UCLouvain

LEPL1503
Projet 3
Rapport

Opérations sur les matrices

Élèves :
Lina El Ouarzazi
Lisa Dony
Aurélien Tilot
Diego Seisdedos Stoz
Eric Van Eetvelde
Thibault Vyt

Enseignants :
Benoit Legat
Charles Van Hees
Matthieu Pigaglio
Olivier Bonaventure

12 mai 2025

Table des matières

1 Algorithmes

1.1 Implémentations

Nous avons implémenté les algorithmes demandés dans l'énoncé de cette production, c'est à dire : les opérations de bases telles que l'addition (`add_v_v`, `add_m_m`), la soustraction (`sub_v_v`, `sub_m_m`), la norme d'un vecteur (`norm`) et le produit scalaire de vecteurs (`dot_prod`). Ensuite, il y a des fonctions plus compliquées, la subsitution arrière pour une matrice triangulaire supérieure (`back_sub`), la décomposition QR avec Gram-Schmidt (`qr`) et la régression polynomiale au sens des moindres carrés pour la résolution d'un système linéaire surdéterminé (`lstsq`). Nous avons aussi implémenté une fonction (non exécutable par l'utilisateur) qui vérifie si une matrice est triangulaire supérieure (`is_upper_triang`).

1.2 Modifications

Pour implémenter ces algorithmes, nous avons suivi les propositions d'amélioration fournies dans les énoncés, en particulier le traitement des cas limites. Par exemple, nous avons pris en compte les systèmes n'ayant aucune solution ou une infinité de solutions dans `back_sub`, qui repose sur nos fonctions de base. De même, dans l'opération `qr`, nous avons intégré la gestion des exceptions à l'aide de l'extension algorithmique proposée. Enfin, la fonction `lstsq` a été implémentée à l'aide de nos fonctions de base, ainsi que des algorithmes de `back_sub` et de `qr`.

1.3 Fonction supplémentaire

Nous avons créé une fonction supplémentaire qui vérifie si la matrice en argument est bien une matrice triangulaire supérieure : `is_upper_triang`.

1.4 Multithreading

Lors du développement du multithreading (parallélisme), nous avons été confrontés à plusieurs choix. Nous nous sommes demandé s'il était judicieux de regrouper toutes les fonctions impliquant des threads dans un fichier séparé au sein du dossier `src`. Finalement, nous avons décidé de les inclure selon la logique déjà présente, c'est-à-dire dans le fichier correspondant au format de sortie de la fonction. Nous avons également choisi de ne pas conserver les versions séquentielles (sans multithreading) des fonctions, car elles étaient quasiment équivalentes aux nouvelles fonctions utilisant un seul thread. Enfin, nous avions initialement intégré des `mutex` dans notre implémentation, mais après avoir analysé les performances avec et sans ceux-ci, nous avons décidé de les retirer : ils ralentissaient l'exécution, alors que l'ensemble de nos codes fonctionnait correctement sans.

2 Tests unitaires

Pour chaque fonction, nous avons écrit une fonction de test correspondante, nommée de la manière suivante : `test_nom_de_la_fonction`. Ces fonctions de test ont pour objectif de valider le bon fonctionnement de l'implémentation dans divers scénarios, y compris les cas limites.

Au début de chaque fonction de test, nous créons les objets nécessaires à l'exécution, tels que des matrices ou des vecteurs, en fonction des besoins spécifiques de la fonction à tester. Pour la création de ces objets, nous utilisons soit `rand()` afin de générer des données aléatoires et générales, soit des valeurs choisies spécifiquement pour simuler des cas limites.

Dans un premier temps, nous évaluons la fonction dans un environnement mono-threadé, c'est-à-dire avec un seul thread. Cela permet de vérifier le bon fonctionnement de base de l'algorithme.

Ensuite, nous testons la fonction en mode multithreadé, avec différentes configurations de nombre de threads. L'objectif est de s'assurer que le comportement reste correct et performant lors d'une exécution parallèle. Ces tests incluent également des cas extrêmes, comme l'utilisation d'un très grand nombre de threads ou, au contraire, d'un nombre nul ou négatif.

Enfin, nous veillons systématiquement à libérer la mémoire allouée à l'aide des fonctions `free`, afin d'éviter toute fuite de mémoire lors de l'exécution des tests.

Les cas limites systématiquement pris en compte sont les suivants :

- Objets de tailles incompatibles : l'exécution est interrompue.
- L'objet destiné à contenir le résultat est `NULL`, ou les deux objets en entrée sont `NULL` : l'exécution est interrompue.
- Si un seul des deux objets en entrée est `NULL`, le calcul est effectué avec l'objet restant.
- Nombre de threads négatif ou nul : l'exécution est interrompue.
- Nombre de threads supérieur à la taille de l'objet : l'exécution se poursuit, mais le nombre de threads est ajusté à la taille de l'objet.

Ainsi, l'ensemble des tests mis en place permet de couvrir une large gamme de cas d'usage tout en assurant la robustesse et la fiabilité de l'implémentation, quelle que soit la configuration d'exécution.

3 Performances

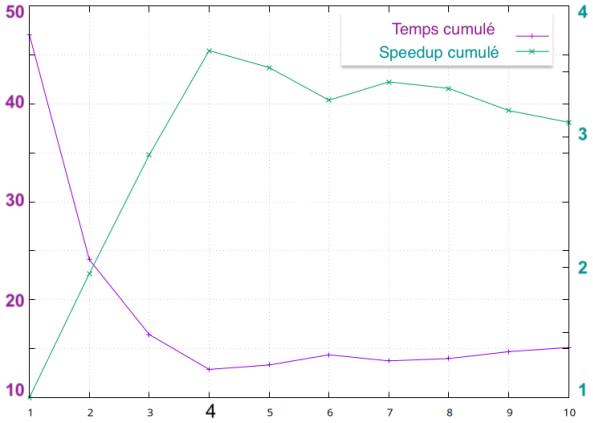
3.1 Temps d'exécution

Temps moyen d'exécution pour une régression polynomiale de degré 4 pour 5 points (gauche) ainsi que pour une régression polynomiale de degré 400 pour 500 points (droite) :

Threads	Temps (s)
1	0.0102
2	0.0166
3	0.0201
4	0.0242

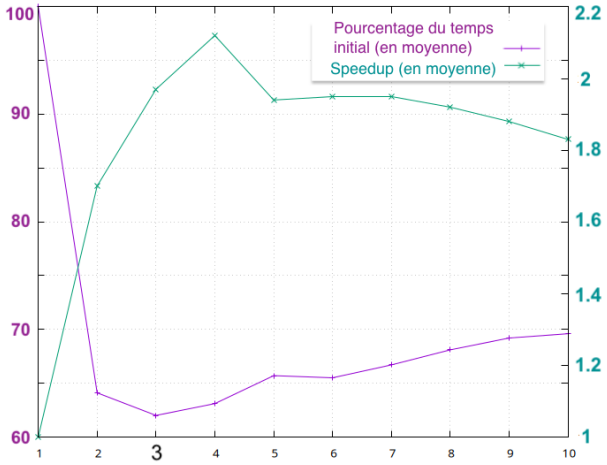
Threads	Temps (s)
1	21.472
2	35.872
3	52.884
4	76.233

Répartition : temps & facteur d'accélération cumulés



(a) Axes de gauche et de droite (respectivement) : temps (en secondes) et facteur d'accélération (speedup, par rapport au temps pour 1 thread) cumulés. Ces valeurs sont mesurées et affichées pour 1-10 threads.

Répartition : temps & facteur d'accélération moyens



(b) Axes de gauche et de droite (respectivement) : temps (% du temps pour 1 thread) et facteur d'accélération (speedup) moyens. Valeurs mesurées pour 1-10 threads.

Nous observons que comme prévu, toutes les fonctions basiques gagnent en vitesse d'exécution. Nous avons en plus essayé de multithreader `lstsq`, mais nous observons que la performance temporelle de cette fonction est impactée négativement par une telle modification. Selon nous, cela est dû au fait que cette fonction utilise plus de ressources pour créer et gérer les threads que ce que permet d'économiser le multithreading.

3.2 Consommation de mémoire

Pour effectuer une régression linéaire de degré 400 de 500 point, nous utilisons la mémoire suivante :

- 323 553 allocations et libération
- 653 770 207 bits alloués

3.3 Consommation d'énergie

Lors de l'exécution d'une régression linéaire de degré 400 de 500 point, notre Raspberry-pi consomme un puissance de $1.55W$.

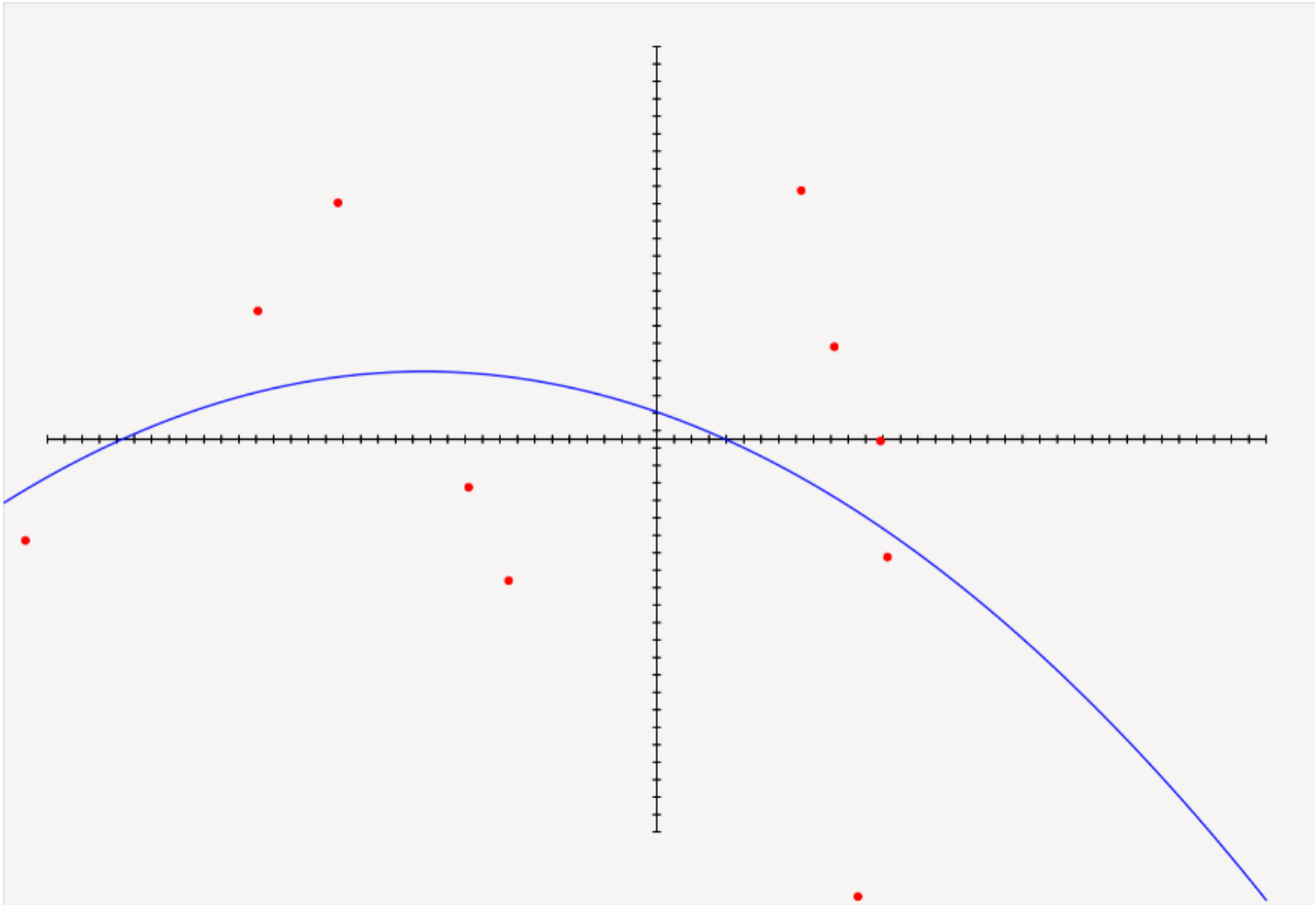
4 Bonus

4.1 Bonus 1

Afin de tester nos fonctions de manière approfondie et sans devoir recréer des vecteurs en permanence, nous avons décidé d'écrire un code permettant d'effectuer rapidement des régressions linéaires. Pour ce faire, nous avons utilisé le package `gtk`, qui nous permet d'afficher un repère orthonormé afin de placer dynamiquement les points à régresser, et ainsi tester nos fonctions de manière interactive. Ce code se trouve sur une branche annexe

`main_bonus1_application`, car certains appels à `#include` (pour charger les packages nécessaires) bloquent la pipeline, ce qui empêche un merge avec la branche principale.

Illustration de notre application



Voici le rendu de l'application ou on a placé à la souris les dix points rouges.

4.2 Bonus 2

Afin de dynamiser un peu notre code, nous avons implémenté deux fichiers supplémentaires : `create_vector.c` et `create_matrix.c`, qui permettent la création de vecteurs et de matrices directement depuis le terminal. Il suffit (par exemple) d'utiliser la commande `./create_matrix "matrix1" "[[1.0 2.0] [3.0 4.0]]"` pour générer un fichier `matrix1.bin` contenant la matrice spécifiée précédemment, utilisable de la même manière que les vecteurs ou matrices générés avec `generator_vector.c` et `generator_matrix.c`. Cette implémentation se trouve sur la branche `main`.

5 Bibliographie

Références

[1] O. Bonaventure, G. Detal et C. Paasch, *Syllabus de Programmation en Langage C*, (2021), Disponible à l'adresse : `https://sites.uclouvain.be/SyllabusC/notes/Theorie/index.html`.

[2] INGInious-UCL, *Projet P3 exercices*, (2025), Disponible à l'adresse : `https://inginious.info.ucl.ac.be/course/LEPL1503`.

[3] OpenAI, *ChatGPT*, (2025), Disponible à l'adresse : `https://chatgpt.com`[1].

[4] Scribens, (2025), Outil utilisé pour la correction orthographique et grammaticale du rapport, Disponible à l'adresse : `https://www.scribens.fr`.

1. ChatGPT a été utilisé pour identifier certains problèmes dans le code. Aucun contenu ni algorithme n'a été généré automatiquement : toutes les implémentations sont le fruit du travail des membres du groupe.