Sup'Galilée
MACS1

6 novembre 2018

Initiation aux projets numériques – TP1

Les exercices marqués d'un astérisque sont à réaliser en priorité.

Avant de commencer ce TP, créer un répertoire `PROJETS_NUMERIQUES` puis un sous répertoire `TP1`.

1 Graphisme élémentaire sous matlab

La commande `plot` de matlab permet de tracer des courbes élémentaires sous matlab : si `a` et `b` sont des vecteurs de taille n, la commande `plot`(a,b) va tracer les points de coordonnées $A_i = (a(i), b(i))$ ainsi que les segments $[A_iA_{i+1}]$. De nombreuses options peuvent être associées à cette commande (type de trait, type de points, couleurs des traits et des points ...). Pour obtenir de l'aide sur cette commande, on pourra taper `help plot` dans la fenêtre de commande de Matlab (voir aussi le lien *Reference page for plot* proposé à la fin de l'aide pour obtenir de nombreux exemples).

Ainsi, pour représenter le graphe d'une fonction $f : [a,b] \to \mathbb{R}$, on sera amener à **choisir une subdivision** $x_1, \dots, x_n$ de $[a,b]$ (en général, on prendra une subdivision à pas constant, avec `x=linspace(a,b,n)`), puis on calculera le vecteur $y_1 = f(x_1), \dots, y_n = f(x_n)$ et on représentera `plot(x,y)`, qui interpole linéairement entre les points (x_i, y_i). Ainsi, **la représentation graphique d'une fonction a une précision limitée**, à la fois par les capacités graphiques matérielles (résolution de l'écran), mais aussi par le nombre de points calculés.

Exercice 1* : graphisme élémentaire 2D

Les codes associés à cet exercice pourront être sauvegardés dans un script `Exercice 1.m`.

1. Tracer le cercle de centre $C = (2,4)$ et de rayon $r = 1$ en **rouge** et en **pointillés** en utilisant une équation paramétrique du cercle :

$$\begin{cases} x(\theta) = x_C + r\cos(\theta) \\ y(\theta) = y_C + r\sin(\theta) \end{cases} \qquad \theta \in [0, 2\pi].$$

Assurez-vous que les axes ont bien la même échelle (commande `axis equal`).

2. Ajouter le titre *' cercle de rayon* 1 *et de centre* $(2,4)$ *'* en police de taille 20.

3. En utilisant *Edit plot*, épaissir le trait (par exemple 2).

4. Tracer sur le même graphique un second cercle de même centre et de rayon 0.5 en noir avec un trait épais d'épaisseur 3 **directement dans votre script**, i.e sans utiliser le menu *edit plot*. (Pour cela, sélectionner le cercle rouge dans la fenêtre de *Edit plot*, cliquer sur *More properties* et chercher le mot clé permettant de contrôler l'épaisseur du trait)

5. Sur une deuxième figure, tracer maintenant le disque de centre $(2,4)$ et le rayon 1 à l'aide de la commande `fill`.

6. Sur une troisième figure, tracer le graphe de la fonction

$$f : x \mapsto f(x) = 3x^2 + \frac{\ln\left((x-\pi)^2\right)}{\pi^4} + 1$$

dans l'intervalle $[\pi - 1, \pi + 1]$.

7. Tracer dans une même fenêtre mais sur deux graphes différents les fonctions $\cos x$ et $\sin x$ entre 0 et 2π (fonction `subplot`). Sauvegarder votre figure au format `fig` et `m`. Ouvrir le fichier `.m` obtenu à l'aide de l'éditeur de texte de Matlab.

Exercice 2 : surfaces et volumes

(Script `Exercice 2.m`)

1. Tracer la sphère de rayon $r = 3$ et de centre $(1,1,1)$ (on pourra utiliser la fonction `sphere` de Matlab).

2. Tracer la surface $z = \sin(x^2 + y)$ pour x et y entre 0 et 4π (Commande `surf` ou `mesh`)

2 Représentation des nombres réels en machine et erreurs d'arrondis

2.1 Représentation des nombres réels en machine

Dans un ordinateur, les nombres réels sont stockés en utilisant la méthode 'virgule flottante' : tout nombre réel est représenté dans une base b ($b = 2$ pour un ordinateur), par son signe ($+$ ou $-$), par une mantisse m et par un exposant e :

$$x = \pm\, m\, b^e \tag{1}$$
$$m = d.dd\cdots d \tag{2}$$
$$e = dd\cdots d \tag{3}$$

où $d \in \{0, 1, \cdots b-1\}$ représente un chiffre. Pour rendre la représentation unique, on utilise une mantisse normalisée : le premier chiffre avant le point décimal de la mantisse est non nul.

Exemples.

- *Le nombre $x = 0.625$ sera stocké en binaire $x = 1.01 \cdot 2^{-1}$ ($m = 1.01$, $e = -1$, $b = 2$). En effet, $y = \frac{1}{2} + \frac{1}{8}$.*
- *Le nombre $y = -0.000345678$ s'écrira en utilisant $b = 10$ comme $y = -\,m\,10^e$ où $m = 3.45678$ et $e = -4$. Comme la mantisse peut stocker au maximum n chiffres, si $n \geq 6$, le nombre est stocké de manière exacte. Au contraire si $n < 6$, alors le nombre y n'est pas stocké de manière exacte. Par exemple si $n = 4$, alors $m = 3.456$. Il y a une **erreur d'arrondi**.*
- *Le nombre $z = \frac{1}{3}$ s'écrit, en décimal, $0.3333333\cdots$. Comme son développement est infini, il est **impossible** de stocker z en base 10 sans faire d'erreur d'arrondi. Le même problème se pose en base 2, où z s'écrit $0.0101010101\cdots$ (en effet, $\frac{1}{3} = \frac{\frac{1}{4}}{1-\frac{1}{4}} = \frac{1}{4} + \frac{1}{4^2} + \frac{1}{4^3} + \cdots$). Attention, le nombre $\frac{1}{5}$ s'écrit en décimal 0.2, mais en binaire son développement est infini : seuls les nombres dyadiques (de la forme $\frac{k}{2^n}$) ont un développement fini en base 2.*

La mantisse m est toujours stockée avec une précision limitée (correspondant à n chiffres significatifs). En effet, la mémoire d'un ordinateur n'est pas infinie. Par conséquent, **les nombres ne sont pas toujours stockés de manière exacte**. En particulier, sur un ordinateur (utilisant la base 2) les nombres irrationnels (π, $\sqrt{2}$, ...), mais aussi tous les nombres non dyadiques ($\frac{1}{3}$, $\frac{1}{5}$, $\frac{2}{3}$,...), ne peuvent pas être stockés de manière exacte. En Matlab, le stockage des nombres est le format IEEE double précision (nombres réels stockés en binaire sur 64 bits, cf. la description en annexe A)

2.2 Erreurs d'arrondis : perte de précision numérique

Exercice 3* : précision machine

Dans cet exercice, on utilisera la commande `format long` pour avoir 16 chiffres significatifs affichés à l'écran.

Comme Matlab utilise des nombres flottants, il travaille avec une précision limitée. Par conséquent, il existe un entier k tel que

- $1 + 2^{-k}$ " $\neq$ " 1
- pour tout $q > k$, $1 + 2^{-q}$ " $=$ "1.

L'égalité précédente doit se comprendre au sens suivant : le nombre $1 + 2^{-q}$ et le nombre 1 ont même représentation en Matlab. Autrement dit, Matlab ne fait pas la différence entre 1 et $1 + 2^{-q}$ dès lors que $q > k$. On appelle **précision machine** le nombre $\varepsilon = 2^{-k}$.

L'objectif de cet exercice est de calculer la **précision machine**.

1. Concevoir un algorithme pour calculer la **précision machine**. On pourra utiliser une méthode de dichotomie : on part de $\eta = 2^0 = 1$, et tant que $\eta + 1$ est différent de 1, on divise η par 2. (cf. script `Exercice3.m` fourni)

2. Coder votre algorithme en Matlab.

3. Comparer les résultats obtenus à la constante prédéfinie `eps` de Matlab.

4. Comparer `eps` et `realmin`. Expliquer.

Remarque : les commandes `realmin` et `realmax` permettent de connaitre les plus petits et plus grands nombres stockés au format IEEE de Matlab.

Exercice 4* : défaut de commutativité

Dans cet exercice nous allons évaluer en Matlab la fonction $f : x \mapsto \frac{(1+x)-1}{x}$. Évidemment, pour tout nombre réel x, $f(x) = 1$.

1. Écrire une fonction qui prend comme argument d'entrée un nombre x et qui calcule la quantité $f(x) = \frac{(1+x)-1}{x}$ (on respectera les parenthèses).
2. Tracer l'evolution du nombre $f(x)$ (Calculer avec Matlab) pour x allant de 10^{-1} à $x = 10^{-30}$ (on pourra utiliser la fonction `semilogx` de matlab pour obtenir un affichage de l'axe des abscisses en échelle logarithmique). Expliquer le phénomène obtenu en utilisant l'exercice précédent, et comparer avec l'écriture $f : x \mapsto \frac{(1-1)+x}{x}$. Pour vous aider, vous pouvez regarder l'evolution des quantités $1 + x$ et $(1 + x) - 1$. Vous pouvez aussi calculer $f(\texttt{eps})$ et $f(\texttt{eps}/2)$ où `eps` est la précision machine.

Exercice 5 : erreurs de compensation

On souhaite estimer le nombre $e(\approx 2.71828)$ à l'aide de la relation

$$\lim_{n\to+\infty} \left(1 + \frac{1}{n}\right)^n = e.$$

1. Écrire une fonction Matlab qui prend comme argument un nombre entier n et qui calcule $\left(1 + \frac{1}{n}\right)^n$.
2. À l'aide de la fonction précédente, tracer l'évolution de la quantité $|(1 + \frac{1}{n})^n - e|$ en fonction de n pour n allant de 10 à 10^{16} (En Matlab `e = exp(1)`). On utilisera la commande `plot`.
3. Expliquer le phénomène obtenu.

Exercice 6 : instabilités numériques

Soit $S_n = \displaystyle\int_0^\pi \left(\frac{x}{\pi}\right)^{2n} \sin x\, dx$. Puisque l'intégrant est une fonction positive sur l'intervalle considéré, la suite $(S_n)_{n\in\mathbb{N}}$ est positive.

1. Vérifier que $S_0 = 2$ et montrer que l'on a la relation de récurrence :

$$S_n = 1 - \frac{2n(2n-1)}{\pi^2} S_{n-1} \text{ pour } n \in \mathbb{N}^*. \tag{4}$$

2. Écrire la fonction Matlab `Calcul_Sn` (fichier `Calcul_Sn.m`) qui calcule tous les S_k pour $k \leq n$.
3. Montrer que la suite $(S_n)_{n\in\mathbb{N}}$ tend vers 0. Vérifier que la formule de récurrence (4) conduit à des instabilités numériques en calculant S_1, S_2, $\cdots$, S_{16}. Expliquer.

3 Nombres pseudo-aléatoires

De nombreux domaines d'application utilisent des nombres aléatoires : sécurité informatique (génération automatique d'identifiants, de clés secrètes), méthodes d'optimisation dans des espaces de grande dimension (recuit simulé, algorithmes génétiques), simulations numériques de systèmes complexes (physique, ingénierie, finance, assurance, météo...), jeux vidéos (paysages aléatoires, intelligence artificielle,...),...

A priori, vouloir utiliser un ordinateur pour obtenir des nombres aléatoires apparaît paradoxal, sinon impossible : par définition, un nombre aléatoire n'est pas prévisible, tandis que l'ordinateur ne peut appliquer qu'une formule ou un algorithme prédéfinis. Il existe heureusement des algorithmes fournissant des suites de nombres ayant **de façon approximative** certaines propriétés d'une suite de variables aléatoires indépendantes et de même loi. On parle de nombres **pseudo-aléatoires**.

On s'attend à ce qu'une suite pseudo aléatoire suive la loi des grands nombres, et soit « bien mélangeante ». Une définition plus précise est délicate, et dépendrait en fait de l'application voulue : parfois, on souhaite simplement une suite de valeurs « bien répartie » dans un ensemble, sans s'inquiéter de la dépendance entre termes successifs, tandis que d'autres contextes exigent des contraintes d'indépendance plus importantes (par exemple en cryptographie). De plus, si on a besoin d'une grande quantité de nombres aléatoires, alors il faut un meilleur générateur, qui peut être plus coûteux en temps de calcul.

Dans Matlab, le générateur de nombres aléatoires prend la forme de la "fonction" `rand` (sans paramètre), qui est particulière car son résultat est un nombre réel dans $]0, 1[$ qui change a priori à chaque fois qu'on appelle cette fonction. On fera l'hypothèse suivante :

Les différents appels à `rand` renvoient une réalisation "générique" d'une suite de variables aléatoires indépendantes et de loi uniforme dans $[0, 1]$.

Autrement dit, il existe une suite $(U_n)_{n\geq 1}$ de variables aléatoires (c'est-à-dire de fonctions mesurables $U_n : \Omega \to \mathbb{R}$ définies sur un espace de probabilités $(\Omega, \mathcal{F}, P)$) indépendantes et suivant chacune la loi uniforme sur $[0,1]$, et il existe un $\omega \in \Omega$ tels que `rand` renvoie $U_1(\omega)$, puis $U_2(\omega)$, etc.

La valeur ω est la **graine** du générateur : si on connaît ω, la suite $(U_n(\omega))_{n\geq 1}$ est entièrement connue ! Cela est utile pour reproduire une simulation avec les mêmes "nombres aléatoires". Dans l'hypothèse ci-dessus, on a précisé "générique", pour signifier que ω réalise ("en pratique") n'importe quel événement presque sûr. On pourra donc utiliser `rand` pour observer des propriétés vraies presque sûrement, notamment la loi forte des grands nombres.

Dans Matlab, la commande pour fixer la graine égale à n (entier positif) est `rng(n)`. On peut aussi vouloir laisser Matlab choisir la graine de façon plus "aléatoire" (en fait la graine est choisie en utilisant l'heure, en millisecondes) avec `rng('shuffle')`, auquel cas le résultat du premier appel à `rand` est à peu près imprévisible.

Remarquons qu'en utilisant des nombres pseudo-aléatoires dans Matlab, on combine donc deux approximations : l'approximation numérique (Partie 2 du TP) liée à la précision limitée (`rand` ne donne pas n'importe quel nombre réel dans $]0,1[$), et l'approximation aléatoire liée à l'imitation imparfaite du hasard.

Exercice 7* : générateur et graine, utilisation de `rand` et `rng`

1. Dans la fenêtre de commandes de Matlab, exécuter `rng(1)`, puis exécuter `rand` plusieurs fois et observer les résultats. Exécuter à nouveau `rng(1)` et faire à nouveau appel à `rand` : que remarque-t-on ?
2. Recommencer en remplaçant `rng(1)` par `rng(2)`. Comparer les résultats.
3. Recommencer en remplaçant `rng(1)` par `rng('shuffle')`, et réessayer plusieurs fois : que remarque-t-on ? Est-ce que utiliser ensemble `rng('shuffle')` et `rand` à chaque fois que l'on veut un nombre aléatoire (par exemple dans une boucle) est une bonne méthode pour obtenir une suite de nombres "vraiment" aléatoires ?

Exercice 8* : simuler des variables aléatoires à loi discrète

Bien que Matlab ne fournisse que des nombres qui suivent la loi uniforme dans $[0,1]$, on verra que cela permet d'obtenir des variables aléatoires réelles, ou dans $\mathbb{R}^d$, suivant n'importe quelle loi. On dira que l'on **simule** une loi de probabilité lorsque l'on fait produire à Matlab un nombre aléatoire suivant cette loi.

On s'intéresse pour le moment à la simulation de quelques exemples de lois discrètes.

1. Écrire une fonction `x=signe_aleatoire(p)` qui renvoie un nombre aléatoire qui vaut 1 avec probabilité `p`, et -1 avec probabilité $1-$`p`. *Indication : si U suit la loi uniforme sur $[0,1]$, quelle est la probabilité que U soit inférieur à p ?*
2. Écrire une fonction `vx=signes_aleatoires(n,p)` qui renvoie un vecteur de n nombres aléatoires donnés par la fonction précédente. Exécuter `N=1000;plot(1:N,cumsum(signes_aleatoires(N,0.5))`. Qu'a-t-on représenté ?
3. Écrire une fonction `x=uniforme(n)` qui renvoie un nombre aléatoire choisi de manière uniforme dans l'ensemble $\{1,2,3,\dots,n\}$. *Indication : si U suit la loi uniforme sur $[0,1]$, montrer que $\lfloor nU \rfloor + 1$ suit la loi uniforme sur $\{1,2,\dots,n\}$, où $\lfloor x \rfloor$ est la partie entière (inférieure) d'un réel x (donnée par `floor` dans Matlab).*
4. Écrire une fonction `x=discrete012(p,q)` qui, étant donnés `p` et `q` tels que `p`≥ 0, `q`≥ 0 et `p+q`≤ 1 renvoie un nombre aléatoire X tel que

$$P(X=0)=p, \qquad P(X=1)=q, \qquad \text{et} \qquad P(X=2)=1-p-q.$$

Indication : Utiliser `rand` une seule fois dans la fonction. On pourra commencer par penser au cas $p=q=\frac{1}{3}$.

Exercice 9 : générateurs de nombres aléatoires par congruence linéaire

Pour bien comprendre comment `rand` fonctionne, écrivons notre propre générateur de nombres aléatoires, bien plus simple que celui de Matlab. Les méthodes du type suivant ont été utilisées jusque dans les années 90. On définit la suite $(X_n)_{n\geq 0}$ par récurrence, par la donnée de $X_0 = \omega \in \{1,\dots,2^{31}-1\}$ (c'est la graine) puis

$$\text{pour tout } n \geq 0, \qquad X_{n+1} = 16807\, X_n \pmod{2^{31}-1}.$$

Alors les valeurs $(X_n)_{n\geq 1}$ se comportent approximativement comme des entiers aléatoires uniformément choisis entre 0 et $2^{31}-1 = 2147483647$, et donc $U_n = 2^{-31} X_n$ suit approximativement la loi uniforme sur $[0,1]$.

Par des propriétés arithmétiques, on peut montrer que la suite $X_0, X_1, \dots, X_{N-1}$ parcourt une fois chacune des valeurs de $1,2,\dots,N-1$, où $N = 2^{31}-1$ (c'est un nombre premier), et $X_N = X_0$.

1. On souhaite écrire une fonction `u=alea()` qui renvoie un nombre aléatoire uniforme dans $[0,1]$ à l'aide de cette méthode, et une fonction `graine(n)` qui donne à la graine la valeur `n`. Reproduire les codes suivants et les tester quelques fois (les résultats ont-ils bien l'air aléatoires ?), et comprendre comment il se fait que `alea()` renvoie des valeurs différentes alors qu'il n'y a pas de paramètre.

```
function u=alea()
    global X;
    X=mod(X*16807, 2147483647);
    u=X/2147483648;
end
```

```
function graine(n)
    global X;
    X=n
end
```

2. Si l'on a besoin de plus que $2^{31} (\simeq 2 \cdot 10^9)$ nombres aléatoires, peut-on utiliser ce générateur ?

3. Écrire une fonction `vecteur_alea(n)` qui renvoie un vecteur $[x_1, \ldots, x_n]$ de nombres réels tirés par la fonction `alea`. Pour $n = 10000$, représenter un histogramme des résultats, à l'aide de la fonction Matlab `histogram(x,k)` (où `x` est le vecteur des données, et `k` est le nombre de classes de l'histogramme) ; que met-on ainsi en évidence ? Comparer les histogrammes obtenus pour diverses valeurs de `k` entre 10 et 1000.
4. Écrire une fonction `points2D(n)` qui représente, sur un graphique, n points (x_1, y_1), (x_2, y_2),..., (x_n, y_n), où $x_1, y_1, x_2, y_2, \ldots$ sont aléatoires, indépendants et uniformes dans $[0, 1]$, obtenus grâce à la fonction `alea`. Apprécier visuellement la répartition des points.
5. Écrire une fonction `points3D(n)` qui représente, sur un graphique, n points (x_1, y_1, z_1), (x_2, y_2, z_2),..., (x_n, y_n, z_n), où $x_1, y_1, z_1, x_2, y_2, z_2, \ldots$ sont aléatoires, indépendants et uniformes dans $[0, 1]$, obtenus grâce à la fonction `alea`. *On pourra utiliser `plot3(x,y,z,'.')` où `x`, `y` et `z` sont des vecteurs de même taille.*
6. Entre les années 60 et les années 80, le générateur de nombres aléatoires de certaines bibliothèques FORTRAN très répandues, nommé RANDU, utilisait la récurrence $X_{n+1} = 65539 X_n \pmod{2^{31}}$, puis $U_n = 2^{-31} X_n$, en partant d'un $X_0 \in \{1, \ldots, 2^{31} - 1\}$. Il est aujourd'hui considéré comme exemple caractéristique de *mauvais* générateur. Modifier le code précédent, et exécuter `points3D(1000)`. Est-ce que les points semblent bien répartis dans le cube ? Cela illustre les corrélations entre tirages successifs.

A Annexe : le système IEEE

Le système IEEE 754 est un standard pour la représentation des nombres en virgule flottante en binaire. Il définit les formats de représentation des nombres à virgule flottante (signe, mantisse, exposant). Le nombre est représenté à l'aide de $1 + N_e + N_m$ bits. Le premier bit est le bit de signe : s'il est égal à 1, le nombre est négatif, s'il est égal à 0 le nombre est positif. Les N_e bit suivants représentent l'exposant et les N_m derniers bits représentent la mantisse. L'exposant est décalé de $2^{N_e - 1} - 1$ afin de le stocker sous la forme d'une nombre non signé (cela permet de stocker les exposants négatifs à l'aide d'un nombre positif).

Simple précision

En simple précision, le nombre est représenté sur 32 bits : 1 bit de signe, 8 bits d'exposants ($N_e = 8$, de -126 à 127) et 23 bits de mantisses ($N_m = 23$) (voir Figure 1).

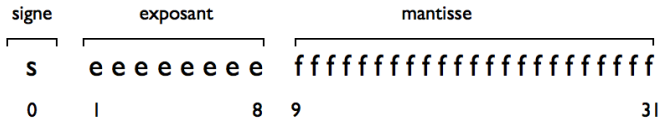

FIGURE 1 – Représentation des nombres flottants suivant le système IEEE simple précision

Exemple. *Écrivons le nombre -118.625 en format IEEE simple précision. Comme le nombre est négatif le bit de signe est 1. La conversion de 118.625 en binaire donne $1110110,101$. On décale alors la virgule sur la gauche pour obtenir un nombre flottant normalisé : $1110110,101 = 1,110110101 \times 2^6$. La mantisse est la partie à droite de la virgule (il est inutile de stocker le chiffre devant la virgule qui est toujours 1 puisqu'on travaille en binaire). On ajoute des 0 à droite pour obtenir 23 bits. On obtient*

$$m = 11011010100000000000000$$

Enfin, on calcul l'exposant décalé. L'exposant est égal à 6. Nous devons décaler l'exposant puis le convertir en binaire. Pour le format simple précision le décalage est de 127 ($2^{8-1} - 1$). Donc $6 + 127 = 133$, qui récrit en binaire donne 10000101. Finalement le nombre -118.625 s'écrit en norme IEEE simple précision

$$1\ 10000101\ 11011010100000000000000$$

Double précision

En double précision, les nombres sont stockés sur 64 bits. 1 bit de signe 11 bits d'exposant ($N_e = 11$, de -1022 à 1023) et 52 bits de mantisse. L'exposant est décalé de $2^{N_e - 1} - 1 = 1023$ (cf. Figure 2).

Exercice : conversions

1. Conversion des nombres entiers :
   - Quel est le nombre entier dont l'écriture en base 7 est 1234 ?

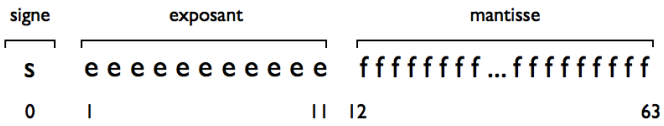

FIGURE 2 – Représentation des nombres flottants suivant le système IEEE double précision

- Écrire en base 2 les nombres 13 et 1286.
- Vérifier vos résultats en Matlab (fonctions `dec2base`,`base2dec` `dec2bin`, `dec2hex`, `base2hex`) .

2. Conversion des nombres décimaux-norme IEEE :
   - Convertir en binaire les nombres 0.625 et 455.40625. Ecrire ces nombres en norme IEEE simple précision.
   - Convertir le nombre binaire b (encodé suivant la norme IEEE simple précision) en nombre décimal :

$$b = 1\,10000011\,0110\,0000\,0000\,0000\,0000\,000$$

   - Afficher le nombre π (`pi`) avec Matlab. Utiliser la commande `format long` pour avoir 16 chiffres significatifs. Le nombre π est-il stocké de manière exacte sous Matlab ?