LINMA1170 – Devoir 2 : Gradient conjugué

V. Degrooff, C. Heneffe, N. Tihon, J.-F. Remacle

2024–2025

Contexte

Grâce à vos compétences récemment acquises dans le cours EPL1100, les éléments finis et l'élasticité linéaire n'ont plus de secret pour vous. Un musicien vous contacte donc au sujet de son nouveau diapason dont il est mécontent. Il a constaté qu'il ne vibrait pas à la bonne fréquence et vous demande de simuler sa déformation.

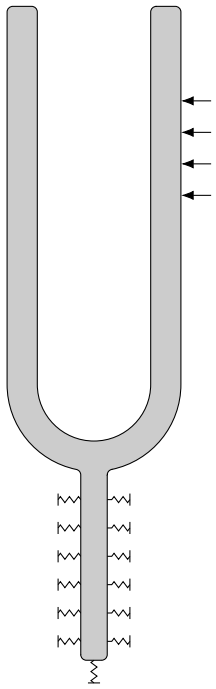

Figure 1: Le diapason et ses conditions frontières

Pour ce devoir, on ne s'intéresse pas encore à la dynamique du système, mais uniquement au cas statique. Pour représenter la mise en vibration du diapason, on considère une force sur une des branches, i.e. une condition de Neumann. Le manche, lui, est maintenu en position grâce à un support modélisé comme un ressort, i.e. une condition de Robin. En suivant la recette habituelle, on obtient finalement un système linéaire

$$Kx = f \quad \text{où}$$

- $K \in \mathbb{R}^{2n \times 2n}$ est la matrice de raideur,
- $x \in \mathbb{R}^{2n}$ est le vecteur de déplacements nodaux u_x, u_y,
- $f \in \mathbb{R}^{2n}$ le vecteur de forces nodales f_x, f_y et
- n est le nombre de noeuds.

Vous n'avez *qu'à* résoudre le système linéaire, vous ne devez pas calculer la matrice de raideur ou le vecteur de forces. Nous vous fournissons le code d'élasticité linéaire qui s'en charge :

```
|-- Makefile
|-- deformation [executable]
|-- gmsh-sdk    [download]
|   |-- include
|   |-- lib
|   |-- ...
|-- include
|   |-- ...
|-- models
|   |-- model_fork.c
|   |-- ...
|-- src
    |-- main.c
    |-- devoir_2.c [TODO]
    |-- model.c
    |-- renumber.c
    |-- utils.c
    |-- utils_gmsh.c
```

Vous devez donc télécharger le sdk gmsh et l'insérer dans le projet. Pour compiler et exécuter le code:

```
make
./deformation <model> <mesh_size_factor>
```

- `model` est `fork` (un des modèles dans `models/`) et
- `mesh_size_factor` est un float influençant la taille de maille et donc le nombre d'inconnues (commencez avec 1.0 et diminuez par la suite).

Dans l'état, le code calcule la solution avec un solveur direct et affiche la déformation dans l'interface graphique gmsh. Vous n'avez plus qu'à implémenter l'algorithme de gradient conjugué pour passer de 1000 noeuds à 100 000 noeuds. *Le projet a été écrit sous Linux, il est donc possible qu'il y ait quelques soucis techniques a régler pour ceux sous Windows, ou macOS.*

Implémentations

On vous demande d'écrire les quatre fonctions suivantes dans un fichier `devoir_2.c` :

1. Une fonction `int csr_sym()` qui renvoie `0` si vous travaillez avec toute la matrice A, ou bien `1` si vous ne travaillez qu'avec sa partie symétrique inférieure.
2. Une fonction qui calcule la solution d'un système $Ax = b$ en utilisant la méthode du gradient conjugué avec comme critère d'arrêt $\sqrt{\frac{r^T r}{r_0^T r_0}} < \epsilon$. La solution est sotckée dans le vecteur x. Pour cette fonction, vous ferez probablement appel à une routine qui calcule un produit matrice-vecteur.

   ```
   int CG(int n, int nnz, const int *rows_idx, const int *cols,
          const double *A, const double *b, double *x, double eps);
   ```

   - `n` indique la taille de la matrice
   - `nnz` indique le nombre d'élément non nul de la matrice
   - `rows_idx`, `cols` et `A` contiennent le stockage d'une matrice A sous format CSR
     - `rows_idx` est de taille $n + 1$
     - `cols` est de taille nnz
     - `A` est de taille nnz

- `b` est un vecteur de taille n
- `x` est un vecteur de taille n
- `eps` est la tolérance de l'erreur relative à atteindre
- En sortie, la fonction retourne le nombre d'itérations nécessaires

3. Une fonction qui calcule la factorisation incomplète $LU = LDL^*$ d'une matrice A.

```
void ILU(int n, int nnz, const int *rows_idx, const int *cols,
         const double *A, const double *L);
```

- `n`, `nnz`, `row_idx`, `cols` et `A` : cf. sup
- `L` de même taille que `A` contient en sortie sa factorisation incomplète
  - si `csr_sym()` renvoie `0` : le tableau csr `L` contient L dans sa partie strictement inférieure et U dans sa partie supérieure
  - si `csr_sym()` renvoie `1` : le tableau csr `L` contient L dans sa partie strictement inférieure et D dans sa diagonale

4. Une fonction qui calcule la solution d'un système $Ax = b$ en utilisant la méthode du gradient conjugué préconditionnée avec la factorisation incomplète de la matrice A. Le critère d'arrêt est le même que celui de CG. La solution est sotckée dans le vecteur x. Pour cette fonction, en plus de la routine matrice-vecteur, vous aurez besoin d'une routine pour résoudre les systèmes triangulaires $L(Ux) = b$.

```
int PCG(int n, int nnz, const int *rows_idx, const int *cols,
        const double *A, const double *b, double *x, double eps);
```

- `n`, `nnz`, `row_idx`, `cols`, `A`, `b`, `x`, `eps` : cf. sup
- En sortie, la fonction retourne le nombre d'itérations nécessaires

Questions théoriques

- Sachant que les normes des erreurs satisfont

$$\|e_i\|_A \leq 2\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^i \|e_0\|_A,$$

obtenez une borne supérieure sur N, le nombre d'itérations nécessaires au gradient conjugué pour obtenir une solution dont l'erreur relative est inférieure à une certaine précision ϵ, ie

$$N \text{ tel que } \frac{\|e_N\|_A}{\|e_0\|_A} \leq \epsilon.$$

L'expression de N ne dépend que de κ et de ϵ.

- Estimez le gain, en nombre d'opérations, apporté par l'utilisation du stockage creux par rapport à un stockage plein et à un stockage bande durant une itération de la méthode de CG et de PCG.

Analyses numériques

- Comparez la complexité temporelle de votre implémentation *CG* et *PCG* pour résoudre des systèmes linéaires. Présentez vos résultats temps/taille sur un plot **log-log**.
- Comparez l'évolution de l'erreur relative des résidus $\frac{\|r_k\|}{\|r_0\|}$ en fonctions du nombre d'itérations k entre l'algorithme *CG* et *PCG* pour résoudre un même système linéaire.
- (Bonus) Calculez numériquement le nombre de condition de votre matrice A et vérifiez si votre borne supérieure du nombre d'opération de CG est une bonne approximation.

En bref

On vous demande de soumettre **par groupe de deux** sur Gradescope :

- votre fichier de code `devoir_2.c` qui contient au moins les fonction demandées
  - La seule librairie externe admise, mais non obligatoire, est la librairie `BLAS`
- votre rapport réalisé avec LaTeX nommé `devoir_2.pdf`, et sa source `devoir_2.tex`
  - vous y répondez aux questions théoriques
  - vous y présentez vos analyses numériques
  - dans un format de maximum 3 pages (4 si bonus) avec `documentclass article [11pt]` en `pagestyleplain`.
- le script (`.py` probablement) qui reproduit les résultats/figures de votre rapport

pour le **18 avril 2025 à 18h**.

Toutes les implémentations et les sources des rapports seront soumises à un logiciel anti-plagiat.

Développer du code se fait rarement sans erreur... C'est pourquoi il faut développer en local. On a donc décidé de brider le nombre de soumissions sur le Gradescope du cours. N'y soumettez votre code que lorsque vous êtes confiant qu'il est correct.

[01 avril 2025 : création]

[01 avril 2025 16h35 : précisions des consignes]

[07 avril 2025 : utils.c non obfusqué]

[10 avril 2025 : correction typo borne question théorique 1]