PARTIE 1:
Nous avons écrit l’ensembles des fonctions écrites grâce aux différentes aides fournis par le cours et
internet.
L’ensemble du main et des tests seront fournies à la fin de cette partie.
Voici l’ensemble des bibliothèques et des define utilisé lors de cette partie 1:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include <assert.h>
#define N 100000
```

Exercice 1:
Voici le premier code écrit et commenté (fonction bitcommun) :

```
// Fonction pour compter les bits communs (méthode non optimisée)
int bitcommun(double x, double y) {
   // Conversion des doubles en unsigned long long (pour manipuler les bits) (et les comparer)
   unsigned long long x_bits = *(unsigned long long*)&x;
   unsigned long long y_bits = *(unsigned long long*)&y;
   // Variable pour stocker le nombre de bits communs
   int common_bits = 0;
   // Masque pour extraire chaque bit, on initialise la variable mask avec la valeur 1
   // sous la forme d’un entier non signé de 64 bits.
   unsigned long long mask = 1ULL;

   // Boucle à travers les 64 bits
   for (int i = 0; i < 64; i++) {
      // Comparaison bit à bit et incrémentation si les bits sont égaux
      // (ici != car les bits sont comparés 1 à 1 après l’autre)
      if ((x_bits & mask) != (y_bits & mask)) {
         common_bits++;
      }
      mask <<= 1; // Décalage du masque pour passer au bit suivant
   }
   return common_bits; // Retourne le nombre de bits communs
}
```

Exercice 2:
Voici le second code écrit et commenté, il s’agit d’une version optimisé (verifié dans l’exercice 3) du
premier code (fonction bitcommmun_opt) :

```
// Fonction pour compter les bits communs (méthode optimisée)
int bitcommun_opt(double x, double y) {
   // Comme pour la première méthode, convertit les double pour les manipuler.
   unsigned long long int x_bits = *((unsigned long long int*)&x);
   unsigned long long int y_bits = *((unsigned long long int*)&y);
   // Réalise une opération XOR (^) entre les valeurs binaires stockées
   // dans x_bits et y_bits et stocke le résultat dans la variable diff
   unsigned long long int diff = x_bits ^ y_bits;
   int count = 0; //initialise le compteur de bits commun.
```

```
    // La boucle while compte simplement le nombre de bits en commun en
poursuivant diff
    while (diff) {
        count += diff & 1;
        diff >>= 1;
    }
    return count; // Retourne le nombre de bits communs
}
```

Exercice 3 :
Nous avons eu un petit problème de compréhension concernant la méthode de résolution de cette
exercice. Nous avions compris qu'il fallait donner le temps en seconde de résolution de chaque
fonction, afin de prouver que l'une est plus optimisée que l'autre. Nous avons pour cela utilisé la
fonction clock().
Voici à présent le main global de la partie :

```
int main() {
    srand(time(NULL)); // Initalisation de la valeur srand à time(NULL)
    double x = 1.41421356;
    double y = 1.41427845;
    double a[N], b[N], un = 1;
    double time_spent = 0.0; // Calcul du temps passé
    clock_t begin = clock();

    for (int i = 0; i < N; i++) {
        a[i] = (un * rand() * 10000) / RAND_MAX;
        b[i] = a[i] + (un * rand() * 0.0001) / RAND_MAX;
        int common_bits = bitcommun_opt(a[i], b[i]);
    }

    clock_t end = clock();

    // calcule le temps écoulé en trouvant la différence (end - begin) et
    // divisant la différence par CLOCKS_PER_SEC pour convertir en secondes
    time_spent += (double)(end - begin) / CLOCKS_PER_SEC;
    printf("Temps écoulé pour la boucle for (fonction simple) est de %f
secondes\n", time_spent);

    double time_spent2 = 0.0;
    clock_t begin2 = clock();

    for (int i = 0; i < N; i++) {
        a[i] = (un * rand() * 10000) / RAND_MAX;
        b[i] = a[i] + (un * rand() * 0.0001) / RAND_MAX;
        int common_bits_opt = bitcommun_opt(a[i], b[i]);
    }

    clock_t end2 = clock();

    // calcule le temps écoulé en trouvant la différence (end - begin) et
    // divisant la différence par CLOCKS_PER_SEC pour convertir en secondes
```

```
    time_spent2 += (double)(end2 - begin2) / CLOCKS_PER_SEC;
    printf("Temps écoulé pour la boucle for (fonction optimisée) est de %f
secondes\n", time_spent2);

    // Appel de la fonction pour compter les bits communs (méthode 1)
    int common_bits = bitcommun(x, y);

    // Appel de la fonction pour compter les bits communs (méthode 2, optimisée)
    int common_bits_opt = bitcommun_opt(x, y);

    // Affichage du résultat
    printf("Nombre de bits en commun entre %f et %f est %d\n", x, y,
common_bits);
    printf("Nombre de bits en commun entre %f et %f est %d (optimisé)\n", x, y,
common_bits_opt);

    //tests des fonctions :
    assert(common_bits == common_bits_opt);
    assert(bitcommun(x, y) == 22);
    assert(bitcommun_opt(x, y) == 22);

    return 0;
}
```

Partie 2
Dans la partie 2,nous avons reussi d'ecrire un programme qui calcule en double précision les termes
de la suite en s’arrêtant dès que la différence entre deux itérés successifs est inférieure à ε.
Nous avons utilisee le schéma de Horner
P (X) = (...(an .x + an−1).x + an−2).x + ...).x + a1).x + a0
Et nous avons faisons lire au clavier l’initialisation de la suite x0 , α et ε.

Exercice 1

```
#include <stdio.h>
#include <math.h>

// Fonction pour évaluer le polynôme en utilisant l'algorithme de Horner
double horner(double *p, int deg, double val) {
    double result = p[0]; // Initialisation du résultat avec le premier
coefficient
    for (int i = 1; i <= deg; i++) {
        result = result * val + p[i]; // Algorithme de Horner
    }
    return result; // Renvoie le résultat final
}

int main() {
    double x0, alpha, epsilon;

    // Demander à l'utilisateur d'entrer les valeurs x0,alpha et epsilon
    printf("Entrez la valeur de x0 : ");
    scanf("%lf", &x0);
    printf("Entrez la valeur de alpha : ");
    scanf("%lf", &alpha);
    printf("Entrez la valeur de epsilon : ");
```

```
    scanf("%lf", &epsilon);

    // Coefficients du polynôme numerateur
    double numerateur[] = {4.0 , -(3.0 * alpha-2.0) , -alpha ,-2.0 * alpha};
    // Coefficients du polynôme denominateur
    double denominateur[] = {5.0 , -(4.0 * alpha - 3.0) , -2.0 * alpha -2.0};

    int deg = 3; // Degré du polynôme

    double x1 = horner(numerateur, deg, x0) / horner(denominateur, deg - 1, x0);
// Calcul de la première itération

    printf("x0 : %.15e\n",x0); // Affichage de x0

    int n = 1;
    while(fabs(x1 - x0) > epsilon){ //si epsilon est inferieur a la valeur
absolue de (x1 - x0),on passe dans le boucle,sinon on s'arrete
        x0 = x1;
        x1 = horner(numerateur, deg, x1) / horner(denominateur, deg - 1, x1); //
Calcul de la prochaine itération xn+1
        printf("x%d = %.15e\n",n++,x0); // Affichage du résultat de l'itération
    }
    printf("x%d = %.15e\n",n++,x1); // Affichage du résultat final

    return 0;
}
```

Et nous avons bien teste la fonction avec α = 0.3, x0 = 2, ε = 0.001,
Apres nous avons saisi les 3 entrees pour initialisation de la suite x0, α et ε d'apres le clavier:

```
Entrez la valeur de x0 : 2
Entrez la valeur de alpha : 0.3
Entrez la valeur de epsilon : 0.001
Il s'affiche bien les resultat:
x0 : 2.000000000000000e+00
x1 = 1.676190476190476e+00
x2 = 1.439696165400490e+00
x3 = 1.273236714165871e+00
x4 = 1.161606844341898e+00
x5 = 1.091029861618567e+00
x6 = 1.049157357720403e+00
x7 = 1.025725214859971e+00
x8 = 1.013191320874718e+00
x9 = 1.006684324452670e+00
x10 = 1.003365238211252e+00
x11 = 1.001688509002766e+00
x12 = 1.000845742570146e+00
```

Le programme s'arrete jusqu'a quand fabs(xn+1 - xn) < epsilon,avec ici xn+1 = x12 , xn = x11 et epsilon = 0.001

Exercice 2 & 3
Dans cet exercice, nous allons d'etudier les 5 cas,et expliquer ses type de convergence
On étudie le rapport (|xn+1 − xs|)/(|xn − xs|) pour différente valeurs de n.

- Si ce rapport augmente, c’est une convergence logarithmique ;
- Si ce rapport reste constant, c’est une convergence linéaire ;
- Si ce rapport diminue, c’est une convergence exponentielle.

Et nous allons retrouver ces résultats en faisant l’étude théorique de la suite.

1. x0 = 2, α = 0.3, ε = 10^(-14):
Type de Convergence : Logarithmique
On lance le programme et les resultat sont:
x0 : 2.000000000000000e+00
x1 = 1.676190476190476e+00
x2 = 1.439696165400490e+00
x3 = 1.273236714165871e+00
x4 = 1.161606844341898e+00
x5 = 1.091029861618567e+00
x6 = 1.049157357720403e+00
x7 = 1.025725214859971e+00
x8 = 1.013191320874718e+00
x9 = 1.006684324452670e+00
x10 = 1.003365238211252e+00
x11 = 1.001688509002766e+00
x12 = 1.000845742570146e+00
x13 = 1.000423245282873e+00
x14 = 1.000211716390457e+00
x15 = 1.000105881663806e+00
x16 = 1.000052946702971e+00
x17 = 1.000026474819744e+00
x18 = 1.000013237776998e+00
x19 = 1.000006618980289e+00
x20 = 1.000003309513093e+00
x21 = 1.000001654762284e+00
x22 = 1.000000827382576e+00
x23 = 1.000000413691647e+00
x24 = 1.000000206845913e+00
x25 = 1.000000103422979e+00
x26 = 1.000000051711495e+00
x27 = 1.000000025855749e+00
x28 = 1.000000012927875e+00
x29 = 1.000000006463938e+00
x30 = 1.000000003231969e+00
x31 = 1.000000001615985e+00
x32 = 1.000000000807992e+00
x33 = 1.000000000403996e+00
x34 = 1.000000000201998e+00
x35 = 1.000000000100999e+00
x36 = 1.000000000050500e+00
x37 = 1.000000000025250e+00
x38 = 1.000000000012625e+00
x39 = 1.000000000006313e+00
x40 = 1.000000000003156e+00
x41 = 1.000000000001578e+00
x42 = 1.000000000000789e+00
x43 = 1.000000000000395e+00
x44 = 1.000000000000198e+00
x45 = 1.000000000000099e+00
x46 = 1.000000000000050e+00
x47 = 1.000000000000025e+00
x48 = 1.000000000000013e+00
x49 = 1.000000000000007e+00
Explication:
Dans le cas, on peut observer que la méthode de Hornerconverge rapidement vers la solution x très proche de 1.
La suite des itérations xn converge vers 1. Après seulement quelques itérations, la valeur de x se rapproche rapidement de 1, et dès la 12ème itération, x est très proche de 1.

2. x0 = 5, α = 3.5, ε = 10^(-14):
Type de Convergence : Linéaire
On lance le programme et les resultat sont:

x0 : 5.000000000000000e+00
x1 = 4.311475409836065e+00
x2 = 3.847043195188532e+00
x3 = 3.592332454460253e+00
x4 = 3.508717383711981e+00
x5 = 3.500087290075656e+00
x6 = 3.500000008865236e+00
x7 = 3.500000000000000e+00
x8 = 3.500000000000001e+00
Explication:
Dans ce cas, on observe que la méthode de Horner converge vers une valeur approximativement égale à 3.5.
La suite des itérations xn se rapproche rapidement de cette valeur,
et à partir de la 6ème itération, la différence entre xn+1 et xn devient très petite.

3. x0 = 0.5, α = 3.5, ε = 10^(-14):
Type de Convergence : Exponentielle
On lance le programme et les resultat sont:
x0 : 5.000000000000000e-01
x1 = 7.830188679245284e-01
x2 = 8.957990913541282e-01
x3 = 9.487454786675618e-01
x4 = 9.745621521751624e-01
x5 = 9.873259468431511e-01
x6 = 9.936738955952065e-01
x7 = 9.968396422598544e-01
x8 = 9.984204902870645e-01
x9 = 9.992104118786879e-01
x10 = 9.996052475540634e-01
x11 = 9.998026341720901e-01
x12 = 9.999013196837326e-01
x13 = 9.999506604911536e-01
x14 = 9.999753304078819e-01
x15 = 9.999876652445151e-01
x16 = 9.999938326324007e-01
x17 = 9.999969163187362e-01
x18 = 9.999984581600021e-01
x19 = 9.999992290801595e-01
x20 = 9.999996145401193e-01
x21 = 9.999998072700697e-01
x22 = 9.999999036350373e-01
x23 = 9.999999518175193e-01
x24 = 9.999999759087598e-01
x25 = 9.999999879543797e-01
x26 = 9.999999939771899e-01
x27 = 9.999999969885949e-01
x28 = 9.999999984942974e-01
x29 = 9.999999992471487e-01
x30 = 9.999999996235743e-01
x31 = 9.999999998117872e-01
x32 = 9.999999999058936e-01
x33 = 9.999999999529467e-01
x34 = 9.999999999764734e-01
x35 = 9.999999999882366e-01
x36 = 9.999999999941184e-01
x37 = 9.999999999970591e-01
x38 = 9.999999999985296e-01
x39 = 9.999999999992648e-01
x40 = 9.999999999996324e-01
x41 = 9.999999999998164e-01
x42 = 9.999999999999083e-01
x43 = 9.999999999999541e-01
x44 = 9.999999999999770e-01

x45 = 9.999999999999886e-01
x46 = 9.999999999999943e-01
Explication:
Dans le cas , on peut observer que la méthode de Horner converge rapidement vers la solution x très proche de 1.
La suite des itérations xn converge vers 1.
Après seulement quelques itérations, la valeur de x se rapproche rapidement de 1,
et dès la 11ème itération, x est très proche de 1.

4. x0 = -3.0, α = 1.0, ε = 10^(-14):
Type de Convergence : Exponentielle
On lance le programme et les resultat sont:
x0 : -3.000000000000000e+00
x1 = -2.636363636363636e+00
x2 = -2.384338433843384e+00
x3 = -2.220139761959379e+00
x4 = -2.120307221576788e+00
x5 = -2.063442351619389e+00
x6 = -2.032676881934435e+00
x7 = -2.016598309217026e+00
x8 = -2.008367090886275e+00
x9 = -2.004200926306228e+00
x10 = -2.002104859707238e+00
x11 = -2.001053535522817e+00
x12 = -2.000527045002281e+00
x13 = -2.000263591914762e+00
x14 = -2.000131813323740e+00
x15 = -2.000065911005081e+00
x16 = -2.000032956588546e+00
x17 = -2.000016478565800e+00
x18 = -2.000008239350785e+00
x19 = -2.000004119692364e+00
x20 = -2.000002059850425e+00
x21 = -2.000001029926273e+00
x22 = -2.000000514963402e+00
x23 = -2.000000257481767e+00
x24 = -2.000000128740900e+00
x25 = -2.000000064370454e+00
x26 = -2.000000032185228e+00
x27 = -2.000000016092614e+00
x28 = -2.000000008046308e+00
x29 = -2.000000004023154e+00
x30 = -2.000000002011577e+00
x31 = -2.000000001005789e+00
x32 = -2.000000000502894e+00
x33 = -2.000000000251447e+00
x34 = -2.000000000125724e+00
x35 = -2.000000000062862e+00
x36 = -2.000000000031431e+00
x37 = -2.000000000015716e+00
x38 = -2.000000000007858e+00
x39 = -2.000000000003929e+00
x40 = -2.000000000001964e+00
x41 = -2.000000000000982e+00
x42 = -2.000000000000491e+00
x43 = -2.000000000000246e+00
x44 = -2.000000000000123e+00
x45 = -2.000000000000062e+00
x46 = -2.000000000000031e+00
x47 = -2.000000000000016e+00
x48 = -2.000000000000008e+00
Explication:
Dans ce cas, on peut observer que la méthode de Horner converge rapidement vers

la solution x est tres proche de -2.
La suite des itérations xn converge vers -2.
Après seulement quelques itérations, la valeur de x se rapproche rapidement de -2, et dès la 6ème itération, x est très proche de -2.

5. x0 = 2.0, α = 1.0, ε = 10^(-14):
Type de Convergence : Instable
Il converge initialement vers 1 , mais il devient instable à partir de l'itération 52.
On lance le programme et les resultat sont:
x0 : 2.000000000000000e+00
x1 = 1.714285714285714e+00
x2 = 1.503246753246753e+00
x3 = 1.350158722022269e+00
x4 = 1.241042377757994e+00
x5 = 1.164490458046522e+00
x6 = 1.111496717833038e+00
x7 = 1.075198285473716e+00
x8 = 1.050534265636377e+00
x9 = 1.033873508602568e+00
x10 = 1.022665762821774e+00
x11 = 1.015148089898183e+00
x12 = 1.010115582129110e+00
x13 = 1.006751258691448e+00
x14 = 1.004504202770433e+00
x15 = 1.003004300898994e+00
x16 = 1.002003534731529e+00
x17 = 1.001335986834971e+00
x18 = 1.000890790003871e+00
x19 = 1.000593918751816e+00
x20 = 1.000395971954741e+00
x21 = 1.000263992914985e+00
x22 = 1.000176000438477e+00
x23 = 1.000117335920214e+00
x24 = 1.000078224966593e+00
x25 = 1.000052150430799e+00
x26 = 1.000034767154168e+00
x27 = 1.000023178193236e+00
x28 = 1.000015452170132e+00
x29 = 1.000010301465206e+00
x30 = 1.000006867649757e+00
x31 = 1.000004578432925e+00
x32 = 1.000003052291465e+00
x33 = 1.000002034871580e+00
x34 = 1.000001356599000e+00
x35 = 1.000000904407917e+00
x36 = 1.000000602926978e+00
x37 = 1.000000401995515e+00
x38 = 1.000000268076683e+00
x39 = 1.000000178726041e+00
x40 = 1.000000119267902e+00
x41 = 1.000000079433799e+00
x42 = 1.000000052800895e+00
x43 = 1.000000034577066e+00
x44 = 1.000000022832820e+00
x45 = 1.000000012966400e+00
x46 = 1.000000011416410e+00
x47 = 1.000000004322133e+00
x48 = 9.999999885835903e-01
x49 = 9.999999935167999e-01
x50 = 9.999999923890602e-01
x51 = 9.999999967583999e-01
x52 = 1.000000000000000e+00
x53 = -nan

Explication:
Dans ce cas , on peut observer que la méthode de Horner converge vers une solution,
mais à partir de l'itération 52, le résultat devient indéfini (not-a-number ou -nan).
La suite des itérations xn converge initialement vers 1.
Cependant, à l'itération 52, le calcul devient instable et le résultat devient indéfini (-nan).
Cela peut se produire lorsque les conditions initiales ou les propriétés du polynôme rendent le calcul numérique de la méthode de Horner instable.
C'est un comportement que l'on observe occasionnellement dans les méthodes numériques.

En résumé:
1. Pour x0 = 2, α = 0.3, ε = 10^(-14), la méthode de Horner converge rapidement vers une solution très proche de 1. La convergence est de type logarithmique.
2. Pour x0 = 5, α = 3.5, ε = 10^(-14), la méthode de Horner converge rapidement vers une solution très proche de 3.5. La convergence est de type linéaire.
3. Pour x0 = 0.5, α = 3.5, ε = 10^(-14), la méthode de Horner converge rapidement vers une solution très proche de 1. La convergence est de type exponentielle.
4. Pour x0 = -3.0, α = 1.0, ε = 10^(-14), la méthode de Horner converge rapidement vers une solution très proche de -2. La convergence est de type exponentielle.
5. Pour x0 = 2.0, α = 1.0, ε = 10^(-14), la méthode de Horner converge initialement vers 1, mais devient instable à partir de l'itération 52, produisant un résultat indéfini (-nan). La convergence est instable.
Ces analyses sont basées sur les observations faites à partir des itérations de la méthode de Horner pour chaque cas.
Chaque type de convergence a été déterminé en évaluant le ratio (|xn+1 − xs|)/(|xn − xs|), où xs est la limite de la séquence.