MARMOTTES

Communications et Systèmes
Systèmes d'Informations
Direction Espace

Édition : **5** Date : **01/02/2002**
Révision : **7** Date : **04/03/2005**

Réf. : ESPACE/MS/CHOPE/MARMOTTES/MU/001

Manuel d'utilisation

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 : attitude, multimission, bibliothèque

TITRE DU DOCUMENT :

Manuel d'utilisation

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

RÉSUMÉ :

Ce manuel décrit l'interface utilisateur de la bibliothèque MARMOTTES, tant au niveau de la programmation (fonctions de bibliothèque) qu'au niveau des données (fichiers de ressources des données senseurs). Les fonctions de CANTOR permettant de traiter les rotations en dimension 3 sont également décrites.

DOCUMENTS RATTACHÉS : ce document vit seul		LOCALISATION :	
VOLUME : 1	NBRE TOTAL DE PAGES : 216 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 entre les versions 9.7 et 9.8			
CONTRAT : Néant			
SYSTÈME HÔTE : Unix – LaTeX 2_ε			

MODIFICATIONS

Éd.	Rév.	Date	Référence, Auteur(s), Causes d'évolution
			Pour l'historique des modifications antérieures:
			se reporter à l'édition 05 / révision 05 de ce document
5	0	01/02/02	L. Maisonobe, création du document CS
5	1	08/04/02	S. Vresk, description des évolutions entre les versions 9.1 et 9.2
5	2	06/09/02	G. Prat, description des évolutions entre les versions 9.2 et 9.3
5	3	05/03/03	S. Vresk, description des évolutions entre les versions 9.3 et 9.4
5	4	28/07/03	L. Maisonobe, description des évolutions entre les versions 9.4 et 9.5
5	5	28/07/03	L. Maisonobe, description des évolutions entre les versions 9.5 et 9.6
5	6	21/06/04	G. Prat, description des évolutions entre les versions 9.6 et 9.7
5	7	04/03/05	L. Maisonobe, description des évolutions entre les versions 9.7 et 9.8

SOMMAIRE

GLOSSAIRE

Marmottes
: Modélisation d'Attitude par Récupération des Mesures d'Orientation pour tout Type d'Engin Spatial

senseur
: équipement bord permettant à un satellite de mesurer son mouvement autour de son centre de gravité

IRES
: Infra-Red Earth Sensor

TM
: télémesure

DOCUMENTS DE RÉFÉRENCE

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

[DR2] *Manuel d'utilisation de la bibliothèque CANTOR*
ESPACE/MS/CHOPE/CANTOR/MU/001 , édition 5.11, 04/03/2005

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

[DR4] *MERCATOR – accès aux fichiers*
CT/TI/MS/SG/95-024 , édition 1.0, 14/02/1995

[DR5] *Traduction de messages et mots-clefs*
CT/TI/MS/SG/95-015 , édition 1.0, 06/01/1995

[DR6] *Iterative Optimal Orthogonalization of the Strap down Matrix*
Itzhack Y. Bar-Itzhack, ieee Transactions on Aerospace and Electronic Systems – Vol AES-11, January 1975

[DR7] *GNU Coding Standards*
R. Stallman, 09/09/1996

[DR8] *GNU Autoconf, Automake, and Libtool*
G. V. Vaughan, B. Elliston, T. Tromey, I. L. Taylor, October 2000

[DR9] *Autoconf*
D. MacKenzie, édition 2.13, 12/1998

[DR10] *Automake*
D. MacKenzie, T. Tromey, édition 1.3, 04/1998

[DR11] *Version Management with CVS*
documentation de CVS 1.10

[DR12] *CVS Client/server*
documentation de CVS 1.10

[DR13] *GNU g++*
R. Stallman

[DR14] *Standard Template Library Programmer's Guide*
Silicon Graphics Computer Systems, http://www.sgi.com/Technology/STL, 1996

DOCUMENTS APPLICABLES

Néant

1 présentation

MARMOTTES (Modélisation d'Attitude par Récupération des Mesures d'Orientation pour Tout Type d'Engin Spatial) est une bibliothèque regroupant des fonctions de simulation de contrôle d'attitude multimissions, c'est-à-dire dont le code n'est pas dédié à une plate-forme particulière de satellite.

Cette bibliothèque regroupe des objets *métier* destinés à être utilisés dans des applications de mécanique spatiale qui peuvent soit être auto-suffisantes, soit être elles-mêmes des bibliothèques plus spécialisées ; il s'agit donc d'une bibliothèque de niveau intermédiaire.

Ce document décrit la version 9.8 de la bibliothèque.

2 description du contexte

Cette bibliothèque récupère les descriptions complètes des senseurs dans des fichiers de ressources ; le paramétrage des fonctions de simulation est donc très simple, il suffit de passer le nom du fichier de ressources, les noms de senseurs, les valeurs des consignes.

Les fonctions simulent un pilotage parfait respectant ces consignes et permettent entre autres de retourner l'attitude (sous forme de quaternion) et le spin. La bibliothèque CANTOR (qui est d'ailleurs utilisée par MARMOTTES) fournit les fonctions de calcul liées aux quaternions dont peut avoir besoin le programme principal.

La bibliothèque est écrite en C++ et dispose d'une interface C++, d'une interface C, et d'une interface FORTRAN (les fonctions de calcul de quaternions de CANTOR disposent des mêmes interfaces).

La liste ordonnée des bibliothèques à spécifier à l'éditeur de liens est la suivante :

```
-lmarmottes -lcantor -lclub
```

Il est indispensable de faire l'édition de liens avec le compilateur C++ qui a été utilisé pour compiler la bibliothèque car celle-ci utilise le mécanisme des exceptions qui nécessite un support de la part de l'éditeur de liens. Si le programme appelant est en FORTRAN ou en FORTRAN 90, il faut spécifier explicitement les bibliothèques fortran (par exemple avec la version 4.2 du compilateur fortran 77 SUN, il faut ajouter `-lF77 -lM77 -lsunmath`).

Cette note décrit les fichiers de ressources des senseurs et toutes les routines de bibliothèque. Les routines liées aux quaternions sont décrites dans la documentation de CANTOR [DR2]. Les principes de résolution et les algorithmes utilisés sont décrits en détail dans la documentation mathématique de MARMOTTES [DR1].

3 conventions

3.1 rotations

La représentation des rotations dans l'espace présente souvent des ambiguïtés selon que l'on considère des vecteurs tournant dans un repère fixe ou un repère tournant au milieu de vecteurs inertiels. Afin d'éviter des

interprétations erronées, nous allons expliciter les conventions de MARMOTTES de façon fonctionnelle, c'est-à-dire en définissant implicitement les rotations par les vecteurs qu'elles consomment et ceux qu'elles produisent [1].

Soit *Att* l'attitude produite par MARMOTTES à la suite d'un calcul. Soit $\vec{u}_{\text{in}}$ les coordonnées en repère inertiel d'un vecteur défini de façon absolue dans l'espace (par exemple la direction du Soleil). Soit $\vec{u}_{\text{sat}}$ les coordonnées en repère satellite de ce même vecteur. Ces trois éléments sont liés par :

$$\vec{u}_{\text{sat}} = Att(\vec{u}_{\text{in}})$$

Cette convention peut être illustrée par les exemples suivants :
- calcul d'une direction de poussée par lecture des télémesures senseurs ;
- recherche des consignes permettant d'aboutir naturellement à l'attitude optimale de poussée ;
- initialisation d'une attitude à partir des données CVI.

3.1.1 Calcul d'une direction de poussée

Supposons que l'on désire intégrer numériquement la poussée réalisée par un satellite en lisant la télémesure. À chaque ligne de télémesure, on extrait les mesures m1, m2, m3 réalisées par trois senseurs en visibilité, et on les utilise comme consignes pour MARMOTTES qui renvoie le tableau *att*. On connait la direction de poussée en repère satellite : *Psat*.

Pour intégrer cette poussée numériquement, on la convertit en repère intertiel par un appel du type (en FORTRAN) :

call AppliqueRotInverse (att, Psat, Pin)

3.1.2 Recherche des consignes pour une poussée optimale

Les logiciels d'optimisation donnent l'attitude au début de poussée. Supposons que la procédure opérationnelle impose que cette attitude soit contrôlée par des consignes géométriques et une consigne cinématique figées depuis au moins 45 minutes (pour la tranquilisation des ergols), et qu'il faille donc extrapoler cette attitude à rebours avec le gyromètre pour trouver les trois consignes géométriques d'initialisation pour la mise en attitude avant le basculement sur gyromètre.

On connait la direction de poussée en repère inertiel *pousseein*, et la direction satellite/terre (et donc l'angle poussée/direction terre θ). Si on suppose que le roulis est nul et que la poussée est sur +Xsat, on en déduit la direction terre en repère satellite :

$$\vec{terre}_{sat} \begin{cases} \cos(\theta) \\ 0 \\ \sin(\theta) \end{cases}$$

L'attitude en début de poussée se calcule par :

call RotU1U2V1V2 (att, pousseein, terrein, Xsat, terresat)

[1] L'auteur s'avoue incapable de dire ce que fait MARMOTTES en termes de *matrices de passage*

(Cette attitude vérifie $att(\vec{pousse}_{\text{in}}) = \vec{X}_{sat}, att(\vec{terre}_{\text{in}}) = \vec{terre}_{\text{sat}}$).

On utilise cette attitude pour initialiser MARMOTTES, et on extrapole à rebours pendant 45 minutes. On obtient l'attitude initiale att0.

Les mesures fournies par les senseurs géométriques dans cette attitude sont obtenues par des appels du type :

```
if ((MarmottesMesure (id, 'IRES_ROLL', roulis, message) .eq. 0)
    .or.
    (MarmottesMesure (id, 'IRES_PITCH', tangage, message) .eq. 0)
    .or.
    (MarmottesMesure (id, 'SSH_YAW', lacet, message) .eq. 0)) then
  write (iaffi, message)
  stop
endif
```

affichage des valeurs roulis, tangage, lacet recherchées

Dans le sens normal du temps, on utiliserait ces valeurs comme consignes, ce qui permettrait d'aboutir à att0, puis en passant sur gyromètre l'attitude évoluerait naturellement jusqu'à l'attitude optimale pour la poussée.

3.1.3 Initialisation d'une attitude à partir des données CVI

Les CVI Ariane donnent l'évolution de l'attitude du lanceur sous forme de trois angles mesurés depuis l'initialisation de la centrale, 9 secondes avant la mise en feu.

Pour déduire l'attitude du satellite à chaque instant, il faut d'une part connaître l'orientation du satellite par rapport au lanceur, et d'autre part connaître l'orientation du lanceur par rapport au référentiel inertiel à l'initialisation de la centrale.

On définit pour cela toute une série de repères intermédiaires.

Repère satellite : on connait les coordonnées de ses axes $\vec{X}_{\text{sat}}$, $\vec{Y}_{\text{sat}}$, $\vec{Z}_{\text{sat}}$ dans le repère lanceur.

Repère lanceur : $\vec{X}_{\text{lanceur}}$ est l'axe de roulis (longitudinal, positif dans le sens de l'avancement), $\vec{Y}_{\text{lanceur}}$ est l'axe de lacet, $\vec{Z}_{\text{lanceur}}$ est l'axe de tangage.

À l'initialisation, l'axe $+\vec{X}_{\text{lanceur}}$ est aligné avec le $+\vec{Zenith}_{\text{rampe}}$, l'axe $+\vec{Y}_{\text{lanceur}}$ est décalé de $azimut_{\text{plateforme}}$" vers l'Est, à partir du $+\vec{Nord}_{\text{rampe}}$.

Repère rampe : C'est le repère (Zenith, Est, Nord) du pas de tir.

Repère trajectoire : C'est un repère équatorial situé dans le méridien du pas de tir (il en est donc décalé du repère rampe de la latitude de tir).

Repère inertiel : L'écart entre le repère trajectoire et le repère inertiel est lié à la longitude du repère trajectoire (c'est-à-dire la longitude du pas de tir puisqu'ils sont dans le même méridien) et à la position de la terre, c'est-à-dire au temps sidéral à $H_0 - 9$ s.

Les angles de roulis, tangage et lacet diffusés dans les CVI peuvent être interprétés comme suit :

Pour passer du repère lanceur à H_0-9 au repère lanceur à la date courante, on tourne le lanceur de $-$tangage autour de Z, puis de $-$lacet autour de Y', puis de $-$roulis autour de X" (dans les CVI, les angles sont donnés dans l'ordre lacet, roulis, tangage).

On peut calculer au préalable la rotation qui appliquée à un vecteur en repère lanceur à $H_0 - 9$ s donne les coordonnées de ce même vecteur en repère inertiel.

Pour des raisons de concision[2], cet exemple est donné en C++.

```
//vecteurs canoniques
VecDBL i (1, 0, 0);
VecDBL j (0, 1, 0);
VecDBL k (0, 0, 1);

//conversion de vecteur satellite en vecteur lanceur
VecDBL xSat (x1, x2, x3); //coordonnées en repère lanceur
VecDBL ySat (y1, y2, y3); //coordonnées en repère lanceur
RotDBL satLanceur (i, j, Xsat, Ysat);

//conversion de vecteur lanceur en vecteur rampe
VecDBL xLanceur = i; //aligné avec Zénith rampe
VecDBL yLanceur (0, sin (azimut), cos (azimut)); //décalé par rapport au Nord
RotDBL lanceurRampe (i, j, Xlanceur, Ylanceur);

//conversion de vecteur rampe en vecteur trajectoire
VecDBL = j; //aligné avec l'est trajectoire
VecDBL zenith (cos (lat), 0, sin (lat)); //décalé par rapport à l'équateur
RotDBL rampeTraj (j, i, est, zenith);

//conversion de vecteur trajectoire en vecteur inertiel
// tsidr : temps sidéral à H0 - 9s
VecDBL pole = k; //aligné avec le pole trajectoire
VecDBL meridien (cos (tsidr + longi), sin (tsidr + longi), 0);
RotDBL trajInert (k, i, pole, meridien);

//combinaison des trois dernières rotations
RotDBL lanceurInert = trajInert (rampeTraj (lanceurRampe));
```

À chaque lecture des angles d'attitude, on calcule l'évolution, en sachant que si le lanceur (c'est-à-dire le repère) à tourné d'abord de $-\theta$ autour de Z, puis de $-\psi$ autour de Y, puis de $-\phi$ autour de X, alors pour convertir un vecteur du repère lanceur courant dans le repère lanceur à H0 - 9s, il faut d'abord le tourner de $+\theta$ autour de X, puis de $+\psi$ autour de Y, puis de $+\phi$ autour de Z.

On calcule donc :

```
RotDBL roulis  (VecDBL (1, 0, 0), phi);
RotDBL lacet   (VecDBL (0, 1, 0), psi);
```

[2]et parce que le module opérationel qui réalise ce calcul est écrit de cette façon

```
RotDBL tangage (VecDBL (0, 0, 1), theta);
RotDBL lanceurTlanceurH09 = tangage (lacet (roulis));
```

Enfin l'attitude compatible avec la convention MARMOTTES se calcule par :

```
RotDBL satInert = lanceurInert (lanceurTlanceurH09 (satLanceur));
RotDBL attitude = -satInert;
```

On peut utiliser directement cette rotation pour initialiser MARMOTTES.

Remarque : On peut faire bien plus court ! Mais optimiser ce calcul impose de savoir reconnaître les rotations de vecteurs des rotations de repère (en construisant certaines rotations directement par un axe et un angle avec le bon signe, comme nous avons dû le faire pour roulis/tangage/lacet), de plus faire plus court serait nettement plus obscur. On pourrait ainsi calculer directement les rotations inverses de ce que l'on a fait, pour aboutir d'emblée à l'attitude plutôt que d'inverser la dernière rotation. Il faut cependant se rappeler qu'avec les quaternions, inverser une rotation ne coûte guère que le temps du changement de signe d'un unique réel.

3.2 unités

Il faut prendre garde au problème des unités. MARMOTTES travaille en kilomètres et kilomètres par secondes en interne, et la norme du vecteur position influe en particulier sur les corrections de parallaxe. Si l'appelant utilise des unités différentes, il doit le signaler à la bibliothèque.

Les senseurs posent un problème un peu plus compliqué car l'un des fondements de MARMOTTES est de masquer le type des senseurs au maximum. Ne connaissant pas le genre de la mesure (angle, vitesse, coordonnée cartésienne) on ne peut la convertir. La conversion est donc du ressort de chaque senseur individuellement, et MARMOTTES introduit uniquement une notion d'unité interne (pour les calculs) et d'unité externe. Certains senseurs comme les senseurs cartésiens ignorent ces notions, d'autres les utilisent par exemple en prenant des radians en unité interne et des degrés en unité externe. L'appelant paradoxalement a plus d'informations que la bibliothèque sur les unités (même s'il n'en a pas sur les senseurs) ! Le développeur sait en effet si les données qu'il manipule sont issues ou destinées à des routines de calcul ou à des routines d'entrées-sorties. Dans un cas il pourra signaler à MARMOTTES qu'il s'agit d'unités internes (cela sous-entend qu'en fait il sait très bien ce que repésente chaque mesure et chaque senseur) et dans l'autre cas il signalera qu'il s'agit d'unités externes (par exemple une lecture de fichier ou de télémesure). Par défaut, les échanges ont lieu dans les unités internes (c'est à dire que personne ne fait de conversion).

4 catalogue

4.1 classes disponibles

BodyEphem permet l'accès aux différentes implémentations utilisateurs du calcul du temps sidéral et des éphémérides du Soleil, de la Lune et de la Terre, par rapport au corps central. Elle permet aussi l'accès à des grandeurs physiques du corps central (rayon équatorial, aplatissement et vitesse de rotation).

BodyEphemC permet l'accès à l'implémentation en C par l'utilisateur, du calcul du temps sidéral et des éphémérides du Soleil, de la Lune et de la Terre, par rapport au corps central.
Elle permet aussi l'accès à des grandeurs physiques du corps central (rayon équatorial, aplatissement et vitesse de rotation).

BodyEphemF permet l'accès à l'implémentation en FORTRAN par l'utilisateur, du calcul du temps sidéral et des éphémérides du Soleil, de la Lune et de la Terre, par rapport au corps central.
Elle permet aussi l'accès à des grandeurs physiques du corps central (rayon équatorial, aplatissement et vitesse de rotation).

Etat mémorise l'état du satellite (date, position, vitesse, attitude) ainsi que quelques données qui lui sont directement liées comme les directions de la lune et du soleil ;

Famille est la classe qui sert d'interface à la classe FamilleAbstraite ;

FamilleAbstraite est une classe abstraite de haut niveau permettant de gérer les différents types de solutions aux modèles analytiques de résolution ;

FamilleAlignementMoins est une classe dérivée de la précédente, elle permet de traiter un cas particulier de singularité ;

FamilleAlignementPlus est une classe dérivée de FamilleAbstraite, elle permet de traiter un cas particulier de singularité ;

FamilleFixe est une classe dérivée de FamilleAbstraite, elle est chargée de la modélisation géométrique dans le cas où le vecteur cible est figé par les deux consignes ;

FamilleGenerale est une classe dérivée de FamilleAbstraite, elle est chargée de la modélisation géométrique dans le cas général, pour un domaine angulaire particulier ;

FamillePrologementPi est une classe dérivée de FamilleAbstraite, elle permet de traiter un cas particulier de singularité ;

FamillePrologementZero est une classe dérivée de FamilleAbstraite, elle permet de traiter un cas particulier de singularité ;

Marmottes est la classe de plus haut niveau de la bibliothèque, elle représente un simulateur d'attitude complet, avec son état courant et les moyens de le faire évoluer ;

MarmottesErreurs gère toutes les erreurs internes de la bibliothèque dans la langue de l'utilisateur ;

Modele est une classe abstraite servant d'interface à des modèles à un degré de liberté décrivant l'ensemble des attitudes respectant deux consignes sur les trois ;

ModeleCine implante la classe Modele dans le cas de consignes portant sur des senseurs cinématiques ;

ModeleGeom implante la classe Modele dans le cas de consignes portant sur des senseurs géométriques ;

Parcelle est une classe abstraite servant d'interface aux descriptions de champs de vue ayant une notion de visibilité booléenne (le limbe doit être dans le scan Nord *et* dans le scan Sud) en plus de la notion de visibilité géométrique ;

ParcelleElementaire implante les Parcelles les plus simples qui soient : réduites à la notion de visibilité géométrique ;

ResolveurAttitude est une classe recherchant numériquement les valeurs du degré de liberté permettant à la classe Modèle de générer une attitude respectant les trois consignes, et triant les vrais solutions des artefacts de modélisation mathématique ;

ReunionEtParcelles implante les Parcelles ayant besoin de deux visibilités simultanées ;

ReunionOuParcelles implante les Parcelles se contentant d'une visibilité parmi deux possibilités ;

Senseur est une classe abstraite servant d'interface à tous les types de senseurs ou de pseudo-senseurs, il s'agit de la classe de base d'une hiérarchie d'héritage simple mais comportant de nombreuses classes (voir la figure 3, page 26) ;

SenseurAlpha implante un pseudo-senseur d'ascension droite, c'est à dire un senseur qui mesure la direction d'un vecteur satellite par rapport à un repère inertiel ;

SenseurCardan implante tous les pseudo-senseurs de Cardan, c'est à dire les senseurs mesurant des rotations successives permettant de passer du repère orbital local au repère satellite, tous les ordres possibles pour les rotations sont supportés ;

SenseurCartesien implante des senseurs optiques dont la mesure est directement une composante du vecteur cible en repère satellite (il s'agit généralement de mesures composites élaborées à bord par le SCAO à partir d'un ensemble de senseurs de base) ;

SenseurCinematique implante les gyromètres et permet la construction des modèles analytiques ModeleCine ;

SenseurDelta implante un pseudo-senseur de déclinaison, c'est à dire un senseur qui mesure la direction d'un vecteur satellite par rapport à un repère inertiel ;

SenseurDiedre implante des senseurs optiques mesurant des angles entre plans autour d'un axe dièdre (la plupart des senseurs optiques sont de ce type) ;

SenseurElevation implante des senseurs mesurant l'élévation d'un vecteur cible au dessus d'un plan de référence (ce type de senseur est essentiellement utilisé comme pseudo-senseur pour produire la seconde coordonnée sphérique d'une direction, en association avec un senseur dièdre qui donne un azimut) ;

SenseurFonction est la classe de base de tous les senseurs représentant des fonctions quelconques sur la sphère unité ;

SenseurFonctionEchant1D permet de modéliser des senseurs dont la mesure est une fonction sur la sphère unité échantillonnée radialement, ce type de senseur est principalement utilisé pour modéliser des gains d'antennes mesurés ;

SenseurFonctionGauss permet de modéliser des senseurs dont la mesure est une fonction gausienne sur la sphère unité, ce type de senseur est principalement utilisé pour modéliser des gains d'antennes mesurés ;

SenseurFonctionSinCard2 permet de modéliser des senseurs dont la mesure est une fonction $(\sin\theta/\theta)^2$ sur la sphère unité, ce type de senseur est principalement utilisé pour modéliser des gains d'antennes mesurés ;

SenseurGeometrique est une classe abstraite regroupant tous les senseurs géométriques et factorisant les données et méthodes nécessaires pour construire les modèles analytiques ModeleGeom ;

SenseurGyroInteg implante les senseurs de type gyromètres intégrateurs ;

SenseurLimbe est une spécialisation de SenseurDiedre pour le cas des senseurs infrarouge observant le limbe du corps central ;

SenseurOptique est une classe abstraite regroupant tous les senseurs géométriques observant une cible en repère inertiel (terre, lune, étoiles, moment cinétique de l'orbite, ...) ;

SenseurVecteur implante des senseurs optiques mesurant des angles entre deux vecteurs (la cible et une référence connue en repère satellite), ces senseurs permettent de modéliser les mesures réalisées à bord de satellites spinnés, mais ils servent surtout de pseudo-senseurs par exemple pour connaître le dépointage d'une antenne par rapport à une station sol ;

SpinAtt permet de mémoriser une attitude et un spin, elle est utilisée comme intermédiaire entre le modèle analytique à un degré de liberté et la recherche numérique de la bonne valeur de ce degré de liberté pour respecter la dernière consigne ;

StationCible permet de modéliser une station sol, qui peut être une cible observée par un senseur afin de savoir si un satellite qui pointe vers elle peut ou non la voir, les masques sols fonctions de l'azimut et la réfraction atmosphérique fonction des conditions météorologiques locales sont pris en compte.

5 environnement

Les données spécifiques aux senseurs sont des données relativement complexes comprenant parfois des structures récursives (pour définir les champs de vue). Ces données seraient assez difficiles à lire dans un programme principal et à transmettre à la bibliothèque. Il a donc été décidé que la bibliothèque serait seule responsable de ces lectures, l'appelant ne voyant plus que le nom du fichier des senseurs et les noms des senseurs eux-mêmes. Il est tolérable que des fonctions de bibliothèques – rarement modifiées – soient assez complexes, aussi a-t-il été possible d'utiliser une syntaxe de fichiers assez riche pour faciliter l'édition manuelle de ces fichiers (possibilités de commentaires, formats variables, ordre de certains éléments non significatif, possibilité d'utiliser plusieurs lignes, inclusions de fichiers, héritage de données entre structures proches, ...).

Cette syntaxe est basée sur la notion de fichiers structurés par blocs telle qu'elle est implantée dans la bibliothèque CLUB (voir [DR3]), la couche MARMOTTES spécialisant cette syntaxe générale en spécifiant les blocs utiles, leur nom et leur structuration.

5.1 Syntaxe générale

Le principe des fichiers structurés a été élaboré afin de permettre une représentation textuelle de structures de données imbriquées avec un nombre de niveaux quelconque non déterminé à l'avance, mais découvert au cours de la lecture du fichier. Ceci permet en particulier de décrire des structures récursives.

L'analyse est contrôlée par le programme, qui va chercher les blocs dont il a besoin les uns après les autres, inspecte leur contenu, et éventuellement fonde ses choix sur les valeurs lues. Il ne s'agit absolument pas d'un *langage*, pour lequel ce serait plutôt le texte du fichier qui contrôlerait le programme lecteur (traduction dirigée par la syntaxe).

Les lexèmes des fichiers structurés sont le marqueur de commentaire `#`, les délimiteurs `{` et `}`, les séparateurs (espace, tabulation, fin de ligne), et les champs (suite de caractères n'appartenant à aucune des catégories précédentes).

Les commentaires s'étendent du marqueur `#` à la fin de la ligne, il peut y avoir des données entre le début de ligne et les commentaires.

Hormis leur rôle de séparation des champs, les séparateurs n'ont aucune fonction dans la syntaxe des fichiers structurés, l'utilisateur peut en user à loisir pour améliorer la lisibilité de son fichier en jouant sur les lignes vides et l'indentation.

5.1.1 Blocs

Toute donnée qu'elle soit élémentaire ou composée est définie dans un bloc entre accolades précédé par un nom, la donnée est repérée dans le fichier par le nom de son bloc. On peut ainsi définir un fichier de constantes par :

Expl. 1 – *blocs de données*

```
pi {3.14}
e  {2.17}
```

Seul le nom est utilisé pour accéder à la donnée, l'ordre des blocs dans le fichier est indifférent. Les blocs sans nom sont autorisés (on utilise alors une chaîne vide en guise de nom dans les routines d'accès). Le bloc étant référencé par son nom, il ne faut pas que deux blocs différents portent le même nom à l'intérieur d'une zone de recherche unique. La casse utilisée pour les caractères du nom *est* significative.

5.1.2 Données élémentaires

Une donnée élémentaire est une donnée ne contenant pas de sous-bloc (pas d'accolades imbriquées). Une donnée élémentaire peut contenir un ou plusieurs champs, alphabétiques et numériques, séparés par des blancs (espace, tabulation, fins de lignes). On accède à ces champs par leur numéro, l'ordre est donc important à l'intérieur d'un bloc élémentaire, comme le montre l'exemple 2 :

Expl. 2 – *différents types de blocs élémentaires*

```
cible      {Soleil}
precision  {0.1}                 # unités : degrés
axe        {0.707  0.707  0.0 }  #  <--  attention à l'ordre !!!
```

5.1.3 Données composées

Une donnée composée est décrite par un bloc contenant des sous-blocs (accolades imbriquées). Chaque sous-bloc a un nom (qui peut être vide) et l'ordre des sous-blocs est indifférent. En fait l'accès à un sous-bloc à partir d'un bloc est similaire à l'accès à un bloc depuis le fichier, exactement comme si un bloc n'était qu'un sous-bloc du fichier[3]. La recherche d'un sous-bloc étant limitée par le bloc englobant, deux blocs différents peuvent contenir un sous-bloc de même nom sans ambiguïté.

Expl. 3 – *bloc composé*

```
cone { # cône d'axe i et d'angle pi/2 : demi-espace x > 0
       axe   {1.0  0.0  0.0}
       angle {90.0}
     }
```

Cette syntaxe permet de décrire une donnée structurée même à définition récursive (comme un arbre) mais pas de donnée engendrant une récursivité infinie (par exemple une liste dont la fin rebouclerait sur le début). Un programme prévu pour lire des données récursives recherche d'abord les blocs principaux, en extrait les

[3] ce n'est pas un hasard, c'est implanté exactement de cette façon ...

sous-blocs, puis les sous-sous-blocs, jusqu'aux blocs élémentaires, à chaque fois par le nom. Selon la façon dont cette structure générale est spécialisée, on peut soit avoir des imbrications figées (un fichier est composé des blocs A, B et C, C étant décomposé en C1 et C2), soit avoir des imbrications variables (un fichier est composé d'un bloc état et d'un bloc liste, un bloc liste étant soit un bloc élémentaire - une tête de liste - soit composé d'un bloc tête élémentaire et d'un bloc queue, la queue étant une liste).

5.1.4 Inclusions

Il est souvent utile de séparer des données ayant trait à des domaines différents (par exemple les vrais senseurs d'un côté, les pseudo-senseurs de l'autre), la syntaxe des fichiers structurés propose donc un mécanisme d'inclusion de fichiers.

Le principe est que seul le nom du fichier *primaire* est fourni aux routines d'accès, mais que si ce fichier contient une chaîne du type : `<autre>`, l'ensemble de cette chaîne (caractères `<` et `>` compris) doit être remplaçé par le contenu du fichier `autre`. Si le nom `autre` commence par un caractère `/` (par exemple `</usr/local/senseurs/ires.fr>`), il est utilisé tel quel. Si le nom ne commence pas par `/`, il est ajouté à la fin du nom du répertoire dans lequel le fichier primaire a été ouvert, pour constituer le nom complet du fichier à ouvrir (avec un `/` entre le nom du répertoire et le nom `autre`).

5.1.5 Héritage

Il est fréquent de prévoir dans un fichier plusieurs configurations pour un même senseur (avec ou sans inhibition, selon plusieurs modes de fonctionnement, en considérant tous les exemplaires montés différemment, ...). Une grande partie des données de chaque configuration est alors similaire, et dupliquer les blocs qui les définissent conduit vite à de très gros fichiers.

Expl. 4 – *duplication d'informations*

```
config_1  { structure_complexe {...}
            valeur {1}
          }
config_2  { structure_complexe {...} # duplication des valeurs de config_1
            valeur {2}
          }

  ...

config_12 { structure_complexe {...} # duplication des valeurs de config_1
            valeur {12}
           }
```

Si le bloc `structure_complexe` est difficile à décrire, le dupliquer dans le fichier est lourd, peu lisible, et peut conduire à des erreurs en cas de modification (il faut bien penser à mettre à jour toutes les occurences distinctes dans le fichier simultanément).

Pour pallier à ce genre de problème, la syntaxe générale propose un mécanisme d'héritage : si au moment de la recherche d'un bloc (ou d'un sous-bloc) on ne trouve pas le nom désiré dans la zone de recherche (le bloc

englobant ou le fichier complet), alors on regarde s'il n'y a pas un bloc élémentaire nommé `=>`[4], le contenu de ce bloc est interprété comme le nom d'un bloc dans lequel on peut puiser les sous-blocs manquants.

L'exemple 4 peut être reproduit en évitant la duplication à l'aide de ce mécanisme :

Expl. 5 – *utilisation de l'héritage pour éviter la duplication*

```
config_1  { structure_complexe { ... }
            valeur { 1 }
          }
config_2  { valeur { 2 }  => { config_1 } }

 ...

config_12 { valeur { 12 } => { config_1 } }
```

Le nom du bloc référencé par le pointeur d'héritage `=>` est interprété au niveau fichier, c'est à dire qu'il est recherché dans tout le fichier et non à l'intérieur d'un bloc particulier qui l'engloberait. Il est cependant possible d'hériter des données d'un sous-bloc si l'on précise son chemin d'accès complet, avec tous ses blocs ancêtres séparés par des points.

Expl. 6 – *héritage de sous-blocs*

```
primaire  { secondaire { tertiaire { i { 1 0 0 }
                                     j { 0 1 0 }
                                     k { 0 0 1 }
                                   }
                         }
          }
repere    { => { primaire.secondaire.tertiaire } } # on hérite de i, j, k
```

Il est important de remarquer que ce mécanisme d'héritage n'est utilisé que pour la recherche de blocs *nommés*, on ne peut pas l'utiliser pour hériter un champ d'un bloc élémentaire[5].

Les mécanismes d'inclusion et d'héritage peuvent être utilisés conjointement pour mettre en place des senseurs complexes possédant des données ajustables profondément enfouies au sein des structures de données. La démarche recommandée est alors de décrire dans un fichier unique la structure complète, en héritant les parties variables d'un bloc de *paramétrage* externe à ce fichier. L'utilisateur voulant utiliser un tel senseur doit alors inclure le fichier générique dans son fichier de senseurs et définir lui-même les données du bloc de paramétrage. Si de plus il décide d'isoler le bloc de paramétrage dans un fichier inclus, il limite considérablement les risques liés à l'édition manuelle de ce fichier pour ajuster le senseur (par exemple pour suivre les ajustements de paramètres selon les télécommandes).

Les exemples 7, 8 et 9 montrent ainsi que dans le (très) complexe fichier `std15.fr` (situé dans le répertoire `exemples` de la distribution), des pointeurs vers des sous-blocs numérotés de `STD15_PARAMETRES.ZPT1` à `STD15_PARAMETRES.ZPT4` ont été prévus pour définir les angles de début des zones de présence terre, des pointeurs vers un sous-bloc `STD15_PARAMETRES.LG` permettant de définir la longueur commune de toutes ces zones. L'utilisateur peut ainsi prendre en compte les capacités de programmation du champ de vue du senseur. D'autre

[4] on peut lire `=>` comme : «*voir aussi*»

[5] ce serait à la fois difficile à mettre en place, peu lisible, et peu utile

part dès lors que le bloc `STD15_PARAMETRES` est défini dans un fichier indépendant inclus par le fichier primaire, la modification des angles de programmation est simple et peu risquée.

Expl. 7 – *extrait de senseurs.fr*

```
<parametrage.fr>
<std15.fr>
IRES_ROULIS { repere { ... } => { STD15_BASE_ROULIS } }
```

Expl. 8 – *extrait de parametrage.fr*

```
STD15_PARAMETRES { ZPT1 { angle { 80.15 } }
                   ZPT2 { angle {  0.00 } }
                   ZPT3 { angle { 81.57 } }
                   ZPT4 { angle {  1.41 } }
                   LG   { angle {  9.84 } }
                 }
```

Expl. 9 – *extrait de std15.fr*

```
STD15_BASE_ROULIS
{ ...
  zone_1
  { { => { STD15_BASE_ROULIS.trace_1_sans_masque } }
    inter
    { { rotation { axe { 0 1 0 } => { STD15_PARAMETRES.ZPT1 } }
        de       { cone { axe { 0 0 1 } angle { 90 }}}
      }
      inter
      { rotation { rotation { axe { 0 1 0 } => { STD15_PARAMETRES.ZPT1 } }
                   de       { axe { 0 1 0 } => { STD15_PARAMETRES.LG   } }
                 }
        de       { cone { axe { 0 0 -1 } angle { 90 } } }
      }
    }
  }
  ...
}
```

5.2 Application aux senseurs

MARMOTTES utilise les fichiers structurés pour représenter les senseurs. Les senseurs sont tous regroupés dans un même fichier, chaque senseur étant décrit dans un bloc de niveau fichier (c'est-à-dire sans bloc père) dont le nom est le nom du senseur.

Les noms des sous-blocs sont des mots-clefs imposés par MARMOTTES et dépendent du type de senseur[6].

Certaines caractéristiques que l'on retrouve dans plusieurs types de senseurs s'appuient sur des éléments de bas niveau, comme les vecteurs ou les champs de vue.

[6] ces mots-clefs sont utilisés à travers le système de traduction (voir [DR5]) on peut donc les modifier en éditant le domaine de traduction `marmottes`

5.2.1 Vecteur

MARMOTTES utilise des vecteurs normés en dimension 3, ces vecteurs peuvent être représentés de plusieurs façon.

cartésiennes : on décrit le vecteur par ses trois coordonnées cartésiennes dans un bloc élémentaire (MARMOTTES se charge de la normalisation) :

```
i { 1 0 0 }
```

sphériques : on décrit le vecteur par ses deux coordonnées sphériques en degrés dans un bloc élémentaire :

```
i { 0 0 }, j { 90 0 }, k { 0 90 }
```

rotation : on décrit le vecteur comme étant l'image par une rotation (sous-bloc `rotation`) d'un autre vecteur (sous-bloc `de`) :

```
u { rotation { ... } de { ... }}
```

5.2.2 Rotations

Comme pour les vecteurs, il existe plusieurs moyens de décrire des rotations en dimension 3, adaptées à des cas différents.

axe et angle : on donne l'axe de la rotation (sous-bloc `axe`) et l'angle en degrés dont doivent tourner les *vecteurs* à qui on applique la rotation (sous-bloc `angle`) :

```
r { axe { 0 0 1 } angle { 25 } }
```

couple et image : on décrit implicitement la rotation par un couple de vecteurs (blocs `v_base_1` et `v_base_2`) et par l'image de ce couple par la rotation (blocs `v_image_1` et `v_image_2`) :

```
r { v_base_1 {1 0 0} v_base_2 {0 1 0} v_image_1 {0 1 0} v_image_2 {0 0 1}}
```

vecteur et image : on décrit implicitement la[7] rotation par un vecteur (sous-bloc `v_base`) et son image (sous-bloc `v_image`) :

```
r { v_base { 1 0 0 } v_image { 0 1 0 } }
```

composition : on décrit la rotation par la composition d'une rotation initiale (sous-bloc `de`) par une rotation opérateur (sous-bloc `rotation`) :

```
r { rotation { ... } de { ... } }
```

image d'un repère : on décrit la rotation par l'image des vecteurs du repère canonique (sous-blocs `i`, `j`, et `k`) :

```
r { i { 0 1 0 } j { -1 0 0 } k { 0 0 1 } }
```

quaternion : on décrit la rotation par les quatre coordonnées de son quaternion dans un bloc élémentaire :

```
r { 0.707 0.0 0.707 0.0 }
```

5.2.3 Champs

On peut définir des zones géométriques sur la sphère unité à partir de zones élémentaires et d'un ensemble d'opérations agissant sur des zones (élémentaires ou non) et produisant des zones plus complexes.

[7] il existe une infinité de rotations transformant `v_base` en `v_image`, MARMOTTES en choisit une arbitrairement

La portion de sphère unitaire la plus simple à définir est la calotte résultant de l'intersection d'un cône dont le sommet est au centre d'une sphère unité avec cette sphère. Une telle zone est entièrement définie par le cône que l'on décrit par son axe et son demi-angle d'ouverture en degrés : `cone { axe { 1 0 0 } angle { 90.0 } }`. L'exemple précédent définit ainsi le demi-espace vérifiant $x > 0$ sous forme d'un cône de demi-angle d'ouverture $\frac{\pi}{2}$.

TAB. 1 – opérations sur les zones géométriques de la sphère unité

nom	arguments		syntaxe
intersection	champ (*c1*)	champ (*c2*)	`zone { {`*c1*`} inter {`*c2*`} }`
réunion	champ (*c1*)	champ (*c2*)	`zone { {`*c1*`} union {`*c2*`} }`
différence	champ (*c1*)	champ (*c2*)	`zone { {`*c1*`} sauf {`*c2*`} }`
marge	scalaire (x)	champ (c)	`zone { marge {`x`} sur {`c`} }`
déplacement	rotation (r)	champ (c)	`zone { rotation {`r`} de {`c`} }`
balayage	rotation (u et a)	champ (c)	`zone { balayage { axe {`u`} angle {`a`} } de {`c`} }`

Les opérations de composition disponibles sont décrites dans la table 1, elles permettent de générer toutes sortes de champs complexes, y compris des champs comportant des trous ou des champs constitués de plusieurs zones séparées les une des autres.

Il est important de remarquer que l'opération de balayage n'accepte pas en argument les rotations générales, mais uniquement la donnée d'un axe et d'un angle. Ceci est dû au fait que si la rotation $(\vec{u}, \alpha)$ est en terme d'opérateur vectoriel indiscernable de la rotation $(-\vec{u}, 2\pi - \alpha)$, ces deux rotations n'engendrent pas les mêmes champs lorsqu'on les utilise dans un balayage[8].

Comme nous l'avons dit lors de la description de la syntaxe générale, l'ordre des sous-blocs dans un bloc composé est indifférent, on peut donc écrire `{{xxxx} inter {yyyy}}` comme cela est indiqué dans la table 1, mais on peut également écrire `{inter {yyyy} {xxxx}}`. La première écriture semble cependant plus lisible.

Expl. 10 – *double dièdre fixe*

```
double_diedre_contenant_I
{ { { cone { axe { 0  65 } angle { 90 } } }
    inter
    { cone { axe { 0 -65 } angle { 90 } } }
  }
  inter
  { { cone { axe {  65 0 } angle { 90 } } }
    inter
    { cone { axe { -65 0 } angle { 90 } } }
  }
}
```

L'exemple 10 montre ainsi comment définir un champ de vue en forme de double dièdre de 50° d'ouverture, l'exemple 11 montrant comment rendre ce même champ de vue paramétrable par un héritage de l'angle de demi-ouverture.

[8] les versions de MARMOTTES antérieures à la version 9.0 utilisaient des rotations quelconques, ce qui conduisait à des erreurs dès que l'amplitude des rotations dépassait π

Expl. 11 – *double dièdre paramétrable*

```
demi { angle { 25.0 } } # <-- demi-angle d'ouverture paramétrable

double_diedre_contenant_I
{ { { cone { axe   { rotation { axe { 0  1 0 } => { demi } } de { 0 0  1 } }
             angle { 90 }
           }
    }
    inter
    { cone { axe   { rotation { axe { 0 -1 0 } => { demi } } de { 0 0 -1 } }
             angle { 90 }
           }
    }
  }
  inter
  { { cone { axe   { rotation { axe { 0 0 -1 } => { demi } } de { 0  1 0 } }
             angle { 90 }
           }
    }
    inter
    { cone { axe   { rotation { axe { 0 0  1 } => { demi } } de { 0 -1 0 } }
             angle { 90 }
           }
    }
  }
}
```

Le champ `double_diedre_contenant_I` de l'exemple 10 est défini par intersection de deux champs. Le premier de ces sous-champs est dans un bloc sans nom et correspond à un dièdre de 50° d'ouverture ($\alpha = 25°$) autour de l'arête $\vec{j}$ et contenant $\vec{i}$, le second de ces sous-champs est dans le bloc de nom `inter` et correspond à un dièdre de 50° d'ouverture ($\alpha = 25°$) autour de l'arête $\vec{k}$ et contenant également $\vec{i}$. L'intersection donne le double dièdre de la figure 1.

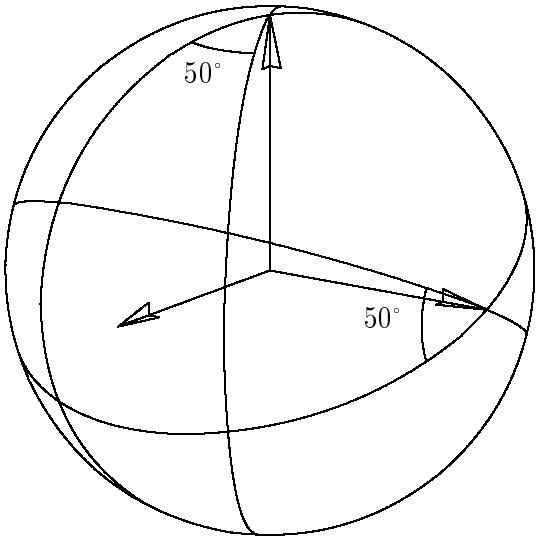
Les champs sont donc des structures récursives pour lesquelles les fonctions de lecture découvrent petit à petit les imbrications, en testant l'existence de sous-bloc nommés `inter`, `union`, `sauf`, `marge`, `balayage`, et `rotation`[9].

5.2.4 Parcelles

Les champs permettent de définir des zones géométriques suffisantes pour déterminer la visibilité d'astres ponctuels (lune ou soleil). Les senseurs de limbes observent quant à eux la totalité du limbe du corps central, et combinent plusieurs mesures élémentaires dans des zones séparées de leur champ de vue pour produire une mesure globale. Il faut alors définir des zones *logiques* sur la sphère unité et des opérateurs entre ces zones, pour

[9] les mots clefs `inter`, `union`, `sauf`, `marge`, `balayage`, et `rotation` sont utilisés au travers du mécanisme de traduction, ils appartiennent au domaine `marmottes`

FIG. 1 – champ de vue en double dièdre



expliciter des comportements du type : le senseur voit le corps central si le limbe est visible dans les zones 1 et 2, ou s'il est visible dans les zones 3 et 4.

MARMOTTES introduit pour couvrir ces besoins la notion de *parcelles*. Une parcelle peut être élémentaire (c'est alors un champ), ou être une combinaison logique de parcelles. Les opérations de composition disponibles sont décrites dans la table 2.

Lorsqu'une parcelle est utilisée pour tester la visibilité d'une cible considérés comme ponctuelle (le soleil, la lune ou le moment cinétique de l'orbite par exemple), la distinction entre les deux types de réunions s'évanouit. Cette interprétation permet de s'abstenir de distinguer champs et parcelles. Dans la pratique, toutes les zones sur la sphère décrites dans les fichiers de description des senseurs par MARMOTTES sont lues comme des parcelles, et seuls les senseurs terre utilisent toutes les possibilités qui leur sont offertes.

TAB. 2 – opérations sur les zones logiques de la sphère unité

nom	arguments		syntaxe
réunion *et*	parcelle (*p1*)	parcelle (*p2*)	`zone { {`*p1*`} et {`*p2*`} }`
réunion *ou*	parcelle (*p1*)	parcelle (*p2*)	`zone { {`*p1*`} ou {`*p2*`} }`
marge	scalaire (x)	parcelle (p)	`zone { marge {`x`} sur {`p`} }`

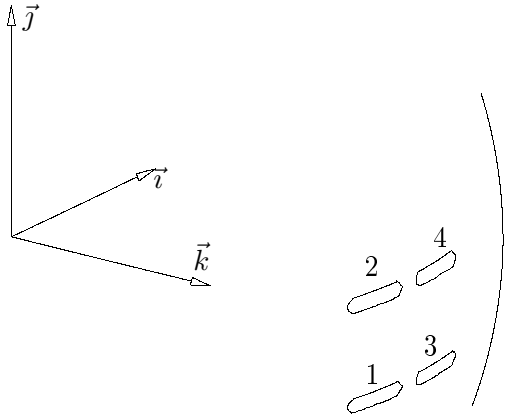
L'exemple 12 montre ainsi comment on peut décrire la logique d'un senseur de limbe et associer un champ de vue pour la mesure roulis et un champ de vue pour la mesure tangage, en supposant que les champs `scan_1` à `scan_4` sont numérotés conformément à la figure 2.

Expl. 12 – *définition de parcelles*

```
champ_de_vue_ROULIS
{ { { => { scan_1 } } et { => { scan_2 } } }
  ou
  { { => { scan_3 } } et { => { scan_4 } } }
}

champ_de_vue_TANGAGE
{ { { => { scan_1 } } et { => { scan_3 } } }
  ou
  { { => { scan_2 } } et { => { scan_4 } } }
}
```

FIG. 2 – champ de vue d'un senseur de limbe à quatre scans



5.2.5 Stations sol

Les stations sols peuvent être décrites dans une structure définie par l'exemple 13 :

Expl. 13 – *structure station*

```
station { longitude    {   1.49939883 }
          latitude     {  43.42869186 }
          altitude     { 261.58       }

          pression     { 1000 }
          temperature  { 15   }
          hygrometrie  { 80   }

          masque       { 0 10 }

        }
```

La description des masques d'antenne des stations sols est constituée d'une simple série de couples azimut/site limite sans aucune structuration. Si la station a un masque constant (par exemple un site minimum d'observation de 10 °), il suffit de donner un seul couple et la valeur de l'azimut n'a pas d'importance :

```
masque { 0 10 }
```

Si le masque dépend de l'azimut d'observation, il faut donner plusieurs points, MARMOTTES interpolera linéairement entre ces points en fermant la courbe elle-même :

Expl. 14 – *définition de masque*

```
masque { # KRN ESX 76
           0.00   0.20
           5.00   0.00
          10.00   0.10
          15.00  18.50
          20.00   0.00
          25.00   0.00
          28.00   2.00

                ...

         350.00   0.20
         355.00   0.20
       }
```

5.2.6 Points d'échantillonnage des senseurs de gain d'antenne

La description des points d'échantillonnage utilisés pour les senseurs de gain d'antenne est constituée d'une simple série de couples angle/gain sans aucune structuration. Si l'antenne a un gain constant, il suffit de donner un seul couple et la valeur de l'angle n'a pas d'importance :

```
echantillon { 0 -3.0 }
```

Si le gain dépend de l'angle de dépointage, il faut donner plusieurs points, MARMOTTES interpolera linéairement entre ces points en considérant que le gain est constant entre 0 et le premier angle ainsi qu'entre le dernier angle et π.

Expl. 15 – *définition d'échantillon*

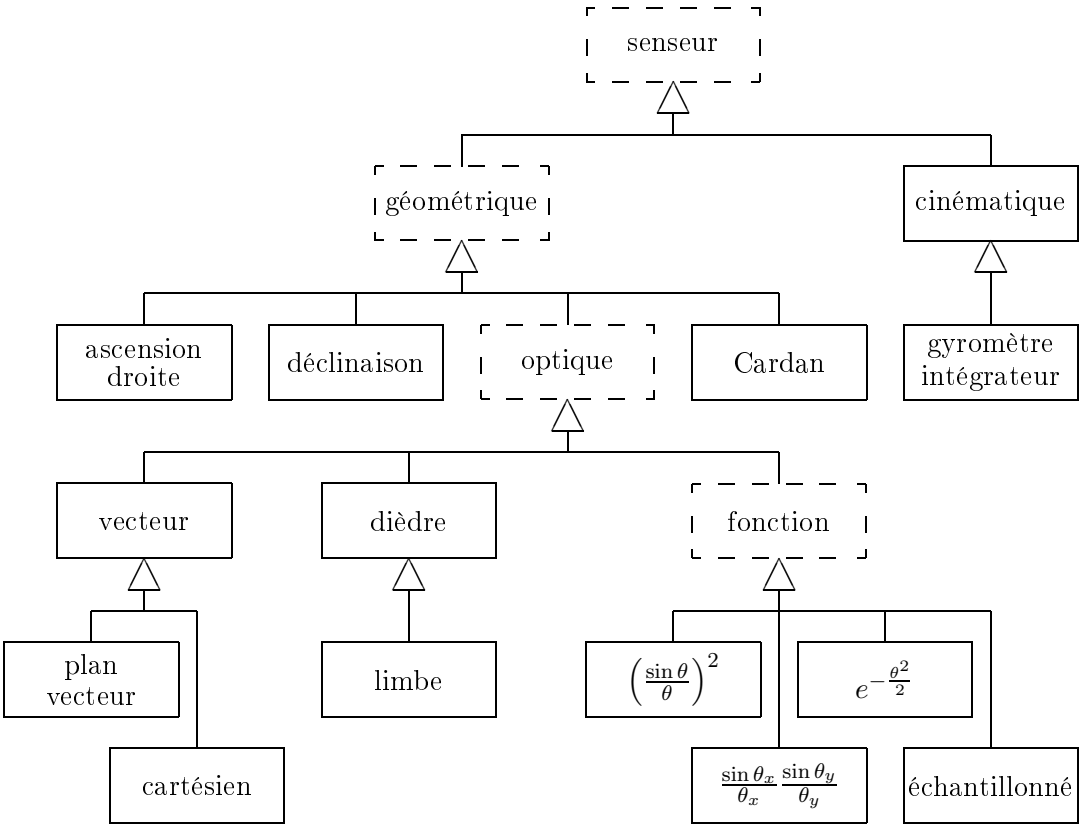
```
echantillon {
              10   0.0
              20  -1.0
              25  -3.0
              30 -10.0
            }
```

5.2.7 Hiérarchie de description des senseurs

Les senseurs sont implantés dans le code par une hiérarchie de classes C++ dont la racine est la classe senseur. D'un point de vue informatique, cette architecture permet de factoriser du code commun dans des

classes de base, et de spécialiser certaines notions par du code propre aux classes dérivées [10].

FIG. 3 – Hiérarchie des senseurs



Les sous-blocs du bloc senseur décrivant les propriétés de base sont lus de la même façon dans les classes dérivées que dans les classes de base. Par contre les spécificités des classes dérivées sont décrites dans les fichiers par des sous-blocs supplémentaires spécifiques à chaque classe. L'ordre des sous-blocs est non significatif, il s'ensuit donc que la structuration des données entre données générales intervenant au niveau de la classe de base et données spécifiques à un type particulier de senseur ne transparaît pas dans le fichier des senseurs (voir exemple 16). Les tables qui suivent masquent cette structuration, au prix de la duplication dans toutes les tables des informations communes (par exemple le sous-bloc `repere`, qui est général).

Expl. 16 – *indépendance de la syntaxe par rapport aux classes*

```
BASS-2_PITCH
{ type         { diedre }  # <-- bloc général hérité de la classe senseur
  cible        { Soleil }  # <-- bloc intermédiaire hérité
                           #     de la classe senseur géométrique
  axe_sensible { 0  1  0 } # <-- bloc spécifique à la classe senseur Dièdre
  precision    { 1.0 }     # <-- bloc général hérité de la classe senseur
 ...
}
```

[10] ainsi la notion de visibilité des senseurs de limbe considère un limbe et la structure logique des parcelles, alors que pour les autres senseurs optiques on se contente de vérifier la présence d'un point dans un champ

5.2.8 Senseurs reconnus par Marmottes

Les tables suivantes décrivent tous les attributs qui doivent être définis pour chaque type de senseur. La signification de ces attributs est décrite dans le document [DR1].

TAB. 3: liste des cibles reconnues pour les senseurs optiques

mots-clef	description
`soleil`	direction du soleil vue du satellite. Cette cible est affectée par les éclipses engendrées par la lune ou la terre
`soleil-sans-eclipse`	cible comparable à `soleil`, mais qui n'est pas affectée par les éclipses
`corps-central-soleil`	direction du soleil vue du corps central (c'est à dire sans corriger la parallaxe liée à la position du satellite)
`lune`	direction de la lune vue du satellite. Cette cible est affectée par les éclipses engendrées par la terre
`lune-sans-eclipse`	cible comparable à `lune`, mais qui n'est pas affectée par les éclipses
`corps-central`	direction du centre du corps central
`terre`	direction du centre terre
`nadir`	direction du point qui voit le satellite exactement à sa verticale, en tenant compte de l'ellipticité du corps central
`vitesse-sol-apparente`	direction du déplacement apparent du point au sol situé dans la direction d'observation, en tenant compte de la vitesse de rotation du corps central et de la vitesse propre du satellite, la direction d'observation est elle-même décrite dans un bloc séparé
`polaris`	direction de l'étoile polaire
`canopus`	direction de l'étoile Canope
`vitesse`	direction de la vitesse intantanée du satellite
`moment`	direction du moment orbital du satellite ($\vec{p} \wedge \vec{v}$)
`devant`	direction perpendiculaire à la fois à la direction du centre attracteur et à la direction du moment (positive dans le sens de la vitesse)
`position`	ce mot-clef indique que l'utilisateur donnera lui-même au cours de l'exécution la position de la cible par rapport au centre attracteur, et qu'il faudra lui appliquer une correction de parallaxe (il s'agit typiquement d'un point au sol variable ou d'un autre satellite). Se reporter au § 11.12 pour des précisions sur le repère d'expression de la position de la cible.
`position-sans-eclipse`	cible comparable à `position`, mais qui n'est pas affectée par les éclipses
`direction`	ce mot-clef indique que l'utilisateur donnera lui-même au cours de l'exécution la direction de la cible et qu'il faudra la prendre telle qu'elle (il s'agit typiquement d'une étoile pour un senseur stellaire pouvant gérer un nombre très important d'étoiles différentes). Se reporter au § 11.12 pour des précisions sur le repère d'expression de la direction de la cible.

à suivre ...

TAB. 3: liste des cibles reconnues pour les senseurs optiques (suite)

mots-clef	description
`direction-sans-eclipse`	cible comparable à `direction`, mais qui n'est pas affectée par les éclipses
`station`	direction d'une station sol, la station étant elle-même décrite dans un bloc séparé
`polaris-sans-eclipse`	cible comparable à `polaris`, mais qui n'est pas affectée par les éclipses
`canopus-sans-eclipse`	cible comparable à `canopus`, mais qui n'est pas affectée par les éclipses

TAB. 4 – senseur d'ascension droite

nom du bloc	type	description
`type`	mot-clef	Identificateur permettant de reconnaître le type de senseur, vaut obligatoirement `ascension_droite`.
`precision`	réel	Précision du senseur en degrés, utilisée pour la convergence.
`repere`	rotation	Définition du repère senseur, le plus pratique est d'utiliser la syntaxe `{i{...} j{...} k{...}}`, on donne alors les coordonnées *en repère satellite* des axes canoniques du senseur.
`axe_calage`	vecteur	Bloc *optionnel* définissant un axe de calage autour duquel le senseur peut tourner (voir 11.11).
`observe`	vecteur	Coordonnées *en repère senseur* du vecteur dont on calcule l'ascension droite

TAB. 5 – senseur de déclinaison

nom du bloc	type	description
`type`	mot-clef	Identificateur permettant de reconnaître le type de senseur, vaut obligatoirement `declinaison`.
`precision`	réel	Précision du senseur en degrés, utilisée pour la convergence.
`repere`	rotation	Définition du repère senseur, le plus pratique est d'utiliser la syntaxe `{i{...} j{...} k{...}}`, on donne alors les coordonnées *en repère satellite* des axes canoniques du senseur.
`axe_calage`	vecteur	Bloc *optionnel* définissant un axe de calage autour duquel le senseur peut tourner (voir 11.11).
`observe`	vecteur	Coordonnées *en repère senseur* du vecteur dont on calcule la déclinaison.

TAB. 6 – senseur vecteur

nom du bloc	type	description
`type`	mot-clef	Identificateur permettant de reconnaître le type de senseur, vaut obligatoirement `vecteur`.
`precision`	réel	Précision du senseur en degrés, utilisée pour la convergence.
`repere`	rotation	Définition du repère senseur, le plus pratique est d'utiliser la syntaxe `{i{...} j{...} k{...}}`, on donne alors les coordonnées *en repère satellite* des axes canoniques du senseur.
`axe_calage`	vecteur	Bloc *optionnel* définissant un axe de calage autour duquel le senseur peut tourner (voir 11.11).
`cible`	mot-clef	Cible du senseur. La table 3, page 27 donne la liste des valeurs reconnues.
`observe`	vecteur	Coordonnées de la direction d'observation du senseur *en repère senseur*, uniquement si `cible` vaut `vitesse-sol-apparente`
`station`	structure	Définition de la station observée, uniquement si `cible` vaut `station`.
`champ_de_vue`	parcelle	Champ de vue du senseur *en repère senseur*.
`champ_d_inhibition_corps_central`	parcelle	Bloc *optionnel* définissant le champ d'inhibition du senseur par un astre de type corps central *en repère senseur*. Vide par défaut.
`champ_d_inhibition_soleil`	parcelle	Bloc *optionnel* définissant le champ d'inhibition par le soleil du senseur *en repère senseur*. Vide par défaut.
`marge_eclipse_soleil`	réel	Bloc *optionnel* définissant la marge angulaire sur la suppression des inhibitions pour les passages en éclipse.
`champ_d_inhibition_lune`	parcelle	Bloc *optionnel* définissant le champ d'inhibition par la lune du senseur *en repère senseur*. Vide par défaut.
`marge_eclipse_lune`	réel	Bloc *optionnel* définissant la marge angulaire sur la suppression des inhibitions pour les passages en éclipse.
`seuil_phase_lune`	scalaire	Bloc *optionnel* définissant le seuil sur l'angle Soleil/Satellite/Lune au *dessous* duquel la lune devient gênante (0 signifie que la lune n'est jamais gênante, 180 signifie qu'elle est toujours gênante).
`reference`	vecteur	Vecteur de référence des mesures.

TAB. 7 – senseur plan-vecteur

nom du bloc	type	description
`type`	mot-clef	Identificateur permettant de reconnaître le type de senseur, vaut obligatoirement `plan_vecteur`.
`precision`	réel	Précision du senseur en degrés, utilisée pour la convergence.
`repere`	rotation	Définition du repère senseur, le plus pratique est d'utiliser la syntaxe `{i{...} j{...} k{...}}`, on donne alors les coordonnées *en repère satellite* des axes canoniques du senseur.
`axe_calage`	vecteur	Bloc *optionnel* définissant un axe de calage autour duquel le senseur peut tourner (voir 11.11).
`cible`	mot-clef	Cible du senseur. La table 3, page 27 donne la liste des valeurs reconnues.
`observe`	vecteur	Coordonnées de la direction d'observation du senseur *en repère senseur*, uniquement si `cible` vaut `vitesse-sol-apparente`
`station`	structure	Définition de la station observée, uniquement si `cible` vaut `station`.
`champ_de_vue`	parcelle	Champ de vue du senseur *en repère senseur*.
`champ_d_inhibition_corps_central`	parcelle	Bloc *optionnel* définissant le champ d'inhibition du senseur par un astre de type corps central *en repère senseur*. Vide par défaut.
`champ_d_inhibition_soleil`	parcelle	Bloc *optionnel* définissant le champ d'inhibition par le soleil du senseur *en repère senseur*. Vide par défaut.
`marge_eclipse_soleil`	réel	Bloc *optionnel* définissant la marge angulaire sur la suppression des inhibitions pour les passages en éclipse.
`champ_d_inhibition_lune`	parcelle	Bloc *optionnel* définissant le champ d'inhibition par la lune du senseur *en repère senseur*. Vide par défaut.
`marge_eclipse_lune`	réel	Bloc *optionnel* définissant la marge angulaire sur la suppression des inhibitions pour les passages en éclipse.
`seuil_phase_lune`	scalaire	Bloc *optionnel* définissant le seuil sur l'angle Soleil/Satellite/Lune au *dessous* duquel la lune devient gênante (0 signifie que la lune n'est jamais gênante, 180 signifie qu'elle est toujours gênante).
`normale_reference`	vecteur	Vecteur de référence des mesures (normale au plan des mesures nulles).

TAB. 8 – senseur cartésien

nom du bloc	type	description
`type`	mot-clef	Identificateur permettant de reconnaître le type de senseur, vaut obligatoirement `cartesien`.
`precision`	réel	Précision du senseur en degrés, utilisée pour la convergence.
`repere`	rotation	Définition du repère senseur, le plus pratique est d'utiliser la syntaxe `{i{...} j{...} k{...}}`, on donne alors les coordonnées *en repère satellite* des axes canoniques du senseur.
`axe_calage`	vecteur	Bloc *optionnel* définissant un axe de calage autour duquel le senseur peut tourner (voir 11.11).
`cible`	mot-clef	Cible du senseur. La table 3, page 27 donne la liste des valeurs reconnues.
`observe`	vecteur	Coordonnées de la direction d'observation du senseur *en repère senseur*, uniquement si `cible` vaut `vitesse-sol-apparente`
`station`	structure	Définition de la station observée, uniquement si `cible` vaut `station`.
`champ_de_vue`	parcelle	Champ de vue du senseur *en repère senseur*.
`champ_d_inhibition_corps_central`	parcelle	Bloc *optionnel* définissant le champ d'inhibition du senseur par un astre de type corps central *en repère senseur*. Vide par défaut.
`champ_d_inhibition_soleil`	parcelle	Bloc *optionnel* définissant le champ d'inhibition par le soleil du senseur *en repère senseur*. Vide par défaut.
`marge_eclipse_soleil`	réel	Bloc *optionnel* définissant la marge angulaire sur la suppression des inhibitions pour les passages en éclipse.
`champ_d_inhibition_lune`	parcelle	Bloc *optionnel* définissant le champ d'inhibition par la lune du senseur *en repère senseur*. Vide par défaut.
`marge_eclipse_lune`	réel	Bloc *optionnel* définissant la marge angulaire sur la suppression des inhibitions pour les passages en éclipse.
`seuil_phase_lune`	scalaire	Bloc *optionnel* définissant le seuil sur l'angle Soleil/Satellite/Lune au *dessous* duquel la lune devient gênante (0 signifie que la lune n'est jamais gênante, 180 signifie qu'elle est toujours gênante).
`reference`	vecteur	Coordonnées *en repère senseur* du vecteur le long duquel on mesure la coordonnée.

TAB. 9 – senseur dièdre

nom du bloc	type	description
`type`	mot-clef	Identificateur permettant de reconnaître le type de senseur, vaut obligatoirement `diedre`.
`precision`	réel	Précision du senseur en degrés, utilisée pour la convergence.
`repere`	rotation	Définition du repère senseur, le plus pratique est d'utiliser la syntaxe `{i{...} j{...} k{...}}`, on donne alors les coordonnées *en repère satellite* des axes canoniques du senseur.
`axe_calage`	vecteur	Bloc *optionnel* définissant un axe de calage autour duquel le senseur peut tourner (voir 11.11).
`cible`	mot-clef	Cible du senseur. La table 3, page 27 donne la liste des valeurs reconnues.
`observe`	vecteur	Coordonnées de la direction d'observation du senseur *en repère senseur*, uniquement si `cible` vaut `vitesse-sol-apparente`
`station`	structure	Définition de la station observée, uniquement si `cible` vaut `station`.
`champ_de_vue`	parcelle	Champ de vue du senseur *en repère senseur*.
`champ_d_inhibition_corps_central`	parcelle	Bloc *optionnel* définissant le champ d'inhibition du senseur par un astre de type corps central *en repère senseur*. Vide par défaut.
`champ_d_inhibition_soleil`	parcelle	Bloc *optionnel* définissant le champ d'inhibition par le soleil du senseur *en repère senseur*. Vide par défaut.
`marge_eclipse_soleil`	réel	Bloc *optionnel* définissant la marge angulaire sur la suppression des inhibitions pour les passages en éclipse.
`champ_d_inhibition_lune`	parcelle	Bloc *optionnel* définissant le champ d'inhibition par la lune du senseur *en repère senseur*. Vide par défaut.
`marge_eclipse_lune`	réel	Bloc *optionnel* définissant la marge angulaire sur la suppression des inhibitions pour les passages en éclipse.
`seuil_phase_lune`	scalaire	Bloc *optionnel* définissant le seuil sur l'angle Soleil/Satellite/Lune au *dessous* duquel la lune devient gênante (0 signifie que la lune n'est jamais gênante, 180 signifie qu'elle est toujours gênante).
`axe_sensible`	vecteur	Coordonnées de l'axe sensible du senseur *en repère senseur*.
`reference_zero`	vecteur	Coordonnées *en repère senseur* d'un vecteur du plan des mesures nulles.

TAB. 10 – senseur de limbe

nom du bloc	type	description
`type`	mot-clef	Identificateur permettant de reconnaître le type de senseur, vaut obligatoirement `limbe`.
`precision`	réel	Précision du senseur en degrés, utilisée pour la convergence.
`repere`	rotation	Définition du repère senseur, le plus pratique est d'utiliser la syntaxe `{i{...} j{...} k{...}}`, on donne alors les coordonnées *en repère satellite* des axes canoniques du senseur.
`axe_calage`	vecteur	Bloc *optionnel* définissant un axe de calage autour duquel le senseur peut tourner (voir 11.11).
`champ_de_vue`	parcelle	Champ de vue du senseur *en repère senseur*.
`champ_d_inhibition_corps_central`	parcelle	Bloc *optionnel* définissant le champ d'inhibition du senseur par un astre de type corps central *en repère senseur*. Vide par défaut.
`champ_d_inhibition_soleil`	parcelle	Bloc *optionnel* définissant le champ d'inhibition par le soleil du senseur *en repère senseur*. Vide par défaut.
`marge_eclipse_soleil`	réel	Bloc *optionnel* définissant la marge angulaire sur la suppression des inhibitions pour les passages en éclipse.
`champ_d_inhibition_lune`	parcelle	Bloc *optionnel* définissant le champ d'inhibition par la lune du senseur *en repère senseur*. Vide par défaut.
`marge_eclipse_lune`	réel	Bloc *optionnel* définissant la marge angulaire sur la suppression des inhibitions pour les passages en éclipse.
`seuil_phase_lune`	scalaire	Bloc *optionnel* définissant le seuil sur l'angle Soleil/Satellite/Lune au *dessous* duquel la lune devient gênante (0 signifie que la lune n'est jamais gênante, 180 signifie qu'elle est toujours gênante).
`axe_sensible`	vecteur	Coordonnées de l'axe sensible du senseur *en repère senseur*.
`reference_zero`	vecteur	Coordonnées *en repère senseur* d'un vecteur du plan des mesures nulles.

TAB. 11 – senseur échantillonné à une dimension

nom du bloc	type	description
`type`	mot-clef	Identificateur permettant de reconnaître le type de senseur, vaut obligatoirement `gain_echantillonne_1D` .
`precision`	réel	Précision du senseur en degrés, utilisée pour la convergence.
`repere`	rotation	Définition du repère senseur, le plus pratique est d'utiliser la syntaxe `{i{...} j{...} k{...}}`, on donne alors les coordonnées *en repère satellite* des axes canoniques du senseur.
`axe_calage`	vecteur	Bloc *optionnel* définissant un axe de calage autour duquel le senseur peut tourner (voir 11.11).
`cible`	mot-clef	Cible du senseur. La table 3, page 27 donne la liste des valeurs reconnues.
`observe`	vecteur	Coordonnées de la direction d'observation du senseur *en repère senseur*, uniquement si `cible` vaut `vitesse-sol-apparente`
`station`	structure	Définition de la station observée, uniquement si `cible` vaut `station`.
`champ_de_vue`	parcelle	Champ de vue du senseur *en repère senseur*.
`champ_d_inhibition_corps_central`	parcelle	Bloc *optionnel* définissant le champ d'inhibition du senseur par un astre de type corps central *en repère senseur*. Vide par défaut.
`champ_d_inhibition_soleil`	parcelle	Bloc *optionnel* définissant le champ d'inhibition par le soleil du senseur *en repère senseur*. Vide par défaut.
`marge_eclipse_soleil`	réel	Bloc *optionnel* définissant la marge angulaire sur la suppression des inhibitions pour les passages en éclipse.
`champ_d_inhibition_lune`	parcelle	Bloc *optionnel* définissant le champ d'inhibition par la lune du senseur *en repère senseur*. Vide par défaut.
`marge_eclipse_lune`	réel	Bloc *optionnel* définissant la marge angulaire sur la suppression des inhibitions pour les passages en éclipse.
`seuil_phase_lune`	scalaire	Bloc *optionnel* définissant le seuil sur l'angle Soleil/Satellite/Lune au *dessous* duquel la lune devient gênante (0 signifie que la lune n'est jamais gênante, 180 signifie qu'elle est toujours gênante).
`axe`	vecteur	Coordonnées *en repère senseur* de la direction du centre de l'échantillonnage.
`origine`	vecteur	Coordonnées *en repère senseur* d'un vecteur définissant avec l'axe le plan d'azimut. La fonction n'étant échantillonnée que selon θ, cet axe peut être n'importe quel vecteur non colinéaire à l'axe.
`echantillon`	tableau	Points d'échantillonnage de la fonction (voir l'exemple 15, page 25)

TAB. 12 – senseur sinus cardinal carré

nom du bloc	type	description
`type`	mot-clef	Identificateur permettant de reconnaître le type de senseur, vaut obligatoirement `gain_sinus_cardinal_2` .
`precision`	réel	Précision du senseur en degrés, utilisée pour la convergence.
`repere`	rotation	Définition du repère senseur, le plus pratique est d'utiliser la syntaxe `{i{...} j{...} k{...}}` , on donne alors les coordonnées *en repère satellite* des axes canoniques du senseur.
`axe_calage`	vecteur	Bloc *optionnel* définissant un axe de calage autour duquel le senseur peut tourner (voir 11.11).
`cible`	mot-clef	Cible du senseur. La table 3, page 27 donne la liste des valeurs reconnues.
`observe`	vecteur	Coordonnées de la direction d'observation du senseur *en repère senseur*, uniquement si `cible` vaut `vitesse-sol-apparente`
`station`	structure	Définition de la station observée, uniquement si `cible` vaut `station`.
`champ_de_vue`	parcelle	Champ de vue du senseur *en repère senseur*.
`champ_d_inhibition_corps_central`	parcelle	Bloc *optionnel* définissant le champ d'inhibition du senseur par un astre de type corps central *en repère senseur*. Vide par défaut.
`champ_d_inhibition_soleil`	parcelle	Bloc *optionnel* définissant le champ d'inhibition par le soleil du senseur *en repère senseur*. Vide par défaut.
`marge_eclipse_soleil`	réel	Bloc *optionnel* définissant la marge angulaire sur la suppression des inhibitions pour les passages en éclipse.
`champ_d_inhibition_lune`	parcelle	Bloc *optionnel* définissant le champ d'inhibition par la lune du senseur *en repère senseur*. Vide par défaut.
`marge_eclipse_lune`	réel	Bloc *optionnel* définissant la marge angulaire sur la suppression des inhibitions pour les passages en éclipse.
`seuil_phase_lune`	scalaire	Bloc *optionnel* définissant le seuil sur l'angle Soleil/Satellite/Lune au *dessous* duquel la lune devient gênante (0 signifie que la lune n'est jamais gênante, 180 signifie qu'elle est toujours gênante).
`axe`	vecteur	Coordonnées *en repère senseur* de la direction de gain maximal.
`origine`	vecteur	Coordonnées *en repère senseur* d'un vecteur définissant avec l'axe le plan d'azimut. La fonction $10 \times \frac{\log K\left(\frac{\sin(\theta/\theta_0)}{(\theta/\theta_0)}\right)^2}{\log 10}$ ne dépendant que de θ, cet axe peut être n'importe quel vecteur non colinéaire à l'axe.
`maximum`	réel	valeur maximale en dB du gain (cette valeur est atteinte sur l'axe).
`angle_3dB`	réel	angle définissant la largeur de lobe à 3 dB : lorsque θ prend

TAB. 13 – senseur sinus cardinal bidimensionnel

nom du bloc	type	description
`type`	mot-clef	Identificateur permettant de reconnaître le type de senseur, vaut obligatoirement `gain_sinus_cardinal_xy` .
`precision`	réel	Précision du senseur en degrés, utilisée pour la convergence.
`repere`	rotation	Définition du repère senseur, le plus pratique est d'utiliser la syntaxe `{i{...} j{...} k{...}}` , on donne alors les coordonnées *en repère satellite* des axes canoniques du senseur.
`axe_calage`	vecteur	Bloc *optionnel* définissant un axe de calage autour duquel le senseur peut tourner (voir 11.11).
`cible`	mot-clef	Cible du senseur. La table 3, page 27 donne la liste des valeurs reconnues.
`observe`	vecteur	Coordonnées de la direction d'observation du senseur *en repère senseur*, uniquement si `cible` vaut `vitesse-sol-apparente`
`station`	structure	Définition de la station observée, uniquement si `cible` vaut `station`.
`champ_de_vue`	parcelle	Champ de vue du senseur *en repère senseur*.
`champ_d_inhibition_corps_central`	parcelle	Bloc *optionnel* définissant le champ d'inhibition du senseur par un astre de type corps central *en repère senseur*. Vide par défaut.
`champ_d_inhibition_soleil`	parcelle	Bloc *optionnel* définissant le champ d'inhibition par le soleil du senseur *en repère senseur*. Vide par défaut.
`marge_eclipse_soleil`	réel	Bloc *optionnel* définissant la marge angulaire sur la suppression des inhibitions pour les passages en éclipse.
`champ_d_inhibition_lune`	parcelle	Bloc *optionnel* définissant le champ d'inhibition par la lune du senseur *en repère senseur*. Vide par défaut.
`marge_eclipse_lune`	réel	Bloc *optionnel* définissant la marge angulaire sur la suppression des inhibitions pour les passages en éclipse.
`seuil_phase_lune`	scalaire	Bloc *optionnel* définissant le seuil sur l'angle Soleil/Satellite/Lune au *dessous* duquel la lune devient gênante (0 signifie que la lune n'est jamais gênante, 180 signifie qu'elle est toujours gênante).
`axe`	vecteur	Coordonnées *en repère senseur* de la direction de gain maximal.
`origine`	vecteur	Coordonnées *en repère senseur* d'un vecteur définissant avec l'axe le plan d'azimut. La fonction $10 \times \frac{\log K\left(\frac{\sin(\theta_x/\theta_{0,x})}{(\theta_x/\theta_{0,x})}\frac{\sin(\theta_y/\theta_{0,y})}{(\theta_y/\theta_{0,y})}\right)}{\log 10}$ ne dépendant indépendamment de θ_x et θ_y, cet axe doit être calé correctement par rapport au lobe d'antenne (il donne la direction de l'axe $\vec{X}$).
`maximum`	réel	valeur maximale en dB du gain (cette valeur est atteinte sur l'axe).

TAB. 14 – senseur gaussien

nom du bloc	type	description
`type`	mot-clef	Identificateur permettant de reconnaître le type de senseur, vaut obligatoirement `gain_gauss`.
`precision`	réel	Précision du senseur en degrés, utilisée pour la convergence.
`repere`	rotation	Définition du repère senseur, le plus pratique est d'utiliser la syntaxe `{i{...} j{...} k{...}}`, on donne alors les coordonnées *en repère satellite* des axes canoniques du senseur.
`axe_calage`	vecteur	Bloc *optionnel* définissant un axe de calage autour duquel le senseur peut tourner (voir 11.11).
`cible`	mot-clef	Cible du senseur. La table 3, page 27 donne la liste des valeurs reconnues.
`observe`	vecteur	Coordonnées de la direction d'observation du senseur *en repère senseur*, uniquement si `cible` vaut `vitesse-sol-apparente`
`station`	structure	Définition de la station observée, uniquement si `cible` vaut `station`.
`champ_de_vue`	parcelle	Champ de vue du senseur *en repère senseur*.
`champ_d_inhibition_corps_central`	parcelle	Bloc *optionnel* définissant le champ d'inhibition du senseur par un astre de type corps central *en repère senseur*. Vide par défaut.
`champ_d_inhibition_soleil`	parcelle	Bloc *optionnel* définissant le champ d'inhibition par le soleil du senseur *en repère senseur*. Vide par défaut.
`marge_eclipse_soleil`	réel	Bloc *optionnel* définissant la marge angulaire sur la suppression des inhibitions pour les passages en éclipse.
`champ_d_inhibition_lune`	parcelle	Bloc *optionnel* définissant le champ d'inhibition par la lune du senseur *en repère senseur*. Vide par défaut.
`marge_eclipse_lune`	réel	Bloc *optionnel* définissant la marge angulaire sur la suppression des inhibitions pour les passages en éclipse.
`seuil_phase_lune`	scalaire	Bloc *optionnel* définissant le seuil sur l'angle Soleil/Satellite/Lune au *dessous* duquel la lune devient gênante (0 signifie que la lune n'est jamais gênante, 180 signifie qu'elle est toujours gênante).
`axe`	vecteur	Coordonnées *en repère senseur* de la direction de gain maximal.
`origine`	vecteur	Coordonnées *en repère senseur* d'un vecteur d'éfinissant avec l'axe le plan d'azimut. La fonction $10 \times \frac{\log K e^{\frac{-\theta^2}{2\theta_0^2}}}{\log 10}$ ne dépendant que de θ, cet axe peut être n'importe quel vecteur non colinéaire à l'axe.
`maximum`	réel	valeur maximale en dB du gain (cette valeur est atteinte dur l'axe)
`angle_3dB`	réel	angle correspondant définissant la largeur de lobe à 3 dB :

TAB. 15 – senseur de Cardan

nom du bloc	type	description
`type`	mot-clef	Identificateur permettant de reconnaître le type de senseur, vaut obligatoirement `cardan`.
`precision`	réel	Précision du senseur en degrés, utilisée pour la convergence.
`repere`	rotation	Définition du repère senseur, le plus pratique est d'utiliser la syntaxe `{i{...} j{...} k{...}}`, on donne alors les coordonnées *en repère satellite* des axes canoniques du senseur.
`axe_calage`	vecteur	Bloc *optionnel* définissant un axe de calage autour duquel le senseur peut tourner (voir 11.11).
`genre`	mot-clef	Identificateur définissant l'ordre d'application des rotations et la rotation considérée, vaut obligatoirement `LRT-lacet`, `LRT-roulis`, ou `LRT-tangage` pour l'ordre lacet puis roulis puis tangage, `LTR-lacet`, `LTR-tangage`, ou `LTR-roulis` pour l'ordre lacet puis tangage puis roulis, `RLT-roulis`, `RLT-lacet`, ou `RLT-tangage` pour l'ordre roulis puis lacet puis tangage, `RTL-roulis`, `RTL-tangage`, ou `RTL-lacet` pour l'ordre roulis puis tangage puis lacet, `TLR-tangage`, `TLR-lacet`, ou `TLR-roulis` pour l'ordre tangage puis lacet puis roulis, `TRL-tangage`, `TRL-roulis`, ou `TRL-lacet` pour l'ordre tangage puis roulis puis lacet.
`reference`	mot-clef	Identificateur définissant le repère à partir duquel sont appliquées les rotations, vaut obligatoirement `geocentrique`, `orbital-TNW`, `orbital-QSW`, `inertiel`, `topocentrique` ou `utilisateur`.

TAB. 16 – senseur cinématique

nom du bloc	type	description
`type`	mot-clef	Identificateur permettant de reconnaître le type de senseur, vaut obligatoirement `cinematique`.
`precision`	réel	Précision du senseur en degrés, utilisée pour la convergence.
`repere`	rotation	Définition du repère senseur, le plus pratique est d'utiliser la syntaxe `{i{...} j{...} k{...}}`, on donne alors les coordonnées *en repère satellite* des axes canoniques du senseur.
`axe_calage`	vecteur	Bloc *optionnel* définissant un axe de calage autour duquel le senseur peut tourner (voir 11.11).
`axe_sensible`	vecteur	Coordonnées *en repère senseur* de l'axe sensible du senseur.

TAB. 17 – gyromètre intégrateur

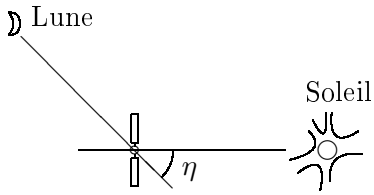
nom du bloc	type	description
`type`	mot-clef	Identificateur permettant de reconnaître le type de senseur, vaut obligatoirement `gyro_integrateur`.
`precision`	réel	Précision du senseur en degrés, utilisée pour la convergence.
`repere`	rotation	Définition du repère senseur, le plus pratique est d'utiliser la syntaxe `{i{...} j{...} k{...}}`, on donne alors les coordonnées *en repère satellite* des axes canoniques du senseur.
`axe_calage`	vecteur	Bloc *optionnel* définissant un axe de calage autour duquel le senseur peut tourner (voir 11.11).
`axe_sensible`	vecteur	Coordonnées *en repère senseur* de l'axe sensible du senseur.

5.2.9 Définition des paramètres d'inhibition des senseurs optiques

Les senseurs optiques observant des cibles particulières (terre, point au sol, autre satellite ...) peuvent être perturbés par la présence d'astres très lumineux comme la lune, le soleil ou un astre de type corps central (terre par exemple) dans le voisinage de leur champ de vue. Ce phénomène est pris en compte dans Marmottes lorsque l'utilisateur définit un champ d'inhibition pour la lune, le soleil, un astre de type corps central (ou une combinaison des trois). Ces champs dénommés `champ_d_inhibition_corps_central` , `champ_d_inhibition_soleil` et `champ_d_inhibition_lune` sont optionnels, par défaut il n'y a pas d'inhibition. Il y a inhibition lorsque l'astre incriminé est dans le champ d'inhibition et n'est pas en éclipse (pour la lune et le soleil). Outre les champs eux-mêmes, l'utilisateur peut définir une marge de sécurité angulaire pour le calcul des éclipses (`marge_eclipse_soleil` et `marge_eclipse_lune`), qui valent 0 degré par défaut. L'astre inhibant est considéré en éclipse si la distance angulaire entre le centre de l'astre occultant et le centre de l'astre inhibant plus la marge est inférieur au rayon de l'astre occultant. Dans la pratique, cela signifie qu'une marge de 1 degré conduira à considérer qu'il y a encore inhibition si l'astre inhibant est derrière la terre mais à moins de 1 degré du limbe.

Enfin dans le cas de la lune, on peut tenir compte de son éclairement (les phases de la lune) et ne pas considérer d'inhibition lorque celui-ci est plus faible qu'un seuil spécifié par le bloc `seuil_phase_lune`. La valeur par défaut de ce seuil est 180 degrés, ce qui signifie que la lune est toujours gênante, même en nouvelle lune.

FIG. 4 – angle de phase lune



η représente, sur ce schéma, l'angle de phase lune.

0 degré : le satellite est entre le soleil et la lune et donc voit la face éclairée de la lune, nous sommes en configuration de pleine lune. 180 degrés : lune et soleil sont du même côté, le satellite voit la face sombre de la lune, nous sommes en configuration de nouvelle lune.

L'utilisateur peut spécifier un seuil angulaire indiquant à partir de quand la brillance de la lune gène le senseur. Lorsque l'angle η est compris entre 0 degré et ce seuil, la lune est considérée comme trop brillante, il y a inhibition. Lorsque l'angle η est compris entre ce seuil et 180 degrés, la lune est assez sombre, il n'y a pas d'inhibition.

Par conséquent si l'utilisateur met ce seuil à 0, il n'y a jamais inhibition quelle que soit la phase de la lune et si *a contratio* il le positionne à 180 degrés il y a toujours inhibition quelle que soit la phase de la lune.

Une valeur de η couramment utilisée est 90 degrés.

6 installation

La bibliothèque MARMOTTES est livrée sous forme d'une archive compressée dont le nom est de la forme `marmottes-`*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[11] l'archive et d'en extraire les fichiers, par une commande du type :

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

Cette commande crée un répertoire `marmottes-`*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.

Par défaut, le script de configuration recherche les bibliothèques CLUB et CANTOR dans l'environnement par défaut de l'utilisateur et dans le préfixe choisi pour MARMOTTES. Les options `--with-club[=PATH]` et `--with-cantor[=PATH]` permet d'aider le script à trouver les bibliothèques lorsqu'elles sont installées dans des répertoires non standards. Lorque l'on spécifie un chemin du style `/racine/specifique`, les fichiers d'en-tête sont recherchés sous `/racine/specifique/include` et sous `/racine/specifique/include/club` (ou `cantor` selon le cas) et les bibliothèques 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[12] 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 `libmarmottes.a`, le répertoire de fichiers d'inclusion `marmottes`, et les fichiers de traduction `en/marmottes` et `fr/marmottes` 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.

[11] l'utilitaire de décompression `gunzip` est disponible librement sur tous les sites `ftp` miroirs du site GNU
[12] l'option `--prefix` s'initialise à l'aide d'une variable shell `prefix`

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 `marmottes-`*`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/marmottes-vv.rr.tar.gz | tar xvf -
cd marmottes-vv.rr
./configure --prefix=/chemin/vers/l/espace/prive
make
make install
cd ..
rm -fr marmottes-vv.rr
```

7 messages d'avertissements et d'erreurs

La bibliothèque MARMOTTES 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[13]. 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.

`"\"%s\" n'est pas un gyromètre intégrateur, initialisation impossible"`
(`"\"%s\" is not a rate integrated gyro, unable to initialize"`) Ce message est généré si l'utilisateur tente de réinitialiser un senseur qui n'est pas un gyromètre intégrateur. Il s'agit soit d'une erreur sur le nom du senseur, soit d'une incohérence entre les fichiers et le code.

`"%s n'est pas un senseur cinématique, initialisation de la dérive impossible"`
(`"%s is not a kinematic sensor, unable to initialize the drift"`) Ce message est généré (par la méthode **MarmottesInitialiseDerive**) lorsque l'utilisateur essaie d'appliquer une dérive à un senseur qui n'est pas un senseur cinématique.

[13] 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)

`"\"%s\" n'est pas un senseur de Cardan, initialisation du repère de référence impossible"` (`"\"%s\" is not a Cardan sensor, unable to initialize reference frame"`) Ce message est généré lorsque l'utilisateur tente de modifier le repère de référence d'un senseur qui n'est pas un senseur de Cardan. Il s'agit soit d'une erreur sur le nom du senseur, soit d'une incohérence entre les fichiers et le code.

`"\"%s\" n'est pas un senseur optique, initialisation de la cible impossible"` (`"\"%s\" is not an optical sensor, unable to initialize target"`) Ce message est généré lorsque l'utilisateur tente de modifier la cible d'un senseur qui n'a pas été prévu pour cela. Il s'agit soit d'une erreur sur le nom du senseur, soit d'une incohérence entre les fichiers et le code.

`"attitude solution théorique incontrôlable (senseur %s)"` (`"solved theoretical attitude cannot be controlled (%s sensor)"`) Ce message est généré lorsque la résolution d'attitude et le filtrage ont sélectionné une attitude unique respectant les consignes, mais que l'un des senseurs ne peut la contrôler, les causes peuvent être diverses :

tout senseur optique : la cible peut être hors du champ de vue ;

senseur solaire : il peut y avoir éclipse (on peut le vérifier en changeant la cible de `soleil` à `pseudo-soleil`) ;

senseur terre : le senseur peut être inhibé par la lune ou le soleil (on peut le vérifier en réduisant les champs d'inhibition) ;

senseur station : le satellite peut être sous le masque d'antenne, en tenant compte de l'effet troposphérique.

Ce message indique un vrai problème de contrôle d'attitude, il faut changer les consignes ou changer de senseurs de contrôle pour l'éliminer.

`"axe de rotation nul dans le bloc \"%s\""` (`"null rotation axis in bloc \"%s\""`) Ce message est produit par une opération lors de la lecture du fichier de description des senseurs. C'est en fait une erreur de CANTOR qui survient pendant la construction d'une rotation. L'axe de la rotation est nul.

`"bloc \"%s\" non terminal"` (`"bloc \"%s\" not terminal"`) Ce message indique une erreur dans les fichiers de description des senseurs, il peut être généré au moment où l'on lit un senseur particulier pour la première fois (si l'ensemble des fichiers sont lus dès la création du simulateur Marmottes, les descriptions de senseurs sont analysées à la demande). MARMOTTES attendait un bloc terminal et a trouvé un bloc ayant une structuration interne.

`"bloc \"%s\" introuvable dans le fichier \"%s\""` (`"no bloc \"%s\" in file \"%s\""`) Ce message n'est pas généré par MARMOTTES mais par CLUB en cas d'échec d'une tentative d'extraction d'un bloc de haut niveau (c'est à dire d'un senseur ou d'un bloc servant à une indirection), il s'agit le plus souvent d'une erreur de nommage de ce bloc (soit dans l'appel soit dans le fichier).

`"cible d'un senseur inconnue : \"%s\"\ncibles connues :"` (`"unknown sensor target : \"%s\"\nknown targets :"`) Ce message indique une erreur dans les fichiers de description des senseurs, la cible d'un senseur optique n'a pas été reconnue. Le message est suivi de la liste des cibles autorisées.

`cible du senseur \"%s\" non initialisée` (`\"%s\" sensor, uninitialized target`) Ce message est généré lors de l'utilisation d'un senseur ayant une cible définie par l'utilisateur lorsque celui-ci utilise le senseur avant d'initialiser sa cible. Il s'agit vraisemblablement d'une erreur de codage.

`"consignes des senseurs \"%s\" et \"%s\" incompatibles"`
(`"inconsistent attitude constraints for sensors \"%s\" and \"%s\""`) Ce message est généré lors d'une tentative de résolution d'attitude si l'on se rend compte que deux consignes géométriques ne sont pas compatibles entre elles (par exemple parce qu'elles sont redondantes et ne permettent pas de construire des familles de modèles à un degré de liberte). Ce message indique généralement une erreur dans le choix des senseurs de consignes ou dans leurs caractéristiques (axes sensibles entre autres).

`"consignes des senseurs \"%s\" et \"%s\" incompatibles avec un omega maximal limité à %f degrés par seconde"`
(`"attitude constraints for sensors \"%s\" et \"%s\" inconsistent with maximal %omega limited to %f degrees per seconde"`) Ce message indique que la combinaison des consignes des deux premiers senseurs cinématiques dépasse la vitesse de rotation maximale du satellite. Il ne peut donc y avoir aucune solution, et ce quelle que soit la valeur de la troisième consigne. Il faut donc vérifier les consignes, ou bien le pas de temps alloué au satellite pour tourner dans le cas des gyromètres intégrateurs (un pas trop court revient à demander une vitesse trop élévée).

`"consigne du senseur \"%s\" dégénérée"`
(`"degenerated attitude constraint for sensor \"%s\""`) Ce message est généré lors d'une tentative de résolution d'attitude si l'on se rend compte que l'un des cônes de consigne géométrique est quasiment ponctuel. Ce message indique généralement une erreur dans le choix de la valeur de la consigne (par exemple $\pm 90^\circ$ pour un senseur de déclinaison).

`"échantillon vide (bloc \"%s\")"`
(`"empty sample (bloc \"%s\")"`) Ce message est généré s'il n'y a pas de point dans l'échantillonnage d'un senseur de fonction échantillonnée. Ce message indique une erreur dans le fichier de description.

`"erreur d'allocation mémoire"`
(`"memory allocation error"`) Ce message indique qu'une tentative d'allocation mémoire s'est soldée par un échec, il s'agit généralement d'un dépassement des capacités de la machine. Il faut alors vérifier l'utilisation des ressources par le programme appelant (par exemple vérifier que l'on demande bien à la bibliothèque de libérer les simulateurs par des appels à **MarmottesDetruire**), ou par les autres programmes pouvant tourner en parallèle.

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

`"erreur non reconnue"`
(`"unknown error"`) Ce message ne devrait normalement pas apparaître ... Il indique qu'une exception inattendue a été récupérée, ce qui temoigne d'une erreur de la bibliothèque. Dans une telle éventualité, il faut prévenir la maintenance.

`"genre non reconnu pour le senseur d'angle de Cardan \"%s\"\ngenres connus :"`
(`"unknown kind of Cardan angle sensor \"%s\"\nknown kinds :"`) Ce message est généré lors de la lecture d'un senseur de Cardan dont le genre n'est pas reconnu, il est suivi de la liste des genres reconnus. Ce message indique une erreur dans les fichiers.

`"gyromètres coaxiaux"`
(`"coaxials rate gyros"`) Ce message est généré lors de la prise en compte des consignes au début d'une

résolution d'attitude par un modèle cinématique lorsque les deux senseurs ont des axes trop proches les uns des autres. Ceci indique vraisemblablement une erreur de paramétrage avec une recopie du même nom de senseur ou la définition du même senseur sous deux noms différents dans le fichier.

`"identificateur marmottes non initialisé : %d"` (`"uninitialized marmottes id : %d"`) Ce message est généré par les interfaces fonctionnelles C et FORTRAN lorsque le numéro utilisé comme identificateur est invalide, ceci indique une erreur de codage, l'identificateur utilisé n'ayant pas été retourné par un appel préalable à **MarmottesCreer** ou ayant été détruit par un appel précédent à **MarmottesDetruire**.

`"la cible du senseur \"%s\" n'est pas modifiable"` (`"target modification is not allowed for sensor \"%s\""`) Ce message est généré lorsque l'utilisateur tente de modifier la cible d'un senseur qui n'a pas été prévu pour cela. Il s'agit soit d'une erreur sur le nom du senseur, soit d'une incohérence entre les fichiers et le code.

`"la norme des vecteurs position/vitesse est temporairement litigieuse et ne doit pas être utilisée"` (`"the size of position/speed vectors is temporarily questionable and should not be used"`) Ce message apparaît lorsque l'utilisateur change les unités d'entrée d'un simulateur (fonction **MarmottesUnitesPositionVitesse** ou méthode **unitesPositionVitesse** de la classe Marmottes) et qu'il fait ensuite un appel nécessitant une position exacte (par exemple mesure d'un senseur solaire, qui passe par un calcul de parallaxe) avant d'avoir redonné de nouvelles position et vitesse dans les nouvelles unités. Il faut noter que toutes les utilisations de senseurs n'engendrent pas cette erreur, ainsi un senseur de Cardan basé sur le repère orbital local n'utilise que les directions des vecteurs, pas leur norme.

`"le bloc \"%s\" a %d champs (%d demandés)"` (`"bloc \"%s\" has %d fields (%d asked for)"`) Ce message indique une erreur dans les fichiers de description des senseurs, il peut être généré au moment où l'on lit un senseur particulier pour la première fois (si l'ensemble des fichiers sont lus dès la création du simulateur Marmottes, les descriptions de senseurs sont analysées à la demande). MARMOTTES attendait un certain nombre de champs dans un bloc terminal et en a trouvé un autre.

`"le repère de référence du senseur \"%s\" n'est pas modifiable"` (`"reference frame modification is not allowed for sensor \"%s\""`) Ce message apparaît lorsque l'utilisateur tente de modifier le repère de référence d'un senseur de Cardan qui n'a pas été prévu pour cela. Il s'agit soit d'une erreur sur le nom du senseur, soit d'une incohérence entre les fichiers et le code.

`"le senseur \"%s\" ne peut être utilisé qu'en mesure, pas en consigne"` (`"\"%s\" sensor can be used only for measurements, not for attitude constraints"`) Ce message indique que l'utilisateur a tenté d'utiliser en consigne un senseur pour lequel seule la fonction de mesure a été implantée, pas la fonction inverse qui permet de créer un modèle d'attitude en partant d'une mesure. Il s'agit par exemple des pseudo-senseurs représentant des gains d'antennes (on ne peut donner le gain et en déduire l'attitude). Il s'agit soit d'une erreur sur le nom du senseur, soit d'une incohérence entre les fichiers et le code.

`"le senseur \"%s\" observe le corps central mais est inhibé par le corps central"` (`"sensor \"%s\" observes the central body but is also inhibited by the central body"`) Ce message est affiché lors de la lecture d'un capteur si sa cible est liée au corps central mais si simultanément le capteur a un champ d'inhibition pour le corps central. Cette situation correspond généralement à une erreur de modélisation ou à un copier-coller malencontreux et est explicitement interdite dans la bibliothèque. Il faut corriger le fichier senseur et éliminer le champ d'inhibition.

"le senseur \"%s\" observe le soleil mais est inhibé par le soleil" ("sensor \"%s\" observes the sun but is also inhibited by the sun") Ce message est affiché lors de la lecture d'un capteur si sa cible est liée au soleil mais si simultanément le capteur a un champ d'inhibition pour le soleil. Cette situation correspond généralement à une erreur de modélisation ou à un copier-coller malencontreux et est explicitement interdite dans la bibliothèque. Il faut corriger le fichier senseur et éliminer le champ d'inhibition.

""le senseur \"%s\" observe la lune mais est inhibé par la lune" ("sensor \"%s\" observes the moon but is also inhibited by the moon") Ce message est affiché lors de la lecture d'un capteur si sa cible est liée à la lune mais si simultanément le capteur a un champ d'inhibition pour la lune. Cette situation correspond généralement à une erreur de modélisation ou à un copier-coller malencontreux et est explicitement interdite dans la bibliothèque. Il faut corriger le fichier senseur et éliminer le champ d'inhibition.

"les senseurs \"%s\" et \"%s\" ne peuvent pas être utilisés en consigne simultanément" ("sensors \"%s\" and \"%s\" cannot be used simultaneously for control") Ce message est généré lors du choix des senseurs utilisés pour le modèle analytique lorsqu'il reste une incompatibilité après le tri. Cette erreur ne devrait normalement pas se produire avec les versions de MARMOTTES actuelles (jusqu'à la 6.1 au moins) car il n'y a que deux types de senseurs et le choix est à effectuer entre trois senseurs, il y en a donc toujours deux de même type. Il est cependant possible que des versions futures introduisent de nouveaux types, éventuellement même des types qui ne soient jamais utilisables en consigne.

"liste des senseurs de contrôle non initialisée" ("list of control sensors not initialized") Cette erreur signale que l'on tente de résoudre une attitude sans avoir initialisé les senseurs de contrôle, normalement ceci ne devrait jamais se produire, car la construction des simulateurs aurait déjà signalé une erreur au préalable. Ceci indique donc une erreur de codage.

"nombre de points d'échantillonnage incorrect (%d points, bloc \"%s\")" (incorrect number of sample points (%d points, bloc \"%s\")) Ce message est généré à la lecture d'une table d'échantillonnage lors de la description d'un senseur de fonction échantillonnée, le nombre doit obligatoirement être pair (et non nul). Ce message indique une erreur dans le fichier de description.

"nombre de points de masque incorrect (%d points, bloc \"%s\")" ("incorrect number of mask points (%d points, bloc \"%s\")") Ce message est généré à la lecture d'un masque d'antenne sol lors de la description d'une station, le nombre doit obligatoirement être pair (et non nul). Ce message indique une erreur dans le fichier de description.

"nombre de tranches de dichotomie négatif ou nul : %d" ("negative or null dichotomy samples number : %d") Ce message est généré lorsque l'utilisateur change le nombre de tranches de dichotomie de l'algorithme de résolution numérique (par la fonction **MarmottesDichotomie** ou par la méthode **dichotomie** de la classe Marmottes) et qu'il donne une valeur irréaliste. Il s'agit vraisemblablement d'une erreur de codage.

"norme du quaternion \"%s\" trop petite" ("too small norm for quaternion \"%s\"") Ce message est généré lors de la lecture d'une rotation sous forme de quaternion dans les fichiers de description des senseurs si la norme est excessivement faible, ceci indique une erreur dans les fichiers.

"norme du vecteur \"%s\" trop petite" ("too small norm for vector \"%s\"") Ce message est généré lors de la lecture d'un vecteur sous forme de

coordonnées cartésiennes dans les fichiers de description des senseurs si la norme est excessivement faible, ceci indique une erreur dans les fichiers.

`"pas d'axe de calage de défini pour ce senseur"`
(`"no defined wedging axis for this sensor"`) Ce message est généré lors d'une tentative de recalage d'un senseur (par la fonction **MarmottesCalage** ou par la méthode **calage** de la classe Marmottes) lorsqu'aucun axe de calage n'a été défini pour ce senseur (l'axe de calage est une donnée optionnelle pour tout senseur). Ceci indique soit une erreur dans les fichiers de description des senseurs, soit une erreur de paramétrage.

`"pas de solution aux consignes d'attitude"`
(`"no solution to attitude constraints found"`) Ce message est de très loin celui qui apparaît le plus fréquemment ! il indique que MARMOTTES n'a pas réussi à trouver d'attitude respectant à la fois les consignes et les contraintes des senseurs (champs de vue, inhibitions, masques, ...) et qu'il a été impossible de déterminer lors du rejet des solutions parasites lesquelles étaient de simples artefacts mathématiques et laquelle correspondait à la solution physique attendue mais rejetée par les contraintes technologiques. Dans la plupart des cas, cela indique que l'attitude n'est effectivement pas contrôlable de la façon souhaitée, on peut investiguer en suivant les conseils donnés dans la description du message `attitude solution théorique incontrôlable ...` , avec la différence que l'on ne sait pas quel senseur a conduit au rejet (car on peut rejetter généralement les artefacts mathématiques sur un senseur et la solution physique sur un autre, les artefacts étant souvent des symétriques de la solution).

`"point d'échantillonnage (%f, %f) hors bornes (bloc \"%s\")"`
(`"sample point (%f, %f) out of bounds (bloc \"%s\")"`) Ce message apparaît lors de la lecture d'un senseur de fonction échantillonnée si l'angle d'échantillonnage est inférieur à 0 ou supérieur à π. Ce message indique une erreur dans le fichier de description.

`"problème dans la mise à jour du repère de référence du senseur \"%s\""`
(`"problem in sensor \"%s\" reference frame update"`) Ce message est généré lors de la mise à jour du repère de référence d'un senseur de type Cardan. C'est en fait une erreur de CANTOR qui survient lors de la construction du repère.

`"référence à un objet inconnu dans le tracé"`
(`"reference to an unknown object in the call trace"`) Ce message indique que lors de la retranscription des traces d'appels, une référence à un objet inconnu a été faite (par exemple lors d'un appel à une méthode). Ceci indique soit que l'activation du mécanisme de tracé a été faite trop tard et que la création de l'objet référence a été manquée (cela peut par exemple arriver si le mécanisme est lancé manuellement depuis un *debugger* pendant une exécution), soit que l'objet a été détruit entre-temps. Il faut prendre garde à activer le mécanisme avant les constructions d'objets.

`"repère de référence du senseur \"%s\" non initialisé"`
(`"\"%s\" sensor, uninitialized reference frame"`) Ce message est généré lors de l'utilisation d'un senseur de Cardan ayant un repère de réference défini par l'utilisateur lorsque celui-ci utilise le senseur avant d'initialiser son repère. Il s'agit vraisemblablement d'une erreur de codage.

`"repère non reconnu pour le senseur d'angle de Cardan \"%s\"\nrepères connus :"`
(`"unknown frame for Cardan angle sensor \"%s\"\nknown frames :"`) Ce message est généré lors de la lecture d'un senseur de Cardan dont le repère de référence n'est pas reconnu, il est suivi de la liste des repères reconnus. Ce message indique une erreur dans les fichiers.

`"seuil de convergence négatif ou nul : %f"`
(`"negative or null convergence threshold : %f"`) Ce message est généré lorsque l'utilisateur change le seuil de convergence de l'algorithme de résolution numérique (par la fonction **MarmottesConvergence** ou par la méthode **convergence** de la classe Marmottes) et qu'il donne une valeur irréaliste. Il s'agit vraisemblablement d'une erreur de codage.

`"sous-bloc \"%s\" introuvable dans le bloc \"%s\" du fichier \"%s\""`
(`"no bloc \"%s\" inside bloc \"%s\" (file \"%s\")"`) Ce message n'est pas généré par MARMOTTES mais par CLUB lors de la recherche d'un sous-bloc dans un bloc non terminal des fichiers de description des senseurs, il s'agit soit d'une erreur de syntaxe dans les fichiers, soit d'un problème de langue, le fichier ayant des mots-clefs dans une langue qui ne correspond pas à la langue couramment utilisée par MARMOTTES. Ce dernier cas est directement visible par la langue utilisée pour afficher le message et par le nom du bloc recherché (le premier bloc recherché dans tous les senseurs est le bloc `repere` en français, traduit par `frame` dans la distribution standard de MARMOTTES).

`"type d'un senseur inconnu : \"%s\"\ntypes connus :"`
(`"unknown sensor type : \"%s\"\nknown types :"`) Ce message est généré lors de la lecture du type d'un senseur dans les fichiers de description des senseurs, il est suivi de la liste des senseurs connus. Ce message indique une erreur dans les fichiers.

`"types des senseurs \"%s\" et \"%s\" incompatibles"`
(`"incompatible types for sensors \"%s\" and \"%s\""`) Ce message est généré lors d'une résolution d'attitude partielle (par le fonction **MarmottesDeuxConsignes** ou par la méthode **deuxConsignes** de la classe Marmottes) si les deux premiers senseurs ne sont pas tous les deux géométriques ou tous les deux cinématiques. En effet dans le cas d'une résolution partielle, on ne peut plus réordonner les senseurs, et il faut que les senseurs permettent de construire le modèle analytique (il n'y a dans ce cas pas de résolution numérique). Il s'agit là d'une limitation de MARMOTTES.

`"unité de position inconnue : \"%s\"\nunités connues :"`
(`"unknown position unit : \"%s\"\nknown units :"`) Ce message apparaît lors du changement des unités d'entrée d'un simulateur (fonction **MarmottesUnitesPositionVitesse** ou méthode **unitesPositionVitesse** de la classe Marmottes) si l'unité donnée par l'utilisateur n'est pas valide, il est suivi de la liste des unités reconnues. Si ce message apparaît pour une unité utile, il faut demander une évolution de la bibliothèque (qui se limite à quelques lignes de code très simples).

`"unité de vitesse inconnue : \"%s\"\nunités connues :"`
(`"unknown speed unit : \"%s\"\nknown units :"`) Ce message apparaît lors du changement des unités d'entrée d'un simulateur (fonction **MarmottesUnitesPositionVitesse** ou méthode **unitesPositionVitesse** de la classe Marmottes) si l'unité donnée par l'utilisateur n'est pas valide, il est suivi de la liste des unités reconnues. Si ce message apparaît pour une unité utile, il faut demander une évolution de la bibliothèque (qui se limite à quelques lignes de code très simples).

`"vitesse de rotation max négative ou nulle : %f deg/s"`
(`"negative or null maximum angular rate : %f deg/s"`) Ce message est généré lorsque l'utilisateur change la limite de vitesse de rotation de l'algorithme de résolution numérique (par la fonction **MarmottesWMax** ou par la méthode **wMax** de la classe Marmottes) et qu'il donne une valeur irréaliste. Il s'agit vraisemblablement d'une erreur de codage.

8 tests

La bibliothèque MARMOTTES 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 le résultat avec le résultat de référence archivé à l'aide de l'utilitaire `difference` disponible avec la bibliothèque CLUB. Les statistiques des différences sont affichées, elles devraient normalement se limiter à des erreurs de l'ordre de grandeur des seuils de convergence (typiquement 10^{-4}) ou de l'ordre de grandeur de la précision numérique de la machine (typiquement 10^{-15}) selon les cas.

Les classes disposant de tests spécifiques sont :
- Champ
- Etat
- Marmottes
- Parcelle

Certains senseurs spécifiques sont également testés :
- modification de la cible des senseurs optiques ;
- modification du repère de référence des senseurs de Cardan ;
- initialisation des gyromètres intégrateurs ;
- mesure des senseurs de gain d'antenne échantillonné.

Enfin, depuis la mise en place du système de trace d'exécution par les fonctions du type **MarmottesActiveTrace** décrites à la section 11, de nombreux cas représentatifs de l'utilisation réelle de la bibliothèque ont été ajoutés à la suite de tests automatiques. Ces cas regroupent aussi bien des cas de référence qui fonctionnaient bien que les cas qui permettaient de reproduire des erreurs de la bibliothèque. D'une façon générale, lorsqu'une erreur est détectée par un utilisateur, un nouveau cas test est ajouté à la fois pour valider la correction et pour éviter des régressions futures.

Dans la pratique, toutes les classes algorithmiques sont testées par le test de la classe de haut niveau Marmottes, mais tous les senseurs ne sont pas utilisés pour ces tests.

Cette panoplie de tests est à l'évidence trop succincte et devrait être améliorée.

9 maintenance

La maintenance de la bibliothèque MARMOTTES 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 [DR7]). 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` [DR8] (version 1.5.14 au moins)
- `autoconf` [DR9] (version 2.59 au moins)
- `automake` [DR10] (version 1.9.5 au moins)
- `cvs` [DR11] et [DR12]
- `g++` [DR13] (version 3.3 au moins)
- `gzip`
- GNU `m4`
- GNU `make`
- `perl`
- TEX/LATEX/`dvips` (de plus `xdvi` est recommandé)
- la classe LATEX «`notecnes`»
- les paquetages LATEX «`babel`» et «`longtable`»

9.3 installation de l'environnement de maintenance

Le développeur récupère tout d'abord le module marmottes par une commande `cvs checkout marmottes`, 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/MarmottesConfig.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 MARMOTTES 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 MARMOTTES 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. [DR11] et [DR12]). 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/MarmottesConfig.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/MarmottesConfig.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). Si l'on utilise l'éditeur `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 [DR7]. 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` [DR9] (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`.

La gestion des dépendances dans le `Makefile.in` de développement (créé par `automake`) impose l'utilisation de `g++` (ou `egcs`) et du `make` de GNU. Pour tester d'autres outils, le plus simple est de créer une distribution (par `make dist`), puis de tenter de l'installer comme un utilisateur standard en retirant les outils GNU de son

`PATH` (après avoir décompressé l'archive). Les distributions n'ont en effet aucun besoin de dépendances fines, les `Makefile.in` créés sont donc simplifiés et portables.

Les fichiers décrivant les spécificités de la bibliothèque MARMOTTES. 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/MarmottesConfig.h` généré par configure, et c'est ce `#define` qui est utilisé par la fonction marmottesVersion.

10 évolutions

10.1 changements depuis les versions précédentes

10.1.1 évolutions entre la version 9.7 et la version 9.8

Correction d'une erreur de recalage d'angle entre 0 et 2π pour les senseurs de Cardan (FA 00230).

10.1.2 évolutions entre la version 9.6 et la version 9.7

Correction d'une erreur d'interpolation pour les senseurs visant une station sol ayant un masque d'antenne non constant (FA 0034, FA 0032).

Correction d'une erreur de changement de repère : la transformation des vecteurs du repère senseur dans le repère satellite n'était pas effectuée (FA 0035).

Ajout d'explications sur le repère d'expression de la cible au niveau de la documentation utilisateur (DM 0033).

10.1.3 évolutions entre la version 9.5 et la version 9.6

Corrections pour améliorer la qualité : utilisation d' une méthode statique et une méthode privée pour résoudre l'attitude. Élimination des méthodes publiques (sB, modele, date, position, famille, seuil et tranches) dans la classe ResolveurAttitude (DM 0029).

Un nouveau service de lecture des paramètres internes à MARMOTTES est disponible via les 3 interfaces (c++, c et fortran) (DM 0027).

Ajout des mots-clefs permettant l'extension de la notion de cible sans éclipse à polaris et canopus (DM 0026).

Ajout de la possibilité de définir une zone d'inhibition pour un astre de type corps central (FA 0030).

Correction des cibles position-sans-eclipse et direction-sans-eclipse qui étaient impossibles à initialiser en raison d'une protection trop sévère générant un message d'erreur (FA 0031).

Ajout de la possibilité de journaliser les appels au constructeur par copie de la classe Marmottes, afin d'autoriser la génération de journaux d'appels pour les applicatifs utilisant MARMOTTES à partir du langage C++ (DM 0028).

Extension du domaine de validité de la cible vitesse-sol-apparente par continuité au delà du limbe du corps central, en considérant que les points visés ont une vitesse nulle. Ceci élimine complètement un cas d'erreur et permet d'utiliser cette cible dès l'initialisation du simulateur (DM 0025).

10.1.4 évolutions entre la version 9.4 et la version 9.5

Des erreurs d'initialisations introduites lors des interventions qualité du projet ATV ont été corrigées.

Les fonctions de calcul par défaut du temps sidéral, de la position du Soleil, de la Terre et de la Lune sont externalisées pour permettre leur utilisation dans le cadre d'un appel à MarmottesEnregistreCorps depuis l'interface fortran.

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.

10.1.5 évolutions entre la version 9.3 et la version 9.4

Une interface Fortran permettant de récupérer le numéro de version de la bibliothèque a été intégré (DM 0009).

Les fonctions MarmottesAutoriseExtrapolation et MarmottesCopie sont dorénavant testées (DM 0010).

Une revue de la bibliothèque a été effectuée pour harmoniser les classes entre elles afin qu'elles soient sur un modèle unique (DM 0012).

L'utilitaire mamottesReplay est maintenant capable de reconnaitre l'utilisation de fonctions utilisateurs ou de fonctions par défaut pour les corps célestes et peut donc les prendre en compte dans son algorithme (DM 0013).

La documentation a été agrémenté d'un schéma explicatif sur la définition de l'angle soleil-satellite-lune (DM 0014), d'une description des repères de référence des senseurs (DM 0015) et d'un lexique français-anglais des mots clés reconnus par marmottes (DM 0016).

L'extension des fichiers a été changé de CC en CPP, cette dernière extension étant reconnue par un plus grand nombre de systèmes (DM 0017).

La bibliothèque est maintenant compilable, jusqu'à la version GCC 3.2.1, de légéres modifications ayant été apporté aux fichiers (DM 0018).

10.1.6 évolutions entre la version 9.2 et la version 9.3

Par défaut, MARMOTTES dispose de modèles internes pour le calcul du temps sidéral, des éphémérides du Soleil, de la Lune et de la Terre, ainsi que des caractéristiques du corps central.
Si ces modèles ne conviennent pas à l'utilisateur (par exemple : dans un cadre interplanétaire), alors ces modèles peuvent être personnalisés en fournissant des valeurs numériques et des fonctions de calculs appropriées (DM 0007).

La documentation (au format Postscript) présente désormais des barres de modification aux endroits modifiés par rapport à la version précédente.

Quelques erreurs rares de configuration dans des cas inhabituels ont été corrigées. Il s'agit de corrections mineures n'affectant pas les utilisateurs habituels.

Un test reproduisant les conditions de la FA 15 a été ajouté et le test du programme Parcelle a été mis à jour. La correction de cette anomalie est faite dans la bibliothéque cantor.

10.1.7 évolutions entre la version 9.1 et la version 9.2

La prise en compte de la dérive d'un senseur cinématique est maintenant possible.

Deux nouvelles méthodes, appelables à partir de l'interface C++, permettent, pour l'une, d'accéder au pointeur d'un senseur, à partir de son nom et du fichier qui le décrit et pour l'autre d'accéder à l'état de l'instance Marmottes.

Des directives throw oubliées dans des signatures de fonctions internes ont été corrigées.

10.1.8 évolutions entre la version 9.0 et la version 9.1

Une erreur sur les modèles géométriques à un vecteur fixe a été corrigée. Cette erreur grave empêchait de trouver les solutions ayant un vecteur fixe opposé à l'axe du cône de consigne.

Une erreur de consignes dégénérées non détectée a été corrigée. L'utilisation d'une consigne à -90 degrés pour un capteur plan-vecteur ou d'une consigne à 180 degrés pour un capteur vecteur n'étaient pas détectées alors qu'elles sont dégénérées.

L'utilisation de consignes cinématiques (gyromètres et gyromètres intégrateurs) conduisant à un modèle de vitesse supérieur au $\omega_{\max}$ génère désormais une erreur. Ce cas peut se rencontrer notamment lorsque l'on spécifie un pas de temps trop court pour qu'un gyromètre intégrateur atteigne une valeur angulaire donnée.

Les capteurs optiques ayant une cible liée au soleil ne peuvent pas être inhibés par le soleil. Il en est de même pour la lune. Les champs d'inhibitions étaient simplement ignorés jusque là. Désormais, une erreur est générée lorsqu'un champ d'inhibition porte explicitement sur la cible du capteur (FA 0008).

Prise en compte de versions récentes des outils de développement GNU. Les outils de développement de la suite GNU ont été mis à jour (autoconf 2.52, automake 1.5 et libtool 1.4.1). Ceci ne devrait avoir aucun impact sur les utilisateurs (qui se contentent de compiler la bibliothèque à partir des fichiers générés par ces outils et inclus dans la distribution). Seuls les développeurs de la bibliothèque qui sont amenés à y apporter des modification sont concernés.

10.1.9 évolutions entre la version 8.5 et la version 9.0

La syntaxe de l'opération de balayage dans les fichiers senseurs a été modifiée. Cette opération permet de créer un champ en *étalant* un autre champ selon une rotation. Les versions précédentes acceptaient toutes les définitions de rotation pour cette opération, mais ceci conduisait à des résultats faux lorsque l'angle de la rotation dépassait 180 degrés, car les rotations $(\vec{u}, \alpha)$ et $(-\vec{u}, 2\pi - \alpha)$ sont indiscernables lorsqu'on se contente de les voir comme opérateurs vectoriels. Désormais, il est *indispensable* de donner un axe et un angle pour cette opération. Les anciens fichiers senseurs qui utilisaient des rotations quelconques doivent être corrigés pour pouvoir être lus par cette version de la bibliothèque (cette incompatibilité est la raison du saut de numérotation de 8.x à 9.y).

Une cible vitesse-sol-apparente a été ajoutée. Cette cible représente la vitesse vue du satellite du point au sol situé dans la direction d'observation du capteur. Elle est typiquement utilisée pour contrôler le lacet d'un satellite d'observation de façon à compenser la vitesse de rotation du corps central et obliger les points observés à se déplacer dans une direction privilégiée au cours d'une prise de vue (perpendiculairement à la barrette CCD).

Une cible `lune-sans-eclipse` a été ajoutée pour les senseurs optiques.

Un capteur de gain d'antenne bidimensionnel en produit de sinus cardinaux a été ajouté. La forme en $\sin(x)/x$ utilisée dans les modélisations simples de gains d'antenne provient de la transformée de Fourier d'une ouverture rectangulaire. Il n'y a donc généralement pas de symétrie axiale pour ces formes, et il faut spécifier séparément la taille à 3 dB selon $\vec{X}$ de la taille selon $\vec{Y}$. L'ancien capteur en sinus cardinal carré, qui spécifiait un seul angle et présentait une symétrie est conservé pour des raisons de compatibilité, mais le nouveau capteur qui a été créé permet des modélisations plus réalistes.

Une erreur dans la formule du temps sidéral a été corrigée. Cette erreur dans les constantes du temps sidéral a un impact numérique minime sur la position des cibles de type station

Un problème potentiel dans la lecture des capteurs de limbe a été corrigé. La description des capteurs de limbe dans les fichiers ne nécessite pas de spécifier que la cible du capteur est le corps central. Le calcul du limbe impose en effet cette cible, de part la conception de l'algorithme. Avant la correction, l'utilisateur pouvait spécifier par erreur une cible qui n'était pas le corps central et qui était lue et utilisée dans certaines parties du code, indépendantes du calcul du limbe. Cela pouvait conduire à des résultats incohérents. La lecture des capteurs de limbe a été protégée, elle ne lit que les bloc explicitement utiles pour la modélisation du capteur.

L'utilitaire `marmottesReplay` est désormais installé en même temps que la bibliothèque.

Certaines méthodes des classes internes étant des prédicats retournent désormais des booléens au lieu d'entiers. Pour les utilisateurs des classes publiques de haut niveau, cette modification n'a d'impact que sur la méthode `Parcelle::` **pointSuivant** .

Les classes AnnotatedArc, Braid, Field, Node et Secteurs ont été transférées vers la bibliothèque CANTOR. La classe CallTrace a été transférée vers la bibliothèque CLUB.

La bibliothèque cantor génère maintenant des exceptions, qui sont remontées au niveau de certaines classes intermédiaires. Cette modification n'a pas d'impact pour les utilisateurs des classes publiques de haut niveau.

10.1.10 évolutions entre la version 8.4 et la version 8.5

des cibles `position-sans-eclipse` et `direction-sans-eclipse` ont été ajoutées pour les senseurs optiques. Le terme `pseudo-soleil` pour les cibles des senseurs optiques a été reformulé en `soleil-sans-eclipse` , plus explicite, de même le terme `terre-soleil` a été remplacé par `corps-central-soleil` , en prévision d'une évolution future vers l'interplanétaire. Les anciens termes sont toujours reconnus pour la compatibilité, mais ne sont plus documentés.

Les inhibition des capteurs par la Lune ou le Soleil étaient jusqu'à présent pris en compte uniquement pour les capteurs de limbe Terre. Désormais, tous les capteurs optiques peuvent avoir des champs d'inhibitions et leur contrôlabilité dépend à la fois de la présence de leur cible dans leur champ de vue mais également de l'absence d'inhibition. Une conséquence de cette modification est que la classe SenseurOptique ne déclare plus le type énuméré typeOpt ni la fonction virtuelle typeOptique qui sont devenue inutiles (toutes les classes descendant de SenseurOptique appartenant désormais à l'ancienne catégorie avecInhibitions).

La classe SenseurTerre a été remplacée par la classe SenseurLimbe. Ce remplacement a été fait à l'occasion de la simplification résultant de la factorisation des calculs d'inhibitions. La nouvelle classe est une version

considérablement simplifiée de l'ancienne, elle a été renommée à la fois pour des raisons de compréhension (la différence entre un senseur d'angle dièdre visant la terre et un senseur terre n'était pas immédiate) et également en prévision d'évolutions futures permettant d'utiliser MARMOTTES dans le cadre de projets interplanétaires.

De nouvelles fonctions ont été ajoutées à la bibliothèque. Elles permettent de bénéficier de toutes les fonctions de post-traitement telles que les calculs de mesures ou les vérifications de controlabilité, y compris lorsque l'attitude est calculée par des moyens externes (par exemple par intégration de la dynamique). Les fonctions ajoutées permettent d'imposer à MARMOTTES l'attitude (auquel cas le spin est déduit par différences finies depuis l'état précédent) ou le spin (auquel cas l'attitude est déduite par intégration depuis l'état précédent).

Des fonctions permettant d'obtenir des détails sur les critères de controlabilité qui sont respectés et ceux qui sont violés ont été ajoutées. Ces fonctions peuvent s'utiliser conjointement ou à la place des fonctions synthétiques qui se contentaient d'un résultat binaire (orientation contrôlable ou non par un capteur donné).

Une erreur dans l'algorithme de test de présence d'un point dans un champ de vue a été corrigée. Elle se manifestait de façon ponctuelle lorsque l'arc le plus court reliant le point à tester et un point intérieur de référence passait trop près d'un sommet de la frontière du champ. Ce cas était pris en compte mais le contournement n'était pas efficace dans tous les cas.

Une double erreur dans les capteurs Terre a été corrigée. Lors des calculs de contrôlabilité par des capteurs Terre, l'équation comportait une arctangente au lieu d'un arcsinus, ce qui conduisait à des résultats d'autant plus faux que le satellite était bas, et la position était considérée comme exprimée en kilomètres, même si l'utilisateur avait opté pour des mètres.

Des erreurs détectées par `purify` ont été corrigées. Ces erreurs étaient des fuites de mémoire, et des boucles non protégées qui conduisaient à lire des zones mémoires qui avaient été libérées en cours de boucle

Le support des compilateurs SUN a été amélioré. La bibliothèque semble compilable avec les compilateurs SUN. *Attention*, il semblerait cependant qu'il subsiste des erreurs graves d'implémentation de la STL par ces compilateurs (au moins jusqu'à la version 6.1). Si la bibliothèque est compilée avec ces versions, *elle ne fonctionne absolument pas* et génère des violations de mémoires dès que l'on utilise des champs de vue. Ces problèmes ont été longuement analysés (entre autre avec `purify`) et la bibliothèque elle-même ne semble pas en cause. L'utilisation de ces compilateurs est donc déconseillée. Les compilateurs GNU récents compilent cette bibliothèque sans aucun problème à la fois sur les plate-formes GNU-LINUX et solaris.

10.1.11 évolutions entre la version 8.3 et la version 8.4

Une erreur commune au constructeur par copie et à l'opérateur d'affectation de la classe Marmottes a été corrigée. Elle conduisait à une duplication de pointeurs dans les tables de senseurs des deux instances, ce qui pouvait engendrer des violations mémoire si l'une des instances avait été allouée dynamiquement puis libérée alors que l'autre était toujours utilisée.

Toutes les utilisations des classes non standards hash_map et hash_set ont été remplacées par des classes standards de la STL. Ces classes étaient des extensions de SGI qui n'étaient pas disponibles sur toutes les implémentations de la STL.

10.1.12 évolutions entre la version 8.2 et la version 8.3

Une erreur dans la modélisation des senseurs de CARDAN a été corrigée. Cette erreur ne se manifestait que pour des senseurs qui n'étaient pas alignés avec le repère satellite et conduisait à des résultats complètement erronés.

La valeur par défaut de la vitesse maximale de rotation utilisée par le modèle cinématique a été abaissée de 2 radians par seconde à 0,4 radians par seconde. La valeur précédente était vraiment trop élevée par rapport au satellites courants (les satellites stabilisés par effet gyroscopique sont de plus en plus rares et tournent à moins de 20 degrés par seconde). Des problèmes de résolution ont d'autre part été rencontrés avec l'ancienne valeur, pouvant aller jusqu'à manquer la solution à vitesse très faible attendue pour converger vers une solution faisant faire un tour complet au satellite entre chaque pas !

10.1.13 évolutions entre la version 8.1 et la version 8.2

Une initialisation oubliée a été corrigée dans la lecture des capteurs. Cet oubli induisait des violations mémoire en cas de lancement d'exception, si le fichier de description comportait certains types d'erreurs.

10.1.14 évolutions entre la version 8.0 et la version 8.1

Les seules modifications introduites dans la version 8.1 de la bibliothèque sont l'ajout du script de configuration `marmottes-config` destiné à faciliter la compilation d'applicatifs dépendant de MARMOTTES (cf 12.3, page 92). La version anglaise du fichier de licence a également été ajoutée dans la distribution.

10.1.15 évolutions entre la version 7.5 et la version 8.0

Une erreur d'estimation de spin lors des changements de loi a été corrigée. Cette erreur était liée à l'utilisation abusive d'états résolus antérieurs au changement de loi d'attitude, bien que des états extrapolés valides à la date du changement d'attitude soient disponibles.

Une mauvaise réinitialisation des senseurs de mesure lors de la réutilisation d'un simulateur désinitialisé au préalable a été corrigée. Si le senseur avait subi des modifications telles que changement du repère de base ou de la précision par exemple, le nouveau simulateur récupérait le senseur dans cet état modifié au lieu de l'état initial obtenu à la lecture du fichier.

Un mécanisme permettant de retranscrire dans un fichier tous les appels aux fonctions de l'interface publique de la bibliothèque a été ajouté (cf 11.21, page 87). Ce mécanisme devrait faciliter la reproduction par les développeurs des problèmes rencontrés par les utilisateurs finaux.

À l'aide du mécanisme précédent, la base des tests de non-régression a été fortement enrichie avec des cas provenant d'applicatifs utilisant la bibliothèque.

La bibliothèque MARMOTTES peut désormais être générée sous forme de bibliothèque partagée. Il faut cependant prendre garde, un problème rencontré sous solaris lors des tests a montré que les exceptions générées dans CLUB ne sont pas récupérées par CLUB lorsque toutes les bibliothèques sont partagées. Le problème ne

se pose ni avec des bibliothèques statiques ni sous GNU/linux. 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.

Un changement profond mais qui ne devrait que peu affecter la majorité des utilisateurs est l'abandon de la classe ChaineSimple de la bibliothèque CLUB au profit de la classe string du standard C++. Seules les interfaces C++ sont affectées. La classe Adressage a également été abandonnée au profit de la classe hash_map, mais ceci n'a aucun impact visible par les utilisateurs.

Quelques attributs internes utilisés comme indicateurs dans diverses classes ont été convertis du type entier vers le type booléen, qui reflète mieux leur usage réel.

Les pseudo-senseurs de gains d'antennes ajoutés à la version précédente ont été documentés. Un nouveau type de pseudo-senseur de gain d'antenne a été ajouté, il permet de modéliser des gains échantillonnés uniquement en fonction de l'angle de dépointage.

Cette version de la bibliothèque ne peut être compilée qu'avec les versions de CLUB 8.0 et ultérieures.

10.1.16 évolutions entre la version 7.4 et la version 7.5

La version 7.5 de la bibliothèque a vu la création de nouveaux types de senseurs : SenseurFonctionGauss et SenseurFonctionSinCard2 pour gérer les bilans de liaison d'antenne.

Des erreurs introduites lors du passage en version 7.3 ont également été corrigées.

Enfin, le support du compilateur Sun WorkShop 5.0 a été amélioré, il n'est cependant pas encore complet.

10.1.17 évolutions entre la version 7.3 et la version 7.4

La seule évolution de la version 7.4 est la correction d'un message d'erreur, qui pouvait conduire à des violations mémoire lors de son affichage. Quelques corrections mineures de la documentation ont également été apportées.

Cette version est la première à être destinée à une diffusion publique.

10.1.18 évolutions entre la version 7.2 et la version 7.3

La version 7.3 de MARMOTTES utilise désormais 5 tranches par défaut au lieu de 50 pour la phase de recherche des solutions, il n'est malheureusement pas sûr que cela suffise à accélérer les calculs ...

Cette version corrige un problème du modèle cinématique qui empêchait de trouver des solutions à spin faible mais non nul. Le problème apparaissait par exemple si un axe restait pointé sur le soleil et si les rotations autour de cet axe était interdit, les vitesses de rotations résultantes étant de un tour par an.

Une erreur dans l'extrapolation d'attitude a été corrigée. La contrôlabilité n'était pas testée, on pouvait donc calculer des attitudes correctes, puis les extrapoler au-delà de leur limite de validité. Un cas typique concerne le pointage terre où les consignes sont constamment nulles mais où le soleil ou la lune peuvent inhiber les senseurs.

Un certain nombre de messages d'erreur d'affichage du nom des senseurs lors de la lecture du fichier les décrivant ont également été corrigés.

Enfin, la documentation des routines de gestion de l'autorisation d'extrapoler a été ajoutée (les fonctions existaient depuis longtemps mais n'avaient pas été documentées) et la documentation des senseurs de Cardan a été améliorée.

10.1.19 évolutions entre la version 7.1 et la version 7.2

La version 7.2 de MARMOTTES n'apporte qu'une correction d'erreur. Les genres LRT-lacet, LRT-roulis et LRT-tangage des senseurs de Cardan étaient bien modélisés dans le code de la bibliothèque mais pas reconnus au niveau de la lecture du fichier des senseurs.

10.1.20 évolutions entre la version 7.0 et la version 7.1

La seule évolution apportée par la version 7.1 de MARMOTTES concerne la génération de la documentation qui est désormais livrée au format PDF au lieu du format PostScript.

Outre cette évolution, deux erreurs ont été corrigées dans la bibliothèque. La première erreur concernait des fuites de mémoire importantes introduites lors du passage en version 7.0. À cette occasion, d'autres fuites ont été détectées dans la bibliothèque CLUB et corrigées.

La seconde erreur était beaucoup plus ancienne mais n'avait jamais été rencontrée jusque là, elle correspondait à des échecs de résolution lors de l'utilisation de consignes à 180^o avec des senseurs dièdres. Un test de non régression correspondant au cas détecté par les utilisateurs a été ajouté.

10.1.21 évolutions entre la version 6.3 et la version 7.0

De très nombreuses évolutions ont été réalisées dans MARMOTTES depuis la version 6.3.

Du point de vue de l'utilisateur final, les changements principaux concernent l'introduction des gyromètres intégrateurs, l'introduction de la cible `nadir`, la possibilité de définir une cible programmable pour les senseurs optiques (par exemple pour pointer un satellite GPS, un site d'observation sol ou une étoile à l'infini) et la possibilité de paramétrer le repère de base des senseurs de Cardan.

Une refonte majeure des algorithmes de résolution des modèles géométriques a été menée à bien, ce qui devrait corriger des problèmes de résolution pour des configurations particulières. Les anciens algorithmes utilisaient une modélisation présentant des singularités les obligeant à prendre des points d'appuis de part et d'autre de certaines singularités et d'interpoler (et donc de commettre des erreurs d'approximation) entre ces points. Les nouveaux algorithmes utilisent une modélisation sans singularité.

Une autre amélioration algorithmique concerne la limitation des domaines de recherche dès le démarrage de la résolution (c'est à dire avant la phase itérative numérique) en propageant certaines contraintes (par exemple les champs de vue lorsqu'il y en a). Ceci devrait éviter de perdre du temps à chercher des solutions qui seront rejetés à terme dans la phase de filtrage des artefacts mathématiques. Il n'est cependant pas toujours possible de limiter ce

Des modifications qui ne concernent guère que les installateurs et les développeurs de la bibliothèque sont le passage aux outils de portabilité GNU (`autoconf` et `automake`), la réorganisation des répertoires de la distribution et le remplacement de la gestion de configuration sous `rcs` par une gestion sous `cvs`.

10.2 évolutions futures

Une évaluation des performances de MARMOTTES en terme de rapidité devrait être menée, et des actions correctives éventuellement réalisées.

MARMOTTES utilise un critère de convergence par senseur lors de la résolution numérique. Ce critère n'est pas très pratique et devrait être remplacé par un critère général sur l'attitude et sur le spin (il faudrait du même coup éliminer l'attribut de précision des senseurs et les méthodes associées).

Une lacune importante de MARMOTTES tient à son incapacité à intégrer la dynamique. Il serait bon de la combler, probablement par une autre bibliothèque utilisée en surcouche (la dynamique risque de se compliquer très vite, en particulier si l'on désire modéliser des modes souples).

Une autre extension intéressante serait l'introduction d'une couche de filtrage permettant de combiner les mesures de nombreux capteurs sur une plage de temps paramétrable, avec minimisation d'un critère de moindres carrés par exemple.

La notion de mode de pilotage serait un apport intéressant, soit dans MARMOTTES soit dans une bibliothèque associée, pour offrir une interface simplifiée dans certains cas comme le pointage terre, le *yaw steering*, ...

Pour accélérer la lecture des fichiers senseurs, MARMOTTES devrait gérer un fichier pré-interprété image des fichiers utilisateur, et ne lire ces derniers que lorsque la mise à jour du fichier image est nécessaire (c'est à dire quand il n'existe pas, quand l'un des fichiers utilisateur a changé, ou quand la version de la bibliothèque a changé). Ce fichier n'a pas à être vu par les utilisateurs (on pourrait le nommer `.senseurs.en` si le fichier de base s'appelait `senseurs.en` ; il pourrait même être binaire.

Il faudrait utiliser la bibliothèque MADONA pour lire le fichier des senseurs (ceci suppose une extension des possibilités de MADONA, en particulier au niveau des inclusions de fichiers et des indirections de blocs). Une autre extension dans le même esprit serait la reconnaissance de fichiers XML. Des utilitaires de conversions de formats devraient également être crées. Des utilitaires d'aide à la modélisation de capteurs seraient également les bienvenus.

Les champs de vue les plus classiques sont des double dièdres ; il serait pratique de disposer d'une méthode plus simple que l'intersection de deux dièdres pour les décrire. Les informations nécessaires sont le demi-angle d'ouverture, la direction de visée et l'orientation autour de cette direction.

Certains senseurs présents sur les satellites ne diffèrent de senseurs modélisés que par une fonction de conversion de la mesure. Il faudrait permettre à l'utilisateur de spécifier de telles fonctions, à la fois en mesure et en

consigne. Deux voies sont possibles pour cela, pas forcément incompatibles. Dans le premier cas l'utilisateur enregistre au niveau de son code la fonction associée à un senseur particulier, la bibliothèque appelant cette fonction en temps utile. Dans le second cas l'utilisateur exprime la fonction de transfert directement dans le fichier senseur, dans un bloc optionnel (sous forme d'une chaîne de caractères). La première méthode est plus facile à mettre en œuvre, plus souple, mais viole complètement le principe d'indépendance du code par rapport aux senseurs. La seconde méthode est plus complexe, mais respecte ce principe.

Les senseurs modélisant un bilan de liaison en fonction de la position dans un lobe d'antenne sont actuellement limités à deux fonctions spécifiques : les lobes gaussiens ou en sinus cardinal carré et aux échantillonnages à symétrie axiale. Il serait intéressant de prendre en compte des lobes définis par la valeur du gain en divers points d'échantillonnages bidimensionnels. Ces points ne sont pas forcément régulièrement répartis mais peuvent très bien être dispersés. Une méthode classique (quoi que peu utilisée sur la sphère unité) pour évaluer le gain en un point quelconque passe par une triangulation de DELAUNAY sur les points d'échantillonnage puis à faire un calcul barycentrique sur les triangles.

Certains instruments de la charge utile du satellite pourraient être modélisés sous forme de capteurs. Il serait souhaitable de pouvoir fournir à l'utilisateur une description de la fauchée de ces instruments sur le sol.

Les modèles de position du soleil ou de corps central ne sont adaptés qu'à une utilisation autour de la terre, il faudrait permettre aux utilisateurs de spécifier leurs propres modèles.

Les tests de non régression internes de la bibliothèque sont trop peu nombreux et de trop haut niveau. Il faudrait étoffer cette batterie et améliorer les tests existants. Un pas a déjà été fait dans le sens de l'enrichissement (mais pas du niveau) avec l'introduction de tests issus de programmes utilisateurs réels.

Il serait souhaitable de traduire l'ensemble de la bibliothèque en anglais, pour faciliter son déploiement.

11 description des routines

La bibliothèque MARMOTTES encapsule complètement les données utiles pour la résolution d'attitude (senseur, modèles à un degré de liberté, conservation des états précédents pour les calculs cinématiques). Elle gère des structures internes regroupant ces informations. Plusieurs instances complètement indépendantes de ces structures peuvent exister simultanément, on qualifie chacune de ces instances de *simulateur marmottes*. Toutes ces instances sont regroupées dans une table. Un programme souhaitant utiliser MARMOTTES doit réserver les simulateurs dont il a besoin dans cette table (on peut imaginer qu'un programme ait besoin de plusieurs simulateurs, par exemple un représentant l'état courant du satellite et un autre représentant un état potentiel si d'autres senseurs étaient utilisés, pour tester quand on peut passer d'un jeu de consignes à un autre). Toutes les interfaces fonctionnelles avec MARMOTTES excepté la gestion des traces d'exécution utilisent un numéro identifiant le simulateur sur lequel l'appelant travaille (il s'agit tout simplement de l'indice dans la table, sachant que l'indice 0 n'est jamais utilisé). Il n'y a qu'une seule table pour tous les simulateurs, partagée par les interfaces C et FORTRAN. En C++, on gère directement des instances de la classe marmottes, il n'y a pas besoin de table.

11.1 Création

Pour réserver une entrée dans la table, on *crée* un simulateur en donnant un état initial (date, position, attitude, ...) et MARMOTTES renvoie un identificateur entier[14] qui permettra par la suite au programme de rappeler à la bibliothèque sur quelle instance il demande des calculs. En cas d'erreur, l'identificateur retourné est nul.

interface FORTRAN :

integer function **MarmottesCreer** (*date*, *position*, *vitesse*, *attitude*, *spin*, *fichier*, *senseur1*, *senseur2*, *senseur3*, *messageErreur*)
double precision *date*, *position* (3), *vitesse* (3), *attitude* (4), *spin* (3)
character*(*) *fichier*, *senseur1*, *senseur2*, *senseur3*, *messageErreur*

interface C :

```
#include "marmottes/InterfaceC.h"
IdentMarmottes MarmottesCreer (double date,
                               const double position [3], const double vitesse [3],
                               const double attitude [4], const double spin [3],
                               const char *fichier, const char *senseur1,
                               const char *senseur2, const char *senseur3,
                               char *messageErreur, int lgMaxMessage)
```

interface C++ :

```
#include "marmottes/Marmottes.h"
Marmottes::Marmottes (double date,
                      const VecDBL &position, const VecDBL &vitesse,
                      const RotDBL &attitude, const VecDBL &spin,
                      const string &fichier,
                      const string &senseur1, const string &senseur2, const string &senseur3)
```

date, *position*, *vitesse*, *attitude*, et *spin* permettent d'initialiser l'instance créée. Si on ignore l'attitude initiale, on peut utiliser la rotation identité (1, 0, 0, 0) et un spin nul (0, 0, 0).

fichier indique le nom du fichier des senseurs. *senseur1*, *senseur2*, *senseur3* décrivent la liste des senseurs utilisés pour contrôler l'attitude. Si une erreur se produit un message d'erreur est retourné à l'appelant.

Il est également possible de *copier* un simulateur. Ceci est en particulier utile pour éviter de réanalyser le fichier des senseurs (ce qui est long) lorsqu'un programme doit créer et détruire un grand nombre de simulateurs. Pour cela on commence par créer un simulateur de référence et on appelle les fonctions de test de contrôlabilité en ignorant leur résultat, uniquement pour leur effet de bord qui est de charger dans le simulateur le senseur s'il n'y est pas déjà. On dispose alors d'un simulateur disposant d'une base de senseurs riche. Il suffit alors de copier ce simulateur pour créer un simulateur de travail sans qu'il soit nécessaire à la bibliothèque de réanalyser le fichier pour trouver les senseurs utiles, ceux-ci seront déjà dans la copie.

interface FORTRAN :

[14] En C et C++, l'identificateur est un IdentMarmottes (qui est un typedef sur un int)

integer function **MarmottesCopie** (*ident*, *messageErreur*)
integer *ident*
character*(*) *messageErreur*

interface C :
#include "marmottes/InterfaceC.h"
IdentMarmottes **MarmottesCopie** (IdentMarmottes *ident*,
char **messageErreur*, int *lgMaxMessage*)

interface C++ :
#include "marmottes/Marmottes.h"
Marmottes::**Marmottes** (const Marmottes &m)
Marmottes& Marmottes::**operator** = (const Marmottes &m)

ident (ou *m*) permet de décrire l'instance à copier.

Si l'on utilise en boucle des séries de simulateurs s'appuyant sur les mêmes senseurs, on peut *réinitialiser* un simulateur de travail plutôt que de le détruire et d'en recréer un à chaque itération. Là encore les informations déjà lues dans le fichier sont préservées ce qui évite de réanalyser le fichier pour trouver les senseurs utiles, ceux-ci seront déjà dans la copie.

interface FORTRAN :
integer function **MarmottesReinitialisation** (*ident*, *date*,
position, *vitesse*,
attitude, *spin*,
senseur1, *senseur2*, *senseur3*,
messageErreur)
double precision *date*, *position* (3), *vitesse* (3), *attitude* (4), *spin* (3)
character*(*) *senseur1*, *senseur2*, *senseur3*, *messageErreur*

interface C :
#include "marmottes/InterfaceC.h"
int **MarmottesReinitialisation** (IdentMarmottes *ident*,
double *date*,
const double *position* [3], const double *vitesse* [3],
const double *attitude* [4], const double *spin* [3],
const char **senseur1*, const char **senseur2*,
const char **senseur3*,
char **messageErreur*, int *lgMaxMessage*)

interface C++ :
#include "marmottes/Marmottes.h"
void Marmottes::**reinitialise** (double *date*,
const VecDBL &*position*, const VecDBL &*vitesse*,
const RotDBL &*attitude*, const VecDBL &*spin*,
const string &*fichier*,
const string &*senseur1*, const string &*senseur2*, const string &*senseur3*)

ident permet de décrire l'instance à réinitialiser.

date, *position*, *vitesse*, *attitude*, et *spin* permettent d'initialiser l'instance créée. Si on ignore l'attitude initiale, on peut utiliser la rotation identité (1, 0, 0, 0) et un spin nul (0, 0, 0).

senseur1, *senseur2*, *senseur3* décrivent la liste des senseurs utilisés pour contrôler l'attitude.

fichier indique le nom du fichier des senseurs. Si ce nom est identique au nom précédent, le fichier n'est pas relu et la table des senseurs n'est pas réinitialisée (c'est ce qui se passe systématiquement pour les interfaces FORTRAN et C, le nom n'étant pas paramétrable à l'appel des fonctions).

Si une erreur se produit un code de retour non nul et un message d'erreur sont retournés à l'appelant.

11.2 Destruction

Lorsqu'une instance devient inutile, on peut la *détruire* pour que l'entrée correspondante de la table interne de la bibliothèque soit à nouveau disponible.

interface FORTRAN :

```
subroutine MarmottesDetruire (ident)
integer ident
```

interface C :

```
#include "marmottes/InterfaceC.h"
void MarmottesDetruire (IdentMarmottes ident)
```

interface C++ :

```
#include "marmottes/Marmottes.h"
Marmottes::~Marmottes ()
```

ident permet de décrire l'instance à détruire.

11.3 Senseurs de contrôle

Si l'on désire changer les senseurs de contrôle d'une instance de simulateur, on redonne les noms des trois senseurs désirés. Il faut redonner les noms des trois senseurs même si un seul change, et l'ordre est important (il est lié à l'ordre des mesures dans les fonctions de résolution d'attitude).

interface FORTRAN :

```
integer function MarmottesSenseurs (ident,
                                    senseur1, senseur2, senseur3,
                                    messageErreur)
integer          ident
character*(*) senseur1, senseur2, senseur3, messageErreur
```

interface C :

```
#include "marmottes/InterfaceC.h"
int MarmottesSenseurs (IdentMarmottes ident, const char *senseur1,
                       const char *senseur2, const char *senseur3,
                       char *messageErreur, int lgMaxMessage)
```

interface C++ :

```
#include "marmottes/Marmottes.h"
void Marmottes::senseurs (const string &fichier, const string &senseur1,
                          const string &senseur2, const string &senseur3)
                  throw (ClubErreurs, MarmottesErreurs)
```

ident permet de décrire l'instance à modifier. *senseur1*, *senseur2*, *senseur3* décrivent la liste des senseurs à utiliser.

Si une erreur survient, les fonctions C et FORTRAN retournent un code non nul et initialisent le message d'erreur. Dans la fonction C++, la gestion des erreurs n'est pas faite par retour d'un code, mais par le mécanisme des exceptions.

Dans la fonction C++, on peut donner un nom de fichier différent de celui de la création. Si on veut réutiliser le même fichier, on peut le retrouver par la fonction :

```
const string &Marmottes::nomFichier () const
```

11.4 Résolution d'attitude

La fonction principale de la bibliothèque MARMOTTES est de résoudre une attitude en fonction des valeurs de consignes imposées. L'ordre dans lequel doivent être données les consignes est important, il correspond à l'ordre dans lequel ont été déclarés les senseurs de contrôle (lors de la création où à la suite d'une modification). Les fonctions suivantes implantent cette résolution.

interface FORTRAN :

```
integer function MarmottesAttitude (ident,
                                    date, position, vitesse,
                                    mesure1, mesure2, mesure3,
                                    attitude, spin,
                                    messageErreur)
integer           ident
double precision  date, position (3), vitesse (3)
double precision  mesure1, mesure2, mesure3
double precision  attitude (4), spin (3)
character*(*)     messageErreur
```

interface C :

```
#include "marmottes/InterfaceC.h"
int MarmottesAttitude (IdentMarmottes ident,
```

double *date*, const double *position* [3], const double *vitesse* [3],
double *mesure1*, double *mesure2*, double *mesure3*,
double *attitude* [4], double *spin* [3],
char* *messageErreur*, int *lgMaxMessage*)

interface C++ :

#include "marmottes/Marmottes.h"
void Marmottes::**attitude** (double *date*,
const VecDBL &*position*, const VecDBL &*vitesse*,
double *m1*, double *m2*, double *m3*,
RotDBL **attit*, VecDBL **spin*)
throw (CantorErreurs, MarmottesErreurs)

ident permet de décrire l'instance sur laquelle porte le calcul. *date*, *position*, *vitesse* donnent l'état courant du satellite. *mesure1*, *mesure2*, *mesure3* sont les consignes d'attitude pour les senseurs de contrôle en radians ou radians/seconde selon les senseurs. *attitude* et *spin* sont les résultats du calcul retournés par la bibliothèque.

Si une erreur survient, les fonctions C et FORTRAN retournent un code non nul et initialisent le message d'erreur. Dans la fonction C++, la gestion des erreurs n'est pas faite par retour d'un code, mais par le mécanisme des exceptions.

11.5 Résolution partielle d'attitude

Certains logiciels n'ont pas besoin d'une résolution complète de l'attitude du satellite, mais uniquement de la direction d'un axe particulier (par exemple la direction de poussée) ou la direction d'un astre unique connaissant deux mesures le concernant. Dans le cas où les deux consignes d'attitude sont du même type (soit géométrique soit cinématique), MARMOTTES peut réaliser une résolution partielle, beaucoup plus rapide que la résolution complète sur les trois degrés de liberté. Ce mode de fonctionnement est à utiliser avec précautions, car aucune garantie n'est fournie sur le degré de liberté non considéré, on ne peut raisonnablement profiter de cette accélération que si l'on sait exactement ce que sont les senseurs utilisés et ce que l'on fait de l'attitude fournie par la bibliothèque.

interface FORTRAN :

integer function **MarmottesDeuxConsignes** (*ident*,
date, *position*, *vitesse*, *mesure1*, *mesure2*,
attitude, *spin*, *messageErreur*)
integer *ident*
double precision *date*, *position* (3), *vitesse* (3)
double precision *mesure1*, *mesure2*
double precision *attitude* (4), argumentspin (3)
character*(*) *messageErreur*

interface C :

#include "marmottes/InterfaceC.h"
int **MarmottesDeuxConsignes** (IdentMarmottes *ident*,

double *date*,
const double *position* [3], const double *vitesse* [3],
double *mesure1*, double *mesure2*,
double *attitude* [4], double *spin* [3],
char* *messageErreur*, int *lgMaxMessage*)

interface C++ :

#include "marmottes/Marmottes.h"
void Marmottes::**deuxConsignes** (double *date*,
const VecDBL &*position*, const VecDBL &*vitesse*,
double *m1*, double *m2*,
RotDBL **attit*, VecDBL **spin*)
throw (MarmottesErreurs)

ident permet de décrire l'instance sur laquelle porte le calcul. *date*, *position*, *vitesse* donnent l'état courant du satellite. *mesure1* et *mesure2* sont les consignes d'attitude pour les senseurs de contrôle en radians ou radians/seconde selon les senseurs. *attitude* et *spin* sont les résultats du calcul retournés par la bibliothèque.

Si une erreur survient, les fonctions C et FORTRAN retournent un code non nul et initialisent le message d'erreur. Dans la fonction C++, la gestion des erreurs n'est pas faite par retour d'un code, mais par le mécanisme des exceptions.

11.6 Forçage d'attitude ou de spin

Lorsque l'évolution de l'attitude est calculée par des moyens externs à MARMOTTES (par exemple par une intégration de la dynamique) mais que l'on désire tout de même utiliser la bibliothèque pour estimer des mesures ou vérifier la contrôlabilité par exemple, alors il faut fournir à la bibliothèque ces valeurs externes pour qu'elle mette à jour ses variables internes.

Si l'on impose l'attitude, le spin est déduit par différences finies depuis l'état précédent.

interface FORTRAN :

integer function **MarmottesImposeAttitude** (*ident*,
date, *position*, *vitesse*,
attitude,
messageErreur)

integer *ident*
double precision *date*, *position* (3), *vitesse* (3)
double precision *attitude* (4)
character*(*) *messageErreur*

interface C :

#include "marmottes/InterfaceC.h"
int **MarmottesImposeAttitude** (IdentMarmottes *ident*,
double *date*, const double *position* [3], const double *vitesse* [3],
const double *attitude* [4],
char* *messageErreur*, int *lgMaxMessage*)

interface C++ :

```
#include "marmottes/Marmottes.h"
void Marmottes::attitude (double date,
                          const VecDBL &position, const VecDBL &vitesse,
                          const RotDBL &attit)
                  throw (MarmottesErreurs)
```

ident permet de décrire l'instance sur laquelle porte le calcul. *date*, *position*, *vitesse* donnent l'état courant du satellite. *attitude* représente la valeur que l'on souhaite fournir à la bibliothèque.

Si l'on impose le spin, l'attitude est déduite par intégration depuis l'état précédent.

interface FORTRAN :

```
integer function MarmottesImposeSpin (ident,
                                      date, position, vitesse,
                                      spin,
                                      messageErreur)
integer           ident
double precision  date, position (3), vitesse (3)
double precision  spin (3)
character*(*)     messageErreur
```

interface C :

```
#include "marmottes/InterfaceC.h"
int MarmottesImposeSpin (IdentMarmottes ident,
                         double date, const double position [3], const double vitesse [3],
                         const double spin [3],
                         char* messageErreur, int lgMaxMessage)
```

interface C++ :

```
#include "marmottes/Marmottes.h"
void Marmottes::imposeSpin (double date,
                            const VecDBL &position, const VecDBL &vitesse,
                            const VecDBL &spin)
                  throw (MarmottesErreurs)
```

ident permet de décrire l'instance sur laquelle porte le calcul. *date*, *position*, *vitesse* donnent l'état courant du satellite. *spin* représente la valeur que l'on souhaite fournir à la bibliothèque.

Si une erreur survient, les fonctions C et FORTRAN retournent un code non nul et initialisent le message d'erreur. Dans la fonction C++, la gestion des erreurs n'est pas faite par retour d'un code, mais par le mécanisme des exceptions.

11.7 Extraction de mesures

Dans un état donné d'une instance de simulateur, on peut calculer les mesures qui seraient produites par n'importe quel senseur (que ce soit un senseur de contrôle ou non).

Remarque : Il n'est pas possible de créer une fonction de mesure prenant en entrée l'attitude et fournissant la mesure, car dans le cas des senseurs cinématiques il faut également l'attitude précédente, et dans le cas des senseurs optiques il faut également la date et la position. En fait toutes les informations qui sont stockées d'un appel à l'autre dans une instance de simulateur sont nécessaires pour ce calcul.

interface FORTRAN :

integer function **MarmottesMesure** (*ident*, *senseur*, *mesure*, *messageErreur*)
integer *ident*
character*(*) *senseur*, *messageErreur*
double precision *mesure*

interface C :

#include "marmottes/InterfaceC.h"
int **MarmottesMesure** (IdentMarmottes *ident*, const char **senseur*,
double **mesure*, char **messageErreur*, int *lgMaxMessage*)

interface C++ :

#include "marmottes/Marmottes.h"
void Marmottes::**mesure** (const string &*fichier*, const string &*senseur*, double **m*)
throw (ClubErreurs, MarmottesErreurs)

ident permet de décrire l'instance sur laquelle porte le calcul. *senseur* donne le nom du senseur dont on désire la mesure. *mesure* est la valeur retournée par la bilbliothèque en radians ou radians par seconde selon le senseur.

Si une erreur survient, les fonctions C et FORTRAN retournent un code non nul et initialisent le message d'erreur. Dans la fonction C++, la gestion des erreurs n'est pas faite par retour d'un code, mais par le mécanisme des exceptions.

Dans la fonction C++, on peut donner un nom de fichier différent de celui de la création. Si on veut réutiliser le même fichier, on peut le retrouver par la fonction :

const string &Marmottes::**nomFichier** () const

11.8 Contrôlabilité

Pour simuler les modes de pilotages successifs d'un satellite ou pour déterminer les créneaux pendant lesquels ces modes sont possibles, il est nécessaire de tester la possibilité de basculement du contrôle d'un senseur à un autre, sans réaliser ce basculement.

interface FORTRAN :

integer function **MarmottesControlable** (*ident*, *senseur*,
controlable, *messageErreur*)
integer *ident*, *controlable*
character*(*) *senseur*, *messageErreur*

interface C :

```
#include "marmottes/InterfaceC.h"
int MarmottesControlable (IdentMarmottes ident, const char *senseur,
                          int *controlable, char *messageErreur, int lgMaxMessage)
```

interface C++ :

```
#include "marmottes/Marmottes.h"
void Marmottes::controlable (const string &fichier, const string &senseur, int *controlable)
                throw (ClubErreurs, MarmottesErreurs)
```

ident permet de décrire l'instance sur laquelle porte le test. *senseur* donne le nom du senseur que l'on désire tester. *controlable* est un entier valant 0 si l'état n'est pas contrôlable par le senseur considéré.

Il est possible d'obtenir des informations plus détaillées.

interface FORTRAN :

```
integer function MarmottesCriteresControlabilite (ident, senseur,
                                                  inhibant, eclipsant,
                                                  ecartFrontiere, amplitudeSignificative,
                                                  messageErreur)
integer           ident, inhibant, eclipsant, amplitudeSignificative
double precision  ecartFrontiere
character*(*)     senseur, messageErreur
```

interface C :

```
#include "marmottes/InterfaceC.h"
int MarmottesCriteresControlabilite (IdentMarmottes ident, const char *senseur,
                                     MarmottesAstre *inhibant, MarmottesAstre *eclipsant,
                                     double *ecartFrontiere, double *amplitudeSignificative,
                                     char *messageErreur, int lgMaxMessage)
```

interface C++ :

```
#include "marmottes/Marmottes.h"
void Marmottes::criteresControlabilite (const string &fichier, const string &senseur,
                Senseur::codeAstre *inhibant, Senseur::codeAstre *eclipsant,
                double *ecartFrontiere, bool *amplitudeSignificative)
                throw (ClubErreurs, MarmottesErreurs)
```

ident permet de décrire l'instance sur laquelle porte le test. *senseur* donne le nom du senseur que l'on désire tester. *inhibant* et *eclipsant* indiquent le code de l'astre posant problème. En FORTRAN ces indicateurs sont des entiers que l'on peut comparer aux `PARAMETER` déclarés dans le fichier `marmottes/marmdefs.f` que l'on peut inclure par `#include` ou par `INCLUDE` au niveau du fichier appelant. En langage C ces indicateurs sont des variables du type énuméré `MarmottesAstre` déclaré dans le fichier d'en-tête `marmottes/InterfaceC.h` . Pour ces deux langages, les valeurs retournées peuvent être comparées aux constantes `MrmNonSig`, `MrmSoleil`, `MrmLune`, `MrmCentral` et `MrmAucun`. En C++ ces indicateurs sont des variables du type énuméré `Senseur::codeAstre` déclaré dans le fichier d'en-tête `marmottes/Senseurs.h` . Dans ce cas, les valeurs retournées peuvent être comparées

aux constantes : `Senseur::nonSignificatif` , `Senseur::soleil`, `Senseur::lune`, `Senseur::corpsCentral` et `Senseur::aucunAstre` . *ecartFrontiere* indique l'écart angulaire entre la cible et la frontière du champ de vue, la valeur est positive lorsque la cible est dans le champ, et négative hors du champ. *amplitudeSignificative* indique si la valeur de *ecartFrontiere* est significative ou si seul le signe est valide (par exemple pour les capteurs de limbe, on ne sait pas calculer une valeur angulaire, on sait juste dire si le limbe est visible ou non). En FORTRAN et en C cette valeur est retournée sous forme d'un entier qui vaut 0 pour indiquer que l'amplitude n'est pas significative, et une valeur non nulle pour indiquer qu'elle est significative.

Si une erreur survient, les fonctions C et FORTRAN retournent un code non nul et initialisent le message d'erreur. Dans la fonction C++, la gestion des erreurs n'est pas faite par retour d'un code, mais par le mécanisme des exceptions.

Dans la fonction C++, on peut donner un nom de fichier différent de celui de la création. Si on veut réutiliser le même fichier, on peut le retrouver par la fonction :

const string &Marmottes::**nomFichier** () const

11.9 Gestion de l'extrapolation dans la résolution d'attitude

La bibliothèque MARMOTTES a la possibilité de tenter de simples extrapolations de l'attitude à partir des états précédents pour accélérer ses résolutions d'attitude, dans ce cas la résolution complète n'est lancée qu'en cas d'échec de l'extrapolation. Il s'agit là du comportement par défaut, les deux routines suivantes permettent à l'application appelante d'activer ou d'inhiber ce mode de fonctionnement.

interface FORTRAN :

integer function **MarmottesAutoriseExtrapolation** (*ident*, *messageErreur*)
integer *ident*
character*(*) *messageErreur*

integer function **MarmottesInterditExtrapolation** (*ident*, *messageErreur*)
integer *ident*
character*(*) *messageErreur*

interface C :

#include "marmottes/InterfaceC.h"
int **MarmottesAutoriseExtrapolation** (IdentMarmottes *ident*,
char* *messageErreur*, int *lgMaxMessage*)

int **MarmottesInterditExtrapolation** (IdentMarmottes *ident*,
char* *messageErreur*, int *lgMaxMessage*)

interface C++ :

#include "marmottes/Marmottes.h"
void Marmottes::**autoriseExtrapolation** ()
void Marmottes::**interditExtrapolation** ()

ident permet de décrire l'instance sur laquelle porte le calcul.

Si une erreur survient, les fonctions C et FORTRAN retournent un code non nul et initialisent le message d'erreur. Dans la fonction C++, la gestion des erreurs n'est pas faite par retour d'un code, mais par le mécanisme des exceptions.

11.10 Récupération de l'orientation des senseurs

L'orientation de base d'un senseur par rapport au satellite peut être récupérée par les fonctions suivantes. Le repère de base est le repère défini dans le fichier des ressources, il ne varie pas lorsque l'appelant modifie l'orientation courante d'un senseur.

interface FORTRAN :

```
integer function MarmottesRepereBase (ident, senseur, r, messageErreur)
integer           ident
double precision  r (4)
character*(*)     senseur, messageErreur
```

interface C :

```
#include "marmottes/InterfaceC.h"
int MarmottesRepereBase (IdentMarmottes ident, const char *senseur,
                         double r [4], char *messageErreur, int lgMaxMessage)
```

interface C++ :

```
#include "marmottes/Marmottes.h"
void Marmottes::repereBase (const string &fichier, const string &senseur, RotDBL *r)
                    throw (ClubErreurs, MarmottesErreurs)
```

ident permet de décrire l'instance sur laquelle porte le calcul. *senseur* donne le nom du senseur que l'on désire tester. *r* donne la rotation définissant l'orientation du senseur.

Si le senseur a été réorienté par rapport à son repère de base, on peut récupérer cette nouvelle orientation par les fonctions suivante.

Si une erreur survient, les fonctions C et FORTRAN retournent un code non nul et initialisent le message d'erreur. Dans la fonction C++, la gestion des erreurs n'est pas faite par retour d'un code, mais par le mécanisme des exceptions.

interface FORTRAN :

```
integer function MarmottesRepere (ident, senseur, r, messageErreur)
integer           ident
double precision  r (4)
character*(*)     senseur, messageErreur
```

interface C :

```
#include "marmottes/InterfaceC.h"
int MarmottesRepere (IdentMarmottes ident, const char *senseur,
                     double r [4], char *messageErreur, int lgMaxMessage)
```

interface C++ :

#include "marmottes/Marmottes.h"
void Marmottes::**repere** (const string &*fichier*, const string &*senseur*, RotDBL *r)
throw (ClubErreurs, MarmottesErreurs)

ident permet de décrire l'instance sur laquelle porte le test. *senseur* donne le nom du senseur que l'on désire tester. r donne la rotation définissant l'orientation du senseur.

Si une erreur survient, les fonctions C et FORTRAN retournent un code non nul et initialisent le message d'erreur. Dans la fonction C++, la gestion des erreurs n'est pas faite par retour d'un code, mais par le mécanisme des exceptions.

Dans les fonctions C++, on peut donner un nom de fichier différent de celui de la création. Si on veut réutiliser le même fichier, on peut le retrouver par la fonction :

const string &Marmottes::**nomFichier** () const

11.11 Modification de l'orientation des senseurs

Le programme appelant peut modifier l'orientation d'un senseur par rapport à l'orientation définie dans le fichier de ressources. Cette modification peut être soit arbitraire (cela peut être utile pour des pseudo-senseurs) auquel cas l'appelant doit définir entièrement le repère, soit limitée à la prise en compte d'un calage autour d'un axe prédéfini (par exemple pour traiter les senseurs fixés sur des panneaux solaires ou sur des réflecteurs d'antennes). Le calage est toujours compté comme un angle absolu à partir du repère de base. Les fonctions de prise en compte de calage ne fonctionnent que si un axe de calage autour duquel tourner a été prédéfini dans le senseur au niveau du fichier de ressources.

interface FORTRAN :

integer function **MarmottesNouveauRepere** (*ident*, *senseur*, *r*, *messageErreur*)
integer *ident*
double precision *r* (4)
character*(*) *senseur*, *messageErreur*

integer function **MarmottesCalage** (*ident*, *senseur*, *c*, *messageErreur*)
integer *ident*
double precision *c*
character*(*) *senseur*, *messageErreur*

interface C :

#include "marmottes/InterfaceC.h"
int **MarmottesNouveauRepere** (IdentMarmottes *ident*, const char *senseur,
double *nouveau* [4], char *messageErreur,
int *lgMaxMessage*)

int **MarmottesCalage** (IdentMarmottes *ident*, const char *senseur,
double *c*, char *messageErreur, int *lgMaxMessage*)

interface C++ :

```
#include "marmottes/Marmottes.h"
void Marmottes::nouveauRepere (const string &fichier, const string &senseur,
                               const RotDBL& nouveau)
                throw (ClubErreurs, MarmottesErreurs)
void Marmottes::calage         (const string &fichier, const string &senseur, double c)
                throw (ClubErreurs, MarmottesErreurs)
```

ident permet de décrire l'instance sur laquelle porte le calcul. *senseur* donne le nom du senseur que l'on désire tester. *nouveau* donne la rotation définissant l'orientation du senseur. *c* donne la valeur du calage en radians.

Si une erreur survient, les fonctions C et FORTRAN retournent un code non nul et initialisent le message d'erreur. Dans la fonction C++, la gestion des erreurs n'est pas faite par retour d'un code, mais par le mécanisme des exceptions.

11.12 Modification de la cible des senseurs optiques

Le programme appelant peut modifier la cible d'un senseur dérivant d'un senseur optique lorsque celui-ci a été prévu en ce sens (c'est-à-dire si la cible spécifiée dans le fichier de ressources est `position` ou `direction`).

interface FORTRAN :

```
integer function MarmottesModifieCible (ident, senseur, cible, messageErreur)
integer          ident
double precision cible (3)
character*(*)    senseur, messageErreur
```

interface C :

```
#include "marmottes/InterfaceC.h"
int MarmottesModifieCible (IdentMarmottes ident, const char *senseur,
                           double cible [3], char *messageErreur,
                           int lgMaxMessage)
```

interface C++ :

```
#include "marmottes/Marmottes.h"
void Marmottes::modifieCible (const string &fichier, const string &senseur,
                              const VecDBL& cible)
                throw (ClubErreurs, MarmottesErreurs)
```

ident permet de décrire l'instance sur laquelle porte le calcul. *senseur* donne le nom du senseur dont on modifie la cible. *cible* donne la nouvelle cible du senseur, dans l'unité de position courante. Si la cible attendue est une `direction`, les unités n'ont pas d'importance (le vecteur sera normalisé à l'utilisation), si la cible attendue est une `position` en revanche, il faut prendre garde que l'unité courante au moment de la mémorisation est utilisée pour convertir la cible en argument, et que l'unité courante au moment des résolutions d'attitude est utilisée pour convertir le position courante du satellite. Il est tout à fait possible de changer d'unités entre le moment où l'on initialise la cible et le moment où on l'utilise, mais il faut faire attention à la cohérence des

arguments que l'on a envoyé aux diverses fonctions. Une erreur est retournée si le senseur ne dérive pas d'un senseur optique ou si sa cible n'est pas modifiable.

Le repère d'expression de la cible doit être identique à celui utilisé pour les position-vitesse du satellite.

Si une erreur survient, les fonctions C et FORTRAN retournent un code non nul et initialisent le message d'erreur. Dans la fonction C++, la gestion des erreurs n'est pas faite par retour d'un code, mais par le mécanisme des exceptions.

11.13 Initialisation des gyromètres intégrateurs

Le programme appelant peut initialiser (ou réinitialiser) les gyromètres intégrateurs en donnant la mesure qu'ils devraient fournir à une date donnée, MARMOTTES se chargeant de faire évoluer le senseur de façon interne même s'il n'est pas utilisé en contrôle et que l'on n'extrait pas ses mesures pendant plusieurs pas.

interface FORTRAN :

```
integer function MarmottesInitialiseGyro (ident, senseur, date, angle, messageErreur)
integer           ident
double precision  date, angle
character*(*)     senseur, messageErreur
```

interface C :

```
#include "marmottes/InterfaceC.h"
int MarmottesInitialiseGyro (IdentMarmottes ident, const char *senseur,
                             double date, double angle,
                             char *messageErreur, int lgMaxMessage)
```

interface C++ :

```
#include "marmottes/Marmottes.h"
void Marmottes::initialiseGyro (const string &fichier, const string &senseur,
                                double date, double angle)
                    throw (ClubErreurs, MarmottesErreurs)
```

ident permet de décrire l'instance sur laquelle porte le calcul. *senseur* donne le nom du gyromètre intégrateur que l'on désire initialiser. *date* et *angle* donnent la référence d'initialisation du gyromètre intégrateur. Si *date* ne correspond pas à la date courante, il faut prendre garde que MARMOTTES extrapolera entre cette date et la date courante avec le spin courant lorsqu'il intègrera le mouvement du premier pas. Il est donc recommandé de limiter l'écart entre ces deux dates si l'on veut conserver une bonne précision. L'identité entre ces dates n'est pas imposée afin de permettre la prise en compte de dates de réinitialisation tombant entre les pas de calculs (cas typique d'une lecture d'écho de télémesure). Une erreur est retournée si le senseur n'est pas un gyromètre intégrateur.

Si une erreur survient, les fonctions C et FORTRAN retournent un code non nul et initialisent le message d'erreur. Dans la fonction C++, la gestion des erreurs n'est pas faite par retour d'un code, mais par le mécanisme des exceptions.

11.14 Initialisation de la dérive d'un senseur cinématique

Le programme appelant peut initialiser la dérive d'un senseur cinématique en donnant la valeur de celle-ci et le nom du senseur auquel elle s'applique.

interface FORTRAN :

```
integer function MarmottesInitialiseDerive (ident, senseur, derive,messageErreur)
integer           ident
double precision  derive
character*(*)     senseur, messageErreur
```

interface C :

```
#include "marmottes/InterfaceC.h"
int MarmottesInitialiseDerive (IdentMarmottes ident, const char *senseur,
                               double derive,
                               char *messageErreur, int lgMaxMessage)
```

interface C++ :

```
#include "marmottes/Marmottes.h"
void Marmottes::initialiseDerive (const string &fichier, const string &senseur,
                                  double derive
                        throw (ClubErreurs, MarmottesErreurs)
```

ident permet de décrire l'instance sur laquelle porte le calcul. *senseur* donne le nom du senseur cinématique auquel la dérive doit être appliquée. *derive* est la valeur de la dérive à appliquer. Une erreur est retournée si le senseur n'est pas un senseur cinématique.

Si une erreur survient, les fonctions C et FORTRAN retournent un code non nul et initialisent le message d'erreur. Dans la fonction C++, la gestion des erreurs n'est pas faite par retour d'un code, mais par le mécanisme des exceptions.

11.15 Modification du repère de référence des senseurs de Cardan

Le programme appelant peut modifier le repère de référence d'un senseur de Cardan lorsque celui-ci a été prévu en ce sens (c'est à dire si la référence spécifiée dans le fichier de ressources est `utilisateur`).

interface FORTRAN :

```
integer function MarmottesModifieReference (ident, senseur, reference, messageErreur)
integer           ident
double precision  reference (4)
character*(*)     senseur, messageErreur
```

interface C :

```
#include "marmottes/InterfaceC.h"
int MarmottesModifieReference (IdentMarmottes ident, const char *senseur,
                               double reference [4], char *messageErreur,
                               int lgMaxMessage)
```

interface C++ :

#include "marmottes/Marmottes.h"

void Marmottes::**modifieReference** (const string &*fichier*, const string &*senseur*, const RotDBL& *reference*)
throw (ClubErreurs, MarmottesErreurs)

ident permet de décrire l'instance sur laquelle porte le calcul. *senseur* donne le nom du senseur dont on modifie le repère de référence. *reference* donne le nouveau repère de référence du senseur. Le repère de référence est une rotation qui transforme un vecteur projeté en repère inertiel en lui-même projeté dans le repère de référence. Cette convention est similaire à la convention sur les attitudes (voir 3.1), ceci permet d'utiliser une attitude issue d'une résolution par un simulateur MARMOTTES pour initialiser les senseurs de Cardan d'un autre simulateur et faire des calculs sur les erreurs de pilotage autour de cette attitude considérée comme une référence théorique. Une erreur est retournée si le senseur n'est pas un senseur de Cardan ou si son repère de référence n'est pas modifiable.

Si une erreur survient, les fonctions C et FORTRAN retournent un code non nul et initialisent le message d'erreur. Dans la fonction C++, la gestion des erreurs n'est pas faite par retour d'un code, mais par le mécanisme des exceptions.

11.16 Modification des modèles d'éphémérides des corps célestes

Par défaut, MARMOTTES dispose de modèles internes pour le calcul du temps sidéral, des éphémérides (par rapport au corps central) du Soleil, de la Lune et de la Terre, ainsi que des caractéristiques du corps central (rayon équatorial, aplatissement, vitesse angulaire de rotation du corps sur lui-même) et des rayons des astres inhibants.
Si ces modèles ne conviennent pas à l'utilisateur (par exemple : corps central différent de la Terre et autre référentiel dans un cadre interplanétaire, ...), alors ces modèles peuvent être personnalisés en fournissant des valeurs numériques et des fonctions de calcul appropriées.

Les unités sont obligatoirement des kilomètres pour les distances, des radians pour les angles et des radians par seconde pour la vitesse angulaire de rotation.
Le temps sidéral doit être donné entre 0 et 2π.
Les éphémérides de la Lune n'ont de sens que dans le cas où le corps central est la Terre.
Les éphémérides de la Terre n'ont de sens que dans le cas où le corps central n'est pas la Terre.

Par défaut, les modèles et rayons des astres utilisés sont tels que :

- corps central = Terre
- rayon équatorial vaut 6378.14 `km`
- aplatissement vaut 1.0/298.257
- vitesse angulaire de rotation de la Terre vaut $7.29211514670519379\,10^{-5}$ `rad/s`
- rayon de la Lune vaut 1737.4 `km`
- rayon du Soleil vaut 695500 `km`
- calcul du temps sidéral (en `rad`) dans le repère de Veis ($\widehat{\gamma}_{50}$)
- théorie de Brown pour les éphémérides de la Lune
- théorie de Newcomb pour les éphémérides du Soleil
- position de la Terre vaut 0 (car corps central = Terre)

Pour personnaliser ces modèles ou/et rayons des astres, l'utilisateur doit donner des valeurs numériques et des fonctions de calcul respectant une certaine convention d'appel. Les fonctions d'interface suivantes lui permettent d'enregistrer à la fois les valeurs et les fonctions de calcul souhaitées.

Il n'est pas obligatoire de tout personnaliser.

interface FORTRAN :

integer function **MarmottesEnregistreCorps** (*ident*, *rayonEquatorial*, *aplatissement*, *vitesseRotation*, *rayonLune*, *rayonSoleil*, *tsidFonc*, *soleilFonc*, *luneFonc*, *terreFonc*, *messageErreur*)

```
integer             ident
double precision    rayonEquatorial
double precision    aplatissement
double precision    vitesseRotation
double precision    rayonLune
double precision    rayonSoleil
double precision    tsidFonc
external            soleilFonc
external            luneFonc
external            terreFonc
character*(*)       messageErreur
```

interface C :

```
#include "marmottes/InterfaceC.h"
int MarmottesEnregistreCorps (IdentMarmottes ident,
                              double rayonEquatorial, double aplatissement,
                              double vitesseRotation, double rayonLune, double rayonSoleil,
                              TypeFuncTsidC *tsidFonc, TypeFuncPosC *soleilFonc,
                              TypeFuncPosC *luneFonc, TypeFuncPosC *terreFonc,
                              char *messageErreur, int lgMaxMessage)
```

avec la définition des signatures des fonctions via les typedef suivants :

```
typedef double TypeFuncTsidC (double, double)
typedef void TypeFuncPosC (double, double [3])
```

interface C++ :

```
#include "marmottes/Marmottes.h"
void Marmottes::EnregistreCorps (double rayonEquatorial, double aplatissement,
                                 double vitesseRotation, double rayonLune,
                                 double rayonSoleil, BodyEphemC::TypeFuncTsid *tsidFonc,
                                 BodyEphemC::TypeFuncPos *soleilFonc, BodyEphemC::TypeFuncPos *luneFonc,
                                 BodyEphemC::TypeFuncPos *terreFonc)
```

avec la définition des signatures des fonctions via des typedef similaires à ceux définis pour l'interface C

ident permet de décrire l'instance sur laquelle porte le calcul.

rayonEquatorial, *aplatissement* et *vitesseRotation* permettent d'initialiser les valeurs du rayon équatorial, de l'aplatissement et de la vitesse angulaire de rotation du corps central. Dans le cas où l'une des valeurs est négative, alors la valeur par défaut est utilisée.

rayonLune et *rayonSoleil* permettent d'initialiser les valeurs du rayon de la Lune et du Soleil. Ces données sont maintenant prise en compte dans le calcul de l'inhibition alors qu'avant n'était pris en compte que le centre de l'astre. Dans le cas où l'une des valeurs est négative, alors la valeur par défaut est utilisée.

tsidFonc donne accès à la fonction utilisateur ou celle par défaut, de calcul du temps sidéral.

soleilFonc, *luneFonc* et *terreFonc* donnent accès (respectivement) à la fonction utilisateur ou celle par défaut, de calcul de la position du Soleil, de la Lune et de la Terre par rapport au corps central. *À noter* : pour le FORTRAN, il s'agit de sous-routines.

Attention : à partir de l'interface C, il suffit de passer des pointeurs nuls pour utiliser les fonctions par défaut.

Dans le cas de l'interface FORTRAN, par contre, l'utilisateur ne peut utiliser de pointeurs de fonction nuls et doit obligatoirement utiliser les méthodes par défaut (fonctions **MarmottesTempsSideralParDefaut** , **MarmottesPositionSoleilParDefaut** , **MarmottesPositionLuneParDefaut** , **MarmottesPositionTerreParDefaut**).

Si une erreur se produit un code retour non nul et un message d'erreur sont retournés à l'appelant.

La signature des fonctions est décrite ci-après en fonction du langage utilisé.

Signature des fonctions et routines utilisateurs pour le FORTRAN

Calcul du temps sidéral

double precision	function	**tsidFonc** (*t*,*decalage*)
double precision	*t*	
double precision	*decalage*	

Calcul des éphémérides du Soleil par rapport au corps central

subroutine	**soleilFonc** (*t*, *corpsSoleil*)
double precision	*t*
double precision	*corpsSoleil* (3)

Calcul des éphémérides de la Lune par rapport au corps central

subroutine **luneFonc** (t, *corpsLune*)
double precision t
double precision *corpsLune* (3)

Calcul des éphémérides de la Terre par rapport au corps central

subroutine **terreFonc** (t, *corpsTerre*)
double precision t
double precision *corpsTerre* (3)

Signature des fonctions utilisateurs pour le C et le C++

Calcul du temps sidéral

double **tsidFonc** (double t, double *decalage*)

Calcul des éphémérides du Soleil par rapport au corps central

void **soleilFonc** (double t, double *corpsSoleil*[3])

Calcul des éphémérides de la Lune par rapport au corps central

void **luneFonc** (double t, double *corpsLune*[3])

Calcul des éphémérides de la Terre par rapport au corps central

void **terreFonc** (double t, double *corpsTerre*[3])

Avec les arguments :

t	:	la date courante en `jour`, par rapport à une date de référence
decalage	:	écart de datation entre l'échelle de temps utilisé pour la date t et l'échelle de temps utilisé par le modèle de calcul du temsp sidéral (unité : en général en `s`).
tsidFonc	:	le temps sidéral à la date t (en `rad` et $\in [0, 2\pi]$)
corpsSoleil(3)	:	la position du Soleil à la date t par rapport au corps central (en `km`).
corpsLune(3)	:	la position de la Lune à la date t par rapport au corps central (en `km`).
corpsTerre(3)	:	la position de la Terre à la date t par rapport au corps central (en `km`).

La notion de date dans MARMOTTES n'impose que deux choses : l'unité est le jour, et la date t, utilisée au niveau des fonctions de calcul utilisateur, doit être cohérente avec les autres dates passées à MARMOTTES. Par ailleurs, les calculs dans MARMOTTES n'utilisent que des différences entre dates. Donc la date de référence est sans importance au niveau de MARMOTTES, on peut utiliser des jours juliens, des jours juliens modifiés, des jours juliens CNES, ... Seuls les modèles définis par l'utilisateur nécessitent une notion de date de référence.

Exemple :

Dans ce paragraphe, nous allons donner un exemple d'utilisation dans le cadre du FORTRAN.

Cet exemple considère la Terre comme corps central, mais avec des grandeurs physiques différentes des valeurs par défaut. Dans notre exemple, nous proposons également une redéfinition du temps sidéral et du calcul des éphémérides du Soleil.

Expl. 17 – *redéfinition de certaines valeurs physiques par l'utilisateur*

```
        ...
       double precision r_terre, aplatissement, vit_bidon
C rayon équatorial terrestre utilisateur
       r_terre = 6378.39d0
C aplatissement utilisateur
       aplatissement = 1.d0/298.256d0
C vitesse de rotation non définie par l'utilisateur
       vit_bidon = -1.d0
        ...
```

Dans cet exemple, l'utilisateur ne souhaite pas redéfinir la vitesse angulaire de rotation, donc il suffit de mettre une valeur négative.

De plus il ne souhaite ni redéfinir les rayons de la Lune et du Soleil ni redéfinir les modèles pour le calcul de la Lune (et dans ce cas le modèle pour la Terre n'a pas de sens).

Expl. 18 – *redéfinition de certaines fonctions par l'utilisateur*

```
C fonction de calcul du temps sidéral
       double precision function monTsid ( t, decalage)
       double precision t, decalage
C calcul du temps sidéral selon le modède retenu par
C l'utilisateur en fonction de la date t et de l'écart de datation
C (la valeur du temps sidéral est recalée dans [0,2pi])
        ...
        monTsid = ...
        end

C fonction de calcul des éphémerides du Soleil
        subroutine monSun ( t, corpsSoleil )
        double precision t, corpsSoleil(3)
C calcul de la position du Soleil par rapport au corps central =
C la Terre, selon le modèle retenu par l'utilisateur.
C Les positions sont calculées en km, en fonction de la date t
        ...
        corpsSoleil(1) = ...
        corpsSoleil(2) = ...
        corpsSoleil(3) = ...
        end
```

Il est alors possible de prendre en compte ces nouvelles caractéristiques du corps central et ces nouveaux modèles au niveau de MARMOTTES de la façon suivante :

Expl. 19 – *utilisation des valeurs et fonctions redéfinies par l'utilisateur*

```
integer NULL
...
double precision monTsid
external monSun
external MarmottesPositionLuneParDefaut
external MarmottesPositionTerreParDefaut
...
if (MarmottesEnregistreCorps (ident, r_terre,
>     aplatissement, vit_bidon, -1, -1,
>     monTsid, monSun,
>     MarmottesPositionLuneParDefaut,
>     MarmottesPositionTerreParDefaut,
>     messageErreur) .ne. 0 ) then

      write (0, *) messageErreur (1:lnblnk (messageErreur))
      stop
endif
```

11.17 Réglage des unités

Par défaut, les positions sont exprimées en kilomètres et les vitesse en kilomètres par seconde et les échanges avec les senseurs ont lieu dans leur unité interne (voir 3.2). On peut changer ces conventions d'interface. L'exemple suivant montre comment utiliser les unités externes pour lire directement des consignes dans la télémesure en degrés ou degrés par seconde pour par exemple faire une calibration (fonction **MarmottesConvertirConsignes**) et pour afficher les résidus de calibration également en degrés ou degrés par seconde sur les mêmes senseurs (fonction **MarmottesConvertirMesures**), en laissant la bibliothèque MARMOTTES gérer seule les conversions :

Expl. 20 – *conversions d'unités*

```
  do 10 i = 1, n
    if ((MarmottesConvertirConsignes (id, senseur (i), msg) .ne. 0)
>       .or.
>       (MarmottesConvertirMesures   (id, senseur (i), msg) .ne. 0)
>      ) then
      write (0, *) msg (1:lnblnk (msg))
      stop
    endif
10  continue
```

interface FORTRAN :

integer function **MarmottesUnitesPositionVitesse** (*ident*, *unitePos*, *uniteVit*, *messageErreur*)

integer *ident*
character*(*) *unitePos*, *uniteVit*
character*(*) *messageErreur*

integer function **MarmottesRespecterConsignes** (*ident*, *senseur*, *messageErreur*)
integer *ident*
character*(*) *senseur*
character*(*) *messageErreur*

integer function **MarmottesConvertirConsignes** (*ident*, *senseur*, *messageErreur*)
integer *ident*
character*(*) *senseur*
character*(*) *messageErreur*

integer function **MarmottesRespecterMesures** (*ident*, *senseur*, *messageErreur*)
integer *ident*
character*(*) *senseur*
character*(*) *messageErreur*

integer function **MarmottesConvertirMesures** (*ident*, *senseur*, *messageErreur*)
integer *ident*
character*(*) *senseur*
character*(*) *messageErreur*

interface C :

```
#include "marmottes/InterfaceC.h"
int MarmottesUnitesPositionVitesse (IdentMarmottes ident,
                                    const char *unitePos, const char *uniteVit,
                                    char *messageErreur, int lgMaxMessage)

int MarmottesRespecterConsignes    (IdentMarmottes ident,
                                    const char *senseur,
                                    char *messageErreur, int lgMaxMessage)

int MarmottesConvertirConsignes    (IdentMarmottes ident,
                                    const char *senseur,
                                    char *messageErreur, int lgMaxMessage)

int MarmottesRespecterMesures      (IdentMarmottes ident,
                                    const char *senseur,
                                    char *messageErreur, int lgMaxMessage)

int MarmottesConvertirMesures      (IdentMarmottes ident,
                                    const char *senseur,
                                    char *messageErreur, int lgMaxMessage)
```

interface C++ :

#include "marmottes/Marmottes.h"
void Marmottes::**unitesPositionVitesse** (const string& *unitePos*,
const string& *uniteVit*)
throw (MarmottesErreurs)

void Marmottes::**respecterConsignes** (const string& *fichier*,
const string& *senseur*)
throw (ClubErreurs, MarmottesErreurs)

void Marmottes::**convertirConsignes** (const string& *fichier*,
const string& *senseur*)
throw (ClubErreurs, MarmottesErreurs)

void Marmottes::**respecterMesures** (const string& *fichier*,
const string& *senseur*)
throw (ClubErreurs, MarmottesErreurs)

void Marmottes::**convertirMesures** (const string& *fichier*,
const string& *senseur*)
throw (ClubErreurs, MarmottesErreurs)

ident permet de décrire l'instance à modifier. *unitePos* est la nouvelle unité de position, et *uniteVit* est la nouvelle unité de vitesse. Les unités supportées sont actuellement "km", "m", "km/s", et "m/s".

Si une erreur survient, les fonctions C et FORTRAN retournent un code non nul et initialisent le message d'erreur. Dans la fonction C++, la gestion des erreurs n'est pas faite par retour d'un code, mais par le mécanisme des exceptions.

senseur donne le nom du senseur à modifier.

Dans les fonctions C++, on peut donner un nom de fichier différent de celui de la création. Si on veut réutiliser le même fichier, on peut le retrouver par la fonction :

const string &Marmottes::**nomFichier** () const

11.18 Réglage de la vitesse maximale du modèle cinématique

Par défaut, la vitesse de rotation instantanée maximale du satellite utilisée dans la modélisation des attitudes contrôlées par deux senseurs cinématiques au moins est de 0,4 radians par seconde (voir [DR1]), on peut changer cette valeur par défaut :

interface FORTRAN :
integer function **MarmottesWMax** (*ident*, *omega*, *messageErreur*)
integer *ident*
double precision *omega*
character*(*) *messageErreur*

interface C :

```
#include "marmottes/InterfaceC.h"
int MarmottesWMax (IdentMarmottes ident, double omega,
                   char *messageErreur, int lgMaxMessage)
```

interface C++ :

```
#include "marmottes/Marmottes.h"
void Marmottes::wMax (double omega)
                  throw (MarmottesErreurs)
```

ident permet de décrire l'instance à modifier. *omega* est la nouvelle vitesse maximale en radians par seconde.

Si une erreur survient, les fonctions C et FORTRAN retournent un code non nul et initialisent le message d'erreur. Dans la fonction C++, la gestion des erreurs n'est pas faite par retour d'un code, mais par le mécanisme des exceptions.

11.19 Réglage du seuil de convergence

Par défaut, le seuil de convergence de la résolution vaut un dixième de la précision du troisième senseur (attention à l'éventuel réordonnancement selon les types de senseurs, voir section A). On peut changer cette valeur par défaut (attention, elle est réinitialisée à chaque changement des senseurs de contrôle) :

interface FORTRAN :

```
integer function MarmotteConvergence (ident, seuil, messageErreur)
integer          ident
double precision seuil
character*(*)    messageErreur
```

interface C :

```
#include "marmottes/InterfaceC.h"
int MarmottesConvergence (IdentMarmottes ident, double seuil,
                          char *messageErreur, int lgMaxMessage)
```

interface C++ :

```
#include "marmottes/Marmottes.h"
void Marmottes::convergence (double seuil)
                  throw (MarmottesErreurs)
```

ident permet de décrire l'instance à modifier. *seuil* est le nouveau seuil de convergence, en radians ou en radians par seconde selon le senseur.

Si une erreur survient, les fonctions C et FORTRAN retournent un code non nul et initialisent le message d'erreur. Dans la fonction C++, la gestion des erreurs n'est pas faite par retour d'un code, mais par le mécanisme des exceptions.

11.20 Réglage de la dichotomie

Par défaut, le nombre de tranches de la recherche dichotomique d'intervalles monotones est de 50, Cette valeur peut être modifiée :

interface FORTRAN :

```
integer function MarmottesDichotomie (ident, tranches, messageErreur)
integer          ident, tranches
character*(*)    messageErreur
```

interface C :

```
#include "marmottes/InterfaceC.h"
int MarmottesDichotomie (IdentMarmottes ident, int tranches,
                                 char *messageErreur, int lgMaxMessage)
```

interface C++ :

```
#include "marmottes/Marmottes.h"
void Marmottes::dichotomie (int tranches)
                    throw (MarmottesErreurs)
```

ident permet de décrire l'instance à modifier. *tranches* est le nouveau nombre de tranches de dichotomie.

Si une erreur survient, les fonctions C et FORTRAN retournent un code non nul et initialisent le message d'erreur. Dans la fonction C++, la gestion des erreurs n'est pas faite par retour d'un code, mais par le mécanisme des exceptions.

11.21 Gestion des traces d'exécution

La bibliothèque MARMOTTES étant par essence destinée à être appelée depuis des programmes extérieurs, il est difficile aux développeurs de reproduire des cas d'erreurs rencontrés par les utilisateurs finaux sans ces programmes.

Pour compenser cette absence, un mécanisme de retranscription des appels aux routines de l'interface publique a été mis en plase. En activant ce mécanisme au début de leur programme, les utilisateurs finaux peuvent créer un fichier qu'ils enverront ensuite à la maintenance qui disposera ainsi de toutes les informations nécessaires pour reproduire ce qui concerne MARMOTTES.

Ce mécanisme peut également être utilisé par les développeurs pour créer des cas tests de non-régression pour la bibliothèque elle-même.

interface FORTRAN :

```
integer function MarmottesActiveTrace (fichier, messageErreur)
character*(*) fichier, messageErreur

integer function MarmottesDesactiveTrace ()
```

interface C :

```
#include "marmottes/InterfaceC.h"
int MarmottesActiveTrace (char *fichier,
                          char *messageErreur, int lgMaxMessage)

int MarmottesDesactiveTrace (char *messageErreur, int lgMaxMessage)
```

interface C++ :

```
#include <string>
#include "marmottes/CallTrace.h"
static CallTrace *CallTrace::getInstance ()
                          throw (MarmottesErreurs)

void          CallTrace::activate (string fichier)
                          throw (MarmottesErreurs)
void          CallTrace::deactivate ()
```

fichier indique le nom du fichier dans lequel retranscrire les appels aux fonctions de l'interface publique.

Le mécanisme de trace d'exécution est géré par un objet unique (un *singleton* selon la terminologie des *design patterns*), pour appeler les méthodes **activate** et **deactivate** de la classe CallTrace, il faut donc passer par une méthode préalable **getInstance** pour récupérer un pointeur sur le singleton. L'activation du mécanisme se fait donc par une séquence d'appel du genre :

Expl. 21 – *activation des traces d'exécution*

```
CallTrace::getInstance ()->activate (string ("marmottes.log"));
```

Si une erreur survient, les fonctions C et FORTRAN retournent un code non nul et initialisent le message d'erreur. Dans la fonction C++, la gestion des erreurs n'est pas faite par retour d'un code, mais par le mécanisme des exceptions.

11.22 Accés à des données

L'accés à des données comme la récupération du pointeur sur un senseur où l'état peut s'avérer nécessaire, aussi ces possibilités sont-elles données par l'intermédiaire des méthodes décrites ci-après.

Seule l'interface C++ peut acceder à ces méthodes (accés directe aux attributs de la classe Marmottes) et c'est donc la seule explicitée.

interface C++ :

```
#include "marmottes/Marmottes.h"
Senseur * accesSenseur (const string& fichier,
                        const string& senseur)
throw (ClubErreurs, CantorErreurs, MarmottesErreurs)
```

interface C++ :

#include "marmottes/Marmottes.h"
const Etat& **etat** () const

11.23 Récupération des valeurs des parametres courants

Il est possible a tout moment de récupérer les valeurs courantes des paramètres internes à MARMOTTES : la date, la position, la vitesse, l'attitude et le spin.

interface FORTRAN :

integer function **MarmottesLireParametres** (*ident*, *date*,
position, *vitesse*,
attitude, *spin*,
messageErreur)
double precision *date*, *position* (3), *vitesse* (3), *attitude* (4), *spin* (3)
messageErreur

interface C :

#include "marmottes/InterfaceC.h"
int **MarmottesLireParametres** (IdentMarmottes *ident*,
double **date*,
const double *position* [3], const double *vitesse* [3],
const double *attitude* [4], const double *spin* [3],
char **messageErreur*, int *lgMaxMessage*)

interface C++ :

#include "marmottes/Marmottes.h"
void Marmottes::**lireParametres** (double *date*,
const VecDBL &*position*, const VecDBL &*vitesse*,
const RotDBL &*attitude*, const VecDBL &*spin*)

date, *position*, *vitesse*, *attitude*, et *spin* permettent de récupérer les valeurs courantes des paramètres au moment de l'appel.

12 description des utilitaires

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

12.1 traduitSenseurs

description générale

L'utilitaire `traduitSenseurs` permet de traduire des fichiers de capteurs de français à anglais et réciproquement.

ligne de commande et options

La ligne de commande a la forme suivante :

```
traduitSenseurs fichier
```

Le seul argument est le nom du fichier de base des capteurs (ce fichier peut en inclure d'autres). Tous les noms de fichiers doivent être de la forme `nom.en` ou `nom.fr`, le suffixe spécifiant la langue.

descriptions des sorties

L'utilitaire n'affiche rien sur sa sortie standard, il crée directement les fichiers traduits dans le même répertoire que les fichiers d'origine, en se basant sur le suffixe pour déterminer si la traduction est du français vers l'anglais ou de l'anglais vers le français.

Seuls les mot-clefs et les noms de fichiers inclus sont traduits, les commentaires sont préservés. Lorsqu'un fichier référence d'autres fichiers, ceux-ci sont traduits également.

conseils d'utilisation

Les fichiers de capteurs sont souvent écrits à la main, et les rédacteurs apportent souvent une certain attention à l'indentation des structures pour améliorer la lisibilité. Lors de la traduction, tous les blancs sont préservés, mais comme la taille des mots-clefs varie entre anglais et français, ceci perturbe l'indentation. Si l'on a besoin de maintenir les fichiers dans les deux langues, il est recommandé de faire les mises à jour manuelles toujours à partir de la même langue et de générer automatiquement les fichiers de l'autre langue. En effet si l'on travaille alternativement sur les deux langues, on finit par obtenir des portions peu lisibles dans tous les fichiers.

12.2 MarmottesReplay

description générale

L'utilitaire `marmottesReplay` permet de rejouer de façon indépendante des séries d'appels à l'interface publique de la bibliothèque MARMOTTES. Cet utilitaire a été créé à l'origine pour l'aide à la mise au point de la bibliothèque elle-même et pour l'inclusion de cas réels d'utilisation dans la suite des tests de non-régression. Il est désormais installé en même temps que la bibliothèque car il permet aux utilisateurs expérimentés d'étudier de façon rapide le comportement local d'une simulation, en changeant les paramètres ou en ajoutant à la main des appels supplémentaires avec d'autres capteurs.

ligne de commande et options

La ligne de commande a la forme suivante :

```
marmottesReplay [-o output] [-d directory] [-v] input
```

Le fichier d'entrée doit être un fichier au format produit par la bibliothèque lorsque la fonction **MarmottesActiveTrace** a été appelée (cf 11.21).

Par défaut, l'utilitaire se contente de réexécuter tous les appels et n'affiche rien. Les développeurs utilisent classiquement ce mode avec un debugger.

L'option `-o` permet d'activer les traces d'exécution, ce qui permet de comparer les nouveaux résultats avec ceux du fichier initial. Ce mode est classiquement utilisé pour les cas tests.

L'option `-d` permet de spécifier le répertoire dans lequel les fichiers senseurs doivent être recherché, la recherche se fait dans le répertoire courant par défaut.

L'option `-v` affiche sur la sortie standard tous les appels.

descriptions des sorties

Les seules sorties de l'utilitaire sont activées par les options `-o` et `-v`.

L'option `-o` crée un fichier de traces d'exécution au même format que l'entrée.

L'option `-v` affiche juste un message à chaque appel effectué.

conseils d'utilisation

Cet utilitaire nécessite une certaine connaissance de la bibliothèque pour être exploitable.

Pour les développeurs, l'intérêt principal est qu'il permet de rejouer des cas envoyés par les utilisateurs de la bibliothèque même s'ils ne disposent pas du code appelant. Ce rejeu se fait traditionnellement une fois à la main avec l'option `-o` pour vérifier que le problème cité par l'utilisateur est bien reproductible dans l'environnement de développement, puis il est refait sous debugger.

Pour les utilisateurs, il est possible d'étudier le comportement local d'une simulation en éditant à la main le fichier avant de la faire rejouer. On peut alors changer les paramètres d'appel ou ajouter des appels (les lignes de résultats des fonctions commençant par `->` étant ignorées par l'utilitaire, il est inutile d'en mettre lorsque l'on ajoute un appel à une fonction, la ligne d'appel suffit).

12.3 marmottes-config

description générale

L'utilitaire `marmottes-config` permet de déterminer les options nécessaires à la compilation d'applicatifs ou de bibliothèques s'appuyant sur MARMOTTES. 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 :

```
marmottes-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% marmottes-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% marmottes-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% marmottes-config --libs
 -lmarmottes -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% marmottes-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) `marmottes-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 \
            `marmottes-config --ldflags` $(LDFLAGS) \
            `marmottes-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 $@ `marmottes-config --cppflags --ldflags --libs` \
            $(CPPFLAGS) $(CXXFLAGS) $(LDFLAGS) $(LIBS) client.cc
```

13 description des classes

13.1 classe BodyEphem

description

Cette classe abstraite est l'interface d'accès aux différentes implémentations utilisateurs (C ou FORTRAN) du calcul du temps sidéral et des éphémérides du Soleil, de la Lune et de la Terre, par rapport au corps central. Elle permet aussi l'accès à des grandeurs physiques du corps central (rayon équatorial, aplatissement et vitesse de rotation).

Elle est dérivée en deux classes différentes.

interface publique

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

TAB. 18: BodyEphem : méthodes publiques

signature	description
BodyEphem * **clone** () const	permet la duplication sans connaître la classe dérivée employée (méthode virtuelle)
~BodyEphem()	destructeur
double **siderealTime** (double *t*, double *offset*)	calcule le temps sidéral en `rad` à la date *t* en `jour`, avec un écart de datation *offset* (généralement en `s`) entre l'échelle de temps utilisé pour la date *t* et l'échelle de temps utilisé par le modèle de calcul du temps sidéral (méthode virtuelle)
VecDBL **sunPosition** (double *t*)	calcule la position du Soleil par rapport au corps central en `km` à la date *t* (méthode virtuelle)
VecDBL **moonPosition** (double *t*)	calcule la position de la Lune par rapport au corps central en `km` à la date *t* (méthode virtuelle)
VecDBL **earthPosition** (double *t*)	calcule la position de la Terre par rapport au corps central en `km` à la date *t* (méthode virtuelle)
double **getEquatorialRadius** () const	retourne le rayon équatorial du corps central en `km` (sans conversion dans les unités utilisateur)
double **getOblateness** () const	retourne l'aplatissement du corps central
double **getRotationVelocity** () const	retourne la vitesse angulaire de rotation du corps central en `km/s` (sans conversion dans les unités utilisateur)
double **defaultFsiderealTime** (double **ptrT*, double **ptrOffset*)	fonction par défaut pour le FORTRAN de calcul du temps sidéral dans le système de référence de Veis, à la date **ptrT* (en jour) avec un écart de datation **ptrOffset* (en s) entre l'échelle de temps TU1 et l'échelle de temps utilisé pour la date **ptrT*. L'écart **ptrOffset* est ajouté à la date **ptrT*
void **defaultFsunPosition** (double **ptrT*, double *BodySun[3]*)	fonction par défaut pour le FORTRAN de calcul de la position *BodySun[3]* (en km) du Soleil par rapport à la Terre (corps central par défaut), à la date **ptrT* (en jour), selon le modèle de Newcomb
void **defaultFmoonPosition** (double **ptrT*, double *BodyMoon[3]*)	fonction par défaut pour le FORTRAN de calcul de la position *BodyMoon[3]* (en km) de la Lune par rapport à la Terre (corps central par défaut), à la date **ptrT* (en jour), selon le modèle de Brown
void **defaultFearthPosition** (double **ptrT*, double *BodyEarth[3]*)	fonction par défaut pour le FORTRAN de calcul de la position *BodyEarth[3]* (en km) de la Terre par rapport à la Terre (corps central par défaut) = 0

exemple d'utilisation

Il n'y a *aucune* utilisation directe de la classe BodyEphem dans toute la bibliothèque MARMOTTES.

conseils d'utilisation spécifiques

Cette classe est abstraite, c'est à dire qu'aucune instance ne peut être créée directement. Tout pointeur sur un objet de ce type pointe en réalité sur un objet d'un des types dérivés : BodyEphemC ou BodyEphemF. Les constructeurs ne servent donc qu'à compléter les constructions d'objets plus gros et ne peuvent être appelés que par les constructeurs des classes dérivées.

Les unités utilisées, les valeurs par défaut, un rappel sur la notion de date utilisée dans MARMOTTES et des indications sur la définition des fonctions par l'utilisateur sont donnés au niveau de la description de l'interface utilisateur (cf 11.16).

implantation

Les attributs protégés sont décrits sommairement dans la table 19, il n'y a pas d'attribut privé.

TAB. 19: attributs protégés de la classe BodyEphem

nom	type	description
defaultEquatorialRadius	static const double	valeur par défaut du rayon équatorial : pour la Terre (en km)
defaultOblateness	static const double	valeur par défaut de l'aplatissement : pour la Terre
defaultRotationVelocity	static const double	valeur par défaut de la vitesse de rotation : pour la Terre (en rad/s)
equatorialRadius_	double	rayon équatorial (en km)
oblateness_	double	aplatissement
rotationVelocity_	double	vitesse de rotation (en rad/s)

Les méthodes protégées sont décrites dans la table 20.

TAB. 20: BodyEphem : méthodes protégées

signature	description
static double **defaultCsiderealTime** (double *t*, double *offset*)	fonction par défaut pour le C de calcul du temps sidéral dans le système de référence de Veis, à la date *t* (en jour) avec un écart de datation *offset* (en s) entre l'échelle de temps TU1 et l'échelle de temps utilisé pour la date *t*. L'écart *offset* est ajouté à la date *t*
static void **defaultCsunPosition** (double *t*, double *BodySun[3]*)	fonction par défaut pour le C de calcul de la position *BodySun[3]* (en km) du Soleil par rapport à la Terre (corps central par défaut), à la date *t* (en jour), selon le modèle de Newcomb
static void **defaultCmoonPosition** (double *t*, double *BodyMoon[3]*)	fonction par défaut pour le C de calcul de la position *BodyMoon[3]* (en km) de la Lune par rapport à la Terre (corps central par défaut), à la date *t* (en jour), selon le modèle de Brown
static void **defaultCearthPosition** (double *t*, double *BodyEarth[3]*)	fonction par défaut pour le FORTRAN de calcul de la position *BodyEarth[3]* (en km) de la Terre par rapport à la Terre (corps central par défaut) = 0
BodyEphem (double *equatorialRadius*, double *oblateness*, double *rotationVelocity*)	construit une instance à partir des caractéristiques physiques du corps central : rayon équatorial *equatorialRadius* (en km), aplatissement *oblateness* et vitesse de rotation *rotationVelocity* (en rad/s)
	à suivre ...

TAB. 20: BodyEphem : méthodes protégées (suite)

signature	description
BodyEphem ()	initialise une instance par défaut
BodyEphem (const BodyEphem& *b*)	constructeur par copie
const BodyEphem& **operator** = (const BodyEphem& *b*)	affectation

13.2 classe BodyEphemC

description

Cette classe dérivée de la classe BodyEphem implante l'interface d'accès, pour l'implémentation en C par l'utilisateur, du calcul du temps sidéral et des éphémérides du Soleil, de la Lune et de la Terre, par rapport au corps central.
Elle permet aussi l'accès à des grandeurs physiques du corps central (rayon équatorial, aplatissement et vitesse de rotation).

interface publique

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

TAB. 21: BodyEphemC : méthodes publiques

signature	description
typedef double **TypeFuncTsid** (double, double)	défini la signature, pour le C, de la fonction de calcul du temps sidéral
typedef void **TypeFuncPos** (double, double [3])	défini la signature pour le C des fonctions de calcul de position des corps (Soleil, Lune et Terre) par rapport au corps central
BodyEphemC ()	initialise une instance par défaut
BodyEphemC (const BodyEphemC& *b*)	constructeur par copie
const BodyEphemC& **operator** = (const BodyEphemC& *b*)	affectation
˜**BodyEphemC** ()	destructeur
BodyEphem * **clone** () const	permet la duplication
à suivre ...	

TAB. 21: BodyEphemC : méthodes publiques (suite)

signature	description
BodyEphemC (double *equatorialRadius*, double *oblateness*, double *rotationVelocity*, TypeFuncTsid **tsidFunc*, TypeFuncPos **sunFunc*, TypeFuncPos **moonFunc*, TypeFuncPos **earthFunc*)	construit une instance à partir des caractéristiques physiques du corps central : rayon équatorial *equatorialRadius* (en km), aplatissement *oblateness* et vitesse de rotation *rotationVelocity* (en rad/s) ; et des fonctions de calcul du temps sidéral **tsidFunc* (en rad), de calcul de la position du Soleil **sunFunc* (en km), de la Lune **moonFunc* (en km) et de la Terre **earthFunc*) (en km)
double **siderealTime** (double *t*, double *offset*)	calcule le temps sidéral en `rad` à la date *t* en `jour`, avec un écart de datation *offset* (généralement en `s`) entre l'échelle de temps utilisé pour la date *t* et l'échelle de temps utilisé par le modèle de calcul du temps sidéral
VecDBL **sunPosition** (double *t*)	calcule la position du Soleil par rapport au corps central en `km` à la date *t*
VecDBL **moonPosition** (double *t*)	calcule la position de la Lune par rapport au corps central en `km` à la date *t*
VecDBL **earthPosition** (double *t*)	calcule la position de la Terre par rapport au corps central en `km` à la date *t*

exemple d'utilisation

L'exemple suivant, directement extrait du code de la bibliothèque montre comment, dans la classe Etat, on initialise les données et fonctions données par l'utilisateur, dans le cas du langage C.

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

    if (ptrBodyEphem_ != 0)
    {
      delete ptrBodyEphem_;
    }
    ptrBodyEphem_ = new BodyEphemC(equatorialRadius, oblateness, rotationVelocity,
                                   tsidFunc, sunFunc, moonFunc, earthFunc);
```

conseils d'utilisation spécifiques

Les unités utilisées, les valeurs par défaut, un rappel sur la notion de date utilisée dans MARMOTTES et des indications sur la définition des fonctions par l'utilisateur sont donnés au niveau de la description de l'interface utilisateur (cf 11.16).

implantation

Les attributs privés sont décrits sommairement dans la table 22, il n'y a pas d'attribut protégé.

TAB. 22: attributs privés de la classe BodyEphemC

nom	type	description
tsidFuncPtr_	TypeFuncTsid *	pointeur sur la fonction de calcul du temps sidéral
sunFuncPtr_	TypeFuncPos *	pointeur sur la fonction de calcul de la position du Soleil
moonFuncPtr_	TypeFuncPos *	pointeur sur la fonction de calcul de la position de la Lune
earthFuncPtr_	TypeFuncPos *	pointeur sur la fonction de calcul de la position de la Terre

13.3 classe BodyEphemF

description

Cette classe dérivée de la classe BodyEphem implante l'interface d'accès, pour l'implémentation en FORTRAN par l'utilisateur, du calcul du temps sidéral et des éphémérides du Soleil, de la Lune et de la Terre, par rapport au corps central.
Elle permet aussi l'accès à des grandeurs physiques du corps central (rayon équatorial, aplatissement et vitesse de rotation).

interface publique

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

TAB. 23: BodyEphemF : méthodes publiques

signature	description
typedef double **TypeFuncTsid** (double *, double *)	défini la signature, pour le FORTRAN, de la fonction de calcul du temps sidéral
typedef void **TypeFuncPos** (double *, double [3])	défini la signature pour le FORTRAN des fonctions de calcul de position des corps (Soleil, Lune et Terre) par rapport au corps central
BodyEphemF ()	initialise une instance par défaut
BodyEphemF (const BodyEphemF& *b*)	constructeur par copie
const BodyEphemF& **operator** = (const BodyEphemF& *b*)	affectation
~BodyEphemF ()	destructeur
BodyEphem * **clone** () const	permet la duplication
	à suivre ...

TAB. 23: BodyEphemF : méthodes publiques (suite)

signature	description
BodyEphemF (double *equatorialRadius*, double *oblateness*, double *rotationVelocity*, TypeFuncTsid **tsidFunc*, TypeFuncPos **sunFunc*, TypeFuncPos **moonFunc*, TypeFuncPos **earthFunc*)	construit une instance à partir des caractéristiques physiques du corps central : rayon équatorial *equatorialRadius* (en km), aplatissement *oblateness* et vitesse de rotation *rotationVelocity* (en rad/s) ; et des fonctions de calcul du temps sidéral **tsidFunc* (en rad), de calcul de la position du Soleil **sunFunc* (en km), de la Lune **moonFunc* (en km) et de la Terre **earthFunc*) (en km)
double **siderealTime** (double *t*, double *offset*)	calcule le temps sidéral en `rad` à la date *t* en `jour`, avec un écart de datation *offset* (généralement en `s`) entre l'échelle de temps utilisé pour la date *t* et l'échelle de temps utilisé par le modèle de calcul du temps sidéral
VecDBL **sunPosition** (double *t*)	calcule la position du Soleil par rapport au corps central en `km` à la date *t*
VecDBL **moonPosition** (double *t*)	calcule la position de la Lune par rapport au corps central en `km` à la date *t*
VecDBL **earthPosition** (double *t*)	calcule la position de la Terre par rapport au corps central en `km` à la date *t*

exemple d'utilisation

L'exemple suivant, directement extrait du code de la bibliothèque montre comment, dans la classe Etat, on initialise les données et fonctions données par l'utilisateur, dans le cas du langage FORTRAN.

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

    if (ptrBodyEphem_ != 0)
    {
      delete ptrBodyEphem_;
    }
    ptrBodyEphem_ = new BodyEphemF(equatorialRadius, oblateness, rotationVelocity,
                                   tsidFunc, sunFunc, moonFunc, earthFunc);
```

conseils d'utilisation spécifiques

Les unités utilisées, les valeurs par défaut, un rappel sur la notion de date utilisée dans MARMOTTES et des indications sur la définition des fonctions par l'utilisateur sont donnés au niveau de la description de l'interface utilisateur (cf 11.16).

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 BodyEphemF

nom	type	description
tsidFuncPtr_	TypeFuncTsid *	pointeur sur la fonction de calcul du temps sidéral
sunFuncPtr_	TypeFuncPos *	pointeur sur la fonction de calcul de la position du Soleil
moonFuncPtr_	TypeFuncPos *	pointeur sur la fonction de calcul de la position de la Lune
earthFuncPtr_	TypeFuncPos *	pointeur sur la fonction de calcul de la position de la Terre

13.4 classe Etat

description

Cette classe mémorise l'état du satellite (date, position, vitesse, attitude) ainsi que quelques données qui lui sont directement liées comme les directions de la lune avec parallaxe, du soleil avec parallaxe et du soleil sans parallaxe.

interface publique

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

TAB. 25: Etat : méthodes publiques

signature	description
Etat ()	construit une instance par défaut. La configuration par défaut est un état utilisable (le satellite est au dessus de la terre, et les astres sont cohérents avec la date), mais son utilisation est déconseillée, il faut passer par la méthode **reinitialise** pour avoir une instance correcte. Ce constructeur est conçu pour la table des simulateurs des interfaces FORTRAN et C de la biliothèque, qui garantisse une réinitialisation correcte. Les utilisateurs C++ doivent plutôt passer par les autres constructeurs.
Etat (double *date*, const VecDBL& *position*, const VecDBL& *vitesse*, const RotDBL& *attitude*, const VecDBL& *spin*, double *coeffPosition*, double *coeffVitesse*) **throw (CantorErreurs)**	construit une instance à partir des éléments donnés en argument, les *coeffPosition* et *coeffVitesse* permettent de convertir les *position* et *vitesse* depuis les unités utilisateur vers les unités internes (kilomètres et kilomètres par secondes), ces coefficients sont appliqués aux arguments passés à la construction et également aux arguments passés par la méthode **reinitialise**.
	à suivre ...

TAB. 25: Etat : méthodes publiques (suite)

signature	description
Etat (double *date*, const VecDBL& *position*, const VecDBL& *vitesse*, const RotDBL& *attitude*, double *coeffPosition*, double *coeffVitesse*) **throw (CantorErreurs)**	construit une instance à partir des éléments donnés en argument (le spin est forcé à zéro dans ce constructeur), les *coeffPosition* et *coeffVitesse* permettent de convertir les *position* et *vitesse* depuis les unités utilisateur vers les unités internes (kilomètres et kilomètres par secondes), ces coefficients sont appliqués aux arguments passés à la construction et également aux arguments passés par la méthode **reinitialise**.
Etat (const Etat& *e*)	constructeur par copie
Etat& **operator** = (const Etat& *e*)	affectation
~Etat()	destructeur
void **desinitialise** ()	remet l'instance dans la configuration du constructeur par défaut
void **reinitialise** (double *date*, const VecDBL& *position*, const VecDBL& *vitesse*, const RotDBL& *attitude*) **throw (CantorErreurs)**	mémorise un nouvel état (sauf la partie spin, qui est recalculée par différences finies) et met à jour les données associées (temps sidéral, astres), les *coeffPosition* et *coeffVitesse* donnés à la construction (ou modifiés par **unitesPositionVitesse**) sont utilisés pour convertir les *position* et *vitesse* depuis les unités utilisateur vers les unités internes (kilomètres et kilomètres par secondes)
void **reinitialise** (const VecDBL& *spin*)	mémorise un nouveau spin
void **unitesPositionVitesse** (const string& *unitePos*, const string& *uniteVit*) **throw (MarmottesErreurs)**	prépare l'instance pour qu'elle considère des *position* et *vitesse* dans les unités utilisateur spécifiées (les unités reconnues sont : `km`, `m`, `km/s` et `m/s`)
double **date** () const	retourne la date
double **tempsSideral** () const	retourne le temps sidéral (entre 0 et 2π)
const VecDBL& **position** () const	retourne la position (dans les unités utilisateur)
const VecDBL& **vitesse** () const	retourne la vitesse (dans les unités utilisateur)
const RotDBL& **attitude** () const	retourne l'attitude
const RotVD1& **attitudeVD1** () const	retourne l'attitude (convertie sous forme d'un RotVD1)
const VecDBL& **spin** () const	retourne le spin
double **aplatissement** () const	retourne l'aplatissement du corps central
double **rayonEquatorial** () const	retourne le rayon équatorial du corps central (dans les unités utilisateur)
double **rayonCorpsCentral** () const	retourne le rayon *angulaire* du corps central
double **vitesseRotation** () const	retourne la vitesse angulaire de rotation du corps central
const VecDBL& **satLune** () const	retourne la direction de la lune (vecteur normé) et tenant compte de la parallaxe
double **distLune** () const	retourne la distance de la lune (dans les unités utilisateur)
double **rayonLune** () const	retourne le rayon angulaire de la lune

à suivre ...

TAB. 25: Etat : méthodes publiques (suite)

signature	description
const VecDBL& **satSoleil** () const	retourne la direction du soleil (vecteur normé) et tenant compte de la parallaxe
double **distSoleil** () const	retourne la distance du soleil (dans les unités utilisateur)
const VecDBL& **terreSoleil** () const	retourne la direction du soleil (vecteur normé) sans tenir compte de la parallaxe
double **coeffPosition** () const	retourne le coefficient de conversion des unités de position utilisateur en kilomètres
double **coeffVitesse** () const	retourne le coefficient de conversion des unités de vitesse utilisateur en kilomètres par seconde
void **normesLitigieuses** () const **throw (MarmottesErreurs)**	indique que les corrections de parallaxes ont été faites avec des vecteurs dont les normes sont litigieuses, puisque les valeurs numériques proviennent de la construction ou d'un appel à **reinitialise** alors que les coefficients de conversions ont été changés après coup
VecDBL **spinExtrapole** (double *date*, const RotDBL& *attitude*) const	retourne le spin qu'il faudrait avoir pour qu'à la *date* donnée on ait l'*attitude* donnée
VecVD1 **spinExtrapole** (double *date*, const RotVD1& *attitude*) const	retourne le spin qu'il faudrait avoir pour qu'à la *date* donnée on ait l'*attitude* donnée
RotDBL **attitudeExtrapolee** (double *date*) const	extrapole l'attitude courante à l'aide du spin courant jusqu'à la *date* fournie
void **enregistreCorps** (double *equatorialRadius*, double *oblateness*, double *rotationVelocity*, double *moonRadius*, double *sunRadius*, BodyEphemC : :TypeFuncTsid * *tsidFunc*, BodyEphemC : :TypeFuncPos * *sunFunc*, BodyEphemC : :TypeFuncPos * *moonFunc*, BodyEphemC : :TypeFuncPos * *earthFunc*,)	donne accès aux valeurs utilisateurs pour le rayon équatorial, l'aplatissement, et la vitesse de rotation du corps central, ainsi qu'aux fonctions utilisateurs, écrites en C, de calcul du temps sidéral et d'éphémérides par rapport au corps central. Les unités sont obligatoirement des `km` pour les distances et des `rad` pour les angles. Le temps sidéral doit être donné entre 0 et 2π.
void **enregistreCorps** (double *equatorialRadius*, double *oblateness*, double *rotationVelocity*, double *moonRadius*, double *sunRadius*, BodyEphemF : :TypeFuncTsid * *tsidFunc*, BodyEphemF : :TypeFuncPos * *sunFunc*, BodyEphemF : :TypeFuncPos * *moonFunc*, BodyEphemF : :TypeFuncPos * *earthFunc*,)	donne accès aux valeurs utilisateurs pour le rayon équatorial, l'aplatissement, et la vitesse de rotation du corps central, ainsi qu'aux fonctions utilisateurs, écrites en FORTRAN, de calcul du temps sidéral et d'éphémérides par rapport au corps central. Les unités sont obligatoirement des `km` pour les distances et des `rad` pour les angles. Le temps sidéral doit être donné entre 0 et 2π.

exemple d'utilisation

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

...

void SenseurOptique::initialiseCible (const Etat& etat)
   throw (MarmottesErreurs)
{ // initialisation de la direction de la cible en repère inertiel

  switch (code_)
  { case codeSoleil       :
      etat.normesLitigieuses ();
      cible_ = etat.satSoleil ();
      rapportDistCentral_ = etat.distSoleil () / etat.position ().norme ();
      rapportDistLune_    = etat.distSoleil () / etat.distLune ();
      break;

    case codeSoleilSansEclipse :
      etat.normesLitigieuses ();
      cible_ = etat.satSoleil ();
      rapportDistCentral_ = 0.0;
      rapportDistLune_    = 0.0;
      break;

    ...

    case codeVitesse :
      cible_ = etat.vitesse () / etat.vitesse ().norme ();
      rapportDistCentral_ = 0.0;
      rapportDistLune_    = 0.0;
      break;

    case codeMoment :
      cible_ = etat.position () ^ etat.vitesse ();
      cible_.normalise ();
      rapportDistCentral_ = 0.0;
      rapportDistLune_    = 0.0;
      break;

    case codeDevant :
      cible_ = (etat.position () ^ etat.vitesse ()) ^ etat.position ();
      cible_.normalise ();
      rapportDistCentral_ = 0.0;
      rapportDistLune_    = 0.0;
      break;
```

```
    ...

    default : // on ne peut passer ici qu'en cas d'incohérence dans le code
      cible_ = VecDBL (1, 0, 0);
      rapportDistCentral_ = 0.0;
      rapportDistLune_    = 0.0;
      throw MarmottesErreurs (MarmottesErreurs::cas_impossible,
                              __LINE__, __FILE__);
      break;
  }

  // on fait la conversion en développement limité une fois pour toutes
  cibleVD1_ = VecDBLVD1 (cible_);

}
```

conseils d'utilisation spécifiques

La classe Etat est utilisée par la classe Marmottes pour mémoriser les résultats des diverses résolutions d'attitude et pour tester des solutions différentes lors de la recherche numérique. Selon le modèle de résolution analytique considéré (géométrique ou cinématique) le paramètre de base est soit l'attitude (on déduit alors le spin par différences finies) soit le spin (on déduit alors l'attitude par extrapolation). Ces deux possibilités expliquent les différentes méthodes de réinitialisation et d'extrapolation fournies par la classe.

Il faut prendre garde au problème des unités. MARMOTTES *travaille en kilomètres et kilomètres par secondes en interne*, et la norme du vecteur position influe en particulier sur les corrections de parallaxe. Si l'appelant utilise des unités différentes, il doit le signaler à la bibliothèque. Ce problème ne s'étant posé qu'après plusieurs années, il n'est pas possible de configurer la classe dès la construction (changer la signature des constructeurs et des fonctions d'interface FORTRAN et C a semblé disproportionné face au problème). Il s'ensuit que l'on doit appeler **unitesPositionVitesse** *a posteriori* sur un état déjà calculé et avec des parallaxes corrigées. On devrait normalement réinitialiser l'état juste après une modification de ce type, et surtout ne pas utiliser les données de parallaxe avant cette réinitialisation. La méthode **normesLitigieuses** permet d'éviter ces erreurs, l'exemple d'utilisation ci-dessus montre dans quels cas la classe SenseurOptique le fait.

implantation

Les attributs privés sont décrits sommairement dans la table 26, il n'y a pas d'attribut protégé.

TAB. 26: attributs privés de la classe Etat

nom	type	description
date_	double	date en jours juliens CNES
tempsSideral_	double	temps sidéral entre 0 et 2π
position_	VecDBL	position du satellite (en km)
vitesse_	VecDBL	vitesse du satellite (en km/s)
attitude_	RotDBL	attitude du satellite
attitudeVD1_	RotVD1	conversion de attitude_ en RotVD1
spin_	VecDBL	spin du satellite
rayonCorpsCentral_	double	rayon angulaire du corps central
satLune_	VecDBL	direction de la lune corrigée de la parallaxe
distLune_	double	distance de la lune (en km)
rayonLune_	double	rayon angulaire de la lune
satSoleil_	VecDBL	direction du soleil corrigée de la parallaxe
distSoleil_	double	distance du soleil (en km)
terreSoleil_	VecDBL	direction du soleil sans correction de la parallaxe
rayonSoleil_	double	rayon angulaire du soleil
ptrBodyEphem_	BodyEphem *	pointeur sur un objet de type BodyEphem
coeffPosition_	double	coefficient de conversion des unités de position utilisateur en km
coeffVitesse_	double	coefficient de conversion des unités de vitesse utilisateur en km/s
normesLitigieuses_	int	indicateur de problèmes potentiels de parallaxe

Les méthodes privées sont décrites dans la table 27.

TAB. 27: Etat : méthodes privées

signature	description
void **miseAJourTempsSideral** (double *decalage* = 0.0)	mémorise le temps sidéral correspondant à la date courante pour un écart entre échelles de temps donné
void **miseAJourAstres** () **throw (CantorErreurs)**	met à jour les directions et rayons des astres avec et sans correction de la parallaxe pour la date courante

13.5 classe Famille

description

Cette classe ne sert que d'interface à la classe FamilleAbstraite. Le langage C++ ne permettant de gérer des instances dérivant d'une classe abstraite qu'à l'aide de pointeurs, cette classe encapsule de tels pointeurs et s'occupe de la gestion des copies et des désallocations mémoires de façon à présenter ces instances comme s'il s'agissait d'objets simples.

Elle est utilisée directement par la classe ModeleGeom qui résoud le modèle à un degré de liberté respectant deux consignes dans le cas de deux senseurs géométriques.

interface publique

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

TAB. 28: Famille : méthodes publiques

signature	description
Famille ()	initialise une instance par défaut inutilisable sans réaffectation (pointeur nul)
Famille (const FamilleAbstraite* *f*)	méthode qui permet de construire une Famille à partir d'un pointeur de FamilleAbstraite (copie profonde de l'objet pointé)
Famille (const Famille& *f*)	constructeur par copie
Famille& **operator** = (const Famille& *f*)	affectation
~**Famille** ()	destructeur, détruit le pointeur
RotVD1 **inertielCanonique** (const ValeurDerivee1& *t*) const	méthode qui retourne le quaternion de passage du repère inertiel au repère canonique de travail défini dans ModeleGeom

exemple d'utilisation

L'exemple suivant, directement extrait du code de la bibliothèque montre comment on a créé un vecteur (au sens de la STL [DR14]) de Famille.

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

{
  Intervalle i = Intervalle (-M_PI , M_PI);
  // création d'une Famille
  Famille f (new FamilleFixe (i, u1, u2,
                              (-canSat_) (VecDBLVD1 (v1a)),
```

```
                              (-canSat_) (VecDBLVD1 (v1a))));
  // table_ est de type vector
  table_.push_back (f);
  nombreFamilles_++;
}
```

conseils d'utilisation spécifiques

Cette classe n'a été implémentée que pour des raisons informatiques. En effet, elle permet de gérer élégamment la notion de pointeur de FamilleAbstraite (copie, destruction ...).

implantation

Les attributs privés sont décrits sommairement dans la table 29, il n'y a pas d'attribut protégé.

TAB. 29: attributs privés de la classe Famille

nom	type	description
ptrFamille_	FamilleAbstraite *	pointeur sur un objet de type FamilleAbstraite

13.6 classe FamilleAbstraite

description

Cette classe abstraite est l'interface d'accès aux différents types de familles de solutions aux modèles analytiques à un degré de liberté respectant deux consignes d'attitude dans le cas de senseurs géométriques.

Elle est dérivée en six classes différentes.

interface publique

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

TAB. 30: FamilleAbstraite : méthodes publiques

signature	description
FamilleAbstraite ()	initialise une instance par défaut inutilisable sans réaffectation
FamilleAbstraite (const Intervalle& *plages*)	construit une FamilleAbstraite à partir d'un intervalle plages
	à suivre ...

TAB. 30: FamilleAbstraite : méthodes publiques (suite)

signature	description
FamilleAbstraite (const FamilleAbstraite& f)	constructeur par copie
FamilleAbstraite& **operator** = (const FamilleAbstraite& f)	affectation
FamilleAbstraite * **copie** () const = 0	opérateur de copie virtuel
~**FamilleAbstraite** ()	destructeur
const Intervalle **plages** () const	retourne l'intervalle de validité de θ
RotVD1 **inertielCanonique** (const ValeurDerivee1& t) const = 0	méthode virtuelle pure retournant le quaternion de passage du repère inertiel au repère canonique de travail défini dans ModeleGeom

exemple d'utilisation

Il n'y a *aucune* utilisation directe de la classe FamilleAbstraite dans toute la bibliothèque MARMOTTES ! Les accès à la classe se font tous par l'intermédiaire de la classe Famille.

conseils d'utilisation spécifiques

Cette classe est abstraite, c'est à dire qu'aucune instance ne peut être créée directement. Tout pointeur sur un objet de ce type pointe en réalité sur un objet d'un des types dérivés : FamilleFixe, FamilleGenerale, FamilleAlignementMoins, FamilleAlignementPlus, FamillePrologementPi, FamillePrologementZero. Les constructeurs ne servent donc qu'à compléter les constructions d'objets plus gros et ne peuvent être appelés que par les constructeurs des classes dérivées.

À la création, le type de famille de solutions est analysé de sorte que la famille courante soit du bon type (fixe, générale, ...), mais après cette mise en place il n'y a plus lieu de différencier les types de familles. La fonction de résolution d'attitude de MARMOTTES passent donc par l'interface de la classe abstraite.

implantation

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

TAB. 31: attributs protégés de la classe FamilleAbstraite

nom	type	description
plages_	Intervalle	intervalle de validité de θ

Les méthodes protégées sont décrites dans la table 32.

TAB. 32: FamilleAbstraite : méthodes protégées

signature	description
ValeurDerivee1 **transforme** (const ValeurDerivee1& *t*) const	méthode qui transforme le paramètre libre t compris entre 0 et 1 en le paramètre libre θ compris entre les bornes du domaine de validité

13.7 classe FamilleAlignementMoins

description

Cette classe implante un cas particulier du modèle analytique d'attitude à un degré de liberté respectant deux consignes géométriques. C'est le cas où l'alignement $\vec{v}_1 = -\vec{a}_2$ est rencontré avec néanmoins une condition supplémentaire qui dit que $\mu_1 = \gamma = \frac{\pi}{2}$ n'est *pas* vérifié. μ_1 est le demi-angle d'ouverture du cône de consigne du premier senseur et γ est l'angle que forment les axes des deux cônes de consignes des deux premiers senseurs (tout deux géométriques). Les notations utilisées sont décrites en détail dans la documentation mathématique de MARMOTTES [DR1].

La Famille correspondante pour laquelle cette égalité sera vérifiée est implantée dans la classe FamilleProlongementPi.

interface publique

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

TAB. 33: FamilleAlignementMoins : méthodes publiques

signature	description
FamilleAlignementMoins ()	initialise une instance par défaut inutilisable sans réaffectation
FamilleAlignementMoins (const Intervalle *plages*, const VecVD1 *u1*, const VecVD1 *u2*, double *signe*, double *sinMu1*, double *cosMu1*, double *sinMu2*, double *cosMu2*)	construit une FamilleAlignementMoins à partir d'un intervalle plages, des vecteurs $\vec{u}_1$ et $\vec{u}_2$ exprimés dans le repère inertiel, d'un signe (± 1), et des constantes technologiques $\sin\mu_1$, $\cos\mu_1$, $\sin\mu_2$ et $\cos\mu_2$
FamilleAlignementMoins (const FamilleAlignementMoins& *f*)	constructeur par copie
FamilleAlignementMoins& **operator** = (const FamilleAlignementMoins& *f*)	affectation
FamilleAbstraite * **copie**() const	opérateur de copie
~**FamilleAlignementMoins** ()	destructeur

à suivre ...

TAB. 33: FamilleAlignementMoins : méthodes publiques (suite)

signature	description
RotVD1 **inertielCanonique** (const ValeurDerivee1& t) const	méthode virtuelle pure de la classe FamilleAbstraite, redéfinie ici et qui retourne le quaternion de passage du repère inertiel au repère canonique de travail défini dans ModeleGeom

exemple d'utilisation

L'exemple suivant, directement extrait du code de la bibliothèque montre comment, dans la classe ModeleGeom, on a créé un vecteur (au sens de la STL [DR14]) de FamilleAlignementMoins.

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

{
  if ((creneauInter.rencontre (Intervalle (M_PI-sinUneSec, M_PI+sinUneSec))) &&
       (fabs (sinPlus) <= sinUneSec))
  { // Theta = Pi est rencontré
    for (int j = 0 ; j < nombreIntervallesSecteur2 ; j++)
    {
      Famille f (new FamilleFixe (creneauConsigne2 [j], u1, u2,
                                  -a2Can, a1Can));
      table_.push_back (f);
      nombreFamilles_++;
    }
    Famille f (new FamilleAlignementMoins (creneauInter [i], u1, u2,
                                           -1.0, sinMu_1, cosMu_1,
                                           sinMu_2, cosMu_2));
    table_.push_back (f);
    nombreFamilles_++;
    f = new FamilleAlignementMoins (creneauInter [i], u1, u2,
                                    1.0, sinMu_1, cosMu_1,
                                    sinMu_2, cosMu_2);
    table_.push_back (f);
    nombreFamilles_++;
  }
}
```

conseils d'utilisation spécifiques

Ce mode de résolution d'attitude permet de prolonger le modèle général dans le cas particulier où l'alignement $\vec{v}_1 = -\vec{a}_2$ est rencontré et l'égalité $\mu_1 = \gamma = \frac{\pi}{2}$ n'est *pas* vérifié.

Comme montré dans l'exemple précédent, il faut créer deux familles de ce type ajoutées à une famille de type fixe qui correspond à $\vec{v}_1 = -\vec{a}_2$.

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 FamilleAlignementMoins

nom	type	description
u1_	VecVD1	premier vecteur cible en repère inertiel
u2_	VecVD1	second vecteur cible en repère inertiel
signe_	double	signe ± 1
sinMu_1_	double	sinus du demi-angle d'ouverture du premier cône de consigne
cosMu_1_	double	cosinus du demi-angle d'ouverture du premier cône de consigne
sinMu_2_	double	sinus du demi-angle d'ouverture du second cône de consigne
cosMu_2_	double	cosinus du demi-angle d'ouverture du second cône de consigne
deuxSinCos_	double	variable qui vaut $2 \times \sin\mu_1 \times \cos\mu_1$
deuxSinSin_	double	variable qui vaut $2 \times \sin\mu_1 \times \sin\mu_1$
coeff_	double	variable qui vaut $\sin\mu_1 \times \cos\mu_2$

13.8 classe FamilleAlignementPlus

description

Cette classe implante un cas particulier du modèle analytique d'attitude à un degré de liberté respectant deux consignes géométriques. C'est le cas où l'alignement $\vec{v}_1 = \vec{a}_2$ est rencontré avec néanmoins une condition supplémentaire qui dit que $\mu_1 = \gamma = \frac{\pi}{2}$ n'est *pas* vérifié. On rapelle que μ_1 est le demi-angle d'ouverture du cône de consigne du premier senseur et que γ est l'angle que forment les axes des deux cônes de consignes des deux premiers senseurs (tout deux géométriques). Les notations utilisées sont décrites en détail dans la documentation mathématique de MARMOTTES [DR1].

La Famille correspondante pour laquelle cette égalité sera vérifiée est implantée dans la classe FamilleProlongementZero.

interface publique

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

TAB. 35: FamilleAlignementPlus : méthodes publiques

signature	description
FamilleAlignementPlus()	initialise une instance par défaut inutilisable sans réaffectation
FamilleAlignementPlus (const Intervalle *plages*, const VecVD1 *u1*, const VecVD1 *u2*, double *signe*, double *sinMu1*, double *cosMu1*, double *sinMu2*, double *cosMu2*)	construit une FamilleAlignementPlus à partir d'un intervalle plages, des vecteurs $\vec{u}_1$ et $\vec{u}_2$ exprimés dans le repère inertiel, d'un signe (± 1), et des constantes technologiques $\sin\mu_1$, $\cos\mu_1$, $\sin\mu_2$ et $\cos\mu_2$
FamilleAlignementPlus (const FamilleAlignementPlus& *f*)	constructeur par copie
FamilleAlignementPlus& **operator** = (const FamilleAlignementPlus& *f*)	affectation
FamilleAbstraite * **copie**() const	opérateur de copie
~**FamilleAlignementPlus**()	destructeur
RotVD1 **inertielCanonique** (const ValeurDerivee1& *t*) const	méthode virtuelle pure de la classe FamilleAbstraite, redéfinie ici et qui retourne le quaternion de passage du repère inertiel au repère canonique de travail défini dans ModeleGeom

exemple d'utilisation

L'exemple suivant, directement extrait du code de la bibliothèque montre comment, dans la classe ModeleGeom, on a créé un vecteur (au sens de la STL [DR14]) de FamilleAlignementPlus.

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

{
  for (int i = 0 ; i < nombreIntervallesSecteur1 ; i++)
  {
    if ((creneauInter.rencontre (Intervalle (-sinUneSec, sinUneSec)))
         && (fabs (sinMoins) <= sinUneSec))
    {// Theta = 0 est rencontré
      for (int j = 0 ; j < nombreIntervallesSecteur2 ;j++)
      {
        Famille f (new FamilleFixe (creneauConsigne2 [j], u1, u2,
                                    a2Can, a1Can));
        table_.push_back (f);
        nombreFamilles_++;
      }
```

```
      Famille f (new FamilleAlignementPlus (creneauInter [i], u1, u2,
                                            -1.0, sinMu_1, cosMu_1,
                                            sinMu_2, cosMu_2));
      table_.push_back (f);
      nombreFamilles_++;
      f = new FamilleAlignementPlus (creneauInter [i], u1, u2,
                                     1.0, sinMu_1, cosMu_1,
                                     sinMu_2, cosMu_2);
      table_.push_back (f);
      nombreFamilles_++;
    }
  }
}
```

conseils d'utilisation spécifiques

Ce mode de résolution d'attitude permet de prolonger le modèle général dans le cas particulier où l'alignement $\vec{v}_1 = \vec{a}_2$ est rencontré et l'égalité $\mu_1 = \gamma = \frac{\pi}{2}$ n'est *pas* vérifié.

Comme montré dans l'exemple précédent, il faut créer deux familles de ce type ajoutées à une famille de type fixe qui correspond à $\vec{v}_1 = \vec{a}_2$.

implantation

Les attributs privés sont décrits sommairement dans la table 36, il n'y a pas d'attribut protégé.

TAB. 36: attributs privés de la classe FamilleAlignementPlus

nom	type	description
u1_	VecVD1	premier vecteur cible en repère inertiel
u2_	VecVD1	second vecteur cible en repère inertiel
signe_	double	signe ± 1
sinMu_1_	double	sinus du demi-angle d'ouverture du premier cône de consigne
cosMu_1_	double	cosinus du demi-angle d'ouverture du premier cône de consigne
sinMu_2_	double	sinus du demi-angle d'ouverture du second cône de consigne
cosMu_2_	double	cosinus du demi-angle d'ouverture du second cône de consigne

à suivre ...

TAB. 36: attributs privés de la classe FamilleAlignementPlus (suite)

nom	type	description
deuxSinCos_	double	variable qui vaut $2 \times \sin\mu_1 \times \cos\mu_1$
deuxSinSin_	double	variable qui vaut $2 \times \sin\mu_1 \times \sin\mu_1$
coeff_	double	variable qui vaut $\sin\mu_1 \times \cos\mu_2$

13.9 classe FamilleFixe

description

Cette classe implante les cas particuliers du modèle analytique à un degré de liberté respectant deux consignes géométriques pour lesquels les vecteurs $\vec{v}_1$ ou $\vec{v}_2$ sont fixés.

interface publique

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

TAB. 37: FamilleFixe : méthodes publiques

signature	description
FamilleFixe()	initialise une instance par défaut inutilisable sans réaffectation
FamilleFixe (const Intervalle *plages*, const VecVD1 *u1*, const VecVD1 *u2*, const VecVD1 *v1*, const VecVD1 *ref*, const VecVD1 *axe*)	construit une FamilleFixe à partir d'un intervalle plages, des vecteurs $\vec{u}_1$ et $\vec{u}_2$ exprimés dans le repère inertiel, du vecteur $\vec{v}_1$ exprimé dans le repère canonique (défini dans la ModeleGeom) et des référence et axe qui définissent le paramètre libre sur le secteur de consigne considéré
FamilleFixe(const FamilleFixe& *f*)	constructeur par copie
FamilleFixe& **operator** =(const FamilleFixe& *f*)	affectation
FamilleAbstraite * **copie**() const	opérateur de copie
~**FamilleFixe**()	destructeur
RotVD1 **inertielCanonique** (const ValeurDerivee1& *t*) const	méthode virtuelle pure de la classe FamilleAbstraite, redéfinie ici et qui retourne le quaternion de passage du repère inertiel au repère canonique de travail défini dans ModeleGeom

exemple d'utilisation

Les exemples vus pour les classes FamilleAlignementMoins, FamilleAlignementPlus, directement extraits du code de la bibliothèque montrent comment, dans la classe ModeleGeom, on a créé un vecteur (au sens de la STL [DR14]) de FamilleFixe.

conseils d'utilisation spécifiques

Par rapport aux anciennes versions de MARMOTTES, cette famille est limitée par les champs de vue et par les domaines de validité du paramètre de consigne. Ceci apporte un gain dans le temps de calcul.

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 FamilleFixe

nom	type	description
axe_	VecVD1	coordonnées en repère canonique de l'axe du cône de consigne
r_	RotVD1	rotation constante qui permet d'amener le vecteur fixe à sa place dans le repère canonique

13.10 classe FamilleGenerale

description

Cette classe implante le cas général du modèle analytique d'attitude à un degré de liberté respectant deux consignes géométriques. On entend par cas général le cas où aucun alignement ne se produit, ni avec les axes des cônes de consignes, ni avec les points des cônes, ni avec les vecteurs cibles.

interface publique

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

TAB. 39: FamilleGenerale : méthodes publiques

signature	description
FamilleGenerale()	initialise une instance par défaut inutilisable sans réaffectation
FamilleGenerale (const Intervalle& *plages*, const VecVD1& *u1*, const VecVD1& *u2*, double *signe*, double *sinMu1*, double *cosMu1*, double *sinMu2*, double *cosMu2*, double *sinGamma*, double *cosGamma*, double *cosAlpha_1_2*)	construit une FamilleGenerale à partir d'un intervalle plages, des vecteurs $\vec{u}_1$ et $\vec{u}_2$ exprimés dans le repère inertiel, d'un signe (± 1), et des constantes technologiques $\sin\mu_1$, $\cos\mu_1$, $\sin\mu_2$, $\cos\mu_2$, $\sin\gamma$, $\cos\gamma$ et $\cos\alpha_{1,2}$
FamilleGenerale(const FamilleGenerale& *f*)	constructeur par copie
FamilleGenerale& **operator** =(const FamilleGenerale& *f*)	affectation
FamilleAbstraite * **copie**() const	opérateur de copie
~**FamilleGenerale**()	destructeur
RotVD1 **inertielCanonique** (const ValeurDerivee1& *t*) const	méthode virtuelle pure de la classe FamilleAbstraite, redéfinie ici et qui retourne le quaternion de passage du repère inertiel au repère canonique de travail défini dans ModeleGeom

exemple d'utilisation

L'exemple suivant, directement extrait du code de la bibliothèque montre comment, dans la classe ModeleGeom, on a créé un vecteur (au sens de la STL [DR14]) de FamilleGenerale.

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

{
  // Ni Theta = 0, ni Theta = Pi ne sont rencontrés
  Famille f (new FamilleGenerale (creneauInter [i], u1, u2, -1.0,
                                  sinMu_1, cosMu_1, sinMu_2, cosMu_2,
                                  sinGamma, cosGamma, cosAlpha_1_2));
  table_.push_back (f);
  nombreFamilles_++;
  f = new FamilleGenerale (creneauInter [i], u1, u2, 1.0,
                           sinMu_1, cosMu_1, sinMu_2, cosMu_2,
                           sinGamma, cosGamma, cosAlpha_1_2);
  table_.push_back (f);
  nombreFamilles_++;
}
```

conseils d'utilisation spécifiques

Une importante amélioration (et correction de bug !) a été apportée à la version 7.0 par rapport aux versions précédentes de MARMOTTES.

La prise en compte des singularités ne se fait plus numériquement mais analytiquement. C'est pourquoi une nouvelle architecture a été implantée et a mis au point ce système de *familles*.

implantation

Les attributs privés sont décrits sommairement dans la table 40, il n'y a pas d'attribut protégé.

TAB. 40: attributs privés de la classe FamilleGenerale

nom	type	description
u1_	VecVD1	premier vecteur cible en repère inertiel
u2_	VecVD1	second vecteur cible en repère inertiel
signe_	double	signe ± 1
sinMu_1_	double	sinus du demi-angle d'ouverture du premier cône de consigne
sinMu_2_	double	sinus du demi-angle d'ouverture du second cône de consigne
cosMu_2_	double	cosinus du demi-angle d'ouverture du second cône de consigne
sinGammaCosMu1_	double	variable qui vaut $\sin\gamma \times \cos\mu_1$
cosGammaSinMu1_	double	variable qui vaut $\cos\gamma \times \sin\mu_1$
sinGammaSinMu1_	double	variable qui vaut $\sin\gamma \times \sin\mu_1$
cosGammaCosMu1_	double	variable qui vaut $\cos\gamma \times \cos\mu_1$
partieConstante_	double	variable qui vaut $\cos\alpha_{1,2} - \cos\mu_2 \times \cos\gamma \times \cos\mu_1$
cosMu2SinGammaSinMu1_	double	variable qui vaut $\cos\mu_2 \times \sin\gamma \times \sin\mu_1$

13.11 classe FamillePrologementPi

description

Cette classe implante un cas particulier du modèle analytique d'attitude à un degré de liberté respectant deux consignes géométriques. C'est le cas où l'alignement $\vec{v}_1 = -\vec{a}_2$ est rencontré avec néanmoins une condition supplémentaire qui dit que $\mu_1 = \gamma = \frac{\pi}{2}$.

On rapelle que μ_1 est le demi-angle d'ouverture du cône de consigne du premier senseur et que γ est l'angle que forment les axes des deux cônes de consignes des deux premiers senseurs (tout deux géométriques). Les notations utilisées sont décrites en détail dans la documentation mathématique de MARMOTTES [DR1].

De plus, la Famille correspondante pour laquelle cette égalité n'est pas vérifiée est implantée dans la classe FamilleAlignementMoins.

interface publique

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

TAB. 41: FamilleProlongementPi : méthodes publiques

signature	description
FamilleProlongementPi()	initialise une instance par défaut inutilisable sans réaffectation
FamilleProlongementPi (const Intervalle& *plages*, const VecVD1& *u1*, const VecVD1& *u2*, double *signe*, double *sinMu2*, double *cosMu2*)	construit une FamilleProlongementPi à partir d'un intervalle plages, des vecteurs $\vec{u}_1$ et $\vec{u}_2$ exprimés dans le repère inertiel, d'un signe (± 1), et des constantes technologiques $\sin\mu_2$ et $\cos\mu_2$
FamilleProlongementPi (const FamilleProlongementPi& *f*)	constructeur par copie
FamilleProlongementPi& **operator** = (const FamilleProlongementPi& *f*)	affectation
FamilleAbstraite * **copie**() const	opérateur de copie
~**FamilleProlongementPi**()	destructeur
RotVD1 **inertielCanonique** (const ValeurDerivee1& *t*) const	méthode virtuelle pure de la classe FamilleAbstraite, redéfinie ici et qui retourne le quaternion de passage du repère inertiel au repère canonique de travail défini dans ModeleGeom

exemple d'utilisation

L'exemple suivant, directement extrait du code de la bibliothèque montre comment, dans la classe ModeleGeom, on a créé un vecteur (au sens de la STL [DR14]) de FamilleProlongementPi.

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

{
  if (creneauInter.rencontre (Intervalle (M_PI-sinUneSec, M_PI+sinUneSec)))
  {
    // Theta = Pi est rencontré
    for (int j = 0 ; j < nombreIntervallesSecteur2 ; j++)
    {
      Famille f (new FamilleFixe (creneauConsigne2 [j], u1, u2,
                                  -a22Can, a1Can));
      table_.push_back (f);
      nombreFamilles_++;
```

```
    }
    Famille f (new FamilleProlongementPi (creneauInter [i], u1, u2,
                                          1.0, sinMu_2, cosMu_2));
    table_.push_back (f);
    nombreFamilles_++;
    f = new FamilleProlongementPi (creneauInter [i], u1, u2,
                                   -1.0, sinMu_2, cosMu_2);
    table_.push_back (f);
    nombreFamilles_++;
  }
}
```

conseils d'utilisation spécifiques

Ce mode de résolution d'attitude permet de prolonger le modèle général dans le cas particulier où l'alignement $\vec{v}_1 = -\vec{a}_2$ est rencontré et on a l'égalité $\mu_1 = \gamma = \frac{\pi}{2}$.

Comme montré dans l'exemple précédent, il faut créer deux familles de ce type ajoutées à une famille de type fixe qui correspond à $\vec{v}_1 = -\vec{a}_2$.

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 FamilleProlongementPi

nom	type	description
u1_	VecVD1	premier vecteur cible en repère inertiel
u2_	VecVD1	second vecteur cible en repère inertiel
signe_	double	signe ± 1
sinMu_2_	double	sinus du demi-angle d'ouverture du second cône de consigne
cosMu_2_	double	cosinus du demi-angle d'ouverture du second cône de consigne

13.12 classe FamilleProlongementZero

description

Cette classe implante un cas particulier du modèle analytique d'attitude à un degré de liberté respectant deux consignes géométriques. C'est le cas où l'alignement $\vec{v}_1 = \vec{a}_2$ est rencontré avec néanmoins une condition supplémentaire qui dit que $\mu_1 = \gamma = \frac{\pi}{2}$.

On rapelle que μ_1 est le demi-angle d'ouverture du cône de consigne du premier senseur et que γ est l'angle que forment les axes des deux cônes de consignes des deux premiers senseurs (tout deux géométriques). Les notations utilisées sont décrites en détail dans la documentation mathématique de MARMOTTES [DR1].

De plus, la Famille correspondante pour laquelle cette égalité n'est pas vérifiée est implantée dans la classe FamilleAlignementPlus.

interface publique

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

TAB. 43: FamilleProlongementZero : méthodes publiques

signature	description
FamilleProlongementZero ()	initialise une instance par défaut inutilisable sans réaffectation
FamilleProlongementZero (const Intervalle& *plages*, const VecVD1& *u1*, const VecVD1& *u2*, double *signe*, double *sinMu2*, double *cosMu2*)	construit une FamilleProlongementZero à partir d'un intervalle plages, des vecteurs $\vec{u}_1$ et $\vec{u}_2$ exprimés dans le repère inertiel, d'un signe (± 1), et des constantes technologiques $\sin\mu_2$ et $\cos\mu_2$
FamilleProlongementZero (const FamilleProlongementZero& *f*)	constructeur par copie
FamilleProlongementZero& **operator** = (const FamilleProlongementZero& *f*)	affectation
FamilleAbstraite * **copie**() const	opérateur de copie
~FamilleProlongementZero ()	destructeur
RotVD1 **inertielCanonique** (const ValeurDerivee1& *t*) const	méthode virtuelle pure de la classe FamilleAbstraite, redéfinie ici et qui retourne le quaternion de passage du repère inertiel au repère canonique de travail défini dans ModeleGeom

exemple d'utilisation

L'exemple suivant, directement extrait du code de la bibliothèque montre comment, dans la classe ModeleGeom, on a créé un vecteur (au sens de la STL [DR14]) de FamilleProlongementZero.

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

{
  for (int i = 0 ; i < nombreIntervallesSecteur1 ; i++)
  {
    if (creneauInter.rencontre (Intervalle (-sinUneSec, sinUneSec)))
    {
      // Theta = 0 est rencontré
```

```
      for (int j = 0 ; j < nombreIntervallesSecteur2 ; j++)
      {
        Famille f (new FamilleFixe (creneauConsigne2 [j], u1, u2,
                                    a2Can, a1Can));
        table_.push_back (f);
        nombreFamilles_++;
      }
      Famille f (new FamilleProlongementZero (creneauInter [i], u1, u2,
                                              1.0, sinMu_2, cosMu_2));
      table_.push_back (f);
      nombreFamilles_++;
      f = new FamilleProlongementZero (creneauInter [i], u1, u2,
                                       -1.0, sinMu_2, cosMu_2);
      table_.push_back (f);
      nombreFamilles_++;
    }
  }
}
```

conseils d'utilisation spécifiques

Ce mode de résolution d'attitude permet de prolonger le modèle général dans le cas particulier où l'alignement $\vec{v}_1 = -\vec{a}_2$ est rencontré et on a l'égalité $\mu_1 = \gamma = \frac{\pi}{2}$.

Comme montré dans l'exemple précédent, il faut créer deux familles de ce type ajoutées à une famille de type fixe qui correspond à $\vec{v}_1 = \vec{a}_2$.

implantation

Les attributs privés sont décrits sommairement dans la table 44, il n'y a pas d'attribut protégé.

TAB. 44: attributs privés de la classe FamilleProlongement-Zero

nom	type	description
u1_	VecVD1	premier vecteur cible en repère inertiel
u2_	VecVD1	second vecteur cible en repère inertiel
signe_	double	signe ± 1
sinMu_2_	double	sinus du demi-angle d'ouverture du second cône de consigne
cosMu_2_	double	cosinus du demi-angle d'ouverture du second cône de consigne

13.13 classe Marmottes

description

Cette classe est la classe de plus haut niveau de la bibliothèque MARMOTTES. Dans la plupart des cas, les utilisateurs n'utiliseront que cette classe pour leurs applicatifs, soit directement s'ils programment en C++, soit à travers les interfaces fonctionnelles s'ils programment en C ou en FORTRAN (ces interfaces fonctionnelles n'encapsulent en effet que les appels à des instances de la classe Marmottes, en gérant ces instances dans un tableau de sorte que les interfaces fonctionnelles ne voient qu'un index).

Les explications données ici sont relativement succinctes, en effet toutes les méthodes nécessaires aux utilisateurs sont décrites en parallèles avec leurs interfaces C et FORTRAN dans la section 11, page 62.

interface publique

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

TAB. 45: Marmottes : méthodes publiques

signature	description
Marmottes ()	construit une instance de simulateur par défaut, non initialisée
Marmottes (double *date*, const VecDBL& *position*, const VecDBL& *vitesse*, const RotDBL& *attitude*, const VecDBL& *spin*, const string& *fichier*, const string& *senseur1*, const string& *senseur2*, const string& *senseur3*)	construit une instance de simulateur à partir d'un état initial
Marmottes (const Marmottes& *m*) **throw (ClubErreurs, MarmottesErreurs)**	constructeur par copie
Marmottes& **operator** = (const Marmottes& *m*) **throw (ClubErreurs, MarmottesErreurs)**	affectation
void **desinitialise** ()	désinitialise l'instance, cette méthode est utilisée par l'interface fonctionnelle pour gérer le tableau des instances pour le C et le FORTRAN, elle n'est pas d'une grande utilité pour le C++ qui a accès aux constructeurs, destructeur, et opérateur d'affectation.
à suivre ...	

TAB. 45: Marmottes : méthodes publiques (suite)

signature	description
void **reinitialise** (double *date*, const VecDBL& *position*, const VecDBL& *vitesse*, const RotDBL& *attitude*, const VecDBL& *spin*, const string& *fichier*, const string& *senseur1*, const string& *senseur2*, const string& *senseur3*) **throw (ClubErreurs, MarmottesErreurs)**	réinitialise une instance comme si elle venait d'être créée avec les arguments fournis
void **senseurs** (const string& *fichier*, const string& *senseur1*, const string& *senseur2*, const string& *senseur3*) **throw (ClubErreurs, MarmottesErreurs)**	change les senseurs de contrôle à utiliser pour la suite des résolutions
void **nouveauRepere** (const string& *fichier*, const string& *senseur*, const RotDBL& *nouveau*) **throw (ClubErreurs, MarmottesErreurs)**	change l'orientation du *senseur* par rapport à ce qui a été lu dans le fichier de configuration
void **calage** (const string& *fichier*, const string& *senseur*, double *c*) **throw (ClubErreurs, MarmottesErreurs)**	positionne le *senseur* par rapport à son repère de base selon l'angle de *calage* (il faut qu'un axe de calage ait été défini pour ce senseur dans son fichier de configuration)
void **modifieCible** (const string& *fichier*, const string& *senseur*, const VecDBL& *cible*) **throw (ClubErreurs, MarmottesErreurs)**	mémorise la *cible* du *senseur* dérivé d'un senseur optique spécifié
void **initialiseGyro** (const string& *fichier*, const string& *senseur*, double *date*, double *angle*) **throw (ClubErreurs, MarmottesErreurs)**	réinitialise le gyromètre intégrateur *senseur* de sorte qu'il donne la mesure *angle* à la *date* spécifiée
void **modifieReference** (const string& *fichier*, const string& *senseur*, const RotDBL& *reference*) **throw (ClubErreurs, MarmottesErreurs)**	mémorise le repère *reference* pour le senseur de Cardan *senseur*
void **deuxConsignes** (double *date*, const VecDBL& *position*, const VecDBL& *vitesse*, double *m1*, double *m2*, RotDBL **attitude*, VecDBL **spin*) **throw (CantorErreurs, MarmottesErreurs)**	résoud une attitude de façon partielle pour qu'elle ne respecte que les consignes des deux premiers senseurs de contrôle ; cette méthode est réservée aux utilisateurs expérimentés
à suivre ...	

TAB. 45: Marmottes : méthodes publiques (suite)

signature	description
void **attitude** (double *date*, const VecDBL& *position*, const VecDBL& *vitesse*, double *m1*, double *m2*, double *m3*, RotDBL **attit*, VecDBL **spin*) **throw (CantorErreurs, MarmottesErreurs)**	résoud l'attitude qui respecte les consignes fournies pour cette date et met à jour l'instance ; cette méthode est la plus importante de la bibliothèque
void **imposeAttitude** (double *date*, const VecDBL& *position*, const VecDBL& *vitesse*, const RotDBL& *attit*) **throw (MarmottesErreurs)**	force l'attitude à la valeur spécifiée par *attit* ; le spin est déduit par différences finies
void **imposeSpin** (double *date*, const VecDBL& *position*, const VecDBL& *vitesse*, const VecDBL& *spin*) **throw (MarmottesErreurs)**	force le spin à la valeur spécifiée par *spin* ; l'attitude est déduite par intégration
void **repereBase** (const string& *fichier*, const string& *senseur*, RotDBL **r*) **throw (ClubErreurs, MarmottesErreurs)**	retourne le repère de base du *senseur*, indépendamment de toute réorientation
void **repere** (const string& *fichier*, const string& *senseur*, RotDBL **r*) **throw (ClubErreurs, MarmottesErreurs)**	retourne le repère courant du *senseur*, en tenant compte des réorientations éventuelles
void **mesure** (const string& *fichier*, const string& *senseur*, double **m*) **throw (ClubErreurs, MarmottesErreurs)**	retourne dans la variable pointée par *m* la mesure produite par le *senseur* dans l'attitude courante *sans prendre en compte les critères de contrôlabilité* ; ceci signifie qu'un senseur ayant un champ de vue de 15 ° peut tres bien fournir une mesure de 164 ° et pas de code d'erreur, les critères de contrôlabilité sont disponibles par une fonction séparée (**controlable**)
void **controlable** (const string& *fichier*, const string& *senseur*, int **c*) **throw (ClubErreurs, MarmottesErreurs)**	retourne dans la variable pointée par *c* un indicateur de contrôlabilité de l'attitude courante par le *senseur* (c'est à dire est ce que l'astre cible est dans le champ de vue, n'est il pas masqué par la terre, le senseur est-il inhibé, ...)
void **unitesPositionVitesse** (const string& *unitePos*, const string& *uniteVit*) **throw (MarmottesErreurs)**	cette méthode permet de modifier les unités de position et de vitesse dans les interfaces externes de la bibliothèque (qui en interne travaille toujours en kilomètres et kilomètres par seconde)
void **respecterConsignes** (const string& *fichier*, const string& *senseur*) **throw (ClubErreurs, MarmottesErreurs)**	Cette méthode permet de signaler à la bibliothèque que les consignes fournies par l'appelant sont déjà dans les unités internes et qu'il ne faut pas y toucher

à suivre ...

TAB. 45: Marmottes : méthodes publiques (suite)

signature	description
void **convertirConsignes** (const string& *fichier*, const string& *senseur*) **throw (ClubErreurs, MarmottesErreurs)**	Cette méthode permet de signaler à la bibliothèque que les consignes fournies par l'appelant sont dans des unités externes et qu'il faut leur appliquer une conversion degrés vers radians pour pouvoir les utiliser (ceci fonctionne aussi avec des vitesses angulaires)
void **respecterMesures** (const string& *fichier*, const string& *senseur*) **throw (ClubErreurs, MarmottesErreurs)**	Cette méthode permet de signaler à la bibliothèque que l'appelant attend les mesures dans les unités internes et qu'il ne faut pas y toucher
void **convertirMesures** (const string& *fichier*, const string& *senseur*) **throw (ClubErreurs, MarmottesErreurs)**	Cette méthode permet de signaler à la bibliothèque que l'appelant attend les mesures dans des unités externes et qu'il faut leur appliquer une conversion radians vers degrés pour qu'il puisse les utiliser (ceci fonctionne aussi avec des vitesses angulaires)
const string& **nomFichier**() const	retourne le nom du fichier de base des senseurs (ce fichier peut en inclure d'autres)
bool **estInitialise** () const	indique si l'instance est initialisée, cette méthode est utilisée par l'interface fonctionnelle pour gérer le tableau des instances pour le C et le FORTRAN, elle n'est pas d'une grande utilité pour le C++ qui a accès aux constructeurs, destructeur, et opérateur d'affectation
void **wMax** (double *omega*) **throw (MarmottesErreurs)**	cette méthode permet de modifier la vitesse de rotation maximale du modèle analytique des senseurs cinématiques
void **convergence** (double *seuil*) **throw (MarmottesErreurs)**	cette méthode permet de modifier le critère de convergence de l'algorithme de résolution numérique
void **dichotomie** (int *tranches*) **throw (MarmottesErreurs)**	cette méthode permet de modifier le nombre de tranches de l'algorithme de séparation des zéros dans la résolution numérique
void **autoriseExtrapolation** () **throw (MarmottesErreurs)**	cette méthode autorise la bibliothèque à accélérer ses résolutions en tentant une simple extrapolation de l'attitude à partir des états précédents et de ne lancer une résolution complète qu'en cas d'échec (il s'agit du comportement par défaut, aussi cette méthode n'est utile que pour annuler l'effet d'un appel préalable à **interditExtrapolation**)
void **interditExtrapolation** () **throw (MarmottesErreurs)**	cette méthode oblige la bibliothèque à refaire une résolution d'attitude complète et l'empêche de se contenter d'une simple extrapolation à partir des pas précédents

à suivre ...

TAB. 45: Marmottes : méthodes publiques (suite)

signature	description
void **enregistreCorps** (double *equatorialRadius*, double *oblateness*, double *rotationVelocity*, double *moonRadius*, double *sunRadius*, BodyEphemC : :TypeFuncTsid * *tsidFunc*, BodyEphemC : :TypeFuncPos * *sunFunc*, BodyEphemC : :TypeFuncPos * *moonFunc*, BodyEphemC : :TypeFuncPos * *earthFunc*)	cette méthode permet l'accès aux valeurs utilisateurs pour le rayon équatorial, l'aplatissement, et la vitesse de rotation du corps central, ainsi qu'aux fonctions utilisateurs, écrites en C, de calcul du temps sidéral et d'éphémérides par rapport au corps central.
void **enregistreCorps** (double *equatorialRadius*, double *oblateness*, double *rotationVelocity*, double *moonRadius*, double *sunRadius*, BodyEphemF : :TypeFuncTsid * *tsidFunc*, BodyEphemF : :TypeFuncPos * *sunFunc*, BodyEphemF : :TypeFuncPos * *moonFunc*, BodyEphemF : :TypeFuncPos * *earthFunc*)	cette méthode permet l'accès aux valeurs utilisateurs pour le rayon équatorial, l'aplatissement, et la vitesse de rotation du corps central, ainsi qu'aux fonctions utilisateurs, écrites en FORTRAN, de calcul du temps sidéral et d'éphémérides par rapport au corps central.
Senseur * **accesSenseur** (const string& *fichier*, const string& *senseur*) **throw (ClubErreurs, CantorErreurs, MarmottesErreurs)**	cette méthode permet de récupérer le pointeur sur un senseur à partir des arguments.
const Etat & **etat** ()	cette méthode retourne l'état de l'instance Marmottes.
void **lireParametres** (double* *ptrDate*, VecDBL* *ptrPosition*, VecDBL* *ptrVitesse*, RotDBL* *ptrAttitude*, VecDBL* *ptrSpin*) **throw (MarmottesErreurs)**	cette méthode permet de récupérer les valeurs courantes des paramètres internes à MARMOTTES : la date, la position, la vitesse, l'attitude et le spin.

exemple d'utilisation

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

// simulation d'un pointage en yaw-steering
// l'axe Z est pointé terre, et l'orientation autour de la
// direction terre est telle que le soleil soit dans le
// plan XZ (les panneaux solailres tournent autour de l'axe Y)
Marmottes simulateur (dateIni, positionIni, vitesseIni,
                      attitude, spin,
                      fichier,
                      "roulis-terre",
```

```
                      "tangage-terre",
                      "lacet-soleil");

// on tient compte d'un biais de pilotage
simulateur.attitude (date, position, vitesse,
                     biaisRoulis, biaisTangage, 0.0,
                     &attitude, &spin);

// recherche de la cible dans le repère satellite
// (on à mis des pseudo senseurs spécifiques)
double x, y, z;
simulateur.mesure (simulateur.nomFichier (), "x-cible", &x);
simulateur.mesure (simulateur.nomFichier (), "y-cible", &y);
simulateur.mesure (simulateur.nomFichier (), "z-cible", &z);
VecDBL cible (x, y, z);

int    visible;
simulateur.controlable (simulateur.nomFichier (), "instrument",
                        &visible);
if (visible)
{ // la cible est dans le champ de l'instrument
  ...
}
```

conseils d'utilisation spécifiques

Une bonne utilisation de la bibliothèque MARMOTTES passe par une modification assez profonde des habitudes de raisonnement et de modélisation. Il est important de raisonner en termes de *senseurs* plutôt qu'en termes d'angles, de vecteurs ou de plans.

On se rend ainsi compte avec l'expérience que l'on n'utilise quasiment jamais l'attitude ou le spin en sortie des méthodes de résolution, on se contente de faire confiance au simulateur qui les mémorise et on lui demande des informations de plus haut niveau par des pseudo-senseurs (l'exemple illustre cette démarche).

Les pseudo-senseurs ne sont pas utilisés uniquement pour faire faire du post-traitement à la bibliothèque en plus des résolutions, ils sont également utilisés pour modéliser certaines attitude où ils interviennent en tant que senseurs de contrôle. On peut citer les pseudo-senseurs de Cardan bien adaptés au pointage terre, mais également les pseudo-senseurs d'ascension droite et de déclinaison qui permettent d'interfacer l'attitude avec l'optimisation des manœuvres inertielles (on modélise la direction de poussée par deux senseurs), et tous les senseurs optiques dont les cibles sont des directions *a priori* non mesurables à bord d'un satellite (direction du moment orbital, direction de la vitesse, direction du soleil pendant l'éclipse, ...).

Il arrive (trop souvent pour les utilisateurs) que la résolution d'attitude s'achève sur un message laconique du type : pas de solution aux consignes d'attitude. Ce message indique un échec de la résolution, qui est généralement lié à une impossibilité physique de contrôle par les senseurs et les consignes fournis par l'utilisateur (utilisation d'un senseur solaire pendant l'éclipse, inhibition du senseur terre par la lune ou le soleil, incompatibilité entre les

positions relatives terre, satellite, soleil, et les consignes ou les champs de vue sur certaines portions de l'orbite, ...). Il faut prendre garde à ces limitations[15], et éventuellement utiliser pour le contrôle des pseudo-senseurs fournissant des mesures compatibles avec les vrais senseurs montés sur le satellite, mais moins limités : champs de vue couvrant toute la sphère unité, possibilité de pointer le soleil même à travers la terre, ...

implantation

Les attributs privés sont décrits sommairement dans la table 46, il n'y a pas d'attribut protégé.

TAB. 46: attributs privés de la classe Marmottes

nom	type	description
initialise_	bool	indicateur d'instance initialisée
extrapolationOk_	bool	indicateur d'extrapolation d'attitude autorisée pour accélérer les résolutions
etat_	Etat	dernier état calculé
solveur_	ResolveurAttitude	moteur de résolution d'attitude
fichier_	FichierStructure	fichier de base des senseurs (ce fichier peut en inclure d'autres)
senseurs_	Adressage<Senseur *>	table des senseurs déjà utilisés

Les méthodes privées sont décrites dans la table 47.

TAB. 47: Marmottes : méthodes privées

signature	description
void **initialiseSenseurs** (const string& *nomFichier*) **throw (ClubErreurs, MarmottesErreurs)**	initialise une table de senseurs vide et mémorise le fichier *nomFichier*
Senseur* **recupParNom** (const string& *nom*) **throw (ClubErreurs, MarmottesErreurs)**	récupère un senseur par son *nom*, soit dans la table interne, soit en le lisant dans le fichier (et en l'insérant dans la table)
void **valideNouvelEtat** (double *date*, const VecDBL& *position*, const VecDBL& *vitesse*, const RotDBL& *attitude*, const VecDBL& *spin*)	propage l'état résolu défini par les *position*, *vitesse*, *attitude* et *spin* aux senseurs de la table (pour permettre par exemple aux gyromètres intégrateurs de se mettre à jour)

[15] qui sont des limitations de la loi de pilotage, pas des limitations de la bibliothèque

13.14 classe MarmottesErreurs

description

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

interface publique

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

Les opérations publiques sont essentiellement 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 :

id_marmottes_non_initialise : `int` (pour l'identificateur) ;

bloc_non_terminal : `char *` (pour le nom du bloc) ;

rotation_bloc : `char *` (pour le nom du bloc) ;

nombre_champs_bloc : `char *` (pour le nom du bloc, `int` (pour le nombre de champs trouvés) , `int` (pour le nombre de champs demandés) ;

bloc_introuvable : `char *` (pour le message issu de la bibliothèque CLUB) ;

vecteur_nul : `char *` (pour le nom du bloc) ;

quaternion_nul : `char *` (pour le nom du bloc) ;

liste_non_initialisee : néant ;

gyros_coaxiaux : néant ;

consignes_gyro_elevees : `char *` (pour le nom du premier senseur), `char *` (pour le nom du second senseur), `double` (pour la valeur de la vitesse de rotation maximale) ;

consigne_degeneree : `char *` (pour le nom du senseur) ;

consignes_incompatibles : `char *` (pour le nom du premier senseur), `char *` (pour le nom du second senseur) ;

type_inconnu : `char *` (pour le type trouvé), `char *` (pour chacun des types connus), `(char *) 0` (pour indiquer la fin de la liste) ;

cible_inconnue : `char *` (pour la cible trouvée), `char *` (pour chacune des cibles connues), `(char *) 0` (pour indiquer la fin de la liste) ;

champ_inhibition_cible_soleil :`char *` (pour le nom du senseur) ;

champ_inhibition_cible_lune :`char *` (pour le nom du senseur) ;

champ_inhibition_cible_central :`char *` (pour le nom du senseur) ;

omega_neg : `double` (pour la vitesse fournie) ;

seuil_neg : `double` (pour le seuil fourni) ;

tranches_neg : `int` (pour le nombre fourni) ;

pas_de_solution : néant ;

controlabilite : `char *` (pour le nom du senseur) ;

calage_interdit : néant ;

types_incompatibles : `char *` (pour le nom du premier senseur), `char *` (pour le nom du second senseur) ;

consigne_interdite : `char *` (pour le nom du premier senseur), `char *` (pour le nom du second senseur) ;

genre_cardan_inconnu : `char *` (pour le nom du senseur), `char *` (pour chacun des genres connus), `(char *) 0` (pour indiquer la fin de la liste) ;

reference_cardan_inconnue : `char *` (pour le nom du senseur), `char *` (pour chacun des repères connus), `(char *) 0` (pour indiquer la fin de la liste) ;

rotation_cardan : `char *` (pour le nom du senseur) ;

points_masque : `int` (pour le nombre de points trouvé), `char *` (pour le nom du bloc) ;

unite_position : `char *` (pour l'unité fournie), `char *` (pour chacune des unités connues), `(char *) 0` (pour indiquer la fin de la liste) ;

unite_vitesse : `char *` (pour l'unité fournie), `char *` (pour chacune des unités connues), `(char *) 0` (pour indiquer la fin de la liste) ;

normes_litigieuses : néant ;

cible_utilisateur : `char *` (pour le nom du senseur) ;

modifie_cible : `char *` (pour le nom du senseur) ;

senseur_sans_cible : `char *` (pour le nom du senseur) ;

pas_gyro_integrateur : `char *` (pour le nom du senseur) ;

pas_senseur_cinematique : `char *` (pour le nom du senseur) ;

reference_utilisateur : `char *` (pour le nom du senseur) ;

modifie_reference : `char *` (pour le nom du senseur) ;

senseur_sans_reference : `char *` (pour le nom du senseur) ;

erreur_non_reconnue : néant ;

senseur_mesure_pure : `char *` (pour le nom du senseur) ;

allocation_memoire : néant ;

points_echantillon : `int` (pour le nombre de points), `char *` (pour le nom du bloc) ;

echantillon_vide : `char *` (pour le nom du bloc) ;

echantillon_rejete : `double` (pour la première coordonnée du point), `double` (pour la seconde coordonnée du point), `char *` (pour le nom du bloc) ;

objet_inconnu : néant ;

cas_impossible : `int` (pour le numéro de ligne), `char *` (pour le nom du fichier source).

exemples d'utilisation

```
#include "marmottes/MarmottesErreurs.h"
#include "marmottes/Lecture.h"

try
{
  Senseur *s = LireSenseur  (fichier, "ires-roll");
}

catch (MarmottesErreurs me)
{
  return me.code ();
}
```

conseils d'utilisation spécifiques

Cette classe est principalement utilisée pour tester la bonne exécution des fonctions de la bibliothèque MARMOTTES 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 MARMOTTES 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.

13.15 classe Modele

description

Cette classe abstraite est l'interface d'accès aux modèles analytiques à un degré de liberté respectant deux consignes d'attitude. Elle est utilisée directement par la résolution numérique qui propose des valeurs tests pour le degré de liberté et attend l'attitude correspondante, ce qui lui permet de trouver la valeur test respectant également la troisième consigne.

Cette classe est destinée à être dérivée en deux classes, une implantant le modèle correspondant à deux consignes géométriques, l'autre implantant le modèle correspondant à deux consignes cinématiques.

interface publique

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

TAB. 48: Modele : méthodes publiques

signature	description
Modele ()	initialise une instance par défaut inutilisable sans réaffectation
Modele (const Modele& *m*)	constructeur par copie
Modele& **operator** = (const Modele& *m*)	affectation
~**Modele** ()	destructeur virtuel, ne fait rien dans la classe de base
const Senseur* **senseur1** () const	retourne un pointeur sur le premier senseur concerné par le modèle
const Senseur* **senseur2** () const	retourne un pointeur sur le second senseur concerné par le modèle
void **miseAJourSenseurs** (Senseur* *s1*, Senseur* *s2*)	change les senseurs concernés par le modèle
void **prendConsignesEnCompte**() **throw (MarmottesErreurs)** = 0	méthode virtuelle pure d'initialisation du modèle à partir des consignes courantes des senseurs concernés, cette méthode est spécifique au type de modèle (géométrique ou cinématique et est donc implantée uniquement dans les classe dérivées)
int **familles** () const = 0	méthode virtuelle pure retournant le nombre de famillles d'attitude disjointes
void **attitude** (const Etat& *etatPrecedent*, double *date*, const ValeurDerivee1& *t*, int *famille*, RotVD1* *ptrAttitude*, VecVD1* *ptrSpin*) const = 0	méthode virtuelle pure utilisée par la résolution numérique et retournant une attitude et un spin à la *date* courante dans les variables pointées par *ptrAttitude* et *ptrSpin* pour une valeur test du degré de liberté *t* (compris entre −1 et +1) de la *famille* en cours d'analyse, connaissant l'*etatPrecedent*

exemple d'utilisation

L'exemple suivant, directement extrait du code de la bibliothèque montre la fonction numérique annulée par la résolution de la troisième consigne. La méthode **modele** de la classe ResolveurAttitude retourne un pointeur sur un Modele qui correspond au modèle courant (géométrique ou cinématique).

```
static ValeurDerivee1 fonc (double t, void* donnee)
{ // récuperation de l'objet de résolution
  ResolveurAttitude* ptr = (ResolveurAttitude *) donnee;

  // calcul de l'attitude modélisée respectant les premières consignes
```

```
  RotVD1 attitude;
  VecVD1 spin;
  ptr->modele ()->attitude (ptr->etatPrecedent (), ptr->date (),
                            ValeurDerivee1 (t, 1.0), ptr->famille (),
                            &attitude, &spin);

  // calcul de l'écart par rapport à la troisième consigne
  return ptr->sB ()->foncEcart (ptr->etatPrecedent (), ptr->date (),
                                attitude, spin);

}
```

conseils d'utilisation spécifiques

Cette classe est abstraite, c'est à dire qu'aucune instance ne peut être créée directement. Tout pointeur sur un objet de ce type pointe en réalité sur un objet d'un des types dérivés : ModeleGeom ou ModeleCine. Les constructeurs ne servent donc qu'à compléter les constructions d'objets plus gros et ne peuvent être appelés que par les constructeurs des classes dérivées.

À la création et à chaque changement de senseur de consigne, le triplet de senseurs est analysé de sorte que le modèle courant soit du bon type (géométrique ou cinématique), mais après cette mise en place il n'y a plus lieu de différencier les deux types de modèles. Toutes les fonctions de résolution de MARMOTTES passent donc par l'interface de la classe abstraite.

implantation

Les attributs privés sont décrits sommairement dans la table 49, il n'y a pas d'attribut protégé.

TAB. 49: attributs privés de la classe Modele

nom	type	description
senseur1_	Senseur*	pointeur sur le premier senseur du modèle
senseur2_	Senseur*	pointeur sur le second senseur du modèle

13.16 classe ModeleCine

description

Cette classe dérivée de la classe Modele implante le modèle analytique d'attitude à un degré de liberté respectant deux consignes cinématiques.

Les consignes cinématiques étant des projections du vecteur de rotation instantané (le spin) sur des axes sensibles, repecter deux consignes revient à dire que le spin est sur une droite (par exemple une droite parallèle

à l'axe $\vec{k}$ si les consignes portent sur $\omega_{\vec{i}}$ et $\omega_{\vec{j}}$). Le degré de liberté est donc une position sur cette droite, et le modèle est donc un modèle de spin. L'attitude se déduit du spin modélisé par intégration à partir de l'état précedent en supposant que le spin est resté constant entre l'état précedent et la date courante.

interface publique

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

TAB. 50: ModeleCine : méthodes publiques

signature	description
ModeleCine ()	construit une instance par défaut inutilisable sans réaffectation
ModeleCine (const ModeleCine& *m*)	constructeur par copie
ModeleCine& **operator** = (const ModeleCine& *m*)	affectation
~**ModeleCine** ()	destructeur virtuel, ne fait rien dans cette classe
void **miseAJourOmegaMax** (double *omega*) **throw (MarmottesErreurs)**	mémorise une nouvelle vitesse de rotation instantané maximale $\omega_{\max}$ qui traduit l'étendue du modèle analytique
void **prendConsignesEnCompte**() **throw (MarmottesErreurs)**	méthode d'initialisation du modèle à partir des consignes courantes des senseurs concernés
int **familles** () const	retourne le nombre de famillles d'attitude disjointes, qui vaut toujours 1 pour les modèles cinématiques
void **attitude** (const Etat& *etatPrecedent*, double *date*, const ValeurDerivee1& *t*, int *famille*, RotVD1* *ptrAtt*, VecVD1* *ptrSpin*) const	retourne l'attitude intégrée et le spin modélisé à la *date* courante dans les variables pointées par *ptrAtti* et *ptrSpin* pour une valeur test du degré de liberté *t* (compris entre -1 et $+1$) de la *famille* en cours d'analyse, l'intégration de l'attitude selon le spin se faisant à partir de l'*etatPrecedent*

exemple d'utilisation

La définition en ligne suivante, extraite du fichier de déclaration de la classe ResolveurAttitude est la *seule* utilisation directe de la classe ModeleCine dans la bibliothèque MARMOTTES si l'on excepte les manipulations de pointeurs pour rendre courant soit le modèle géométrique soit le modèle cinématique. Tous les autres accès à la classe se font par l'intermédiaire de la classe de base Modele.

```
void  miseAJourOmegaMax    (double omega)
  throw (MarmottesErreurs)
               { modeleCine_.miseAJourOmegaMax (omega); }
```

conseils d'utilisation spécifiques

L'arborescence d'héritage des classes de modèles est un moyen de masquer aux algorithmes de résolution numérique le type de modèle sous-jacent. La seule chose qui soit utile à ce niveau est la possibilité d'initialiser le modèle en début de résolution, et la possibilité d'appliquer le modèle à une valeur test pour obtenir une attitude correspondante.

Ce principe simplifie énormément les résolutions et doit être conservé ; il est donc fortement déconseillé de faire évoluer la classe ModeleCine indépendamment des classes Modele et ModeleGeom. La présence d'une méthode **miseAJourOmegaMax** peut déjà être considérée comme un défaut qu'il faudrait supprimer à terme.

implantation

Les attributs privés sont décrits sommairement dans la table 51, il n'y a pas d'attribut protégé.

TAB. 51: attributs privés de la classe ModeleCine

nom	type	description
u_	VecVD1	partie du spin indépendante du degré de liberté
v_	VecVD1	direction de la partie variable du spin : $\vec{\Omega} = \vec{u} + \theta\vec{v}$
thetaMax_	double	amplitude maximale de θ
omegaMax_	double	vitesse de rotation maximale

13.17 classe ModeleGeom

description

Cette classe dérivée de la classe Modele implante le modèle analytique d'attitude à un degré de liberté respectant deux consignes géométriques.

Les consignes géométriques étant des cônes (ou des secteurs sur des cônes), le degré de liberté est une position angulaire sur le premier des cônes, et le modèle est un modèle d'attitude. Le spin se déduit de l'attitude par différences finies par rapport à l'état précédent.

interface publique

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

TAB. 52: ModeleGeom : méthodes publiques

signature	description
ModeleGeom ()	construit une instance par défaut inutilisable sans réaffectation
ModeleGeom (const ModeleGeom& *m*) ModeleGeom& **operator** = (const ModeleGeom& *m*)	constructeur par copie affectation
~**ModeleGeom** ()	destructeur virtuel, ne fait rien dans cette classe
void **prendConsignesEnCompte**() **throw (MarmottesErreurs)**	méthode d'initialisation du modèle à partir des consignes courantes des senseurs concernés
int **familles** () const	retourne le nombre de familles d'attitude disjointes
void **attitude** (const Etat& *etatPrecedent*, double *date*, const ValeurDerivee1& *t*, int *famille*, RotVD1* *ptrAtt*, VecVD1* *ptrSpin*) const	retourne l'attitude modélisée et le spin dérivé à la *date* courante dans les variables pointées pa *ptrAtt* et *ptrSpin* pour une valeur test du degré de liberté *t* (compris entre -1 et $+1$) de la *famille* en cours d'analyse, la dérivation du spin se faisant à partir de l' *etatPrecedent*

exemple d'utilisation

Il n'y a *aucune* utilisation directe de la classe ModeleGeom dans toute la bibliothèque MARMOTTES ! Si l'on excepte les manipulations de pointeurs pour rendre courant soit le modèle géométrique soit le modèle cinématique, les accès à la classe se font tous par l'intermédiaire de la classe de base Modele.

conseils d'utilisation spécifiques

L'arborescence d'héritage des classes de modèles est un moyen de masquer aux algorithmes de résolution numérique le type de modèle sous-jacent. La seule chose qui soit utile à ce niveau est la possibilité d'initialiser le modèle en début de résolution, et la possibilité d'appliquer le modèle à une valeur test pour obtenir une attitude correspondante.

Ce principe simplifie énormément les résolutions et doit être conservé ; il est donc fortement déconseillé de faire évoluer la classe ModeleGeom indépendamment des classes Modele et ModeleCine.

implantation

Les attributs privés sont décrits sommairement dans la table 53, il n'y a pas d'attribut protégé.

TAB. 53: attributs privés de la classe ModeleGeom

nom	type	description
nombreFamilles_	int	nombre total de familles d'attitude pour les consignes courantes
table_	vector<Famille>	élément de type *vector* (STL) qui contient l'ensemble des solutions possibles, celles-ci étant de type Famille
canSat_	RotVD1	rotation permettant de convertir un vecteur exprimé dans le repère canonique de résolution en vecteur exprimé dans le repère satellite

Les méthodes privées sont décrites dans la table 54.

TAB. 54: ModeleGeom : méthodes privées

signature	description
Creneau **intersection** (double *cosMin*, double *cosMax*, const Creneau& *creneauConsigne*) const	retourne le créneau d'intersection entre le domaine de validité de θ et le créneau de consigne du premier senseur

13.18 classe Parcelle

description

Cette classe abstraite est l'interface de haut niveau avec les champs de vue possédant une logique booléenne. Ces champs de vue sont utilisés par les senseurs terre pour modéliser des notions telles que : la terre est considérée comme visible si une partie du limbe est dans le scan Nord *et* qu'une autre partie est dans le scan Sud.

interface publique

Le fichier `cantor/Field.h` qui est inclus par par `marmottes/Parcelle.h` définit deux types fonctions qui peuvent être passées à certaines méthodes de la classe :

```
typedef void TypeFuncConstField (const Field& f, void* data);
typedef void TypeFuncField      (      Field& f, void* data);

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

TAB. 55: Parcelle : méthodes publiques

signature	description
Parcelle ()	constructeur par défaut (la classe étant abstraite ne peut être instanciée, ce constructeur est destiné à être appelé par les constructeurs des classes dérivées)
Parcelle* **copie** () const = 0	opérateur de copie virtuel
~**Parcelle** ()	destructeur virtuel
bool **inclus** (const VecDBL& *u*) const = 0	indique si le vecteur $\vec{u}$ est inclus dans la parcelle
double **ecartFrontiere** (const VecDBL& *u*) const = 0	calcule l'écart angulaire signé entre le vecteur $\vec{u}$ et la frontière (positif si le point est à l'intérieur de la parcelle, négatif sinon)
bool **visible** (const Cone& *c*) **throw (CantorErreurs)** const = 0	indique si le cône *c* est visible au moins partiellement
Secteurs **visible** (const Secteurs& *s*) **throw (CantorErreurs)** const = 0	filtre la partie visible du secteur *s*
void **appliqueRotation** (const RotDBL& *r*) = 0	transforme l'instance en lui appliquant la rotation *r*
void **integreRotation** (const VecDBL& *axe*, double *angle*) **throw (CantorErreurs, MarmottesErreurs)** = 0	transforme l'instance en la *trainant* selon la rotation définie par l'*axe* et l'*angle*
void **appliqueMarge** (double *m*) **throw (CantorErreurs, MarmottesErreurs)** = 0	transforme l'instance en lui appliquant la marge angulaire *m*
void **applique** (TypeFoncConstChamp **f*, void **d*) const = 0	applique la fonction *f* (qui ne modifie pas ses arguments) à tous les champs qui composent la parcelle, l'argument anonyme *d* est passé à *f* à chaque appel
void **applique** (TypeFoncChamp **f*, void **d*) = 0	applique la fonction *f* (qui peut modifier ses arguments) à tous les champs qui composent la parcelle, l'argument anonyme *d* est passé à *f* à chaque appel
void **rechercheChamp** (Champ ***adressePtr*) = 0	recherche le champ suivant celui dont l'adresse est donnée en argument, et le retourne dans *adressePtr* (cette fonction est utilisée en interne par les classes dérivées, elle n'est publique qu'en raison de limitations d'accès complexes propres au C++)
void **initParcours** (double *tolerance* = 1.0e-4)	initialise les fonctions de parcours de sorte que l'erreur maximale respecte la *tolerance*
bool **pointSuivant** (VecDBL **u*, bool **dernier*)	retourne le point suivant du parcours dans le vecteur $\vec{u}$, et indique dans la booléen pointé par *dernier* si ce point est le dernier d'un tronçon intermédiaire (c'est à dire s'il faudra lever le crayon dans un tracé après ce point). Retourne un booléen faux lorsque le parcours est terminé
void **arreteParcours** ()	arrête le parcours courant

exemple d'utilisation

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

for (int i = 0; i < nbParcelles; i++)
{ // boucle sur toutes les parcelles à tracer
  bool dernier;
  VecDBL point;

  // tracé en repère senseur
  tableParcelles [i]->initParcours (tolerance);
  ptrTraceur->modePointille  ();
  ptrTraceur->commenceCourbe ();
  while (tableParcelles [i]->pointSuivant (&point, &dernier))
  { ptrTraceur->ajoutePoint (satSens (point), -1.0e-4);
    if (dernier)
      ptrTraceur->termineCourbe ();
  }
  ptrTraceur->termineCourbe  ();
  ptrTraceur->modeContinu    ();
  tableParcelles [i]->arreteParcours ();

  // nettoyage
  delete tableParcelles [i];
  tableParcelles [i] = 0;

}
```

conseils d'utilisation spécifiques

Cette classe est abstraite, c'est à dire qu'aucune instance ne peut être créée directement. Tout pointeur sur un objet de ce type pointe en réalité sur un objet d'un des types dérivés : ParcelleElementaire, ReunionEtParcelle ou ReunionOuParcelle. Les constructeurs ne servent donc qu'à compléter les constructions d'objets plus gros et ne peuvent être appelés que par les constructeurs des classes dérivées.

Seules les fonctions de lecture des senseurs ont besoin de connaître les types de base, pour construire les parcelles petit à petit. Une fois ces parcelles construites toutes les autres fonctions doivent passer par la classe de base Parcelle.

implantation

Les attributs privés sont décrits sommairement dans la table 56, il n'y a pas d'attribut protégé.

TAB. 56: attributs privés de la classe Parcelle

nom	type	description
courant_	Champ *	pointeur sur le champ courant lors d'un parcours
tolerance_	double	tolérance sur le parcours

Les méthodes privées sont décrites dans la table 57.

TAB. 57: Parcelle : méthodes privées

signature	description
bool **champSuivant** ()	prépare le parcours du champ suivant
Parcelle (const Parcelle& p)	constructeur par copie
Parcelle& **operator** = (const Parcelle& p)	affectation

13.19 classe ParcelleElementaire

description

Cette classe implante les parcelles les plus simples qui ne sont composées que d'un champ.

interface publique

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

TAB. 58: ParcelleElementaire : méthodes publiques

signature	description
ParcelleElementaire (const Champ& c)	construit une parcelle à partir du champ c
Parcelle* **copie**() const	opérateur de copie virtuel
~**ParcelleElementaire**()	destructeur virtuel
bool **inclus** (const VecDBL& u) const	indique si le vecteur $\vec{u}$ est inclus dans la parcelle
double **ecartFrontiere** (const VecDBL& u) const	calcule l'écart angulaire signé entre le vecteur $\vec{u}$ et la frontière (positif si le point est à l'intérieur de la parcelle, négatif sinon)
bool **visible** (const Cone& c) const **throw (CantorErreurs)**	indique si le cône c est visible au moins partiellement
Secteurs **visible** (const Secteurs& s) const **throw (CantorErreurs)**	filtre la partie visible du secteur s
	à suivre ...

TAB. 58: ParcelleElementaire : méthodes publiques (suite)

signature	description
void **appliqueRotation** (const RotDBL& *r*)	transforme l'instance en lui appliquant la rotation *r*
void **integreRotation** (const VecDBL& *axe*, double *angle*) **throw (CantorErreurs, MarmottesErreurs)**	transforme l'instance en la *trainant* selon la rotation définie par l'*axe* et l'*angle*
void **appliqueMarge** (double *m*) **throw (CantorErreurs, MarmottesErreurs)**	transforme l'instance en lui appliquant la marge angulaire *m*
void **applique** (TypeFoncConstChamp *f, void *d) const	applique la fonction *f* (qui ne modifie pas ses arguments) à tous les champs qui composent la parcelle, l'argument anonyme *d* est passé à *f* à chaque appel
void **applique** (TypeFoncChamp *f, void *d)	applique la fonction *f* (qui peut modifier ses arguments) à tous les champs qui composent la parcelle, l'argument anonyme *d* est passé à *f* à chaque appel
void **rechercheChamp** (Champ **adressePtr*)	recherche le champ suivant celui dont l'adresse est donnée en argument, et le retourne dans *adressePtr* (cette fonction est utilisée en interne par les classes dérivées, elle n'est publique qu'en raison de limitations d'accès complexes propres au C++)

exemple d'utilisation

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

Parcelle *LireParcelle (FichierStructure *blocPere, const string& nom)
  throw (MarmottesErreurs)
{
  Parcelle *p1 = 0;
  Parcelle *p2 = 0;
  try
  {
    // extraction d'une parcelle depuis un sous-bloc nommé d'un bloc père.

  ...
        { // ce doit être une parcelle élémentaire: un champ
          Field c;
          LireChamp (blocPere, nom, &c);
          return (Parcelle *) new ParcelleElementaire (c);
        }
  ...

  }
  ...

}
```

conseils d'utilisation spécifiques

Seules les fonctions de lecture des senseurs ont besoin de connaître les types de base, pour construire les parcelles petit à petit. Une fois ces parcelles construites toutes les autres fonctions doivent passer par la classe de base Parcelle.

implantation

Les attributs privés sont décrits sommairement dans la table 59, il n'y a pas d'attribut protégé.

TAB. 59: attributs privés de la classe ParcelleElementaire

nom	type	description
c_	Champ	champ sous-jacent à la parcelle

Les méthodes privées sont décrites dans la table 60.

TAB. 60: ParcelleElementaire : méthodes privées

signature	description
ParcelleElementaire ()	constructeur par défaut. Il est défini explicitement uniquement pour prévenir celui créé automatiquement par le compilateur et ne doit pas être utilisé.
ParcelleElementaire (const ParcelleElementaire& *p*)	constructeur par copie
ParcelleElementaire& **operator** = (const ParcelleElementaire& *p*)	affectation

13.20 classe ResolveurAttitude

description

Cette classe implante la phase de résolution numérique d'une attitude à partir d'un modèle n'ayant plus qu'un degré de liberté.

interface publique

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

TAB. 61: ResolveurAttitude : méthodes publiques

signature	description
ResolveurAttitude ()	constructeur par défaut
ResolveurAttitude (const ResolveurAttitude& *r*)	constructeur par copie
ResolveurAttitude& **operator** = (const ResolveurAttitude& *r*)	affectation
~ResolveurAttitude ()	destructeur, libère la mémoire allouée pour la table des solutions
Senseur* **s1** () const	retourne un pointeur sur le premier senseur de contrôle
Senseur* **s2** () const	retourne un pointeur sur le deuxième senseur de contrôle
Senseur* **s3** () const	retourne un pointeur sur le troisième senseur de contrôle
void **reinitialise** (Senseur* *s1*, Senseur* *s2*, Senseur* *s3*) **throw (MarmottesErreurs)**	modifie les senseurs de contrôle (ceci affecte le seuil de convergence)
void **modeliseConsignes** (const Etat& *etatPrecedent*, const Etat& *etatResolution*, double *m1*, double *m2*) **throw (CantorErreurs, MarmottesErreurs)**	met en place le modèle analytique pour une résolution d'attitude partielle à deux senseurs uniquement
void **modeliseConsignes** (const Etat& *etatPrecedent*, const Etat& *etatResolution*, double *m1*, double *m2*, double *m3*) **throw (CantorErreurs, MarmottesErreurs)**	met en place le modèle analytique pour une résolution d'attitude complète
void **miseAJourOmegaMax** (double *omega*) **throw (MarmottesErreurs)**	modifie la vitesse de rotation instantanée maximale du satellite utilisée dans la modélisation des attitudes contrôlées par deux senseurs cinématiques
void **miseAJourConvergence** (double *seuil*) **throw (MarmottesErreurs)**	modifie le seuil de convergence de la résolution numérique
void **miseAJourDichotomie** (int *tranches*) **throw (MarmottesErreurs)**	modifie le nombre de tranches de dichotomie pour la séparation des extrema locaux
void **deuxConsignes** (SpinAtt* *ptrSpinAtt*) **throw (CantorErreurs, MarmottesErreurs)**	résolution partielle d'attitude sur uniquement deux senseurs
void **trouveTout** () **throw (MarmottesErreurs)**	résolution complète d'attitude (on utilise trois senseurs, et on cherche toutes les solutions)
void **elimineExcedentaires** () **throw (CantorErreurs, MarmottesErreurs)**	élimine les artefacts de résolution mathématiques issus d'un appel préalable à **trouveTout**
int **nombreSolutions** () const	retourne le nombre total de solutions mémorisées pour la résolution courante

à suivre ...

TAB. 61: ResolveurAttitude : méthodes publiques (suite)

signature	description
const SpinAtt& **selection** () const	sélectionne la *meilleure* solution issue d'un appel préalable à **trouveTout** (s'il reste plusieurs solutions, le critère de sélection est la modification de spin nécessaire pour aboutir à l'attitude testée depuis l'attitude précédente)

exemple d'utilisation

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

void Marmottes::attitude (double date,
                          const VecDBL& position, const VecDBL& vitesse,
                          double m1, double m2, double m3,
                          RotDBL *attitude, VecDBL *spin)
  throw (CantorErreurs, MarmottesErreurs)
{ // calcul d'une attitude donnée par trois consignes

  try
  {
  ...

    // mise à jour de la modélisation des mesures dans les senseurs
    solveur_.modeliseConsignes (etatResolu_, etatExtrapole_,
                                m1, m2, m3);
    ...

    // résolution du modèle pour respecter la troisième consigne
    solveur_.trouveTout ();

    ...

    // filtrage des solutions engendrées par la modélisation mathématique
    solveur_.elimineExcedentaires ();

    ...

    // récupération de la "meilleure" solution
    SpinAtt sol = solveur_.selection ();
    *attitude   = sol.attitude ();
    *spin       = sol.spin     ();

    // mise à jour de l'état de référence et de l'état extrapolé
```

```
    etatResolu_.reinitialise (date, position, vitesse, *attitude);
    etatResolu_.reinitialise (*spin);
    etatExtrapole_ = etatResolu_;

  }

  catch (...)
  {
    etatExtrapole_ = etatResolu_;
    throw;
  }
}
```

conseils d'utilisation spécifiques

La classe de résolution numérique n'a de raison d'être que pour une résolution donnée ; elle ne mémorise pas les états précédent et courant par exemple. Elle est conçue comme un moyen de sérialiser les étapes nécessaires, en mémorisant les résultats intermédiaires (le modèle analytique, les solutions mathématiques, les solutions physiques, *la* solution finale). Il est nécessaire d'appeler ses différentes méthodes dans le bon ordre. Cette classe n'étant absolument pas réutilisable et n'étant appelée que par la classe de haut niveau Marmottes, aucun mécanisme de protection particulier n'a été mis en place pour garantir que le bon ordre soit respecté. Cet ordre est le suivant : mettre en place le modèle analytique, chercher l'ensemble des solutions, éliminer les artefacts de modélisation, sélectionner une solution physique respectant les contraintes technologiques.

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 ResolveurAttitude

nom	type	description
senseursConsigne_	Senseur* [3]	table des pointeurs sur les senseurs de contrôle (ces senseurs ne sont pas alloués dans la classe, ils doivent exister par ailleurs)
sA1_	Senseur*	pointeur vers le premier senseur du modèle analytique courant (c'est l'un des pointeurs de la table senseursConsigne_)
sA2_	Senseur*	pointeur vers le second senseur du modèle analytique courant (c'est l'un des pointeurs de la table senseursConsigne_)
sB_	Senseur*	pointeur vers le senseur utilisé pour la résolution numérique (c'est l'un des pointeurs de la table senseursConsigne_)

à suivre ...

TAB. 62: attributs privés de la classe ResolveurAttitude (suite)

nom	type	description
modeleCourant_	Modele*	pointeur sur le modèle courant (il s'agit de l'un des attributs modeleGeom_ ou modeleCine_)
modeleGeom_	ModeleGeom	modèle analytique pour les senseurs géométriques
modeleCine_	ModeleCine	modèle analytique pour les senseurs cinématiques
etatPrecedent_	Etat	état précédent
etatResolution_	Etat	état courant de la résolution (ni l'attitude ni le spin ne peuvent y être significatifs bien sûr)
famille_	int	nombre de familles du modèle analytique courant
tailleTable_	int	taille allouée totale de la table des solutions
nbSol_	int	nombre de solutions mémorisées
solutions_	SpinAtt *	table des solutions
seuil_	double	seuil de convergence
tranches_	int	nombre de tranches de dichotomie de la séparation des extrema

Les méthodes privées sont décrites dans la table 63.

TAB. 63: ResolveurAttitude : méthodes privées

signature	description
void **ajouteSolution** (double *t*)	ajoute une solution à la table pour la valeur *t* de la variable libre du modèle analytique
ValeurDerivee1 **f** (double *t*)	fonction à annuler pour résoudre l'attitude
static ValeurDerivee1 **f** (double *t*, void * *arg*)	fonction à annuler pour résoudre l'attitude, *arg* étant un pointeur vers une instance

13.21 classe ReunionEtParcelles

description

Cette classe implante les parcelles devant voir leur cible dans deux sous-parcelles simultanément (notion de *et* logique).

interface publique

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

TAB. 64: ReunionEtParcelles : méthodes publiques

signature	description
ReunionEtParcelles (Parcelle* *p1*, Parcelle* *p2*)	construit une parcelle à partir de deux sous-parcelles
Parcelle* **copie** () const	opérateur de copie virtuel
~ReunionEtParcelles ()	destructeur virtuel
bool **inclus** (const VecDBL& *u*) const	indique si le vecteur $\vec{u}$ est inclus dans la parcelle
double **ecartFrontiere** (const VecDBL& *u*) const	calcule l'écart angulaire signé entre le vecteur $\vec{u}$ et la frontière (positif si le point est à l'intérieur de la parcelle, négatif sinon)
bool **visible** (const Cone& *c*) const **throw (CantorErreurs)**	indique si le cône *c* est visible au moins partiellement
Secteurs **visible** (const Secteurs& *s*) const **throw (CantorErreurs)**	filtre la partie visible du secteur *s*
void **appliqueRotation** (const RotDBL& *r*)	transforme l'instance en lui appliquant la rotation *r*
void **integreRotation** (const VecDBL& *axe*, double *angle*) **throw (CantorErreurs, MarmottesErreurs)**	transforme l'instance en la *trainant* selon la rotation définie par l'*axe* et l'*angle*
void **appliqueMarge** (double *m*) **throw (CantorErreurs, MarmottesErreurs)**	transforme l'instance en lui appliquant la marge angulaire *m*
void **applique** (TypeFoncConstChamp *f, void *d) const	applique la fonction *f* (qui ne modifie pas ses arguments) à tous les champs qui composent la parcelle, l'argument anonyme *d* est passé à *f* à chaque appel
void **applique** (TypeFoncChamp *f, void *d)	applique la fonction *f* (qui peut modifier ses arguments) à tous les champs qui composent la parcelle, l'argument anonyme *d* est passé à *f* à chaque appel
void **rechercheChamp** (Champ **adressePtr*)	recherche le champ suivant celui dont l'adresse est donnée en argument, et le retourne dans *adressePtr* (cette fonction est utilisée en interne par les classes dérivées, elle n'est publique qu'en raison de limitations d'accès complexes propres au C++)

exemple d'utilisation

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


Parcelle *LireParcelle (FichierStructure *blocPere, const string& nom)
  throw (MarmottesErreurs)
{
  Parcelle *p1 = 0;
  Parcelle *p2 = 0;
  try
  {
```

```
    // extraction d'une parcelle depuis un sous-bloc nommé d'un bloc père.

    ...

    string nomEssai (TraduitVersExterne ("et"));
    if (blocFils.contientSousBloc (nomEssai.c_str ()))
    { // c'est une réunion "et" de parcelles
      p1 = LireParcelle (&blocFils, string (""));
      p2 = LireParcelle (&blocFils, nomEssai);

      return new ReunionEtParcelles (p1, p2);
    }

    ...

  }

  ...

}
```

conseils d'utilisation spécifiques

Seules les fonctions de lecture des senseurs ont besoin de connaître les types de base, pour construire les parcelles petit à petit. Une fois ces parcelles construites toutes les autres fonctions doivent passer par la classe de base Parcelle.

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 ReunionEtParcelles

nom	type	description
p1_	Parcelle*	pointeur vers la première sous-parcelle
p2_	Parcelle*	pointeur vers la deuxième sous-parcelle

Les méthodes privées sont décrites dans la table 66.

TAB. 66: ReunionEtParcelles : méthodes privées

signature	description
ReunionEtParcelles ()	constructeur par défaut. Il est défini explicitement uniquement pour prévenir celui créé automatiquement par le compilateur et ne doit pas être utilisé.
ReunionEtParcelles (const ReunionEtParcelles& *p*)	constructeur par copie
ReunionEtParcelles& **operator** = (const ReunionEtParcelles& *p*)	affectation

13.22 classe ReunionOuParcelles

description

Cette classe implante les parcelles devant voir leur cible dans l'une ou l'autre de deux sous-parcelles indifféremment (notion de *ou* logique).

interface publique

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

TAB. 67: ReunionOuParcelles : méthodes publiques

signature	description
ReunionOuParcelles (Parcelle* *p1*, Parcelle* *p2*)	construit une parcelle à partir de deux sous-parcelles
Parcelle* **copie** () const	opérateur de copie virtuel
~**ReunionOuParcelles** ()	destructeur virtuel
bool **inclus** (const VecDBL& *u*) const	indique si le vecteur $\vec{u}$ est inclus dans la parcelle
double **ecartFrontiere** (const VecDBL& *u*) const	calcule l'écart angulaire signé entre le vecteur $\vec{u}$ et la frontière (positif si le point est à l'intérieur de la parcelle, négatif sinon)
bool **visible** (const Cone& *c*) const **throw (CantorErreurs)**	indique si le cône *c* est visible au moins partiellement
Secteurs **visible** (const Secteurs& *s*) const **throw (CantorErreurs)**	filtre la partie visible du secteur *s*
void **appliqueRotation** (const RotDBL& *r*)	transforme l'instance en lui appliquant la rotation *r*
void **integreRotation** (const VecDBL& *axe*, double *angle*) **throw (CantorErreurs, MarmottesErreurs)**	transforme l'instance en la *trainant* selon la rotation définie par l'*axe* et l'*angle*
	à suivre ...

TAB. 67: ReunionOuParcelles : méthodes publiques (suite)

signature	description
void **appliqueMarge** (double *m*) **throw (CantorErreurs, MarmottesErreurs)**	transforme l'instance en lui appliquant la marge angulaire *m*
void **applique** (TypeFoncConstChamp *f, void *d) const	applique la fonction *f* (qui ne modifie pas ses arguments) à tous les champs qui composent la parcelle, l'argument anonyme *d* est passé à *f* à chaque appel
void **applique** (TypeFoncChamp *f, void *d)	applique la fonction *f* (qui peut modifier ses arguments) à tous les champs qui composent la parcelle, l'argument anonyme *d* est passé à *f* à chaque appel
void **rechercheChamp** (Champ **adressePtr*)	recherche le champ suivant celui dont l'adresse est donnée en argument, et le retourne dans *adressePtr* (cette fonction est utilisée en interne par les classes dérivées, elle n'est publique qu'en raison de limitations d'accès complexes propres au C++)

exemple d'utilisation

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


Parcelle *LireParcelle (FichierStructure *blocPere, const string& nom)
  throw (MarmottesErreurs)
{
  Parcelle *p1 = 0;
  Parcelle *p2 = 0;
  try
  {
    // extraction d'une parcelle depuis un sous-bloc nommé d'un bloc père.

    ...

    string nomEssai (TraduitVersExterne ("ou"));
    if (blocFils.contientSousBloc (nomEssai.c_str ()))
    { // c'est une réunion "ou" de parcelles
        p1 = LireParcelle (&blocFils, string (""));
        p2 = LireParcelle (&blocFils, nomEssai);

        return new ReunionOuParcelles (p1, p2);
    }

    ...
  }

  ...
```

```
}
```

conseils d'utilisation spécifiques

Seules les fonctions de lecture des senseurs ont besoin de connaître les types de base, pour construire les parcelles petit à petit. Une fois ces parcelles construites toutes les autres fonctions doivent passer par la classe de base Parcelle.

implantation

Les attributs privés sont décrits sommairement dans la table 68, il n'y a pas d'attribut protégé.

TAB. 68: attributs privés de la classe ReunionOuParcelles

nom	type	description
p1_	Parcelle*	pointeur vers la première sous-parcelle
p2_	Parcelle*	pointeur vers la deuxième sous-parcelle

Les méthodes privées sont décrites dans la table 69.

TAB. 69: ReunionOuParcelles : méthodes privées

signature	description
ReunionOuParcelles ()	constructeur par défaut. Il est défini explicitement uniquement pour prévenir celui construit par le compilateur et ne doit pas être utilisé.
ReunionOuParcelles (const ReunionOuParcelles& *p*)	constructeur par copie
ReunionOuParcelles& **operator** = (const ReunionOuParcelles& *p*)	affectation

13.23 classe Senseur

description

Cette classe abstraite est l'interface d'accès aux senseurs d'attitude. Elle est utilisée à de nombreux endroits dans MARMOTTES (modèles analytiques, résolution numérique, interface utilisateur).

Cette classe est destinée à être dérivée pour implanter les fonctions de mesure, de modélisation de consigne, et de calcul d'écart à la consigne.

interface publique

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

Le fichier d'en-tête déclare les types énumérés suivants :

```
enum typeMethode { intersectionCones, integrationSpin, aucuneMethode };
enum codeAstre   { nonSignificatif, soleil, lune, corpsCentral, aucunAstre };
```

TAB. 70: Senseur : méthodes publiques

signature	description
Senseur (const string &*nom*, const RotDBL& *repere*, const VecDBL& *axeCalage*, double *precision*)	construit une instance à partir des composantes communes à tous les types de senseurs (la classe étant abstraite ne peut être instanciée, ce constructeur est destiné à être appelé par les constructeurs des classes dérivées)
Senseur (const Senseur& *s*)	constructeur par copie
Senseur& **operator** = (const Senseur& *s*)	affectation
~Senseur ()	destructeur virtuel (ne fait rien dans cette classe)
Senseur ***copie** () const = 0	opérateur de copie virtuel
const string& **nom** () const	retourne le nom du senseur
const RotDBL& **repereBase** () const	retourne le repère de base du senseur (celui qui a été lu dans le fichier, indépendamment de toute rotation appliquée ultérieurement)
const RotDBL& **repere** () const	retourne le repère courant du senseur (cette rotation convertit les vecteurs exprimés en repère satellites en vecteurs exprimés en repère senseur)
const VecDBL& **axeCalage** () const	retourne l'axe de calage du senseur
double **precision** () const	retourne la précision du senseur (pour pouvoir être comparée directement aux valeurs mesurées, la précision est mémorisée dans l'unité de *mesure*)
bool **conversionConsignes** () const	indique si le senseur doit convertir les consignes en entrée dans son unité interne
bool **conversionMesures** () const	indique si le senseur doit convertir les mesures en sortie dans les unités externes
double **valeurConsigne** () const	retourne la valeur courante de la consigne (pour pouvoir être comparée directement aux valeurs mesurées, la consigne courante est mémorisée dans l'unité de *mesure*)
void **respecterConsignes** ()	force le senseur à respecter les unités de consignes fournies en entrée
void **convertirConsignes** ()	force le senseur à convertir les unités de consignes fournies en entrée
void **respecterMesures** () = 0	force le senseur à respecter les unités de mesures dans ses sorties

à suivre ...

TAB. 70: Senseur : méthodes publiques (suite)

signature	description
void **convertirMesures** () = 0	force le senseur à convertir les unités de mesures dans ses sorties
void **nouveauRepere** (const RotDBL& *nouveau*)	remplace le repère du senseur par le *nouveau*
void **calage** (double *c*) **throw (MarmottesErreurs)**	applique l'angle de calage *c* au senseur, retourne une erreur si le senseur n'a pas d'axe de calage prédéfini
VecDBL **satSens** (const VecDBL& *sat*) const	convertit le vecteur exprimé en repère satellite en vecteur exprimé en repère senseur
VecDBL **sensSat** (const VecDBL& *sens*) const	convertit le vecteur exprimé en repère senseur en vecteur exprimé en repère satellite
double **mesure** (const Etat& *etat*) **throw (MarmottesErreurs)** = 0	retourne la mesure que produirait le senseur dans l'*etat* fourni
int **controlable** (const Etat& *etat*) **throw(MarmottesErreurs**, **CantorErreurs)** = 0	indique si le senseur serait capable de contrôler le satellite dans l'*etat* fourni
int **criteresControlabilite** (const Etat& *etat*, codeAstre **ptrInhibant*, codeAstre **ptrEclipsant*, double **ptrEcartFrontiere*, bool **ptrAmplitudeSignificative*) **throw(MarmottesErreurs**, **CantorErreurs)**	indique si le senseur serait capable de contrôler le satellite dans l'*etat* fourni en précisant les valeurs de chaque critère (présence d'un astre inhibant, éclipsant), écart par rapport à la frontière du champ de vue
int **methode** () const = 0	retourne la méthode à utiliser pour constituer le modèle analytique à un degré de liberté avec le senseur courant (la valeur est un énuméré typeMethode convertit en entier)
void **modeliseConsigne** (const Etat& *etat*, double *valeur*) **throw(CantorErreurs**, **MarmottesErreurs)** = 0	modélise la consigne *valeur* dans l'*etat* fourni
ValeurDerivee1 **foncEcart** (const Etat& *etatPrecedent*, const Etat& *etatResolution*, const RotVD1& *attitude*, const VecVD1& *spin*) = 0	retourne l'écart entre la consigne et la mesure que produirait le senseur dans l'*attitude* et le *spin* fournis, connaissant l'*etatPrecedent* du satellite
void **modifieCible** (const VecDBL& *cible*) **throw (MarmottesErreurs)**	mémorise la *cible* pour les senseurs optiques (les autres senseurs retournent une erreur)
void **initialiseGyro** (double *date*, double *angle*) **throw (MarmottesErreurs)**	initialise les gyromètres intégrateurs pour qu'ils produisent la mesure *angle* à la *date* spécifiée (les autres senseurs retournent une erreur)
void **initialiseDerive** (double *derive*) **throw (MarmottesErreurs)**	initialise la dérive d'un senseur cinématique (les autres senseurs retournent une erreur)

à suivre ...

TAB. 70: Senseur : méthodes publiques (suite)

signature	description
void **modifieReference** (const RotDBL& *reference*) **throw (MarmottesErreurs)**	mémorise la *reference* pour les senseurs de Cardan (les autres senseurs retournent une erreur)
void **prendEtatEnCompte** (const Etat& *etat*)	prend l'*etat* de résolution en compte dans le senseur (les gyromètres intégrateurs s'en servent pour modifier leur angle mesuré à chaque appel, les autres senseurs ne font rien)

exemple d'utilisation

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

void ResolveurAttitude::modeliseConsignes (const Etat& etatPrecedent,
                                           const Etat& etatResolution,
                                           double m1, double m2, double m3)
  throw  (CantorErreurs, MarmottesErreurs)
{ if ((sA1_ == 0) || (sA2_ == 0) || (sB_ == 0))
    throw MarmottesErreurs (MarmottesErreurs::liste_non_initialisee);

  senseursConsigne_ [0]->modeliseConsigne (etatResolution, m1);
  senseursConsigne_ [1]->modeliseConsigne (etatResolution, m2);
  senseursConsigne_ [2]->modeliseConsigne (etatResolution, m3);

  // sauvegarde de l'état précédent complet
  etatPrecedent_ = etatPrecedent;

  // sauvegarde de l'état de résolution (l'attitude n'y est pas significative)
  etatResolution_ = etatResolution;

  // prise en compte des consignes dans le modèle
  modeleCourant_->prendConsignesEnCompte ();
}

static ValeurDerivee1 fonc (double t, void* donnee)
{ // récuperation de l'objet de résolution
  ResolveurAttitude* ptr = (ResolveurAttitude *) donnee;

  // calcul de l'attitude modélisée respectant les premières consignes
  RotVD1 attitude;
  VecVD1 spin;
  ptr->modele ()->attitude (ptr->etatPrecedent (), ptr->date (),
                            ValeurDerivee1 (t, 1.0), ptr->famille (),
                            &attitude, &spin);
```

```
  // calcul de l'écart par rapport à la troisième consigne
  return ptr->sB ()->foncEcart (ptr->etatPrecedent (), ptr->date (),
                                attitude, spin);

}

void ResolveurAttitude::trouveTout ()
  throw (MarmottesErreurs)
{ // vidage des solutions éventuellement trouvées précédemment
  nbSol_ = 0;

  // recherche des solutions dans toutes les familles d'attitudes modélisées
  for (famille_ = 0; famille_ < modeleCourant_->familles (); famille_++)
  { Resolution1Iterateur iter (fonc, (void *) this,
                               -1.0, 1.0, tranches_, -1.0, seuil_);
    double t0;

    // recherche de toutes les solutions d'une famille
    while ((t0 = iter.zeroSuivant ()) < 2.0)
      ajouteSolution (t0);

    ...

  }

  ...

}
```

conseils d'utilisation spécifiques

Cette classe est abstraite, c'est à dire qu'aucune instance ne peut être créée directement. Tout pointeur sur un objet de ce type pointe en réalité sur un objet d'un des types dérivés. Les constructeurs ne servent donc qu'à compléter les constructions d'objets plus gros et ne peuvent être appelés que par les constructeurs des classes dérivées.

Les fonctions de lecture des senseurs ont besoin de connaître les types de base, pour les construire. Les modèles analytiques doivent avoir une information un peu plus fine et différencient les senseurs géométriques des senseurs cinématiques grace à la méthode **methode**. Des logiciels de tracé de champ de vue s'appuyant sur MARMOTTES ont besoin d'une information encore plus fine et utilisent une méthode propre aux senseurs géométriques pour reconnaître les senseurs optiques, avec ou sans inhibitions.

Hormis ces cas particuliers, les senseurs sont tous utilisés de la même façon. L'exemple précédent montre leur initialisation lors de l'exécution de ResolveurAttitude : :**modeliseConsignes**, et l'appel à leur fonction

foncEcart pour la résolution. Cette similitude rend inutiles des exemples et des conseils spécifiques à chaque senseur, les sections correspondantes sont donc omises dans les descriptions individuelles.

Pour implanter un nouveau senseur, les fonctions virtuelles à implanter sont les suivantes : le destructeur, **copie**, **respecterMesures**, **convertirMesures**, **nouveauRepere**, **mesure**, **controlable**, **methode**, **modeliseConsigne**, et **foncEcart**. Certaines des ces fonctions sont déjà définies dans les classes intermédiaires SenseurGeometrique, SenseurOptique, ... et n'ont pas besoin d'être spécialisées plus avant. Il faut regarder ce qui reste à implanter au niveau d'arborescence où se situe le nouveau senseur.

implantation

Les attributs privés sont décrits sommairement dans la table 71, il n'y a pas d'attribut protégé.

TAB. 71: attributs privés de la classe Senseur

nom	type	description
nom_	string	nom du senseur
repereBase_	RotDBL	repère de base du senseur (lu dans le fichier)
repere_	RotDBL	repère courant du senseur
axeCalage_	VecDBL	axe de calage du senseur (nul s'il n'est pas défini dans le fichier)
precision_	double	précision du senseur (en unités de mesure)
convertirConsignes_	bool	indicateur de conversion des consignes en entrée
convertirMesures_	bool	indicateur de consversion des mesures en sortie
valeurConsigne_	double	valeur courante de la consigne (en unités de mesure)

Les méthodes protégées sont décrites dans la table 72.

TAB. 72: Senseur : méthodes protégées

signature	description
Senseur ()	constructeur par défaut. Il est défini explicitement uniquement pour prévenir celui créé automatiquement par le compilateur et ne doit pas être utilisé.
void **reinitialisePrecision** (double *precision*)	réinitialise la précision du senseur (cette fonction est utilisée par les méthodes **respecterMesures** et **convertirMesures** de sorte que la précision soit toujours mémorisée dans les bonnes unités)

13.24 classe SenseurAlpha

description

Cette classe implante les pseudo-senseurs d'ascension droite, elle est pratique par exemple pour optimiser une direction de poussée inertielle (le vecteur interne de référence est alors la direction de poussée).

interface publique

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

TAB. 73: SenseurAlpha : méthodes publiques

signature	description
SenseurAlpha (const string& *nom*, const RotDBL& *repere*, const VecDBL& *axeCalage*, double *precision*, const VecDBL& *observe*)	construit une instance à partir des données technologiques
SenseurAlpha (const SenseurAlpha& *s*)	constructeur par copie
SenseurAlpha& **operator** = (const SenseurAlpha& *s*)	affectation
~**SenseurAlpha** ()	destructeur, ne fait rien dans cette classe
void **respecterMesures** ()	force le senseur à respecter les unités de mesures dans ses sorties
void **convertirMesures** ()	force le senseur à convertir les unités de mesures dans ses sorties
int **controlable** (const Etat& *etat*) **throw (MarmottesErreurs)**	indique si le senseur serait capable de contrôler le satellite dans l'*etat* fourni (toujours vrai pour un senseur d'ascension droite)
Senseur* **copie** () const	opérateur de copie virtuel
typeGeom **typeGeometrique** () const	retourne le type de senseur géométrique (pseudoSenseur)
void **nouveauRepere** (const RotDBL& *nouveau*)	remplace le repère du senseur par le *nouveau*
void **modeliseConsigne** (const Etat& *etat*, double *valeur*) **throw (CantorErreurs, MarmottesErreurs)**	modélise la consigne *valeur* dans l'*etat* fourni
double **mesure** (const Etat& *etat*) **throw (MarmottesErreurs)**	retourne la mesure que produirait le senseur dans l'*etat* fourni
ValeurDerivee1 **foncEcart** (const Etat& *etatPrecedent*, const Etat& *etatResolution*, const RotVD1& *attitude*, const VecVD1& *spin*)	retourne l'écart entre la consigne et la mesure que produirait le senseur dans l'*attitude* et le *spin* fournis, connaissant l'*etatPrecedent* du satellite

implantation

Les attributs privés sont décrits sommairement dans la table 74, il n'y a pas d'attribut protégé.

TAB. 74: attributs privés de la classe SenseurAlpha

nom	type	description
observe_	VecDBL	vecteur interne observé (en repère satellite)
alphaConsigne_	double	sauvegarde de la dernière consigne passée

Les méthodes protégées sont décrites dans la table 75.

TAB. 75: SenseurAlpha : méthodes protégées

signature	description
SenseurAlpha ()	constructeur par défaut. Il est défini explicitement uniquement pour prévenir celui créé automatiquement par le compilateur et ne doit pas être utilisé.

Les méthodes privées sont décrites dans la table 76.

TAB. 76: SenseurAlpha : méthodes privées

signature	description
void **initialiseCible** (double *alpha*)	initialise la cible en repère inertiel dans le plan orthogonal à la direction $(\alpha, 0)$

13.25 classe SenseurCardan

description

Cette classe implante les senseurs de Cardan, c'est à dire les senseurs qui modélisent des rotations successives à partir d'un repère de référence. Plusieurs repères de référence sont disponibles :

géocentrique : ce repère est défini par l'axe $\vec{Z}$ qui pointe vers le centre terre et l'axe $\vec{Y}$ qui est opposé au moment orbital, l'axe $\vec{X}$ déduit des deux précédents est dans le plan de l'orbite, dans le même sens que la vitesse ;

orbital TNW : ce repère est défini par l'axe $\vec{X}$ qui est porté par le vecteur vitesse (on le note souvent $\vec{T}$ dans ce cas) et l'axe $\vec{Z}$ qui est porté par le moment orbital (on le note souvent $\vec{W}$ dans ce cas), l'axe $\vec{Y}$ déduit des deux précédents est dans le plan de l'orbite, dans le sens du nadir (on le note souvent $\vec{N}$ dans ce cas) ;

orbital QSW : ce repère est défini par l'axe $\vec{X}$ qui est pointe à l'opposé du centre terre (on le note souvent $\vec{Q}$ dans ce cas) et l'axe $\vec{Z}$ qui est porté par le moment orbital (on le note souvent $\vec{W}$ dans ce cas),

l'axe $\vec{Y}$ déduit des deux précédents est dans le plan de l'orbite, dans le sens de la vitesse (on le note souvent $\vec{S}$ dans ce cas) ;

inertiel : ce repère est directement le repère inertiel à partir duquel les position, vitesse et attitude sont exprimées :

topocentrique : ce repère est défini par l'axe $\vec{Z}$ qui pointe vers le centre terre et l'axe $\vec{Y}$ qui pointe vers l'Est, l'axe $\vec{X}$ déduit des deux précédents est dans le plan méridien, positif vers le Sud ;

utilisateur : ce repère n'est pas prédéfini dans le code, il est spécifié par l'utilisateur dynamiquement, il peut très bien être modifié à chaque pas de calcul (par exemple à partir de l'attitude donnée par un autre simulateur MARMOTTES, pour évaluer des erreurs de pilotages autour d'une attitude nominale).

Si les mesures de roulis, tangage et lacet sont simultanément nulles, le repère senseur est aligné avec le repère de référence.

interface publique

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

  enum typeGenre     { GenreInconnu,
                       LRTLacet,   LRTRoulis,  LRTTangage,
                       LTRLacet,   LTRTangage, LTRRoulis,
                       RLTRoulis,  RLTLacet,   RLTTangage,
                       RTLRoulis,  RTLTangage, RTLLacet,
                       TLRTangage, TLRLacet,   TLRRoulis,
                       TRLTangage, TRLRoulis,  TRLLacet
                     };

  enum typeReference { ReferenceInconnue,
                       Geocentrique, OrbitalTNW,    OrbitalQSW,
                       Inertiel,     Topocentrique, Utilisateur
                     };
```

TAB. 77: SenseurCardan : méthodes publiques

signature	description
SenseurCardan (int *variation*, const string& *nom*, const RotDBL& *repere*, const VecDBL& *axeCalage*, double *precision*)	construit une instance à partir des données technologiques
SenseurCardan (const SenseurCardan& *s*)	constructeur par copie
SenseurCardan& **operator** = (const SenseurCardan& *s*)	affectation
~**SenseurCardan** ()	destructeur, ne fait rien dans cette classe
	à suivre ...

TAB. 77: SenseurCardan : méthodes publiques (suite)

signature	description
void **respecterMesures** ()	force le senseur à respecter les unités de mesures dans ses sorties
void **convertirMesures** ()	force le senseur à convertir les unités de mesures dans ses sorties
Senseur* **copie** () const	opérateur de copie virtuel
typeGeom **typeGeometrique** () const	retourne le type de senseur géométrique (pseudoSenseur)
double **mesure** (const Etat& *etat*) **throw (MarmottesErreurs)**	retourne la mesure que produirait le senseur dans l' *etat* fourni
int **controlable** (const Etat& *etat*) **throw (MarmottesErreurs)**	indique si le senseur serait capable de contrôler le satellite dans l'*etat* fourni
void **modeliseConsigne** (const Etat& *etat*, double *valeur*) **throw (CantorErreurs, MarmottesErreurs)**	modélise la consigne *valeur* dans l'*etat* fourni
void **modifieReference** (const RotDBL& *reference*) **throw (MarmottesErreurs)**	mémorise le repère de *reference*. Cette méthode n'est utilisable que si le fichier de description du senseur indique que le repère doit être fourni par l'utilisateur (type de référence : `Utilisateur`).

implantation

Les attributs privés sont décrits sommairement dans la table 78, il n'y a pas d'attribut protégé.

TAB. 78: attributs privés de la classe SenseurCardan

nom	type	description
genre_	typeGenre	indique le type de senseur de Cardan modélisé
reference_	typeReference	indique le type de repère de référence
refUtilInitialisee_	bool	indique si la référence utilisateur est initialisée
referenceUtilisateur_	RotDBL	repère de référence donné par l'utilisateur
xIn_	VecDBL	projection de l'axe $\vec{x}$ du repère de référence en repère inertiel
yIn_	VecDBL	projection de l'axe $\vec{y}$ du repère de référence en repère inertiel
zIn_	VecDBL	projection de l'axe $\vec{z}$ du repère de référence en repère inertiel

Les méthodes protégées sont décrites dans la table 79.

TAB. 79: SenseurCardan : méthodes protégées

signature	description
SenseurCardan ()	constructeur par défaut. Il est défini explicitement uniquement pour prévenir celui créé automatiquement par le compilateur et ne doit pas être utilisé.

Les méthodes privées sont décrites dans la table 80.

TAB. 80: SenseurCardan : méthodes privées

signature	description
int **metAJourReference** (const Etat& *etat*) **throw (MarmottesErreurs)**	met à jour le repère de référence en fonction de l'*etat* spécifié.

13.26 classe SenseurCartesien

description

Cette classe implante les senseurs cartésiens, c'est à dire les senseurs qui mesurent des coordonnées cartésiennes de leur direction cible. Ces pseudo-senseurs permettent par exemple de faire des tracés de passage d'astre dans des champs de vue de senseurs réels, en associant un triplet de tels senseur pour obtenir les trois coordonnées de l'astre.

interface publique

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

TAB. 81: SenseurCartesien : méthodes publiques

signature	description
SenseurCartesien (const string& *nom*, const RotDBL& *repere*, const VecDBL& *axeCalage*, double *precision*, codeCible *code*, const StationCible **ptrStation*, const VecDBL& *observe*, Parcelle* *ptrChampDeVue*, Parcelle* *ptrChampInhibitionSoleil*, Parcelle* *ptrChampInhibitionLune*, Parcelle* *ptrChampInhibitionCentral*, double *margeEclipseSoleil*, double *margeEclipseLune*, double *seuilPhaseLune*, const VecDBL& *reference*)	construit une instance à partir des données technologiques
SenseurCartesien (const SenseurCartesien& *s*)	constructeur par copie
SenseurCartesien& **operator** = (const SenseurCartesien& *s*)	affectation
~**SenseurCartesien** ()	destructeur, ne fait rien dans cette classe
void **respecterMesures** ()	force le senseur à respecter les unités de mesures dans ses sorties
void **convertirMesures** ()	force le senseur à convertir les unités de mesures dans ses sorties
Senseur* **copie** () const	opérateur de copie virtuel
void **modeliseConsigne** (const Etat& *etat*, double *valeur*) **throw (CantorErreurs, MarmottesErreurs)**	modélise la consigne *valeur* dans l'*etat* fourni
double **mesure** (const Etat& *etat*) **throw (MarmottesErreurs)**	retourne la mesure que produirait le senseur dans l' *etat* fourni

implantation

Il n'y a ni attribut privé, ni attribut protégé.

Les méthodes protégées sont décrites dans la table 82.

TAB. 82: SenseurCartesien : méthodes protégées

signature	description
SenseurCartesien ()	constructeur par défaut. Il est défini explicitement uniquement pour prévenir celui créé automatiquement par le compilateur et ne doit pas être utilisé.

13.27 classe SenseurCinematique

description

Cette classe implante les senseurs cinématiques (gyromètres), c'est à dire les senseurs qui mesurent la projection d'une vitesse angulaire sur un axe sensible. Ces senseurs constituent une famille complètement disjointe des senseurs géométriques.

interface publique

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

TAB. 83: SenseurCinematique : méthodes publiques

signature	description
SenseurCinematique (const string& *nom*, const RotDBL& *repere*, const VecDBL& *axeCalage*, double *precision*, const VecDBL& *axeSensible*) **throw (CantorErreurs)**	construit une instance à partir des données technologiques
SenseurCinematique (const SenseurCinematique& *s*)	constructeur par copie
SenseurCinematique& **operator** = (const SenseurCinematique& *s*)	affectation
~SenseurCinematique ()	destructeur, ne fait rien dans cette classe
void **respecterMesures** ()	force le senseur à respecter les unités de mesures dans ses sorties
void **convertirMesures** ()	force le senseur à convertir les unités de mesures dans ses sorties
Senseur* **copie** () const	opérateur de copie virtuel
void **nouveauRepere** (const RotDBL& *nouveau*)	remplace le repère du senseur par le *nouveau*
const VecDBL& **sensible** () const	retourne l'axe sensible du senseur
double **omega** () const	retourne la consigne courante
double **derive** () const	retourne la dérive courante
int **methode** () const	retourne la méthode à utiliser pour constituer le modèle analytique à un degré de liberté avec le senseur courant (integrationSpin)
void **modeliseConsigne** (const Etat& *etat*, double *valeur*) **throw (CantorErreurs, MarmottesErreurs)**	modélise la consigne *valeur* dans l'*etat* fourni

à suivre ...

TAB. 83: SenseurCinematique : méthodes publiques (suite)

signature	description
double **mesure** (const Etat& *etat*) **throw (MarmottesErreurs)**	retourne la mesure que produirait le senseur dans l'*etat* fourni
int **controlable** (const Etat& *etat*) **throw (MarmottesErreurs)**	indique si le senseur serait capable de contrôler le satellite dans l'*etat* fourni (toujours vrai pour un senseur cinématique)
ValeurDerivee1 **foncEcart** (const Etat& *etatPrecedent*, const Etat& *etatResolution*, const RotVD1& *attitude*, const VecVD1& *spin*)	retourne l'écart entre la consigne et la mesure que produirait le senseur dans l'*attitude* et le *spin* fournis, connaissant l'*etatPrecedent* du satellite
void **initialiseDerive** (double *derive*) **throw (MarmottesErreurs)**	initialise la dérive d'un senseur cinématique

implantation

Les attributs privés sont décrits sommairement dans la table 84, il n'y a pas d'attribut protégé.

TAB. 84: attributs privés de la classe SenseurCinematique

nom	type	description
sensible_	VecDBL	axe sensible du senseur en repère satellite
sensibleVD1_	VecVD1	conversion de sensible_ en VecVD1
omega_	double	consigne courante
derive_	double	dérive courante

Les méthodes protégées sont décrites dans la table 85.

TAB. 85: SenseurCinematique : méthodes protégées

signature	description
SenseurCinematique ()	constructeur par défaut. Il est défini explicitement uniquement pour prévenir celui créé automatiquement par le compilateur et ne doit pas être utilisé.

13.28 classe SenseurDelta

description

Cette classe implante les pseudo-senseurs de déclinaison, elle est pratique par exemple pour optimiser une direction de poussée inertielle (le vecteur interne de référence est alors la direction de poussée).

interface publique

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

TAB. 86: SenseurDelta : méthodes publiques

signature	description
SenseurDelta (const string& *nom*, const RotDBL& *repere*, const VecDBL& *axeCalage*, double *precision*, const VecDBL& *observe*)	construit une instance à partir des données technologiques
SenseurDelta (const SenseurDelta& *s*)	constructeur par copie
SenseurDelta& **operator** = (const SenseurDelta& *s*)	affectation
~**SenseurDelta** ()	destructeur, ne fait rien dans cette classe
void **respecterMesures** ()	force le senseur à respecter les unités de mesures dans ses sorties
void **convertirMesures** ()	force le senseur à convertir les unités de mesures dans ses sorties
int **controlable** (const Etat& *etat*) **throw (MarmottesErreurs)**	indique si le senseur serait capable de contrôler le satellite dans l'*etat* fourni (toujours vrai pour un senseur de déclinaison)
Senseur* **copie** () const	opérateur de copie virtuel
typeGeom **typeGeometrique** () const	retourne le type de senseur géométrique (pseudoSenseur)
void **nouveauRepere** (const RotDBL& *nouveau*)	remplace le repère du senseur par le *nouveau*
void **modeliseConsigne** (const Etat& *etat*, double *valeur*) **throw (CantorErreurs, MarmottesErreurs)**	modélise la consigne *valeur* dans l'*etat* fourni
double **mesure** (const Etat& *etat*) **throw (MarmottesErreurs)**	retourne la mesure que produirait le senseur dans l'*etat* fourni

implantation

Les attributs privés sont décrits sommairement dans la table 87, il n'y a pas d'attribut protégé.

TAB. 87: attributs privés de la classe SenseurDelta

nom	type	description
observe_	VecDBL	vecteur interne observé (en repère satellite)

Les méthodes protégées sont décrites dans la table 88.

TAB. 88: SenseurDelta : méthodes protégées

signature	description
SenseurDelta ()	constructeur par défaut. Il est défini explicitement uniquement pour prévenir celui créé automatiquement par le compilateur et ne doit pas être utilisé.

13.29 classe SenseurDiedre

description

Cette classe modélise les senseurs d'angles dièdres, qui représentent la plupart des senseurs réels embarqués.

interface publique

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

TAB. 89: SenseurDiedre : méthodes publiques

signature	description
SenseurDiedre (const string& *nom*, const RotDBL& *repere*, const VecDBL& *axeCalage*, double *precision*, codeCible *code*, const StationCible **ptrStation*, const VecDBL& *observe*, Parcelle* *ptrChampDeVue*, Parcelle* *ptrChampInhibitionSoleil*, Parcelle* *ptrChampInhibitionLune*, Parcelle* *ptrChampInhibitionCentral*, double *margeEclipseSoleil*, double *margeEclipseLune*, double *seuilPhaseLune*, const VecDBL& *referenceZero*, const VecDBL& *axeSensible*)	construit une instance à partir des données technologiques
SenseurDiedre (const SenseurDiedre& *s*)	constructeur par copie
SenseurDiedre& **operator** = (const SenseurDiedre& *s*)	affectation
~SenseurDiedre ()	destructeur, ne fait rien dans cette classe
void **respecterMesures** ()	force le senseur à respecter les unités de mesures dans ses sorties
void **convertirMesures** ()	force le senseur à convertir les unités de mesures dans ses sorties
Senseur* **copie** () const	opérateur de copie virtuel
void **nouveauRepere** (const RotDBL& *nouveau*)	remplace le repère du senseur par le *nouveau*
void **modeliseConsigne** (const Etat& *etat*, double *valeur*) **throw (CantorErreurs, MarmottesErreurs)**	modélise la consigne *valeur* dans l'*etat* fourni
double **mesure** (const Etat& *etat*) **throw (MarmottesErreurs)**	retourne la mesure que produirait le senseur dans l'*etat* fourni
ValeurDerivee1 **foncEcart** (const Etat& *etatPrecedent*, const Etat& *etatResolution*, const RotVD1& *attitude*, const VecVD1& *spin*)	retourne l'écart entre la consigne et la mesure que produirait le senseur dans l'*attitude* et le *spin* fournis, connaissant l'*etatPrecedent* du satellite

implantation

Les attributs privés sont décrits sommairement dans la table 90, il n'y a pas d'attribut protégé.

TAB. 90: attributs privés de la classe SenseurDiedre

nom	type	description
dansPlan0_	VecDBL	vecteur situé le plan des mesures nulles
normalPlan0_	VecDBL	vecteur normal au plan des mesures nulles

Les méthodes protégées sont décrites dans la table 91.

TAB. 91: SenseurDiedre : méthodes protégées

signature	description
SenseurDiedre ()	constructeur par défaut. Il est défini explicitement uniquement pour prévenir celui créé automatiquement par le compilateur et ne doit pas être utilisé.

13.30 classe SenseurElevation

description

Cette classe implante les senseurs mesurant des angles d'élévation par rapport à un plan de référence, elle permet d'obtenir les coordonnées sphériques d'un astre en associant un SenseurDiedre et un SenseurElevation.

interface publique

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

TAB. 92: SenseurElevation : méthodes publiques

signature	description
SenseurElevation (const string& *nom*, const RotDBL& *repere*, const VecDBL& *axeCalage*, double *precision*, codeCible *code*, const StationCible **ptrStation*, const VecDBL& *observe*, Parcelle* *ptrChampDeVue*, Parcelle* *ptrChampInhibitionSoleil*, Parcelle* *ptrChampInhibitionLune*, Parcelle* *ptrChampInhibitionCentral*, double *margeEclipseSoleil*, double *margeEclipseLune*, double *seuilPhaseLune*, const VecDBL& *reference*)	construit une instance à partir des données technologiques
SenseurElevation (const SenseurElevation& *s*)	constructeur par copie
SenseurElevation& **operator** = (const SenseurElevation& *s*)	affectation
~SenseurElevation ()	destructeur, ne fait rien dans cette classe
Senseur* **copie** () const	opérateur de copie virtuel
void **modeliseConsigne** (const Etat& *etat*, double *valeur*) **throw (CantorErreurs, MarmottesErreurs)**	modélise la consigne *valeur* dans l'*etat* fourni
double **mesure** (const Etat& *etat*) **throw (MarmottesErreurs)**	retourne la mesure que produirait le senseur dans l'*etat* fourni

implantation

Il n'y a ni attribut privé, ni attribut protégé.

Les méthodes protégées sont décrites dans la table 93.

TAB. 93: SenseurElevation : méthodes protégées

signature	description
SenseurElevation ()	constructeur par défaut. Il est défini explicitement uniquement pour prévenir celui créé automatiquement par le compilateur et ne doit pas être utilisé.

13.31 classe SenseurFonction

description

Cette classe abstraite est l'interface d'accès aux senseurs mesurant une fonction scalaire définie sur la sphère unité. Ce type de senseurs a été créé à l'origine pour faciliter le calcul de bilans de liaison, les fonctions modélisées représentant les gains d'antennes bord. C'est en raison de cette application que ces senseurs sont des spécialisations de la classe SenseurOptique, afin de bénéficier de la notion de champ de vue, généralement associée à une iso-valeur du gain.

Les fonctions implantées par les classes dérivées étant généralement difficiles à inverser, on considère par défaut que l'on ne *peut pas utiliser ces senseurs en consigne* , ils sont destinés à être utilisés en *mesure uniquement*.

Cette classe introduit la méthode fonction que les classes dérivée doivent implémenter pour le calcul explicite de la fonction mesurée.

interface publique

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

TAB. 94: SenseurFonction : méthodes publiques

signature	description
SenseurFonction (const string& *nom*, const RotDBL& *repere*, const VecDBL& *axeCalage*, double *precision*, codeCible *code*, const StationCible **ptrStation*, const VecDBL& *observe*, Parcelle* *ptrChampDeVue*, Parcelle* *ptrChampInhibitionSoleil*, Parcelle* *ptrChampInhibitionLune*, Parcelle* *ptrChampInhibitionCentral*, double *margeEclipseSoleil*, double *margeEclipseLune*, double *seuilPhaseLune*, const VecDBL& *axe*, const VecDBL& *origine*) **throw (CantorErreurs)**	construit une instance à partir des données technologiques
SenseurFonction (const SenseurFonction& *s*)	constructeur par copie
SenseurFonction& **operator** = (const SenseurFonction& *s*)	affectation
~SenseurFonction ()	destructeur, ne fait rien dans cette classe
	à suivre ...

TAB. 94: SenseurFonction : méthodes publiques (suite)

signature	description
void **respecterMesures** ()	la classe n'ayant aucune information sur la signification des mesures, cette fonction est sans effet ici
void **convertirMesures** ()	la classe n'ayant aucune information sur la signification des mesures, cette fonction est sans effet ici
void **nouveauRepere** (const RotDBL& *nouveau*)	remplace le repère du senseur par le *nouveau*
int **methode** () const	les fonctions implantées par les classes dérivées étant généralement difficiles à inverser, l'implantation de cette méthode fournie ici est de retourner la constante `aucuneMethode`, qui implique que l'on ne *peut pas utiliser ces senseurs en consigne*
void **modeliseConsigne** (const Etat& etat, double valeur) **throw (CantorErreurs, MarmottesErreurs)**	ce senseur ne peut pas être utilisé en consigne, cette méthode retourne donc systématiquement une erreur
double **mesure** (const Etat& etat) **throw (MarmottesErreurs)**	retourne la mesure que fournirait le senseur dans l'état fourni, c'est à dire la valeur de la fonction sous-jacente dans la direction de la cible
ValeurDerivee1 **foncEcart** (const Etat& *etatPrecedent*, const Etat& *etatResolution*, const RotVD1& *attitude*, const VecVD1& *spin*)	ce senseur ne peut pas être utilisé en consigne, on retourne donc systématiquement la valeur 1.0
double **fonction** (double *azimut*, double *depointage*) const **throw (MarmottesErreurs)** = 0	cette méthode virtuelle évalue la valeur de fonction sous-jacente au point défini par *azimut* et *depointage*. Elle doit être implantée par les classes dérivées. Le *depointage* est l'angle entre le point de calcul et l'axe donné à la construction, l'*azimut* est l'angle autour de l'axe, compté à partir de l'origine donnée à la construction. À titre d'exemple, si l'on considère que l'axe était $\vec{k}$ et que l'origine était $\vec{i}$, alors la direction $\vec{i}$ correspond à *azimut*=0 et *depointage*=$\pi/2$, la direction $\vec{j}$ correspond à *azimut*=$\pi/2$ et *depointage*=$\pi/2$, et la direction $\vec{k}$ correspond à *azimut* quelconque et *depointage*=0

implantation

Les attributs privés sont décrits sommairement dans la table 95, il n'y a pas d'attribut protégé.

TAB. 95: attributs privés de la classe SenseurFonction

nom	type	description
i_	VecDBL	premier axe canonique du repère de calcul de la fonction sous-jacente
		à suivre ...

TAB. 95: attributs privés de la classe SenseurFonction (suite)

nom	type	description
j_	VecDBL	deuxième axe canonique du repère de calcul de la fonction sous-jacente
k_	VecDBL	troisième axe canonique du repère de calcul de la fonction sous-jacente

Les méthodes protégées sont décrites dans la table 96.

TAB. 96: SenseurFonction : méthodes protégées

signature	description
SenseurFonction ()	constructeur par défaut. Il est défini explicitement uniquement pour prévenir celui créé automatiquement par le compilateur et ne doit pas être utilisé.

13.32 classe SenseurFonctionEchant1D

description

Cette classe implante les pseudo-senseurs mesurant des gains d'antennes bord dans une direction particulière définie par la cible (typiquement une station sol). Ce type de senseur permet de calculer la part liée uniquement à l'attitude dans un bilan de liaison. La forme du gain modélisée par ce senseur est un échantillonnage à symétrie axiale spécifié par l'utilisateur dans le fichier de description du senseur. La valeur retournée est une interpolation linéaire entre les points échantillonnés.

Si le plus petit angle de dépointage de l'échantillon est non nul, la fonction est considérée comme constante entre 0 et ce plus petit angle. Si le plus grand angle de dépointage de l'échantillon est inférieur à π, la fonction est considérée comme constante entre ce plus grand angle et π. Une conséquence est qu'un échantillon ne comportant qu'un seul point conduira à la modélisation (peu utile *a priori*) d'une fonction constante entre 0 et π.

La fonction n'étant pas inversible pour tous les échantillonnages et un souci d'homogénéité avec la classe SenseurFonctionSinCard2 nous ont poussés à *interdire l'utilisation de ces senseurs en consigne* , ils sont destinés à être utilisés en *mesure uniquement*.

interface publique

```
#include "marmottes/SenseurFonctionEchant1D.h"
```

TAB. 97: SenseurFonctionEchant1D : méthodes publiques

signature	description
SenseurFonctionEchant1D (const string& *nom*, const RotDBL& *repere*, const VecDBL& *axeCalage*, double *precision*, codeCible *code*, const StationCible **ptrStation*, const VecDBL& *observe*, Parcelle* *ptrChampDeVue*, Parcelle* *ptrChampInhibitionSoleil*, Parcelle* *ptrChampInhibitionLune*, Parcelle* *ptrChampInhibitionCentral*, double *margeEclipseSoleil*, double *margeEclipseLune*, double *seuilPhaseLune*, const VecDBL& *axe*, const VecDBL& *origine*, int *nbEchantillons*, double **angles*, double **valeurs*) **throw (CantorErreurs, MarmottesErreurs)**	construit une instance à partir des données technologiques
SenseurFonctionEchant1D (const SenseurFonctionEchant1D& *s*)	constructeur par copie
SenseurFonctionEchant1D& **operator** = (const SenseurFonctionEchant1D& *s*)	affectation
~SenseurFonctionEchant1D ()	destructeur, ne fait rien dans cette classe
Senseur* **copie** () const	opérateur de copie virtuel
double **fonction** (double *azimut*, double *depointage*) const **throw (MarmottesErreurs)**	cette méthode évalue la fonction échantillonnée pour la valeur θ de *depointage*

implantation

Les attributs privés sont décrits sommairement dans la table 98, il n'y a pas d'attribut protégé.

TAB. 98: attributs privés de la classe SenseurFonctionEchant1D

nom	type	description
echantillon_	map<double, double>	échantillon de points (dépointage, valeur)

Les méthodes protégées sont décrites dans la table 99.

TAB. 99: SenseurFonctionEchant1D : méthodes protégées

signature	description
SenseurFonctionEchant1D ()	constructeur par défaut. Il est défini explicitement uniquement pour prévenir celui créé automatiquement par le compilateur et ne doit pas être utilisé.

13.33 classe SenseurFonctionGauss

description

Cette classe implante les pseudo-senseurs mesurant des gains d'antennes bord dans une direction particulière définie par la cible (typiquement une station sol). Ce type de senseur permet de calculer la part liée uniquement à l'attitude dans un bilan de liaison. La forme du gain modélisée par ce senseur est :

$$g = 10 \times \frac{\log K e^{\frac{-x^2}{2}}}{\log 10}$$

où θ est le dépointage entre l'axe d'antenne et la cible. La valeur retournée est donc directement une valeur en dB.

La fonction est inversible, mais sur un intervalle un peu étrange $[g_{\max} - \frac{5\pi^2}{\log 10}; g_{\max}]$, cette raison et un souci d'homogénéité avec la classe SenseurFonctionSinCard2 nous ont poussés à *interdire l'utilisation de ces senseurs en consigne*, ils sont destinés à être utilisés en *mesure uniquement*.

interface publique

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

TAB. 100: SenseurFonctionGauss : méthodes publiques

signature	description
SenseurFonctionGauss (const string& *nom*, const RotDBL& *repere*, const VecDBL& *axeCalage*, double *precision*, codeCible *code*, const StationCible **ptrStation*, const VecDBL& *observe*, Parcelle* *ptrChampDeVue*, Parcelle* *ptrChampInhibitionSoleil*, Parcelle* *ptrChampInhibitionLune*, Parcelle* *ptrChampInhibitionCentral*, double *margeEclipseSoleil*, double *margeEclipseLune*, double *seuilPhaseLune*, const VecDBL& *axe*, const VecDBL& *origine*, double *maximum*, double *angle3dB*) **throw (CantorErreurs)**	construit une instance à partir des données technologiques
SenseurFonctionGauss (const SenseurFonctionGauss& *s*)	constructeur par copie
SenseurFonctionGauss& **operator** = (const SenseurFonctionGauss& *s*)	affectation
~SenseurFonctionGauss ()	destructeur, ne fait rien dans cette classe
Senseur* **copie** () const	opérateur de copie virtuel
double **fonction** (double *azimut*, double *depointage*) const **throw (MarmottesErreurs)**	cette méthode évalue la fonction de gain $g = 10 \times \frac{\log K e^{\frac{-x^2}{2}}}{\log 10}$ où θ est le *depointage*

implantation

Les attributs privés sont décrits sommairement dans la table 101, il n'y a pas d'attribut protégé.

TAB. 101: attributs privés de la classe SenseurFonctionGauss

nom	type	description
maximum_	double	valeur maximale en dB du gain dans la direction de l'axe
angle3dB_	double	angle de dépointage pour lequel le gain chute de 3 dB (c'est à dire que la fonction dans le logarithme est divisée par deux)

Les méthodes protégées sont décrites dans la table 102.

TAB. 102: SenseurFonctionGauss : méthodes protégées

signature	description
SenseurFonctionGauss ()	constructeur par défaut. Il est défini explicitement uniquement pour prévenir celui créé automatiquement par le compilateur et ne doit pas être utilisé.

13.34 classe SenseurFonctionSinCard2

description

Cette classe implante les pseudo-senseurs mesurant des gains d'antennes bord dans une direction particulière définie par la cible (typiquement une station sol). Ce type de senseur permet de calculer la part liée uniquement à l'attitude dans un bilan de liaison. La forme du gain modélisée par ce senseur est :

$$g = 10 \times \frac{\log K \left(\frac{\sin\theta}{\theta}\right)^2}{\log 10}$$

où θ est le dépointage entre l'axe d'antenne et la cible. La valeur retournée est donc directement une valeur en dB.

La fonction n'étant pas inversible (à moins de se restreindre au lobe primaire), on ne *peut pas utiliser ces senseurs en consigne*, ils sont destinés à être utilisés en *mesure uniquement*.

interface publique

```
#include "marmottes/SenseurFonctionSinCard2.h"
```

TAB. 103: SenseurFonctionSinCard2 : méthodes publiques

signature	description
SenseurFonctionSinCard2 (const string& *nom*, const RotDBL& *repere*, const VecDBL& *axeCalage*, double *precision*, codeCible *code*, const StationCible **ptrStation*, const VecDBL& *observe*, Parcelle* *ptrChampDeVue*, Parcelle* *ptrChampInhibitionSoleil*, Parcelle* *ptrChampInhibitionLune*, Parcelle* *ptrChampInhibitionCentral*, double *margeEclipseSoleil*, double *margeEclipseLune*, double *seuilPhaseLune*, const VecDBL& *axe*, const VecDBL& *origine*, double *maximum*, double *angle3dB*) **throw (CantorErreurs)**	construit une instance à partir des données technologiques
SenseurFonctionSinCard2 (const SenseurFonctionSinCard2& *s*)	constructeur par copie
SenseurFonctionSinCard2& **operator** = (const SenseurFonctionSinCard2& *s*)	affectation
~SenseurFonctionSinCard2 ()	destructeur, ne fait rien dans cette classe
Senseur* **copie** () const	opérateur de copie virtuel
double **fonction** (double *azimut*, double *depointage*) const **throw (MarmottesErreurs)**	cette méthode évalue la fonction de gain $g = 10 \times \frac{\log K \left(\frac{\sin\theta}{\theta}\right)^2}{\log 10}$ où θ est le *depointage*

implantation

Les attributs privés sont décrits sommairement dans la table 104, il n'y a pas d'attribut protégé.

TAB. 104: attributs privés de la classe SenseurFonctionSinCard2

nom	type	description
maximum_	double	valeur maximale en dB du gain dans la direction de l'axe
		à suivre ...

TAB. 104: attributs privés de la classe SenseurFonctionSinCard2 (suite)

nom	type	description
angle3dB_	double	angle de dépointage pour lequel le gain chute de 3 dB (c'est à dire que la fonction dans le logarithme est divisée par deux)

Les méthodes protégées sont décrites dans la table 105.

TAB. 105: SenseurFonctionSinCard2 : méthodes protégées

signature	description
SenseurFonctionSinCard2 ()	constructeur par défaut. Il est défini explicitement uniquement pour prévenir celui créé automatiquement par le compilateur et ne doit pas être utilisé.

13.35 classe SenseurGeometrique

description

Cette classe abstraite est l'interface d'accès aux senseurs géométriques. Elle est utilisée par le modèle analytique spécifique aux senseurs géométriques.

Cette classe implante la fonction **methode**, elle implante une version générale de la fonction de calcul d'écart à la consigne (certaines classes dérivées s'en contentent, d'autres la spécialisent), et introduit de nouvelles méthodes spécifiques aux senseurs géométriques (**secteursConsigne**, **cible**, **typeGeometrique**, **nouvelleReferenceSecteurs**).

interface publique

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

enum typeGeom { optique, pseudoSenseur };
```

TAB. 106: SenseurGeometrique : méthodes publiques

signature	description
SenseurGeometrique (const string& *nom*, const RotDBL& *repere*, const VecDBL& *axeCalage*, double *precision*)	construit une instance à partir des données technologiques
SenseurGeometrique (const SenseurGeometrique& *s*)	constructeur par copie
SenseurGeometrique& **operator** = (const SenseurGeometrique& *s*)	affectation
~**SenseurGeometrique** ()	destructeur, ne fait rien dans cette classe
void **nouveauRepere** (const RotDBL& *nouveau*)	remplace le repère du senseur par le *nouveau*
void **nouvelleReferenceSecteurs** (const VecDBL& *u*)	remplace la référence des secteurs de consigne par *u*
const Secteurs& **secteursConsigne** () const	retourne le secteur de consigne (pour le modèle analytique)
const VecDBL& **cible** () const	retourne le vecteur cible en repère inertiel
typeGeom **typeGeometrique** () const = 0	retourne le type de senseur géométrique (méthode virtuelle pure devant être implantée par les classes dérivées)
int **methode** () const	retourne la méthode à utiliser pour constituer le modèle analytique à un degré de liberté avec le senseur courant (intersectionCones)
ValeurDerivee1 **foncEcart** (const Etat& *etatPrecedent*, const Etat& *etatResolution*, const RotVD1& *attitude*, const VecVD1& *spin*)	retourne l'écart entre la consigne et la mesure que produirait le senseur dans l'*attitude* et le *spin* fournis, connaissant l'*etatPrecedent* du satellite

implantation

Les attributs protégés sont décrits sommairement dans la table 107, il n'y a pas d'attribut privé.

TAB. 107: attributs protégés de la classe SenseurGeometrique

nom	type	description
secteursConsigne_	Secteurs	secteur de consigne pour la consigne courante
axeVD1_	VecVD1	axe du cône de consigne convertit en VecVD1
cible_	VecDBL	vecteur cible en repère inertiel
cibleVD1_	VecVD1	conversion de cible_ en VecVD1

Les méthodes protégées sont décrites dans la table 108.

TAB. 108: SenseurGeometrique : méthodes protégées

signature	description
void **verifieConsigne** () const **throw (MarmottesErreurs)**	vérifie si le cône de consigne n'est pas dégénéré ; cette méthode doit être appelée par toutes les classes dérivées générant des cônes de demi-angle d'ouverture variable à la fin de la prise en compte de la consigne
SenseurGeometrique ()	constructeur par défaut. Il est défini explicitement uniquement pour prévenir celui créé automatiquement par le compilateur et ne doit pas être utilisé.

13.36 classe SenseurGyroInteg

description

Cette classe implante les gyromètres intégrateurs, c'est à dire les senseurs qui accumulent les rotations élémentaires autour d'un axe sensible depuis l'instant de leur remise à zéro.

Bien que ces senseurs produisent des mesures angulaires il s'agit de senseurs cinématiques et non de senseurs géométriques : les mesures qu'ils produisent dépendent de tous les états intermédiaires entre le moment de la remise à zéro et les instants de mesures.

Cette classe est la seule a implanter une version non triviale de la méthode **prendEtatEnCompte**, pour tenir compte du pilotage de l'attitude à chaque résolution.

interface publique

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

TAB. 109: SenseurGyroInteg : méthodes publiques

signature	description
SenseurGyroInteg (const string& *nom*, const RotDBL& *repere*, const VecDBL& *axeCalage*, double *precision*, const VecDBL& *axeSensible*)	construit une instance à partir des données technologiques
SenseurGyroInteg (const SenseurGyroInteg& *s*)	constructeur par copie
SenseurGyroInteg& **operator** = (const SenseurGyroInteg& *s*)	affectation
~SenseurGyroInteg ()	destructeur, ne fait rien dans cette classe
Senseur* **copie** () const	opérateur de copie virtuel
	à suivre ...

TAB. 109: SenseurGyroInteg : méthodes publiques (suite)

signature	description
void **modeliseConsigne** (const Etat& *etat*, double *valeur*) **throw (CantorErreurs, MarmottesErreurs)**	modélise la consigne *valeur* dans l'*etat* fourni
double **mesure** (const Etat& *etat*) **throw (MarmottesErreurs)**	retourne la mesure que produirait le senseur dans l'*etat* fourni
ValeurDerivee1 **foncEcart** (const Etat& *etatPrecedent*, const Etat& *etatResolution*, const RotVD1& *attitude*, const VecVD1& *spin*)	retourne l'écart entre la consigne et la mesure que produirait le senseur dans l'*attitude* et le *spin* fournis, connaissant l'*etatPrecedent* du satellite
void **initialiseGyro** (double *date*, double *angle*) **throw (MarmottesErreurs)**	initialise le senseur pour qu'il produise la mesure *angle* à la *date* spécifiée
void **prendEtatEnCompte** (const Etat& *etat*)	prend l'*etat* de résolution en compte dans le senseur, pour mettre à jour la mesure en intégrant le spin depuis la dernière mise à jour

implantation

Les attributs privés sont décrits sommairement dans la table 110, il n'y a pas d'attribut protégé.

TAB. 110: attributs privés de la classe SenseurGyroInteg

nom	type	description
t0_	double	date de la dernière mise à jour du senseur
alpha0_	double	angle courant à la date t0_
initialise_	bool	indique si le senseur a été initialisé

Les méthodes protégées sont décrites dans la table 111.

TAB. 111: SenseurGyroInteg : méthodes protégées

signature	description
SenseurGyroInteg ()	constructeur par défaut. Il est défini explicitement uniquement pour prévenir celui créé automatiquement par le compilateur et ne doit pas être utilisé.

13.37 classe SenseurLimbe

description

Cette classe implante les senseurs de limbe, c'est à dire des senseurs d'angles dièdres classiques observant obligatoirement le bord d'un astre ayant un diamètre apparent suffisant.

interface publique

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

TAB. 112: SenseurLimbe : méthodes publiques

signature	description
SenseurLimbe (const string& *nom*, const RotDBL& *repere*, const VecDBL& *axeCalage*, double *precision*, Parcelle **ptrChampDeVue*, Parcelle **ptrChampInhibitionSoleil*, Parcelle **ptrChampInhibitionLune*, Parcelle **ptrChampInhibitionCentral*, double *margeEclipseSoleil*, double *margeEclipseLune*, double *seuilPhaseLune*, const VecDBL& *referenceZero*, const VecDBL& *axeSensible*)	construit une instance à partir des données technologiques
SenseurLimbe (const SenseurLimbe& *s*)	constructeur par copie
SenseurLimbe& **operator** = (const SenseurLimbe& *s*)	affectation
~SenseurLimbe ()	destructeur
Senseur* **copie** () const	opérateur de copie virtuel
int **controlable** (const Etat& *etat*) **throw (MarmottesErreurs)**	indique si le senseur serait capable de contrôler le satellite dans l'*etat* fourni

implantation

Il n'y a ni attribut protégé ni attribut privé. Les méthodes protégées sont décrites dans la table : 113.

TAB. 113: SenseurLimbe : méthodes protégées

signature	description
SenseurLimbe ()	constructeur par défaut. Il est défini explicitement uniquement pour prévenir celui créé automatiquement par le compilateur et ne doit pas être utilisé.
void **ecartFrontiere** (const Etat& *etat*, double **ptrEcartFrontiere*, bool **ptrAmplitudeSignificative*) const **throw (CantorErreurs)**	pour les capteurs de limbe, on ne sait pas calculer l'écart angulaire entre la cible et la frontière. Cette méthode se contente donc de retourner $+1$ si le limbe est visible et -1 s'il ne l'est pas et d'indiquer par une valeur fausse dans la variable pointée par *ptrAmplitudeSignificative* que la valeur numérique est non significative.

13.38 classe SenseurOptique

description

Cette classe abstraite est l'interface d'accès aux senseurs optiques.

Cette classe implante la fonction **typeGeometrique**, elle implante une version générale de la fonction de vérification de la contrôlabilité (certaines classes dérivées s'en contentent, d'autres la spécialise), et introduit de nouvelles méthodes spécifiques aux senseurs optiques (**champDeVue**, **typeOptique**, **visible**, **typeOptique**).

interface publique

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

enum codeCible {codeInvalide,              codeSoleil,
                codeSoleilSansEclipse,     codeCorpsCentralSoleil,
                codeLune,                  codeLuneSansEclipse,
                codeCorpsCentral,          codeTerre,
                codeVitesseSolApparente,   codeNadir,
                codePolaire,               codeCanope,
                codeVitesse,               codeMoment,
                codeDevant,                codePosition,
                codePositionSansEclipse,   codeDirection,
                codeDirectionSansEclipse,  codeStation};
```

TAB. 114: SenseurOptique : méthodes publiques

signature	description
SenseurOptique (const string& *nom*, const RotDBL& *repere*, const VecDBL& *axeCalage*, double *precision*, codeCible *code*, const StationCible **ptrStation*, const VecDBL& *observe*, Parcelle* *ptrChampDeVue*, Parcelle* *ptrChampInhibitionSoleil*, Parcelle* *ptrChampInhibitionLune*, Parcelle* *ptrChampInhibitionCentral*, double *margeEclipseSoleil*, double *margeEclipseLune*, double *seuilPhaseLune*)	construit une instance à partir des données technologiques
SenseurOptique (const SenseurOptique& *s*)	constructeur par copie
SenseurOptique& **operator** = (const SenseurOptique& *s*)	affectation
~SenseurOptique ()	destructeur, libère la mémoire allouée aux champs de vue et d'inhibition
void **nouveauRepere** (const RotDBL& *nouveau*)	remplace le repère du senseur par le *nouveau*
const Parcelle* **champDeVue** () const	retourne un pointeur sur le champ de vue
typeGeom **typeGeometrique** () const	retourne le type de senseur géométrique (optique)
int **controlable** (const Etat& *etat*) **throw (MarmottesErreurs, CantorErreurs)**	indique si le senseur serait capable de contrôler le satellite dans l'*etat* fourni (par défaut, ceci correspond à la cible dans le champ de vue et non éclipsée et à l'absence d'inhibition, les classes dérivées peuvent bien sûr redéfinir ce comportement)
void **criteresControlabilite** (const Etat& *etat*, codeAstre **ptrInhibant*, codeAstre **ptrEclipsant*, double **ptrEcartFrontiere*, bool **ptrAmplitudeSignificative*) **throw (MarmottesErreurs, CantorErreurs)**	donne le détail des critères de contrôlabilité : corps inhibant, cors éclipsant, écart angulaire entre la cible et la frontière (positif à l'intérieur du champ de vue et négatif au dehors), et indicateur permettant de savoir si la valeur numérique de cet écart est significative (ce calcul ne pouvant pas être fait pour certains types de senseurs, typiquement les senseurs de limbe) ou si seul le signe est calculé
const Parcelle* **champInhibitionSoleil** () const	retourne un pointeur sur le champ d'inhibition associé au soleil
const Parcelle* **champInhibitionLune** () const	retourne un pointeur sur le champ d'inhibition associé à la lune
const Parcelle* **champInhibitionCentral** () const	retourne un pointeur sur le champ d'inhibition associé au corps central
double **margeEclipseSoleil** () const	retourne la marge à prendre sur les éclipses solaires
double **margeEclipseLune** () const	retourne la marge à prendre sur les éclipses lunaires
void **modifieCible** (const VecDBL& *cible*) **throw (MarmottesErreurs)**	mémorise la *cible* donnée par l'utilisateur

implantation

Les attributs protégés sont décrits sommairement dans la table 115.

TAB. 115: attributs protégés de la classe SenseurOptique

nom	type	description
visee_	VecDBL	point visé par l'optique ; peut être différent de la cible, par exemple quand on vise optiquement un point au sol mais que l'on est sensible à sa vitesse plutôt qu'à sa direction, comme dans le cas de la cible `codeVitesseSolApparente` .
ptrChampDeVue_	Parcelle*	pointeur sur le champ de vue
ptrChampInhibitionSoleil_	Parcelle*	pointeur sur le champ d'inhibition associé au soleil
ptrChampInhibitionLune_	Parcelle*	pointeur sur le champ d'inhibition associé à la lune
ptrChampInhibitionCentral_	Parcelle*	pointeur sur le champ d'inhibition associé au corps central
margeEclipseSoleil_	double	marge à prendre sur les éclipses solaires
margeEclipseLune_	double	marge à prendre sur les éclipses lunaires
seuilPhaseLune_	double	Seuil sur l'angle Soleil/Satellite/Lune au *dessous* duquel la lune devient gênante (0 signifie que la lune n'est jamais gênante, 180 signifie qu'elle est toujours gênante)
code_	codeCible	code de la cible inertielle
station_	StationCible	caractéristiques de la station cible, utilisées uniquement lorsque code_ vaut `codeStation`
utilisateur_	VecDBL	cible donnée par l'utilisateur
utilisateurInitialise_	bool	indique si la cible utilisateur a été initialisée
secteursFiltrables_	bool	indique s'il est possible d'accélérer les calculs en filtrant les secteurs de consigne avant d'entamer la résolution. Ceci n'est en effet possible ni pour les senseurs de limbe (la cible pouvant être hors du champ de vue, qui observe le limbe et pas le centre de l'astre), ni pour la cible `codeVitesseSolApparente` (la cible du modèle ne correspond pas à la direction de la cible optique, mais à la direction de son déplacement).

Les méthodes protégées sont décrites dans la table 116.

TAB. 116: SenseurOptique : méthodes protégées

signature	description
SenseurOptique ()	constructeur par défaut. Il est défini explicitement uniquement pour prévenir celui créé automatiquement par le compilateur et ne doit pas être utilisé.
void **reinitVitesseSolApparente** (const Etat& *etatPrecedent*, const RotVD1& *attitude*) **throw (MarmottesErreurs)**	réinitialise la cible spécialisée vitesse-sol-apparente, en tenant compte de l'*attitude* test et de ses dérivées
void **initialiseCible** (const Etat& *etat*) **throw (MarmottesErreurs)**	initialise la cible en repère inertiel
void **ecartFrontiere** (const Etat& *etat*, double **ptrEcartFrontiere*, bool **ptrAmplitudeSignificative*) const **throw (CantorErreurs)**	calcule l'écart angulaire entre la cible et la frontière (positif à l'intérieur du champ de vue et négatif au dehors), et un indicateur permettant de savoir si la valeur numérique de cet écart est significative (ce calcul ne pouvant pas être fait pour certains types de senseurs, typiquement les senseurs de limbe) où si seul le signe est calculé
void **filtreSecteurs**() **throw (CantorErreurs)**	méthode de filtrage des secteurs de consigne par le champ de vue, pour accélérer les calculs

13.39 classe SenseurVecteur

description

Cette classe implante les senseurs mesurant des angles entre vecteurs. De tels senseurs représentent par exemple les senseurs des satellites spinnés qui utilisent le mouvement du satellite sur lui-même pour produire leur mesure. La classe sert également de classe de base pour définir des pseudo-senseurs (SenseurCartesien, SenseurElevation).

interface publique

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

TAB. 117: SenseurVecteur : méthodes publiques

signature	description
SenseurVecteur (const string& *nom*, const RotDBL& *repere*, const VecDBL& *axeCalage*, double *precision*, codeCible *code*, const StationCible **ptrStation*, const VecDBL& *observe*, Parcelle **ptrChampDeVue*, Parcelle* *ptrChampInhibitionSoleil*, Parcelle* *ptrChampInhibitionLune*, Parcelle* *ptrChampInhibitionCentral*, double *margeEclipseSoleil*, double *margeEclipseLune*, double *seuilPhaseLune*, const VecDBL& *reference*) **throw (CantorErreurs)**	construit une instance à partir des données technologiques
SenseurVecteur (const SenseurVecteur& *s*)	constructeur par copie
SenseurVecteur& **operator** = (const SenseurVecteur& *s*)	affectation
~**SenseurVecteur** ()	destructeur, ne fait rien dans cette classe
void **respecterMesures** ()	force le senseur à respecter les unités de mesures dans ses sorties
void **convertirMesures** ()	force le senseur à convertir les unités de mesures dans ses sorties
Senseur* **copie** () const	opérateur de copie virtuel
void **nouveauRepere** (const RotDBL& *nouveau*)	remplace le repère du senseur par le *nouveau*
void **modeliseConsigne** (const Etat& *etat*, double *valeur*) **throw (CantorErreurs, MarmottesErreurs)**	modélise la consigne *valeur* dans l'*etat* fourni
double **mesure** (const Etat& *etat*) **throw (MarmottesErreurs)**	retourne la mesure que produirait le senseur dans l' *etat* fourni

implantation

Les attributs privés sont décrits sommairement dans la table 118, il n'y a pas d'attribut protégé.

TAB. 118: attributs privés de la classe SenseurVecteur

nom	type	description
reference_	VecDBL	vecteur de référence en repère satellite

Les méthodes protégées sont décrites dans la table 119.

TAB. 119: SenseurVecteur : méthodes protégées

signature	description
SenseurVecteur ()	constructeur par défaut. Il est défini explicitement uniquement pour prévenir celui créé automatiquement par le compilateur et ne doit pas être utilisé.

13.40 classe SpinAtt

description

Cette classe est un conteneur permettant de mémoriser un couple attitude et spin. Elle est utilisée par la classe ResolveurAttitude pour stocker toutes les solutions trouvées.

interface publique

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

TAB. 120: SpinAtt : méthodes publiques

signature	description
SpinAtt () **SpinAtt** (const RotDBL& *attitude*, const VecDBL& *spin*)	constructeur par défaut construit une instance à partir d'une *attitude* et d'un *spin*
SpinAtt (const SpinAtt& *sa*) SpinAtt& **operator** = (const SpinAtt& *s*) **~SpinAtt()**	constructeur par copie affectation destructeur
const RotDBL& **attitude** () const const VecDBL& **spin** () const	retourne le spin mémorisé retourne l'attitude mémorisée

exemple d'utilisation

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

void  Marmottes::attitude (double date,
                           const VecDBL& position, const VecDBL& vitesse,
                           double m1, double m2, double m3,
                           RotDBL *attitude, VecDBL *spin)
  throw (CantorErreurs, MarmottesErreurs)
{ // calcul d'une attitude donnée par trois consignes
```

```
  try
  {

    ...

    // récupération de la "meilleure" solution
    SpinAtt sol = solveur_.selection ();
    *attitude   = sol.attitude ();
    *spin       = sol.spin     ();

    ...
  }

  catch (...)
  {
    etatExtrapole_ = etatResolu_;
    throw;
  }
}
```

conseils d'utilisation spécifiques

Cette classe est un simple conteneur conservant des copies des éléments avec lesquels l'instance est construite, elle ne présente aucune difficulté particulière d'utilisation.

implantation

Les attributs privés sont décrits sommairement dans la table 121, il n'y a pas d'attribut protégé.

TAB. 121: attributs privés de la classe SpinAtt

nom	type	description
attitude_	RotDBL	attitude mémorisée
spin_	VecDBL	spin mémorisé

13.41 classe StationCible

description

Cette classe modélise une station sol telle qu'elle peut voir un satellite ; elle est utilisée en tant que cible de certains senseurs optiques.

interface publique

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

TAB. 122: StationCible : méthodes publiques

signature	description
StationCible ()	constructeur par défaut
StationCible (double *pression*, double *temperature*, double *hygrometrie*, double *longitude*, double *latitude*, double *altitude*, int *nbPtsMasque*, const double masqueAz [], const double masqueSi [])	construit une station à partir de ses coordonnées et des conditions atmosphériques associées
StationCible (const StationCible& *s*)	constructeur par copie
StationCible& **operator** = (const StationCible& *s*)	affectation
~StationCible ()	destructeur, libère la mémoire allouée pour le masque d'antenne
VecDBL **position** () const	retourne la position par rapport au repère terrestre
void **correctionTropo** (double *siteMesure*, double* *dSite*, double* *dDist*) const	calcule les corrections troposphériques en site et en distance pour le site mesuré
void **correctionTropoInverse** (double *siteTheorique*, double* *dSite*, double* *dDist*) const	calcule l'inverse des corrections troposphériques en site et en distance pour le site théorique
double **siteObservePt** (const VecDBL& *p*) const	retourne le site observé pour un vecteur regardant le point $\vec{p}$ en repère terrestre
double **siteTheoriquePt** (const VecDBL& *p*) const	retourne le site théorique pour un vecteur regardant le point $\vec{p}$ en repère terrestre
double **azimutPt** (const VecDBL& *p*) const	retourne l'azimut pour un vecteur regardant le point $\vec{p}$ en repère terrestre
double **masque** (double *azimut*) const	retourne le masque d'antenne (site minimal) dans l'azimut spécifié
int **visiblePt** (const VecDBL& *p*) const	indique si le point $\vec{p}$ est visible (c'est à dire s'il est au-dessus du masque d'antenne, en tenant compte de l'effet troposphérique)

exemple d'utilisation

```
#include "marmottes/StationCible.h"
...
void SenseurOptique::initialiseCible (const Etat& etat)
  throw (MarmottesErreurs)
```

```
{ // initialisation de la direction de la cible en repère inertiel

  switch (code_)
  {
    ...

    case codeStation :
      { // encapsulation du cas entre "{}" pour limiter la portée
        // des variables locales
        etat.normesLitigieuses ();

        RotDBL terreInert (VecDBL (0.0, 0.0, 1.0), etat.tempsSideral ());
        cible_ = terreInert (station_.position ()) - etat.position ();
        cible_.normalise ();

      }
      break;

    ...

  }

  // on fait la conversion en développement limité une fois pour toutes
  cibleVD1_ = VecDBLVD1 (cible_);

}

int SenseurOptique::visible (const Etat& etat, const VecDBL& u) const
{ if (ptrChampDeVue_)
  { if (code_ == codeStation)
    { VecDBL v = etat.attitude () (u);

      RotDBL inertTerre (VecDBL (0.0, 0.0, 1.0), -(etat.tempsSideral ()));

      return (ptrChampDeVue_->inclus (v)
              &&
              station_.visiblePt (inertTerre (etat.position ())));

    }
    else
    { VecDBL v = etat.attitude () (u);
      return ptrChampDeVue_->inclus (v);
    }
  }
  else
    return 0;
```

```
}
```

conseils d'utilisation spécifiques

La classe station est destinée principalement à modéliser des liaisons bord-sol par l'intermédiaire de pseudo-senseurs. La visibilité de la station dans le lobe d'antenne bord peut être modélisée par un champ de vue. La visibilité du satellite dans le lobe d'antenne sol est calculée en tenant compte de la forme ellipsoïdale de la terre et des effets troposphériques. Il faut donc prendre garde que la visibilité est limitée par le sol et par le bord de façon indépendante.

implantation

Les attributs privés sont décrits sommairement dans la table 123, il n'y a pas d'attribut protégé.

TAB. 123: attributs privés de la classe StationCible

nom	type	description
pression_	double	pression atmosphérique au sol
temperature_	double	température au sol
hygrometrie_	double	hygrométrie au sol
altitude_	double	altitude de la station au dessus de l'ellipsoïde terrestre
position_	VecDBL	position de la station en repère terrestre
nordEstNadir_	RotDBL	orientation du repère topocentrique local par rapport au repère terrestre
nbPtsMasque_	int	nombre de points du masque d'antenne
masqueAz_	double *	table des azimuts du masque d'antenne
masqueSi_	double *	table des sites du masque d'antenne

Les méthodes privées sont décrites dans la table 124.

TAB. 124: StationCible : méthodes privées

signature	description
void **initCoord** (double *longitude*, double *lagitude*, double *altitude*)	initialise la position de la station à partir de ses coordonnées sur l'ellipsoïde
void **initMasque** (int *nbPtsMasque*, const double masqueAz [], const double masqueSi [])	initialise le masque d'antenne

A Réordonnancement des senseurs

Pour résoudre l'attitude, Marmottes regroupe deux senseurs de même genre (cinématiques ou géométriques) et utilise le troisième senseur pour annuler une fonction (avec un seuil de convergence dépendant de la précision de ce troisième senseur).

Afin de permettre à l'utilisateur de savoir quel senseur est isolé parmi les trois senseurs de consigne, voici l'algorithme utilisé par Marmottes.

Soient s1, s2 et s3 les senseurs dans l'ordre utilisateur

Soient sa1, sa2 et sb les senseurs dans l'ordre de résolution

```
Si (s1 et s2 sont de même type)
   sa1 = s1
   sa2 = s2
   sb = s3
Sinon
   Si (s1 et s3 sont de même type)
      sa1 = s1
      sa2 = s3
      sb = s2
   Sinon
      sa1 = s2
      sa2 = s3
      sb = s1
   finsi
finsi
```

B exemple de fichier senseurs en francais

```
# Senseur solaire 1 (tangage)
SOLAIRE_1_TANGAGE
{ type                {diedre}
  cible               {soleil}
  precision           { 0.01 }

  # définition des axes senseurs en repère satellite
  repere { i { -1  0  0 } j { 0 0 1 } k { 0 1 0 }}

  # définition des axes particuliers en repère senseur
  axe_calage      { 0 1 0 }
  axe_sensible    { 0 0 1 }
  reference_zero  { 1 0 0 }


  # définition du champ de vue (vecteurs notés angulairement)
  champ_de_vue
  { { # dièdre d'axe j senseur (ouverture +/- 32 degrés)
      { cone { axe { 0.0  58.0 } angle { 90 }}}
      inter
      { cone { axe { 0.0 -58.0 } angle { 90 }}}
    }
    inter
    { # dièdre d'axe k senseur (ouverture +/- 32 degrés)
      { cone { axe {  58.0 0.0 } angle { 90 }}}
      inter
      { cone { axe { -58.0 0.0 } angle { 90 }}}
    }
  }
}

# Senseur solaire 1 (lacet)
SOLAIRE_1_YAW
{ => { SOLAIRE_1_TANGAGE}

  # seuls les axes de mesure diffèrent entre SOLAIRE_1_TANGAGE SOLAIRE_1_YAW
  axe_calage      { 0 0 1 }
  axe_sensible    { 0 1 0 }
  reference_zero  { 1 0 0 }

}
```

```
IRES_ROLL
{ type           { limbe }
  precision      { 0.2 }

  # le repère IRES s'obtient en tournant le repère satellite
  # de -0.4702 degrés autour de l'axe Ysat
  repere         { axe { 0 1 0 } angle { -0.4702} }

  axe_sensible   { -1 0 0 }
  reference_zero {  0 0 1 }

  # l'élément de détection de l'IRES est un bolomètre carré de 1.3 degrés
  # de largeur tourné de 45 degrés autour de Z : c'est un double dièdre
  bolometre_fictif { { { cone { axe {  45 0.65 } angle { 90 } } }
                       inter
                       { cone { axe { 225 0.65 } angle { 90 } } }
                     }
                     inter
                     { { cone { axe { 135 0.65 } angle { 90 } } }
                       inter
                       { cone { axe { 315 0.65 } angle { 90 } } }
                     }
                   }

  # pour définir un scan, on place un bolomètre au milieu du scan
  # on le déplace de -1/2 scan, puis on le traîne le long du scan
  # l'angle total vaut 8 degrés en champ large, 5.3 degrés en champ étroit
  centre_scan_1 { rotation { axe { 0 1 0 } angle { -6.2 } }
                  de       { axe { 1 0 0 } angle {  6.2 } }
                }
  centre_scan_2 { rotation { axe { 0 1 0 } angle { -6.2 } }
                  de       { axe { 1 0 0 } angle { -6.2 } }
                }
  centre_scan_3 { rotation { axe { 0 1 0 } angle {  6.2 } }
                  de       { axe { 1 0 0 } angle {  6.2 } }
                }
  centre_scan_4 { rotation { axe { 0 1 0 } angle {  6.2 } }
                  de       { axe { 1 0 0 } angle { -6.2 } }
                }
  scan          { axe { 0 1 0 } angle {  8.00 } }
  demi_scan     { axe { 0 1 0 } angle { -4.00 } }
```

```
# scans élémentaires du champ de vue
scan_1
{ balayage { => { IRES_ROLL.scan } }
  de       { rotation { => { IRES_ROLL.demi_scan } }
             de       { rotation { => { IRES_ROLL.centre_scan_1 } }
                        de       { => { IRES_ROLL.bolometre_fictif } }
                      }
           }
}

scan_2
{ balayage { => { IRES_ROLL.scan } }
  de       { rotation { => { IRES_ROLL.demi_scan } }
             de       { rotation { => { IRES_ROLL.centre_scan_2 } }
                        de       { => { IRES_ROLL.bolometre_fictif } }
                      }
           }
}

scan_3
{ balayage { => { IRES_ROLL.scan } }
  de       { rotation { => { IRES_ROLL.demi_scan } }
             de       { rotation { => { IRES_ROLL.centre_scan_3 } }
                        de       { => { IRES_ROLL.bolometre_fictif } }
                      }
           }
}

scan_4
{ balayage { => { IRES_ROLL.scan } }
  de       { rotation { => { IRES_ROLL.demi_scan } }
             de       { rotation { => { IRES_ROLL.centre_scan_4 } }
                        de       { => { IRES_ROLL.bolometre_fictif } }
                      }
           }
}
```

```
  champ_de_vue
  { { { => { IRES_ROLL.scan_1 } } et { => { IRES_ROLL.scan_2 } } }
    ou
    { { => { IRES_ROLL.scan_3 } } et { => { IRES_ROLL.scan_4 } } }
  }

  champ_d_inhibition_soleil
  { { marge { 3.0 }
      sur   { { { => { IRES_ROLL.scan_1 } }
                union
                { => { IRES_ROLL.scan_2 } }
              }
              union
              { { => { IRES_ROLL.scan_3 } }
                union
                { => { IRES_ROLL.scan_4 } }
              }
            }
    }
    sauf
    { cone { axe { 0 0 1 } angle { 8.2 } } }
  }

  champ_d_inhibition_lune { => { IRES_ROLL.champ_d_inhibition_soleil } }

}

IRES_PITCH
{ axe_sensible { 0 -1 0 }

  champ_de_vue
  { { { => { IRES_ROLL.scan_1 } } et { => { IRES_ROLL.scan_3 } } }
    ou
    { { => { IRES_ROLL.scan_2 } } et { => { IRES_ROLL.scan_4 } } }
  }

  => {IRES_ROLL}

}
```

```
AEF_Ascension   { type      { ascension_droite }
                  precision { 0.001 }
                  repere    { i { 1 0 0 } j { 0 1 0 } k { 0 0 1 } }
                  observe   { 0 0 1 }
                }

AEF_Declination { => {AEF_Ascension} type { declinaison }}

COMMUNS-PSEUDOS
{ precision { 0.0001 }
  repere    { 1 0 0 0 } # quaternion identité
}

ALPHA_X { type { ascension_droite } observe { 1 0 0 } => {COMMUNS-PSEUDOS}}
DELTA_X { type { declinaison }      observe { 1 0 0 } => {COMMUNS-PSEUDOS}}
ALPHA_Y { type { ascension_droite } observe { 0 1 0 } => {COMMUNS-PSEUDOS}}
DELTA_Y { type { declinaison }      observe { 0 1 0 } => {COMMUNS-PSEUDOS}}
ALPHA_Z { type { ascension_droite } observe { 0 0 1 } => {COMMUNS-PSEUDOS}}
DELTA_Z { type { declinaison }      observe { 0 0 1 } => {COMMUNS-PSEUDOS}}

GYRO_X  { type { cinematique } axe_sensible { 1 0 0 } => {COMMUNS-PSEUDOS}}
GYRO_Y  { type { cinematique } axe_sensible { 0 1 0 } => {COMMUNS-PSEUDOS}}
GYRO_Z  { type { cinematique } axe_sensible { 0 0 1 } => {COMMUNS-PSEUDOS}}
```

C exemple de fichier senseurs en anglais

```
# Pitch sun sensor 1
SUN_1_PITCH
{ type                {dihedral}
  target              {sun}
  accuracy            { 0.01 }

  # definition of sensor axis in satellite frame
  frame { i { -1  0  0 } j { 0 0 1 } k { 0 1 0 }}

  # definition of special vectors in sensor frame
  wedging_axis      { 0 1 0 }
  sensitive_axis    { 0 0 1 }
  zero_reference    { 1 0 0 }


  # field of view definition (vectors are described angularly)
  field of view
  { { # j sensor axis dihedra (opening +/- 32 degrees)
      { cone { axis { 0.0  58.0 } angle { 90 }}}
      inter
      { cone { axis { 0.0 -58.0 } angle { 90 }}}
    }
    inter
    { # k sensor axis dihedra (opening +/- 32 degrees)
      { cone { axis {  58.0 0.0 } angle { 90 }}}
      inter
      { cone { axis { -58.0 0.0 } angle { 90 }}}
    }
  }
}

# Yaw sun sensor 1
SUN_1_YAW
{ => { SUN_1_PITCH}

  # only measurements axis differ from SUN_1_PITCH
  wedging_axis      { 0 0 1 }
  sensitive_axis    { 0 1 0 }
  zero_reference    { 1 0 0 }

}
```

```
IRES_ROLL
{ type           { limb }
  accuracy       { 0.2 }

  # IRES frame is satellite frame rotated
  # -0.4702 degrees around Ysat
  frame            { axis { 0 1 0 } angle { -0.4702} }

  sensitive_axis   { -1 0 0 }
  zero_reference   {  0 0 1 }

  # IRES detector is a 1.3 degrees square bolometer
  # rotated by 45 degrees around Z : it is a double-dihedra
  fictious_bolometer { { { cone { axis {  45 0.65 } angle { 90 } } }
                         inter
                         { cone { axis { 225 0.65 } angle { 90 } } }
                       }
                       inter
                       { { cone { axis { 135 0.65 } angle { 90 } } }
                         inter
                         { cone { axis { 315 0.65 } angle { 90 } } }
                       }
                     }

  # in order to define a scan, one places the bolometer at the middle
  # of the scan, then shift it -1/2 scan, and then one spread it over
  # all scan long. total angle is 8 degrees in wide scan mode and 5.3
  # degrees in narrow scan mode
  center_scan_1 { rotation { axis { 0 1 0 } angle { -6.2 } }
                  of       { axis { 1 0 0 } angle {  6.2 } }
                }
  center_scan_2 { rotation { axis { 0 1 0 } angle { -6.2 } }
                  of       { axis { 1 0 0 } angle { -6.2 } }
                }
  center_scan_3 { rotation { axis { 0 1 0 } angle {  6.2 } }
                  of       { axis { 1 0 0 } angle {  6.2 } }
                }
  center_scan_4 { rotation { axis { 0 1 0 } angle {  6.2 } }
                  of       { axis { 1 0 0 } angle { -6.2 } }
                }
  scan           { axis { 0 1 0 } angle {  8.00 } }
  half_scan      { axis { 0 1 0 } angle { -4.00 } }
```

```
# elementary scans
scan_1
{ spread { => { IRES_ROLL.scan } }
  of       { rotation { => { IRES_ROLL.half_scan } }
             of       { rotation { => { IRES_ROLL.center_scan_1 } }
                        of       { => { IRES_ROLL.fictious_bolometer } }
                      }
           }
}

scan_2
{ spread { => { IRES_ROLL.scan } }
  of       { rotation { => { IRES_ROLL.half_scan } }
             of       { rotation { => { IRES_ROLL.center_scan_2 } }
                        of       { => { IRES_ROLL.fictious_bolometer } }
                      }
           }
}

scan_3
{ spread { => { IRES_ROLL.scan } }
  of       { rotation { => { IRES_ROLL.half_scan } }
             of       { rotation { => { IRES_ROLL.center_scan_3 } }
                        of       { => { IRES_ROLL.fictious_bolometer } }
                      }
           }
}

scan_4
{ spread { => { IRES_ROLL.scan } }
  of       { rotation { => { IRES_ROLL.half_scan } }
             of       { rotation { => { IRES_ROLL.center_scan_4 } }
                        of       { => { IRES_ROLL.fictious_bolometer } }
                      }
           }
}
```

```
  field_of_view
  { { { => { IRES_ROLL.scan_1 } } and { => { IRES_ROLL.scan_2 } } }
    or
    { { => { IRES_ROLL.scan_3 } } and { => { IRES_ROLL.scan_4 } } }
  }

  sun_field_of_inhibition
  { { margin { 3.0 }
      upon   { { { => { IRES_ROLL.scan_1 } }
                   union
                   { => { IRES_ROLL.scan_2 } }
               }
               union
               { { => { IRES_ROLL.scan_3 } }
                 union
                 { => { IRES_ROLL.scan_4 } }
               }
             }
     }
     except
     { cone { axis { 0 0 1 } angle { 8.2 } } }
  }

  moon_field_of_inhibition { => { IRES_ROLL.sun_field_of_inhibition } }

}

IRES_PITCH
{ sensitive_axis { 0 -1 0 }

  field_of_view
  { { { => { IRES_ROLL.scan_1 } } and { => { IRES_ROLL.scan_3 } } }
    or
    { { => { IRES_ROLL.scan_2 } } and { => { IRES_ROLL.scan_4 } } }
  }

  => {IRES_ROLL}

}
```

```
AEF_Ascension   { type      { right_ascension }
                  accuracy  { 0.001 }
                  frame     { i { 1 0 0 } j { 0 1 0 } k { 0 0 1 } }
                  observed  { 0 0 1 }
                }

AEF_Declination { => {AEF_Ascension} type { declination }}

PSEUDOS-COMMONS
{ accuracy { 0.0001 }
  frame    { 1 0 0 0 } # identity quaternion
}

ALPHA_X { type { right_ascension } observed { 1 0 0 } => {PSEUDOS-COMMONS}}
DELTA_X { type { declination }     observed { 1 0 0 } => {PSEUDOS-COMMONS}}
ALPHA_Y { type { right_ascension } observed { 0 1 0 } => {PSEUDOS-COMMONS}}
DELTA_Y { type { declination }     observed { 0 1 0 } => {PSEUDOS-COMMONS}}
ALPHA_Z { type { right_ascension } observed { 0 0 1 } => {PSEUDOS-COMMONS}}
DELTA_Z { type { declination }     observed { 0 0 1 } => {PSEUDOS-COMMONS}}

GYRO_X  { type { kinematic } sensitive_axis { 1 0 0 } => {PSEUDOS-COMMONS}}
GYRO_Y  { type { kinematic } sensitive_axis { 0 1 0 } => {PSEUDOS-COMMONS}}
GYRO_Z  { type { kinematic } sensitive_axis { 0 0 1 } => {PSEUDOS-COMMONS}}
```

D Lexique Français-Anglais des mots clés du fichier Senseurs

TAB. 125: Mots-clés du fichier senseurs

Mots clés en Français	Mots clés en Anglais
altitude	altitude
angle	angle
angle_3dB	angle_3dB
angle_3dB_x	angle_3dB_x
angle_3dB_y	angle_3dB_y
angle_zero	zero_angle
axe	axis
axe_calage	wedging_axis
axe_sensible	sensitive_axis
balayage	spread
champ_de_vue	field_of_view
champ_d_inhibition_corps_central	central_body_field_of_inhibition
champ_d_inhibition_lune	moon_field_of_inhibition
champ_d_inhibition_soleil	sun_field_of_inhibition
cible	target
cone	cone
de	of
echantillon	sample
genre	kind
hygrometrie	hygrometry
i	i
inter	inter
j	j
k	k
longitude	longitude
latitude	latitude
marge_eclipse_lune [16]	moon_eclipse_margin
marge_eclipse_soleil [17]	sun_eclipse_margin
masque	mask
maximum	maximum
normale_reference	normal_reference

à suivre ...

[16] Pour plus de précisions : se reporter au § 5.2.9
[17] Pour plus de précisions : se reporter au § 5.2.9

TAB. 125: Mots-clés du fichier senseurs (suite)

Mots clés en Français	Mots clés en Anglais
observe	observed
origine	origin
precision	accuracy
pression	pressure
reference	reference
reference_zero	zero_reference
repere	frame
rotation	rotation
sauf	except
seuil_phase_lune [18]	moon_phase_threshold
temperature	temperature
type	type
union	union
v_base	v_base
v_base_1	v_base_1
v_base_2	v_base_2
v_image	v_image
v_image_1	v_image_1
v_image_2	v_image_2

TAB. 126: Types de senseurs reconnus

Mots clés en Français	Mots clés en Anglais
ascension_droite	right_ascension
cardan	cardan
cartesien	cartesian
cinematique	kinematic
declinaison	declination
diedre	dihedral
gain_echantillonne_1D	sampled_1D_gain
gain_gauss	gauss_gain
gain_sinus_cardinal_2	square_cardinal_sine_gain
	à suivre ...

[18] Pour plus de précisions : se reporter au § 5.2.9

TAB. 126: Types de senseurs reconnus (suite)

Mots clés en Français	Mots clés en Anglais
gain_sinus_cardinal_xy	xy_cardinal_sine_gain
gyro_integrateur	integrating_gyro
limbe	limb
plan_vecteur	plane_vector
terre	earth
vecteur	vector

TAB. 127: Types de senseurs cardans

Mots clés en Français	Mots clés en Anglais
LRT-lacet	YRP-yaw
LRT-roulis	YRP-roll
LRT-tangage	YRP-pitch
LTR-lacet	YPR-yaw
LTR-roulis	YPR-roll
LTR-tangage	YPR-pitch
RLT-lacet	RYP-yaw
RLT-roulis	RYP-roll
RLT-tangage	RYP-pitch
RTL-lacet	RPY-yaw
RTL-roulis	RPY-roll
RTL-tangage	RPY-pitch
TLR-lacet	PYR-yaw
TLR-roulis	PYR-roll
TLR-tangage	PYR-pitch
TRL-lacet	PRY-yaw
TRL-roulis	PRY-roll
TRL-tangage	PRY-pitch

TAB. 128: Repères de référence

Mots clés en Français	Mots clés en Anglais
geocentrique	geocentric
inertiel	inertial
orbital-TNW	TNW-orbital
orbital-QSW	QSW-orbital
topocentrique	topocentric
utilisateur	user

TAB. 129: Astres et cibles connus

Mots clés en Français	Mots clés en Anglais
canopus	canopus
canopus-sans-eclipse	eclipse-free-canopus
corps-central	central-body
corps-central-soleil	central-body-sun
devant	along-track
direction	direction
direction-sans-eclipse	eclipse-free-direction
nadir	nadir
lune	moon
lune-sans-eclipse	eclipse-free-moon
moment	momentum
polaris	polaris
polaris-sans-eclipse	eclipse-free-polaris
position	position
position-sans-eclipse	eclipse-free-position
pseudo-soleil	pseudo-sun
soleil	sun
soleil-sans-eclipse	eclipse-free-sun
station	station
terre-soleil	earth-sun
vitesse	velocity
vitesse-sol-apparente	apparent-ground-velocity

E Lexique Anglais-Français des mots clés du fichier Senseurs

TAB. 130: Mots-clés du fichier senseurs

Mots clés en Anglais	Mots clés en Français
accuracy	precision
altitude	altitude
angle	angle
angle_3dB	angle_3dB
angle_3dB_x	angle_3dB_x
angle_3dB_y	angle_3dB_y
axis	axe
central_body_field_of_inhibition	champ_d_inhibition_corps_central
cone	cone
except	sauf
field_of_view	champ_de_vue
frame	repere
hygrometry	hygrometrie
i	i
inter	inter
j	j
k	k
kind	genre
longitude	longitude
latitude	latitude
mask	masque
maximum	maximum
moon_eclipse_margin	marge_eclipse_lune [19]
moon_field_of_inhibition	champ_d_inhibition_lune
moon_phase_threshold	seuil_phase_lune [20]
normal_reference	normale_reference
observed	observe
of	de
origin	origine
pressure	pression
reference	reference
	à suivre ...

[19] Pour plus de précisions : se reporter au § 5.2.9
[20] Pour plus de précisions : se reporter au § 5.2.9

TAB. 130: Mots-clés du fichier senseurs (suite)

Mots clés en Anglais	Mots clés en Français
rotation	rotation
sample	echantillon
sensitive_axis	axe_sensible
spread	balayage
sun_eclipse_margin	marge_eclipse_soleil [21]
sun_field_of_inhibition	champ_d_inhibition_soleil
target	cible
temperature	temperature
type	type
union	union
v_base	v_base
v_base_1	v_base_1
v_base_2	v_base_2
v_image	v_image
v_image_1	v_image_1
v_image_2	v_image_2
wedging_axis	axe_calage
zero_angle	angle_zero
zero_reference	reference_zero

TAB. 131: Types de senseurs reconnus

Mots clés en Anglais	Mots clés en Français
cardan	cardan
cartesian	cartesien
declination	declinaison
dihedral	diedre
earth	terre
gauss_gain	gain_gauss
integrating_gyro	gyro_integrateur
limb	limbe
kinematic	cinematique
	à suivre ...

[21] Pour plus de précisions : se reporter au § 5.2.9

TAB. 131: Types de senseurs reconnus (suite)

Mots clés en Anglais	Mots clés en Français
plane_vector	plan_vecteur
right_ascension	ascension_droite
sampled_1D_gain	gain_echantillonne_1D
square_cardinal_sine_gain	gain_sinus_cardinal_2
vector	vecteur
xy_cardinal_sine_gain	gain_sinus_cardinal_xy

TAB. 132: Types de senseurs cardans

Mots clés en Anglais	Mots clés en Français
PRY-pitch	TRL-tangage
PRY-roll	TRL-roulis
PRY-yaw	TRL-lacet
PYR-pitch	TLR-tangage
PYR-roll	TLR-roulis
PYR-yaw	TLR-lacet
RPY-pitch	RTL-tangage
RPY-roll	RTL-roulis
RPY-yaw	RTL-lacet
RYP-pitch	RLT-tangage
RYP-roll	RLT-roulis
RYP-yaw	RLT-lacet
YPR-pitch	LTR-tangage
YPR-roll	LTR-roulis
YPR-yaw	LTR-lacet
YRP-pitch	LRT-tangage
YRP-roll	LRT-roulis
YRP-yaw	LRT-lacet

TAB. 133: Repères de référence

Mots clés en Anglais	Mots clés en Français
geocentric	geocentrique
inertial	inertiel
QSW-orbital	orbital-QSW
TNW-orbital	orbital-TNW
topocentric	topocentrique
user	utilisateur

TAB. 134: Astres et cibles connus

Mots clés en Anglais	Mots clés en Français
along-track	devant
apparent-ground-velocity	vitesse-sol-apparente
canopus	canopus
central-body	corps-central
central-body-sun	corps-central-soleil
direction	direction
earth-sun	terre-soleil
eclipse-free-canopus	canopus-sans-eclipse
eclipse-free-direction	direction-sans-eclipse
eclipse-free-moon	lune-sans-eclipse
eclipse-free-polaris	polaris-sans-eclipse
eclipse-free-position	position-sans-eclipse
eclipse-free-sun	soleil-sans-eclipse
momentum	moment
moon	lune
nadir	nadir
polaris	polaris
position	position
pseudo-sun	pseudo-soleil
station	station
sun	soleil
velocity	vitesse

F Définitions des repères utilisés

Marmottes utilise principalement trois grandes catégorie de repère.

Le repère inertiel est le repère dont l'origine est au centre du corps attracteur et les axes sont fixes dans l'espace (gamma50 CNES, J2000, ...). Toutes les positions et vitesses passées en argument à Marmottes sont exprimées dans ce repère. L'attitude est la rotation qui, appliquée aux coordonnées d'un vecteur exprimé dans ce repère, donne les coordonnées, de ce même vecteur, exprimées en repère satellite.

Le repère satellite est le repère dont on cherche à déterminer l'orientation par rapport au repère inertiel. Il est défini par le constructeur et correspond au coprs du satellite.

Les repères senseurs sont les repères propres aux équipements de mesure de l'attitude (typiquement la tête optique des senseurs ou le boîtier des gyromètres). Ce repère est calé, par construction, par rapport au repère satellite. C'est ce repère qui doit être défini pour chaque senseur dans le fichier des senseurs. Tous les vecteurs de définition des axes de visée, de mesure, de champ de vue des senseurs sont exprimés dans ce repère dans le fichier senseurs.

Outre ces repères généraux, les capteurs de Cardan utilisent des repères spécifiques. Les capteurs d'angles de Cardan mesurent les angles de rotation successives permettant de passer d'un repère de référence au repère satellite. Plusieurs repère de référence sont prédéfinis et peuvent être spécifiés dans le fichier senseurs. Ces repères sont définis de la façon suivante, par rapport au repère inertiel (dans ces définitions, $\vec{P}$ est le vecteur position du satellite, compté du centre du corps attracteur vers le satellite et $\vec{V}$ est le vecteur vitesse du satellite).

repère géocentrique
: Ce repère dépend de la position du satellite et tourne à la fréquence orbitale.
Ce repère est défini par :
- $\vec{Z}$ est dirigé du satellite vers le centre du corps attracteur ($\vec{Z} = -\vec{P}/||\vec{P}||$),
- $\vec{Y}$ est porté par l'opposé du moment cinétique ($\vec{Y} = -\vec{P} \wedge \vec{V}/||\vec{P} \wedge \vec{V}||$),
- $\vec{X}$ complète le trièdre ($\vec{X} = \vec{Y} \wedge \vec{Z}$).

repère QSW
: Ce repère dépend de la position du satellite et tourne à la fréquence orbitale.
Ce repère est défini par :
- $\vec{X}$ pointe vers l'opposé du centre du corps attracteur ($\vec{X} = -\vec{P} \wedge ||\vec{P}||$),
- $\vec{Z}$ est porté par le moment orbital ($\vec{Z} = \vec{P} \wedge \vec{V}/||\vec{P} \wedge \vec{V}||$),
- $\vec{Y}$ complète le trièdre ($\vec{Y} = \vec{Z} \wedge \vec{X}$).

repère topocentrique
: Ce repère dépend de la position du satellite et tourne à la fréquence orbitale.
Ce repère est défini par :
- $\vec{Z}$ pointe vers le centre du corps attracteur ($\vec{Z} = -\vec{P}/||\vec{P}||$),
- $\vec{Y}$ pointe vers l'Est, ses coordonnées sont $(-P_y/\sqrt{P_x^2 + P_y^2}, P_x/\sqrt{P_x^2 + P_y^2}, 0)$ en repère inertiel,
- $\vec{X}$ complète le triédre ($\vec{X} = \vec{Y} \wedge \vec{Z}$).

repère inertiel
: Ce repère ne dépend de rien et est fixe.

Ce repère est le repère de définition. Ses axes sont donc les axes canoniques ($\vec{X}$ (1,0,0),$\vec{Y}$ (0,1,0),$\vec{Z}$ (0,0,1)).

repère TNW

Ce repère dépend de la position du satellite et tourne à la fréquence orbitale.
Ce repère est défini par :
- $\vec{X}$ est porté par la vitesse ($\vec{X} = \vec{V}/||\vec{V}||$),
- $\vec{Z}$ est porté par le moment orbital ($\vec{Z} = \vec{P} \wedge \vec{V}/||\vec{P} \wedge \vec{V}||$),
- $\vec{Y}$ complète le triédre ($\vec{Y} = \vec{Z} \wedge \vec{X}$).

repère utilisateur

Ce repère est entiérement paramétré par l'utilisateur à l'aide de la fonction **MarmottesModifieReference**. Il s'agit typiquement de l'attitude retournée par un appel préalable à **MarmottesAttitude**. Ceci permet alors de considérer les mesures des senseurs de Cardan comme les *écarts* (ou les erreurs de pilotage) par rapport à cette attitude de référence.