CANTOR

Communications et Systèmes
Systèmes d'Informations
Direction Espace

Édition : **5** Date : **04/09/2000**
Révision : **11** Date : **04/03/2005**

Réf. : ESPACE/MS/CHOPE/CANTOR/MU/001

Manuel d'utilisation de la bibliothèque CANTOR

Rédigé par : le : L. Maisonobe, O. Queyrut CS SI/ESPACE/FDS G. Prat, F. Auguie, S. Vresk	
Validé par : le : G. Prat CS SI/ESPACE/FDS	
Pour application : le : C. Fernandez-Martin CS SI/ESPACE/FDS	

Pièces jointes :

DIFFUSION INTERNE

Nom	Sigle	BPi	Observations
CS SI	ESPACE/FDS		2 exemplaires

DIFFUSION EXTERNE

Nom	Sigle	Observations
CNES	DCT/SB/MS	3 exemplaires

BORDEREAU D'INDEXATION

CONFIDENTIALITÉ : NC	MOTS CLEFS : bibliothèque, mathématiques, C++

TITRE DU DOCUMENT :

Manuel d'utilisation de la bibliothèque CANTOR

AUTEUR(S) : L. Maisonobe, O. Queyrut
G. Prat, F. Auguie, S. Vresk

RÉSUMÉ :
Ce manuel d'utilisation provient d'un document CNES rédigé par Luc Maisonobe. Il décrit la bibliothèque des Composants d'Analyse Numérique Traduits sous forme d'Objets Réutilisables CANTOR. Cette bibliothèque fournit des classes C++ implantant des composants géométriques de base comme les vecteurs et rotations en dimension 3 et les éléments simples sur la sphère unité, des classes d'analyse comme les dérivées automatiques à l'ordre un ou deux, les tableaux de variations et des algorithmes de résolution de zéros, ainsi que quelques composants classiques commes les polynômes, les méthodes de moindres carrés linéaires et les approximations fonctionnelles.

DOCUMENTS RATTACHÉS : ce document vit seul		LOCALISATION :	
VOLUME : 1	NBRE TOTAL DE PAGES : 106 DONT PAGES LIMINAIRES : 3 NBRE DE PAGES SUPPL. :	DOCUMENT COMPOSITE : N	LANGUE : FR
GESTION DE CONF. : Oui	RESP. GEST. CONF. : Luc Maisonobe - CSSI		

CAUSES D'ÉVOLUTION : description des évolutions introduites en version 7.3

CONTRAT : Néant

SYSTÈME HÔTE : Unix – LaTeX 2_ε

MODIFICATIONS

Éd.	Rév.	Date	Référence, Auteur(s), Causes d'évolution
4	0	03/08/1999	SCS/CANTOR/MU/99-001, passage de la documentation en version CS
4	1	12/08/1999	SCS/CANTOR/MU/99-001, modification de la taille du tableau des méthodes publiques de Arc et d'Intervalle
4	2	12/07/2000	SCS/CANTOR/MU/99-001, ajout de la section concernant les évolutions entre les versions
5	0	04/09/2000	SCS/CANTOR/MU/2000-001, passage en feuille de style notechope
5	1	22/11/00	SCS/CANTOR/MU/2000-001, description des modifications de configuration entre les versions 5.3 et 5.4
5	2	05/12/00	SCS/CANTOR/MU/2000-001, description des modifications de configuration entre les versions 5.4 et 5.5, ajout de la description de `cantor-config`
5	3	16/02/01	SCS/CANTOR/MU/2000-001, description des modifications de configuration entre les versions 5.5 et 5.6
5	4	22/06/01	SCS/CANTOR/MU/2000-001, description des modifications de configuration entre les versions 5.6 et 6.0
5	5	23/08/01	SCS/CANTOR/MU/2000-001, description des corrections entre les versions 6.0 et 6.1
5	6	23/08/01	SCS/CANTOR/MU/2000-001, description des corrections entre les versions 6.1 et 6.2
5	7	31/01/02	ESPACE/MS/CHOPE/CANTOR/MU/001, description des corrections entre les versions 6.2 et 6.3
5	8	09/09/02	ESPACE/MS/CHOPE/CANTOR/MU/001, description des corrections entre les versions 6.3 et 7.0
5	9	28/03/03	ESPACE/MS/CHOPE/CANTOR/MU/001, description des corrections entre les version 7.0 et 7.1 et ajout des descriptions des constructeurs et destructeurs ajoutés dans la version 7.1
5	10	28/07/03	ESPACE/MS/CHOPE/CANTOR/MU/001, description des évolutions introduites en version 7.2
5	11	04/03/05	ESPACE/MS/CHOPE/CANTOR/MU/001, description des évolutions introduites en version 7.3

SOMMAIRE

GLOSSAIRE

CLUB
: CLasses Utilitaires de Base

CANTOR
: Composants et Algorithmes Numériques Traduits sous forme d'Objets Réutilisables

MARMOTTES
: Modélisation d'Attitude par Récupération des Mesures d'Orientation pour Tout Type d'Engin Spatial

DOCUMENTS DE RÉFÉRENCE

[DR1] *MARMOTTES – documentation mathématique*
ESPACE/MS/CHOPE/MARMOTTES/DM/001 , édition 4.0, 01/02/2002

[DR2] *MARMOTTES – manuel d'utilisation*
ESPACE/MS/CHOPE/MARMOTTES/MU/001 , édition 5.7, 04/03/2005

[DR3] *Manuel d'utilisation de la bibliothèque CLUB*
ESPACE/MS/CHOPE/CLUB/MU/001 , édition 6.9, 04/03/2005

[DR4] *Racines d'une équation par une méthode hybride – méthode de Brent*
J.-C. Bergès, dts/mpi/ms/mn/98-092 , édition 1.0, 27/11/1998

[DR5] *GNU Coding Standards*
R. Stallman, 09/09/1996

[DR6] *Libtool*
G. Matzigkeit, A. Oliva, T. Tanner et G. Vaughan, édition 1.4.1, 06/04/2001

[DR7] *Autoconf*
D. MacKenzie et B. Elliston, édition 2.52, 17/07/2001

[DR8] *Automake*
D. MacKenzie et T. Tromey, édition 1.5, 22/08/2001

[DR9] *Version Management with CVS*
documentation de CVS 1.10

[DR10] *CVS Client/server*
documentation de CVS 1.10

[DR11] *GNU g++*
R. Stallman

[DR12] *egcs*

DOCUMENTS APPLICABLES

Néant

1 présentation

La bibliothèque CANTOR (Composants et Algorithmes Numériques Traduits sous forme d'Objets Réutilisables) regroupe des classes et des fonctions mathématiques de base. Ces fonctions sont principalement orientées autour de la géométrie sur la sphère unité, mais on trouve aussi des notions plus générales comme les polynômes, les fonctions, et des algorithmes de filtrage ou de recherche de racines.

Ce document décrit la version 7.3 de la bibliothèque.

2 description du contexte

La bibliothèque CANTOR s'appuie sur la bibliothèque CLUB [DR3], sur les bibliothèques optionnelles dont CLUB dépend et sur les bibliothèques standards C++ et C. La liste ordonnée des bibliothèques à spécifier à l'éditeur de liens varie selon le compilateur utilisé pour générer la bibliothèque et selon la disponibilité des bibliothèques optionnelles.

La bibliothèque CLUB ayant subi des évolutions importantes (notamment sur les traitements des erreurs), le tableau 1 résume les compatibilités entre CLUB et CANTOR.

TAB. 1 – compatibilité entre les versions de CLUB et de CANTOR

versions de	
CANTOR	CLUB
< 5.1	< 6.0
5.1	6.0
5.2	6.1 - 6.2 - 6.3
≥ 5.3	≥ 7.0

Les versions de CANTOR ultérieures à la version 5.1 utilisent le mécanisme des exceptions, qui n'est mis en place correctement que si l'édition de liens est réalisée par le compilateur C++ qui a servi à compiler la bibliothèque. Cette condition impérative a un impact à l'exécution, pas lors de l'édition de liens elle-même, ce qui la rend insidieuse. En effet, en cas d'absence du mécanisme de récupération, aucune exception lancée ne peut pas être interceptée, le programme s'arrête alors sur une erreur fatale dès la première exception. Le résultat typique est une violation mémoire avec génération d'un fichier `core`). Il faut noter également que le mélange des compilateurs (SUN et GNU par exemple) ne fonctionne pas.

La bibliothèque MARMOTTES [DR1] s'appuie sur CANTOR.

3 conventions

Un certain nombre de principes ont été respectés lors de la conception et du développement de cette bibliothèque. Ces principes sont placés en exergue ici et ne seront pas répétés lors de la description des classes et des méthodes.

3.1 conventions de nommage

- chaque mot constitutif d'un nom de classe est capitalisé (exemple : `ArcIterateur`) ;
- le premier mot constitutif des méthodes et des attributs est en minuscules, mais les mots suivants sont capitalisés (exemple : `ArcIterateur::nbPoints ()`) ;
- les noms des attributs se terminent par le caractère '_' (exemple : `ArcIterateur::arc_`) ;
- les noms des fichiers déclarant ou implantant les classes sont les noms des classes, auxquels on ajoute le suffixe `.cc` ou `.h` (exemple : `ArcIterateur.cc`) ;
- les noms des fichiers sources n'implantant pas de classes suivent la même convention de nommage, par homogénéité (exemple : `Resolution1.cc`).

3.2 retour des erreurs

Traditionnellement, les fonctions C générant une erreur retournent un entier non nul représentant le code d'erreur, et retournent 0 en cas de succès. Cette convention est respectée, cependant il s'avère que dans une bibliothèque un entier n'est pas suffisant pour que l'appelant génère un message d'erreur pertinent ; de plus en C++ les constructeurs ne retournent aucune valeur qui puisse être testée. La convention adoptée est donc que le code de l'erreur (destiné à être testé par l'appelant) et le message d'erreur (destiné à être lu par l'opérateur) sont également retournés dans une instance de classe spécialisée (dérivée de BaseErreurs) fournie par l'appelant.

- Si la fonction ne retourne pas de valeur testable simplement (par exemple un constructeur), l'instance d'erreur peut elle-même être testée ;
- L'appelant peut s'abstenir de fournir l'instance pour le retour des erreurs[1], auquel cas l'erreur est générée dans un objet local qui est soit retourné à l'appelant par le mécanisme des exceptions soit construit puis détruit localement ;
- Lorsqu'une instance d'erreur contenant un compte rendu d'echec est détruite, son destructeur a pour effet de bord de réaliser l'affichage du message, à l'aide d'une fonction qui peut être personnalisée par l'utilisateur.

Ce mécanisme offre l'avantage de couvrir des besoins relativement larges. Dans le cadre d'un développement rapide de code *jetable*, on ne se préoccupe de rien, on ne donne pas les instances pour les erreurs, et on peut même s'abstenir de tester les codes de retour. Les erreurs sont alors générées et affichées à très bas niveau par la bibliothèque elle-même.

Dans le cadre d'un développement ayant des objectifs de qualité élevés on peut dissocier la génération du message de son affichage en permettant à l'appelant de récupérer l'erreur (soit par exceptions soit en fournissant lui-même les instances pour les erreurs) et de décider s'il doit la corriger (pour éviter son affichage au moment de la destruction de l'objet) ou arrêter proprement ses traitements.

Il est intéressant de remarquer que dans tous les cas (même pour un développement rapide) la génération du message est faite à très bas niveau, là où l'on dispose de l'information la plus complète, et que le formatage de la chaîne de caractères peut être réalisé dans la langue de l'utilisateur si l'environnement comporte les fichiers de traduction des formats internes (si l'environnement n'est pas en place, un message est tout de même affiché, mais bien sûr sans traduction).

Un exemple typique de traitement d'erreur est donné par les lignes suivantes, dans lesquelles l'utilisateur donne l'instance d'erreur (la variable `ce`) dans le seul but de pouvoir tester le résultat de la construction de l'objet RotDBL, mais ne fait pas grand chose de plus (il sait que la destruction de cet objet au moment où le `return 1` sera exécuté affichera proprement les problèmes) :

[1] l'instance est généralement fournie sous la forme d'un pointeur optionnel en dernier argument

```
#include "cantor/DeclDBL.h"

int main (int argc, char **argv)
{ CantorErreurs ce;

  double matrice [3][3];
  litMatrice (matrice);

  RotDBL rotation (matrice, 1.0e-6, &ce);
  if (ce.existe ())
    return 1;

  VecDBL axe = rotation.axe ();
  (void) printf ("la rotation a pour axe :"
                 " (%9.6f, %9.6f, %9.6f), et pour angle :"
                 " %8.4f degrés\n",
                 axe.x (), axe.y (), axe.z (), degres (rotation.angle ()));

  // fin normale du programme
  return 0;

}
```

3.3 utilisation de la STL

Comme celle du C, la norme C++ comprend la définition d'une bibiliothèque standard dont la majorité des fonctions et des patrons de classes provient de la Standard Template Library (STL).

Cette bibiliothèque fournit un ensemble de composants génériques structurés en C++. Ces composants ont été créés dans le but de pouvoir être utilisés à la fois sur les structures fournies par la bibliothèque et sur les structures du langage C++.

La bibiliothèque CLUB implémente des classes et fonctions dont certaines s'apparentent à celles présentées par la STL. Comme il est fortement conseillé de suivre les standards de codage, la version 5.3 de CANTOR utilise les classes de la STL plutôt que leur équivalent CLUB et notamment la classe string au lieu de ChaineSimple.

L'implémentation de la classe CantorErreurs (seule utilisatrice de ChaineSimple) s'en trouve ainsi modifiée. Néanmoins, comme les méthodes concernées sont déclarées protégées, aucune évolution sur les codes utilisateurs de CANTOR n'a besoin d'être réalisée.

Par ailleurs, le document [DR3] fournit les informations nécessaires pour substituer efficacement les ChaineSimples par des strings dans les codes utilisateurs des bibliothèques CHOPE.

4 catalogue

4.1 classes disponibles

AnnotatedArc permet de marquer des arcs frontières lors de la combinaison de champs de vue, les marques permettant ensuite de sélectionner les arcs selon l'opération réalisée (réunion ou intersection) ;

Arc permet de représenter des arcs de grands ou de petits cercles sur la sphère unité ;

ArcIterateur permet d'itérer sur les points d'un Arc avec une précision donnée ;

Braid est une classe utilitaire réalisant l'entrelacement de deux frontières et gérant le marquage des arcs frontières ;

CantorErreurs est une classe de gestion des erreurs internes de la bibliothèque s'appuyant sur la gestion des erreurs de la bibliothèque `club` ;

Cone est une classe gérant la zone la plus simple que l'on puisse définir sur la sphère unité : la calotte sphérique issue de l'intersection de la sphère et d'un cône issu de son centre ;

Creneau gère des ensembles d'intervalles de $\mathbb{R}$ disjoints ;

Field modélise les champs de vue sur la sphère unité, ces champs pouvant être non connexes ;

FonctionApprochee est quant à elle une classe d'approximation fonctionnelle dans un espace vectoriel de fonctions s'appuyant sur MoindreCarreLineaire ;

Intervalle implante la notion d'intervalle dans $\mathbb{R}$;

MoindreCarreLineaire est une classe élémentaire pour les moindres carrés linéaires ;

Node est utilisée conjointement avec AnnotatedArc pour réaliser la propagation des marques selon la topologie de l'entrelacement des arcs frontières ;

Polynome implante la notion de polynôme à une variable, dont les cooefficients et les monômes peuvent être exprimés par des nombres réels ou bien par des valeurs de fonctions [2] (et celles de leurs dérivées premières et éventuellement secondes) ;

Resolution1Iterateur et Resolution2Iterateur permettent de parcourir l'ensemble des zéros d'une fonction retournant un ValeurDerivee1 (respectivement un ValeurDerivee2) sur un intervalle spécifié à la construction de l'instance ;

Rotation implante la notion de rotations (dans un espace de dimension 3), dont les coordonnées peuvent être exprimées par des nombres réels ou bien par des valeurs de fonctions [2] (et celles de leurs dérivées premières et éventuellement secondes) ;

Secteurs permet de découper un Cone en tronçons, et de modéliser les portions d'un cône visibles à travers un objet de type Field ;

StatistiqueEchantillon permet de constituer et d'analyser des échantillons statistiques (valeurs extrêmes, moyenne, écart type) ;

ValeurDerivee1 et ValeurDerivee2 permettent de calculer simultanément les valeurs et les dérivées (première et éventuellement deuxième) d'une fonction sans approximation ;

Variation1 et Variation2 sont des classes gérant des tableaux de variations de fonctions, elles sont surtout destinées à répondre à des besoins internes des algorithmes de recherche de zéros ;

Vecteur3 implante la notion de vecteurs (dans un espace de dimension 3), dont les coordonnées peuvent être exprimées par des nombres réels ou bien par des valeurs de fonctions [2] (et celles de leurs dérivées premières et éventuellement secondes).

Outre ces classes, CANTOR fournit également des paquetages de convertisseurs permettant la conversion entre les différents scalaires utilisables (réels, valeurs de fonctions avec dérivées premières et secondes) ainsi que des paquetages de combinaisons d'opérandes permettant d'utiliser les principales opérations arithmétiques et géométriques avec des opérandes de types différents.

Des algorithmes de recherche de zéros et d'extremums de fonctions sont disponibles sous forme de fonctions, de même qu'une fonction de factorisation de matrice symétrique définie positive sous forme $L.D.L^t$ et une fonction de résolution de système linéaire utilisable après une telle factorisation.

[2] classe `template`

Enfin quelques fonctions utilitaires simples (conversions d'angles en degrés et en radians, recalage d'un angle entre $\alpha - \pi$ et $\alpha + \pi$, fonctions minimum et maximum) sont définies sous forme *inline* dans un fichier d'interface de la classe.

4.2 routines d'interfaçage

Seuls les calculs de rotation en dimension 3 disposent d'une interface avec les langages C et FORTRAN.

4.2.1 interface C

RotAxeAngle permet de construire une rotation à partir d'un axe et d'un angle ;

RotU1U2V1V2 permet de construire une rotation à partir de deux vecteurs $\vec{u_1}$ et $\vec{u_2}$ et de leurs images $\vec{v_1}$ et $\vec{v_2}$;

RotU1V1 permet de construire une rotation à partir d'un vecteur $\vec{u_1}$ et de son image $\vec{v_1}$ (un choix arbitraire est fait au sein de l'ensemble infini des solutions) ;

RotMatrice permet de construire une rotation à partir d'une matrice 3×3 (en corrigeant son éventuelle non-orthogonalité) ;

RotTroisAngles permet de construire une rotation à partir de trois rotations élémentaires (toutes les conventions classiques de CARDAN et d'EULER sont supportées) ;

RotInverse permet de construire la rotation inverse d'une rotation donnée[3] ;

RotComposee permet de construire une rotation par composition de deux autres rotations ;

AxeRot extrait l'axe de la rotation fournie en argument ;

AngleRot extrait l'angle de la rotation fournie en argument ;

AxeAngleRot extrait simultanément l'axe et l'angle de la rotation fournie en argument ;

MatriceRot construit la matrice de la rotation fournie en argument ;

TroisAnglesRot extrait les trois angles des rotations élémentaires qui donnent la rotation fournie en argument une fois combinés (toutes les conventions classiques de CARDAN et d'EULER sont supportées) ;

AppliqueRot calcule l'image d'un vecteur par une rotation ;

AppliqueRotInverse calcule l'image réciproque d'un vecteur par une rotation.

4.2.2 interface FORTRAN

RotAxeAngle permet de construire une rotation à partir d'un axe et d'un angle ;

RotU1U2V1V2 permet de construire une rotation à partir de deux vecteurs $\vec{u_1}$ et $\vec{u_2}$ et de leurs images $\vec{v_1}$ et $\vec{v_2}$;

RotU1V1 permet de construire une rotation à partir d'un vecteur $\vec{u_1}$ et de son image $\vec{v_1}$ (un choix arbitraire est fait au sein de l'ensemble infini des solutions) ;

RotMatrice permet de construire une rotation à partir d'une matrice 3×3 (en corrigeant son éventuelle non-orthogonalité) ;

[3] il faut préciser que les rotations étant représentées en internes par des quaternions, l'opération d'inversion est extrêmement peu coûteuse : elle se borne à changer le signe d'un réel !

RotTroisAngles permet de construire une rotation à partir de trois rotations élémentaires (toutes les conventions classiques de CARDAN et d'EULER sont supportées) ;

RotInverse permet de construire la rotation inverse d'une rotation donnée[3] ;

RotComposee permet de construire une rotation par composition de deux autres rotations ;

AxeRot extrait l'axe de la rotation fournie en argument ;

AngleRot extrait l'angle de la rotation fournie en argument ;

AxeAngleRot extrait simultanément l'axe et l'angle de la rotation fournie en argument ;

MatriceRot construit la matrice de la rotation fournie en argument ;

TroisAnglesRot extrait les trois angles des rotations élémentaires qui donnent la rotation fournie en argument une fois combinés (toutes les conventions classiques de CARDAN et d'EULER sont supportées) ;

AppliqueRot calcule l'image d'un vecteur par une rotation ;

AppliqueRotInverse calcule l'image réciproque d'un vecteur par une rotation.

5 environnement

Seule la bibliothèque CLUB et les bibliothèques standards du langage C++ sont utilisées. Il faut rappeler que CLUB utilise deux variables d'environnement pour gérer la traduction des messages d'erreur, variables dont le nom est configurable à la compilation de la bibliothèque (cf [DR3]).

Le système de traduction de CLUB n'est utilisé dans CANTOR que pour les messages d'erreur. Le domaine de traduction associé est `cantor`. Lors de l'installation de la bibliothèque, les fichiers de traduction anglais et français sont installés.

6 installation

La bibliothèque CANTOR est livrée sous forme d'une archive compressée dont le nom est de la forme `cantor-`*vv.rr*`.tar.gz`, où *vv* est le numéro de version et *rr* est le numéro de révision.

L'installation de la bibliothèque est similaire à l'installation des produits GNU. En premier lieu, il importe de décompresser[4] l'archive et d'en extraire les fichiers, par une commande du type :

```
gunzip -c cantor-vv.rr.tar.gz | tar xvf -
```

Cette commande crée un répertoire `cantor-`*vv.rr* à l'endroit d'où elle a été lancée. Il faut ensuite se placer dans ce répertoire, et configurer les makefiles par une commande du type :

```
./configure
```

On peut modifier les choix du script de configuration par plusieurs moyens. Le cas de loin le plus courant est de vouloir installer la bibliothèque à un endroit spécifique (l'espace par défaut est `/usr/local`), on doit pour cela utiliser l'option `--prefix` comme dans l'exemple suivant :

```
./configure --prefix=/racine/espace/installation
```

Il arrive beaucoup plus rarement que l'on désire modifier les options de compilation, il faut là encore passer par des variables d'environnement (`CXXCPP`, `CPPFLAGS`, `CXX`, `CXXFLAGS`, `LDFLAGS` ou `LIBS`) avant de lancer le script.

[4] l'utilitaire de décompression `gunzip` est disponible librement sur tous les sites `ftp` miroirs du site GNU

Par défaut, le script de configuration recherche la bibliothèque CLUB dans l'environnement par défaut de l'utilisateur et dans le préfixe choisi pour CANTOR. L'option `--with-club[=PATH]` permet d'aider le script à trouver la bibliothèque lorsqu'elle est installée dans des répertoires non standards. Lorque l'on spécifie un chemin du style `/racine/specifique`, les fichiers d'en-tête sont recherchés dans un premier temps sous `/racine/specifique/include` et ensuite sous `/racine/specifique/include/club` et la bibliothèque est recherchée sous `/racine/specifique/lib`.

Si l'on désire partager les options ou les variables d'environnement entre plusieurs scripts `configure` (par exemple ceux des bibliothèques CLUB, INTERFACE, CANTOR et MARMOTTES), il est possible d'initialiser les variables[5] dans un ou plusieurs scripts Bourne-shell. La variable `CONFIG_SITE` si elle existe donne une suite de noms de tels scripts séparés par des blancs, ceux qui existent sont chargés dans l'ordre. Si la variable n'existe pas on utilise la liste par défaut des fichiers `$(prefix)/share/config.site` puis `$(prefix)/etc/config.site` ; cette liste par défaut permet de gérer plusieurs configurations en spécifiant manuellement l'option `--prefix` sans le risque de confusion inhérent aux variables d'environnement peu visibles.

La compilation est ensuite réalisée par la commandes `make` et l'installation par la commande `make install`. La première commande compile localement tous les éléments de la bibliothèque et les programmes de tests, seule la seconde commande installe des fichiers hors de l'arborescence de compilation. Les fichiers installés sont l'archive `libcantor.la`, le répertoire de fichiers d'inclusion `cantor`, et les fichiers de traduction `en/cantor` et `fr/cantor` dans les sous-répertoires de la racine spécifiée par l'option `--prefix` de `configure`, valant `/usr/local` par défaut.

Si l'utilisateur le désire, il peut faire un `make check` après le `make` pour lancer localement les tests internes de la distribution.

Il est possible de désinstaller complètement la bibliothèque par la commande `make uninstall`.

Pour régénérer l'arborescence telle qu'issue de la distribution (en particulier pour éliminer les fichiers de cache de `configure`), il faut faire un `make distclean`.

Une fois l'installation réalisée, le répertoire `cantor-`*`vv.rr`* ne sert plus (sauf si l'on a compilé avec une option de déboguage) et peut être supprimé.

La démarche de compilation normale est de désarchiver, de configurer, de compiler, d'installer, puis de supprimer le répertoire des sources, pour ne conserver que l'archive compressée. Le répertoire des sources n'est pas un espace de stockage permanent, les directives du `Makefile` ne supportent en particulier pas les compilations simultanées sur un espace unique, et le `Makefile` lui-même dépend de la configuration (il n'est d'ailleurs pas livré pour cette raison). Si un utilisateur désire installer la bibliothèque sur plusieurs machines ayant chacune un espace privé pour les bibliothèques (typiquement `/usr/local`) mais se partageant le répertoire d'archivage par un montage de disque distant, il ne faut pas décompresser l'archive dans l'espace commun. On préfèrera dans ce cas une série de commandes du type :

```
cd /tmp
gunzip -c /chemin/vers/l/espace/partage/cantor-vv.rr.tar.gz | tar xvf -
cd cantor-vv.rr
./configure --prefix=/chemin/vers/l/espace/prive
make
make install
cd ..
rm -fr cantor-vv.rr
```

[5] l'option `--prefix` s'initialise à l'aide d'une variable shell `prefix`

7 messages d'avertissements et d'erreurs

La bibliothèque CANTOR peut générer des messages d'erreurs en fonction de la langue de l'utilisateur. La liste suivante est triée par ordre alphabétique du format d'écriture dans le fichier de traduction français d'origine[6]. Le format de la traduction anglaise d'origine est indiqué entre parenthèses. Si un utilisateur modifie les fichiers de traduction d'origine (ce qui est tout à fait raisonnable), la liste suivante devra être interprétée avec une certaine tolérance.

`"ajustement des moindres carrés impossible (matrice singulière)"`
(`"impossible adjustment of least squares (singular matrix)"`) Cette erreur se produit lors d'une tentative d'ajustement d'un filtre de moindres carrés. La forme d'une matrice des moindres carrés conduit nécessairement cette dernière à être symétrique définie positive dès que suffisamment de points ont été accumulés dans l'échantillon. Cette erreur témoigne donc d'un échantillon trop restreint (ce qui peut être dû au nombre total de points inférieur à la dimension du problème ou du fait que le même point a été mis plusieurs fois dans l'échantillon).

`"ajustement du filtre des moindres carrés non effectué"`
(`"least squares filter adjustment not yet available"`) Ce message signale que l'utilisateur a tenté d'extraire les coefficients d'un ajustement par un filtre de moindres carrés, d'évaluer l'erreur quadratique de l'ajustement, ou d'appliquer la fonction filtrée en ayant oublié de refaire cet ajustement depuis le dernier ajout ou retrait de points. Cette erreur indique normalement une erreur de codage de l'appelant.

`"axe d'une rotation nul"`
(`"null rotation axis"`) Cette erreur se produit lorsque l'axe donné en argument d'un constructeur de rotation par un axe et angle a une norme trop faible.

`"cas non implanté\nveuillez contacter la maintenance de CANTOR"`
(`"unimplemented case\nplease contact CANTOR support"`) Ce message est produit par une opération interne de la classe Field appelée notamment lors de la construction d'un champ ou lors des opérations entre champs (intersection, réunion, balayage, ...). Elle provient de la détection de l'échec de toutes les méthodes devant rejeter les arcs frontières parasites créés lors de ces opérations. L'apparition de ce message signale un bug interne de CANTOR !

`"cônes coaxiaux"`
(`"coaxials cones"`) Ce message apparaît lorsque l'on tente de calculer l'intersection de deux cônes coaxiaux.

`"cônes disjoints"`
(`"separated cones"`) Ce message apparaît lorsque l'on tente de calculer l'intersection de deux cônes disjoints.

`"dimension de l'espace d'approximation nulle"`
(`"null dimension of approximation space"`) Cette erreur se produit lors d'une tentative d'ajustement d'un filtre de moindres carrés, lorsque la dimension de l'espace vectoriel d'ajustement n'a pas été initialisée correctement. Ceci peut provenir de l'utilisation directe d'une instance créée par le constructeur par défaut (typiquement un élément de tableau) qui n'a pas été réaffectée par une instance créée normalement avec un argument de dimension.

`"ensemble vide d'intervalles"`
(`"empty set of intervals"`) Ce message est produit lorsque l'on tente d'accéder aux éléments internes (valeur minimale, maximale, ou intervalle d'index donné) dans un créneau vide.

[6] la langue interne est le français sans les accents et c'est le fichier de traduction français qui introduit les accents, le fichier de traduction anglais reste ainsi lisible par un utilisateur anglophone qui n'aurait pas configuré son éditeur de texte de sorte qu'il affiche correctement les caractères utilisant le codage iso-8859-1 (IsoLatin 1)

`"erreur %d dans les fonctions de base de l'approximation"`
(`"error %d in base functions for approximation"`) Cette erreur se produit lors de l'ajout ou du retrait d'un point de référence lors de la constitution d'un échantillon pour une approximation fonctionnelle, ou lors de l'évaluation de la fonction ajustée. Cette erreur signale que la fonction que l'utilisateur avait spécifiée à la construction de l'instance et qui est appelée à ces occasions retourne un code d'erreur (c'est à dire une valeur non nulle).

`"erreur interne de CANTOR, veuillez contacter la maintenance (ligne %d, fichier %s)"`
(`"CANTOR internal error, contact the support (line %d, file %s)"`) Ce message ne devrait normalement pas apparaître ... Il indique qu'un cas non supporté a été rencontré, ce qui temoigne d'une erreur de la bibliothèque. Dans une telle éventualité, il faut prévenir la maintenance.

`"extremité d'un arc alignée avec l'axe"`
(`"endpoint of an arc aligned with the axis"`) Ce message est généré lorsqu'à la construction d'un arc sur la sphère unité, le début ou la fin de l'arc sont alignés avec l'axe, interdisant ainsi de définir correctement les plans délimitant le dièdre de l'arc. Ceci arrive en particulier pour les arcs appartenant à des cônes dégénérés d'angle d'ouverture proche de 0 ou de π. Ce type d'erreur devrait être traité par l'application, elle ne devrait normalement pas remonter jusqu'à l'utilisateur.

`"frontière de champ ouverte"`
(`"open field boundary"`) Ce message indique que deux arcs successifs de la liste composant la frontière d'un champ de vue (ou d'un champ d'inhibition) sont trop éloignés l'un de l'autre, ce qui temoigne d'une erreur de la bibliothèque. Dans une telle éventualité, il faut prévenir la maintenance.

`"indice \"%s\" = %d, hors bornes [%d; %d]"`
(`"index \"%s\" = %d, out of range [%d; %d]"`) Ce message est généré lors de l'utilisation d'un mauvais index de monôme dans un polynôme, ou lors de l'accès à un intervalle inexistant dans un créneau. La chaîne affichée précise le type de l'indice fautif.

`"matrice non orthogonalisable"`
(`"non orthogonalisable matrix"`) Ce message est généré lors de la construction d'une rotation par sa matrice, si celle ci ne peut être orthogonalisée après dix itérations. Ceci indique probablement un problème important sur la matrice, ou une erreur dans le seuil de convergence à respecter.

`"matrice singulière"`
(`"singular matrix"`) Ce message est généré lors de la tentative de factorisation d'une matrice sous forme LDL^T. Ceci indique probablement un problème important sur la matrice, ou une erreur dans le seuil de convergence à respecter.

`"norme d'un vecteur nulle"`
(`"null vector"`) Ce message provient d'une tentative de normalisation (ou d'orthonormalisation) d'un vecteur de norme trop faible.

`"opération sur un polynôme invalide"`
(`"operation upon invalid polynom"`) Ce message est généré lors d'une tentative d'utilisation d'un polynôme qui n'a pas encore été initialisé. Ceci peut provenir de l'utilisation directe d'une instance créée par le constructeur par défaut (typiquement un élément de tableau) qui n'a pas été réaffectée par une instance créée normalement avec des coefficients de monômes.

`"ordre de rotations élémentaires inconnu : %d\nordres connus :"`
(`"unknown elementary rotations ordre: %d\nknown orders:"`) Ce message est généré lors de la construction d'une rotation par trois angles élémentaires ou dans l'extraction de ces angles, lorsque l'ordre des rotations élémentaires n'est pas reconnu. Cette erreur ne peut survenir qu'avec l'interface FORTRAN qui utilise

des entiers, les interfaces C et C++ utilisent des types énumérés. Ce message est suivi de la liste des ordres reconnus, sous la forme des PARAMETER déclarés dans le fichier `cantdefs.f`.

`"pas de base de fonctions d'approximation"`
(`"no base functions for approximation"`) Cette erreur se produit lors de l'ajout ou du retrait d'un point de référence lors de la constitution d'un échantillon pour une approximation fonctionnelle. Cette erreur signale que l'utilisateur n'a pas spécifié de pointeur de fonction à la construction de l'instance. Ceci peut provenir de l'utilisation directe d'une instance créée par le constructeur par défaut (typiquement un élément de tableau) qui n'a pas été réaffectée par une instance créée normalement avec un pointeur de fonction non nul.

`"singularite au niveau du calcul des angle de rotation"`
(`"rotation angle computation: singularity found"`) Ce message est généré lorsque, lors du calcul des anges de rotation d'Euler ou de Cardan, des incertitudes sur les angles apparaissent notamment au voisinage de 0 degré pour les angles d'Euler ou au voisinage de 90 degrés pour les angles de Cardan. Ces singularités ne permettent pas de calculer les angles de rotation avec précision mais ce problème peut être contourné en changeant le type de la rotation.

8 tests

La bibliothèque CANTOR dispose d'un certain nombre de tests de non régression automatiques que l'on peut exécuter par la commande `make tests` disponible dans la distribution standard. Cette commande lance les tests existants et compare automatiquement (par un `diff`) le résultat avec le résultat de référence archivé. Seules les différences sont affichées, elles devraient normalement se limiter aux lignes contenant les chaînes du système de contrôle de versions RCS.

Les classes disposant de tests spécifiques sont :
- Arc (et ArcIterateur)
- Cone
- Creneau
- Field
- FonctionApprochee
- Intervalle
- Polynome
- Resolution1Iterateur
- Resolution2Iterateur
- Rotation (plus des tests spécifiques pour les angles de CARDAN et d'EULER)
- StatistiqueEchantillon
- ValeurDerivee1
- ValeurDerivee2
- Vecteur3

La classe MoindreCarreLineaire est testée par le biais des tests de la classe FonctionApprochee laquelle n'est qu'une surcouche utilisée dans un cas particulier d'ajustement. Les classes AnnotatedArc, Braid et Node sont testées par l'intermédiaire de la classe Field. De même les fonctions de résolution et les classes Variation1 et Variation2 sont testées par l'intermédiaire des classes Resolution1Iterateur et Resolution2Iterateur.

9 maintenance

La maintenance de la bibliothèque CANTOR s'effectue selon quelques principes qui concernent la portabilité, les outils utilisés, les procédures de maintenance, les fichiers descriptifs et l'archivage sous `cvs`.

9.1 portabilité

La bibliothèque a dépassé le stade de l'outil spécifique d'un département et est utilisée sur plusieurs sites dans des environnements différents. La portabilité est donc un point fondamental à garder en permanence à l'esprit.

Le modèle suivi a donc naturellement été celui de la lignée de produits GNU. Le document "GNU coding standards" est très utile pour comprendre cette organisation (cf [DR5]). Dans cet esprit, l'environnement de maintenance nécessite beaucoup d'outils mais les utilisateurs finaux n'ont guère besoin que de `gunzip` pour décompresser la distribution et d'un compilateur C++.

9.2 environnement de maintenance

Les produits suivants sont indispensables :

- `libtool` [DR6] (version 1.5.14 au moins)
- `autoconf` [DR7] (version 2.59 au moins)
- `automake` [DR8] (version 1.9.5 au moins)
- `cvs` [DR9] et [DR10]
- `g++` [DR11] (version 3.3 au moins)
- `gzip`
- GNU `m4`
- GNU `make`
- `perl`
- TEX/LATEX/`dvips` (de plus `xdvi` est recommandé)
- la classe LATEX `notechope`
- les paquetages LATEX `babel` et `longtable`

9.3 installation de l'environnement de maintenance

Le développeur récupère tout d'abord le module cantor par une commande `cvs checkout cantor`, il lui faut ensuite générer certains fichiers. Il suffit de passer quatre commandes pour obtenir un environnement complet :

```
aclocal
autoheader
autoconf
automake
```

aclocal génère le fichier `aclocal.m4`

autoheader génère le fichier `src/CantorConfig.h.in`

autoconf génère le script `configure`

automake génère tous les fichiers `Makefile.in`

9.4 compilation

Une fois les fichiers indiqués au paragraphe précédent créés, on se retrouve dans une situation similaire à celle d'un utilisateur qui reçoit la distribution (on a même quelques fichiers en plus, par exemple ceux liés à la gestion `cvs`). Il suffit alors de générer les fichiers `Makefile` par la commande :

```
./configure
```

ou bien

```
./configure --prefix=$HOME
```

si l'on préfère travailler entièrement dans l'environnement de maintenance.

Il faut noter que les `Makefile` générés savent non seulement compiler la bibliothèque, mais qu'ils savent également relancer les commandes initialisant le mécanisme, ceci signifie que d'éventuelles modifications des fichiers `configure.ac` ou `Makefile.am` utilisés par les commandes précédentes seront correctement répercutées partout.

Par défaut, la bibliothèque CANTOR est générée sous forme partagée. Ceci comporte de nombreux avantages par rapport aux bibliothèques statiques mais impose également des contraintes aux développeurs : le temps de compilation d'une bibliothèque partagée est deux fois plus long, le déboguage est plus difficile ... Il peut donc être intéressant de générer CANTOR en statique, et ceci peut être réalisé en passant l'option «–disable-shared» à `configure`.

9.5 procédures de maintenance

L'ensemble des sources (que ce soient les sources C++ ou les fichiers de configuration des outils de génération de scripts) sont gérés sous `cvs` (cf. [DR9] et [DR10]). Les fichiers pouvant être générés automatiquement ne *sont pas gérés sous* `cvs`.

Il ne faut bien sûr pas éditer les fichiers générés, mais éditer les fichiers sources correspondant. Ces fichiers sources sont de plus considérablement plus simples à comprendre. La difficulté est de savoir quels fichiers sont générés et à partir de quels fichiers sources. On ne peut pas toujours se fier au nom, ainsi `src/CantorConfig.h.in` et tous les fichiers `Makefile.in` sont générés, leur suffixe .in signifie simplement qu'une fois générés (par `autoheader` et par `automake` respectivement) ils servent de sources à `autoconf` (qui génère alors `src/CantorConfig.h` et `Makefile`). Les fichiers éditables sont donc : `configure.ac`, et tous les `Makefile.am`.

D'autre part la bibliothèque est aussi maintenue à l'aide du mécanisme de `ChangeLog` qui présente un avantage majeur : les modifications sont présentées dans l'ordre historique des actions de maintenance, ce qui d'une part est en corrélation avec le processus de maintenance et d'autre part peut aider à déterminer par exemple à quels moments certains bugs ont pu être introduits.

Pour tout changement de fichier, il est recommandé de mettre une entrée dans le fichier `ChangeLog` (il y a un fichier de ce type pour chaque sous-répertoire). Sous `emacs` il suffit d'utiliser la commande `M-x add-change-log-entry` en étant à l'endroit où l'on a fait la modification, `emacs` remplissant seul la date, l'auteur, le nom de fichier, et le contexte (nom de fonction, de classe, ...). Pour savoir comment remplir ce fichier, il est recommandé de lire le document décrivant le standard [DR5]. Ces modifications de niveau source ne doivent pas être mises dans le fichier `NEWS`, qui contient les nouveautés de niveau utilisateur, pas développeur.

Pour savoir ce qui peut poser des problèmes de portabilité et comment résoudre ces problèmes, il est fortement recommandé de lire le manuel `autoconf` [DR7] (à cette occasion, on pourra également se pencher

sur le manuel `automake`). On peut également utiliser `autoscan` (qui fait partie de la distribution `autoconf`) pour détecter automatiquement les problèmes communs et proposer des macros les prenant en compte pour `configure.ac`.

Pour faire les tests, il faut utiliser la cible `check` du `Makefile`.

Les fichiers décrivant les spécificités de la bibliothèque CANTOR. Ces fichiers concernent soit l'utilisation soit la maintenance de la bibliothèque.

9.5.1 fichiers descriptifs destinés à l'utilisateur

`README` donne une définition globale de la bibliothèque et indique les particularités de l'installation pour certains environnements.

`NEWS` décrit l'évolution de la bibliothèque, il indique les changements visibles par l'utilisateur.

9.5.2 fichiers descriptifs destinés au mainteneur

`README.dev` définit les principes de maintenance, décrit l'environnement nécessaire pour maintenir la bibliothèque, rappelle les commandes à exécuter à l'aide des produits GNU pour créer un espace de développement et compiler la bibliothèque.

9.6 archivage

La politique d'archivage dans le serveur `cvs` est qu'il faut archiver à chaque nouvelle fonctionnalité ajoutée, ou à chaque bug corrigé. Ceci permet de cerner plus facilement les portions de code touchées (pour les `cvs diff` et autres `cvs update -j`). Il n'est pas recommandé d'archiver des versions intermédiaires non compilables (on peut cependant y être obligé si plusieurs développeurs doivent s'échanger les fichiers).

A priori, les incrémentations de versions ne se font qu'à l'occasion de distributions hors de l'équipe de développement et lors des créations de branches pour des corrections de bugs ou des évolutions susceptibles de durer. Pour les distributions, les tags doivent être de la forme : release-4-5.

À chaque nouvelle distribution, le fichier `NEWS` doit être mis à jour avec toutes les informations pertinentes pour les utilisateurs (l'objectif est donc différent de celui des fichiers `ChangeLog`).

Pour générer une distribution, utiliser la cible `dist` du `Makefile` (il existe également une cible `distcheck` qui permet de vérifier cette distribution). Le numéro de version de la distribution est paramétré par la macro `AC_INIT` dans le fichier `configure.ac`, ce numéro est ensuite propagé sous forme d'un `#define` dans le fichier `src/CantorConfig.h` généré par configure, et c'est ce `#define` qui est utilisé par la fonction cantorVersion.

10 changements depuis les versions précédentes

10.1 évolutions entre la version 7.2 et la version 7.3

Correction d'une erreur dans l'utilisation des champs de vue. Lors de la réduction de l'arc test pour les calculs de présence d'un point dans un champ, l'arc pouvait changer de direction et passer par les points qui avaient été interdits au préalable (BIBMS-FA-2811-199-CN).

10.2 évolutions entre la version 7.1 et la version 7.2

Correction d'une erreur de calcul de l'étendue d'un arc vu d'un point quelconque de la sphère, qui conduisait à prendre de mauvaises directions de test lors des constructions de champs de vue (FA 22).

Un test de configuration a été corrigé. Ce test estimait que g++ 3.x ne respectait pas la spécification de complexité de `list::splice` alors qu'il la respecte tout à fait (FA 25).

Une erreur d'initialisation introduite lors des interventions qualité du projet ATV a été corrigée.

Les scripts de configuration ont été mis à niveau par rapport aux versions courantes des outils de développement GNU (`autoconf` version 2.57, `automake` version 1.7.5 et `libtool` version 1.5). Cette modification n'a pas d'impact pour les utilisateurs (DM 24).

10.3 évolutions entre la version 7.0 et la version 7.1

Pour répondre aux exigences qualité du projet ATV, les constructeurs, destructeurs et opérateurs d'assignation ont été explicitement définis dans chaque classe de la bibliothèque. Si ces méthodes ne devaient pas être disponibles, elles ont été créées en limitant leur accès (protected ou private). L'ordre des sections public, protected et private a par ailleurs été uniformisé afin de faciliter les activités de maintenance. Ces modifications n'ont aucun impact sur le code appelant des bibliothèques.

Enfin les extensions des fichiers sources qui étaient de la forme .cc ont été modifiés en .cpp afin de faciliter un portage ultérieur de la bibliothèque sous Windows.

10.4 évolutions entre la version 6.3 et la version 7.0

Une erreur du compilateur C++ de SUN (qui ne respecte pas le standard du langage) est désormais automatiquement détecté et contourné (FA 0017).

Le traitement des champs sur la sphère unité a été largement amélioré de façon à corriger des erreurs de tests d'appartenance dans des cas limites (FA 0015). Les arcs frontière sont désormais orientés (l'intérieur est à gauche de la frontière, l'extérieur à droite) et les tests d'appartenance sont basés sur le sens de la première traversée d'une trajectoire quelconque partant du point test et franchissant la frontière. Cette nouvelle méthode réduit la complexité des classes, permet de détecter de façon naturelle des cas considérés comme litigieux par la méthode précédente et devrait être plus robuste dans les cas limites.

Les droits de copies ont été mis à jour vis à vis des différents outils GNU utilisés dans le cadre de ce projet.

Le test prgm_FonctionApprochee, utilisant un générateur de nombre pseudo-aléatoire, pouvait ne pas passer sous certaines plateformes du fait de sa sensibilité numérique (FA 0010). Le générateur a été modifié afin de supprimer l'étape la plus sensible et ainsi supprimer une partie des erreurs numériques potentielles.

10.5 évolutions entre la version 6.2 et la version 6.3

Des erreurs d'intersections d'arcs dans des cas limites ont été corrigées. Lorsque des intersections d'arcs faisaient intervenir des points très proches des extrémités de l'arc, le fait de déclarer le point d'intersection dans ou hors de l'arc dépendait de la taille de l'arc, ce qui introduisait des incohérences lors du traitement

de trois arcs de rayons très différents dans des calculs deux à deux. Ceci pouvait en particulier se produire avec l'introduction de petits arcs de contournement de sommets dans les tests de champs de vue (FA 0002).

Une erreur d'échelle dans la classe Secteurs a été corrigée. Cette erreur conduisait les vecteurs debut et fin calculés par la méthode vecteurs à ne pas être normés lorsque le demi angle d'ouverture du cône n'était pas $\pi/2$.

Ajout d'une protection contre les singularites dans le calcul des angles de rotation de Cardan et d'Euler. Les angles de Cardan ou d'Euler ne peuvent pas être extrait de façon fiable dans tous les cas de figures. Il faut gérer ces cas et une erreur est maintenant générée si un tel cas se produit.

Le calcul des cônes de rejet (dispositif permettant d'accélérer les tests de présence d'un point dans un champ) conduisait à sélectionner un cône au sein d'un ensemble et à ne tester sa validité qu'après coup et à désactiver cette accélération dans ce cas. Désormais, seuls des cônes valides peuvent être construits, l'accélération est donc active dès lors que l'on peut trouver au moins un tel cône.

Le contexte dans lequel l'erreur révélée par la FA 0002 se produisait résultait entre autres d'algorithmes de calculs de rapprochements et d'évitements de sommets trop grossiers. Ces algorithmes ont été améliorés. Désormais, moins d'évitement sont nécessaires lors d'un test de présence d'un point dans un champ. De plus, ces évitements ne nécessitent plus de passer par des arcs microscopiques qui poussent les algorithmes à leurs limites.

La correction des cas limites dans les intersections d'arcs et la refonte des champs de vue permet désormais de modéliser des arcs complets comme ayant réellement un angle de balayage de 2π et des extrémités confondues et pas comme des arcs de balayage $2\pi - \varepsilon$ et ayant des extrémités proches. On construit donc les arcs complets de façon plus exacte, ce qui évite d'introduire artificiellement des arcs frontières dégénérés mais non nuls dans la construction des champs de vue.

Correction de l'oubli de protection des caractères spéciaux dans le fichier acinclude.m4, à l'occassion du passage aux derniéres versions des outils GNU, causant la non récupération de la variables de positionnement de la langue via la configuration de Club.

La suppression du fichier CantorConfig.h, de la liste des fichiers constituant le paquetage, a été effectuée dans un soucis de clarté. En effet, ce fichier est crée à partir du fichier CantorConfig.h.in et n'a nul besoin d'être livré.

10.6 évolutions entre la version 6.1 et la version 6.2

Les tests d'appartenance d'un point ou d'un secteur à un champ sur la sphère (classe Field) utilisent désormais un premier test grossier basé sur un cône unique englobant complètement le champ pour accélérer les traitements. Les calculs fins ne sont déclenchés que si ce premier test grossier ne permet pas de trancher. Cette modification est une optimisation, elle ne change pas les fonctionnalités de la bibliothèque.

Une erreur mémoire détectée par `purify` a été corrigée. Un itérateur était incrémenté en fin d'une boucle alors qu'il avait été effacé lors de la dernière itération

Les versions des outils de la suite de développement GNU (`libtool`, `autoconf` et `automake`) ont été prises en compte. Ces versions facilitent le développement en réduisant les dépendances par rapport aux autres produits GNU et en simplifiant les macros de test.

10.7 évolutions entre la version 6.0 et la version 6.1

Il est désormais possible de créer des arcs en spécifiant un angle de balayage négatif. Dans ce cas, l'arc débute bien au niveau du vecteur de début spécifié, mais l'axe spécifié est inversé, l'angle d'ouverture est remplacé par son complémentaire et l'angle de balayage est inversé. L'arc construit correspond donc aux points de la sphère spécifiés et parcourus dans le même sens, mais les éléments internes sont rendus plus conformes à une repésentation canonique.

Il est désormais possible de spécifier des valeurs négatives pour l'angle d'étalement lors de la construction de champs sur la sphère unité par balayage.

La construction de champs sur la sphère unité par balayage a été corrigée de façon à gérer correctement le cas où l'axe de la rotation est aligné avec l'axe de l'un des arcs de la frontière du champ qui sert de base au balayage.

10.8 évolutions entre la version 5.6 et la version 6.0

Les classes AnnotatedArc, Braid, Field, Node et Secteurs ont été transférées depuis la bibliothèque MARMOTTES.

L'opération de traînage d'un arc sur la sphère pour construire des champs de vue a été complètement revue. L'ancienne méthode n'étais pas considérée comme suffisamment robuste dans les cas dégénérés (rotation nulle, tour complet, axe de la rotation passant par la frontière, ...). De plus cette méthode engendrait en cours de traitement des arcs exactement superposés les uns aux autres, qu'il était parfois difficile d'identifier avec les heuristiques existantes. La nouvelle méthode préserve plus d'information sur l'opération en cours et devrait se comporter correctement dans les cas dégénérés.

Les rotations en dimension 3 peuvent désormais être construite à partir de trois angles correspondant à des rotations élémentaires autour d'axes canoniques. Toutes les ordres possibles pour les angles de Cardan et d'Euler sont supportées. Il est également possible d'extraire ces angles à partir d'une rotation déjà construite.

Deux erreurs dans la classe Field ont été corrigées. La première se produisait lors du filtrage de secteurs par un champ, qui pouvait conduire à des résultats complètement faux, la seconde se produisait lors de l'étalement d'un champ (qui est également utilisé lors de l'application d'une marge) et pouvait également conduire à des résultats complètement faux.

Une erreur dans la classe Arc a été corrigée. Lors de l'inversion d'un arc de longueur nulle, on obtenait un arc faisant un tour complet au lieu d'un autre arc de longueur nulle.

La bibliothèque cantor était la seule de la trilogie CLUB/CANTOR/MARMOTTES à ne pas encore utiliser le mécanisme des exceptions pour la gestion de ses erreurs. C'est désormais chose faite. Cette évolution conduit à des changements de signatures de plusieurs méthodes et fonctions globales (toutes les fonctions qui prenaient un pointeur sur un objet de type CantorErreurs en dernier argument optionnel n'ont plus cet argument et lancent désormais une exception. Parmis ces fonctions, celles qui retournaient un entier pour indiquer un status d'erreur sont désormais de type void.

Le passage aux exceptions a conduit à éliminer la valeur entière de retour et le pointeur d'erreur optionnel de la méthode MoindreCarreLineaire::**erreurQuadratique**, il ne restait alors plus que le pointeur sur la valeur réelle à mettre à jour, ce qui correspondait à une signature un peu étrange. Pour cette méthode, il a donc été décidé de lui faire retourner la valeur réelle de l'erreur directement, sans passer par un argument.

Le passage aux exceptions a conduit à éliminer la valeur entière de retour et le pointeur d'erreur optionnel de la méthode FonctionApprochee::**erreurQuadratique**, il ne restait alors plus que le pointeur sur la valeur

réelle à mettre à jour, ce qui correspondait à une signature un peu étrange. Pour cette méthode, il a donc été décidé de lui faire retourner la valeur réelle de l'erreur directement, sans passer par un argument.

Un certain nombre de méthodes de la bibliothèque sont des prédicats. Ces méthodes retournaient jusqu'à présent un entier, ce qui était supporté par tous les compilateurs. Depuis quelques versions, la bibliothèque sous-jacente club nécessite l'utilisation d'un compilateur respectant le standard ANSI, lequel spécifie le type bool, qui représente de façon plus explicite ces résultats. L'opportunité des changements de signatures liés aux exceptions a donc été saisie pour changer également les signatures des prédicats. Ce changement ne devrait pas trop affecter les utilisateurs grâce aux conversions implicites du langage.

Les signatures des fonctions de l'interface C de la bibliothèque (qui est limitée aux rotations) pouvait poser des problèmes de violation mémoire si l'appelant n'allouait pas une chaîne de caractères de taille suffisante pour contenir les messages d'erreurs. Désormais, et en conformité avec l'interface C de la bibliothèque associée marmottes, l'interface C prend en dernier argument après le pointeur sur le message d'erreur un entier indiquant la taille disponible pour écrire ce message. Si le message est plus long que la chaîne fournie par l'appelant, le message est tronqué.

10.9 évolutions entre la version 5.5 et la version 5.6

Cette version n'apporte que des modifications de configuration et de compilation conditionnelle permettant de compiler la bibliothèque avec les compilateurs SUN.

10.10 évolutions entre la version 5.4 et la version 5.5

Les seules modifications introduites dans la version 5.5 de la bibliothèque sont l'ajout du script destiné à faciliter la compilation d'applicatifs dépendant de CANTOR (cf 13.1, page 93) : `cantor-config` . La version anglaise du fichier de licence a également été ajoutée dans la distribution.

10.11 évolutions entre la version 5.3 et la version 5.4

Les modifications apportées dans la version 5.4 sont uniquement des modifications de configuration. Les fonctionnalités du code n'ont pas évolué.

La première modification de configuration concerne le test de la bibliothèque CLUB. Les nouvelles macros couvrent plus de cas et les dépendances entre bibliothèques sont mieux gérées (par exemple les dépendances entre CLUB et XERCES ou MADONA).

La seconde modification de configuration provient d'un problème rencontré sous solaris lors des tests de MARMOTTES. Les exceptions générées dans CLUB ne sont pas récupérées par CLUB lorsque toutes les bibliothèques sont partagées. Le problème ne se pose pas avec des bibliothèques statiques. Dans l'attente d'une meilleure compréhension du phénomène, la configuration par défaut sous solaris consiste donc à ne construire que des bibliothèques statiques. L'utilisateur aventureux peut toujours construire des bibliothèques partagées en utilisant l'option `--enable-shared` du script de configuration.

10.12 évolutions entre la version 5.2.2 et la version 5.3

La seule évolution de la version 5.3 est l'utilisation de la classe string de la STL au lieu de ChaineSimple de CLUB. Cette évolution ne concerne que la classe CantorErreurs dont la méthode formate change de signature. Cette méthode étant déclarée protégée aucun impact sur l'interface publique de CANTOR n'est à mentionner.

10.13 évolutions entre la version 5.2.1 et la version 5.2.2

La seule évolution de la version 5.2.2 est la correction du traitement des cas dégénérés de partage d'arcs autour d'un point situé à proximité d'une des extrémités. Dans ces cas là, un des sous-arcs résultants est de taille nulle. Cet arc est désormais le premier si le point est proche du début et le second si ce n'est pas le cas. Dans les versions précédentes de la bibliothèque, le premier arc était systématiquement complet et seul le second arc pouvait être dégénéré. Le changement de comportement est lié à des modifications dans la bibliothèque Marmottes.

10.14 évolutions entre la version 5.2 et la version 5.2.1

Aucune évolution de code n'a eu lieu à l'occasion du passage en version 5.2.1. La distribution a simplement été mise en conformité avec les exigences d'une diffusion publique.

11 évolutions possibles

Il serait souhaitable d'augmenter la couverture de tests de la bibliothèque CANTOR.

Il serait utile de disposer dans la bibliothèque d'un support pour des notions liées au pavage de la sphère unité. On pense en particulier à l'extension de la triangulation de DELAUNAY à la sphère unité et à sa notion duale le pavage de VORONOI ainsi qu'aux interpolations sur un échantillon de points.

12 description des classes

12.1 classe AnnotatedArc

description

Cette classe est une extension de la class Arc utilisée lors de la combinaison de champs (classe Field, cf 12.8) par réunion ou intersection.

La combinaison des frontières de deux champs se fait en redécoupant tous les arcs frontières aux niveau des points d'intersection entre frontières, puis en marquant différemment les arcs qui font partie de l'intersection et ceux qui font partie de la réunion. La détermination de la marque à apposer sur l'arc provenant d'un champ en fonction de sa position par rapport à l'autre champ représente un volume de calcul non négligeable et est parfois difficile à calculer précisément. C'est typiquement le cas lorsque les deux champs ont une frontière commune, tous les points de certains arcs sont alors rigoureusement à cheval sur la frontière de l'autre. On peut éviter une partie de ces calculs en préservant l'information topologique reliant les arcs les uns aux autres.

La classe AnnotatedArc est destinée à mémoriser la marque apposée sur les arcs. Avec l'aide de la classe Node (cd 12.12), elle réalise la propagation de ces marques selon la topologie de l'entrelacs.

interface publique

```
#include "cantor/AnnotatedArc.h"
```

Les types publics sont décrits sommairement dans la table 2.

TAB. 2: types publics de la classe AnnotatedArc

nom	type	description
Annotation	enum	énumération des marques possibles, les valeurs autorisées sont : `notAnnotated`, `unionAnnotated` et `intersectionAnnotated`

TAB. 3: AnnotatedArc : méthodes publiques

signature	description
AnnotatedArc ()	crée un arc par défaut
AnnotatedArc (const Arc& *a*, const void **origin*)	crée un arc à partir de l'arc argument et de son champ de provenance
AnnotatedArc (const AnnotatedArc& *a*)	constructeur par copie
AnnotatedArc& **operator** = (const AnnotatedArc& *a*)	affectation
AnnotatedArc ()	destructeur de la classe.
bool **confines** (const AnnotatedArc& *a*, VecDBL **ptrEscape*) const	teste si l'argument est confiné dans un couloir autour de l'instance, s'il ne l'est pas, met à jour la variable pointée par *ptrEscape* avec le point où l'arc s'échappe
bool **parallelConnections** (const AnnotatedArc& *a*, Node **n*) const	teste si l'argument et l'instance convergent parallèlement vers le nœud spécifié
void **replace** (AnnotatedArc **ptrNew*) **throw**(CantorErreurs)	remplace l'instance par un arc équivalent
void **replace** (AnnotatedArc **ptrUp*, AnnotatedArc **ptrDown*) **throw**(CantorErreurs)	remplace l'instance par les deux arcs équivalents résultant de sa coupure au niveau d'un point intermédiaire
bool **removeIfPossible** ()	élimine l'arc de la topologie si c'est possible (c'est à dire si l'arc est de longueur nulle et que les nœuds qui le limitent peuvent être reconnectés
void **connect** (AnnotatedArc **ptrDown*) **throw**(CantorErreurs)	connecte l'instance avec l'arc spécifié en argument (qui doit démarrer au niveau où l'instance s'arrête)
void **connect** (AnnotatedArc **ptrDown1*, AnnotatedArc **ptrUp2*, AnnotatedArc **ptrDown2*) **throw**(CantorErreurs)	connecte l'instance avec les arcs spécifiés en argument (qui doivent selon les cas démarrer ou s'arrêter au niveau où l'instance s'arrête)
bool **isConnected** (AnnotatedArc **ptrA*) const	teste si l'instance est connectée à l'arc spécifié
const void * **origin** () const	retourne le champ de provenance de l'arc
Node * **upstreamNode** ()	retourne le nœud amont
const Node * **upstreamNode** () const	retourne le nœud constant
Node * **downstreamNode** ()	retourne le nœud aval
const Node * **downstreamNode** () const	retourne le nœud constant aval
Node * **oppositeNode** (const Node **n*) const	retourne le nœud opposé au nœud spécifié
static Annotation **oppositeAnnotation** (Annotation *a*)	retourne la marque opposée à la marque spécifiée
Annotation **annotation** () const	retourne la marque courante de l'instance
void **annotate** (Annotation *a*)	appose une marque sur l'instance
void **propagate** ()	propage le marquage aussi loin que possible à partir de l'instance
void **propagateThroughOppositeNode** (const Node **n*)	propage le marquage aussi loin que possible à partir de l'instance, mais uniquement dans la direction opposée au nœud spécifié

à suivre ...

TAB. 3: AnnotatedArc : méthodes publiques (suite)

signature	description
AnnotatedArc ***ptrUpstreamArc** () const	retourne un pointeur sur l'arc amont
AnnotatedArc ***ptrDownstreamArc** () const	retourne un pointeur sur l'arc aval

exemple d'utilisation

```
#include "marmottes/AnnotatedArc.h"

void
Braid::identifyArcs (const Field *ptrF1, const Field *ptrF2)
{
  for (iter i = arcs_.begin (); i != arcs_.end (); ++i)
    if (i->annotation () == AnnotatedArc::notAnnotated)
    { // look for the arc inclusion according to one test point at the middle
      bool inter, degenerated;
      if (i->origin () != ptrF1)
        inter = ptrF1->isInside (TestPoint (i), &degenerated);
      else
        inter = ptrF2->isInside (TestPoint (i), &degenerated);

      if (! degenerated)
      {
        i->annotate (inter
                     ? AnnotatedArc::intersectionAnnotated
                     : AnnotatedArc::unionAnnotated);
        i->propagate ();
      }
    }
}
```

conseils d'utilisation spécifiques

La classe AnnotatedArc est utilisée par la classe Braid qui gère globalement le découpage des arcs de plusieurs frontières et le marquage.

La classe a été conçue uniquement pour assurer les services nécessaires à la classe Braid pour déterminer la frontière d'une combinaison de deux champs. Les services qu'elle implémente sont donc très spécifiques, pour une utilisation plus générique, il est plutôt conseiller d'utiliser la classe de base Arc de la bibliothèque CANTOR.

implantation

Les attributs privés sont décrits sommairement dans la table 4, il n'y a pas d'attribut protégé.

TAB. 4: attributs privés de la classe AnnotatedArc

nom	type	description
upstreamNode_	TamponPartage	nœud amont (partagé avec d'autres arcs)
downstreamNode_	TamponPartage	nœud aval (partagé avec d'autres arcs)
annotation_	Annotation	marque courante
origin_	const void *	pointeur vers le champ d'origine de l'arc ce pointeur n'est utilisé que pour reconnaître si deux arcs ont la même origine ou non, il est parfois entièrement fabriqué et ne correspond pas systématiquement à un objet de type Field, c'est la raison pour laquelle on utilise le type const void *

12.2 classe Arc

description

Cette classe modélise des arcs de cercles (arc de petit ou de grand cercle) sur la sphère unité dans un espace vectoriel de dimension 3. Ces arcs sont définis par l'intersection de cônes issus du centre de la sphère avec la sphère, les arcs sont des extraits de tels cônes.

interface publique

```
#include "cantor/Arc.h"
```

TAB. 5: Arc : méthodes publiques

signature	description
Arc ()	construit un arc par défaut
Arc (const VecDBL& *axe*, double *angle* = 0.5 * `M_PI`) **throw**(CantorErreurs)	construit un arc correspondant à la totalité du cône d' *axe* et de demi-angle d'ouverture donné
Arc (const Cone& *c*) **throw**(CantorErreurs)	construit un arc correspondant à la totalité du cône *c*
Arc (const VecDBL& *debut*, const VecDBL& *fin*) **throw**(CantorErreurs)	construit l'arc de grand cercle joignant les deux points par le chemin le plus court
Arc (const VecDBL& *axe*, const VecDBL& *debut*, const VecDBL& *fin*) **throw**(CantorErreurs)	construit un arc de grand cercle à partir de son *axe* et de deux points définissant les plans méridiens limites (l'arc ne passera rigoureusement par ces points que s'ils sont orthogonaux à l'axe)
Arc (const VecDBL& *axe*, double *angle*, const VecDBL& *debut*, const VecDBL& *fin*) **throw**(CantorErreurs)	construit un arc de petit cercle à partir de son *axe*, de son demi-angle d'ouverture et de deux points définissant les plans méridiens limites (l'arc ne passera rigoureusement par ces points que s'ils sont à la bonne distance de l'axe)
à suivre ...	

TAB. 5: Arc : méthodes publiques (suite)

signature	description
Arc (const VecDBL& *axe*, double *angle*, const VecDBL& *debut*, double *balayage*) **throw**(CantorErreurs)	construit un arc de petit cercle à partir de son *axe*, de son demi-angle d'ouverture, d'un point définissant le plan méridiens de départ et du *balayage* angulaire (l'arc ne passera rigoureusement par le point début que s'il est à la bonne distance de l'axe)
Arc (const Arc& *a*)	constructeur par copie
Arc& **operator** = (const Arc& *a*)	affectation
Arc ()	destructeur.
const VecDBL& **axe** () const	retourne une référence constante sur l'axe de l'arc
const VecDBL& **debut** () const	retourne une référence constante sur le point initial de l'arc
const VecDBL& **fin** () const	retourne une référence constante sur le point final de l'arc
VecDBL **intermediaire** (double *alpha*) const	retourne le point intermédiaire d'azimut α de l'arc (ce point ne fait rigoureusement partie de l'arc que si α est compris entre 0 et **balayage** ())
double **cosinus** () const	retourne le cosinus de **angle** ()
double **sinus** () const	retourne le sinus de **angle** ()
double **angle** () const	retourne le demi-angle d'ouverture du cône dont l'arc est un extrait
const VecDBL& **u** () const	retourne une référence constante sur le premier vecteur du repère orthogonal non normé permettant de décrire l'arc
const VecDBL& **v** () const	retourne une référence constante sur le deuxième vecteur du repère orthogonal non normé permettant de décrire l'arc
const VecDBL& **w** () const	retourne une référence constante sur le troisième vecteur du repère orthogonal non normé permettant de décrire l'arc
double **balayage** () const	retourne l'étendue de l'arc comptée sous forme de l'angle autour de l'axe du cône
double **longueur** () const	retourne l'étendue de l'arc comptée sous forme d'intégrale curviligne ; la valeur retournée est donc égale à **balayage** () × **sinus** ()
VecDBL **proche** (const VecDBL& *p*) const	retourne le point de l'arc le plus proche de $\vec{p}$
double **distance** (const VecDBL& *p*) const	retourne la distance angulaire entre $\vec{p}$ et le point de l'arc le plus proche (cette distance est donc nulle si $\vec{p}$ appartient à l'arc)
bool **diedreContient** (const VecDBL& *p*) const	indique si $\vec{p}$ est inclus dans le dièdre délimité par les plans constitués par l'axe du cône et les points extrèmes de l'arc
void **partage** (const VecDBL& *p*, Arc **ptrAv*, Arc **ptrAp*) const	découpe l'arc en deux sous-arcs de part et d'autre du plan défini par l'axe et le vecteur $\vec{p}$ et met les résultats dans les variables pointées par *ptrAv* et *ptrAp* en respectant le sens de balayage de l'arc initial. Si $\vec{p}$ n'est pas dans le dièdre de l'arc initial, l'un des arcs créés aura un **balayage** () nul. Il est possible d'utiliser un pointeur sur l'arc courant pour *ptrAv* ou pour *ptrAp*

à suivre ...

TAB. 5: Arc : méthodes publiques (suite)

signature	description
void **intersection** (const VecDBL& *a*, double *cosinus*, int **ptrNbInt*, VecDBL **ptrV1*, VecDBL **ptrV2*) const	calcule les points d'intersection de l'arc courant et de l'arc défini par *a* et *cosinus*. Le nombre de points d'intersection (entre 0 et 2) est retourné dans la variable pointée par *ptrNbInt*, les points étant retournés dans les variables pointées par *ptrV1* (s'il y a au moins un point) et *ptrV2* (s'il y a au moins deux points). S'il y a deux points d'intersection, * *ptrV1* et **ptrV2* respectent le sens de balayage de l'arc initial.
void **intersection** (const Cone& *c*, int **ptrNbInt*, VecDBL **ptrV1*, VecDBL **ptrV2*) const	méthode identique à la précedente pour un arc correspondant à la totalité du cône *c*
void **intersection** (const Arc& *a*, int **ptrNbInt*, VecDBL **ptrV1*, VecDBL **ptrV2*) const	méthode identique à la précedente pour l'arc *a*
bool **recouvre** (const Arc& *a*, double *epsilon*) const	indique si tous les points de *a* sont à moins de ε de l'instance (c'est à dire si l'instance recouvre complètement *a*)
bool **balaye** (const VecDBL& *point*, const VecDBL& *axe*, double *balayage*)	vérifie l'arc recouvre le point spécifié lors d'un balayage
void **appliqueRotation** (const RotDBL& *r*)	transforme l'arc courant par la rotation *r*
Arc **operator -** () const	retourne un arc contenant les mêmes points que l'instance, mais parcouru en sens inverse

exemple d'utilisation

```
#include "cantor/Arc.h"
...

// constitution d'un triangle sphérique
Arc cote1 (pointA, pointB);
Arc cote2 (pointB, pointC);
Arc cote3 (pointC, pointA);

cout << "périmètre : "
     << (cote1.longueur () + cote2.longueur () + cote3.longueur ())
     << endl;

Arc moitieAv, moitieAp;
cote1.partage (cote1.intermediaire (0.5 * cote1.balayage ()),
               &motieAv, &motieAp);

...
```

conseils d'utilisation spécifiques

Les calculs sur les arcs ne sont réalisables que dans certaines plages (les cônes trop fins par exemples ne permettent pas des calculs précis). Il est donc nécessaire de gérer certains seuils. Ceci est également vrai pour les calculs sur les champs (voir la classe Champ, [DR2]). La classe Champ s'appuyant sur la classe Arc, il est nécessaire que les ε de Champ soient au moins légèrement supérieurs à ceux de Arc. Il a été décidé de prendre 4.85e-6 pour Champ, ce qui représente une seconde d'arc (ou son sinus, à ce niveau la distinction n'est pas perceptible) et de prendre 3.16e-6 pour Arc, ce qui représente la racine de 1.0e-11 (qui est utilisé dans des soustractions de nombres aux alentours de 1.0).

Les autres classes permettant de travailler sur la sphère unité sont ArcIterateur(cf 12.3), Champ (cf [DR2]), Cone (cf 12.6), et Secteurs (cf [DR2]).

implantation

Les attributs privés sont décrits sommairement dans la table 6, il n'y a pas d'attribut protégé.

TAB. 6: attributs privés de la classe Arc

nom	type	description
axe_	VecDBL	axe du cône dont l'arc est un extrait
cos_	double	cosinus de angle_
sin_	double	sinus de angle_
angle_	double	demi-angle d'ouverture du cône dont l'arc est un extrait
deb_	VecDBL	début de l'arc sur le cône (l'arc est décrit en tournant dans le sens trigonométrique entre deb_ et fin_ autour de axe_).
fin_	VecDBL	fin de l'arc sur le cône (l'arc est décrit en tournant dans le sens trigonométrique entre deb_ et fin_ autour de axe_)
u_	VecDBL	vecteur non normé dans le plan méridien du début de l'arc
v_	VecDBL	vecteur non normé orthogonal à u_
w_	VecDBL	vecteur non normé colinéaire à l'axe du cône dont l'arc est un extrait
balayage_	double	amplitude de l'arc, comptée autour de l'axe du cône (entre 0 et 2π)

Les méthodes privées sont décrites dans la table 7.

TAB. 7: Arc : méthodes privées

signature	description
void **initVecteurs** (const VecDBL& *axe*, double *alpha*, const VecDBL& *debut*, const VecDBL& *fin*) **throw**(CantorErreurs)	initialisation des attributs de la classe, appelée par presque tous les constructeurs. Cette fonction calcule les fonctions trigonométriques, copie et normalise l'axe, copie normalise et recale les vecteurs limites de sorte qu'ils soient bien séparés de l'axe par l'angle donné en argument (on considère qu'au départ ils ne font que délimiter le plan dans lequel se trouve la limite réelle).

12.3 classe ArcIterateur

description

Cette classe permet de parcourir point par point un arc, la ligne polygonale obtenue en reliant ces points par des segments de droite étant assurée de ne pas s'écarter de l'arc courbe réel de plus de ε (en distance), tolérance fixée à la construction. La tolérance est considérée par rapport à des segments de droite dont seuls les points extrèmes sont sur la sphère unité, les points intermédiaires sont sous la surface de la sphère.

L'itérateur est séparé de la classe Arc afin de permettre d'avoir simultanément plusieurs itérateurs complètement indépendants parcourant le même arc.

Le principe adopté est qu'un itérateur est dans un état indéfini juste après la construction (ou la réinitialisation), et qu'il devient valide après le premier appel à **suivant** () ou à **operator ()** ().

interface publique

```
#include "cantor/ArcIterateur.h"
```

TAB. 8: ArcIterateur : méthodes publiques

signature	description
ArcIterateur (const Arc& *a*, double *tolerance* = 1.0e-4)	construit un itérateur sur l'arc *a* tel que la plus grosse erreur réalisée en approximant l'arc par la succession de segments de droites soit inférieure à la *tolerance* fixée, interprétée comme une norme euclidienne en dimension 3
ArcIterateur (const ArcIterateur& *i*)	constructeur par copie
ArcIterateur& **operator** = (const ArcIterateur& *i*)	affectation
ArcIterateur ()	destructeur
int **nbSegments** () const	retourne le nombre de segments de droites approximant l'arc
int **nbPoints** () const	retourne le nombre de points approximant l'arc
void **reinitialise** ()	remet l'itérateur en début de parcours. Après réinitialisation, l'itérateur est dans un état indéfini, il faut appeler une première fois **suivant** () ou **operator ()** () avant de pouvoir récupérer le premier point.
int **suivant** ()	avance l'itérateur d'un pas, et retourne un indicateur de validité
int **operator ()** ()	avance l'itérateur d'un pas, et retourne un indicateur de validité
VecDBL **point** () const	retourne une copie du point courant. Si l'itérateur est dans un état invalide (juste après construction ou réinitialisation), cette fonction retourne le premier point. Dans ce cas, après avoir mis l'itérateur dans un état valide (par appel à **suivant** () ou à **operator ()** ()), le premier appel suivant à **point** () redonnera une seconde fois ce premier point. Si l'itérateur est arrivé en fin de parcours, **point** () redonne toujours le dernier point, même si on s'obstine à appeler **suivant** () ou **operator ()** ()

exemple d'utilisation

```
#include "cantor/ArcIterateur.h"

...

Arc a (VecDBL (1.0, 1.0, 1.0),
       VecDBL (1.0, 0.0, 0.0), VecDBL (0.0, 1.0, 0.0));

ArcIterateur iter (a, 1.0e-6);

while (iter ())
  cout << iter.point ().x () << ' '
       << iter.point ().y () << ' '
       << iter.point ().z () << endl;

...
```

conseils d'utilisation spécifiques

Cette classe permet d'approximer des arcs par des lignes polygonales en dimension trois ; les points intérieurs de chaque segment n'appartiennent donc pas à la sphère unité, ils sont sous la surface. Il n'est donc pas conseillé d'interpoler dans les segments pour faire des calculs, à moins d'utiliser une tolérance très petite et donc un très grand nombre de points.

Si l'utilisateur désire faire des calculs sur des arcs, il lui est conseillé de passer plutôt par les méthodes de la classe Arc (distance, longueur, ...) ou par les méthodes des classes associées Champ (cf [DR2]), Cone (cf 12.6), et Secteurs (cf [DR2]. Cette classe réalise une approximation qui ne peut être adaptée à des calculs précis qu'au prix d'une tolérance très petite et donc d'une grande quantité de calculs pour générer un nombre important de points.

L'itérateur est basé sur un parcours par points de l'arc, les première et dernière itération donnant rigoureusement les points extrèmes de l'arc. Il faut conserver à l'esprit que si l'arc est approximé en n segments, alors il y aura $n+1$ points à récupérer.

Il faut noter que l'itérateur ne mémorise jamais de points, il les crée un par un à la demande, il n'y a donc pas de surcoût en mémoire à utiliser une tolérance petite (à moins que l'appelant ne les mémorise lui-même), il y a par contre un surcoût en temps de calcul.

implantation

Les attributs privés sont décrits sommairement dans la table 10, il n'y a pas d'attribut protégé.

TAB. 9: méthodes privées de la classe ArcIterateur

signature	description
ArcIterateur ()	constructeur par défaut

TAB. 10: attributs privés de la classe ArcIterateur

nom	type	description
arc_	const Arc*	pointeur sur l'arc à parcourir
pas_	double	pas angulaire de parcours
indice_	int	indice courant du parcours
segments_	int	nombre de segments approximant l'arc complet

12.4 classe Braid

description

Cette classe modélise un entrelacement d'arcs frontières sur la sphère unité. Elle est utilisée pour calculer la frontière résultant de la combinaison de deux champs par réunion ouintersection.

La combinaison des frontières de deux champs se fait en redécoupant tous les arcs frontières aux niveau des points d'intersection entre frontières, puis en marquant différemment les arcs qui font partie de l'intersection et ceux qui font partie de la réunion. Cette classe propose les services permettant de réaliser toutes ces opérations (découpage, marquage, sélection des arcs selon l'opération désirée).

interface publique

```
#include "cantor/Braid.h"
```

TAB. 11: Braid : méthodes publiques

signature	description
Braid ()	construit un entrelacs par défaut
Braid (const Arc& *a*, const VecDBL& *axis*,double *spreading*) **throw**(CantorErreurs)	construit un entrelacs par *étalement* d'un arc à l'aide d'une rotation définie par *axis* et *spreading*
Braid (const Braid& *b*)	constructeur par copie
Braid& **operator** = (const Braid& *b*)	affectation
Braid ()	destructeur
static void **intertwine** (Braid *ptrB1, Braid *ptrB2) **throw**(CantorErreurs)	entrelace deux instances
void **absorb** (Braid **ptrOther*)	absorbe tout le contenu d'un entrelacs dans l'instance
void **initAnnotations** ()	initialise toutes les marques de tous les arcs à la valeur AnnotatedArc::notAnnotated
void **identifyArcs** (const Field **ptrF1*, const Field **ptrF2*)	identifie et marque les arcs non ambigüs
bool **isCompletelyAnnotated** () const	teste si l'ensemble des arcs de l'entrelacs est marqué
	à suivre ...

TAB. 11: Braid : méthodes publiques (suite)

signature	description
void **simplify** () **throw**(CantorErreurs)	élimine les arcs et nœuds inutiles de l'entrelacs (arcs nuls)
bool **coveringHeuristic** (const Arc& *a*, const VecDBL& *axis*, double *spreading*) **throw**(CantorErreurs)	tente de marquer les arcs recouverts par l'étalement de l'arc origine
bool **closeParallelsHeuristic** () **throw**(CantorErreurs)	tente de marquer les arcs parallèles proches tente de marquer les arcs recouverts par l'étalement de l'arc origine
bool **splittingHeuristic** () **throw**(CantorErreurs)	tente de redécouper les arcs dont seule une zone est ambigüe de façon à permettre leur marquage
bool **equivalentPathsHeuristic** ()	tente de marquer les arcs parallèles qui se perturbent mutuellement
void **intersectionBoundary** (vector< vector<Arc> > **ptrB*) const	initialise la frontière pointée par l'argument avec les arcs représentant la frontière de l'intersection des champs initiaux
void **unionBoundary** (vector< vector<Arc> > **ptrB*) const	initialise la frontière pointée par l'argument avec les arcs représentant la frontière de la réunion des champs initiaux

exemple d'utilisation

```
#include "marmottes-utilisateur/Braid.h"

const Field&
Field::combine (const Field& c, bool intersection)
  throw (CantorErreurs)
{ // generic function to compute both reunion and intersection

  ...

  // create the two strands of a braid from the boundaries of both fields
  Braid b1 (this, boundary_.begin (), boundary_.end ());
  Braid b2 (&c, c.boundary_.begin (), c.boundary_.end ());
  Braid::intertwine (&b1, &b2);

  // gather the strands in a uniq list
  b1.absorb (&b2);
  b1.simplify ();

  // at the beginning, no arc is annotated
  b1.initAnnotations ();

  // annotation loop
  for (bool modification = true; modification;)
  {
```

```
    modification = false;

    // annotate unambiguous arcs
    b1.identifyArcs (this, &c);

    if (! b1.isCompletelyAnnotated ())
    {
      // there are some ambiguous arcs left
      // we apply several heuristics to remove the ambiguities

      // heuristic 1:
      // splitting arcs where only one part is ambiguous
      modification = b1.splittingHeuristic ();

      // heuristic 2:
      // detect parallel paths
      if (!modification)
        modification = b1.equivalentPathsHeuristic ();

      // more heuristics can be added here

      if (! modification)
        throw CantorErreurs (CantorErreurs::heuristic_failure);

    }

  }

  // select the arcs according to the desired operation (intersection vs union)
  Boundary b;
  if (intersection)
    b1.intersectionBoundary (&b);
  else
    b1.unionBoundary        (&b);

  ...

}
```

conseils d'utilisation spécifiques

La classe Braid est utilisée par la classe Field pour calculer la frontière résultant de la combinaison de deux champs par réunion ou intersection. Elle n'a pas d'autre utilisation.

implantation

Les attributs privés sont décrits sommairement dans la table 12, il n'y a pas d'attribut protégé.

TAB. 12: attributs privés de la classe Braid

nom	type	description
arcs_	list<AnnotatedArc>	liste de stockage des arcs

12.5 classe CantorErreurs

description

Cette classe permet de formater et traduire dans la langue de l'utilisateur des messages d'erreur liés à la bibliothèque CANTOR. Elle utilise les mécanismes qui lui sont fournis par sa classe de base.

interface publique

```
#include "cantor/CantorErreurs.h"
```

Les opérations publiques sont essentiellements celles de la classe de base `BaseErreurs`, qui appartient à la bibliothèque CLUB (voir [DR3]). Les méthodes qui ne peuvent être héritées (les constructeurs et les méthodes de classe) ont été redéfinies avec des sémantiques équivalentes. Dans ces méthodes redéfinies, les codes d'erreurs (déclarés comme type énuméré public interne) attendent les arguments suivants dans la liste des arguments variables :

norme_nulle : néant ;

axe_rotation_nul : néant ;

ordre_inconnu : `int` (pour le code de l'ordre), liste de `char *` terminée par un pointeur nul pour les noms des ordres connus ;

matrice_non_orthogonalisable : néant ;

matrice_singuliere : néant ;

repere_indirect : néant ;

alignement_axe_extremite : néant ;

cones_coaxiaux : néant ;

cones_disjoints : néant ;

creneau_vide : néant ;

indice_hors_bornes : `char *` (pour le nom de l'indice), `int` (pour la valeur courante de l'indice), `int` (pour la valeur minimale autorisée), `int` (pour la valeur maximale autorisée) ;

polynome_invalide : néant ;

aucune_fonction_base : néant ;

erreur_fonction_base : `int` (pour le code d'erreur de la fonction utilisateur) ;

ajuste_dimension_nulle : néant ;

non_ajustable : néant ;

non_ajustee : néant ;

echec_heuristique : néant ;

frontiere_ouverte : néant ;

desequilibre_connexions : néant ;

erreur_interne : `int` (pour le numéro de ligne, utiliser la macro `__LINE__`), `char *` (pour le nom du fichier source, utiliser la macro `__FILE__`) ;

singularite_angle : néant.

exemples d'utilisation

```
#include "cantor/CantorErreurs.h"
#include "cantor/MoindreCarreLineaire.h"

static double x [] = { 1.0,  2.0,  3.0,  4.0, 5.0,   6.0,   7.0,  8.0  };
static double y [] = { 0.97, 3.01, 2.93, 6.9, 8.99, 11.03, 13.0, 14.98 };

CantorErreur ce;

// ajustement d'une droite sur l'échantillon
// on a phi (k, 1) = 1 et phi (k, 2) = x (i)
double fki [2];
fki [0] = 1;
MoindreCarreLineaire initial (2);
for (int i = 0; i < sizeof (x) / sizeof (double); i++)
{ fki [1] = x [i];
  initial.ajouteResidu (y [i], fki);
}

if (initial.ajuste (&ce))
  return ce.code ();
```

conseils d'utilisation spécifiques

Cette classe est principalement utilisée pour tester la bonne exécution des fonctions de la bibliothèque CANTOR elle-même. Son utilisation se résume donc à tester correctement la présence ou l'absence d'erreurs (méthode `existe ()`), et à décider du comportement à adopter en présence d'une erreur.

Si la même instance d'erreur est utilisée pour tester le retour de plusieurs fonctions, il faut prendre garde de la tester au bon moment ; il est en effet possible qu'une erreur soit générée par le premier appel, qu'elle soit ignorée par l'appelant, qu'une seconde fonction de CANTOR se termine ensuite normalement et que l'appelant ne détecte la première erreur qu'à cet instant.

implantation

La classe dérive publiquement de `BaseErreurs`, elle ne possède aucun attribut propre.

12.6 classe Cone

description

Cette classe décrit l'intersection d'un cône et de la sphère unité dans un espace vectoriel de dimension 3.

interface publique

```
#include "cantor/Cone.h"
```

TAB. 13: Cone : méthodes publiques

signature	description
Cone ()	construit un cône par défaut (d'axe aligné avec l'axe des abscisses et de demi-angle d'ouverture nul)
Cone (const VecDBL& *a*, double *mu*) **throw**(CantorErreurs)	construit le cône d'axe $\vec{a}$ et de demi-angle d'ouverture μ (l'angle est ramené entre 0 et π au besoin). Une exception est lancée si l'axe est de norme quasi-nulle
Cone (const VecDBL& *p1*, const VecDBL& *p2*, const VecDBL& *p3*) **throw**(CantorErreurs)	construit le cône passant par les points $\vec{p_1}$, $\vec{p_2}$ $\vec{p_3}$. Une exception est lancée si les points ne sont pas distincts
Cone (const Cone& *c*)	constructeur par copie
Cone& **operator** = (const Cone& *c*)	affectation
Cone ()	destructeur
Cone **operator** - () const	retourne le cône complémentaire de l'instance par rapport à la sphère unité
const VecDBL& **axe** () const	retourne une référence constante sur l'axe du cône
double **angle** () const	retourne le demi-angle d'ouverture du cône
double **cosinus** () const	retourne le cosinus du demi-angle d'ouverture du cône
double **sinus** () const	retourne le sinus du demi-angle d'ouverture du cône
void **corrige** (double *ecart*)	corrige le demi-angle ouverture du cône de l'*ecart* (le demi-angle est ramené entre 0 et π après correction)
bool **inclus** (const VecDBL &*u*) const	indique si le vecteur $\vec{u}$ est inclus dans le cône
void **intersection** (const Cone &*v*, VecDBL* *p_deb*, VecDBL* *p_fin*) const **throw**(CantorErreurs)	initialise les variables pointées par *p_deb* et *p_fin* aux vecteurs définissant l'intersection de l'instance et du cône *v*. La portion de l'instance comprise dans *v* va de *p_deb* à *p_fin* en tournant positivement autour de l'axe du cône

exemple d'utilisation

```
#include "cantor/Cone.h"
#include "cantor/util.h"

...
```

```
Cone c1 (VecDBL (1.0, 0.0, 0.0), radians (60.0));
Cone c2 (VecDBL (0, 1, 0), radians (90.0));
VecDBL deb, fin;
c1.intersection (c2, &deb, &fin);
cout << "deb = " << deb << endl;
cout << "fin = " << fin << endl;
```

conseils d'utilisation spécifiques

Les cônes sont les éléments de base pour les calculs sur la sphère unité. Les triangles sphériques ne peuvent pas être considérés comme des éléments de base, car ils peuvent être construits par des grands cercles de la sphère qui ne sont que des cas particuliers de cônes[7].

Il faut prendre garde que l'angle considéré dans les cônes est toujours le demi-angle d'ouverture, c'est à dire qu'un cône qui définit un grand cercle sur la sphère a un demi-angle d'ouverture de $\pi/2$, et qu'un cône qui recouvre la totalité de la sphère a un demi-angle d'ouverture de π.

Les autres classes permettant de travailler sur le sphère unité sont Arc (cf 12.2), ArcIterateur(cf 12.3), Champ (cf [DR2]), et Secteurs (cf [DR2]).

implantation

Les attributs privés sont décrits sommairement dans la table 14, il n'y a pas d'attribut protégé.

TAB. 14: attributs privés de la classe Cone

nom	type	description
a_	VecDBL	axe du cône
cosMu_	double	cosinus du demi-angle d'ouverture du cône
sinMu_	double	sinus du demi-angle d'ouverture du cône
mu_	double	demi-angle d'ouverture du cône

12.7 classe Creneau

description

La classe Creneau implante la notion d'ensemble d'intervalles disjoints ordonnés. Elle généralise ainsi la notion d'intervalle en proposant des services similaires mais en autorisant les trous.

interface publique

```
#include "cantor/Creneau.h"
```

[7] ce sont des cônes de demi-angle d'ouverture $\pi/2$

TAB. 15: Creneau : méthodes publiques

signature	description
Creneau()	construit un créneau vide
Creneau(const Intervalle& *i*)	construit un créneau connexe similaire à l'intervalle *i*
Creneau (const Intervalle& *i1*, const Intervalle& *i2*)	construit un créneau contenant les intervalles *i1* et *i2*
Creneau(double *a*, double *b*)	construit le créneau connexe similaire à l'intervalle de bornes *a* et *b* (l'ordre des bornes est indifférent)
Creneau (const Creneau& *c*)	constructeur par copie
Creneau& **operator** = (const Creneau& *c*)	affectation
~**Creneau** ()	destructeur, libère la mémoire allouée
void **nettoie** ()	enlève tous les intervalles contenus dans l'instance
double **inf** () const **throw**(CantorErreurs)	retourne une copie de la borne inférieure du plus petit intervalle (l'erreur est activée si le créneau est vide)
double **sup** () const **throw**(CantorErreurs)	retourne une copie de la borne supérieure du plus grand intervalle (l'erreur est activée si le créneau est vide)
int **nombre** () const	retourne une copie du nombre d'intervalles contenus dans l'instance
const Intervalle& **operator []** (int *i*) const **throw**(CantorErreurs)	retourne une copie de l'intervalle d'indice *i* de l'instance. Une erreur est générée si l' indice n'est pas compris entre 0 et *nombre* () - 1)
bool **connexe** () const	indique si l'instance est connexe (c'est à dire si elle contient exactement un intervalle)
bool **vide** () const	teste si l'instance est vide (la vacuité est considérée avec une certaine marge, et correspond au fait que les intervalles contenus dans l'instance aient tous une longueur inférieure à la constante `cantorEpsilon` définie dans `"cantor/Util.h"`
bool **contient** (double *x*) const	indique si l'instance contient le réel *x*
bool **contient** (const Intervalle& *i*) const	indique si l'instance contient les points de l'intervalle *i*
bool **rencontre** (const Intervalle& *i*) const	indique si l'instance et l'intervalle *i* ont au moins un point commun
bool **disjoint** (const Intervalle& *i*) const	indique si aucun point de l'intervalle *i* n'est dans l'instance
void **decale** (double *delta*) const	décale tous les intervalles de la valeur *delta*
Creneau& **operator** += (const Intervalle& *i*)	remplace l'instance par le résultat de sa réunion avec l'intervalle *i*
Creneau& **operator** -= (const Intervalle& *i*)	remplace l'instance par le résultat de sa disjonction avec l'intervalle *i*
Creneau& **operator** *= (const Intervalle& *i*)	remplace l'instance par le résultat de son intersection avec l'intervalle *i*
Creneau& **operator** += (const Creneau& *c*)	remplace l'instance par le résultat de sa réunion avec le créneau *c*
Creneau& **operator** -= (const Creneau& *c*)	remplace l'instance par le résultat de sa disjonction avec le créneau *c*

TAB. 16: Creneau : opérations non membre

signature	description
Creneau **operator** + (const Creneau& *c*, const Intervalle& *i*)	construit le créneau réunion du créneau *c* et de l'intervalle *i*
Creneau **operator** - (const Creneau& *c*, const Intervalle& *i*)	construit le créneau disjonction du créneau *c* et de l'intervalle *i*
Creneau **operator** + (const Creneau& *c*, const Creneau& *c2*)	construit le créneau réunion des créneaux *c* et *c2*
Creneau **operator** - (const Creneau& *c*, const Creneau& *c2*)	construit le créneau disjonction des créneaux *c* et *c2*
Creneau **operator** * (const Creneau& *c*, const Intervalle& *i*)	construit le créneau intersection du créneau *c* et de l'intervalle *i*
Creneau **operator** * (const Creneau& *c*, const Creneau& *c2*)	construit le créneau intersection des créneaux *c* et *c2*
inline ostream& **operator** << (ostream& *s*, const Creneau& *c*)	formate une chaîne de caractères représentant le créneau *c* sur le flot *s*

exemple d'utilisation

```
#include "cantor/creneau.h"
...
Creneau c1 (-2.0, -1.0);
        c1 += Intervalle (-0.95, -0.85);
Intervalle i (-0.9, -0.8);
Creneau c2 (i);
        c2 += Intervalle (10.0, 12.0);
        c2 *= Intervalle (-11.0, 11.0);

cout << c1
     << (c1.connexe () ?  " ":  " non ") <<  "connexe, "
     << (c1.vide    () ?  " ":  " non ") <<  "vide "
     << '(' << c1.nombre () << " intervalle"
     << ((c1.nombre () > 1) ? "s" : "") << ")\n";

cout << c1
     << (c1.contient (-1.3) ? " contient " :  " ne contient pas ")
     << -1.3 << endl;

for (int i = 0; i < c2.nombre (); i++)
{ cout << c1
       << (c1.rencontre (c2 [i]) ? " rencontre "
                                 : " ne rencontre pas ")
       << c2 [i] << endl;
}
```

conseils d'utilisation spécifiques

En règle générale en C++, les opérateurs conduisant à une modification de l'instance (+=, *=, ...) sont plus efficaces que leurs homologues sans affectation car ils évitent la création d'une variable temporaire, sa recopie, puis sa destruction. La classe Creneau présente une exception : l'opérateur *= lorsque l'argument est un autre créneau. Cette opération étant très difficile à implanter en place, l'opérateur se contente d'appeler l'opérateur * et le compilateur génère une variable temporaire et tout ce qui s'ensuit. Cette opération n'apporte donc pas de réelle économie par rapport aux opérateurs infixés, elle n'a été rajoutée que pour des raisons de symétrie de la classe.

implantation

Les attributs privés sont décrits sommairement dans la table 17, il n'y a pas d'attribut protégé.

TAB. 17: attributs privés de la classe Creneau

nom	type	description
tailleTable_	int	taille de la table des intervalles
nbIntervalles_	int	nombre d'intervalles utilisés dans la table
table_	Intervalle *	pointeur sur la table des intervalles allouée

Les méthodes privées sont décrites dans la table 18.

TAB. 18: Creneau : méthodes privées

signature	description
void **etendTable** ()	réalloue la table des intervalles

12.8 classe Field

description

Cette classe implante la notion de champ sur la sphère unité, par exemple pour définir des champs de vue de senseurs optiques. Ces champs peuvent être définis de façon directe dans des cas simples (lorsqu'ils sont limités à des cônes), ou par étapes successives en modifiant ou combinant des champs plus simples à l'aide d'opérateurs ensemblistes : réunion, intersection, inversion mais aussi différence, déplacement, étalement, dilatation ou contraction.

Les champs modélisés ne sont limités ni à des zones convexes ni même à des zones connexes. Ils peuvent contenir des trous et être constitués de plusieurs morceaux indépendants dont la frontière peut être relativement complexe.

interface publique

```
#include "cantor/Field.h"
```

Les types publics sont décrits sommairement dans la table 19.

TAB. 19: types publics (typedef) de la classe Field

nom	type	description
Loop	vector<Arc>	frontière d'une zone connexe
Boundary	vector<Loop>	frontière globale, pouvant contenir plusieurs zones connexes
TypeFuncConstField	void f (const Field&, void*)	fonction applicable à un champ constant
TypeFuncField	void f (Field&, void*)	fonction applicable à un champ

TAB. 20: Field : méthodes publiques

signature	description
Field ()	construit un champ par défaut, vide
Field (const Cone& *c*)	construit un champ simple constitué par le cône *c*
Field (const Arc& *a*, const VecDBL& *axis*, double *spreading*) **throw**(CantorErreurs)	construit un champ résultant du balayage à l'aide de la rotation d'axe *axis* et de d'amplitude *spreading* de l'arc *a* sur la sphère unité
Field (const Field& *f*)	constructeur par copie
Field& **operator** = (const Field& *f*)	affectation
Field ()	destructeur
bool **isEmpty**() const	indique si l'instance est vide
bool **isFull**() const	indique si l'instance recouvre la totalité de la sphère
bool **isInside** (const VecDBL& *point*, bool **ptrDegenerated* = 0) const	indique si le *point* est inclus dans l'instance, si *ptrDegenerated* est non nul une indication de calcul est fiable ou dégénéré est écrite dans la variable pointée
double **offsetFromBoundary** (const VecDBL& *point*) const **throw**(CantorErreurs)	calcule l'écart angulaire signé entre le point et la frontière (positif si le point est à l'intérieur du champ, négatif sinon)
Secteurs **selectInside** (const Secteurs& *s*) const **throw**(CantorErreurs)	retourne les parties du secteur *s* visibles à travers le champ courant
Secteurs **selectInside** (const Cone& *c*) const **throw**(CantorErreurs)	retourne les parties du cône *c* visibles à travers le champ courant
Field **operator -** () const	retourne le champ opposé à l'instance, c'est à dire dont l'intérieur et l'extérieur sont permutés
const Field& **operator *=** (const Field& *f*) **throw**(CantorErreurs)	remplace l'instance par son intersection avec le champ *f*
const Field& **operator** += (const Field& *f*) **throw**(CantorErreurs)	remplace l'instance par sa réunion avec le champ *f*
const Field& **operator** -= (const Field& *f*) **throw**(CantorErreurs)	remplace l'instance par sa différence avec le champ *f*
void **rotate** (const RotDBL& *r*)	remplace l'instance par son image à travers la rotation *r*
	à suivre ...

TAB. 20: Field : méthodes publiques (suite)

signature	description
void **spread** (const VecDBL& *axis*, double *spreading*) **throw**(CantorErreurs)	remplace l'instance par son balayage à l'aide de la rotation d'axe *axis* et d'amplitude *spreading*
void **applyMargin** (double *m*) **throw**(CantorErreurs)	applique la marge angulaire *m* à l'instance (ceci étend l'instance si *m* est positif et la réduit si *m* est négatif)
void **initWalk** (double *tolerance* = 1.0e-4)	initialise un itérateur interne permettant de parcourir la frontière de l'instance
bool **nextPoint** (VecDBL **ptrPoint*, bool **ptrLast*)	calcule le point suivant lors du parcours de la frontière et retourne un indicateur de fin de parcours. Si la frontière est formée de plusieurs courbes fermées disjointes, la variable pointée par *ptrLast* est initialisée à une valeur non nulle à chaque fois qu'un point est le dernier d'une courbe (ceci permet à un logiciel de tracé de « lever le crayon » entre les courbes)
void **stopWalk** ()	arrête l'itérateur interne sur la frontière

exemple d'utilisation

```
#include "marmottes/Field.h"

...

  VecDBL v1 (radians (180.0), radians (25.0));
  Cone planLimitationScan1 (v1, radians (90.0));
  VecDBL v2 (radians (0.0),   radians (25.0));
  Cone planLimitationScan2 (v2, radians (90.0));
  Field diedreOuvertureScan    (planLimitationScan1);
  diedreOuvertureScan *= Field (planLimitationScan2);

  Field scanSud (Cone (VecDBL (0.0, -1.0, 0.0), radians (96.491)));
  scanSud  *= Field (Cone (VecDBL (0.0,  1.0, 0.0), radians (84.159)));
  scanSud  *= diedreOuvertureScan;

  scanSud.applyMargin (radians (0.3));

  ...

  VecDBL point;
  bool   dernier;

  scanSud.initWalk (1.0e-4);
  while (scanSud.nextPoint (&point, &dernier))
  {
    cout << point.x () << ' ' << point.y () << ' ' << point.z () << endl;
    if (dernier)
      cout << "&\n";
  }
```

```
scanSud.stopWalk ();
```

conseils d'utilisation spécifiques

Pour l'utilisateur, la classe Field est très simple car elle modélise des notions ensemblistes de relativement haut niveau. On peut modéliser facilement des champs complexes en les construisant petit à petit à l'aide des divers opérateurs. La plupart du temps il y a de nombreuses façons différentes de réaliser la construction.

La classe Field est une refonte complète de l'ancienne classe Champ utilisées dans les versions de MARMOTTES antérieures à la 8.0. Elle corrige un problème grave mais à occurrence très rare.

implantation

Les attributs privés sont décrits sommairement dans la table 21, il n'y a pas d'attribut protégé.

TAB. 21: attributs privés de la classe Field

nom	type	description
empty_	bool	indicateur de vacuité du champ, utilisé lorsque le nombre d'arcs frontières est nul (dans ce cas soit le champ est vide, soit il recouvre toute la sphère unité)
boundary_	Boundary	frontière séparant l'intérieur de l'extérieur
rejectionCone_	Cone	cône permettant de rejeter par un test rapide une grande partie des points extérieurs au champ
activeRejectionCone_	bool	indicateur de validité de l'attribut rejectionCone_
targetPoint_	VecDBL	point de la frontière permettant de tester si un point de la sphère appartient ou nom au champ
activeTargetPoint_	bool	indicateur de validité de l'attribut targetPoint_
iter_	ArcIterateur *	itérateur sur l'arc frontière courant
init_	bool	indicateur d'initialisation de l'itérateur
i_	Boundary::const_iterator	itérateur sur la boucle courante
j_	Loop::const_iterator	itérateur sur la frontière courante
tolerance_	double	tolérance sur le parcours de la frontière

Les méthodes privées sont décrites dans la table 22.

TAB. 22: Field : méthodes privées

signature	description
void **combine** (const Field& *f*, bool *intersection*) **throw**(CantorErreurs)	combine la frontière de l'instance et celle de l'argument selon l'opération spécifiée (intersection ou réunion)
void **simplifyBoundary** ()	simplifie la frontière en éliminant les arcs nuls
void **select** (list<Arc> **ptrList*) const **throw**(CantorErreurs)	filtre les arcs qui sont à l'intérieur du champ, en les redécoupant au besoin
bool **selectClosest** (VecDBL **ptrP*, const VecDBL& *p*, int *nbPointsToAvoid*, const VecDBL ***pointsToAvoid*) const	sélectionne le point de la frontière le plus proche du point test *p*, en évitant les points listés dans la table *pointsToAvoid*, retourne une valeur vraie si un point a été trouvé.
Cone **recursRejectionCone** (const VecDBL& *p1*, const VecDBL& *p2*, const VecDBL& *p3*, int *depth*) const **throw**(CantorErreurs)	recherche récursivement un cône de rejet en partant des trois points de base spécifiés. La profondeur est mise à zéro par l'appelant de plus haut niveau et est incrémentée et testée au dessous de façon à limiter la recherche à quelques niveaux seulement.
void **computeRejectionCone** () **throw**(CantorErreurs)	Calcule un cône de rejet permettant lorsqu'on en trouve un d'accélérer les tests d'appartenance.
void **computeTargetPoint** (const VecDBL& *point*) **throw**(CantorErreurs)	Calcule un point cible optimisé pour le point test courant (ce point cible pourra cependant être réutilisé pour d'autres points tests)
void **firstCrossing** (Arc **ptrArc*, Loop::const_iterator **ptrCrossing*) **throw**(CantorErreurs)	tronque l'arc pointé par *ptrArc* au niveau du premier franchissement de la frontière et retourne un itérateur sur l'arc frontière correspondant dans la variable pointée par *ptrCrossing*

12.9 classe FonctionApprochee

description

La classe FonctionApprochee implante la notion d'approximation sur un espace vectoriel de fonctions, c'est à dire la modélisation d'une fonction exacte $f(x)$ par une combinaison linéaire de fonctions de base $\varphi_i(x)$.Les φ_i sont les fonctions de bases définissant l'espace vectoriel fonctionnel.

$$\tilde{f}(x) = \sum_{i=1}^{n} \alpha_i \varphi_i(x)$$

L'approximation est faite à partir d'un échantillon de points $x_1, \ldots, x_p$, des valeurs exactes $y_k = f(x_k)$ et des poids w_k que l'on attribue à chacun de ces points. Les coefficients α_i sont choisis tels qu'ils minimisent l'erreur quadratique ε :

$$\varepsilon = \sum_{k=1}^{p} w_k \left(y_k - \tilde{f}(x_k) \right)^2$$

La démarche de calcul consiste à créer une instance de la classe en lui donnant les φ_i sous forme d'un pointeur de fonction C, puis à constituer l'échantillon en accumulant dans l'instance les triplets (x_k, y_k, w_k), et enfin à faire l'ajustement (calcul des α_i) minimisant l'erreur quadratique.

Une fois l'ajustement réalisé on peut l'utiliser soit en demandant le tableau des coefficients α_i soit en chargeant l'instance elle-même de calculer la valeur de la fonction ajustée $\tilde{f}$ en n'importe quel point.

Il est possible d'ajouter ou de retrancher des points de l'échantillon (triplets (x_k, y_k, w_k)) à tout moment, même après ajustement, il faut bien sûr refaire l'ajustement après ces modifications.

Si l'on considère que les résidus $f(x_k) - \tilde{f}(x_k)$ sont dûs uniquement à un bruit aléatoire et si tous les poids sont identiques ($\forall_{k,l} w_k = w_l$), alors on peut convertir l'erreur quadratique en écart type de ce bruit par la formule :

$$\sigma = \sqrt{\frac{\sum_{k=1}^{p} \left(y_k - \tilde{f}(x_k)\right)^2}{p-1}}$$

La classe ne se sert jamais directement des x_k, elle se contente de prendre ceux que l'appelant lui donne pour les fournir à la fonction C de génération de l'espace vectoriel fonctionnel. La classe ne voit cet argument que sous forme d'un pointeur anonyme (`void *`).

La fonction C générant l'espace vectoriel fontionnel doit respecter le type TypeFonctionsBase, qui est défini dans le fichier `cantor/FonctionApprochee` par :

```
typedef int (*TypeFonctionsBase) (void *x, void *argsBase, double *ptrY);
```

Cette fonction sera appelée par la classe à chaque ajout de point (méthode **ajoutePointReference**), à chaque élimination de point (méthode **otePointReference**), et à chaque évaluation de la fonction approchée (opérateur **()**). Les pointeurs anonymes x et *argsBase* sont les pointeurs fournis par l'utilisateur de la classe, x étant fourni à chaque appel et *argsbase* étant fourni au moment de la construction de la classe. La fonction doit retourner les valeurs de chaque fonction de base dans le table pointée par $ptrY$ (la dimension du tableau correspond à ce qui a été défini au moment de la construction de l'instance).

interface publique

```
#include "cantor/FonctionApprochee.h"
```

TAB. 23: FonctionApprochee : méthodes publiques

signature	description
FonctionApprochee ()	prépare une instance non initialisée (nécessaire pour construire des tableaux de FonctionApprochee, ces instance ne sont pas utilisables directement, elles doivent être initialisées par affectation)
FonctionApprochee (int *dimension*, TypeFonctionsBase *fonctionsBase*, void **argsBase* = 0)	construit une instance permettant de modéliser une fonction dans l'espace vectoriel de dimension *dimension* engendré par les fonctions de bases calculées par *fonctionsBase*
FonctionApprochee (const FonctionApprochee& *f*)	constructeur par copie
FonctionApprochee& **operator** = (const FonctionApprochee& *f*)	affectation
~**FonctionApprochee** ()	destructeur, libère la mémoire allouée
	à suivre ...

TAB. 23: FonctionApprochee : méthodes publiques (suite)

signature	description
void **ajoutePointReference** (void *xk, double yk, double $wk = 1.0$) **throw**(CantorErreurs)	ajoute le triplet (x_k, y_k, w_k) à l'échantillon courant
void **otePointReference** (void *xk, double yk, double $wk = 1.0$) **throw**(CantorErreurs)	élimine le triplet (x_k, y_k, w_k) de l'échantillon courant
void **oteTousPointsReferences** ()	élimine tous les points de références de l'échantillon
int **nbPointsReferences** () const	retourne le nombre de points de références de l'échantillon
void **ajuste** (double *seuil* = 1.0e-10) **throw**(CantorErreurs)	réalise l'ajustement de la fonction au sens des moindres carrés sur l'échantillon (le seuil est utilisé pour tester les termes diagonaux lors de l'inversion de la matrice des moindres carrés, il doit être positif)
bool **estAjustee** () const	indique si la fonction a été ajustée
void **coefficients** (double *coeffs* []) const **throw**(CantorErreurs)	mets les α_i résultant de l'ajustement dans le tableau *coeffs*, lance une exception si la fonction n'avait pas été ajustée au préalable
void **erreurQuadratique** () const **throw**(CantorErreurs)	met l'erreur d'ajustement ε dans la variable pointée par *ptrErrQuad*, lance une exception si la fonction n'avait pas été ajustée au préalable
void **operator ()** (void *xk, double *$ptrY$) const **throw**(CantorErreurs)	évalue la fonction approchée $\tilde{f}$ au point x et met le résultat dans la variable pointée par $ptrY$, lance une exception si la fonction n'avait pas été ajustée au préalable
int **dimension** () const	retourne la dimension de l'espace d'ajustement
TypeFonctionsBase **fonctionsBase** () const	retourne le pointeur sur la fonction de génération de l'espace vectoriel fonctionnel
void ***argsBase** () const	retourne le pointeur anonyme passé en deuxième argument de la fonction de génération de l'espace vectoriel fonctionnel

exemple d'utilisation

```
#include "cantor/FonctionApprochee.h"

int fonctionsBase (void *x, void *argsBase, double *ptrY)
{ // fonctions de base d'un espace vectoriel
  // f1 : constante, f2 : linéaire, f3 : sinus, f4 : cosinus
  double t = *((double *) x);

  ptrY [0] = 1.0;
  ptrY [1] = t;
  ptrY [2] = sin (t);
  ptrY [3] = cos (t);

  return 0;
}
```

```
...

FonctionApprochee approx (4, fonctionsBase);

for (double x = 0.0; x < 4.0; x += 0.01)
  approx.ajoutePointReference ((void *) &x, fonctionReelle (x));

approx.ajuste ();

// récupération des éléments de l'ajustement
double coeffs [4];
approx.coefficients (coeffs);
(void) printf ("modèle ajusté : a + b x + c sin (x) + d cos (x)\n");
(void) printf ("a = %f\n", coeffs [0]);
(void) printf ("b = %f\n", coeffs [1]);
(void) printf ("c = %f\n", coeffs [2]);
(void) printf ("d = %f\n", coeffs [3]);

double errQuad = approx.erreurQuadratique ();
(void) printf ("écart type de l'ajustement : %f\n",
               sqrt (errQuad / (1.0 + approx.nbPointsReferences ())));
...
```

conseils d'utilisation spécifiques

Il est important pour réaliser un ajustement correct de bien répartir les points de l'échantillon pour que la matrice soit bien inversible, se contenter de tester que le nombre de points est supérieur à la dimension du problème peut être insuffisant si certains points sont trop proches et donc n'apportent pas d'information distincte.

implantation

Les attributs privés sont décrits sommairement dans la table 24, il n'y a pas d'attribut protégé.

TAB. 24: attributs privés de la classe FonctionApprochee

nom	type	description
moindresCarres_	MoindreCarreLineaire	modèle linéaire sous-jacent
fonctionsBase_	TypeFonctionsBase	pointeur vers la fonction générant l'espace vectoriel fonctionnel d'approximation
argsBase_	void *	pointeur anonyme servant d'argument à fonctionsBase_
YXk_	double *	table de travail stockant les $\varphi_i(x_k)$

Les méthodes privées sont décrites dans la table 25.

TAB. 25: FonctionApprochee : méthodes privées

signature	description
void **alloueTableau** (int *dimension*)	alloue le tableau de travail adapté à la *dimension* de l'espace vectoriel dans une instance vide
void **libereTableau** ()	libère la mémoire allouée pour le tableau interne

12.10 classe Intervalle

description

La classe Intervalle implante la notion d'intervalle sur $\mathbb{R}$. Les opérations les plus importantes qu'elle propose sont les opérations de réunion (par les opérateurs '*=' et '*'), d'intersection (par les opérateurs '+=' et '+'), et de test d'inclusion d'un scalaire (ou d'un intervalle).

Les réels de l'informatique ayant une imprécision incontournable, aucune distinction entre intervalle ouvert, semi-ouvert, ou fermé n'est faite. Si l'utilisateur veut implanter ces notions, c'est à lui de gérer l'intervalle de confiance qu'il utilise pour l'égalité, une classe d'usage général ne peut pas faire d'hypothèse réaliste sur ce point.

On peut noter que presque toutes les méthodes de la classe sont `inline`, c'est à dire peu coûteuses. En fait la seule méthode à ne pas être `inline` est l'opérateur d'affectation.

interface publique

```
#include "cantor/Intervalle.h"
```

TAB. 26: Intervalle : méthodes publiques

signature	description
Intervalle ()	construit l'intervalle $[0;1]$
Intervalle (double *a*, double *b*)	construit l'intervalle de bornes *a* et *b* (l'ordre est indifférent)
Intervalle (const Intervalle& *i*)	constructeur par copie
Intervalle& **operator** = (const Intervalle& *i*)	affectation
Intervalle ()	destructeur
double **inf** () const	retourne une copie de la borne inférieure de l'intervalle
double **sup** () const	retourne une copie de la borne supérieure de l'intervalle
double **longueur** () const	retourne une copie de la longueur de l'intervalle
bool **contient** (double *x*) const	indique si l'instance contient le réel *x*
bool **contient** (const Intervalle& *i*) const	indique si l'instance contient tous les points de l'intervalle *i*
bool **rencontre** (const Intervalle& *i*) const	indique si l'instance rencontre l'intervalle *i* (c'est à dire s'ils ont au moins un point commun)
bool **disjoint** (const Intervalle& *i*) const	indique si l'instance et l'intervalle *i* sont totalement disjoints

à suivre ...

TAB. 26: Intervalle : méthodes publiques (suite)

signature	description
void **decale** (double *delta*) const	décale les deux bornes de l'intervalle de la valeur *delta*
bool **operator** < (const Intervalle& *i*) const	indique si tous les points de l'instance sont strictements inférieurs à tous les points de *i*
bool **operator** > (const Intervalle& *i*) const	indique si tous les points de l'instance sont strictements supérieurs à tous les points de *i*
Intervalle& **operator** += (const Intervalle& *i*)	remplace l'instance par l'intervalle résultant de la réunion de l'instance et de *i*. *Attention*, si au départ les intervalles étaient disjoints, cette opération comble le trou qui existait originellement entre les intervalles !
Intervalle& **operator** *= (const Intervalle& *i*)	remplace l'instance par l'intervalle résultant de l'intersection de l'instance et de *i*. Si les intervalles étaient disjoints au départ, l'intervalle résultant est de longueur nulle, et ses deux bornes sont égales à la plus grande des bornes inférieures des intervalles initiaux

TAB. 27: Intervalle : opérations non membres

signature	description
inline Intervalle **operator** + (const Intervalle& *i*, const Intervalle& *j*)	retourne l'intervalle réunion des intervalles *i* et *j*. *Attention*, si au départ les intervalles étaient disjoints, cette opération comble le trou qui existait originellement entre les intervalles !
inline Intervalle **operator** * (const Intervalle& *i*, const Intervalle& *j*)	retourne l'intervalle intersection des intervalles *i* et *j*. Si les intervalles étaient disjoints au départ, l'intervalle résultant est de longueur nulle, et ses deux bornes sont égales à la plus grande des bornes inférieurs des intervalles initiaux.
inline ostream& **operator** << (ostream& *s*, const Intervalle& *i*)	formate une chaîne de caractères représentant l'intervalle *i* sur le flot *s*

exemple d'utilisation

```
#include "cantor/Intervalle.h"
...
Intervalle i (-1.0, 2.0);
Intervalle j (1.0, 3.5);
Intervalle ij = i * j;
cout << i
     << (i.contient (1.2) ? " contient " : " ne contient pas ")
     << 1.2 << endl;

j = Intervalle (4.0, 5.0);
if (i < j)
  cout << i << '<' << j << endl;

j = Intervalle (-0.5, 0.5);
if (i.contient (j))
```

```
  cout << i <<  " contient " << j << endl;
...
```

conseils d'utilisation spécifiques

La classe Intervalle est d'utilisation relativement simple, la seule opération pouvant poser problème est la réunion d'intervalle (opérateurs += et +), qui comble le trou entre deux intervalles si ceux ci étaient disjoints au départ. Si l'on désire pouvoir gérer ce cas correctement d'un point de vue mathématique, il faut passer par la classe Creneau, qui est décrite à la section 12.7.

implantation

Les attributs privés sont décrits sommairement dans la table 28, il n'y a pas d'attribut protégé.

TAB. 28: attributs privés de la classe Intervalle

nom	type	description
inf_	double	borne inférieure de l'intervalle
sup_	double	borne supérieure de l'intervalle

12.11 classe MoindreCarreLineaire

description

La classe MoindreCarreLineaire implante la notion d'ajustement au sens des moindres carrés d'un modèle linéaire de résidus de mesures d'un système.

$$r_k = m_k - m_k^{\text{th}} = \sum_{i=1}^{n} \alpha_i \varphi_{k,i}$$

Les m_k étant les mesures réelles, les m_k^{th} étant les mesures théoriques, et les $\varphi_{k,i}$ étant les coefficients de linéarité connus[8].

L'ajustement est fait à partir d'un échantillon de résidus $r_1, \ldots, r_p$, des coefficients $\varphi_{k,i}$ et des poids w_k que l'on attribue à chacun de ces points. Les coefficients α_i déterminés au cours de l'ajustement minimisent l'erreur quadratique ε :

$$\varepsilon = \sum_{k=1}^{p} w_k r_k^2$$

La démarche de calcul consiste à créer une instance de la classe en lui donnant la dimension p du modèle linéaire puis à constituer l'échantillon en accumulant dans l'instance les triplets $(r_k, \varphi_{k,i}, w_k)$, et enfin à faire l'ajustement (calcul des α_i) minimisant l'erreur quadratique.

Une fois l'ajustement réalisé on peut l'utiliser soit en demandant le tableau des coefficients α_i soit en chargeant l'instance elle-même de calculer la valeur du résidu linéarisé r en n'importe quel point.

[8] ce sont souvent les dérivées partielles des mesures par rapport au vecteur d'état pour une itération donnée de la résolution d'un problème de moindres carrés non linéaire

Il est possible d'ajouter ou de retrancher des points de l'échantillon (triplets $(r_k, \varphi_{k,i}, w_k)$) à tout moment, même après ajustement, il faut bien sûr refaire l'ajustement après ces modifications.

Si l'on considère que les résidus $r_k = m_k - m_k^{\text{th}}$ sont dûs uniquement à un bruit aléatoire et si tous les poids sont identiques ($\forall_{k,l} w_k = w_l$), alors on peut convertir l'erreur quadratique en écart type de ce bruit de mesure par la formule :

$$\sigma = \sqrt{\frac{\sum_{k=1}^{p} r_k^2}{p-1}}$$

interface publique

```
#include "cantor/MoindreCarreLineaire.h"
```

TAB. 29: MoindreCarreLineaire : méthodes publiques

signature	description
MoindreCarreLineaire ()	prépare une instance non initialisée (nécessaire pour construire des tableaux de MoindreCarreLineaire, ces instance ne sont pas utilisables directement, elles doivent être initialisées par affectation)
MoindreCarreLineaire (int *dimension*)	construit une instance permettant de modéliser des résidus dans un espace vectoriel de dimension *dimension*
MoindreCarreLineaire (const MoindreCarreLineaire& *m*)	constructeur par copie
MoindreCarreLineaire& **operator** = (const MoindreCarreLineaire& *m*)	affectation
~**MoindreCarreLineaire** ()	destructeur, libère la mémoire allouée
void **ajouteResidu** (double *rk*, double fki [], double *wk* = 1.0)	ajoute le triplet $(r_k, f_{k,i}, w_k)$ à l'échantillon courant
void **oteResidu** (double *rk*, double fki [], double *wk* = 1.0)	élimine le triplet $(r_k, f_{k,i}, w_k)$ de l'échantillon courant
void **oteTousResidus** ()	élimine tous les résidus de l'échantillon
int **nbResidus** () const	retourne le nombre de résidus de l'échantillon
void **ajuste** (double *seuil* = 1.0e-10) **throw**(CantorErreurs)	réalise l'ajustement du modèle de résidus au sens des moindres carrés sur l'échantillon (le seuil est utilisé pour tester les termes diagonaux lors de l'inversion de la matrice des moindres carrés, il doit être positif), lance une exception en cas de problème
bool **estAjuste** () const	indique si le modèle a été ajusté
void **coefficients** (double *ai* []) const **throw**(CantorErreurs)	mets les α_i résultant de l'ajustement dans le tableau *ai*, lance une exception si la fonction n'avait pas été ajustée au préalable
double **erreurQuadratique** () const **throw**(CantorErreurs)	met l'erreur d'ajustement ε dans la variable pointée par *ptrErrQuad*, lance une exception si la fonction n'avait pas été ajustée au préalable

à suivre ...

TAB. 29: MoindreCarreLineaire : méthodes publiques (suite)

signature	description
void **operator ()** (double *$ptrFk$, double fki []) const **throw**(CantorErreurs)	évalue le modèle linéaire des résidus en fonction des coefficients $f_{k,i}$ et met le résultat dans la variable pointée par $ptrRk$, lance une exception si la fonction n'avait pas été ajustée au préalable
int **dimension** () const	retourne la dimension de l'espace d'ajustement

TAB. 30: MoindreCarreLineaire : fonctions globales associées

signature	description
void **factLDLt** (double *m, int n, double *seuil*) **throw**(CantorErreurs)	factorise sous forme LDL^T et *en place* la matrice symétrique définie positive m de dimension n, en utilisant *seuil* pour tester les pivots lors de l'inversion ; lance une exception en cas d'impossibilité
void **resoudLDLt** (const double *m, int n, double x [])	résoud *en place* l'équation $y = mx$, où m a déjà été factorisée sous forme LDL^T

exemple d'utilisation

```
#include "cantor/MoindreCarreLineaire.h"

static double x [] = { 1.0,  2.0,  3.0,  4.0, 5.0,   6.0,   7.0,  8.0  };
static double y [] = { 0.97, 3.01, 2.93, 6.9, 8.99, 11.03, 13.0, 14.98 };

// ajustement d'une droite sur l'échantillon
// on a phi (k, 1) = 1 et phi (k, 2) = x (i)
double fki [2];
fki [0] = 1;
MoindreCarreLineaire initial (2);
for (int i = 0; i < sizeof (x) / sizeof (double); i++)
{ fki [1] = x [i];
  initial.ajouteResidu (y [i], fki);
}

initial.ajuste ();

// on se donne pour l'exemple un seuil de rejet à 2 sigmas
double seuil = initial.erreurQuadratique ();
seuil = 2.0 * sqrt (seuil / (1.0 + initial.nbResidus ()));

// on refait un échantillon sans les points aberrants
MoindreCarreLineaire final (initial);
for (int i = 0; i < initial.nbResidus (); i++)
{ double estime;
  fki [1] = x [i];
  initial (&estime, fki);
```

```
  if (abs (estime - y [i]) > seuil)
    final.oteResidu (y [i], fki);
}

final.ajuste ();

double coeffs [2];
final.coefficients (coeffs);
cout << coeffs [0] << " + " << coeffs [1] << " x" << endl;
```

conseils d'utilisation spécifiques

La classe MoindreCarreLineaire utilise les deux fonctions globales **factLDLt** et **resoudLDLt** pour réaliser l'ajustement. Ces fonctions ont été rendues globales de façon à pouvoir être appelées directement par un utilisateur de CANTOR.

Dans la fonction **factLDLt**, la matrice initiale est symétrique définie positive, seule une moitié de cette matrice est donc nécessaire pour les calculs. Afin d'optimiser la place, le nombre de transferts mémoire et les calculs, la fonction **factLDLt** attend uniquement la moitié utile de la matrice, rangée dans un vecteur de dimension $n(n+1)/2$. Les n premières composantes doivent contenir les éléments $m_{1,1}$ à $m_{1,n}$, les $n-1$ composantes suivantes doivent contenir les éléments $m_{2,2}$ à $m_{2,n}$, ... Ce type de rangement est traditionnel dans les codes d'algèbre linéaire.

implantation

Les attributs privés sont décrits sommairement dans la table 31, il n'y a pas d'attribut protégé.

TAB. 31: attributs privés de la classe MoindreCarreLineaire

nom	type	description
dimension_	int	dimension de l'espace vectoriel d'ajustement
nbResidus_	int	taille courante de l'échantillon
sommeRk2_	double	scalaire pour le calcul de l'erreur
vecSommeRkFk_	double *	second membre des moindres carrés
matSommeFkFk_	double *	matrice des moindres carrés
estAjuste_	bool	indicateur d'ajustement
ai_	double *	coefficients d'ajustement
errQuad_	double	erreur quadratique d'ajustement

Les méthodes privées sont décrites dans la table 32.

TAB. 32: MoindreCarreLineaire : méthodes privées

signature	description
void **allouеTableaux** (int *dimension*)	alloue les tableaux adaptée à la *dimension* de l'espace vectoriel dans une instance vide
void **libereTableaux** ()	libère la mémoire allouée pour les tableaux internes

12.12 classe Node

description

Cette classe est utilisée conjointement avec la classe AnnotatedArc (cf 12.1) pour réaliser la propagation des marques apposées sur les arcs de façon cohérente avec la topologie de l'entrelacement des frontières.

Un nœud représente la jonction entre deux ou quatre arcs annotés. Les nœuds simples ne reliant qu'un arc amont à un arc aval correspondent à une portion de frontière classique, séparant une zone intérieure d'une zone extérieure. Les nœuds multiples reliant quatre arcs correspondent à l'intersection de deux frontières, il y a donc un arc amont et un arc aval issus de la première frontière, un arc amont et un arc aval issus de la seconde frontière.

Les nœds simples sont crées champ par champ avant leur combinaison alors que les nœuds multiples sont crées au début de la combinaison des frontières. Tous les nœuds sont ensuite utilisés lors du marquage des arcs frontière. Lorsqu'un arc est identifié comme faisant partie soit de la frontière de l'intersection soit de la frontière de la réunion des champs, cette information est propagée par l'intermédiaire des nœuds aux arcs voisins, et éventuellement au reste de l'entrelacs, de proche en proche en suivant la structure topologique.

interface publique

```
#include "cantor/Node.h"
```

TAB. 33: Node : méthodes publiques

signature	description
Node ()	crée un nœud par défaut
Node (const VecDBL& *u*, AnnotatedArc **upstream*, AnnotatedArc **downstream*) **throw**(CantorErreurs)	crée un nœud reliant les deux arcs passés en argument
Node (const VecDBL& *u*, AnnotatedArc **upstream1*, AnnotatedArc **downstream1*, AnnotatedArc **upstream2*, AnnotatedArc **downstream2*) **throw**(CantorErreurs)	crée un nœud reliant les quatre arcs passés en argument

à suivre ...

TAB. 33: Node : méthodes publiques (suite)

signature	description
Node (const Node& *n*)	constructeur par copie
Node& **operator** = (const Node& *n*)	affectation
Node ()	destructeur
const VecDBL& **direction** () const	retourne la direction du nœud sur la sphère unité
int **nbArcs** () const	retourne le nombre d'arcs reliés par ce nœud
AnnotatedArc ***arc** (int *i*) const	retourne l'arc spécifié par son index
AnnotatedArc ***ptrUpstreamArc** () const	retourne l'arc amont s'il est unique, retourne un pointeur nul sinon
AnnotatedArc ***ptrDownstreamArc** () const	retourne l'arc aval s'il est unique, retourne un pointeur nul sinon
void **replaceUpstreamArc** (AnnotatedArc **ptrOld*, AnnotatedArc **ptrNew*)	remplace l'arc amont *ptrOld* par *ptrNew*
void **replaceDownstreamArc** (AnnotatedArc **ptrOld*, AnnotatedArc **ptrNew*)	remplace l'arc aval *ptrOld* par *ptrNew*
void **detach** (AnnotatedArc **upstream*, AnnotatedArc **downstream*)	détache les deux arcs spécifiés de l'instance (qui change ainsi de type)
void **absorbDownstream** (AnnotatedArc **ptrNull1*, AnnotatedArc **ptrNull2*) **throw**(CantorErreurs)	absorbe les deux arcs nuls ainsi que leur nœud commun suivant
bool **propagate** ()	propage les marques des arcs connectés à l'instance le plus loin possible
bool **isConvergent** (AnnotatedArc **ptrTest*, set<const AnnotatedArc *> **ptrAlreadyTested*) const	test si l'arc *ptrTest* converge vers l'instance, soit directement soit avec des arcs nuls intermédiaires. Utilise l'ensemble pointé par *ptrAlreadyTested* pour éviter de tester plusieurs fois les mêmes arcs.
bool **select** (AnnotatedArc::Annotation **annotation*, const AnnotatedArc ***ptrA*, bool **ptrDirect*, set<const AnnotatedArc *> **ptrAlreadySelected*) const	selectionne un arc selon l'*annotation* spécifiée et le retourne dans la variable pointée par *ptrA*, en indiquant dans la variable pointée par *ptrDirect* si l'arc est parcouru dans le sens direct ou inverse en partant du nœud courant. Utilise l'ensemble pointé par *ptrAlreadySelected* pour éviter de sélectionner plusieurs fois les mêmes arcs.

exemple d'utilisation

```
#include "marmottes/Node.h"

bool
AnnotatedArc::isConnected (AnnotatedArc *ptrA) const
{ // check if the instance is connected with another arc
  Node *downstream = (Node *) downstreamNode_.memoire ();
  Node *upstream   = (Node *) upstreamNode_.memoire ();
```

```
  set<const AnnotatedArc *> alreadyTested;
  alreadyTested.insert (this);

  return (((downstream != 0)
           && downstream->isConvergent (ptrA, &alreadyTested))
          ||  ((upstream != 0)
               && upstream->isConvergent (ptrA, &alreadyTested)));

}

void
AnnotatedArc::propagate ()
{ // propagate the annotation of the instance to the neighboring
  // arcs (recursively as far as possible) through the nodes

  // propagate downstream
  Node *downstream = (Node *) downstreamNode_.memoire ();
  if (downstream)
    downstream->propagate ();

  // propagate upstream
  Node *upstream = (Node *) upstreamNode_.memoire ();
  if (upstream)
    upstream->propagate ();

}
```

conseils d'utilisation spécifiques

La classe Node est utilisée par les classes Braid et AnnotatedArc pour réaliser la propagation des marques lors de la détermination de la frontière d'une combinaison de deux champs. elle n'a pas d'autre utilisation.

implantation

Les attributs privés sont décrits sommairement dans la table 34, il n'y a pas d'attribut protégé.

TAB. 34: attributs privés de la classe Node

nom	type	description
type_	Type	type du nœud (inconnu, simple ou multiple)
direction_	VecDBL	direction du nœud sur la sphère unité
arcs_	AnnotatedArc * [4]	table des arcs connectés au nœud

Les méthodes privées sont décrites dans la table 35.

TAB. 35: Node : méthodes privées

signature	description
bool **propagate2** ()	propage les marques dans le cas d'un nœud simple reliant deux arcs
bool **propagate4** ()	propage les marques dans le cas d'un nœud multiple reliant quatre arcs

12.13 classe Polynome

description

La classe Polynome<T> implante les polynômes à coefficients et valeurs de type T.

interface publique

Il est recommandé de ne pas inclure directement le fichier de déclaration de la classe (voir la section 12), mais de passer plutôt par l'une des directives :

```
#include "cantor/DeclDBL.h"
#include "cantor/DeclVD1.h"
#include "cantor/DeclVD2.h"
```

Ces fichiers réalisent l'inclusion du fichier de déclaration et définissent les types PolDBL en tant qu'alias de Polynome<double>, PolVD1 en tant qu'alias de Polynome<ValeurDerivee1>, et PolVD2 en tant qu'alias de Polynome<ValeurDerivee2>.

Outre les méthodes publiques décrites dans la table 36 et les fonctions décrites dans la table 37, tous les opérateurs +=, -=, *=, /=, +, -, ^, |, *, /, combinant des instances de Polynome<T> et des scalaires de type T sont définis[9]. L'opérateur - unaire est également défini.

TAB. 36: Polynome : méthodes publiques

signature	description
Polynome ()	construit un polynôme invalide (de degré -1), un tel polynôme n'est utilisable qu'après avoir été écrasé par une affectation d'un polynôme valide ; ce constructeur permet cependant de créer des tableaux de polynômes, chaque cellule étant correctement initialisée après sa construction
Polynome (int *degre*, const T coeff []) **throw**(CantorErreurs)	construit un polynôme de *degre* et de *coeff*icients donnés. Une exception est lancée si le degré est hors des bornes tolérées (de 0 à `polynome_max_degre`).
Polynome (const T& *a0*)	construit le polynôme a_0
Polynome (const T& *a0*, const T& *a1*)	construit le polynôme $a_0 + a_1 X$
Polynome (const T& *a0*, const T& *a1*, const T& *a2*)	construit le polynôme $a_0 + a_1 X + a_2 X^2$
	à suivre ...

[9] l'opérateur /= n'est défini que pour un argument de type T, pas pour un argument de type Polynome<T>

TAB. 36: Polynome : méthodes publiques (suite)

signature	description
Polynome (const T& *a0*, const T& *a1*, const T& *a2*, const T& *a3*)	construit le polynôme $a_0 + a_1X + a_2X^2 + a_3X^3$
Polynome (const T& *a0*, const T& *a1*, const T& *a2*, const T& *a3*, const T& *a4*)	construit le polynôme $a_0 + a_1X + a_2X^2 + a_3X^3 + a_4X^4$
Polynome (const T& *a0*, const T& *a1*, const T& *a2*, const T& *a3*, const T& *a4*, const T& *a5*)	construit le polynôme $a_0 + a_1X + a_2X^2 + a_3X^3 + a_4X^4 + a_5X^5$
Polynome (const Polynome<T>& *p*)	constructeur par copie
Polynome<T>& **operator** = (const Polynome<T>& *p*)	affectation
int **degre** () const	retourne le degré du polynôme
T **coeff** (int *i*) const **throw**(CantorErreurs)	retourne le coefficient du monôme de degré *i* (une erreur est générée si l'index n'est pas entre 0 et **degre** () - 1)
T **operator ()** (const T& *x*) const **throw**(CantorErreurs)	calcule la valeur du polynôme en *x*, une erreur est générée si le polynôme est invalide
Polynome<T> **operator ()** (const Polynome<T>& *x*) const **throw**(CantorErreurs)	calcule la composition de *x* par le polynôme, une erreur est générée si le polynôme ou si *x* est invalide
T **derivee** (const T& *x*) const **throw**(CantorErreurs)	retourne la valeur de la dérivée de l'instance en *x*, une erreur est générée si le polynôme est invalide
Polynome<T> **derivee** () const **throw**(CantorErreurs)	retourne le polynôme dérivé de l'instance, une erreur est générée si le polynôme est invalide
Polynome<T> **integrale** (const T& *y* = T (0.0), const T& *x* = T (0.0)) const **throw**(CantorErreurs)	retourne le polynôme résultat de l'intégration de l'instance, en positionnant les constantes d'intégration de sorte que ce polynôme prenne la valeur *y* en *x*, une erreur est générée si le polynôme est invalide
bool **zerosCalculables**() const	indique si les zéros de l'instance sont calculables par une méthode directe (c'est vrai quand le degré du polynôme est inférieur à cinq).
void **zeros** (int* *n*, T z [], int o []) const **throw**(CantorErreurs)	calcule les zéros réels de l'instance par une méthode directe, et stocke dans le tableau *z* la valeur des zéros (dans l'ordre croissant) et dans le tableau *o* l'ordre de ces zéros. Au cas où tous les zéros seraient séparés, l'appelant doit fournir des tableaux pouvant contenir **degre** () éléments. Le nombre d'éléments réellement utilisés est retourné dans l'entier pointé par *n*. On assure que la somme des *o*[i] pour i variant de 0 à (**n*)-1 est inférieure ou égale au degré du polynôme.
Polynome ()	destructeur.

TAB. 37: Polynome : fonctions non membres

signature	description
inline ostream& **operator** << (ostream& *s*, const Polynome<T>& *u*)	formate une chaîne de caractères représentant le polynôme sur le flot *s*

exemple d'utilisation

```
#include "cantor/DeclDBL.h"
...
double epsilon = 1.0e-5;
PolDBL pa (4.0 + 2.0 * epsilon, -2.0);
PolDBL pb (epsilon - 2.0, 1.0);

// polynôme s'annulant en 2-epsilon et en 2+epsilon
PolDBL p2 = pa * pb * 5.0;

double zeros [polynome_max_degre];
int    n, ordre [polynome_max_degre];

p2.zeros (&n, zeros, ordre);
cout << n <<  " zéros :\n";
for (int i = 0; i < n; i++)
{ cout << ' ' << zeros [i] << " d'ordre : " << ordre [i];
  cout << " (P (" << zeros [i] << ") = " << p2 (zeros [i]) << ")\n";
}

...

// Calcul des premiers polynômes de Chebyshev
PolDBL chebyshev [7];
chebyshev [0] = PolDBL (1.0);
chebyshev [1] = PolDBL (0.0, 1.0);
PolDBL deuxX (0.0, 2.0);
for (i = 2; i < 7; i++)
  chebyshev [i] = deuxX * chebyshev [i-1] - chebyshev [i-2];
```

conseils d'utilisation spécifiques

La constante entière `polynome_max_degre` est définie par le fichier d'en-tête de la classe, elle peut être utilisée par l'appelant pour dimensionner ses tableaux. Sa valeur est de 10 pour la version 5.1 de CANTOR.

La classe Polynome<T> s'appuie sur le mécanisme de `template` pour s'abstraire du type T des coordonnées des polynômes. Ainsi, peuvent être utilisés des polynômes dont les coordonnés sont représentées par des doubles, des ValeurDerivee1 ou encore des ValeurDerivee2. Le mécanisme de `template`, s'il offre des avantages indéniables, est parfois trop souple et peut engendrer des erreurs (que penser d'un Polynome<char *> ?). Il est donc recommandé aux utilisateurs d'utiliser les types prédéfinis PolDBL, PolVD1, et PolVD2 définis dans les fichiers `cantor/DeclDBL.h` , `cantor/DeclVD1.h` et `cantor/DeclVD2.h` , et de ne pas instancier eux-mêmes

les `templates`. Si les utilisateurs doivent utiliser simultanément des double et des ValeurDerivee1, il leur faudra inclure le fichier `cantor/DeclDBLVD1.h`, qui outre inclure les fichiers précédents, définit également un certain nombre d'opérateurs de conversions permettant par example d'ajouter un VecDBL à un VecVD1. Les fichiers `cantor/DeclDBLVD2.h` et `cantor/DeclVD1VD2.h` existent aussi avec des rôles symétriques.

implantation

Les attributs privés sont décrits sommairement dans la table 38, il n'y a pas d'attribut protégé.

TAB. 38: attributs privés de la classe Polynome

nom	type	description
degre_	int	degré du polynôme (limité par la constante entière polynome_max_degre)
coeff_	T [polynome_max_degre + 1]	tableau des coefficients du polynôme

Les méthodes privées sont décrites dans la table 39.

TAB. 39: Polynome : méthodes privées

signature	description
void **affecter** (const Polynome<T>& *p*)	copie le polynôme donné en argument dans l'instance. Cette fonction est utilisée par le constructeur par copie et l'affectation
void **ajusteDegre** ()	cette fonction diminue le degré de l'instance tant que le terme de plus haut degré est supérieur à la constante symbolique `cantorEpsilon` en valeur absolue. Elle est utilisée par presque tous les constructeurs et par les fonctions membres qui modifient le terme de plus haut degré (operator +=, ...).

12.14 classe Resolution1Iterateur

description

La classe Resolution1Iterateur permet d'itérer sur tous les zéros d'une fonction, qu'elle calcule à l'aide des fonctions globales de `cantor/Resolution1.h`.

interface publique

```
#include "cantor/Resolution1Iterateur.h"
```

TAB. 40: Resolution1Iterateur : méthodes publiques

signature	description
Resolution1Iterateur (TypeFoncVD1 *f*, void* *arg*, double *a*, double *b*, double *pas*, double *convergenceX*, double *convergenceY*)	construit une instance d'itérateur sur les zéros de f dans l'intervalle $[a;b]$ (voir section 12.14 pour plus les détails)
Resolution1Iterateur (TypeFoncVD1 *f*, void* *arg*, double *a*, double *b*, int *n*, double *convergenceX*, double *convergenceY*)	construit une instance d'itérateur sur les zéros de f dans l'intervalle $[a;b]$ (voir section 12.14 pour plus les détails)
double **convergenceX** () const	retourne le seuil de convergence en abscisse
double **convergenceY** () const	retourne le seuil de convergence en ordonnée
void **reinitialise** ()	reinitialise l'itérateur au début de la recherche
ValeurDerivee1 **evalueFonction** (double *x*) const	retourne la valeur en x de la fonction à annuler (valeur de retour de f (x, *arg*), où f et *arg* sont les arguments stockés dans l'instance à la construction
double **zeroSuivant** ()	retourne une copie du zéro suivant, ou une valeur largement supérieure à la borne supérieure de l'intervalle de recherche si aucun zéro n'est trouvé (la valeur est de l'ordre de la borne sup plus un million de fois la taille de l'intervalle, de façon à permettre de faire des tests de fin de recherche qui ne prennent pas un zéro égal à la borne sup pour un indicateur de fin; l'utilisateur peut utiliser pour ses tests une valeur au delà de l'intervalle).
Resolution1Iterateur ()	destructeur.

interfaces protégées

TAB. 41: Resolution1Iterateur : méthodes protégées

signature	description
Resolution1Iterateur ()	constructeur par défaut.
Resolution1Iterateur (const Resolution1Iterateur & *other*)	constructeur par copie.
Resolution1Iterateur & **operator** = (const Resolution1Iterateur & *other*)	affectation.

exemple d'utilisation

```
#include "cantor/Resolution1Iterateur.h"
...

ValeurDerivee1 SinusCroissant (double t, void* d)
{ // fonction s'annulant 9 fois entre -11 et 11
  // avec deux séries de zéros très proches (+/-10.907, +/-10.904)

  // incrementation du compteur d'appels
```

```
  *((int *) d) += 1;
  ValeurDerivee1 x (t, 1.0);
  return sin (x) + x * 0.091325;
}
...
int compteur = 0;
Resolution1Iterateur iter (SinusCroissant, (void *) &compteur,
                           -12.0, 12.0, 10, 1.0e-4, 1.0e-4);
double z;
while ((z = iter.zeroSuivant ()) < 13.0)
  cout << z << :  << iter.evalueFonction (z).f0 () << endl;
cout << compteur <<  " appels à la fonction f\n";
...
```

conseils d'utilisation spécifiques

La méthode utilisée pour parcourir les zéros d'une fonction sur un intervalle donné est la suivante : on commence par chercher des intervalles monotones encadrant des zéros, soit à l'aide du *pas* soit en découpant l'intervalle en n tronçons (*pas* et n étant des paramètres de construction de l'instance). Une fois un intervalle encadrant une racine trouvé, cet intervalle est réduit jusqu'à trouver le zéro.

La phase de séparation peut conduire à rechercher finement certains extremums locaux s'ils sont proches d'un zéro, la méthode permet donc de trouver des zéros dans des cas limites où la fonction ne change de signe que très brièvement, c'est à dire quand deux zéros successifs sont très proches. Elle est limitée par le fait que deux extremums très proches peuvent ne pas être vus, selon la valeur de *pas* (ou de n) utilisée.

Le critère d'arrêt porte sur les bornes d'un intervalle encadrant la racine, à la fois en abscisse (longueur de l'intervalle) et en ordonnée (valeur maximale atteinte aux bornes). Dès que les bornes vérifient l'un des seuils *convergenceX* ou *convergenceY* la réduction de l'intervalle est stoppée et le *meilleur* point est retourné. Si l'un des critères est négatif, il ne sera jamais respecté et c'est donc l'autre qui sera déterminant.

Une description détaillée des algorithmes se trouve dans [DR1].

implantation

Les attributs privés sont décrits sommairement dans la table 42, il n'y a pas d'attribut protégé.

TAB. 42: attributs privés de la classe Resolution1Iterateur

nom	type	description
convergenceX_	double	seuil de convergence en x (s'il est négatif, on ne convergera jamais en x)
convergenceY_	double	seuil de convergence en y (s'il est négatif, on ne convergera jamais en y)
tMin_	double	borne inférieure de l'intervalle de recherche des zéros
tMax_	double	borne supérieure de l'intervalle de recherche des zéros
pas_	double	pas de dichotomie pour la recherche des extremums
		à suivre ...

TAB. 42: attributs privés de la classe Resolution1Iterateur (suite)

nom	type	description
min_	Variation1	minimum courant (ce n'est en fait pas forcément un minimum, mais c'est un point suffisamment bas pour être utilisable dans la recherche des zéros)
max_	Variation1	maximum courant (ce n'est en fait pas forcément un maximum, mais c'est un point suffisamment haut pour être utilisable dans la recherche des zéros)
precedente_	Variation1	variation de la fonction au point précédent
f_	TypeFoncVD1	pointeur sur la fonction à annuler
arg_	void*	second argument d'appel de la fonction pointée par f_ (le premier argument est le t variant dans l'intervalle [tMin_ ; tMax_])

12.15 classe Resolution2Iterateur

description

La classe Resolution2Iterateur permet d'itérer sur tous les zéros d'une fonction, qu'elle calcule à l'aide des fonctions globales de `cantor/Resolution2.h` .

interface publique

```
#include "cantor/Resolution2Iterateur.h"
```

TAB. 43: Resolution2Iterateur : méthodes publiques

signature	description
Resolution2Iterateur (TypeFoncVD2 *f*, void* *arg*, double *a*, double *b*, double *pas*, double *convergenceX*, double *convergenceY*)	construit une instance d'itérateur sur les zéros de *f* dans l'intervalle $[a;b]$ (voir section 12.15 pour plus les détails)
Resolution2Iterateur (TypeFoncVD2 *f*, void* *arg*, double *a*, double *b*, int *n*, double *convergenceX*, double *convergenceY*)	construit une instance d'itérateur sur les zéros de *f* dans l'intervalle $[a;b]$ (voir section 12.15 pour plus les détails)
double **convergenceX** () const	retourne le seuil de convergence en abscisse
double **convergenceY** () const	retourne le seuil de convergence en ordonnée
void **reinitialise** ()	reinitialise l'itérateur au début de la recherche
ValeurDerivee2 **evalueFonction** (double *x*) const	retourne la valeur en *x* de la fonction à annuler (valeur de retour de *f* (*x*, *arg*), où *f* et *arg* sont les arguments stockés dans l'instance à la construction
à suivre ...	

TAB. 43: Resolution2Iterateur : méthodes publiques (suite)

signature	description
double **zeroSuivant** ()	retourne une copie du zéro suivant, ou une valeur largement supérieure à la borne supérieure de l'intervalle de recherche si aucun zéro n'est trouvé (la valeur est de l'ordre de la borne sup plus un million de fois la taille de l'intervalle, de façon à permettre de faire des tests de fin de recherche qui ne prennent pas un zéro égal à la borne sup pour un indicateur de fin; l'utilisateur peut utiliser pour ses tests une valeur au delà de l'intervalle).
Resolution2Iterateur ()	destructeur.

interfaces protégées

TAB. 44: Resolution2Iterateur : méthodes protégées

signature	description
Resolution2Iterateur ()	constructeur par défaut.
Resolution2Iterateur (const Resolution2Iterateur & *other*)	constructeur par copie.
Resolution2Iterateur & **operator** = (const Resolution2Iterateur & *other*)	affectation.

exemple d'utilisation

```
#include "cantor/Resolution2Iterateur.h"
...

ValeurDerivee2 SinusCroissant (double t, void* d)
{ // fonction s'annulant 9 fois entre -11 et 11
  // avec deux séries de zéros très proches (+/-10.907, +/-10.904)

  // incrementation du compteur d'appels
  *((int *) d) += 1;
  ValeurDerivee2 x (t, 1.0, 0.0);
  return sin (x) + x * 0.091325;
}
...
int compteur = 0;
Resolution2Iterateur iter (SinusCroissant, (void *) &compteur,
                           -12.0, 12.0, 10, 1.0e-4, 1.0e-4);
double z;
while ((z = iter.zeroSuivant ()) < 13.0)
  cout << z << :  << iter.evalueFonction (z).f0 () << endl;
cout << compteur <<  " appels à la fonction f\n";
...
```

conseils d'utilisation spécifiques

La méthode utilisée pour parcourir les zéros d'une fonction sur un intervalle donné est la suivante : on commence par chercher des intervalles monotones encadrant des zéros, soit à l'aide du *pas* soit en découpant l'intervalle en n tronçons (*pas* et n étant des paramètres de construction de l'instance). Une fois un intervalle encadrant une racine trouvé, cet intervalle est réduit jusqu'à trouver le zéro.

La phase de séparation peut conduire à rechercher finement certains extremums locaux s'ils sont proches d'un zéro, la méthode permet donc de trouver des zéros dans des cas limites où la fonction ne change de signe que très brièvement, c'est à dire quand deux zéros successifs sont très proches. Elle est limitée par le fait que deux extremums très proches peuvent ne pas être vus, selon la valeur de *pas* (ou de n) utilisée.

Le critère d'arrêt porte sur les bornes d'un intervalle encadrant la racine, à la fois en abscisse (longueur de l'intervalle) et en ordonnée (valeur maximale atteinte aux bornes). Dès que les bornes vérifient l'un des seuils *convergenceX* ou *convergenceY* la réduction de l'intervalle est stoppée et le *meilleur* point est retourné. Si l'un des critères est négatif, il ne sera jamais respecté et c'est donc l'autre qui sera déterminant.

Une description détaillée des algorithmes se trouve dans [DR1].

implantation

Les attributs privés sont décrits sommairement dans la table 45, il n'y a pas d'attribut protégé.

TAB. 45: attributs privés de la classe Resolution2Iterateur

nom	type	description
convergenceX_	double	seuil de convergence en x (s'il est négatif, on ne convergera jamais en x)
convergenceY_	double	seuil de convergence en y (s'il est négatif, on ne convergera jamais en y)
tMin_	double	borne inférieure de l'intervalle de recherche des zéros
tMax_	double	borne supérieure de l'intervalle de recherche des zéros
pas_	double	pas de dichotomie pour la recherche des extremums
min_	Variation2	minimum courant (ce n'est en fait pas forcément un minimum, mais c'est un point suffisamment bas pour être utilisable dans la recherche des zéros)
max_	Variation2	maximum courant (ce n'est en fait pas forcément un maximum, mais c'est un point suffisamment haut pour être utilisable dans la recherche des zéros)
precedente_	Variation2	variation de la fonction au point précédent
f_	TypeFoncVD2	pointeur sur la fonction à annuler
arg_	void*	second argument d'appel de la fonction pointée par f_ (le premier argument est le t variant dans l'intervalle [tMin_ ; tMax_])

12.16 classe Rotation

description

La classe Rotation<T> implante les rotations dans un espace de dimension 3 et propose toutes les opérations mathématiques permettant de les manipuler (construction à partir d'éléments variés, application à des vecteurs, compositions...).

interface publique

Il est recommandé de ne pas inclure directement le fichier de déclaration de la classe (voir la section 13), mais de passer plutôt par l'une des directives :

```
#include "cantor/DeclDBL.h"
#include "cantor/DeclVD1.h"
#include "cantor/DeclVD2.h"
```

Ces fichiers réalisent l'inclusion du fichier de déclaration et définissent les types RotDBL en tant qu'alias de Rotation<double>, RotVD1 en tant qu'alias de RotVD1<ValeurDerivee1>, et RotVD2 en tant qu'alias de RotVD2<ValeurDerivee2>.

```
#include "cantor/Rotation.h"
```

La classe Rotation définit un type énuméré AxesRotation qui permet de spécifier quelle convention l'utilisateur désire utiliser pour spécifier la succession des trois rotations élémentaires permettant définir des rotations à l'aides d'angles de CARDAN ou d'EULER. On rappelle que les angles de CARDAN correspondent à trois rotations autour d'axes tous différents alors que les angles d'EULER correspondent à trois rotation autour de deux axes différents (le même axe étant utilisé pour la première et la dernière rotation). Une confusion courante, en particulier dans les domaines de l'aéronautique et du contrôle d'attitude, est d'utiliser le terme angle d'EULER pour parler des angles de roulis, tangage et lacet alors qu'il s'agit d'angles de CARDAN. L'expérience montre qu'en ce qui concerne les angles de CARDAN de nombreuses conventions différentes sont utilisées parmi les six possibles[10], alors que pour les angles d'EULER, la convention ZXZ est pratiquement la seule qui soit utilisée parmi les six possibles.

Le type énuméré défini dans la classe Rotation est utilisé en argument du constructeur par trois angles élémentaires et dans la méthode **initAngles** d'extraction des angles élémentaires d'une rotation existante.

Les constantes de ce type énuméré sont conformes à la déclaration suivante, les six premières constantes correspondant à des angles de CARDAN et les six autres correspondant à des angles d'EULER.

```
enum AxesRotation { XYZ, XZY, YXZ, YZX, ZXY, ZYX,
                    XYX, XZX, YXY, YZY, ZXZ, ZYZ };
```

TAB. 46: Rotation : méthodes publiques

signature	description
Rotation ()	construit la rotation identité
	à suivre ...

[10] on a même rencontré l'utilisation simultanée de trois conventions différentes pour un même projet

TAB. 46: Rotation : méthodes publiques (suite)

signature	description
Rotation (const Vecteur3<T>& *axe*, const T& *angle*, T *norme* = T (-1.0)) **throw**(CantorErreurs)	construit une rotation d'*axe* et d'*angle* donnés, c'est à dire que si l'on construit r à partir de l'axe $\vec{z}$ et de l'angle $\pi/2$, alors l'image de $\vec{x}$ est $\vec{y}$, l'image de $\vec{y}$ est $-\vec{x}$ et l'image de $\vec{z}$ est $\vec{z}$ lui-même. Si la *norme* n'est pas passée en argument (c'est à dire si la valeur reçue est négative), le constructeur la recalcule lui-même
Rotation (AxesRotation *ordre*, const T& *alpha1*, const T& *alpha2*, const T& *alpha3*) **throw**(CantorErreurs)	construit une rotation en appliquant successivement trois rotations élémentaires d'angles *alpha1*, *alpha2* et *alpha3* autour des axes spécifiés par l'*ordre*. *Attention*, les rotations finales sont réalisées en tenant compte de celles qui précèdent et qui ont modifié les axes. À titre d'exemple, si on tourne d'abord de 45 degrés autour de $\vec{x}$ puis de 15 degrés autour de $\vec{z}$, la seconde rotation est faite autour d'un axe qui n'est pas l'axe $\vec{z}$ original mais un axe situé à mi-chemin entre les axes $\vec{z}$ et $-\vec{y}$ originaux.
Rotation (T matrice [3][3], T *convergence* = T (1.0e-6)) **throw**(CantorErreurs)	construit une rotation à partir de sa *matrice* (c'est à dire construit la rotation qui transforme $\vec{u}$ en $\vec{v}$ si $(\vec{v}) = M \times (\vec{u})$). La *matrice* peut ne pas être unitaire, le constructeur se chargeant de trouver la matrice unitaire la plus proche au sens de la norme de FROBENIUS de la *matrice* passées en argument. Une erreur est générée si la correction dépasse le seuil de *convergence* toléré.
Rotation (Vecteur3<T> *u1*, Vecteur3<T> *u2*, Vecteur3<T> *v1*, Vecteur3<T> *v2*) **throw**(CantorErreurs)	construit une rotation transformant le couple de vecteurs $(\vec{u}_1, \vec{u}_2)$ en $(\vec{v}_1, \vec{v}_2)$. La copie du vecteur $\vec{v}_2$ utilisée dans le constructeur peut être corrigée si les deux couples n'ont pas le même écartement
Rotation (Vecteur3<T> *u*, Vecteur3<T> *v*) **throw**(CantorErreurs)	construit une rotation transformant $\vec{u}$ en $\vec{v}$. Il y a une infinité de telles rotations, l'algorithme de choix utilisé par le constructeur consiste à prendre l'axe orthogonal au plan $(\vec{u}, \vec{v})$, ce qui revient à choisir la rotation d'angle minimal
Rotation (const T& *q0*, const T& *q1*, const T& *q2*, const T& *q3*)	construit une rotation à partir des composantes d'un quaternion. Cette fonction en ligne a été simplifiée au maximum pour améliorer sa rapidité (elle est utilisée pour l'interface C/C++) et n'effectue aucune vérification sur ses arguments, c'est l'appelant qui doit garantir que les quatre composantes forment bien un quaternion normé ($q_0^2 + q_1^2 + q_2^2 + q_3^2 = 1$)
Rotation (const Rotation<T>& *r*)	constructeur par copie
Rotation<T>& **operator** = (const Rotation<T>& *r*)	affectation
const T& **q0** () const	retourne une copie de la première composante (partie scalaire) du quaternion
const T& **q1** () const	retourne une copie de la seconde composante (abscisse de la partie vectorielle) du quaternion
const T& **q2** () const	retourne une copie de la troisième composante (ordonnée de la partie vectorielle) du quaternion
const T& **q3** () const	retourne une copie de la quatrième composante (cote de la partie vectorielle) du quaternion
Vecteur3<T> **axe** () const	retourne une copie de l'axe normé de la rotation (le sens est tel que l'angle associé soit compris entre 0 et π)
T **angle** () const	retourne l'angle entre 0 et π de la rotation

à suivre ...

TAB. 46: Rotation : méthodes publiques (suite)

signature	description
void **initMatrice** (T *m* [3][3]) const	initialise la matrice *m* à partir de l'instance. Cette méthode peut être considérée comme l'inverse du constructeur par matrice.
void **initAngles** (AxesRotation *ordre*, T **ptrAlpha1*, T **ptrAlpha2*, T **ptrAlpha3*) const **throw**(CantorErreurs)	retourne une copie des angles de rotation successifs correspondant à la succession de rotations élémentaires spécifiée par *ordre*. Cette méthode peut être considérée comme l'inverse du constructeur par trois angles.
Rotation<T> **operator ()** (const Rotation<T>& *r*) const	construit la rotation composée de *r* suivie de l'instance
Vecteur3<T> **operator ()** (const Vecteur3<T>& *u*) const	retourne l'image de $\vec{u}$ par l'instance
Rotation ()	destructeur.

TAB. 47: Rotation : fonctions non membres

signature	description
Rotation<T> **operator -** (const Rotation<T>& *r*)	construit la rotation réciproque de *r*
void **corrigeOrthog** (const T *m* [3][3], T *orth* [3][3], T *convergence*) **throw**(CantorErreurs)	recherche la matrice orthogonale *orth* la plus proche de *m* au sens de la norme de FROBENIUS

exemple d'utilisation

```
#include "cantor/DeclDBL.h"
#include "cantor/Util.h"
...
VecDBL i (1.0, 0.0, 0.0);
VecDBL j (0.0, 1.0, 0.0);
VecDBL k = i ^ j;

RotDBL r (i, j, j, k);
cout << axe: << r.axe ()
     << , angle: << degres (r.angle ())
     << endl;
cout << r (i)= << r (i) << endl;

RotDBL q (VecDBL (3.0, 2.0, 1.0), radians (56.0));
RotDBL rPuisQ q (r);
cout << q (r (i))= << q (r (i)) << endl;
cout << qOr (i)= << rpuisQ (i) << endl;
```

conseils d'utilisation spécifiques

Les rotations sont implantées dans CANTOR sous forme de quaternions, mais il s'agit là d'un détail d'implémentation dont l'utilisateur ne devrait pas se soucier. La classe a été implantée de façon à garder une vision de haut niveau, les rotations étant de simples opérateurs pouvant agir soit sur d'autres rotations soit sur des vecteurs. Les constructeurs définissent implicitement les rotations par leurs effets sur des vecteurs et des couples de vecteurs, ou par analogie avec des opérateurs linéaires (matrices). Le constructeur par composantes de quaternion est une exception dont l'utilisation devrait être limitée à des cas bien précis où l'utilisateur sait ce qu'il fait (par exemple lorsqu'il connaissait déjà les composantes *au sens de* CANTOR par un appel préalable.

C'est ce point de vue d'opérateur qui a conduit à ne pas implanter la notion de multiplication. La multiplication est une opération liée aux matrices, et c'est dans le cadre de l'isomorphisme entre l'espace des rotations et un sous-espace des matrices 3×3 que cette notion se ramène à la composition des rotations, il ne s'agit pas d'une notion canonique. Cette notion peut par ailleurs conduire à des erreurs puisque la multiplication des matrices et la multiplication des quaternions se font en sens inverse lorsqu'on les rapportent à la notion canonique de composition des rotations. La composition quant à elle à l'avantage d'une notation qui conserve la dissymétrie et indique clairement quelle rotation est appliquée en premier et quelle rotation est appliquée en second : on utilise directement la règle de composition $(r_2(r_1))(\vec{u}) = r_2(r_1(\vec{u}))$.

Lorsque l'on parle d'angle et d'axe de la rotation, on adopte la convention qu'ils sont orientés selon le produit vectoriel $\vec{v}\vec{v}'$, où $\vec{v}'$ est l'image de $\vec{v}$ par la rotation. L'orientation naturelle est donc celle de rotation de vecteurs dans un repère fixe, si on veut modéliser une rotation de repère dans un champ de vecteurs fixes, il faut inverser l'angle (ou l'axe). Cette convention peut différer des conventions d'autres ouvrages ou bibliothèques, il est donc prudent de ne pas chercher à utiliser directement les composantes des quaternions et de rester à l'interface de haut niveau où une rotation est un opérateur.

Les constructeurs de Rotation<T> sont très variés car la méthode la plus naturelle pour définir une rotation dépend beaucoup du problème à résoudre. Ces constructeurs sont robustes, ils acceptent des vecteurs non normés, des couples de vecteurs non isomorphes, des matrices non unitaires. Ce choix permet de les utiliser naturellement y compris lorsque les arguments sont peu précis comme par exmple lorsqu'ils sont lus dans des fichiers d'entrée.

Dans le constructeur de rotation à partir d'un axe et d'un angle, la norme n'est calculée par le constructeur que si la valeur passée en argument est négative (c'est le cas de la valeur par défaut). Si l'appelant connait à l'avance cette norme (par exemple quand l'axe est une constante), alors il est recommandé de la passer, ce qui économise des calculs. Dans le constructeur par matrice,

La classe Rotation<T> s'appuie sur le mécanisme de `template` pour s'abstraire du type T des données qui caractérisent les rotations. Ainsi, peuvent être utilisées des rotations dont les coordonnées sont des doubles, des ValeurDerivee1 ou des ValeurDerivee2. Le mécanisme de `template`, s'il offre des avantages indéniables, est parfois trop souple et peut engendrer des erreurs (que penser d'un Rotation<char *> ?). Il est donc recommandé aux utilisateurs d'utiliser les types prédéfinis RotDBL, RotVD1, et RotVD2 définis dans les fichiers `cantor/DeclDBL.h`, `cantor/DeclVD1.h` et `cantor/DeclVD2.h`, et de ne pas instancier lui-même les `templates`. Si l'utilisateur doit utiliser simultanément des double et des ValeurDerivee1, il lui faudra inclure le fichier `cantor/DeclDBLVD1.h`, qui outre inclure les fichiers précédents, définit un certain nombre d'opérateurs de conversions permettant par example de tester l'égalité d'un RotDBL et d'un RotVD1. Les fichiers d'en-tête `cantor/DeclDBLVD2.h` et `cantor/DeclVD1VD2.h` existent aussi avec des rôles symétriques.

implantation

Les attributs privés sont décrits sommairement dans la table 48, il n'y a pas d'attribut protégé.

TAB. 48: attributs privés de la classe Rotation

nom	type	description
q_	Vecteur3<T>	partie vectorielle du quaternion (elle correspond à l'axe de la rotation multiplié par le sinus du demi-angle de la rotation)
q0_	T	partie scalaire du quaternion (elle correspond au cosinus du demi-angle de la rotation

12.17 classe Secteurs

description

Cette classe est une surcouche à la classe Cone (cf 12.6) permettant de filtrer des portions considérées comme visibles du cône. Elle est utilisée par exemple pour modéliser la visibilité du limbe d'un corps central à travers les champs de vue d'un senseur infrarouge dans la bibliothèque MARMOTTES.

interface publique

```
#include "cantor/Secteurs.h"
```

TAB. 49: Secteurs : méthodes publiques

signature	description
Secteurs ()	construit un secteur constitué de la totalité du cône par défaut
Secteurs (const Cone& *base*)	construit un secteur constitué de la totalité du cône *base*
Secteurs (const Cone& *base*, const VecDBL& *reference*, const Creneau& *visible*) **throw**(CantorErreurs)	construit un secteur constitué des portions angulaires *visible* (comptées à partir du vecteur de *reference*) autour du cône *base*
Secteurs (const Secteurs& *s*)	constructeur par copie
Secteurs& **operator** = (const Secteurs& *s*)	affectation
Secteurs ()	destructeur
void **intersection** (const Cone& *c*)	filtre dans le secteur courant les parties visibles à travers le cône *c*
bool **diedreContient** (const VecDBL& *u*) const	teste si le vecteur *u* est dans l'un des dièdres du secteur
	à suivre ...

TAB. 49: Secteurs : méthodes publiques (suite)

signature	description
int **nombre** () const	retourne le nombre de tronçons dont est constitué le secteur
bool **vide** () const	indique si le secteur est vide (c'est à dire ne contient aucun tronçon visible)
bool **nonVide** () const	indique si le secteur n'est pas vide (c'est à dire contient au moins un tronçon visible)
const Cone& **cone** () const	retourne une référence constante sur le cône de base sur lequel s'appuie le secteur
double **angle** () const	retourne le demi angle d'ouverture du cône de base sur lequel s'appuie le secteur
const VecDBL& **reference** () const	retourne une référence constante sur le vecteur de référence à partir duquel sont comptés les angles décrivant les portions visibles
const Creneau& **creneau** () const	retourne une référence constante sur les portions angulaires visibles
void **diedre** (int *i*, VecDBL **ptrAxe*, VecDBL **ptrPlanDebut*, VecDBL **ptrPlanFin*, double **ptrBalayage*) const	initialise les variables pointées par les arguments *ptrAxe*, *ptrPlanDebut*, *ptrPlanFin*, et *ptrBalayage* avec les paramètres du dièdre contenant le tronçon *i*. Il faut noter ques les vecteurs *ptrPlanDebut* et *ptrPlanFin* sont des vecteurs appartenant aux plans limitant le dièdre de l'arc (ils sont orthogonaux à l'axe, quel que soit l'ouverture du cône), ce ne sont *pas* les vecteurs de début et de fin de l'arc lui-même
void **modifieReference** (const VecDBL& *reference*)	ramène la référence de l'instance dans le plan méridien défini par *reference* et par l'axe du cône
void **tourne** (const RotDBL& *r*)	transforme l'instance par la rotation *r*

exemple d'utilisation

```
#include "marmottes/Secteurs.h"

// constitution d'un limbe terre complet
Secteurs limbe (Cone (directionTerre, rayonTerre));

// élimination des portions invisibles
for (int i = 0; i < n; i++)
  limbe.intersection (coneFiltre [i]);

// la direction du limbe observée est-elle visible ?
int ok = 0;
for (int i = 0; i < limbe.nombre (); i++)
{ VecDBL axe, debut, fin;
  double balayage;
  limbe.vecteurs (i, &axe, &debut, &fin, &balayage);
  Arc arc (axe, debut, fin);
  if (arc.diedreContient (directionTest))
    ok = 1;
}
```

conseils d'utilisation spécifiques

L'utilisation de cette classe suit généralement une démarche séquentielle. On commence par créer une instance à partir d'un cône complet, puis on filtre les portions visibles en faisant passer l'instance à travers des successions de cônes, et enfin on parcourt les portions restantes, soit sous forme de créneaux angulaires autour de l'axe du cône de base, soit sous forme d'arcs sur la sphère unité.

Les autres classes permettant de travailler sur le sphère unité sont Arc (cf 12.2), ArcIterateur(cf 12.3), Cone (cf 12.6), Field (cf 12.8), AnnotatedArc (cf 12.1), Node (cf 12.12) et AnnotatedArc (cf 12.1).

implantation

Les attributs privés sont décrits sommairement dans la table 50, il n'y a pas d'attribut protégé.

TAB. 50: attributs privés de la classe Secteurs

nom	type	description
base_	Cone	cône de base sur lequel s'appuie le secteur
x_	VecDBL	vecteur de référence (non normé) à partir duquel sont comptés les angles
y_	VecDBL	vecteur orthogonal à l'axe du cône de base et à x_ et permettant de construire un repère de l'espace (non normé)
zNonNorme_	VecDBL	vecteur non normé parallèle à l'axe du cône de base
visible_	Creneau	portions angulaires visibles, comptées à partir de x_ positivement autour de zNonNorme_

12.18 classe StatistiqueEchantillon

description

Cette classe permet de constituer un échantillon statistique par enrichissement progressif (on peut ajouter des points individuellement ou d'autres échantillons), et d'extraire ses caractéristiques (nombre de points, moyenne, écart type, valeurs extrémales).

interface publique

```
#include "cantor/StatistiqueEchantillon.h"
```

TAB. 51: StatistiqueEchantillon : méthodes publiques

signature	description
StatistiqueEchantillon ()	construit un échantillon vide
StatistiqueEchantillon (const StatistiqueEchantillon& *s*)	constructeur par copie
StatistiqueEchantillon& **operator** = (const StatistiqueEchantillon& *s*)	affectation
StatistiqueEchantillon ()	destructeur
const StatistiqueEchantillon& **operator** += (double *x*)	ajout d'un point dans l'échantillon
const StatistiqueEchantillon& **operator** += (const StatistiqueEchantillon& *s*)	combinaison de deux échantillons
int **nbPoints** () const	retourne le nombre de points de l'échantillon
double **min** () const	retourne la valeur minimale de l'échantillon
double **max** () const	retourne la valeur maximale de l'échantillon
double **moyenne** () const	retourne la valeur moyenne de l'échantillon
double **ecartType** () const	retourne l'écart type de la loi que suivent les points de l'échantillon

exemple d'utilisation

```
#include "cantor/StatistiqueEchantillon.h" ...

StatistiqueEchantillon se;

for (int i = 0; i < n; i++)
  se += essai [i];

cout << "échantillon de " << se.nbPoints () << " points" << endl;
cout << "  min        = " << se.min       () << endl;
cout << "  max        = " << se.max       () << endl;
cout << "  moyenne    = " << se.moyenne   () << endl;
cout << "  écart type = " << se.ecartType () << endl;
```

conseils d'utilisation spécifiques

Il faut prendre garde à l'extrapolation hâtive des formules statistiques. La variance d'une loi aléatoire X vaut $v(X) = E(X^2) - E^2(X)$, et un estimateur non biaisé de l'espérance mathématique sur un échantillon est donné par la formule : $E(X) = \frac{\sum_1^n x_i}{n}$. La combinaison directe de ces formules donne bien un estimateur de la variance, mais il est biaisé : $\tilde{v} = \frac{n\sum x_i^2 - \left(\sum x_i\right)^2}{n^2}$. Si l'on considère que tous les x_i proviennent de tirages indépendants selon une loi unique, l'espérance de l'estimateur vaut en effet : $E(\tilde{v}) = \frac{n-1}{n} v(X)$. L'erreur commise décroît lorsque la taille de l'échantillon croît : elle passe au dessous de 2% (soit 1% sur l'écart type) pour des échantillons de plus de cinquante points. La fonction **ecartType** utilise donc la formulation sans

biais (mais moins intuitive) :

$$\sigma = \sqrt{\frac{n\sum x_i^2 - \left(\sum x_i\right)^2}{n(n-1)}}$$

Cette classe est très simple, et aucune gestion d'erreur sophistiquée n'a été implantée. Les seules circonstances particulières apparaissent lorsque l'on demande les caractéristiques d'un échantillon vide, les méthodes de la classe sont protégées contre ces cas et retournent arbitrairement 0.

implantation

Les attributs privés sont décrits sommairement dans la table 52, il n'y a pas d'attribut protégé.

TAB. 52: attributs privés de la classe StatistiqueEchantillon

nom	type	description
s1_	int	nombre de points de l'échantillon ($\sum_1^n 1$)
min_	double	valeur minimum de l'échantillon
max_	double	valeur maximum de l'échantillon
sX_	double	$\sum_1^n x_i$
sX2_	double	$\sum_1^n x_i^2$

12.19 classe ValeurDerivee1

description

La classe ValeurDerivee1 est une classe de différentiation automatique à l'ordre 1 de fonctions réelles à une variable. Une instance de ValeurDerivee1 représente la valeur d'une fonction en un point, ainsi que la valeur de sa dérivée première au même point. La classe ValeurDerivee1 fournit un ensemble d'opérations permettant de calculer la valeur de toute fonction explicite (et de sa dérivée première) en un point par combinaison des fonctions mathématiques de base.

interface publique

Pour bénéficier à la fois des déclarations de la classe ValeurDerivee1 et des paquetages de conversion et de combinaison d'opérandes qui vont avec, il faut utiliser la directive :

```
#include "cantor/DeclVD1.h"
```

Ce fichier inclut lui-même le fichier déclarant la classe : `"cantor/ValeurDerivee1.h"` .

Outre les méthodes publiques décrites dans la table 53, tous les opérateurs +=, -=, *=, /=, +, -, *, /, <, ≤, >, ≥ combinant des instances de ValeurDerivee1, des instances de ValeurDerivee2 et des réels en double précision sont définis. L'opérateur - unaire est également défini ainsi que les fonctions mathématiques $\sqrt{a}$, $\sin(a)$, $\cos(a)$, $\tan(a)$, $\arcsin(a)$, $\arccos(a)$, $\arctan(a)$, e^a, $\ln(a)$, $\text{atan2}(y,x)$, $\text{pow}(a,b)$, $\text{fabs}(x)$, $\max(a,b)$ et $\min(a,b)$.

TAB. 53: ValeurDerivee1 : méthodes publiques

signature	description
ValeurDerivee1 (double *f0* = 0, double *f1* = 0)	construit une instance à partir de la valeur *f0* et de la dérivée *f1*
ValeurDerivee1 (const ValeurDerivee1& *a*)	constructeur par copie
ValeurDerivee1& **operator** =(const ValeurDerivee1& *a*)	affectation
ValeurDerivee1 ()	destructeur
double **f0** () const	retourne la valeur de la fonction au point considéré
double **f1**() const	retourne la dérivée première de la fonction au point considéré
ValeurDerivee1& **operator** += (const ValeurDerivee1& *a*)	incrémente l'instance de *a* et retourne une référence sur l'instance modifiée
ValeurDerivee1& **operator** -= (const ValeurDerivee1& *a*)	décrémente l'instance de *a* et retourne une référence sur l'instance modifiée
ValeurDerivee1& **operator** *= (const ValeurDerivee1& *a*)	multiplie l'instance par *a* et retourne une référence sur l'instance modifiée
ValeurDerivee1& **operator** /= (const ValeurDerivee1& *a*)	divise l'instance par *a* et retourne une référence sur l'instance modifiée
double **taylor** (double *h*) const	renvoie une estimation de la valeur de la fonction à la distance *h* du point de calcul par la formule de Taylor

TAB. 54: ValeurDerivee1 : fonctions non membres

signature	description
ostream& **operator** << (ostream& *s*, const ValeurDerivee1& *d*)	formate une chaîne de caractères représentant le ValeurDerivee1 *d* sur le flot *s*
ValeurDerivee1 **Approximation** (const ValeurDerivee1& *t*, const ValeurDerivee1& *ta*, const ValeurDerivee1& *fa*, const ValeurDerivee1& *tb*, const ValeurDerivee1& *fb*)	retourne une copie d'une approximation de la valeur en *t* de la fonction *f* prenant en *ta* la valeur *fa* et en *tb* la valeur *fb*, cette approximation est donnée par un polynôme de degré 3 ayant en *ta* et *tb* mêmes valeurs et dérivées que *f*

exemple d'utilisation

```
#include cantor/DeclVD1.h
...
ValeurDerivee1 f (double t)
{ // conversion de la variable en fonction identite
  ValeurDerivee1 vdt (t, 1.0);

  // calcul de la fonction avec ses derivees
  ValeurDerivee1 y  = sin (vdt) - 3.6 * vdt * vdt;
                y /= atan2 (vdt * 2.0, vdt + 17.0 * cos (vdt));
```

```
  // retour a la fonction appelante
  return y;

}
...
for (double t = 0.0; t < 10.0; t += 1.0)
{ ValeurDerivee1 y = f (t);
  cout << t << y.f0 ()<< ' ' << y.f1 () << endl;
}
```

conseils d'utilisation spécifiques

La classe ValeurDerivee1 est destinée à être utilisée exactement comme un réel. Toute variable de type `double` peut être remplacée par un ValeurDerivee1 en ne changeant que sa déclaration (et éventuellement les déclarations des fonctions utilisateurs qui l'utilisent).

Lorsque l'on désire calculer simultanément la valeur et la dérivée d'une fonction f par rapport à une variable x, il suffit de coder f en utilisant des variables de type ValeurDerivee1 pour tous les calculs intermédiaires dépendant de x, et de construire x par :

```
ValeurDerivee1 x (x0, 1.0);
```

où `x0` est le point où l'on désire évaluer la fonction. La variable `x` est donc considérée comme la fonction identité, évaluée en `x0`.

Il est possible de déclarer une classe de calcul `template class <T> class MaClasse { ... }` puis de la spécialiser plusieurs fois :

```
typedef MaClasse<double> MCDBL;
MaClasse<ValeurDerivee1> MCVD1;
MaClasse<ValeurDerivee2> MCVD2;
```

Si l'on a besoin de dérivées d'ordre supérieur, la classe ValeurDerivee2 existe également. Il faut cependant noter que si le calcul d'une dérivée première par la classe ValeurDerivee1 est relativement peu coûteux (trois multiplications et une addition pour calculer $a \times b$), le passage à la dérivée seconde augmente très sensiblement le nombre d'opérations élémentaires (cinq multiplications et quatre additions pour calculer $a \times b$). Il est parfois plus économique d'utiliser des algorithmes plus grossiers n'utilisant que la dérivée première même s'ils doivent évaluer la fonction un grand nombre de fois que des algorithmes utilisant les dérivées première et seconde et n'évaluant la fonction qu'un petit nombre de fois.

implantation

Les attributs privés sont décrits sommairement dans la table 55, il n'y a pas d'attribut protégé.

TAB. 55: attributs privés de la classe ValeurDerivee1

nom	type	description
f1_	double	dérivée première de la fonction
f0_	double	valeur de la fonction

12.20 classe ValeurDerivee2

description

La classe ValeurDerivee2 est une classe de différentiation automatique à l'ordre 2 de fonctions réelles à une variable. Une instance de ValeurDerivee2 représente la valeur d'une fonction en un point, ainsi que la valeur de sa dérivée première et de sa dérivée seconde au même point. La classe ValeurDerivee2 fournit un ensemble d'opérations permettant de calculer la valeur de toute fonction explicite (et de ses dérivée première et seconde) en un point par combinaison des fonctions mathématiques de base.

interface publique

Pour bénéficier à la fois des déclarations de la classe ValeurDerivee2 et des paquetages de conversion et de combinaison d'opérandes qui vont avec, il faut utiliser la directive :

```
#include "cantor/DeclVD2.h"
```

Ce fichier inclut lui-même le fichier déclarant la classe : `"cantor/ValeurDerivee2.h"` .

Outre les méthodes publiques décrites dans la table 56, tous les opérateurs +=, -=, *=, /=, +, -, *, /, <, ≤, >, ≥ combinant des instances de ValeurDerivee2, des instances de ValeurDerivee2 et des réels en double précision sont définis. L'opérateur - unaire est également défini ainsi que les fonctions mathématiques $\sqrt{a}$, $\sin(a)$, $\cos(a)$, $\tan(a)$, $\arcsin(a)$, $\arccos(a)$, $\arctan(a)$, e^a, $\ln(a)$, $\text{atan2}(y,x)$, $\text{pow}(a,b)$, $\text{fabs}(x)$, $\max(a,b)$ et $\min(a,b)$.

TAB. 56: ValeurDerivee2 : méthodes publiques

signature	description
ValeurDerivee2 (double *f0* = 0, double *f1* = 0, double *f2* = 0)	construit une instance à partir de la valeur *f0*, de la dérivée première *f1*, et de la dérivée seconde *f2*
ValeurDerivee2(const ValeurDerivee2& *a*)	constructeur par copie
ValeurDerivee2& **operator** =(const ValeurDerivee2& *a*)	affectation
signature **ValeurDerivee2** ()	destructeur
hline double **f0** () const	retourne la valeur de la fonction au point considéré
double **f1**() const	retourne la dérivée première de la fonction au point considéré
double **f2**() const	retourne la dérivée seconde de la fonction au point considéré
ValeurDerivee2& **operator** += (const ValeurDerivee2& *a*)	incrémente l'instance de *a* et retourne une référence sur l'instance modifiée
ValeurDerivee2& **operator** -= (const ValeurDerivee2& *a*)	décrémente l'instance de *a* et retourne une référence sur l'instance modifiée
ValeurDerivee2& **operator** *= (const ValeurDerivee2& *a*)	multiplie l'instance par *a* et retourne une référence sur l'instance modifiée
ValeurDerivee2& **operator** /= (const ValeurDerivee2& *a*)	divise l'instance par *a* et retourne une référence sur l'instance modifiée

à suivre ...

TAB. 56: ValeurDerivee2 : méthodes publiques (suite)

signature	description
double **taylor** (double *h*) const	renvoie une estimation de la valeur de la fonction à la distance *h* du point de calcul par la formule de Taylor

TAB. 57: ValeurDerivee2 : fonctions non membres

signature	description
ostream& **operator** << (ostream& *s*, const ValeurDerivee2& *d*)	formate une chaîne de caractères représentant le ValeurDerivee2 *d* sur le flot *s*
ValeurDerivee2 **Approximation** (const ValeurDerivee2& *t*, const ValeurDerivee2& *ta*, const ValeurDerivee2& *fa*, const ValeurDerivee2& *tb*, const ValeurDerivee2& *fb*)	retourne une copie d'une approximation de la valeur en *t* de la fonction *f* prenant en *ta* la valeur *fa* et en *tb* la valeur *fb*, cette approximation est donnée par un polynôme de degré 5 ayant en *ta* et *tb* mêmes valeurs et dérivées que *f*

exemple d'utilisation

```
#include cantor/DeclVD2.h
...
ValeurDerivee2 f (double t)
{ // conversion de la variable en fonction identite
  ValeurDerivee2 vdt (t, 1.0, 0.0);

  // calcul de la fonction avec ses derivees
  ValeurDerivee2 y  = sin (vdt) - 3.6 * vdt * vdt;
                 y /= atan2 (vdt * 2.0, vdt + 17.0 * cos (vdt));

  // retour a la fonction appelante
  return y;

}
...
for (double t = 0.0; t < 10.0; t += 1.0)
{ ValeurDerivee2 y = f (t);
  cout << t << y.f0 ()<< ' ' << y.f1 () << endl;
}
```

conseils d'utilisation spécifiques

La classe ValeurDerivee2 est destinée à être utilisée exactement comme un réel. Toute variable de type `double` peut être remplacée par un ValeurDerivee2 en ne changeant que sa déclaration (et éventuellement les déclarations des fonctions utilisateurs qui l'utilisent).

Lorsque l'on désire calculer simultanément la valeur et les dérivées première et seconde d'une fonction f par rapport à une variable x, il suffit de coder f en utilisant des variables de type ValeurDerivee2 pour tous les calculs intermédiaires dépendant de x, et de construire x par :

```
ValeurDerivee2 x (x0, 1.0, 0.0);
```

où `x0` est le point où l'on désire évaluer la fonction. La variable `x` est donc considérée comme la fonction identité, évaluée en `x0`.

Il est possible de déclarer une classe de calcul `template class <T> class MaClasse { ... }` puis de la spécialiser plusieurs fois :

```
typedef MaClasse<double> MCDBL;
MaClasse<ValeurDerivee2> MCVD1;
MaClasse<ValeurDerivee2> MCVD2;
```

Si l'on a besoin de dérivées d'ordre inférieur, la classe ValeurDerivee1 existe également. Il faut cependant noter que si le calcul d'une dérivée première par la classe ValeurDerivee1 est relativement peu coûteux (trois multiplications et une addition pour calculer $a \times b$), le passage à la dérivée seconde augmente très sensiblement le nombre d'opérations élémentaires (cinq multiplications et quatre additions pour calculer $a \times b$). Il est parfois plus économique d'utiliser des algorithmes plus grossiers n'utilisant que la dérivée première même s'ils doivent évaluer la fonction un grand nombre de fois que des algorithmes utilisant les dérivées première et seconde et n'évaluant la fonction qu'un petit nombre de fois.

implantation

Les attributs privés sont décrits sommairement dans la table 58, il n'y a pas d'attribut protégé.

TAB. 58: attributs privés de la classe ValeurDerivee2

nom	type	description
f2_	double	dérivée seconde de la fonction
f1_	double	dérivée première de la fonction
f0_	double	valeur de la fonction

12.21 classe Variation1

description

La classe Variation1 modélise la variation d'une fonction scalaire en un point (au sens d'un élément de tableau de variation). Cette classe est utilisée de façon interne par la classe Resolution1Iterateur.

interface publique

```
#include "cantor/Variation1.h"
```

TAB. 59: Variation1 : méthodes publiques

signature	description
Variation1 ()	construit la variation d'une fonction à valeur et dérivée nulle en 0
Variation1 (double x, const ValeurDerivee1& y)	construit la variation de la fonction valant y en x
Variation1 (const Variation1& v)	constructeur par copie
Variation1& **operator** = (const Variation1& v)	affectation
Variation1 ()	destructeur
int **sens** () const	retourne une copie du sens de variation (positif si la fonction est croissante, négatif sinon)
int **croissante** () const	indique si la fonction est croissante
int **decroissante** () const	indique si la fonction est décroissante
int **sensValide** () const	indique si le sens de variation est cohérent avec le signe de la dérivée (le sens peut être simulé par les fonctions membres **simuleCroissante** et **simuleDecroissante**)
double **x** () const	retourne une copie du point de calcul de la fonction
const ValeurDerivee1& **y**() const	retourne une référence constante sur la valeur de la fonction au point de calcul
void **reinitialise** (double x, const ValeurDerivee1& y)	remplace l'instance par la variation de la fonction valant y en x
void **simuleCroissante** ()	modifie le sens de la variation pour simuler une fonction croissante (on peut reconnaître une variation simulée à l'aide de la fonction **sensValide**). Cette fonction facilite l'utilisation des Variation1 dans des algorithmes utilisant des extremums, pour lesquels la convergence peut aboutir à un point croissant ou décroissant de façon aléatoire
void **simuleDecroissante** ()	modifie le sens de la variation pour simuler une fonction décroissante (on peut reconnaître une variation simulée à l'aide de la fonction **sensValide**). Cette fonction facilite l'utilisation des Variation1 dans des algorithmes utilisant des extremums, pour lesquels la convergence peut aboutir à un point croissant ou décroissant de façon aléatoire

exemple d'utilisation

```
#include "cantor/Variation1.h"

Variation1 NewtonSecante (TypeFoncVD1   f,           void* arg,
                          TypeFoncConv1 convergence, void* arg_conv,
                          const Variation1& a, const Variation1& b)
{ // remise en ordre des bornes
  Variation1 inf = (a.x () < b.x ()) ? a : b;
  Variation1 sup = (a.x () < b.x ()) ? b : a;
  int        last_inf;

  // correction du sens de variation donne par les derivees
  // (si a et b sont des extremums, le signe des derivees n'est pas fiable)
```

```
if (inf.y () <= sup.y ())
{ inf.simuleCroissante   ();
  sup.simuleCroissante   ();
}
else
{ inf.simuleDecroissante ();
  sup.simuleDecroissante ();
}

...

do
{ double t;

  ...

  // recherche d'un nouveau t par la methode de Newton
  if (abs (sup.y ().f0 ()) >= abs (inf.y ().f0 ()))
  { // c'est la borne inf qui est la plus proche
    double dt = - inf.y ().f0 () / inf.y ().f1 ();

    ...

    // calcul du nouveau point de test
    t = inf.x () + dt;

  }
  else
    ...

  // calcul de la fonction pour le nouveau t
  Variation1 b (t, (*f) (t, arg));

  // mise a jour des bornes de l'intervalle encadrant la racine
  if (inf.croissante ())
    b.simuleCroissante   ();
  else
    b.simuleDecroissante ();
  if (((b.y () <= 0.0) && (inf.croissante   ()))
   || ((b.y () >  0.0) && (inf.decroissante ())))
  { inf      = b;
    last_inf = 1;
  }
  else
  { sup      = b;
    last_inf = 0;
  }

  // test de la convergence de l'algorithme
  code = (*convergence) (inf, sup, arg_conv);
```

```
  } while ((code == CONV1_AUCUNE)
           &&
           ((sup.x () - inf.x ()) > eps_inf + eps_sup));

  // retour de la meilleure estimee du zero
  ...

}
```

conseils d'utilisation spécifiques

Cette classe est essentiellement destinée à être utilisée dans des algorithmes de recherche de zéros ou d'extremums de fonctions à une variable[11]. Il faut prendre garde dans ce type d'algorithme que les points de calculs résultent parfois de critères de convergence variés, il faut donc étudier minutieusement le comportement de l'algorithme dans les cas limites. À titre d'exemple, on peut citer le cas d'un partitionnement d'intervalle en un tronçon croissant et un tronçon décroissant ; ceci conduit à voir le point intermédiaire dans le premier sous-intervalle comme le dernier point décroissant et dans l'autre comme le premier point croissant. Les méthodes **simuleDecroissante** et **simuleDecroissante** peuvent être utilisées pour éviter les phénomènes de bord dans ce cas.

implantation

Les attributs privés sont décrits sommairement dans la table 60, il n'y a pas d'attribut protégé.

TAB. 60: attributs privés de la classe Variation1

nom	type	description
sens_	int	sens de variation : $+1 \Rightarrow$ croissante, $-1 \Rightarrow$ décroissante
sensValide_	int	indicateur de cohérence entre sens_ et le signe de la dérivée
x_	double	abscisse à laquelle est calculée la fonction
y_	ValeurDerivee1	valeur de la fonction en x_

12.22 classe Variation2

description

La classe Variation2 modélise la variation d'une fonction scalaire en un point (au sens d'un élément de tableau de variation). Cette classe est utilisée de façon interne par la classe Resolution1Iterateur.

interface publique

```
#include "cantor/Variation2.h"
```

[11] elle a été créée pour les besoins internes de CANTOR

TAB. 61: Variation2 : méthodes publiques

signature	description
Variation2 ()	construit la variation d'une fonction à valeur et dérivée nulle en 0
Variation2 (double x, const ValeurDerivee1& y)	construit la variation de la fonction valant y en x
Variation2 (const Variation2& v)	constructeur par copie
Variation2& **operator** = (const Variation2& v)	affectation
Variation2 ()	destructeur
int **sens** () const	retourne une copie du sens de variation (positif si la fonction est croissante, négatif sinon)
int **croissante** () const	indique si la fonction est croissante
int **decroissante** () const	indique si la fonction est décroissante
int **sensValide** () const	indique si le sens de variation est cohérent avec le signe de la dérivée (le sens peut être simulé par les fonctions membres **simuleCroissante** et **simuleDecroissante**)
double **x** () const	retourne une copie du point de calcul de la fonction
const ValeurDerivee1& **y**() const	retourne une référence constante sur la valeur de la fonction au point de calcul
void **reinitialise** (double x, const ValeurDerivee1& y)	remplace l'instance par la variation de la fonction valant y en x
void **simuleCroissante** ()	modifie le sens de la variation pour simuler une fonction croissante (on peut reconnaître une variation simulée à l'aide de la fonction **sensValide**). Cette fonction facilite l'utilisation des Variation2 dans des algorithmes utilisant des extremums, pour lesquels la convergence peut aboutir à un point croissant ou décroissant de façon aléatoire
void **simuleDecroissante** ()	modifie le sens de la variation pour simuler une fonction décroissante (on peut reconnaître une variation simulée à l'aide de la fonction **sensValide**). Cette fonction facilite l'utilisation des Variation2 dans des algorithmes utilisant des extremums, pour lesquels la convergence peut aboutir à un point croissant ou décroissant de façon aléatoire

exemple d'utilisation

```
#include "cantor/Variation2.h"

Variation2 NewtonSecante (TypeFoncVD2   f,           void* arg,
                          TypeFoncConv2 convergence, void* arg_conv,
                          const Variation2& a, const Variation2& b)
{ // remise en ordre des bornes
  Variation2 inf = (a.x () < b.x ()) ? a : b;
  Variation2 sup = (a.x () < b.x ()) ? b : a;
  int        last_inf;

  // correction du sens de variation donne par les derivees
  // (si a et b sont des extremums, le signe des derivees n'est pas fiable)
```

```
if (inf.y () <= sup.y ())
{ inf.simuleCroissante   ();
  sup.simuleCroissante   ();
}
else
{ inf.simuleDecroissante ();
  sup.simuleDecroissante ();
}

...

do
{ double t;

  ...

  // recherche d'un nouveau t par la methode de Newton améliorée
  if (abs (sup.y ().f0 ()) >= abs (inf.y ().f0 ()))
  { // c'est la borne inf qui est la plus proche
    double dt = - (2.0 * inf.y ().f0 () * inf.y ().f1 ())
                / (2.0 * inf.y ().f1 () * inf.y ().f1 ()
                       - inf.y ().f0 () * inf.y ().f2 ());

    ...

    // calcul du nouveau point de test
    t = inf.x () + dt;

  }
  else
    ...

  // calcul de la fonction pour le nouveau t
  Variation2 b (t, (*f) (t, arg));

  // mise a jour des bornes de l'intervalle encadrant la racine
  if (inf.croissante ())
    b.simuleCroissante   ();
  else
    b.simuleDecroissante ();
  if (((b.y () <= 0.0) && (inf.croissante   ()))
   || ((b.y () >  0.0) && (inf.decroissante ())))
  { inf      = b;
    last_inf = 1;
  }
  else
  { sup      = b;
    last_inf = 0;
  }
```

```
    // test de la convergence de l'algorithme
    code = (*convergence) (inf, sup, arg_conv);

  } while ((code == CONV2_AUCUNE)
           &&
           ((sup.x () - inf.x ()) > eps_inf + eps_sup));

  // retour de la meilleure estimee du zero
  ...

}
```

conseils d'utilisation spécifiques

Cette classe est essentiellement destinée à être utilisée dans des algorithmes de recherche de zéros ou d'extremums de fonctions à une variable[12]. Il faut prendre garde dans ce type d'algorithme que les points de calculs résultent parfois de critères de convergence variés, il faut donc étudier minutieusement le comportement de l'algorithme dans les cas limites. À titre d'exemple, on peut citer le cas d'un partitionnement d'intervalle en un tronçon croissant et un tronçon décroissant ; ceci conduit à voir le point intermédiaire dans le premier sous-intervalle comme le dernier point décroissant et dans l'autre comme le premier point croissant. Les méthodes **simuleDecroissante** et **simuleDecroissante** peuvent être utilisées pour éviter les phénomènes de bord dans ce cas.

implantation

Les attributs privés sont décrits sommairement dans la table 62, il n'y a pas d'attribut protégé.

TAB. 62: attributs privés de la classe Variation2

nom	type	description
sens_	int	sens de variation : $+1 \Rightarrow$ croissante, $-1 \Rightarrow$ décroissante
sensValide_	int	indicateur de cohérence entre sens_ et le signe de la dérivée
x_	double	abscisse à laquelle est calculée la fonction
y_	ValeurDerivee1	valeur de la fonction en x_

12.23 classe Vecteur3

description

La classe Vecteur3<T> implante les vecteurs dans un espace de dimension 3 et propose toutes les opérations mathématiques permettant de les manipuler (additions, produits scalaire et vectoriel...).

[12] elle a été créée pour les besoins internes de CANTOR

interface publique

Il est recommandé de ne pas inclure directement le fichier de déclaration de la classe (voir la section 12.23), mais de passer plutôt par l'une des directives :

```
#include "cantor/DeclDBL.h"
#include "cantor/DeclVD1.h"
#include "cantor/DeclVD2.h"
```

Ces fichiers réalisent l'inclusion du fichier de déclaration et définissent les types VecDBL en tant qu'alias de Vecteur3<double>, VecVD1 en tant qu'alias de Vecteur3<ValeurDerivee1>, et VecVD2 en tant qu'alias de Vecteur3<ValeurDerivee2>.

Outre les méthodes publiques décrites dans la table 63 et les fonctions décrites dans la table 64, tous les opérateurs +=, -=, ^=, *=, /=, +, -, ^, |, *, /, combinant des instances de Vecteur3<T> et des scalaires de type T sont définis. L'opérateur - unaire est également défini. Il faut noter que pour éviter des erreurs d'interprétation, ni le produit scalaire ni le produit vectoriel n'utilisent le symbole *, celui-ci est réservé à la multiplication d'un vecteur par un scalaire. Le produit vectoriel utilise le symbole ^, tandis que le produit scalaire utilise le symbole | (par homogénéité avec la notation $< x|y >$ rencontrée dans certains ouvrages d'algèbre linéaire).

TAB. 63: Vecteur3 : méthodes publiques

signature	description
Vecteur3 ()	construit un vecteur nul
Vecteur3 (T *alpha*, T *delta*)	construit un vecteur normé à partir de ses coordonnées sphériques
Vecteur3 (T *x*, T *y*, T *z*)	construit un vecteur à partir de ses coordonnées
Vecteur3 (const Vecteur3<T>& *u*)	constructeur par copie
Vecteur3<T>& **operator** = (const Vecteur3<T>& *u*)	affectation
const T& **x** () const	retourne une référence constante sur l'abscisse
const T& **y** () const	retourne une référence constante sur l'ordonnéé
const T& **z** () const	retourne une référence constante sur la cote
T **norme** () const	retourne la norme
T **alpha** () const	retourne l'azimut entre $-\pi$ et $+\pi$
T **delta** () const	retourne l'élévation entre $-\pi/2$ et $\pi/2$
int **estNorme** () const	indique si le vecteur est normé (avec une marge de l'ordre de la constante symbolique `cantorEpsilon`)
int **estNul** () const	indique si le vecteur est nul (avec une marge de l'ordre de la constante symbolique `cantorEpsilon`)
void **normalise** () **throw**(CantorErreurs)	normalise l'instance (une erreur est générée si l'instance est nulle)
void **orthonormalise** () **throw**(CantorErreurs)	remplace l'instance par un vecteur normé qui lui est orthogonal (une erreur est générée si l'instance est nulle)
T **angleAvecVecteur** (const Vecteur3<T>& *u*) const	retourne l'angle entre l'instance et le vecteur $\vec{u}$
Vecteur3<T> **orthogonal** () const **throw**(CantorErreurs)	retourne un vecteur orthogonal à l'instance (une erreur est générée si l'instance est nulle)

à suivre ...

TAB. 63: Vecteur3 : méthodes publiques (suite)

signature	description
Vecteur3 ()	destructeur

TAB. 64: Vecteur3 : fonctions non membres

signature	description
T **angle** (const Vecteur3<T> & u, const Vecteur3<T>& v)	retourne l'angle entre les vecteurs $\vec{u}$ et $\vec{u}$
Vecteur3<T> **vecteurOrthogonalNorme** (const Vecteur3<T>& u) **throw**(CantorErreurs)	retourne un vecteur normé orthogonal à $\vec{u}$
Vecteur3<T> **vecteurNorme** (const Vecteur3<T>& u) **throw**(CantorErreurs)	retourne un vecteur normé colinéaire à $\vec{u}$
inline ostream& **operator** << (ostream& s, const Vecteur3<T>& u)	formate une chaîne de caractères représentant le vecteur $\vec{u}$ sur le flot s

exemple d'utilisation

```
#include "cantor/DeclDBL.h"
#include "cantor/Util.h"

...

VecDBL u (1.0, 0.0, 0.0);
VecDBL v (radians (90.0), radians (45.0));
VecDBL w = u ^ v;
cout << angle (u,v): << degres (angle (u, v)) << endl;
cout << u << '^' << v << '=' << w;
cout << "u + v/||v|| - 3 w";
v.normalise ();
cout << u + v - 3 * w << endl;

...
```

conseils d'utilisation spécifiques

La classe Vecteur3<T> s'appuie sur le mécanisme de `template` pour s'abstraire du type T des coordonnées des vecteurs. Ainsi, dans MARMOTTES, sont utilisés des Vecteur3 dont les coordonnés sont représentées par des doubles, des ValeurDerivee1 ou encore des ValeurDerivee2. Le mécanisme de `template`, s'il offre des avantages indéniables, est parfois trop souple et peut engendrer des erreurs (que penser d'un Vecteur3<char *> ?). Il est donc recommandé aux utilisateurs d'utiliser les types prédéfinis VecDBL, VecVD1, et VecVD2 définis dans les fichiers `cantor/DeclDBL.h` , `cantor/DeclVD1.h` et `cantor/DeclVD2.h` , et de ne pas instancier

eux-mêmes les `templates`. Si les utilisateurs doivent utiliser simultanément des double et des ValeurDerivee1, il leur faudra inclure le fichier `cantor/DeclDBLVD1.h` , qui outre inclure les fichiers précédents, définit également un certain nombre d'opérateurs de conversions permettant par example d'ajouter un VecDBL à un VecVD1. Les fichiers `cantor/DeclDBLVD2.h` et `cantor/DeclVD1VD2.h` existent aussi avec des rôles symétriques.

implantation

Les attributs privés sont décrits sommairement dans la table 65, il n'y a pas d'attribut protégé.

TAB. 65: attributs privés de la classe Vecteur3

nom	type	description
x_	T	abscisse
y_	T	ordonnée
z_	T	cote

12.24 paquetages de conversion d'opérandes

L'utilisation des mécanismes de `template`, si elle offre une grande généralité pour la définition des classes Vecteur3 , Rotation et Polynome, tend à rendre délicate la pleine maîtrise d'un code.

En effet, il est *a priori* possible de définir des Vecteur3 des Rotation et des Polynome dont les éléments sont quelconques. Par exemple, on pourrait fort bien définir des vecteurs de chaînes de caractères !

Bien qu'elles ne soient jamais explicitées dans le code, il existe cependant des restrictions à l'extension des classes Vecteur3, Rotation et Polynome : en effet, le code des classes `template` utilise les opérateurs arithmétiques de base, qui doivent absolument exister pour le type T[13]. Cette restriction est formalisée par la règle suivante :

> tout type ou classe T dont on souhaite pouvoir manipuler des Vecteur3, des Rotation, ou des Polynome doit être doté des opérateurs arithmétiques et logiques de base du type réel : +, -, *, /, +=, -=, *=, /=, <, <=, >, >= et ==

A côté de cela, étant donnée la nature scalaire des éléments de base des Vecteur3, Rotation, et Polynome, il est nécessaire d'offrir un large éventail de possibilités permettant une manipulation conjointe non seulement de ces éléments de base, mais aussi de vecteurs de rotations et de polynômes construits à partir de ces éléments de base. C'est le rôle des paquetages de conversion et des paquetages de combinaison d'opérandes. La définition d'un paquetage de conversion est formalisée ainsi :

> on appelle paquetage de conversion des types T et T', un fichier contentant les six opérations suivantes : opération de conversion d'un objet de type T en un objet de type T' (et opération inverse), opération de conversion d'un objet de type Vecteur3<T> en un objet de type Vecteur3<T'> (et opération inverse), opération de conversion d'un objet de type Rotation<T> en un objet de type Rotation<T'> (et opération inverse), opération de conversion d'un objet de type Polynome<T> en un objet de type Polynome<T'> (et opération inverse)

[13] l'absence d'un opérateur engendrerait une erreur de compilation

L'existence de ces opérations de conversion garantit que l'on ne sera jamais limité, au cours d'une utilisation de la bibliothèque CANTOR par l'incompatibilité d'opérandes de types différents au cours de manipulations d'objets, que ceux-ci appartiennent aux types de base, ou soient des vecteurs, des rotations ou des polynômes dérivés pour ces types de base.Si les paquetages de conversion existent, on pourra en effet toujours effectuer les conversions.

Les conversions multiples peuvent être coûteuses en temps de calcul, alors que dans certains cas, des calculs directs avec des types hétérogènes sont possibles. Pour ces cas simples, l'utilisation des paquetages de combinaisons d'opérandes (décrits plus loin) permettra de se dispenser d'invoquer les opérateurs des paquetages de conversion.

Si l'utilisateur de la bibliothèque CANTOR a besoin d'introduire un nouveau type de scalaire, il devra, après s'être assuré que ce type dispose des opérateurs base nécessaires, écrire autant de paquetages de conversion qu'il existe préalablement de types de scalaires dans la bibliothèque.

La version 3.13 de la bibliothèque CANTOR comprend ainsi trois types de paquetages de conversion : DBLVD1 (conversions entre double et ValeurDerivee1), DBLVD2 (conversions entre double et ValeurDerivee2) et VD1VD2 (conversions entre ValeurDerivee1 et ValeurDerivee2).

Si un utilisateur désire introduire une nouvelle classe ayant une sémantique de scalaire, par exemple une valeur associée à une incertitude (appelons là ici ValeurIncertaine), il devra écrire trois paquetages de conversion : DBLVIN (conversions entre double et ValeurIncertaine), VD1VIN (conversions entre ValeurDerivee1 et ValeurIncertaine) et VD2VIN (conversions entre ValeurDerivee2 et ValeurIncertaine). Cette action est formalisée par la procédure suivante :

> pour introduire un nouveau type de scalaire T dans la bibliothèque CANTOR, l'utilisateur doit définir des paquetages de conversion entre le type T (après s'être assuré qu'il dispose des opérateurs de base cités plus haut) et chacun des types scalaires déjà présents dans CANTOR

La définition d'un paquetage de conversion peut se faire de facon assez systématique, par exemple en copiant un paquetage de conversion déjà existant, en y remplacant toutes les occurrences d'un type par un autre, en rajoutant celles qui manquent, puis en écrivant le corps des fonctions du paquetage (qui sont en général limitées à quelques lignes).

L'intérêt de cette procédure est que l'on n'est pas obligé de toucher aux paquetages de conversion existant lorsque l'on introduit un nouveau type de scalaire.

Si en revanche, l'utilisateur souhaite introduire un nouveau type de classe `template` (par exemple, Nappe<T>), il faudra modifier tous les paquetages de conversion pour y inclure les conversions de nappes. Ces modifications seront de même nature pour tous les paquetages.

De par leur nature orthogonale au reste du code, tous les paquetages de conversion sont regroupés dans le sous-répertoire `cantor/conversions` des fichiers d'en-tête.

12.25 paquetages de combinaison d'opérandes

La notion de paquetage de combinaison d'opérandes complète celle de paquetage de conversion pour offrir une grande souplesse dans l'utilisation combinée des différents types de scalaires proposés par la bibliothèque CANTOR. De ce fait, il est fortement conseillé de ne lire cette section qu'après une lecture approfondie de la section qui se rapporte aux paquetages de conversion (12.24).

La définition d'un paquetage de combinaison d'opérande est formalisée ainsi :

> on appelle paquetage de combinaison d'opérandes associé à la classe C pour les types T et T', un fichier contenant toutes les définitions qui permettent de combiner, lors de l'appel d'opérations arithmétiques et logiques de base, des opérandes des types C<T> et C<T'>. Cette définition est étendue aux scalaires par un abus de langage, puisque dans la réalité le type double n'est pas une classe, et que les scalaires sont représentés par les types T et T' et non par des types double<T> et double<T'>.

On notera également que la définition précédente limite les combinaisons d'opérandes à deux types scalaires de base. Bien que la combinaison d'opérandes soit théoriquement possible pour plus de trois types de base, la complexité de telles pratiques risque fort d'être prohibitive. De fait, en utilisant de façon explicite les opérations des paquetages de conversion, on pourra toujours se ramener au cas des combinaisons d'opérandes à deux types de base.

Le principe général de conversion est le suivant : si le type B est une restriction du type A (par exemple A est un ValeurDerivee1 et B est un double), alors une opération entre un A et un B donnera un A, le B ayant été complété par des 0. Ceci n'est bien sûr pas vrai pour une opération du type B += A ; dans ce cas on restreint le type A pour réaliser l'opération. Sous ces mêmes hypothèses, une comparaison entre un A et un B (<, <=, ...) sera réalisée entre une restriction du A et le B original. Dans la pratique, les calcul n'est pas fait en créant un A avec des 0, mais par des formules adaptées à chaque cas, pour des raisons d'efficacité. Ce principe est l'extension naturelle des formules donnant la dérivée d'une opération arithmétique entre une dérivée et un réel, obtenue en considérant le réel comme une fonction constante.

Si un utilisateur désire introduire une classe de scalaires munis d'une valeur incertitude (appelons là ici ValeurIncertaine), il devra écrire trois paquetages de combinaison d'opérandes : DBLVIN (combinaisons entre double et ValeurIncertaine), VD1VIN (combinaisons entre ValeurDerivee1 et ValeurIncertaine) et VD2VIN (combinaisons entre ValeurDerivee2 et ValeurIncertaine). Cette action est formalisée par la procédure suivante :

> pour introduire un nouveau type de scalaire T dans la bibliothèque CANTOR, l'utilisateur doit définir un paquetage de conversion selon la prcédure décrite à la section 12.24, puis il doit définir des paquetages de combinaison d'opérandes associés à chacune des classes `template` pour le type T et chacun des types déjà présents dans bibliothèque CANTOR

La définition des paquetages de combinaison peut ainsi se faire de facon assez systématique, par exemple en copiant un paquetage de combinaison déjà existant, en y remplacant toutes les occurrences d'un type par un autre, en rajoutant celles qui manquent, et en écrivant le corps des fonctions du paquetage (qui sont en général limitées à quelques lignes).

De par leur nature orthogonale au reste du code, tous les paquetages de combinaison sont regroupés dans le sous-répertoire `cantor/conversions` des fichiers d'en-tête.

13 description des utilitaires

Cette section décrit les utilitaires fournis par la bibliothèque CANTOR. Cette description concerne l'utilisation des utilitaires et les principes généraux auxquels ils répondent.

13.1 cantor-config

description générale

L'utilitaire `cantor-config` permet de déterminer les options nécessaires à la compilation d'applicatifs ou de bibliothèques s'appuyant sur CANTOR. Il est principalement destiné à être utilisé dans les règles des fichiers de directives du type `Makefile`.

ligne de commande et options

La ligne de commande a la forme suivante :

```
cantor-config [--cppflags | --ldflags | --libs | --version]
```

Si aucune des options `--cppflags`, `--ldflags`, `--libs` ou `--version` n'est utilisée, l'utilitaire indique les options disponibles et s'arrête avec un code d'erreur.

descriptions des sorties

Les sorties de l'utilitaire dépendent des options sélectionnées dans la ligne de commande.

L'option `--cppflags` permet d'obtenir sur la sortie standard les options de compilation, comme dans l'exemple suivant :

```
(lehrin) luc% cantor-config --cppflags
 -I/home/luc/include
(lehrin) luc%
```

L'option `--ldflags` permet d'obtenir sur la sortie standard les options d'édition de liens, comme dans l'exemple suivant :

```
(lehrin) luc% cantor-config --ldflags
 -L/home/luc/lib
(lehrin) luc%
```

L'option `--libs` permet d'obtenir sur la sortie standard les bibliothèques nécessaires à l'édition de liens, comme dans l'exemple suivant :

```
(lehrin) luc% cantor-config --libs
 -lcantor -lclub -lxerces
(lehrin) luc%
```

L'option `--version` permet d'obtenir sur la sortie standard le numéro de version de la bibliothèque, comme dans l'exemple suivant :

```
(lehrin) luc% cantor-config --version
 5.5
(lehrin) luc%
```

conseils d'utilisation

L'utilisation classique de `--cppflags` est dans une règle de compilation de fichier `Makefile` du type :

```
client.o : client.cc
     $(CXX) `cantor-config --cppflags` $(CPPFLAGS) $(CXXFLAGS) -c client.cc
```

L'utilisation classique de `--ldflags` et `--libs` est dans une règle d'édition de liens de fichier `Makefile` du type :

```
client : client.o
     $(CXX) -o $@ client.o \
            `cantor-config --ldflags` $(LDFLAGS) \
            `cantor-config --libs`    $(LIBS)
```

Il est possible de combiner les options `--cppflags`, `--ldflags` et `--libs` dans une règle de fichier `Makefile` du type :

```
client : client.cc
     $(CXX) -o $@ `cantor-config --cppflags --ldflags --libs` \
            $(CPPFLAGS) $(CXXFLAGS) $(LDFLAGS) $(LIBS) client.cc
```

14 description des routines

Les fonctions externes disponibles dans la bibliothèque sont la fonction de récupération de la version de la bibliothèque, les fonctions matricielles utilisées également par la classe MoindreCarreLineaire, les fonctions de recherche de racines, et les interfaces C et FORTRAN d'utilisation des rotations en dimension 3.

14.1 fonction C++ de recherche de mimimum

Le fichier `cantor/SectionDoree.h` déclare une fonction implantant l'algorithme de la section dorée pour la recherche d'un mimimum local de fonction à une variable.

14.1.1 interface

```
#include "cantor/SectionDoree.h"
```

TAB. 66: fonction de recherche de minimum par la section dorée

signature	description
void **SectionDoree** (double *fonc* (double x, void *), void **arg*, double *eps*, double *xa*, double *xb*, double **ptrXMin* = 0, double **ptrFMin* = 0)	recherche le minimum de *fonc* dans l'intervalle [*xa* ; *xb*], avec un critère de convergence en abscisse de *eps*. Si les pointeurs *ptrXMin* et *ptrFMin* sont non nuls, l'abscisse et l'ordonnée du point minimum y sont copiées. Le pointeur anonyme *arg* est passé à la fonction utilisateur à chaque appel, l'utilisateur peut s'en servir pour passer d'autres arguments ou récupérer des informations (la dernière évaluation de la fonction est garantie être faite au point de convergence).

14.2 fonctions C++ de résolution utilisant des dérivées premières

Le fichier `cantor/Resolution1.h` regroupe les algorithmes liés à la recherche des zéros et des extremums d'une fonction scalaire à une variable implantée sous forme de fonction renvoyant un objet de type ValeurDerivee1.

14.2.1 interface

```
#include "cantor/Resolution1.h"
```

Le fichier `cantor/Resolution1.h` définit les types :

```
typedef ValeurDerivee1 (*TypeFoncVD1) (double t, void* donnee);

enum CodeConvergence1 { CONV1_AUCUNE, CONV1_INF, CONV1_SUP };

typedef CodeConvergence1 (*TypeFoncConv1) (const Variation1& inf,
                                           const Variation1& sup,
                                           void* donnee);
```

`TypeFoncVD1` permet de déclarer des pointeurs sur des fonctions prenant un double et un void* en argument et retournant un ValeurDerivee1. On peut ainsi modéliser des fonctions mathématiques scalaires à une variable `t`, le pointeur anonyme `donnee` permettant de passer les arguments constants de la fonction dans une structure. La fonction informatique calcule simultanément la valeur et la dérivée de la fonction mathématique, qu'elle renvoie dans un ValeurDerivee1.

`CodeConvergence1` énumère les cas de convergence possibles pour un intervalle lors d'une recherche de zéro ou d'extremum. `CONV1_AUCUNE` signifie qu'aucune des bornes de l'intervalle ne vérifie les conditions de convergence, `CONV1_INF` signifie que la borne inférieure de l'intervalle vérifie les conditions de convergence, `CONV1_SUP` signifie que la borne supérieure vérifie les conditions de convergence.

`TypeFoncVD1` permet de déclarer des pointeurs sur des fonctions prenant en arguments deux références constantes sur des Variation1 et un void* et retournant un CodeConvergence1. Ceci permet de modéliser des fonctions de test de convergence sur un intervalle de recherche, fonctions pouvant indiquer si la convergence est ou non atteinte sur l'une ou l'autre des bornes de l'intervalle.

TAB. 67: fonctions de résolution utilisant des dérivées premières

signature	description
Variation1 **NewtonSecante** (TypeFoncVD1 *f*, void* *arg*, TypeFoncConv1 *convergence*, void* *arg_conv*, const Variation1& *a*, const Variation1& *b*)	retourne une copie de la variation de *f* pour le zéro compris dans l'intervalle [*a*.x () ; *b*.x ()]
double **NewtonSecante** (TypeFoncVD1 *f*, void* *arg*, TypeFoncConv1 *convergence*, void* *arg_conv*, double *a*, double *b*)	retourne le zéro de *f* dans l'intervalle [*a* ; *b*]
Variation1 **ExtremumNewSec** (TypeFoncVD1 *f*, void* *arg*, TypeFoncConv1 *convergence*, void* *arg_conv*, const Variation1& *a*, const Variation1& *b*)	retourne une copie de la variation de *f* pour l'extremum compris dans l'intervalle [*a*.x () ; *b*.x ()]
double **ExtremumNewSec** (TypeFoncVD1 *f*, void* *arg*, TypeFoncConv1 *convergence*, void* *arg_conv*, double *a*, double *b*)	retourne l'abscisse de l'extremum de *f* compris dans l'intervalle [*a* ; *b*]

La fonction globale **NewtonSecante** recherche le zéro de la fonction f dans l'intervalle $[a\,;\,b]$ par une méthode de Newton utilisant les dérivées premières calculées de façon automatique dans f aux deux bornes de l'intervalle. La recherche d'un nouveau point entre les deux bornes s'appuie sur une approximation cubique inverse. Cet algorithme est sécurisé par un algorithme de sécante. Ceci permet de garantir que l'argument t de la fonction f (cf description de `TypeFoncVD1`) soit entre a et b (inclus) pour tous les appels. L'argument *arg* peut être utilisé pour envoyer des paramètres constants à f (*arg* est passé à f sans être touché par l'algorithme).

La fonction globale **ExtremumNewSec** recherche un extremum de la fonction f dans l'intervalle $[a\,;\,b]$ par la méthode de Brent (cf. [DR4]). ATTENTION, contrairement à ce que le nom de cette fonction pourrait faire croire, il n'y a pas d'algorithme de Newton dans cette fonction, le New apparaissant dans le nom est destiné à garantir la compatibilité avec la fonction équivalente declarée dans le fichier `cantor/resolution2.h`, et qui elle utilise un Newton en deux points sécurisé par une sécante. Cette compatibilité permet de passer d'une signature à l'autre en changeant uniquement les types des arguments et des fonctions dans son programme, ce qui peut être réalisé de façon automatique. L'argument t de la fonction f (cf description de `TypeFoncVD1`) est garanti être entre a et b (inclus) pour tous les appels. L'argument *arg* peut être utilisé pour envoyer des paramètres constants à f (*arg* est passé à f sans être touché par l'algorithme).

Dans aucune des fonctions il n'est garanti que le point retourné soit le résultat du dernier appel à f, si par exemple la fonction (resp. la dérivée) s'annule en $a+\varepsilon$, le meilleur point peut très bien être le $f\ (a)$ calculé dès le début, la convergence n'étant détectée qu'après avoir appelé f de très nombreuses fois, la borne b étant la seule à évoluer au cours de la recherche, jusqu'à se rapprocher suffisamment de a. Il est donc très dangereux de se servir de *arg* pour que f retourne des paramètres à la fonction appelante[14], *arg* est destiné à être utilisé dans l'autre sens, pour que la fonction appelante envoie des paramètres à f. Dans le cas où la première signature est utilisée, les arguments a et b doivent être les variations de f aux bornes de l'intervalle de recherche (ce qui suppose que f a déjà été appelée), la valeur de retour sera alors la variation de f au meilleur point. Dans le cas où la seconde signature est utilisée, a et b sont les bornes de l'intervalle de recherche où il faut calculer la fonction, et la valeur de retour est l'abscisse du meilleur point.

Les algorithmes s'arrêtent soit lorsque la fonction convergence signale que l'une des bornes de l'intervalle courant vérifie les critères de convergence, soit lorsque l'intervalle courant est du même ordre de grandeur

[14] il est bien sûr possible d'utiliser *arg* pour retourner un compteur du nombre d'appels

que la précision de la machine (c'est à dire quand l'intervalle correspond à deux nombres modèles consécutifs — ou plûtot très proches — de la représentation machine des réels double précision). Il est donc possible (mais sûrement pas efficace en rapidité) d'utiliser une fonction convergence renvoyant systématiquement `CONV1_AUCUNE`. L'argument *arg_conv* peut être utilisé pour envoyer des paramètres constants à convergence (*arg_conv* est passé à convergence sans être touché par l'algorithme). Une copie du meilleur point trouvé par l'algorithme est retourné à l'appelant.

14.2.2 exemple d'utilisation

```
#include "cantor/resolution1.h"
#include "cantor/util.h"
...
ValeurDerivee1 SinusCroissant (double t, void* d)
{ // fonction s'annulant 9 fois entre -11 et 11
  // avec deux séries de zéros très proches (+/-10.907, +/-10.904)

  // incrémentation du compteur d'appels
  *((int *) d) += 1;
  ValeurDerivee1 x (t, 1.0);
  return sin (x) + x * 0.091325;
}

CodeConvergence1 ConvergenceMin (const Variation1& inf,
                                 const Variation1& sup,
                                 void* donnee)
{ if (abs (inf.y ().f1 ()) < 1.0e-6)
    return CONV1_INF;
  else if (abs (sup.y ().f1 ()) < 1.0e-6)
    return CONV1_SUP;
  else
    return CONV1_AUCUNE;
}

CodeConvergence1 Convergence0 (const Variation1& inf,
                               const Variation1& sup,
                               void* donnee)
{ // on teste la convergence sur x!
  if (sup.x () - inf.x () < 1.0e-6)
  { if (abs (inf.y ().f0 ()) < abs (sup.y ().f0 ())
      return CONV1_INF;
    else
      return CONV1_SUP;
  }
  else
    return CONV1_AUCUNE;
}
...
int compteur = 0;
tmin = ExtremumNewSec (SinusCroissant, (void *) &compteur,
```

```
                        ConvergenceMin, (void *) 0,
                        -11.0, -10.0);
cout << "tmin = " << tmin
     <<  "trouvé en : " << compteur << " itérations\n";

compteur = 0;
t0 = NewtonSecante (SinusCroissant, (void *) &compteur,
                    Convergence0, (void *) 0,
                    -11.0, tmin);
cout << "t0 = " << t0
     <<  "trouvé en : " << compteur << " itérations\n";
```

14.3 fonctions C++ de résolution utilisant des dérivées secondes

Le fichier `cantor/resolution2.h` regroupe les algorithmes liés à la recherche des zéros et des extremums d'une fonction scalaire à une variable implantée sous forme de fonction renvoyant un objet de type ValeurDerivee2.

14.3.1 interface

```
#include "cantor/resolution2.h"
```

Le fichier `cantor/resolution2.h` définit les types :

```
typedef ValeurDerivee2 (*TypeFoncVD2) (double t, void* donnee);

enum CodeConvergence2 { CONV2_AUCUNE, CONV2_INF, CONV2_SUP };

typedef CodeConvergence2 (*TypeFoncConv2) (const Variation2& inf,
                                           const Variation2& sup,
                                           void* donnee);
```

`TypeFoncVD2` permet de déclarer des pointeurs sur des fonctions prenant un double et un void* en argument et retournant un ValeurDerivee2. On peut ainsi modéliser des fonctions mathématiques scalaires à une variable `t`, le pointeur anonyme `donnee` permettant de passer les arguments constants de la fonction dans une structure. La fonction informatique calcule simultanément la valeur et la dérivée de la fonction mathématique, qu'elle renvoie dans un ValeurDerivee2.

`CodeConvergence2` énumère les cas de convergence possibles pour un intervalle lors d'une recherche de zéro ou d'extremum. `CONV2_AUCUNE` signifie qu'aucune des bornes de l'intervalle ne vérifie les conditions de convergence, `CONV2_INF` signifie que la borne inférieure de l'intervalle vérifie les conditions de convergence, `CONV2_SUP` signifie que la borne supérieure vérifie les conditions de convergence.

`TypeFoncVD2` permet de déclarer des pointeurs sur des fonctions prenant en arguments deux références constantes sur des Variation2 et un void* et retournant un CodeConvergence2. Ceci permet de modéliser des fonctions de test de convergence sur un intervalle de recherche, fonctions pouvant indiquer si la convergence est ou non atteinte sur l'une ou l'autre des bornes de l'intervalle.

La fonction globale **NewtonSecante** recherche le zéro de la fonction f dans l'intervalle $[a\,;\,b]$ par une méthode de Newton améliorée utilisant les dérivées première et seconde calculées de façon automatique dans

f, et sécurisée par un algorithme de sécante. Ceci permet de garantir que l'argument t de la fonction f (cf description de `TypeFoncVD2`) soit entre a et b (inclus) pour tous les appels. L'argument *arg* peut être utilisé pour envoyer des paramètres constants à f (*arg* est passé à f sans être touché par l'algorithme).

La fonction globale **ExtremumNewSec** recherche un extremum de la fonction f dans l'intervalle $[a\,;\,b]$ par une méthode de Newton utilisant les dérivées premières calculées de façon automatique dans f aux deux bornes de l'intervalle. La recherche d'un nouveau point entre les deux bornes s'appuie sur une approximation cubique inverse. Cet algorithme est sécurisé par un algorithme de sécante. L'argument t de la fonction f (cf description de `TypeFoncVD2`) est garanti être entre a et b (inclus) pour tous les appels. L'argument *arg* peut être utilisé pour envoyer des paramètres constants à f (*arg* est passé à f sans être touché par l'algorithme).

TAB. 68: fonctions de résolution utilisant des dérivées secondes

signature	description
Variation2 **NewtonSecante** (TypeFoncVD2 *f*, void* *arg*, TypeFoncConv2 *convergence*, void* *arg_conv*, const Variation2& *a*, const Variation2& *b*)	retourne une copie de la variation de f pour le zéro compris dans l'intervalle [*a*.x () ; *b*.x ()]
double **NewtonSecante** (TypeFoncVD2 *f*, void* *arg*, TypeFoncConv2 *convergence*, void* *arg_conv*, double *a*, double *b*)	retourne le zéro de f dans l'intervalle $[a\,;\,b]$
Variation2 **ExtremumNewSec** (TypeFoncVD2 *f*, void* *arg*, TypeFoncConv2 *convergence*, void* *arg_conv*, const Variation2& *a*, const Variation2& *b*)	retourne une copie de la variation de f pour l'extremum compris dans l'intervalle [*a*.x () ; *b*.x ()]
double **ExtremumNewSec** (TypeFoncVD2 *f*, void* *arg*, TypeFoncConv2 *convergence*, void* *arg_conv*, double *a*, double *b*)	retourne l'abscisse de l'extremum de f compris dans l'intervalle $[a\,;\,b]$

Dans aucune des fonctions il n'est garanti que le point retourné soit le résultat du dernier appel à f, si par exemple la fonction (resp. la dérivée) s'annule en $a + \varepsilon$, le meilleur point peut très bien être le f (a) calculé dès le début, la convergence n'étant détectée qu'après avoir appelé f de très nombreuses fois, la borne b étant la seule à évoluer au cours de la recherche, jusqu'à se rapprocher suffisamment de a. Il est donc très dangereux de se servir de *arg* pour que f retourne des paramètres à la fonction appelante[15], *arg* est destiné à être utilisé dans l'autre sens, pour que la fonction appelante envoie des paramètres à f. Dans le cas où la première signature est utilisée, les arguments a et b doivent être les variations de f aux bornes de l'intervalle de recherche (ce qui suppose que f a déjà été appelée), la valeur de retour sera alors la variation de f au meilleur point. Dans le cas où la seconde signature est utilisée, a et b sont les bornes de l'intervalle de recherche où il faut calculer la fonction, et la valeur de retour est l'abscisse du meilleur point.

Les algorithmes s'arrêtent soit lorsque la fonction convergence signale que l'une des bornes de l'intervalle courant vérifie les critères de convergence, soit lorsque l'intervalle courant est du même ordre de grandeur que la précision de la machine (c'est à dire quand l'intervalle correspond à deux nombres modèles consécutifs — ou plûtot très proches — de la représentation machine des réels double précision). Il est donc possible (mais sûrement pas efficace en rapidité) d'utiliser une fonction convergence renvoyant systématiquement `CONV2_AUCUNE`. L'argument *arg_conv* peut être utilisé pour envoyer des paramètres constants à convergence (*arg_conv* est passé à convergence sans être touché par l'algorithme). Une copie du meilleur point trouvé par l'algorithme est retourné à l'appelant.

[15] il est bien sûr possible d'utiliser *arg* pour retourner un compteur du nombre d'appels

14.3.2 exemple d'utilisation

```
#include "cantor/resolution2.h"
#include "cantor/util.h"
...
ValeurDerivee2 SinusCroissant (double t, void* d)
{ // fonction s'annulant 9 fois entre -11 et 11
  // avec deux séries de zéros très proches (+/-10.907, +/-10.904)

  // incrémentation du compteur d'appels
  *((int *) d) += 1;
  ValeurDerivee2 x (t, 1.0);
  return sin (x) + x * 0.091325;
}

CodeConvergence2 ConvergenceMin (const Variation2& inf,
                                 const Variation2& sup,
                                 void* donnee)
{ if (abs (inf.y ().f1 ()) < 1.0e-6)
    return CONV2_INF;
  else if (abs (sup.y ().f1 ()) < 1.0e-6)
    return CONV2_SUP;
  else
    return CONV2_AUCUNE;
}

CodeConvergence2 Convergence0 (const Variation2& inf,
                               const Variation2& sup,
                               void* donnee)
{ // on teste la convergence sur x!
  if (sup.x () - inf.x () < 1.0e-6)
  { if (abs (inf.y ().f0 ()) < abs (sup.y ().f0 ())
      return CONV2_INF;
    else
      return CONV2_SUP;
  }
  else
    return CONV2_AUCUNE;
}
...
int compteur = 0;
tmin = ExtremumNewSec (SinusCroissant, (void *) &compteur,
                       ConvergenceMin, (void *) 0,
                       -11.0, -10.0);
cout << "tmin = " << tmin
     <<  "trouvé en : " << compteur << " itérations\n";

compteur = 0;
t0 = NewtonSecante (SinusCroissant, (void *) &compteur,
                    Convergence0, (void *) 0,
```

```
                    -11.0, tmin);
cout << "t0 = " << t0
     <<  "trouvé en : " << compteur << " itérations\n";
```

14.4 autres fonctions C++

Les fonctions décrites dans la table 69 sont déclarées dans les fichiers suivants :
- **cantorVersion** :`cantor/CantorVersion.h` ;
- fonctions d'algèbre linéaire : `cantor/MoindreCarreLineaire.h` ;
- fonctions utilitaires : `cantor/Util.h`.

TAB. 69: fonctions C++ diverses

signature	description
const char ***cantorVersion** ()	retourne la version de la bibliothèque
void **factLDLt** (double *m, int n, double *seuil*) **throw**(CantorErreurs)	factorise sous forme $L.D.L^t$ la matrice symétrique définie positive m de taille $n \times n$ (dont seul le triangle supérieur est stocké dans le tableau), et remet le résultat dans m, lance une exception si l'un des éléments de la diagonale est inférieur au *seuil*
void **resoudLDLt** (const double *m, int n, double x [])	résoud le système linéaire $M.X = B$, où m contient la factorisation de M par la fonction précédente, et où x contient au départ le vecteur B et au retour le vecteur X solution
double **min** (double a, double b)	retourne le minimum de a et de b
int **min** (int a, int b)	retourne le minimum de a et de b
double **max** (double a, double b)	retourne le maximum de a et de b
int **max** (int a, int b)	retourne le maximum de a et de b
double **radians** (double a)	convertit un angle de degrés en radians
double **degres** (double a)	convertit un angle de radians en degrés
double **recaleAngle** (double a, double *ref*)	recale l'angle a d'un nombre entier de tours de sorte qu'il soit entre $ref - \pi$ et $ref + \pi$

14.5 fonctions C

TAB. 70: fonctions C

signature	description
int **RotAxeAngle** (double q [4], double *axe* [3], double *angle*, char* *message*, int *lgMaxMessage*)	construit la rotation q à partir de l'*axe* et de l'*angle*, retourne un code non nul en cas de problème, et initialise le *message* d'erreur
int **RotU1U2V1V2** (double q [4], double *u1* [3], double *u2* [3], double *v1* [3], double *v2* [3], char* *message*, int *lgMaxMessage*)	construit la rotation q à partir des deux vecteurs *u1* et *u2* et de leurs images *v1* et *v2*, retourne un code non nul en cas de problème, et initialise le *message* d'erreur
	à suivre ...

TAB. 70: fonctions C (suite)

signature	description
int **RotU1V1** (double *q* [4], double *u1* [3], double *v1* [3], char* *message*, int *lgMaxMessage*)	construit la rotation *q* à partir du vecteur *u1* et de son image *v1* (une solution est choisie arbitrairement dans l'infinité possible), retourne un code non nul en cas de problème, et initialise le *message* d'erreur
int **RotMatrice** (double *q* [4], double *m* [3][3], double *seuil*, char* *message*, int *lgMaxMessage*)	construit la rotation *q* à partir de la matrice *m* en corrigeant éventuellement sa non-orthogonalité d'au plus *seuil* (au sens de la norme de FROBENIUS), retourne un code non nul en cas de problème, et initialise le *message* d'erreur
int **RotTroisAngles** (double *q* [4], CantorAxesRotation *ordre*, double *alpha1*, double *alpha2*, double *alpha3*, char **message*, int *lgMaxMessage*)	construit la rotation *q* à partir de trois rotations élémentaires dans l'*ordre* spécifié, retourne un code non nul en cas de problème, et initialise le *message* d'erreur
void **RotInverse** (double *q* [4], double *qInitiale* [4])	construit la rotation *q* inverse de *qInitiale*, en se contentant d'inverser un élément du quaternion (cette opération est donc très peu coûteuse en temps de calcul)
void **RotComposee** (double *q* [4], double *q1* [4], double *q2* [4])	construit la rotation *q* résultant de la composition $q2 \circ q1$
void **AxeRot**(double *q* [4], double *axe* [3])	extrait l'axe de la rotation *q* et le met dans le tableau *axe*
void **AngleRot**(double *q* [4], double* *pAngle*)	extrait l'angle de la rotation *q* et le met dans la variable pointée par *pAngle*
void **AxeAngleRot** (double *q* [4], double *axe* [3], double* *pAngle*)	extrait l'axe et l'angle de la rotation *q* et les met dans le tableau *axe* et dans la variable pointée par *pAngle*
void **MatriceRot**(double *q* [4], double *m* [3][3])	extrait la matrice de la rotation *q* et la met dans le tableau *m*
int **TroisAnglesRot** (double *q* [4], CantorAxesRotation *ordre*, double **pAlpha1*, double **pAlpha2*, double **pAlpha3*, char **message*, int *lgMaxMessage*)	extrait de la rotation *q* les angles des rotations élémentaires dans l'*ordre* spécifié, retourne un code non nul en cas de problème, et initialise le *message* d'erreur
void **AppliqueRot** (double *q* [4], double *u* [3], double *uPrime* [3])	applique la rotation *q* au vecteur *u* et met l'image dans le tableau *uPrime*
void **AppliqueRotInverse** (double *q* [4], double *uPrime* [3], double *u* [3])	applique l'inverse de la rotation *q* au *uPrime* et met l'image réciproque dans le tableau *u*

14.6 sous-routines FORTRAN

Seules les fonctions d'utilisation des rotations en dimension 3 sont disponibles à partir du langage FORTRAN. Ces fonctions nécessitent le passage d'un ou de plusieurs tableaux de quatre réels pour contenir les composantes du quaternion représentant la rotation. Ces composantes sont rarement utiles pour l'appelant, il se contente généralement de passer les tableaux à la bibliothèque CANTOR, laquelle les initialise dans le cas de fonctions construisant des rotations, et les utilise dans les autres cas.

TAB. 71: fonctions FORTRAN

signature	description
integer function **RotAxeAngle** (*q*, *axe*, *angle*, *message*) double precision *q* (4) double precision *axe* (3), *angle* character*(*) *message*	construit la rotation *q* à partir de l'*axe* et de l'*angle*, retourne un code non nul en cas de problème, et initialise le *message* d'erreur
integer function **RotU1U2V1V2** > (*q*, *u1*, *u2*, *v1*, *v2*, *message*) double precision *q* (4) double precision *u1* (3), *u2* (3) double precision *v1* (3), *v2* (3) character*(*) *message*	construit la rotation *q* à partir des deux vecteurs *u1* et *u2* et de leurs images *v1* et *v2*, retourne un code non nul en cas de problème, et initialise le *message* d'erreur
integer function **RotU1V1** (*q*, *u1*, *v1*, *message*) double precision *q* (4) double precision *u1* (3), *v1* (3) character*(*) *message*	construit la rotation *q* à partir du vecteur *u1* et de son image *v1* (une solution est choisie arbitrairement dans l'infinité possible), retourne un code non nul en cas de problème, et initialise le *message* d'erreur
integer function **RotMatrice** (*q*, *m*, *seuil*, *message*) double precision *q* (4) double precision *m* (3, 3), *seuil* character*(*) *message*	construit la rotation *q* à partir de la matrice *m* en corrigeant éventuellement sa non-orthogonalité d'au plus *seuil* (au sens de la norme de FROBENIUS), retourne un code non nul en cas de problème, et initialise le *message* d'erreur
integer function **RotTroisAngles** > (*q*, *ordre*, *alpha1*, *alpha2*, *alpha3*, *message*) double precision *q* (4) double precision *alpha1*, *alpha2*, *alpha3* character*(*) *message*	construit la rotation *q* à partir de trois rotations élémentaires dans l'*ordre* spécifié (qui doit être conforme à l'un des `PARAMETER` déclarés dans le fichier `cantor/cantdefs.f` que l'on peut inclure par `#include` ou par `INCLUDE` au niveau du fichier appelant), retourne un code non nul en cas de problème, et initialise le *message* d'erreur
subroutine **RotInverse** (*q*, *qInitiale*) double precision *q* (4), *qInitiale* (4)	construit la rotation *q* inverse de *qInitiale*, en se contentant d'inverser un élément du quaternion (cette opération est donc très peu coûteuse en temps de calcul)
subroutine **RotComposee** (*q*, *q1*, *q2*) double precision *q* (4), *q1* (4), *q2* (4)	construit la rotation *q* résultant de la composition $q2 \circ q1$
subroutine **AxeRot** (*q*, *axe*) double precision *q* (4), *axe* (3)	extrait l'*axe* de la rotation *q*
subroutine **AngleRot** (*q*, *angle*) double precision *q* (4), *angle*	extrait l'*angle* de la rotation *q*
subroutine **AxeAngleRot** (*q*, *axe*, *angle*) double precision *q* (4), *axe* (3), *angle*	extrait l'*axe* et l'*angle* de la rotation *q*
subroutine **MatriceRot** (*q*, *m*) double precision *q* (4), *m* (3, 3)	extrait la matrice *m* de la rotation *q*
integer function **TroisAnglesRot** > (*q*, *ordre*, *alpha1*, *alpha2*, *alpha3*, *message*) double precision *q* (4) double precision *alpha1*, *alpha2*, *alpha3* character*(*) *message*	extrait de la rotation *q* les angles des rotations élémentaires dans l'*ordre* spécifié (qui doit être conforme à l'un des `PARAMETER` déclarés dans le fichier `cantor/cantdefs.f` que l'on peut inclure par `#include` ou par `INCLUDE` au niveau du fichier appelant), retourne un code non nul en cas de problème, et initialise le *message* d'erreur
subroutine **AppliqueRot** (*q*, *u*, *uPrime*) double precision *q* (4) double precision *u* (3), *uPrime* (3)	applique la rotation *q* au vecteur *u* et met l'image dans le tableau *uPrime*
subroutine **AppliqueRotInverse** (*q*, *uPrime*, *u*) double precision *q* (4) double precision *uPrime* (3), *u* (3)	applique l'inverse de la rotation *q* au *uPrime* et met l'image réciproque dans le tableau *u*