MÉTHODES DE MONTE-CARLO
CORRIGÉ DE L'EXAMEN BLANC

par Rémi Peyre

PROBLÈME 1 — La récompense de Mickey Thunk

Mickey Thunk*, génial informaticien, vient d'achever la programmation du logiciel T$\mathcal{O}$C, destiné à révolutionner la monde de la typographie. Il est tellement fier de sa programmation qu'il défie quiconque de trouver le moindre bug dans son programme ! Une personne qui trouverait un tel bug remporterait ainsi un prix de 100 €. Mieux, une personne qui trouverait un second bug après le premier remporterait pour sa part 200 €, la troisième 400 €, et ainsi de suite en doublant la prime à chaque nouveau bug trouvé.*

En fait, Mickey Thunk ne pense pas que son programme soit totalement exempt de bugs, mais il estime qu'il ne doit plus en rester que « trois ou quatre ». Il modélise le nombre total de bugs dans son programme par une loi de Poisson de paramètre 3,5. Notons X une variable aléatoire suivant, sous une probabilité $\mathbb{P}$, la loi $\mathcal{P}oisson(3{,}5)$, et Y la variable aléatoire correspondant au montant total de la récompense, qui est une fonction de X. Mickey Thunk se demande combien va coûter sa récompense en moyenne ; autrement dit, il souhaite calculer $\mathbb{E}(Y)$ (il estime que tous les bugs finiront par être trouvés).

1. Écrire une fonction MATLAB appelée `Poisson` qui prend un argument λ et renvoie un nombre aléatoire suivant la loi $\mathcal{P}oisson(\lambda)$.
Indication : On rappelle que, si $T_1, T_2, \dots$ sont des v.a. i.i.d. suivant une loi $\mathcal{E}xponentielle(\lambda)$, alors

$$\sup\left\{k \in \mathbb{N} : \sum_{i=1}^{k} T_i \leqslant 1\right\} \tag{1}$$

suit la loi $\mathcal{P}oisson(\lambda)$.

Corrigé.

```
function X = Poisson(lambda)
% La fonction "Poisson" prend en argument un réel positif lambda et renvoie
% une variable aléatoire simulée selon la loi Poisson(lambda).

% On utilise la méthode de la simulation de la loi de Poisson à partir
% d'une suite de lois exponentielles. "somme" est la somme d'un certain
% nombre de variables exponentielles indépendantes, et "indice" le nombre
% de telles variables qu'on a sommées.
somme = 0;
indice = 0;
% On commence une boucle "while" dans laquelle on somme les variables
% exponentielles jusqu'à ce qu'elles dépassent 1.
while(1)
    % T est une variable aléatoire de loi Exponentielle(lambda), simulée
    % selon la méthode de la fonction de répartition (inverse).
    T = -log(rand)/lambda;
    somme = somme + T;
    if(somme > 1)
        break;
    else
        indice = indice + 1;
```

```
    end
end
% La valeur retournée est celle de l'indice de la dernière variable T
% sommée avant que la somme atteigne 1.
X = indice;
end
```

✓

2. Écrire une fonction MATLAB appelée `Y` qui prend un argument X et renvoie la valeur de la récompense totale Y correspondant au nombre de bugs X. On essaiera de programmer cette fonction de façon qu'elle soit la plus rapide possible.

Corrigé. La valeur de la i-ème récompense attribuée par Mickey Thunk est $100 \times 2^{i-1}$. Par conséquent, la récompense totale s'il y a X erreurs est

$$Y = \sum_{i=1}^{X} 100 \times 2^{i-1} = 100 \times (2^X - 1)$$

d'après la règle de sommation des séries exponentielles.

D'où le programme suivant :

```
function recompense = Y(X)
% La fonction "Y" calcule, pour un nombre d'erreurs X, le montant de la
% récompense totale correspondante.

recompense = 100*(2^X-1);
end
```

✓

3. Évaluer $\mathbb{E}(Y)$ par une méthode de Monte-Carlo « naïve ». On écrira une fonction MATLAB appelée `MCnaif` qui prend en argument le nombre N de simulations demandées et renvoie un intervalle de confiance à 90 % (soit $\pm 1{,}65$ sigmas) pour $\mathbb{E}(Y)$. (On admettra que Y est L^2).

Corrigé. Le programme est le suivant :

```
function I = MCnaif(N)
% La fonction "MCnaif" évalue la valeur moyenne de la récompense totale
% offerte par Mickey Thunk, par la méthode de Monte-Carlo avec N
% simulations.

% Pour faciliter la lecture, on définit "lambda" comme valant 3,5.
lambda = 3.5;
% "somme" et "sommecarrés" stockeront respectivement la somme et la somme
% du carré des quantités simulées.
somme = 0;
sommecarres = 0;
% Voici la boucle principale dans laquelle on simule à N reprises la
% récompense totale.
for ii = 1:N
    recompense = Y(Poisson(lambda));
    somme = somme + recompense;
    sommecarres = sommecarres + recompense*recompense;
end
```

```
% On calcule la moyenne et l'écart-type empiriques de la variable
% "recompense".
moyenne = somme / N;
ecarttype = sqrt(sommecarres / N - moyenne*moyenne);
% On renvoie l'intervalle de confiance à 90%, soit 1.65 sigmas.
I = moyenne + ecarttype/sqrt(N) * [-1.65,1.65];
end
```

Avec $N = 1\,000\,000$, je trouve un intervalle de confiance égal à [3 177 €, 3 231 €]. ✓

On cherche maintenant à évaluer $\mathbb{E}(Y)$ à l'aide d'un échantillonnage préférentiel. La loi d'échantillonnage retenue est une loi de Poisson de paramètre 5.

4. Quelle est la densité relative de la loi $\mathcal{P}oisson(5)$ par rapport à la loi $\mathcal{P}oisson(3{,}5)$?
Indication : On rappelle que si P est une loi de Poisson de paramètre λ, alors pour tout $n \in \mathbb{N}$

$$P(\{n\}) = \frac{\lambda^n}{n!e^\lambda}. \tag{2}$$

Corrigé. Dans le cas de lois discrètes comme ici, la densité relative au point n de $\mathcal{P}oisson(5)$ (que nous noterons ici P_5 pour faciliter la lecture) par rapport à $P_{3,5}$ est simplement le quotient

$$densité(n) = \frac{P_5(\{n\})}{P_{3,5}(\{n\})} = \frac{5^n \,/\, (n!e^5)}{3{,}5^n \,/\, (n!e^{3,5})} = e^{3,5-5}\left(\frac{5}{3{,}5}\right)^n.$$

✓

5. Évaluer $\mathbb{E}(Y)$ par la méthode de Monte-Carlo avec échantillonnage préférentiel suggerée ci-dessus. On écrira une fonction MATLAB appelée `MCpref` qui prend en argument le nombre N de simulations demandées et renvoie un intervalle de confiance à 90 % pour $\mathbb{E}(Y)$. (On admettra qu'il n'y a toujours pas de souci avec les questions d'intégrabilité L^2).

Corrigé. Notant $\mathbb{E}'$ l'espérance sous la loi d'échantillonnage préférentiel $\mathcal{P}oisson(5)$, il faut donc évaluer

$$\mathbb{E}'\left(\frac{Y(X)}{densité(X)}\right).$$

Cela se fait par le programme suivant :

```
function I = MCpref(N)
% La fonction "MCpref" évalue également la valeur moyenne de la récompense
% totale offerte par Mickey Thunk, mais en utilisant l'échantillonnage
% préférentiel.

% Pour faciliter la lecture, on définit "lambda" comme valant 3,5 et
% "lambdapref" comme valant 5. Il sera également utile de définir le
% quotient de ces deux grandeurs, ainsi que la quantité
% exp(lambda-lambdapref).
lambda = 3.5;
lambdapref = 5;
quotient = lambdapref / lambda;
Cte0 = exp(lambda - lambdapref);
```

```
% "somme" et "sommecarrés" stockeront respectivement la somme et la somme
% du carré des quantités simulées.
somme = 0;
sommecarres = 0;
% Voici la boucle principale.
for ii = 1:N
    % On simule la loi biaisée.
    X = Poisson(lambdapref);
    % On évalue la récompense pour cette simulation biaisée.
    recompense = Y(X);
    % Il faut corriger par la densité relative de la loi biaisée par
    % rapport à la loi véritable.
    densite = Cte0 * quotient^X;
    quantitedinteret = recompense / densite;
    somme = somme + quantitedinteret;
    sommecarres = sommecarres + quantitedinteret*quantitedinteret;
end
% On calcule la moyenne et l'écart-type empiriques de la variable
% "quantitedinteret".
moyenne = somme / N;
ecarttype = sqrt(sommecarres / N - moyenne*moyenne);
% On renvoie l'intervalle de confiance à 90%, soit 1.65 sigmas.
I = moyenne + ecarttype/sqrt(N) * [-1.65,1.65];
end
```

Quand j'exécute ce programme avec $N = 1\,000\,000$, je trouve un intervalle de confiance de $[3\,206\ €, 3\,218\ €]$. (C'est nettement plus précis qu'avec l'échantillonnage naïf ; cela dit les calculs sont un peu plus longs, car notre méthode de simulation de la loi de Poisson est d'autant plus lente que le paramètre est élevé). ✓

PROBLÈME 2 — Mouvement brownien fractionnaire

On définit le mouvement brownien fractionnaire de paramètre $7/8$ *comme un processus* gaussien centré $(H_t)_{t\geqslant 0}$ *à valeurs réelles tel que* H_0 *soit égal à* 0 *presque-sûrement et que, pour tous* $0 \leqslant s \leqslant t$, $(H_t - H_s)$ *ait pour écart-type* $(t-s)^{7/8}$. *On admettra qu'un tel processus existe avec des trajectoires continues.*

1. Montrer que pour tous $t_1, t_2 \in \mathbb{R}_+$,

$$\operatorname{Cov}(H_{t_1}, H_{t_2}) = \tfrac{1}{2}\big(t_1^{7/4} + t_2^{7/4} - |t_2 - t_1|^{7/4}\big). \tag{3}$$

Corrigé. Pour fixer les idées, supposons que $t_1 \leqslant t_2$: la formule (3) étant clairement invariante par permutation de t_1 et t_2, cela suffira à prouver le cas général. Une formule faisant intervenir $\operatorname{Cov}(H_{t_1}, H_{t_2})$ est celle de la variance de $(H_{t_2} - H_{t_1})$:

$$\operatorname{Var}(H_{t_2} - H_{t_1}) = \operatorname{Var}(H_{t_1}) + \operatorname{Var}(H_{t_2}) - 2\operatorname{Cov}(H_{t_1}, H_{t_2}).$$

Dans cette formule, nous connaissons $\operatorname{Var}(H_{t_2} - H_{t_1})$: c'est $\big((t_2 - t_1)^{7/8}\big)^2 = (t_2 - t_1)^{7/4}$. Pour trouver $\operatorname{Var}(H_{t_1})$, on observe que c'est aussi $\operatorname{Var}(H_{t_1} - H_0)$, ce qui permet d'établir par le même raisonnement que cette quantité vaut $t_1^{7/4}$; de même, $\operatorname{Var}(H_{t_2}) = t_2^{7/4}$. En injectant toutes ces valeurs dans la formule de $\operatorname{Var}(H_{t_2} - H_{t_1})$, on en déduit alors la formule (3) recherchée. ✓

2. Simuler ce mouvement brownien fractionnaire (en le plottant) sur l'intervalle $[0, 1]$ par la méthode de Cholesky. Le code-source sera une fonction MATLAB appelée `MBf`, prenant en argument le nombre points retenus pour la discrétisation, ceux-ci étant régulièrement espacés. Lancer cette simulation pour un nombre de points au moins égal à 1 024, et faire une copie d'écran du diagramme obtenu (au format `.png` de préférence), la figure étant affichée en grande fenêtre.

Corrigé. Voici le programme d'une simulation avec exactement 1 024 points :

```
function MBf
% La fonction MBf simule un mouvement brownien fractionnaire (de paramètre
% 7/8) sur l'intervalle [0,1], par la méthode de Cholesky.

% M est le nombre de sous-intervalles de temps dans la simulation, que je
% prends ici égal à 1023.
M = 1023;
% Pour gagner du temps, je calcule une fois pour toutes la valeur de 7/4.
septquarts = 7/4;
% On crée la matrice C des covariances entre H_{i/M} et H_{j/M}, càd. la
% matrice des 1/2 * [(i/M)^{7/4} + (j/M)^{7/4} - |j/M-i/M|^{7/4}].
% J'utilise ici le formalisme matriciel, car MATLAB est particulièrement
% rapide dans ce cadre. A est la matrice des (i/M), et B la matrice des
% (j/M).
A = (1:M)'/M * ones(1,M);
B = ones(M,1) * (1:M)/M;
C = (A.^septquarts + B.^septquarts - abs(B-A).^septquarts) /2;
% J'applique la méthode de Cholesky pour construire un vecteur(-ligne)
% gaussien centré de matrice de covariance C.
vecteur = randn(1,M) * chol(C);
% Je trace mon plot, en n'oubliant pas d'ajouter le point de temps 0 qui
% n'a pas été simulé.
vecteur = [0 vecteur];
temps = (0:M)/M;
plot(temps,vecteur);
end
```

La figure 1 présente le résultat que j'ai obtenu en lançant ce programme. ✓

3. Le mouvement brownien fractionnaire n'est pas un processus markovien. Sauriez-vous expliquer pourquoi (sans faire une preuve formelle), rien qu'en regardant l'allure de la courbe obtenue ?
Indication : Il pourra éventuellement être utile de zoomer sur le diagramme.

Corrigé. Un processus markovien est un processus qui « ne se souvient que de son présent » : pour prévoir son évolution future, seul son état présent compte ; sa trajectoire passée n'a aucune influence. Or ici, on a l'impression (juste) que le mouvement brownien fractionnaire a une « inertie » : quand il commence à descendre, il reste sur une tendance descendante longtemps, et vice-versa. Par conséquent, pour prédire son futur, il est utile de connaître son passé qui nous dira sur quelle tendance il est : ce n'est donc pas un processus markovien ! Cette tendance à l'inertie, si nous faisons une simulation plus raffinée, est confirmée par le fait que la trajectoire du mouvement brownien fractionnaire est beaucoup moins découpée (comme on le voit en zoomant) que celle d'un processus markovien comme le mouvement

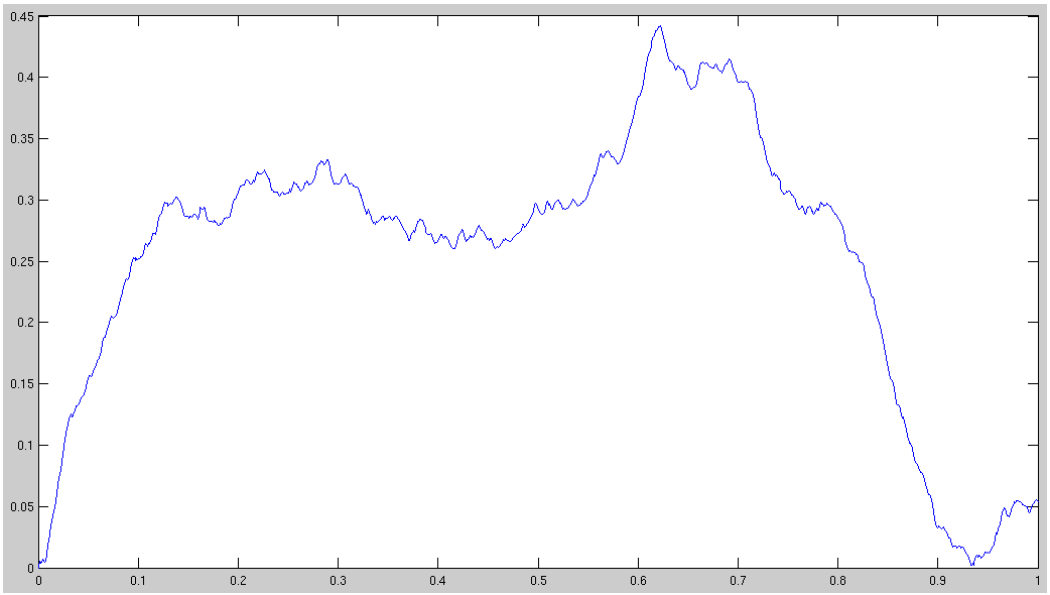

FIGURE 1 – Tracé par MATLAB d'un mouvement brownien fractionnaire.

brownien (non fractionnaire) : le fait que le mouvement brownien fractionnaire ait de l'inertie a tendance à lisser sa trajectoire. ✓

4. On définit la variable aléatoire $X = \sup_{t\in[0,1]} |H_t|$. En admettant que 1024 points de simulation constituent une approximation suffisante pour la question qui nous intéresse, évaluer $\mathbb{E}(X)$ par la méthode de Monte-Carlo. Le programme s'appellera `supMBf_a`, prendra comme seul argument le nombre de simulations demandé, et reverra un intervalle de confiance à 2 sigmas pour $\mathbb{E}(X)$.

Corrigé. Il faut simuler N trajectoires de mouvement brownien fractionnaire, calculer $\sup_{t\in[0,1]} |H_t|$ pour chacun de ces trajectoires et en faire la moyenne. Pour la simulation des trajectoires, nous nous inspirons du programme `MBf`, mais nous ne l'utilisons pas tel quel, d'abord parce que nous avons besoin de récupérer les données numériques des trajectoires et pas seulement leurs tracés, ensuite parce que les tracés ne servent à rien et qu'ils prendraient un temps considérable, et aussi parce que certains calculs intermédiaires comme celui de la matrice de Cholesky gagnent à être faits une fois pour toutes.

Mon programme (qui inclut déjà la réponse à la question 7) est le suivant :

```
function I = supMBf_a(N)
% La fonction "supMBF_a" évalue E(X) par la méthode de Monte-Carlo naïve,
% en faisant N simulations. Ce programme recyclant en grande partie "MBf",
% je ne commenterai pas les parties communes aux deux.

M = 1023;
septquarts = 7/4;
A = (1:M)'/M * ones(1,M);
B = ones(M,1) * (1:M)/M;
C = (A.^septquarts + B.^septquarts - abs(B-A).^septquarts) /2;
% Ici je calcule une fois pour toutes la matrice de Cholesky, sous peine
% d'une perte de temps considérable!
Cholesky = chol(C);
```

```
% On va maintenant passer à la boucle de Monte-Carlo proprement dite. Les
% variables "somme" et "sommecarres" font ce que leurs noms suggèrent.
somme = 0;
sommecarres = 0;

% L'efficacité sera évaluée en chronométrant uniquement la durée de la
% boucle. Cette partie du programme est à commenter si on ne souhaite pas
% calculer l'efficacité.
tic;

for ii= 1:N
    % Simulation d'une trajectoire du mouvement brownien fractionnaire.
    trajectoire = randn(1,M) * Cholesky;
    % Calcul de la variable X correspondant à la simulation. Noter qu'on
    % n'a pas besoin d'utiliser la valeur de la trajectoire en 0
    % puisqu'elle ne contribue pas au supremum.
    X = max(abs(trajectoire));
    somme = somme + X;
    sommecarres = sommecarres + X*X;
end

% Arrêt du chronomètre. Cette partie du programme est à commenter si on ne
% souhaite pas calculer l'efficacité.
t = toc;

% Moyenne, écart-type et intervalle de confiance.
moyenne = somme / N;
variance = sommecarres / N - moyenne*moyenne;
I = moyenne + sqrt(variance / N)*[-2,2];

% On calcule enfin l'efficacité. Cette partie du programme est à commenter
% si on ne souhaite pas calculer l'efficacité.
efficacite = N / t / variance;
disp('Efficacité :')
disp(efficacite)

end
```

Quand je lance ce programme avec $N = 1\,000$, je trouve un intervalle de confiance de $[0{,}843; 0{,}916]$. ✓

5. Calculer exactement $\mathbb{E}(|H_1|)$.

Corrigé. D'après la formule (3) appliquée avec $t_1 = 0$ et $t_2 = 1$, on a $\mathrm{Var}(H_1) = 1$, et comme le processus est gaussien centré, cela signifie que H_1 suit une loi $\mathcal{N}(0,1)$. D'après l'expression de la densité de cette loi, $\mathbb{E}(|H_1|)$ vaut donc

$$\int_{\mathbb{R}} |x| \, \frac{e^{-x^2/2}}{\sqrt{2\pi}} \, dx.$$

Dans cette expression, la fonction à intégrer est paire, donc l'intégrale se simplifie en

$$2\int_0^{+\infty} x \, \frac{e^{-x^2/2}}{\sqrt{2\pi}} \, dx = \frac{2}{\sqrt{2\pi}} \big[-e^{-x^2/2}\big]_0^{+\infty} = \sqrt{2/\pi}(1-0) = \sqrt{2/\pi}.$$

✓

6. En utilisant $|H_1|$ comme variable de contrôle, améliorer la méthode de Monte-Carlo précédente. Le programme s'appellera `supMBf_b`.

Corrigé. La méthode de la variable de contrôle consiste, pour un β bien choisi, à réécrire $\mathbb{E}[X]$ comme $\mathbb{E}[X - \beta|H_1|] + \beta\mathbb{E}[|H_1|]$, sachant que $\mathbb{E}[H_1]$ vaut $\sqrt{2/pi}$. La méthode pour trouver le β optimal est expliquée par le théorème 35.a du cours (et la remarque 35.d qui le suit). Dans le programme qui suit, comme les calculs de simulation sont assez coûteux, j'utilise une seule et même boucle pour évaluer β et pour appliquer la méthode de Monte-Carlo (voir les commentaires pour plus d'explications) :

```
function I = supMBf_b(N)
% La fonction "supMBF_b" évalue E(X) par la méthode de Monte-Carlo avec la
% variable de contrôle |H_1|. Ce programme recyclant en grande partie
% "supMBf_a", je ne commenterai pas les parties communes aux deux.

M = 1023;
septquarts = 7/4;
A = (1:M)'/M * ones(1,M);
B = ones(M,1) * (1:M)/M;
C = (A.^septquarts + B.^septquarts - abs(B-A).^septquarts) /2;
Cholesky = chol(C);
% À la fin de la boucle qui suit, on veut être en mesure de faire deux
% choses : d'abord, d'évaluer beta, puis d'appliquer à proprement parler la
% méthode de Monte-Carlo avec variable de contrôle. Du coup, on va calculer
% un grand nombre de quantités : la "sommeX" des X, la somme "sommeX2" des
% X^2, la somme "sommeH1" des |H_1|, la somme "sommeH12" des |H_1|^2 et la
% somme "sommeH1X" des produits |H_1|X.
sommeX = 0;
sommeX2 = 0;
sommeH1 = 0;
sommeH12 = 0;
sommeH1X = 0;

% À commenter si on ne souhaite pas calculer l'efficacité.
tic;

for ii= 1:N
    trajectoire = randn(1,M) * Cholesky;
    X = max(abs(trajectoire));
    H1 = abs(trajectoire(M));
    sommeX = sommeX + X;
    sommeX2 = sommeX2 + X*X;
    sommeH1 = sommeH1 + H1;
    sommeH12 = sommeH12 + H1*H1;
    sommeH1X = sommeH1X + H1*X;
end

% À commenter si on ne souhaite pas calculer l'efficacité.
t = toc;

% Je calcule maintenant les moyennes empiriques associées aux sommes que
% nous venons de calculer.
moyempX = sommeX / N;
moyempX2 = sommeX2 / N;
moyempH1 = sommeH1 / N;
```

```
moyempH12 = sommeH12 / N;
moyempH1X = sommeH1X / N;
% Grâce aux calculs précédents, j'évalue la variance empirique de |H_1| et
% la covariance empirique de |H_1| avec X :
varempH1 = moyempH12 - moyempH1*moyempH1;
covH1X = moyempH1X - moyempH1*moyempX;
% On en déduit une évaluation du beta optimal :
beta = covH1X / varempH1;
% Maintenant, l'astuce est qu'à partir des résultats de la boucle, nous
% pouvons dire sans faire de calculs supplémentaires quelles auraient été
% les moyenne et moyenne des carrés (empiriques) trouvées pour la variable
% X - beta*|H_1| (pour la moyenne des carrés, il faut développer le carré
% (X-beta*|H_1|)^2) :
moyenne = moyempX - beta * moyempH1;
moyennecarres = moyempX2 + beta*beta * moyempH12 - 2*beta * moyempH1X;
% On en déduit l'intervalle de confiance pour E(X-beta*|H_1|) :
variance = moyennecarres - moyenne*moyenne;
Ibrut = moyenne + sqrt(variance / N) * [-2,2];
% Et on rectifie pour tenir compte de la variable de contrôle :
I = Ibrut + beta * sqrt(2 / pi);

% À commenter si on ne souhaite pas calculer l'efficacité.
efficacite = N / t / variance;
disp('Efficacité :')
disp(efficacite)

end
```

Quand j'exécute ce programme, je trouve un intervalle de confiance de [0,852 ; 0,864]. ✓

7. Modifier les fonctions `supMBf_a` et `supMBf_b` pour que, au cours de leur exécution, elles affichent aussi leur efficacité (inutile de changer les noms des fonctions). Comparer les efficacités et commenter.

Corrigé. Concernant le calcul de l'efficacité, il est déjà pris en compte dans les codes ci-dessus. Les efficacité que je trouve sur ma machine sont d'environ 1 800 s^{-1} pour `supMBf_a` et de 80 000 s^{-1} pour `supMBf_b` : on gagne plus d'un facteur quarante ! Si le gain en efficacité est aussi important, c'est parce qu'ici le coefficient de corrléation de Pearson entre X et H_1 est très proche de 1, comme on peut le voir en s'amusant à modifier le programme `supMBf_b` pour la calculer : il vaut environ 0,987. Cette très forte corrélation entre X et H_1 s'explique par le fait que, comme le mouvement brownien fractionnaire a une forte inertie, sa trajectoire a généralement tendance à suivre à peu près la même direction tout du long, et par conséquent le supremum de $|H_t|$ sur $[0,1]$ est atteint pour une valeur de t très proche de 1 [∗], en vertu de quoi X est très proche de H_1.

Par ailleurs, on aurait pu craindre que le gain d'efficacité par étape qu'apporte la variable de contrôle soit altéré par la complexité accrue des calculs, mais en fait dans nos deux boucles la partie la plus chronophage est de loi la simulation de la trajectoire du mouvement brownien fractionnaire, ce qui fait que le temps de calcul

[∗]. Même si ce n'est pas du tout le cas sur la trajectoire particulière que j'ai dessinée à la question 2.

par étape n'est pratiquement pas ralenti par la prise en compte de la variable de contrôle. ✓

PROBLÈME 3 — Rétroingénierie

Question. Voici un programme. Dites tout ce que vous trouverez d'intelligent à dire dessus.

```
function A = mystere(B)
C = min(B,4096);
D = B-C;
E = 0;
F = 0;
for G = 1:C
    H = sqrt(1-rand^2);
    E = E+H;
    F = F+H*H;
end
I = F/C-(E/C)*(E/C);
for J = 1:D
    K = sqrt(1-rand^2);
    E = E+K;
end
A = 4*(E/B+I/sqrt(B)*[-3,3]);
end
```

Corrigé. On peut raisonnablement soupçonner, vu le sujet du cours, que la fonction `mystere` fera un calcul de Monte-Carlo et que son argument B sera le nombre de simulations. Cela est conforté par l'exécution du programme : `mystere(1000)` renvoie la matrice [3,172; 3,206], et `mystere(1000000)` renvoie [3,140 3; 3,141 5], lesquelles matrices évoquent fortement des intervalles de confiance. Toutefois ces intervalles, bien que proches, ne s'intersectent pas, ce qui est bizarre... Laissons ce problème de côté pour l'instant.

Essayons de voir quelle est la quantité que ce programme calcule. `mystere(100000000)` renvoie un intervalle centré sur environ 3,141 7, ce qui suggère fortement que ce programme estime la constante d'Archimède π. Gardons cela en tête pour vérifier plus tard si cela est bien le cas.

Le programme comporte deux boucles qui se ressemblent beaucoup. La première de ces boucles évoque fortement celles que nous avons vues dans notre cours, puisqu'on y évalue une quantité aléatoire H à chaque itération, dont on calcule la somme et la somme des carrés. À ce stade, on peut essayer de réécrire le programme avec des noms de variables plus parlants afin d'y voir plus clair :

```
function intervalle = pi(N)
C = min(N,4096);
D = N-C;
somme = 0;
sommecarres = 0;
for ii = 1:C
    X = sqrt(1-rand^2);
    somme = somme+X;
    sommecarres = sommecarres+X*X;
end
```

```
I = sommecarres/C-(somme/C)*(somme/C);
for J = 1:D
    K = sqrt(1-rand^2);
    somme = somme+K;
end
intervalle = 4*(somme/N+I/sqrt(N)*[-3,3]);
end
```

Après cette réécriture, il est flagrant que I doit désigner une variance. Par ailleurs, K joue le même rôle que X et peut être renommée en X sans altérer le fonctionnement du programme. On peut également renommer sans danger l'indice J également en ii [†]. Un dernier mystère reste toutefois à résoudre : pourquoi y a-t-il deux boucles ?

La première boucle comporte C itérations et la seconde D. D ayant été défini comme $N - C$, cela fait N itérations en tout, ce qui conforte notre déduction que N correspond à un nombre de simulations. Si nous regardons l'effet de la première boucle et l'effet de la seconde sur *somme*, celui-ci est le même. La différence est que dans la première boucle, on calcule également *sommecarrés*, mais pas dans la seconde ! Par conséquent, la première boucle permet de déterminer un intervalle de confiance, alors que la seconde concerne simplement la méthode de Monte-Carlo « L^1 ».

Nous nous rappelons alors ce qui est écrit dans le cours en page 9, sur la réponse à la « seconde objection » et dans la remarque 9.a : pour calculer l'intervalle de confiance, on n'a besoin d'évaluer la variance de la quantité d'intérêt qu'avec une précision moyenne ! Et là, tout s'illumine : l'idée de ce programme est de gagner du temps en évitant de passer trop de temps à calculer la variance : si l'exécutant a demandé beaucoup de simulations, on ne se servira que des premières d'entre elles (en l'occurrence, les 4 096 premières) pour calculer la variance, et ensuite on ne calculera que l'espérance !

Bref, notre programme rétro-ingénié est finalement le suivant :

```
function intervalle = pi(N)
% Cette fonction évalue pi par la méthode de Monte-Carlo à partir de N
% simulations.

% On va se servir des N1 premières simulations pour estimer la variance de
% la quantité d'intérêt, puis pour les N2 suivantes on ne cherchera plus
% qu'à affiner l'estimation de l'espérance. On prend N1 = min(N,4096).
N1 = min(N,4096);
N2 = N-N1;

% "somme" va stocker la somme de N simulations de la quantité d'intérêt.
somme = 0;
% "sommecarres" ne va servir que dans la première boucle, et stockera la
% somme de N1 simulations de la quantité d'intérêt.
sommecarres = 0;
% Première boucle.
for ii = 1:N1
```

[†]. J'utilise ici le nom de variable "ii" plutôt que "i" simplement à cause du fait que pour MATLAB, i est par défaut égal au nombre complexe de carré -1. On pourrait réécrire sur la variable i sans bug, mais cela empêcherait ensuite de l'utiliser en tant que racine carrée de -1 s'il y en avait besoin — ce ne sera pas le cas ici, mais autant prendre tout de suite de bonnes habitudes...

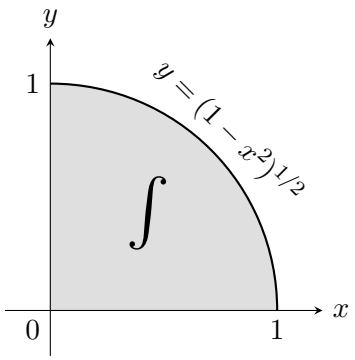

FIGURE 2 – L'intégrale $\int_0^1 (1-u^2)^{1/2}\,du$ égale à $\mathbb{E}[X]$ est l'aire du quart de cercle dessiné ci-dessus, soit $\pi/4$.

```
    X = sqrt(1-rand^2);
    somme = somme+X;
    sommecarres = sommecarres+X*X;
end
% À ce stade on évalue la variance de la quantité d'intérêt.
variance = sommecarres/C-(somme/C)*(somme/C);
% Seconde boucle.
for ii = 1:N2
    X = sqrt(1-rand^2);
    somme = somme+X;
end
% On renvoie un intervalle de confiance à 3 sigmas pour l'espérance de la
% quantité d'intérêt, ou plus précisément pour cette espérance multipliée
% par 4.
intervalle = 4*(somme/N+sqrt(variance)/sqrt(N)*[-3,3]);
end
```

Ici, il convient de noter une dernière modification : j'avais oublié [‡] une racine carrée dans la formule de l'intervalle de confiance ! Cela explique pourquoi nos intervalles avaient tendance à tomber légèrement à côté de π... On vérifie que rajouter la racine carrée règle bien le problème.

Reste encore quelques observations à faire. La première : l'espérance de la variable aléatoire X que nous avons évaluée est-elle bien (avant multiplication par 4) égale à $\pi/4$? Si nous regardons bien, X est donné par la formule $X = (1-U^2)^{1/2}$, où U est uniforme sur $[0,1]$. Par conséquent,

$$\mathbb{E}[X] = \int_0^1 (1-u^2)^{1/2}\,du.$$

On pourrait essayer de calculer algébriquement cette intégrale, mais c'est un peu compliqué... Par contre, observant que la courbe $y = (1-x)^2$ décrit un quart de cercle unité dans le premier quadrant quand x parcourt $[0,1]$ (voir figure 2), l'interprétation d'une intégrale comme aire nous montre que l'intégrale ci-dessus est simplement l'aire de ce quart de cercle unité : c'est bien $\pi/4$.

À ce sujet : est-il utile de la préciser, cette méthode de calcul de π est assez moisie. Déjà, rien que si on veut calculer π comme une intégrale, il n'est pas judicieux d'utiliser la métode de Monte-Carlo, attendu qu'il s'agit d'une intégrale de

[‡]. C'était fait exprès... !

fonction régulière en dimension 1 (cf. § I.4.c du cours). Et même, les méthodes de calculs efficaces de π utilisent plutôt des formules de série[§] qui convergent avec une précision exponentielle, bien meilleure que la précision des formules d'intégrale, laquelle est toujours polynomiale (avec ou sans Monte-Carlo)...

Pénultièmement, vous remarquerez que la programmation a été ici optimisée pour ne perdre aucun calcul. Je veux dire, on aurait pu faire dans un premier temps une boucle de 4 096 simulations pour calculer $\mathrm{Var}(X)$, puis ensuite lancer N simulations pour la méthode de Monte-Carlo à proprement parler : ici, on a été plus malin en réutilisant les 4 096 premières simulations pour le calcul de l'espérance, conformément à ce que suggèrait la remarque 11.a du cours. Notez aussi que, quand le nombre de simulations demandées est petit, notre programme ne s'acharne pas à lancer 4 096 simulations juste pour avoir l'intervalle de confiance : c'est à cela que correspond le choix de poser "`N1 = min(N,4096)`".

Et pour finir : cette astuce de programmation était-elle utile ? Pour le savoir, j'ai modifié le programme pour afficher son temps de calcul, et j'ai fait des tests avec ou sans l'utilisation de la seconde boucle (je ne reproduis pas les codes ici). Avec $N = 10^8$, je trouve un temps de calcul de 9,61 s (avec des fluctuations de quelques millisecondes d'une exécution à l'autre) pour le programme "amélioré", contre un temps de calcul pour le programme "naïf"... exactement égal ! (compte tenu de la marge d'erreur). Bref, ces astuces diaboliques n'avaient en fait aucun intérêt pratique ! La raison en est vraisemblablement que les opérations de génération aléatoire et de calcul de racine carrée prennent énormément plus de temps que l'opération de mise à jour de la variable $sommecarrés$. ✓

[§]. Certains algorithmes modernes de calcul de π reponsent sur des algorithmes itératif à convergence exponentielle d'exponentielle. Mais ceux-ci demandent quand même une très grande puissance de calcul, ce qui fait qu'ils n'ont pas ringardisé pour autant les formules de séries.