Record Number: 122260
Author, Monographic: Fortin, V.
Author Role:
Title, Monographic: Une librairie d'objets C++ pour l'arithmétique et la régression floue
Translated Title:
Reprint Status:
Edition:
Author, Subsidiary:
Author Role:
Place of Publication: Québec
Publisher Name: INRS-Eau, Terre & Environnement
Date of Publication: 1996
Original Publication Date: 11 juin 1996
Volume Identification:
Extent of Work: iii, 75
Packaging Method: pages incluant 2 annexes et une disquette
Series Editor:
Series Editor Role:
Series Title: INRS-Eau, rapport de recherche
Series Volume ID: 468
Location/URL:
ISBN: 2-89146-440-0
Notes: Rapport annuel 1996-1997
Abstract: Dépot BNC et BNQ, prix 9.00 $
Call Number: R000468
Keywords: rapport/ ok/ dl

UNE LIBRAIRIE D'OBJETS C++ POUR L'ARITHMÉTIQUE ET LA RÉGRESSION FLOUE

UNE LIBRAIRIE D'OBJETS C++ POUR L'ARITHMÉTIQUE ET LA RÉGRESSION FLOUE

par

Vincent Fortin

Institut national de la recherche scientifique, INRS-Eau
2800, rue Einstein, Case postale 7500, SAINTE-FOY (Québec), G1V 4C7

Rapport de recherche No R-468

11 juin 1996

ISBN 2-89146-440-0

TABLE DES MATIÈRES

1. INTRODUCTION

Ce rapport présente une librairie de classes d'objets C++ développée pour permettre de facilement insérer des calculs flous à l'intérieur d'un programme C++. On peut ainsi appliquer aisément les opérateurs arithmétiques courants ainsi que le modèle de régression linéaire à des nombres flous. L'interface de chaque objet de la librairie sera présentée en détail, en incluant des détails techniques qui pourraient permettre d'y apporter des modifications plus facilement. Nous assumons que le lecteur est familier avec les concepts de nombre flou et de régression floue, ainsi qu'avec la programmation orientée-objet en C++. Un projet de conférence annexé à ce rapport présente les principes de l'arithmétique et de la régression floue, et donne les principales références bibliographiques. Une seconde annexe propose une interface-utilisateur pour un logiciel de régression floue, qui pourrait être réalisé en C++ ou à l'aide de MATLAB.

2. STRUCTURE DE LA LIBRAIRIE

Les objectifs suivants étaient poursuivis lors de la conception de la librairie:

1. **transparence**: un nombre flou devait pouvoir être manipulé de la même façon qu'un nombre réel;
2. **efficacité**: une représentation interne permettant des calculs rapides et utilisant une quantité minimale de mémoire devrait être utilisée;
3. **portabilité**: le code doit être indépendant du compilateur et de la machine utilisée.

Pour respecter ces objectifs, nous avons opté pour le langage orienté-objet C++, qui permet de créer un type de données (une classe) pour représenter un nombre flou et définir les opérateurs s'y rapportant. S'il n'existe pas encore de standard définitif pour ce langage, en particulier en ce qui concerne les gabarits (*template*), le compilateur Borland C++ 4.5 était au moment de concevoir la librairie très près du standard proposé; c'est pourquoi nous avons opté pour ce compilateur.

Pour pouvoir utiliser une librairie de classes d'objets C++, il faut comprendre l'interface de celles-ci, qui correspond aux fichiers entêtes (*header files*). Nous présenterons ces fichiers en détail, mais il est important auparavant de développer une vision d'ensemble de la librairie et des liens entre les classes la composant. La Figure 1 présente la hiérarchie des classes. On compte 15 classes, dont trois sont génériques (array, table et fuzzy_reg), et trois sont abstraites (fuzzy, fuzzy_convex et graph).

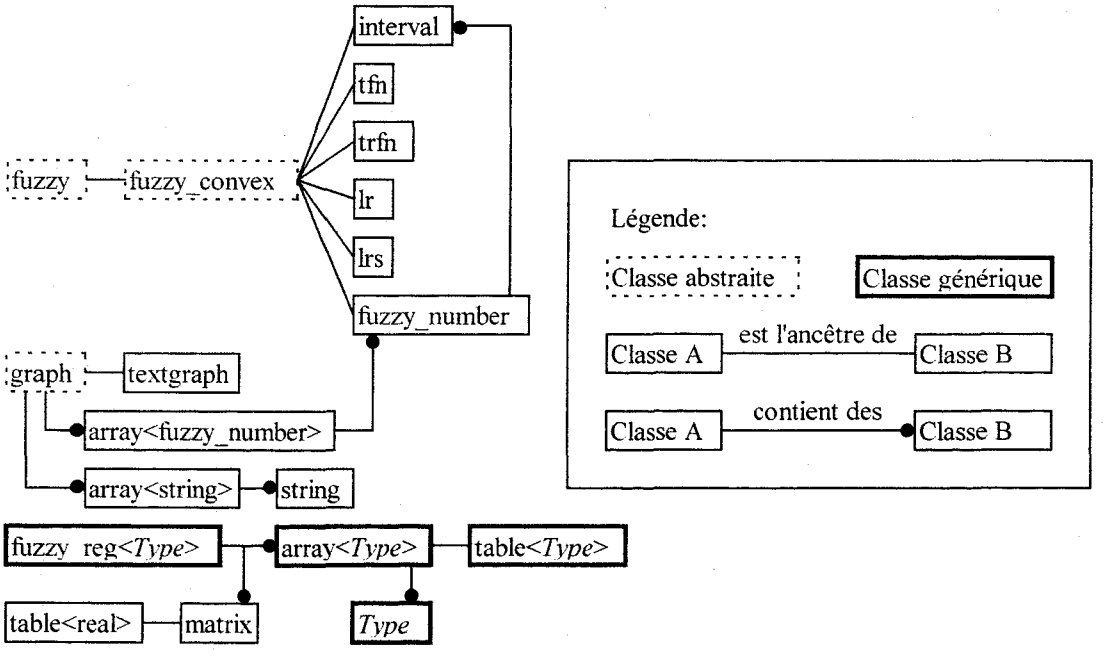


Figure 1: Hiérarchie des classes

Pour les besoins de ce travail, nous avons développé une série de classes permettant de gérer des matrices d'objets. La classe générique "array<*Type*>" permet de stocker un ensemble d'objets de même type "*Type*". le descendant "table<*Type*>" de cette classe permet de stocker des objets numériques, par exemple des nombres réels ou des nombres flous: on doit pouvoir effectuer des opérations arithmétiques sur les objets de type "*Type*" pour pouvoir les stocker dans une "table". Enfin, la classe "matrix" permet de stocker des nombres réels, de type "real", un type défini par l'usager qui correspond soit à un réel simple précision (float) ou à un réel double précision (double).

La classe "fuzzy" permet de représenter des ensembles flous. Cette classe a un descendant spécialisé, "fuzzy_convex", qui permet de représenter des ensembles flous convexes. On distingue plusieurs types d'ensembles flous convexes: l'intervalle (classe "interval"), le nombre flou triangulaire (classe "tfn"), le nombre flou trapézoïdal (classe "tfn"), le nombre L-R de Dubois et Prade (classe "lr"), le nombre L-R de Bárdossy (classe "lrs") et enfin le nombre flou général (classe "fuzzy_number"), modélisé par un ensemble d'intervalles.

La classe "graph" permet de tracer sur un même graphique les fonctions d'appartenance de plusieurs nombres flous. Un objet de cette classe maintient donc une liste des nombres flous à tracer (array<fuzzy_number>), et une liste des identificateurs de ces nombres pour pouvoir imprimer une légende (array<string>). La classe "string" est simplement une encapsulation d'une chaîne de caractères; elle permet de les manipuler plus facilement.

Finalement, la classe "fuzzy_reg<*Type*>" permet d'appliquer le modèle de régression linéaire flou. Le type "*Type*" correspond au type des paramètres flous du modèle, soit "tfn", "trfn", "lr", "lrs" ou "fuzzy_number".

Nous pouvons maintenant décrire en détail l'interface de chacune des classes, en étudiant les fichiers entêtes.

3. INTERFACE DE LA LIBRAIRIE

Avant de décrire l'interface de chacune des classes présentées à la Figure 1, nous présenterons les quelques fichiers entêtes non reliés à des classes. Il s'agit des fichiers suivants:

- errors.h: références aux structures contenant les messages d'erreur du système;
- floating.h: définition du type "real";
- template.h: définition de quelques gabarits d'utilisation générale;
- typedef.h: définition de quelques types de base.

Le seul fichier dont la compréhension est importante pour un usager de la librairie est "floating.h". Il s'agit aussi du plus simple. Les sections identifiées comme étant "techniques" ne sont pas utiles à un usager de la librairie, mais uniquement à un programmeur désirant la modifier.

3.1 Entête "errors.h" (technique)

Ce fichier d'entête crée des références globales aux structures contenant les messages d'erreur du système. Un usager de la librairie n'a pas besoin de connaître ces structures. On y retrouve les définitions suivantes:

```
const nberrmes=20;
const nbmes=20;

extern char *errmes[nberrmes+1];
extern char *message[nbmes];
```

Le système peut stocker dans les structures "errmes" et "message" respectivement "nberrmes" et "nbmes" messages d'erreur et avertissements. Une case supplémentaire est allouée dans la structure "errmes" pour toujours laisser la place pour le message "erreur inconnue", placé à l'adresse errmes[nberrmes]. En ce moment, le système utilise réellement 6 espaces pour les messages d'erreurs et 0 pour les avertissements. Si jamais des modifications à la librairies nécessitaient plus de 20 messages d'erreurs ou plus de 20 avertissements, il faudrait modifier les constantes "nberrmes" et "nbmes" en conséquence.

3.2 Entête "floating.h"

Ce fichier d'entête contient la définition du type "real", utilisé fréquemment dans la librairie. La seule définition qu'on y retrouve est la suivante:

```
typedef double        real;
```

Ceci permet de spécifier que tous les calculs doivent se faire en double précision. Pour réduire le temps d'exécution et la précision des calculs (et donc l'espace mémoire

nécessaire à l'exécution d'un programme utilisant la librairie), on peut remplacer cette ligne par:

```
typedef float        real;
```

Si une plus grande précision est nécessaire, on peut aussi utiliser des "long double" mais certaines fonctions mathématiques complexes seront tout de même effectuées en précision "double". Ceci vient du fait que certaines fonctions C++ opérant sur des "long double" ne sont pas transportables.

3.3 Entête "template.h" (technique)

Ce fichier d'entête définit quelques gabarits couramment utilisés:

template <class T1, class T2> T1 max(T1 x[], T2 n)	retourne le maximum de la liste "x" de "n" éléments;
template <class T1, class T2> T1 min(T1 x[], T2 n)	retourne le minimum de la liste "x" de "n" éléments;
template <class T1, class T2> T1 avg(T1 x[], T2 n)	retourne la moyenne de la liste "x" de "n" éléments;
template <class T1, class T2> T1 mid(T1 x[], T2 n)	retourne le milieu de l'étendue de la liste "x" de "n" éléments, donc la moitié de min(x,n)+max(x,n);
template <class T1, class T2> T2 posmax(T1 x[], T2 n)	retourne la position du maximum de la liste "x" de "n" éléments;
template <class T1, class T2> T2 posmin(T1 x[], T2 n)	retourne la position du minimum de la liste "x" de "n" éléments;
template <class T> T max(T x, T y)	retourne le maximum de "x" et de "y";
template <class T> T min(T x, T y)	retourne le minimum de "x" et de "y";
template <class T> T avg(T x, T y)	retourne la moyenne de "x" et de "y";
template <class T> int sign (T a)	retourne le signe de a, i.e. |a|/a et 1 si "a"=0;
template <class T> void swap(T &a, T &b)	échange les contenus de "a" et "b";
template <class T> T sqr(T x)	retourne le carré de "x";
template <class T> T appop(T a, char o, T b)	retourne le résultat de l'application de l'opérateur "o"sur "a" et "b"; "o" peut être '+', '-', '*', '/' ou '^';
template <class T> T interpol(T x0, T x1, T y0, T y1, T x)	sachant que y0=f(x0) et y1=f(x1), retourne y=f(x) en supposant f lineaire; si x0=x1, on ne peut calculer y, on retourne alors max(y0,y1);
template <class T1, class T2> void tribulle(modetri mode, T2 n, T1 x[])	trie les "n" observations du vecteur "x" en ordre croissant si "mode"=croissant, en ordre décroissant si "mode"=decroissant;
template <class T1, class T2> T1 median(T1 x[], T2 n)	retourne la médiane de la liste "x" de "n" éléments.

3.4 Entête "typedef.h" (technique)

Ce fichier d'entête définit quelques types de base utilisés à l'intérieur de la librairie. On y retrouve les définitions suivantes:

```
typedef unsigned charbyte;
typedef unsigned int  word;
typedef int     boole;
const  true    =1;
const  false   =0;
```

On donne respectivement aux types "unsigned char" et "unsigned int" les noms abrégés "byte" et "word". On définit ensuite le type "boole" comme étant "int", et les constantes "true" et "false" à leur valeur usuelle. Les autres fichiers entête de la librairies décrivent tous des classes. Nous pouvons maintenant les présenter.

3.5 Classe générique "array" (entête "array.h")

Cette classe permet de stocker un tableau à une ou deux dimensions d'objets de même type (générique). La taille de ce tableau peut être changée dynamiquement, ce qui constitue un avantage important sur les tableaux standards du C++. Cette classe peut aussi contrôler l'accès à ses membres et s'assurer que les indices ne dépassent jamais les bornes du tableau.

3.5.1 Attributs d'instance (technique)

La classe "array" comporte 4 attributs d'instance protégés: m, n, flags et mat, dont voici la définition:

```
unsigned m,n;              // m=nombre de lignes, n=nombre de colonnes
struct {
        word errcal:1;     // indique les erreurs de calcul (1=erreur, 0=pas d'erreur)
        word copie:1;      // indique si c'est une copie (1=copie, 0=original)
        word verbose:1;    // indique si on imprime les erreurs (1=oui, 0=non)
        word erreur:8;     // specifie le no du message d'erreur
} flags;
Type *mat;                 // pointeur aux valeurs du tableau
```

Les attributs "m" et "n" donnent respectivement le nombre de lignes et de colonnes du tableau. La structure "flags" permet de spécifier l'état du tableau. Lorsqu'une opération non conforme est effectuée sur un tableau, "errcal" est positionné à 1 et "erreur" indique le numéro de l'erreur. Le message d'erreur correspondant est alors affiché seulement si "verbose" vaut 1.

L'indicateur "copie" permet de savoir si le tableau est l'original ou la copie d'un autre tableau. Cet indicateur est utilisé par l'opérateur []. On peut en effet manipuler une colonne d'un tableau existant comme s'il s'agissait d'un objet indépendant, à l'aide de l'opérateur []. Par exemple A[1] réfère à la première colonne du tableau A. L'opérateur []

retourne une colonne de A et indique que cet objet est une copie de l'original. En effet, comme l'objet A[1] pointe vers les données de A, on doit le traiter de façon spéciale: lors de sa destruction, on ne doit pas détruire ses données. Ce n'est que lorsque A est détruit que ces données doivent être éliminées. L'indicateur "copie" influence donc le destructeur de la classe "array".

L'attribut "mat" est simplement un pointeur aux données du tableau, et est de type "real", un type défini par l'usager. Évidemment, l'usager de la librairie n'a pas besoin de gérer les attributs de cette classe, puisqu'ils sont protégés.

3.5.2 Attribut de classe (technique)

La classe "array" comporte un seul attribut de classe "repint", qui indique le numéro du code d'erreur à placer dans l'indicateur "erreur" lorsque se produit une erreur de représentation interne (i.e. lorsque l'objet ne peut effectuer la transformation demandée sur ses éléments, en général parce que les arguments de la méthode appelée sont incorrectement spécifiés). La définition de "repint" est:

static const byte repint;

3.5.3 Méthodes d'instance protégées (technique)

La classe "array" comporte 5 méthodes d'instance protégées permettant d'initialiser des objets de cette classe et de gérer les erreurs. En voici une courte description:

void init(void);	met à zéro tous les attributs d'instance de la classe;
void allouer(unsigned i, unsigned j);	alloue un tableau de m=i lignes par n=j colonnes;
void unseterr();	remet "errcal" et "erreur" à zéro pour oublier l'erreur
boole verbose(void);	retourne l'indicateur "verbose";
boole verbose(boole etat);	met l'indicateur "verbose" à la valeur "état" et retourne "état".

3.5.4 Constructeurs et desctructeur

La classe "array" comporte 3 constructeurs et un destructeur, tous très simples:

array(void);	construit un tableau vide (m=0 et n=0);
array(unsigned i, unsigned j=1);	construit un tableau de m=i lignes par n=j colonnes;
array(const array &a);	copie-constructeur;
virtual ~array(void);	détruit le tableau si copie=0, sinon ne fait rien.

3.5.5 Méthodes d'instance publiques

La classe "array" comporte un bon nombre de méthodes d'instance, toutes relativement simples, que nous avons classé en 5 groupes:

3.5.5.1 Gestion d'erreur

virtual char *errmes() const;	retourne un message correspondant à l'état de la matrice;

void err(unsigned char no);	indique à la matrice que l'erreur "no" s'est produite; affiche le message correspondant si "verbose"=1;
void err(const array &a);	copie l'erreur de la matrice "a" dans l'objet; affiche le message correspondant si "verbose"=1;
boole err(void) const;	retourne "errcal", l'état de l'objet;
boole errno(void) const;	retourne "erreur", le numéro de l'erreur;
virtual boole ok(void);	indique l'état de la matrice: vaut 1 s'il n'y a pas d'erreur ("errcal"=0) et que la matrice contient quelque chose ("mat" n'est pas NULL).

3.5.5.2 Modification de la taille de la matrice

void size(unsigned i, unsigned j=1);	change les dimensions de la matrice pour m=i et n=j; le contenu est préservé; "mat" est mis à NULL s'il n'y a pas suffisamment de mémoire pour l'opération;
void size(unsigned i, unsigned j, const Type &x);	change les dimensions de la matrice pour m=i et n=j, et initialise les nouveaux éléments à x; le contenu est préservé; "mat" est mis à NULL s'il n'y a pas suffisamment de mémoire pour l'opération;
void add(const array &a, elem_array=colonnes);	augmente l'objet de la matrice a, en augmentant le nombre de colonnes si "elem_array" vaut "colonnes", et en augmentant le nombre de lignes si "elem_array" vaut "lignes".

3.5.5.3 Modification du contenu de la matrice

array &operator =(const array &a);	remplace le contenu par celui de la matrice "a"; change la taille de la matrice si nécessaire;
void symetrie(void);	rend la matrice symétrique en copiant tout ce qui est au dessus de la diagonale en-dessous;
void fill(const Type &x);	remplit la matrice de "x".

3.5.5.4 Accès aux éléments de la matrice

unsigned l(void) const;	retourne "m", le nombre de lignes;
unsigned c(void) const;	retourne "n", le nombre de colonnes;
Type &operator()(unsigned i, unsigned j) const;	retourne l'élément situé à la ligne i et la colonne j; cause une erreur d'exécution si (i,j) est à l'extérieur des limites de la matrice;
Type &operator()(unsigned i) const;	retourne l'élément situé à la ligne i et la colonne 1; cause une erreur d'exécution si (i,1) est à l'extérieur des limites de la matrice;
Type &operator()(int i, int j) const;	retourne l'élément situé à la ligne i et la colonne j; cause une erreur d'exécution si (i,j) est à l'extérieur des limites de la matrice;

Type &operator()(unsigned i) const;	retourne l'élément situé à la ligne i et la colonne 1; cause une erreur d'exécution si (i,1) est à l'extérieur des limites de la matrice;
Type &operator()(int i, unsigned j) const;	retourne l'élément situé à la ligne i et la colonne j; cause une erreur d'exécution si (i,j) est à l'extérieur des limites de la matrice;
Type &operator()(unsigned i, int j) const;	retourne l'élément situé à la ligne i et la colonne j; cause une erreur d'exécution si (i,j) est à l'extérieur des limites de la matrice;
array<Type> &operator[](unsigned i) const;	retourne la colonne i de la matrice;
array<Type> &operator[](int i) const;	retourne la colonne i de la matrice.

3.5.5.5 Opérateurs logiques

boole operator ==(const array &a);	retourne 1 si l'objet est identique à "a", 0 autrement;
boole operator !=(const array &a);	retourne 1 si l'objet est différente de "a", 0 autrement.

3.5.6 Fonctions amies

Ces fonctions permettent d'effectuer aisément la lecture et l'écriture d'une matrice:

friend ostream &operator <<(ostream &f, const array<Type> &a);
imprime la matrice "a";

friend istream &operator >>(istream &f, array<Type> &a);
lit la matrice "a".

3.6 Classe générique "table" (entête "table.h")

Cette classe générique, qui hérite de la classe générique "array", spécialise son ancêtre en se consacrant à la gestion d'éléments numériques. Une "table" ne contient donc que des nombres, c'est à dire des objets pour lesquels les opérations arithmétiques courantes sont définies. Il peut donc s'agir de nombres réels ou de nombres flous. Cette classe ne définit pas de nouveaux attributs, seulement de nouvelles méthodes.

3.6.1 Constructeurs

Les constructeurs de la classe générique "array" sont redéfinis pour plaire au compilateur, mais sans rien changer:

table(void) : array<Type>()	construit une "table" vide (0x0)
table(unsigned i, unsigned j=1): array<Type>(i,j) {}	construit une "table" de m=i par n=j
table(const table &a): array<Type>(a) {}	copie-constructeur

3.6.2 Méthodes d'instance

Les méthodes d'instance de l'objet "table" sont simples; nous en avons fait 3 catégories.

3.6.2.1 Accès aux éléments de la matrice

unsigned l(void) const;	retourne "m", le nombre de lignes;
unsigned c(void) const;	retourne "n", le nombre de colonnes;
table l(unsigned i) const;	retourne une copie de la ligne i de la matrice;
table l(int i) const;	retourne une copie de la ligne i de la matrice;
table c(unsigned j) const;	retourne une copie de la colonne j de la matrice; diffère de l'opérateur []: la méthode "c" fournit une copie indépendante; les modifications dans cette copie ne seront pas reflétées dans l'original;
table c(int j) const;	retourne une copie de la colonne j de la matrice; diffère de l'opérateur []: la méthode "c" fournit une copie indépendante; les modifications dans cette copie ne seront pas reflétées dans l'original.

3.6.2.2 Opérateurs arithmétiques

table &operator +=(const table&);	
table &operator -=(const table&);	
table &operator *=(const table&);	
table &operator +=(const Type&);	
table &operator -=(const Type&);	
table &operator *=(const Type&);	
table &operator /=(const Type&);	
table &operator ^=(int);	
table operator +(const table&) const;	
table operator -(const table&) const;	
table operator *(const table&) const;	
table operator +(const Type&) const;	
table operator -(const Type&) const;	
table operator *(const Type&) const;	
table operator /(const Type&) const;	
table operator ^(int r) const;	si "r" est égal à la constante "T", alors la matrice est transposée;
table operator -(void) const;	
Type max(void) const;	retourne le maximum de tout le tableau;
Type min(void) const;	retourne le minimum de tout le tableau;
Type avg(void) const;	retourne la moyenne de tout le tableau;
Type mid(void) const;	retourne le milieu de l'étendue du tableau;
Type median(void) const;	retourne la médiane du tableau.

3.6.2.3 Modification du contenu de la matrice

void transformation(real r);	élève chaque élément du tableau à la puissance "r";
void identity(void);	transforme le tableau en matrice identité;
void zero(void);	remplit le tableau de zéros.

3.6.3 Fonctions utilisant la classe "table"

template <class Type> Type max(const table<Type> &a);	retourne le maximum de la table "a";
template <class Type> Type min(const table<Type> &a);	retourne le minimum de la table "a";
template <class Type> Type avg(const table<Type> &a);	retourne la moyenne de la table "a";
template <class Type> Type mid(const table<Type> &a);	retourne le milieu de l'étendue du tableau, soit la moitié de max(a)+min(a);
template <class Type> Type median(const table<Type> &a);	retourne la médiane du tableau "a".

3.7 Classe "matrix" (entête "matrix.h")

La classe "matrix" spécialise la classe générique "table" en l'appliquant à des nombres réels de type "real". Plusieurs fonctions spécifiques aux matrices de nombres réels sont ajoutées: l'inversion de matrice, la factorisation LU, et la programmation linéaire.

3.7.1 Attributs de classe publics

La classe "matrix" comporte 5 nouveaux attributs de classe, mais aucun nouvel attribut d'instance. 4 d'entre eux correspondent simplement à des numéros de messages d'erreur; un seul pourrait être manipulé par l'utilisateur: il s'agit de "eps", un attribut qui spécifie l'écart accepté entre la solution exacte et la solution approximative trouvée à l'aide de la méthode du simplex. Il doit s'agir toujours d'une faible valeur positive. Voici la définition des attributs de classe:

static real eps;	epsilon pour le simplex;
static const byte detnul;	code d'erreur correspondant à un déterminant nul;
static const byte x0faux;	code d'erreur correspondant à un point de départ non faisable pour la méthode du simplex;
static const byte nosol;	code d'erreur indiquant que le système d'équations à résoudre n'a pas de solution;
static const byte tropeq;	code d'erreur indiquant que le système d'inéquations à résoudre est trop complexe pour l'algorithme du simplex généralisé.

3.7.2 Méthodes d'instance protégées (technique)

Deux méthodes d'instance reliées à la procédure du simplex sont protégées, car peu utiles pour l'usager. La première, "extremefaisable", vérifie si un point du domaine est extrême faisable (i.e. s'il fait partie de la limite du domaine respectant les contraintes). La seconde, "simplex", applique la procédure du simplex une fois que la base de départ a été construite. Cette méthode est utilisée par plusieurs versions accessibles à l'usager de la procédure du simplex. Voici la définition de ces deux méthodes d'instance protégées:

boole extremefaisable(const matrix&, const matrix&, real) const;
matrix simplex(const matrix&, const matrix&, const matrix&, const table<int>&) const;

3.7.3 Constructeurs

Encore une fois, les constructeurs des ancêtres de "matrix" sont redéfinis pour plaire au compilateur:

matrix(void);	construit une matrice vide;
matrix(unsigned i, unsigned j=1);	construit une matrice de m=i par n=j;
matrix(const table<real> &x);	copie-constructeur;
matrix(const matrix &x);	copie-constructeur.

3.7.4 Méthodes d'instance publiques

La plupart des méthodes de la classe "matrix" sont complexes. Elles implantent l'inversion de matrice, la factorisation LU et la programmation linéaire.

3.7.4.1 Gestion d'erreur

virtual char *errmes() const; retourne un message correspondant à l'état de la matrice.

3.7.4.2 Modification de la taille de la matrice

void size(unsigned i, unsigned j=1, real x=real(0)); change les dimensions de la matrice pour m=i et n=j, et initialise les nouveaux éléments à x; le contenu est préservé; "mat" est mis à NULL s'il n'y a pas suffisamment de mémoire pour l'opération.

3.7.4.3 Accès aux éléments de la matrice

matrix &operator[](unsigned i) const;	retourne la colonne i de la matrice;
matrix &operator[](int i) const;	retourne la colonne i de la matrice.

3.7.4.4 Opérateurs arithmétiques

matrix &operator ^=(int i);	si "i" est négatif la matrice est d'abord élevée à la puissance |i| puis inversée;
matrix operator ^(int i) const;	si "i" est négatif la matrice est d'abord élevée à la puissance |i| puis inversée; si "i" est égal à la constante "T", la matrice est simplement transposée.

3.7.4.5 Inversion de matrice et résolution de système d'équations linéaires

Factorisation LU:

virtual void LU(matrix &lu, table<int> &perm, int &signe) const;

Décompose la matrice (*this) en une matrice L et une matrice U de telle sorte que "lu"=L*U soit une permutation des lignes de (*this) (la permutation des lignes est gardée dans le vecteur "perm") (*this) doit être une matrice nxn, tout comme "lu"; "perm" est un vecteur de "n" éléments; signe est égal à 1 si le nombre de permutations est pair, 0 sinon. Inspiré de "Numerical Recipes in C", W.H. Press (1988).

Codes d'erreur gardé dans lu.erreur:
repint: la matrice n'est pas carré, il manque de mémoire, ou "perm" ne comporte pas n lignes;
detnul: la matrice est singulière (déterminant=0).

Résolution d'un système d'équations factorisé:
virtual matrix LUresolution(const table<int> &perm, const matrix &b) const;

Retourne la solution "x" du système linéaire A*x=b, où (*this) est la factorisation LU de A, et "perm" est le vecteur de permutations retourné par la méthode LU lors de la factorisation.

Résolution d'un système d'équations linéaires par factorisation LU:
virtual matrix resolution(const matrix &b) const;
Retourne la solution "x" du système linéaire (*this)*x=b, en procédant par factorisation LU.

virtual void inverse(void);	inverse la matrice;
virtual real determinant(void) const;	retourne le déterminant de la matrice.

3.7.4.6 Programmation linéaire

La programmation linéaire permet de résoudre des systèmes du type A*x<=b.
Pour éviter des difficultés numériques, les tests d'égalité du type v=w sont remplacés par |v-w|<e. Il faut donc choisir une valeur de "e" correcte pour le problème.

Code d'erreur gardé dans x.erreur:

repint:	erreur de représentation interne ou manque de mémoire;
detnul:	problème de multicollinéarité entre les contraintes;
x0faux:	x0 n'est pas un point extrême faisable
nosol:	pas de solution

Algorithme du simplex avec point de départ extrême faisable connu:
virtual matrix simplex(const matrix &b, const matrix &c, const matrix &x0, real e=eps) const;

Détermine la solution x minimisant c'x telle que (*this)*x<=b et x>=0. Il s'agit de la méthode itérative classique, qui commence avec un point de départ x0 qui doit être extrême faisable, c'est à dire être situé sur la limite du domaine respectant les contraintes.

Algorithme du simplex avec point de départ extrême faisable inconnu:
virtual matrix simplex(const matrix &b, const matrix &c, real e=eps) const;

Détermine la solution x minimisant c'x telle que (*this)*x<=b et x>=0, sans qu'il soit nécessaire de fournir un point de départ. L'algorithme cherche lui-même un point de départ extrême faisable par énumération. Peut être très lent. Il est recommandé d'utiliser "approx_simplex" pour augmenter la vitesse; les pertes en précision sont minimales.

Algorithme approximatif du simplex
virtual matrix approx_simplex(const matrix &b, const matrix &c, real e=eps) const;

Détermine approximativement la solution x minimisant c'x telle que (*this)*x<=b et x>=0, sans qu'il soit nécessaire de fournir un point de départ. L'algorithme ajoute des variables pour créer un point de départ extrême faisable, et modifie la fonction de coût c'x de façon à ce que ces nouvelles variables soient très près de zéro pour une solution optimale. Il s'agit de l'approche classique.

Algorithme du simplex généralisé
virtual matrix generalized_simplex(const matrix &b, const matrix &c, unsigned nbpos=0, real e=eps) const;

Détermine approximativement la solution x minimisant c'x telle que (*this)*x<=b. La contrainte x>=0 n'est maintenue que pour les dernières "nbpos" variables (par défaut aucune). L'algorithme procède en résolvant $2^{n-nbpos}$ problèmes du simplex (par l'algorithme "approx_simplex" pour gagner du temps), où "n" est le nombre de variables.

3.7.5 Fonction amie

friend matrix I(unsigned i,unsigned j);	retourne la colonne "j" d'une matrice identité comportant "i" lignes et "i" colonnes.

3.7.6 Fonctions utilisant la classe "matrix"

real det(const matrix &a);	retourne le déterminant de la matrice "a";
matrix I(unsigned i);	retourne une matrice identité de "i" par "i";
matrix zero(unsigned i,unsigned j=1);	retourne une matrice nulle de "i" par "j";
real min(const matrix &a);	retourne le minimum de la matrice "a";
real max(const matrix &a);	retourne le maximum de la matrice "a";
real avg(const matrix &a);	retourne la moyenne de la matrice "a";
real mid(const matrix &a);	retourne le milieu de l'étendue de "a";
real median(const matrix &a);	retourne la médiane de la matrice "a".

3.8 Classe abstraite "fuzzy" (entête "fuzzy.h")

La classe "fuzzy" est très simple. Il s'agit d'un ancêtre abstrait de tous les ensembles flous. La classe ne dispose d'aucun attribut, et de seulement deux méthodes d'instance et deux fonctions amies. On définit tout de même avec celles-ci les rudiments d'une gestion d'erreur, d'une fonction d'appartenance, ainsi que les opérations d'entrée-sortie.

3.8.1 Méthodes d'instance publiques

Les méthodes d'instance de cette classe doivent obligatoirement être redéfinies par les descendants.

virtual boole ok(void) const =0;	retourne l'état de l'ensemble flou; permet d'indiquer les erreurs de représentation interne;
virtual real operator ()(real x) const =0;	retourne la valeur de la fonction d'appartenance de l'ensemble flou évaluée en "x".

3.8.2 Fonctions amies

Ces fonctions amies servent à définir la structure des opérations d'entrée-sortie, et doivent être redéfinies par les descendants.

friend ostream &operator <<(ostream&, const fuzzy&);	imprime "Erreur inconnue";
friend istream &operator >>(istream& f, fuzzy&);	retourne simplement "f".

3.9 Classe abstraite "fuzzy_convex" (entête "fuzzy.h")

La classe "fuzzy_convex" est aussi une classe abstraite. Il s'agit d'une spécialisation de "fuzzy" qui englobe tous les ensembles flous qui peuvent être décomposés en une série d'intervalles emboîtés. Tous les descendants de cette classe sont des nombres flous. Cette classe ne comporte aucun attribut, et toutes ses méthodes sont publiques.

3.9.1 Méthodes d'instance publiques

Les méthodes d'instance de la classe "fuzzy_convex" définissent la structure des accès aux intervalles composant un ensemble flou convexe, en plus de permettre quelques opérations logiques (binaires - non floues) de comparaison d'ensembles flous convexes.

3.9.1.1 Méthodes permettant l'accès aux intervalles composant un ensemble convexe

virtual interval operator [](real alpha) const =0;	retourne l'intervalle de niveau alpha;
virtual interval supp() const =0;	retourne le support de l'ensemble convexe (qui correspond à l'intervalle de niveau zéro).

3.9.1.2 Opérateurs logiques binaires

boole operator <(const fuzzy_convex &a) const;	indique si tous les éléments de l'ensemble sont strictement plus petits que tous les éléments de "a";
boole operator >(const fuzzy_convex &a) const;	indique si tous les éléments de l'ensemble sont strictement plus grands que tous les éléments de "a";
boole operator <=(const fuzzy_convex &a) const;	indique si tous les éléments de l'ensemble sont plus petits que tous les éléments de "a";
boole operator >=(const fuzzy_convex &a) const;	indique si tous les éléments de l'ensemble sont plus petits que tous les éléments de "a";

boole operator <(real x) const;	indique si tous les éléments de l'ensemble sont strictement plus petits que "x";
boole operator >(real x) const;	indique si tous les éléments de l'ensemble sont strictement plus grands que "x";
boole operator <=(real x) const;	indique si tous les éléments de l'ensemble sont plus petits que "x";
boole operator >=(real x) const;	indique si tous les éléments de l'ensemble sont plus grands que "x".

3.9.2 Fonctions utilisant la classe "fuzzy_convex"

Ces fonctions consistent en la redéfinition de quatre opérateurs logiques permettant de comparer un nombre réel avec un ensemble flou convexe. On retrouve aussi la fonction "supp" qui retourne le support d'un ensemble flou convexe.

boole operator <(real x, const fuzzy_convex &a);	indique si "x" est strictement plus petit que tous les éléments de l'ensemble "a";
boole operator >(real x, const fuzzy_convex &a);	indique si "x" est strictement plus grand que tous les éléments de l'ensemble "a";
boole operator <=(real x, const fuzzy_convex &a);	indique si "x" est plus petit que tous les éléments de l'ensemble "a";
boole operator >=(real x, const fuzzy_convex &a);	indique si "x" est plus grand que tous les éléments de l'ensemble "a";
interval supp(const fuzzy_convex &a);	retourne le support de l'ensemble "a".

3.10 Classe "interval" (entête "interval.h")

Un intervalle peut-être vu comme un cas particulièrement simple d'un nombre flou. Par ailleurs, comme nous le verrons plus tard, un nombre flou complexe peut être décomposé en intervalles.

3.10.1 Attributs d'instance protégés (technique)

La classe "interval" comporte deux attributs d'instance "l" et "u", qui correspondent à la borne inférieure et à la borne supérieure de l'intervalle:

```
real l, u;
```

3.10.2 Objet de la classe "interval"

Un objet de type "interval", nommé "interr" est retourné par plusieurs méthodes de la classe "interval" et de d'autres classes lorsqu'une erreur se produit dans le traitement. Il est défini par l=LONG_MAX et u=LONG_MIN.

3.10.3 Constructeurs

La classe "interval" comporte quatre constructeurs:

interval(void);	construit l'intervalle [0,0];
interval(real a, real b);	construit l'intervalle [a,b];
interval(pair p);	"pair" étant défini comme un tableau de 2 "real", construit l'intervalle [p[0],p[1]];
interval(const interval&);	copie-constructeur.

3.10.4 Méthodes d'instance publiques

Les méthodes d'instance de toutes les classes de nombres flous sont nombreuses mais relativement simples. Nous les avons classées pour plus de clarté.

3.10.4.1 Gestion d'erreur

virtual boole ok(void) const;	retourne 1 si "l"<="u", 0 autrement.

3.10.4.2 Méthodes permettant l'accès et la modification des attributs

real binf(void) const;	retourne "l", la borne inférieure de l'intervalle;
real bsup(void) const;	retourne "u", la borne supérieure de l'intervalle;
real binf(real bi);	affecte à "l" la valeur "bi" et retourne "l";
real bsup(real bs);	affecte à "u" la valeur "bs" et retourne "u";
real &operator [](int i);	retourne "l" si "i" vaut 0 et "u" si "i" vaut 1, retourne "interr" pour d'autres valeurs de "i";
real &operator [](unsigned i);	retourne "l" si "i" vaut 0 et "u" si "i" vaut 1, retourne "interr" pour d'autres valeurs de "i";
virtual interval supp() const;	retourne le support de l'intervalle, soit l'intervalle lui-même;
virtual interval operator [](real alp) const;	retourne la coupe de niveau "alp" de l'intervalle, soit l'intervalle lui même si 0<="alp"<=1, "interr" sinon;
virtual real operator ()(real x) const;	retourne l'appartenance (0 ou 1) de "x" à l'intervalle;
interval &operator =(const interval &x);	copie l'intervalle "x";
interval &operator =(real x);	construit l'intervalle [x,x].

3.10.4.3 Opérateurs logiques

boole operator ==(const interval&) const;
boole operator !=(const interval&) const;
boole operator ==(real) const;
boole operator !=(real) const;

3.10.4.4 Opérateurs arithmétiques

interval &operator +=(const interval&);

```
interval &operator -=(const interval&);
interval &operator *=(const interval&);
interval &operator /=(const interval&);
interval &operator ^=(const interval&);
interval &operator +=(real);
interval &operator -=(real);
interval &operator *=(real);
interval &operator /=(real);
interval &operator ^=(real);
interval operator +(const interval&) const;
interval operator -(const interval&) const;
interval operator *(const interval&) const;
interval operator /(const interval&) const;
interval operator ^(const interval&) const;
interval operator +(real) const;
interval operator -(real) const;
interval operator *(real) const;
interval operator /(real) const;
interval operator ^(real) const;
interval operator -(void) const;
```

3.10.5 Fonctions amies et fonctions utilisant la classe "interval"

3.10.5.1 Principe d'extension

Ces quatre méthodes permettent d'appliquer le principe d'extension à "x" et "y" pour l'opérateur "o" qui peut être soit '+', '-', '*', '/' ou '^', et retournent l'intervalle correspondant:

```
friend interval extension (const interval &x, char o, const interval &y);
friend interval extension (const interval &x, char o, real y);
friend interval extension (real x, char o, const interval &y);
real extension (real, char, real);
```

3.10.5.2 Opérateurs logiques

```
friend boole operator ==(real, const interval&);
friend boole operator !=(real, const interval&);
```

3.10.5.3 Opérateurs arithmétiques

```
friend interval operator +(real, const interval&);
friend interval operator -(real, const interval&);
friend interval operator *(real, const interval&);
friend interval operator /(real, const interval&);
friend interval operator ^(real, const interval&);
friend interval pow(interval, interval);
friend interval pow(real, const interval&);
```

```
friend interval pow(const interval&, real);
friend interval fabs(const interval&);
friend interval log(const interval&);
friend interval log10(const interval&);
friend interval exp(const interval&);
```

3.10.5.4 Opérateurs d'entrée-sortie

friend ostream &operator <<(ostream&, const interval&);	affiche un intervalle dans un fichier, sous le format "[l,u]";
friend istream &operator >>(istream&, interval&);	lit un intervalle dans un fichier, sous le format "l u".

3.10.5.5 Gestion d'erreur

Ces quatre méthodes permettent de vérifier si une opération "o" peut être appliquée sur deux intervalles ou réels "x" et "y". L'opérateur "o" peut être soit '+', '-', '*', '/' ou '^'. Par exemple, possible(x,'/',0) retourne faux parce que une division par zéro est impossible.

```
friend boole possible (const interval &x, char o, const interval &y);
friend boole possible (const interval &x, char o, real y);
friend boole possible (real x, char o, const interval &y);
boole possible (real x, char o, real y);
```

3.11 Classe "tfn" (entête "tfn.h")

La classe "tfn" permet de créer des nombres flous triangulaires, que l'on note (a,b,c)(TFN), où $[a,c]$ est le support et b le mode.

3.11.1 Attributs d'instance publics

La position d'un nombre flou triangulaire est déterminée par son support [a,c] et son mode "b", d'où les trois attributs suivants:

```
real a,b,c;
```

3.11.2 Constructeurs

La classe "tfn" comporte trois constructeurs:

tfn(real p, real q, real r);	construit le nombre (p,q,r)(TFN);
tfn(real x=0);	construit le nombre (x,x,x)(TFN);
tfn(const tfn&);	copie-constructeur.

3.11.3 Méthodes d'instance publiques

3.11.3.1 Gestion d'erreur

virtual boole ok(void) const;	retourne 1 si "a"<="b"<="c", 0 autrement.

3.11.3.2 Méthodes permettant l'accès et la modification des attributs

virtual interval supp() const;	retourne le support du nombre flou, soit [a,c];
virtual real operator ()(real x) const;	retourne l'appartenance de "x" au nombre flou;
virtual interval operator [](real alp) const;	retourne la coupe de niveau "alp" du nombre flou;
tfn &operator =(real x);	construit le nombre flou (x,x,x)(TFN);
tfn &operator =(const fuzzy_convex &x);	approxime le nombre convexe "x" par un "tfn";
tfn &operator =(const tfn &x);	copie le "tfn" x.

3.11.3.3 Opérateurs logiques

```
boole operator ==(real) const;
boole operator !=(real) const;
boole operator ==(const interval&) const;
boole operator !=(const interval&) const;
boole operator ==(const tfn&) const;
boole operator !=(const tfn&) const;
boole operator ==(const trfn&) const;
boole operator !=(const trfn&) const;
boole operator ==(const lr&) const;
boole operator !=(const lr&) const;
boole operator ==(const lrs&) const;
boole operator !=(const lrs&) const;
```

3.11.3.4 Opérateurs arithmétiques

```
tfn &operator +=(const tfn&);
tfn &operator -=(const tfn&);
tfn &operator +=(real);
tfn &operator -=(real);
tfn &operator *=(real);
tfn &operator /=(real);
tfn operator +(const tfn&) const;
tfn operator -(const tfn&) const;
tfn operator +(real) const;
tfn operator -(real) const;
tfn operator *(real) const;
tfn operator /(real) const;
trfn operator +(const trfn&) const;
trfn operator -(const trfn&) const;
tfn operator -(void) const;
```

3.11.4 Fonctions amies et fonctions utilisant la classe "tfn"

3.11.4.1 Opérateurs d'entrée-sortie

friend ostream &operator <<(ostream&, const tfn&);	affiche un "tfn" dans un fichier, sous le format "(a,b,c)(TFN)";
friend istream &operator >>(istream&, tfn&);	lit un "tfn" dans un fichier, sous le format "a b c".

3.11.4.2 Opérateurs arithmétiques

tfn operator +(real, const tfn&);
tfn operator -(real, const tfn&);
tfn operator *(real, const tfn&);

3.12 Classe "trfn" (entête "trfn.h")

La classe "trfn" permet de créer des nombres flous trapézoïdaux, que l'on note (a,b_1,b_2,c)(TRFN), où $[a,c]$ est le support et $[b_1,b_2]$ le noyau.

3.12.1 Attributs d'instance publics

La position d'un nombre flou trapézoïdal est déterminée par son support [a,c] et son noyau [b1,b2], d'où les quatre attributs suivants:

real a,b1,b2,c;

3.12.2 Constructeurs

La classe "trfn" comporte quatre constructeurs:

trfn(real p, real q1, real q2, real r);	construit le nombre (p,q1,q2,r)(TRFN);
trfn(real x=0);	construit le nombre (x,x,x,x)(TRFN);
trfn(const interval &x);	construit le nombre (x.l,x.l,x.u,x.u)(TRFN);
trfn(const tfn &x);	construit le nombre (x.a,x.b,x.b,x.c)(TRFN);
trfn(const trfn &);	copie-constructeur.

3.12.3 Méthodes d'instance publiques

3.12.3.1 Gestion d'erreur

virtual boole ok(void) const;	retourne 1 si "a"<="b1"<="b2"<="c", 0 autrement.

3.12.3.2 Méthodes permettant l'accès et la modification des attributs

virtual interval supp() const;	retourne le support du nombre flou, soit [a,c];
virtual real operator ()(real x) const;	retourne l'appartenance de "x" au nombre flou;

virtual interval operator [](real alp) const;	retourne la coupe de niveau "alp" du nombre flou;
trfn &operator =(real x);	construit le nombre flou (x,x,x,x)(TRFN);
trfn &operator =(const fuzzy_convex &x);	approxime le nombre convexe "x" par un "trfn";
trfn &operator =(const trfn&);	copie le "trfn" x.

3.12.3.3 Opérateurs logiques

```
boole operator ==(real) const;
boole operator !=(real) const;
boole operator ==(const interval &x) const;
boole operator !=(const interval &x) const;
boole operator ==(const tfn&) const;
boole operator !=(const tfn&) const;
boole operator ==(const trfn&) const;
boole operator !=(const trfn&) const;
boole operator ==(const lr&) const;
boole operator !=(const lr&) const;
boole operator ==(const lrs&) const;
boole operator !=(const lrs&) const;
```

3.12.3.4 Opérateurs arithmétiques

```
trfn &operator +=(const trfn&);
trfn &operator -=(const trfn&);
trfn &operator +=(const tfn&);
trfn &operator -=(const tfn&);
trfn &operator +=(real);
trfn &operator -=(real);
trfn &operator *=(real);
trfn &operator /=(real);
trfn operator +(const trfn&) const;
trfn operator -(const trfn&) const;
trfn operator +(const tfn&) const;
trfn operator -(const tfn&) const;
trfn operator +(real) const;
trfn operator -(real) const;
trfn operator *(real) const;
trfn operator /(real) const;
trfn operator -(void) const;
```

3.12.4 Fonctions amies et fonctions utilisant la classe "trfn"

3.12.4.1 Opérateurs d'entrée-sortie

friend ostream &operator <<(ostream&, const trfn&);	affiche un "trfn" dans un fichier, sous le format "(a,b1,b2,c)(TRFN)";
friend istream &operator >>(istream&, trfn&);	lit un "trfn" dans un fichier, sous le format "a b1 b2 c".

3.12.4.2 Opérateurs arithmétiques

trfn operator +(real, const trfn&);
trfn operator -(real, const trfn&);
trfn operator *(real, const trfn&);

3.13 Classe "lr" (entête "lr.h")

La classe "lr" supporte les nombres flous LR définis par Dubois et Prade. Ces nombres ont un noyau $[m^-,m^+]$ et un support $[m^- - \alpha, m^+ + \beta]$, donc une étendue α à gauche du noyau, et β à droite. Les valeurs de la fonction d'appartenance $\mu(x)$ entre $m^- - \alpha$ et m^-, et entre m^+ et $m^+ + \beta$ sont déterminées respectivement par deux fonctions décroissantes L et R définies sur [0,1]. Ainsi, pour $m^- - \alpha < x < m^-$, $\mu(x) = L[(m^- - x)/\alpha]$, et pour $m^+ < x < m^+ + \beta$, $\mu(x) = R[(x - m^+)/\beta]$. Il est difficile de permettre à l'usager de déterminer complètement la forme des fonctions L et R. Nous avons choisi de limiter l'usager aux formes paramètriques suivantes: $L(u) = 1 - u^{qL}$ et $R(u) = 1 - u^{qR}$. Seuls les paramètres qL et qR doivent être spécifiés pour déterminer les fonctions L et R. On dénote un nombre flou LR par $(m^-, m^+, \alpha, \beta, qL, qR)(LR)$.

3.13.1 Attributs d'instance publics

La position d'un nombre flou LR est déterminée par son noyau ["minf","msup"], l'étendue de part et d'autre de ce noyau ("alpha" et "beta"), ainsi que la forme des fonctions L et R, déterminée par les paramètres "qL" et "qR", d'où les six attributs suivants:

real minf, msup, alpha, beta, qL, qR;

3.13.2 Constructeurs

La classe "lr" comporte sept constructeurs:

lr(real mi,real ms,real a,real b,real ql=1,real qr=1);	construit le nombre flou (mi,ms,a,b,ql,qr)(LR);
lr(real x=0);	construit le nombre flou (x,x,0,0,1,1)(LR);
lr(const interval &x);	construit le nombre flou (x.l,x.u,0,0,1,1)(LR);
lr(const tfn &x);	construit le nombre flou (x.b,x.b,x.b-x.a,x.c-x.b,1,1)(LR);
lr(const trfn &x);	construit le nombre flou (x.b1,x.b2,x.b1-x.a,x.c-x.b2,1,1)(LR);
lr(const lrs &x);	construit le nombre flou (x.m,x.m,x.alpha,x.beta,x.qL,x.qR)(LR);

lr(const lr&); copie-constructeur.

3.13.3 Méthodes d'instance publiques

3.13.3.1 Gestion d'erreur

virtual boole ok(void) const; indique si les attributs du nombre LR sont cohérents.

3.13.3.2 Méthodes permettant l'accès et la modification des attributs

real L(real u) const;	retourne $L(u)$;
real R(real u) const;	retourne $R(u)$;
real Linv(real alp) const;	retourne $L^{-1}(alp)$;
real Rinv(real alp) const;	retourne $R^{-1}(alp)$;
real Larea(void) const;	retourne l'aire sous la fonction L entre 0 et 1;
real Rarea(void) const;	retourne l'aire sous la fonction R entre 0 et 1;
virtual interval supp() const;	retourne le support du nombre LR, soit l'intervalle [minf-alpha,msup+beta];
virtual real operator ()(real x) const;	retourne l'appartenance de "x" au nombre LR;
virtual interval operator [](real alp) const;	retourne la coupe de niveau "alp" du nombre LR;
lr &operator =(real x);	construit le nombre (x,x,0,0,1,1)(LR);
lr &operator =(const fuzzy_number &x);	approxime le nombre flou par un nombre LR, en respectant le support et le noyau de "x", et en optimisant qL et qR par moindres carrés;
lr &operator =(const interval &x);	construit le nombre (x.l,x.u,0,0,1,1)(LR);
lr &operator =(const tfn &x);	construit le nombre (x.b,x.b,x.b-x.a,x.c-x.b,1,1)(LR);
lr &operator =(const trfn &x);	construit le nombre (x.b1,x.b2,x.b1-x.a,x.c-x.b2,1,1)(LR);
lr &operator =(const lr &x);	construit le nombre (x.m,x.m,x.alpha,x.beta,x.qL,x.qR)(LR);
lr &operator =(const lrs &x);	copie le nombre LR "x".

3.13.3.3 Opérateurs logiques

boole operator ==(real) const;
boole operator !=(real) const;
boole operator ==(const interval&) const;
boole operator !=(const interval&) const;
boole operator ==(const tfn&) const;
boole operator !=(const tfn&) const;
boole operator ==(const trfn&) const;
boole operator !=(const trfn&) const;
boole operator ==(const lr&) const;
boole operator !=(const lr&) const;

boole operator ==(const lrs&) const;
boole operator !=(const lrs&) const;

3.13.3.4 Opérateurs arithmétiques

lr &operator +=(const lr&);
lr &operator -=(const lr&);
lr &operator +=(const lrs&);
lr &operator -=(const lrs&);
lr &operator +=(real);
lr &operator -=(real);
lr &operator *=(real);
lr &operator /=(real);
lr operator +(const lr&) const;
lr operator -(const lr&) const;
lr operator +(const lrs&) const;
lr operator -(const lrs&) const;
lr operator +(real) const;
lr operator -(real) const;
lr operator *(real) const;
lr operator /(real) const;
lr operator -(void) const;

3.13.4 Fonctions amies et fonctions utilisant la classe "lr"

3.13.4.1 Opérateurs d'entrée-sortie

friend ostream &operator <<(ostream&, const lr&);
affiche un nombre LR dans un fichier, sous le format:
"(minf,msup,alpha,beta)(LR) L(u)=1-u^qL R(u)=1-u^qR"

friend istream &operator >>(istream&, lr&);
lit un nombre LR dans un fichier, sous le format "minf msup alpha beta qL qR".

3.13.4.2 Opérateurs arithmétiques

lr operator +(real, const lr&);
lr operator -(real, const lr&);
lr operator *(real, const lr&);

3.14 Classe "lrs" (entête "lrs.h")

La classe "lrs" supporte les nombres flous LR simplifiés par Bárdossy, en supposant un noyau ponctuel, c'est à dire en posant $m^+=m^-=m$. On peut noter un nombre flou LR simplifié par (m,α,β,qL,qR)(LR).

3.14.1 Attributs d'instance publics

La position d'un nombre flou LR simplifié est déterminée par son noyau "m", l'étendue de part et d'autre de ce noyau ("alpha" et "beta"), ainsi que la forme des fonctions L et R, déterminée par les paramètres "qL" et "qR", d'où les cinq attributs suivants:

real m, alpha, beta, qL, qR;

3.14.2 Constructeurs

La classe "lr" comporte quatre constructeurs:

lrs(real m, real a, real b, real ql=1, real qr=1);	construit le nombre flou (m,a,b,ql,qr)(LR);
lrs(real x=0);	construit le nombre flou (x,0,0,1,1)(LR);
lrs(const tfn &x);	construit le nombre flou (x.b,x.b-x.a,x.c-x.b,1,1)(LR);
lrs(const lrs&);	copie-constructeur.

3.14.2.1 Gestion d'erreur

virtual boole ok(void) const;	indique si les attributs du nombre LR sont cohérents.

3.14.2.2 Méthodes permettant l'accès et la modification des attributs

real L(real u) const;	retourne L(u);
real R(real u) const;	retourne R(u);
real Linv(real alp) const;	retourne L^{-1}(alp);
real Rinv(real alp) const;	retourne R^{-1}(alp);
real Larea(void) const;	retourne l'aire sous la fonction L entre 0 et 1;
real Rarea(void) const;	retourne l'aire sous la fonction R entre 0 et 1;
virtual interval supp() const;	retourne le support du nombre LR simplifié, soit l'intervalle [m-alpha,m+beta];
virtual real operator ()(real x) const;	retourne l'appartenance de "x" au nombre LR simplifié;
virtual interval operator [](real alp) const;	retourne la coupe de niveau "alp" du nombre LR simplifié;
lrs &operator =(real x);	construit le nombre (x,0,0,1,1)(LR);
lrs &operator =(const fuzzy_number &x);	approxime le nombre flou par un nombre LR simplifié, en respectant le support de "x", en fixant m au milieu du noyau de "x", et en optimisant qL et qR par moindres carrés;
lrs &operator =(const interval &x);	approxime l'intervalle "x" par le nombre (x.l/2+x.u/2,x.u/2-x.l/2,x.u/2-x.l/2,1,1)(LR); théoriquement, on devrait faire tendre qL et qR vers -∞, mais c'est évidemment impossible numériquement; il s'agit donc d'une très mauvaise approximation; à éviter;

lrs &operator =(const tfn &x);	construit le nombre (x.b,x.b-x.a,x.c-x.b,1,1)(LR);
lrs &operator =(const trfn &x);	approxime le TRFN "x" par un nombre LR simplifié, en respectant le support de "x", en fixant m au milieu du noyau de "x" et en optimisant qL et qR par moindres carrés;
lrs &operator =(const lr &x);	approxime le nombre LR "x" par un nombre LR simplifié, en respectant le support de "x", en fixant m au milieu du noyau de "x" et en optimisant qL et qR par moindres carrés;
lrs &operator =(const lrs &x);	copie le nombre LR simplifié "x".

3.14.2.3 Opérateurs logiques

```
boole operator ==(real) const;
boole operator !=(real) const;
boole operator ==(const interval&) const;
boole operator !=(const interval&) const;
boole operator ==(const tfn&) const;
boole operator !=(const tfn&) const;
boole operator ==(const trfn&) const;
boole operator !=(const trfn&) const;
boole operator ==(const lr&) const;
boole operator !=(const lr&) const;
boole operator ==(const lrs&) const;
boole operator !=(const lrs&) const;
```

3.14.2.4 Opérateurs arithmétiques

```
lrs &operator +=(const lrs&);
lrs &operator -=(const lrs&);
lrs &operator +=(real);
lrs &operator -=(real);
lrs &operator *=(real);
lrs &operator /=(real);
lrs operator +(const lrs&) const;
lrs operator -(const lrs&) const;
lrs operator +(real) const;
lrs operator -(real) const;
lrs operator *(real) const;
lrs operator /(real) const;
lr operator +(const lr&) const;
lr operator -(const lr&) const;
lrs operator -(void) const;
```

3.14.3 Fonctions amies et fonctions utilisant la classe "lrs"

3.14.3.1 Opérateurs d'entrée-sortie

friend ostream &operator <<(ostream&, const lr&);
affiche un nombre LR simplifié dans un fichier, sous le format
"(m,alpha,beta)(LR) L(u)=1-u^qL R(u)=1-u^qR"

friend istream &operator >>(istream&, lr&);
lit un nombre LR simplifié dans un fichier, sous le format "m alpha beta qL qR".

3.14.3.2 Opérateurs arithmétiques

lr operator +(real, const lr&);
lr operator -(real, const lr&);
lr operator *(real, const lr&);

3.15 Classe "fuzzy_number" (entête "fuzynmbr.h")

La classe "fuzzy_number" permet d'approximer tout nombre flou ayant un support fini, en le décomposant en une série d'intervalles emboîtés, chaque intervalle étant associé à une coupe-α du nombre flou. Le nombre d'intervalles utilisés pour approximer le nombre flou peut être fixé par l'usager, et varier dynamiquement. Les nombres flous de cette classe peuvent aussi être imprimés dans un fichier texte sous la forme d'un graphique ou d'un tableau.

3.15.1 Attributs de classe

static unsigned nlev;
nombre de coupes alpha utilisées par défaut pour créer un nombre flou (vaut 10 initialement); ATTENTION: ne compte pas le support; l'espace alloué est donc pour nlev+1 intervalles; les coupes sont équidistantes, donc nlev=10 permet d'avoir une coupe-α à tous les 0.1 entre 0 et 1, donc 11 intervalles;

static modeprint printmode;
mode d'impression, choisi parmi les 3 constantes suivantes: (il est initialisé à "graphic"):
- fnapp: imprime sous forme de tableau la fonction d'appartenance;
- alphacuts: imprime sous forme de tableau les coupes-α;
- graphic: imprime sous forme de graphique la fonction d'appartenance.

3.15.2 Attributs d'instance protégés (technique)

interval *cut;	pointeur vers les coupes-α;
unsigned n;	nombre de coupes-α;
boole okfuz;	indique si les coupes-α sont coherentes.

3.15.3 Constructeurs et destructeur

fuzzy_number(real x=0,	construit un "fuzzy_number" ayant "niv" coupes-α

unsigned niv=nlev);	toutes égales à "x";
fuzzy_number(pair mu[], unsigned nmu,	construit un "fuzzy_number" ayant "niv" coupes-α
unsigned niv=nlev);	à partir d'un tableau de "nmu" paires $(x_i,\mu(x_i))$ de valeur de la fonction d'appartenance, où x_i est donné par mu[i][0], et $\mu(x_i)$ est donné par mu[i][1]; le tableau doit être trié en ordre croissant des x_i;
fuzzy_number(const fuzzy_convex &x,	construit un "fuzzy_number" ayant "niv" coupes-α
unsigned niv=nlev);	à partir d'un "fuzzy_convex"; permet une conversion implicite de tous les autres types de nombres flous;
fuzzy_number(const fuzzy_number &x);	copie-constructeur;
virtual ~fuzzy_number(void);	libère l'espace mémoire utilisé par l'objet.

3.15.4 Méthodes d'instance publiques

3.15.4.1 Gestion d'erreur

virtual boole ok(void) const;	indique si les coupes-α sont cohérentes et qu'un espace mémoire est réservé pour celles-ci.

3.15.4.2 Méthodes permettant l'accès et la modification des attributs

unsigned levels(void) const;	retourne "n", le nombre de coupes-α utilisées pour approximer le nombre flou;
boole setlevels(unsigned=nlev);	modifie le nombre de coupes-α utilisées pour approximer le nombre flou; retourne ok();
virtual interval operator [](real alp) const;	retourne la coupe de niveau "alp";
virtual interval &operator [](unsigned i) const;	retourne la ième coupe, donc mat[i];
virtual interval &operator [](int i) const;	retourne la ième coupe, donc mat[i];
virtual interval supp() const;	retourne le support, donc mat[0];
virtual real operator ()(real x) const;	retourne l'appartenance de "x" au nombre flou;
fuzzy_number &operator =(const fuzzy_number &x);	copie le nombre flou "x";
fuzzy_number &operator =(const fuzzy_convex &x);	copie l'ensemble convexe "x";
fuzzy_number &operator =(real x);	construit un nombre flou égal à "x".

3.15.4.3 Opérateurs logiques

boole operator ==(const fuzzy_number&) const;
boole operator !=(const fuzzy_number&) const;
boole operator ==(const fuzzy_convex&) const;
boole operator !=(const fuzzy_convex&) const;
boole operator ==(real) const;

```
boole operator !=(real) const;
```

3.15.4.4 Opérateurs arithmétiques

```
fuzzy_number &operator +=(const fuzzy_number&);
fuzzy_number &operator -=(const fuzzy_number&);
fuzzy_number &operator *=(const fuzzy_number&);
fuzzy_number &operator /=(const fuzzy_number&);
fuzzy_number &operator ^=(const fuzzy_number&);
fuzzy_number &operator +=(const fuzzy_convex&);
fuzzy_number &operator -=(const fuzzy_convex&);
fuzzy_number &operator *=(const fuzzy_convex&);
fuzzy_number &operator /=(const fuzzy_convex&);
fuzzy_number &operator ^=(const fuzzy_convex&);
fuzzy_number &operator +=(real);
fuzzy_number &operator -=(real);
fuzzy_number &operator *=(real);
fuzzy_number &operator /=(real);
fuzzy_number &operator ^=(real);
fuzzy_number operator +(const fuzzy_number&) const;
fuzzy_number operator -(const fuzzy_number&) const;
fuzzy_number operator *(const fuzzy_number&) const;
fuzzy_number operator /(const fuzzy_number&) const;
fuzzy_number operator ^(const fuzzy_number&) const;
fuzzy_number operator +(const fuzzy_convex&) const;
fuzzy_number operator -(const fuzzy_convex&) const;
fuzzy_number operator *(const fuzzy_convex&) const;
fuzzy_number operator /(const fuzzy_convex&) const;
fuzzy_number operator ^(const fuzzy_convex&) const;
fuzzy_number operator +(real) const;
fuzzy_number operator -(real) const;
fuzzy_number operator *(real) const;
fuzzy_number operator /(real) const;
fuzzy_number operator ^(real) const;
fuzzy_number operator -(void) const;
```

3.15.4.5 Méthodes d'entrée-sortie

`void print_mu(ostream&) const;`	force une impression de la fonction d'appartenance du nombre flou sous forme de tableau;
`void print_alphacuts(ostream&) const;`	force une impression des coupes-α du nombre flou sous forme de tableau;
`void print_textgraph(ostream&) const;`	force une impression de la fonction d'appartenance du nombre flou sous forme de graphique.

3.15.5 Fonctions amies

3.15.5.1 Opérateurs logiques

```
friend boole operator ==(real, const fuzzy_number&);
friend boole operator !=(real, const fuzzy_number&);
friend boole operator ==(const fuzzy_convex&, const fuzzy_number&);
friend boole operator !=(const fuzzy_convex&, const fuzzy_number&);
```

3.15.5.2 Opérateurs arithmétiques

```
friend fuzzy_number operator +(real, const fuzzy_number&);
friend fuzzy_number operator -(real, const fuzzy_number&);
friend fuzzy_number operator *(real, const fuzzy_number&);
friend fuzzy_number operator /(real, const fuzzy_number&);
friend fuzzy_number operator ^(real, const fuzzy_number&);
friend fuzzy_number operator +(const fuzzy_convex&, const fuzzy_number&);
friend fuzzy_number operator -(const fuzzy_convex&, const fuzzy_number&);
friend fuzzy_number operator *(const fuzzy_convex&, const fuzzy_number&);
friend fuzzy_number operator /(const fuzzy_convex&, const fuzzy_number&);
friend fuzzy_number operator ^(const fuzzy_convex&, const fuzzy_number&);
friend fuzzy_number pow(fuzzy_number, fuzzy_number);
friend fuzzy_number pow(const fuzzy_number&, real);
friend fuzzy_number pow(real, const fuzzy_number&);
friend fuzzy_number pow(fuzzy_number, real);
friend fuzzy_number pow(real, fuzzy_number);
friend fuzzy_number fabs(const fuzzy_number&);
friend fuzzy_number log(const fuzzy_number&);
friend fuzzy_number log10(const fuzzy_number&);
friend fuzzy_number exp(const fuzzy_number&);
```

3.15.5.3 Opérateurs d'entrée-sortie

Les fonctions d'entrée-sortie de la classe "fuzzy_number" sont plus complexes que pour les autres types de nombres flous.

```
friend ostream &operator <<(ostream&, const fuzzy_number&);
```

Imprime l'objet selon le format spécifié par l'attribut de classe "printmode", qui peut soit être "fnapp" (pour imprimer la fonction d'appartenance sous forme de tableau), "alphacuts" (pour imprimer les coupes-α sous forme de tableau), ou "graphic" (pour imprimer la fonction d'appartenance sous forme de graphique).

```
friend istream &operator >>(istream&, fuzzy_number&);
```

Construit un "fuzzy_number" à partir d'un tableau de paires $(x_i,\mu(x_i))$, trié en ordre croissant des x_i, contenant des valeurs de la fonction d'appartenance. On doit d'abord retrouver le nombre de coupes-α (qui deviendra la valeur de l'attribut "n"), et ensuite le

nombre de couples (x_i, $\mu(x_i)$) que contient le tableau. Viennent ensuite les valeurs de ce tableau. Par exemple, on pourrait lire dans un fichier:

```
10
5
1     0
2     0.3
3     1
4     0.8
5     0
```

On construirait alors un "fuzzy_number" comportant 10 coupes-α (en plus du support), à partir d'un tableau de 5 couples (x_i,$\mu(x_i)$) donnant les valeurs de la fonction d'appartenance pour x variant de 1 à 5.

3.16 Classe "string" (entête "strings.h")

Cette classe permet une manipulation aisée des chaînes de caractères. Elle est utilisée par la classe "graph" pour construire des graphiques de nombres flous.

3.16.1 Attributs d'instance protégés (technique)

La classe "string" encapsule une chaîne de caractère C++ standard, et conserve aussi sa longueur.

char *str;	pointeur vers les caractères de la chaîne;
unsigned len;	longueur de la chaîne.

Lors de la construction d'une chaîne, "len"+1 espaces mémoires sont réservés, pour permettre à la chaîne de se terminer par un caractère nul, comme toutes les chaînes C++ standard, tout en allouant véritablement "len" espaces pour le contenu de la chaîne.

3.16.2 Attribut de classe public

Lors de la création d'une chaîne, la taille n'a pas besoin d'être spécifiée. Elle est alors fixée par l'attribut de classe "default_length", dont la valeur initiale est 255:

```
static unsigned default_length;
```

3.16.3 Constructeurs et destructeur

La classe "string" comporte cinq constructeurs différents et un destructeur:

string(void);	construit une chaîne de longueur "default_length";
string(unsigned l);	construit une chaîne de longueur "l";
string(char *s)	construit une chaîne contenant une copie de "s";
string(char *s, unsigned l);	construit une chaîne de longueur "l", contenant les "l" premiers caractères de "s";
string(const string &s);	copie-constructeur;

~string(void);	libère l'espace réservé par l'objet.

3.16.4 Méthodes d'instance publiques

unsigned strlen(void);	retourne "len", la longueur de la chaîne;
char *const operator()(unsigned i);	retourne le ième caractère de la chaîne, ou NULL si l'index est hors des bornes de la chaîne;
string &operator=(const string &s);	copie les "len" premiers caractères de "s";
string &operator=(char *s)	copie les "len" premiers caractères de "s";
boole operator==(const string &s);	indique si les deux chaînes sont identiques;
boole operator!=(const string &s);	indique si les deux chaînes sont différentes;
string operator+(const string &s);	concatène deux chaînes de caractères.

3.16.5 Fonctions amies

friend istream &operator>> (istream&, string&);	lit une chaîne dans un fichier (l'usager est responsable de ne pas lire une chaîne plus longue que "len";
friend ostream &operator<< (ostream&, const string&);	imprime une chaîne dans un fichier.

3.17 Classe abstraite "graph" (entête "graph.h")

Cette classe abstraite propose une interface pour l'affichage graphique de nombres flous. Pour spécialiser cette classe, seule la méthode "plot" doit être redéfinie.

3.17.1 Attributs d'intance protégés (technique)

array<fuzzy_number> courbes;	tableau de nombre flous à tracer sur le graphique;
array<string> legende;	légende associée à chaque nombre flou à tracer;
array<char> symboles;	symbole associé à chaque nombre flou à tracer;
real xmin, xmax;	bornes de l'abcisse du graphique;
boole bornes_fixes;	indique si les bornes fixes "xmin" et "xmax" doivent être utilisés ou si le système doit les fixer lui-même;
ostream &s;	fichier ou doit être expédié le graphique.

3.17.2 Méthodes d'instance protégées (technique)

string default_tag(unsigned i);	légende par défaut pour le nombre flou numéro "i";
char default_symbol(unsigned i);	symbole par défaut pour le nombre flou numéro "i";
string default_tag(void);	légende par défaut pour le nombre flou suivant;
char default_symbol(void);	symbole par défaut pour le nombre flou suivant.

3.17.3 Constructeur

graph(ostream &f);	initialise un graphique en spécifiant le fichier de sortie.

3.17.4 Méthodes d'instance publiques:

boole ok(void);	indique si les nombres flous à tracer sont cohérents, ainsi que la légende et la liste des symboles;
unsigned n(void);	retourne la quantité de nombres flous à tracer;
boole set(unsigned i, const fuzzy_number &x);	spécifie que la série "i" du graphique doit correspondre au nombre flou "x";
boole set(unsigned i, string s);	spécifie que la légende de la série "i" doit être "s";
boole set(unsigned i, char c);	spécifie que le symbole de la série "i" doit être "c";
boole set(real xinf, real xsup);	indique que la borne inférieure de l'abcisse doit être "xinf" et la borne supérieure "xsup";
void reset_limits(void);	indique que les bornes de l'abcisse doivent être déterminés par le système;
void reset(void);	remet à zéro le graphique: élimine les séries, la légende, les symboles choisis ainsi que les bornes;
void reset(unsigned i);	élimine du graphique la série "i"; ATTENTION: les numéros de toutes les séries suivantes sont alors décalées vers le bas;
boole add(const fuzzy_number &x);	ajoute le nombre flou "x" au graphique (en dernier);
boole add(const fuzzy_number &x, string s);	ajoute le nombre flou "x" au graphique, en spécifiant que la légende doit être "s";
boole add(const fuzzy_number &x, char c);	ajoute le nombre flou "x" au graphique, en spécifiant que le symbole doit être "c";
boole add(const fuzzy_number &x, string s, char c);	ajoute le nombre flou "x" au graphique, en spécifiant que la légende doit être "s" et le symbole "c";
boole add(const fuzzy_number &x, char c, string s);	ajoute le nombre flou "x" au graphique, en spécifiant que la légende doit être "s" et le symbole "c";
graph &operator<< (const fuzzy_number &x);	ajoute le nombre flou "x" au graphique, et expédie le résultat du graphique dans le fichier "f";
virtual void plot(void)=0;	expédie le résultat du graphique dans le fichier "f".

3.18 Classe "textgraph" (entête "txtgraph.h")

La classe "textgraph" est une spécialisation de la classe abstraite "graph" qui permet d'imprimer des graphiques en mode texte.

3.18.1 Constructeur

Le constructeur de la classe "textgraph" n'ajoute rien au constructeur de la classe "graph":

textgraph(ostream &f);	initialise un graphique qui sera expédié au fichier "f".

3.18.2 Méthodes d'instance publiques

La seule méthode d'instance de la classe "textgraph" permet d'expédier le graphique au fichier:

void plot(void);

3.19 Classe générique "fuzzy_reg" (entête "fuzzyreg.h")

La dernière classe de la librairie est celle dont l'implantation est la plus complexe: elle permet la régression linéaire floue, simple ou multiple. L'interface de cette classe est tout de même assez simple. Pour l'utiliser, on doit spécifier le type des paramètres flous de la régression, parmi "interval", "tfn", "trfn", "lr", "lrs" et "fuzzy_number". Notons que les nombres flous obtenus par la régression sont toujours des nombres LR simplifiés; ils sont simplement convertis ensuite dans le format désiré.

3.19.1 Attributs d'instance protégés (technique)

Les attributs de la classe "fuzzy_reg" déterminent la configuration de la régression, et permettent de stocker les paramètres issus de la régression:

real belief;	niveau h de la régression (entre 0 et 1);
real k;	coefficient déterminant le rapport entre l'étendue à gauche et à droite du noyau des paramètres flous; avec "k" fixé à 1, les paramètres sont symétriques; pour "k" fixé à 2 par exemple, l'étendue à gauche du noyau des paramètres flous sera le double de l'étendue à droite;
real qL;	coefficient de la fonction L des paramètres flous, tel que L(u)=1-uqL;
real qR;	coefficient de la fonction R des paramètres flous, tel que R(u)=1-uqR;
vagueness criterion;	détermine le type de fonction objectif utilisé pour optimiser les paramètres de la régression:
"pred_vagueness":	flou de prédiction minimisé;
	"avg_vagueness": flou moyen minimisé;
array<Type> param;	paramètres résultant de la régression floue;
pivot pos_pivot;	type de pivot utilisé pour la régression floue:
	"mid_pivot": milieu de l'étendue des données;
	"min_pivot": minimum observé dans les données;
	"max_pivot": maximum observé dans les données;
	"med_pivot": médiane des données observées;
	"avg_pivot": moyenne des données observées;
	"usr_pivot": pivot fixé par l'usager;
	"opt_pivot": pivot minimisant le flou (optimisé);
matrix val_pivot;	valeur du pivot utilisé pour la régression floue.

3.19.2 Constructeur

Le constructeur de la classe "fuzzy_reg" permet de spécifier la configuration de la régression, qui pourra aussi être modifiée par la suite.

fuzzy_reg(vagueness v=pred_vagueness,real h=0,pivot p=opt_pivot,real ql=1,real qr=1,real c=1);

- la fonction objectif est fixée à "v", par défaut "pred_vagueness", i.e. flou de prédiction;
- le niveau de la régression est fixé à "h", par défaut 0;
- le type de pivot utilisé pour la régression est fixé à "p", par défaut "opt_pivot", i.e. pivot optimisé;
- les coefficients des fonctions L et R des paramètres de la régression sont respectivement fixés à "ql" et "qr", par défaut 1;
- le rapport entre l'étendue à gauche et à droite du noyau des paramètres flous de la régression est fixé à "c", par défaut 1, i.e. paramètres symétriques.

3.19.3 Méthodes d'instance publiques

On peut classer les méthodes d'instance de la classe "fuzzy_reg" en quatre groupes: les méthodes permettant de spécifier la configuration de la régression, les méthodes permettant de connaître la configuration de la régression, les méthodes permettant d'effectuer la régression, et les méthodes permettant d'effectuer des prédictions à partir des paramètres estimés.

3.19.3.1 Méthodes permettant de spécifier la configuration de la régression

Ces méthodes vérifient que le paramètre qui leur est passé est correct, et retournent un indicateur "vrai" si c'est le cas, et "faux" si le paramètre est erronné. Dans ce cas, la configuration initiale n'est pas modifiée. Tout le travail effectué par ces méthodes peut être réalisé lors de l'appel du constructeur.

boole set_belief(real h);	spécifie le niveau de la régression (attribut "belief");
boole set_asymmetry(real c);	spécifie le rapport entre l'étendue à gauche et à droite du noyau des paramètres flous (attribut "k");
boole set_pos_pivot(pivot p);	spécifie le type de pivot à utiliser pour la régression: "mid_pivot": milieu de l'étendue des données; "min_pivot": minimum observé dans les données; "max_pivot": maximum observé dans les données; "med_pivot": médiane des données observées; "avg_pivot": moyenne des données observées; "usr_pivot": pivot fixé par l'usager; "opt_pivot": pivot minimisant le flou (optimisé);
boole set_qL(real ql);	spécifie le coefficient de la fonction L des paramètres flous (attribut "qL");
boole set_qR(real qr);	spécifie le coefficient de la fonction R des paramètres flous (attribut "qR");
boole set_criterion(vagueness v);	spécifie le type de fonction objectif à utiliser pour optimiser les paramètres flous:
"pred_vagueness":	flou de prédiction minimisé;
"avg_vagueness":	flou moyen minimisé.

3.19.3.2 Méthodes permettant de connaître la configuration de la régression

real get_belief() const;	retourne le niveau de la régression (attribut "belief");
real get_asymmetry() const;	retourne le rapport entre l'étendue à gauche et à droite du noyau des paramètres flous (attribut "k");
pivot get_pos_pivot() const;	retourne le type de pivot utilisé pour la régression: "mid_pivot": milieu de l'étendue des données; "min_pivot": minimum observé dans les données; "max_pivot": maximum observé dans les données; "med_pivot": médiane des données observées; "avg_pivot": moyenne des données observées; "usr_pivot": pivot fixé par l'usager; "opt_pivot": pivot minimisant le flou (optimisé);
matrix &get_val_pivot();	retourne la valeur du pivot utilisé pour la régression;
real get_qL() const;	retourne le coefficient de la fonction L des paramètres flous (attribut "qL");
real get_qR() const;	retourne le coefficient de la fonction L des paramètres flous (attribut "qR");
vagueness get_criterion();	retourne le type de fonction objectif utilisé pour optimiser les paramètres flous: "pred_vagueness": flou de prédiction minimisé; "avg_vagueness": flou moyen minimisé.
real get_vagueness (const matrix &x);	retourne la valeur numérique de la fonction objectif pour les données "x" (ne doit pas être utilisé avant d'effectuer la régression);
array<Type> &operator ()();	retourne les paramètres de la régression floue (ne doit pas être utilisé avant d'effectuer la régression).

3.19.3.3 Méthodes permettant d'effectuer la régression floue

Toutes les méthodes permettant d'effectuer la régression floue sont semblables. Le premier paramètre "y", obligatoire, est un vecteur de réels ou de nombres flous spécifiant les valeurs de la variable dépendante. Le second paramètre "x", aussi obligatoire, est une matrice de réels spécifiant les valeurs des variables indépendantes, celles-ci étant organisées en colonnes. Le troisième paramètre "p" est optionnel. Il s'agit d'un vecteur ou d'un nombre réel contenant le pivot à utiliser pour effectuer la régression. Il ne peut s'agir évidemment d'un nombre réel que dans le cas de la régression simple, à une seule variable indépendante. Si le pivot n'est pas spécifié, il sera déterminé à partir de la valeur de l'attribut "pos_pivot".

array<Type> &operator ()(const matrix &y, const matrix &x);
array<Type> &operator ()(const matrix &y, const matrix &x, real p);
array<Type> &operator ()(const matrix &y, const matrix &x, const matrix &p);
array<Type> &operator ()(const array<Type> &y, const matrix &x);
array<Type> &operator ()(const array<Type> &y, const matrix &x, real p);
virtual array<Type> &operator ()

(const array<Type>&y, const matrix &x, const matrix &p);

3.19.3.4 Méthodes permettant d'effectuer des prédictions

Ces trois méthodes permettent d'effectuer une ou plusieurs prédictions à partir du modèle déterminé par la régression. Elle ne doivent donc être utilisées qu'après l'application de la régression.

Type prediction(real x) const;	retourne, dans le cas de la régression simple, le nombre flou prédit pour le réel "x";
virtual Type prediction(const matrix &x) const;	retourne, dans le cas de la régression multiple, le nombre flou prédit pour le vecteur "x";
array<Type> predictions(const matrix &x) const;	retourne un tableau de nombre flou prédits pour un ensemble de valeurs des variables explicatives; à chaque ligne doit correspondre une observation pour laquelle on désire une prédiction; les variables indépendantes sont donc disposées en colonnes.

4. EXEMPLES D'UTILISATION DE LA LIBRAIRIE

Nous voulons donner quelques exemples d'utilisation de la librairie développée pour montrer sa simplicité, et pour aider l'utilisateur novice. Ces exemples seront basés sur l'utilisation du disque de Secchi pour estimer le coefficient d'extinction η de l'eau d'un lac, qui dépend principalement des particules en suspension et des particules dissoutes. Il s'agit d'une procédure courante qui consiste à immerger un disque métallique réfléchissant la lumière jusqu'à ce qu'on ne puisse plus le voir. On peut montrer qu'à cette profondeur, on a la relation suivante:

$k=10^{-\eta(2z)}$

où z est la profondeur du disque, et k est une constante de calibration, qui peut varier de 0.02 à 0.05, et qui est usuellement fixée à 0.03. On peut facilement isoler η:

$\eta=-\log_{10}(k)/(2z)$

Connaissant k et z, on peut alors calculer le coefficient d'extinction à l'aide du programme C++ suivant (les sections en gras sont celles qui devront être changées pour appliquer le calcul d'intervalle ou le calcul flou - c'est à dire seulement le type des variables):

```
#include <math.h>              // fonctions mathématiques
#include <iostream.h>          // entrées-sorties

void main()
{
      float k = 0.03;          // constante estimée à 0.03
      float z;                 // profondeur à laquelle on perd le disque
      float n;

      cin >> z;                // lecture de la profondeur

      n = -log10(k)/(2*z);     // calcul du coefficient d'extinction

      cout << n;               // écriture du coefficient d'extinction
}
```

Par exemple, en entrant z=4m, on obtient un coefficient d'extinction de η=0.19.

4.1 Application au calcul d'intervalle

Le calcul flou est une généralisation du calcul d'intervalle. On peut donc utiliser la librairie pour effectuer du calcul d'intervalle, qui est un cas particulier. On peut ainsi étudier la propagation d'une erreur de mesure dans un calcul complexe. Dans le cas du calcul du coefficient d'extinction, on peut étudier l'incertitude sur la constante de calibration k ainsi que sur la profondeur mesurée z. On sait déjà que k peut varier de 0.02 à 0.05. On peut remplacer dans le programme précédent le nombre réel k=0.03 par un

intervalle [0.02,0.05]. Pour ce qui est de z, on supposera qu'il est possible d'atteindre une précision de l'ordre du centimètre.

```
#include <math.h>              // fonctions mathématiques
#include <iostream.h>          // entrées-sorties
#include "interval.h"          // entête décrivant l'objet "interval"

void main()
{
      interval k(0.02,0.05);   // cte estimée par l'intervalle [.02,.05]
      interval z;              // profondeur à laquelle on perd le disque
      interval n;              // coeff.d'extinction: aussi un intervalle

      cin >> z;                // lecture de la profondeur

      n = -log10(k)/(2*z);     // calcul du coefficient d'extinction

      cout << n;               // écriture du coefficient d'extinction
}
```

À part l'ajout d'une librairie et le changement du type des variables (les sections indiquées en gras), rien n'a changé: ni les entrées-sorties, ni le calcul du coefficient d'extinction. Il est donc facile d'adapter des programmes existants pour le calcul d'intervalle. Pour z=[3.99,4.01]m (i.e. 4m±1cm), on obtient comme coefficient d'extinction l'intervalle [0.16,0.21], ce qui est plus instructif que simplement 0.19. On peut avoir une idée de la précision.

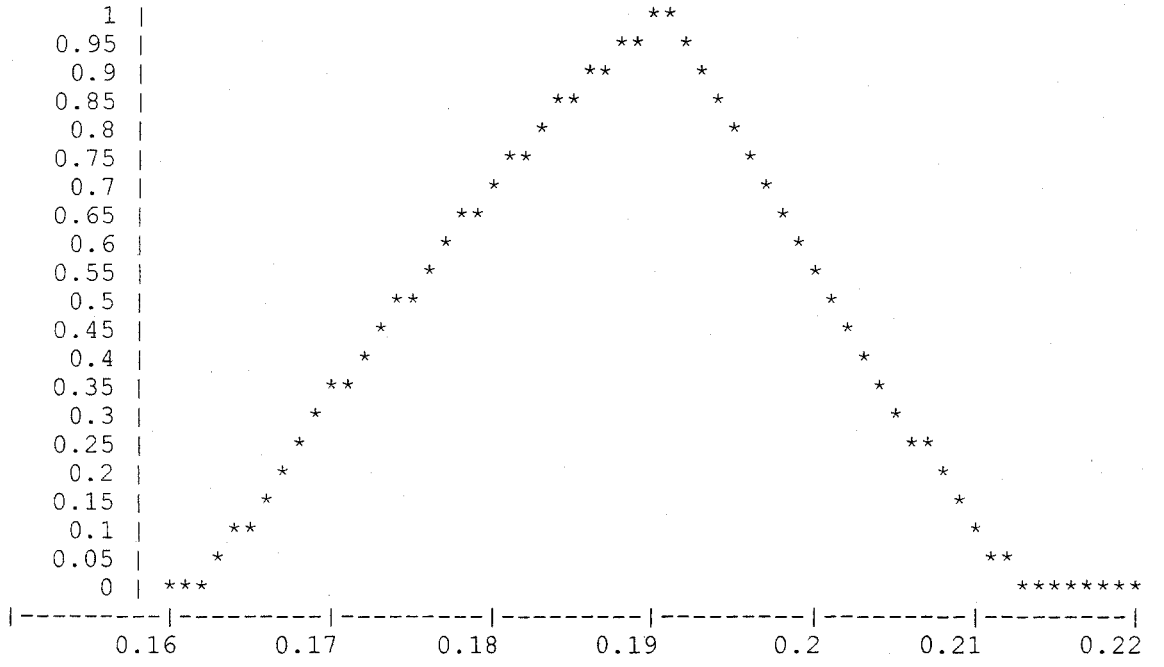
4.2 Application au calcul flou

Le calcul d'intervalle permet de déterminer les cas extrêmes. Si k est bel et bien entre 0.02 et 0.05, que l'erreur sur z n'est pas supérieure à 1 cm et que le modèle est valide, il est absolument certain que le coefficient d'extinction est situé entre 0.16 et 0.21. Le calcul flou permet de nuancer ce résultat, en appliquant le calcul d'intervalle à plusieurs niveaux de confiance. Ceci permet d'introduire de l'information supplémentaire concernant les valeurs plus vraisemblables des paramètres.

Dans l'exemple du disque de Secchi, on sait que la valeur de k=0.03 est plus vraisemblable. Dans un tel cas, on choisit usuellement de représenter l'incertitude sur k par un nombre flou triangulaire, où le mode est fixé à la valeur la plus vraisemblable. On pourrait alors supposer que k=(.02,.03,.05)(TFN). Les modifications à apporter au programme précédent sont alors mineures (elles sont indiquées en gras) pour permettre de calculer un nombre flou représentant le coefficient d'extinction.

```
#include <math.h>              // fonctions mathématiques
#include <iostream.h>          // entrées-sorties
#include "interval.h"          // entête décrivant l'objet "interval"
#include "tfn.h"               // entête décrivant l'objet "tfn"
#include "fuzynmbr.h"          // entête décrivant l'objet "fuzzy_number"

void main()
{
      tfn k(0.02,0.03,0.05);   // cte estimée par (.02,.03,.05)(TFN)
      interval z;              // profondeur à laquelle on perd le disque
      fuzzy_number n;          // coeff. d'extinction: un nombre flou

      cin >> z;                // lecture de la profondeur

      n = -log10(k)/(2*z);     // calcul du coefficient d'extinction

      cout << n;               // écriture du coefficient d'extinction
}
```

Il faut ajouter deux fichier d'entête, l'un pour la description des nombres flous triangulaires ("tfn"), l'autre pour les nombres flous généraux ("fuzzy_number"). En effet, le coefficient d'extinction n'est pas un nombre flou triangulaire, mais un nombre flou général. On modifie aussi le type des paramètres "k" et "n", mais encore une fois on ne touche ni aux entrées-sorties, ni aux calculs. Cette fois, la sortie du programme est un graphique:



On observe que le nombre flou représentant l'incertitude sur le coefficient d'extinction serait assez bien modélisé par un TFN. On peut donc dire que l'incertitude sur le coefficient augmente à peu près de la même façon que l'incertitude sur la constante de calibration. D'autres exemples d'application sont donnés à l'Annexe A, y compris un exemple d'utilisation de la régression floue.

ANNEXE A: COMMUNICATION POUR LA CONFÉRENCE AIENG 96

Cette communication a été acceptée pour publication à la conférence AIENG 96. Elle présente la librairie proposée dans ce travail, et détaille les principes de l'arithmétique floue et de la régression floue.

A C++ library for fuzzy sensitivity analysis and multiple fuzzy linear regression

V. Fortin & B. Bobée
NSERC/Hydro-Québec Chair on Statistical Hydrology
Institut National de la Recherche Scientifique
Sainte-Foy, Québec, Canada, G1V 4C7
EMail: chaire_hydro@uquebec.ca

Abstract

Fuzzy sets theory has proven over the years to be a valuable tool for modeling uncertainty in engineering. It is used extensively in control, in expert systems and in rule-based models. However, applications to sensitivity analysis and regression are still few, mainly because there is no appropriate software available. A C++ library of objects has been developed to easily and efficiently introduce fuzzy sensitivity analysis into new or existing C/C++ code, and to perform multiple fuzzy linear regression. An outline of the library is given, together with examples of applications in hydrologic engineering. For problems involving only fuzzy regression, a more user-friendly interface is currently being developed.

Introduction

In any modeling task it is vital to study the uncertainty on the inputs and on the model itself, so as to assess its effects on the outputs. For simple models, such as linear models, it is possible using probability theory to perform this sensitivity analysis analytically, given that there is sufficient data. But for more elaborate models, it is necessary to resort to Monte-Carlo simulations, usually with a hypothesis of independence between inputs to reduce complexity. When data is scarce or when the validity of the hypothesis of independence between inputs is not so obvious, probability theory collapses, leaving the scientist in an awkward position: he may feel that there is sufficient physical evidence to justify the model, but cannot defend it on statistical grounds.

An interesting alternative to probability theory in uncertainty analysis is fuzzy sets theory [7,9,15]. It allows for an approximate assessment of uncertainty even when data is scarce or the model very complex by modeling imprecise knowledge, even intuition, and combining it with the available numerical data. It is not a panacea, nor the only way of dealing with the problem. Bayesian analysis [3] or imprecise probabilities [4,12,14] may well prove as efficient and more appealing theoretically, but fuzzy sets theory as the clear advantage of being much simpler. At least as a first approximation, it is a simple solution to a complex problem.

After briefly reviewing and discussing basic concepts of fuzzy sets theory, we discuss the application of fuzzy sensitivity analysis to complex models, and the application of fuzzy regression to simple models when data is scarce. We present a C++ library of objects which can be easily used to build fuzzy sensitivity analysis into new or existing models developed in C or C++, and to perform fuzzy regression. Examples of applications of fuzzy sensitivity analysis and fuzzy regression using the proposed library to hydrologic engineering problems are provided, and possible enhancements are discussed.

Fuzzy sets - a brief review

Fuzzy sets theory was introduced by Zadeh [15] to model imprecision in language. The concept has since been extended to allow for modeling of imprecision in measurements [5] and in probability assessments [6]. A fuzzy set A is defined by its membership function $\mu_A(x)$, which associates with each element x of the universe Ω a number between 0 and 1. This number reflects the degree of membership of x to the fuzzy set A, with $\mu_A(x)=0$ meaning that x does not belong to A and with $\mu_A(x)=1$ meaning that x belongs completely to A. Usual, non fuzzy (or crisp) sets are a particular case of fuzzy sets having membership functions which can only take the values 0 or 1. When there is no ambiguity, we shall write simply $\mu(x)$ for the membership function so as to simplify the notation.

Fuzzy sets theory uses extensively the concept of an h-cut, which is defined, for a fuzzy set A, as the crisp subset $A(h)$ of all values of Ω for which $\mu_A(x)\geq h$. A fuzzy set is completely defined by the collection of its h-cuts, and is said to be convex if all h-cuts are convex, i.e. can be represented by intervals $A(h)=[l_h,u_h]$. A fuzzy set A is said to be normal if $A(1)\neq\varnothing$, or (equivalently) if $\sup\{\mu_A(x)=1\}$. Finally, $A(0)$ is named the support of A, and $A(1)$ its kernel.

Assessing degrees of membership of fuzzy sets

Alone, the preceding definition of a fuzzy set is of little use since it does not associate a clear physical interpretation with degrees of membership. Without such an interpretation, they would be impossible to measure. Just as probabilities cannot be measured (only frequencies, betting rates, or degrees of belief - interpretations of probabilities - can be), degrees of membership cannot be measured. And just as there are many interpretations of probabilities, there are many possible interpretations of degrees of membership.

In engineering, imprecise measurements provide the basis for a useful interpretation of these degrees. Suppose you dispose of a measuring device, which cannot measure precisely the value of a random variable Y, but only tell that it is surely inside an interval $X=Y\pm\Delta Y$ (where ΔY can vary from one observation to the next). A fuzzy set modeling this imprecision could have as membership function $\mu(x)$ the probability that a given value x would be included in an imprecise observation X:

$$\mu(x)=\Pr[x\in X] \tag{1}$$

There are strong links between probability and fuzzy sets theories, but a membership function is nothing like a probability density function (*pdf*): it does not integrate to one, and it can be interpreted directly both in the discrete and continuous cases. In contrast, to obtain a probability value from a continuous *pdf*, it is necessary to integrate the function between two bounds.

Possibility theory

A membership function may define a possibility distribution Π on a random variable, which is an upper bound of its probability distribution. When all imprecise measurements

$\{X_1,X_2,...,X_n\}$ of a random variable Y are nested (meaning that they can be ordered so that $X_{(1)}\subseteq X_{(2)}\subseteq...X_{(n)}\subseteq\Omega$), then $\mu(x)=\Pr[x\in X]$ is a possibility assignment of Y in Ω. Indeed, it can be shown [6] that for any subset B of Ω, the maximum value of $\mu(x)$ over that subset is superior or equal to the probability of Y belonging to B:

$$\Pi(Y\in B)=\sup_{x\in B}\mu(x)\geq\Pr[Y\in B] \tag{2}$$

When no imprecise nested data is available (even when almost no information is available), it is nonetheless easy to obtain a possibility assignment, much more easily in fact than a probability assignment. Indeed, even if nothing is known about the probability of $Y\in B$, a precise possibility assignment $\mu(x)$ can easily be devised to let $\Pi(Y\in B)=1$, so that no value of probability is excluded. Bayesian statisticians have however found that justifying any precise probability assignment in that case is next to impossible [12]. There is of course a price to pay for using possibilities: the resulting inferences are less precise than with probability theory, so that possibility theory should only be used when it is not feasible to obtain a probability distribution, either because of limited knowledge or because of limited resources (such as time and money). However, it may be a good approach to use possibility distributions in first approximation, which could then be refined using probability theory.

Fuzzy numbers

A fuzzy number is simply a convex and normal fuzzy set. Note that a real number z is a particular case of a fuzzy number having a membership function $\mu_z(x)=1$ if $x=z$ and $\mu_z(x)=0$ if $x\neq z$. Fuzzy numbers are interesting because a complete arithmetic has been defined on them [10], which generalizes interval arithmetic [11]. The result of any operator $\perp$ applied on two fuzzy numbers A and B is a fuzzy set C having the following h-cuts:

$$C(h)=A(h)\perp B(h) \tag{3}$$

Notice that C is a fuzzy set but not necessarily a fuzzy number. However, for most operators used in practice, except for set operators such as union and intersection, C is a fuzzy number (if it exists - division by zero is still not allowed). It should also be noted that some basic results of real number arithmetic do not hold for fuzzy arithmetic. For example in general even if $A=B$, $A-B\neq 0$ and $A/B\neq 1$.

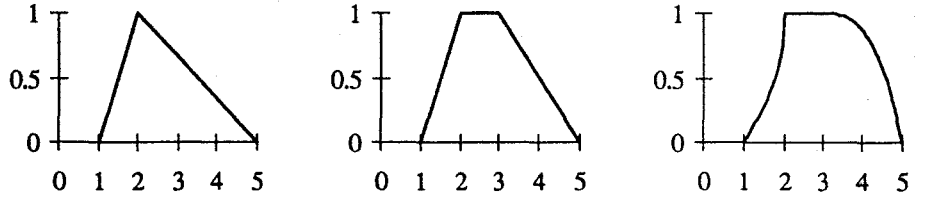


Figure 1: membership function of various fuzzy numbers

If an infinite number of h-cuts are in theory necessary to completely specify a fuzzy number, much simpler fuzzy numbers are used in practice. LR numbers, proposed by Dubois & Prade [6], are a very general family of parametric fuzzy numbers. They are completely specified by their kernel $[\underline{m},\overline{m}]$, their support $[\underline{m}-\alpha,\overline{m}+\beta]$ and two functions $L(u)$ and $R(u)$ defined on [0,1] and decreasing from 1 to 0. LR numbers are denoted $(\underline{m},\overline{m},\alpha,\beta)$(LR) and their membership function is given by eq. (4).

$$\mu(x)=\begin{cases} L[(\underline{m}-x)/\alpha] & \text{if} \quad \underline{m}-\alpha \le x < \underline{m} \\ 1 & \text{if} \quad \underline{m} \le x \le \overline{m} \\ R[(x-\overline{m})/\beta] & \text{if} \quad \overline{m} < x \le \overline{m}+\beta \\ 0 & \text{elsewhere} \end{cases} \tag{4}$$

The family of functions $1-u^q$ is often used for L and R functions, so that the notation $(\underline{m},\overline{m},\alpha,\beta,a,b)$(LR) can be used to completely specify an LR fuzzy number if $L(u)=1-u^a$ and $R(u)=1-u^b$. Simpler LR numbers for which $\underline{m}=\overline{m}=m$ are also often used. These will be called LRS numbers and denoted (m,α,β,a,b)(LRS). Triangular fuzzy numbers (TFNs) and trapezoidal fuzzy numbers (TRFNs) are respectively particular cases of LRS and LR numbers having linear L and R functions. They are consequently completely specified by their support and kernel. A TFN with support $[a,c]$ and kernel $\{b\}$ is denoted (a,b,c)(TFN). A TRFN with support $[a,c]$ and kernel $[b_1,b_2]$ is denoted (a,b_1,b_2,c)(TRFN). Figure 1 shows examples of various fuzzy numbers.

Fuzzy sensitivity analysis

Once a model is devised and calibrated to simulate a physical system, sensitivity analysis consists in studying the effect of small variations of the inputs on the outputs of the model. When the *joint* probability distribution of all inputs is precisely known, the probability distribution of the outputs can be found, either analytically if the model is simple (e.g. linear) or by Monte-Carlo simulation if the model is more complex. However, in most cases it is virtually impossible to determine the probability distribution of the input parameters unless a very large amount of data is available. Furthermore Monte-Carlo simulation involves a fair amount of computing to obtain only reasonably accurate responses.

An alternative is to obtain possibility distributions for each inputs (we have seen that this can be done precisely even when no data is available) and to replace every operation in the model by its fuzzy counterpart. Then, the outputs of the model will be fuzzy numbers, which can themselves be interpreted as possibility distributions. Not only do we then simplify the modeling of uncertainty, but we also eliminate the need for a lengthy simulation. It should however be mentioned that fuzzy operations are more costly than the corresponding operations on real numbers, leading to longer computing times, but still nowhere near the time needed for a Monte-Carlo simulation.

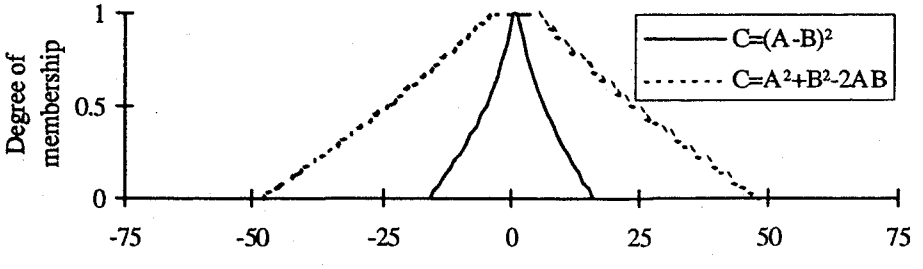


Figure 2: Computing the uncertainty on $(A-B)^2$ using two different formulae

Using fuzzy sensitivity analysis may seem simple, but there are two major difficulties to be overcome before it can be applied to a model. First, operators used in the model must be replaced by their fuzzy counterparts, which unlike Monte-Carlo simulation involves potentially major modifications to the code if the model is already programmed. The second difficulty is that the fuzzy numbers representing the outputs of the model may vary depending on the way the algorithms are coded, if any input variable appears more than once in the code [6]. Efforts should therefore be made, when coding the algorithms, to refrain from introducing

the same input variable more than once. However, introducing needlessly the same variable many times in the model, such as replacing $C=(A-B)^2$ by $C=A^2+B^2-2AB$, only increases uncertainty on the outputs, never decreases it, making the possibility assessments more vague, never less. It is consequently not crucial to avoid using the same variables many times, as far as the validity of the results is concerned, but it may help reduce the uncertainty on the outputs substantially.

Consider for example the fuzzy numbers $A=(1,2,5)$(TFN) and $B=(1,2,3,5)$(TRFN) (see Figure 1 for their graphical representation). Figure 2 shows the membership function of the fuzzy numbers $(A-B)^2$ and A^2+B^2-2AB. It is readily seen that computing C as A^2+B^2-2AB gives a much more imprecise result than computing $C=(A-B)^2$, but both results are coherent, i.e. $(A-B)^2 \subseteq A^2+B^2-2AB$. In this case, using more than once the same variables increases considerably the uncertainty of the output. However, since fuzzy sensitivity analysis gives only an upper bound on the probability, the result is still valid, only less useful.

While overcoming this second difficulty may imply modifying the structure of the code, simply replacing usual operators by fuzzy ones can be much simpler with an object-oriented language such as C++. Indeed, with an object-oriented language a new type (or class) of objects representing fuzzy numbers can be defined, together with all needed operators, and only the type of the variables used in the model then needs to be changed. We have developed such a new type in C++. Before presenting the main features of this library, we shall discuss the problem of modeling linear relationships when data is scarce.

Multiple fuzzy linear regression

An important class of physical systems can be represented by linear models. When numerical data is sufficient the problem of taking into account uncertainty can be solved by statistical regression techniques. However, this process requires the verification of a number of hypotheses, including independence between inputs and between each observation of a given input variable, and homogeneous random variations normally distributed about the mean. Usually, these hypotheses can only be verified by looking at the data, before fitting the model (for cross-correlations between inputs) and after (for the distribution of errors). When data is scarce, it is often impossible to determine the validity of those hypotheses. If it is furthermore needed to validate the model itself by analyzing the quality of the fit, even more observations are needed to obtain powerful statistical fitting tests.

Again fuzzy sets theory provides an alternative to probability theory. Fuzzy regression, developed by Tanaka et al. [13] and improved by Bárdossy [1], can often be helpful to model scarce data. It does not require the hypothesis of independence between observations of a same variable nor the independence and identical distribution of errors. In fact, known patterns in the distribution of errors can be explicitly modeled. Instead of adding a white noise to allow for model uncertainty, in fuzzy regression the parameters are taken to be fuzzy numbers. Let $f(\mathbf{x}|\mathbf{A},\overline{\mathbf{x}})$ be the fuzzy output of a linear model of K input parameters $\mathbf{x}=\{x_1,x_2,\ldots,x_k\}$:

$$f(\mathbf{x}|\mathbf{A},\overline{\mathbf{x}})=A_0+\sum_{k=1}^{K} A_k(x_k-\overline{x}_k) \qquad (5)$$

where $\mathbf{A}=\{A_0,A_1,\ldots,A_K\}$ are parameters of the model, and $\overline{\mathbf{x}}=\{\overline{x}_1,\overline{x}_2,\ldots,\overline{x}_K\}$ is the point in the input space where the model is believed to be most accurate, usually in the middle of the range of interest of the input variables. The basic principle of fuzzy regression is to find the most precise parameters such that a given h-cut of $f(\cdot|\mathbf{A},\overline{\mathbf{x}})$ includes all observed output of the system being modeled. Let $\{(Y_j,\mathbf{x}_j),\ j=1,2,\ldots,n\}$ denote a sample of n observations with $\mathbf{x}_j=\{x_{1,j},x_{2,j},\ldots,x_{k,j}\}$ being the jth observation of the K inputs, and let $f(\mathbf{x}_j,h=h_0|\mathbf{A},\overline{\mathbf{x}})$ denote an h_0-cut of $f(\mathbf{x}_j|\mathbf{A},\overline{\mathbf{x}})$. Note that each Y_j can either be real or fuzzy, allowing for explicit modeling of sampling error structure. Only fuzzy models respecting the following conditions are acceptable:

$$Y_j(h_0) \subseteq f(x_j, h=h_0 | \mathbf{A}, \bar{x}), j=1,2,...,n \quad (6)$$

where (6) holds for a given level h_0. Of all values of A compatible with (6), parameters as precise as possible should be chosen. One possibility is to minimize the total width of these parameters, given by (7):

$$H(\mathbf{A}) = \sum_{k=0}^{K} \{\sup A_k(0) - \inf A_k(0)\} \quad (7)$$

but Bárdossy [1] has shown that minimizing prediction vagueness, given by (8), is preferable. It can be interpreted as the total area of the fuzzy model:

$$H(\mathbf{A}) = \int_{x_K^-}^{x_K^+} \int_{x_{K-1}^-}^{x_{K-1}^+} \cdots \int_{x_1^-}^{x_1^+} \int_{-\infty}^{+\infty} \mu_{f(x|\mathbf{A},\bar{x})}(y) \cdot dy \cdot dx_1 \cdot dx_2 \ldots dx_K \quad (8)$$

where $[x_k^-, x_k^+]$ defines the range of interest of the variable x_k. Minimizing $H(\mathbf{A})$ defined by (7) or (8) subject to the constraints (6) is a linear programming problem, which can be solved by standard techniques. Additional constraints must however be put on the shape of the fuzzy parameters for the solution to be unique [1]. If the parameters are taken to be LRS numbers such that $A_k=(m_k,\alpha_k,\beta_k)$(LRS), with L and R functions specified beforehand, then it is sufficient to impose $\alpha_k=c_k \cdot \beta_k$, where c_k is a constant also chosen beforehand. Usually, the parameters are supposed to be symmetrical, with $\alpha_k=\beta_k$.

The choice of h_0, L and R is a controversial issue in fuzzy regression; their interpretation is unclear. The value of h_0 influences directly the precision of the fuzzy parameters. Small values of h_0 make the model more precise, larger ones make the model less precise. In fact for crisp data (non fuzzy Y_j) the only effect of choosing a non-zero value of h_0 is to multiply the coefficients α_k and β_k respectively by $1/L^{-1}(h_0)$ and $1/R^{-1}(h_0)$. In that case it seems that h_0, L and R simply introduce an implicit security factor into the computations. Quite often in the literature, fuzzy regressions are computed with linear L and R functions, and with h_0=0.5, leading to an implicit multiplication by two of the regresssion coefficients α_k and β_k, and thus of the support of the fuzzy model.

As for fuzzy sensitivity analysis, lack of adequate software has limited the application of fuzzy regression. We hope to lift some of these difficulties by offering a simple, easy to use, yet powerful C++ library for performing fuzzy arithmetic, fuzzy sensitivity analysis and multiple fuzzy linear regression.

Outline of the C++ library

The simplest way of adapting models for fuzzy sensitivity analysis and of programming fuzzy regression is to develop a new data type (or class) representing a fuzzy number, with all associated operators. This is only possible in an object-oriented language such as C++. We have in fact developed a hierarchy of classes modeling different types of fuzzy numbers: intervals, TFNs, TRFNs, LR and LRS fuzzy numbers, and even completely general fuzzy numbers, which are modeled by nested intervals, the number of which can be set by the user. The C++ class names chosen for those new types are respectively `interval`, `tfn`, `trfn`, `lr`, `lrs` and `fuzzy_number`. With this architecture, simpler types of fuzzy numbers can therefore be used whenever possible, minimizing both memory and computing time.

All necessary logic and arithmetic operators, together with a number of useful mathematical functions, have been programmed so that these different types of fuzzy numbers can be manipulated easily. It is also possible to explicitly cast any type of fuzzy number into another one. In particular, a general fuzzy number can be cast into an LR number, with the L and R functions automatically optimizing themselves to obtain a least-square fit. Standard input-output C++ operators have also been overloaded, and a class providing graphical

outputs has also been devised. For now however it can only produce text-based graphics. To store collections of fuzzy numbers, a template for a generic array class has been devised, complete with operators allowing for direct arithmetic manipulation of matrix and vectors of fuzzy numbers. The multiple fuzzy linear regression algorithm, using a modified simplex algorithm for linear programming, has been implemented as a functor, which is a C++ encapsulated function. The library should be entirely portable; it has however only been tested with Borland C++™ 4.5 for Microsoft Windows™.

Using the library for fuzzy sensitivity analysis

With the C++ library presented previously, adapting a model coded in C or C++ for fuzzy sensitivity analysis is quite simple. Of course, as mentioned earlier, code optimization to avoid using the same input variable more than once may be needed to decrease the fuzziness of the outputs, but is not necessary to ensure validity of the results. An application of fuzzy sensitivity analysis to a ground water hydrology problem illustrates the use of the library.

Pumping at constant rate in a homogeneous confined aquifer lowers water pressure, producing a steady-state drawdown of s meters at a distance of r meters from the well, which depends on the pumping rate Q (m^3/s), the range of action R (m) of the well, and the transmissivity T (m^2/s) of the aquifer [8]:

$$s=\frac{Q}{2\pi T}\log(R/r) \tag{9}$$

where log(·) denotes Napierian logarithms. For example, with T=0.005 m^2/s, R=28 km, Q=0.02 m^3/s the drawdown can be computed to be s=7.5 m at a distance of r=20 cm from the well, using the following C++ code (characters in bold identify instructions which will need to be changed to adapt the program for fuzzy sensitivity analysis):

```
#include <math.h>          // LIBRARY FOR log FUNCTION
#include <iostream.h>      // LIBRARY FOR i/o
void main() {
      // INITIALIZATION OF THE INPUT PARAMETERS
      float T(.005);
      float R(28000);
      float Q(.02);
      float r(.2);
      float s;

      // COMPUTING AND PRINTING THE OUTPUT OF THE MODEL
      s = Q/(2*3.1416*T) * log(R/r);
      cout << s;}
```

In daily hydrogeology practice, the main difficulty in solving such problems is the lack of accuracy on input parameters, especially transmissivity (T) which depends both on the depth of the aquifer and its hydraulic conductivity. Suppose now that T is only known to be between 0.002 and 0.01 m^2/s, and is probably around 0.005 m^2/s. T=(.002,.005,.01)(TFN) could model this information. In the same manner, the following fuzzy numbers could be obtained for R and Q:

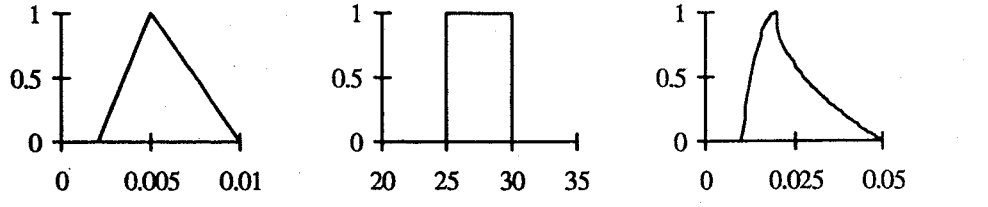


Figure 3: membership function of R, Q and T

Modifying the preceding program to take into account uncertainty is straightforward. It involves adding a new library for fuzzy arithmetic and modifying the initialization of the input parameters, but not the actual code responsible for computing and printing the outputs. Here is the same model coded for fuzzy sensitivity analysis, with modifications in bold (note that only one line was added, to include the new library):

```
#include <math.h>          // LIBRARY FOR log FUNCTION
#include <iostream.h>      // LIBRARY FOR i/o
#include "fuzynmbr.h"      // LIBRARY FOR FUZZY ARITHMETIC
void main() {
      // INITIALIZATION OF THE INPUT PARAMETERS
      tfn                  T(.002,.005,.01);
      interval             R(25000,30000);
      lrs                  Q(.02,.01,.03,2,.5);
      float                r(.2);
      fuzzy_number         s;
      // COMPUTING AND PRINTING THE OUTPUT OF THE MODEL
      s = Q/(2*3.1416*T) * log(R/r);
      cout << s;}
```

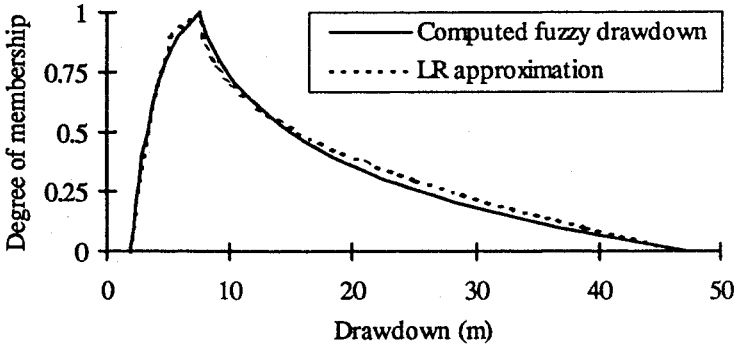


Figure 4: Uncertainty on computed drawdown

Notice that various types of fuzzy numbers can be freely mixed in a same formula. The output is of type `fuzzy_number` since no simpler parametric type can model it exactly, given the operators used in the model. Except for very simple models, such as linear one, this is always the case. However, if optimizing memory is of concern, s could be modeled as an LR number, with minimal loss of accuracy. Figure 4 shows the output of the fuzzy model, both if s is forced to be an LR number, and if it is modeled as a general fuzzy number, represented by 10 nested intervals (the default choice - which can be overridden). In most cases, using LR numbers to model uncertainty seems sufficient. Note that the mode of the fuzzy number,

which corresponds to the most possible value of s is around 7,5 m, but s could be anywhere between 2 and 47 m. An h-cut of h=0.5 shows that values less than 3 m and larger than 15 m have a possibility of 0.5, meaning that the probability of these extreme events could be as much as one half (since the possibility measure is an upper bound on probability).

With little efforts, a model coded in C++ has been adapted for fuzzy sensitivity analysis using the proposed library, allowing for a quick estimation of uncertainty on the output of the model. The resulting fuzzy number gives both an idea of the most likely value and of worst possible cases. Modeling of uncertainty on the inputs is much simpler than using probability theory, since only an upper bound on the probability is needed, and computations are by far simpler and faster.

Using the library for fuzzy linear regression

The advantages of fuzzy regression over statistical regression are not simplicity and speed. Compared with statistical regression, fuzzy regression is complex and slow, especially for large sample sizes. Fuzzy regression is however helpful precisely when data is scarce, since statistical regression cannot be applied, for it is then impossible the verify its underlying hypotheses or to assess the quality of the fit. As an example of application of fuzzy regression, we discuss the problem of finding the relationship between average annual flow at two measuring points (SM2 and SM3) located on the Sainte-Marguerite river, on the north shore of the Saint-Lawrence river in Québec.

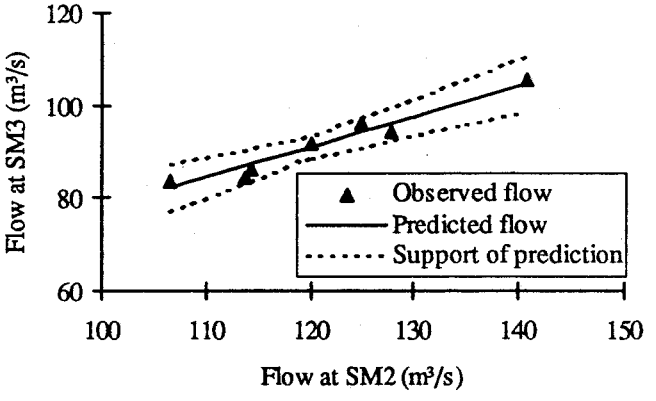


Figure 5: Fuzzy linear regression predicting flow at SM3 from flow at SM2

The sample size of only 7 years (1987-1993) would be considered too small for a meaningful statistical regression. However, for fuzzy linear regression it is quite reasonable. Let Q_2 be the flow at SM2, and Q_3 be the flow at SM3. To estimate the flow at SM3 from SM2, we propose the following fuzzy model, where the parameters $\mathbf{A}=\{A_0,A_1\}$ are symmetrical TFNs, and $\bar{x}$ is the average interannual flow at SM2:

$$Q_3=f(Q_2|\mathbf{A},\bar{x})=A_0+A_1(Q_2-\bar{x}) \qquad (10)$$

With the degree of belief h in the fuzzy model set to h=0,5 and prediction vagueness being minimized over the range of observed flow, the parameters are A_0=(89.3,91.7,94.0)(TFN) and A_1=(.46,.65,.83)(TFN). Figure 5 shows the fuzzy regression obtained.

Since the observed flow fall quite on a straight line, the width of the predicted fuzzy number is relatively small, and the predicted flow, which corresponds to the value of the fuzzy prediction with maximum possibility (the mode), seems sensible. It should be stressed that the support of the predictions, $f(Q_2,h=0|\mathbf{A},\bar{x})$, is not a confidence interval, neither for the prediction line nor for the observations. It rather limits the space containing the set of possible

relationships between Q_2 and Q_3, given the data and the degree of belief in the model (h=0,5 in this case). The shape of these limits has been shown [2] to be a good indicator of the quality of the fit. Indeed, when the model is not adequate, the support of the prediction becomes clearly out of proportion since the parameters must be very vague for the model to include all observations. A consequence of that is of course a sensitivity of fuzzy regression to outliers. Care should therefore be taken to avoid including erroneous data when using fuzzy regression.

Discussion and Conclusion

Fuzzy number arithmetic provides a simple framework for sensitivity analysis in complex models, but has until now found few applications for lack of adequate software tools. We developed a C++ library of object for easily incorporating fuzzy sensitivity analysis into new or existing models coded in C or C++. The library also has capabilities for multiple fuzzy linear regression, and can be used to help study uncertainty in linear models when data is scarce. Examples of applications show that adapting a model for fuzzy sensitivity analysis requires minimal modifications to existing code, and that using fuzzy regression is relatively simple.

While C++ classes are adapted tools for fuzzy sensitivity analysis, since they need to be incorporated into the code, it would be useful to possess a more user-friendly tool for fuzzy regression. Fuzzy regression could be needed in a more complex model, but most applications only require the fuzzy relationship to be established from the available data. We are currently devising a software dedicated to fuzzy linear regression which will use the available libraries, and should be available by the end of the year.

Acknowledgment

The financial support of the Natural Science and Engineering Research Council of Canada and of Hydro-Québec is gratefully acknowledged.

References

1. Bárdossy, Á. Note on fuzzy regression, *Fuzzy Sets and Systems*, 1990, **37**, 65-75.
2. Bárdossy, Á., Bogárdi, I. & Duckstein, L. Fuzzy Regression in Hydrology. *Water Resources Research*, 1990, **26**(7), 1497-1508.
3. Berger, J.O. *Statistical Decision Theory and Bayesian Analysis*, 2nd ed., Springer-Verlag, New York, 1985.
4. Caselton, W.C. & Luo, W. Decision Making with Imprecise Probabilities - Dempster-Shafer Theory and Application, *Water Resources Research*, 1992, **28**(12), 3071-3083.
5. Dubois, D. & Prade, H. Fuzzy sets and statistical data. *European Journal of Operational Research*, 1986, **25**: 345-356.
6. Dubois, D. & Prade, H. *Possibility Theory: An Approach to Computerized Processing of Uncertainty*, Plenum Press, New York, 1988.
7. Ferson, S. & Kuhn, R. Propagating Uncertainty in Ecological Risk Analysis Using Interval and Fuzzy Arithmetic, in Envirosoft/92 (ed. P. Zanetti), pp. 387-401, *Proceedings of the 4th Int. Conf. on Computer Techniques in Environmental Studies*, Computational Mechanics Publications, Southampton, 1992.
8. Heath, R.C. *Basic Ground-Water Hydrology*, U.S. Geological Survey Water-Supply Paper 2220, 1983.
9. Kaufmann, A. *Introduction to the Theory of Fuzzy Subsets*, Academic Press, New York, 1975.

10. Kaufmann, A. & Gupta, M.M. *Introduction to fuzzy arithmetic: Theory and applications*, Van Nostrand Rheinhold, New York, 1991.
11. Moore, R. *Methods and Applications of Interval Analysis*, SIAM Studies on Applied Mathematics, Vol. 2, Philadelphia, 1979.
12. Shafer, G. *A Mathematical Theory of Evidence*, Princeton University Press, New Jersey, 1976.
13. Tanaka, H., Uejima, S. & Asai, K. Linear regression analysis with fuzzy model, *IEEE Trans. Syst. Man Cybern.*, 1982, SMC-12, 903-907.
14. Walley, P. *Statistical Reasoning with Imprecise probabilities*, Chapman and Hall, London, 1990.
15. Zadeh, L.A. Fuzzy sets. *Information and Control*, 1965, **8**, 338.

ANNEXE B: PROJET DE LOGICIEL POUR LA RÉGRESSION FLOUE

Il existe très peu de logiciels permettant d'effectuer une régression floue. Bien sûr, il est possible d'utiliser la librairie proposée dans ce travail. Mais il serait préférable de disposer d'un outil plus facile à manipuler par quelqu'un ne connaissant pas le C++. Le rapport présenté dans cette section propose une interface pour un tel logiciel.

MuFLeR (Multiple Fuzzy Linear Regression): Spécifications de l'interface

Ce document présente les spécifications de l'interface du logiciel MuFLeR pour la régression linéaire multiple floue. L'objectif de MuFLeR est de permettre une utilisation aisée de la partie consacrée à la régression de la librairie C++ de nombres flous conçue par Fortin & Bobée (1996).

Le problème de la régression floue multiple peut être résumé ainsi. Soit un ensemble de K variables indépendantes réelles $\mathbf{X}=\{X_1,X_2,...,X_K\}$ et une variable dépendante réelle ou floue Y. Soit une relation linéaire floue $f(\mathbf{X}|\mathbf{A})$ définie sur le domaine de $\mathbf{X}$ et qui dépend d'un ensemble $\mathbf{A}=\{A_0,A_1,A_2,...,A_K\}$ de K+1 paramètres flous, habituellement supposés être des nombres LRS (Fortin & Bobée 1996) tels que $A_k=(m_k,\alpha_k,c\cdot\alpha_k,a,b)$(LRS), où a, b et c sont des paramètres fixés à priori:

$$f(\mathbf{X}|\mathbf{A})=A_0+\sum_{k=1}^{K} A_k(X_k-\bar{x}_k)$$

où $\bar{\mathbf{x}}=\{\bar{x}_1,\bar{x}_2,\ldots,\bar{x}_K\}$ est le point du domaine de $\mathbf{X}$ où la relation linéaire floue est supposée être la plus précise. Soit un échantillon $\mathbf{w}=\{(\mathbf{x}_j,y_j), j=1,2,...,n\}$ de n observations du couple $(\mathbf{X},Y)$, où $\mathbf{x}_j=\{x_{j,1},x_{j,2},...,x_{j,K}\}$ est un vecteur d'observations des K variables indépendantes. Un modèle de régression flou admissible pour $\mathbf{w}$ est une relation floue $f(\mathbf{X}|\mathbf{A})$ dont la coupe $f(\mathbf{X}|\mathbf{A})(h)$ de niveau h contient toutes la coupe $y_j(h)$ de niveau h de chaque observation y_j:

$$y_j(h)\subseteq f(\mathbf{x}_j|\mathbf{A})(h)$$

Parmi tous les modèles admissibles, on choisit un modèle dont les paramètres flous $\mathbf{A}$ minimisent une fonction objectif $H(\mathbf{A})$. Pour effectuer une régression floue, il faut donc connaître le nombre n d'observations, le nombre K de variables indépendantes, et le type de la variable dépendante (réelle ou floue - et si floue quel type particulier: TFN, TRFN, LR, LRS, nombre flou). Il faut évidemment disposer du tableau $n\times(K+1)$ des observations. De plus, certains paramètres de la régression floue doivent être fixés: le niveau h de crédibilité de la relation, le point $\bar{\mathbf{x}}$ de l'espace des variables dépendantes pour lequel la relation floue est supposée être la plus précise, la fonction objectif $H(\mathbf{A})$ à minimiser (flou moyen des paramètres, flou de prédiction), et enfin la forme générale des paramètres flous, spécifiée par a, b et c.

Une fois tous ces paramètres déterminés, un peu de programmation linéaire permet d'estimer $\mathbf{A}$, c'est à dire l'ensemble des m_k et des α_k. Quelques graphiques peuvent alors être intéressants: un graphique d'un paramètre, un graphique des valeurs prédites de Y en fonction des valeurs observées, et un graphique de $f(\mathbf{X}|\mathbf{A})$ en fonction de $\mathbf{X}$, en y indiquant les observations y_j. Il faut donc développer une interface permettant de spécifier tous les paramètres de la régression floue et d'afficher les résultats appropriés. Dans un premier temps, le logiciel n'aura pas le devoir de construire les fichiers de données, mais seulement de pouvoir en importer à partir d'un fichier ou du presse-papiers. Nous débuterons par une modélisation conceptuelle des données, et nous poursuivrons par une modélisation de l'interface.

Modélisation conceptuelle des données

L'objectif de MuFLeR est de permettre à l'usager de spécifier tous les paramètres d'une régression floue: n, K, $\mathbf{w}$, le type de Y (que nous nommerons *TypeY*), $\bar{\mathbf{x}}$, h, $H(\cdot)$, a, b et c. Il doit aussi être possible d'effectuer des transformation sur les $\mathbf{x}_j$, qui consistent à élever chaque observation à une puissance quelconque, à prendre le logarithme ou l'exponentielle. Celles-ci pourront être représentées par un vecteur de K transformations $\mathbf{r}=\{r_1,r_2,...,r_K\}$. Il doit aussi afficher les résultats de la régression: $\mathbf{A}$ et $f(\mathbf{X}|\mathbf{A})$, qui seront conservés jusqu'à leur élimination par l'usager. Voici la structure proposée d'un fichier de format MuFLeR:

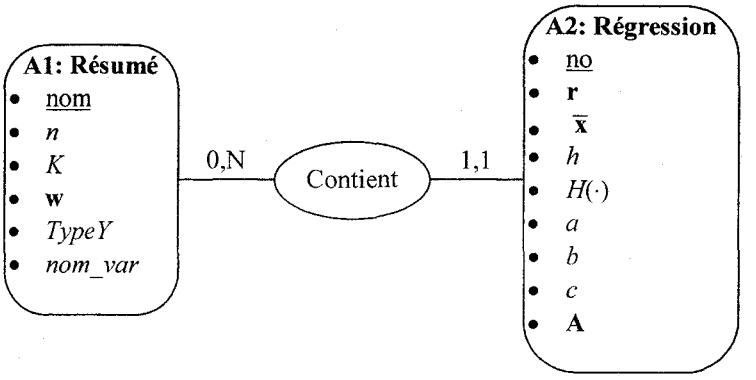


Figure 1: Modélisation conceptuelle des données d'un fichier .FLR

Nous avons ajouté aux structures déjà connues *nom_var*, un vecteur de K+1 chaînes de caractères pour nommer les K variables indépendentes et la variable dépendante. Ces noms se retrouveront comme titres d'axe dans les graphiques. Notons qu'il ne semble pas nécessaire que le logiciel gère plus d'un fichier à la fois en mémoire.

Modélisation de l'interface

Menu de départ

Voici la structure de menu proposée lors de l'ouverture de MuFLeR. Les options entre parenthèses seront désactivées au départ:

1. Fichier	**2. Édition**	**3. (Analyse)**	**4. (Régression)**	**5. Outils**
1.1 Ouvrir	2.1 (Copier)	3.1 Corrélations	4.1 Régression floue	5.1 Éditeur de texte
1.2 Importer	2.2 Coller	3.2 Dispersion		5.2 Options
1.3 (Enregistrer)				
1.4 (Enregistrer sous)				
1.5 (Nommer les variables)				
1.6 (Imprimer)				
1.7 Langue				
1.7.1 Français				
1.7.2 Anglais				
1.8 Quitter				

Figure 2: Menu de l'application MuFLeR

On pourra ajouter par la suite un menu "Fenêtre" standard et un menu d'aide. À chacune des feuilles de l'arbre des menus se rattachera un processus dont le comportement est décrit dans ce qui suit.

Processus reliés au menu

1.1 Ouvrir

Cette option permettra d'aller rechercher un fichier .FLR enregistré précédemment, et d'afficher un résumé de son contenu: son nom, *n*, *K*, le nom de chaque variable (*nom_var*) et le résultat de chaque régression effectuée et conservée. D'abord, s'il y avait déjà un fichier ouvert, le système demandera à l'usager s'il veut enregistrer le fichier précédent avant d'en ouvrir un autre. Le format du fichier sera vérifié, et un message d'erreur sera affiché si le format est erroné. Dans le cas d'une ouverture de fichier réussie, toutes les options désactivées du menu seront activées.

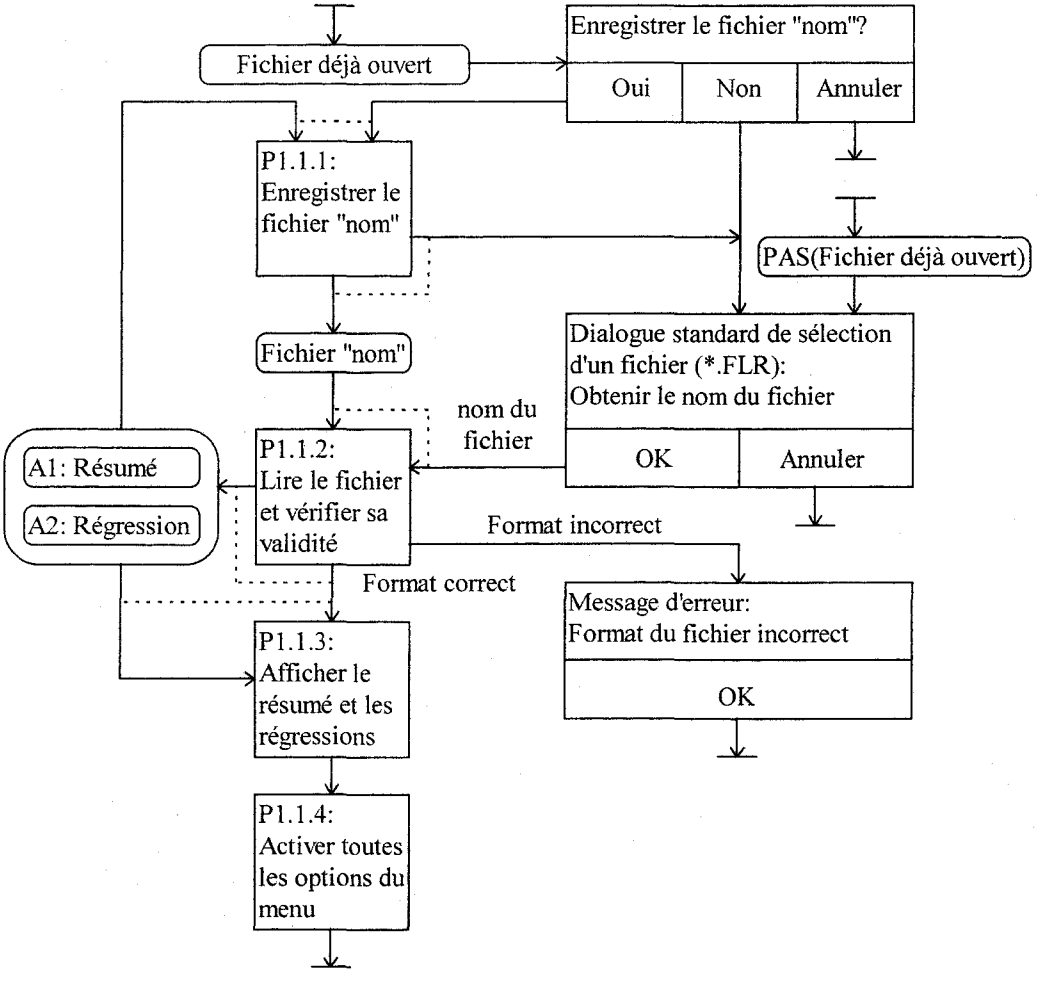


Figure 3: Comportement du système pour l'option 1.1: Ouvrir

1.2 Importer

Cette option permet de lire des fichiers textes pour en importer les données. La seule restriction posée sur la structure de ces fichiers sera qu'ils puissent être lus par l'algorithme suivant à l'aide de l'operateur >> du C++, et que l'on retrouve une observation ($\mathbf{x}$,y) par ligne:

Pour j de 1 à n:
    Pour k de 1 à K:
        Lire $x_{j,k}$
    Lire y_j

MuFLeR devra déterminer lui-même n et K en analysant le fichier, mais devra demander à l'usager de spécifier le type de Y, qui pourra soit être réel, TFN, TRFN, LR, LRS ou nombre flou. D'abord, s'il y avait déjà un fichier ouvert, le système demandera à l'usager s'il veut enregistrer le fichier précédent avant d'en ouvrir un autre. Le format du fichier sera vérifié, et un message d'erreur sera affiché si le format est erroné. Dans le cas d'une ouverture de fichier réussite, toutes les options désactivées du menu seront activées. Notons que le processus P2.1.1 devra donner comme nom au fichier dans l'accumulation A1 le code "NONAME.FLR".

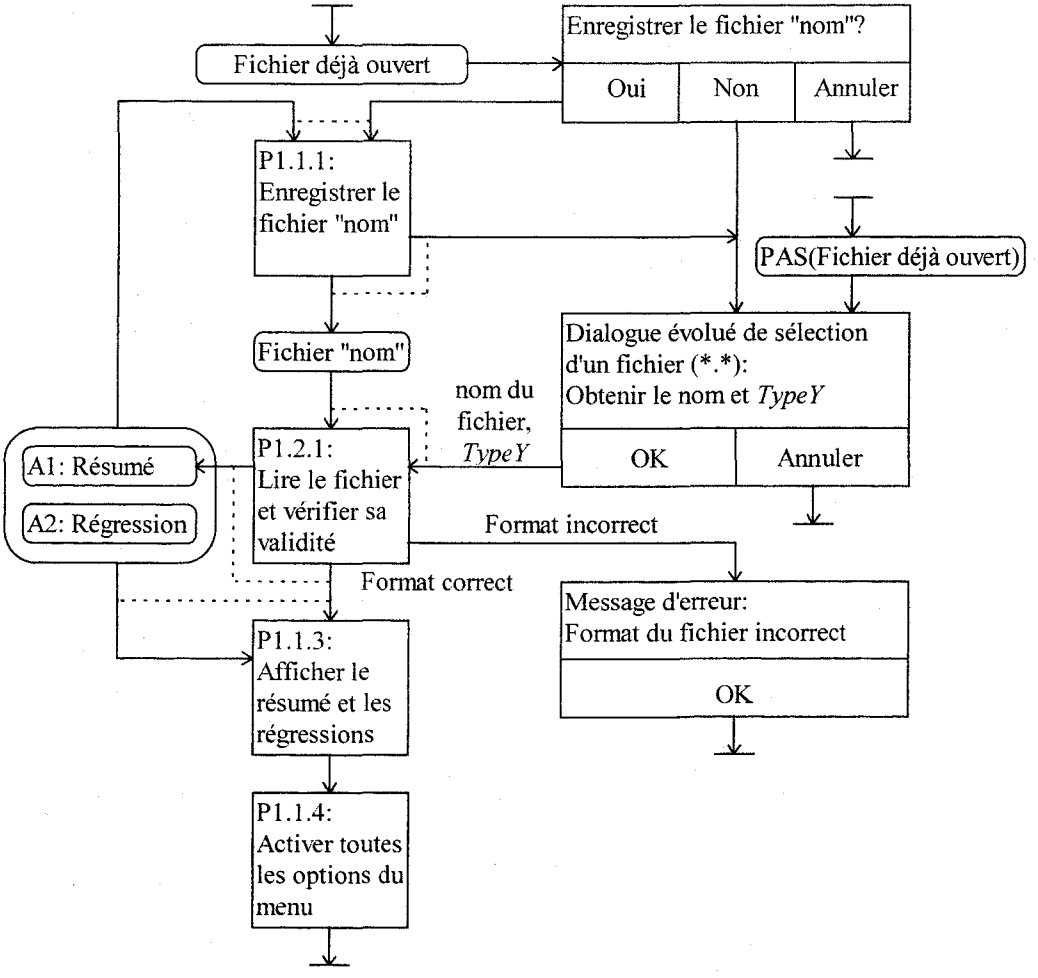


Figure 4: Comportement du système pour l'option 1.2: Importer

1.3 Enregistrer

Cette option consiste à enregistrer le fichier si le nom du résumé est différent de "NONAME.FLR", ou sinon à appeler le processus 1.4: Enregistrer sous. Le nouveau nom du fichier est ensuite inscrit dans le résumé.

1.4 Enregistrer sous

Cette option consiste à demander un nouveau nom de fichier et à enregistrer le travail sous ce nouveau nom. Celui-ci est ensuite inscrit dans le résumé. Un bouton "Annuler" sera prévu.

1.5 Nommer les variables

Cette option permet, à l'aide d'une boîte de dialogue, de modifier les noms des variables (dont le nom par défaut est X1, X2, ..., X*K* et Y). On prévoira un bouton "Annuler", et un bouton "Valeurs initiales" pour revenir aux valeurs par défaut. Il faudra par la suite mettre à jour la fenêtre résumé.

1.6 Imprimer

Cette option permet d'imprimer le contenu des accumulations A1 et A2, ou encore le contenu de la fenêtre active (qui pourra être un graphique). Il faudra demander à l'usager ce qu'il veut, et ensuite utiliser le dialogue standard d'impression. Ou mieux encore, inclure le choix dans le dialogue standard sous la forme de boutons radios. Un bouton "Annuler" sera prévu.

1.7 Langue

L'usager pourra choisir 1.7.1: Français ou 1.7.2: Anglais pour que tous les affichages se fassent dans la langue de son choix. Une librairie a déjà été développée pour gérer l'affichage en plusieurs langues; il faudra en tenir compte lors de la conception du programme entier. Le choix de langue actif sera spécifié par un crochet dans le menu.

1.8 Quitter

Cette option demandera s'il faut enregistrer le fichier (s'il y en a un d'ouvert) avant de fermer l'application. Un bouton "Annuler" sera prévu.

2.1 Copier

Cette option ne sera active que lorsque la fenêtre active à l'intérieur de MuFLeR sera un graphique. Elle copiera le graphique sur le presse-papiers.

2.2 Coller

Cette option, semblable à 1.2: Importer, permettra de coller des données du presse-papiers vers MuFLeR. Les données seront supposées être du même format que pour 1.2: Importer, et le comportement du processus sera le même.

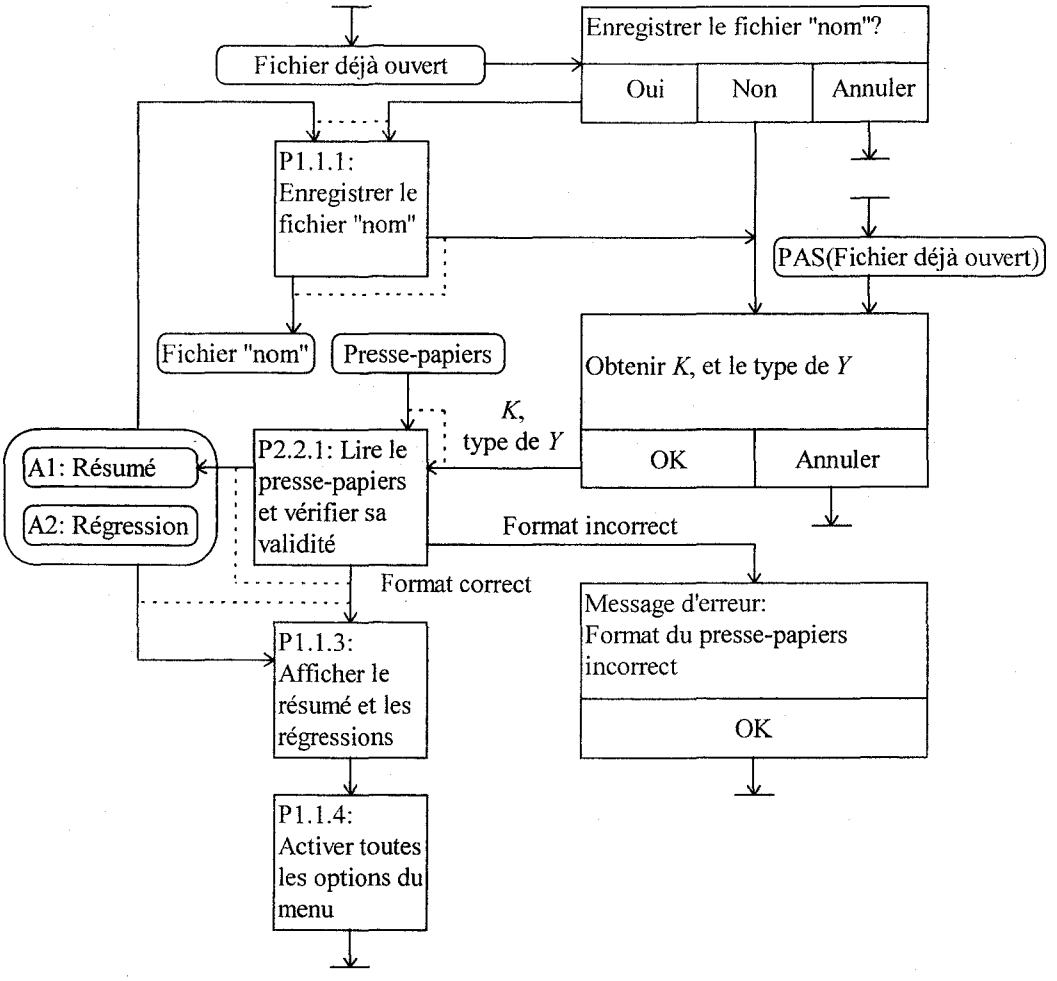


Figure 5: Comportement du système pour l'option 2.2: Coller

3.1 Corrélations

Cette option affiche la matrice des corrélations entre les K+1 variables. Pour chaque variable, il est possible de spécifier une transformation.

3.2 Dispersion

Le but de cette option est de permettre un examen sommaire des données avant de poursuivre avec la régression floue. Il s'agira de s'assurer qu'un modèle linéaire est bel et bien raisonnable, ou que des transformations sont appropriées.

Le comportement du système sera différent selon que K sera plus grand que 1 ou égal à 1. Si K>1, l'usager choisira d'abord à partir de deux listes quelles variables il veut visualiser (incluant les K variables dépendantes et la variable indépendante). La première liste correspondra aux variables qu'il veut voir apparaître en abcisse, et la deuxième aux variables d'ordonnée. Plusieurs variables pourront être choisies dans chaque liste. Chaque variable sera identifiée par son nom dans *nom_var*. Le produit cartésien des deux ensembles de variables sélectionnées formera un ensemble de paires de variables, duquel on enlèvera les répétitions du type (X1,X2) et (X2,X1)

ainsi que les couples de variables semblables du type (X1,X1). Ceci donnera l'ensemble final des paires de variables.

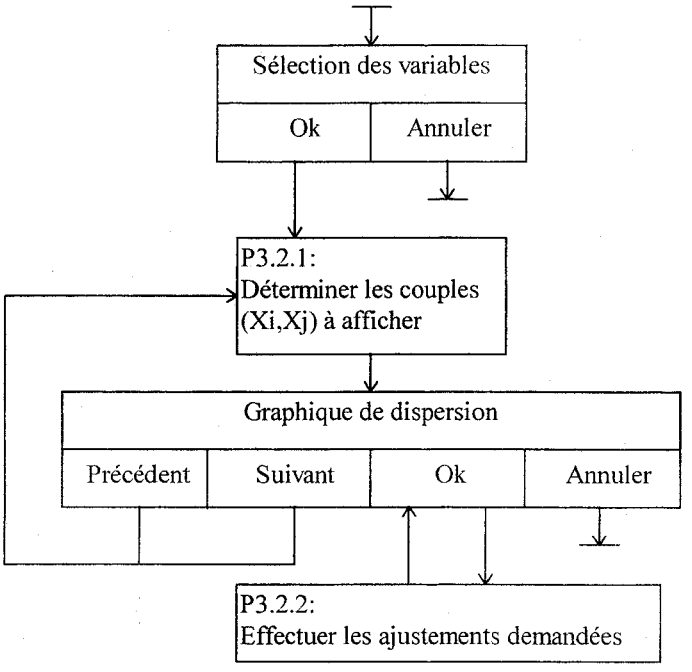


Fig. 6: Comportement du système pour l'option 3.2: Dispersion

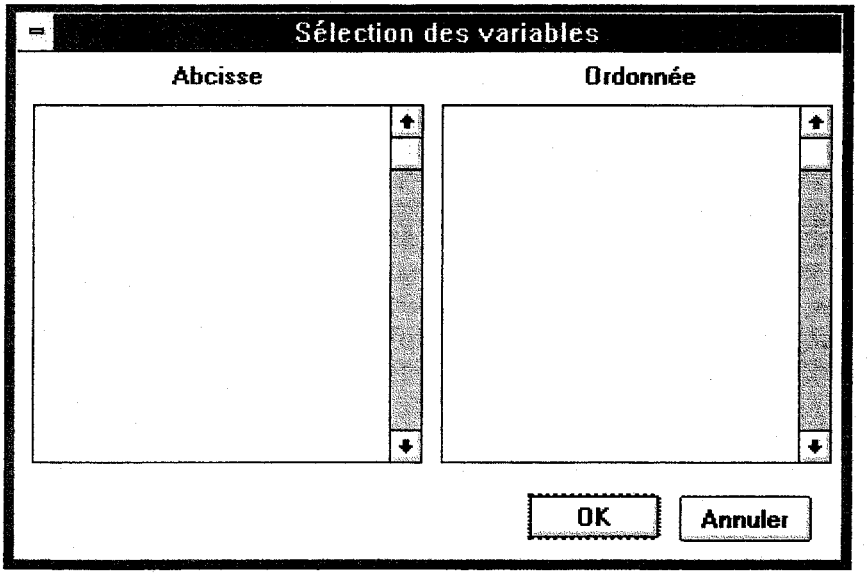


Figure 7: Dialogue de sélection des variables à utiliser pour les graphiques de dispersion

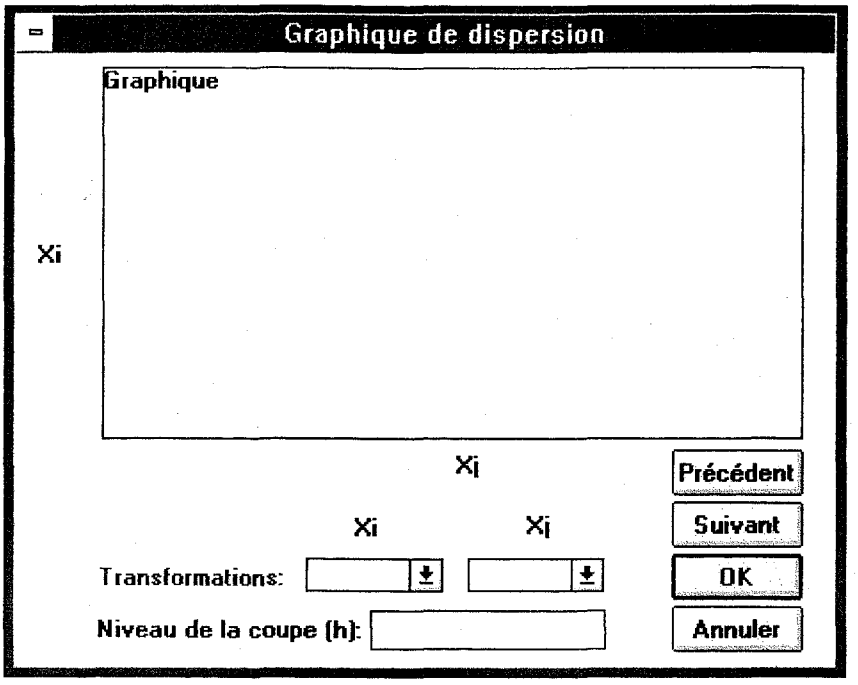


Figure 8: Fenêtre pour l'affichage des graphiques de dispersion

On affichera le graphique de la première de ces paires, (X*i*,X*j*), dans une fenêtre avec les boutons "Précédent", "Suivant" et avec deux champs pour spécifier des transformations sur ces variables. Avec les deux boutons précédent et suivant, l'usager pourra passer à d'autres paires de variables sélectionnées. Dans le cas où K=1, il n'y a qu'un seul graphique possible, Y vs X_1. Ce graphique sera affiché automatiquement. Quelque soit la valeur de K, lorsqu'un graphique contiendra la variable Y et que celle-ci sera floue, on affichera pour chaque observation une coupe h, où h pourra être spécifié dans la fenêtre:

4.1 Régression floue

Lors de l'appel de cette option, une boîte de dialogue s'affichera pour demander à l'usager les valeurs des paramètres $\bar{\mathbf{x}}$, h, $H(\cdot)$, a, b, c et $\mathbf{r}$. La régression sera ensuite calculée, et les résultats seront affichés, à moins qu'un message d'erreur indique un problème numérique. Les résultats incluront tous les paramètres spécifiés auparavant ainsi que la valeur des paramètres flous $\mathbf{A}$. Il sera possible à partir de cette fenêtre de demander le graphique d'un paramètre sélectionné dans la liste (bouton "Tracer"), le graphique des valeurs observées en fonction des valeurs prédites (bouton "Préd. vs Obs.") et si K=1, le graphique de Y en fonction de $\mathbf{X}$ (bouton "Y vs X"). Pour le bouton "Tracer", toutes les variables seront tracées sur un même graphique si aucune n'est sélectionnée. Un bouton "Annuler" permettra d'oublier les résultats de la régression et de retourner à la boîte de dialogue, les paramètres étant conservés. Un bouton "Fermer" permettra de fermer cette fenêtre ainsi que toutes les fenêtres de graphiques se rattachant à cette régression.

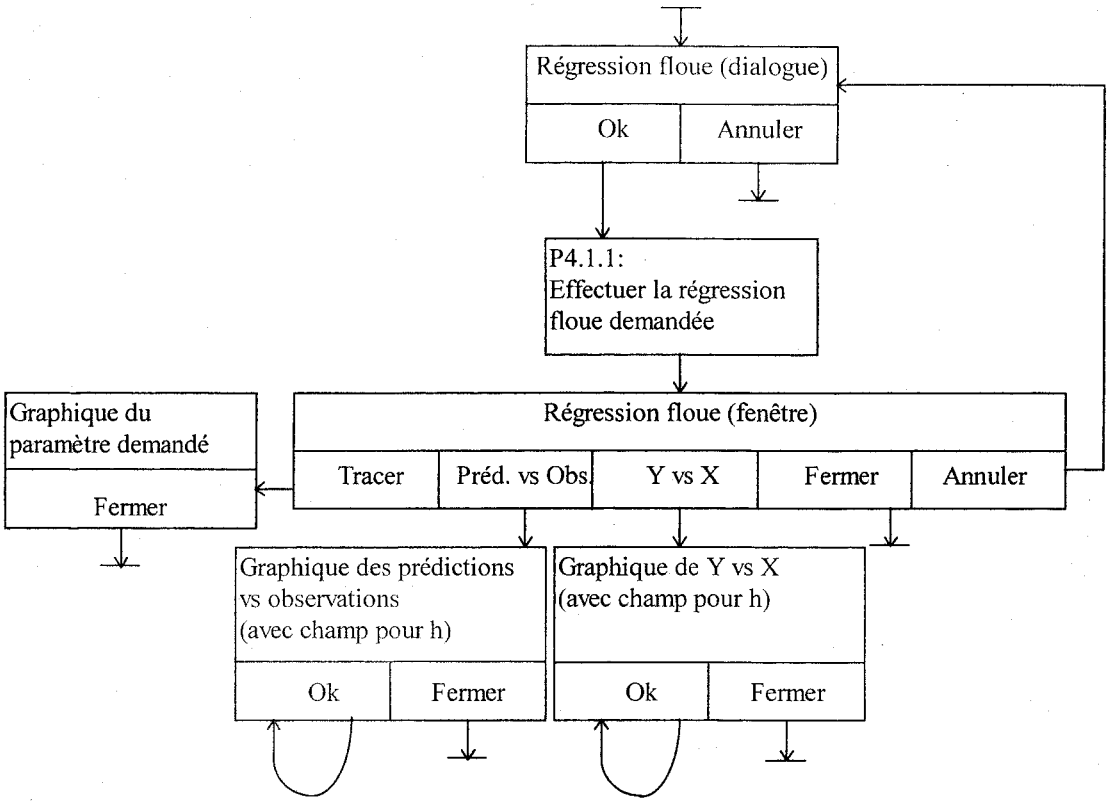


Fig. 9: Comportement du système pour l'option 4.1: Régression floue

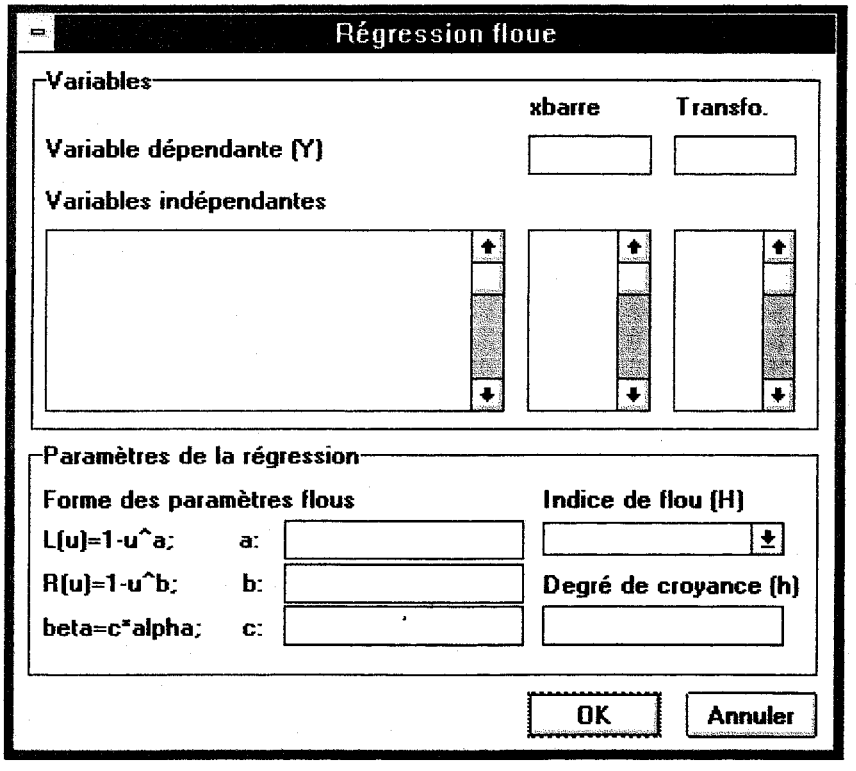


Figure 10: Dialogue de sélection des paramètres d'une régression

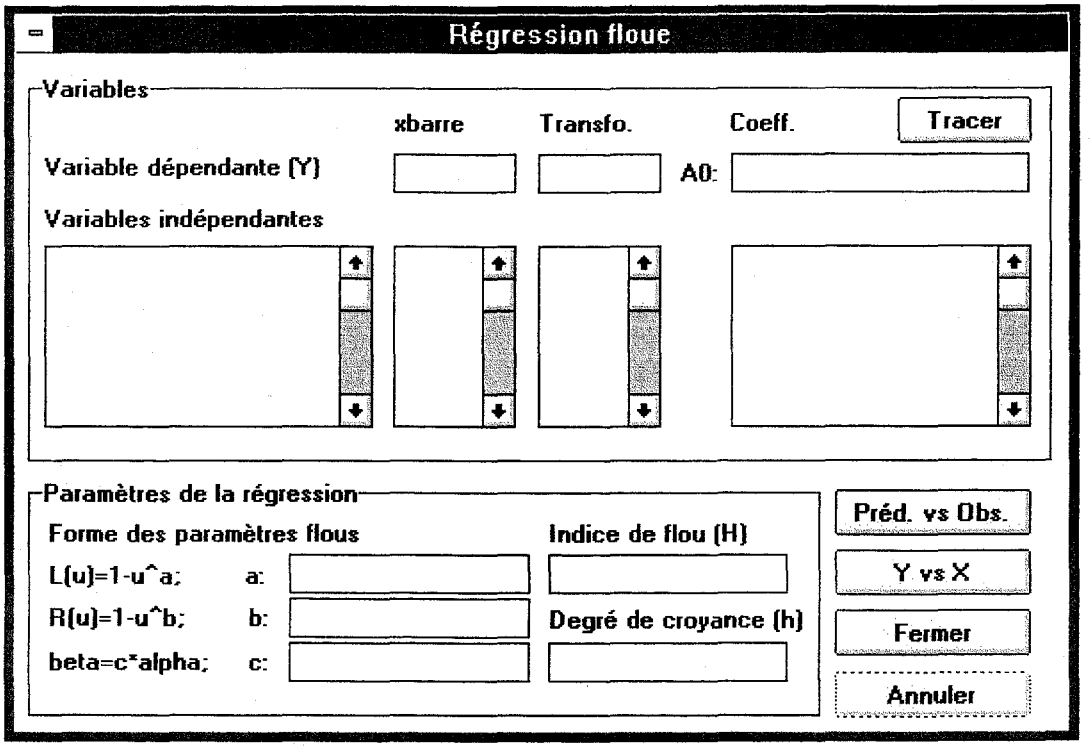


Figure 11: Fenêtre de résultats d'une régression floue

Le bouton "Tracer" permettra d'afficher les fonctions d'appartenance des paramètres graphiquement. Ces paramètres sont des nombres LRS. Le bouton "Préd. vs Obs." permettra de tracer le graphique des prédictions du modèle pour les observations disponibles. Les prédictions, qui seront floues seront représentées par leur coupe de niveau *h*. Si les observations sont aussi floues, il faudra utiliser des ellipses correspondant aux coupes de niveau *h*. Le bouton "Y vs X" ne sera actif que si K=1, et permettra de tracer le modèle de régression accompagné des observations. Voici un exemple de ce type de graphique:

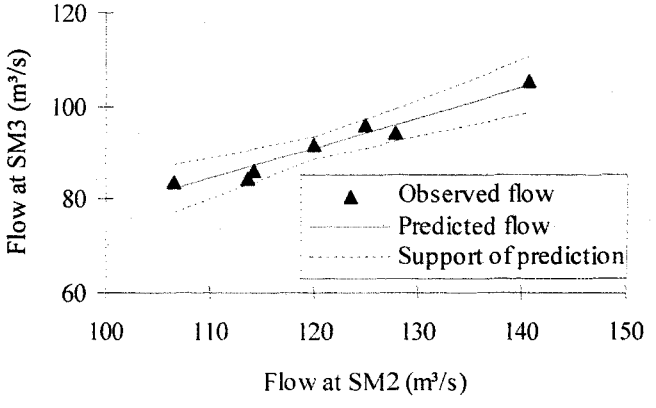


Figure 12: Exemple de graphique Y vs X

5.1 Éditeur de texte

Cette option de menu chargera un éditeur de texte dont le nom sera spécifié dans 3.4: Options. Le but est de permettre facilement à un utilisateur de créer ou de modifier un fichier de données brutes, qu'il pourra ensuite importer.

5.2 Options

Il s'agit ici de permettre à l'usager de définir dans une boîte de dialogue des options de configuration du logiciel, incluant:

- le nom de l'éditeur de texte (par défaut notepad.exe);
- la valeur de h (par défaut 0,5);
- le type de Y parmi: réel,TFN,TRFN,LR,LRS,nombre flou (par défaut réel);
- la valeur de $\bar{x}$ parmi: min, médiane, moyenne, max, (min+max)/2 (par défaut moyenne);
- la valeur de H parmi: flou moyen, flou de prédiction (par défaut flou de prédiction);
- les valeurs de a, b et c (par défaut 1).

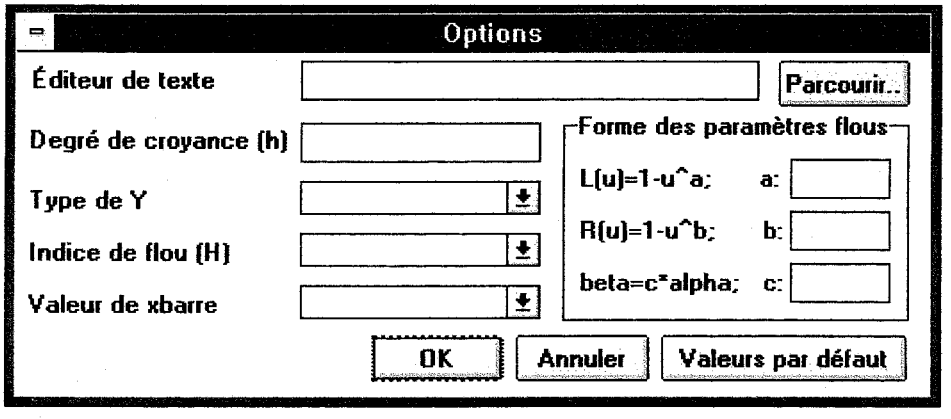


Figure 13: Dialogue de sélection des options

Priorités

L'interface sera développée de façon incrémentale, de façon à ce qu'elle puisse être testée pendant son développement. Les options de menu suivantes seront construites en premier lieu:

Fichier:	Importer, Quitter
Régression:	Régression floue

Dans un second temps, on ajoutera les options suivantes:

Édition:	Copier
Fichier:	Imprimer

On pourra ensuite ajouter les options usuelles:

Fichier:	Ouvrir, Enregistrer, Enregistrer sous
Édition:	Coller

On ajoutera par la suite le menu d'analyse:

Analyse:	Corrélations, Dispersion

Enfin, on ajoutera les options:

Fichier:	Nommer les variables, Langue
Outils:	Éditeur de texte, Options

Une fois le logiciel fonctionnant correctement, on pourra ajouter un menu Fenêtre et un menu d'aide. Cependant, s'il s'avère utile d'incorporer de l'aide contextuelle, il faudrait en tenir compte lors de la conception des dialogues et fenêtres.

Références

Fortin V. et B. Bobée (1996). A C++ library for fuzzy sensitivity analysis and multiple fuzzy linear regression. Accepté pour présentation à la conférence AIENG 96, Clearwater, Floride.