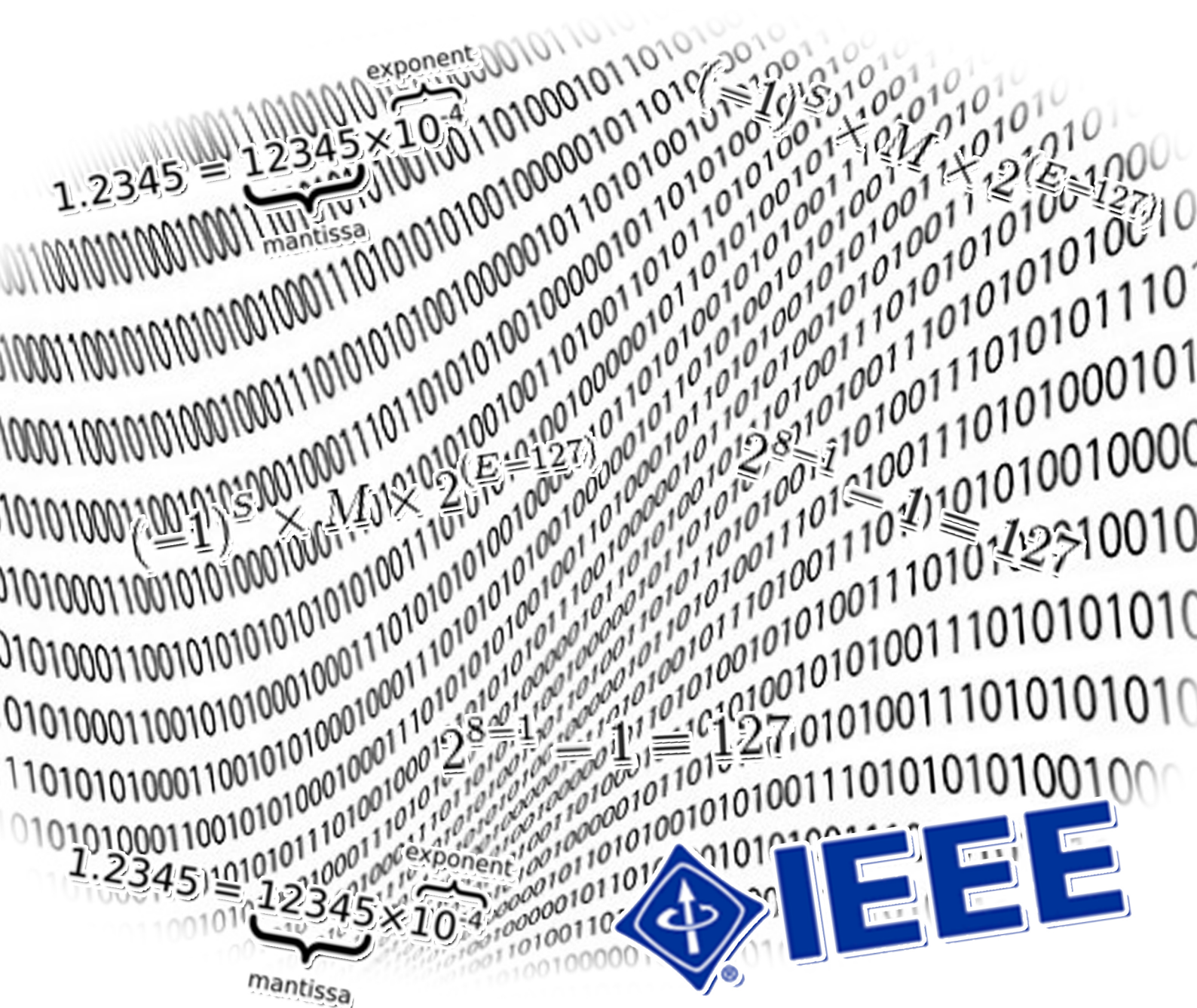


GM3

TP1 D'ALGORITHMIQUE NUMÉRIQUE



Introduction

Depuis la création des « bombes » destinées à déchiffrer des messages codés pendant la seconde guerre mondiale, les appareils informatiques ont connu un essor exponentiel. Non seulement l'informatique est aujourd'hui omniprésente, mais elle continue d'évoluer de jour en jour (les conjectures de Moore évaluent que la puissance brute des machines doublera tout les deux ans). D'abord pensés pour réaliser des calculs, il est logique de vouloir que nos ordinateurs manient des objets aussi abstraits que les nombres réels. Or, il est difficile de gérer des nombres à virgules en binaire, d'où la nécessité d'imposer une norme de codage de ces nombres (on pense notamment aux incidents de la bourse de Vancouver, du missile Patriote ou encore de la fusée Ariane). C'est la raison pour laquelle on a vu apparaître ces dernières années la norme *IEEE 754* destinée à régulariser la représentation des nombres à virgule flottante en binaire.

Cette convention consiste à représenter en machine les nombres à virgules flottantes (ou « flottants ») de la manière suivante : $(-1)^s \times (1 + m) \times 2^{(e-d)}$.

Avec :

- un nombre s sur un bit pour déterminer le signe (1 pour négatif, 0 pour positif) ;
- m pour la valeur décimale de la « mantisse » : la partie fractionnaire en notation binaire scientifique. La mantisse tient sur plusieurs bits ; la convention est de 23 bits pour un stockage du nombre sur 32 bits (respectivement 52 pour 64) ;
- e pour l'exposant signé de la puissance de 2, codé sur 8 bits en simple précision (respectivement 11 en double) ;
- d pour le décalage de l'exposant afin de le stocker sous forme d'un nombre non-signé (codage de l'exposant par excès). Si l'on note b le nombre de bits codant l'exposant, on a $d = 2^{b-1} - 1$.

Dans ce cas, on dira que le nombre flottant est « normalisé ». Pour un nombre très proche de 0 (i.e. dont l'exposant dépasse la capacité du format), on qualifiera le flottant de « dénormalisé » et on prendra une représentation légèrement différente pour pouvoir le coder : $(-1)^s \times m \times 2^{(1-d)}$ (l'exposant est représenté comme étant minimal mais sa valeur est $1 - d$ pour assurer une sorte de continuité entre le zéro, les nombres dénormalisés et les nombres normalisés). Le premier bit de la mantisse qui est implicitement 1 en normalisé est ici, de manière implicite aussi, 0.

Dans ce TP, on se propose d'étudier différents flottants ; tant bien en s'intéressant à leurs représentations en binaire, mais aussi en étudiant leurs valeurs exactes et décimales. Nous nous attarderons, dans un premier temps, sur des flottants caractéristiques de la norme *IEEE 754* :

- l' $\epsilon_{machine}$;
- X_{min} normalisé/dénormalisé ;
- X_{max} normalisé/dénormalisé ;
- le zéro : 0 et l'unité : 1.

Puis, nous prendrons un autre exemple plus concret : nous nous intéresserons au codage de 0.1. Ensuite, nous regarderons comment est codé une somme de termes inférieurs à l' $\epsilon_{machine}$. Enfin, nous étudierons le codage de l'unité ajouté à cette somme.

Table des matières

Introduction	2
1 Étude de flottants caractéristiques	4
1.1 $L'\epsilon_{machine}$	4
1.2 X_{min}	7
1.3 X_{max}	12
1.4 Le zéro et l'unité	18
2 Le nombre flottant 0.1	23
2.1 Le 0.1 machine en simple précision	23
2.2 Le 0.1 machine en double précision	25
3 Étude d'une somme de termes inférieurs à $l'\epsilon_{machine}$	27
3.1 Pour une somme de 2 termes	28
3.2 Pour une somme de 10 termes	29
3.3 Pour une somme de 100 termes	31
3.4 Pour une somme de 1000 termes	32
3.5 Remarque : pour n assez grand	33
4 Ajout de l'unité à une somme de termes inférieurs à $l'\epsilon_{machine}$	35
4.1 Pour une somme de 2 termes	36
4.2 Pour les autres sommes	36
Conclusion	38
Bibliographie	39

Étude de flottants caractéristiques

Dans cette partie, on s'intéresse à des flottants particuliers. Ils sont importants dans le sens où ils définissent les limites du calcul scientifique sur machine.

1.1 $L'\epsilon_{machine}$

$L'\epsilon_{machine}$ est le plus petit réel positif-machine tel que : $1 + \epsilon_{machine} > 1$ sur machine. On développe un petit programme C pour pouvoir l'approximer, à un ordre de magnitude près :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  float epsilonMachineSimple() {
5      float z = 1.;
6      while(1 + z/2 > 1) {
7          z/=2;
8      }
9      return z;
10 }
11
12 double epsilonMachineDouble() {
13     double z = 1.;
14     while(1 + z/2 > 1) {
15         z/=2;
16     }
17     return z;
18 }
19
20 int main(int argc, char* argv[]){
21     float eps1 = epsilonMachineSimple();
22     double eps2 = epsilonMachineDouble();
23     printf("Epsilon machine (simple precision) : %g\n", eps1);
24     printf("Epsilon machine (double precision) : %g\n", eps2);
25     return EXIT_SUCCESS;
26 }
```

Ce qui nous donne comme sortie à l'écran :

```
Epsilon machine (simple precision) : 1.19209e-07
Epsilon machine (double precision) : 2.22045e-16
```

FIGURE 1.1 – Résultat affiché par le programme

On voudrait connaître la représentation binaire de ce flottant. Pour cela on passe par le débogueur gdb (sans oublier de rajouter l'option -g en flag dans le makefile). On commence par poser un break sur la ligne où est affichée la variable que l'on souhaite étudier (ici, ligne 23 pour la simple précision et 24 pour la double précision), puis on lance le programme avec la commande *run*. La commande *x/tw* pour un simple, respectivement *x/tg* pour un double, suivie de la valeur de l'adresse de cette variable (opérateur &) nous permet de récupérer son codage en binaire.

1.1.1 Simple précision

Codage binaire

En codage 32 bits, on a :

```
(gdb) x/tw &eps1
0x7fffffffdddf4: 00110100000000000000000000000000
(gdb) █
```

FIGURE 1.2 – Codage binaire de notre approximation de l' $\epsilon_{machine}$ en simple précision

On en déduit donc que le nombre flottant, en simple précision :

- est bien composé de 32 bits ;
- est positif (bit de signe à 0) ;
- est normalisé (l'exposant n'est pas codé par 00000000) ;
- a un exposant dont le codage binaire est 01101000 ;
- a une mantisse dont le codage binaire est 000000000000000000000000.

Valeur exacte

On a clairement :

- le bit de signe $s = 0$;
- la valeur de la mantisse qui vaut $m = 0$.

Reste à trouver la valeur de l'exposant e , puis à le décaler de $d = 2^{8-1} - 1 = 127$. On a l'expression binaire de l'exposant $(e)_2 = (01101000)_2 = (e_7 e_6 \dots e_0)_2$.

On en déduit la valeur en base 10,

$$\begin{aligned} e &= \sum_{i=0}^7 e_i 2^i \\ &= 2^3 + 2^5 + 2^6 \\ &= 8 + 32 + 64 \\ &= 104 \end{aligned}$$

D'où $e - d = 104 - 127 = -23$.

Finalement, la valeur v du flottant binaire précédent est :

$$\begin{aligned} v &= (-1)^0 \times 1.0 \times 2^{-23} \\ &= 2^{-23} \end{aligned}$$

Valeur décimale

On en déduit la valeur décimale suivante :

$$\begin{aligned} v &= 2^{-23} \\ &\approx 1.19 \times 10^{-7} \end{aligned}$$

Ce qui confirme le résultat affiché à l'écran :

```
Epsilon machine (simple precision) : 1.19209e-07
```

FIGURE 1.3 – Valeur décimale affichée de notre approximation de l' $\epsilon_{machine}$ simple

En fait, les valeurs trouvées pour le $\epsilon_{machine}$ sont très logiques dans le sens où elle définissent la limite qu'impose la mantisse : après 23 bits en simple précision et 52 bits en double (respectivement 2^{-23} et 2^{-52} pour l'épsilon machine), on ne peut plus rien mettre dans la mantisse.

1.2 X_{min}

X_{min} est le plus petit nombre flottant que l'on peut coder sur machine. On développe un programme C pour pouvoir le calculer en normalisé et en dénormalisé :

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  float xMinNormaliseSimple(){
5      float z = 1.;
6      int n = 0;
7      while(z/2 > 0 && n < 126){
8          z/=2;
9          n++;
10     }
11     return z;
12 }
13
14 double xMinNormaliseDouble(){
15     double z = 1.;
16     int n = 0;
17     while(z/2 > 0 && n < 1022){
18         z/=2;
19         n++;
20     }
21     return z;
22 }
23
24 float xMinDenormaliseSimple(){
25     float z = 1.;
26     while(z/2 > 0){
27         z/=2;
28     }
29     return z;
30 }
31
32 double xMinDenormaliseDouble(){
33     double z = 1.;
34     while(z/2 > 0){
35         z/=2;
36     }
37     return z;
38 }
39
40 int main(int argc, char* argv[]){
41     float xMin1 = xMinNormaliseSimple();
42     double xMin2 = xMinNormaliseDouble();
43     float xMin3 = xMinDenormaliseSimple();
44     double xMin4 = xMinDenormaliseDouble();
45     printf("X min normalise (simple precision) : %g\n", xMin1);
46     printf("X min normalise (double precision) : %g\n", xMin2);
47     printf("X min denormalise (simple precision) : %g\n", xMin3);
48     printf("X min denormalise (double precision) : %g\n", xMin4);
49     return EXIT_SUCCESS;

```

On va donc poser un break sur les lignes 45, 46, 47 et 48.

Ce qui nous donne comme sortie à l'écran :

```
X min normalise (simple precision) : 1.17549e-38
X min normalise (double precision) : 2.22507e-308
X min denormalise (simple precision) : 1.4013e-45
X min denormalise (double precision) : 4.94066e-324
```

FIGURE 1.6 – Résultat affiché par le programme

1.2.1 X_{min} normalisé en simple précision

Codage binaire

En codage 32 bits, on a :

```
(gdb) x/tw &xMin1
0x7fffffffde18: 00000000100000000000000000000000
(gdb) 
```

FIGURE 1.7 – Codage binaire du X_{min} normalisé en simple précision

On en déduit donc que le nombre flottant, en simple précision :

- est bien composé de 32 bits ;
- est positif (bit de signe à 0) ;
- est normalisé (l'exposant n'est pas codé par 00000000) ;
- a un exposant dont le codage binaire est 00000001 ;
- a une mantisse dont le codage binaire est 000000000000000000000000.

Valeur exacte

On a, là encore, clairement :

- le bit de signe $s = 0$;
- la valeur de la mantisse qui vaut $m = 0$.

Trouvons la valeur de l'exposant e , avant de le décaler de $d = 127$. On a l'expression binaire de l'exposant $(e)_2 = (00000001)_2 = (e_7 e_6 \dots e_0)_2$.

On en déduit la valeur en base 10,

$$\begin{aligned} e &= \sum_{i=0}^7 e_i 2^i \\ &= 2^0 \\ &= 1 \end{aligned}$$

D'où $e - d = 1 - 127 = -126$.

Finalement, la valeur v du X_{min} normalisé en simple précision est :

$$\begin{aligned} v &= (-1)^0 \times 1.0 \times 2^{-126} \\ &= 2^{-126} \end{aligned}$$

Valeur décimale

On en déduit la valeur décimale suivante :

$$\begin{aligned} v &= 2^{-1022} \\ &\approx 2.225 \times 10^{-308} \end{aligned}$$

Ce qui confirme le résultat affiché à l'écran :

```
X min normalise (double precision) : 2.22507e-308
```

FIGURE 1.10 – Valeur décimale affichée du X_{min} normalisé double

1.2.3 X_{min} dénormalisé en simple précision

Codage binaire

En codage 32 bits, on a :

```
(gdb) x/tw &xMin3
0x7fffffffde1c: 00000000000000000000000000000001
(gdb)
```

FIGURE 1.11 – Codage binaire du X_{min} dénormalisé en simple précision

On en déduit donc que le nombre flottant, en simple précision :

- est bien composé de 32 bits ;
- est positif (bit de signe à 0) ;
- est bien normalisé (l'exposant est bien codé par 00000000) ;
- a une mantisse dont le codage binaire est 000000000000000000000001.

Valeur exacte

On a maintenant :

- le bit de signe $s = 0$;
- l'exposant qui ne vaut pas -127 comme on pourrait le penser mais bien -126 (cf introduction).

Trouvons la valeur de la mantisse en base 10. On rajoute le bit implicite et alors on a $(m)_2 = (0, 000000000000000000000001)_2$, d'où :

$$m = 2^{-23}$$

Finalement, la valeur v du X_{min} dénormalisé en simple précision est :

$$\begin{aligned} v &= (-1)^0 \times 2^{-23} \times 2^{-126} \\ &= 2^{-149} \end{aligned}$$

Valeur décimale

On en déduit la valeur décimale suivante :

$$\begin{aligned} v &= 2^{-149} \\ &\approx 1.401 \times 10^{-45} \end{aligned}$$

Ce qui confirme le résultat affiché à l'écran :

```
X min denormalise (simple precision) : 1.4013e-45
```

FIGURE 1.12 – Valeur décimale affichée du X_{min} dénormalisé simple

1.2.4 X_{min} dénormalisé en double précision

Codage binaire

En codage 64 bits, on a :

[illegible]FIGURE 1.13 – Codage binaire du X_{min} dénormalisé en double précision

On en déduit donc que le nombre flottant, en double précision :

- [illegible]

Valeur exacte

On a maintenant :

- le bit de signe $s = 0$;
- l'exposant qui ne vaut pas -1023 comme on pourrait le penser mais bien -1022 (cf introduction).

[illegible]

$$m = 2^{-52}$$

Finalement, la valeur v du X_{min} dénormalisé en double précision est :

$$\begin{aligned} v &= (-1)^0 \times 2^{-52} \times 2^{-1022} \\ &= 2^{-1074} \end{aligned}$$

Valeur décimale

On en déduit la valeur décimale suivante :

$$v = 2^{-1074}$$

$$\approx 4.940 \times 10^{-324}$$

Ce qui confirme le résultat affiché à l'écran :

```
X min denormalise (double precision) : 4.94066e-324
```

FIGURE 1.14 – Valeur décimale affichée du X_{min} dénormalisé double

1.3 X_{max}

X_{max} est le plus grand nombre flottant que l'on peut coder sur machine. On développe un programme C pour pouvoir le calculer en normalisé et en dénormalisé :

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  float xMaxNormaliseSimple(){
5      float z = 1., inf = 1./0., x;
6      while(2*z < inf){
7          z*=2;
8      } // Plus grand exposant
9      x = z/2;
10     while(z + x < inf){
11         z+=x;
12         x/=2;
13     } // Plus grande mantisse
14     return z;
15 }
16
17 double xMaxNormaliseDouble(){
18     double z = 1., inf = 1./0., x;
19     while(2*z < inf){
20         z*=2;
21     } // Plus grand exposant
22     x = z/2;
23     while(z + x < inf){
24         z+=x;
25         x/=2;
26     } // Plus grande mantisse
27     return z;
28 }
29
30 float xMaxDenormaliseSimple(){
31     float xMinNorm = 1., xMinDenorm = 1.;
32     int n = 0;
33     while(xMinDenorm/2 > 0){
34         xMinDenorm/=2;
35     }
36     while(xMinNorm/2 > 0 && n < 126){
37         xMinNorm/=2;
38         n++;
39     }
40     return (xMinNorm - xMinDenorm);
41 }
42
43 double xMaxDenormaliseDouble(){
44     double xMinNorm = 1., xMinDenorm = 1.;
45     int n = 0;
46     while(xMinDenorm/2 > 0){
47         xMinDenorm/=2;
48     }
49     while(xMinNorm/2 > 0 && n < 1022){
50         xMinNorm/=2;
51         n++;
52     }
53     return (xMinNorm - xMinDenorm);
54 }
55

```

```

56 int main(int argc, char* argv[]){
57     float xMax1 = xMaxNormaliseSimple();
58     double xMax2 = xMaxNormaliseDouble();
59     float xMax3 = xMaxDenormaliseSimple();
60     double xMax4 = xMaxDenormaliseDouble();
61     printf("X Max normalise (simple precision) : %g\n", xMax1);
62     printf("X Max normalise (double precision) : %g\n", xMax2);
63     printf("X Max denormalise (simple precision) : %g\n", xMax3);
64     printf("X Max denormalise (double precision) : %g\n", xMax4);
65     return EXIT_SUCCESS;
66 }

```

On va donc poser un break sur les lignes 61, 62, 63 et 64.

Ce qui nous donne comme sortie à l'écran :

```

X Max normalise (simple precision) : 3.40282e+38
X Max normalise (double precision) : 1.79769e+308
X Max denormalise (simple precision) : 1.17549e-38
X Max denormalise (double precision) : 2.22507e-308

```

FIGURE 1.15 – Résultat affiché par le programme

1.3.1 X_{max} normalisé en simple précision

Codage binaire

En codage 32 bits, on a :

```

(gdb) x/tw &xMax1
0x7fffffffde18: 01111111011111111111111111111111
(gdb)

```

FIGURE 1.16 – Codage binaire du X_{max} normalisé en simple précision

On en déduit donc que le nombre flottant, en simple précision :

- est bien composé de 32 bits;
- est positif (bit de signe à 0);
- est bien dénormalisé (l'exposant n'est pas codé par 00000000);
- a un exposant dont le codage binaire est 11111111 (et non pas 11111111, sinon ce serait infinity ou NaN);
- a une mantisse dont le codage binaire est 111111111111111111111111.

Valeur exacte

On a :

— le bit de signe $s = 0$;

Trouvons la valeur de l'exposant e . On a l'expression binaire de l'exposant $(e)_2 = (11111110)_2 = (e_7 e_6 \dots e_0)_2$.

On en déduit la valeur en base 10,

$$\begin{aligned} e &= \sum_{i=0}^7 e_i 2^i \\ &= 2^1 + 2^2 + 2^3 + 2^5 + 2^6 + 2^7 \\ &= 2 \times \frac{1 - 2^7}{1 - 2} \\ &= 254 \end{aligned}$$

D'où $e - d = 254 - 127 = 127$.

Trouvons la valeur de la mantisse m . On a le codage binaire de la mantisse qui est : $(m)_2 = (0, 111111111111111111111111)_2 = (0, m_1 m_2 \dots m_{23})_2$.

On en déduit la valeur en base 10,

$$\begin{aligned} m &= \sum_{i=1}^{23} m_i 2^{-i} \\ &= 2^{-23} \frac{1 - 2^{23}}{1 - 2} \\ &\approx 0.99999988079071044921875 \end{aligned}$$

D'où $1 + m = 1.99999988079071044921875$.

Finalement, la valeur v du X_{max} normalisé en simple précision est :

$$\begin{aligned} v &\approx (-1)^0 \times 1.99999988079071044921875 \times 2^{127} \\ &= 1.99999988079071044921875 \times 2^{127} \end{aligned}$$

Valeur décimale

On en déduit la valeur décimale suivante :

$$\begin{aligned} v &\approx 1.99999988079071044921875 \times 2^{127} \\ &\approx 3.40282346638528859811704183484516925440 \times 10^{38} \end{aligned}$$

Ce qui confirme le résultat affiché à l'écran :

X Max normalise (simple precision) : 3.40282e+38

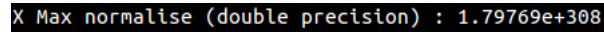
FIGURE 1.17 – Valeur décimale affichée du X_{max} normalisé simple

Valeur décimale

On en déduit la valeur décimale suivante :

$$\begin{aligned} v &\approx 1.99999999999999997779553950749686919152736663818359375 \times 2^{1023} \\ &\approx 1.797693134862315708145274237317043567980705675258449965 \times 10^{308} \end{aligned}$$

Ce qui confirme le résultat affiché à l'écran :

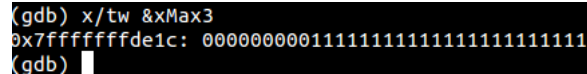


```
X Max normalise (double precision) : 1.79769e+308
```

FIGURE 1.19 – Valeur décimale affichée du X_{max} normalisé double

1.3.3 X_{max} dénormalisé en simple précision**Codage binaire**

En codage 32 bits, on a :



```
(gdb) x/tw &xMax3
0x7fffffffde1c: 00000000111111111111111111111111
(gdb)
```

FIGURE 1.20 – Codage binaire du X_{max} dénormalisé en simple précision

On en déduit donc que le nombre flottant, en simple précision :

- est bien composé de 32 bits ;
- est positif (bit de signe à 0) ;
- est bien dénormalisé (l'exposant est bien codé par 00000000) ;
- a une mantisse dont le codage binaire est 111111111111111111111111.

Valeur exacte

On a maintenant :

- le bit de signe $s = 0$;
- l'exposant qui ne vaut pas -127 comme on pourrait le penser mais bien -126 (cf introduction).

Trouvons la valeur de la mantisse en base 10. On rajoute le bit implicite et alors on a $(m)_2 = (0, 111111111111111111111111)_2 = (0, m_1 m_2 \dots m_{23})_2$, d'où :

$$\begin{aligned} m &= \sum_{i=1}^{23} m_i 2^{-i} \\ &= 2^{-23} \frac{1 - 2^{23}}{1 - 2} \\ &\approx 0.99999988079071044921875 \end{aligned}$$

Finalement, la valeur v du X_{max} dénormalisé en simple précision est :

$$v \approx 0.99999988079071044921875 \times 2^{-126}$$

Valeur décimale

On en déduit la valeur décimale suivante :

$$v \approx 0.99999999999997779553950749686919152736663818359375 \times 2^{-1022}$$
$$\approx 2.225074 \times 10^{-308}$$

Ce qui confirme le résultat affiché à l'écran :

```
X Max denormalise (double precision) : 2.22507e-308
```

FIGURE 1.23 – Valeur décimale affichée du X_{max} dénormalisé double

1.4 Le zéro et l'unité

On s'intéresse ici à des nombres très important car ils sont constamment utilisés en calcul scientifique. Il s'agit bien des éléments neutres des lois de compositions du corps des réels : 0 pour l'addition et 1 pour la multiplication. On développe un programme C pour pouvoir les afficher :

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(int argc, char* argv[]){
5     float zero1 = 0.;
6     double zero2 = 0.;
7     float un1 = 1.;
8     double un2 = 1.;
9     printf("Zero (simple precision) : %g\n", zero1);
10    printf("Zero (double precision) : %g\n", zero2);
11    printf("Un (simple precision) : %g\n", un1);
12    printf("Un (double precision) : %g\n", un2);
13    return EXIT_SUCCESS;
14 }
```

On va donc poser un break sur les lignes 9, 10, 11 et 12.

Ce qui nous donne comme sortie à l'écran :

```
Zero (simple precision) : 0
Zero (double precision) : 0
Un (simple precision) : 1
Un (double precision) : 1
```

FIGURE 1.24 – Résultat affiché par le programme

1.4.1 Le zéro machine en simple précision

Codage binaire

En codage 32 bits, on a :

```
(gdb) x/tw &zero1
0x7fffffffde18: 00000000000000000000000000000000
```

FIGURE 1.25 – Codage binaire du zéro machine en simple précision

On en déduit donc que le nombre flottant, en simple précision :

- est bien composé de 32 bits ;
- est positif (bit de signe à 0), mais il existe une autre valeur du zéro qui est négative (bit de signe à 1) qui est traitée par la machine comme étant la même valeur ;
- est dénormalisé (l'exposant est codé par 00000000) ;
- a une mantisse dont le codage binaire est 000000000000000000000000.

Valeur exacte

On a maintenant :

- le bit de signe $s = 0$ ou $s = 1$ (peu importe) ;
- l'exposant qui vaut -126 car le nombre est dénormalisé ;
- la mantisse qui vaut 0.

Finalement, la valeur v du zéro machine en simple précision est :

$$v = \pm 0.0 \times 2^{-126}$$

Valeur décimale

On en déduit la valeur décimale suivante :

$$\begin{aligned} v &= \pm 0.0 \times 2^{-126} \\ &= 0 \end{aligned}$$

Clairement confirmée par notre programme :

```
Zero (simple precision) : 0
```

FIGURE 1.26 – Valeur décimale affichée du zéro machine en simple précision

1.4.2 Le zéro machine en double précision

Codage binaire

En codage 64 bits, on a :

```
(gdb) x/tg &zero2
0x7fffffffde20: 0000000000000000000000000000000000000000000000000000000000000000
```

FIGURE 1.27 – Codage binaire du zéro machine en double précision

On en déduit donc que le nombre flottant, en simple précision :

- est bien composé de 64 bits ;
- est positif (bit de signe à 0), mais il existe une autre valeur du zéro qui est négative (bit de signe à 1) qui est traitée par la machine comme étant la même valeur ;
- est dénormalisé (l'exposant est codé par 00000000000) ;
- a une mantisse dont le codage binaire est 000.

Valeur exacte

On a maintenant :

- le bit de signe $s = 0$ ou $s = 1$ (peu importe) ;
- l'exposant qui vaut -1022 car le nombre est dénormalisé ;
- la mantisse qui vaut 0 .

Finalement, la valeur v du zéro machine en simple précision est :

$$v = \pm 0.0 \times 2^{-1022}$$

Valeur décimale

On en déduit la valeur décimale suivante :

$$v = \pm 0.0 \times 2^{-1022}$$

$$= 0$$

Clairement confirmée par notre programme :

```
Zero (double precision) : 0
```

FIGURE 1.28 – Valeur décimale affichée du zéro machine en double précision

1.4.3 L'unité machine en simple précision

Codage binaire

En codage 32 bits, on a :

```
(gdb) x/tw &un1
0x7fffffffde1c: 00111111100000000000000000000000
```

FIGURE 1.29 – Codage binaire de l'unité machine en simple précision

On en déduit donc que le nombre flottant, en simple précision :

- est bien composé de 32 bits ;
- est bien positif (bit de signe à 0) ;
- est normalisé ;
- a un exposant dont le codage binaire est 01111111 ;
- a une mantisse dont le codage binaire est 000000000000000000000000.

Valeur exacte

On a maintenant :

- le bit de signe qui est $s = 0$;

— la mantisse qui vaut 0, donc $1 + m = 1$.
 La valeur de l'exposant est l'expression décimale du nombre binaire $(e)_2 = (01111111)_2 = (e_7 e_6 \dots e_0)_2$:

$$\begin{aligned} e &= \sum_{i=0}^7 e_i 2^i \\ &= \frac{1 - 2^7}{1 - 2} \\ &= 127 \end{aligned}$$

D'où $e - d = 127 - 127 = 0$. Finalement, la valeur v de l'unité machine en simple précision est :

$$v = 1 \times 2^0$$

Valeur décimale

On en déduit la valeur décimale suivante :

$$\begin{aligned} v &= 1 \times 2^0 \\ &= 1 \end{aligned}$$

Clairement confirmée par notre programme :

```
Un (simple precision) : 1
```

FIGURE 1.30 – Valeur décimale affichée de l'unité machine en simple précision

1.4.4 L'unité machine en double précision

Codage binaire

En codage 64 bits, on a :

```
(gdb) x/tg &un2
0x7fffffffde28: 0011111111110000000000000000000000000000000000000000000000000000
```

FIGURE 1.31 – Codage binaire de l'unité machine en double précision

On en déduit donc que le nombre flottant, en double précision :

- est bien composé de 64 bits ;
- est bien positif (bit de signe à 0) ;
- est normalisé ;
- a un exposant dont le codage binaire est 0111111111 ;
- a une mantisse dont le codage binaire est 00.

Valeur exacte

On a maintenant :

- le bit de signe qui est $s = 0$;
- la mantisse qui vaut 0, donc $1 + m = 1$.

La valeur de l'exposant est l'expression décimale du nombre binaire $(e)_2 = (0111111111)_2 = (e_{10}e_9 \dots e_0)_2$:

$$\begin{aligned} e &= \sum_{i=0}^{10} e_i 2^i \\ &= \frac{1 - 2^{10}}{1 - 2} \\ &= 1023 \end{aligned}$$

D'où $e - d = 1023 - 1023 = 0$. Finalement, la valeur v de l'unité machine en double précision est :

$$v = 1 \times 2^0$$

Valeur décimale

On en déduit la valeur décimale suivante :

$$\begin{aligned} v &= 1 \times 2^0 \\ &= 1 \end{aligned}$$

Clairement confirmée par notre programme :

Un (double precision) : 1

FIGURE 1.32 – Valeur décimale affichée de l'unité machine en double précision

Le nombre flottant 0.1

Dans cette partie, on étudie le flottant 0.1 et nous allons voir que certaines constantes sont inexactes... On développe un programme C pour pouvoir l'afficher :

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(int argc, char* argv[]){
5     float zeroUn1 = 0.1;
6     double zeroUn2 = 0.1;
7     printf("0.1 (simple precision) : %g\n", zeroUn1);
8     printf("0.1 (double precision) : %g\n", zeroUn2);
9     return EXIT_SUCCESS;
10 }
```

On va donc poser un break sur les lignes 7 et 8.

Ce qui nous donne comme sortie à l'écran :

```
0.1 (simple precision) : 0.1
0.1 (double precision) : 0.1
```

FIGURE 2.1 – Résultat affiché par le programme

2.1 Le 0.1 machine en simple précision

2.1.1 Codage binaire

En codage 32 bits, on a :

```
(gdb) x/tw &zeroUn1
0x7fffffffde34: 00111101110011001100110011001101
```

FIGURE 2.2 – Codage binaire du 0.1 machine en simple précision

On en déduit donc que le nombre flottant, en simple précision :

- est bien composé de 32 bits;
- est bien positif (bit de signe à 0);
- est normalisé (l'exposant n'est pas codé par 00000000);
- a un exposant dont le codage binaire est 01111011.
- a une mantisse dont le codage binaire est 10011001100110011001101.

2.1.2 Valeur exacte

On a maintenant :

- le bit de signe $s = 0$;

La valeur de l'exposant est l'expression décimale du nombre binaire $(e)_2 = (01111011)_2 = (e_7 e_6 \dots e_0)_2$:

$$\begin{aligned} e &= \sum_{i=0}^7 e_i 2^i \\ &= 2^0 + 2^1 + 2^3 + 2^4 + 2^5 + 2^6 \\ &= 123 \end{aligned}$$

D'où $e - d = 123 - 127 = -4$ La valeur de la mantisse est l'expression décimale du nombre binaire $(m)_2 = (0, 10011001100110011001101)_2 = (0, m_1 m_2 \dots m_{23})_2$:

$$\begin{aligned} m &= \sum_{i=1}^{23} m_i 2^{-i} \\ &= 2^{-1} + 2^{-4} + 2^{-5} + 2^{-8} + 2^{-9} + 2^{-12} + 2^{-13} + 2^{-16} + 2^{-17} + 2^{-20} + 2^{-21} + 2^{-23} \\ &\approx 0.60000002384 \end{aligned}$$

D'où $1 + m \approx 1.60000002384$. Finalement, la valeur v du 0.1 machine en simple précision est :

$$v \approx 1.60000002384 \times 2^{-4}$$

2.1.3 Valeur décimale

On en déduit la valeur décimale suivante :

$$\begin{aligned} v &\approx 1.60000002384 \times 2^{-4} \\ &\approx 0.10000000149 \end{aligned}$$

Ce qui n'est, ni la valeur exacte recherchée, ni la valeur affichée par notre programme :

0.1 (simple precision) : 0.1

FIGURE 2.3 – Valeur décimale affichée du 0.1 machine en simple précision

En fait, une approximation de 0.1 a été faite par la machine car l'expression de la mantisse en notation binaire scientifique est infinie : elle suit un schéma régulier qui se répète indéfiniment. Pour avoir une valeur exacte en binaire, il nous faudrait une quantité infinie de bits sur la mantisse pour pouvoir le stocker sans l'approximer $(m)_2 = (0, 1001\ 1001\ 1001\dots)_2 = 0.6$. En effet, $1.6 \times 2^{-4} = 0.1$. La machine affiche bien 0.1 car elle fait elle-même l'approximation à l'affichage que $0.10000000149 \approx 0.1$, mais si on lui demande d'afficher la variable avec 50 décimales (avec `%.50f` dans le `printf`), on a bien :

0.1 (simple precision) : 0.1000000014901161193847656250000000000000000000000000000

FIGURE 2.4 – Affichage de 50 décimales du 0.1 machine en simple précision

2.2 Le 0.1 machine en double précision

2.2.1 Codage binaire

En codage 64 bits, on a :

```
(gdb) x/tg &zeroUn2
0x7fffffffde38: 0011111110111001100110011001100110011001100110011001100110011010
```

FIGURE 2.5 – Codage binaire du 0.1 machine en double précision

On en déduit donc que le nombre flottant, en double précision :

- est bien composé de 64 bits ;
- est bien positif (bit de signe à 0) ;
- est normalisé (l'exposant n'est pas codé par 00000000) ;
- a un exposant dont le codage binaire est 0111111011.
- a une mantisse dont le codage binaire est 100110011001100110011001100110011001100110011010.

2.2.2 Valeur exacte

On a maintenant :

- le bit de signe $s = 0$;

La valeur de l'exposant est l'expression décimale du nombre binaire $(e)_2 = (0111111011)_2 = (e_{10}e_9 \dots e_0)_2$:

$$\begin{aligned} e &= \sum_{i=0}^{10} e_i 2^i \\ &= 2^0 + 2^1 + 2^3 + 2^4 + 2^5 + 2^6 + 2^7 + 2^8 + 2^9 \\ &= 1019 \end{aligned}$$

D'où $e - d = 1019 - 1023 = -4$ La valeur de la mantisse est l'expression décimale du nombre binaire $(m)_2 = (0, 100110011001100110011001100110011001100110011010)_2 = (0, m_1 m_2 \dots m_{52})_2$:

$$\begin{aligned} m &= \sum_{i=1}^{52} m_i 2^{-i} \\ &= 2^{-1} + 2^{-4} + 2^{-5} + 2^{-8} + \dots + 2^{-45} + 2^{-48} + 2^{-49} + 2^{-51} \\ &\approx 0.6000000000000000088817841970012523233890533447265625 \end{aligned}$$

D'où $1 + m \approx 1.6$. Finalement, la valeur v du 0.1 machine en double précision est :

$$v \approx 1.6000000000000000088817841970012523233890533447265625 \times 2^{-4}$$

2.2.3 Valeur décimale

On en déduit la valeur décimale suivante :

$$\begin{aligned} v &\approx 1.600000000000000088817841970012523233890533447265625 \times 2^{-4} \\ &\approx 0.1000000000000000055511151231257827021181583404541015625 \end{aligned}$$

On a bien la même approximation à l’affichage (à cause du %g dans le printf) qui est réalisée :

```
0.1 (double precision) : 0.1
```

FIGURE 2.6 – Valeur décimale affichée du 0.1 machine en double précision

L’approximation est bien plus précise ici, si l’on affiche la variable avec 50 décimales (avec %.50f dans le printf), on a bien :

```
0.1 (double precision) : 0.10000000000000000555111512312578270211815834045410
```

FIGURE 2.7 – Affichage de 50 décimales du 0.1 machine en double précision

Étude d'une somme de termes inférieurs à $\epsilon_{machine}$

Dans cette partie, on se propose d'étudier une somme de termes inférieurs à l'épsilon machine. On se donne un nombre flottant $Y \in]0; \epsilon_{machine}[$ et on va s'intéresser à $Z = \sum_{i=1}^n Y$ pour certaines valeurs de $n \in \mathbb{N}^*$. On prend $Y = 2^{-55} \approx 2.7755576 \times 10^{-17}$ car ce nombre est inférieur à l'épsilon machine strictement en simple et double précision et il est supérieur au zéro machine et normalisé en simple et double précision. On développe un programme C pour pouvoir calculer cette somme :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  double calculY(){
5      double y = 1.;
6      int i;
7      for(i = 0; i < 55 ; i++){
8          y/=2;
9      }
10     return y;
11 }
12
13 float sommeEpsilonSimple(int n){
14     float y = calculY(), z = y;
15     int i;
16     for (i = 1 ; i < n ; i++){
17         z+=y;
18     }
19     return z;
20 }
21
22 double sommeEpsilonDouble(int n){
23     double y = calculY(), z = y;
24     int i;
25     for (i = 1 ; i < n ; i++){
26         z+=y;
27     }
28     return z;
29 }
30
31 int main(int argc, char* argv[]){
32     float sum2S = sommeEpsilonSimple(2);
33     float sum10S = sommeEpsilonSimple(10);
34     float sum100S = sommeEpsilonSimple(100);
35     float sum1000S = sommeEpsilonSimple(1000);
36     double sum2D = sommeEpsilonDouble(2);
37     double sum10D = sommeEpsilonDouble(10);
38     double sum100D = sommeEpsilonDouble(100);
39     double sum1000D = sommeEpsilonDouble(1000);
40     printf("Somme pour n = 2 (simple precision) : %g\n", sum2S);
41     printf("Somme pour n = 2 (double precision) : %g\n", sum2D);
42     printf("Somme pour n = 10 (simple precision) : %g\n", sum10S);
43     printf("Somme pour n = 10 (double precision) : %g\n", sum10D);
44     printf("Somme pour n = 100 (simple precision) : %g\n", sum100S);
```

Ce qui nous donne comme sortie à l'écran :

```

Somme pour n = 2 (simple precision) : 5.55112e-17
Somme pour n = 2 (double precision) : 5.55112e-17
Somme pour n = 10 (simple precision) : 2.77556e-16
Somme pour n = 10 (double precision) : 2.77556e-16
Somme pour n = 100 (simple precision) : 2.77556e-15
Somme pour n = 100 (double precision) : 2.77556e-15
Somme pour n = 1000 (simple precision) : 2.77556e-14
Somme pour n = 1000 (double precision) : 2.77556e-14

```

FIGURE 3.1 – Résultat affiché par le programme

3.1.1 Codage binaire

```
(gdb) x/tw &sum25
0x7fffffffdf20: 00100100100000000000000000000000
(gdb)
```

FIGURE 3.2 – Somme de 2 termes en simple précision

[illegible]

FIGURE 3.3 – Somme de 2 termes en double précision

Clairément, les nombres sont normalisés.

On devrait avoir :

$$2 \times 2^{-55} = 2^{-54}$$

Pour les deux précisions, on remarque que le bit de signe est 0 ($s = 0$), et que $m = 0$ (tous les bits de la mantisse sont à zéros). On a pour l'exposant en simple précision $(e)_2 = (01001001)_2$. D'où :

$$e = 1 + 8 + 64 = 73$$

Ainsi $e - d = 73 - 127 = -54$. Finalement on a bien en simple précision

$$\sum_{i=1}^2 Y = 1.0 \times 2^{-54} = 2^{-54}$$

Pour les deux précisions, on remarque que le bit de signe est 0 ($s = 0$). On a pour l'exposant en simple précision $(e)_2 = (01001011)_2$. D'où :

$$e = 1 + 2 + 8 + 64 = 75$$

Ainsi $e - d = 75 - 127 = -52$. Et, $(m)_2 = (0,0100000\dots)_2$ D'où :

$$m = 2^{-2} = 0.25$$

Finalement on a bien en simple précision

$$\sum_{i=1}^{10} Y = 1.25 \times 2^{-52}$$

De même, en double précision, $(e)_2 = (01111001011)_2$ D'où :

$$e = 1 + 2 + 8 + 64 + 128 + 256 + 512 = 971$$

Ce qui implique que $e - d = 971 - 1023 = -52$. Et, $(m)_2 = (0,0100000\dots)_2$ D'où :

$$m = 2^{-2} = 0.25$$

On a donc aussi en double précision :

$$\sum_{i=1}^{10} Y = 1.25 \times 2^{-52}$$

3.2.3 Valeur décimale

On a en simple et double précision :

$$\sum_{i=1}^{10} Y = 1.25 \times 2^{-52} \approx 2.7755576 \times 10^{-16}$$

Ce qui est confirmé à l'affichage :

```
Somme pour n = 10 (simple precision) : 2.77556e-16
Somme pour n = 10 (double precision) : 2.77556e-16
```

FIGURE 3.7 – Somme de 10 termes affichée en simple et double précision

3.3.3 Valeur décimale

On a en simple et double précision :

$$\sum_{i=1}^{100} Y = 1.5625 \times 2^{-49} \\ \approx 2.77555756156 \times 10^{-15}$$

Ce qui est confirmé à l'affichage :

```
Somme pour n = 100 (simple precision) : 2.77556e-15
Somme pour n = 100 (double precision) : 2.77556e-15
```

FIGURE 3.10 – Somme de 100 termes affichée en simple et double précision

3.4 Pour une somme de 1000 termes

3.4.1 Codage binaire

Pour ce qui est du codage binaire, on a en simple et double précision :

```
(gdb) x/tw &sum1000S
0x7fffffffdf2c: 00101000111110100000000000000000
(gdb) █
```

FIGURE 3.11 – Somme de 1000 termes en simple précision

```
(gdb) x/tg &sum1000D
0x7fffffffdf48: 0011111010001111101000000000000000000000000000000000000000000000
(gdb) █
```

FIGURE 3.12 – Somme de 1000 termes en double précision

Clairement, les nombres sont normalisés.

3.4.2 Valeur exacte

On devrait avoir :

$$1000 \times 2^{-55} = 10 \times 1.5625 \times 2^{-49} = 1.25 \times 1.5625 \times 2^{-46} = 1.953125 \times 2^{-46}$$

Pour les deux précisions, on remarque que le bit de signe est 0 ($s = 0$). On a pour l'exposant en simple précision $(e)_2 = (01010001)_2$. D'où :

$$e = 1 + 16 + 64 = 81$$

Ainsi $e - d = 81 - 127 = -46$. Et, $(m)_2 = (0, 11110100 \dots)_2$ D'où :

$$m = 2^{-1} + 2^{-2} + 2^{-3} + 2^{-4} + 2^{-6} = 0.953125$$

Finalement on a bien en simple précision

$$\sum_{i=1}^{1000} Y = 1.953125 \times 2^{-46}$$

De même, en double précision, $(e)_2 = (01111010001)_2$ D'où :

$$e = 1 + 16 + 64 + 128 + 256 + 512 = 977$$

Ce qui implique que $e - d = 977 - 1023 = -46$. Et, $(m)_2 = (0, 11110100 \dots)_2$ D'où :

$$m = 0.953125$$

On a donc aussi en double précision :

$$\sum_{i=1}^{1000} Y = 1.953125 \times 2^{-46}$$

3.4.3 Valeur décimale

On a en simple et double précision :

$$\sum_{i=1}^{1000} Y = 1.953125 \times 2^{-46} \approx 2.77555756156 \times 10^{-14}$$

Ce qui est confirmé à l'affichage :

```
Somme pour n = 1000 (simple precision) : 2.77556e-14
Somme pour n = 1000 (double precision) : 2.77556e-14
```

FIGURE 3.13 – Somme de 1000 termes affichée en simple et double précision

3.5 Remarque : pour n assez grand

Pour un n assez grand (de l'ordre de la centaine de milliers de milliards de milliards), on aurait eu une somme qui atteint 1. Et alors, on pourrait penser que Z se stabilise à 1 par définition du $\epsilon_{machine}$. En effet, si l'on choisit plutôt $Y = 2^{-24}$ et que l'on effectue 10^8 fois la somme :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  double calculY() {
5      double y = 1.;
6      int i;
7      for(i = 0; i < 24 ; i++){
8          y/=2;
9      }
10     return y;
11 }
```

```

12
13 float sommeEpsilonSimple(int n){
14     float y = calculY(), z = y;
15     int i;
16     for (i = 1 ; i < n ; i++){
17         z+=y;
18     }
19     return z;
20 }
21
22 double sommeEpsilonDouble(int n){
23     double y = calculY(), z = y;
24     int i;
25     for (i = 1 ; i < n ; i++){
26         z+=y;
27     }
28     return z;
29 }
30
31 int main(int argc, char* argv[]){
32     float sumS = sommeEpsilonSimple(100000000);
33     double sumD = sommeEpsilonDouble(100000000);
34     printf("Somme pour n = 100000000 (simple precision) : %.50f\n", sumS);
35     printf("Somme pour n = 100000000 (double precision) : %.50f\n", sumD);
36     return EXIT_SUCCESS;
37 }

```

On a comme sortie à l'écran :

[illegible]

FIGURE 3.14 – Résultat affiché par le programme

Et comme Y est strictement inférieur au $\epsilon_{machine}$ simple, il est normal de trouver 1 exactement pour la somme en simple précision et quelque chose de strictement plus grand que 1 pour la somme en double. En effet, en double précision on a bien :

$$10^8 \times 2^{-24} = (1.25)^8 \times 2^0 \approx 5.96046447754$$

D'où les codes binaires (on reconnaît bien l'unité machine et on peut vérifier le résultat en double par un calcul similaire aux précédents) :

```
(gdb) x/tw &sumS
0x7fffffffdf44: 00111111100000000000000000000000
(gdb) break 35
Breakpoint 2 at 0x40067f: file src/somme.c, line 35.
(gdb) x/tg &sumD
0x7fffffffdf48: 0100000000010111110101111000010000000000000000000000000000000000
(gdb)
```

FIGURE 3.15 – Codes binaires des deux variables précédentes

Au vu de ces résultats, il nous semble logique de penser que la prochaine partie nous donnera comme résultat 1 pour toute valeurs de n .

Ajout de l'unité à une somme de termes inférieurs à $l'\epsilon_{machine}$

On s'intéresse maintenant au résultat de l'ajout de 1 à la somme précédente. On va donc regarder la valeur de $1 + Z$. On produit un programme C pour pouvoir calculer cette somme :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  double calculY(){
5      double y = 1.;
6      int i;
7      for(i = 0; i < 55 ; i++){
8          y/=2;
9      }
10     return y;
11 }
12
13 float sommeEpsilonSimple(int n){
14     float y = calculY(), z = y;
15     int i;
16     for (i = 1 ; i < n ; i++){
17         z+=y;
18     }
19     return z;
20 }
21
22 double sommeEpsilonDouble(int n){
23     double y = calculY(), z = y;
24     int i;
25     for (i = 1 ; i < n ; i++){
26         z+=y;
27     }
28     return z;
29 }
30
31 int main(int argc, char* argv[]){
32     float sum2S = sommeEpsilonSimple(2) + 1;
33     float sum10S = sommeEpsilonSimple(10) + 1;
34     float sum100S = sommeEpsilonSimple(100) + 1;
35     float sum1000S = sommeEpsilonSimple(1000) + 1;
36     double sum2D = sommeEpsilonDouble(2) + 1;
37     double sum10D = sommeEpsilonDouble(10) + 1;
38     double sum100D = sommeEpsilonDouble(100) + 1;
39     double sum1000D = sommeEpsilonDouble(1000) + 1;
40     printf("Somme pour n = 2 (simple precision) : %.50f\n", sum2S);
41     printf("Somme pour n = 2 (double precision) : %.50f\n", sum2D);
42     printf("Somme pour n = 10 (simple precision) : %.50f\n", sum10S);
43     printf("Somme pour n = 10 (double precision) : %.50f\n", sum10D);
44     printf("Somme pour n = 100 (simple precision) : %.50f\n", sum100S);
45     printf("Somme pour n = 100 (double precision) : %.50f\n", sum100D);
46     printf("Somme pour n = 1000 (simple precision) : %.50f\n", sum1000S);
47     printf("Somme pour n = 1000 (double precision) : %.50f\n", sum1000D);
48     return EXIT_SUCCESS;
```

On va donc poser un break sur les lignes 40 à 47.

Ce qui nous donne comme sortie à l'écran :

```
Somme pour n = 2 (single precision) : 1.0000000000000000000000000000000000000000000000000000000000000000
Somme pour n = 2 (double precision)   : 1.0000000000000000000000000000000000000000000000000000000000000000
Somme pour n = 10 (simple precision)    : 1.0000000000000000000000000000000000000000000000000000000000000000
Somme pour n = 10 (double precision)    : 1.00000000000000002226404692503130808472633361816406
Somme pour n = 100 (simple precision)    : 1.0000000000000000000000000000000000000000000000000000000000000000
Somme pour n = 100 (double precision)   : 1.000000000000000266453525910037569701671600341796875
Somme pour n = 1000 (simple precision)   : 1.0000000000000000000000000000000000000000000000000000000000000000
Somme pour n = 1000 (double precision)  : 1.0000000000000002775557561562891351059079170227050781
```

FIGURE 4.1 – Résultat affiché par le programme

Hérésie ! Par définition de l'epsilon machine on devrait avoir $(\dots(((1+y)+y)+y)+y\dots) = 1$. Or, on remarque que les valeurs de Z pour lesquelles la somme dépasse l'epsilon machine en double précision nous donne $1 + Z > 1$. En fait, cela nous montre la non-associativité de l'additivité machine, car sinon on aurait $(\dots(((1+y)+y)+y)+y\dots) = 1 + Z = 1$ peu importe si la valeur de Z dépasse l'epsilon machine !

4.1 Pour une somme de 2 termes

Le code binaire confirme que le résultat trouvé pour les deux sommes (en simple et double précision) est le 1 machine.

```
(gdb) x/tw &sum2S
0x7fffffffdf10: 00111111100000000000000000000000
```

FIGURE 4.2 – $1 + Z$ pour $n = 2$ en simple précision

```
(gdb) x/tg &sum2D  
0x7fffffffdf20: 0011111111110000000000000000000000000000000000000000000000000000
```

FIGURE 4.3 – $1 + Z$ pour $n = 2$ en double précision

4.2 Pour les autres sommes

Clairement, on retrouve l'unité machine en simple précision, on s'intéressera donc à la double précision.

```
(gdb) x/tg &sum10D  
0x7fffffffdf28: 0011111111110000000000000000000000000000000000000000000000000000
```

FIGURE 4.4 – $1 + Z$ pour $n = 10$ en double précision

Conclusion

Dans ce TP, nous avons pu découvrir la spécificité de certains flottants. Ceci nous a permis, dans un premier temps, de mieux comprendre le fonctionnement de la norme *IEEE 754*. Puis, nous avons pu montrer les défauts de la représentation des nombres flottants sur machine, en montrant notamment qu'il est impossible de représenter exactement en machine certains réels tels que 0.1. Enfin, nous avons démontré la non-associativité de l'addition machine en étudiant une somme de termes inférieurs à l'épsilon machine.

Tout ce travail nous aura permis de comprendre que l'utilisation de la machine comme outil de calcul scientifique doit se faire en prenant compte des approximations réalisées par celle-ci. La plupart des accidents industriels informatiques était du à des erreurs d'approximations machine, c'est pourquoi il est important de comprendre son fonctionnement avant de l'utiliser à tout-va.

Bibliographie

- [1] *Bernard Gleyse, Cours d'Algorithmique Numérique*, Institut National des Sciences Appliquées de Rouen.
- [2] <http://web2.0calc.fr/>, La calculatrice en ligne. (Valide à la date du 08/03/2014).
- [3] <http://progdupeu.pl/tutoriels/85/les-nombres-a-virgule-flottante/>, Très bon tutoriel pour la norme IEEE 754. (Valide à la date du 10/03/2014).
- [4] http://fr.wikipedia.org/wiki/IEEE_754, Pour comparer mes résultats avec la valeur des flottants caractéristiques affichés sur Wikipédia. (Valide à la date du 10/03/2014).
- [5] http://fr.wikipedia.org/wiki/Nombre_d%C3%A9normalis%C3%A9, Pour mieux comprendre la notion de nombre dénormalisé. (Valide à la date du 12/03/2014).
- [6] <http://fr.openclassrooms.com/informatique/cours/nombres-flottants-et-processeurs/rappels-sur-la-norme-ieee-754>, Un tutoriel complet et bien expliqué pour comprendre la représentation des nombres flottants. (Valide à la date du 13/03/2014).