Nombres en informatique

Séance pratique (2 × 1h30) à faire par binôme.

Dans cette séance, vous allez observer le codage des nombres en informatique, puis différents problèmes que peuvent engendrer ce codage sur un ordinateur.

Créez un répertoire nommé `TP_NOMBRES`, démarrez Scilab et placez-vous dans ce répertoire. Récupérez par le WEB le fichier-script `nombres.sce` :

allez à l'adresse `http://chamilo1.grenet.fr/ujf/courses/MAP101/`
entrez vos login et mot de passe
cliquez sur les liens *Documents* puis *TP*
cliquez sur le fichier et enregistrez-le dans votre répertoire de travail

Ce fichier contient des *procédures* de codages binaires à partir de valeurs, et inversement, ainsi qu'une procédure d'écriture exacte d'une valeur. Placez ce fichier dans le répertoire `TP_NOMBRES`.

Une fois ce fichier récupéré, tapez dans la console Scilab la commande

```
exec("nombres.sce", -1);
```

Cette commande devra être ré-exécutée à chaque redémarrage de Scilab pour ce thème.

Dans un premier temps, on va observer le codage de valeurs sur un ordinateur, puis voir le résultat exact de certaines opérations arithmétiques simples, et certains problèmes rencontrés lors du calcul des éléments d'une suite numérique.

1 - Valeurs numériques avec Scilab

Le logiciel Scilab code toutes les valeurs numériques en utilisant le codage *réel flottant sur 64 bits*.

Dans la console, tapez les instructions suivantes :

```
a=1 , b=0.3 , c=10^100 , d = exp(-20) , e = sqrt(2)
```

Les valeurs sont affichées dans la console avec quelques chiffres après la virgule et éventuellement avec une puissance de 10 (`1.00D+100` correspond à $1,00 \times 10^{100}$).

Le nombre de chiffres après la virgule étant limité, la valeur affichée peut ne pas être une valeur exacte mais une valeur arrondie.

Pour avoir plus de précision à l'affichage d'une valeur, il est préférable d'utiliser l'instruction `mprintf` avec un *format flottant* :

`mprintf("%`*n*`.`*p*`f",`*expression*`)`

où n est le nombre total de caractères pour l'affichage, p le nombre de chiffres affiché après la virgule et *expression* une expression dont on veut afficher la valeur.
Si dans le format, on ajoute les deux caractères `\n`, un retour à la ligne est fait.

Tapez les instructions suivantes et observez le résultat :

```
mprintf("%10.7f\n",e)
mprintf("%15.12f\n",e)
mprintf("%20.17f\n",e)
```

on obtient l'affichage de la valeur $\sqrt{2}$ avec différentes précisions.

Tapez les instructions suivantes et observez le résultat :

```
mprintf("%10.7f\n",b)
mprintf("%15.12f\n",b)
mprintf("%20.17f\n",b)
mprintf("%60.57f\n",b)
```

on remarque qu'avec une précision suffisante à l'affichage, on obtient la valeur exacte codée par Scilab pour $b = 0,3$.

L'instruction `ecrire` définie dans le script `nombres.sce` permet d'écrire la valeur exacte sous forme scientifique en utilisant la précision nécessaire pour la mantisse.
Tapez les instructions suivantes et observez le résultat :

```
ecrire(b)
ecrire(c)
```

ceci montre que les valeurs $b = 0,3$ et $c = 10^{100}$ ne sont pas codées de manière exacte avec le logiciel Scilab.

2 - Codage de réels flottants, ordres de grandeur et précision

Le codage des valeurs en utilisant le codage *réel flottant* sur 64 bits implique des limitations sur les valeurs qu'on peut manipuler.

Le script `nombres.sce` fournit des procédures pour obtenir le codage binaire à partir d'une valeur réelle, et inversement.

La procédure `ecrire_codage64` permet d'écrire à l'écran le codage binaire sur 64 bits d'une valeur `v`.

Testez cette procédure en tapant les instructions suivantes :

```
ecrire_codage64(-101)
ecrire_codage64(0.3)
ecrire_codage64(0)
```

Observez ce qui ce passe au niveau du codage quand on passe d'une valeur v à $v \times 2$ et $v/2$, par exemple

```
ecrire_codage64(1)
ecrire_codage64(2)
ecrire_codage64(0.5)
```

ou bien

```
ecrire_codage64(0.3)
ecrire_codage64(0.6)
ecrire_codage64(0.15)
```

On peut aussi utiliser la procédure `codage_reel64` pour récupérer le codage d'une valeur `v` sous forme de trois chaînes de caractères `s` (*signe* - 1 caractère), `e` (*exposant* - 11 caractères) et `m` (*mantisse* - 52 caractères).

On peut aussi obtenir une valeur à partir d'un codage donné sous la forme de 3 chaînes de caractères.

Tapez les instructions suivant dans un fichier-script nommé `ex1.sce`

```
s = "0"
e = "01111111111"
m = "0000000000000000000000000000000000000000000000000000"
v = valeur_reel64(s,e,m);
ecrire(v)
```

(pour s'assurer que `m` a bien 52 caractères, placez le curseur en début et fin de chaîne, et regardez les numéros de colonne correspondant dans la barre d'état).

Les ordres de grandeur qu'on peut atteindre (en min. et en max.) pour une valeur réelle correspondant aux limites que les parties *exposant* (`e`) et *mantisse* (`m`) peuvent atteindre.

Tapez les instructions suivantes dans un fichier-script nommé `ex1.sce`

```
s = "0"
e = "00000000000"
m = "0000000000000000000000000000000000000000000000000001"
v = valeur_reel64(s,e,m);
ecrire(v)
s = "0"
e = "00000000001"
m = "0000000000000000000000000000000000000000000000000000"
v = valeur_reel64(s,e,m);
ecrire(v)
s = "0"
e = "11111111110"
m = "1111111111111111111111111111111111111111111111111111"
v = valeur_reel64(s,e,m);
ecrire(v)
```

et exécutez ce fichier.

La précision est donnée par le nombre de bits de la mantisse. Tapez les instructions suivantes dans un fichier-script nommé `ex1.sce`

```
s = "0"
e = "01111111111"
m = "0000000000000000000000000000000000000000000000000000"
v = valeur_reel64(s,e,m);
ecrire(v)
s = "0"
e = "01111111111"
m = "0000000000000000000000000000000000000000000000000001"
v = valeur_reel64(s,e,m);
ecrire(v)
```

et exécutez ce fichier.
Avec ce codage, la valeur immédiatement supérieure à 1, et codée différemment de 1 est $1+2^{-52}$.

La précision relative est de $\varepsilon = 1/2^{52} \simeq 2.10^{-16}$ soit 15 à 16 chiffres décimaux après la virgule, c'est à dire qu'en deça de cette précision relative, deux nombres voisins non nuls x et y tels que $\left|\dfrac{x-y}{x}\right| < \varepsilon$ seront codés sur ordinateur de la manière identique et donc considérés comme égaux.

On dit parfois que ce codage des réels flottants sur 64 bits à une précision de 15 à 16 chiffres décimaux.

Tapez l'instruction `ecrire_codage64(4)` puis `ecrire_codage64(4.1)` puis `ecrire_codage64(4.01)` puis `ecrire_codage64(4.001)` etc ... et arrêtez-vous dès que vous trouvez la valeur $4.000\ldots001 = 4 + 10^{-m}$ dont le codage est identique à la valeur 4. Que vaut m ?

3 - Erreurs numériques lors de calcul sur ordinateur

Le fait que certaines valeurs numériques (mêmes simples) ne peuvent pas être codées de manière exacte avec Scilab, peut entrainer des problèmes de calcul numérique.

Exemple : On a $0,9 - 0,6 - 0,3 = 0$ mais avec Scilab ce n'est pas le cas.
Tapez l'instruction suivante et observez le résultat :

```
r1=0.9-0.6-0.3
```

En effet les valeurs $0,3$, $0,6$ et $0,9$ ne sont pas représentées de manière exacte en Scilab (tapez les instructions suivantes)

```
ecrire(0.3)
ecrire(0.6)
ecrire(0.9)
ecrire(0.9-0.6)
```

on voit notamment que les valeurs $0,9-0,6$ et $0,3$ sont différentes avec Scilab, et qu'un simple calcul peut engendrer une (petite) erreur numérique. Si on effectue alors un grand nombre de calcul à la suite, ces différentes (petites erreurs) peuvent alors se cumuler.

De plus l'ordre dans lequel on fait les opérations peut influer sur le résultat. Tapez l'instruction suivante et observez le résultat :

```
r2=0.9-0.3-0.6
```

et comparez `r2` avec `r1` .

Exemple : On va effectuer le processus suivant : on part avec la valeur $s = 0$, puis on va lui ajouter la valeur $0,3$ cent mille fois, puis lui retrancher la valeur 3 dix mille fois. Normalement on doit retrouver $s = 0$ car

$$0 + 0,3 \times 100000 - 3 \times 10000 = 0$$

Avec l'éditeur de texte Scilab, créez le fichier suivant en le nommant `ex_numerique1.sce`, exécutez-le et observez le résultat :

```
s = 0;
```

```
for k=1:10000
  s = s+0.3;
end
for k=1:1000
  s = s-3;
end
disp(s)
```

la valeur de s n'est pas nulle mais négative (proche de 0).
Cet exemple simple montre le problème d'accumulation des *erreurs numériques*.

Exemple : Dans cet exemple, on va partir de la valeur $b = 1$ puis on va la diviser par 2 un certain nombre de fois. Comme la valeur de départ est non nulle et positive, en la divisant par deux, elle reste non nulle et positive

$$\begin{array}{ccccccccc} b=1 & \rightarrow & \boxed{\text{division par 2}} & \rightarrow & b=1/2 & \rightarrow & \boxed{\text{division par 2}} & \rightarrow & b=1/4 \quad \rightarrow \boxed{\text{division par 2}} \rightarrow \quad \ldots \\ \ldots & \rightarrow & \boxed{\text{division par 2}} & \rightarrow & b=1/2^n & \rightarrow & \boxed{\text{division par 2}} & \rightarrow & b=1/2^{n+1} \quad \rightarrow \boxed{\text{division par 2}} \rightarrow \quad \ldots \end{array}$$

Avec l'éditeur de texte Scilab, créez le fichier suivant en le nommant `ex_numerique2.sce`

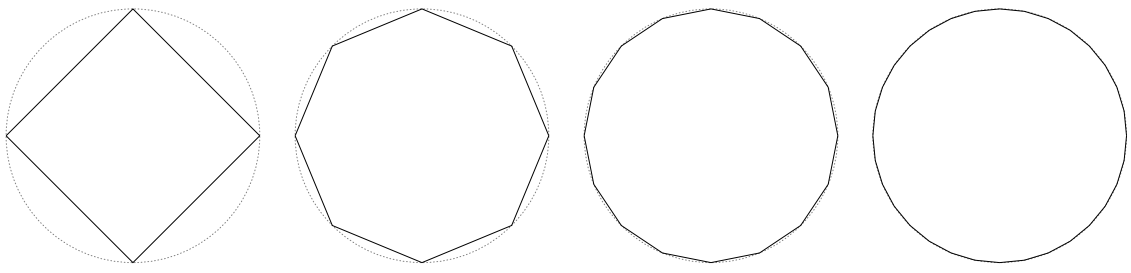
```
b = 1;
n = 0; // nombre d'iterations
while b>0
  b = b/2;
  n = n+1;
end
mprintf("La boucle s''est arretee apres %d iterations", n)
```

Normalement si on exécute ces instructions, on ne devrait jamais sortir de la boucle car la valeur de b devrait rester strictement supérieure à 0.
Exécutez ce script, et commentez le résultat obtenu.

Modifiez le script précédent en remplaçant l'instruction `while b>0` par `while b+1>1`
Le test $b > 0$ étant équivalent au test $b + 1 > 1$, le script devrait avoir le même comportement.
Exécutez ce script, et commentez le résultat.

Exemple : Dans cet exemple, on va calculer une approximation du nombre π à l'aide d'une suite de nombres sensée converger vers π.

La valeur π est le périmètre d'un cercle de diamètre 1, et pour avoir une approximation de π, on peut considérer les périmètres de polygones réguliers inscrits dans un cercle de diamètre 1.



Polygone P_1 (4 cotés) — Polygone P_2 (8 cotés) — Polygone P_3 (16 cotés) — Polygone P_4 (32 cotés)

On considère la suite de polygones réguliers à partir de P_1 (carré inscrit dans le cercle), et on passe de P_n à P_{n+1} en doublant le nombre de cotés. On note v_n le périmètre de P_n.

La suite de polygones P_n *tend* vers le cercle unité et donc la suite des valeurs de périmètre v_n tend vers la valeur π.

On peut montrer que :

$$v_1 = 2\sqrt{2} \qquad \text{et} \qquad v_{n+1} = 2^{n+1}\sqrt{2 - \sqrt{4 - \left(\frac{v_n}{2^n}\right)^2}}$$

Avec l'éditeur de texte Scilab, créez le fichier suivant en le nommant `ex_suite_pi.sce`, exécutez-le et observez le résultat :

```
vn = 2*sqrt(2);
for n=1:30
    mprintf("n = %2d , v(%2d) = %20.15f\n", n, n, vn);
    vn = 2^(n+1)*sqrt(2-sqrt(4-(vn/2^n)^2));
end
```

Cette boucle calcule et écrit les 30 premiers termes de la suite v_n.
Les valeurs v_n s'approchent de π puis continuent à croître jusqu'à 4 et deviennent ensuite nulles. L'explication est la suivante : le terme $\left(\frac{v_n}{2^n}\right)^2$ va tendre vers 0 donc numériquement va devenir négligeable par rapport à 4, et à partir d'une certaine valeur de n

$$2^{n+1}\sqrt{2 - \sqrt{4 - \left(\frac{v_n}{2^n}\right)^2}} \simeq 2^{n+1}\sqrt{2 - \sqrt{4 - 0}} = 0$$

Pour éviter ce problème, il faut modifier la formule de récurrence entre v_n et v_{n+1} :

$$v_{n+1} = 2^{n+1}\sqrt{2 - \sqrt{4 - \left(\frac{v_n}{2^n}\right)^2}} = \frac{2^{n+1}\sqrt{2 - \sqrt{4 - \left(\frac{v_n}{2^n}\right)^2}}\sqrt{2 + \sqrt{4 - \left(\frac{v_n}{2^n}\right)^2}}}{\sqrt{2 + \sqrt{4 - \left(\frac{v_n}{2^n}\right)^2}}} \qquad (1)$$

Exercice 1 :

Cet exercice fera l'objet d'un compte-rendu (par binôme) noté.

Simplifiez le numérateur de la formule (1) précédente, afin de trouver une nouvelle expression pour v_{n+1} en fonction de v_n puis modifiez le script `ex_suite_pi.sce` en conséquence, et exécutez-le.

Pour le compte-rendu, rendre à la fin de la séance une feuille A4 manuscrite, en indiquant, les noms et prénoms de votre binôme avec

- la formule (1) simplifiée,
- la modification faite dans le script `ex_suite_pi.sce`,
- la valeur de v_{30} obtenue.