Décomposition d'une fraction rationnelle en éléments simples

Xavier Dupré
http://www.xavierdupre.fr/

3 mai 2013

Résumé

Ce document décrit une façon de décomposer une fraction rationnelle en élément simple.

Il existe une méthode systématique qui permet de décomposer une fraction rationelle en éléments simples : la première étape est d'écrire la forme a priori de la décomposition, puis, après mise sous même dénominateur, d'identifier les coefficients de la décomposition. Cette méthode revient à résoudre un système linéaire de n inconnues à n équations. Les n inconnues sont les n coefficients de la décomposition de la fraction en éléments simples. Un théorème affirme que cette décomposition existe et est unique ce qui nous assure que le système linéaire à résoudre admet nécessairement une solution unique.

Les chapitres qui suivront ont pour de présenter un algorithme de décomposition des fractions rationnelles en éléments simples.

0.1 Opérations sur les polynômes

La multiplication de polynômes sera la seule opération sur les polynômes utilisée par l'algorithme. Néanmoins, l'addition sera également présentée.

Un polynôme P est défini par :

$$\begin{cases} \text{un degré } d_P \\ \text{un tableau de coefficients } \{a_0, \cdots, a_{d_P}\} \end{cases}$$

D'après ces notations :

$$P(x) = \sum_{i=0}^{d_P} a_i x^i$$

Par définition, un polynôme est constant si et seulement si $d_P = 0$, par convention, le polynôme nul aura pour degré 0.

0.1.1 L'addition

On considère deux polynômes :

P de degré d_P et de coefficients $\{a_0, \cdots, a_{d_P}\}$ et Q de degré d_Q et de coefficients $\{b_0, \cdots, b_{d_Q}\}$.

Soit $R = P + Q$, on sait que $d_R \leqslant \max(d_P, d_Q)$, et es coefficients de R sont $\{c_0, \cdots, c_{d_R}\}$.

$$\forall i \in \{0, \cdots, d_R\},\ c_i = \begin{cases} a_i + b_i \text{ si } i \leqslant \min\{d_P, d_Q\} \\ a_i \text{ si } i \leqslant d_P \text{ et } i > d_Q \\ b_i \text{ si } i > d_P \text{ et } i \leqslant d_Q \end{cases}$$

La valeur précise de d_R est donnée par :

$$d_R = \begin{cases} \max\{d_P, d_Q\} \text{ si } d_P \neq d_Q \\ \inf\{i \mid i \in \{0, \cdots, d_R\} \text{ et } \forall k > i,\ c_k = 0\} \text{ si } d_P \neq d_Q \end{cases}$$

On prend cette précaution pour s'assurer que le coefficient du terme de plus haut degré est non nul. Dans le programme, cette addition n'est pas utilisée.

0.1.2 La multiplication

On considère deux polynômes P de degré d_P et de coefficients $\{a_0, \cdots, a_{d_P}\}$ et Q de degré d_Q et de coefficients $\{b_0, \cdots, b_{d_Q}\}$.

$$P = \sum_{i=0}^{d_P} a_i x^i$$
$$Q = \sum_{i=0}^{d_Q} b_i x^i$$

Par conséquent, soit R le polynôme vérifiant $R = PQ$, de degré d_R et de coefficients $\{c_0, \cdots, c_{d_R}\}$, alors :

$$d_R = \begin{cases} 0 \text{ si } P = 0 ou Q = 0 \\ d_P + d_Q \text{ sinon} \end{cases}$$

Dans le cas où $P \neq 0$ et $Q \neq 0$:

$$R = \left(\sum_{i=0}^{d_P} a_i x^i\right)\left(\sum_{i=0}^{d_Q} b_i x^i\right)$$
$$R = \sum_{i=0}^{d_R} \sum_{\substack{k+l=i \\ k \leqslant d_R \\ l \leqslant d_Q}} a_k\, b_l\, x^i$$
$$R = \sum_{i=0}^{d_R} \sum_{k=\max\{0, i-d_Q\}}^{\min\{i, d_P\}} a_k\, b_{i-k}\, x^i$$

0.2 Opérations sur les matrices

L'algorithme fait intervenir deux opérations sur les matrices : la multiplication et l'inversion.
Une matrice M est représentée de manière informatique par :

$$\begin{cases} \text{un nombre de lignes } L_M \\ \text{un nombre de colonnes } C_M \\ \text{un tableau de coefficients } (a_{ij})_{\substack{1\leqslant i\leqslant L_M \\ 1\leqslant j\leqslant C_M}} \end{cases}$$

0.2.1 La multiplication

On considère deux matrices : M de coefficients $(a_{ij})_{\substack{1\leqslant i\leqslant L_M \\ 1\leqslant j\leqslant C_M}}$ et N de coefficients $(b_{ij})_{\substack{1\leqslant i\leqslant L_N \\ 1\leqslant j\leqslant C_N}}$.
On peut effectuer le produit $K = MN$ si $C_M = L_N$:

$$\begin{cases} K = (c_{ij})_{\substack{1\leqslant i\leqslant L_M \\ 1\leqslant j\leqslant C_N}} \\ \forall i \in \{1, \cdots, L_M\}, \forall j \in \{1, \cdots, C_N\}, \ c_{ij} = \sum_{k=0}^{C_M} a_{ik} b_{kj} \end{cases}$$

0.2.2 Multiplication et opérations sur les lignes et colonnes

Les multiplications entre matrices permettent à l'aide de matrices simples de traduire des opérations sur les lignes et les colonnes.
Exemple :

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & -5 \\ 2 & 0 & 1 \end{pmatrix} \qquad \underbrace{\begin{pmatrix} L_1 \\ L_2 \\ L_3 \end{pmatrix}}_{\substack{\text{matrice quelconque} \\ \text{composée de 3 lignes}}} \qquad = \begin{pmatrix} L_1 \\ L_2 - 5L_3 \\ L_3 + 2L_1 \end{pmatrix}$$

Dans le cas général, multiplier à gauche par une matrice carrée $M \in \mathbf{M}_n,\ M = (a_{ij})_{\substack{1\leqslant i\leqslant n \\ 1\leqslant j\leqslant n}}$ revient à faire des opérations sur les lignes sur la matrice de droite :

$$(a_{ij}) \begin{pmatrix} L_1 \\ L_2 \\ \vdots \\ L_n \end{pmatrix} = \begin{pmatrix} \sum_{j=1}^{n} a_{1j} L_j \\ \sum_{j=1}^{n} a_{2j} L_j \\ \vdots \\ \sum_{j=1}^{n} a_{nj} L_j \end{pmatrix}$$

De même, multiplier à droite par une matrice carrée $M \in \mathbf{M}_n, \quad M = (a_{ij})_{\substack{1\leqslant i\leqslant n \\ 1\leqslant j\leqslant n}}$ revient à faire des opérations sur les colonnes sur la matrice de gauche :

$$\begin{pmatrix} C_1 & C_2 & \dots & C_n \end{pmatrix} (a_{ij}) = \begin{pmatrix} \sum_{i=1}^{n} a_{i1}C_i & \sum_{i=1}^{n} a_{i2}C_i & \dots & \sum_{i=1}^{n} a_{in}C_i \end{pmatrix}$$

Ces précisions sont importantes en terme de coût algorithime . En effet, pour effectuer le produit de matrices suivant :

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & -5 \\ 2 & 0 & 1 \end{pmatrix} \qquad \underbrace{\begin{pmatrix} L_1 \\ L_2 \\ L_3 \end{pmatrix}}_{\substack{\text{matrice quelconque} \\ \text{composée de 3 lignes}}}$$

Il est plus judicieux de faire des opérations sur les lignes plutôt que d'effectuer le produit matriciel complet. Le coût d'un produit matriciel est en $O\left(n^3\right)$ ceci signifie que pour deux matrices carrées d'ordre n, le nombre d'opérations (au sens informatique) est un multiple constant (indépendant des matrices choisies) de n^3, ou 3 boucles imbriquées de n itérations. Or, effectuer les opérations suivantes sur les lignes :

$$\begin{pmatrix} L_1 \\ L_2 - 5L_3 \\ L_3 + 2L_1 \end{pmatrix}$$

est beaucoup moins coûteux. Dans ce cas, chaque ligne de la matrice résultat est une somme coefficientée de deux autres lignes. Le coût de ce produit sera donc en $O\left(n^2\right)$. Ce sera justement le seul cas rencontré lors du pivot de Gauss.

0.2.3 Matrice de passage du pivot de Gauss

Lors du calcul de l'inverse, il est nécessaire de calculer la matrice de passage qui permet de passer de la matrice initiale à la matrice triangulaire supérieure obtenue par pivot de Gauss.

$$\underbrace{\begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -3 & -5 & 1 \end{pmatrix}}_{\text{matrice de passage}} \begin{pmatrix} L_1 \\ L_2 \\ L_3 \end{pmatrix} = \underbrace{\begin{pmatrix} L_1 \\ L_2 - 2L_1 \\ L_3 - 3L_1 - 5L_2 \end{pmatrix}}_{\text{obtenue directement par pivot de Gauss}} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a_{22} & a_{23} \\ 0 & 0 & a_{33} \end{pmatrix}$$

On peut écrire également :

$$\begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ -3 & -5 & 1 \end{pmatrix} \underbrace{\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}}_{\text{matrice identité}} \begin{pmatrix} L_1 \\ L_2 \\ L_3 \end{pmatrix} = \begin{pmatrix} L_1 \\ L_2 - 2L_1 \\ L_3 - 3L_1 - 5L_2 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a_{22} & a_{23} \\ 0 & 0 & a_{33} \end{pmatrix}$$

Si on note la matrice identité :

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} I_1 \\ I_2 \\ I_3 \end{pmatrix}$$

On peut écrire le résultat précédent sous la forme :

$$\begin{pmatrix} I_1 & & \\ I_2 - 2I_1 & & \\ I_3 - 3I_1 - 5I_2 & & \end{pmatrix} \begin{pmatrix} L_1 \\ L_2 \\ L_3 \end{pmatrix} = \begin{pmatrix} L_1 & & \\ L_2 - 2L_1 & & \\ L_3 - 3L_1 - 5L_2 & & \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a_{22} & a_{23} \\ 0 & 0 & a_{33} \end{pmatrix}$$

Le pivot de Gauss consiste à appliquer une série d'opérations sur les lignes à partir de la matrice initiale pour la rendre triangulaire supérieure. La matrice de passage s'obtient donc en effectuant exactement les mêmes opérations sur les lignes à partir de la matrice identité.

0.2.4 L'inversion

L'inversion d'une matrice utilise le pivot de Gauss. Cette méthode permet de rendre les matrices triangulaires supérieures. C'est également cette méthode qui est utilisée pour calculer son déterminant.

La méthode utilisée se décrit simplement, nous supposons que nous avons une matrice de la forme :

$$\begin{pmatrix} L_1 \\ L_2 \\ L_3 \\ L_4 \\ L_5 \end{pmatrix} = \begin{pmatrix} a_{11} & * & * & * & * \\ 0 & a_{22} & * & * & * \\ 0 & 0 & a_{33} & a_{34} & * \\ 0 & 0 & a_{43} & a_{44} & * \\ 0 & 0 & a_{53} & a_{54} & * \end{pmatrix}$$

les * désignent un coefficient quelconque)

L'objectif est de rendre nul les coefficients a_{43} et a_{53}, pour ce faire, on effectue les opérations sur les lignes comme si on résolvait un système linéaire, ces opérations correspondent également à un produit de matrice :

$$\begin{aligned} L_4 &\leftarrow L_4 - L_3 \frac{a_{43}}{a_{33}} \\ L_5 &\leftarrow L_5 - L_3 \frac{a_{53}}{a_{33}} \end{aligned}$$

On suppose que $a_{33} \neq 0$:

$$\underbrace{\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & -a_{43} & 1 & 0 \\ 0 & 0 & -a_{53} & 0 & 1 \end{pmatrix}}_{P_3} \underbrace{\begin{pmatrix} a_{11} & * & * & * & * \\ 0 & a_{22} & * & * & * \\ 0 & 0 & a_{33} & a_{34} & * \\ 0 & 0 & a_{43} & a_{44} & * \\ 0 & 0 & a_{53} & a_{54} & * \end{pmatrix}}_{M_3} = \underbrace{\begin{pmatrix} a_{11} & * & * & * & * \\ 0 & a_{22} & * & * & * \\ 0 & 0 & a_{33} & a_{34} & * \\ 0 & 0 & 0 & a_{44} - \frac{a_{43}}{a_{33}} a_{34} & * \\ 0 & 0 & 0 & a_{54} - \frac{a_{53}}{a_{33}} a_{44} & * \end{pmatrix}}_{M_4}$$

Dans le cas où $a_{33} = 0$ et que $a_{43} \neq 0$, on fait d'abord : $L_3 \leftarrow L_4$ puis on continue le pivot de Gauss :

$$\underbrace{\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & -a_{43} & 1 & 0 \\ 0 & 0 & -a_{53} & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}}_{P_3} \underbrace{\begin{pmatrix} a_{11} & * & * & * & * \\ 0 & a_{22} & * & * & * \\ 0 & 0 & 0 & a_{34} & * \\ 0 & 0 & a_{43} & a_{44} & * \\ 0 & 0 & a_{53} & a_{54} & * \end{pmatrix}}_{M_3} =$$

$$\underbrace{\begin{pmatrix} a_{11} & * & * & * & * \\ 0 & a_{22} & * & * & * \\ 0 & 0 & a_{43} & a_{43}+a_{44} & * \\ 0 & 0 & 0 & -a_{34} & * \\ 0 & 0 & 0 & a_{54} - \frac{a_{53}}{a_{43}} a_{34} & * \end{pmatrix}}_{M_4}$$

Dans le cas général, on suppose que M est une matrice carrée d'ordre n, on suppose que $M_1 = M$ et la suite (P_i) est construite de telle sorte que :

$$\begin{aligned} P_i M_i &= M_{i+1} \\ M_i &= \left(a_i^{kl}\right)_{1 \leqslant k,l \leqslant n} \\ \forall\, (k,l) \text{ tel que } i > k > l \geqslant 1,\ a_i^{kl} &= 0 \end{aligned}$$

On obtient finalement que $(P_n P_{n-1} \dots P_1)\, M = N$ avec N matrice triangulaire supérieure. Comme la matrice $P_n P_{n-1} \dots P_1$ est inversibles (c'est un produit de matrice inversibles), la matrice M est inversible si la matrice N l'est aussi et N est inversible si tous ses coefficients sur la diagonale sont non nuls.

En effet, on note $N = \left(b^{kl}\right)_{1 \leqslant k,l \leqslant n}$, N est triangulaire supérieure donc :

$$\begin{aligned} \det N &= \prod_{i=1}^{n} b^{ii} \\ &= \underbrace{\det\left(P_n P_{n-1} \dots P_1\right)}_{=1 \text{ par construction}} \times \det M \\ &= \det M \end{aligned}$$

On suppose maintenant que la matrice M est inversible, tous les coefficients diagonaux de N sont donc non nuls. Dans cette première phase, nous avons, dans l'ordre croissant des lignes, rempli la matrice M de 0 en dessous de sa diagonale, il est alors possible en appliquant le même pivot de Gauss mais cette fois dans l'ordre décroissant des lignes de remplir la matrice de 0 au-dessus de sa diagonale.

Il existe donc une suite de matrice (Q_i) tel que $(Q_n Q_{n-1} \dots Q_1)\, N = D_n$ où D_n est une matrice diagonale. De plus, on peut choisir les matrices Q_i de telle sorte que $D_n = I_n$ où I_n est la matrice identité. Par exemple :

$$\underbrace{\begin{pmatrix} 1 & 0 & -b_{13} & 0 & 0 \\ 0 & 1 & -b_{23} & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{b_{33}} & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}}_{Q_3} \underbrace{\begin{pmatrix} b_{11} & b_{12} & b_{13} & 0 & 0 \\ 0 & b_{22} & b_{23} & 0 & 0 \\ 0 & 0 & b_{33} & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}}_{N_3} =$$

$$\underbrace{\begin{pmatrix} b_{11} & b_{12} & 0 & 0 & 0 \\ 0 & b_{22} & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}}_{N_4}$$

Donc :

$$(Q_n Q_{n-1} \dots Q_1)(P_n P_{n-1} \dots P_1) M = I_n$$

D'où :

$$\boxed{M^{-1} = (Q_n Q_{n-1} \dots Q_1)(P_n P_{n-1} \dots P_1)}$$

0.3 Application aux fractions rationnelles

Trouver tous les coefficients de la décomposition a priori d'une fonction rationnelle revient à résoudre un système linéaire.

Soit P et Q deux polynômes à coefficients réels, Q se décompose en :

$$Q = \alpha \prod_{i=1}^{s} (X - a_i)^{d_i} \prod_{i=1}^{t} \left(X^2 + p_i X + q_i\right)^{d'_i}$$

La fraction $\frac{P}{Q}$ se décompose donc en :

$$\frac{P}{Q} = E + \sum_{i=1}^{s} \sum_{k=1}^{d_i} \frac{\alpha_i^k}{(X - a_i)^k} + \sum_{i=1}^{t} \sum_{k=1}^{d'_i} \frac{\beta_i^k X + \gamma_i^k}{\left(X^2 + p_i X + q_i\right)^k}$$

où E est la partie entière.

On met tout sous le même dénominateur :

$$\begin{aligned} P = \quad & EQ \\ + \quad & \alpha \sum_{i=1}^{s} \left(\prod_{j \neq i} (X + a_j)^{d_j} \prod_{j} \left(X^2 + p_i X + q_i\right)^{d'_j} \right) \left(\sum_{k=1}^{d_i} \alpha_i^k (X - a_i)^{d_i - k} \right) \\ + \quad & \alpha \sum_{i=1}^{t} \left(\prod_{j} (X + a_j)^{d_j} \prod_{j \neq i} \left(X^2 + p_i X + q_i\right)^{d'_j} \right) \left(\sum_{k=1}^{d'_i} \left(\beta_i^k X + \gamma_i^k\right) \left(X^2 + p_i X + q_i\right)^{d'_i - k} \right) \end{aligned}$$

Il suffit d'identifier ces deux polynômes, ce qui revient à résoudre un système linéaire. Pour que l'algorithme marche, il faut néanmoins connaître la factorisation en éléments simples du dénominateur.

0.3.1 Construction du système linéaire

L'illustration par un exemple permet de mieux comprendre l'algorithme qu'une description théorique utilisant une multitude d'indice.

Nous allons donc décortiquer chaque étape de l'algorithme sur la fraction rationnelle :

$$\frac{P}{Q} = \frac{X^9}{(X+1)^3 (X^2+X+2)^2} \tag{1}$$

0.3.2 Numérotation des coefficients

La partie entière est de degré 2 (=9-7).

La forme a priori de la décomposition de la fraction rationnelle est :

$$\frac{P}{Q} = a_1 + a_2 X + a_3 X^2 + \frac{a_4}{X+1} + \frac{a_5}{(X+1)^2} + \frac{a_6}{(X+1)^3} + \frac{a_7 + a_8 X}{X^2+X+2} + \frac{a_9 + a_{10} X}{(X^2+X+2)^2}$$

La numérotation suit le sens croissant des puissances en commençant par la partie entière.

0.3.3 Construction du système linéaire

En mettant tout sous le même dénominateur, on obtient :

$$P = a_1 Q + a_2 X Q + a_3 X^2 Q + \frac{a_4 Q}{X+1} + \frac{a_5 Q}{(X+1)^2} + \frac{a_6 Q}{(X+1)^3} + \frac{(a_7 + a_8 X)\, Q}{X^2+X+2} + \frac{(a_9 + a_{10} X)\, Q}{(X^2+X+2)^2}$$

On construit la matrice suivante :

$$\begin{pmatrix} & \\ \mathbf{a}_1\times & Q \\ \mathbf{a}_2\times & XQ \\ \mathbf{a}_3\times & X^2Q \\ \mathbf{a}_4\times & Q/(X+1) \\ \mathbf{a}_5\times & Q/(X+1)^2 \\ \mathbf{a}_6\times & Q/(X+1)^3 \\ \mathbf{a}_7\times & Q/\left(X^2+X+2\right) \\ \mathbf{a}_8\times & XQ/\left(X^2+X+2\right) \\ \mathbf{a}_9\times & Q/\left(X^2+X+2\right)^2 \\ \mathbf{a}_{10}\times & XQ/\left(X^2+X+2\right)^2 \end{pmatrix} = \underbrace{\begin{pmatrix} & \mathbf{X}^0 & \mathbf{X}^1 & \mathbf{X}^2 & \mathbf{X}^3 & \mathbf{X}^4 & \mathbf{X}^5 & \mathbf{X}^6 & \mathbf{X}^7 & \mathbf{X}^8 & \mathbf{X}^9 \\ & \times & \times & \times & \times & \times & \times & \times & \times & \times & \times \\ \mathbf{a}_1\times & 4 & 16 & 29 & 33 & 26 & 14 & 5 & 1 & & \\ \mathbf{a}_2\times & & 4 & 16 & 29 & 33 & 26 & 14 & 5 & 1 & \\ \mathbf{a}_3\times & & & 4 & 16 & 29 & 33 & 26 & 14 & 5 & 1 \\ \mathbf{a}_4\times & 4 & 12 & 17 & 16 & 10 & 4 & 1 & & & \\ \mathbf{a}_5\times & 4 & 8 & 9 & 7 & 3 & 1 & & & & \\ \mathbf{a}_6\times & 4 & 4 & 5 & 2 & 1 & & & & & \\ \mathbf{a}_7\times & 1 & 4 & 6 & 4 & 1 & & & & & \\ \mathbf{a}_8\times & & 1 & 4 & 6 & 4 & 1 & & & & \\ \mathbf{a}_9\times & 1 & 3 & 3 & 1 & & & & & & \\ \mathbf{a}_{10}\times & & 1 & 3 & 3 & 1 & & & & & \end{pmatrix}}_{=M \text{ (10 lignes et 10 colonnes)}}$$

On peut écrire ce système sous la forme : (M^T est la transposée de M)

$$\begin{pmatrix} a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & a_9 & a_{10} \end{pmatrix} M \begin{pmatrix} X^0 \\ X^1 \\ X^2 \\ X^3 \\ X^4 \\ X^5 \\ X^6 \\ X^7 \\ X^8 \\ X^9 \end{pmatrix} = \underbrace{\begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}^T \begin{pmatrix} X^0 \\ X^1 \\ X^2 \\ X^3 \\ X^4 \\ X^5 \\ X^6 \\ X^7 \\ X^8 \\ X^9 \end{pmatrix}}_{=P \text{ numérateur de la fraction}}$$

Donc par identification des deux polynômes, on obtient :

$$\begin{pmatrix} a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & a_9 & a_{10} \end{pmatrix} M = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

D'où :

$$\underbrace{M^T \begin{pmatrix} a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & a_9 & a_{10} \end{pmatrix}^T = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}^T}_{\text{c'est le système linéaire cherché}}$$

0.3.4 Résolution

D'après le système précédent, on déduit que :

$$\begin{pmatrix} a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & a_9 & a_{10} \end{pmatrix}^T = \left(M^T\right)^{-1} \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}^T$$

La décomposition de la fraction rationnelle est donc :

$$\frac{P}{Q} = 11 - 5X + X^2 - \frac{6,6875}{X+1} + \frac{2}{(X+1)^2} - \frac{0,25}{(X+1)^3} - \frac{15 + 4,3125X}{X^2 + X + 2} + \frac{5,75 + 5,625X}{(X^2 + X + 2)^2}$$

0.4 Un exemple de programme informatique

La fraction (1) peut être décrite comme dans la figure 1, le résultat est imprimé dans la figure 2.

Quelques différences peuvent être observées entre la solution fournies par le programme et la véritable solution, ceci est dû aux approximations de calcul des ordinateurs qui code les nombres réels avec environ 15 chiffres de précision. Certains termes nuls peuvent alors être non nuls mais très petits par rapport aux autres coefficients.

numerateur degre 9
d0 0
d1 0
d2 0
d3 0
d4 0
d5 0
d6 0
d7 0
d8 0
d9 1
.....denominateur produit 2
denominateur puissance 3
denominateur degre 1
d0 1
d1 1
denominateur puissance 2
denominateur degre 2
d0 2
d1 1
d2 1

Figure 1 : *Description d'une fraction rationnelle*

fraction a decomposer :	
numerateur :	$X\hat{}9$
nombre de terme au denominateur :	2
terme 1 :	$(1+X)\hat{}\,3$
terme 2 :	$(2+X+X\hat{}2)\hat{}\,2$
decomposition en elements simples	
partie entiere :	$11-5*X+1*X\hat{}2$
terme en	$(1+X) : -6.6875$
terme en	$(1+X)\hat{}\,2 : 2$
terme en	$(1+X)\hat{}\,3 : -0.25$
terme en	$(2+X+X\hat{}2) : -15-4.3125*X$
terme en	$(2+X+X\hat{}2)\hat{}\,2 : 5.75+5.625*X$

Figure 2 : *Résultat de la décomposition*

0.5 Le programme informatique

0.5.1 Tableaux

Etant que les tableaux informatique de dimension n sont indicés de 0 à $n-1$ inclus, un polynôme devient :

$$\begin{cases} a\,[50] \text{ est un tableau appelé } a \text{ à 50 éléments numérotés de 0 à 49} \\ a[i] \text{ représente l'élément d'indice } i \\ \text{un polynôme s'écrit } P = \sum_{i=0}^{n} a\,[i]\,X^i \end{cases}$$

Et une matrice :

$$\begin{cases} a\,[40]\,[50] \text{ est un tableau appelé } a \text{ de 40 lignes et 50 colonnes} \\ a[i][j] \text{ représente l'élément placé sur la ligne } i \text{ et la colonne } j \\ \text{le dernier élément est l'élément } a\,[39]\,[49] \\ \text{une matrice s'écrit } M = \begin{pmatrix} a\,[0]\,[0] & \cdots & a\,[0]\,[n-1] \\ \vdots & \ldots & \vdots \\ a\,[m-1]\,[0] & \cdots & a\,[m-1]\,[n-1] \end{pmatrix} = (a\,[i]\,[j])_{\substack{0 \leqslant i \leqslant m-1 \\ 0 \leqslant j \leqslant n-1}} \end{cases}$$

0.5.2 Structures possibles

Deux types spéciaux peuvent être créés pour contenir matrices et polynômes.

Structure d'une matrice :

```
typedef struct {
 double *C;
 int Ligne;
 int Colonne;
} Matrice;
```

Structure d'un polynôme :

```
typedef struct {
 double *C;
 int Degre;
} Polynome;
```

0.5.3 Expression de la décomposition à l'aide de ces structures

$$\frac{P}{Q} = partie\_entiere + \sum_{n=0}^{facteur-1} \sum_{k=0}^{puissance[n]-1} \frac{resultat[n][k]}{(element[n])^{k+1}}$$

$$\frac{P}{Q} = \sum_{i=0}^{partie\_entiere.Degre} partie\_entiere.C\,[i] \times X^i +$$

$$\sum_{n=0}^{facteur-1} \sum_{k=0}^{puissance[n]-1} \frac{\sum_{i=0}^{resultat[n][k].Degre} resultat[n][k].C\,[i] \times X^i}{\left(\sum_{i=0}^{element[n].Degre} element[n].C\,[i] \times X^i\right)^{k+1}}$$

Correction

```
#include "stdio.h"
#include "string.h"


//////////////////////////////////////////////////////////////////////

typedef struct {
  double *C ;
  int Ligne ;
  int Colonne ;
} Matrice ;

typedef struct {
  double *C ;
  int Degre ;
} Polynome ;

//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////

// fonction matricielle

// crée une matrice
Matrice matrice_new (int ligne, int colonne) ;
// crée une matrice carrée identité
Matrice matrice_new_identite (int ligne) ;
// libère une matrice
void matrice_free (Matrice *mat) ;
// change la valeur d'un coefficient
void matrice_change (Matrice *mat, int i, int j, double val) ;
// retourne la valeur d'un coefficient
double matrice_donne (Matrice *mat, int i, int j) ;
// fait une copie de matrice
Matrice matrice_copie (Matrice *mat) ;
```

```
// produit de matrice
Matrice matrice_produit (Matrice *mat1, Matrice *mat2) ;
// écrire une matrice
char *matrice_ecrit (Matrice *mat) ;

// réalise le pivot de Gauss
// et retourne la matrice de passage
Matrice matrice_pivot_gauss (Matrice *mat) ;
// réalise le pivot de Gauss de haut en bas
// et retourne la matrice de passage
Matrice matrice_pivot_gauss_inverse (Matrice *mat) ;
// utilise les deux fonctions précédentes pour calculer l'inverse d'une matrice
// la matrice mat rste inchangée
Matrice matrice_inverse (Matrice *mat) ;


////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////

// crée un polynôme
Polynome polynome_new (int degre) ;
// libère un polynôme
void polynome_free (Polynome *p) ;
// change la valeur d'un coefficient (celui devant le monôme de degré n)
void polynome_change (Polynome *p, int n, double val) ;
// retourne la valeur d'un coefficient
double polynome_donne (Polynome *p, int n) ;
// retourne une chaîne
char *polynome_ecrit (Polynome *p) ;
// multiplie deux polynôme
Polynome polynome_produit (Polynome *p1, Polynome *p2) ;

// effectue la decomposition d'une fraction rationnelle
// resultat contient les resultats
// première dimension : facteur
// deuxième dimension : puissance [n]
// résultat doit déjà être alloué
void polynome_decomposition (Polynome *numerateur, int facteur, Polynome *element, int *puissance,
                    Polynome partie_entiere,
                    Polynome **resultat) ;

////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////

Polynome fichier_lecture_polynome (FILE *f)
{
  char buffer [200] ;
  Polynome r ;
  fscanf (f, "%s %s %d", buffer, buffer, &r.Degre) ;
  r = polynome_new (r.Degre) ;
  double d ;
  int n ;
  for (n = 0 ; n <= r.Degre ; n++) {
    fscanf (f, "%s %lf", buffer, &d) ;
    polynome_change (&r, n, d) ;
```

```
  }
  return r ;
}

// pour écrire dans un fichier
FILE *Sortie = NULL ;

////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////

void main (int argc, char *param [])
{
  char *c = "fraction.txt" ;
  char *res = "decompo.txt" ;
  if (argc > 1) c = param [1] ;
  if (argc > 2) res = param [2] ;

  // on lit le fichier
  FILE *f = fopen (c, "r") ;
  if (f == NULL) printf ("impossible de lire le fichier %s\n", c) ;
  else {
    char buffer [200] ;

    // lecture du numérateur
    Polynome Numerateur = fichier_lecture_polynome (f) ;

    // lecture du nombre de polynôme au dénominateur
    int Denominateur ;
    fscanf (f, "%s %s %d", buffer, buffer, &Denominateur) ;

    int *Puissance = new int [Denominateur] ;
    Polynome *Element = new Polynome [Denominateur] ;

    // lecture de ces polynôme
    int n ;
    for (n = 0 ; n < Denominateur ; n++) {
      fscanf (f, "%s %s %d", buffer, buffer, &Puissance [n]) ;
      Element [n] = fichier_lecture_polynome (f) ;
    }

    fclose (f) ;

    // vérification de la lecture
    char *s = polynome_ecrit (&Numerateur) ;
    printf ("numerateur  : %s\n",s) ;
    delete [] s ;
    printf ("\nnombre de terme au denominateur : %d\n",Denominateur) ;
    for (n = 0 ; n < Denominateur ; n++) {
      s = polynome_ecrit (&(Element [n])) ;
      printf ("terme %d puissance %d : %s\n", n+1, Puissance [n], s) ;
      delete [] s ;
    }

    // création de la structure de résultat
    // degré total du dénominateur
    int degre = 0 ;
    for (n = 0 ; n < Denominateur ; n++)
```

```
  degre += Puissance [n] * Element [n].Degre ;

// degré de la partie entière
int degre_ent = Numerateur.Degre - degre ;
if (degre_ent < 0) degre_ent = -1 ; // pas de partie entère

int k ;
Polynome PartieEntiere = polynome_new (degre_ent) ;
Polynome **Resultat = new Polynome* [Denominateur] ;
for (n = 0 ; n < Denominateur ; n++) {
  Resultat [n] = new Polynome [Puissance [n]] ;
  for (k = 0 ; k < Puissance [n] ; k++)
    Resultat [n][k] = polynome_new (Element [n].Degre - 1) ;
      // les coefficients des éléments
      // simple sont des polynômes de degré 0 (constante) pour un élément simple de 1ř espèce
      // et de degré 1 pour des éléments simples de 2ř espèce
}


// traitement de la décomposition
FILE *g = fopen (res, "w") ;
Sortie = g ;

polynome_decomposition (&Numerateur,
                        Denominateur,
                        Element,
                        Puissance,
                        PartieEntiere,
                        Resultat) ;


// écriture du résultat


// précision sur la fraction à décomposer

s = polynome_ecrit (&Numerateur) ;
fprintf (g, "fraction à décomposer : \n\n") ;
fprintf (g, "numerateur  : %s\n",s) ;
delete [] s ;
fprintf (g, "\nnombre de terme au denominateur : %d\n",Denominateur) ;
for (n = 0 ; n < Denominateur ; n++) {
  s = polynome_ecrit (&(Element [n])) ;
  if (Puissance [n] > 1) fprintf (g, "terme %d : (%s)^%d\n", n+1, s, Puissance [n]) ;
  else fprintf (g, "terme %d : (%s)\n", n+1, s) ;
  delete [] s ;
}
fprintf (g, "\n\ndécomposition en éléments simples \n\n") ;

// partie entière
s = polynome_ecrit (&PartieEntiere) ;
fprintf (g, "partie entière : \t\t%s\n",s) ;
delete [] s ;

// les autres termes
for (n = 0 ; n < Denominateur ; n++) {
  for (k = 0 ; k < Puissance [n] ; k++) {
    s = polynome_ecrit (&(Element [n])) ;
    if (k == 0) fprintf (g, "terme en (%s) : \t", s) ;
```

```
        else fprintf (g, "terme en (%s)^%d : \t", s, k+1) ;
        delete [] s ;

        s = polynome_ecrit (&(Resultat [n][k])) ;
        fprintf (g, "%s\n",s) ;
        delete [] s ;
      }
    }



    // fin du programme
    Sortie = NULL ;
    fclose (g) ;

    polynome_free (&Numerateur) ;
    for (n = 0 ; n < Denominateur ; n++) polynome_free (&(Element [n])) ;
    delete [] Element ;

    polynome_free (&PartieEntiere) ;
    for (n = 0 ; n < Denominateur ; n++) {
      for (k = 0 ; k < Puissance [n] ; k++)
        polynome_free (&(Resultat [n][k])) ;
      delete [] Resultat [n] ;
    }
    delete [] Puissance ;
    delete [] Resultat ;

    // fin
//    getchar () ;
  }
}

////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////

Matrice matrice_new (int ligne, int colonne)
{
  if (ligne * colonne > 0) {
    Matrice r ;
    r.Ligne = ligne ;
    r.Colonne = colonne ;
    r.C = new double [r.Ligne * r.Colonne] ;
    return r ;
  }
  else {
    Matrice r ;
    r.Ligne = 0 ;
    r.Colonne = 0 ;
    return r ;
  }
}

Matrice matrice_new_identite (int ligne)
{
  Matrice r = matrice_new (ligne, ligne) ;
  int i,j ;
```

```
  for (i = 0 ; i < r.Ligne ; i++)
    for (j = 0 ; j < r.Colonne ; j++)
      if (i == j) matrice_change (&r, i,j, 1) ;
      else matrice_change (&r, i,j, 0) ;
  return r ;
}

void matrice_free (Matrice *mat)
{
  if (mat->Ligne * mat->Colonne > 0) {
    delete [] mat->C ;
    mat->Ligne = 0 ;
    mat->Colonne = 0 ;
  }
}

void matrice_change (Matrice *mat, int i, int j, double val)
{
  if ((i >= 0) && (i < mat->Ligne) && (j >= 0) && (j < mat->Colonne))
    mat->C [i * mat->Colonne + j] = val ;
  else printf ("erreur matrice change %d %d\n",i,j) ;
}
double matrice_donne (Matrice *mat, int i, int j)
{
  if ((i >= 0) && (i < mat->Ligne) && (j >= 0) && (j < mat->Colonne))
    return mat->C [i * mat->Colonne + j] ;
  else { printf ("erreur matrice donne %d %d\n",i,j) ; return 0 ; }
}

Matrice matrice_produit (Matrice *mat1, Matrice *mat2)
{
  if (mat1->Colonne == mat2->Ligne) {
    Matrice r = matrice_new (mat1->Ligne, mat2->Colonne) ;
    double d ;
    int i,j,k ;
    for (i = 0 ; i < r.Ligne ; i++)
      for (j = 0 ; j < r.Colonne ; j++) {
        d = 0 ;
        for (k = 0 ; k < mat1->Colonne ; k++)
             d +=  matrice_donne (mat1, i,k)
                 * matrice_donne (mat2, k,j) ;
        matrice_change (&r, i,j, d) ;
      }
    return r ;
  }
  else {
    Matrice r ;
    r.Ligne = 0 ;
    r.Colonne = 0 ;
    return r ;
  }
}

char *matrice_ecrit (Matrice *mat)
{
  char *res = new char [mat->Ligne * mat->Colonne * 20] ;
  *res = 0 ;
  char *c ;
  int i,j ;
```

```
  for (i = 0 ; i < mat->Ligne ; i++) {
    for (j = 0 ; j < mat->Colonne ; j++) {
      c = res + strlen (res) ;
      sprintf (c, "%g\t",matrice_donne (mat, i,j)) ;
    }
    c = res + strlen (res) ;
    sprintf (c, "\n") ;
  }
  return res ;
}

Matrice matrice_copie (Matrice *mat)
{
  Matrice r = matrice_new (mat->Ligne, mat->Colonne) ;
  int i,j ;
  for (i = 0 ; i < mat->Ligne ; i++)
    for (j = 0 ; j < mat->Colonne ; j++)
      matrice_change (&r, i,j, matrice_donne (mat, i,j)) ;
  return r ;
}

Matrice matrice_pivot_gauss (Matrice *mat)
{
  // la matrice de passage est carrée et est l'identité au début
  Matrice passage = matrice_new_identite (mat->Ligne) ;

  // pivot
  int n,k,l ;
  double d,div,fact ;
  int pos ;
  bool faire_pivot ;

  // nombre de lignes sur lesquels il faut faire le pivot
  // car le rang d'une matrice M est vérifié r (M) <= MIN (Ligne, Colonne)
  int ligne = mat->Ligne ;
  if (mat->Colonne < ligne) ligne = mat->Colonne ;

  // pivot sur toutes les lignes
  for (n = 0 ; n < ligne ; n++) {
    // on teste le terme diagonale
    faire_pivot = true ;  // par défaut, on effectue le pivot sur la colonne n
    d = matrice_donne (mat, n,n) ;
    if (d == 0) {
      // si ce terme est nul, on regarde si un terme dans la même colonne ne l'est pas
      pos = n ;
      while ((pos < mat->Ligne) && (matrice_donne (mat, pos, n) == 0)) pos++ ;
      if (pos >= mat->Ligne) faire_pivot = false ; // car tous les termes de la colonne sont nuls
      else {
        // sinon, on ajoute la ligne pos à la ligne n
        for (k = n ; k < mat->Colonne ; k++) {
          d = matrice_donne (mat, n, k) ;
          d += matrice_donne (mat, pos, k) ;
          matrice_change (mat, n, k, d) ;
        }

        for (k = 0 ; k < passage.Colonne ; k++) {
          // on répercute également ces changements dans la matrice de passage
          // sur toute la ligne cette fois
          d = matrice_donne (&passage, n, k) ;
```

```
          d += matrice_donne (&passage, pos, k) ;
          matrice_change (&passage, n, k, d) ;
        }
      }
    }

    if (faire_pivot) {
      fact = matrice_donne (mat, n,n) ; // on est sûr que fact != 0

      // on fait le pivot sur toutes les lignes après n
      for (l = n+1 ; l < mat->Ligne ; l++) {
        div = matrice_donne (mat, l, n) ;
        // si ce coefficient n'est pas nul
        if (div != 0) {
          for (k = n ; k < mat->Colonne ; k++) {
            d = matrice_donne (mat, l, k) ;
            d -= matrice_donne (mat, n, k)*div/fact ;
            matrice_change (mat, l, k, d) ;
          }

          for (k = 0 ; k < passage.Colonne ; k++) {
            // on répercute également ces changements dans la matrice de passage
            // sur toute la ligne
            d = matrice_donne (&passage, l, k) ;
            d -= matrice_donne (&passage, n, k)*div/fact ;
            matrice_change (&passage, l, k, d) ;
          }
        }
      }
    }

  }
  return passage ;
}

Matrice matrice_pivot_gauss_inverse (Matrice *mat)
{
  // la matrice de passage est carrée et est l'identité au début
  Matrice passage = matrice_new_identite (mat->Ligne) ;

  // pivot
  int n,k,l ;
  double d,div,fact ;
  int ligne = mat->Ligne ;

  // l'inversibilité ne s'applique qu'à des matrices carrées
  bool inversible = mat->Ligne == mat->Colonne ;

  // on s'arrange pour n'avoir que des uns sur les diagonales
  for (n = 0 ; n < ligne ; n++) {
    fact = matrice_donne (mat, n,n) ;
    inversible &= fact != 0 ;
    if (inversible) {
      for (k = n ; k < mat->Colonne ; k++) {
        d = matrice_donne (mat, n, k) ;
        d /= fact ;
        matrice_change (mat, n, k, d) ;
      }
      for (k = 0 ; k < passage.Colonne ; k++) {
```

```
        // on répercute également ces changements dans la matrice de passage
        d = matrice_donne (&passage, n, k) ;
        d /= fact ;
        matrice_change (&passage, n, k, d) ;
      }
    }
    else break ;
  }

  // pivot remontant sur toutes les lignes si la matrice est inversible
  if (inversible) {
    for (n = ligne-1 ; n >= 0 ; n--) {
      fact = matrice_donne (mat, n,n) ;
      // on est sûr que fact != 0
      // même, c'est forcément 1 maintenant

      // on fait le pivot sur toutes les lignes après n
      for (l = n-1 ; l >= 0 ; l--) {
        div = matrice_donne (mat, l, n) ;
        // si ce coefficient n'est pas nul
        if (div != 0) {
          for (k = n ; k < mat->Colonne ; k++) {
            d = matrice_donne (mat, l, k) ;
            d -= matrice_donne (mat, n, k)*div/fact ;
            matrice_change (mat, l, k, d) ;
          }

          for (k = 0 ; k < passage.Colonne ; k++) {
            // on répercute également ces changements dans la matrice de passage
            // sur toute la ligne
            d = matrice_donne (&passage, l, k) ;
            d -= matrice_donne (&passage, n, k)*div/fact ;
            matrice_change (&passage, l, k, d) ;
          }
        }
      }
    }
  }
  return passage ;
}

Matrice matrice_inverse (Matrice *mat)
{
  Matrice copie = matrice_copie (mat) ;
  Matrice pas1 = matrice_pivot_gauss (&copie) ;
  Matrice pas2 = matrice_pivot_gauss_inverse (&copie) ;
  Matrice prod = matrice_produit (&pas2, &pas1) ;
  matrice_free (&copie) ;
  matrice_free (&pas2) ;
  matrice_free (&pas1) ;
  return prod ;
}

////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////

Polynome polynome_new (int degre)
```

```
{
  if (degre >= 0) {
    Polynome r ;
    r.Degre = degre ;
    r.C = new double [degre+1] ;
    return r ;
  }
  else {
    Polynome r ;
    r.Degre = -1 ;
    return r ;
  }
}

void polynome_free (Polynome *p)
{
  if (p->Degre >= 0) {
    delete [] p->C ;
    p->Degre = -1 ;
  }
}


void polynome_change (Polynome *p, int n, double val)
{
  if ((n >= 0) && (n <= p->Degre)) p->C [n] = val ;
  else printf ("erreur polynome_change %d\n",n) ;
}
double polynome_donne (Polynome *p, int n)
{
  if ((n >= 0) && (n <= p->Degre)) return p->C [n] ;
  else { printf ("erreur polynome_donne %d\n",n) ; return 0 ; }
}

char *polynome_ecrit (Polynome *p)
{
  if (p->Degre >= 0)  {
    char *res = new char [p->Degre * 20 + 20] ;
    *res = 0 ;
    char *c = res ;
    int n ;
    for (n = 0 ; n <= p->Degre ; n++) {
      if (polynome_donne (p, n) != 0) {
        if ((c > res) && (polynome_donne (p, n) > 0)) sprintf (c, " + ") ;
        else sprintf (c, " ") ;
        c = res + strlen (res) ;
        if (n == 0) sprintf (c, "%g", polynome_donne (p, n)) ;
        else if (n == 1) {
          if (polynome_donne (p, n) != 1) sprintf (c, "%g*X", polynome_donne (p, n)) ;
          else sprintf (c, "X") ;
        }
        else {
          if (polynome_donne (p, n) != 1) sprintf (c, "%g*X^%d", polynome_donne (p, n), n) ;
          else sprintf (c, "X^%d",n) ;
        }
      }
      c = res + strlen (res) ;
    }
    return res ;
```

```
  }
  else {
    char *res = new char [20] ;
    strcpy (res, "nul") ;
    return res ;
  }
}

Polynome polynome_produit (Polynome *p1, Polynome *p2)
{
  Polynome p = polynome_new (p1->Degre + p2->Degre) ;
  int n ;
  int k ;
  double d ;
  // mise à zéro du polynôme produit
  for (n = 0 ; n <= p.Degre ; n++) polynome_change (&p, n, 0) ;
  // calcul du produit
  for (n = 0 ; n <= p1->Degre ; n++) {
    for (k = 0 ; k <= p2->Degre ; k++) {
      d = polynome_donne (&p, n+k) ;
      d += polynome_donne (p1, n) * polynome_donne (p2, k) ;
      polynome_change (&p, n+k, d) ;
    }
  }
  return p ;
}

////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
// méthode consacrée à la décomposition

// calcul le produit de tous les polynômes du dénominateur
// en évitant la puissance evite_puissance de l'élément evite_el
// si evite_puissance == -1, calcule tous les produits

Polynome polynome_produit_denominateur (int facteur, Polynome *element, int *puissance,
                          int evite_el, int evite_puissance)
{
  // c'est la constante égale à un
  Polynome prod = polynome_new (0) ;
  polynome_change (&prod, 0, 1) ;

  Polynome temp ;

  int n ;
  int k ;
  int fin ;
  for (n = 0 ; n < facteur ; n++) {

    if (evite_el == n) fin = puissance [n] - evite_puissance ;
    else fin = puissance [n] ;

    for (k = 0 ; k < fin ; k++) {
      temp = polynome_produit (&prod, &(element [n])) ;
      polynome_free (&prod) ;
      prod = temp ;
    }
```

```
  }
  return prod ;
}

void polynome_decale_puissance (Polynome *p, int k)
{
  if (k > 0) {
    Polynome res = polynome_new (p->Degre+k) ;
    int n ;
    for (n = 0 ; n <= res.Degre ; n++) {
       if (n < k) polynome_change (&res, n, 0) ;
       else polynome_change (&res, n, polynome_donne (p, n-k)) ;
    }
    polynome_free (p) ;
    *p = res ;
  }
}

void polynome_decomposition (Polynome *numerateur, int facteur, Polynome *element, int *puissance,
                   Polynome partie_entiere,
                   Polynome **resultat)
{
  // calcul du nombre de coefficient et du degré maximal
  int nb_coef = partie_entiere.Degre + 1 ;
  int n ;
  for (n = 0 ; n < facteur ; n++)
    nb_coef += element [n].Degre * puissance [n] ;

  // pour stocker les produits de polynômes
  Polynome *stock = new Polynome [nb_coef] ;

  // on fait les produits
  int k ;
  // on numérote les coefficients en prenant les éléments simpels les uns après les autres
  // par ordre de puissances croissantes
  // et par ordre de monômes de degré croissant

  // la partie entière

  for (k = 0 ; k <= partie_entiere.Degre ; k++) {
    stock [k] = polynome_produit_denominateur (facteur, element, puissance, -1, -1) ;
    // on décale les coefficients vers la droite de k pas
    // pour une multiplicateur de X^k
    polynome_decale_puissance (&(stock [k]), k) ;
  }

  // suite des calculs
  int numcoef = partie_entiere.Degre + 1 ;
  int l ;
  for (n = 0 ; n < facteur ; n++) {
    for (k = 0 ; k < puissance [n] ; k++) { // boucle sur 1 ou 2 coefficients
      for (l = 0 ; l < element [n].Degre ; l++) {
        stock [numcoef] = polynome_produit_denominateur (
                   facteur, element, puissance, n, k+1) ;
        polynome_decale_puissance (&(stock [numcoef]), l) ;
        numcoef++ ;
      }
    }
  }
```

```
  // construction de la matrice du système linéaire
  // ressemble à une transposition

  Matrice systeme = matrice_new (nb_coef, nb_coef) ;

  for (n = 0 ; n < nb_coef ; n++) {
    for (k = 0 ; k < nb_coef ; k++) {
      if (n <= stock [k].Degre) matrice_change (&systeme, n,k,
                                  polynome_donne (&(stock [k]), n)) ;
      else matrice_change (&systeme, n,k,0) ;
    }
  }

  // inversion de la matrice

  Matrice inverse = matrice_inverse (&systeme) ;

  // calcul des coefficients

  Matrice constante = matrice_new (nb_coef, 1) ;

  for (n = 0 ; n < nb_coef ; n++)
    if (n <= numerateur->Degre) matrice_change (&constante, n, 0,
                                  polynome_donne (numerateur, n)) ;
    else matrice_change (&constante, n, 0, 0) ;

  // résolution du système

  Matrice solution = matrice_produit (&inverse, &constante) ;

  // on place les solutions dans la structure censée les recevoir

  // partie entière
  for (n = 0 ; n <= partie_entiere.Degre ; n++)
    polynome_change (&partie_entiere, n, matrice_donne (&solution, n, 0)) ;

  // les autres coefficients
  numcoef = partie_entiere.Degre + 1 ;
  for (n = 0 ; n < facteur ; n++)
    for (k = 0 ; k < puissance [n] ; k++)
      for (l = 0 ; l < element [n].Degre ; l++) {
        polynome_change (&(resultat [n][k]), l, matrice_donne (&solution, numcoef, 0)) ;
        numcoef++ ;
      }

  // fin
  matrice_free (&inverse) ;
  matrice_free (&constante) ;
  matrice_free (&systeme) ;
  matrice_free (&solution) ;

  for (n = 0 ; n < nb_coef ; n++) polynome_free (&(stock [n])) ;
  delete [] stock ;
}



////////////////////////////////////////////////////////////////////
```

```
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////////////
```

Le fichier texte suivant contient une fraction décrite selon le format que le programme précédent peut lire.

```
numérateur degré 9
d0 0
d1 0
d2 0
d3 0
d4 0
d5 0
d6 0
d7 0
d8 0
d9 1
..................dénominateur produit 2
dénominateur puissance 3
dénominateur degré 1
d0 1
d1 1
dénominateur puissance 2
dénominateur degré 2
d0 2
d1 1
d2 1
```