Rapport

LU3IN028 - Algorithmique numérique
Rapport des travaux sur machines encadrés

Licence de Sciences et Technologies

Bi-Disciplinaire Informatique et Mathématiques

Année universitaire 2019 - 2020

Encadrant : Pr Stef Graillat
Groupe : Mr Vincent Fu
Mr Yuhao Liu

Table des matières

Préambule :

Tous les algorithmes et leurs tests sont dans le dossier contenant les fichier .m. Les algorithmes sont repérables par leurs noms tandis que les tests sont nommées de la manière suivante tmeA_ exoB_ ... avec A et B les numéros qui les correspondent, les ... indiquent les numéros des questions.

Première partie

Travaux sur machines encadrés

I.1 TME n°0 : Introduction à MATLAB et à l'arithmétique à virgule flottante

Exercice 1

1.

```
format longE;
function res = Higham(x)
   for i=1:52
      x=sqrt(x);
   end
   for i=1:52
      x = x.^2;
   end
   res = x;
end
Higham(4)
```

Le programme ci-dessus doit normalement afficher 4.

2.
Le résultat affiché sur l'ordinateur est `2.718281808182473e+00`.
Lors de l'exécution du programme, à la sortie de la première boucle, `x` devient une valeur très proche 1, c'est-à-dire $1 + \epsilon$ avec ϵ un certain erreur très petit (proche de 0). ϵ vaut en fait 2^{-52}. Sachant que $(1 + \frac{1}{n})^n \to exp(1)$, on en déduit qu'en sortie d'algorithme, on obtient une valeur proche de $exp(1)$. D'où le résultat.

3.

```
x = logspace(0, 1, 2013);
y = Higham(x);
plot(x, y, 'k.', x, x, '--')
```

En exécutant le programme ci-dessus, on obtient le graphique suivant :

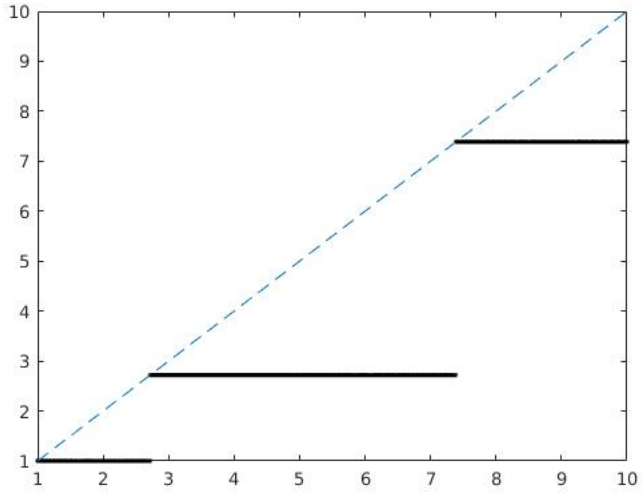

FIGURE 1 – *Le graphe de Higham en gras et le graphe identité en pointillé*

Sachant que $(1 + \frac{a}{n})^n \rightarrow exp(a)$, en fonction des valeurs entrées dans le programme `Higham`, a est nécessairement une puissance de 2. On en déduit alors selon l'intervalle où se situe la valeur d'entrée, on obtient en sortie $exp(a)$. Ces intervalles sont justement indiquées par les points $y = x$.

Exercice 2

1.
Voici l'algorithme permettant de calculer la précision machine ϵ :

```
function res = eps_pratique
   res = 1;
   while 1 + res > 1
      res = res / 2;
   end
   res = res * 2;
end
```

On obtient en pratique $\epsilon =$ `2.220446049250313e-16`. En comparant avec la constante de `eps` de MATLAB, on obtient le même résultat.

2.
Voici l'algorithme permettant de calculer le plus petit nombre flottant positif normalisé α :

```
function res = alpha_pratique
   tmp = 1;
   while tmp > 0
      res = tmp;
      tmp = tmp / 2;
   end
   res = res / eps_pratique;
end
```

On obtient en pratique $\alpha =$ `2.225073858507201e-308`. En comparant avec `realmin`, on obtient le même résultat.

3.
Le plus petit nombre flottant positif dénormalisé correspond à `eps * realmin`.
On obtient comme valeur `4.940656458412465e-324`.

4.
Voici l'algorithme permettant de calculer le plus grand nombre flottant positif :

```
function res = fmax_pratique
   tmp = 1;
   while tmp < inf;
      res = tmp;
      tmp = tmp * 2;
   end
end
```

On obtient en pratique `8.988465674311580e+307`. `realmax` vaut `1.797693134862316e+308`, ce qui correspond à `2 * fmax_pratique`.

Exercice 3

1.
Soit la fonction f définie par $f(x) = \frac{exp(x)-1-x}{x^2}$.
Lorsque x est proche de 0, nous observons des erreurs d'élimination. En effet, $exp(x)$ - 1 - x et x^2 sont proches de 0.

2.
Pour x proche de 0, on peut appliquer le développement limité de Taylor à l'ordre 2 (ou d'ordre supérieur) à $exp(x)$, puis en simplifiant f, on obtient uniquement une somme de plusieurs termes. Sachant que les termes du développement de Taylor sont placés dans l'ordre décroissante en valeur absolue (si $|x| < 1$), on obtient les valeurs de $f(x)$ en commençant par additionner le dernier terme à son précédent, puis d'additionner ce dernier par le l'avant dernier terme et ainsi de suite jusqu'au premier terme.
Voici l'algorithme décrivant la méthode ci-dessus permettant de calculer f au voisinage de 0 :

```
function res = f_voisinzero(x, n)
   res = 0;
   fact = 1;
   i = 1;
   while i <= n
      fact = fact * i;
      i = i + 1;
   end
   i = i - 1;
   while i >= 2
      res = (x^(i - 2)) / fact + res;
      fact = fact / i;
      i = i - 1;
   end
end
```

Exercice 4

1.
Voici l'algorithme qui calcule la fonction sinus en utilisant l'approximation de Taylor à l'ordre n au voisinage de 0 :

```
function res = sin_taylor(x, n)
   res = 0;
   fact = 1;
   signe = 1;
   i = 1;
   while i <= n
      res = res + signe * (x^i) / fact;
      signe = signe * (-1);
      i = i + 2;
      fact = fact * (i - 1) * i;
   end
end
```

2.

Voici l'algorithme qui calcule partiellement la série de sinus :

```
function res = sin_serie(x)
   res = x;
   fact = 2 * 3;
   signe = -1;
   i = 3;
   tmp = res + signe * (x^i) / fact;
   while res ~= tmp
      res = tmp;
      i = i + 2;
      fact = fact * (i - 1) * i;
      signe = signe * (-1);
      tmp = res + signe * (x^i) / fact;
   end
end
```

3.
Voici le programme permettant de tracer le graphe :

```
format longE
x = 0:1:2000;
z = sin(-10 + x / 100);
y = 0:1:2000;
for i = 1 : 2001
   y(i) = sin_serie(-10 + (i - 1) / 100);
end
result = (y - z) ./ z;
plot (x, result)
```

En exécutant le programme ci-dessus, on obtient le graphique suivant :

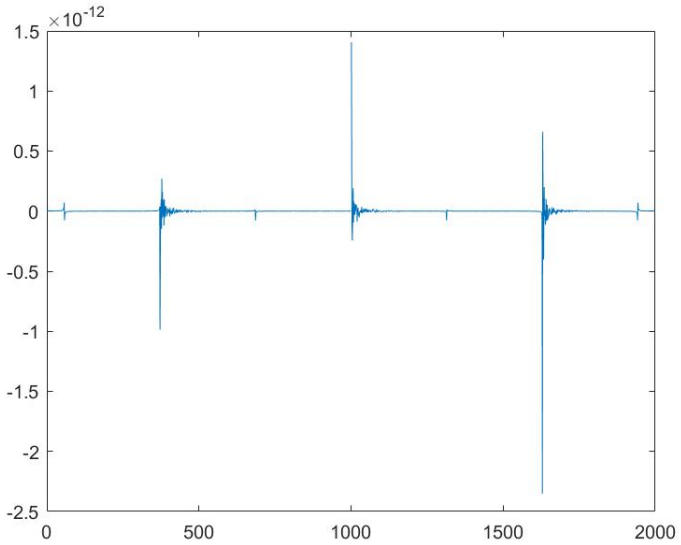

FIGURE 2 – *Le graphe des erreurs relatives de sin_ taylor(-10 + $\frac{k}{100}$) par rapport à sin(-10 + $\frac{k}{100}$)*

Graphiquement, on observe que les erreurs relatives sont de l'ordre de 10^{-12} et que les erreurs relatives sont beaucoup plus importantes sur quelques valeurs particulières.

Exercice 5

1.
Voici l'algorithme qui calcule les s_i :

```
function res = norme1_ligne_oriente_colonne(M)
   res = [];
   i = 1;
   while i <= size(M, 1)
      somme = 0;
      j = 1;
      while j <= size (M, 2)
          somme = somme + abs(M(i, j));
          j = j + 1;
      end
   res = [res somme];
   i = i + 1;
   end
end
```

Voici le programme permettant de comparer l'efficacité :

```
A = rand(10000, 20000);
B = rand(20000, 10000);
tA = transpose(A);
tB = trandpose(B);
tic; norme1_ligne_oriente_colonne(A); toc
tic; vecnorm(tA, 1); toc
tic; norme1_ligne_oriente_colonne(B); toc
tic; vecnorm(tB, 1); toc
```

Dans les deux cas, il faut environ `1.1` secondes pour `norme1_ligne_oriente_colonne` tandis que la commande `vecnorm` ne prend qu'environ `0.08` secondes.

---

2. Voici l'algorithme effectuant le produit matriciel pour deux matrices de taille identique :

```
function res = produit_matriciel(A, B)
   n = size (A, 1);
   res = zeros(n);
   i = 1;
   while i <= n
      j = 1;
      while j <= n
          k = 1;
          somme = 0;
          while k <= n
              somme = somme + A(i, k) * B(k, j);
              k = k + 1;
          end
          res(i, j) = somme;
          j = j + 1;
      end
   i = i + 1;
```

```
    end
end
```
Pour $n = 2000$, il faut environ `31` secondes pour la fonction `produit_matriciel(A, B)` tandis que `A*B` ne prend qu'environ 0.8 secondes.

Fin du TME n°0

I.2 TME n°1 : Introduction à l'optimisation (1/2)

Exercice 11

1.
Voici l'algorithme de la méthode de section dorée :

```
function [res_a, res_b] = sect_doree(f, a, b)
   res_a = a;
   res_b = b;
   tho = (sqrt(5) - 1) / 2;
   x1 = res_a + (1 - tho) * (res_b - res_a);
   x2 = res_a + tho * (res_b - res_a);
   f1 = vpa(subs(f, x1));
   f2 = vpa(subs(f, x2));
   i = 0;
   while res_b - res_a > tho & i < 100
      if f1 > f2
          res_a = x1;
          x1 = x2;
          f1 = f2;
          x2 = res_a + tho * (res_b - res_a);
          f2 = vpa(subs(f, x2));
      else
          res_b = x2;
          x2 = x1;
          f2 = f1;
          x1 = res_a + (1 - tho) * (res_b - res_a);
          f1 = vpa(subs(f, x1));
      end
      i = i + 1;
   end
end
```

2.
Voici l'algorithme de la méthode de Newton :

```
function x1 = newton(f, x0, eps_erreur)
   fp1 = diff(f);
   fp2 = diff(fp1);
   f1 = vpa(subs(fp1, x0));
   f2 = vpa(subs(fp2, x0));
   x1 = x0 - f1 / f2;
   while abs(vpa(subs(f, x1)) - vpa(subs(f, x0))) >= eps_erreur
      x0 = x1;
      f1 = vpa(subs(fp1, x0));
      f2 = vpa(subs(fp2, x0));
      x1 = x0 - f1 / f2;
   end
end
```

3.
On obtient avec les différentes méthodes :
a) `res_a` = 0, `res_b` = 0.6, `Newton(f, 0.1, 10^-4)` $\simeq$ 10.9955

b) `res_a` = `-0.2361`, `res_b` = `0.2361`, `Newton(f, 0.1, 10^-4)` $\simeq$ `0.0`
c) `res_a` = `0.8156`, `res_b` = `1.3262`
d) `res_a` = `-0.2361`, `res_b` = `0.2361`

Fin du TME n°1

I.3 TME n°2 : Introduction à l'optimisation (2/2)

Exercice 12

1.
Soit la fonction f définie par $f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$.
Voici le programme permettant de calculer le gradient $g(x)$ et la Hessienne $H(x)$ de la fonction f :

```
syms x1 x2
f = 100 * (x2 - x1^2)^2 + (1 - x1)^2;
G = gradient(f, [x1, x2])
H = hessian(f, [x1, x2])
```

Voici les résultat obtenus :

```
G =
2*x1 - 400*x1*(- x1^2 + x2) - 2
            - 200*x1^2 + 200*x2

H =
[  1200*x1^2 - 400*x2 + 2, -400*x1]
[                -400*x1,     200]
```

2.
Vérifions que $x^* = [1, 1]^T$ est un minimum local :

```
Gxe = vpa(subs(G, [x1, x2], [1,1]))
Gxe =
0
0

Hxe = vpa(subs(H, [x1, x2], [1,1]))
Hxe =
[  802.0, -400.0]
[ -400.0,  200.0]

[M_vect_p, MD_val_p] = eig(Hxe);
MD_val_p ≃
[  1001.6006,       0]
[          0,  0.3993]
```

Les deux valeurs propres sont strictement positives. x^* est bien un minimum local.

3.
Voici l'algorithme de la méthode de Newton sur plusieurs variables :

```
function res = newton_pv(f, vars, x0)
   G = gradient(f, vars);
   H = hessian(f, vars);
   i = 1;
   while i <= 5
      A = vpa(subs(H, vars, x0));
      B = vpa(subs(G, vars, x0));
      s = A \ (-B);
```

```
        x0 = x0 + s
        i = i + 1;
    end
    res = x0;
end
```

Voici les résultats des 5 premiers itérés en matrice (les colonnes représente les itérés) sans l'utilisation de `vpa` :

```
res = newton_pv(f,[x1; x2], [-1; -2])
-0.9967    0.9956    0.9956    1.0000    1.0000
0.9933    -2.9779    0.9912    1.0000    1.0000
```

Voici sur le même graphe des lignes de niveau de f ainsi que les points itérés (tracés à l'aide la commande `hold on`) :

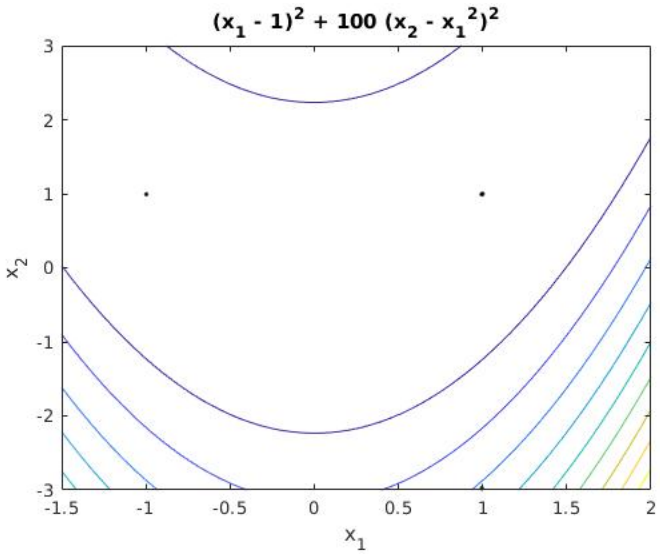

FIGURE 3 – *Le graphe des lignes de niveau de f ainsi que les 5 premières itérés (seulement 3 points peuvent être distingués sur le graphe)*

4.
A l'aide la fonction `norm`, on obtient les valeurs d'erreur suivantes (arrondis) à chaque itération :

```
1.996683305570962
3.977906695091106
0.009836920652712
0.000019421451765
0.000000000000002
```

On remarque qu'à chaque itération, l'itéré gagne le double de chiffres significatifs que son précédent. On en déduit que la converge est quadratique.

Exercice 13

1.
Voici l'algorithme de la méthode du gradient :

```
function x1 = metho_gradient(f, vars, x0, alpha, eps_erreur)
```

```
   G = gradient(f, vars);
   Gx0 = vpa(subs(G, vars, x0));
   x1 = x0 - alpha * Gx0;
   while abs(vpa(subs(f, vars, x1)) - vpa(subs(f, vars, x0))) >= eps_erreur
      x0 = x1;
      Gx0 = vpa(subs(G, vars, x0));
      x1 = x0 - alpha * Gx0;
      end
end
```

Pour le cas de `eps_erreur` $= 10^{-5}$, nous avons réussi à appliquer l'algorithme à $\alpha = 0.1$ et $\alpha = 0.5$ mais pas pour $\alpha = 1$. Il faut augmenter l'erreur pour réussir à appliquer l'algorithme pour $\alpha = 1$. Voici les résultats (arrondis) :

```
syms x1 x2
f = x1^2 + 2 * x2^2;

res = metho_gradient(f, [x1; x2], [-1; -1], 0.1, 10^-5)
-0.003777
-0.000002

res = metho_gradient(f, [x1; x2], [-1; -1], 0.5, 10^-5)
   0
-1.0

res = metho_gradient(f, [x1; x2], [-1; -1], 1, 10^2)
1.0
3.0
```

Au vu de la fonction f, on peut en déduire que le minimal est atteint en (0, 0). De ce fait, plus on augmente la valeur de α plus on s'éloigne de ce minimum car à cause de l'erreur, le point ne peut être atteint par la méthode du gradient sous peine d'avoir une boucle infinie.

2.
En appliquant de même avec la fonction de Rosenbrock f, on obtient :

```
res = metho_gradient(f, [x1; x2], [-1; 1.2], 0.001, 10^-2)
-1.0728052096
1.1595552
```

Le résultat obtenu n'est pas celui qu'on attendait puisque le minimum est atteint en (1, 1) d'après l'exercice précédent. Il y a donc un problème dans la méthode de gradient sans pas optimum.

3.
Voici l'algorithme de la méthode de Wolfe :

```
function [res_tg, res_td] = wolfe(f, vars, x0, t, tg, td, m1, m2)
   res_tg = tg;
   res_td = td;
   G = gradient(f, vars);
   d = - vpa(subs(G, vars, x0));
   g0 = vpa(subs(f, vars, x0));
   gt = vpa(subs(f, vars, x0 + t * d));
   gp0 = transpose(d) * vpa(subs(G, vars, x0));
```

```
   gpt = transpose(d) * vpa(subs(G, vars, x0 + t * d));
   while 1
      if gt <= g0 + m1 * t * gp0 & gpt >= m2 * gp0
          break;
      end
      if gt > g0 + m1 * t * gp0
          td = t;
      end
      if gt <= g0 + m1 * t * gp0 & gpt < m2 * gp0
          tg = t;
      end
      if td == inf
          t = 10 * td;
      else
          t = (td + tg) / 2;
      end
      gt = vpa(subs(f, vars, x0 + t * d));
      gpt = transpose(d) * vpa(subs(G, vars, x0 + t * d));
   end
   res_tg = tg;
   res_td = td;
end
```

Puis, voici l'algorithme de la méthode du gradient à pas optimum :

```
function x1 = metho_opt(f, vars, x0, eps_erreur)
   G = gradient(f, vars);
   Gx0 = vpa(subs(G, vars, x0));
   [res_tg, res_td] = wolfe(f, vars, x0, 1, 0, inf, 0.1, 0.9);
   alpha = (res_tg + res_td) / 2;
   x1 = x0 - alpha * Gx0;
   while abs(vpa(subs(f, vars, x1)) - vpa(subs(f, vars, x0))) >= eps_erreur
      x0 = x1;
      Gx0 = vpa(subs(G, vars, x0));
      [res_tg, res_td] = wolfe(f, vars, x0, 1, 0, inf, 0.1, 0.9);
      alpha = (res_tg + res_td) / 2;
      x1 = x0 - alpha * Gx0;
      end
end
```

En appliquant la méthode du gradient à pas optimum pour la fonction de Rosenbrock f, on obtient : (la durée d'exécution est un peu longue, environ 6 mins)

```
res = metho_opt(f, [x1; x2], [-1; 1.2], 10^-5)
0.902976032464759
0.814399904572787
```

Fin du TME n°2

I.4 TME n°3 : Résolution d'équations non linéaires

Exercice 3

1.
Voici l'algorithme de la méthode de Newton pour les fonction non linéaires :

```
function x1 = newton_non_lineaire(f, x0, eps_erreur)
   fp = diff(f);
   f1 = vpa(subs(f, x0));
   f2 = vpa(subs(fp, x0));
   x1 = x0 - f1 / f2;
   while abs(x1 - x0) >= eps_erreur
      x0 = x1;
      f1 = vpa(subs(f, x0));
      f2 = vpa(subs(fp, x0));
      x1 = x0 - f1 / f2;
   end
end
```

2.
Voici une solution approchée de f définie par $f(x) = x^3$ - $cos(x)$:

```
syms x
f = x^3 - cos(x);

res = newton_non_lineaire(f, 1, 10^-5)
0.865474033101616
```

En observant les valeurs des itérés et en modifiant l'erreur (par exemple : `eps_erreur` = `10^-20`), l'algorithme se termine à seulement 6 itérations et on remarque à chaque itération, `res` gagne le double de chiffres significatifs que son itéré précédent. On en déduit une convergence très rapide, elle est donc quadratique.

Exercice 4

1.
Pour implémenter la méthode de Newton p-adique, nous aurons besoin de d'une fonction qui calcule l'inverse modulaire, voici son algorithme :

```
function res = inverse_mod(a, p)
   u = [1 a];
   v = [0 p];
   while v(2) ~= 0
      q = fix(u(2) / v(2));
      t = u - q * v;
      u = v;
      v = t;
   end
   res = mod(u(1), p);
end
```

Voici à présent l'algorithme de la méthode de Newton p-adique :

```
function xi = newton_p_adique(f, x1, p, k)
   fp = diff(f);
   fp1 = mod(subs(fp, x1), p);
   s = inverse_mod(fp1, p);
   f1 = subs(f, x1);
   xi = mod(x1 - s * f1, p^2)
   i = 2;
   while i < k
      x1 = xi;
      fp1 = mod(subs(fp, x1), p);
      s = inverse_mod(fp1, p^i);
      f1 = subs(f, x1);
      i = i + 1;
      xi = mod(x1 - s * f1, p^i)
   end
   res = xi;
end
```

2.
Voici les résultats obtenus sur les relèvements de $x_1 = 3 \mod 7^k$ pour $k = 2,3,4$ et $f(x) = x^2$ - 2 :

```
xi = newton_p_adique(f, 3, 7, 4)
10
108
2166
```

Exercice 5

1.
Soit A une matrice de taille n inversible.
Soit f la fonction définie par $f(X) = A$ - X^{-1}.
Montrons que l'itération de Newton vérifie $X_{n+1} = 2X_n$ - X_nAX_n.
Soit H une matrice de taille n.

$$\begin{aligned} f(X + H) \text{ - } f(X) &= X^{-1} \text{ - } (X + H)^{-1} \\ &= X^{-1} \text{ - } [(I + HX^{-1})X]^{-1} \\ &= X^{-1} \text{ - } X^{-1}(I + HX^{-1})^{-1} \end{aligned}$$

Pour une norme de H assez petite alors, on a en utilisant les développements limités au voisinage de la matrice 0 :
$(I + HX^{-1})^{-1} = I - HX^{-1}$
On a donc :

$$\begin{aligned} f(X + H) \text{ - } f(X) &= X^{-1} \text{ - } X^{-1}(I - HX^{-1}) \\ &= X^{-1}HX^{-1} \end{aligned}$$

La fonction L définie par $L(H) = X^{-1}HX^{-1}$ est linéaire. f est donc différentiable en tout point et on a par définition :
$J_f(X)H = X^{-1}HX^{-1}$ avec J_f la matrice Jacobienne de f.
L'itération de Newton correspond à $X_{n+1} = X_n + S_n$ telle que S_n est solution de $J_f(X_n)S_n = -f(X_n)$.
En remplaçant H par S_n, on obtient : $X_n^{-1}S_nX_n^{-1} = -A + X_n^{-1} \iff S_n = -X_nAX_n + X_n$.
On obtient bien $X_{n+1} = 2X_n$ - X_nAX_n.

2.
Voici l'algorithme de la méthode de Newton pour f :

```
function R = newton_f(A, eps_erreur)
```

```
   tA = transpose(A);
   R = tA / trace (tA * A);
   I = eye(size(A, 1));
   while norm(I - R * A, 2) >= eps_erreur
      R = 2 * R - R * A * R;
   end
end
```

Fin du TME n°3

Deuxième partie

Travaux sur machines encadrés (suite)

II.5 TME n°4 : Transformée de Fourier Discrète

Exercice 5

1.
Voici l'algorithme récursif permettant de calculer un polynôme par séparation en sa partie paire et sa partie impaire :

```
function res = poly_eval_dp(p, x)
   res = 0;
   n = length(p);
   if n == 1
      res = p(1);
      return
   end
   if real(x) == 1
      for i = 1:n
          res = res + p(i);
      end
      return
   end
   pp = p(1:2:n);
   pi = p(2:2:n);
   res = poly_eval_dp(pp, x^2) + x * poly_eval_dp(pi, x^2);
end
```

Voici l'algorithme récursive de la FFT :

```
function res = fft_recu(p, racine_n)
   res = [];
   n = length(p);
   for j = 0:(n - 1)
      res = [res poly_eval_dp(p, racine_n^j)];
   end
end
```

2.
Voici l'algorithme itératif de la FFT :

```
function r = fft_ite(a, racine_n)
   if racine_n == 1
      r = a;
      return;
   end
   n = length(a);
   if n == 2
      r = [a(1) + a(2) a(1) + racine_n * a(2)];
      return
   end
   ap = a(1:2:n);
   ai = a(2:2:n);
   m = length(ap);
   ap_trie = [];
   ai_trie = [];
   for i = 1:2(m / 2)
      ap_trie = [ap_trie ap(i) ap(i + m / 2)];
```

```
        ai_trie = [ai_trie ap(i) ai(i + m / 2)];
    end
    for k = 2:2:(m / 2)
        ap_trie = [ap_trie ap(k) ap(k + m / 2)];
        ai_trie = [ai_trie ap(k) ai(k + m / 2)];
    end
    fap = [];
    fai = [];
    for l = 0:(m - 1)
        coef_fap = ap_trie;
        coef_fai = ai_trie;
        e = length(coef_fap);
        while e  = 1
            tmpcap = [];
            tmpcai = [];
            e = length(coef_fap);
            for u = 1:2:e
                tmpcap = [tmpcap coef_fap(u) + ((racine_n^2)^(e / 2))^l * coef_fap(u + 1)];
                tmpcai = [tmpcai coef_fai(u) + ((racine_n^2)^(e / 2))^l * coef_fai(u + 1)];
            end
            coef_fap = tmpcap;
            coef_fai = tmpcai;
        end
        fap = [fap coef_fap];
        fai = [fai coef_fai];
    end
    r = zeros(1, n);
    for j = 1:(n / 2)
        r(j) = fap(j) + racine_n^(j - 1) * fai(i);
        r(j + n / 2) = fap(j) - racine_n^(j - 1) * fai(j);
    end
end
```

---

3. et 4.
Voici les graphes des temps d'exécution des différents FFT en fonction de la taille des polynômes (en rouge, le récursif, en bleu, l'itératif et en vert, le fft de MATLAB) :

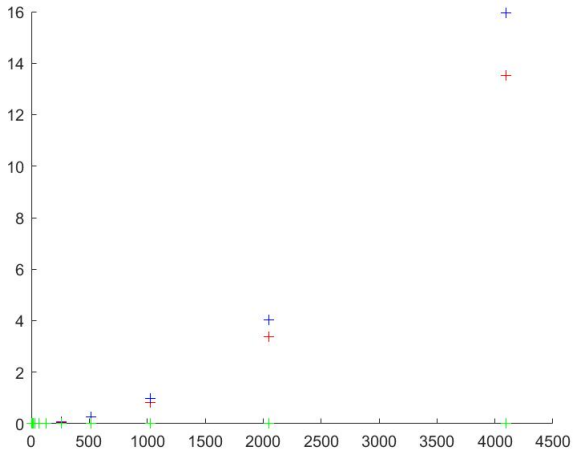

FIGURE 4 – *Graphes expérimentales des temps d'exécution des différents FFT en fonction de la taille des polynômes*

Graphiquement, on en déduit que le FFT récursif est légèrement plus rapide que le FFT itératif sans pour autant

être performant tandis que le FFT de MATLAB est très bien optimisé et beaucoup plus efficace que les autres FFT.

Exercice 6

Voici l'algorithme pour compresser une image par FFT :

```
function ffX = compression_fft(X)
   fX = [];
   n = size(X, 2);
   for j = 1:n
      fX = [fX fft(X(:, j))];
   end
   ffX = [];
   m = size(X, 1);
   for i = 1:m
      ffX = [ffX; fft(fX(i, :))];
   end
end
```

2.
Le taux de compression correspond aux nombre total de valeurs stockées divisée par la taille de la matrice.
Voici l'algorithme calculant le taux :

```
function res = taux(ffX, seuil)
   m = size(ffX, 1);
   n = size(ffX, 2);
   cpt = 0;
   for i = 1:m
      for j = 1:n
          if abs(ffX(i, j)) < seuil | abs(ffX(i, j)) == 0
              cpt = cpt + 1;
          end
      end
   end
   res = ((m * n) - cpt) / (m * n);
end
```

Exercice 7

1.
Voici l'algorithme de multiplication de polynômes avec la méthode naïve :

```
function res = mult_naive(P, Q)
   n = length(P);
   res = zeros(1, 2 * n - 1);
   for i = 1:n
      for j = 1:n
          res(i + j - 1) = res(i + j - 1) + P(i) * Q(j);
      end
   end
```

```
end
```

Voici l'algorithme de multiplication de polynômes en utilisant les FFT :

```
function res = mult_fft(P, Q)
   n = length(P);
   fP = fft(P, 2 * n);
   fQ = fft(Q, 2 * n);
   fPfQ = fP .* fQ;
   res = ifft(fPfQ);
end
```

Voici les courbes expérimentales des temps d'exécution en fonction du degré des polynômes (en bleu, celle de la multiplication naïve et en rouge, celle de la multiplication par FFT) :

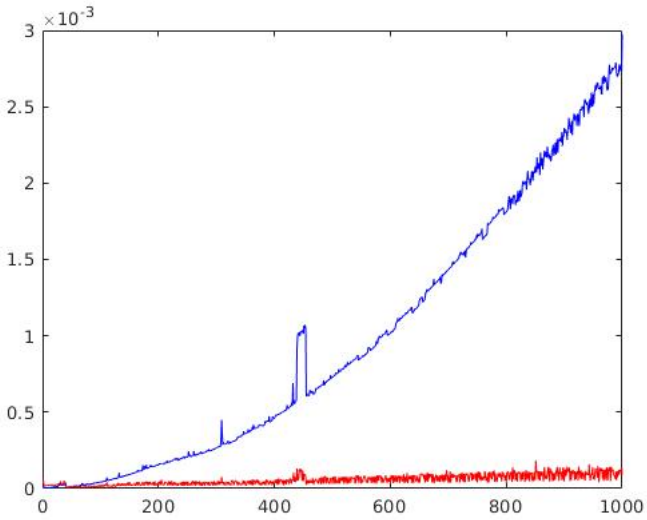

FIGURE 5 – *Courbes expérimentales des temps d'exécution des multiplications de polynômes en fonction de leurs degrés*

Graphiquement, on observe que la multiplication par FFT est beaucoup plus efficace que celle de la multiplication naïve.

Fin du TME n°4

II.6 TME n°5 : Méthodes itératives pour la résolution de systèmes linéaires

Exercice 5

L'algorithme de Jacobi et l'algorithme de Gauss-Seidel sont dans le cours (ou sinon voir nos implémentations). Voici cependant l'algorithme SOR (ligne 6 et 7 du code en une seule ligne de code, tout est à la suite) :

```
function x = SOR(A, y, x0, omega, itmax)
   n = length(y);
   x = x0;
   for it = 1:itmax
      for i = 1:n
          x(i) = omega * (y(i) - A(i, 1:(i - 1)) * x(1:(i - 1)) - A(i, (i + 1):n) * x((i + 1):n))
/ A(i, i) + (1 - omega) * x(i);
      end
   end
end
```

Voici l'algorithme du gradient conjugue :

```
function xk = gradient_conjugue(A, b, x0)
   xk = x0;
   rk = b - A * x0;
   pk = rk;
   for i = 1:length(b)
      alphak = (pk.' * rk) / (pk.' * A * pk);
      xk = xk + alphak * pk;
      rk = rk - alphak * pk;
      betak = - (pk.' * A * rk) / (pk.' * A * pk);
      pk = rk + betak * pk;
   end
end
```

Voici les courbes expérimentales des différents méthodes pour résoudre $Ax + B$ (en rouge, Jacobi, en bleu, Gauss-Seidel, en vert, SOR et en jaune, le gradient conjugué) :

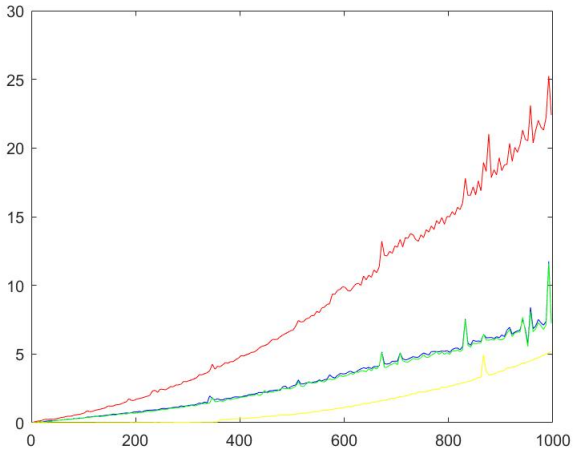

FIGURE 6 – *Courbes expérimentales des temps d'exécution des différentes méthodes de résolution en fonction de la taille des matrices*

Graphiquement, on remarque que la méthode du gradient conjugué est le plus performant de toutes pour des matrices de taille plus petit que 1000. Au delà, cela semble être équivalent, à celui de Gauss-Seidel ou SOR. La méthode de Gauss-Seidel et SOR sont équivalentes tandis que la méthode de Jacobi est le moins performant de toutes pour des matrices de toute taille.

Fin du TME n°5