DIRO
IFT 2425

EXAMEN INTRA

Max Mignotte

DIRO, Département d'Informatique et de Recherche Opérationnelle, local 2377
Http: //www.iro.umontreal.ca/∼mignotte/ift2425/
E-mail: mignotte@iro.umontreal.ca

Date: *19/02/2025*

I	Erreur/Amplification d'erreur en arithmétique flottante (16 pts)
II	Erreur en arithmétique flottante (47 pts)
III	Recherche des racines d'une équation (47 pts)
Total	110 points.

Tous documents personnels, calculatrices et calculateurs autorisés

I. Amplification d'erreur en arithmétique flottante (16 **pts**)

Soit l'algorithme suivant en C:

```
. float Val=10.0/9.0;
. for(int k=0;  k<25;  k++)
.   {
.      printf("%f",Val);
.      Val = (Val-1.0) * 10.0;
.   }
```

ou la suite: $x_{n+1} = (x_n - 1) \times 10$ avec $x_0 = 10.0/9.0$ que l'on peut considérer et implémenter sur sa calculatrice car les résultats seront à peu près identique à ceux donnés par un ordinateur (pour le reste de l'exercice, le raisonnement restera le même).

1. Donner quelle serait le résultat affiché par ce programme (exécuté sur l'ordinateur ou la calculatrice) si la numérotation utilisée par l'ordinateur ou la calculatrice n'était pas en numérotation flottante pour Val (et pour son affichage) mais en numérotation réelle (*i.e.*, sans approximation ou utilisant une infinité de précision).
   <2 pts>

2. Donner maintenant les résultats des 25 itérations données par votre calculatrice (en mode normal, *i.e.*, en écriture décimale ou scientifique <*non écriture fractionnaire*> si vous n'avez pas de calculatrice, vous pouvez utiliser votre ordinateur avec des flottants).
   <4 pts>

3. Expliquez précisément ce qui se passe en utilisant vos connaissances en modélisation mathématique sur la propagation des erreurs (en notation flottante simple précision) sur les termes de cette suite itérative.
   <5 pts>

4. Expliquez ce qui se passe en utilisant maintenant seulement vos connaissances, cette fois ci, en architecture des ordinateurs et en notation flottante (plus précisément au niveau de la mantisse de la représentation flottante de chaque terme de la suite).
   <5 pts>

Réponse

1.

Supposons que la numérotation est réelle (*i.e.*, une numérotation sans approximation ou utilisant une infinité de précision), chacune des 25 itérations de l'algorithme afficherait $1.11111..._{\text{infiniment}} = 1.\overline{1}$ (avec le 1 après la virgule se répétant indéfiniment ou encore $= 10.0/9.0$ si on utilisait une notation fractionnaire).

<2 pts>

2.

Sur ma calculatrice du type *CASIO fx-991MS*, en la programmant, je trouve:
$x_0=1.111111111$ $x_1=1.111111111$ $x_2=1.111111111$ $x_3=1.11111111$ $x_4=1,1111111$ $x_5=1.111111$ $x_6=1.11111$ $x_7=1.1111$ $x_8=1.111$ $x_9=1.11$ $x_{10}=1.1$ $x_{11}=1$ $x_{12}=0$ $x_{13}=-10$ $x_{14}=-110$ $x_{15}=-1110$ $x_{16}=-11110$ $x_{17}=-111110$ $x_{18}=-1111110$ $x_{19}=-11111110$ $x_{20}-111111110$ $x_{21}=-1111111110$ $x_{22}=-1.111111111\cdot 10^{11}$ $x_{23}=-1.111111111\cdot 10^{11}$ $x_{24}=-1.111111111\cdot 10^{12}$
Ce qui est assez différent de ce qui est trouvé précédemment dans la cas d'une représentation réelle sans erreur.
<4 pts>

Nota -1-: Avec une calculatrice du type *CASIO fx-991MS*, par exemple, je rappelle que cette suite itérative se programme facilement en calculant 10/9 puis en appuyant sur la touche $\boxed{=}$ pour afficher 1.111111111 puis en appuyant sur la touche $\boxed{(}$, $\boxed{ANS}$, $\boxed{-}$, $\boxed{1}$, $\boxed{)}$, $\boxed{*}$ puis sur $\boxed{1}$ $\boxed{0}$. Chaque appel à la touche $\boxed{=}$ nous permet ensuite d'avoir une itération de la suite.

Nota -2-: En vrai flottant simple précision, on trouve: $x_0=1.111111$, $x_1=1.111112,\ldots$ $x_{10}=5.309302\cdot 10^2$, $x_{20}=5.298190\cdot 10^{12},\ldots,x_{24}=5.309302\cdot 10^{16}$.

$\boxed{\textbf{3.}}$

Mathématiquement ou analytiquement (pour un numéricien connaissant la théorie de l'amplification d'erreur), ▷ avec la relation itérative $x_{n+1}=f(x_n)=\mathbf{10}\,(x_n-1)$, une erreur ou imprécision sur x_n de ϵ_n est **amplifiée par 10 à l'itération suivante**.
En partant donc de $x_0=10/9$ représentée en flottant (simple précision utilisant seulement 24 bits pour la mantisse) avec une imprécision de ϵ_0, cette imprécision sera donc amplifiée, à chaque itération de la suite, de 10, affectant ainsi x_1 avec une erreur numérique (absolue) de $10\,\epsilon_0$, puis x_2 de $10^2\epsilon_0$, puis x_3 de $10^3\epsilon_0$, ainsi de suite ... Au final, en partant de x_0 avec une imprécision de ϵ_0, cette erreur sera amplifiée, par 10^k à la k-ième itération de la suite et **$\mathbf{x_k}$ sera donc entaché d'une imprécision de $\mathbf{10^k}\epsilon_\mathbf{0}$ sur x_k à la k-ième itérations, avec $\epsilon_\mathbf{0}\approx \mathbf{2^{-24}}$**.
<5 pts>

Nota: Avec le programme demandé à la question II.1, on peut avoir une approximation d'une borne supérieure de l'erreur numérique entachant $x_0=10.0/9.0=1.\overline{1}$. Il suffit de demander à notre programme la valeur que 10.0/9.0 "*absorbera*" ou "*qu'il considérera*" comme négligeable dans une addition avec lui même. Le programme nous donne:
$0.00000047683715820312\approx 0.47\times 10^{-7}$.
La calculatrice utilise une notation proche de celle du DOUBLE qui possède approximativement deux fois plus de bits pour la mantisse. Donc, pour une calculatrice, l'erreur entachant x_0 est approximativement de $\epsilon_0/2^{24}\approx 10^{-15}$. Ce qui veut dire qu'après la 15-ième itération, l'itérée de notre suite n'aura, assurément, plus aucun chiffre significatif ce qui est le cas dans notre exemple.

$\boxed{\textbf{4.}}$

Concrètement, physiquement, *i.e.*; matériellement (ou électroniquement), ▷ à partir de $x_0 = 1.\overline{1}_{10} = 0.xxx...xxx_2$ (avec 24 **bits de précision pour la mantisse, approximant le codage de** $1.\overline{1}_{10}$), la multiplication de notre suite par 10, **à chaque itération, va décaler nos bits informatifs de** x_0 **vers la gauche et faire rentrer à l'extrémité droite de la mantisse des** 0 **supprimant ainsi, d'itération en itération, toute la précision que l'on avait initialement (lors de** x_0**) sur ce nombre**; $x_0 = 1.111111111$, $x_1 = 1.111111111, \dots$ ..., ▶ $x_{12} = 0$

Ensuite une fois que *Val* est devenu 0 (nulle), on rentre dans les négatif avec un décalage de bit créant successivement; $x_{13} = -10$, $x_{14} = -110, \dots$ ..., $x_{24} = -1.111111111 \cdot 10^{12}$.

<5 pts>

II. Erreur/Amplification d'Erreur en Arithmétique Flottante (47 pts)

1. Pour sensibiliser les utilisateurs de machines informatiques que le calcul numérique en notation flottante sur ordinateur peut entraîner des erreurs numériques, on aimerait faire le programme en langage C suivant qui, affiche à l'écran:

   > Entrez une valeur flottante ?:

   Attend une valeur numérique (avec l'instruction en langage C: SCANF), par exemple : 10 (entrée au clavier) puis, l'appuie sur la touche $\boxed{\text{RETURN}}$), permet d'afficher à l'écran:

   > En flottant, [0.00000047683715820312] sera considéré négligeable (nulle) une fois additionnée
   > à: [10.00000] et lui même sera négligeable (ou nulle) lorsqu'il sera additionné avec le float:
   > [268435456.00000]

   Avec, dans cet exemple; 0.00000047683715820312 la plus grande valeur (à 1 % d'erreur relative près [à peu près]) de la véritable plus grande valeur *absorbée* (ou considérée comme nulle ou négligeable) lors de son addition avec 10 (en flottant simple précision). Et 268435456 la plus petite valeur flottante (toujours à **environ** 1 % d'erreur relative près) qui *absorbera* 10 (*i.e.*, sur ordinateur, en flottant simple précision; 268435456+10 = 268435456).

   Il faut bien sur que le programme marche pour différentes entrées au clavier. Par exemple si 0.1 est entré au clavier au lieu de 10, on obtiendra à l'écran:

   > En flottant, [0.00000000372528985437] sera considéré négligeable (nulle) une fois additionnée
   > à: [0.10000] et lui même sera négligeable (ou nulle) lorsqu'il sera additionnée avec le float:
   > [2097152.00000]

   Pour simplifier le programme, on considérera que l'on entre nécessairement au clavier une valeur positive supérieure à 10^{-5}.

   Attention: Votre calculatrice utilise peut-être une numérotation qui est meilleure que la numérotation flottante, et qui est en tout cas différente de la numérotation flottante simple précision d'un ordinateur (en fait, la numérotation flottante d'une calculatrice est plus proche de la numérotation en double *(flottant double précision).*

Faire ce programme en langage C.
<15 pts>

2. Expliquer pourquoi le calcul numérique de ces quatre expressions en notation flottante:

   (a) $1.0 - \sin(x)$
   (b) $\ln(\sin(x)) - \ln(\cos(x)) \qquad x \neq 0 \qquad x \in]0\ ,\ \pi/2[$
   (c) $\frac{1}{\sqrt{x^2+1}-x}$
   (d) $\exp(x) - \exp(-x)$

   peuvent conduire, pour certaine(s) valeur(s) de x, que l'on **oubliera pas d'identifier**, à des problèmes d'erreurs numériques.
   Identifier et citer (tous) ce(s) problème(s) numérique(s) associé(s) à chacune de ces expressions et proposer une expression mathématiquement équivalente qui permettrait d'éviter ces problèmes numériques et/ou augmenter la précision des calculs de ces quatre expressions.
   Pour ces quatre cas, je ne vous demande pas de trouver des expressions qui sont seulement *approximativement* équivalentes, ni encore d'utiliser les développements limités.
   <20 pts>

3. On veut estimer informatiquement, avec le maximum de précision, le produit suivant:

$$\prod_{i=0}^{i<500} x[i]$$

   Dans lequel, $x[i]$ est une valeur flottante aléatoire et distribuée uniformément entre $[10^{-4}\ , 2.6]$ et rangée dans le vecteur x de façon croissante ($x[0] < x[1] < x[2] < \ldots < x[499]$).

   (a) Préciser le(s) type(s) de problème(s) numérique(s) auxquel(s) on sera confronté si on calcule ce produit informatiquement classiquement ou naïvement, tel quel (*i.e.*; $\text{res} \leftarrow x[0]$, $\text{res} \leftarrow \text{res} \times x[1]$, $\text{res} \leftarrow \text{res} \times x[2]$, $\ldots$, $\text{res} \leftarrow, \text{res} \times x[499]$) ou dans le sens inverse.
   <3 pts>

   (b) Proposez trois solutions (assez différentes) en les classant par ordre d'intérêt (et en justifiant votre réponse) permettant d'éviter ou de minimiser fortement ces problèmes numériques et/ou augmenter la précision du calcul de ce produit de flottants.
   <9 pts>

Réponse

$\boxed{\textbf{1.}}$

On pourrait avoir tout simplement le programme suivant:

```
int k;
float Val, Eps;
printf("Entrez une valeur flottante:");
scanf("%f",&Val);
for(Eps=1.0,k=0; ;k++)
  { Eps/=1.0000001;
    if ((Val+Eps)==Val) break; }
printf(" En float, %.20f sera considéré négligeable/nulle une fois additionné à: %.5f",Eps,Val);
Eps=Val;
Val=1.0;
for(k=0; ;k++)
  { Val*=1.0000001;
    if ((Val+Eps)==Val) break; }
printf(" ...et lui même sera négligeable/nulle lorsqu'il sera additionné avec: %.5f",Val);
```

Nota : Si on veut plus de précision, on doit remplacer $eps/= 1.0000001$ et $Val *= 1.0000001$ par deux instructions comportant plus de zéro (*mais pas trop !, sinon l'ordinateur risquerait d'approximer le diviseur par* 1.0 *et du coup, l'ordinateur bouclerait à l'infini en ne faisant strictement rien !*). Dans mon cas, la précision sera plus grande que 1% d'erreur relative. Le programme s'exécute en moins de 2 secondes. **<15 pts>**

Notation ▷: 5 Pts pour la notion de boucle, 5 Pts pour la notion de variable qui croit/décroît et 5 Pts pour la notion de condition d'arrêt.

2.

Notation ▷: 2 pts pour le bon problème, 1 pt pour le bon endroit où il y a problème numérique et 2 pts pour la bonne équivalence. Dans la suite "soustraction de deux nombres approximés quasi égaux" ou "annulation de cse" concerne la même erreur numérique.

[-a-]

Pour la première expression, lorsque $x \to (\pi/2))$, on a un problème numérique de SOUSTRACTION DE DEUX NOMBRES APPROXIMÉS QUASI ÉGAUX (en fait, un seul de ces deux nombres sera approximés mais cela suffit à créer le problème).

On se propose de trouver une expression mathématiquement équivalente, minimisant ce problème, en écrivant simplement:

<5 pts>
$$1.0 - \sin(x) = \frac{(1-\sin(x))(1+\sin(x))}{1+\sin(x)} = \frac{(1-\sin^2(x))}{1+\sin(x)} = \frac{\cos^2(x)}{1+\sin(x)}$$

Nota :

- Une autre possibilité serait d'écrire: $[1.0 - \sin(x)] = 2\sin^2\{(\pi/4) - (x/2))\}$.

[-b-]

Lorsque $x \to (\pi/4)\,[+2\,k\,\pi]$ $(k \in I\!N)$, on a encore un problème numérique de SOUSTRACTION DE DEUX NOMBRES APPROXIMÉS QUASI ÉGAUX que l'on peut simplement éviter en réécrivant l'expression de la façon équivalente suivante:

<5 pts>
$$\ln(\sin(x)) - \ln(\cos(x)) = \ln(\tan(x))$$

[-c-]

Lorsque $x \to \infty$, on a encore un problème numérique de SOUSTRACTION DE DEUX NOMBRES APPROXIMÉS VOISINS (auquel s'ajoute un problème moins grand d'OVERFLOW/UNDERFLOW possible) que l'on peut simplement éviter en réécrivant l'expression de la façon équivalente suivante:

$$\frac{1}{\sqrt{x^2+1}-x} = \frac{\left(\sqrt{x^2+1}+x\right)}{\left(\sqrt{x^2+1}-x\right)\left(\sqrt{x^2+1}+x\right)} = \frac{\sqrt{x^2+1}+x}{1} = \sqrt{x^2+1}+x$$

$$= \text{ou, } |x|\sqrt{1+(1/x^2)}+x$$

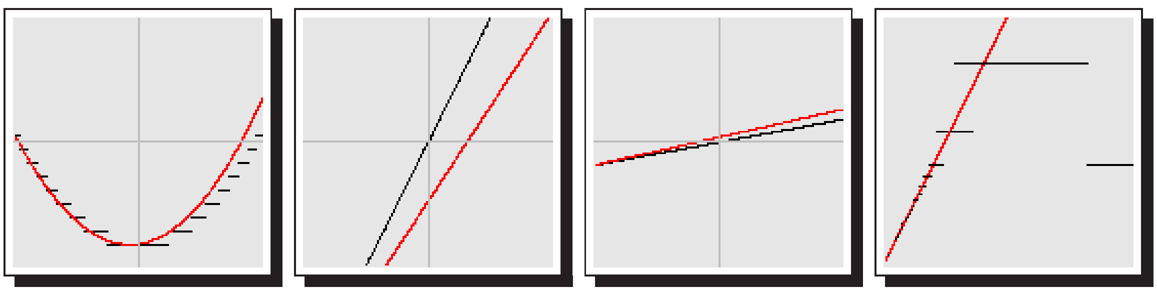
<5 pts>

[-d-]

SOUSTRACTION DE DEUX NOMBRES APPROXIMÉS VOISINS au voisinage de 0, (en plus aussi d'un problème d'UNDERFLOW et d'OVERFLOW quand $x \to \pm\infty$ et d'un problème de soustraction de deux nombres d'ordre de grandeur très différents quand $x \to \pm\infty$) et que l'on peut facilement éviter en utilisant la définition de la fonction sinus hyperbolique, $\mathrm{sh}\,(x)$:

$$\exp(x) - \exp(-x) = 2\,\mathrm{sh}\,(x)$$

<5 pts>



De gauche à droite: expression (a-b-c-d) pour: (a) $[x \in \pi/2 \pm 10^{-4}]$ et $[y \in \pm 10^{-7}]$ (b) $[x \in \pi/4 \pm 10^{-5}]$, et $[y \in \pm 10^{-5}]$ (c) $[(x,y) \in [0.5{\cdot}10^{1}\ \ 0.5{\cdot}10^{3}]$ (d) $[(x,y) \in \pm 0.2 \cdot 10^{-7}]$.
Avec, la courbe en rouge correspondant à cette expression sans erreur numérique [ou correspondant à l'expression mathématiquement équivalente que l'on a trouvé en FLOAT] et en noire, l'expression flottante initiale entaché d'erreurs numériques.

$\boxed{\textbf{3.(a)}}$

Dans le pire des cas, on sera probablement confronté à un problème d'**UNDERFLOW** si on calcule ce produit dans le sens direct et un problème d'**OVERFLOW** si on calcule ce produit dans le sens inverse (débordement/sous-débordement). Dans tous les cas, on aura une **PERTE DE PRÉCISION** qui sera d'autant plus importante que le nombre de termes de ce produit sera important.
<3 pts>

$\boxed{\textbf{3.(a)}}$

Pour minimiser fortement ces problèmes numériques et ainsi augmenter la précision du calcul de ce produit on peut proposer, en ordre décroissant de préférence ou d'intérêt Celle qui permettant d'éviter ou de minimiser le mieux les problèmes numériques de UNDERFLOW/OVERFLOW et le manque de précision:

1. Multipliez le terme d'indice 0 avec celui d'indice 499 puis multipliez le résultat par le terme d'indice 1 avec celui d'indice 498, …, et enfin $x[249] \times x[250]$.

2. Mélanger les termes de ce produit pour que ceux-ci ne soit plus classés par ordre croissant ou décroissant.

3. Procéder en deux étapes:

   (a) Estimer en premier : $S_{\log} = \sum_{i=0}^{i<500} \log(x[i])$ (log pour logarithme népérien)

   (b) Calculer finalement: $S = \exp(S_{\log})$

<9 pts>

Nota -1-: J'ai implémenter le calcul de ce produit sur ordinateur en notation flottante et j'ai bien obtenu un UNDERFLOW pour le calcul en sens direct et un OVERFLOW pour le calcul dans le sens inverse et une estimation assez précise pour les deux solutions proposées. J'ai programmé ce produit avec ces deux méthodes et en deux numérotations; la numérotation FLOTTANTE et LONG DOUBLE supposé sans (trop) d'erreur surtout avec la méthode Exponentiel-Log car on dispose en C d'une fonction logarithmique et exponentiel en Long Double très précise. j'obtiens comme estimation la plus précise $S = 0.0262207\,268\,669249$. Pour la version *aux deux extrémités*; $S = 0.\underline{0262207}\,314...$, pour la version *mélangée*; $S = 0.\underline{0262207}370...$ et la version *Log-Exponentiel*; $S = 0.\underline{02622}\,11710...$

Nota -2-: *D'autres pistes intéressante mais demandant un petit peu d'algorithmie:*

• Une autre possibilité serait d'essayer de conserver/garder le produit partiel dans un intervalle, disons [0.1 : 10] en prenant le maximum étant dans la liste quand on atteint une valeur (partielle du produit) plus petite que notre borne inf. c-à-d; 0.1 et en prenant la plus petite valeur encore existants dans la liste lorsqu'on atteint notre borne max, c-à-d; 10. Cela demanderait un petit peu d'algorithmie simple à programmer et une complexité $\mathcal{O}(n^2)$.

• On pourrait multiplier toute les valeurs par une constante; par exemple K et ensuite dire que le total doit être divisé par K^{500}, avec un peu d'algorithmie, cela se fait bien.

• Une autre possibilité serait de diviser ce produit en plusieurs plus petits blocs puis utiliser ensuite une stratégie *ad-hoc* permettant de minimiser le problème d'**UNDERFLOW/OVERFLOW** ... en les classant par ordre croissant et en alternant le produit entre les termes extrêmes par exemple.

• Une autre possibilité serait de faire cette multiplication en séparant les valeurs flottantes et leurs exposant (mais cela ne serait pas si simple que cela algorithmiquement....

III. Méthode du Point Fixe/Newton/Bissection (47 pts)

On se propose de trouver numériquement une valeur approchée d'une racine de la fonction

$$f(x) = \ln(x) - \exp(-x) \tag{1}$$

1. Méthode du Point Fixe

   (a) Montrer qu'il existe une racine unique r pour cette Eq. (1) dans l'intervalle $J = [1\,,\,2]$. et donner une forme $x = g_1(x)$, mathématiquement équivalente à $f(x) = 0$ (Eq. (1)), pour laquelle la suite itérative $r_{n+1} = g_1(r_n)$ converge lorsque, le premier élément de cette suite est le milieu de l'intervalle J, c.-à-d.; $r_0 = 1.5$.
   <5 pts>

   (b) Utiliser le résultat de la question précédente pour calculer les 6 premières estimées $r_1, \ldots, r_6$, en partant de $r_0 = 1.5$.
   <5 pts>

   (c) Pour obtenir à l'avance le nombre d'itérations nécessaires de la méthode du point fixe pour arriver à la racine souhaitée avec la précision voulue, on peut essayer de trouver (en utilisant le théorème des accroissements finis ou de la valeur moyenne) une majoration du type $|r_n - r| \leq K$, ou r_n désigne la valeur approchée, à la nième itération, de cette racine. Trouver cette majoration et en déduire le nombre d'itérations nécessaires pour obtenir, par cette méthode, une valeur approchée à 10^{-5} près de cette racine.
   <5 pts>

(d) Trouver une fonction $g_2(x)$ équivalente à l'équation $f(x)=0$ et pour laquelle vous démontrerez qu'elle n'est pas contractante sur J (i.e., pour laquelle la convergence vers une solution unique par la méthode du point fixe n'est pas assurée).
<**5 pts**>

2. Méthode de la Bissection

    (a) Si on avait utilisé la méthode de la bissection, avec combien d'itérations serait-on arrivé à une valeur approchée de la racine à 10^{-5} près ? (Nota: on vous demande d'obtenir une estimation de ce nombre en utilisant les propriétés de la méthode de la bissection mais sans calculer les différentes estimations $r_0, r_1, \ldots,$ données par cette méthode.)
    <**5 pts**>

3. Méthode de Newton

    (a) Donner la relation itérative $r_{n+1} = g_{\text{newt.}}(r_n)$ intervenant dans la méthode itérative de Newton pour la résolution itérative de la racine r de l'équation $f(x) = 0$.
    <**5 pts**>

    (b) Montrer qu'avec $r_0 = 1.5$, la relation itérative de Newton converge.
    <**5 pts**>

    (c) Calculer les 4 premières estimées $r_1, \ldots, r_4$, en partant de $r_0 = 1.5$.
    <**5 pts**>

4. Méthode de Householder
    Soit la relation itérative de HouseHolder:

$$r_{n+1} = r_n - \frac{2\, f(r_n) \cdot f'(r_n)}{-f(r_n) \cdot f''(r_n) + 2\, [f'(r_n)]^2}$$

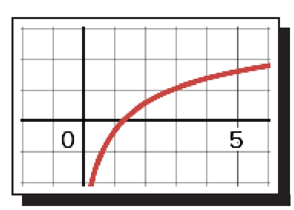
    (a) Si on voulait implémenter sur ordinateur cette relation itérative de Householder avec des dérivés numériques les plus simples (pour la dérivée première et la dérivée seconde), quel est le nombre d'évaluations de fonctions (minimal) que demanderait une itération de cette relation itérative de Householder.
    <**3 pts**>

    (b) Supposons maintenant que $N_{\text{iters}}^{\text{Newt}}$ représente le nombre d'itérations que demande l'approche itérative de Newton pour arriver à convergence de la racine solution avec 9 cse, et que $N_{\text{iters}}^{\text{Hous}}$ désigne, quand à lui, le nombre d'itérations de la méthode de Householder pour arriver à cette même précision.
    Définir la relation entre $N_{\text{iters}}^{\text{Newt}}$ et $N_{\text{iters}}^{\text{Hous}}$ (et ce que vous avez estimez à la question précédente), pour pouvoir affirmer que la méthode de Householder est finalement plus rapide (en terme de coût calculatoire et pour cet exemple) que la méthode (simple) de Newton ?
    <**4 pts**>

Réponse

$\boxed{\textbf{1.(a)}}$

L'étude des variations de la fonction f sur $J=[1\,,\,2]$ montre que la fonction est continue et **strictement croissante sur** J (donc monotone) car $f'(x) = (1/x) + (\exp(-x)) > 0$ sur $J = [1\,,\,2]$.
De plus, on a $f(1) = -\exp(-1) \approx -0.36$ et $f(2) \approx 0.55$ donc $\mathbf{f(1)f(2)} < \mathbf{0}$ et, puisque la fonction f est continue et croissante sur J, il existe **donc une racine r unique dans cet intervalle** J.
<**2 pts**>



$f(x)=\ln(x) - \exp(-x)$

On trouve facilement que: $f(x)=0$ est équivalente à l'équation $\ln(x) = \exp(-x)$ ou encore $x = \exp(\exp(-x) = g_1(x)$ avec $g_1'(x) = -\exp(-x)\cdot \exp\big(\exp(-x)\big) = -\exp\big(\exp(-x)-x\big)$.

$\forall\, x \in [1\,,\,2]$, $\exp(-x)-x < \exp(-1)-1$ et donc $\forall\, x \in [1\,,\,2]$, $\exp(\exp(-x)-x) < \exp(\exp(-1)-1) \approx 0.53$. De ce fait, $|g_1'(x)| < 0.53 < 1$, $\forall\, x \in J$, et la convergence de ce point fixe est assurée.

<3 pts>

1.(b)

En partant de $r_0 = 1.5$, on a, $r_{n+1} = g_1(r_n) = \exp\big(\exp(-r_n)$ et,

$$
\begin{aligned}
r_1 &= 1.249983261\\
r_2 &= 1.331770943\\
r_3 &= 1.302140308\\
r_4 &= 1.312520212\\
r_5 &= 1.308839555\\
r_6 &= 1.310139145
\end{aligned}
$$

qui va converger doucement vers la valeur $r = 1.309799586$.

<5 pts>

Nota : 1 cse après la virgule pour la seconde itération, 2 cse après la virgule pour la quatrième itération, 3 cse pour la sixième itération; il s'agit bien d'un schéma itératif associé à une convergence typiquement *linéaire*, avec une constante asymptotique plus proche de 1 que de 0.

1.(c)

En utilisant le théorème de la valeur moyenne, on obtient, puisque $g(r) = r$ et $r_n = g_1(r_{n-1})$ avec r_n la valeur approchée de la racine à la nième itération;

$$
\begin{aligned}
r_n - r &= \frac{g_1(r_{n-1}) - g_1(r)}{(r_{n-1}-r)} \times (r_{n-1} - r)\\
&= g_1'(\zeta) \times (r_{n-1} - r) \qquad \text{avec} \quad \zeta \in J
\end{aligned}
$$

En utilisant l'inégalité $|g_1'(\xi)| < 0.956$ (cf. question 1.(a)), on obtient les inégalités suivantes;

$$
|r_n - r| \le\; 0.53\,|r_{n-1} - r| \;\le\; 0.53^2\,|r_{n-2} - r| \;\le \ldots \le\; 0.53^n\,\underbrace{(r_0 - r)}_{\le 1}
$$

On obtiendra donc $|r_n - r| < 10^{-5}$ dès que $(0.53)^n < 10^{-5}$, *i.e.*, dès que $\mathbf{n = 19}$. Il s'agit bien sûr d'une borne supérieure. Si on veut quelque chose de plus précis, alors on doit prendre $g_1'(\zeta = r = 1.309799586) \approx 0.353$ conduisant à $n = 12$ itérations (les deux réponses sont acceptables).

<5 pts>

1.(d)

$f(x){=}0$ est équivalent à $x = g_2(x) = -\ln(\ln(x)$. Sur J, $g_2(x)$ ne convergera pas car $g_2(x)$ est non défini sur J (particulièrement en $x=0$) et aussi car $|g_2'(x)| = \big|-1/[x\,\ln(x)]\big| > 1$ sur J, car $\forall\, x \in [1\,,\,2]$, $x\,\ln(x) \in [0\,,\,2\ln(2)]$ et donc $|g_2'(x)| = [x\,\ln(x)]^{-1} \in [(2\ln(2))^{-1}\ ,\ \infty]$. Par exemple, pour $x = 1.5$ $|g_2'(x=1.5)| \approx 1.64 > 1$ (en fait ce seul contre exemple ou la non définition de $g_2(x)$ sur J suffisait à le prouver).

<5 pts>

Nota :

- $f(x){=}0$ est équivalent à $x = g_3(x) = x + \ln(x) - \exp(-x)$. Sur J, $g_3(x)$ ne convergera pas car $|g_3'(x)| =$

$\big|1 + (1/x) + \exp(-x)\big| > 1$ sur J. Par exemple, pour $x = 2.0$; $|g_3'(x = 2.0)| \approx 1.63 > 1$ (un seul contre exemple suffit à le prouver).
• Même chose pour $f(x) = 0$, équivalent à $x = g_4(x) = x + K \cdot \{\ln(x) - \exp(-x)\}$ avec $K \in]0, +\infty[$

2.

La méthode de la bissection permet de construire à partir de l'intervalle $[1\ ,\ 2]$ de largeur ≈ 1.0 contenant r, un nouvel intervalle de longueur moitié contenant r. En appliquant n fois consécutives la méthode, on obtient un intervalle de longueur $\approx 1.0/2^n$ contenant r. À l'itération n, on a donc un majorant de l'erreur absolu donné par $\Delta r = 1/2^n$. Or on veut $\Delta r < 10^{-5}$, donc $2^n > 1.0 \times 10^5$, c-à-d., $\boxed{\mathbf{n = 17}}$, ce qui, dans notre cas est à comparer avec $n = 24$ obtenu pour la méthode itérative du point fixe précédente.
<5 pts>

3.(a)

La fonction $f(x)$ est dérivable sur $\mathbb{R}$ et la méthode itérative de Newton permet d'écrire:

<5 pts>

$$r_{n+1} = r_n - \frac{f(r_n)}{f'(r_n)} = r_n - \frac{\ln(r_n) - \exp(-r_n)}{(1/r_n) + \exp(-r_n)}$$

3.(b)

Dans ce cas, le plus simple est de montrer que la fonction $f(x)$, sur l'intervalle $J = [1\ ,\ 2]$ (intervalle dans lequel on prendra $r_0 = 1.5$ et dans lequel une racine unique s'y trouve) ne présente pas d'*extrema*, *i.e.*, de valeurs pour laquelle, $f'(x)$ s'annule.

Dans notre cas, $f(x) = \ln(x) - \exp(-x) = 0$ et $f'(x) = (1/x) + \exp(-x)) > 0$, toujours positif sur J donc aucun problème !
<5 pts>

3.(c)

En partant de $r_0 = 1.5$, on a, $r_n = g_{\text{newt.}}(r_{n-1}) = r_n - \frac{\ln(r_n) - \exp(-r_n)}{(1/r_n) + \exp(-r_n)}$, on trouve:

$$\begin{aligned} r_1 &= 1.295082492 \\ r_2 &= 1.309710106 \\ r_3 &= 1.309799583 \\ r_4 &= 1.309799586 \end{aligned}$$

qui va converger (quadratiquement) en 4 itérations vers la valeur $r = 1.309799586$. Cela nous indique aussi que la fonction dont on cherche la racine est très très linéaire au voisinage de la racine.
<5 pts>

Nota : 4 cse après la virgule pour la seconde itération, 8 cse (*i.e.*; le double) après la virgule pour la troisième itération; il s'agit bien d'un schéma itératif associé à une convergence typiquement *quadratique*.

4.(a)

• Une évaluation de fonction pour $f(x_i)$.

• Une de plus; pour calculer $f(x_i + \epsilon)$ dans l'expression $f'(x_i)$, en utilisant la dérivé numérique la plus simple, vue en TP ou en cours, du style: $f'(x_i) \approx [f(x_i + \epsilon) - f(x_i)]/\epsilon$, avec ϵ petit.

• Et une de plus; pour calculer $f''(x_i) \approx [f(x_i + 2\epsilon) - 2\,f(x_i + \epsilon) - f(x_i)]/\epsilon^2$, que l'on trouve facilement en disant que $f''(.)$ est une différence numérique de dérivée première.

▷ Donc en tout **TROIS** évaluations de fonctions (par itération).

<3 pts>

4.(b)

Rappelons que la relation itérative de Newton ne demande que deux évaluations de fonctions. Soit $N_{\text{iters}}^{\text{Newt}}$, le nombre d'itérations que demande l'approche itérative de Newton pour arriver à convergence (de la racine) solution avec 9 cse; Il faudrait donc que la relation de Householder converge en moins de $2/3 \cdot N_{\text{iters}}^{\text{Newt}}$ itérations pour que cette méthode soit computationellement intéressante en terme de coût calculatoire), *i.e.*, qu'elle converge avec moins de $2/3 \cdot N_{\text{iters}}^{\text{Newt}}$ itérations pour qu'elle converge avec un plus faible coût calculatoire que la méthode de Newton.

La question est donc: ▷ $\mathbf{N}_{\text{iters}}^{\text{Hous}} < \mathbf{2/3} \cdot \mathbf{N}_{\text{iters}}^{\text{Newt}}$?

<4 pts>

Nota -1-: La question 3.(c) nous a montré que l'approche itérative de Newton demandait 4 itérations pour arriver à convergence (de la racine) solution avec 9 cse. Il faudrait que la méthode de Householder puisse arriver à ce même résultat donc en moins de $2/3 \times N_{\text{iters}}^{\text{Newt}} = 8/3 = 2.\overline{6}$ itérations, c-à-d en 2 itérations.

Nota -2-: Testons l'efficacité de la méthode de Householder:

$$r_{n+1} = r_n - \frac{2\, f(r_n) \cdot f'(r_n)}{-f(r_n) \cdot f''(r_n) + 2\, [f'(r_n)]^2} = r_n - \frac{2\left(\ln(r_n) - e^{-r_n}\right)\left((1/r_n) + e^{-r_n}\right)}{-\left(\ln(r_n) - e^{-r_n}\right)\left((-1/r_n^2) - e^{-r_n}\right) + 2\left(1/r_n + e^{-r_n}\right)^2}$$

En partant de $r_0 = 1.5$, on trouve:

$$\begin{aligned}
r_1 &= 1.5 - \frac{2\left(\ln(1.5) - e^{-1.5}\right)\left((1/1.5) + e^{-1.5}\right)}{-\left(\ln(1.5) - e^{-1.5}\right)\left((-1/(1.5)^2) - e^{-1.5}\right) + 2\left(1/1.5 + e^{-1.5}\right)^2} \\
&= 1.3097101450 \\
r_2 &= 1.3097995520 \\
r_3 &= 1.3097995520
\end{aligned}$$

qui converge bien en moins de trois itérations (c-à-d en 2 itérations). Les résultats après le 8-ième chiffre après la virgule sont ici différents de ceux données à la question 1.(b) car ceux donnés par mon ordinateur en flottant (*versus* ma calculatrice pour la question 1.(b).)
Si je demande à mon ordinateur de faire ces mêmes calculs en double; j'obtiens:

$$\begin{aligned}
r_0 &= 1.50 \\
r_1 &= 1.3097101145 \\
r_2 &= 1.3097995858 \\
r_3 &= 1.3097995858
\end{aligned}$$

qui sont les mêmes que ceux donnés par ma calculatrice, mais avec un chiffre de plus après la virgule.

Remarque: 4 cse après la virgule pour la première itération et plus du triple cse après la virgule pour la seconde itération (en fait; toute la précision de ma calculatrice); il s'agit bien d'un schéma itératif associé à une convergence typiquement *cubique*.

En conclusion: inutile de prendre une méthode plus compliquée, qui converge sans doute, en moins d'itérations, mais qui, si elle demande plus d'opérations (*i.e.*, évaluation de fonctions dans notre cas) par itération, n'est peut être pas finalement *beaucoup* plus rapide (en terme de coût calculatoire ou de rapidité). Ceci dit cela dépend, bien sur du problème considéré. Dans notre exemple, les deux méthodes demandent approximativement le même coût calculatoire pour arriver à la précision maximale d'un résultat avec 9 cse mais la méthode de Newton est beaucoup moins difficile à implémenter...