LISACode

version 1.4

Un simulateur scientifique de LISA

Guide développeur (description du fonctionnement de LISACode)

LISA France
Antoine Petiteau

May 15, 2008

Contents

1 Introduction

Ce document a pour objectif de décrire le fonctionnement du simulateur scientifique LISACode de façon à guider les utilisateurs et les développeur de LISACode.

2 Généralités

LISACode est un simulateur scientifique du détecteur d'ondes gravitationnelles LISA. Ses objectifs s'étendent de la simulation rapide pour l'analyse de données réaliste à la simulation technologique allégée. Cette partie présente de manière générale LISACode en exposant ses principes de base puis sa structure et enfin son implémentation.

2.1 Principe de base

Le concept principal qui a guidé le développement de LISACode est un subtil compromis entre la modélisation réaliste de LISA et l'efficacité de la génération de signaux réalistes pour l'analyse de données.

2.1.1 Un simulateur réaliste du détecteur LISA ...

Ce simulateur doit être suffisamment proche de l'instrument pour prendre en compte le maximum d'effets intervenants dans les signaux de mesures. Mais LISACode n'est pas un simulateur technologique car il ne simule pas le détecteur dans le détail. Il utilise seulement les fonctions de réponse des principaux composants. D'autre part, la définition de la mission n'est pas finalisée et le design technologique du détecteur est en constante évolution. La modélisation utilisée dans LISACode est donc basée sur une description de LISA intègrant tous les principes de base de l'instrument. L'implémentation de cette modélisation est faite de telle sorte qu'il est facile de la modifier pour permettre au simulateur d'évoluer conformément au design technologique de la mission, comme on le verra par la suite.

LISACode simule LISA en suivant une évolution temporelle comme lors du déroulement de la mission. Dans une simulation cette évolution est discrète, c'est-à-dire que le temps est échantillonné selon un pas de temps fixe Δt et les valeurs associées à chaque processus sont calculées pour un temps t_i puis pour un temps $t_{i+i} = t_i + \Delta t$ et ainsi de suite.

Dans LISA, il existe deux types de processus. D'une part il y a les processus continus qui sont les ondes gravitationnelles, la circulation des faisceaux laser et par conséquent tous les bruits induits sur ces faisceaux : bruit laser, bruit de masse inertielle, bruit de chemin optique et *shot noise*. D'autre part il y a les processus échantillonnés qui sont associés aux mesures du phasemètre et à l'application de *TDI*. C'est en s'appuyant sur cette constatation que la gestion de l'évolution temporelle dans LISACode a été mise en place. En effet, il y a deux pas de temps dans LISACode : un pas de temps physique

$\Delta t_{physique}$ qui permet de modéliser au mieux les processus continus et un pas de temps de mesure Δt_{mesure} pour les processus échantillonnés. Il faut bien voir que ce second pas de temps a une existence réelle dans LISA puisqu'il correspond au pas de temps avec lequel les phasemètres échantillonnent les signaux de mesures et c'est donc celui avec lequel les données scientifiques reçues sur Terre sont échantillonnées. Dans LISACode, le fonctionnement est similaire à celui de LISA : pour chaque pas de temps de mesure, les signaux du phasemètre sont calculés à partir d'une modélisation des processus physiques réalisée au pas de temps physique, et sont ensuite enregistrés dans des fichiers de sorties. Ainsi la méthode *TDI* s'applique de la même manière sur les signaux des phasemètres de LISA que sur ceux de LISACode. Le pas de temps principal qui régit l'évolution temporelle dans LISACode est donc le pas de temps de mesure Δt_{mesure}. Pour modéliser au mieux les processus physiques, le pas de temps physique doit être très petit de manière à rapprocher la description discrète du cas continu.

Il faut bien voir que l'ensemble photodiode-phasemètre est l'élément central de LISACode, puisque c'est lui qui effectue la transition entre les deux types de processus. En effet, la photodiode réalise une mesure en continue de l'interférence entre les faisceaux laser que le phasemètre analyse pour fournir un signal de mesure échantillonné. Cet ensemble clé dans LISA est donc l'élément central de LISACode. Il constitue, avec la distinction des pas de temps et l'utilisation d'un pas de temps physique, les bases d'une simulation réaliste.

2.1.2 ... efficace dans la génération de données grâce à sa flexibilté

En plus de décrire le détecteur de manière réaliste, LISACode doit pouvoir générer rapidement des flux de données notamment pour l'analyse de données. Le problème est que ces deux objectifs ne sont pas compatibles. En effet, pour qu'une simulation soit réaliste et modélise finement l'instrument, le pas de temps physique doit être très petit, mais plus ce pas de temps est petit, plus le temps de calcul nécessaire à la simulation est important. La solution de ce problème réside en partie dans le compromis à faire sur le choix des pas de temps mais surtout dans la grande flexibilité du simulateur.

En effet, LISACode est basé sur une structure modulaire qui s'adapte au type de simulation que l'utilisateur souhaite faire. Cela lui donne la potentialité de faire un certain nombre d'approximations et de ne construire que les éléments nécessaires à la simulation. Pour une simulation précise et réaliste, dont l'objectif serait, par exemple, l'étude de l'élimination du bruit laser par *TDI* et l'impact de cette méthode sur les bruits et les ondes gravitationnelles, l'ensemble des éléments de LISACode est créé et le pas de temps est petit. A l'opposé, pour une simulation rapide des réponses *TDI* à quelques ondes gravitationnelles, seuls les éléments nécessaires aux calculs des ondes et à l'application sont utilisés. Cette flexibilité, dont on détaillera l'implémentation dans la prochaine partie (cf. sous-section 2.2), donne à LISACode la possibilité de réaliser des simulations très variées, en des temps de calculs tout à fait raisonnables et avec un réalisme adapté au besoin de l'étude à mener.

2.2 Structure

LISACode est basé sur une structure flexible organisée autour du phasemètre. Cette organisation est exposée dans cette partie et on montrera comment elle permet de reproduire le fonctionnement de LISA.

La structure de LISACode présentée sur la figure 1, est organisée en modules de différents types : les modules principaux (blocs en trait plein rouge) et les modules d'interface (blocs en tirets verts). Ces modules interagissent entre eux, pour créer, en sortie du simulateur, des séquences temporelles qui correspondent aux flux de données, c'est-à-dire les signaux de mesure des phasemètres et les signaux obtenus après un prétraitement par la méthode *TDI*, que fournira LISA.

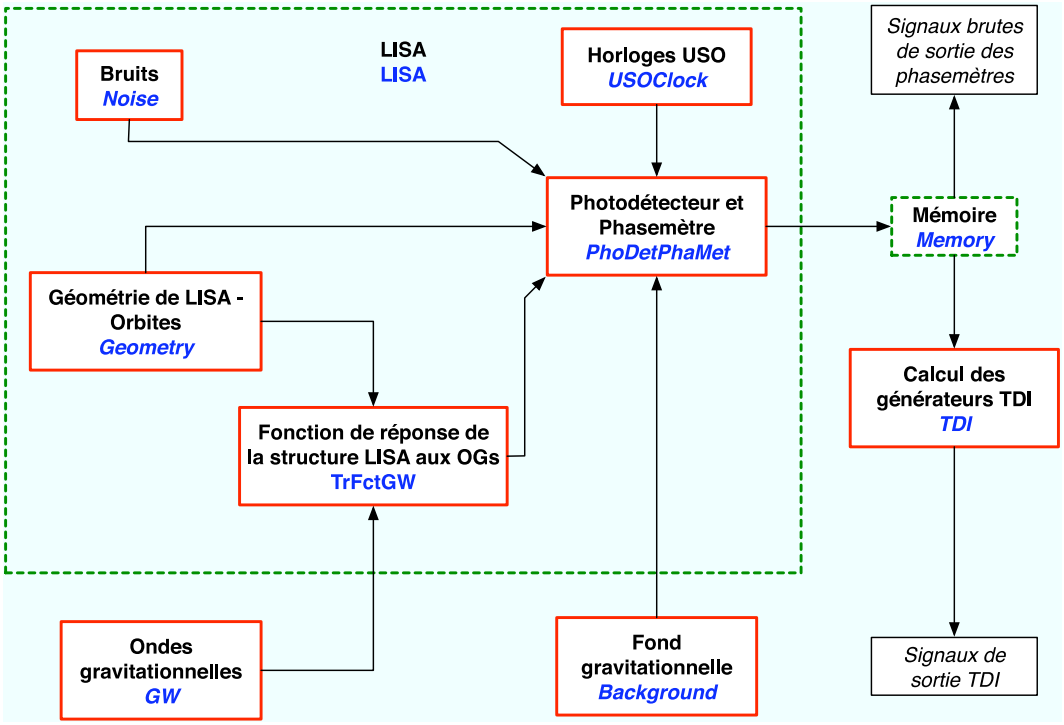

Figure 1: Structure du simulateur LISACode. Les boites en trait plein rouge représentent les modules principaux et les boites en tirets verts les modules d'interface. Le nom générique des modules sont écrits en bleu italique .

Les 7 modules principaux qui modélisent les différents types de phénomènes participants aux signaux de mesures, sont :

- *GW* : Modélisation des ondes gravitationnelles,
- *Background* : Modélisation des fonds d'ondes gravitationnelles,
- *TrFctGW* : Fonction de réponse des bras aux ondes gravitationnelles,

- *Geometry* : Modélisation des orbites des satellites et des temps de parcours,
- *Noise* : Modélisation des bruits,
- *USOClock* : Modélisation des horloges ultra-stables (*USO*),
- *PhoDetPhaMet* : Gestion des mémoires pour les sorties de chaque satellite,
- *TDI* : Application des générateurs TDI.

Ces modules principaux sont gérer par 2 modules d'interface qui sont :

- *LISA* : Organisation des éléments composant le détecteur que sont les bruits, les orbites, le calcul des fonctions de réponse des bras, les *USO* et les phasemètres ,
- *Memory* : gestion des signaux de mesures en sortie des phasemètres, pour les enregistrer dans des fichiers et les mémoriser pour permettre l'application de *TDI*.

Le module *LISA* gère les modules principaux qui modélisent le détecteur LISA et plus particulièrement le module phasemètre (*PhoDetPhaMet*), qui est l'élément central de l'organisation de LISACode comme on le constate sur la figure 1. Ce module utilise les autres modules de LISA, *Noise*, *Geometry*, *TrFctGW* et *USOClock* pour calculer son signal de mesure. Le module *Noise* ne décrit pas des bruits en particulier, comme un bruit laser ou un bruit de masse inertielle, mais il décrit un type de bruit comme par exemple un bruit blanc, un bruit filtré passe-bas, un bruit obtenu à partir d'un filtre polynomial, un bruit lu dans un fichier, etc. C'est le module phasemètre (*PhoDetPhaMet*) qui utilise le module bruit (*Noise*) pour générer les bruits intervenant dans la mesure.

Il y a trois modules principaux en-dehors des modules du détecteur LISA : le module onde gravitationnelle (*GW*), le module fond gravitationnel *Background* et le module *TDI*. Le module onde gravitationnelle (*GW*) est utilisé par le module de LISA *TrFctGW* pour calculer le signal sur un bras induit par les ondes gravitationnelles. Ce module fonctionne de la même manière que le module bruit (*Noise*). Il ne contient pas la description d'une onde gravitationnelle associée à une source en particulier mais la description de différents types d'ondes telles qu'une onde monochromatique, une onde émise par un système binaire calculée dans l'approximation Post-Newtonienne, etc. LISACode utilise ce module lors de sa phase d'initialisation pour créer les ondes gravitationnelles utilisées dans la simulation. Le deuxième module est le module fond gravitationnel *Background* qui est utilisé pour modéliser un fond gravitationnel diffus dans certaine situation. Le troisième module principal externe à LISA est le module *TDI* qui applique la méthode *TDI* sur les signaux de mesures mémorisés dans le module mémoire *Memory*. Comme pour les autres modules, il n'applique pas un générateur *TDI* en particulier mais est un support général à cette application. Il est utilisé par LISACode pour appliquer les générateurs requis par l'utilisateur.

LISACode utilise donc une structure possédant toutes les bases nécessaires à la simulation pour construire un modèle de LISA, des ondes gravitationnelles ainsi que les générateurs *TDI* correspondants aux exigences de l'utilisateur.

2.3 Implémentation

Afin de modéliser LISA au plus près de la réalité, j'ai eu besoin d'utiliser un langage de programmation qui permet de former des objets représentant les différents éléments de LISACode en les organisant dans la structure modulaire décrite précédemment. J'ai donc choisi d'écrire ce logiciel en C++. Cette partie détaillera comment la modularité de la programmation objet en C++ a permis de construire LISACode.

La programmation objet est basée sur des classes à partir desquelles sont créés des objets. Une classe est une structure composée d'un ensemble de variables et de méthodes. Chaque objet construit à partir d'une classe possède cet ensemble de variables et de méthodes qui lui sont propres. Ainsi, à partir d'une seule classe, une multitude d'objets peuvent être créés. Pour reprendre des notions de programmation classique telle que le FORTRAN ou le C, la classe serait en quelque sorte un type ou une structure, l'objet serait alors la variable créée à partir de ce type ou de cette structure. Dans LISACode, il y a au moins une classe par module. Par exemple le module onde gravitationnelle, (GW), (cf. partie précédente 2.2 et figure 1) est décrit par une classe contenant trois variables, qui sont les deux angles de position et l'angle de polarisation, et deux méthodes, qui décrivent les évolutions temporelles des deux composantes de polarisation $h_+(t)$ et $h_\times(t)$. A partir de cette seule classe, il est possible de créer autant d'ondes gravitationnelles que nécessaire.

L'autre notion importante du C++ qui est utilisé dans LISACode est l'héritage. A partir d'une classe que l'on appellera classe mère, il est possible de dériver plusieurs autres classes qui possèdent obligatoirement les variables et les fonctions de la classe mère, plus d'autres variables et d'autres fonctions spécifiques à la classe dérivée. Ainsi cette classe dérivée, ou classe fille, peut être vue depuis l'extérieur comme étant de la même nature que la classe mère puisqu'elle possèdent les mêmes attributs. Autrement dit, un pointeur de type classe mère peut également pointer sur un objet construit à partir de la classe mère ou de la classe fille. Dans l'exemple du module onde gravitationnelle, (GW), la classe mère correspond à la classe décrite dans le paragraphe précédent. Une classe fille décrit un type d'onde gravitationnelle plus spécifique, comme par exemple le type d'une onde émise par une binaire en calcul post-newtonien (cf. partie **??**). Cette classe fille possède les 3 variables de la classe mère, mais elle possède en plus les masses des corps composant la source, le temps de coalescence, l'inclinaison et la distance. Elle possède également les deux fonctions qui décrivent $h_+(t)$ et $h_\times(t)$ et le calcul qu'elles effectuent correspondant au calcul post-newtonien. Il est possible de créer d'autres classes fille de la même manière comme une onde monochromatique, une binaire de fréquence fixe, etc. Le module de LISA qui calcule la réponse aux bras en fonction des ondes gravitationnelles,

(*TrFctGW*), pointe[1] sur le type onde gravitationnelle en générale c'est-à-dire sur le type de la classe mère. Mais l'objet sur lequel il pointe peut très bien être une des classes filles puisque les seules choses dont ce module a besoin, ce sont les trois variables et les deux fonctions que possèdent aussi bien la classe mère que la classe fille.

Chaque classe ou groupe de classes (classe mère et classes filles dérivées) forme un module. Ces modules sont regroupés dans une bibliothèque et constituent alors des briques élémentaires qui peuvent être utilisées par LISACode comme par d'autres programmes. Au sein même de LISACode, il existe d'ailleurs trois exécutables qui correspondent à des agencements différents des modules un exécutable principal *LISACode* et deux exécutables annexes, *DnonGW* et *TDIApply* sur lesquelles on reviendra par la suite.

Il est important de voir que chaque classe, constituant les ondes gravitationnelles ou le détecteur LISA, modélise un élément physique. C'est donc une réflexion sur la physique mise en jeu dans l'élément qui détermine quelle sont ses variables et quelles sont ses fonctions. Cet effort fait sur la compréhension et la modélisation des processus physiques et le rôle de chaque élément, assure à LISACode un maximum de réalisme.

En utilisant la programmation objet, il est donc possible de programmer la structure modulaire de LISACode décrite dans la partie précédente 2.2. L'organisation des classes constituant LISACode est présentée sur la figure 2. Les modules présentés reprennent ceux de la figure 1 à quelques différences près. Les modules précédents, réponse d'un bras *TrFctGW*, phasemètre *PhoDetPhaMet* et *LISA*, correspondent chacun à une classe mais ces trois classes ont été regroupées dans un nouveau module appelé *Detecteur*. Trois autres modules ont été ajoutés : le module *InputData* pour la configuration du simulateur, le module *Generalites* pour les constantes globales et le module *Outils_Maths* pour les différents outils mathématiques. Ce schéma présente également les connexions entre classes mères (flèches à pointeur plein) qui ordonnent les échanges de données. On distingue le pas de temps avec lequel ces échanges s'effectuent par le style de flèche. Ces connexions forment la structure de base où les échanges sont réduits aux seules données physiques nécessaires. Les classes filles ne font qu'ajouter des détails supplémentaires dans la définition de certains modules. On constate que le phasemètre, modélisé par la classe *PhoDetPhaMet*, est toujours au centre de cette structure et sépare les processus selon le pas de temps utilisé, comme on l'a déjà vu.

Sans modifier les modules de LISACode, il est possible de former deux exécutables autres que l'exécutable principal *LISACode*. Le premier exécutable,*DnonGW*, utilise le module dédié aux ondes gravitationnelles *GW* et celui dédié à la réponse d'un bras *TrFctGW* pour calculer uniquement le signal gravitationnel induit sur chaque bras. Le deuxième exécutable, *TDIApply*, utilise les modules de gestion de mémoires, *Memoire*, et d'application de *TDI*, *TDI*, pour appliquer la méthode *TDI* sur des fichiers contenant les signaux de mesure.

LISACode est donc un logiciel écrit en C++ qui utilise la modularité de la program-

[1]Il pointe c'est-à-dire qu'il sait quelles ondes gravitationnelles utiliser, qu'ils connaît l'adresse en machine des objets correspondant à ces ondes

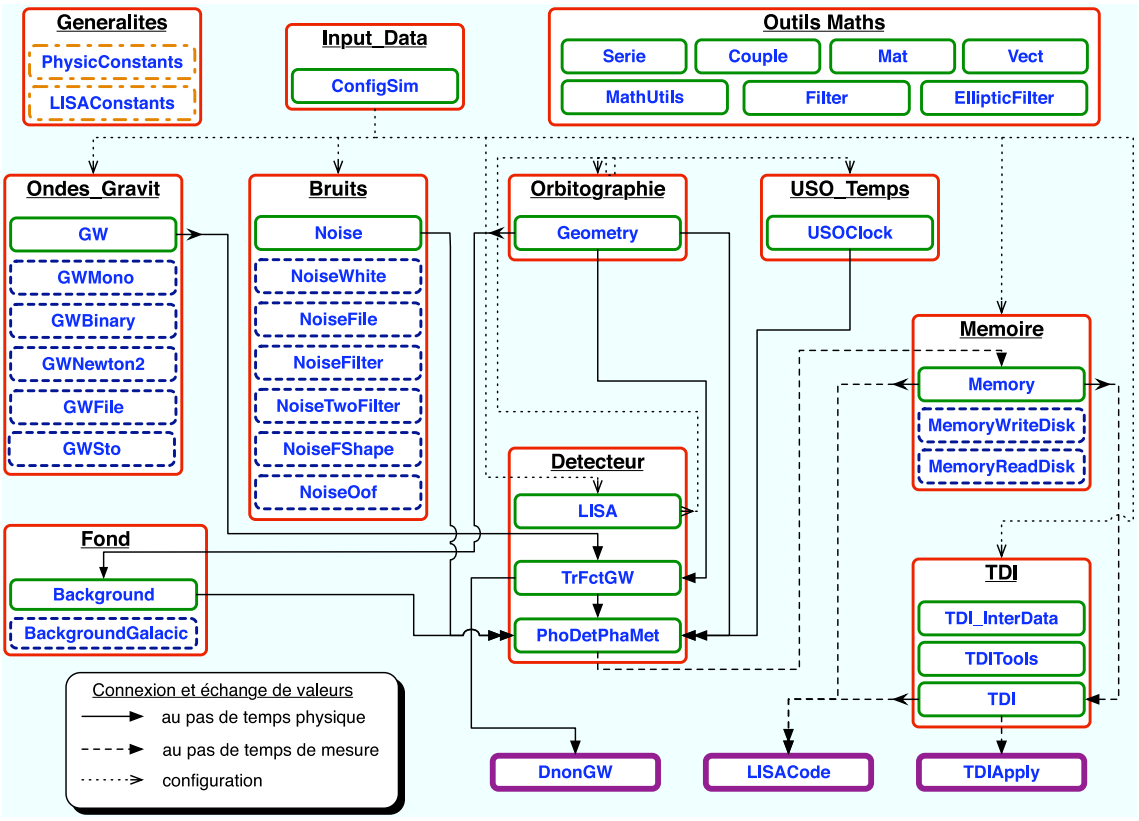

Figure 2: Organisation des classes constituant LISACode. Les classes mères sont représentées par les blocs en trait plein vert et les classes filles par des blocs en tirets bleus. Ces classes sont regroupées par module (bloc rouge et nom en gras souligné). Les trois exécutables correspondent aux trois blocs en trait plein épais violet en bas du schéma. Les connexions entre classes ou entre modules sont représentées par des flèches. Elles se traduisent par le fait que le bloc pointé par la flèche interroge des fonctions du bloc de départ de la flèche qui lui transmet ainsi des données. On distingue les échanges qui se font au pas de temps physique par des flèches en trait plein, ceux qui se font au pas de temps de mesure par des flèches en tirets et ceux qui permettent l'initialisation par des flèches en pointillés. Les modules *Generalites* et *Outils_Maths* sont connectés à tous les autres blocs mais ces connexions n'ont pas été représentées, pour alléger la figure. Les blocs en points-tirets orange correspondent à des fichiers de constantes. Ce schéma correspond à celui de LISACode version 1.4 .

mation objet pour créer une structure au plus proche de la réalité du détecteur LISA puisqu'elle est issue d'une réflexion sur la physique et le rôle de chaque élément. La flexibilité de cette structure permet à LISACode de s'adapter à l'usage que l'on souhaite en faire.

3 Détails du fonctionnement

LISACode est un logiciel écrit en C++ dont la structure flexible est formée de différents modules. Dans la partie précédente, les réflexions sur les processus physiques mis en jeu et le rôle de chaque élément, qui sont à l'origine de cette structure, ont été exposées. Dans cette partie, chaque module sera décrit et l'accent sera mis sur quelques points de détails importants.

On décrira d'abord le module modélisant les ondes gravitationnelles et celui modélisant les bruits. Puis on présentera rapidement le module modélisant un fond gravitationnel, le module modélisant les orbites et celui modélisant les horloges ultra-stable (*USO*). Ensuite on s'attardera sur le module central modélisant le détecteur et plus particulièrement le phasemètre. Enfin le module de gestion des mémoires et le module d'application de *TDI* seront exposés. On terminera par une présentation des autres modules que sont les constantes, les outils et les exécutables.

Les descriptions qui suivent concernent la version actuelle de LISACode, c'est-à-dire la version 1.4. Ce simulateur évoluant constamment, certains détails auront peut-être changer dans les versions à venir mais les grandes lignes seront inchangées. Les changements concernent généralement des ajouts de classes filles.

3.1 Module onde gravitationnelle GW

Le module onde gravitationnelle a déjà été en partie présenté, puisqu'il a servi d'exemple pour illustrer les principes de base de LISACode et leur implémentation, mais nous allons ici reprendre sa description dans le détail.

Ce module, *Ondes_Gravit* sur la figure 2, est formé par une classe mère de laquelle dérivent cinq classes filles. La classe mère possède les variables et les fonctions communes à tous les types d'onde gravitationnelle localisée. Le point commun de toutes ces description d'ondes est en premier lieu la description de la position. La classe mère possède donc trois variables qui sont les deux angles écliptiques qui définissent la position, soient la déclinaison β et l'ascension droite λ, et l'angle de polarisation ψ. Elle dispose également des fonctions associées à cette position, comme par exemple le calcul des composantes du vecteur normal à la direction de la source dans le référentiel barycentrique. Le second point commun concerne le fait que toutes les ondes gravitationnelles sont décrites par deux états de polarisation qui évoluent dans le temps, $h_+(t)$ et $h_\times(t)$. La description de ces deux fonctions ne peut pas être faite dans la classe mère car celle-ci ne dispose pas d'informations suffisantes sur la source. La classe mère impose donc que ces fonctions existent dans les classes filles mais ne les décrit pas. Elles sont définies comme des fonctions virtuelles. Une fonction virtuelle est un type de fonction en C++ qui correspond à une fonction exigée par la classe mère et décrite par les classes filles. Cette classe décrit des ondes dont la position est bien définie. Le cas des ondes gravitationnelles diffuses sera traité soit en utilisant le module *Fond* (cf. partie 3.3) soit, dans certains cas, en utilisant

ce même module *GW* d'une manière bien particulière que l'on présentera dans la partie **??** (mettre la ref quand fait dans la partie resultat).

Les cinq classes filles définissent les fonctions d'évolution des états de polarisation $h_+(t)$ et $h_\times(t)$ en fonction du type de source qu'elles représentent. Ces classes possèdent des paramètres propres au type d'onde qu'elles décrivent.

Parmi ces cinq classes, trois décrivent des sources dont les fonctions $h_+(t)$ et $h_\times(t)$ ont des expressions analytiques :

- *GWMono* : Cette classe modélise une onde monochromatique quelconque définie par la fréquence f, l'amplitude des composantes h_{0+} et $h_{0\times}$ et les phases initiales ϕ_{0+} et $\phi_{0\times}$. Les évolutions temporelles des deux composantes de polarisation sont alors décrites par :

$$h_{S+}(t) = h_{0+}\sin(2\pi f t + \phi_{0+}) \quad (1)$$
$$h_{S\times}(t) = h_{0\times}\sin(2\pi f t + \phi_{0\times}) \quad (2)$$

- *GWBinary* : Cette classe modélise une onde monochromatique issue d'une source binaire présentée dans la partie **??**. La binaire évolue si lentement que la fréquence de l'onde est considérée fixe. Les évolutions temporelles des deux composantes de polarisation sont alors décrites par les équations (**??**), (**??**) et (**??**) qui dépendent de la masse totale de la binaire m_{tot}, de son rapport de masse ν, de son inclinaison i, de sa fréquence orbitale f_{orb} qui est la moitié de la fréquence de l'onde f_{OG}, de sa phase initiale ϕ_0 et de la distance r entre la source et le Soleil .

- *GWNewton2* : Cette classe modélise une onde issue d'une binaire spiralante décrite par l'approximation post-newtonienne (cf. partie **??**). Les fonctions décrivant les deux composantes de polarisation utilisent au choix les formulations 1 PN (**??**) et (**??**) (cf. partie **??**) ou les formulations (**??**) et (**??**) plus précises à 2.5 PN (cf. partie **??**). Ces formulations dépendent du temps de coalescence[2] t_{coal}, de la masse totale de la binaire m_{tot}, de son rapport de masse ν, de son inclinaison i, de sa phase initiale ϕ_0 et de la distance r entre la source et le Soleil.

Les différents paramètres dont dépendent les fonctions $h_+(t)$ et $h_\times(t)$ sont des paramètres propres à chacune des classes qui possèdent également les fonctions permettant de les gérer. On rappelle que tout ce qui concerne la position est géré par la classe mère et n'a donc pas besoin d'être redéfini dans les classes filles puisque par définition elles possèdent tout ce que possède la classe mère.

Les deux autres classes filles ont des types un peu particuliers. Tout d'abord la classe *GWFile* récupère les évolutions temporelles des composantes de polarisation dans un fichier dont les trois colonnes sont le temps t, $h_+(t)$ et $h_\times(t)$. Le pas de temps des données dans le fichier n'a pas de réel importance car une interpolation lagrangienne permet de les

[2]Temps entre le début de l'observation de la binaire et sa coalescence.

adapter au pas de temps de la simulation. La potentialité de la lecture dans un fichier est très importante car elle permet de modéliser des ondes gravitationnelles complexes résultant de calcul fait avec d'autres simulateurs. Cette potentialité est utilisée pour la modélisation des EMRIs dans LISACode. En effet, les ondes issues de ces sources sont calculées à partir d'un programme qui réalise l'intégration numérique des équations du modèle de Barack et Cutler **?** . Ce programme du nom de CodeEMRI, écrit par Philippe GrandClément puis modifié par Guillaume Trap et moi-même, sera dans l'avenir intégré à LISACode, mais le passage par fichier permet d'étudier la détection des EMRIs par LISA en attendant que cette intégration soit réalisée.

Enfin la dernière classe fille de ce module onde gravitationnelle est la classe *GWSto*. Elle modélise une onde gravitationnelle comme un bruit. Ce bruit gravitationnel est caractérisé par sa distribution spectrale. Il est modélisé en créant dans les variables de la classe deux objets de type bruit, un pour chaque composante de polarisation. Ainsi, on utilise le module bruit dans le module onde gravitationnelle, ce qui illustre bien la flexibilité de l'architecture de LISACode. Cette classe est notamment utilisée pour la génération du fond stochastique défini dans les parties **??** et **??**. On reviendra sur cette génération dans la partie **??** (Mettre la ref quand la partie sur la generation du fond sto sera ecrite). Plus de détails sur la façon de générer ces bruits seront évidemment fournis dans la partie suivante qui concerne le module bruit.

3.2 Module bruit **Bruits**

Le module bruit permet de générer les bruits instrumentaux du détecteur LISA mais aussi des bruits gravitationnels comme on vient de le voir. Il génère un bruit de manière générale, qui peut ensuite être utilisé n'importe où, aussi bien pour un bruit laser, que pour un bruit de masse inertielle, ou que pour n'importe quel autre bruit. Dans cette partie, on détaillera ce module bruit en s'intéressant plus particulièrement aux particularités imposées par LISA, puis en présentant les méthodes utilisées pour modéliser les différents types de bruits.

3.2.1 Aspect général d'un bruit dans LISA (classe mère)

Le schéma de ce module bruit, *Bruits* sur la figure 2, est similaire à celui du module onde gravitationnelle. Il y a une classe mère et six classes filles. La classe mère contient les variables et les fonctions communes à tous les types de bruits. La manière dont un élément extérieur interroge un bruit doit être la même, quelque soit le type de ce bruit. La méthode qui fournit un bruit appartient donc à la classe mère. Cette méthode est assez complexe du fait de l'utilisation particulière des bruits dans LISA, à savoir qu'un même bruit peut intervenir sur plusieurs signaux à des instants différents. En effet, comme on l'a vu dans la partie **??** sur la formulation des signaux, une partie d'un faisceau laser est détectée quasi instantanément par des photodiodes du satellite contenant le laser et

l'autre partie de ce même faisceau est envoyée vers un autre satellite et est détectée par une photodiode de celui-ci plus de 16 secondes après l'émission et la détection dans le satellite émetteur. Les bruits transportés par ce faisceau, et notamment le bruit laser, interviendront donc sur les signaux de mesure des deux satellites à deux instants décalés du temps de parcours du faisceau le long du bras. Une modélisation précise et réaliste de cet effet est nécessaire pour que la méthode *TDI* puisse réduire le bruit laser dans des conditions aussi proches que possible de celles de la future mission et pour que l'efficacité des différents générateurs puisse être étudiée. Cette modélisation est effectuée au niveau de la construction des signaux par le phasemètre et sera détaillée dans la partie 3.6. La conséquence sur les bruits de cet effet de non-instantanéité est le fait qu'ils peuvent être interrogés à différents instants et non pas uniquement au moment de leur génération. Pour répondre à cette exigence, chaque bruit est mis en mémoire sur une certaine période. L'utilisation de cette mémoire se décompose en deux fonctions qui sont d'une part la génération et le stockage du bruit et d'autre part le renvoie de la valeur du bruit pour un temps compris entre $t - L/c$ et t où t est le temps courant et L/c la durée de propagation le long d'un bras.

Pour la mise en mémoire du bruit, la classe mère possède un tableau dans lequel les dernières valeurs de bruit générées sont stockées. Ce tableau de taille fixe est glissant, c'est-à-dire qu'à chaque pas de temps physique, les valeurs sont décalées d'une case, la dernière valeur est éliminée et le bruit nouvellement généré est stocké comme première valeur[3]. Ainsi la première valeur du tableau correspond toujours au temps courant t et la dernière au temps $t - T_{Mem.\ Bruit}$ où $T_{Mem.\ Bruit}$ est la durée sur laquelle le bruit est mémorisée. Toute cette gestion étant réalisée par la classe mère, celle-ci doit posséder une fonction qui permet d'ajouter une valeur de bruit à chaque pas de temps physique. Cette fonction est exécutée par la classe *LISA* qui gère l'avancement temporelle . Elle fait glisser le tableau et appelle une fonction de génération du bruit pour obtenir la nouvelle valeur à stocker. La fonction de génération du bruit dépend du type de bruit et est définie dans les classes filles mais la classe mère impose son existence par une fonction virtuelle (type de fonction du C++ définie dans la partie précédente 3.1).

Une autre fonction de la classe mère permet de renvoyer la valeur du bruit pour un temps compris entre $t - T_{Mem.\ Bruit}$ et t ou plus exactement pour un retard par rapport au temps courant T_L compris entre 0 et $T_{Mem.\ Bruit}$. Le problème est que ce retard T_L n'a aucune raison d'être un multiple du pas de temps physique. La valeur de bruit est alors interpolée à partir des valeurs mémorisées. Typiquement, l'interpolation utilisée est une interpolation de Lagrange d'ordre 7 et la durée de mémorisation $T_{Mem.\ Bruit}$ est d'environ 20 secondes ,ce qui permet sans problème d'interpoler pour des valeurs de retards entre 16 à 17 secondes. Cette interpolation est une limite au réalisme de la simulation qui est imposée par les contraintes d'une modélisation numérique.

[3]En fait, on utilise un vecteur en C++ (classe vector) qui permet d'ajouter une case au début du vecteur et d'éliminer la dernière case, ce qui revient au principe d'un tableau glissant.

3.2.2 Différents types de bruit (classes filles)

Les six classes filles du module bruit définissent les différents types de bruit que LISACode peut générer.

La plus simple est certainement la classe *NoiseWhite* qui génère un bruit blanc défini par le niveau de la densité spectrale de puissance (DSP) [4]. Ce bruit blanc est créé par un tirage dans une fonction gaussienne dont l'écart-type est :

$$\sigma = \sqrt{\frac{PSD}{2\ \Delta t_{physique}}} \tag{3}$$

Une autre classe de bruit assez simple est la classe *NoiseFile* qui récupère les valeurs de bruit dans un fichier à deux colonnes : $t\ bruit(t)$. De la même manière que dans la classe *GWFile*, les données sont adaptées au pas de temps physique de la simulation par une interpolation. La potentialité offerte par cette classe est très importante dans le cadre d'une utilisation de LISACode en interaction avec des développements expérimentaux car elle permet d'utiliser de vraies mesures de bruit dans une simulation. Cela permet, d'une part, d'ajouter au réalisme de la simulation, et d'autre part, de tester l'impact de ces bruits dans un modèle global de LISA.

Les quatre autres classes filles génèrent des bruits à partir d'une description plus ou moins élaborée de leur DSP. Elles sont basées sur le filtrage d'un bruit blanc et utilisent un ou plusieurs filtres selon le type de description de la DSP :

- *NoiseFilter* : Cette classe modélise un bruit à partir d'un bruit blanc qui est ensuite filtré. Ce bruit est directement défini par les coefficients du filtre. Il est typiquement utilisé pour décrire des bruits dont la racine carrée de la DSP est proportionnelle à une puissance entière de la fréquence.

$$\sqrt{DSP} = A\ f^{\alpha} \tag{4}$$

- *NoiseTwoFilter* : Cette classe modélise un bruit de la même manière que la précédente mais en générant deux bruits blancs qui sont ensuite filtrés par deux filtres et sommés. Elle permet typiquement de décrire des bruits dont la racine carrée de la DSP est une somme de deux puissances entières de la fréquence :

$$\sqrt{DSP} = A\ f^{\alpha} + B\ f^{\beta} \tag{5}$$

- *NoiseFShape* : Cette classe modélise un bruit dont la racine carrée de la DSP est une somme de puissances entières de la fréquence, soit :

$$\sqrt{DSP} = \sum_{i=1}^{M} A_{-i} f^{-i} + \sum_{j=0}^{N} A_j f^j = A_{-M} f^{-M} + ... + A_{-1} f^{-1} + A_0 + A_1 f^1 + ... + A_{-N} f^{-N} \tag{6}$$

[4] La DSP d'un bruit blanc est plate et n'est donc définie que par une seule valeur.

Ce bruit est généré à partir d'un bruit blanc qui est filtré simultanément par autant de filtre qu'il y a de puissances de la fréquence dans la DSP, puis les résultats de tous ces filtrages sont sommés.

- *NoiseOof* : Cette classe modélise un bruit dont la racine carrée de la DSP est une puissance non entière de la fréquence. Ce bruit est généré à partir d'un bruit blanc filtré par un filtre dont les coefficients sont calculés en fonction de la valeur de la puissance de la fréquence. Cette technique de génération de bruit est présentée dans l'article de S. Plaszczynski **?**.

Le ou les filtres utilisés dans toutes ces classes filles sont des objets construits à partir de la classe *Filter* du module *Outils_Maths* qui applique un filtre récursif décrit par des coefficients récursifs α_i et directs β_i. L'application de ce filtre sur des données temporelles échantillonnées x_n donnent les données filtrées y_n par :

$$y_n = \sum_{k=1}^{N_\alpha} \alpha_k \; y_{n-k} + \sum_{k=0}^{N_\beta} \beta_k x_{n-k} \tag{7}$$

Le calcul d'une nouvelle valeur de bruit filtré dans le tableau de bruits de la classe mère se fait en utilisant les données d'entrée x_n qui correspondent ici aux valeurs du bruit blanc et les valeurs de bruit déjà calculés y_n. Les classes doivent donc posséder plusieurs tableaux de données intermédiaires pour l'application du ou des filtres, qui contiennent par exemple les valeurs de bruit blanc. Un filtre peut être décomposé en de multiples sous-filtres (ou cellules), chacun défini par un jeu de coefficients. Cette décomposition permet de réaliser des filtrages brutaux en évitant des problèmes d'imprécision numérique.

Le calcul des coefficients récursifs α_i et des coefficients directs β_i d'un filtre en fonction de la forme de la racine carrée de la DSP peut se faire en utilisant la transformation bilinéaire. Prenons l'exemple d'un bruit standard de masse inertielle défini dans la partie **??**. La racine carrée de la DSP de ce bruit est de la forme suivante :

$$\sqrt{S_{\frac{\delta\nu}{\nu},MI}} = \mathcal{A} \; f^{-1} \tag{8}$$

La fonction de transfert s'écrit alors : expliquer un peu pourquoi $2\pi\mathcal{A}(i\omega)^{-1}$

$$H(\omega) = \frac{\mathcal{A}}{f} = 2\pi\mathcal{A}(i\omega)^{-1} \tag{9}$$

où ω est la pulsation ?. La transformation bilinéaire est définie comme :

$$s = i\omega = \frac{2}{\Delta t}\frac{1 - Z^{-1}}{1 + Z^{-1}} \tag{10}$$

où Δt est le pas de temps et Z la variable complexe de la transformation en Z qui est l'équivalent discret en traitement du signal de la transformée de Laplace. La fonction de

transfert s'écrit alors :

$$H(Z) = \pi \ \mathcal{A} \ \Delta t \ \frac{1 + Z^{-1}}{1 - Z^{-1}} \tag{11}$$

Si Y représente les données filtrées et X les données brutes, la transformation s'écrit $Y = HX$, soit :

$$y_n = y_{n-1} + \pi \ \mathcal{A} \ \Delta t \ \ (x_n + x_{n-1}) \tag{12}$$

Les coefficients α_i et β_i du filtre, définis par la formulation (7) sont donc pour le filtre qui permet de générer le bruit de masse inertielle :

$$\alpha_{1,MI} = 1 \quad \text{et} \quad \beta_{0,MI} = \beta_{1,MI} = \pi \ \mathcal{A} \ \Delta t \tag{13}$$

Verifier et etayer tout ca de quelques justifications en utilisant notamment comme référence le livre de traitement du signal

Un calcul similaire permet d'obtenir les coefficients du filtre permettant de générer un bruit de *shot noise* défini dans la partie **??** et dont la racine carrée de la DSP est de la forme :

$$\sqrt{S_{\frac{\delta\nu}{\nu},SN}} = \mathcal{A} \ f \tag{14}$$

Ces coefficients sont alors :

$$\alpha_{1,SN} = -1 \quad \text{et} \quad \beta_{0,SN} = -\beta_{1,SN} = \frac{\mathcal{A}}{\pi \ \Delta t} \tag{15}$$

3.3 Module fond gravitationnel **Fond**

Ce module est prévu pour modéliser un fond d'onde gravitationnelle diffus à partir d'une modélisation éventuelle de la réponse de chaque bras à ce fond. Un fond d'ondes gravitationnelles est différent d'une onde gravitationnelle classique car il n'a pas de direction de propagation précise. La modélisation des fonds gravitationnels est un problème complexe dans LISA qui n'a pas encore de solution précise car il faut à la fois tenir compte de la dispersion de la source et du mouvement de LISA. La solution exposée ici propose d'injecter directement dans le phasemètre le signal gravitationnel induit sur chaque bras par un fond. Mais cela suppose de disposer d'une modélisation analytique ou numérique de ce signal ce qui est loin d'être évident.

Le module *Fond* est constitué d'une classe mère, la classe *Background*, qui possède seulement un pointeur sur les orbites et une fonction virtuelle qui impose aux classes filles de renvoyer une réponse pour chaque bras à chaque instant. Les classes filles décrivent les différentes modélisations de cette réponse. Etant donné qu'aujourd'hui il n'existe pas de solution réellement propre et efficace pour décrire la réponse de LISA à un fond gravitationnel, il n'y a actuellement qu'une classe fille qui récupère cette réponse dans un fichier. Ainsi le calcul de la réponse à ce fond est déporté sur une modélisation extérieure à LISACode ou qui utilise LISACode d'une manière spécifique comme on le verra dans la partie **??** (Mettre quand ecrit) qui expose les résultats que j'ai obtenus pour le fond

galactique. La seule classe fille est la classe *BackgroundGalactic*. Son nom vient du fait qu'elle a été initialement utilisée pour introduire la réponse de LISA au fond Galactique. Elle récupère les signaux de réponse des six liens induits sur les six signaux externe-interne dans un fichier à sept colonnes : *temps* s_1^{FOB} s_2^{FOB} s_3^{FOB} $s'_1{}^{FOB}$ $s'_2{}^{FOB}$ $s'_3{}^{FOB}$ où FOG signifie Fond d'Onde Gravitationnelle. La limite de cette méthode est que les signaux de réponse au fond gravitationnel dépendent des orbites des satellites. Il faut donc que les orbites utilisés pour la génération du fichier soient similaires à ceux de la simulation .

3.4 Module orbitographie **Orbitographie**

Avant d'aborder la question de la modélisation du détecteur, et plus particulièrement du phasemètre, dans la partie 3.6, il est nécessaire de dire quelques mots sur les modules *Orbitographie* et *USO_Temps* inclus dans la modélisation de LISA et utilisés par le module détecteur pour construire les signaux de mesure. Dans cette partie, les grandes lignes du module *Orbitographie* seront présentées.

Ce module est formé d'une seule classe[5], la classe *Geometry*. Les orbites qu'elle modélise sont celles de Dhurandhar, Nayak et Vinet détaillées dans la partie **??**. Ces orbites étant analytiques, il n'y a pas de complexité particulière. Elles sont définies par les trois paramètres suivants qui sont donc des variables de la classe :

- $t_{0,\ orb}$: temps compris entre 0 et un an qui repère la position initiale du barycentre de LISA par rapport à la position initiale standard correspondant au point vernal soit le centre sur l'axe x du référentiel barycentrique.

- $\phi_{rot,\ orb}$: angle de rotation entre la configuration initiale du triangle et la configuration initiale standard défini dans la partie **??** [6].

- L_0 : longueur nominale des bras.

La classe possède des fonctions qui renvoient la position d'un satellite, sa vitesse, le temps de parcours le long d'un bras, la vitesse relative entre deux satellites d'un même bras, etc. Il existe plusieurs options possibles décrites par deux paramètres. Le premier définit si on considère que les satellites bougent, c'est-à-dire le cas réaliste, ou, au contraire, si on considère une configuration fixe du détecteur, ce qui s'avère utile, notamment pour des tests sur la méthode *TDI* ou pour des calculs de sensibilité, comme on le verra par la suite. Le deuxième paramètre définit l'ordre d'approximation utilisé dans le calcul des temps de parcours présenté dans la partie **??** :

- ordre 0 : *flexing*

[5]Une classe mère mais pas de classe fille pour reprendre le schéma des modules déjà exposés.

[6]La configuration initiale standard est la configuration où, à $t = 0$, le triangle est pointe en bas, avec le satellite 1 sur cette pointe, le satellite 2 du coté $y < 0$ et le satellite 3 sur la troisième pointe.

- ordre 1 : *flexing* + effet Sagnac
- ordre 2 : *flexing* + effet Sagnac + corrections relativistes

La classe contient aussi d'autres variables et fonctions qui permettent d'optimiser la gestion des orbites. Dans l'avenir, ce module sera certainement décomposé en une classe mère et plusieurs classes filles qui offriront la potentialité d'utiliser d'autres types d'orbites tels que des orbites intégrant des éphémérides pour prendre en compte les différents objets du système solaire par exemple.

3.5 Module horloges : USO

Pour compléter la description des modules nécessaires au détecteur, il reste à décrire le module *USO_Temps* qui modélise les horloges ultra-stables de chaque satellite. Ce module très simple ne comporte qu'une seule classe, la classe *USOClock*.

Chaque satellite dispose d'une horloge utilisée par le phasemètre comme référence pour la mesure de phase, comme on l'a évoqué dans les parties **??** et **??**. La grande précision requise sur ces mesures nécessite une extrême stabilité de l'horloge d'où son appellation *USO*, ce qui veut dire *Ultra-Stable Oscillator*. Cette horloge sert également au phasemètre pour étiqueter temporellement les mesures c'est-à-dire pour fournir le temps auquel chaque mesure a été effectuée. Cette étiquette temporelle doit être très précise pour que la réduction des bruits lasers dans *TDI* se fasse efficacement.

Aujourd'hui, les caractéristiques techniques des *USO* sont mal connues puisque les principes technologiques employés pour l'USO ne sont pas encore clairement définis. Mais on peut tout de même estimer qu'il y a trois types de bruit possibles : un offset, un bruit gaussien stationnaire et une dérive.

La classe *USOClock* possède une fonction qui, à chaque pas de temps, renvoie un temps dans lequel les différents types de bruit ont été pris en compte. La prise en compte de cet effet dans LISACode permet de tester par exemple les conséquences d'un décalage entre les horloges des satellites.

3.6 Module détecteur dont phasemètre

Dans les parties précédentes, le module modélisant les ondes gravitationnelles ainsi que les modules modélisant les bruits, les orbites et les *USOs* ont été décrits. Tous ces modules décrivent des éléments qui interviennent dans les signaux de mesures que fournit le détecteur LISA par l'intermédiaire du phasemètre. Dans cette partie, le module détecteur sera décrit et plus particulièrement l'élément central de LISACode qu'est le phasemètre.

Ce module *Detecteur* (sur la figure 2) est composé de trois classes qui permettent de construire les signaux de mesures. Ces classes sont *LISA* qui gère l'ensemble des éléments composants LISA, *TrFctGW* qui calcule le signal gravitationnel et *PhoDetPhaMet* qui compose le signal de mesure et modélise un phasemètre.

3.6.1 Gestion du détecteur (classe **LISA**)

La classe *LISA* ne modélise aucun élément particulier mais gère l'ensemble du détecteur. Elle possède différents objets construits à partir des classes des modules de bruits (*Bruits*), d'orbitographie (*Orbitographie*) et d'horloge (*USO_Temps*) qui ont été exposés précédemment et des classes de réponse gravitationnelle (*TrFctGW*) et de phasemètre (*PhoDetPhaMet*) qui le seront dans les prochaines sous-parties. Ces objets sont créés en fonction de la configuration de la simulation (classe *Input_Data*) et modélisent l'ensemble des trois satellites du détecteur.

Deux objets concernent des éléments d'ensemble de LISA : un objet, de type *Geometry*, qui décrit les orbites des trois satellites (cf. partie 3.4) et un objet, de type *TrFctGW*, qui calcule le signal gravitationnel à partir de la liste des ondes gravitationnelles (cf. sous-partie 3.6.2 suivante).

La classe possède également des objets qui représentent les éléments des satellites agissant sur les signaux de mesures. La plupart de ces éléments interviennent au niveau de la circulation des faisceaux sous la forme de bruits qui ont été détaillés dans la partie **??** du chapitre **??**. Il y a, par banc optique, un élément de chaque type qui influe sur le faisceau : le laser, la masse inertielle, la photodiode et le banc optique lui-même. Les bruits associés sont le bruit laser (cf. **??**), le bruit d'accélération de la masse inertielle (cf. **??**), le *shot noise* pour l'interférence avec le faisceau externe (cf. **??**), des bruits de chemin optique sur le faisceau externe (cf. **??**) et le bruit d'accélération du banc optique (c f. **??**). Les bruits concernant le faisceau externe sont regroupés dans un même terme dit bruit de chemin optique. Il y a donc 4 bruits par banc optique et 6 bancs optiques soit un total de 24 bruits. Ces bruits sont listés et groupés par 6 dans un tableau de pointeurs sur le type *Noise*. Les objets pointés sont construits à partir des classes filles dérivées de la classe *Noise* en fonction de la configuration de la simulation (classe *Input_Data*) [7]. La participation de ces objets au signal se fait au niveau du phasemètre.

La classe *LISA* possède également trois objets de type *USOClock* qui modélise les horloges des trois satellites

C'est également dans cette classe que sont les objets modélisant les phasemètres construits à partir de la classe *PhoDetPhaMet* qui sera détaillée dans une prochaine partie. Il y a deux phasemètres par banc optique (cf. partie **??**), un phasemètre externe-interne dit s et un phasemètre interne-interne dit τ, soit 12 phasemètres qui sont listés dans un tableau. Chacun de ces phasemètres connaît, grâce à des pointeurs, les orbites, l'objet calculant le signal gravitationnel, le signal de réponse à d'éventuels fonds gravitationnels, la liste des bruits, la liste des *USOs* et la liste des mémoires vers lesquelles il doit envoyer son signal de mesure.

Enfin la classe *LISA* possède une fonction qui, à chaque pas de temps, fait évoluer le détecteur. Cette fonction demande à chaque objet bruit de générer du bruit (cf. par-

[7] Plus précisément, c'est un pointeur sur une liste de pointeurs sur le type *Noise*. La classe *Input_Data* crée les objets représentant les bruits et transmet l'adresse mémoire aux autres modules par le pointeur.

tie 3.2.1) et aux phasemètres d'effectuer une mesure.

3.6.2 Calcul du signal gravitationnel (classe **TrFctGW**)

La classe *TrFctGW* calcule le signal gravitationnel induit sur un faisceau laser à la suite de sa propagation le long d'un bras. Ce signal est du à la déformation de l'espace au niveau du bras par les ondes gravitationnelles. La réponse d'un bras à une onde gravitationnelle a été étudiée dans la partie **??** à la fin du chapitre **??**. Cette réponse correspond à une variation relative de fréquence formulée par l'expression (**??**). Pour calculer cette réponse, il est nécessaire de connaître la direction de propagation de l'onde, son angle de polarisation et les évolutions des deux composantes de polarisation. Il faut également connaître les positions des satellites qui forment le bras et le temps de propagation du faisceau le long du bras. La classe obtient ses informations en utilisant un pointeur sur l'onde gravitationnelle et un pointeur les orbites[8]. Une des particularités de LISA est qu'il y a plusieurs ondes gravitationnelles simultanément. Le signal gravitationnel global est alors la somme des variations relatives de fréquence induites par chaque onde (cf. équations (**??**) et (**??**)). Pour obtenir les informations sur toutes les ondes gravitationnelles, la classe utilise un pointeur sur la liste des ondes[9].

3.6.3 Calcul du signal de mesure (classe **PhoDetPhaMet**)

Comme on l'a déjà indiqué à plusieurs reprises, cette classe *PhoDetPhaMet*, qui modélise le phasemètre, est l'élément central de LISACode. Son rôle principal est de construire un signal de mesure à partir des différents éléments du détecteur que sont les bruits, les réponses des bras aux ondes gravitationnelles, les orbites des satellites et les horloges ultra-stables. Il effectue alors le passage entre le domaine des processus continus fonctionnant au pas de temps physique et le domaine des mesures échantillonnées au pas de temps de mesure qui comprend l'application de *TDI*.

La classe construit le signal d'un phasemètre en combinant les différents éléments contributifs auxquels il peut accéder par l'intermédiaire de pointeurs. Ce signal est une mesure de phase effectuée à partir du signal de battement hétérodyne de la photodiode. Celle-ci mesure l'interférence entre deux faisceaux laser qui ont circulé entre les différents éléments du détecteur par l'intermédiaire des bancs optiques. Chacun de ces éléments a ajouté du bruit sur les faisceaux et par conséquent du bruit sur le signal. La classe *PhoDetPhaMet* combine les bruits selon les formulations données dans la partie **??** du chapitre **??**. Selon le phasemètre qu'elle représente, elle applique la formulation (**??**),

[8] C'est-à-dire un pointeur sur l'objet représentant les orbites de la classe *LISA*

[9] Plus précisément, c'est un pointeurs sur une liste de pointeur sur le type *GW*. Les objets pointés sont construits à partir des classes filles dérivées de la classe *GW* en fonction de la configuration de la simulation c'est-à-dire de la classe *Input_Data*. C'est cette classe qui créé les objets représentant les ondes gravitationnelles et transmet ensuite l'adresse mémoire aux autres modules par le pointeur de la même manière que pour les bruits (cf. note 7).

(??), (??) ou (??). La valeur de chacun des bruits est obtenue soit pour le temps donné par l'horloge du satellite $x(t_{USO})$, soit pour ce même temps retardé $D_i x(t_{USO}) = x(t_{USO} - L_i/c)$, par la fonction de renvoie de valeur pour un temps quelconque que possède chaque bruit pointé par la liste des bruits (cf. partie 3.2.1). Pour les phasemètres externe-interne s, la valeur du signal gravitationnel ${s_1}^{OG}$ correspond à la somme entre la valeur renvoyée par l'objet de type *TrFctGW*, et la valeur de la réponse du bras à d'éventuels fonds gravitationnels (cf. partie 3.3). On rappelle que l'objet de type *TrFctGW* calcule la réponse du bras aux ondes gravitationnelles.

Le signal qui a été décrit est le signal que devrait mesurer le phasemètre dans le cas d'une mesure parfaite. Mais, comme on l'a vu dans la partie **??**, c'est un appareil complexe dont la mesure ne peut être parfaite notamment parce qu'il doit fournir un signal échantillonné au pas de temps de mesure à partir du signal continu de la photodiode. Cette imperfection se modélise au niveau de LISACode par une fonction de transfert associée au phasemètre. Du fait qu'il est actuellement en cours de développement, sa fonction de transfert réelle n'est pas connue. Pour le moment, celle utilisée dans LISACode est très simple mais influe tout de même sur le signal.

Dans LISACode, tous les processus physiques sont modélisés avec un pas de temps physique qui est inférieur au pas de temps de mesure en sortie des phasemètres. La fonction de transfert du phasemètre doit donc sous-échantillonner le signal physique pour obtenir le signal de mesure. Le pas de temps physique est choisi comme un sous-multiple entier du pas de temps de mesure. Ainsi on prélève une valeur tous les pas de temps de mesure sur les signaux physiques pour obtenir les signaux de mesure. Mais avec un tel sous-échantillonnage les hautes fréquences sont repliées dans les basses fréquences et perturbent complètement le signal de mesure ; c'est le problème bien connu de l'*aliasing*, ou repliement de spectre. Pour éviter ce problème, il est nécessaire de filtrer brutalement les signaux physiques avant le sous-échantillonnage pour éliminer les fréquences supérieures à la moitié de la fréquence de mesure, d'après le théorème de Nyquist-Shannon. Ce filtre passe-bas est modélisé dans la fonction de transfert du phasemètre de LISACode par un filtre passe-bas écliptique. Ce filtre numérique est appliqué par la classe *Filter* qui a été présentée dans la partie 3.2.2 sur la génération des bruits. Ces coefficients dépendent de 5 paramètres qui sont : le pas de temps physique $\Delta t_{physique}$, la fréquence de coupure haute $f_{c,h}$ ($\sim 0.1 \times \Delta t_{mesure}$), la fréquence de coupure basse $f_{c,b}$ ($\sim 0.3 \times \Delta t_{mesure}$), l'oscillation en bande passante P_{dB} ($\sim 0.1\ dB$) et l'atténuation A_{dB} ($\sim 180\ dB$). Mais les pas de temps étant définis par la configuration de la simulation, les coefficients du filtre doivent être calculés dans LISACode : c'est le rôle de la classe *EllipticFilter* écrit par Hubert Halloin à partir de Demande a Hubert. La figure 3 donne la forme du filtre elliptique du phasemètre en illustrant la signification des paramètres dont il dépend. Elle donne également le résultat de son application sur un bruit blanc avant et après le sous-échantillonnage au pas de temps de mesure. On constate que le repliement de spectre est très faible, il provoque juste une très légère remontée en bande atténuée. Il faut bien voir que ce filtre n'est pas une nécessité imposée par la modélisation numérique mais qu'il

existera dans le phasemètre de LISA, le problème d'*aliasing* étant le même. Ce filtre est une spécificité importante de LISACode qui lui permet de fournir des données réalistes.

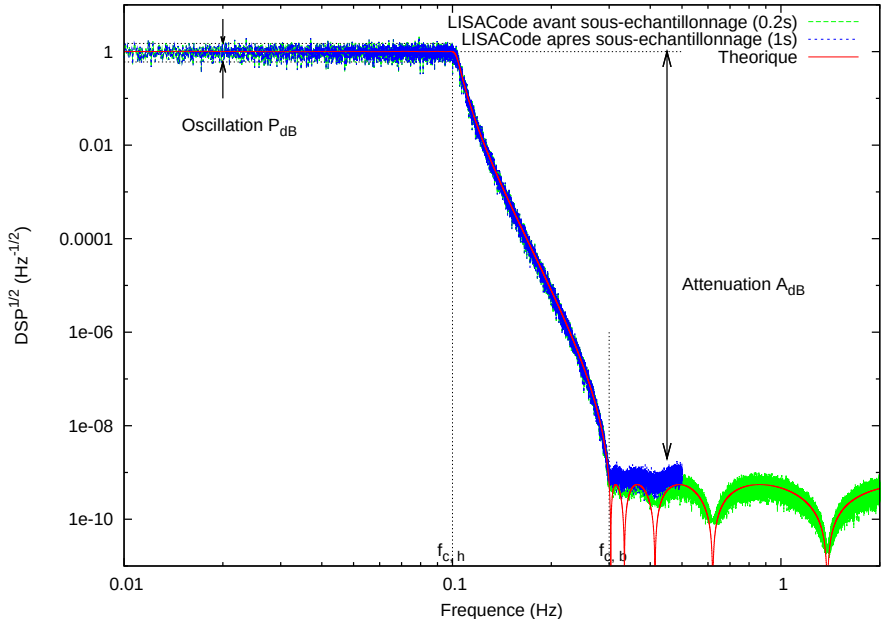

Figure 3: Réponses spectrales du filtre elliptique utilisé dans la fonction de transfert du phasemètre pour permettre un sous-échantillonnage à $\Delta t_{mesure} = 1s$ à partir de données à $\Delta t_{physique} = 0.2s$. Ses paramètres, illustrés sur la figure, sont : fréquence de coupure haute $f_{c,h} = 0.1\ Hz$, fréquence de coupure basse $f_{c,b} = 0.3\ Hz$, oscillation en bande passante $P_{dB} = 0.1\ dB$ et atténuation $A_{dB} = 180\ dB$. La réponse théorique présentant les oscillations caractéristiques d'un filtre elliptique est représentée en trait plein rouge. La DSP d'un bruit blanc échantillonné à $\Delta t_{physique}$ filtré est en tirets verts et suit la réponse théorique. La DSP du même bruit filtré après sous-échantillonnage est en pointillés bleus.

Chaque mesure du phasemètre est faite à un temps donné par l'horloge du satellite (*USO*). Si l'*USO* est bruité, son temps est décalé par rapport au temps courant. L'envoi des mesures vers la mémoire du satellite, puis vers la Terre, étant repéré par le temps courant, l'erreur des *USO* est prise en compte dans les mesures et influe sur l'efficacité de TDI, comme dans lors de la mission. Mais cette erreur est considérée uniquement comme un décalage temporel et non comme une imprécision dans la mesure du phasemètre, puisque la relation entre le bruit de l'USO et la mesure de phase n'est pas encore connue. Pour l'instant, l'imprécision due à ce bruit de l'*USO* est juste ajoutée comme un bruit dans les bruits de chemin optique (cf. partie **??** et équation (**??**)) sans tenir compte de la modélisation de l'*USO*. Mais quand les développements du phasemètre auront suffisament progressé pour que la relation entre le bruit de l'*USO* et l'erreur du phasemètre soit connue, il sera très facile de l'inclure dans LISACode.

La classe *PhoDetPhaMet* modélise donc à la fois la **circulation dans les bancs optiques** des deux faisceaux qui interférent, la **photodiode** et le **phasemètre**, c'est-à-dire l'ensemble du système de mesures de LISA. C'est pourquoi cet élément est central dans LISACode, en effet, il construit le signal de mesure, le filtre et l'envoi vers la mémoire associée aux satellites au pas de temps de mesure.

3.7 Module mémoire Memoire

La description du phasemètre précédente a exposé la construction des signaux de mesures. Le rôle du module mémoire est de faire le lien entre la modélisation du détecteur et l'application de la méthode *TDI*. Il récupère les signaux envoyés par les quatre phasemètres du satellite, les enregistre dans un fichier de sortie et les mémorise pour l'application de *TDI*.

Le module *Memoire* est composé d'une classe mère *Memory* et de deux classes filles. Le rôle principal d'un objet de type *Memory* est de mémoriser les quatre signaux de mesures pendant une certaine durée pour qu'ils puissent ensuite être utilisés par la méthode *TDI*. En effet, cette méthode, présentée dans la partie **??**, élimine le bruit laser en retardant les signaux de mesures puis en les combinant, d'où la nécessité de mémoriser les signaux, un peu comme le module *Bruit* mémorise les bruits pour permettre aux phasemètres de construire les signaux (cf. partie 3.2.1). La classe mère *Memory* gère cette mémoire par une méthode d'ajout de données, qui peut être redéfinie par les classes filles, et par une méthode d'envoi de données. A la différence des classes mères précédemment exposées, celle-ci ne définie pas seulement des bases communes mais peut être utilisée pour créer un objet à part entière ; les deux classes filles ne sont alors que des options supplémentaires sur la manière de récupérer les signaux de mesure. Cette classe gère plusieurs signaux et possède pour chacun une série de données glissante qui est un objet construit à partir de la classe *Serie* du module *Outils_Math*. Cette classe *Serie* gère un tableau glissant de la même manière que celui du module *Bruit* (cf. partie 3.2.1) et offre de nombreuses potentialités pour l'interpolation des valeurs. La méthode d'envoi, qui est appelée par le module *TDI*, interpole les signaux mémorisés pour retourner la valeur correspondant au retard [10], spécifié par *TDI*. Le choix de l'interpolation utilisée est très importante pour garantir une bonne élimination du bruit laser, mais, par ailleurs, une interpolation trop poussée augmente le temps de simulation, c'est pourquoi ce choix est présent dans les options de configuration de la simulation.

La méthode de récupération de données de la classe mère est tout simplement de stocker les signaux de mesures des phasemètres dans le tableau. Les deux classes filles offrent des versions différentes de cette méthode. La classe *MemoryWriteDisk* stocke les signaux et les enregistre dans un fichier en ASCII ou en binaire. Etant donné qu'il y a un objet mémoire associé à chaque satellite, il y a trois fichiers de sortie possibles qui possèdent chacuns cinq colonnes : $temps\ s_i\ s'_i\ \tau_i\ \tau'_i$ où i est l'indice du satellite.

[10] Le retard par rapport au temps courant de la simulation, comme toujours !

La deuxième classe fille, la classe *MemoryReadDisk*, ne récupère pas les signaux des phasemètres mais les lit dans un fichier. Cette classe est uniquement utilisée pour former l'exécutable *TDIApply*, qui permet d'appliquer *TDI* sur des signaux lus dans trois fichiers en utilisant les retards lus dans un quatrième fichier.

La mémorisation par le module *Memory* n'est pas spécialisée pour les signaux des phasemètres. En effet, le nombre de signaux stockés s'adapte aux besoins de l'objet construit à partir de la classe. Ce module peut donc être utilisé à des fins de mémorisation ou d'enregistrement de données. Il sert notamment à stocker les six valeurs de temps de parcours le long des bras.

3.8 Module *TDI*

Le module *TDI* applique la méthode *TDI* détaillée dans la partie **??** sur les signaux de mesures mémorisés, pour obtenir de nouveaux flux de données dans lesquels les bruits laser sont éliminés. Il retarde les signaux grâce aux retards mémorisés puis les combine. Ce module applique *TDI* d'une manière très générique et peut ainsi fournir le flux de données correspondant à n'importe quel générateur. Il faut bien voir que l'application de *TDI* ne relève plus du domaine de la modélisation mais de celui de la pré-analyse. La méthode présentée ici fonctionne donc de la même manière sur les signaux modélisés que sur les signaux réels !

Le module *TDI* comporte deux classes principales, *TDI_InterData* et *TDI*, ainsi qu'une classe outils, *TDITools*, qui permet d'accélérer l'application de *TDI* dans certaines conditions.

La première étape de l'application de *TDI* consiste à créer les six flux de données intermédiaires η_i et ${\eta'}_i$ correspondant aux formulations (**??**) et (**??**), présentées dans la partie **??**. Elles ramènent le problème de la réduction du bruit laser de deux bruits laser par satellite, p_i et ${p'}_i$, à un seul bruit laser par satellite, p_i. Le rôle de la classe *TDI_InterData* est de calculer ces flux intermédiaires. Elle possède six séries de données glissantes (tableau glissant de type *Serie*), c'est à dire une pour chaque flux de données. A chaque pas de temps de mesure, les nouvelles valeurs sont calculées en utilisant les mémoires des signaux des trois satellites [11] et la mémoire des temps de parcours [12], autrement dit des retards. Ce calcul doit être très précis pour réduire au mieux les bruits ${p'}_i$ des deuxièmes lasers de chaque satellite et notamment lors de l'application de l'opérateur retard D_{i+2} sur les signaux de mesures τ_{i+1} et ${\tau'}_{i+1}$ qui utilise la méthode d'interpolation de la classe *Memory*. Cette classe possède également une méthode qui permet de renvoyer une valeur en interpolant dans la série de données des flux intermédiaire[13]. Il n'y a qu'un seul objet de

[11] Plus précisément, la classe *TDI_InterData* possède un pointeur sur une liste de trois pointeurs sur des objets de type *Memory*, un pour chaque satellite, contenant chacun quatre signaux de mesures.

[12] Là aussi, la classe *TDI_InterData* possède un pointeur sur un objet de type *Memory* contenant les six valeurs de temps de parcours

[13] L'interpolation est du même type que celle de la classe *Memory* puisqu'en fait c'est celle de l'objet

type *TDI_InterData* qui est utilisé par l'ensemble des générateurs *TDI*.

La deuxième étape est effectuée par la classe *TDI*. Elle consiste en l'application des générateurs sur les flux de données intermédiaires. Cette classe applique un seul générateur à la fois : il y a donc un objet de type *TDI* par générateur. Un générateur *TDI* peut être développé en une somme de packs. Un pack est un ensemble d'opérateurs retards appliqués à un flux de données. Par exemple, le générateur $X_{1.5st}$ donné par le 6-uples de la formulation (**??**) se développe en une somme de huit packs qui est :

$$X_{1.5st} = \eta_1 + D_3\,\eta'_2 + D_3D_{3'}\,\eta'_1 + D_3D_{3'}D_{2'}\,\eta_3 - \eta'_1 - D_{2'}\,\eta_3 - D_{2'}D_2\,\eta_1 - D_{2'}D_2D_3\,\eta_{2'} \quad (16)$$

On remarque que dans ce développement, l'interprétation du générateur en deux boucles de faisceaux est nettement visible (cf. partie **??**). Afin de faciliter l'utilisation des packs dans LISACode, on exprime chaque pack par un nombre entier à partir des conventions suivantes :

- le signe du nombre correspond au signe du pack dans la sommation,
- le dernier chiffre est un nombre entre 1 et 6 qui indique le signal sur lequel s'applique les retards avec la correspondance suivante :

$$\eta_1 \to 1\,, \quad \eta_2 \to 2\,, \quad \eta_3 \to 3\,, \quad \eta'_1 \to 4\,, \quad \eta'_2 \to 5\,, \quad \eta'_3 \to 6 \qquad (17)$$

- les autres chiffres indiquent les opérateurs de retard à appliquer dans l'ordre : le chiffre des dizaines indique l'indice du bras du premier opérateur à appliquer, le chiffre des centaines, l'indice du deuxième, etc. La correspondance entre le chiffre et le bras concerné par l'opérateur de retard est donnée par le tableau 1.

Opérateur	Bras	Emetteur → Récepteur	Chiffre du pack
D_1	1	3 → 2	1
D_2	2	1 → 3	2
D_3	3	2 → 1	3
D'_1	1'	2 → 3	4
D'_2	2'	3 → 1	5
D'_3	3'	1 → 2	6

Table 1: Tableau donnant la correspondance entre les chiffres du pack et le bras concerné par l'opérateur de retard.

construit à partir de la classe *Serie*

En suivant ces conventions, le pack -5235 correspond à :

$$\begin{aligned} -5235 &\equiv -D_{2'}D_2D_3\ \eta_{2'} \\ &\equiv -{\eta'}_2\left[t-\frac{L_3}{c}\left(t-\frac{L_2}{c}\left(t-\frac{{L'}_2}{c}(t)\right)\right)-\frac{L_2}{c}\left(t-\frac{{L'}_2}{c}(t)\right)-\frac{{L'}_2}{c}(t)\right] \end{aligned} \tag{18}$$

Les chiffres de la suite de pack du générateur $X_{1.5st}$ sont alors :

$$X_{1.5st} \equiv 1,\ 35,\ 364,\ 3653,\ -4,\ -53,\ -521,\ -5235 \tag{19}$$

De la même manière, les packs des autres générateurs couramment utilisés sont :

$$Y_{1.5th} \equiv 2,\ 16,\ 145,\ 1461,\ -5,\ -61,\ -632,\ -6316 \tag{20}$$

$$Z_{1.5th} \equiv 3,\ 24,\ 256,\ 2542,\ -6,\ -42,\ -413,\ -4124 \tag{21}$$

$$P_{1.5th} \equiv 25,\ -63,\ -22,\ 66,\ 642,\ -216,\ 1463,\ -1425 \tag{22}$$

$$E_{1.5th} \equiv 542,\ 56,\ -316,\ -32,\ -144,\ 141,\ 4,\ -1 \tag{23}$$

$$U_{1.5th} \equiv 145,\ 1464,\ -5,\ -64,\ 16,\ 2,\ -6542,\ -656 \tag{24}$$

$$\begin{aligned} X_{2nd} \equiv\ & 1,\ 35,\ 364,\ 3653,\ 36524,\ 365253,\ 3652521,\ 36525235, \\ & -4,\ -53,\ -521,\ -5235,\ -52361,\ -523635,\ -5236364,\ -52363653 \end{aligned} \tag{25}$$

La classe *TDI* décompose chaque chiffre de pack en une variable de signe, une liste d'indices de retard et un indice de signal intermédiaire, η_i ou ${\eta'}_i$, qui est identique à l'indice du signal de mesure dans les développements de *TDI*, s_i ou ${s'}_i$ (cf. partie **??**). A chaque pas de temps de mesure, il calcule la valeur de chaque pack puis somme l'ensemble de ces valeurs. Le calcul de la valeur d'un pack, qui est l'application des opérateurs de retard sur le flux de données intermédiaires, doit se faire en respectant l'ordre des retards. Pour cela, le retard total est calculé par l'algorithme itératif suivant : après avoir initialisé à 0 la variable retard total, on effectue une boucle sur les retards où, à chaque itération, on ajoute au retard total le temps de parcours le long du bras pris au temps retardé du retard total, soit pour chaque itération i l'opération :

$$\Delta t_{total} = \Delta t_{total} + \frac{L_i}{c}\left(t - \Delta t_{total}\right) \tag{26}$$

L'application de cet algorithme sur le pack de l'exemple (18) donne :

1. Initialisation : Retard total à 0 : $\Delta t_{total} = 0$
2. Itération 1 : Ajout du retard du bras ${L'}_2$ pris au temps t :

$$\Delta t_{total} = \frac{{L'}_2}{c}(t)$$

3. Itération 2 : Ajout du retard du bras L_2 pris au temps $t - \frac{L'_2}{c}(t)$:

$$\Delta t_{total} = \frac{L'_2}{c}(t) + \frac{L_2}{c}\left(t - \frac{L'_2}{c}(t)\right)$$

4. Itération 3 : Ajout du retard du bras L_3 pris au temps $t - \frac{L'_2}{c}(t) + \frac{L_2}{c}\left(t - \frac{L'_2}{c}(t)\right)$:

$$\Delta t_{total} = \frac{L'_2}{c}(t) + \frac{L_2}{c}\left(t - \frac{L'_2}{c}(t)\right) + \frac{L_3}{c}\left(t - \frac{L'_2}{c}(t) + \frac{L_2}{c}\left(t - \frac{L'_2}{c}(t)\right)\right)$$

Ce retard total est utilisé pour retarder le flux de données intermédiaires η_i (ou η'_i), et ainsi obtenir la valeur du pack. A chaque itération de l'algorithme, le temps de propagation L_i/c retardé de Δt_{total} s'obtient en utilisant l'interpolation de la méthode de renvoi de valeur de l'objet de type *Memory* qui stocke les retards. La même méthode, mais utilisée à propos de l'objet construit à partir de *TDI_InterData*, est utilisée pour obtenir la valeur du pack. L'interpolation sur les retards est lagrangienne d'ordre 6 et celle sur les signaux *TDI* intermédiaires est spécifiée par la configuration mais, typiquement, on obtient une bonne élimination des bruits laser avec une interpolation lagrangienne d'ordre 20, comme on le verra dans la partie **??**.

Il y a donc par pack, à chaque pas de temps, une interpolation très précise et autant d'interpolation moins précise que d'opérateur de retard. Au total, cela représente un grand nombre d'interpolation et par conséquent un temps de calcul très important. Par exemple, pour le générateur $X_{1.5st}$, il y a 12 interpolations de retard et 6 interpolations de mesure par pas de temps, soit l'équivalent en temps de calcul d'environ 96 interpolations linéaires[14] et 264 pour le générateur X_{2nd}. Ce nombre d'interpolations est indispensable à l'élimination du bruit laser dans un cas réaliste. Mais dans le cas d'une simulation sans bruits laser ou avec sans *flexing*, il n'y a pas lieu d'avoir autant de précision dans l'application de *TDI*. Une option dans LISACode permet alors de supprimer l'interpolation sur les retards en prenant la valeur de chaque retard au temps courant. L'application de *TDI* pour le pack de l'exemple (18) est alors approximée par :

$$- D_{2'}D_2D_3\ \eta_{2'} \approx -\eta'_2\left[t - \frac{L_3}{c}(t) - \frac{L_2}{c}(t) - \frac{L'_2}{c}(t)\right] \tag{27}$$

Les temps de parcours utilisés pour les retards, sont mémorisés dans un objet de type *Memory* géré par le programme principal (cf. partie 3.11). L'exactitude de ces temps de parcours conditionne fortement l'efficacité de *TDI* (cf. partie **??**). Ils sont donc directement obtenus à partir du module d'orbitographie. Mais dans la réalité les orbites exactes ne seront pas connu avec une infinie précision et par conséquent il y

[14] On considère que le temps de calcul est proportionnel à l'ordre de l'interpolation et que l'interpolation lagrangienne d'ordre 2 est équivalente en temps de calcul à une interpolation linéaire.

aura des imprécisions sur la valeur exacte des temps de parcours. Pour simuler cette méconnaissance, on verra qu'une erreur peut être introduite (cf. partie 3.11).

Les résultats de l'ensemble des générateurs *TDI* sont enregistrés dans un même fichier à chaque pas de temps, chaque objet de type *TDI* ayant un pointeur vers ce fichier.

3.9 Configuration (module et classe Input_Data)

LISACode est un simulateur flexible qui se construit à partir des modules qui viennent d'être décrits en fonction des exigences de la simulation. Cette construction est réalisée par le module *Input_Data* qui ne contient qu'une seule classe, qui a le même nom. C'est cette classe qui fait l'interface avec les exigences de l'utilisateur par l'intermédiaire d'un fichier de configuration.

Ce fichier de configuration est détaillé dans le manuel utilisateur de LISACode **?**. Il comporte des instructions qui décrivent les différents éléments de la simulation :

- éléments temporels : pas de temps physique, pas de temps de mesure, temps de simulation,
- paramètres de *TDI* : type d'interpolation, imprécision sur la connaissance des retards, approximation ou non dans le calcul du retard total (cf. partie 3.8),
- orbitographie : temps de la position initiale, phase initiale de rotation, longueur nominale des bras, ordre d'approximation dans le calcul des temps de parcours (cf. partie **??** et 3.4),
- description du détecteur : puissance laser, activation et paramètres du filtre du phasemètre (cf. partie 3.6.3),
- enregistrement : un fichier d'enregistrement ASCII ou binaire différents pour chacun des éléments de mesures : un fichier pour les signaux de mesures pour chaque satellite, un pour les temps de parcours, un pour les signaux *TDI* et un pour les positions des satellites,
- horloges (*USO*) : offset, bruit et dérive (cf. partie 3.5),
- ondes gravitationnelles : localisation de la source [β,λ,ψ] et description (savoir si elle est monochromatique, binaire à fréquence fixe, binaire en calcul PN, lu dans un fichier, etc) (cf. partie **??** et 3.1),
- fond gravitationnel : fichier des réponses des bras (cf. partie **??** et 3.3),
- bruits : localisation du bruit (que ce soit bruit laser, bruit de masse inertielle, shot noise ou autres bruits de chemin optique) avec le repérage du banc optique et description (savoir s'il est blanc, lu dans un fichier, filtré en f, filtré en 1/f, filtré avec une dépendance en puissance et en longueur des bras, etc) (cf. partie **??** et 3.2),

- générateur *TDI* : nom et description des packs par chiffre, s'il n'est pas prédéfini[15](cf. partie **??** et 3.8).

La classe *Input_Data* lit le fichier de configuration et construit les objets au fur et à mesure. Pour les éléments temporels, les paramètres de *TDI*, l'orbitographie, la description du détecteur et les horloges, si aucune instruction n'est lue à leur sujet, une valeur par défaut leur est attribuée. Pour les autres, c'est-à-dire les ondes gravitationnelles, les bruits et les générateurs *TDI*, si aucune instruction ne les concerne, les objets ne sont pas créés. Ainsi, il est par exemple possible de faire une simulation sans bruit tout simplement en ne spécifiant aucune instruction de bruit. De la même manière, les données sont écrites dans les fichiers de sortie seulement s'il y a une instruction concernant ce fichier.

Quelques précisions sont à apporter au sujet des bruits définis par leur DSP qui sont modélisés en utilisant un ou plusieurs filtres. La DSP est définie de façon très simple par un mot clé et quelques valeurs qui sont mémorisées au moment de leur lecture. C'est une fois le fichier de configuration entièrement lu que les coefficients des filtres sont calculés à partir des valeurs mémorisés, du pas de temps physique et d'éventuels autres paramètres. Par exemple, le *shot noise* peut être décrit par un bruit qui dépend de la longueur des bras et de la puissance laser, comme le spécifie la formulation (**??**), et donc ces deux paramètres sont inclus dans le calcul des coefficients du filtre. Ensuite les objets représentant les bruits sont construits. C'est également à ce moment que les bruits de *shot noise* autres bruits de chemin optique (ABCO) sont regroupés dans un seul et même bruit dit bruit de chemin optique.

Le fichier de configuration est écrit au format ASCII simple ou au format XML. Le format ASCII simple se compose des instructions détaillées dans le manuel utilisateur **?** qui sont propres à LISACode. Son langage se veut aussi clair que possible pour permettre à l'utilisateur de rapidement prendre en main le simulateur. Le format XML est celui utilisé par le *Mock LISA Data Challenge* (*MLDC*) qui est la structure visant à coordonner l'analyse des données de LISA. Ce format, assez complexe, est utilisé par les deux autres simulateurs, SyntheticLISA et LISASimulator. LISACode peut se configurer à partir de fichier des mêmes fichiers XML, et fournir le fichier XML correspondant à une simulation, ce qui lui permet d'être pleinement intégré à la simulation de données dans la communauté LISA.

3.10 Autres modules

Les modules précédemment décrits utilisent différents outils définis dans deux modules outils. Le premier est le module *Generalites* qui contient un fichier de constantes physiques, *PhysicConstants*, et un fichier de constantes concernant LISA et LISACode, *LISAConstants*. Le second module outils est le module *Outils_Maths*, qui a déjà été évoqué à plusieurs reprises. Ce module contient plusieurs classes qui décrivent des outils mathématiques

[15]Les générateurs les plus courants sont prédéfinis dans LISACode. Seul leur nom est alors nécessaire.

tel que des couples de données (classe *Couple*), des vecteurs (classe *Vect*), des matrices (classe *Mat*), etc. La classe *Serie* décrit un tableau glissant et les méthodes d'interpolation associées. Elle est notamment utilisée par les modules *Memoire* et *TDI*. La classe *Filter* décrit et applique un filtre qui a été présenté dans la partie 3.2.2 concernant les bruits. La classe *EllipticFilter* calcule automatiquement les coefficients d'un filtre elliptique passe-pas décrit dans la partie 3.6.3 sur le phasemètre puisque c'est essentiellement pour sa fonction de transfert que ce filtre est utilisé. Enfin un sous-programme en C, *randlib*, écrit par Barry W. Brown et James Lovato, gère la génération de nombre aléatoire **?**.

3.11 Applications

A partir de l'ensemble des modules qui ont été présentés et qui forment une sorte de bibliothèque, il est possible de construire le simulateur LISACode, bien entendu, mais aussi plusieurs applications, ou exécutables.

L'application principale est le simulateur LISACode dans son ensemble. Il est créé à partir d'un programme principal qui possède les variables temporelles, un objet de type *Input_Data* qui configure la simulation, la liste des mémoires des satellites, le fichier d'enregistrement des signaux *TDI*, une mémoire sur les temps de parcours utilisés dans l'application de *TDI*, une mémoire pour les positions des satellites, un objet de type *LISA* qui représente l'ensemble du détecteur, un objet de type *TDI_InterData* qui calcule les signaux *TDI* intermédiaires et une liste d'objets de type *TDI* qui représente les générateurs.

Ce programme principal qui effectue l'avancement temporel au pas de temps de mesure, a une progression séquentielle en trois phases :

1. Il configure le simulateur avec l'objet de type *Input_Data* qui interprète le fichier de configuration. Puis il initialise toutes les variables et les objets. L'initialisation de certains objets nécessite un fonctionnement à vide sans avancement temporel pour stabiliser les filtres notamment.

2. La simulation de LISA est en fonction et fournit des données, au pas de temps de mesure, aux mémoires des satellites et des temps de parcours. Mais *TDI* n'est pas encore appliqué car il n'y a pas assez de signaux de mesures mémorisés pour pouvoir appliquer les opérateurs de retard.

3. La simulation de LISA poursuit son fonctionnement et la méthode *TDI* est appliquée et fournit les flux de données correspondant aux générateurs.

Dans la mémorisation des temps de parcours, une erreur peut être ajoutée. Elle permet de simuler la méconnaissance des retards dans l'application de *TDI* et d'estimer les limites acceptables de celle-ci comme on le verra dans la partie **??**.

Il existe deux exécutables secondaires basés sur les mêmes modules. Le premier, *DnonGW*, permet de calculer les signaux gravitationnels, c'est-à-dire les variations relatives de fréquence induites par des ondes gravitationnelles sur les bras de LISA. Il possède

une liste des ondes gravitationnelles, un objet de type *TrFctGW* qui calcule le signal gravitationnel et le fichier de sortie des signaux ainsi que différents autres fichiers de sortie. Il possède également un objet de type *Input_Data* qui crée les ondes gravitationnelles et spécifie les paramètres de simulation à partir d'un fichier identique à celui utilisé pour LISACode mais, cette fois, seules les ondes gravitationnelles sont vraiment utilisées.

Le deuxième exécutable secondaire, *TDIApply*, permet d'appliquer *TDI* sur des signaux de mesures enregistrés dans des fichiers en utilisant un fichier de temps de parcours pour obtenir les retards. Il possède le module mémoire, le module *TDI* et un objet de type *Input_Data* qui spécifie les générateurs *TDI* ainsi que les fichiers d'entrées et de sorties. Les fichiers d'entrée des signaux de mesures et des temps de parcours sont dans le même format que ceux produits par LISACode. Ainsi on peut appliquer de nouveaux générateurs *TDI* sans refaire de simulation complète. Mais on peut également utiliser ce programme pour appliquer *TDI* sur les signaux de mesures réels que fournira la mission !

Il existe aussi un exécutable de test par module qui permet de tester son fonctionnement et de faire des applications rapides.

La gestion de l'ensemble de ces modules et exécutables est basée sur le système automake. Il permet de créer une distribution de LISACode qui peut s'installer sous les systèmes UNIX et Mac comme une application UNIX standard.

LISACode est un simulateur scientifique de LISA qui tient son réalisme d'une organisation proche de celle du détecteur. Son architecture flexible lui permet d'aborder la simulation aussi bien d'un point technologique que d'un point de vue traitement du signal. De plus sa simplicité d'utilisation et sa compatibilité lui permettent d'être utilisé par toute la communauté scientifique.