Floating point numbers in Scilab

Michaël Baudin

November 2010

Abstract

This document is a small introduction to floating point numbers in Scilab. In the first part, we describe the theory of floating point numbers. We present the definition of a floating point system and of a floating point number. Then we present various representation of floating point numbers, including the sign-significand representation. We present the extreme floating point of a given system and compute the spacing between floating point numbers. Finally, we present the rounding error of representation of floats. In the second part, we present floating point numbers in Scilab. We present the IEEE doubles used in Scilab and explain why 0.1 is rounded with binary floats. We show some examples of lost properties of arithmetic and present overflow and gradual underflow in Scilab. Then we present the infinity and Nan numbers in Scilab and explore the signed zeros. Many examples are provided throughout this document, which ends with a set of exercizes, with their answers.

Contents

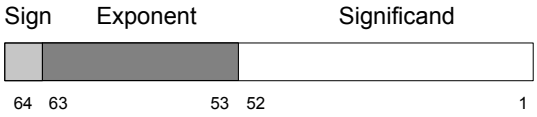

Figure 1: An IEEE-754 64 bits floating point number.

1 Introduction

This document is an open-source project. The LaTeX sources are available on the Scilab Forge:

http://forge.scilab.org/index.php/p/docscifloat/

The LaTeX sources are provided under the terms of the Creative Commons Attribution-ShareAlike 3.0 Unported License:

http://creativecommons.org/licenses/by-sa/3.0

The Scilab scripts are provided on the Forge, inside the project, under the `scripts` sub-directory. The scripts are available under the CeCiLL licence:

http://www.cecill.info/licences/Licence_CeCILL_V2-en.txt

2 Floating point numbers

In this section, we focus on the fact that real variables are stored with limited precision in Scilab.

Floating point numbers are at the core of numerical computations (as in Scilab, Matlab and Octave, for example), as opposed to symbolic computations (as in Maple, Mathematica or Maxima, for example). The limited precision of floating point numbers has a fundamental importance in Scilab. Indeed, many algorithms used in linear algebra, optimization, statistics and most computational fields are deeply modified in order to be able to provide the best possible accuracy.

The first section is a brief overview of floating point numbers. Then we present the `format` function, which allows to see the significant digits of double variables.

2.1 Overview

Real variables are stored by Scilab with 64 bits floating point numbers. That implies that there are 52 significant bits, which correspond to approximately 16 decimal digits. One digit allows to store the sign of the number. The 11 binary digits left are used to store the signed exponent. This way of storing floating point numbers is defined in the IEEE 754 standard [19, 11]. The figure 1 presents an IEEE 754 64 bits double precision floating point number.

This is why we sometimes use the term *double* to refer to real variables in Scilab. Indeed, this corresponds to the way these variables are implemented in Scilab's

source code, where they are associated with *double precision*, that is, twice the precision of a basic real variable.

The set of floating point numbers is not a continuum, it is a finite set. There are $2^{64} \approx 10^{19}$ different doubles in Scilab. These numbers are not equally spaced, there are holes between consecutive floating point numbers. The absolute size of the holes is not always the same ; the size of the holes is relative to the magnitude of the numbers. This relative size is called the *machine precision*.

The pre-defined variable `%eps`, which stands for epsilon, stores the relative precision associated with 64 bits floating point numbers. The relative precision `%eps` can be associated with the number of exact digits of a real value which magnitude is 1. In Scilab, the value of `%eps` is approximately 10^{-16}, which implies that real variables are associated with approximately 16 exact decimal digits. In the following session, we check the property of ϵ_M which satisfies is the smallest floating point number satisfying $1 + \epsilon_M \neq 1$ and $1 + \frac{1}{2}\epsilon_M = 1$, when 1 is stored as a 64 bits floating point number.

```
-->format(18)
-->%eps
 %eps  =
    2.22044604925D-16
-->1+%eps==1
 ans  =
  F
-->1+%eps/2==1
 ans  =
  T
```

2.2 Controlling the precision of the display

In this section, we present the `format` function, which allows to control the precision of the display of variables. Indeed, when we use floating point numbers, we strive to get accurate significant digits. In this context, it is necessary to be able to actually see these digits and the `format` function is designed for that purpose.

By default, 10 characters are displayed by Scilab for each real number. These ten characters include the sign, the decimal point and, if required, the exponent. In the following session, we compute Euler's constant e and its opposite $-e$. Notice that, in both cases, no more that 10 characters are displayed.

```
-->x = exp(1)
 x  =
    2.7182818
-->-exp(1)
 ans  =
  - 2.7182818
```

It may happen that we need to have more precision for our results. To control the display of real variables, we can use the `format` function. In order to increase the number of displayed digits, we can set the number of displayed digits to 18.

```
-->format(18)
-->exp(1)
 ans  =
```

```
    2.718281828459045
```

We now have 15 significant digits of Euler's constant. To reset the formatting back to its default value, we set it to 10.

```
-->format(10)
-->exp(1)
 ans  =
    2.7182818
```

When we manage a large or small number, it is more convenient to process its scientific notation, that is, its representation in the form $a \cdot 10^b$, where the coefficient a is a real number and the exponent b is a negative or positive integer. In the following session, we compute e^{100}.

```
-->exp(100)
 ans  =
    2.688D+43
```

Notice that 9 characters are displayed in the previous session. Since one character is reserved for the sign of the number, the previous format indeed corresponds to the default number of characters, that is 10. Four characters have been consumed by the exponent and the number of significant digits in the fraction is reduced to 3. In order to increase the number of significant digits, we often use the `format(25)` command, as in the following session.

```
-->format(25)
-->x = exp(100)
 x  =
    2.68811714181613544841262555158001358736111187737D+43
```

We now have a lot more digits in the result. We may wonder how many of these digits are correct, that is, what are the significant digits of this result. In order to know this, we compute e^{100} with a symbolic computation system [18]. We get

$$2.688117141816135448412625551580013587361111877 37\ldots \times 10^{43}. \quad (1)$$

We notice that the last digits `609` in the result produced by Scilab are not correct. In this particular case, the result produced by Scilab is correct up to 15 decimal digits. In the following session, we compute the relative error between the result produced by Scilab and the exact result.

```
-->y = 2.68811714181613544841262555158001358736111187737d43
 y  =
    2.68811714181613544841262555158001358736111187737D+43
-->abs(x-y)/abs(y)
 ans  =
    0.
```

We conclude that the floating point representation of the exact result, i.e. `y` is equal to the computed result, i.e. `x`. In fact, the wrong digits `609` displayed in the console are produced by the algorithm which converts the internal floating point binary number into the output decimal number. Hence, if the number of digits required by the `format` function is larger than the number of actual significant digits in the floating point number, the last displayed decimal digits are wrong. In

general, the number of correct significant digits is, at best, from 15 to 17, because this approximately corresponds to the relative precision of binary double precision floating point numbers, that is %eps=2.220D-16.

Another possibility to display floating point numbers is to use the "e"-format, which displays the exponent of the number.

```
-->format("e")
-->exp(1)
 ans  =
    2.718D+00
```

In order to display more significant digits, we can use the second argument of the format function. Because some digits are consumed by the exponent "D+00", we must now allow 25 digits for the display.

```
-->format("e",25)
-->exp(1)
 ans  =
    2.71828182845904509 1D+00
```

2.3 Portable formatting of doubles

The format function is so that the console produces a string which exponent does not depend on the operating system. This is because the formatting algorithm configured by the format function is implemented at the Scilab level.

By opposition, the %e format of the strings produced by the printf function (and the associated mprintf, msprintf, sprintf and ssprintf functions) is operating system dependent. As an example, consider the following script.

```
x=1.e-4;mprintf("%e",x)
```

On a Windows 32 bits system, this produces

```
1.000000e-004
```

while on a Linux 64 bits system, this produces

```
1.000000e-04
```

On many Linux systems, three digits are displayed while two or three digits (depending on the exponent) are displayed on many Windows systems.

Hence, in order to produce a portable string, we can use a combination of the format and string functions. For example, the following script always produces the same result whatever the platform. Notice that the number of digits in the exponent depends on the actual value of the exponent.

```
-->format("e",25)
-->x=-1.e99
 x  =
  - 9.999999999999999 73D+98
-->mprintf("x=%s",string(x))
x=-9.9999999999999973D+98
-->y=-1.e308
 y  =
  - 1.000000000000000 11+308
-->mprintf("y=%s",string(y))
y=-1.0000000000000011+308
```

2.4 Definition

In this section, we give a mathematical definition of a floating point system. We give the example of the floating point system used in Scilab. On such a system, we define a floating point number. We analyse how to compute the floating point representation of a double.

Definition 2.1. *(*Floating point system*) A floating point system is defined by the four integers β, p, e_{min} and e_{max} where*

- $\beta \in \mathbb{N}$ *is the radix and satisfies* $\beta \geq 2$,
- $p \in \mathbb{N}$ *is the precision and satisfies* $p \geq 2$,
- $e_{min}, e_{max} \in \mathbb{N}$ *are the extremal exponents such that*

$$e_{min} < 0 < e_{max}. \tag{2}$$

Example 2.1 Consider the following floating point system:

- $\beta = 2$,
- $p = 53$,
- $e_{min} = -1022$
- $e_{max} = 1023$.

This corresponds to IEEE double precision floating point numbers. We will review this floating point system extensively in the next sections.

A lot of floating point numbers can be represented using the IEEE double system. This is why, in the examples, we will often consider simpler floating point systems. For example, we will consider the floating point system with radix $\beta = 2$, precision $p = 3$ and exponent range $e_{min} = -2$ and $e_{max} = 3$.

With a floating point system, we can represent floating point numbers as introduced in the following definition.

Definition 2.2. *(*Floating point number*) A floating point number x is a real number $x \in \mathbb{R}$ for which there exists at least one representation (M, e) such that*

$$x = M \cdot \beta^{e-p+1}, \tag{3}$$

where

- $M \in \mathbb{N}$ *is called the integral significand and satisfies*

$$|M| < \beta^p, \tag{4}$$

- $e \in \mathbb{N}$ *is called the exponent and satisfies*

$$e_{min} \leq e \leq e_{max}. \tag{5}$$

Example 2.2 Consider the floating point system with radix $\beta = 2$, precision $p = 3$ and exponent range $e_{min} = -2$ and $e_{max} = 3$.

The real number $x = 4$ can be represented by the floating point number $(M, e) = (4, 2)$. Indeed, we have

$$x = 4 \cdot 2^{2-3+1} = 4 \cdot 2^0 = 4. \tag{6}$$

Let us check that the equations 4 and 5 are satisfied. The integral significand M satisfies $M = 4 \leq \beta^p - 1 = 2^3 - 1 = 7$ and the exponent e satisfies $e_{min} = -2 \leq e = 2 \leq e_{max} = 3$ so that this number is a floating point number.

In the previous definition, we state that a floating point number is a real numer $x \in \mathbb{R}$ for which there exists at least one representation (M, e) such that the equation 3 holds. By *at least*, we mean that it might happen that the real number x is either too large or too small. In this case, no couple (M, e) can be found to satisfy the equations 3, 4 and 5. This point will be reviewed later, when we will consider the problem of overflow and underflow.

Moreover, we may be able to find more than one floating point representation of x. This situation is presented in the following example.

Example 2.3 Consider the floating point system with radix $\beta = 2$, precision $p = 3$ and exponent range $e_{min} = -2$ and $e_{max} = 3$.

Consider the real number $x = 3 \in \mathbb{R}$. It can be represented by the floating point number $(M, e) = (6, 1)$. Indeed, we have

$$x = 6 \cdot 2^{1-3+1} = 6 \cdot 2^{-1} = 3. \tag{7}$$

Let us check that the equations 4 and 5 are satisfied. The integral significand M satisfies $M = 6 \leq \beta^p - 1 = 2^3 - 1 = 7$ and the exponent e satisfies $e_{min} = -2 \leq e = 1 \leq e_{max} = 3$ so that this number is a floating point number. In order to find another floating point representation of x, we could divide M by 2 and add 1 to the exponent. This leads to the floating point number $(M, e) = (3, 2)$. Indeed, we have

$$x = 3 \cdot 2^{2-3+1} = 3 \cdot 2^0 = 3. \tag{8}$$

The equations 4 and 5 are still satisfied. Therefore the couple $(M, e) = (3, 2)$ is another floating point representation of $x = 3$. This point will be reviewed later when we will present normalized numbers.

The following definition introduces the *quantum*. This term will be reviewed in the context of the notion of *ulp*, which will be analyzed in the next sections.

Definition 2.3. *(*Quantum*) Let $x \in \mathbb{R}$ and let (M, e) its floating point representation. The quantum of the representation of x is*

$$\beta^{e-p+1} \tag{9}$$

and the quantum exponent is

$$q = e - p + 1. \tag{10}$$

There are two different types of limitations which occur in the context of floating point computations.

- The finiteness of p is limitation on *precision*.
- The inequalities 5 on the exponent implies a limitation on the *range* of floating point numbers.

This leads to consider the set of all floating point numbers as a subset of the real numbers, as in the following definition.

Definition 2.4. *(*Floating point set*) Consider a floating point system defined by β, p, e_{min} and e_{max}. Let $x \in \mathbb{R}$. The set of all floating point numbers is F as defined by*

$$F = \{M \cdot \beta^{e-p+1} | |M| \leq \beta^p - 1, e_{min} \leq e \leq e_{max}\}. \tag{11}$$

2.5 Sign-significand floating point representation

In this section, we present the sign-significand floating point representation which is often used in practice.

Proposition 2.5. *(*Sign-significand floating point representation*) Assume that x is a nonzero floating point number. Therefore, the number x can be equivalently defined as*

$$x = (-1)^s \cdot m \cdot \beta^e \tag{12}$$

where

- *$s \in \{0, 1\}$ is the sign of x,*
- *$m \in \mathbb{R}$ is the normal significand and satisfies the inequalities*

$$0 \leq m < \beta, \tag{13}$$

- *$e \in \mathbb{N}$ is the exponent and satisfies the inequalities*

$$e_{min} \leq e \leq e_{max}. \tag{14}$$

Obviously, the exponent in the representation of the definition 2.2 is the same as in the proposition 2.5. In the following proof, we find the relation between m and M.

Proof. We must proove that the two representations 3 and 12 are equivalent.

First, assume that $x \in F$ is a nonzero floating point number in the sense of the definition 2.2. Then, let us define the sign $s \in \{0, 1\}$ by

$$s = \begin{cases} 0, & \text{if } x > 0, \\ 1, & \text{if } x < 0. \end{cases} \tag{15}$$

Let us define the normal significand by

$$m = |M| \beta^{1-p}. \tag{16}$$

This implies $|M| = m\beta^{p-1}$. Since the integral significand M has the same sign as x, we have $M = (-1)^s m\beta^{p-1}$. Hence, by the equation 3, we have

$$\begin{aligned} x &= (-1)^s m\beta^{p-1}\beta^{e-p+1} && (17)\\ &= (-1)^s m\beta^e, && (18) \end{aligned}$$

which concludes the first part of the proof.

Second, assume that $x \in F$ is a nonzero floating point number in the sense of the proposition 2.5. Let us define the integral significand M by

$$M = (-1)^s m\beta^{p-1}. \tag{19}$$

This implies $(-1)^s m = M\beta^{1-p}$. Hence, by the equation 12, we have

$$\begin{aligned} x &= M\beta^{1-p}\beta^e && (20)\\ &= M\beta^{e-p+1}, && (21) \end{aligned}$$

which concludes the second part of the proof. □

Notice that we carefully assumed that x be nonzero. Indeed, if $x = 0$, then m must be equal to zero, while the two different signs $s = 0$ and $s = 1$ produce the same result. This leads in practice to consider the two signed zeros `-0` and `+0`. This point will be reviewed lated in the context of the analyzis of IEEE 754 doubles.

Example 2.4 Consider the floating point system with radix $\beta = 2$, precision $p = 3$ and exponent range $e_{min} = -2$ and $e_{max} = 3$.

The real number $x = 4$ can be represented by the floating point number $(s, m, e) = (0, 1, 2)$. Indeed, we have

$$x = (-1)^0 \cdot 1 \cdot 2^2 = 1 \cdot 2^2 = 4. \tag{22}$$

The equation 13 is satisfied, since $m = 1 < \beta = 2$.

2.6 Normal and subnormal numbers

In order to make sure that each real number has a unique representation, we must impose bounds on the integral significand M.

Proposition 2.6. *(*Normalized floating point numbers*) Floating point numbers are normalized if the integral significand satisfies*

$$\beta^{p-1} \leq |M| < \beta^p. \tag{23}$$

If x is a nonzero normalized floating point number, therefore its floating point representation (M, e) is unique and the exponent e satisfies

$$e = \lfloor \log_\beta(|x|) \rfloor, \tag{24}$$

while the integral significand M satisfies

$$M = \frac{x}{\beta^{e-p+1}}. \tag{25}$$

Proof. First, we prove that a normalized nonzero floating point number has a unique (M, e) representation. Assume that the floating point number x has the two representations (M_1, e_1) and (M_2, e_2). By the equation 3, we have

$$x = M_1 \cdot \beta^{e_1 - p + 1} = M_2 \cdot \beta^{e_2 - p + 1}, \tag{26}$$

which implies

$$|x| = |M_1| \cdot \beta^{e_1 - p + 1} = |M_2| \cdot \beta^{e_2 - p + 1}. \tag{27}$$

We can compute the base-β logarithm of $|x|$, which will lead us to an expression of the exponent. Notice that we assumed that $x \neq 0$, which allows us to compute $\log(|x|)$. The previous equation implies

$$\log_\beta(|x|) = \log_\beta(|M_1|) + e_1 - p + 1 = \log_\beta(|M_2|) + e_2 - p + 1. \tag{28}$$

We now extract the largest integer lower or equal to $\log_\beta(x)$ and get

$$\lfloor \log_\beta(|x|) \rfloor = \lfloor \log_\beta(|M_1|) \rfloor + e_1 - p + 1 = \lfloor \log_\beta(|M_2|) \rfloor + e_2 - p + 1. \tag{29}$$

We can now find the value of $\lfloor \log_\beta(M_1) \rfloor$ by using the inequalities on the integral significand. The hypothesis 23 implies

$$\beta^{p-1} \leq |M_1| < \beta^p, \qquad \beta^{p-1} \leq |M_2| < \beta^p. \tag{30}$$

We can take the base-β logarithm of the previous inequalities and get

$$p - 1 \leq \log_\beta(|M_1|) < p, \qquad p - 1 \leq \log_\beta(|M_2|) < p. \tag{31}$$

By the definition of the function $\lfloor \cdot \rfloor$, the previous inequalities imply

$$\log_\beta(|M_1|) = \log_\beta(|M_2|) = p - 1. \tag{32}$$

We can finally plug the previous equality into 29, which leads to

$$\lfloor \log_\beta(|x|) \rfloor = p - 1 + e_1 - p + 1 = p - 1 + e_2 - p + 1, \tag{33}$$

which implies

$$e_1 = e_2. \tag{34}$$

The equality 26 immediately implies $M_1 = M_2$.

Moreover, the equality 33 implies 24 while 25 is necessary for the equality $x = M \cdot \beta^{e-p+1}$ to hold. □

Example 2.5 Consider the floating point system with radix $\beta = 2$, precision $p = 3$ and exponent range $e_{min} = -2$ and $e_{max} = 3$.

We have seen in example 2.3 that the real number $x = 3$ can be represented both by $(M, e) = (6, 1)$ and $(M, e) = (3, 2)$. In order to see which floating point representation is normalized, we evaluate the bounds in the inequalities 23. We have $\beta^{p-1} = 2^{3-1} = 4$ and $\beta^p = 2^3 = 8$. Therefore, the floating point representation $(M, e) = (6, 1)$ is normalized while the floating point representation $(M, e) = (3, 2)$ is not normalized.

The proposition 2.6 gives a way to compute the floating point representation of a given nonzero real number x. In the general case where the radix β is unusual, we may compute the exponent from the formula $\log_\beta(|x|) = \log(|x|)/\log(\beta)$. If the radix is equal to 2 or 10, we may use the Scilab functions `log2` and `log10`.

Example 2.6 Consider the floating point system with radix $\beta = 2$, precision $p = 3$ and exponent range $e_{min} = -2$ and $e_{max} = 3$.

By the equation 24, the real number $x = 3$ is associated with the exponent

$$e = \lfloor \log_2(3) \rfloor. \tag{35}$$

In the following Scilab session, we use the `log2` and `floor` functions to compute the exponent e.

```
-->x = 3
 x  =
    3.
-->log2(abs(x))
 ans  =
    1.5849625
-->e = floor(log2(abs(x)))
 e  =
    1.
```

In the following session, we use the equation 25 to compute the integral significant M.

```
-->M = x/2^(e-3+1)
 M  =
    6.
```

We emphasize that the equations 24 and 25 hold only when x is a floating point number. Indeed, for a general real number $x \in \mathbb{R}$, there is no reason why the exponent e, computed from 24, should satisfy the inequalities $e_{min} \leq e \leq e_{max}$. There is also no reason why the integral significand M, computed from 25, should be an integer.

There are cases where the real number x cannot be represented by a normalized floating point number, but can still be represented by some couple (M, e). These cases lead to the subnormal numbers.

Definition 2.7. *(*Subnormal floating point numbers*) A subnormal floating point number is associated with the floating point representation* (M, e) *where* $e = e_{min}$ *and the integral significand satisfies the inequality*

$$|M| < \beta^{p-1}. \tag{36}$$

The term *denormal* number is often used too.

Example 2.7 Consider the floating point system with radix $\beta = 2$, precision $p = 3$ and exponent range $e_{min} = -2$ and $e_{max} = 3$.

Consider the real number $x = 0.125$. In the following session, we compute the exponent e by the equation 24.

```
-->x = 0.125
 x  =
    0.125
-->e = floor(log2(abs(x)))
 e  =
  - 3.
```

We find an exponent which does not satisfy the inequalities $e_{min} \leq e \leq e_{max}$. The real number x might still be representable as a subnormal number. We set $e = e_{min} = -2$ and compute the integral significand by the equation 25.

```
-->e=-2
 e  =
  - 2.
-->M = x/2^(e-3+1)
 M  =
    2.
```

We find that the integral significand $M = 2$ is an integer. Therefore, the couple $(M, e) = (2, -2)$ is a subnormal floating point representation for the real number $x = 0.125$.

Example 2.8 Consider the floating point system with radix $\beta = 2$, precision $p = 3$ and exponent range $e_{min} = -2$ and $e_{max} = 3$.

Consider the real number $x = 0.1$. In the following session, we compute the exponent e by the equation 24.

```
-->x = 0.1
 x  =
    0.1
-->e = floor(log2(abs(x)))
 e  =
  - 4.
```

We find an exponent which is too small. Therefore, we set $e = -2$ and try to compute the integral significand.

```
-->e=-2
 e  =
  - 2.
-->M = x/2^(e-3+1)
 M  =
    1.6
```

This time, we find a value of M which is not an integer. Therefore, in the current floating point system, there is no exact floating point representation of the number $x = 0.1$.

If $x = 0$, we select $M = 0$, but any value of the exponent allows to represent x. Indeed, we cannot use the expression $\log(|x|)$ anymore.

2.7 B-ary representation and the implicit bit

In this section, we present the β-ary representation of a floating point number.

Proposition 2.8. *(*B-ary representation*) Assume that x is a floating point number. Therefore, the floating point number x can be expressed as*

$$x = \pm \left(d_1 + \frac{d_2}{\beta} + \ldots + \frac{d_p}{\beta^{p-1}} \right) \cdot \beta^e, \tag{37}$$

which is denoted by

$$x = \pm (d_1.d_2 \cdots d_p)_\beta \cdot \beta^e. \tag{38}$$

Proof. By the definition 2.2, there exists (M, e) so that $x = M \cdot \beta^e$ with $e_{min} \leq e \leq e_{max}$ and $|M| < \beta^p$. The inequality $|M| < \beta^p$ implies that there exists at most p digits d_i which allow to decompose the positive integer $|M|$ in base β. Hence,

$$|M| = d_1 \beta^{p-1} + d_2 \beta^{p-2} + \ldots + d_p, \tag{39}$$

where $0 \leq d_i \leq \beta - 1$ for $i = 1, 2, \ldots, p$. We plug the previous decomposition into $x = M \cdot \beta^e$ and get

$$\begin{aligned} x &= \pm \left(d_1 \beta^{p-1} + d_2 \beta^{p-2} + \ldots + d_p \right) \beta^{e-p+1} && (40) \\ &= \pm \left(d_1 + \frac{d_2}{\beta} + \ldots + \frac{d_p}{\beta^{p-1}} \right) \beta^e, && (41) \end{aligned}$$

which concludes the proof. □

The equality of the expressions 37 and 12 allows to see that the digits d_i are simply computed from the β-ary expansion of the normal significand m.

Example 2.9 Consider the floating point system with radix $\beta = 2$, precision $p = 3$ and exponent range $e_{min} = -2$ and $e_{max} = 3$.

The normalized floating point number $x = -0.3125$ is represented by the couple (M, e) with $M = -5$ and $e = -2$ since $x = -0.3125 = -5 \cdot 2^{-2-3+1} = -5 \cdot 2^{-4}$. Alternatively, it is represented by the triplet (s, m, e) with $m = 1.25$ since $x = -0.3125 = (-1)^1 \cdot 1.25 \cdot 2^{-2}$. The binary decomposition of m is $1.25 = (1.01)_2 = 1 + 0 \cdot \frac{1}{2} + 1 \cdot \frac{1}{4}$. This allows to write x as $x = -(1.01)_2 \cdot 2^{-2}$.

Proposition 2.9. *(*Leading bit of a b-ary representation*) Assume that x is a floating point number. If x is a normalized number, therefore the leading digit d_1 of the β-ary representation of x defined by the equation 37 is nonzero. If x is a subnormal number, therefore the leading digit is zero.*

Proof. Assume that x is a normalized floating point number and consider its representation $x = M \cdot \beta^{e-p+1}$ where the integral significand M satisfies $\beta^{p-1} \leq |M| < \beta^p$. Let us proove that the leading digit of x is nonzero. We can decompose $|M|$ in base β. Hence,

$$|M| = d_1 \beta^{p-1} + d_2 \beta^{p-2} + \ldots + d_p, \tag{42}$$

where $0 \leq d_i \leq \beta - 1$ for $i = 1, 2, \ldots, p$. We must prove that $d_1 \neq 0$. The proof will proceed by contradiction. Assume that $d_1 = 0$. Therefore, the representation of $|M|$ simplifies to

$$|M| = d_2 \beta^{p-2} + d_3 \beta^{p-3} + \ldots + d_p. \tag{43}$$

Since the digits d_i satisfy the inequality $d_i \leq \beta - 1$, we have the inequality

$$|M| \leq (\beta - 1)\beta^{p-1} + (\beta - 1)\beta^{p-2} + \ldots + (\beta - 1). \tag{44}$$

We can factor the term $\beta - 1$ in the previous expression, which leads to

$$|M| \leq (\beta - 1)(\beta^{p-1} + \beta^{p-2} + \ldots + 1). \tag{45}$$

From calculus, we know that, for any number y and any positive integer n, we have $1 + y + y^2 + \ldots + y^n = (y^{n+1} - 1)/(y - 1)$. Hence, the inequality 45 implies

$$|M| \leq \beta^{p-1} - 1. \tag{46}$$

The previous inequality is a contradiction, since, by assumption, we have $\beta^{p-1} \leq |M|$. Therefore, the leading digit d_1 is nonzero, which concludes the first part of the proof.

We now prove that, if x is a subnormal number, therefore its leading digit is zero. By the definition 2.7, we have $|M| < \beta^{p-1}$. This implies that there exist $p - 1$ digits d_i for $i = 2, 3, \ldots, p$ such that

$$|M| = d_2\beta^{p-2} + d_3\beta^{p-3} + \ldots + d_p. \tag{47}$$

Hence, we have $d_1 = 0$, which concludes the proof. □

The proposition 2.9 implies that, in radix 2, a normalized floating point number can be written as

$$x = \pm(1.d_2 \cdots d_p)_\beta \cdot \beta^e, \tag{48}$$

while a subnormal floating point number can be written as

$$x = \pm(0.d_2 \cdots d_p)_\beta \cdot \beta^e. \tag{49}$$

In practice, a special encoding allows to see if a number is normal or subnormal. Hence, there is no need to store the first bit of its significand. This hidden bit or implicit bit is frequently used.

For example, the IEEE 754 standard for double precision floating point numbers is associated with the precision $p = 53$ bits, while 52 bits only are stored in the normal significand.

Example 2.10 Consider the floating point system with radix $\beta = 2$, precision $p = 3$ and exponent range $e_{min} = -2$ and $e_{max} = 3$.

We have already seen that the normalized floating point number $x = -0.3125$ is associated with a leading 1, since it is represented by $x = -(1.01)_2 \cdot 2^{-2}$.

On the other side, consider the floating point number $x = 0.125$ and let us check that the leading bit of the normal significand is zero. It is represented by $x = (-1)^0 \cdot 0.5 \cdot 2^{-2}$, which leads to $x = (-1)^0 \cdot (0.10)_2 \cdot 2^{-2}$.

2.8 Extreme floating point numbers

In this section, we focus on the extreme floating point numbers associated with a given floating point system.

Proposition 2.10. *(*Extreme floating point numbers*) Consider the floating point system β, p, e_{min}, e_{max}.*

- *The smallest positive normal floating point number is*

$$\mu = \beta^{e_{min}}. \tag{50}$$

- *The largest positive normal floating point number is*

$$\Omega = (\beta - \beta^{1-p})\beta^{e_{max}}. \tag{51}$$

- *The smallest positive subnormal floating point number is*

$$\alpha = \beta^{e_{min}-p+1}. \tag{52}$$

Proof. The smallest positive normal integral significand is $M = \beta^{p-1}$. Since the smallest exponent is e_{min}, we have $\mu = \beta^{p-1} \cdot \beta^{e_{min}-p+1}$, which simplifies to the equation 50.

The largest positive normal integral significand is $M = \beta^p - 1$. Since the largest exponent is e_{max}, we have $\Omega = (\beta^p - 1) \cdot \beta^{e_{max}-p+1}$ which simplifies to the equation 51.

The smallest positive subnormal integral significand is $M = 1$. Therefore, we have $\alpha = 1 \cdot \beta^{e_{min}-p+1}$, which leads to the equation 52. □

Example 2.11 Consider the floating point system with radix $\beta = 2$, precision $p = 3$ and exponent range $e_{min} = -2$ and $e_{max} = 3$.

The smallest positive normal floating point number is $\mu = 2^{-2} = 0.25$. The largest positive normal floating point number is $\Omega = (2 - 2^{-2}) \cdot 2^3 = 16 - 2 = 14$. The smallest positive subnormal floating point number is $\alpha = 2^{-4} = 0.0625$.

2.9 A toy system

In order to see this, we consider the floating point system with radix $\beta = 2$, precision $p = 3$ and exponent range $e_{min} = -2$ and $e_{max} = 3$. The following Scilab script defines these variables.

```
radix = 2
p = 3
emin = -2
emax = 3
```

On such a simple floating point system, it is easy to compute all representable floating point numbers. In the following script, we compute the minimum and maximum integral significand M of positive normalized numbers, as defined by the inequalities 23, that is $M_{min} = \beta^{p-1}$ and $M_{max} = \beta^p - 1$.

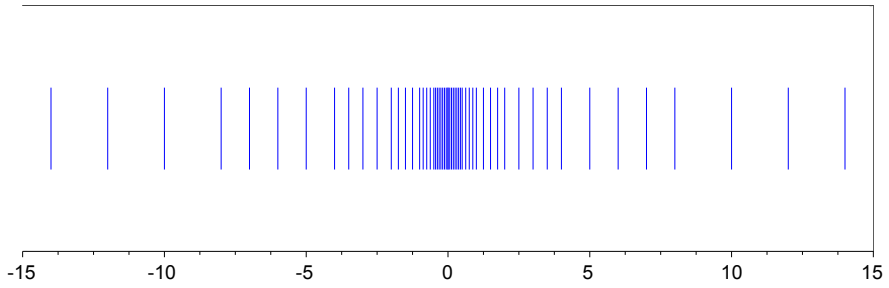

Figure 2: Floating point numbers in the floating point system with radix $\beta = 2$, precision $p = 3$ and exponent range $e_{min} = -2$ and $e_{max} = 3$.

```
-->Mmin = radix^(p - 1)
 Mmin  =
    4.
-->Mmax = radix^p - 1
 Mmax  =
    7.
```

In the following script, we compute all the normalized floating point numbers which can be computed from the equation $x = M \cdot \beta^{e-p+1}$, with M in the intervals $[-M_{max}, -M_{min}]$ and $[M_{min}, M_{max}]$ and e in the interval $[e_{min}, e_{max}]$.

```
f = [];
for e = emax : -1 : emin
  for M = -Mmax : -Mmin
    f($+1) = M * radix^(e - p + 1);
  end
end
f($+1) = 0;
for e = emin : emax
  for M = Mmin : Mmax
    f($+1) = M * radix^(e - p + 1);
  end
end
```

The previous script produces the following numbers;
-14, -12, -10, -8, -7, -6, -5, -4, -3.5, -3, -2.5, -2, -1.75, -1.5, -1.25, -1, -0.875, -0.75, -0.625, -0.5, -0.4375, -0.375, -0.3125, -0.25, 0, 0.25, 0.3125, 0.375, 0.4375, 0.5, 0.625, 0.75, 0.875, 1, 1.25, 1.5, 1.75, 2, 2.5, 3, 3.5, 4, 5, 6, 7, 8, 10, 12, 14.
These floating point numbers are presented in the figure 2. Notice that there are much more numbers in the neighbourhood of zero, than on the left and right hand sides of the picture. This figure shows clearly that the space between adjacent floating point numbers depend on their magnitude.

In order to see more clearly what happens, we present in the figure 3 only the positive normal floating point numbers. This figure shows more clearly that, whenever a floating point numbers is of the form 2^e, then the space between adjacent numbers is multiplied by a factor 2.

The previous list of numbers included only normal numbers. In order to include subnormal numbers in our list, we must add two loops, associated with $e = e_{min}$

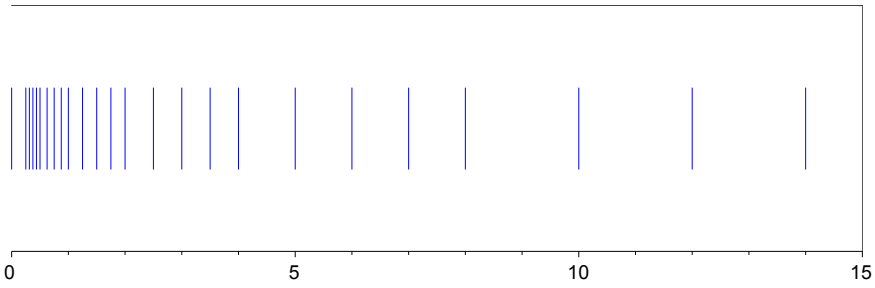

Figure 3: Positive floating point numbers in the floating point system with radix $\beta = 2$, precision $p = 3$ and exponent range $e_{min} = -2$ and $e_{max} = 3$ – Only normal numbers (denormals are exluded).

and integral significands from $-M_{min}$ to -1 and from 1 to M_{min}.

```
f = [];
for e = emax : -1 : emin
  for M = -Mmax : -Mmin
    f($+1) = M * radix^(e - p + 1);
  end
end
e = emin;
for M = -Mmin + 1 : -1
  f($+1) = M * radix^(e - p + 1);
end
f($+1) = 0;
e = emin;
for M = 1 : Mmin - 1
  f($+1) = M * radix^(e - p + 1);
end
for e = emin : emax
  for M = Mmin : Mmax
    f($+1) = M * radix^(e - p + 1);
  end
end
```

The previous script produces the following numbers, where we wrote in bold face the subnormal numbers.
-14, -12, -10, -8, -7, -6, -5, -4, -3.5, -3, -2.5, -2, -1.75, -1.5, -1.25, -1, -0.875, -0.75, -0.625, -0.5, -0.4375, -0.375, -0.3125, -0.25, **-0.1875**, **-0.125**, **-0.0625**, 0., **0.0625**, **0.125**, **0.1875**, 0.25, 0.3125, 0.375, 0.4375, 0.5, 0.625, 0.75, 0.875, 1, 1.25, 1.5, 1.75, 2, 2.5, 3, 3.5, 4, 5, 6, 7, 8, 10, 12, 14.

The figure 4 present the list of floating point numbers, where subnormal numbers are included. Compared to the figure 3, we see that the subnormal numbers allows to fill the space between zero and the smallest normal positive floating point number.

Finally, the figures 5 and 6 present the positive floating point numbers in (base 10) logarithmic scale, with and without denormals. In this scale, the numbers are more equally spaced, as expected.

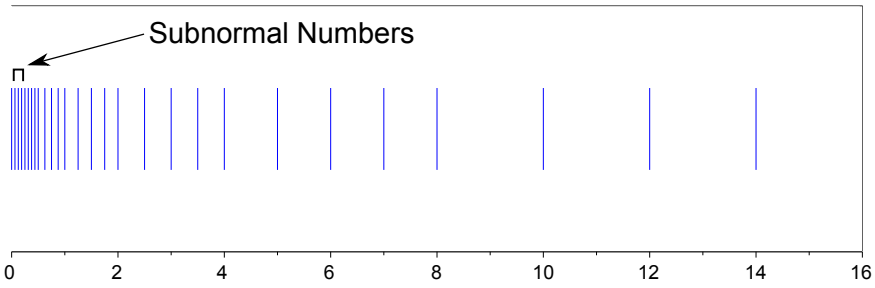

Figure 4: Positive floating point numbers in the floating point system with radix $\beta = 2$, precision $p = 3$ and exponent range $e_{min} = -2$ and $e_{max} = 3$ – Denormals are included.

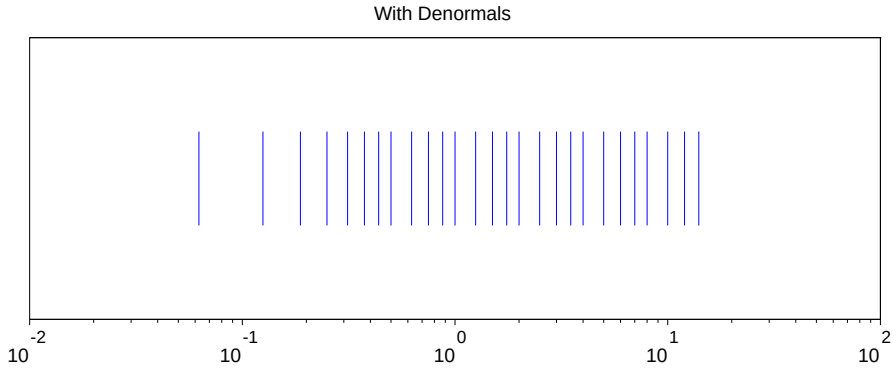

Figure 5: Positive floating point numbers in the floating point system with radix $\beta = 2$, precision $p = 3$ and exponent range $e_{min} = -2$ and $e_{max} = 3$ – With denormals. Logarithmic scale.

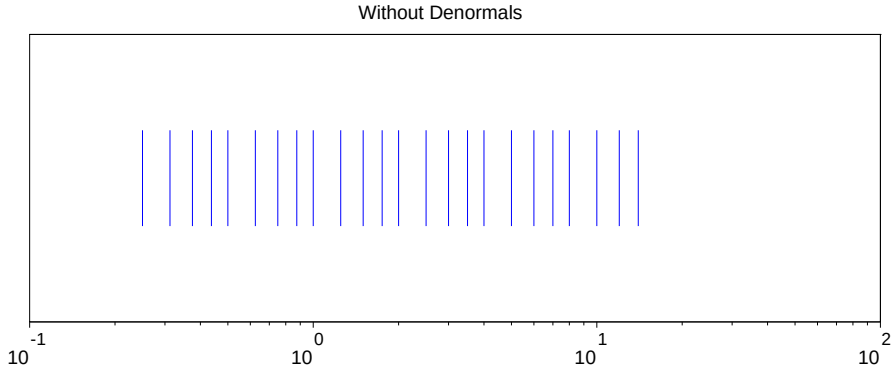

Figure 6: Positive floating point numbers in the floating point system with radix $\beta = 2$, precision $p = 3$ and exponent range $e_{min} = -2$ and $e_{max} = 3$ – Without denormals. Logarithmic scale.

2.10 Spacing between floating point numbers

In this section, we compute the spacing between floating point numbers and introducing the machine epsilon.

The floating point numbers in a floating point system are not equally spaced. Indeed, the difference between two consecutive floating point numbers depend on their magnitude.

Let x be a floating point number. We denote by x^+ the next larger floating point number and x^- the next smaller. We have

$$x^- < x < x^+, \tag{53}$$

and we are interested in the distance between these numbers, that is, we would like to compute $x^+ - x$ and $x - x^-$.

We are particularily interested in the spacing between the number $x = 1$ and the next floating point number.

Example 2.12 Consider the floating point system with radix $\beta = 2$, precision $p = 3$ and exponent range $e_{min} = -2$ and $e_{max} = 3$.

The floating point number $x = 1$ is represented by $M = 4$ and $e = 0$, since $1 = 4 \cdot 2^{0-3+1}$. The next floating point number is represented by $M = 5$ and $e = 0$. This leads to $x^+ = 5 \cdot 2^{0-3+1} = 5 \cdot 2^{-2} = 1.25$. Hence, the difference between these two numbers is $0.25 = 2^{-2}$.

The previous example leads to the definition of the *machine epsilon*.

Proposition 2.11. *(*Machine epsilon*) The spacing between the floating point number $x = 1$ and the next larger floating point number x^+ is the machine epsilon, which satisfies*

$$\epsilon_M = \beta^{1-p}. \tag{54}$$

Proof. We first have to compute the floating point representation of $x = 1$. Consider the integral significand $M = \beta^{p-1}$ and the exponent $e = 0$. We have $M \cdot \beta^{e-p+1} = \beta^{p-1} \cdot \beta^{1-p} = 1$. This shows that the floating point number $x = 1$ is represented by $M = \beta^{p-1}$ and $e = 0$.

The next floating point number x^+ is therefore represented by $M^+ = M + 1$ and $e = 0$. Therefore, $x^+ = (1 + \beta^{p-1}) \cdot \beta^{1-p}$. The difference between these two numbers is $x^+ - x = 1 \cdot \beta^{1-p}$, which concludes the proof. □

In the following example, we compute the distance between x and x^-, in the particular case where x is of the form β^e. We consider the particular case $x = 2^0 = 1$.

Example 2.13 Consider the floating point system with radix $\beta = 2$, precision $p = 3$ and exponent range $e_{min} = -2$ and $e_{max} = 3$.

The floating point number $x = 1$ is represented by $M = 4$ and $e = 0$, since $1 = 4 \cdot 2^{0-3+1}$. The previous floating point number $x^- = 0.875$ is represented by $M = 7$ and $e = -1$, since $x^- = 7 \cdot 2^{-1-3+1} = 7 \cdot 2^{-3}$. Hence, the difference between these two numbers is $0.125 = \frac{1}{2} \cdot 0.25$ which can be written $0.125 = \frac{1}{2}\epsilon_M$, since $\epsilon_M = 0.25$ for this system.

The next proposition allows to know the distance between x and its adjacent floating point number, in the general case.

Proposition 2.12. *(*Spacing between floating point numbers*) Let x be a normalized floating point number. Assume that y is an adjacent normalized floating point number, that is, $y = x^+$ or $y = x^-$. Assume that neither x nor y are zero. Therefore,*

$$\frac{1}{\beta}\epsilon_M|x| \leq |x - y| \leq \epsilon_M|x|. \tag{55}$$

Proof. We separate the proof in two parts, where the first part focuses on $y = x^+$ and the second part focuses on $y = x^-$.

Assume that $y = x^+$ and let us compute $x^+ - x$. Let (M, e) be the floating point representation of x, so that $x = M \cdot \beta^{e-p+1}$. Let us denote by (M^+, e^+) the floating point representation of x^+. We have $x^+ = M^+ \cdot \beta^{e^+-p+1}$.

The next floating point number x^+ might have the same exponent e as x, and a modified integral significand M, or an increased exponent e and the same M. Depending on the sign of the number, an increased e or M may produce a greater or a lower value, thus changing the order of the numbers. Therefore, we must separate the case $x > 0$ and the case $x < 0$. Since neither of x or y is zero, we can consider the case $x, y > 0$ first and prove the inequality 55. The case $x, y < 0$ can be processed in the same way, which lead to the same inequality.

Assume that $x, y > 0$. If the integral significand M of x is at its upper bound, the exponent e must be updated: if not, the number would not be normalized anymore. So we must separate the two following cases: (i) the exponent e is the same for x and $y = x^+$ and (ii) the integral significand M is the same.

Consider the case where $e^+ = e$. Therefore, $M^+ = M + 1$, which implies

$$\begin{aligned} x^+ - x &= (M+1) \cdot \beta^{e-p+1} - M \cdot \beta^{e-p+1} && (56) \\ &= \beta^{e-p+1}. && (57) \end{aligned}$$

By the equality 54 defining the machine epsilon, this leads to

$$x^+ - x = \epsilon_M \beta^e. \tag{58}$$

In order to get an upper bound on $x^+ - x$ depending on $|x|$, we must bound $|x|$, depending on the properties of the floating point system. By hypothesis, the number x is normalized, therefore $\beta^{p-1} \leq |M| < \beta^p$. This implies

$$\beta^{p-1} \cdot \beta^{e-p+1} \leq |M| \cdot \beta^{e-p+1} < \beta^p \cdot \beta^{e-p+1}. \tag{59}$$

Hence,

$$\beta^e \leq |x| < \beta^{e+1}, \tag{60}$$

which implies

$$\frac{1}{\beta}|x| < \beta^e \leq |x|. \tag{61}$$

We plug the previous inequality into 58 and get

$$\frac{1}{\beta}\epsilon_M|x| < x^+ - x \leq \epsilon_M|x|. \tag{62}$$

Therefore, we have proved a slightly stronger inequality than required: the left inequality is strict, while the left part of 55 is less or equal than.

Now consider the case where $e^+ = e+1$. This implies that the integral significand of x is at its upper bound $\beta^p - 1$, while the integral significand of x^+ is at its lower bound β^{p-1}. Hence, we have

$$\begin{aligned} x^+ - x &= \beta^{p-1} \cdot \beta^{e+1-p+1} - (\beta^p - 1) \cdot \beta^{e-p+1} && (63)\\ &= \beta^p \cdot \beta^{e-p+1} - (\beta^p - 1) \cdot \beta^{e-p+1} && (64)\\ &= \beta^{e-p+1} && (65)\\ &= \epsilon_M \beta^e. && (66) \end{aligned}$$

We plug the inequality 61 in the previous equation and get the inequality 62.

We now consider the number $y = x^-$. We must compute the distance $x - x^-$. We could use our previous inequality, but this would not lead us to the result. Indeed, let us introduce $z = x^-$. Therefore, we have $z^+ = x$. By the inequality 62, we have

$$\frac{1}{\beta}\epsilon_M|z| < z^+ - z \leq \epsilon_M|z|, \tag{67}$$

which implies

$$\frac{1}{\beta}\epsilon_M|x^-| < x - x^- \leq \epsilon_M|x^-|, \tag{68}$$

but this is not the inequality we are searching for, since it uses $|x^-|$ instead of $|x|$.

Let (M^-, e^-) be the floating point representation of x^-. We have $x^- = M^- \cdot \beta^{e^- - p+1}$. Consider the case where $e^- = e$. Therefore, $M^- = M - 1$, which implies

$$\begin{aligned} x - x^- &= M \cdot \beta^{e-p+1} - (M-1) \cdot \beta^{e-p+1} && (69)\\ &= \beta^{e-p+1} && (70)\\ &= \epsilon_M \beta^e. && (71) \end{aligned}$$

We plug the inequality 61 into the previous equality and we get

$$\frac{1}{\beta}\epsilon_M|x| < x - x^- \leq \epsilon_M|x|. \tag{72}$$

Consider the case where $e^- = e - 1$. Therefore, the integral significand of x is at its lower bound β^{p-1} while the integral significand of x^- is at is upper bound $\beta^p - 1$. Hence, we have

$$\begin{aligned} x - x^- &= \beta^{p-1} \cdot \beta^{e-p+1} - (\beta^p - 1) \cdot \beta^{e-1-p+1} && (73)\\ &= \beta^{p-1}\beta^{e-p+1} - (\beta^{p-1} - \frac{1}{\beta})\beta^{e-p+1} && (74)\\ &= \frac{1}{\beta}\beta^{e-p+1} && (75)\\ &= \frac{1}{\beta}\epsilon_M \beta^e. && (76) \end{aligned}$$

Figure 7: Rounding modes

The integral significand of $|x|$ is β^{p-1}, which implies that

$$|x| = \beta^{p-1} \cdot \beta^{e-p+1} = \beta^e. \tag{77}$$

Therefore,

$$x - x^- = \frac{1}{\beta}\epsilon_M |x|. \tag{78}$$

The previous equality proves that the lower bound $\frac{1}{\beta}\epsilon_M|x|$ can be attained in the inequality 55 and concludes the proof. □

2.11 Rounding modes

In this section, we present the four rounding modes defined by the IEEE 754-2008 standard, including the default round to nearest rounding mode.

Assume that x is an arbitrary real number. It might happen that there is a floating point number in F which is exactly equal to x. In the general case, there is no such exact representation of x and we must select a floating point number which approximates x at best. The rules by which we make the selection leads to *rounding*. If x is a real number, we denote by $fl(x) \in F$ the floating point number which represents x in the currrent floating point system. The $fl(\cdot)$ function is the rounding function.

The IEEE 754-2008 standard defines four rounding modes (see [11], section 4.3 "Rounding-direction attributes"), that is, four rounding functions $fl(x)$.

- round to nearest: $RN(x)$ is the floating point number that is the closest to x.
- round toward positive: $RU(x)$ is the largest floating point number greater than or equal to x.
- round toward negative: $RD(x)$ is the largest floating point number less than or equal to x.
- round toward zero: $RZ(x)$ is the closest floating point number to x that is no greater in magnitude than x.

The figure 7 presents the four rounding modes defined by the IEEE 754-2008 standard.

The round to nearest mode is the default rounding mode and this is why we focus on this particular rounding mode. Hence, in the remaining of this document, we will consider only $fl(x) = RN(x)$.

It may happen that the real number x falls exactly between two adjacent floating point numbers. In this case, we must use a tie-breaking rule to know which floating point number to select. The IEEE 754-2008 standard defines two tie-breaking rules.

- round ties to even: the nearest floating point number for which the integral significand is even is selected.
- round ties to away: the nearest floating point number with larger magnitude is selected.

The default tie-breaking rule is the round ties to even.

Example 2.14 Consider the floating point system with radix $\beta = 2$, precision $p = 3$ and exponent range $e_{min} = -2$ and $e_{max} = 3$.

Consider the real number $x = 0.54$. This number falls between the two normal floating point numbers $x_1 = 0.5 = 4 \cdot 2^{-1-3+1}$ and $x_2 = 0.625 = 5 \cdot 2^{-1-3+1}$. Depending on the rounding mode, the number x might be represented by x_1 or x_2. In the following sesssion, we compute the absolute distance between x and the two numbers x_1 and x_2.

```
-->x = 0.54;
-->x1 = 0.5;
-->x2 = 0.625;
-->abs(x-x1)
 ans  =
    0.04
-->abs(x-x2)
 ans  =
    0.085
```

Hence, the four different rounding of x are

$$RZ(x) = RN(x) = RD(x) = 0.5, \tag{79}$$

$$RU(x) = 0.625. \tag{80}$$

Consider the real number $x = 4.5 \cdot 2^{-1-3+1} = 0.5625$. This number falls exactly between the two normal floating point numbers $x_1 = 0.5$ and $x_2 = 0.625$. This is a tie, which must, by default, be solved by the round ties to even rule. The floating point number with even integral significand is x_1, associated with $M = 4$. Therefore, we have $RN(x) = 0.5$.

2.12 Rounding error of representation

In this section, we compute the rounding error committed with the round to nearest rounding mode and introduce the *unit roundoff*.

Proposition 2.13. *(*Unit roundoff*) Let x be a real number. Assume that the floating point system uses the round to nearest rounding mode. If x is in the normal range, therefore*

$$fl(x) = x(1 + \delta), \qquad |\delta| \leq u, \tag{81}$$

where u is the unit roundoff defined by

$$u = \frac{1}{2}\beta^{1-p}. \tag{82}$$

If x is in the subnormal range, therefore

$$|fl(x) - x| \leq \beta^{e_{min}-p+1}. \tag{83}$$

We emphasize that the previous proposition states that, if x is in the normal range, the relative error can be bounded, while if x is in the subnormal range, the absolute error can be bounded. Indeed, if x is in the subnormal range, the relative error can become large.

Proof. In the first part of the proof, we analyze the case where x is in the normal range and in, the second part, we analyze the subnormal range.

Assume that x is in the normal range. Therefore, the real number x is nonzero, and we can define the real number δ by the equation

$$\delta = \frac{fl(x) - x}{x}. \tag{84}$$

We must prove that $|\delta| \leq \frac{1}{2}\beta^{1-p}$. In fact, we will prove that

$$|fl(x) - x| \leq \frac{1}{2}\beta^{1-p}|x|. \tag{85}$$

Let $\overline{M} = \frac{x}{\beta^{e-p+1}} \in \mathbb{R}$ be the infinitely precise integral significand associated with x. By assumption, the floating point number $fl(x)$ is normal, which implies that there exist two integers M_1 and M_2 such that

$$\beta^{1-p} \leq M_1, M_2 < \beta^p \tag{86}$$

and

$$M_1 \leq \overline{M} \leq M_2 \tag{87}$$

with $M_2 = M_1 + 1$.

One of the two integers M_1 or M_2 has to be the nearest to $\overline{M}$. This implies that either $\overline{M} - M_1 \leq 1/2$ or $M_2 - \overline{M} \leq 1/2$ and we can prove this by contradiction. Indeed, assume that $\overline{M} - M_1 > 1/2$ and $M_2 - \overline{M} > 1/2$. Therefore,

$$(\overline{M} - M_1) + (M_2 - \overline{M}) > 1. \tag{88}$$

The previous inequality simplifies to $M_2 - M_1 > 1$, which contradicts the assumption $M_2 = M_1 + 1$.

Let M be the integer which is the closest to $\overline{M}$. We have

$$M = \begin{cases} M_1, \text{ if } \overline{M} - M_1 \leq 1/2, \\ M_2 \text{ if } M_2 - \overline{M} \leq 1/2. \end{cases} \tag{89}$$

Therefore, we have $|M - \overline{M}| \leq 1/2$, which implies

$$\left| M - \frac{x}{\beta^{e-p+1}} \right| \leq \frac{1}{2}. \tag{90}$$

Hence,

$$\left| M \cdot \beta^{e-p+1} - x \right| \leq \frac{1}{2}\beta^{e-p+1}, \tag{91}$$

which implies

$$|fl(x) - x| \leq \frac{1}{2}\beta^{e-p+1}. \tag{92}$$

By assumption, the number x is in the normal range, which implies $\beta^{p-1} \leq |\overline{M}|$. This implies

$$\beta^{p-1} \cdot \beta^{e-p+1} \leq |\overline{M}| \cdot \beta^{e-p+1}. \tag{93}$$

Hence,

$$\beta^e \leq |x|. \tag{94}$$

We plug the previous inequality into the inequality 92 and get

$$|fl(x) - x| \leq \frac{1}{2}\beta^{1-p}|x|, \tag{95}$$

which concludes the first part of the proof. □

Assume that x is in the subnormal range. We still have the equation 92. But, by opposition to the previous case, we cannot rely the expressions β^e and $|x|$. We can only simplify the equation 92 by introducing the equality $e = e_{min}$, which concludes the proof.

Example 2.15 Consider the floating point system with radix $\beta = 2$, precision $p = 3$ and exponent range $e_{min} = -2$ and $e_{max} = 3$. For this floating point system, with round to nearest, the unit roundoff is $u = \frac{1}{2}2^{1-3} = 0.125$.

Consider the real number $x = 4.5 \cdot 2^{-1-3+1} = 0.5625$. This number falls exactly between the two normal floating point numbers $x_1 = 0.5$ and $x_2 = 0.625$. The round-ties-to-even rule states that $fl(x) = 0.5$. In this case, the relative error is

$$\left| \frac{fl(x) - x}{x} \right| = \left| \frac{0.5 - 0.5625}{0.5625} \right| = 0.11111111\ldots. \tag{96}$$

We see that this relative error is lower than the unit roundoff, which is consistent with the proposition 2.13.

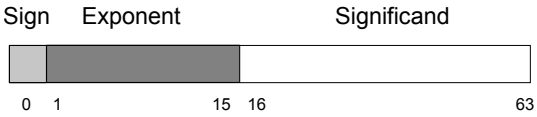

Figure 8: The floating point data format of the single precision number on CRAY-1.

2.13 Other floating point systems

As an example of another floating point system, let us consider the Cray-1 machine. This is will illustrate the kind of difficulties which existed before the IEEE standard. The Cray-1 Hardware reference [7] presents the floating point data format used in this system. The figure 8 presents this format.

The radix of this machine is $\beta = 2$. The figure 8 implies that the precision is $p = 63 - 16 + 1 = 48$. There are 15 bits for the exponent, which would implied that the exponent range would be from $-2^{14} + 1 = -16383$ to $2^{14} = 16384$. But the text indicate that the bias that is added to the exponent is $40000_8 = 4 \cdot 8^4 = 16384$, which shows that the exponent range is from $e_{min} = -2 \cdot 8^4 + 1 = -8191$ to $e_{max} = 2 \cdot 8^4 = 8192$. This corresponds to the approximate exponent range from $2^{-8191} \approx 10^{-2466}$ to $2^{8192} \approx 10^{2466}$. This system is associated with the epsilon machine $\epsilon_M = 2^{1-48} \approx 7 \times 10^{-15}$ and the unit roundoff is $u = \frac{1}{2}2^{1-48} \approx 4 \times 10^{-15}$.

On the Cray-1, the double precision floating point numbers have 96 bits for the significand and the same exponent range as the single. Hence, doubles on this system are associated with the epsilon machine $\epsilon_M = 2^{1-96} \approx 2 \times 10^{-29}$ and the unit roundoff is $u = \frac{1}{2}2^{1-96} \approx 1 \times 10^{-29}$.

3 Floating point numbers in Scilab

In this section, we present the floating point numbers in Scilab. The figure 9 presents the functions related to the management of floating point numbers.

3.1 IEEE doubles

In this section, we present the floating point numbers which are used in Scilab, that is, IEEE 754 doubles.

The IEEE standard 754, published in 1985 and revised in 2008 [19, 11], defines a floating point system which aims at standardizing the floating point formats. Scilab uses the double precision floating point numbers which are defined in this standard.

The parameters of the doubles are presented in the figure 10. The benefits of the IEEE 754 standard is that it increases the portability of programs written in the Scilab language. Indeed, all all machines where Scilab is available, the radix, the precision and the number of bits in the exponent are the same. As a consequence, the machine precision ϵ_M is always equal to the same value for doubles. This is the same for the largest positive normal Ω and the smallest positive normal μ, which are always equal to the values presented in the figure 10. The situation for the smallest positive denormal α is more complex and is detailed in the end of this section.

%inf	Infinity
%nan	Not a number
%eps	Machine precision
fix	rounding towards zero
floor	rounding down
int	integer part
round	rounding
double	convert integer to double
isinf	check for infinite entries
isnan	check for "Not a Number" entries
isreal	true if a variable has no imaginary part
imult	multiplication by i, the imaginary number
complex	create a complex from its real and imaginary parts
nextpow2	next higher power of 2
log2	base 2 logarithm
frexp	computes fraction and exponent
ieee	set floating point exception mode
nearfloat	get previous or next floating-point number
number_properties	determine floating-point parameters

Figure 9: Scilab commands to manage floating point values

Radix β	2
Precision p	53
Exponent Bits	11
Minimum Exponent e_{min}	-1022
Maximum Exponent e_{max}	1023
Largest Positive Normal Ω	$(2 - 2^{1-53}) \cdot 2^{1023} \approx 1.79$D+308
Smallest Positive Normal μ	$2^{-1022} \approx 2.22$D-308
Smallest Positive Denormal α	$2^{-1022-53+1} \approx 4.94D - 324$
Machine Epsilon ϵ_M	$2^{1-53} \approx 2.220$D-16
Unit roundoff u	$2^{-53} \approx 1.110$D-16

Figure 10: Scilab IEEE 754 doubles

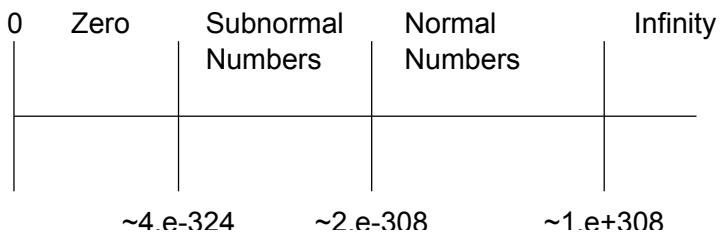

Figure 11: Positive doubles in an exagerately dilated scale.

`x = number_properties(key)`	
`key="radix"`	the radix β
`key="digits"`	the precision p
`key="huge"`	the maximum positive normal double Ω
`key="tiny"`	Smallest Positive Normal μ
`key="denorm"`	a boolean (%t if denormalised numbers are used)
`key="tiniest"`	if denorm is true, the Smallest Positive Denormal α, if not μ
`key="eps"`	Unit roundoff $u = \frac{1}{2}\epsilon_M$
`key="minexp"`	emin
`key="maxexp"`	emax

Figure 12: Options of the `number_properties` function.

The figure 11 presents specific doubles in an an exagerately dilated scale.

In the following script, we compute the largest positive normal, the smallest positive normal, the smallest positive denormal and the machine precision.

```
-->(2-2^(1-53))*2^1023
 ans  =
    1.79D+308
-->2^-1022
 ans  =
    2.22D-308
-->2^(-1022-53+1)
 ans  =
    4.94D-324
-->2^(1-53)
 ans  =
    2.220D-16
```

In practice, it is not necessary to re-compute these constants. Indeed, the `number_properties` function, presented in the figure 12, can do it for us.

In the following session, we compute the largest positive normal double, by calling the `number_properties` function with the `"huge"` key.

```
-->x=number_properties("huge")
 x  =
    1.79D+308
```

In the following script, we perform a loop over all the available keys and display all the properties of the current floating point system.

```
for key = ["radix" "digits" "huge" "tiny" ..
    "denorm" "tiniest" "eps" "minexp" "maxexp"]
  p = number_properties ( key );
  mprintf("%-15s= %s\n",key,string(p))
end
```

In Scilab 5.3, on a Windows XP 32 bits system with an Intel Xeon processor, the previous script produces the following output.

```
radix           = 2
digits          = 53
huge            = 1.79D+308
tiny            = 2.22D-308
denorm          = T
tiniest         = 4.94D-324
eps             = 1.110D-16
minexp          = -1021
maxexp          = 1024
```

Almost all these parameters are the same on most machines on the earth where Scilab is available. The only parameters which might change are `"denorm"`, which can be false, and `"tiniest"`, which can be equal to `"tiny"`. Indeed, gradual underflow is an optional part of the IEEE 754 standard, so that there might be machines which do not support the subnormal numbers.

For example, here is the result of the same Scilab script with a different, non-default, compiling option of the Intel compiler.

```
radix           = 2
digits          = 53
huge            = 1.798+308
tiny            = 2.225-308
denorm          = F
tiniest         = 2.225-308
eps             = 1.110D-16
minexp          = -1021
maxexp          = 1024
```

The previous session appeared in the context of the bugs [6, 1], which have been fixed during the year 2010.

Subnormal floating point numbers are reviewed in more detail in the section 3.4.

3.2 Why 0.1 is rounded

In this section, we present a brief explanation for the following Scilab session. It shows that the mathematical equality $0.1 = 1 - 0.9$ is not exact with binary floating point integers.

```
-->format(25)
-->x1=0.1
 x1  =
    0.1000000000000000055511
-->x2 = 1.0-0.9
 x2  =
```

```
    0.099999999999999777955
-->x1==x2
 ans  =
  F
```

We see that the real decimal number 0.1 is displayed as 0.100000000000000005. In fact, only the 17 first digits after the decimal point are significant : the last digits are a consequence of the approximate conversion from the internal binary double number to the displayed decimal number.

In order to understand what happens, we must decompose the floating point number into its binary components. The IEEE double precision floating point numbers used by Scilab are associated with a radix (or basis) $\beta = 2$, a precision $p = 53$, a minimum exponent $e_{min} = -1023$ and a maximum exponent $e_{max} = 1024$. Any floating point number x is represented as

$$fl(x) = M \cdot \beta^{e-p+1}, \tag{97}$$

where

- e is an integer called the exponent,
- M is an integer called the integral significant.

The exponent satisfies $e_{min} \leq e \leq e_{max}$ while the integral significant satisfies $|M| \leq \beta^p - 1$.

Let us compute the exponent and the integral significant of the number $x = 0.1$. The exponent is easily computed by the formula

$$e = \lfloor \log_2(|x|) \rfloor, \tag{98}$$

where the $\log_2$ function is the base-2 logarithm function. In the case where an underflow or an overflow occurs, the value of e is restricted into the minimum and maximum exponents range. The following session shows that the binary exponent associated with the floating point number 0.1 is -4.

```
-->format(25)
-->x = 0.1
 x  =
    0.1000000000000000055511
-->e = floor(log2(x))
 e  =
  - 4.
```

We can now compute the integral significant associated with this number, as in the following session.

```
-->M = x/2^(e-p+1)
 M  =
    7205759403792794.
```

Therefore, we deduce that the integral significant is equal to the decimal integer $M = 7205759403792794$. This number can be represented in binary form as the 53 binary digit number

$$M = 11001100110011001100110011001100110011001100110011010. \tag{99}$$

We see that a pattern, made of pairs of 11 and 00 appears. Indeed, the real value 0.1 is approximated by the following infinite binary decomposition:

$$0.1 = \left(\frac{1}{2^0} + \frac{1}{2^1} + \frac{0}{2^2} + \frac{0}{2^3} + \frac{1}{2^4} + \frac{1}{2^5} + \dots \right) \cdot 2^{-4}. \tag{100}$$

We see that the decimal representation of $x = 0.1$ is made of a finite number of digits while the binary floating point representation is made of an infinite sequence of digits. But the double precision floating point format must represent this number with 53 bits only.

Notice that, the first digit is not stored in the binary double format, since it is assumed that the number is *normalized* (that is, the first digit is assumed to be one). Hence, the leading binary digit is *implicit*. This is why there is only 52 bits in the mantissa, while we use 53 bits for the precision p. For the sake of simplicity, we do not consider denormalized numbers in this discussion.

In order to analyze how the rounding works, we look more carefully to the integer M, as in the following experiments, where we change only the last decimal digit of M.

```
-->7205759403792793 * 2^(-4-53+1)
 ans  =
    0.099999999999999916733
-->7205759403792794 * 2^(-4-53+1)
 ans  =
    0.10000000000000000055511
```

We see that the exact number 0.1 is between two consecutive floating point numbers:

$$7205759403792793 \cdot 2^{-4-53+1} < 0.1 < 7205759403792794 \cdot 2^{-4-53+1}. \tag{101}$$

There are four rounding modes in the IEEE floating point standard. The default rounding mode is *round to nearest*, which rounds to the nearest floating point number. In case of a tie, the rounding is performed to the only one of these two consecutive floating point numbers whose integral significant is even. In our case, the distance from the exact x to the two floating point numbers is

$$|0.1 - 7205759403792793 \cdot 2^{-4-53+1}| = 8.33 \cdots 10^{-18}, \tag{102}$$

$$|0.1 - 7205759403792794 \cdot 2^{-4-53+1}| = 5.55 \cdots 10^{-18}. \tag{103}$$

(The previous computation is performed with a symbolic computation system, not with Scilab). Therefore, the nearest is $7205759403792794 \cdot 2^{-4-53+1}$, which leads to $fl(0.1) = 0.10000000000000005$.

On the other side, $x = 0.9$ is also not representable as an exact binary floating point number (but 1.0 is exactly represented). The floating point binary representation of $x = 0.9$ is associated with the exponent $e = -1$ and an integral significant between 8106479329266892 and 8106479329266893. The integral significant which is nearest to $x = 0.9$ is 8106479329266893, which is associated with the approximated decimal number $fl(0.9) \approx 0.90000000000000002$.

Then, when we perform the subtraction "1.0-0.9", the decimal representation of the result is $fl(1.0) - fl(0.9) \approx 0.09999999999999997$, which is different from $fl(0.1) \approx 0.10000000000000005$.

We have shown that the mathematical equality $0.1 = 1 - 0.9$ is not exact with binary floating point integers. There are many other examples where this happens. In the next section, we consider the sine function with a particular input.

3.3 Lost properties of arithmetic

In this section, we present some simple experiments where we see the difference between the exact arithmetic and floating point arithmetic.

We emphasize that all the examples presented in this section are showing practical consequences of floating point arithmetic: these are not Scilab bugs, but implicit limitations caused by the limited precision of floating point numbers.

In the following session, we see that the mathematical equality $0.7 - 0.6 = 0.1$ is not satisfied exactly by doubles.

```
-->0.7-0.6 == 0.1
 ans  =
  F
```

This is a straightforward consequence of the fact that 0.1 is rounded, as was presented in the section 3.2. Indeed, let us display more significant digits, as in the following session.

```
-->format("e",25)
-->0.7
 ans  =
    6.9999999999999556D-01
-->0.6
 ans  =
    5.9999999999999778D-01
-->0.7-0.6
 ans  =
    9.9999999999997780D-02
-->0.1
 ans  =
    1.0000000000000056D-01
```

We see that 0.7-0.6 is represented by the floating point number 9.9999999999997780D-02, which is the double before 0.1. Now 0.1 is represented by the double after 0.1, and these two doubles are different.

Let us consider the following session.

```
-->1-0.9 == 0.1
 ans  =
  F
```

This is the same issue as previously, as is shown by the following session, where display more significant digits.

```
-->format("e",25)
-->1-0.9
 ans  =
    9.9999999999997780D-02
-->0.1
 ans  =
    1.0000000000000056D-01
```

In binary floating point arithmetic, multiplication and division by 2 is exact, provided that there is no overflow or underflow. An example is provided in the following session.

```
-->0.9 / 2 * 2 == 0.9
 ans  =
  T
```

This is because this only changes the exponent of floating point representation of the double. Provided that the updated exponent stays within the bounds given by doubles, the updated double is exact.

But this is not true for all multipliers. For example, consider the following session, as shown in the following session.

```
-->0.9 / 3 * 3 == 0.9
 ans  =
  F
```

Commutativity of addition, i.e. $x + y = y + x$, is satisfied by floating point numbers. Associativity with respect to addition,i.e. $(x+y)+z = x+(y+z)$, is not true with floating point numbers.

```
-->(0.1 + 0.2) + 0.3 == 0.1 + (0.2 + 0.3)
 ans  =
  F
```

Associativity with respect to multiplication, i.e. $x * (y * z) = (x * y) * z$ is not true with floating point numbers.

```
-->(0.1 * 0.2) * 0.3 == 0.1 * (0.2 * 0.3)
 ans  =
  F
```

Distributivity of multiplication over addition, i.e. x*(y+z) = x*y + x*z, is lost with floating point numbers.

```
-->0.3 * (0.1 + 0.2) == (0.3 * 0.1) + (0.3 * 0.2)
 ans  =
  F
```

3.4 Overflow and gradual underflow

In the following session, we present numerical experiments with Scilab and extreme numbers.

When we perform arithmetic operations, it may happen that we produce values which are not representable as doubles. This may happen especially in the two following situations.

- If we increase the magnitude of a double, which then becomes too large to be representable as a double, we get an "overflow" and the number is represented by an Infinity IEEE number.
- If we reduce the magnitude of a double, which becomes too small to be representable as a normal double, we get an "underflow" and the accuracy of the floating point number is progressively reduced.

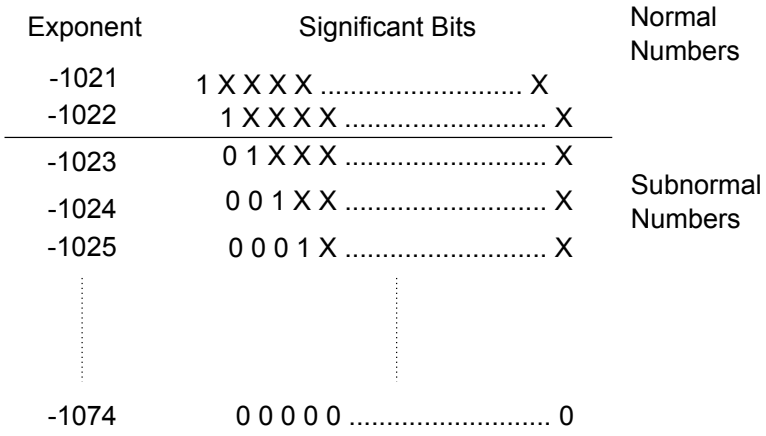

Figure 13: Normal and subnormal numbers.

- If we further reduce the magnitude of a double, even subnormal floating point numbers cannot represent the number and we get zero.

In the following session, we call the `number_properties` function and get the largest positive double. Then we multiply it by two and get the Infinity number.

```
-->x = number_properties("huge")
 x  =
    1.79D+308
-->2*x
 ans  =
    Inf
```

In the following session, we call the `number_properties` function and get the smallest positive subnormal double. Then we divide it by two and get zero.

```
-->x = number_properties("tiniest")
 x  =
    4.94D-324
-->x/2
 ans  =
    0.
```

In the subnormal range, the doubles are associated with a decreasing number of significant digits. This is presented in the figure 13, which is similar to the figure presented by Coonen in [5].

This can be experimentally verified with Scilab. Indeed, in the normal range, the relative distance between two consecutive doubles is either the machine precision $\epsilon_M \approx 10^{-16}$ or $\frac{1}{2}\epsilon_M$: this has been proved in the proposition 2.12.

In the following session, we check this for the normal double `x1=1`.

```
-->format("e",25)
-->x1=1
 x1  =
    1.0000000000000000D+00
-->x2=nearfloat("succ",x1)
 x2  =
    1.0000000000000222D+00
```

```
-->(x2-x1)/x1
 ans  =
    2.22044604925031308 1D-16
```

This shows that the spacing between `x1=1` and the next double is ϵ_M.

The following session shows that the spacing between `x1=1` and the previous double is $\frac{1}{2}\epsilon_M$.

```
-->format("e",25)
-->x1=1
 x1  =
    1.0000000000000000 00D+00
-->x2=nearfloat("pred",x1)
 x2  =
    9.9999999999999889 0D-01
-->(x1-x2)/x1
 ans  =
    1.1102230246251565 40D-16
```

On the other hand, in the subnormal range, the number of significant digits is reduced from 53 to zero. Hence, the relative distance between two consecutive subnormal doubles can be much larger. In the following session, we compute the relative distance between two subnormal doubles.

```
-->x1=2^-1050
 x1  =
    8.28D-317
-->x2=nearfloat("succ",x1)
 x2  =
    8.28D-317
-->(x2-x1)/x1
 ans  =
    5.960D-08
```

Actually, the relative distance between two doubles can be as large as 1. In the following experiment, we compute the relative distance between two consecutive doubles in the extreme range of subnormal numbers.

```
-->x1=number_properties("tiniest")
 x1  =
    4.94D-324
-->x2=nearfloat("succ",x1)
 x2  =
    9.88D-324
-->(x2-x1)/x1
 ans  =
    1.
```

3.5 Infinity, Not-a-Number and the IEEE mode

In this section, we present the `%inf`, `%nan` constants and the `ieee` function.

For obvious practical reasons, it is more convenient if a floating point system is closed. This means that we do not have to rely on an exceptionnal routine if an invalid operation is performed. This is why the Infinity and the NaN floating point numbers are defined by the IEEE 754 standard.

m=ieee() ieee(m)	
0	floating point exception produces an error
1	floating point exception produces a warning
2	floating point exception produces Inf or Nan

Figure 14: Options of the `ieee` function.

The IEEE 754 2008 standard states that "the behavior of infinity in floating-point arithmetic is derived from the limiting cases of real arithmetic with operands of arbitrarily large magnitude, when such a limit exists."

In Scilab, the Infinity number is represented by the `%inf` variable. This number is associated with an arithmetic, such that the +, -, * and / operators are available for this particular double. In the following example, we perform some common operations on the infinity number.

```
-->1+%inf
 ans  =
    Inf
-->%inf + %inf
 ans  =
    Inf
-->2 * %inf
 ans  =
    Inf
-->%inf / 2
 ans  =
    Inf
-->1 / %inf
 ans  =
    0.
```

The infinity number may also be produced by dividing a non-zero double by zero. But, by default, if we simply perform this division, we get a warning, as shown in the following session.

```
-->1/+0
     !--error 27
Division by zero...
```

We can configure the behavior of Scilab when it encounters IEEE exceptions, by calling the `ieee` function which is presented in the figure 14.

In the following session, we configure the IEEE mode so that Inf or Nan are produces instead of errors or warnings. Then we divide 1 by 0 and produce the Infinity number.

```
-->ieee(2)
-->1/+0
 ans  =
    Inf
```

Most elementary functions associated with singularities are sensitive to the IEEE mode. In the following session, we check that we can produce an Inf double by

computing the logarithm of zero.

```
-->ieee(0)
-->log(0)
       !--error 32
Singularity of log or tan function.
-->ieee(2)
-->log(0)
 ans  =
  - Inf
```

All invalid operations return a NaN, meaning Not-A-Number. The IEEE 754 2008 standard defines two NaNs, the *signaling* and the *quiet* NaN.

- Signaling NaNs cannot be the result of arithmetic operations. They can be used, for example, to signal the use of uninitialized variables.
- Quiet NaNs are designed to propagate through all operations without signaling an exception. A quiet NaN is produced whenever an invalid operation occurs.

In the Scilab language, the variable `%nan` contains a quiet NaN. There are several simple operations which are producing quiet NaNs. In the following session, we perform an operation which may produce a quiet Nan.

```
-->0/0
    !--error 27
Division by zero...
```

This is because the default behavior of Scilab is to generate an error. If, instead, we configure the IEEE mode to 2, we are able to produce a quiet Nan.

```
-->ieee(2)
-->0/0
 ans  =
    Nan
```

In the following session, we perform various arithmetic operations which produce NaNs.

```
-->ieee(2)
-->%inf - %inf
 ans  =
    Nan
-->0*%inf
 ans  =
    Nan
-->%inf*0
 ans  =
    Nan
-->%inf/%inf
 ans  =
    Nan
-->modulo(%inf,2)
 ans  =
    Nan
```

In the IEEE standard, it is suggested that the square root of a negative number should produce a quiet NaN. In Scilab, we can manage complex numbers, so that

the square root of a negative number is not a NaN, but is a complex number, as in the following session.

```
-->sqrt(-1)
 ans  =
    i
```

Once that a NaN has been produced, it is propagated through the operations until the end of the computation. This is because it can be the operand of any arithmetic statement or the input argument of any elementary function.

```
-->%nan+2
 ans  =
    Nan
-->sqrt(%nan)
 ans  =
    Nan
```

The NaN has a particular property: this is the only double `x` for which the condition `x==x` is false. This is shown in the following session.

```
-->%nan == %nan
 ans  =
  F
```

This is why we cannot use the condition `x==%nan` to check if `x` is a Nan. Fortunately, the `isnan` is designed for this specific purpose, as shown in the following session.

```
-->isnan(1)
 ans  =
  F
-->isnan(%nan)
 ans  =
  T
```

3.6 Machine epsilon

The machine epsilon ϵ_M is a parameter which gives the spacing between floating point numbers. The theory behind this topic has been presented in the section 2.10.

In Scilab, the machine epsilon is given by the `%eps` variable. In the following session, we check that the floating point number which is next to 1, that is 1^+ is `1+%eps`.

```
-->1+%eps > 1
 ans  =
  T
-->1+%eps/2 == 1
 ans  =
  T
```

On the other hand, the number which is just before 1, that is 1^-, is `1-%eps/2`.

```
-->1-%eps/2 < 1
 ans  =
  T
-->1-%eps/4 == 1
 ans  =
  T
```

We emphasize that the number_properties("eps") statement returns the unit roundoff, which is twice the machine epsilon given by %eps.

```
-->eps = number_properties("eps")
 eps  =
    1.110D-16
-->%eps
 %eps  =
    2.220D-16
```

3.7 Not a number

In this section, we emphasize that, in general, any arithmetic operations involving a Nan on input propagates this Nan on output.

We have already seen how an invalid operations, such as 0/0 for example, produces a Not-a-number. In earlier floating point systems, on some machines, such an exception generated an error message and stoped the computation. In general, this is a behavior which is considered as annoying, since the computation may have been continued beyond this exception. In the IEEE 754 standard, the Nan number is associated with an arithmetic, so that the computation can be continued. The standard states that quiet Nans should be propagated through arithmetic operations, with the output equal to another quiet Nan.

In the following session, we perform several basic arithmetic operations, where the input argument is a Nan. We can check that, each time, the output is also equal to Nan.

```
-->1+%nan
 ans  =
    Nan
-->2*%nan
 ans  =
    Nan
-->2/%nan
 ans  =
    Nan
```

3.8 Signed zeros

There are two zeros, the negative zero -0 and the positive zero +0.

Indeed, the IEEE 754 standard states that zero must be associated by a zero significand M and a minimum exponent e. For double precision floating point numbers, this corresponds to $e = -1022$. But this leaves two possible representations of zero: the sign $s = 0$, which leads to the positive zero $(-1)^0 \times 0 \times 2^{-1022}$ and the sign $s = 0$, which leads to the negative zero $(-1)^1 \times 0 \times 2^{-1022}$. Hence, the sign bit s leads to two different zeros.

In Scilab, the statement $+0$ creates a positive zero, while the statement -0 creates a negative zero. These two numbers are different, as shown in the following session. We invert the positive and negative zeros, which leads to the positive and negative infinite numbers.

```
-->ieee(2)
-->1/+0
 ans  =
    Inf
-->1/-0
 ans  =
  - Inf
```

The previous session corresponds to the mathematical limit of the function $1/x$, when x comes near zero. The positive zero corresponds to the limit $\lim 1/x = +\infty$ when $x \to 0^+$, and the negative zero corresponds to the limit $\lim 1/x = -\infty$ when $x \to 0^-$. Hence, the negative and positive zeros lead to a behavior which corresponds to the mathematical sense.

Still, there is a specific point which is different from the mathematical point of view. For example, the equality operator `==` ignores the sign bit of the positive and negative zeros, and consider that they are equal.

```
-->-0 == +0
 ans  =
  T
```

It may create weird results, such as with the `gamma` function.

```
-->gamma(-0) == gamma(+0)
 ans  =
  F
```

The explanation is simple: the negative and positive zeros lead to different values of the `gamma` function, as show in the following session.

```
-->gamma(-0)
 ans  =
  - Inf
-->gamma(+0)
 ans  =
    Inf
```

All in all, we have found two numbers `x` and `y` such that `x==y` but `gamma(x)<>gamma(y)`.

We emphasize that the sign bit of zero can be used to consistently take into account for branch cuts of inverse complex functions, such as `asin` for example. This use of signed zeros is presented by Kahan in [12] and will not be presented further in this document.

3.9 Infinite complex numbers

In this section, we consider exceptionnal complex numbers involving `%inf` or `%nan` real or imaginary parts. We present the `complex` function, which is designed to manage this situation.

Assume that we want to create a complex number, where the real part is zero and the imaginary part is infinite. In the following session, we try to do this, by multiplying the imaginary number `%i` and the infinite number `%inf`.

```
-->ieee(2)
-->%i * %inf
 ans  =
    Nan + Inf
```

The output number is obviously not what we wanted. Surprisingly, the result is consistent with ordinary complex arithmetic and the arithmetic of IEEE exceptional numbers. Indeed, the multiplication `%i*%inf` is performed as the multiplication of `0+%i*1` and `%inf+%i*0`. Then Scilab applies the rule $(a+ib)*(c+id) = (ac-bd)+i(ad+bc)$. Therefore, the real part is `0*%inf-1*0`, which simplifies into `%nan-0` and produces `%nan`. On the other hand, the imaginary part is `0*0+1*%inf`, which simplifies into `0+%inf` and produces `%inf`.

In this case, the `complex` function must be used. Indeed, for any doubles `a` and `b`, the statement `x=complex(a,b)` creates the complex number `x` which real part is `a` and which imaginary part is `b`. Hence, it creates the number `x` without applying the statement `x=a+%i*b`, that is, without using common complex arithmetic. In the following session, we create the number $1+2i$.

```
-->complex(1,2)
 ans  =
    1. + 2.i
```

With this function, we can now create a number with a zero real part and an infinite imaginary part.

```
-->complex(0,%inf)
 ans  =
    Infi
```

Similarily, we can create a complex number where both the real and imaginary part are infinite. With common arithmetic, the result is wrong, while the `complex` function produces the correct result.

```
-->%inf + %i*%inf
 ans  =
    Nan + Inf
-->complex(%inf,%inf)
 ans  =
    Inf + Inf
```

3.10 Notes and references

The "Handbook of floating point arithmetic" [17] is a complete reference on the subject. Higham presents in [10] an excellent discussion on this topic. Moler's book [15] is a lively and concise discussion of this topic in the context of Matlab. Stewart gives in [20] a discussion on floating point numbers. In his Lecture 6, he presents floating point numbers, overflow and underflow and rounding errors. In [8], Forsythe, Malcolm and Moler present floating point numbers in the chapter 2 "Floating point numbers". Their examples are extremely interesting. Goldberg presents in [9] a complete overview of floating point numbers and associated issues. The pioneering work of Wilkinson on this topic is presented in [22]. The section 3.2 was first published in [4] and is reproduced here for consistency.

In the exercizes of the section 3.11, we are going to analyze the Pythagorean sum $\sqrt{a^2+b^2}$. The paper by Moler and Morrison 1983 [16] gives an algorithm to compute the Pythagorean sum without computing their squares or their square roots. Their algorithm is based on a cubically convergent sequence. The BLAS

linear algebra suite of routines [13] includes the SNRM2, DNRM2 and SCNRM2 routines which conpute the euclidian norm of a vector. These routines are based on Blue [2] and Cody [3]. In his 1978 paper [2], James Blue gives an algorithm to compute the Euclidian norm of a n-vector $\|x\| = \sqrt{\sum_{i=1,n} x_i^2}$. The exceptionnal values of the *hypot* operator are defined as the Pythagorean sum in the IEEE 754 standard [11, 19]. The *__ieee754_hypot(x,y)* C function is implemented in the Fdlibm software library [21] developed by Sun Microsystems and available at netlib. This library is used by Matlab [14] and its *hypot* command.

3.11 Exercises

Exercise 3.1 (*Operating systems*) Consider a 32-bits operating system on a personal computer where we use Scilab.

- How many bits are used to store a double precision floating point number? Consider a 64-bits operating system where we use Scilab.
- How many bits are used to store a double ?

Exercise 3.2 (*The hypot function (part 1)*) In this exercize, we analyse the computation of the Pythagorean sum, which is used in two different computations, that is the norm of a complex number and the 2-norm of a vector of real values.

The Pythegorean sum of two real numbers a and b is defined by

$$h(a,b) = \sqrt{a^2 + b^2}. \tag{104}$$

- Define a Scilab function to compute this function.
- Test it with the input $a = 1$, $b = 1$. Does the result corresponds to the expected result ?
- Test it with the input $a = 10^{200}$, $b = 1$. Does the result corresponds to the expected result and why ?
- Compute it with the input $a = 10^{-200}$, $b = 10^{-200}$. Does the result corresponds to the expected result and why ?

Exercise 3.3 (*The hypot function (part 2)*) We see that the Pythagorean sum function h overflows when a or b is large. To solve this problem, we suggest to scale the computation by a or b, depending on which has the largest magnitude. If a has the largest magnitude, we consider the expression

$$h(a,b) = \sqrt{a^2(1 + \frac{b^2}{a^2})} \tag{105}$$

$$= |a|\sqrt{1 + r^2}, \tag{106}$$

where $r = \frac{b}{a}$.

- Derive the expression when b has the largest magnitude.
- Create a Scilab function which implements this algorithm.
- Test it with the input $a = 1$, $b = 0$.
- Test it with the input $a = 10^{200}$, $b = 1$.

Exercise 3.4 (*The hypot function (part 3)*) The Pythagorean sum of a and b is obviously equal to the magnitude of the complex number $a + ib$.

- Compute it with the input $a = 1$, $b = 0$.
- Compute it with the input $a = 10^{200}$, $b = 1$.

- What can you deduce ?

Exercise 3.5 (*The hypot function (part 4)*) The paper by Moler and Morrison 1983 [16] gives an algorithm to compute the Pythagorean sum $a \oplus b = \sqrt{a^2 + b^2}$ without computing their squares or their square roots. Their algorithm is based on a cubically convergent sequence. The following Scilab function implements their algorithm.

```
function y = myhypot2(a,b)
  p = max(abs(a),abs(b))
  q = min(abs(a),abs(b))
  while (q<>0.0)
    r = (q/p)^2
    s = r/(4+r)
    p = p + 2*s*p
    q = s * q
  end
  y = p
endfunction
```

- Print the intermediate iterations for `p` and `q` with the inputs $a = 4$ and $b = 3$.
- Test it with the inputs from exercize 3.2.

Exercise 3.6 (*Decomposition of* π) The `floatingpoint` module is an ATOMS module which allows to analyze floating point numbers. The goal of this exercize is to use this module to compute the decomposition of the double precision floating point representation of π. To install it, please use the statement:

```
atomsInstall("floatingpoint");
```

and restart Scilab.

- The `flps_systemnew` function allows to create a new virtual floating point system. Create a floating point system associated with the parameters of IEEE-754 doubles.
- The `flps_numbernew` function allows to create a floating point number associated with a given floating point sytem. The `flpn = flps_numbernew ( "double" , flps , x )` calling sequence creates a floating point number associated with a given double `x`. Use this function to compute the floating point decomposition of the double `%pi`.
- Manually check the result computed by `flps_numbernew`.
- Find the two consecutive doubles p_1 and p_2 which are so that $p_1 < \pi < p_2$.
- Cross-check the value of p_2 with the `nearfloat` function, by searching for the double which is just after `%pi`.
- Compute $\pi - p_1$ and $p_2 - \pi$.
- Why does `%pi` is equal to p_1 ?

Exercise 3.7 (*Value of* $\sin(\pi)$) In the following session, we compute the numerical value of $sin(\pi)$.

```
-->sin(%pi)
 ans  =
    1.225D-16
```

- Compute `p2` with the `nearfloat` function, by searching for the double which is just after `%pi`.
- Compute `sin(p2)`.
- Based on what we learned in the exercize 3.6, can you explain this ?

3.12 Answers to exercises

Answer of Exercise 3.1 (*Operating systems*) Consider a 32-bits operating system where we use Scilab. How many bits are used to store a double ? Given that most personal computers are IEEE-754 compliant, it must use 64 bits to store a double precision floating point number. This is why the standard calls them *64 bits binary floating point numbers*. Consider a 64-bits operating system where we use Scilab. How many bits are used to store a double ? This machine also uses *64 bits binary floating point numbers* (like the 32 bits machine). The reason behind this is that 32 bits and 64 bits operating systems refer to the way the memory is organized: it has nothing to do with the number of bits for a floating point number. □

Answer of Exercise 3.2 (*The hypot function (part 1)*) The following function is a straightforward implementation of the hypot function.

```
function y = myhypot_naive(a,b)
  y = sqrt(a^2+b^2)
endfunction
```

As we are going to check soon, this implementation is not very robust. First, we test it with the input $a = 1$, $b = 1$.

```
-->myhypot_naive(1,1)
 ans  =
    1.4142136
```

This is the expected result, so that we may think that our implementation is correct.

Second, we test it with the input $a = 10^{200}$, $b = 1$.

```
-->myhypot_naive(1.e200,1)
 ans  =
    Inf
```

This is obviously the wrong result, since the expected result must be exactly equal to 10^{200}, since 1 is much smaller than 10^{200}. Hence, the relative precision of doubles is so that the result must exactly be equal to 10^{200}. This is not the case here. Obviously, this is caused by the overflow of a^2.

```
-->a=1.e200
 a  =
    1.00D+200
-->a^2
 ans  =
    Inf
```

Indeed, the mathematical value $a^2 = 10^{400}$ is much larger than the overflow threshold Ω which is roughly equal to 10^{308}. Hence, the floating point representation of a^2 is the floating point number `Inf`. Then the sum $a^2 + b^2$ is computed as `Inf+1`, which evaluates as `Inf`. Finally, we compute the square root of $a^2 + b^2$, which is equal to `sqrt(Inf)`, which is equal to `Inf`.

Third, let us test it with the input $a = 10^{-200}$, $b = 10^{-200}$.

```
-->myhypot_naive(1.e-200,1.e-200)
 ans  =
    0.
```

This time, again, the result is wrong since the exact result is 10^{-200}. The cause of this failure is presented in the following session.

```
-->a=1.e-200
 a  =
    1.00D-200
-->a^2
 ans  =
    0.
```

Indeed, the exact result is 10^{-400}, which is much smaller than the smallest nonzero double, which is roughly equal to 10^{-324}. Hence, the floating point representation of a^2 is zero, that is, an underflow occured. □

Answer of Exercise 3.3 (*The hypot function (part 2)*) If b has the largest magnitude, we consider the expression

$$h(a,b) = \sqrt{b^2(\frac{a^2}{b^2}+1)} \quad (107)$$

$$= |b|\sqrt{r^2+1}, \quad (108)$$

where $r = \frac{a}{b}$. The following function implements this algorithm.

```
function y = myhypot(a,b)
  if (a==0 & b==0) then
    y = 0;
  else
    if (abs(b)>abs(a)) then
      r = a/b;
      t = abs(b);
    else
      r = b/a;
      t = abs(a);
    end
    y = t * sqrt(1 + r^2);
  end
endfunction
```

Notice that we must take into account the case where a and b are zero, since, in this case, we cannot scale neither by a, nor by b. □

Answer of Exercise 3.4 (*The hypot function (part 3)*) Let us analyze the following Scilab session.

```
-->abs(1+%i)
 ans  =
    1.4142136
```

The previous session shows that the absolute value algorithm is correct in the case where $a = 1$, $b = 1$. Let us see what happens with the input $a = 10^{200}$, $b = 1$.

```
-->abs(1.e200+%i)
 ans  =
    1.00D+200
```

The previous result shows that a robust implementation of the Pythagorean sum algorithm is provided by the `abs` function. This is easy to check, since we have access to the source code of Scilab. If we look at the file `elementary_functions/sci_gateway/fortran/sci_f_abs.f`, we see that, in the case where the input argument is complex, we call the `dlapy2` function. This function comes from the Lapack API and computes $\sqrt{a^2+b^2}$, using the scaling method that we have presented. More precisely, a simplified source code is presented below.

```
      DOUBLE PRECISION FUNCTION DLAPY2( X, Y )
          [...]
      XABS = ABS( X )
      YABS = ABS( Y )
      W = MAX( XABS, YABS )
      Z = MIN( XABS, YABS )
      IF( Z.EQ.ZERO ) THEN
         DLAPY2 = W
      ELSE
```

```
            DLAPY2 = W*SQRT( ONE+( Z / W )**2 )
         END IF
         END
```

□

Answer of Exercise 3.5 (*The hypot function (part 4)*)
The following modified function prints the intermediate variables `p` and `q` and returns the extra argument `i`, which contains the number of iterations.

```
function [y,i] = myhypot2_print(a,b)
  p = max(abs(a),abs(b))
  q = min(abs(a),abs(b))
  i = 0
  while (q<>0.0)
    mprintf("%d %.17e %.17e\n",i,p,q)
    r = (q/p)^2
    s = r/(4+r)
    p = p + 2*s*p
    q = s * q
        i = i + 1
  end
  y = p
endfunction
```

The following session shows the intermediate iterations for the input $a = 4$ and $b = 3$. Moler and Morrison state their algorithm never performs more than 3 iterations for numbers with less that 20 digits.

```
-->myhypot2_print(4,3)
0 4.00000000000000000e+000 3.00000000000000000e+000
1 4.98630136986301410e+000 3.69863013698630120e-001
2 4.99999997418825260e+000 5.08052632941535820e-004
3 5.00000000000000090e+000 1.31137265239709110e-012
4 5.00000000000000090e+000 2.25516523372845020e-038
5 5.00000000000000090e+000 1.14692522126238280e-115
```

The following session shows the result for the difficult cases presented in the exercize 3.2.

```
-->myhypot2(1,1)
 ans  =
    1.4142136
-->myhypot2(1,0)
 ans  =
    1.
-->myhypot2(0,1)
 ans  =
    1.
-->myhypot2(0,0)
 ans  =
    0.
-->myhypot2(1,1.e200)
 ans  =
    1.00D+200
-->myhypot2(1.e-200,1.e-200)
 ans  =
    1.41D-200
```

We can see that the algorithm created by Moler and Morrison perfectly works in these cases. □

Answer of Exercise 3.6 (*Decomposition of π*) In the following session, we create a virtual floating point system associated with IEEE doubles.

```
-->flps = flps_systemnew ( "IEEEdouble" )
 flps  =
Floating Point System:
======================
radix= 2
p=     53
emin=  -1022
emax=  1023
vmin=  2.22D-308
vmax=  1.79D+308
eps=   2.220D-16
r=     1
gu=    T
alpha= 4.94D-324
ebits= 11
```

In the following session, we compute the floating point representation of π with doubles. First, we configure the formatting of numbers with the `format` function, so that all the digits of long integers are displayed. Then we compute the floating point representation of π.

```
-->format("v",25)
-->flpn = flps_numbernew ( "double" , flps , %pi )
 flpn  =
Floating Point Number:
======================
s= 0
M= 7074237752028440
m= 1.5707963267948965579990
e= 1
flps= floating point system
======================
Other representations:
x= (-1)^0 * 1.5707963267948965579990 * 2^1
x= 7074237752028440 * 2^(1-53+1)
Sign= 0
Exponent= 10000000000
Significand= 1001001000011111101101010100010001000010110100011000
Hex= 400921FB54442D18
```

It is straightforward to check that the result computed by the `flps_numbernew` function is exact.

```
-->7074237752028440 * 2^-51 == %pi
 ans  =
  T
```

We know that the decimal representation of the mathematical π is defined by an infinite number of digits. Therefore, it is obvious that it cannot be be represented by a finite number of digits. Hence, $7074237752028440 * 2^-51 \neq \pi$, which implies that `%pi` is not exactly equal to π. Instead, `%pi` is the best possible double precision floating point representation of π. In fact, we have

$$p_1 = 7074237752028440 \cdot 2^{-51} < \pi < p_2 = 7074237752028441 \cdot 2^{-51}, \tag{109}$$

which means that π is between two doubles. This is easy to check this with a symbolic computing system, such as XCas or Maxima, for example, or with an online system such as http://www.wolframalpha.com. We can cross-check our result with the `nearfloat` function.

```
-->p2 = nearfloat("succ",%pi)
 p2  =
    3.1415927
-->7074237752028441 * 2^-51 == p2
```

```
ans  =
 T
```

We have

$$p_2 - \pi = 3.2162\ldots \cdot 10^{-16}, \quad (110)$$

$$\pi - p_1 = 1.22464\ldots \cdot 10^{-16}. \quad (111)$$

Which means that p_1 is closer to π than p_2. Scilab rounds to the nearest float, and this is why `%pi` is equal to p_1. □

Answer of Exercise 3.7 (*Value of* $\sin(\pi)$) In the following session, we compute `p2`, the double which is just after `%pi`. Then we compute `sin(p2)` and compare our result with `sin(%pi)`.

```
-->p2 = nearfloat("succ",%pi)
 p2  =
    3.1415927
-->sin(p2)
 ans  =
  - 3.216D-16
-->sin(%pi)
 ans  =
    1.225D-16
```

Since `%pi` is not exactly equal to π but is on the left of π, it is obvious that `sin(%pi)` cannot be zero. Since the sin function is decreasing in the neighbourhood of π, this explains why `sin(%pi)` is positive and explains why `p2` is negative. □

References

[1] Michael Baudin. Denormalized floating point numbers are not present in scilab's master. http://bugzilla.scilab.org/show_bug.cgi?id=6934, April 2010.

[2] James L. Blue. A portable fortran program to find the euclidean norm of a vector. *ACM Trans. Math. Softw.*, 4(1):15–23, 1978.

[3] W. J. Cody. *Software for the Elementary Functions*. Prentice Hall, 1971.

[4] Michael Baudin Consortium Scilab Digitéo. Scilab is not naive. http://forge.scilab.org/index.php/p/docscilabisnotnaive/.

[5] J. T. Coonen. Underflow and the denormalized numbers. *Computer*, 14(3):75–87, March 1981.

[6] Allan Cornet. There are no subnormal numbers in the master. http://bugzilla.scilab.org/show_bug.cgi?id=6937, April 2010.

[7] CRAY. Cray-1 hardware reference, November 1977.

[8] George Elmer Forsythe, Michael A. Malcolm, and Cleve B. Moler. Computer methods for mathematical computations.

[9] David Goldberg. *What Every Computer Scientist Should Know About Floating-Point Arithmetic*. Association for Computing Machinery, Inc., March 1991. http://www.physics.ohio-state.edu/~dws/grouplinks/floating_point_math.pdf.

[10] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002.

[11] IEEE Task P754. *IEEE 754-2008, Standard for Floating-Point Arithmetic*. IEEE, New York, NY, USA, August 2008.

[12] W. Kahan. Branch cuts for complex elementary functions, or much ado about nothing's sign bit., 1987.

[13] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. Technical report, University of Texas at Austin, Austin, TX, USA, 1977.

[14] The Mathworks. Matlab - hypot : square root of sum of squares. http://www.mathworks.com.

[15] Cleve Moler. Numerical computing with matlab.

[16] Cleve B. Moler and Donald Morrison. Replacing square roots by pythagorean sums. *IBM Journal of Research and Development*, 27(6):577–581, 1983.

[17] Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010.

[18] Wolfram Research. Wolfram alpha. http://www.wolframalpha.com.

[19] David Stevenson. Ieee standard for binary floating-point arithmetic, August 1985.

[20] G. W. Stewart. *Afternotes on Numerical Analysis*. SIAM, 1996.

[21] Inc. Sun Microsystems. A freely distributable c math library, 1993. http://www.netlib.org/fdlibm.

[22] J. Wilkinson. *Rounding Errors In Algebraic Processes*. Prentice-Hall, 1964.

Index