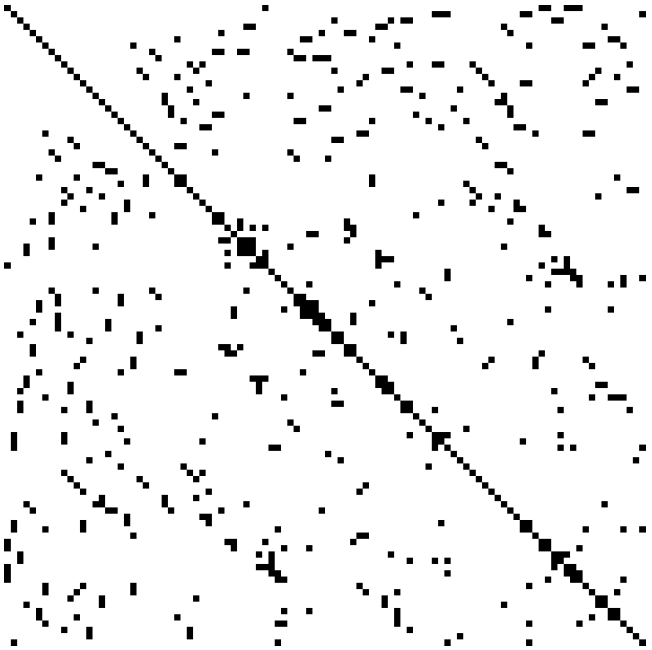
TP 4 : Optimisation et unittesting

Contexte

De nombreuses applications nécessitent l'utilisation de matrices de très grande taille. Avec par exemple des millions de lignes et de colonnes. C'est nottament le cas en sciences et ingénierie lors de la résolution d'équations aux dérivées partielles (EDPs), par exemple avec la méthode des éléments finis. Par chance, ces très grandes matrices contiennent une large majorité de zéros. C'est ce que l'on appelle des **matrices creuse** (sparse) en anglais. Ainsi, leur facteur de remplissage est inférieure à 1%. Par exemple, on représente ci-dessous les valeurs non nulles d'une matrice d'éléments finis.



Puisqu'il y a en fait très peu d'information à stoquer, c'est à dire les éléments non nuls de la matrice, on ne va pas utiliser des tableaux numpy de taille (n,m) (dit "plein") mais une format spécifique. Parmis les plus classiques, on va implémenter le format **COO** (coordinate list) qui n'est forcément le plus efficace mais qui présente de bonnes performances. Il se présente de la façon suivante:

> pour chaque valeur non nulle on stoque le triplet (ligne, colonne, valeur)

Le produit matrice vecteur est une des opérations de base pour tout bibliothèque matricielle. Pour rappel

Si $A = (a_{ij})$ est une matrice de type (m, n) et $\mathbf{b} = (b_i)$ est un vecteur de type taille n, alors leur produit, noté $A\mathbf{b} = \mathbf{c}^T$ est un vecteur de taille m donnée par :

$$\forall i : c_i = \sum_{k=1}^{n} a_{ik} b_k = a_{i1} b_1 + a_{i2} b_2 + \cdots + a_{in} b_n$$

Exercice: benchmarker l'implémentation de la matrice COO

Puisque la performance est clé dans ce contexte, nous allons nous efforcer d'obtenir les meilleurs résultats possibles en termes de vitesse. Même si tous les exemples vus dans le cours instrumentent des fonctions, on ne peut pas faire l'économie d'une structure particulière et donc d'une classe. À vous de choisir la meilleure option pour tester et instrumenter vos codes.

> Attention, dans cet exercice on va chercher à "optimiser" notre représentation de matrice. Cependant, n'oubliez pas:
>
> - Make it work, make it right, make it fast (https://wiki.c2.com/?MakeItWorkMakeItRightMakeItFast) dans cet ordre. On évitera donc l'optimisation prématurée.
> - Faites fonctionner les classes.
> - Mettre en place le benchmark.
> - Optimiser

Benchmark

Pour tous les exemples ci-dessous, on comparera l'usage mémoire et le temps de calcul.

1. Comparaison avec numpy.ndarray(plein)
   - Créer un matrice (pleine) aléatoire de taille 10000×10000 et la multiplier avec un vecteur de taille 10000.
   - Faire de même pour une matrice creuse avec un remplissage de 1/1000.
2. Pour les matrices creuses:
   - Générer une matrice de dimension $10^6 \times 10^8$ avec 10^7 entrés non nulles et la multiplier avec un vecteur de taille correspondante.
   - Mesurer une fois l'usage mémoire pour `MatCooPure` et `MatCooNumpy`
   - Comparer les implémentations de la question suivante.

On fera en sorte que les sorties produisent un rapport lisible pour chacun des benchmarks, par exemple en utilisant des f-strings ou un plot.

> *Remarque*: Il peut être judicieux (ou non) de placer ces tests dans des fonctions (!mémoire!)

Classe(s) matrice COO

> L'utilisation de fichier python séparée semble opportune.

Créer

1. une classe matrice COO en python pur `MatCooPure`
2. une classe matrice COO en python utilisant numpy `MatCooNumpy` Si vous le souhaitez, vous pouvez programmer une classe mère `MatCoo` pour simplifier les appels. Ce n'est cependant pas nécessaire pour notre benchmark.

Pour les deux classes, vous programmerez:

- une initialisation à au hasard avec comme argument : (n,m, n_non_nul)
- une initialisation de la matrice identité (à tester formellement)
- un affichage adapté (extrait + dimensions réelles et mémoire) comprenant un test
- un produit matrice vecteur naïf (qui fonctionne) avec test
- un ou des produits matrice vecteur plus rapides. **Pour ce dernier point:** on pourra au choix coder des méthodes **ou** des fonctions dont le premier argument est une instance de ces classes.