Bibliothèque Mathutil

Michel Llibre

17 février 2004

Table des matières

Introduction

Mathutil est un sous-ensemble de routines en langage C qui mettent en oeuvre des résultats mathématiques de base en géométrie et algèbre linéaire. Ces routines sont regroupées en plusieurs fichiers généralement indépendants :

- Fichiers `sp3.c` et `sp3.h` : Ces fichiers regroupent les routines qui manipulent les vecteurs de $\mathbb{R}^3$ et les matrices 3×3.
- Fichiers `rotation.c` et `rotation.h` : Ces fichiers regroupent les routines qui traitent des rotations dans $\mathbb{R}^3$ (angles, paramètres d'Euler, matrices 3×3 et vecteurs vitesses de rotation) et des déplacements (combiné rotation-translation). Elles font appel à `sp3.h`.
- Fichiers `quatern.c` et `quatern.h` : Ces fichiers regroupent les routines qui traitent des quaternions et de leurs utilisation pour les rotations.
- Fichiers `spn.c` et `spn.h` : Ces fichiers regroupent les routines qui manipulent les vecteurs de $\mathbb{R}^n$ et les matrices $m \times n$.
- Fichiers `linsolve.c` et `linsolve.h` : Ces fichiers regroupent les routines qui résolvent l'équation matricielle $\mathbf{y} = \mathbf{Ax}$. Elles font appel à `spn.h`.
- Fichiers `eigen.c` et `eigen.h` : Ces fichiers regroupent les routines qui calculent les vecteurs propres et valeurs propres d'une matrice.
- Fichiers `polynomes.c` et `polynomes.h` : Ces fichiers regroupent les routines qui manipulent les polynômes. Le fichier `polynomes.c` inclut le fichier `roots234.c` pour le calcul des racines des polynômes de degré inférieur ou égal à `4`.
- Fichiers `racines.c` et `racines.h` : Calcul précis des racines d'un polynome. Le fichier `racines.c` inclut le fichier `roots234.c` pour le calcul des racines des polynômes de degré inférieur ou égal à `4`.
  Traitement des erreurs : Certains algorithmes peuvent être mis en échec, par exemple suite à une allocation mémoire qui échoue. Dans ce cas un message d'erreur est émis et est éventuellemnt mémorisé dans un buffeur fourni par l'utilisateur. Ce mécanisme, configurable par l'utilisateur, est décrit dans le fichier `deferreur.h`. Ce fichier est inclut par `linsolve.c`, `eigen.c`, `polynomes.c` et `racines.c`.

1 Conventions

Dans nos routines en C, nous essayons de respecter les conventions suivantes :

– L'argument résultat est généralement placé en tête de la liste des arguments afin qu'elle rappelle l'expression mathématique calculée. A titre d'exemple `m3m3(A,B,C)` réalise la fonction $A = BC$ (multiplication de matrice 3×3 de $\mathbb{R}^3$).
– Quand le résultat est un scalaire, il n'est pas dans la liste des arguments, c'est la valeur de retour de la fonction.
– Les arguments de type vecteur ou matrice sont passés par l'intermédiaire d'un pointeur sur le premier élément.
– Les éléments des matrices sont supposées être rangés en mémoire, colonne par colonne. Exemple pour une matrice 2×3 :

$$\begin{pmatrix} a_0 & a_2 & a_4 \\ a_1 & a_3 & a_5 \end{pmatrix}$$

Cette convention est celle utilisée en Fortran et en Matlab. En la suivant, on peut faire des applications mélangeant ces divers langages.

– Dans le nom des fonctions, le chiffre 2 se lit généralement "to" comme dans `diag2m3(A,V)` qu'il faut lire diag**to**m3. Cette fonction génère la matrice diagonale A à partir des composantes du vecteur V. La fonction inverse qui extrait le vecteur V de la diagonale de A est `m3todiag(V,A)`. Ici nous n'utilisons pas le chiffre 2 comme raccourci, car il serait précédé du chiffre 3 ce qui gênerait l'effet mnémonique.

Signification des chaînes de caractères

– abs : valeur absolue
– add : addition terme à terme
– asm3 : matrice antisymétrique 3×3
– av3 : paramètres d'Euleur d'une rotation : angle et vecteur unitaire de l'axe
– c : colonne
– cmax : colonne regroupant la plus grande composante de chaque ligne d'une matrice
– cmin : colonne regroupant la plus petite composante de chaque ligne d'une matrice
– csum : colonne regroupant la somme des composantes de chaque ligne d'une matrice
– carol : angles de cap/z_0, assiette/y_1, roulis/x_2
– carre : carré d'un vecteur
– cpy : copie terme à terme
– cross : produit vectoriel croisé
– cum : cumul terme à terme (addition avec un autre vecteur qui est multiplié par un scalaire)
– det : déterminant
– diag : vecteur des éléments de la diagonale d'une matrice
– div : division terme à terme
– dot : produit scalaire d'un vecteur
– euler : angles de précession/z_0, nutation/x_1 et rotation propre/z_2
– eye : matrice identité (ou pseudo-identité si rectangulaire)
– i : index
– imax : index de la plus grande composante
– iamax : index de la plus grande composante en valeur absolue
– imin : index de la plus petite composante
– iamin : index de la plus petite composante en valeur absolue
– iqn : quaternion inverse
– l : ligne
– lmax : ligne regroupant la plus grande composante de chaque colonne d'une matrice
– lmin : ligne regroupant la plus petite composante de chaque colonne d'une matrice
– lsum : ligne regroupant la somme des composantes de chaque colonne d'une matrice
– long : longueur d'un vecteur
– lunit : longueur et unitarisation d'un vecteur
– ltr : angles lacet/z_0, tangage/y_1 et roulis/x_2
– m3 : matrice 3×3
– mat : matrice quelconque
– mrot : matrice de rotation 3×3
– mul : multiplication terme à terme
– neg : changement de signe de tous les termes
– print : impression sur stdout

– por : position et orientation
– pori : por inverse
– omm : vecteur omega (composantes dans le repère mobile Rm)
– omo : vecteur omega (composantes dans le repère fixe Ro)
– qn : quaternion (vecteur de $\mathbb{R}^4$)
– qnu : quaternion unitaire (vecteur de $\mathbb{R}^4$)
– sca : multiplication de tous les termes par le même scalaire
– set : même scalaire affecté à tous les termes
– sub : soustraction terme à terme
– t, tr : transposition
– tm3 : matrice 3×3 transposée
– tmat : matrice transposée
– tens : produit tensoriel de vecteur
– tm3 : matrice transposée de $\mathbb{R}^3$
– tmat : matrice transposée
– tqn : quaternion conjugué
– tv3 : vecteur de $\mathbb{R}^3$ transposé
– v3 : vecteur de $\mathbb{R}^3$
– vec : vecteur de $\mathbb{R}^n$
– x : opération sur les composantes
– xmax : plus grande composante
– xamax : plus grande composante en valeur absolue
– xmin : plus petite composante
– xamin : plus petite composante en valeur absolue
– xsum : somme des composantes

2 Routines des fichiers sp3.c et sp3.h

Manipulation de vecteurs de $\mathbb{R}^3$

Affectation s.	$u_i = s$	`double *setv3 (double* u, double s)`
Affectation t.à.t.	$u_i = v_i$	`double *cpyv3 (double* u, double* v)`
Négation	$u_i = -v_i$	`double *negv3 (double* u, double* v)`
Addition.	$u_i = v_i + w_i$	`double *addv3 (double* u, double* v, double* w)`
Soustraction	$u_i = v_i - w_i$	`double *subv3 (double* u, double* v, double* w)`
Cumul	$u_i = v_i + sw_i$	`double *cumv3 (double* u, double* v, double s, double* w)`
Multiplication t.à.t.	$u_i = v_i w_i$	`double *mulv3 (double* u, double* v, double* w)`
Division t.à.t. (1)	$u_i = \frac{v_i}{w_i}$	`double *divv3 (double* u, double* v, double* w)`
Multiplication s.	$u_i = sv_i$	`double *scav3 (double *u, double s, double *v)`
Valeur Absolue t.à.t.	$u_i = \lvert v_i \rvert$	`double *absv3 (double* u, double* v)`
Maximum	$s = \max u_i$	`double xmaxv3 (double* u)`
Maximum en v.a.	$s = \max_i \lvert u_i \rvert$	`double xamaxv3 (double* u)`
Minimum	$s = \min u_i$	`double xminv3 (double* u)`
Minimum	$s = \min \lvert u_i \rvert$	`double xaminv3 (double* u)`
Index du max	$i = \arg\max u_i$	`int imaxv3 (double* u)`
Index du max en v.a.	$i = \arg\max \lvert u_i \rvert$	`int iamaxv3 (double* u)`
Index du min	$i = \arg\min u_i$	`int iminv3 (double* u)`
Index du min en v.a.	$i = \arg\min \lvert u_i \rvert$	`int iaminv3 (double* u)`
Produit scalaire	$s = \sum u_i v_i$	`double dotv3 (double* u, double* v)`
Produit tensoriel	$a_{ij} = u_i v_j$	`double *tensv3 (double* a, double* u, double* v)`
Produit vectoriel	$u_i = v_i \times w_i$	`double *crossv3 (double* u, double* v, double* w)`
Carré du vecteur	$s = \sum u_i^2$	`double carrev3 (double* u)`
Somme des u_i	$s = \sum_i u_i$	`double xsumv3 (double* u)`
Longueur	$s = \sqrt{\sum u_i^2}$	`double longv3 (double* u)`
Unitarisation	$u_i = \frac{v_i}{s}$	`double *unitv3 (double* u, double *v)`
Unitarisation	$s = \lVert \mathbf{u} \rVert$, $u_i = \frac{v_i}{s}$	`double lunitv3 (double* u, double *v)`
Impression		`voint printv3 (char *, double *v)`

s et sca. : Signifie scalaire. Le même scalaire s intervient au niveau de chaque composante.
t.à t. : Signifie terme à terme.
La fonction `crossv3` (produit vectoriel est spécifique à $\mathbb{R}^3$.
La fonction `divv3` est d'une utilisation **dangereuse** car la division par zéro n'est pas testée.

Dans les opérations du type $u_i = \diamond v_i$, ou $u_i = s \diamond v_i$ ou $u_i = v_i \diamond w_i$ où $\diamond$ représente un opérateur quelconque, le résultat **u** peut occuper la même place mémoire qu'un des arguments **v** ou **w**. Par contre pour le produit tensoriel $\mathbf{a} = \mathbf{u}\mathbf{v}^T$, la matrice **a** ne doit pas occuper le même espace mémoire que les vecteurs **v** ou **w**.

La fonction d'unitarisation retourne la longueur s du vecteur **v**. Si $s = 0$, le vecteur **u** est tel que tous les $u_i = 0$ sauf $u_1 = 1$.

Manipulation de matrice 3×3

Affectation s.	$a_{ij} = s$	`double *setm3 (double* a, double s)`
Affectation t.à t.	$\mathbf{A} = \mathbf{B}$	`double *cpym3 (double* a, double* b)`
Négation	$\mathbf{A} = -\mathbf{B}$	`double *negm3 (double* a, double* b)`
Addition.	$\mathbf{A} = \mathbf{B} + \mathbf{C}$	`double *addm3 (double* a, double* b, double* c)`
Soustraction	$\mathbf{A} = \mathbf{B} - \mathbf{C}$	`double *subm3 (double* a, double* b, double* c)`
Cumul	$\mathbf{A} = \mathbf{B} + s\mathbf{C}$	`double *cumm3 (double* a, double* b, double s, double* c)`
Multiplication s.	$\mathbf{A} = s\mathbf{B}$	`double *scam3 (double *a, double s, double *b)`
Matrice identité	$\mathbf{A} = \mathbf{I}$	`double *eyem3(double *a)`
Matrice diagonale	$a_{ii} = v_i$	`double *diag2m3(double *a, double *v)`
Partie diagonale	$v_i = a_{ii}$	`double *m3todiag(double *v, double *a)`
Matrice antisym	$\mathbf{A} = \tilde{\mathbf{v}}$	`double *v3toasm3(double *a, double *v)`
Partie antisym	$\mathbf{v} = \frac{1}{2}(\widetilde{\mathbf{A} - \mathbf{A}^T})$	`double *asm3tov3(double *v, double *a)`
Transposition	$\mathbf{A} = \mathbf{B}^T$	`double *tm3(double *a, double *b)`
Produit mad x vec	$\mathbf{A} = \mathbf{B}\mathbf{v}$	`double *m3v3(double* a, double* b, double* v)`
Produit mat x vec	$\mathbf{A} = \mathbf{B}^T\mathbf{v}$	`double *tm3v3(double* a, double* b, double* v)`
Produit vec x mat	$\mathbf{A} = \mathbf{v}^T\mathbf{B}$	`double *tv3m3(double* a, double* v, double* b)`
Produit direct×direct	$\mathbf{A} = \mathbf{B}\mathbf{C}$	`double *m3m3 (double *a, double *b, double *c)`
Produit direct×transp.	$\mathbf{A} = \mathbf{B}\mathbf{C}^T$	`double *m3tm3 (double *a, double *b, double *c)`
Produit transp.×direct	$\mathbf{A} = \mathbf{B}^T\mathbf{C}$	`double *tm3m3(double* a, double* b, double* c)`
Trace	$s = \sum_i a_{ii}$	`double tracem3(double *a)`
Déterminant	$s = \|\mathbf{A}\|$	`double detm3(double *a)`
Dét. et inverse	$d = \|\mathbf{A}\|, \mathbf{A}^{-1}$	`double dinvm3(double *a)`
Impression		`void printm3(char *, double *a)`

Les fonctions `v3toasm3` et `asm3tov3` sont spécifiques de $\mathbb{R}^3$. Elles mettent en correspondance une matrice antisymétrique 3×3 `asm3` et un vecteur `v3` de $\mathbb{R}^3$.

3 Routines des fichiers rotation.c et rotation.h

Matrices de rotation

Rotation a / $\vec{e}_x$	$a_x \to R$	`double *mrotx(double *m, double a)`
Rotation a / $\vec{e}_y$	$a_y \to R$	`double *mroty(double *m, double a)`
Rotation a / $\vec{e}_z$	$a_z \to R$	`double *mrotz(double *m, double a)`
Rot. $\psi_z, \theta_x, \varphi_z$	$(\psi_z, \theta_x, \varphi_z) \to R$	`double *euler2mrot (double *m, double *eul)`
Rot. $\psi_z, \theta_y, \varphi_x$	$(\psi_z, \theta_y, \varphi_x) \to R$	`double *carol2mrot (double *m, double *car)`
Rot. l_z, t_x, r_y	$(l_z, t_x, r_y) \to R$	`double *ltr2mrot (double *m, double *ltr)`
Rotation a / $\vec{v}$	$(a, v) \to R$	`double *av3tomrot(double* m, double a, double* v)`
Rotation $\vec{v}$	$\vec{v} \to R$	`double *v3tomrot(double* m, double* v)`
$\psi_z, \theta_x, \varphi_z$ d'1 rot.	$R \to (\psi_z, \theta_x, \varphi_z)$	`double *mrot2euler (double *eul, double *m)`
$\psi_z, \theta_y, \varphi_x$ d'1 rot.	$R \to (\psi_z, \theta_y, \varphi_x)$	`double *mrot2carol (double *car, double *m)`
l_z, t_x, r_y d'1 rot.	$R \to (l_z, t_x, r_y)$	`double *mrot2ltr (double *ltr, double *m)`
a / $\vec{u}$ d'1 rot.	$R \to (a, u)$	`double mrot2av3(double* v, double* m)`
$\vec{v}$ d'1 rot.	$R \to \vec{v}$	`double *mrot2v3(double* v, double* m)`

Le passage des angles (carol, euler ou ltr) vers la matrice de rotation est toujours défini et univoque. Le passage inverse offre deux solutions et peut poser un problème singulier. Pour lever ces ambiguïtés, l'utilisateur doit passer des angles de référence par l'argument (qui seront remplacés par les angles résultats). Dans le cas régulier, on choisit la solution résultat dans la même nappe que les angles de référence. En configuration singulière (jonction des nappes) les angles de références sont utilisés pour calculer la solution qui leur est la plus proche.

Pour la fonction `av3tomrot` qui calcule la matrice de rotation d'angle a, autour de l'axe de vecteur unitaire v, il faut faire attention à ce que v soit effectivement de norme 1.

Vecteur vitesse de rotation

Vit. rotation ds Ro	$(\Omega)_0 = F_{zxz}(\dot{\psi},\theta,\dot{\phi})$	`double deuler2omo(double *omo, double *ltrp, double *ltr)`
Vit. rotation ds Rm	$(\Omega)_m = F_{zxz}(\dot{\psi},\theta,\dot{\phi})$	`double deuler2omm(double *omm, double *ltrp, double *ltr)`
Vit. rotation ds Ro	$(\Omega)_0 = F_{zyx}(\dot{\psi},\theta,\dot{\phi})$	`double dcarol2omo(double *omo, double *ltrp, double *ltr)`
Vit. rotation ds Rm	$(\Omega)_m = F_{zyx}(\dot{\psi},\theta,\dot{\phi})$	`double dcarol2omm(double *omm, double *ltrp, double *ltr)`
Vit. rotation ds Ro	$(\Omega)_0 = F_{zxy}(l,\dot{t},\dot{r})$	`double dltr2omo(double *omo, double *ltrp, double *ltr)`
Vit. rotation ds Rm	$(\Omega)_m = F_{zxy}(l,\dot{t},\dot{r})$	`double dltr2omm(double *omm, double *ltrp, double *ltr)`
Dérivées Euler	$\dot{\psi},\theta,\dot{\phi} = F^{-1}_{zxz}(\Omega)_0$	`double omo2deuler(double *ltrp, double *omo, double *ltr)`
Dérivées Euler	$\dot{\psi},\theta,\dot{\phi} = F^{-1}_{zxz}(\Omega)_m$	`double omm2deuler(double *ltrp, double *omm, double *ltr)`
Dérivées aero.	$\dot{\psi},\theta,\dot{\phi} = F^{-1}_{zyx}(\Omega)_0$	`double omo2dcarol(double *dcarol, double *omo, double *ltr)`
Dérivées aero.	$\dot{\psi},\theta,\dot{\phi} = F^{-1}_{zyx}(\Omega)_m$	`double omm2dcarol(double *dcarol, double *omm, double *ltr)`
Dérivées ltr	$l,\dot{t},\dot{r} = F^{-1}_{zxy}(\Omega)_0$	`double omo2dltr(double *dltr, double *omo, double *ltr`
Dérivées ltr	$l,\dot{t},\dot{r} = F^{-1}_{zxy}(\Omega)_m$	`double omm2dltr(double *dltr, double *omm, double *ltr)`

Le passage des dérivées des angles aux composantes du vecteur vitesse de rotation est toujours régulier. Le passage inverse est singulier lorsque le deuxième angle de rotation est tel que les premier et dernier axes sont alignés. Au voisinage de cette situation, nous utilisons les valeurs des dérivées des angles fournies en entrée pour fournir une solution voisine de ces valeurs.

position+orientation = por

Les *pors* sont des agrégats concaténant un vecteur translation 3×1 et une matrice rotation 3×3. Au niveau des produits, ils se comportent comme des opérateurs de déplacement, enchaînant la translation, puis la rotation. Les opérations successives sont effectuées dans les repères transformés. Dans OpenGL ils sont utilisés sous la forme classique de matrices homogènes 4×4, avec la matrice rotation dans le premier bloc diagonal 3×3 et le vecteur translation dans les 3 premières composantes du quatrième vecteur colonne. Les 3 premières composantes de la dernière ligne sont nulles. Sous cette forme le déplacement nul est la matrice 4×4 identité et le produit de déplacement est égal au produit de matrice 4×4. Ces matrices 4×4 présentent pour nous l'inconvénient d'alourdir les calculs. En effet, les translations simples ne sont pas effectuées par l'addition d'un vecteur 3×1, mais par multiplication par une 4×4 ayant une matrice identité dans la partie rotation et les rotations simples ne sont pas effectuées par la multiplication par une matrice 3×3, mais par multiplication par une 4×4 ayant un vecteur nul dans la partie translation. De plus, pour manipuler la matrice de rotation de la matrice 4×4, il ne faut pas oublier de prendre en compte les zéros de la quatrième ligne.

Nos *pors* mémorisent la translation et la rotation, à la suite l'une de l'autre, sans ajout ni modification de structure de manière à permettre l'utilisation des routines du cas $\mathbb{R}^3$.

Affectation t.à t.	$D_1 = D_2$	`double *cpypor(double* D1, double* D2);`
Produit direct.	$A = BC$	`double *porpor(double* A, double* B, double* C)`
Produit dir*inv	$A = BC^{-1}$	`double *porpori(double* a, double* b, double* c)`
Produit inv*dir	$A = B^{-1}C$	`double *poripor(double* a, double* b, double* c)`
Dépl. direct de v3	$w = Av$	`double *porv3(double* a, double* b, double* c)`
Dépl. inv. de v3	$w = A^{-1}v$	`double *poriv3(double* a, double* b, double* c)`
Impression		`void printpor(char *s, double *a)`

4 Routines des fichiers quatern.c et quatern.h

Fonctions spéciales quaternions

Les quaternions sont programmés comme des tableaux de 4 `double`(s).

Rotation a / $\vec{e}_x$	$a_x \to Q$	`double *qrotx(double *q, double a)`
Rotation a / $\vec{e}_y$	$a_y \to Q$	`double *qroty(double *q, double a)`
Rotation a / $\vec{e}_z$	$a_z \to Q$	`double *qrotz(double *q, double a)`
Quaternion standard	$Q \to Q$ avec $q[0] > 0$	`double *qnstd (double *q)`
Proximité	$Q \to Q$ proche de Q_{ref}	`double *qnprox (double *q, double *qref)`
Copie de quaternion	$u_i = v_i$	`double *cpyqn(double*u, double* v)`
Conjugué $\bar{Q}$	$q[1:3] = -q[1:3]$	`double *tqn(double* q)`
Inverse	$Q/\|\|Q\|\|^2$	`double *iqn(double* q)`
Produit normal	$Q = Q_1 * Q_2$	`double *qnqn (double *q, double *q1, double *q2)`
Produit conj×direct	$Q = Q_1 * Q_2$	`double *tqnqn (double *q, double *q1, double *q2)`
Produit direct×conj	$Q = Q_1 * Q_2$	`double *qntqn (double *q, double *q1, double *q2)`
Produit inv×direct	$Q = Q_1^{-1} * Q_2$	`double *iqnqn (double *q, double *q1, double *q2)`
Produit direct×inv	$Q = Q_1 * Q_2^{-1}$	`double *qniqn (double *q, double *q1, double *q2)`
Rotation $\vec{v}$ par Quat. unit.	$w = Q * v * \bar{Q}$	`double *qnuv3tqnu (double *w, double *q, double *v)`
Mat.Rot ->Quaternion	$Q = Q(R)$	`double *mrot2qn (double *qn, double *mrot)`
Quat. unit->Mat.Rot	$R = R(Q)$ suppose $\|\|Q\|\| = 1$	`double *qnu2mrot (double *mrot, double *qnu)`
a / $\vec{u}$ rot. d'1 quat. unit.	$Q \to (a,u)$	`double qnu2av3(double* v, double* qnu)`
$\vec{v}$ rot. d'1 quat. unit.	$Q \to \vec{v}$	`double *qnu2v3(double* v, double* qnu)`
Norme (Carre) de Q	$n(Q) = \|\|Q\|\|^2$	`double carreqn (double *qn)`
Longueur de Q	$l(Q) = \sqrt{\|\|Q\|\|^2}$	`double longqn (double *qn)`
Unitarisation de Q	$q_i = \frac{q_i}{\|\|\mathbf{q}\|\|}$	`double unitqn (double *qn)`
Unitarisation de Q	$l = \|\|\mathbf{q}\||, u_i = \frac{q_i}{l}$	`double lunitqn (double *qn)`
Dérivée de Q	$\dot{Q} = \frac{1}{2}Q\Omega$	`double omm2dqn(double *dqn, double *omm, double *qn)`
Dérivée de Q	$\dot{Q} = \frac{1}{2}\Omega Q$	`double omo2dqn(double *dqn, double *omo, double *qn)`
Vitesse de rotation	$\Omega = 2Q^{-1}\dot{Q}$	`double dqn2omm(double *omm, double *dqn, double *qn)`
Vitesse de rotation	$\Omega = 2\dot{Q}Q^{-1}$	`double dqn2omo(double *omo, double *dqn, double *qn)`

5 Routines des fichiers spn.c et spn.h

Manipulation de vecteurs de $\mathbb{R}^n$

Affectation par sca.	$u_i = s$	`double *setvec (double* u, double s, int I)`
Affectation t.à t.	$u_i = v_i$	`double *cpyvec (double* u, double* v, int I)`
Echange 2 vecteurs	$u_i \longleftrightarrow v_i$	`double *swapvec (double* u, double* v, int I)`
Négation	$u_i = -v_i$	`double *negvec (double* u, double* v, int I)`
Addition.	$u_i = v_i + w_i$	`double *addvec (double* u, double* v, double* w, int I)`
Soustraction	$u_i = v_i - w_i$	`double *subvec (double* u, double* v, double* w, int I)`
Cumul	$u_i = v_i + sw_i$	`double *cumvec (double* u, double* v, double s, double* w, int I)`
Multiplication t.à.t.	$u_i = v_i w_i$	`double *mulvec (double* u, double* v, double* w, int I)`
Division t.à.t. (1)	$u_i = \frac{v_i}{w_i}$	`double *divvec (double* u, double* v, double* w, int I)`
Multiplication par sca.	$u_i = sv_i$	`double *scavec (double *u, double s, double *v, int I)`
Valeur Absolue t.à.t.	$u_i = \|v_i\|$	`double *absvec (double* u, double* v, int I)`
Maximum	$s = \max_i u_i$	`double xmaxvec (double* u, int I)`
Maximum en v.abs.	$s = \max_i \|u_i\|$	`double xamaxvec (double* u, int I)`
Index du max.	$i = \arg\max_i u_i$	`int imaxvec (double* u, int I)`
Index du max. en v.a.	$i = \arg\max_i \|u_i\|$	`int iamaxvec (double* u, int I)`
Minimum	$s = \min_i u_i$	`double xminvec (double* u, int I)`
Minimum	$s = \min_i \|u_i\|$	`double xaminvec (double* u, int I)`
Index du min	$i = \arg\min_i u_i$	`int iminvec (double* u, int I)`
Index du min en v.a.	$i = \arg\min_i \|u_i\|$	`int iaminvec (double* u, int I)`
Produit scalaire	$s = \sum u_i v_i$	`double dotvec (double* u, double* v, int I)`
Produit tensoriel	$a_{ij} = u_i v_j$	`double *tensvec (double* a, double* u, double* v, int I, int J)`
Carré du vecteur	$s = \sum u_i^2$	`double carrevec (double* u, int I)`
Somme des u_i	$s = \sum u_i$	`double xsumvec (double* u, int I)`
Longueur	$s = \sqrt{\sum u_i^2}$	`double longvec (double* u, int I)`
Unitarisation	$u_i = \frac{v_i}{s}$	`double *unitvec (double* u, double *v, int I)`
Unitarisation	$s = \|\|\mathbf{u}\||, u_i = \frac{v_i}{s}$	`double lunitvec (double* u, double *v, int I)`

La fonction `divvec` est d'une utilisation **dangereuse** car la division par zéro n'est pas testée.

Dans les opérations du type $u_i = \diamond v_i$, ou $u_i = s \diamond v_i$ ou $u_i = v_i \diamond w_i$ où $\diamond$ représente un opérateur quelconque, le résultat **u** peut occuper la même place mémoire qu'un des arguments **v** ou **w**. Par contre pour le produit tensoriel $\mathbf{a} = \mathbf{u}\mathbf{v}^T$, la matrice **a** ne doit pas occuper le même espace mémoire que les vecteurs **v** ou **w**.

La fonction d'unitarisation retourne la longueur s du vecteur **v**. Si $s = 0$, le vecteur **u** est tel que tous les $u_i = 0$ sauf $u_1 = 1$.

Manipulation de matrices de $\mathbb{R}^{n\times m}$

Les opérations entre scalaire et matrice ou entre matrices terme à terme ne font pas l'objet de fonctions spécifiques. On utilisera celles écrites pour les vecteurs en donnant comme dimension le produit du nombre de ligne par le nombre de colonnes. Ainsi pour remplir une matrice 3×4 avec des 1, on fera :

```
setvec(a, 1., 12);
```

Les dimensions sont fournies en queue de la liste des arguments. Lorsque plusieurs dimensions sont en jeux (trois au plus), les dimensions communes ne sont pas répétées. La première dimension est toujours le nombre de lignes de la matrice résultat et la dernière dimension est toujours le nombre de colonnes de la matrice résultat (ce qui peut être troublant dans le cas de la transposition, cf. `trmat`).

Transposition	$a_{ij} = b_{ji}$	`double *tmat(double *a, int n)`
Transposition	$a_{ij} = b_{ji}$	`double *trmat(double *a, double *b, int lia, int coa)`
Matrice identité	$a_{ii} = 1, a_{ij} = 0$	`double eyemat(double *a, int I, int J)`
Matrice diagonale	$a_{ii} = u_i$	`double *mat2diag(double *u, double *a, int I, int J)`
Partie diagonale	$u_i = a_{ii}$	`double *diag2mat(double *a, double *u, int I, int J)`
Col. maximale	$u_i = \max_j a_{ij}$	`double *cmaxmat (double *u, double *a, int n, int m)`
Col. minimale	$u_i = \min_j a_{ij}$	`double *cminmat (double *u, double *a, int n, int m)`
Colonne somme	$u_i = \sum_j a_{ij}$	`double *csummat (double *u, double *a, int n, int m)`
Ligne maximale	$u_j = \max_i a_{ij}$	`double *lmaxmat (double *u, double *a, int n, int m)`
Ligne minimale	$u_j = \min_i a_{ij}$	`double *lminmat (double *u, double *a, int n, int m)`
Ligne somme	$u_j = \sum_i a_{ij}$	`double *lsummat (double *u, double *a, int n, int m)`
Produit direct×direct	$a_{ij} = \sum_k b_{ik} c_{kj}$	`double *matmat (double *a, double *b, double *c, int I, int K, int J)`
Produit direct×transp.	$a_{ij} = \sum_k b_{ik} c_{jk}$	`double *mattmat (double *a, double *b, double *c, int I, int K, int J)`
Produit transp.×direct	$a_{ij} = \sum_k b_{ki} b_{kj}$	`double *tmatmat (double *a, double *b, double *c, int I, int K, int J)`
Produit direct×transp.	$a_{ij} = \sum_k b_{ik} b_{jk}$	`double *AAtmat (double *a, double *b, int I, int K)`
Produit transp.×direct	$a_{ij} = \sum_k b_{ki} c_{kj}$	`double *tAAmat (double *a, double *b, int I, int K)`
Trace	$s = \sum_i a_{ii}$	`double tracemat(double *a, int I, int J)`
Impression		`void printmat(char *, double *a, int li, int co)`

Deux fonctions pour la transposition : `tmat` pour les matrices carrées qui sont transposées en place et `trmat` pour les matrices quelconques *qui ne doivent pas occuper la même zone mémoire*.

Dans le cas particulier des produits de matrice, la chaîne tmat signifie que la matrice est utilisée sous forme *transposée*. Ainsi le produit $\mathbf{A} = \mathbf{BC}$ est réalisé par la fonction matmat(A,B,C,...), le produit $\mathbf{A} = \mathbf{B}^T\mathbf{C}$ est réalisé par la fonction tmatmat(A,B,C,...) et le produit $\mathbf{A} = \mathbf{BC}^T$ est réalisé par la fonction mattmat(A,B,C,...). Au niveau des dimensions, les première et dernière sont toujours celles du résultat. Ainsi, si les dimensions passées sont dans l'ordre I, J et K, on aura toujours $\mathbf{A}$ de dimension $I \times J$, $\mathbf{B}$ de dimension $I \times K$ pour les produits $\mathbf{BC}$ et $\mathbf{BC}^T$ et de dimension $K \times I$ pour le produit $\mathbf{B}^T\mathbf{C}$, et $\mathbf{C}$ de dimension $K \times J$ pour les produits $\mathbf{BC}$ et $\mathbf{B}^T\mathbf{C}$ et de dimension $J \times K$ pour le produit $\mathbf{BC}^T$.

Dans les trois opérations de produit, *la matrice résultat* $\mathbf{A}$, *ne doit pas occuper la même place mémoire que les deux autres matrices*.

6 Routines des fichiers linsolve.c et linsolve.h

Résolutions (inversion) de systèmes linéaires

Ces routines résolvent le système linéaire $\mathbf{Y} = \mathbf{AX}$ par diverses méthodes. La donnée $\mathbf{Y}$ de dimension $m \times p$ est transmise dans la matrice `XY` et le résultat $\mathbf{X}$ de dimension $m \times p$ (ou $n \times p$) écrase cette donnée. La matrice $\mathbf{A}$ de dimension $m \times m$ (ou $m \times n$) est toujours détruite par le calcul. Pour obtenir l'inverse de $\mathbf{A}$, il suffit de mettre une matrice identité dans $\mathbf{Y}$.

Pour les algorithmes de décomposition en valeur singulière, le nombre d'itération pour converger vers ces valeurs est limité à la valeur `MAXIT`. Si ces algorithmes ne convergent pas, on peut tenter d'augmenter cette valeur dans le fichier header `linsolve.h` et recompiler.

CHOLESKI avec $\sqrt{}$	$\mathbf{X} = \mathbf{A}^{-1}\mathbf{Y}$ $s = \lvert\mathbf{A}\rvert$	`double solvchsq (double *XY, double *A, int m, int p)`
CHOLESKI UDU	$\mathbf{X} = \mathbf{A}^{-1}\mathbf{Y}$	`double solvchud (double *XY, double *A, int m, int p)`
Factorisation LU	$\begin{cases} \mathbf{X} = \mathbf{A}^{-1}\mathbf{Y} \text{ si ftr=0} \\ \mathbf{X} = \left(A^T\right)^{-1}\mathbf{Y} \text{ si ftr=1} \end{cases}$	`double solvlu (double *XY, double *A, int m, int p, int ftr)`
Factorisation QR	$\min_{\mathbf{X}} \lVert\mathbf{Y} - \mathbf{AX}\rVert^2$	`int solvqr (double *XY, double *A, int m, int n, int p, double eps)`
Dec. Val. Sing. (SVD)	$\min_{\mathbf{X}} \begin{cases} \lVert\mathbf{Y} - \mathbf{AX}\rVert^2 \\ \lVert\mathbf{X}\rVert^2 \end{cases}$	`int solvsvd (double *XY, double *A, int m,int n, int p, double eps)`

Hypothèses :

– `solvchsq` suppose $\mathbf{A}$ symétrique définie positive,

– `solvchud` suppose **A** symétrique définie,
– `solvlu` suppose **A** carrée régulière

Ces 3 routines retournent le déterminant de **A**. S'il est nul le résultat **X** n'a pas de sens. Pour `solvchsq`, s'il est négatif le résultat est incertain, y compris le signe du déterminant.

Pour `solvqr` et `solvsvd`, l'argument `eps` est un epsilon relatif à partir duquel on considère que les nouvelles lignes ou colonnes sont dépendantes des précédentes. Ces deux routines renvoient le rang r de la matrice **A**. Dans le cas de solvqr, si **A** est carrée régulière, on peut calculer $\det(\mathbf{A})$ par $\det(\mathbf{A}) = \prod_{i=0}^{r}(-a_{ii})$ avec les a_{ii} retournés dans `A`. Dans le cas de solvsvd, si **A** est carrée régulière, on peut calculer $\det(\mathbf{A})$ par $|\det(\mathbf{A})| = \prod_{i=0}^{r}(a_{ii})$ avec les a_{ii} retournés dans `A`.

Fonctions matricielles diverses

Déterminant (par LU)	$\|\mathbf{a}\|$	`double detmat(double *A, int m)`
Rang d'une matrice (par QR)		`int rankqr (double *A, int m, int n, double eps)`
Noyau orth. à gauche de A	$\mathbf{N}^T\mathbf{A} = \mathbf{0}$	`int nullqr (double *N, double *A, int m, int n, double eps)`
Noyaux orth. de A	$\mathbf{AN} = \mathbf{0}$ si ftr=0 $\mathbf{N}^T\mathbf{A} = \mathbf{0}$ si fl=1	`int nullsvd (double *N, double *A, int m, int n,double eps, int fl)`
Factorisation QR	Entrée : AR=**A** Sortie : AR=**R** $\mathbf{QR} = \mathbf{A}$	`void mat2QR (double *Q, double *AR, int m, int n)`
Permutation de colonnes de A selon les indices jpvt		`void permcol(double *A, int *jpvt, int nlin, int ncol)`
Permutation de lignes de A selon les indices jpvt		`void permlin(double *A, int *jpvt, int nlin, int ncol)`
Factorisation QRE	Entrée : AR=**A** Sortie : AR=**R** $\mathbf{QRE}^T = \mathbf{A}$	`int mat2QRE (double *Q, double *AR, int *jpvt, int m, int n, double eps)`
Dec. en val. sing.	$\mathbf{UDV}^T = \mathbf{A}$	`int mat2SVD (double *U, double *S, double *V, double *A, int m,int n,double eps)`
Dét. et inverse (par LU)	$s = \|\mathbf{A}\|$, $\mathbf{A}^{-1}$	`double invmat(double *A, int m)`
Pseudo-inverse (SVD)	$\mathbf{X} = \mathbf{A}^+$	`int pinvmat(double *X, double *A, int m, int n)`

Remarques :
– Noyau : Le nombre de colonnes du noyau est de $m-r$ pour le noyau à gauche et de $n-r$ pour le noyau à droite, r étant le rang de **A** qui est renvoyé par les routines `rankqr`, `nullqr`, `nullsvd`, `mat2QR` et `mat2svd`.
– Factorisation QRE : La matrice **E** telle que $\mathbf{QRE}^T = \mathbf{A}$ s'obtient par exemple par :
`permlin(eyemat(E,J,J),jpvt,J,J)`.
On peut retrouver **A** à partir du produit **QR** par `permcol(QR,jpvt,I,J)`.
– Décomposition en valeurs singulières : La matrice diagonale **D** telle que $\mathbf{UDV}^T = \mathbf{A}$ s'obtient à partir de `S` par :
`D=diag2mat(S,I,J)`.

Attention : Toutes ces routines **détruisent la matrice A!!!**

7 Routines des fichiers eigen.c et eigen.h

Valeurs et vecteurs propres

Valeurs propres λ_i	$\mathbf{Av}_i = \lambda_i\mathbf{v}_i$	`int eigval (double *wr, double *wi, double *A, int n, int mode)`
Valeurs propres λ_i Vecteurs propres $\mathbf{v}_i$	$\mathbf{Av}_i = \lambda_i\mathbf{v}_i$	`int eigvec (double *wr, double *wi, double *M, double *A, int n, int mode)`

L'algorithme utilisé transforme la matrice **A** sous forme de Hessenberg, puis effectue des itérations de factorisation QR avec décalage spectral. L'indicateur `mode` permet de spécifier un balancement préalable de la matrice, et de choisir le type de factorisation (Houseolder ou Givens). En sortie `wr` et `wi` sont les parties réelles et imaginaires des valeurs propres.

Dans le cas de `eigvec`, en sortie, M est la matrice des vecteurs propres et A contient une matrice bloc diagonale avec les valeurs propres réelles sur la diagonale ou un bloc du type $\begin{pmatrix} a & b \\ -b & a \end{pmatrix}$ si $a+ib$ est valeur propre complexe. Dans le cas de valeurs propres complexes conjuguées, M contient d'abord le vecteur partie réelle et après le vecteur partie imaginaire. Cette disposition est telle que MAM^{-1} redonne la matrice A initiale.

8 Routines des fichiers polynomes.c et polynomes.h

Les coefficients des polynômes sont ranges par degré croissant

Division euclidienne	$\mathbf{a}(x) = \mathbf{b}(x)\mathbf{q}(x) - \mathbf{r}(x)$	`int divpol (double *a, int na, double *b, int nb, double *q, int *Nq,double *r)`
Valeur polynome	$s = \mathbf{P}(x)$	`double valpol (double *P, int n, double x)`
Valeur dérivée	$s = \frac{d}{dx}\mathbf{P}(x)$	`double valderpol (double *P, int n, double x)`
Polynome dérivé	$\mathbf{P}' = \frac{d}{dx}\mathbf{P}$	`int derpol (double *Pd, double *P, int np)`
Multipl. réelle	$\mathbf{P} = (x-a)\mathbf{P}$	`int apend_realroot (double *P, int n, double a)`
Multipl. cpx	$\mathbf{P} = (z-(a+ib))\mathbf{P}$	`int apend_cpxroot (double *P, int n, double a, double b)`
Racines ->Polynome	$\mathbf{P} = \prod(z_k - (a_k + ib_k))$	`void rac2pol (double *P, double *a, double *b, int n)`
Racines réelles pol.	$X_{inf} \le x_i \le X_{sup}$ $\mathbf{P}(x_i) = 0$	`int roots_sturmod (double *x, double *P, int n, double Xinf, double Xsup, double epsx)`
Racines réelles pol.	x_i | $\mathbf{P}(x_i) = 0$	`int roots_sturm (double *x, double *P, int n, double epsx)`
Racines cpx pol.	$a_k + ib_k$ | $\mathbf{P}(a_k + ib_k) = 0$	`int roots_newton (double *a, double *b, double *P, int n)`
Racines cpx pol.	$a_k + ib_k$ | $\mathbf{P}(a_k + ib_k) = 0$	`int roots_qd (double *a, double *b, double *P, int n)`

Les 4 dernières routines calculent les racines des polynomes. Ils sont, dans l'ordre, de plus en plus lents et de plus en plus précis, tout en étant peu performants en cas de racines multiples. Pour ces cas délicats, on utilsera les algorithmes du fichier `racines.c`.

`roots_sturm` et `roots_sturmod` ne calculent que les racines réelles, à l'aide des suites de Sturm.

`roots_newton` et `roots_qd` calculent toutes les racines (réelles ou complexes) par l'algorithme de Newton et par l'algorithme quotient-difference de Rutishauser.

9 Routines des fichiers racines.c et racines.h

Calcul précis des racines d'un polynôme

Racines cpx pol.	$a_k + ib_k$	$\mathbf{P}(a_k + ib_k) = 0$	`int roots_hqr (double *a, double *b, double *P, int n)`
Racines cpx pol.	$a_k + ib_k$	$\mathbf{P}(a_k + ib_k) = 0$	`int roots_balhqr (double *a, double *b, double *P, int n)`

Ces deux algorithmes calculent les valeurs propres d'une matrice compagne dont $\mathbf{P}(x)$ est le polynôme caractéristique. Le deuxième algorithme (`roots_balhqr`) effectue en plus un balancement préalable de la matrice, ce qui permet de surmonter des cas difficiles. L'algorithme de calcul des valeurs propres est le même que celui qui est utilisé dans le fichier `eigen.c`.