Master Calcul Haute Performance et Simulation

Projet de Green computing

Étude de la compression des données sur les performances et la consommation d'énergie pour des problèmes de calcul scientifique

par Jean-Baptiste Gaillot

version 1.0

Table des matières

Organisation du projet

Lien vers le GitHub du projet : `https://github.com/gaillot18/M2-CHPS-Green-computing.git`

Structure du projet

- Le répertoire `Fonctions-communes` contient des fichiers de fonctions qui sont appelées pour chaque problème (affichages, opérations sur des tableaux, lecture et écritures de tableaux dans un fichier, sauvegardes de résultats dans un fichier `csv`, ...).
- Le répertoire `Problemes` contient un sous-répertoire par problème. Pour chaque répertoire problème, il contient un sous-répertoire par version (de base, avec `zfp`, avec `sz` ou d'autres variantes). Pour chaque répertoire version, il y a les répertoires suivants :
  - `Source` contient les codes sources. Il y a généralement le fichier `resolution.c` qui contient les fonctions principales de la résolution du problème et le fichier `main.cpp` qui est le programme principal qui récupère les arguments de l'exécutable, initialise les variables, appelle la fonction principale de résolution et calcule les résultats de performances. Pour les versions de base, il y a le fichier `test-unitaire.cpp` qui fait un test des fonctions de résolutions sur un petit exemple.
  - `Objets` contient les fichiers assembleurs, `Librairies` contient les fichiers de définitions des fonctions appelés depuis `main.cpp`, `Binaires` contient les fichiers exécutables.
  - `Resultats-calcul` contient les solutions du problème.
  - `Resultats-performance` contient les fichiers `csv` de résultats de performances de l'exécution.
  - `Resultats-energie` contient les fichiers `csv` de résultats de consommation d'énergie de l'exécution.
- Le répertoire `Donnees` contient les données téléchargées.
- Le répertoires `Compression-zfp` et `Compression-sz` contiennent ce qui est en rapport avec la compression et décompression (voir la suite).
- Le répertoire `Traitement-resultats` contient ce qui est en rapport avec le traitement des résultats (voir la suite).
- Le répertoire `Recuperation-matrices` contient ce qui est en rapport avec la récupération et l'adaptation des données (voir la suite).

Langages de programmation

- Les programmes de compression, de décompression et de résolution sont écrits en `C`.
- Les programmes principaux, les tests unitaires et les fonctions communes sont écrits en `C` et `C++` (hybride).
- Les scripts d'exécutions sont écrits en `Bash` (shell).
- Les scripts et règles de compilations sont écrits en `make`.
- Les programmes de traitements des résultats sont écrits en `Python`, `LaTeX` et `tikz`.

1 Récupération des données, compression, décompression, mesures d'énergie et traitement des résultats

L'approche générale est la suivante :

- On récupère des matrices depuis `SDRBench`, on les adapte pour les rendre un peu plus compatible avec les problèmes (à diagonale strictement dominante, symétrique, ...). On écrit ces nouvelles matrices dans des fichiers.
- À partir des fichiers de l'étape précédente, on les compresses avec `zfp` et `sz`. On écrit les matrices compressées dans des fichiers.
- On traite les problèmes avec comme entrées les fichiers obtenus aux étapes précédentes, et en sortie les solutions du problème. On mesure des résultats et on les écrit dans des fichiers `csv`.
- On traite les résultats pour créer des graphiques.

1.1 Récupération et adaptations des matrices de `SDRBench`

On se place dans le répertoire `Recuperation-matrices`. On utilise le site `https://sdrbench.github.io` avec l'ensemble de datasets `CESM-ATM` qui comporte 79 matrices de taille $1800 \cdot 3600$. Pour chaque problème, on lit depuis un fichier la matrice correspondante, et on utilise la fonction commune `lire_donnee` pour importer uniquement la sous-partie "haut/gauche" de taille $n \cdot n$ de la matrice d'origine rectangulaire.

Ensuite, on applique un traitement à ces matrices pour certains problèmes.

Problème 1 : Calcul d'un produit matriciel
Aucun traitement n'est appliqué à A et B.

Problème 2 : Résolution d'un système linéaire avec la méthode de Gauss
Résoudre un système linéaire avec la méthode de Gauss est instable sur ces matrices et donne des résultats faux, par conséquent on utilise la fonction commune `construire_matrice_diagonale_strictement_dominante` qui modifie les coefficients diagonaux de A pour la rendre à diagonale strictement dominante.

Problème 3 : Résolution d'un système linéaire avec la méthode de la factorisation de Cholesky
Résoudre un système linéaire avec la méthode de la factorisation de Cholesky est également instable sur ces matrices et donne des résultats faux, par conséquent on utilise les fonctions communes `construire_matrice_symetrique` et qui modifie les coefficients sous-diagonaux et diagonaux de A pour la rendre symétrique et la fonction `construire_matrice_diagonale_strictement_dominante`.

Pour les résolutions de systèmes linéaires, A est moins smooth uniquement sur les blocs diagonaux, ce qui est négligeable lorsque n est suffisamment grand.
On calcule b tel que $A \cdot \begin{pmatrix} 1 & \dots & 1 \end{pmatrix}^T = b$. Pour les calculs des erreurs, on pourra s'appuyer sur le fait que la solution exacte est $x = \begin{pmatrix} 1 & \dots & 1 \end{pmatrix}^T$. On écrit dans un fichier, pour différentes valeurs de n, les matrices et second membres associées à chaque problème dans le répertoire `Resultats`.

1.2 Compression des matrices avec `zfp` et `sz`

On se place dans le répertoire `Compression-zfp` ou `Compression-sz`.

Notations On utilisera les conventions suivantes :

- `A` représente une matrice normale, `A_comp` représente la matrice `A` compressée (de type `void *` pour `zfp` et de type `unsigned char *` pour `sz`),
- `taille_A` représente la taille (en octets) de `A`, `taille_A_comp` représente la taille (en octets) de `A_comp`.

Compression
Signatures des fonctions de compressions :

- Compression d'une matrice :

```
void compresser_matrice
(int n, int mode, double parametre_mode, float *A, void **ptr_A_comp,
int *ptr_taille_A_comp)
```

Procédure qui modifie `ptr_A_comp` (pointeur de `A_comp`) pour y écrire le flux de bits compressé représentant `A` et qui modifie `ptr_taille_A_comp` pour y écrire la taille en octets de `A_comp`. Elle est adaptée si `A` représente une matrice.

— Compression d'un vecteur :

```
void compresser_vecteur
(int n, int mode, double parametre_mode, float *b, void **ptr_b_comp,
int *ptr_taille_b_comp)
```

Procédure qui modifie `ptr_b_comp` (pointeur de `b_comp`) pour y écrire le flux de bits compressé représentant `b` et qui modifie `ptr_taille_b_comp` pour y écrire la taille en octets de `b_comp`. Elle est adaptée si `b` représente un vecteur.

— Compression d'un bloc d'une matrice :

```
void compresser_matrice_bloc
(int n, double rate, float *A_bloc, int i_bloc, int j_bloc, void *A_bloc_comp)
```

Procédure qui modifie `A_bloc_comp` pour y écrire le flux de bits compressé représentant `A_bloc` pour le bloc de coordonnées (`i_bloc`, `j_bloc`).

Décompression

Signatures des fonctions de décompressions :

— Décompression d'une matrice :

```
float *decompresser_matrice
(int n, int mode, double parametre_mode, void *A_comp, int taille_A_comp)
```

Fonction qui retourne `A_decomp` qui représente la décompression du flux de bits compressé `A_comp`. Elle est adaptée si `A_comp` représente une matrice.

— Décompression d'un vecteur :

```
float *decompresser_vecteur
(int n, int mode, double parametre_mode, void *b_comp, int taille)
```

Fonction qui retourne `b_decomp` qui représente la décompression du flux de bits compressé `b_comp`. Elle est adaptée si `b_comp` représente un vecteur.

— Décompression d'un bloc d'une matrice :

```
float *decompresser_matrice_bloc
(int n, double rate, void *A_bloc_comp, int i_bloc, int j_bloc)
```

Procédure qui modifie `A_bloc_comp` pour y écrire le flux de bits décompressé représentant `A_bloc_comp` pour le bloc de coordonnées (`i_bloc`, `j_bloc`).

Commentaires

— `mode` représente le mode de compression (pour `zfp` : `0` pour le mode `rate`, `1` pour le mode `precision` et `2` pour le mode `tolerance` (absolue), pour `sz` : `0` pour le mode `tolerance` (absolue) et `1` pour le mode `tolerance` (relative)).

— `parametre_mode` représente la valeur du paramètre associé au mode de compression (`rate`, `precision` ou `tolerance`).

— Pour les compressions et décompressions par blocs, les fonctions ne fonctionnent uniquement que pour le mode `rate` car c'est le seul mode qui permet d'accéder précisément à un bloc par un simple calcul.

— Pour les compressions et décompressions par blocs, les fonctions ne fonctionnent uniquement que pour `zfp` car `sz` ne propose pas de mode `rate`.

— Les signatures des fonctions sont pratiquement identiques (à certains types de variables près) entre `zfp` et `sz`.
— Pour `zfp` et `sz`, il y a un fichier `test-unitaire.cpp` qui teste chaque fonction sur un exemple.

Pour chaque fichier obtenu avec la sous-section 1.1, on génère ses versions compressées avec `zfp` et `sz`, pour un certains nombres de valeurs de `n` et de `parametre_mode`.

1.3 Exécutions des tests

1.3.1 Programmes principaux

L'approche générale d'une exécution de la résolution d'un problème est la suivante :

— On récupère les paramètres du problème et les fichiers d'entrées avec les arguments de l'exécutable (`argv`).
— On lit depuis un fichier d'entrée le(s) fichier(s) de données.
— On appelle la fonction principale de résolution du problème.
— On écrit dans un fichier de sortie la solution du problème.
— Si la mesure d'énergie est activée, on arrête le programme. Sinon, on calcule les variables de performances, on les écrit dans un fichier `csv` et on arrête le programme.

Commentaire Si la mesure d'énergie est activée, on arrête le programme sans calculer les variables de performances pour éviter de mesurer la consommation du calcul de ces variables (même si elle est négligeable).

Les variables de performances sont les suivantes :

— `taille_totale` : taille totale sur le disque dur des entrées et sorties du problème (si on est sur un mode compressé, on la calcule quand même pour pouvoir calculer le facteur de compression), on ne prend pas en compte la taille en RAM.
— `taille_compressee` : taille totale sur le disque dur des entrées et sorties du problème (on la calcule uniquement si on est sur un mode compressé).
— `erreur_L2_relative` : erreur en norme 2 (Frobenius) relative entre la solution calculée et la solution exacte.
— `erreur_infty` : erreur en norme ∞ entre la solution calculée et la solution exacte (on ne l'utilisera pas pour les résultats, elle sert surtout pour le debuggage et les tests).
— `temps_execution` : temps d'exécution (incluant la lecture des entrées depuis un fichier, le calcul et la solution et l'écriture de la solution dans un fichier, excluant le calcul des résultats de performances).

On rajoute au fichier `csv` les variables d'entrées (paramètres) suivantes :

— `n` : taille du problème
— `rate` ou `tolerance` : paramètre de compression (on l'utilise uniquement si on est sur un mode compressé).

1.3.2 Scripts et Makefile

Pour chaque problème, les exécutions sont appelées par un script, pour des valeurs de `n`, de `rate` ou `tolerance` différentes. Les scripts sont les suivants :

— `Script-performance.sh` : exécute les programmes pour toutes les versions et tout les paramètres testés, en mesurant uniquement les variables de performances.
— `Script-energie.sh` : exécute les programmes pour toutes les versions et tout les paramètres testés, en mesurant uniquement la consommation d'énergie.

Commentaires

— Pour chaque version et paramètres identiques, on exécute le programme plusieurs fois d'affilée. On utilise la moyenne pour avoir un résultat unique.
— Les deux types de scripts compilent et lient chaque version du programme avant de lancer les exécutions.
— On est obligé d'exécuter les deux scripts l'un après l'autre pour ne pas fausser les résultats.
— Les fichiers `csv` sont écrits dans les répertoires `Resultats-performance` et `Resultats-energie`, relatifs au problème. Pour les deux types de fichiers, il y a un fichier par version.
— Il y a un script global à la racine du projet qui appelle tout les scripts de tout les problèmes.

1.3.3 Mesures d'énergie

Dans un script de type énergie, pour chaque groupe d'exécutions (c'est-à-dire une version et paramètres identiques qu'on exécute plusieurs fois d'affilée), on mesure la consommation d'énergie totale en J avec `powerjoular`, qu'on divise ensuite par le nombre d'exécutions pour avoir la consommation d'énergie en J par exécution. On ne s'intéresse qu'à cette valeur, car mesurer la puissance instantanée est trop complexe à traiter et inutile (on en cherche pas à voir les variations de puissance au cours d'une même exécution). Par facilité, on mesure la consommation d'énergie totale du système (mais on verra qu'on ajustera légèrement son résultat pour refléter au mieux la consommation d'énergie du programme). Pour éviter de mesurer la consommation d'énergie du calcul des résultats de performances (même si elle est négligeable), on active une macro `MESURE_ENERGIE` (depuis le script) qui fait faire le `return 0` du `main` et les libérations de la mémoire avant ces calculs.

1.3.4 Traitement des résultats

Après les exécutions des scripts, on obtient, pour chaque version de chaque problème, deux types fichiers de résultats `csv`.

Exemple d'un fichier de type `Resultats-performance` pour les paramètres $n \in \{768, 896\}$ et `rate = 8` avec 5 exécutions :

```
n,   rate, taille_totale, taille_compressee, erreur_L2_relative, temps_execution
768, 8,    2365440,       595968,           0.0028706871,       0.509508
768, 8,    2365440,       595968,           0.0028706871,       0.510247
768, 8,    2365440,       595968,           0.0028706871,       0.510663
768, 8,    2365440,       595968,           0.0028706871,       0.510448
768, 8,    2365440,       595968,           0.0028706871,       0.508611
896, 8,    3218432,       809984,           0.0039201236,       0.795485
896, 8,    3218432,       809984,           0.0039201236,       0.801256
896, 8,    3218432,       809984,           0.0039201236,       0.801953
896, 8,    3218432,       809984,           0.0039201236,       0.794647
896, 8,    3218432,       809984,           0.0039201236,       0.801283
```

Exemple d'un fichier de type `Resultats-energie` pour les paramètres $n \in \{768, 896\}$ et `rate = 8` avec 5 exécutions :

```
n,   rate, consommation_energie
768, 8,    8.240
896, 8,    13.915
```

Commentaire Pour un fichier de type `Resultats-energie`, la moyenne des exécutions est déjà faite directement dans le script.

On se place dans le répertoire `Traitement-resultats`.

Pour ces deux fichiers obtenus, on génère un nouveau fichier avec le programme `main-traitement.py`.

Signature d'une fonction (avec `Python` et `pandas`) qui prend en entrée les deux types de fichier générés et :

- Fait la moyenne des résultats pour le fichier de type `Resultats-performance` pour n'avoir qu'une seule ligne par groupe d'exécution et écraser le nouveau résultat par celui-ci.
- Concatène le résultat précédent avec le fichier de type `Resultats-energie`, sans dupliquer les colonnes communes.
- Ajoute une colonne qui calcule la puissance moyenne (égale à la consommation d'énergie par le temps d'exécution).

```
def traiter_csv
(fichier_entree_performances : str, fichier_entree_energie : str, compresseur : int,
mode : int, fichier_sortie : str)
```

Commentaires

— Pour refléter la consommation réelle du programme, on soustrait à chaque valeur de `consommation_energie` la puissance au repos de la machine (qui a été mesurée de manière bien stable) multipliée par le temps d'exécution.

— Calculer la moyenne n'a du sens que pour le temps d'exécution. Pour les variables de types erreurs ou tailles, la moyenne est égale aux valeurs car les exécutions sont déterministes.

Exemple d'un fichier de résultat (traité) avec les entrées des exemples précédents :

```
n,   rate, taille_totale, taille_compressee, erreur_L2_relative, temps_execution, consommation_energie puissance_moyenne
768, 8,    2365440,       595968,           0.002870,           0,509895,        8.240000             16,160190
896, 8,    3218432,       809984,           0.003920,           0,798925,        13.915000            17,417154
```

1.3.5 Visualisation des résultats

Pour chaque fichier obtenu avec la sous-sous-section 1.3.4, on génère des graphiques `tikz` pour les deux axes (correspondants à deux paramètres) voulus avec le programme `main-visualisation.py`.

Signature d'une fonction (avec `Python` et `numpy`) qui prend en entrée un fichier obtenu à la fin de la sous-sous-section 1.3.4 et qui génère la partie de graphique `tikz` (en `LaTeX`) prête à être copié / collée dans ce rapport :

```
def csv_vers_tikz
(fichier_entree_resultats : str, colonne_x : int, colonne_y : int,
fichier_sortie_tikz) -> str:
```

2 Problèmes

On se place dans le répertoire `Problemes`.

2.1 Calcul d'un produit matriciel

On se place dans le sous-répertoire `Produit-matriciel`.

Notations

- Entrées du problème : A et B sont les matrices à multiplier.
- Sortie du problème : C est la solution telle que : $C = AB$.
- La taille du problème est : taille(A) + taille(B) + taille(C). Pour les versions compressés, on utilisera les tailles des entrées et sorties compressées.

2.1.1 Version de base

On lit depuis un fichier les données crées en sous-section 1.1, on calcule $C = AB$ et on écrit la solution obtenue (qu'on considérera comme étant la solution exacte, à quelques erreurs d'arrondis du aux opérations flottantes près) dans un fichier. L'écriture de la solution obtenue servira également à être utilisée comme entrée pour les versions avec compression, afin de comparer la solution exacte avec les solutions obtenus avec les versions compressées).

Fonction d'une version de base :

```
float *calculer_produit_matrices(float *A, float *B, int n){

    float *C;

    C = (float *)malloc(n * n * sizeof(float));

    for (int i = 0 ; i < n ; i ++){
        for (int j = 0 ; j < n ; j ++){
            float accumulateur = 0.0;
            for (int k = 0 ; k < n ; k ++){
                accumulateur += A[IDX(i, k, n)] * B[IDX(k, j, n)];
            }
            C[IDX(i, j, n)] = accumulateur;
        }
    }

    return C;

}
```

Commentaire Pour cette version, la taille du problème est de $3n^2 \cdot$ taille(flottant) octets.

2.1.2 Versions avec `zfp` et `sz`

On lit depuis un fichier les données (compressées) crées en sous-section 1.2, on calcule $C_{\text{comp}} = A_{\text{comp}} B_{\text{comp}}$.

Fonction d'une version avec `zfp` :

```
void calculer_produit_matrices
(void *A_comp, void *B_comp, int n, double rate, int taille_A_comp,
int taille_B_comp, void **ptr_C_comp, int *ptr_taille_C_comp){

    float *A;
    float *B;
    float *C;
    void *C_comp;
    int taille_C_comp;

    C = (float *)malloc(n * n * sizeof(float));

    A = decompresser_matrice(n, 0, rate, A_comp, taille_A_comp);
    B = decompresser_matrice(n, 0, rate, B_comp, taille_B_comp);

    for (int i = 0 ; i < n ; i ++){
        for (int j = 0 ; j < n ; j ++){
            float accumulateur = 0.0;
            for (int k = 0 ; k < n ; k ++){
                accumulateur += A[IDX(i, k, n)] * B[IDX(k, j, n)];
            }
            C[IDX(i, j, n)] = accumulateur;
        }
    }

    compresser_matrice(n, 0, rate, C, &C_comp, &taille_C_comp);

    *ptr_C_comp = C_comp;
    *ptr_taille_C_comp = taille_C_comp;

    free(A);
    free(B);
    free(C);

}
```

Commentaires

- Pour la version avec `zfp`, on utilise le mode `rate` et pour la version avec `sz`, on utilise le mode `tolerance` (absolue).
- On décompresse `A_comp` (resp. `B_comp`) dans `A` (resp. `B`), on effectue les calculs sur les matrices décompressées et on compresse la solution `C` dans `C_comp`.
- Le calcul de l'erreur se fait entre `C_exact` (où `C_exact` est obtenu précédemment avec la version de base) et `C_comp`.
- On a besoin de modifier `ptr_taille_C_comp` pour l'écriture de la solution dans un fichier.
- Pour la version avec `sz`, la fonction `calculer_produit_matrices` est identique (à types de valeurs près).

2.1.3 Version avec `zfp` locale

On lit depuis un fichier les données (compressées) crées en sous-section 1.2, on calcule $C_{\text{comp}} = A_{\text{comp}} B_{\text{comp}}$. L'approche est de décompresser seulement les blocs qu'on a besoin. On calcule C par blocs. Soit $C_{i,j}$ (resp. $A_{i,j}$ et $B_{i,j}$) le (i,j)-ème bloc $4 \cdot 4$ de C (resp. A et B). Alors : $C_{i,j} = \sum_{k=1}^{n/4} A_{i,k} B_{k,j}$. Pour un bloc fixé, à chaque itération de la somme sur k, on décompresse les blocs de A et B correspondants et on calcule le résultat partiel. Lorsque le bloc solution est calculé, on le compresse.

Fonction d'une version avec `zfp` locale :

```c
void *calculer_produit_matrices(void *A_comp, void *B_comp, int n, double rate){

    int nb_blocs;
    void *C_comp;
    int taille_C_comp;

    nb_blocs = n / 4;
    taille_C_comp = (nb_blocs * nb_blocs * 16 * rate + 7) / 8;
    C_comp = malloc(taille_C_comp);

    for (int i_bloc = 0 ; i_bloc < nb_blocs ; i_bloc ++){
        for (int j_bloc = 0 ; j_bloc < nb_blocs ; j_bloc ++){

            float *C_bloc;
            C_bloc = (float *)calloc(4 * 4, sizeof(float));

            for (int k_bloc = 0 ; k_bloc < nb_blocs ; k_bloc ++){

                float *A_bloc;
                float *B_bloc;
                float *C_bloc_accumulateur;
                A_bloc = decompresser_matrice_bloc(n, rate, A_comp, i_bloc, k_bloc);
                B_bloc = decompresser_matrice_bloc(n, rate, B_comp, k_bloc, j_bloc);
                C_bloc_accumulateur
                = calculer_produit_matrices_bloc(A_bloc, B_bloc, 4);
                free(A_bloc);
                free(B_bloc);

                for (int i = 0 ; i < 4 * 4 ; i ++){
                    C_bloc[i] += C_bloc_accumulateur[i];
                }

                free(C_bloc_accumulateur);
            }

            compresser_matrice_bloc(n, rate, C_bloc, i_bloc, j_bloc, C_comp);
            free(C_bloc);
        }
    }

    return C_comp;

}
```

Commentaires

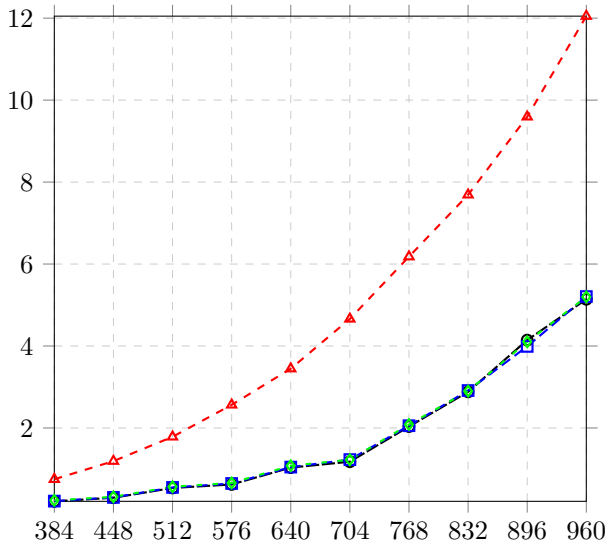
- Un avantage de cette méthode est qu'on effectue aucune décompression totale, donc il y a une faible consommation de la RAM. Il n'y a besoin de stocker que A_{comp}, B_{comp}, les 3 buffers temporaires (de taille $4 \cdot 4 \cdot$ taille(flottant)) et C_{comp}. Cette méthode ne permet pas de réduire la taille du problème.
- Cette méthode n'est faisable qu'avec le mode `rate` car c'est le seul mode qui permet d'accéder à un bloc par une formule de calcul.

2.1.4 Comparaison des performances des méthodes

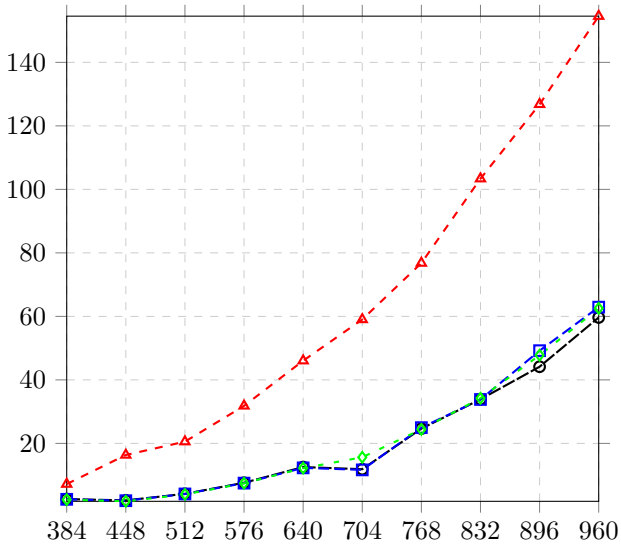
On obtient les résultats suivants (moyennes pour 5 exécutions) avec les paramètres suivants :

- A est représentée par `AEROD_v_1_1800_3600.f32` et B est représentée par `CLDHGH_1_1800_3600.f32`,
- $n \in \{384, 448, 512, 640, 704, 768, 832, 896, 960\}$,
- versions `zfp` et `zfp` locale : `rate = 3`,
- version avec `sz` : `tolerance = 1e-3` (absolue).

Temps d'exécution (s) en fonction de n

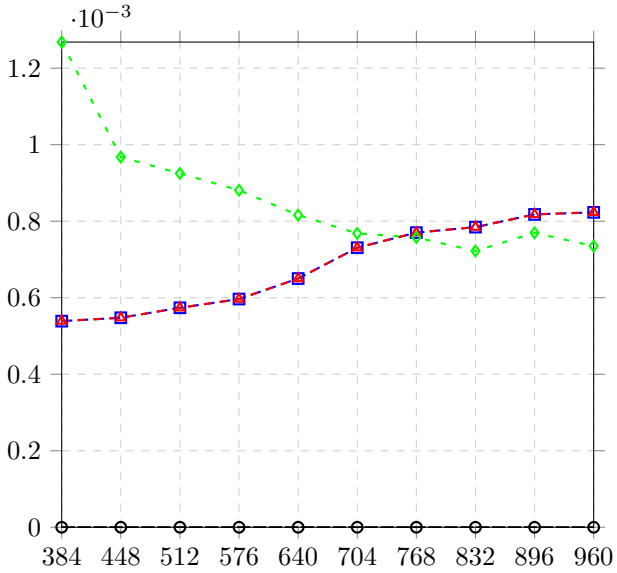


Consommation d'énergie (J) en fonction de n

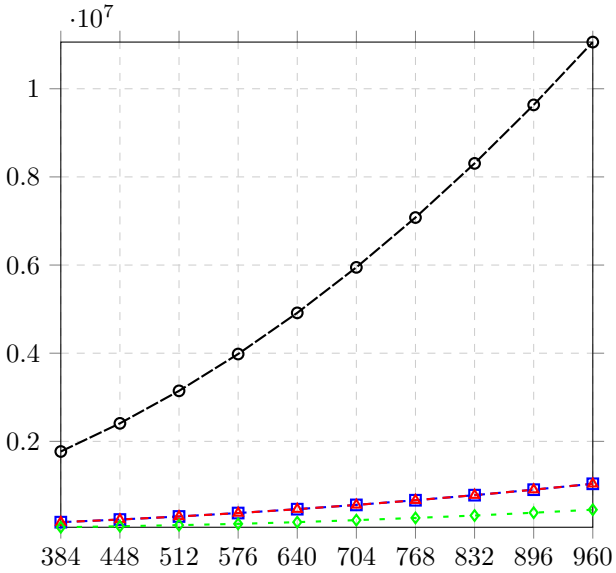


Version de base
Version avec `zfp`
Version avec `zfp` locale
Version avec `sz`

Erreur en fonction de n



Taille (octets) en fonction de n



Version de base
Version avec `zfp`
Version avec `zfp` locale
Version avec `sz`

Commentaires

— Pour $n = 960$, les facteurs de compressions sont les suivants : 10.7 pour les versions avec `zfp` et `zfp` locale et 24.7 pour la version avec `sz`.

— La version avec `zfp` locale est la moins performante car la résolution est constamment ralentie par les appels des fonctions de compression et décompression locales.

— Les versions avec `zfp` et `sz` n'augmentent pas significativement le temps d'exécution et la consommation d'énergie.

— Les versions avec `zfp` et `sz` diminuent fortement la taille.

— Les versions avec `zfp` et `sz` donnent une erreur par rapport à la version de base.

— Il n'y a pas de différences d'erreurs entre les versions avec `zfp` et `zfp` locale, ce qui est attendu.

— La meilleure version est la version avec `sz` car elle donne les mêmes temps d'exécutions et consommations d'énergie que la version de base, et l'erreur est meilleure que pour les versions avec `zfp` (si n est suffisamment grand), et le facteur de compression est meilleur que la version avec `zfp`.

— Pour les versions avec `zfp`, d'autres valeurs pour `rate` ont été testées : si `rate = 2`, l'erreur est de l'ordre de 10^{-3}, et si `rate = 1`, l'erreur est de l'ordre de 10^{-1}.

— Pour les versions avec `zfp`, le mode `tolerance` (absolue) a été testé donne de meilleures erreurs. On s'intéressera plus en détails à ce mode dans les problèmes suivants. Le but ici était d'utiliser le mode `rate` car c'est le seul mode qui permet de faire une version locale.

2.2 Résolution d'un système linéaire

On se place dans le sous-répertoire `Resolution-Gauss` ou `Resolution-Cholesky` (les approches pour traiter les deux problèmes sont identiques).

Notations

— Entrées du problème : A est la matrice représentant le système linéaire et b est le vecteur du second membre représentant le système linéaire.

— Sortie du problème : x est la solution telle que : $Ax = b$.

— b est construit de sorte que $x = \big(1 \dots 1\big)^T$.

— La taille du problème est : taille(A) + taille(b) + taille(x). Pour les versions compressés, on utilisera les tailles des entrées et sorties compressées.

2.2.1 Version de base

On lit depuis un fichier les données crées en sous-section 1.1, on calcule x tel que $Ax = b$ et on écrit la solution obtenue dans un fichier.

Fonction d'une version de base (méthode de Gauss) :

```
float *resoudre_gauss(float *A, float *b, int n){

    float *A_copie;
    float *b_copie;
    float *x;

    A_copie = (float *)malloc(n * n * sizeof(float));
    b_copie = (float *)malloc(n * sizeof(float));
    memcpy(A_copie, A, n * n * sizeof(float));
    memcpy(b_copie, b, n * sizeof(float));

    for (int d = 0 ; d < n - 1 ; d ++){
        for (int i = d + 1 ; i < n ; i ++){
            float pivot = A[IDX(i, d, n)] / A[IDX(d, d, n)];
            for (int j = d ; j < n ; j ++){
                A[IDX(i, j, n)] -= pivot * A[IDX(d, j, n)];
            }
            b[i] -= pivot * b[d];
        }
    }

    x = resoudre_systeme_triangulaire_superieur(A, b, n);
```

```
    memcpy(A, A_copie, n * n * sizeof(float));
    memcpy(b, b_copie, n * sizeof(float));
    free(A_copie);
    free(b_copie);

    return x;

}
```

Fonction d'une version de base (méthode de la factorisation de Cholesky) :

```
float *resoudre_cholesky(float *A, float *b, int n){

    float *L;
    float *L_T;
    float *y;
    float *x;

    L = (float *)calloc(n * n, sizeof(float));

    for (int j = 0 ; j < n ; j ++){

        float accumulateur = A[IDX(j, j, n)];
        for (int k = 0 ; k < j ; k ++){
            accumulateur -= pow(L[IDX(j, k, n)], 2);
        }
        L[IDX(j, j, n)] = sqrt(accumulateur);

        for (int i = j + 1 ; i < n ; i ++){
            double accumulateur = A[IDX(i, j, n)];
            for (int k = 0 ; k < j ; k ++){
                accumulateur -= L[IDX(i, k, n)] * L[IDX(j, k, n)];
            }
            L[IDX(i, j, n)] = accumulateur / L[IDX(j, j, n)];
        }

    }
    L_T = transposer_matrice_triangulaire_inferieure(L, n);

    y = resoudre_systeme_triangulaire_inferieur(L, b, n);
    x = resoudre_systeme_triangulaire_superieur(L_T, y, n);

    free(L);
    free(L_T);
    free(y);

    return x;

}
```

Signature d'une fonction pour résoudre un système triangulaire supérieur :

```
float *resoudre_systeme_triangulaire_superieur(float *A, float *b, int n)
```

Signature d'une fonction pour résoudre un système triangulaire inférieur :

```
float *resoudre_systeme_triangulaire_inferieur(float *A, float *b, int n)
```

Signature d'une fonction pour transposer une matrice triangulaire inférieure :

```
float *transposer_matrice_triangulaire_inferieure(float *A, int n)
```

Commentaire Pour cette version, la taille du problème est de $\left(n^2 + 2n\right) \cdot$ taille(flottant) octets.

2.2.2 Versions avec `zfp` et `sz`

On lit depuis un fichier les données (compressées) crées en sous-section 1.2, on calcule x tel que $A_{\text{comp}}x = b$.

Fonction d'une version avec **`zfp`** (méthode de Gauss) :

```
float *resoudre_gauss
(void *A_comp, float *b, int n, double tolerance, int taille_A_comp){

    float *A;
    float *b_copie;
    float *x;

    b_copie = (float *)malloc(n * sizeof(float));
    memcpy(b_copie, b, n * sizeof(float));

    A = decompresser_matrice(n, 2, tolerance, A_comp, taille_A_comp);

    for (int d = 0 ; d < n - 1 ; d ++){
        for (int i = d + 1 ; i < n ; i ++){
            float pivot = A[IDX(i, d, n)] / A[IDX(d, d, n)];
            for (int j = d ; j < n ; j ++){
                A[IDX(i, j, n)] -= pivot * A[IDX(d, j, n)];
            }
            b[i] -= pivot * b[d];
        }
    }

    x = resoudre_systeme_triangulaire_superieur(A, b, n);

    memcpy(b, b_copie, n * sizeof(float));

    free(A);
    free(b_copie);

    return x;

}
```

Signature d'une fonction d'une version avec **`zfp`** (méthode de la factorisation de Cholesky) :

```
float *resoudre_cholesky
(void *A_comp, float *b, int n, double tolerance, int taille_A_comp)
```

Commentaires

- Pour cette version, on compresse seulement A.
- Pour la version avec **`sz`**, les fonctions **`resoudre_gauss`** et **`resoudre_cholesky`** sont identiques (à types de valeurs près).

2.2.3 Version avec `zfp` avec compression des vecteurs

On lit depuis un fichier les données (compressées) crées en sous-section 1.2, on calcule x_{comp} tel que $A_{\text{comp}}x_{\text{comp}} = b_{\text{comp}}$. L'implémentation est la même que pour la version avec **`zfp`**, à la différence de la décompression de b et la compression de x.

Signature d'une fonction d'une version avec **`zfp`** avec compression des vecteurs (méthode de Gauss) :

```
void resoudre_gauss
(void *A_comp, void *b_comp, int n, double tolerance, int taille_A_comp,
int taille_b_comp, void **ptr_x_comp, int *ptr_taille_x_comp)
```

Signature d'une fonction d'une version avec `zfp` avec compression des vecteurs (méthode de la factorisation de Cholesky) :

```
void resoudre_cholesky
(void *A_comp, void *b_comp, int n, double tolerance, int taille_A_comp,
int taille_b_comp, void **ptr_x_comp, int *ptr_taille_x_comp)
```

Commentaire Pour cette version, A, b et x sont compressés. Le calcul de l'erreur se fait entre x_{comp} et $\big(1 \dots 1\big)^T$.

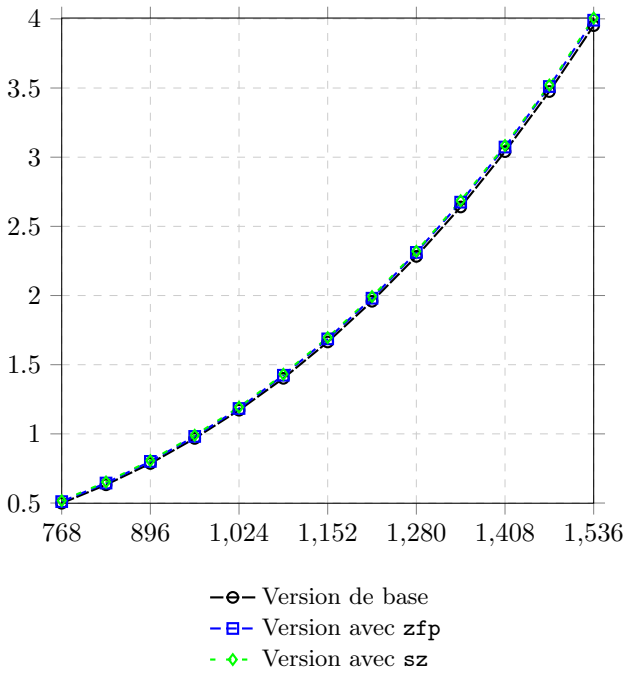
2.2.4 Comparaison des performances des méthodes

Résultats généraux pour la méthode de Gauss

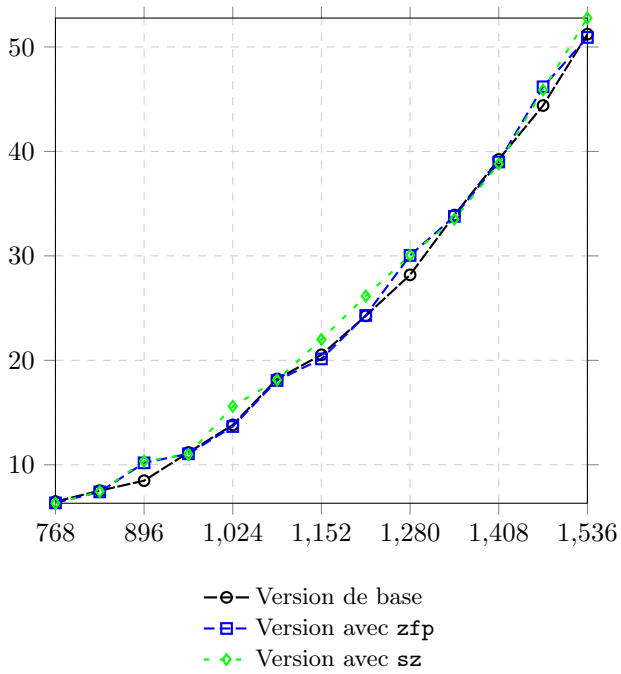
On obtient les résultats suivants (moyennes pour 10 exécutions) avec les paramètres suivants :

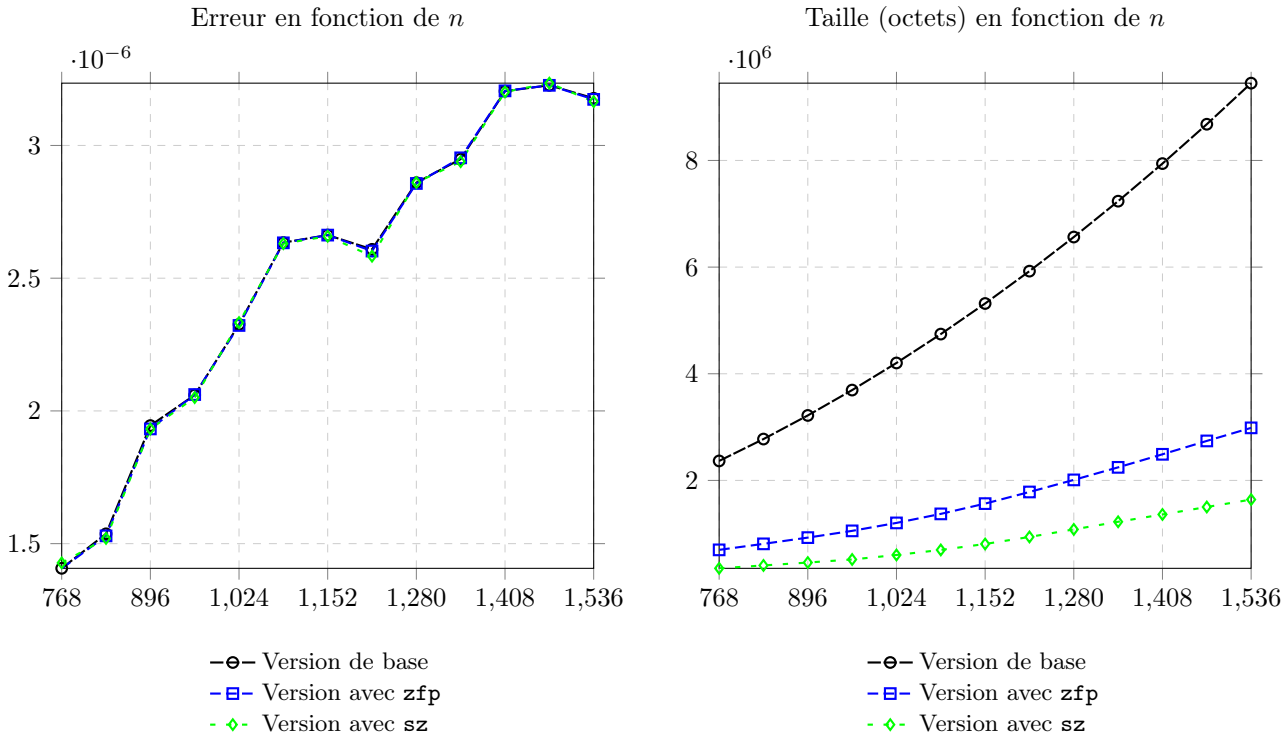
- A est représentée par `TSMX_1_1800_3600.f32` (traitée),
- $n \in \{768, 832, 896, 960, 1024, 1088, 1152, 1216, 1280, 1344, 1408, 1472, 1536\}$,
- versions `zfp` : `tolerance = 1e-3` (absolue),
- version avec `sz` : `tolerance = 1e-3` (absolue).

Temps d'exécution (s) en fonction de n



Consommation d'énergie (J) en fonction de n





Commentaires

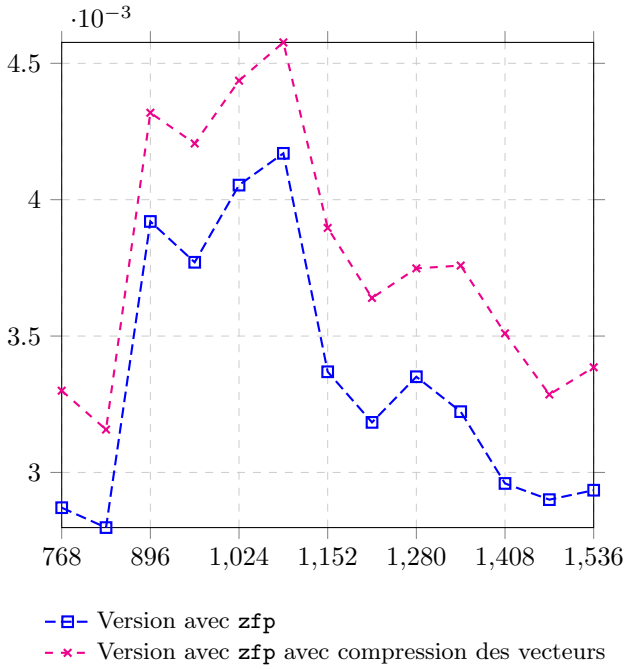
- Pour $n = 1536$, les facteurs de compressions sont les suivants : 3.2 pour la version avec `zfp` et 5.8 pour la version avec `sz`. C'est difficile de faire mieux car ce problème est beaucoup plus sensible aux erreurs que le problème 1. Si la valeur de `tolerance` est moins élevée, alors l'erreur est trop élevée.
- Les versions avec `zfp` et `sz` n'augmentent pas significativement le temps d'exécution et la consommation d'énergie.
- Les versions avec `zfp` et `sz` donnent une erreur presque identique par rapport aux versions de base.
- La meilleure version est la version avec `sz` car elle donne les même temps d'exécutions, consommations d'énergie et erreurs que la version de base, avec la meilleure taille. La version avec `zfp` reste aussi très bonne.

Résultats de la version avec `zfp` avec compression des vecteurs pour la méthode de Gauss

On cherche analyser les erreurs entre la version avec `zfp` et `zfp` avec compression des vecteurs en testant 2 datasets. On obtient les résultats suivants avec les paramètres suivants :

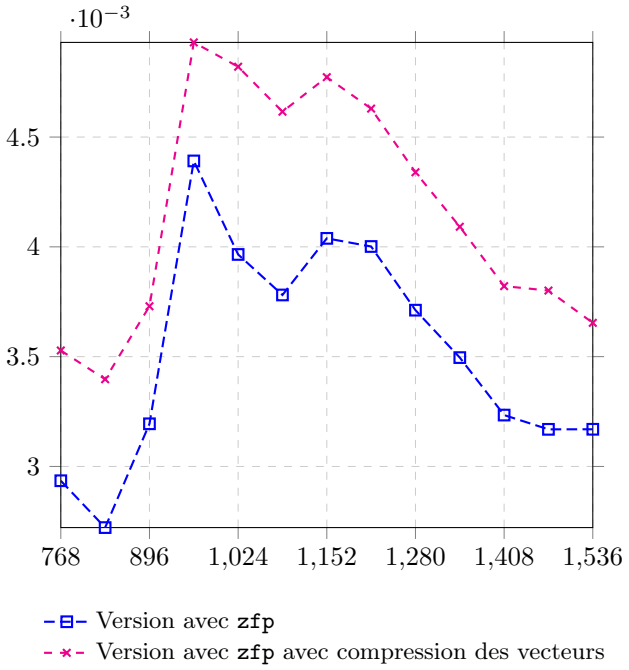
- A est représentée par `TSMX_1_1800_3600.f32` (traitée) et par `TSMN_1_1800_3600.f32` (traitée),
- $n \in \{768, 832, 896, 960, 1024, 1088, 1152, 1216, 1280, 1344, 1408, 1472, 1536\}$,
- versions `zfp` et `zfp` locale : `rate = 8` (pour mieux voir les différences),

Erreur en fonction de n
pour A représentée par `TSMX_1_1800_3600.f32`



Version avec `zfp`
Version avec `zfp` avec compression des vecteurs

Erreur en fonction de n
pour A représentée par `TSMN_1_1800_3600.f32`



Version avec `zfp`
Version avec `zfp` avec compression des vecteurs

Commentaires

— Pour les 2 datasets et pour n fixé, la différence d'erreurs entre les versions avec `zfp` et `zfp` avec compression des vecteurs semble presque constante. Il pourrait être intéressant de tester si ce phénomène est présent pour les 77 autres datasets.

— La version avec `zfp` avec compression des vecteurs est inutile car elle ajoute une erreur et la taille n'est presque pas diminuée (car la taille de 2 vecteurs est négligeable par rapport à la taille d'une matrice).

Résultats généraux pour la méthode de la factorisation de Cholesky

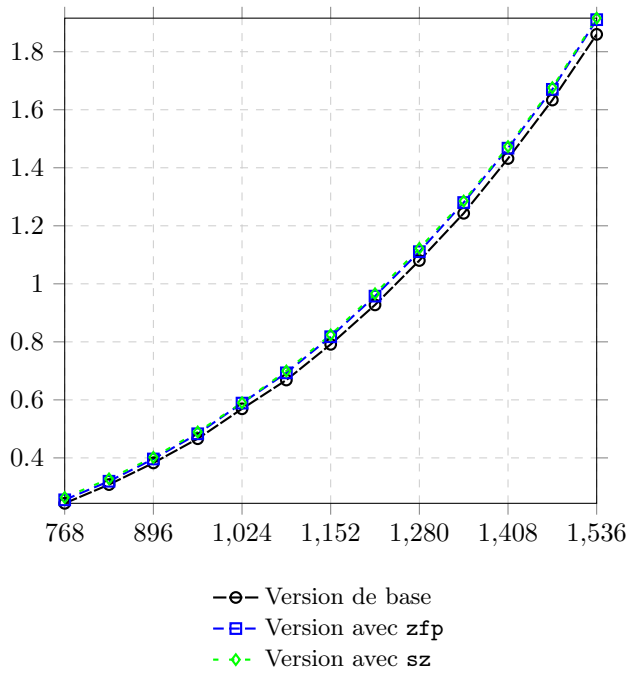
On obtient les résultats suivants (moyennes pour 20 exécutions) avec les paramètres suivants :

— $n \in \{768, 832, 896, 960, 1024, 1088, 1152, 1216, 1280, 1344, 1408, 1472, 1536\}$,

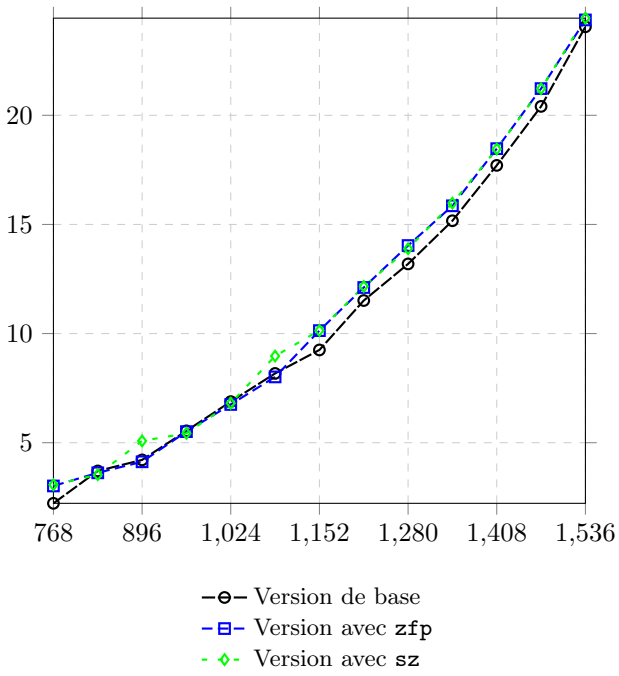
— version avec `zfp` : `tolerance = 1e-3` (absolue),

— version avec `sz` : `tolerance = 1e-3` (absolue).

Temps d'exécution (s) en fonction de n

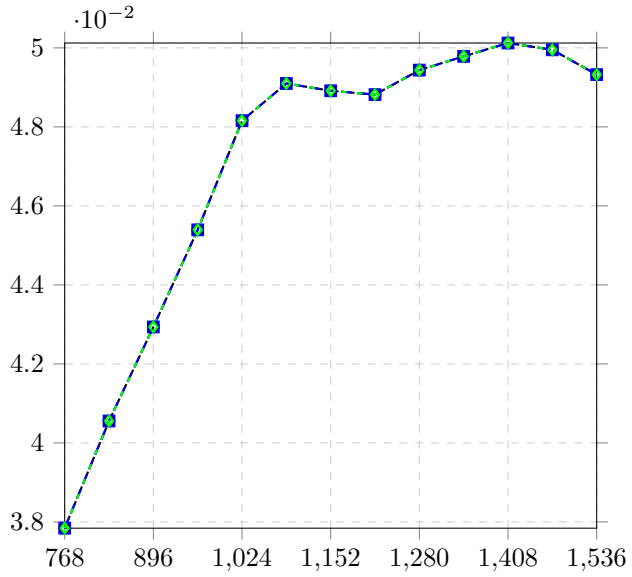


Consommation d'énergie (J) en fonction de n

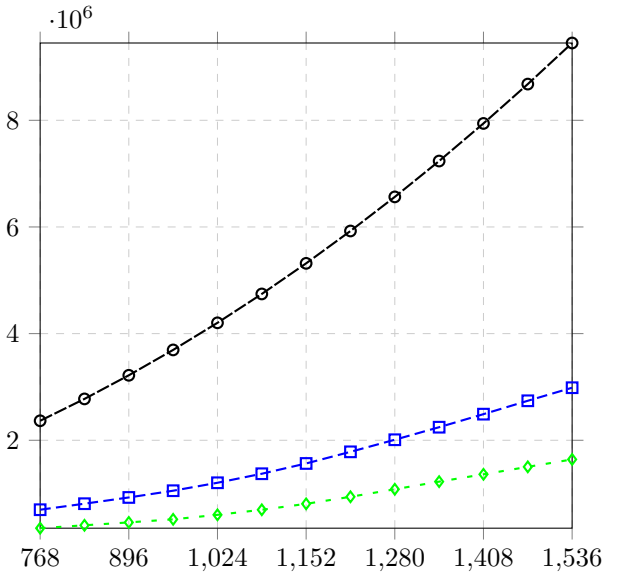


Version de base
Version avec `zfp`
Version avec `sz`

Erreur en fonction de n



Taille (octets) en fonction de n



Version de base
Version avec `zfp`
Version avec `sz`

Commentaires

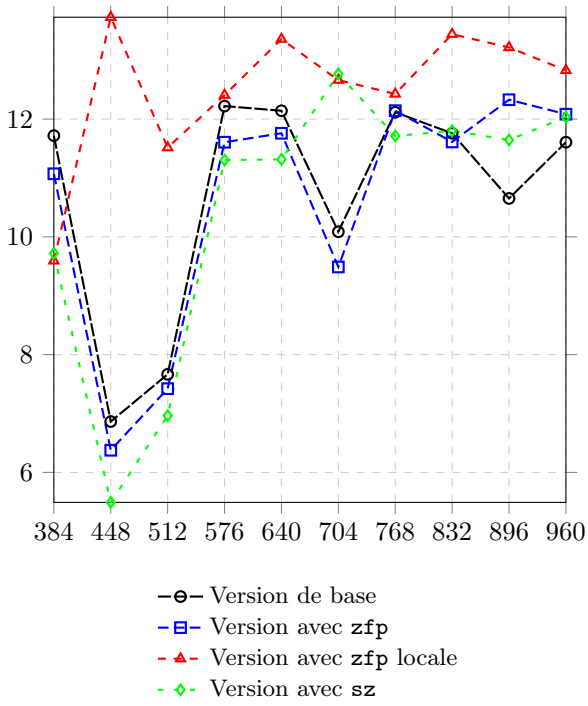
— Les commentaires sont les mêmes que pour le problème 2.

— Pour chaque version, l'erreur est un peu élevée (de l'ordre de 10^{-2}) car malgré le traitement initial des matrices, la méthode de la factorisation de Cholesky est peu stable même pour des matrices non compressés, l'important est de voir la différence d'erreurs entre la version de base et les versions avec `zfp` et `sz`.

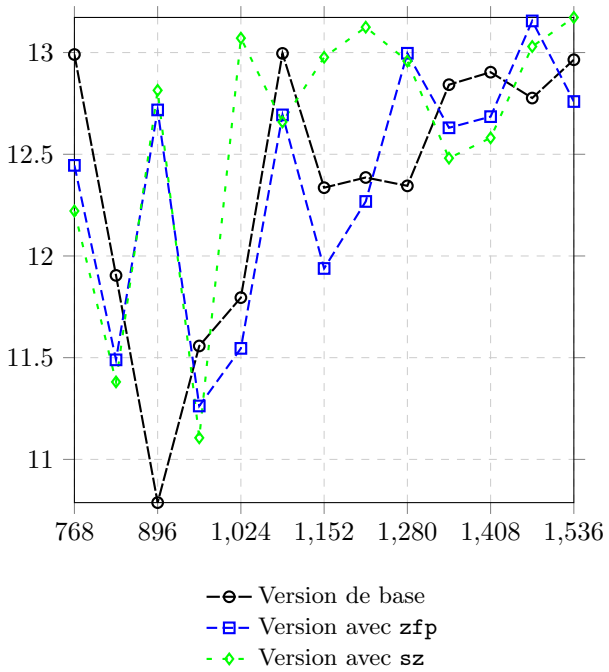
2.3 Autres commentaires et conclusion sur les performances

— Pour chaque problème, la puissance moyenne est bornée (entre environ 6 W et 13 W pour le problème 1 et entre environ 9 W et 14 W pour les problèmes 2 et 3).

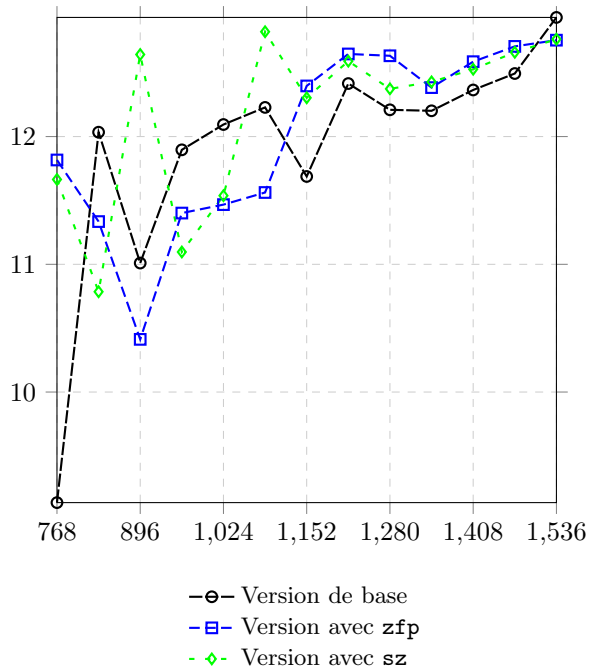
Puissance moyenne (W) en fonction de n
pour le problème 1



Puissance moyenne (W) en fonction de n
pour le problème 2



Puissance moyenne (W) en fonction de n
pour le problème 3



— Pour chaque problème, l'erreur est très sensible aux entrées et au choix du paramètre de compression.

— Pour la résolution des systèmes linéaires, pour le mode `tolerance` (absolue), à `tolerance` égale, la version avec `sz` donne un meilleur facteur de compression que la version avec `zfp`.

— Pour la version avec `zfp`, à facteur de compression égal, le mode `rate` donne une erreur plus élevée que le mode `tolerance` (absolue).

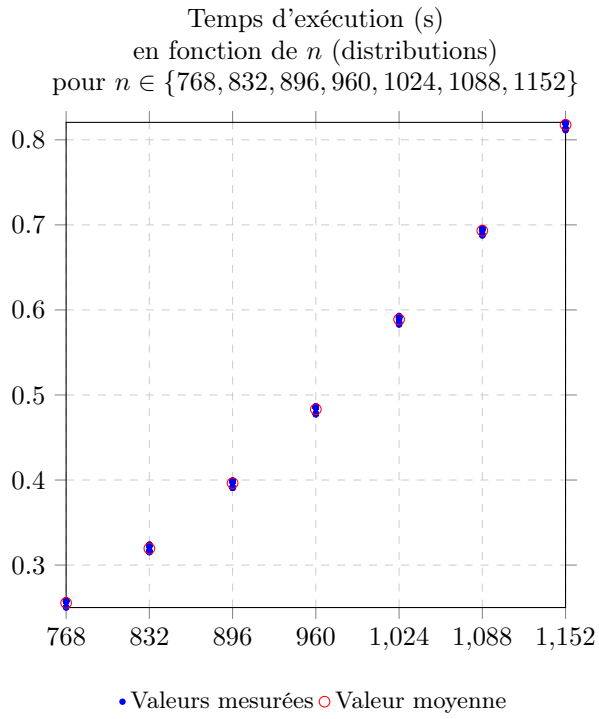
— Pour chaque problème, la décompression des entrées et la compression des sorties n'augmente pas significativement le temps d'exécution et la consommation d'énergie.

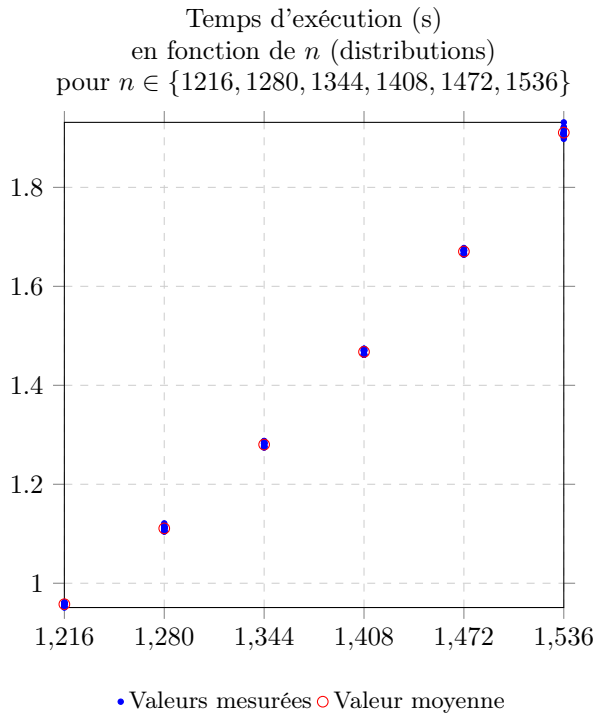
— Pour la version avec `zfp`, une implémentation locale est difficile à mettre en place et augmente les temps d'exécutions et consommations d'énergies. Une version `zfp` locale a été tentée pour le problème 2 mais a donné des résultats non exploitables.

— Ce rapport donne un aperçu d'un certains nombre de tests effectués, mais on pourrait mesurer encore une infinité de choses.

— Le temps d'exécution pour faire tout les tests de ce rapport (script principal) est d'environ 1h20 (du aux plusieurs exécutions à entrées et paramètres identiques, voir le fichier `Demonstrations/execution-complete.txt` qui a enregistré la sortie standard de tout les tests).

— Pour chaque problème, à entrées et paramètres identiques, faire plusieurs exécutions n'a pas été utile pour mesurer les temps d'exécutions, car ils sont pratiquement toujours identique. Par conséquent, l'écart-type est proche de 0. Exemple de la distribution des temps d'exécutions pour problème 3 avec la version `zfp` pour 20 exécutions :





Faire plusieurs exécutions a été plus utile pour mesurer les consommations d'énergies car il est difficile de mesurer des valeurs sur des temps d'exécutions faibles.

— Les tests ont été faites sur une machine ancienne (CPU Intel Core i5 3570, 8G octets de RAM et système Linux). Des exécutions ont été faites sur une machine récente (CPU ARM Apple M3, 16G octets de RAM, système conteneurisé Linux et système hôte macOS) et donnent des temps d'exécutions largement meilleurs mais proportionnels par rapport à la machine ancienne, mais il a été impossible de mesurer la consommation d'énergie sur cette machine du à l'architecture (l'accès aux compteurs matériel n'est pas autorisé sur les Mac).