CHAPITRE XVIII

Systèmes linéaires et matrices

Systèmes linéaires. Considérons un *système d'équations linéaires* :

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= y_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= y_2 \\ & \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n &= y_m \end{cases}$$

En algèbre linéaire on développe la théorie de ces systèmes sur un corps ou un anneau $\mathbb{K}$, et on apprend certaines méthodes pour leur résolution, notamment la méthode de Gauss rappelée plus bas. Ici les données a_{ij} et y_i sont des coefficients dans $\mathbb{K}$ et les x_j sont les inconnues, à déterminer dans la suite. Pour $\mathbb{K} = \mathbb{Z}$ ou $\mathbb{K} = \mathbb{Q}$ ou $\mathbb{K}$ un corps fini, on peut effectuer ces calculs de manière exacte sur ordinateur. Lorsque $\mathbb{K} = \mathbb{R}$ ou $\mathbb{K} = \mathbb{C}$, par contre, on fait recours au calcul numérique arrondi : les données initiales, les calculs intermédiaires et les résultats finaux ne sont que des valeurs approchées.

Structuration du problème. Comme vous en avez l'habitude, la matrice $A = (a_{ij})_{j=1,\dots,n}^{i=1,\dots,m}$ et les vecteurs $x = (x_j)_{j=1,\dots,n}$ et $y = (y_i)_{i=1,\dots,m}$ permettent d'écrire ce système plus succinctement comme

$$Ax = y.$$

C'est bien plus qu'une notation commode. Cette structuration des données est le point de départ de tous les algorithmes pour la résolution de systèmes linéaires. Ayant fixé les données A et y, il s'agit de trouver les solutions x vérifiant $Ax = y$. Or, les problèmes liés à la résolution des systèmes linéaires sont divers. Il y a d'abord la question de la représentations des données et du choix des algorithmes adaptés (petites matrices denses, grandes matrices creuses, matrices trigonales, en blocs, en bandes, etc.). Il y a d'une part des questions habituelles de la complexité algorithmique. Il y a d'autre part, lors du calcul *numérique*, les problèmes liés au conditionnement du système et à la propagation d'erreurs.

Résolution d'un système linéaire. Si A est une matrice inversible, de taille $n \times n$, le système $Ax = y$ est équivalent à $x = A^{-1}y$. C'est cette méthode qui est souvent utilisée dans les petits exemples, disons $n = 3$ ou $n = 4$. On développera cette démarche au §1 pour les matrices de petite taille, disons $n \leq 10$, par la méthode de Faddeev. Soulignons deux avertissements :

- ☞ Le calcul de la matrice inverse A^{-1} est équivalent à la résolution de n systèmes linéaires $Av_i = e_i$. Ceci peut être assez coûteux, à savoir d'ordre $O(n^3)$ opérations si A est dense, c'est-à-dire, ne comporte que peu de coefficients nuls. Ce n'est pas toujours la meilleure solution.
- ☞ La formule de Cramer qui nécessite le calcul de $n+1$ déterminants, soit $(n+1)!$ opérations, est, quant à elle totalement inexploitable. (Calculer 10! ou 20! voire 50! pour vous en convaincre.) Aussi importante qu'elle soit pour la théorie, ne songez pas à l'utiliser sur ordinateur.

Ce chapitre rappelle et implémente quelques méthodes élémentaires, notamment l'élimination de Gauss, permettant de résoudre des systèmes de taille moyenne, disons $n \leq 100$. Nous n'indiquerons qu'en passant des problèmes possibles et des variantes qui remédient aux inconvénients les plus fréquents.

Sommaire

1. Implémentation de matrices en C++

1.1. Matrices denses vs creuses. Commençons par les éléments de base de tout traitement algorithmique : la représentation des données et les opérations élémentaires. Cette première étape est de grande importance, car la modélisation dépend fortement du champs d'application envisagé :

- Veut-on modéliser des matrices *denses*, c'est-à-dire comportant peu de coefficients nuls ? S'agit-il des matrices génériques ? ou symétriques ? Ont-elles des propriétés particulières ? ...
- Veut-on modéliser des matrices *creuses*, c'est-à-dire comportant beaucoup de zéros ? S'agit-il des matrices concentrées autour de la diagonale ? Ou des matrices triangulaires ? Ou en blocs ? ...

Question 1.1 (à titre d'exemple)**.** Une matrice dense de taille $m \times n$ nécessite le stockage de mn coefficients, ce qui peut facilement déborder la mémoire disponible. Discuter si l'on peut stocker puis additionner et multiplier des matrices denses de taille 100×100, 1000×1000, ou $10^4 \times 10^4$, ou $10^5 \times 10^5$, etc.

Jusqu'où est-ce possible pour des matrices creuses ? Précisez quel genre de matrices creuses vous considérez puis esquissez comment les stocker et comment les additionner et multiplier.

1.2. Une implémentation faite maison. Dans ce chapitre on considérera des matrices denses, alors que le projet annexe sur Google traitera un cas de matrices creuses.

Exercice/P 1.2. Le fichier `matrix.cc` implémente une classe générique `Matrix<T>` pour stocker des matrices denses de taille $m \times n$ dont les coefficients sont d'un type `T` et indexés par (i, j) avec $i \in \{1, \dots, m\}$ et $j \in \{1, \dots, n\}$. Essayez de comprendre le code déjà implémenté. Ajouter les opérations usuelles :

```
Matrix<T> operator + ( const Matrix<T>& a, const Matrix<T>& b )
Matrix<T> operator - ( const Matrix<T>& a, const Matrix<T>& b )
Matrix<T> operator * ( const Matrix<T>& a, const Matrix<T>& b )
```

Conseil. — Quand vous utilisez des constantes, il vaut mieux écrire `T(0)` et `T(1)` au lieu de `0` et `1` de type `int` : la conversion explicite assurera le bon résultat, quelque soit le type `T` utilisé. Sans cette précaution, vous obligez le compilateur à deviner lui-même la conversion, ce qui n'est pas toujours possible. Même si c'est possible, la conversion implicite choisie n'est pas forcément celle que vous souhaitez.

Gestion d'exceptions. — Quand les dimensions de `a` et `b` correspondent, l'implémentation ne pose pas de problème. Sinon, il faut décider comment gérer cette exception. On peut renvoyer la matrice vide, de dimension 0×0, pour signaler l'erreur ou bien afficher un message d'erreur et abandonner le calcul. On pourrait aussi « tronquer » convenablement les matrices, ou bien les « stabiliser » c'est-à-dire les élargir convenablement en ajoutant des coefficients zéros.

Remarque 1.3 (complexité)**.** Pour estimer le coût des calculs ultérieurs il est important de connaître d'abord la complexité des opérations élémentaires. On considère les matrices $n \times n$ et on compte le nombre d'opérations effectuées sur les coefficients.

(1) Pour l'addition $C = A + B$ on parcourt toutes les paires (i, j) par deux boucles imbriquées pour $i = 1, \dots, n$ et $j = 1, \dots, n$. Pour chaque (i, j) on calcule $c_{ij} \leftarrow a_{ij} + b_{ij}$. Ceci fait n^2 additions.

(2) Pour la multiplication scalaire $C = \lambda A$ on utilise deux boucles imbriquées et calcule $c_{ij} \leftarrow \lambda a_{ij}$ pour chaque (i, j). Ceci nécessite n^2 multiplications, et on ne peut pas espérer de faire mieux.

(3) La multiplication $C = A * B$, par contre, est plus complexe. Le calcul directe de $c_{ij} \leftarrow \sum_{k=1}^{n} a_{ik} b_{kj}$ nécessite *trois* boucles imbriquées, pour i et j et k. Cet algorithme de multiplication nécessite donc n^3 multiplication et $n^2(n-1)$ additions, au total donc presque $2n^3$ opérations.

1.3. Multiplication d'après Strassen. La définition du produit $C = A * B$ par la formule $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$ donne immédiatement lieu à un algorithme de multiplication de matrices. Si cet algorithme est satisfaisant pour les matrices de petite taille, le coût cubique se fait sentir pour les grandes matrices. En 1969 V. Strassen découvrit une multiplication plus rapide.

Puisque les multiplications sont plus coûteuse que les additions, on essaiera d'économiser les multiplications — même au prix de quelques additions supplémentaires. Concrètement, pour calculer $C = A * B$ on suppose A, B, C de taille $n \times n$ avec $n = 2m$. On les décompose en blocs de taille $m \times m$:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \qquad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \qquad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}.$$

On veut implémenter les formules qui définissent la multiplication des matrices :

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}, \qquad C_{12} = A_{11}B_{12} + A_{12}B_{22},$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21}, \qquad C_{22} = A_{21}B_{12} + A_{22}B_{22}.$$

Comme telles, ces formules utilisent 8 multiplications et 4 additions. On peut faire avec 7 multiplications et 18 additions. D'abord on calcule les sept produits auxiliaires suivants :

$$P_1 := (A_{11} + A_{22})(B_{11} + B_{22}), \qquad P_2 := (A_{21} + A_{22})B_{11},$$
$$P_3 := A_{11}(B_{12} - B_{22}), \qquad P_4 := A_{22}(B_{21} - B_{11}),$$
$$P_5 := (A_{11} + A_{12})B_{22}, \qquad P_6 := (A_{21} - A_{11})(B_{11} + B_{12}),$$
$$P_7 := (A_{12} - A_{22})(B_{21} + B_{22}).$$

Le calcul de $P_1, \ldots, P_7$ utilise 7 multiplications et 10 additions / soustractions. On peut en déduire les coefficients de C avec les 8 additions / soustractions suivantes :

$$C_{11} = P_1 + P_4 - P_5 + P_7, \qquad C_{12} = P_3 + P_5,$$
$$C_{21} = P_2 + P_4, \qquad C_{22} = P_1 - P_2 + P_3 + P_6.$$

Quel est donc l'intérêt de remplacer 8 multiplications et 4 additions par 7 multiplications et 18 additions ? N'est-ce pas une perte d'efficacité ? C'est sans doute vrai pour les matrices 2×2, mais pour les matrices plus grandes il faut regarder de plus prêt :

	Formules de la définition :		Formules de Strassen :	
	multiplications	additions	multiplications	additions
taille n	n^3	$n^2(n+1)$	$7(\frac{n}{2})^3$	$7(\frac{n}{2})^2(\frac{n}{2}-1)+18(\frac{n}{2})^2$
2	8	4	7	18
4	64	48	56	100
6	216	180	189	288
8	512	448	448	624
10	1000	900	875	1150
12	1728	1584	1512	1908
14	2744	2548	2401	2940
16	4096	3840	3584	4288
18	5832	5508	5103	5994
20	8000	7600	7000	8100
22	10648	10164	9317	10648
24	13824	13248	12096	13680
26	17576	16900	15379	17238
28	21952	21168	19208	21364
30	27000	26100	23625	26100

On voit que la méthode de Strassen commence à s'amortir à partir d'une certaine taille n_0 entre 16 et 30. Le point exact dépend du coût respectif de la multiplication et de l'addition des coefficients. Mais quelque soit cette pondération, au plus tard pour $n = 30$ la méthode de Strassen devient plus efficace.

Application récursive. — Pour les grandes matrices on peut appliquer la méthode de Strassen de manière récursive. Pour les additions / soustractions on utilise l'algorithme évident qui ne laisse rien à désirer. Pour les 7 multiplications, par contre, on applique à nouveau la méthode de Strassen aux matrices de tailles $\frac{n}{2} \times \frac{n}{2}$. Ainsi si $c(n)$ dénote le côut de la multiplication de deux matrices de taille $n \times n$, on obtient la formule de récurrence $c(n) = 7c(\lceil \frac{n}{2} \rceil) + 18\alpha(\frac{n}{2})^2$ où α est le coût d'une addition de coefficients. On en déduit que $c(n)$ est d'ordre $O(n^\sigma)$ avec un exposant $\sigma = \log_2(7) \approx 2.808$.

Exercice 1.4. Le fichier `strassen.cc` implémente la multiplication de matrices d'après Strassen comme expliquée ci-dessus. Aux additions et multiplications s'ajoute le coût pour *copier* les sous-matrices. Pour la complexité asymptotique c'est négligeable, mais dans une implémentation conrète ce problème décale le point d'amortissement. Vous pouvez empiriquement déterminer les champs d'applications des deux méthodes : dans notre implémentation non-optimisée Strassen commence à s'amortir à partir de $n \approx 100$. Le gain est de 10% pour $n \approx 200$, de 20% pour $n \approx 300$, et de 40% pour $n \approx 500$.

Remarque 1.5. Malheureusement, l'algorithme de Strassen est numériquement instable : à cause des additions / soustractions supplémentaires il introduit typiquement plus d'erreurs d'arrondi que l'algorithme classique. Pour cette raison il n'est en général pas utilisé pour le calcul numérique. Lors d'un calcul exact, par contre, il est nettement plus efficace pour les grandes matrices que l'algorithme cubique.

Remarque 1.6. En 1987 D. Coppersmith et S. Winograd ont publié un algorithme de complexité $O(n^{2.376})$, mais jusqu'ici c'est resté un résultat purement théorique. Rien n'indique que l'exposant est optimal, et on pourrait soupçonner que l'on puisse arriver à $O(n^{2+\varepsilon})$ pour tout $\varepsilon > 0$. Il est claire, par contre, que l'exposant ne peut être inférieur à 2 puisque l'algorithme doit au moins écrire les n^2 coefficients du résultat.

1.4. Inversion d'après Faddeev. À partir des opérations élémentaires, que nous venons de discuter, on peut déjà calculer l'inverse d'une matrice par une méthode simple mais astucieuse découverte par L. Faddeev. On la présente ici parce qu'elle s'implémente très facilement — et que sa preuve est amusante.

Soit A une matrice $n \times n$. Son *polynôme caractéristique* est $P(X) := \det(A - XI) \in \mathbb{K}[t]$. La méthode de Faddeev permet de calculer P et, lorsque A est inversible, l'inverse de A.

Proposition 1.7. *Soit $A \in \mathrm{Mat}(n \times n; \mathbb{K})$. On pose $p_0 = 1$ et $B_0 = I$, puis pour $k = 1, \dots, n$ on définit*

$$C_k := A \cdot B_{k-1} = A^k - p_1 A^{k-1} - \cdots - p_{k-2} A^2 - p_{k-1} A$$

ainsi que $p_k = \frac{1}{k} \operatorname{tr} C_k$ et $B_k = C_k - p_k I$. L'algorithme s'arrête avec $B_n = 0$, et le polynôme caractéristique de A est $P = (-1)^n (X^n - \sum_{i=1}^n p_i X^{n-i})$. De plus, si $p_n \neq 0$, l'inverse de A n'est autre que $B = \frac{1}{p_n} B_{n-1}$.

On se propose d'implémenter cette méthode afin de la *tester* empiriquement. De manière complémentaire vous pouvez la *prouver* afin de justifier la correction du programme.

Exercice/M 1.8. Développer une preuve de la proposition par les étapes suivantes :

(1) Vérifier la proposition pour une matrice diagonale $A = \mathrm{diag}(\lambda_1, \lambda_2)$, puis $A = \mathrm{diag}(\lambda_1, \lambda_2, \lambda_3)$. Si vous voulez, vous pouvez tenter une preuve pour $A = \mathrm{diag}(\lambda_1, \dots, \lambda_n)$ quelconque.

(2) Supposons que l'énoncé est vrai pour les matrices diagonales. Reste-t-il vrai pour toute matrice triangulaire A ? Puis pour TAT^{-1}, la matrice conjuguée par $T \in \mathrm{GL}_n \mathbb{K}$? Conclure.

Exercice/P 1.9. Implémenter la méthode de Faddeev en une fonction

```
void faddeev( const Matrix<T>& a, Matrix<T>& b, vector<T>& poly )
```

et l'appliquer aux matrices définies dans les fichiers d'exemples. Vérifier la correction de vos calculs en multipliant A par l'inverse calculé. Quel est le nombre d'opérations aritmétiques nécessaire pour calculer l'inverse d'une matrice $n \times n$ par la méthode de Faddeev ?

Jusqu'ici, au moins au niveau théorique, tout va bien. Pour le calcul numérique il existe pourtant une difficulté supplémentaire : les erreurs d'arrondi ! Voici un exemple classique et assez frappant :

Exercice/P 1.10 (matrices de Vandermonde)**.** Pour $n \in \mathbb{N}$ on définit la matrice V_n de taille $n \times n$ comme ayant i^{j-1} pour coefficient (i, j). Écrire une fonction `Matrix<T> vandermonde( Dim n )` qui construit la matrice V_n, puis calculer l'inverse de $V_6, V_7, V_8, \dots$ (avec vérification bien sûr). Qu'observer vous avec le type `double` ? Pour quels rangs n le résultat est-il acceptable ? Pour quels n est-il grossièrement faux ? En quoi est-ce surprenant ? Comparer avec des calculs exacts utilisant les types `Rationnel` et `RReal`.

Ce test exhibe une difficulté typique : pour n grand, les coefficients de V_n varient violemment, et la matrice se révèle *mal conditionnée*, c'est-à-dire difficile à résoudre par des méthodes numériques (§2.2).

Exercice/P 1.11 (matrices de Hilbert)**.** Voici un deuxième exemple de matrices mal conditionnées. Pour $n \in \mathbb{N}$ on définit la matrice de Hilbert H_n de taille $n \times n$ par les coefficients $\frac{1}{i+j-1}$ pour $i, j = 1, \dots, n$. Écrire une fonction `Matrix<T> hilbert( Dim n )` qui construit la matrice H_n, puis calculer l'inverse de H_n pour n de plus en plus grand. (Ne pas oublier la vérification du résultat.) Pour quels rangs n le résultat est-il acceptable ? Pour quels n est-il grossièrement faux ? En quoi est-ce surprenant ?

A priori ces difficultés numériques pourraient être des artefacts de la méthode de Faddeev. On essaiera plus bas d'invertir ces matrices par la méthode de Gauss, pour comparer laquelle des deux méthodes gère mieux les erreurs d'arrondis. Puis on discutera au §2.2 les propriétés mathématiques qui font que l'inversion d'une matrice peut être numériquement méchante.

2. La méthode de Gauss

Motivation. Comme expliqué au début du chapitre on veut résoudre un système $Ax = y$. Si A est triangulaire supérieure, la solution est immédiate : il suffit de remonter. Plus explicitement, on résout $a_{nn}x_n = y_n$, puis on remonte pour résoudre $a_{n-1,n-1}x_{n-1} + a_{n-1,n}x_n = y_{n-1}$, et ainsi de suite :

$$Ax = y \qquad \text{avec} \qquad A = \begin{pmatrix} a_{11} & \dots & \dots & a_{1n} \\ 0 & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & a_{nn} \end{pmatrix}.$$

Dans ce cas on suppose que tous les coefficients diagonaux a_{kk} sont non nuls, et donc inversibles dans le corps $\mathbb{K}$. Plus généralement la solution est facile si A est échelonnée : ici la matrice A n'est plus supposée inversible, ni même carrée. Rappelons qu'une matrice est dite *échelonnée* si le nombre de zéros précédant le premier coefficients non nul d'une ligne augmente ligne par ligne. C'est le cas dans l'exemple ci-dessous, avec des *pivots* a_{11}, a_{23}, a_{34}, a_{47} non nuls et des coefficients $*$ quelconques :

$$Ax = y \qquad \text{avec} \qquad A = \begin{pmatrix} a_{11} & * & * & * & * & * & * \\ 0 & 0 & a_{23} & * & * & * & * \\ 0 & 0 & 0 & a_{34} & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & a_{47} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

À noter que l'application $f_A \colon \mathbb{K}^n \to \mathbb{K}^m$, $x \mapsto Ax$ n'est en général ni surjective ni injective. Dans l'exemple précédent l'équation $Ax = y$ n'a pas de solution si $y_5 \neq 0$. Si $y_5 = 0$, par contre, les solutions x telles que $Ax = y$ forment un sous-espace affine de $\mathbb{K}^n$ de dimension 3. (Le détailler.)

2.1. L'algorithme de Gauss. La méthode de Gauss pour un système $Ax = y$ consiste à déterminer une matrice inversible M telle que la matrice $U = MA$ soit trigonale supérieure, ou bien échelonnée si A n'est pas nénessairement inversible. Ensuite le système $Ax = y$ est équivalent à $Ux = My$, ce qui se résout par la méthode des remontées.

Plus explicitement, on effectue des opérations sur les lignes afin d'obtenir une matrice échelonnée. Trois opérations sont à notre disposition :

$A_i \leftrightarrow A_j$ **:** échanger la ligne i et la ligne j,

$A_i \leftarrow aA_i$ **:** multiplier la ligne i par un facteur inversible a,

$A_i \leftarrow A_i + aA_j$ **:** ajouter un multiple de la ligne j à la ligne i.

L'invariant. On pose $U_0 = A$ et $M_0 = I$, la matrice identité de taille $m \times m$ afin d'assurer la condition initiale $U_0 = M_0 A$. Chacune des trois opérations de base correspond à multiplier à gauche par une certaine matrice inversible T_k. (Expliciter T_k et T_k^{-1}.) Dans une implémentation on ne créera pas T_k explicitement, mais on l'applique à U_{k-1} pour obtenir $U_k := T_k U_{k-1}$, et simultanément à M_{k-1} afin d'obtenir $M_k := T_k M_{k-1}$. Ainsi M_k est à nouveau inversible et vérifie toujours $U_k = M_k A$. Si finalement on arrive à une matrice $U = U_k$ échelonnée, alors on a trouvé $M = M_k$ inversible de sorte que $U = MA$ comme souhaité.

L'algorithme. Pour la première colonne, on choisit un élément $a_{i,1} \neq 0$, que l'on appelle alors le *pivot*, puis on échange la ligne i et la première ligne : c'est l'opération $L_1 \leftrightarrow L_i$. Maintenant $a_{1,1} \neq 0$ et on peut diviser la première ligne par $a_{1,1}$, c'est-à-dire $L_1 \leftarrow \frac{1}{a_{1,1}} L_1$, afin d'obtenir $a_{1,1} = 1$. Finalement, pour tout $i = 2, \dots, n$, on effectue $L_i \leftarrow L_i - a_{i,1} L_1$ de façon à obtenir, dans la première colonne, des termes nuls. On itère l'algorithme en l'appliquant à la matrice $(n-1) \times (n-1)$ obtenue en ignorant la première ligne et première colonne de A. Si jamais une colonne est entièrement zéro, on procède à la suivante.

Exercice 2.1. Vérifier que dans la méthode de Gauss le nombre d'opérations arithmétiques est d'ordre $O(n^3)$ pour une matrice $n \times n$. Si vous voulez, vous pouvez préciser le coût exact en comptant les différentes opérations sur les coefficients : copie, addition, soustraction, multiplication, division. Une fois les matrices U et M calculées, quel est le coût de résoudre $Ax = y$? À noter qu'il est souvent utile de garder U et M afin de résoudre $Ax = y$ pour plusieurs y.

Exercice 2.2. Cette méthode, appliquée à une matrice carrée, permet aussi de calculer efficacement le déterminant. Vérifier que les trois opérations élémentaires ci-dessus changent le déterminant respectivement par un facteur de -1, de a ou pas du tout. Le déterminant de la matrice finale U est le produit de ses éléments diagonaux, ce qui permet de calculer $\det(A)$. Quelle est la complexité de cette méthode ? Comparer avec le calcul via les permutations : $\det(A) = \sum_{\sigma\in\mathfrak{S}_n} \operatorname{sign}(\sigma)\cdot a_{1,\sigma(1)}\cdot a_{2,\sigma(2)}\cdots a_{n,\sigma(n)}$. Cette formule *polynomiale* est très importante pour la théorie des déterminants, mais elle s'avère catastrophique pour des calculs concrets ! Expliciter pourquoi.

L'algorithme de Gauss-Jordan. Si A est inversible on ne peut tomber, au cours de l'algorithme de Gauss, sur une colonne entièrement zéro. Supposons de plus que dans l'algorithme on élimine les coefficients en dessous *et* au dessus du pivot. Dans ce cas la matrice finale sera $U = I$, donc $M = A^{-1}$. Ceci donne une méthode pratique pour inverser des matrices.

Exercice 2.3. Détailler cette variante de l'algorithme. Montrer ainsi qu'il est possible d'inverser une matrice $n\times n$ avec $O(n^3)$ opérations arithmétiques seulement. Comparer à la méthode de Faddeev (théoriquement puis empiriquement après implémentation, voir plus bas).

Préparation numérique. Afin de minimiser les erreurs d'arrondi, on peut préparer les coefficients de A en divisant la ligne L_i par $\sup_j |a_{i,j}|$. On assure ainsi que $\sup_j |a_{i,j}| = 1$ dans le souci de minimiser les variations dans les coefficients et les pertes de précision qui peuvent en résulter. Il faut aussi remarquer que le choix des pivots *n'est pas anodin* : on choisira en général le pivot de module maximum.

Exercice 2.4. On considère le système $\varepsilon x_1 + x_2 = 1$ et $x_1 + x_2 = 2$ avec $\varepsilon = 10^{-9}$. Résoudre ce système en prenant ε pour pivot, puis en prenant 1 comme pivot. Dans ces calculs on ne travaillera qu'avec 8 chiffres significatifs (type `float`). Comparer puis discuter les résultats.

Implémentation. Après ces préparations, passons à l'implémentation de l'algorithme de Gauss ou, si vous préférez, de la variante de Gauss-Jordan.

Exercice/P 2.5. Écrire une fonction mettant en œuvre la méthode de Gauss :

```
template <typename T>
T gauss( Matrix<T>& a, Matrix<T>& m )
```

Ici `a` est la matrice initiale que sera transformée au cours de l'algorithme, et `m` est la matrice accompagnatrice de sorte que `m*a` reste constant. La valeur renvoyée est le déterminant de la matrice initiale.

Attention. — L'implémentation proposée ici opère directement sur les deux matrices `a` et `m` passées par référence. (Justifier le choix de ce mode de passage.) Aucune matrice auxiliaire, notée T_k plus haut, n'est construite explicitement : ce serait très coûteux et inutilement compliqué. (Expliquer pourquoi.)

Vérification. — Pour une application numérique, veillez en particulier à préparer la matrice `a` comme indiqué ci-dessus, et à choisir à chaque étape un pivot de module maximal. Tester votre fonction sur quelques-uns des exemples précédents, en utilisant le type `double` puis le type `Rationnel`.

Exercice/P 2.6. Écrire une fonction qui permet d'inverser une matrice par la méthode de Gauss. Tester les matrices de Vandermonde V_n et de Hilbert H_n vues plus haut. (Ne pas oublier la vérification du résultat.) Pour quels rangs n le résultat est-il acceptable ? Pour quels n est-il grossièrement faux ? Que dire du temps du calcul ? (Vous pouvez utiliser le fichier `timer.hh` pour chronométrer.)

Exercice/P 2.7. Parfois on ne veut que calculer le déterminant $\det(A)$ sans pour autant calculer M de sorte que MA soit échelonnée. Pour ce cas particulier on pourrait implémenter une fonction optimisée : `template <typename T> T det( const Matrix<T>& a )`. Discuter la complexité de ce calcul. À votre avis, est-ce que le gain de performance justifie une implémentation séparée ?

Exercice/P 2.8 (optionnel)**.** Écrire un programme qui permet de lire une matrice A et un vecteur y d'un fichier, puis résout le système $Ax = y$. Améliorez votre implémentation afin de déterminer noyau et image d'une matrice, par exemple

$$A = \begin{pmatrix} 1 & -8 & -9 & -18 \\ 2 & -11 & -12 & -29 \\ 1 & -8 & -9 & -10 \\ 0 & 5 & 6 & -1 \end{pmatrix}.$$

2.2. Conditionnement. Pour toute méthode numérique il est important de comprendre la propagation d'erreurs afin d'éviter un usage inapproprié. Ainsi tout algorithme a ses limites inhérentes, parfois dues à la méthode, parfois dictées par les données elles-mêmes.

Exemple 2.9. On considère l'équation $Ax = y$ avec

$$A = \begin{pmatrix} 0{,}780 & 0{,}563 \\ 0{,}913 & 0{,}659 \end{pmatrix} \quad \text{et} \quad y = \begin{pmatrix} 0{,}217 \\ 0{,}254 \end{pmatrix}.$$

Lequel des deux résultats approchés suivants est meilleur,

$$\tilde{x} = \begin{pmatrix} +0{,}999 \\ -1{,}001 \end{pmatrix} \quad \text{ou} \quad \hat{x} = \begin{pmatrix} +0{,}341 \\ -0{,}087 \end{pmatrix} ?$$

Comme la solution exacte est souvent inconnue, on pourrait simplement calculer les erreurs $|A\tilde{x} - y|$ et $|A\hat{x} - y|$ et choisir la solution qui minimise cette erreur. Vérifier qu'ici c'est $\hat{x}$. En sachant que la solution exacte est $x = \binom{+1}{-1}$, c'est pourtant $\tilde{x}$ qui est nettement plus proche.

Comment expliquer puis quantifier ce phénomène étrange ? Supposons, comme dans l'exemple, que l'on travaille sur $E = \mathbb{R}^n$ ou $E = \mathbb{C}^n$. On munit cet espace d'une norme $E \to \mathbb{R}_+$, $x \mapsto |x|$, habituellement la norme euclidienne $|x| = \sqrt{\sum_{k=1}^n |x_k|^2}$. On regarde une matrice A de taille $n \times n$ comme application linéaire $E \to E$, et on définit sa norme par $|A| := \sup\{|Ax|; |x| \leq 1\}$.

Remarque 2.10. La norme $|A|$ mesure l'effet de « distorsion » : on a $|Ax| \leq |A| \cdot |x|$ pour tout $x \in E$, et $|A|$ est la plus petite valeur possible pour cette inégalité. Si A est diagonalisable, la norme $|A|$ est simplement la plus grande valeur propre en valeur absolue. (Exercice !)

Dans la suite on considère une matrice A *inversible*, et on s'intéresse à la stabilité numérique du système $Ax = y$. Comment varie la solution x si l'on perturbe le vecteur y ? Soit $y' = y + \delta y$ et x' solution de $Ax' = y'$; on a donc $x' = x + \delta x$ avec $A\delta x = \delta y$. Que dire de l'erreur relative $\frac{|\delta x|}{|x|}$ par rapport à l'erreur relative $\frac{|\delta y|}{|y|}$? D'une part on a $|\delta y| \leq |A| \cdot |\delta x|$ et $|\delta x| \leq |A^{-1}| \cdot |\delta y|$, donc

$$\frac{1}{|A|} \frac{|\delta y|}{|x|} \leq \frac{|\delta x|}{|x|} \leq |A^{-1}| \frac{|\delta y|}{|x|}.$$

D'autre part on a $|y| \leq |A| \cdot |x|$ et $|x| \leq |A^{-1}| \cdot |y|$, donc

$$\frac{1}{\mathrm{cond}(A)} \frac{|\delta y|}{|y|} \leq \frac{|\delta x|}{|x|} \leq \mathrm{cond}(A) \frac{|\delta y|}{|y|}.$$

Ici on a introduit $\mathrm{cond}(A) := |A| \cdot |A^{-1}|$, appelé le *conditionnement* de la matrice A.

Remarque 2.11. Chacune des inégalités précédentes devient une égalité pour un choix convenable de y et δy. Ainsi le conditionnement décrit comment une perturbation de y s'amplifie en une perturbation de x.

- On a toujours $\mathrm{cond}(A) = |A| \cdot |A^{-1}| \geq |AA^{-1}| = |I| = 1$.
- Si $\mathrm{cond}(A)$ est proche de 1, alors une petite perturbation de y entraîne une petite perturbation de x.
- Si $\mathrm{cond}(A)$ est grand, une petite perturbation de y peut entraîner une grande perturbation de x.

Ainsi un mauvais conditionnement de la matrice A implique en général une perte de précision :

☞ *Typiquement* $\mathrm{cond}(A) \approx 10^c$ *veut dire que la donnée de y avec une précision de* ℓ *décimales mène à une solution x avec une précision de* $\ell - c$ *décimales.* ✍

Exemple 2.12. On considère $A = \begin{pmatrix} 7 & 10 \\ 5 & 7 \end{pmatrix}$ avec $A^{-1} = \begin{pmatrix} -7 & 10 \\ 5 & -7 \end{pmatrix}$. Les valeurs propres de A sont $\lambda_1 = 7 - 5\sqrt{2} \approx -0{,}071068$ et $\lambda_2 = 7 + 5\sqrt{2} \approx 14{,}071$, donc $|A| \approx 14{,}071$. Les valeurs propres de A^{-1} sont $-7 \pm 5\sqrt{2}$, donc $|A^{-1}| \approx 14{,}071$ et $\mathrm{cond}(A) \approx 198$. Ceci indique que $Ax = y$ peut être sensible aux perturbations de y. Effectivement $Ax = \begin{pmatrix} 1{,}00 \\ 0{,}70 \end{pmatrix}$ donne $x = \begin{pmatrix} 0{,}00 \\ 0{,}10 \end{pmatrix}$, alors que $Ax' = \begin{pmatrix} 1{,}01 \\ 0{,}69 \end{pmatrix}$ donne $x' = \begin{pmatrix} -0{,}17 \\ 0{,}22 \end{pmatrix}$. On constate que le changement relatif en x est plus grand que le changement relatif en y.

Exercice 2.13. En utilisant un logiciel de calcul formel, calculer le conditionnement des matrices de Vandermonde V_n et de Hilbert H_n de taille $n \times n$. Expliquer ainsi les difficultés numériques rencontrées lors de l'inversion de ces matrices. Est-ce que les mêmes problèmes se font sentir lors d'un calcul exact utilisant le type `Rationnel` ? En revanche, quels problèmes peuvent se présenter dans le calcul exact ?

2.3. Factorisation *LU*. La méthode de Gauss admet de nombreuses variantes et spécialisations à des situations particulières. Nous n'en mentionnons que deux : la factorisation LU puis la méthode de Cholesky.

La factorisation dite *LU* est un cas particulier de la méthode de Gauss où l'on choisit toujours $a_{i,i}$ comme pivot, sans jamais échanger de lignes. En général ce coefficient peut s'annuler, il faut donc une hypothèse supplémentaire :

Proposition 2.14. *Si $A = (a_{i,j})$ est une matrice $n \times n$ telle que les n sous-matrices diagonales $\Delta_k = \begin{pmatrix} a_{1,1} & \cdots & a_{1,k} \\ \vdots & & \vdots \\ a_{k,1} & \cdots & a_{k,k} \end{pmatrix}$ soient inversibles, alors il existe une matrice triangulaire inférieure L et une matrice triangulaire supérieure U telles que $A = LU$. Si l'on impose en outre que tous les éléments diagonaux de L soient égaux à 1, alors il y a unicité.*

Exercice/M 2.15. Montrer la proposition en suivant la méthode de Gauss : étant donnée l'hypothèse, on peut toujours choisir $a_{i,i}$ comme pivot. Vérifier que l'on construit ainsi L et U comme souhaité. Pour l'unicité remarquer que le produit de deux matrices triangulaires inférieures (resp. supérieures) est à nouveau triangulaire inférieure (resp. supérieure).

Remarque 2.16. L'hypothèse de la proposition est vérifiée pour les matrices *à diagonale dominante*, c'est-à-dire vérifiant $|a_{i,i}| > \sum_{j \neq i} |a_{i,j}|$. De telles matrices sont inversibles. Les hypothèses sont encore vérifiées pour les matrices *symétriques définies positives*, c'est-à-dire $A^t = A$ et $v^t A v > 0$ pour tout $v \neq 0$.

Remarque 2.17. L'intérêt de la factorisation *LU* réside dans le fait suivant : le système $LUx = y$ est équivalent à $Lw = y$ et $Ux = w$; on est donc ramené à résoudre deux systèmes triangulaires.

Exercice/P 2.18. Écrire un programme qui réalise la décomposition *LU* d'une matrice carrée A et qui utilise cette décomposition pour résoudre des systèmes linéaires $Ax = y$.

2.4. Méthode de Cholesky. La méthode de Cholesky est un cas particulier de la factorisation *LU* appliquée aux matrices symétriques définies positives :

Proposition 2.19. *Si A est une matrice symétrique définie positive, alors il existe une matrice triangulaire inférieure B telle que $A = BB^t$. Cette matrice est unique si l'on impose la condition $b_{i,i} > 0$ pour tout i.*

Exemple 2.20. Regardons $A = \left(\begin{smallmatrix} 4 & -2 \\ -2 & 10 \end{smallmatrix}\right)$. S'il existe $B = \left(\begin{smallmatrix} b_{11} & 0 \\ b_{21} & b_{22} \end{smallmatrix}\right)$ de sorte que $A = BB^t$, alors on peut déterminer ses coefficients un par un. D'abord $b_{11}^2 = 4$, on pose donc $b_{11} = 2$. Ensuite $b_{11} b_{21} = -2$, donc $b_{21} = -1$. Finalement $b_{21}^2 + b_{22}^2 = 10$, donc $b_{22} = 3$. On vérifie aisément que $BB^t = A$, comme souhaité.

Exercice/M 2.21. Montrer que la méthode esquissée se généralise à toute matrice symétrique définie positive de taille $n \times n$, ce qui démontre la proposition. Vérifier aussi que toute matrice $A = BB^t$, avec B inversible, est symétrique définie positive. La construction précédente de B, dans le cas de réussite, constitue donc une preuve que A est définie positive.

Exercice/P 2.22. Écrire un programme qui vérifie si une matrice A est symétrique définie positive et qui, dans l'affirmative, donne une matrice triangulaire inférieure B à coefficients diagonaux positifs telle que $A = BB^t$. À cet effet on cherchera B par la méthode des coefficients indéterminés.

Exercice/P 2.23. Adapter la méthode de Cholesky pour décomposer, quand cela est possible, une matrice symétrique sous la forme BDB^t où D est diagonale et B est triangulaire inférieure avec $b_{i,i} = 1$. Comme application analyser la matrice suivante. Est-elle définie positive ?

$$A = \begin{pmatrix} 20 & 5 & -1 & 20 \\ 5 & 1 & 0 & 5 \\ -1 & 0 & -2 & -1 \\ 20 & 5 & -1 & 20 \end{pmatrix}.$$

Exercice/P 2.24. Calculer l'inverse de la matrice $X = \left(\begin{smallmatrix} A & B \\ B & A \end{smallmatrix}\right)$ où $A = \begin{pmatrix} 1 & 0 & -2 & -2 \\ 0 & 1 & -2 & -2 \\ -2 & -2 & 1 & 0 \\ -2 & -2 & 0 & 1 \end{pmatrix}$ et $B = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$.

Expliquer la méthode suivie.

PROJET XVIII

Classement de pages web à la Google

Objectifs

- Comprendre le fonctionnement de Google, un outil de recherche omniprésent.
- Résoudre un système linéaire creux par une méthode itérative bien adaptée.

Ce projet implémente la technique utilisée par Google, un moteur de recherche généraliste qui a vu un succès fulgurant depuis sa naissance en 1998. Le point fort de Google est qu'il trie *par ordre d'importance* les résultats d'une requête, c'est-à-dire les pages web associées aux mots-clés donnés. On s'intéresse ici de plus près à l'algorithme de classement, qui est à la fois simple et ingénieux. Il s'agit essentiellement de résoudre un immense système d'équations linéaires.

[1] Vous trouvez un développement détaillé dans l'article *Comment fonctionne Google ?* dont ce projet ne donne qu'un résumé. Cet article discute les motivations et la modélisation générale, alors que ce projet se concentrera sur l'implémentation.

[2] Si vous préférez aller aux sources, et avoir une présentation plus informatique, vous pouvez consulter l'article fondateur : S. Brin et L. Page, *The Anatomy of a Large-Scale Hypertextual Web Search Engine*, Stanford University 1998. (Pour le trouver, utiliser Google. ;-)

[3] Si vous voulez savoir plus sur la foudroyante histoire de l'entreprise Google, ses légendes et anecdotes, vous lirez avec profit le récent livre de David Vise et Mark Malseed, *Google story*, Dunod, Paris 2006.

Sommaire

1. Marche aléatoire sur la toile

1.1. Que fait un moteur de recherche ? À première vue, le principe d'un moteur de recherche est simple : on copie les pages web concernées en mémoire locale, puis on trie le contenu (les mots-clés) par ordre alphabétique afin d'effectuer des recherches lexiques. Une *requête* est la donnée d'un ou plusieurs mots-clés ; la *réponse* est une liste des pages contenant les mots-clés recherchés. C'est en gros ce que faisaient les moteurs de recherche, dits de première génération, dans les années 1990.

L'énorme quantité des données entraîne de sérieux problèmes car le nombre des documents à gérer est énorme et rien que le stockage et la gestion efficaces posent des défis considérables. Plus délicat encore : les pages trouvées sont souvent trop nombreuses, il faut donc en choisir les plus pertinentes. La grande innovation apportée par Google en 1998 est le tri des pages par ordre d'importance. Ce qui est frappant est que cet ordre correspond assez précisément aux attentes des utilisateurs.

Exemple 1.1. Par exemple, si vous vous intéressez à la programmation et vous faites chercher les mots-clés « C++ compiler », vous trouverez quelques millions de pages. Des pages importantes comme `gcc.gnu.org` se trouvent quelque part en tête du classement, ce qui est très raisonnable. Par contre, une petite page personnelle, où l'auteur mentionne qu'il ne connaît rien du C++ et n'arrive pas à compiler, ne figurera que vers la fin de la liste, ce qui est également raisonnable. Comment Google distingue-t-il les deux ?

Selon les informations fournies par l'entreprise elle-même (voir `www.google.com/corporate`), l'index de Google porte sur plus de 8 milliards d'adresses web (en avril 2007). Une bonne partie des informations répertoriées, pages web et documents annexes, changent fréquemment. Il est donc hors de question de les classer manuellement, par des êtres humains : ce serait trop coûteux, trop lent et jamais à jour. L'importance d'une page doit donc être déterminée de manière automatisée, par un algorithme. Comment est-ce possible ?

On ne va pas chercher à définir exactement ce qui est l'importance d'une page web. (Peut-il y en avoir une définition objective précise ?) Notre approche sera plus modeste : le mieux que l'on puisse espérer est que notre modèle dégage un résultat qui *approche* bien l'importance *ressentie* par les utilisateurs. L'idée est plutôt de considérer la popularité des pages web, c'est-à-dire leur fréquentation moyenne.

1.2. Matrice de transition. Dans la suite nous modélisons un surfeur aléatoire qui ne lit jamais rien mais qui clique au hasard. Ainsi ce n'est pas le contenu des pages web qui soit pris en compte, mais uniquement la structure du graphe formé par les pages et les liens entre elles. On renvoie à l'article [1] pour des arguments en faveur de ce modèle.

On considère des pages numérotées par $1,\dots,n$. Chaque page j émet un certain nombre ℓ_j de liens. On peut supposer $\ell_j \geq 1$; si jamais une page n'émet pas de liens on peut la faire pointer vers elle-même. Pour tout couple d'indices $i,j \in [\![1,n]\!]$ on définit un coefficient a_{ij} par

$$a_{ij} := \begin{cases} \frac{1}{\ell_j} & \text{si la page } j \text{ émet un lien vers la page } i, \\ 0 & \text{sinon.} \end{cases}$$

On interprète a_{ij} comme la probabilité d'aller de la page j à la page i, en suivant un des ℓ_j liens au hasard. La *marche aléatoire* associée consiste à se balader sur le graphe suivant les probabilités a_{ij}.

Selon sa définition, notre matrice $A = (a_{ij})$ vérifie

$$\begin{aligned} a_{ij} &\geq 0 \qquad \text{pour tout } i,j \text{ et} \\ \sum_i a_{ij} &= 1 \qquad \text{pour tout } j, \end{aligned}$$

ce que l'on appelle une *matrice stochastique*. À noter que la somme de chaque colonne vaut 1, mais on ne peut en général rien dire sur la somme dans une ligne.

1.3. Mesures invariantes. Supposons qu'un vecteur $x \in \mathbb{R}^n$ vérifie

$$x_j \geq 0 \quad \text{pour tout } j \text{ et} \quad \sum_j x_j = 1,$$

ce que l'on appelle un *vecteur stochastique* ou une *mesure de probabilité* sur les pages $1,\dots,n$: on interprète x_j comme la probabilité de se trouver sur la page j.

Effectuons un pas dans la marche aléatoire : avec probabilité x_j on démarre sur la page j, puis on suit le lien $j \to i$ avec probabilité a_{ij}. Ce chemin nous fait tomber sur la page i avec une probabilité $a_{ij}x_j$. Au total, la probabilité d'arriver sur la page i, par n'importe quel chemin, est la somme

$$y_i = \sum_j a_{ij}x_j.$$

Autrement dit, un pas dans la marche aléatoire correspond à l'application linéaire

$$T : \mathbb{R}^n \to \mathbb{R}^n, \quad x \mapsto y = Ax.$$

Exercice/M 1.2. Soit A une matrice stochastique. Majorer la norme de $y = Ax$ en fonction de la norme de x. Que peut-on en déduire pour les valeurs propres de A, ou plus précisément pour le rayon spectral de A ? Si x est un vecteur stochastique ($x_i \geq 0$, $|x| = 1$), vérifier que l'image $y = Ax$ est à nouveau stochastique.

Définition 1.3. Une mesure de probabilité μ vérifiant $\mu = T(\mu)$ est appelée une *mesure invariante* ou une *mesure d'équilibre*. En termes d'algèbre linéaire c'est un vecteur propre associé à la valeur propre 1. En termes d'analyse, μ est un point fixe de l'application T.

1.4. Le modèle utilisé par Google. Pour des raisons expliquées dans [1], Google utilise un modèle plus raffiné, dépendant d'un paramètre $c \in [0,1]$:

– Avec probabilité c, le surfeur abandonne la page actuelle et recommence sur une des n pages du web, choisie de manière équiprobable.
– Avec probabilité $1-c$, le surfeur suit un des liens de la page actuelle j, choisi de manière équiprobable parmi tous les ℓ_j liens émis. (C'est la marche aléatoire discutée ci-dessus.)

Ce modèle se formalise comme l'application

$$T : \mathbb{R}^n \to \mathbb{R}^n, \qquad T(\mu) = c\varepsilon + (1-c)A\mu$$

où A est la matrice stochastique définie en §1.2, et le vecteur stochastique $\varepsilon = (\frac{1}{n}, \dots, \frac{1}{n})$ correspond à l'équiprobabilité sur toutes les pages. Pour $c = 0$ on obtient donc exactement le modèle initial.

Proposition 1.4. *Soit A une matrice stochastique et $T : \mu \mapsto c\varepsilon + (1-c)A\mu$ avec une constante $c \in]0,1]$. Alors l'application T admet une unique mesure invariante $\mu = T(\mu)$. De plus, pour toute mesure initiale μ^0 la suite itérée $\mu^{n+1} = T(\mu^n)$ converge vers l'unique point fixe $\mu = T(\mu)$.*

C'est cette mesure invariante μ qui nous intéressera dans la suite et que l'on interprétera comme mesure d'importance. On la calculera d'ailleurs par la méthode itérative de la proposition.

Exercice/M 1.5. Prouver cette proposition en montrant que T est contractante pour la norme $|\mu| = \sum_i |\mu_i|$. (L'intérêt n'est pas de recopier la preuve de quelqu'un d'autre ; essayez plutôt de refaire la démonstration vous-mêmes et d'ainsi vérifier votre compréhension des arguments.) Quelle est la constante de contraction ? Majorer l'erreur $|\mu - \mu^n|$ en fonction de μ^n et μ^{n-1}. Que peut-on dire de la vitesse de convergence ?

Exercice/M 1.6. Est-ce une bonne idée de prendre $c = 1$ pour calculer le classement des pages web ? D'un autre coté, montrer par un exemple que la proposition est fausse pour $c = 0$.

Un bon choix de c se situe donc quelque part entre 0 et 1. En termes probabilistes, $\frac{1}{c}$ est le *nombre moyen* de liens suivis avant de recommencer sur une page aléatoire. En général on choisira la constante c positive mais proche de zéro. Par exemple, $c = 0,15$ correspond à suivre 7 liens en moyenne.

2. Implémentation en C++

Passons à l'implémentation de l'algorithme discuté ci-dessus. Le logiciel qui en résulte sera plutôt court (environ 40 lignes pour le calcul de μ plus quelques fonctions auxiliaires d'entrée-sortie, voir le fichier `graphe.cc`). Néanmoins il est important de préméditer la façon comment nous nous y prendrons.

2.1. Matrices creuses provenant de graphes. Rappelons qu'en réalité la matrice A est très grande : en 2004 Google affirmait que « le classement est effectué grâce à la résolution d'une équation de 500 millions de variables et de plus de 3 milliards de termes. » Comment est-ce possible ?

Certes, il est envisageable de stocker une matrice 1000×1000 sous le format usuel, c'est-à-dire dans un grand tableau de 10^6 coefficients indexés par $(i,j) \in [\![1,1000]\!]^2$. Ceci est hors de question pour une matrice $n \times n$ avec $n \approx 10^6$, voire $n \approx 10^8$.

Heureusement dans notre cas la plupart des coefficients vaut zéro, car une page n'émet que quelques dizaines de liens typiquement. Dans ce cas il suffit de stocker les coefficients non nuls, dont le nombre est d'ordre n et non n^2. Une telle matrice est appelée *creuse* (ou *sparse* en anglais).

☞ *Pour des applications réalistes il est donc nécessaire d'implémenter des structures et des méthodes spécialisées aux matrices creuses.* ✍

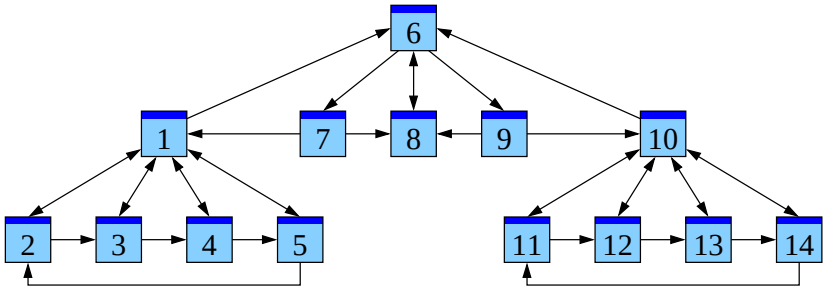
Exercice/P 2.1. Pour simplifier, nous allons spécialiser notre implémentation aux matrices creuses provenant de graphes. On peut implémenter la structure de graphe comme suit :

```
typedef unsigned int   Page;   // les pages forment les sommets du graphe
typedef vector<Page>   Liste;  // les liens forment les arrêtes (orientées)
typedef vector<Liste>  Graphe; // les pages et les liens forment le graphe
typedef vector<double> Mesure; // mesure de probabilité sur les sommets
```

Pour l'entrée-sortie il faut fixer une notation convenable. Dans le graphe ci-dessous la page `1` émet des liens vers les pages `2,3,4,5,6`. Ceci est noté simplement par `1:2,3,4,5,6;` comme dans le fichier `graphe1.mat`. Le graphe tout entier s'écrit comme

```
Graphe( 1:2,3,4,5,6; 2:1,3; 3:1,4; 4:1,5; 5:1,2; 6:7,8,9; 7:8,1; 8:6;
9:8,10; 10:6,11,12,13,14; 11:10,12; 12:10,13; 13:10,14; 14:10,11; )
```

L'entrée sous ce format-ci est déjà implémentée dans le fichier `graphe.cc`. Essayez de comprendre son fonctionnement, puis ajouter l'opérateur de sortie.



2.2. La méthode itérative.

Exercice/P 2.2. Implémenter une fonction `mesure_invariante` qui calcule la mesure invariante μ d'un graphe G. Comme motivé plus haut, on utilise un paramètre c, qui prend la valeur $0,15$ par défaut.

```
typedef vector<double> Mesure;
void mesure_invariante( const Graph& g, Mesure& m, const double c=0.15 );
```

Essayer de n'utiliser que les deux objets `g` et `m` ainsi qu'une copie de `m` durant sa mise à jour ; ni la matrice A ni le vecteur ε ne figureront explicitement dans l'implémentation. Nous rappelons que la construction explicite de la matrice dense T sera catastrophique pour toute application réaliste.

Exercice/P 2.3 (tests en taille minuscule)**.** Testez votre implémentation sur les deux exemples donnés dans les fichiers `graphe1.mat` et `graphe2.mat`. Avant les calculs, quels résultats prédirez-vous. Quels résultats obtient-on pour $c = 0$ et pour $c = 1$? Pour $c = 0,05,\ldots,0.95$? Ces résultats sont-ils plausibles ?

Exercice/P 2.4 (tests en taille moyenne)**.** Pour des exemples un peu plus grands et donc un peu plus réalistes, regardez les fichiers `graphe100.mat` et `graphe1000.mat`, qui donnent deux graphes à 100 et 1000 sommets respectivement. Peut-on deviner sans calcul quelles pages se dégageront comme les plus populaires ? Est-il envisageable de résoudre l'équation $T\mu = \mu$ par la méthode de Gauss ? En utilisant votre implémentation, trouvez les cinq pages les plus fréquentées (= populaires = importantes ?) ainsi que leur mesure calculée. Est-ce que le calcul s'effectue dans un temps raisonnable ?

Exercice/P 2.5 (extrapolation à une échelle réaliste)**.** Votre implémentation marchera-t-elle pour des graphes encore plus grands ? Quel temps d'exécution faudra-t-il environ et de quels paramètres dépend-il ? (Vous pouvez extrapoler ou, pour être plus précis, créer de graphes aléatoires via la fonction `random` implémentée dans `graphe.cc` puis mesurer le temps d'exécution.) Est-ce une méthode praticable à l'échelle « grandeur nature » de l'internet ?

Exercice/P 2.6 (expérience de manipulation)**.** Créez une copie du fichier `graphe1000.mat` nommée `graphe1000bis.mat`. Essayez de manipuler ce graphe pour que votre page préférée (disons la page 1) arrive en tête du classement. Dans une situation réaliste, vous ne pouvez évidemment pas changer les pages des autres, mais vous pouvez adapter les vôtres. La technique d'ajouter des pages et des liens à des fins stratégiques s'appelle « link farming » en anglais. Comment le faire de manière bien camouflée ? Comment l'entreprise Google peut-elle réagir à ces tentatives de manipulations ? De manière générale, quelles autres stratégies voyez-vous pour améliorer le classement de votre page préférée ?