TELECOM Nancy *(ESIAL)*
Maths Numériques

TP1 : arithmétique flottante

Exercice 1 $(0.2)_{10}$ *suite et fin*

Dans le td1, on a vu qu'en double précision (cad les flottants usuels stockés sur 8 octets, alias $\mathbb{F}(2, 53, -1022, 1023)$), on a :

$$fl(0.2) = (\underbrace{1,100}_{1}\ \underbrace{1100}_{2}\dots\underbrace{1100}_{12}\ \underbrace{1101}_{13}\ 0)_2\ 2^{-3}$$

En base 10 ce nombre s'écrit exactement 0.200000000000000011102230246251565404236316680908203125. L'exercice (2 mn max) consiste à initialiser une variable à cette valeur (genre `x = 0.2`). La mémoire interne doit contenir en fait le nombre $fl(0.2)$ ce que l'on découvre si on sort une impression (en base 10) avec suffisamment de chiffres.

1. Sortir la valeur contenue dans la variable `x` à l'aide de `print(x)` ou encore en utilisant une f-string `print(f"x = {x}")`.
2. Expérimenter en autorisant plus de chiffres, ce qui peut se faire avec `print(f"x = {x:fmt}")` où fmt est une chaîne de caractères du type :

   longueur_totale_champ.*longueur_partie_décimale*f

   Par exemple avec `14.10f` on obtiendra une sortie de type :

   $$\underbrace{xxx.\overbrace{xxxxxxxxxx}^{10\ c}}_{14\ c}$$

   On découvre que le nombre stocké n'est pas exactement 0.2 avec fmt = `19.17f` et pour obtenir tous les chiffres il faut fmt = `56.54f`.

Exercice 2 *Seuil d'overflow, mise en pratique !*

Dans le cours nous avons vu que le plus grand nombre flottant de $\mathbb{F}(\beta, p, e_{min}, e_{max})$ est :

$$M = (1 - \beta^{-p})\beta^{e_{max}+1} = \beta^{e_{max}+1} - \beta^{e_{max}+1-p}$$

et que le seuil d'overflow est :

$$\widetilde{M} = \frac{M + \beta^{e_{max}+1}}{2}$$

tout nombre réel x tel que $|x| < \widetilde{M}$ est codé par un nombre flottant "usuel" (ni $\pm Inf$, ni un Nan). Par contre $fl(\widetilde{M}) = Inf$. Le but de cet exercice est de profiter des entiers longs de python pour calculer exactement ces nombres et de vérifier si la pratique correspond à la théorie ! Il est clair que M est un entier lorsque $e_{max} + 1 \geq p$, de même si β est pair et si $e_{max} + 1 > p$ il est aussi évident que $\widetilde{M}$ est un entier. Ces contraintes sont respectées par tous les systèmes flottants usuels et en particulier pour $\mathbb{F}(2, 53, -1022, 1023)$ celui des "doubles" dans lequel nous allons travailler.

1. Ecrire une fonction python qui, étant donné les paramètres, β, p et e_{max} retourne ces deux entiers. On rappelle que l'opérateur puissance en python est `**` et que la division entière s'obtient avec l'opérateur `//`.
2. Dans une console Python :

(a) appeler votre fonction pour obtenir M et $\widetilde{M}$ comme des entiers python ;

(b) récupérer M directement en double en utilisant :

```
import sys
Msys = sys.float_info.max
```

le convertir en entier python (avec la fonction `int`) puis comparer le avec celui de votre fonction (ces deux entiers doivent être égaux !) ;

(c) de même convertir $\widetilde{M}$ en double (avec la fonction `float`), une erreur est générée avec le bon message (i.e. la fonction ne renvoie pas Inf) ;

(d) enfin convertir $\widetilde{M} - 1$ en double et vérifier que $fl(\widetilde{M} - 1) = M$.

Rmq : M et $\widetilde{M}$ dans $\mathbb{F}(2, 53, -1022, 1023)$ sont les entiers suivants :

$$
\begin{aligned}
M = \ & 179769313486231570814527423731704356798070567525844996598917476 8\\
& 03157260780028538760589558632766878171540458953514382464234321 32\\
& 68894641827684675467035375169860499105765512820762454900903893 28\\
& 94407586850845513394230458323690322294816580855933212334827479 78\\
& 26204144723168738177180919299881250404026184124858368
\end{aligned}
$$

$$
\begin{aligned}
\widetilde{M} = \ & 1797693134862315807937289714053034150799341327100378269361737789\\
& 8044496829276475094664901797758720709633028641669288791094655554\\
& 7851940402630657488671505820681908902000708383676273854845817711\\
& 5317644757302700698555713669596228429148198608349364752927190741\\
& 684443655107043427115596995080930428801779041744977 92
\end{aligned}
$$

Contrairement à ce que j'ai pu dire en cours, la gestion des exceptions et du comportement de la FPU n'est pas si triviale avec python. Elle s'obtient avec le module fpectl *mais ce dernier n'est pas "compilé/inclus/construit" par défaut dans python, son usage n'étant recommandé qu'aux spécialistes... Par contre le module* numpy *propose certaines choses dans ce domaine, voir l'exemple ci après.*

```
import numpy as np

# contrôle (ou pas) pour les divisions par 0
x = np.array([5.8, -1, 3])
np.seterr(divide = 'raise')  # retourne les anciens réglages
np.geterr() # retourne les nouveaux réglages
x / 0  # doit générer une exception
np.seterr(divide = 'warn')
x / 0  # doit générer un warning
np.seterr(divide = 'ignore')
x / 0  # pas de message

# contrôle (ou pas) sur les overflow
x = np.array([1.2*10**300])
np.seterr(over='raise')
x**2 # doit générer une exception
np.seterr(over='warn')
x**2 # doit générer un warning (défault)
np.seterr(over='ignore)
x**2 # pas de message
```

Exercice 3 *Une étude de précision*

Voici un exercice provenant du TD1. On cherchait à calculer la fonction :

$$\phi(x) = \frac{1}{1+x} - \frac{1}{1-x} = \frac{-2x}{(1+x)(1-x)}$$

pour un nombre flottant x tel que $|x|$ est assez petit (on suppose que $|x| < 0.1$). Le résultat de cet exercice est qu'une évaluation basée sur la deuxième formule présente une très faible erreur relative alors que la première comporte un risque d'erreur lorsque x se rapproche de zéro. Vous allez réaliser une étude numérique pour mettre en évidence les défauts de la première formule (en se servant de la deuxième comme référence). Pour cela vous allez écrire un script/module python qui :

1. va balayer des petits nombres grâce à la fonction `logspace` du module numpy :

   ```
   n = 2000
   x = np.logspace(-17,-1,n)
   ```

   permet d'obtenir un vecteur x dont les composantes vont de 10^{-17} à 10^{-1} avec n composantes en tout et une répartition logarithmique (on a $x_i/x_{i+1} = Cte$).

2. calcule ensuite les ordonnées correspondantes aux deux formules. Avec y_2 supposée “exacte”, calculer l'erreur absolue (`ea = abs(y1 - y2)`) puis l'erreur relative.

3. et finalement affiche l'erreur relative en échelle log. Attention dans certains cas les deux formules donneront le même résultat (c-a-d que la première fonction marche bien sur certains arguments) et l'erreur relative obtenue (0 donc) ne peut pas s'afficher en échelle log (pourquoi ?). Lors de l'affichage de la courbe vous pouvez sélectionner les composantes des vecteurs qui correspondent à une erreur absolue non nulle grâce à un indiçage booléen obtenu avec l'expression `ea>0` (la courbe en échelle log-log s'obtient avec `loglog(x[ea>0], er[ea>0], ... ))`

   Une “quasi” borne pour l'erreur relative est :

   $$|e_r| \lesssim \frac{2\mathbf{u}}{|x|}$$

   Visualiser cette borne dans le même graphe.

   Rmq : pour obtenir un graphe en échelle loglog, il suffit de remplacer `plot` par `loglog`. Comme python travaille en double précision, on a $\epsilon_m = 2^{-53}$.

Question : pourquoi la partie gauche de la courbe est constante et égale à 1 lorsque $|x|$ est suffisament petit (vous pouvez mieux mettre ce phénomène en évidence en partant de 10^{-18} plutôt que de 10^{-17}) ?