PARTIE 1:

```
#include<stdio.h>
#include<stdlib.h>
#include <time.h>

double horner(double *a, double x, int n)
{
    double y;
    int i;
    y = a[n];
    for(i = n - 1; i >= 0; i--)
        y = y * x + a[i];
    return y;
}

void eval_Horner_1(double a, double* coeff_f, int n, double* f_a){
    // Vérifier si les coefficients sont valides
    if (coeff_f == NULL) {
        printf("Erreur : pointeur invalide\n");
        return;
    }

    // Vérifier si le degré du polynôme est valide
    if (n < 0) {
        printf("Erreur : degré du polynôme invalide\n");
        return;
    }

    // Utiliser la méthode de Horner pour évaluer le polynôme
    *f_a = horner(coeff_f, a, n);
}

double horner2(double *a, double x, int n)
{
    double yi, yp, x2;
    int i;
    x2 = x * x;
    if (n % 2 != 0){ // ou "if (n/2)"
        yi = a[n]; yp = a[n - 1]; i = n - 2;
    }else{
        yi = 0; yp = a[n]; i = n - 1;
    }

    for(; i >= 0; i -= 2){
        yi = yi * x2 + a[i];
        yp = yp * x2 + a[i-1];
    }
    return yp + x * yi;
}

void eval_Horner_2(double a, double* coeff_f, int n, double* f_a, double*
f_minus_a) {
    // Vérifier si les] coefficients sont valides
```

```
    if (coeff_f == NULL) {
        printf("Erreur : pointeur invalide\n");
        return;
    }

    // Vérifier si le degré du polynôme est valide
    if (n < 0) {
        printf("Erreur : degré du polynôme invalide\n");
        return;
    }

    // Utiliser la méthode de Horner2 pour évaluer le polynôme en a
    *f_a = horner2(coeff_f, a, n);

    // Utiliser la méthode de Horner2 pour évaluer le polynôme en -a
    *f_minus_a = horner2(coeff_f, -a, n);
}

int main() {
    const int max_degree = 16;
    const int repetitions = 100000; // Nombre de répétitions pour chaque mesure

    // Initialiser le générateur de nombres aléatoires
    srand(time(NULL));

    double result_1, result_2, result_minus_2; // Déclarer les variables à
l'extérieur de la boucle

    for (int i = 0; i <= max_degree; i++) {
        int degree = 2 * i;
        double coefficients[degree + 1];

        // Générer des coefficients aléatoires dans l'intervalle [-1, 1]
        for (int j = 0; j <= degree; j++) {
            coefficients[j] = (double)rand() / RAND_MAX * 2.0 - 1.0;
        }

        // Mesurer le temps d'évaluation pour eval_Horner_1
        clock_t start_time_1 = clock();
        for (int rep = 0; rep < repetitions; rep++) {
            eval_Horner_1(2.0, coefficients, degree, &result_1);
        }
        clock_t end_time_1 = clock();

        // Mesurer le temps d'évaluation pour eval_Horner_2
        clock_t start_time_2 = clock();
        for (int rep = 0; rep < repetitions; rep++) {
            eval_Horner_2(2.0, coefficients, degree, &result_2, &result_minus_2);
        }
        clock_t end_time_2 = clock();

        // Calculer les temps d'exécution moyens en secondes
        double time_1 = ((double)(end_time_1 - start_time_1)) / CLOCKS_PER_SEC /
```

```
repetitions;
       double time_2 = ((double)(end_time_2 - start_time_2)) / CLOCKS_PER_SEC /
repetitions;

       printf("Degré %d:\n", degree);
       printf("eval_Horner_1 : %.10lf secondes\n", time_1);
       printf("eval_Horner_2 : %.10lf secondes\n", time_2);

       printf("Résultat(Horner_1): %f\n", result_1);
       printf("Résultat(Horner_2): %f\n", result_2);
       printf("Résultat(Horner_2(minus_a_2): %f\n", result_minus_2);

       printf("\n");
   }

   return 0;
}
```

On a fait les tests et on a trouvé les résultat suivants:

```
Degré 0:
eval_Horner_1 : 0.0000000226 secondes
eval_Horner_2 : 0.0000000225 secondes
Résultat(Horner_1): -0.268114
Résultat(Horner_2): -0.268114
Résultat(Horner_2(minus_a_2): -0.268114

Degré 2:
eval_Horner_1 : 0.0000000096 secondes
eval_Horner_2 : 0.0000000164 secondes
Résultat(Horner_1): 0.331642
Résultat(Horner_2): 0.331642
Résultat(Horner_2(minus_a_2): 3.407597

Degré 4:
eval_Horner_1 : 0.0000000137 secondes
eval_Horner_2 : 0.0000000268 secondes
Résultat(Horner_1): -8.779283
Résultat(Horner_2): -8.779283
Résultat(Horner_2(minus_a_2): 3.138385

Degré 6:
eval_Horner_1 : 0.0000000195 secondes
eval_Horner_2 : 0.0000000276 secondes
Résultat(Horner_1): -17.798754
Résultat(Horner_2): -17.798754
Résultat(Horner_2(minus_a_2): -86.790309

Degré 8:
eval_Horner_1 : 0.0000000252 secondes
eval_Horner_2 : 0.0000000337 secondes
Résultat(Horner_1): 272.544204
Résultat(Horner_2): 272.544204
Résultat(Horner_2(minus_a_2): 68.334216

Degré 10:
eval_Horner_1 : 0.0000000309 secondes
eval_Horner_2 : 0.0000000419 secondes
Résultat(Horner_1): 886.770179
Résultat(Horner_2): 886.770179
Résultat(Horner_2(minus_a_2): -126.963875
```

```
Degré 12:
eval_Horner_1 : 0.0000000351 secondes
eval_Horner_2 : 0.0000000486 secondes
Résultat(Horner_1): 324.644963
Résultat(Horner_2): 324.644963
Résultat(Horner_2(minus_a_2): -271.782374

Degré 14:
eval_Horner_1 : 0.0000000455 secondes
eval_Horner_2 : 0.0000000514 secondes
Résultat(Horner_1): 4141.086539
Résultat(Horner_2): 4141.086539
Résultat(Horner_2(minus_a_2): -5919.312661

Degré 16:
eval_Horner_1 : 0.0000000444 secondes
eval_Horner_2 : 0.0000000596 secondes
Résultat(Horner_1): 10577.378387
Résultat(Horner_2): 10577.378387
Résultat(Horner_2(minus_a_2): 24676.128818

Degré 18:
eval_Horner_1 : 0.0000000502 secondes
eval_Horner_2 : 0.0000000621 secondes
Résultat(Horner_1): 200198.367819
Résultat(Horner_2): 200198.367819
Résultat(Horner_2(minus_a_2): 213061.904262

Degré 20:
eval_Horner_1 : 0.0000000559 secondes
eval_Horner_2 : 0.0000000688 secondes
Résultat(Horner_1): -637844.317591
Résultat(Horner_2): -637844.317591
Résultat(Horner_2(minus_a_2): -447698.703095

Degré 22:
eval_Horner_1 : 0.0000000673 secondes
eval_Horner_2 : 0.0000000743 secondes
Résultat(Horner_1): 439264.303713
Résultat(Horner_2): 439264.303713
Résultat(Horner_2(minus_a_2): 3490530.724774

Degré 24:
eval_Horner_1 : 0.0000000712 secondes
eval_Horner_2 : 0.0000000807 secondes
Résultat(Horner_1): -3431337.212383
Résultat(Horner_2): -3431337.212383
Résultat(Horner_2(minus_a_2): 9858260.846539

Degré 26:
eval_Horner_1 : 0.0000000800 secondes
eval_Horner_2 : 0.0000000863 secondes
Résultat(Horner_1): 14887787.261187
Résultat(Horner_2): 14887787.261187
Résultat(Horner_2(minus_a_2): 44229972.613893

Degré 28:
eval_Horner_1 : 0.0000000843 secondes
eval_Horner_2 : 0.0000000931 secondes
Résultat(Horner_1): 245239972.832049
Résultat(Horner_2): 245239972.832049
Résultat(Horner_2(minus_a_2): 209983461.932048
```

```
Degré 30:
eval_Horner_1 : 0.0000000948 secondes
eval_Horner_2 : 0.0000001000 secondes
Résultat(Horner_1): -1181033521.506705
Résultat(Horner_2): -1181033521.506705
Résultat(Horner_2(minus_a_2): -390928452.713832

Degré 32:
eval_Horner_1 : 0.0000001023 secondes
eval_Horner_2 : 0.0000001098 secondes
Résultat(Horner_1): -3904375733.249982
Résultat(Horner_2): -3904375733.249982
Résultat(Horner_2(minus_a_2): -1038674418.801278

Conclusion: En mesurant le temps d'exécution des deux fonctions sur une seule
exécution, il est difficile de comparer leur efficacité.
Ainsi, si on fait un grand nombre d'appels sur ces deux fonctions, (100000 ici)
on commence à voir une différence et la première fonction de Horner semble être
plus rapide pour effectuer les calculs selon les conditions de l'exercice.

PARTIE 2:

#include <stdio.h>
#include <stdlib.h>

void mulpol(double* coeff_f, int n, double* coeff_g, int p, double* coeff_h,
int* q) {
    // Vérifier si les coefficients sont valides
    if (coeff_f == NULL || coeff_g == NULL || coeff_h == NULL || q == NULL) {
        printf("Erreur : pointeurs invalides\n");
        return;
    }

    // Vérifier si les degrés des polynômes sont valides
    if (n < 0 || p < 0) {
        printf("Erreur : degrés des polynômes invalides\n");
        return;
    }

    // Calculer le degré du polynôme résultant
    *q = n + p;

    // Initialiser le tableau des coefficients du polynôme résultant à zéro
    for (int i = 0; i <= *q; i++) {
        coeff_h[i] = 0.0;
    }

    // Calculer le produit des polynômes
    for (int i = 0; i <= n; i++) {
        for (int j = 0; j <= p; j++) {
            coeff_h[i + j] += coeff_f[i] * coeff_g[j];
        }
    }
}

int main() {
    // Exemple d'utilisation
```

```
    int degre_f = 2;
    double coefficients_f[] = {1.0, 2.0, 3.0}; // f(x) = 1 + 2x + 3x^2

    int degre_g = 1;
    double coefficients_g[] = {4.0, 5.0}; // g(x) = 4 + 5x

    int degre_h; // Le degré du polynôme résultant
    double coefficients_h[degre_f + degre_g + 1]; // Le tableau des coefficients du polynôme résultant

    // Appel de la fonction mulpol
    mulpol(coefficients_f, degre_f, coefficients_g, degre_g, coefficients_h, &degre_h);

    // Affichage du polynôme résultant
    printf("Le produit h(x) : ");
    for (int i = degre_h; i >= 0; i--) {
        printf("%.2f", coefficients_h[i]);
        if (i > 0) {
            printf("x^%d + ", i);
        }
    }
    printf("\n");

    return 0;
}
```

PARTIE 3:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

void gauss(float *a, float *b, int n) {
    int i, j, k, il;
    float aux, aux2;

    for (i = 0; i < n - 1; i++) {
        aux = a[i * n + i];
        for (k = i + 1; k < n; k++)
            a[i * n + k] /= aux;
        b[i] /= aux;

        for (k = i + 1; k < n; k++) {
            aux2 = a[k * n + i];
            for (j = i + 1; j < n; j++)
                a[k * n + j] -= aux2 * a[i * n + j];
            b[k] -= aux2 * b[i];
        }
    }

    // Rétro-substitution
    for (i = n - 1; i >= 0; i--) {
        for (il = i - 1; il >= 0; il--) {
```

```
            b[il] -= b[i] * a[il * n + i];
        }
    }
}

void interpol_linalg(double *a, double *f_a, int n, double *coeff_f) {
    // Vérifier si les pointeurs sont valides
    if (a == NULL || coeff_f == NULL) {
        printf("Erreur : pointeurs invalides\n");
        return;
    }

    // Vérifier si n est valide
    if (n < 0) {
        printf("Erreur : degré invalide\n");
        return;
    }

    // Construire la matrice du système linéaire
    int m = n + 1; // Nombre d'équations
    float *matrix_A = (float *)malloc(m * (n + 1) * sizeof(float));
    for (int i = 0; i < m; i++) {
        double current_a = 1.0;
        for (int j = 0; j <= n; j++) {
            matrix_A[i * (n + 1) + j] = current_a;
            current_a *= a[i];
        }
    }

    // Construire le vecteur colonne B
    float *vector_B = (float *)malloc(m * sizeof(float));
    for (int i = 0; i < m; i++) {
        vector_B[i] = f_a[i];
    }

    // Appliquer la méthode de Gauss pour résoudre le système linéaire
    gauss(matrix_A, vector_B, m);

    // Copier les coefficients résultants dans le tableau coeff_f
    for (int i = 0; i <= n; i++) {
        coeff_f[i] = vector_B[i];
    }

    // Libérer la mémoire allouée
    free(matrix_A);
    free(vector_B);
}

void interpol_Lagrange(double *a, double *f_a, int n, double *coeff_f) {
    // Vérifier si les pointeurs sont valides
    if (a == NULL || coeff_f == NULL) {
        printf("Erreur : pointeurs invalides\n");
```

```
        return;
    }

    // Vérifier si n est valide
    if (n < 0) {
        printf("Erreur : degré invalide\n");
        return;
    }

    // Initialiser les coefficients à zéro
    for (int i = 0; i <= n; i++) {
        coeff_f[i] = 0.0;
    }

    // Calculer les coefficients du polynôme interpolateur de Lagrange
    for (int i = 0; i <= n; i++) {
        double term = f_a[i];

        for (int j = 0; j <= n; j++) {
            if (j != i) {
                term *= 1.0 / (a[i] - a[j]);
            }
        }

        coeff_f[i] = term;
    }
}

int main() {
    const int max_degree = 16;
    srand(time(NULL));

    for (int i = 0; i <= max_degree; i++) {
        int n = 2 * i;

        // Générer des entrées aléatoires dans [−1, 1]
        double a[n + 1];
        double f_a[n + 1];
        for (int j = 0; j <= n; j++) {
            a[j] = (double)rand() / RAND_MAX * 2.0 - 1.0;
            f_a[j] = (double)rand() / RAND_MAX * 2.0 - 1.0;
        }

        // Mesurer le temps d'exécution pour interpol_linalg
        clock_t start_time_linalg = clock();
        double coeff_f_linalg[n + 1];
        interpol_linalg(a, f_a, n, coeff_f_linalg);
        clock_t end_time_linalg = clock();

        // Mesurer le temps d'exécution pour interpol_Lagrange
        clock_t start_time_Lagrange = clock();
        double coeff_f_Lagrange[n + 1];
```

```
        interpol_Lagrange(a, f_a, n, coeff_f_Lagrange);
        clock_t end_time_Lagrange = clock();

        // Afficher les résultats
        printf("Degré %d:\n", n);
        printf("interpol_linalg : ");
        for (int j = 0; j <= n; j++) {
            printf("%.2f ", coeff_f_linalg[j]);
        }
        printf("\n");

        printf("interpol_Lagrange : ");
        for (int j = 0; j <= n; j++) {
            printf("%.2f ", coeff_f_Lagrange[j]);
        }
        printf("\n");

        // Calculer les temps d'exécution en secondes
        double time_linalg = ((double)end_time_linalg - start_time_linalg) /
CLOCKS_PER_SEC;
        double time_Lagrange = ((double)end_time_Lagrange -
start_time_Lagrange) / CLOCKS_PER_SEC;

        printf("Temps d'exécution interpol_linalg : %.10lf secondes\n",
time_linalg);
        printf("Temps d'exécution interpol_Lagrange : %.10lf secondes\n",
time_Lagrange);

        printf("\n");
    }

    return 0;
}
```

En conclusion, les différences observées entre `interpol_linalg` et `interpol_Lagrange` en termes de précision et de rapidité peuvent être expliquées par les méthodes numériques spécifiques utilisées dans chaque fonction.

1. **Précision :**
   - **`interpol_linalg` (Gauss/Moindres carrés) :** La méthode des moindres carrés peut être sensible aux erreurs numériques, en particulier pour des polynômes de degrés élevés. La construction de la matrice du système linéaire et la résolution du système peuvent introduire des erreurs d'approximation.
   - **`interpol_Lagrange` (Méthode de Lagrange) :** En utilisant directement la formule mathématique du polynôme interpolateur de Lagrange, cette méthode évite la résolution d'un système linéaire, réduisant ainsi les erreurs numériques potentielles.

2. **Rapidité :**
   - **`interpol_linalg` :** La méthode des moindres carrés implique la construction d'une matrice et la résolution d'un système linéaire, opérations potentiellement coûteuses en termes de temps de calcul, surtout pour des polynômes de degrés élevés.
   - **`interpol_Lagrange` :** En effectuant des calculs directs sans la nécessité de résoudre un système linéaire, la méthode de Lagrange peut être plus

rapide, nécessitant moins d'itérations.

Bien que la méthode de Lagrange puisse offrir une précision supérieure et une exécution plus rapide dans certains cas, le choix entre les deux dépend des exigences spécifiques du problème, de la nature des données d'entrée, et des compromis entre précision et rapidité. La méthode de Lagrange est souvent préférable dans des contextes où la stabilité numérique est cruciale, tandis que la méthode des moindres carrés peut être utilisée efficacement dans d'autres situations.

PARTIE FINALE:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

double horner(double *a, double x, int n){
    double y;
    int i;
    y = a[n];
    for(i = n - 1; i >= 0; i--)
        y = y * x + a[i];
    return y;
}

void eval_Horner_1(double a, double* coeff_f, int n, double* f_a){
    // Vérifier si les coefficients sont valides
    if (coeff_f == NULL) {
        printf("Erreur : pointeur invalide\n");
        return;
    }

    // Vérifier si le degré du polynôme est valide
    if (n < 0) {
        printf("Erreur : degré du polynôme invalide\n");
        return;
    }

    // Utiliser la méthode de Horner pour évaluer le polynôme
    *f_a = horner(coeff_f, a, n);
}

double horner2(double *a, double x, int n)
{
    double yi, yp, x2;
    int i;
    x2 = x * x;
    if (n % 2 != 0){ // ou "if (n/2)"
        yi = a[n]; yp = a[n - 1]; i = n - 2;
    }else{
        yi = 0; yp = a[n]; i = n - 1;
    }

    for(; i >= 0; i -= 2){
        yi = yi * x2 + a[i];
```

```
        yp = yp * x2 + a[i-1];
    }
    return yp + x * yi;
}

void eval_Horner_2(double a, double* coeff_f, int n, double* f_a, double*
f_minus_a) {
    // Vérifier si les] coefficients sont valides
    if (coeff_f == NULL) {
        printf("Erreur : pointeur invalide\n");
        return;
    }

    // Vérifier si le degré du polynôme est valide
    if (n < 0) {
        printf("Erreur : degré du polynôme invalide\n");
        return;
    }

    // Utiliser la méthode de Horner2 pour évaluer le polynôme en a
    *f_a = horner2(coeff_f, a, n);

    // Utiliser la méthode de Horner2 pour évaluer le polynôme en -a
    *f_minus_a = horner2(coeff_f, -a, n);
}

void mulpol(double* coeff_f, int n, double* coeff_g, int p, double* coeff_h,
int* q) {
    // Vérifier si les coefficients sont valides
    if (coeff_f == NULL || coeff_g == NULL || coeff_h == NULL || q == NULL) {
        printf("Erreur : pointeurs invalides\n");
        return;
    }

    // Vérifier si les degrés des polynômes sont valides
    if (n < 0 || p < 0) {
        printf("Erreur : degrés des polynômes invalides\n");
        return;
    }

    // Calculer le degré du polynôme résultant
    *q = n + p;

    // Initialiser le tableau des coefficients du polynôme résultant à zéro
    for (int i = 0; i <= *q; i++) {
        coeff_h[i] = 0.0;
    }

    // Calculer le produit des polynômes
    for (int i = 0; i <= n; i++) {
        for (int j = 0; j <= p; j++) {
            coeff_h[i + j] += coeff_f[i] * coeff_g[j];
        }
```

```
    }
}

void gauss(double *a, double *b, int n) {
    int i, j, k, il;
    double aux, aux2;

    for (i = 0; i < n - 1; i++) {
        aux = a[i * n + i];
        for (k = i + 1; k < n; k++)
            a[i * n + k] /= aux;
        b[i] /= aux;

        for (k = i + 1; k < n; k++) {
            aux2 = a[k * n + i];
            for (j = i + 1; j < n; j++)
                a[k * n + j] -= aux2 * a[i * n + j];
            b[k] -= aux2 * b[i];
        }
    }

    // Rétro-substitution
    for (i = n - 1; i >= 0; i--) {
        for (il = i - 1; il >= 0; il--) {
            b[il] -= b[i] * a[il * n + i];
        }
    }
}


void interpol_linalg(double *a, double *f_a, int n, double *coeff_f) {
    // Vérifier si les pointeurs sont valides
    if (a == NULL || coeff_f == NULL) {
        printf("Erreur : pointeurs invalides\n");
        return;
    }

    // Vérifier si n est valide
    if (n < 0) {
        printf("Erreur : degré invalide\n");
        return;
    }

    // Construire la matrice du système linéaire
    int m = n + 1; // Nombre d'équations
    double *matrix_A = (double *)malloc(m * (n + 1) * sizeof(double));
    for (int i = 0; i < m; i++) {
        double current_a = 1.0;
        for (int j = 0; j <= n; j++) {
            matrix_A[i * (n + 1) + j] = current_a;
            current_a *= a[i];
        }
```

```
    }

    // Construire le vecteur colonne B
    double *vector_B = (double *)malloc(m * sizeof(double));
    for (int i = 0; i < m; i++) {
        vector_B[i] = f_a[i];
    }

    // Appliquer la méthode de Gauss pour résoudre le système linéaire
    gauss(matrix_A, vector_B, m);

    // Copier les coefficients résultants dans le tableau coeff_f
    for (int i = 0; i <= n; i++) {
        coeff_f[i] = vector_B[i];
    }

    // Libérer la mémoire allouée
    free(matrix_A);
    free(vector_B);
}

void interpol_Lagrange(double *a, double *f_a, int n, double *coeff_f) {
    // Vérifier si les pointeurs sont valides
    if (a == NULL || coeff_f == NULL) {
        printf("Erreur : pointeurs invalides\n");
        return;
    }

    // Vérifier si n est valide
    if (n < 0) {
        printf("Erreur : degré invalide\n");
        return;
    }

    // Initialiser les coefficients à zéro
    for (int i = 0; i <= n; i++) {
        coeff_f[i] = 0.0;
    }

    // Calculer les coefficients du polynôme interpolateur de Lagrange
    for (int i = 0; i <= n; i++) {
        double term = f_a[i];

        for (int j = 0; j <= n; j++) {
            if (j != i) {
                term *= 1.0 / (a[i] - a[j]);
            }
        }

        coeff_f[i] = term;
    }
}
```

```
void multipointeval_Horner_1(double *a, double *coeff_f, int n, double
*resultats, int m) {
    // Vérifier si les pointeurs sont valides
    if (a == NULL || coeff_f == NULL || resultats == NULL) {
        printf("Erreur : pointeurs invalides\n");
        return;
    }

    // Vérifier si n est valide
    if (n < 0) {
        printf("Erreur : degré invalide\n");
        return;
    }

    // Évaluer le polynôme aux points ±x1, …, ±xn en utilisant le schéma de
Horner d'ordre 1
    for (int i = 0; i < m; i++) {
        eval_Horner_1(a[i], coeff_f, n, &resultats[i]);
    }
}

void multipointeval_Horner_2(double *a, double *coeff_f, int n, double
*resultats, int m) {
    // Vérifier si les pointeurs sont valides
    if (a == NULL || coeff_f == NULL || resultats == NULL) {
        printf("Erreur : pointeurs invalides\n");
        return;
    }

    // Vérifier si n est valide
    if (n < 0) {
        printf("Erreur : degré invalide\n");
        return;
    }

    // Évaluer le polynôme aux points ±x1, …, ±xn en utilisant le schéma de
Horner d'ordre 2
    for (int i = 0; i < m; i++) {
        double f_a, f_minus_a;
        eval_Horner_2(a[i], coeff_f, n, &f_a, &f_minus_a);
        resultats[i] = 0.5 * (f_a + f_minus_a);
    }
}

void multipointeval_linalg(double *a, double *coeff_f, int n, double *resultats,
int m) {
    // Vérifier si les pointeurs sont valides
    if (a == NULL || coeff_f == NULL || resultats == NULL) {
        printf("Erreur : pointeurs invalides\n");
        return;
    }

    // Vérifier si n est valide
```

```
    if (n < 0) {
        printf("Erreur : degré invalide\n");
        return;
    }

    // Construire la matrice du système linéaire
    int system_size = n + 1;
    double *matrix_A = (double *)malloc(system_size * system_size *
sizeof(double));
    for (int i = 0; i < system_size; i++) {
        double current_a = 1.0;
        for (int j = 0; j <= n; j++) {
            matrix_A[i * system_size + j] = current_a;
            current_a *= a[i];
        }
    }

    // Résoudre le système linéaire pour obtenir les coefficients du polynôme
    gauss(matrix_A, coeff_f, system_size);

    // Évaluer le polynôme aux points ±x1, …, ±xn en utilisant le produit de
matrice vecteur
    for (int i = 0; i < m; i++) {
        double current_a = 1.0;
        resultats[i] = 0.0;

        for (int j = 0; j <= n; j++) {
            resultats[i] += current_a * coeff_f[j];
            current_a *= a[i];
        }
    }

    // Libérer la mémoire allouée
    free(matrix_A);
}

int main() {
    srand(time(NULL));  // Initialiser le générateur de nombres aléatoires

    for (int i = 0; i <= 16; i++) {
        int n = 2 * i;

        // Générer des entrées aléatoires pour a et coeff_f
        double *a = (double *)malloc((n + 1) * sizeof(double));
        double *coeff_f = (double *)malloc((n + 1) * sizeof(double));

        for (int j = 0; j <= n; j++) {
            a[j] = ((double)rand() / RAND_MAX) * 2.0 - 1.0;
            coeff_f[j] = ((double)rand() / RAND_MAX) * 2.0 - 1.0;
        }

        // Mesurer le temps d'exécution pour multipointeval_Horner_1
        clock_t start = clock();
```

```
        double *resultats1 = (double *)malloc((n + 1) * sizeof(double));
        multipointeval_Horner_1(a, coeff_f, n, resultats1, n + 1);
        clock_t end = clock();
        double time1 = ((double)(end - start)) / CLOCKS_PER_SEC;

        // Mesurer le temps d'exécution pour multipointeval_Horner_2
        start = clock();
        double *resultats2 = (double *)malloc((n + 1) * sizeof(double));
        multipointeval_Horner_2(a, coeff_f, n, resultats2, n + 1);
        end = clock();
        double time2 = ((double)(end - start)) / CLOCKS_PER_SEC;

        // Mesurer le temps d'exécution pour multipointeval_linalg
        start = clock();
        double *resultats3 = (double *)malloc((n + 1) * sizeof(double));
        multipointeval_linalg(a, coeff_f, n, resultats3, n + 1);
        end = clock();
        double time3 = ((double)(end - start)) / CLOCKS_PER_SEC;

        // Afficher les résultats
        printf("n = %d\tmultipointeval_Horner_1: %f s\tmultipointeval_Horner_2:
%f s\tmultipointeval_linalg: %f s\n",n, time1, time2, time3);
        // Libérer la mémoire allouée
        free(a);
        free(coeff_f);
        free(resultats1);
        free(resultats2);
        free(resultats3);
    }

    return 0;
}
```

On a trouvé les résultat suivants:

```
n = 0  multipointeval_Horner_1: 0.000002 s   multipointeval_Horner_2: 0.000001
s  multipointeval_linalg: 0.000001 s
n = 2  multipointeval_Horner_1: 0.000001 s   multipointeval_Horner_2: 0.000001
s  multipointeval_linalg: 0.000002 s
n = 4  multipointeval_Horner_1: 0.000001 s   multipointeval_Horner_2: 0.000001
s  multipointeval_linalg: 0.000002 s
n = 6  multipointeval_Horner_1: 0.000001 s   multipointeval_Horner_2: 0.000002
s  multipointeval_linalg: 0.000002 s
n = 8  multipointeval_Horner_1: 0.000001 s   multipointeval_Horner_2: 0.000001
s  multipointeval_linalg: 0.000006 s
n = 10 multipointeval_Horner_1: 0.000002 s   multipointeval_Horner_2: 0.000002
s  multipointeval_linalg: 0.000005 s
n = 12 multipointeval_Horner_1: 0.000001 s   multipointeval_Horner_2: 0.000002
s  multipointeval_linalg: 0.000006 s
n = 14 multipointeval_Horner_1: 0.000002 s   multipointeval_Horner_2: 0.000003
s  multipointeval_linalg: 0.000013 s
n = 16 multipointeval_Horner_1: 0.000003 s   multipointeval_Horner_2: 0.000003
s  multipointeval_linalg: 0.000010 s
n = 18 multipointeval_Horner_1: 0.000003 s   multipointeval_Horner_2: 0.000003
s  multipointeval_linalg: 0.000014 s
n = 20 multipointeval_Horner_1: 0.000005 s   multipointeval_Horner_2: 0.000003
s  multipointeval_linalg: 0.000020 s
```

```
n = 22 multipointeval_Horner_1: 0.000004 s   multipointeval_Horner_2: 0.000004 s  multipointeval_linalg: 0.000022 s
n = 24 multipointeval_Horner_1: 0.000004 s   multipointeval_Horner_2: 0.000004 s  multipointeval_linalg: 0.000027 s
n = 26 multipointeval_Horner_1: 0.000004 s   multipointeval_Horner_2: 0.000004 s  multipointeval_linalg: 0.000038 s
n = 28 multipointeval_Horner_1: 0.000004 s   multipointeval_Horner_2: 0.000005 s  multipointeval_linalg: 0.000043 s
n = 30 multipointeval_Horner_1: 0.000006 s   multipointeval_Horner_2: 0.000005 s  multipointeval_linalg: 0.000052 s
n = 32 multipointeval_Horner_1: 0.000005 s   multipointeval_Horner_2: 0.000006 s  multipointeval_linalg: 0.000059 s
```

Conclusion :
En analysant les résultats fournis, nous pouvons tirer quelques conclusions :

1. **Complexité temporelle :**

   - Pour `multipointeval_Horner_1` et `multipointeval_Horner_2`, les temps d'exécution restent relativement constants avec une légère augmentation à mesure que `n` augmente. Cela suggère une complexité temporelle linéaire ou presque linéaire.

   - En revanche, pour `multipointeval_linalg`, le temps d'exécution augmente de manière plus significative avec `n`. Cela pourrait indiquer une complexité temporelle légèrement supérieure, en particulier en raison de la résolution d'un système linéaire.

2. **Comparaison des méthodes de Horner :**

   - Les deux méthodes de Horner (`multipointeval_Horner_1` et `multipointeval_Horner_2`) montrent des performances similaires avec des temps d'exécution généralement très bas. Cependant, `multipointeval_Horner_2` utilise une version optimisée de la méthode de Horner, ce qui pourrait expliquer sa légère amélioration par rapport à `multipointeval_Horner_1`.

3. **Méthode de résolution de système linéaire :**

   - La méthode `multipointeval_linalg`, qui résout un système linéaire, montre une augmentation significative des temps d'exécution à mesure que `n` augmente. Résoudre un système linéaire a une complexité temporelle plus élevée que l'évaluation directe par la méthode de Horner.

4. **Optimisations possibles :**

   - Les performances pour `n = 8` semblent être une exception, où `multipointeval_linalg` prend plus de temps que prévu. Cela pourrait être dû à des facteurs aléatoires dans les données d'entrée ou à d'autres considérations spécifiques.

5. **Choix de la méthode en fonction des besoins :**

   - Si la résolution d'un système linéaire n'est pas nécessaire, les méthodes de Horner (`multipointeval_Horner_1` et `multipointeval_Horner_2`) semblent être des choix plus efficaces.

   - Si la résolution d'un système linéaire est nécessaire, `multipointeval_linalg` peut toujours être une option valable pour des tailles de problèmes raisonnables.