TELECOM Nancy *(ESIAL)*
Maths Numériques

TP1 : arithmétique flottante et systèmes linéaires

Pour ces TP nous utiliserons l'environnement spyder qui facilite l'écriture et l'exécution de scripts python, en particulier lorsqu'on utilise le module matplotlib pour tracer des courbes. Sous linux, l'application spyder peut se lancer à partir du menu (→ Programmation → Spyder) ou en ouvrant un terminal (cliquer sur l'icône correspondante en bas à gauche) et en entrant la commande `spyder &`.

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)`.
2. Expérimenter en autorisant plus de chiffres, ce qui peut se faire avec `print(format(x,fmt))` où fmt est une chaîne de caractères du type :

   $$longueur\_totale\_champ\mathbf{.}longueur\_partie\_décimale\mathrm{f}$$

   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 le module python `sys` (voir page 7 du tutoriel), convertir M en entier python (avec la fonction `int`) et comparer 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 :

$$
\begin{aligned}
M = \; & 179769313486231570814527423731704356798070567525844996598917476 8\\
& 031572607800285387605895586327668781715404589535143824642343213 2\\
& 688946418276846754670353751698604991057655128207624549009038932 8\\
& 944075868508455133942304583236903222948165808559332123348274797 8\\
& 26204144723168738177180919299881250404026184124858368
\end{aligned}
$$

$$
\begin{aligned}
\widetilde{M} = \; & 1797693134862315807937289714053034150799341327100378269361737789\\
& 8044496829276475094664901797758720709633028641669288791094655554\\
& 7851940402630657488671505820681908902000708383676273854845817711\\
& 5317644757302700698555713669596228429148198608349364752927190741\\
& 68444365510704342711559699508093042880177904174497792
\end{aligned}
$$

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 numpi[1] :

   ```
   n = 2000
   x = 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 :

   $$|e_r| \lesssim \frac{2\epsilon_m}{|x|}$$

   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}) ?

1. Avec spyder et en exécutant le fichier via l'interface, vous disposez directement des fonctionnalités des modules numpi et matplotlib (sans avoir à rajouter de préfixe) il est donc en fait inutile de rajouter l'instruction `from pylab import *` dans votre fichier.

Exercice 4 *Mini module d'algèbre linéaire*

Le but de cet exercice est de coder un petit module d'algèbre linéaire[2] appelé syslin contenant les fonctions suivantes :

1. une fonction d'entête :

```
def lu_fact(A):
    # A est une matrice n x n dont on calcule
    # la factorisation LU simple
    # on renvoie 2 objets: le tableau LU qui
    # contient la factorisation de maniere compacte
    # et un entier donnant le statut de calcul
    # istat = -1 si A n'est pas carree (noter qu'il est possible de
    #            conduire une factorisation sur une matrice rectangulaire)
    #          0 OK
    #          k > 0 : le pivot naturel a l'etape k est nul
```

   qui effectue la factorisation $A = LU$ de la matrice A.

   Remarques :

   (a) le tableau d'entrée ne doit pas être modifié, la factorisation sera faite dans un tableau obtenu initialement par copie : `LU = A.copy()` ;

   (b) les dimensions peuvent se récupérer avec `n,m = np.shape(A)`.

   Pour tester cette première fonction, une fois votre module (exécuté/chargé via l'interface graphique de spyder) vous pourrez rentrer les instructions suivantes dans la fenêtre de commande de spyder :

```
A = rand(4,4)                  # une matrice 4x4 avec des réalisations indépendantes de U(0,1)
LU, istat = lu_fact(A)         # factorisation
L = tril(LU,-1) + identity(4) # reconstitution de L
U = triu(LU)                   # reconstitution de U
A - dot(L,U)                   # doit donner une matrice proche de 0
```

2. une autre fonction d'entête :

```
def lu_solve(LU,b):
    # resoud Ax=b connaissant la factorisation LU de A
    y = b.copy()
    n = len(y)
```

   pour résoudre un système linéaire, lorsque l'on connaît la factorisation $A = LU$. Pour tester votre fonction vous pouvez essayer de retrouver la solution du système linéaire de l'exercice 3 de la feuille 2 de TD. Dans ce but vous pouvez écrire la fonction suivante dans le module syslin qui retourne la matrice, le second membre et la solution de ce système linéaire :

```
def exercice3():
    """retourne la matrice, le second membre et la solution du systeme lineaire de l'exercice 3"""
    # on suppose que numpy est importe sous le nom np (import numpy as np)
    A = np.array([[1,2,3,4],[1,4,9,16],[1,8,27,64],[1,16,81,256]],float)
    b = np.array([2,10,44,190],float)
    x = np.array([-1,1,-1,1],float)
    return A,b,x
```

2. Ici l'exercice est d'écrire un vrai module python capable de fonctionner en dehors de l'environnement spyder. Ainsi vous devrez donc importer le module numpy dans votre module, vous pouvez utiliser `import numpy as np` et ainsi toutes les fonctions de `numpy` sont accessibles avec le préfixe `np.`.