Centre national de la recherche scientifique

Observatoire de Paris

Stage de fin d'étude

Développements logiciels autour de GDL

Auteur :
Nodar Kasradze

Responsable :
M. Alain Coulais

Université Paris 8

Année scolaire 2012-2013

Table des matières

Remerciements

Résumé

Mon stage s'est déroulé au sein de L.E.R.M.A. (Laboratoire d'Etudes du Rayonnement et de la Matière en Astrophysique) – Observatoire de Paris. Son but a été de contribuer au développement de GDL pour atteint un niveau de maturité et de facilité d'utilisation pour assurer la préservation à long terme des capacités d'analyse pour de nombreuses expérimentations.

Ce stage était principalement destiné à la mise en place des :

1) fonctionnalités manquante,

2) correction des bugs connu,

3) tests de régression et performance.

Ce stage nécessitait une connaissance solide de langage C++, car le projet contient plus de 120 000 lignes et est écrit en C++. Je aussi eu à apprendre la syntaxe IDL/GDL pour pouvoir écrire des tests qui vont assurer la qualité des nouvelles fonctionnalités.

Introduction

La première année de master informatique à l'Université Paris 8 Vincennes-Saint-Denis se termine par un stage d'une durée minimum de 3 mois. J'ai choisis de réaliser ce stage au seine du LERMA (Laboratoire d'Études du Rayonnement et de la Matière en Astrophysique) – Observatoire de Paris, situé à Paris 14ème. Les chercheurs de ce laboratoire travaillent avant tout à faire avancer le front des connaissances dans plusieurs axes de l'astronomie, notamment : la formation et l'évolution des galaxies, la formation de certaines étoiles, la chimie de la poussier interstellaire. Du logiciel libre est utilisé largement et est aussi produit : par exemple pour ALMA et pour GDL destiné à être en libre accès pour tout le monde.

C'est ainsi que du 13 mai 2013 au 14 août 2013, je travaille sur un logiciel nommé GDL. Ce logiciel permet de faciliter le traitement des données. Ce projet utilise plusieurs bibliothèques open source, ce qui diminue le travail à réaliser et préserve l'utilisation de code testé pendant plusieurs années par une communauté ouverte.

Ma première approche sur ce projet a été de consacrer une première partie brève du stage à prendre en main les librairies utilisée tout en faisant les diverses installations qui me seront nécessaires pour la conception sur mon poste (GSL, Eigen, PLPlot, GraphicsMagick, ...). Ensuite je prévoie une première ébauche de conception me permettant mieux d'assimiler les bibliothèques, suivi d'une étude approfondie de besoin qui précède la conception spécifique de GDL.

Le travail effectué fut aussi réalisé à trois, avec mon tuteur de stage Alain Coulais et Mouadh Ayad, étudiant à l'Université Paris-Sud, IUT de Cachan et lui aussi stagiaire au LERMA travaillant sur un sujet complémentaire au mien, notamment l'élaboration d'une fonction SMOOTH et suite de tests de régression. En plus les contacts par courriel électronique avec Marc Schellens et d'autres membres de communauté. Le fait d'avoir travaillé en collaboration a renforcé le cadre professionnel de ce stage car il est aujourd'hui impossible de travailler seul sur des projets de grande ou même petite envergure.

Dans la première partie de ce rapport je vais décrire le lieu du stage et l'environnement de travail. Après je vais présenter le cahier de charge et le but du stage. Dans la partie prochaine je vais expliquer le travail que j'ai réalisé pendant ces trois mois, les difficultés que j'ai rencontré, comment j'ai choisi de les résoudre et les résultats obtenus. Pour conclure le rapport, je résume les apports de ce stage, aussi bien personnels que professionnels.

1 Le cadre de travail

1.1 Observatoire de Paris

L'Observatoire de Paris est un Grand établissement sous tutelle du ministère de l'Enseignement supérieur et de la Recherche. Il a le statut d'EPSCP (Établissement public à caractère scientifique, culturel et professionnel). Ses trois missions principales sont : 1) la recherche, en contribuant au progrès de la connaissance de l'Univers par l'acquisition de données d'observation, le développement et l'exploitation des moyens appropriés, l'élaboration des outils théoriques nécessaires, 2) la formation initiale et continue, 3) la diffusion des connaissances. L'Observatoire de Paris abrite l'École doctorale Astronomie et Astrophysique d'Île-de-France.

L'Observatoire de Paris compte un grand total d'environ 600 emplois permanents. Du coté MESR - 89 CNAP, 10 EC, 2 PRAG et 232 personnels de soutien ; 248 titulaires du CNRS y sont affectés. L'Observatoire a le statut d'université et dispense un enseignement universitaire de haut niveau en astronomie et astrophysique allant du master au doctorat, avec possibilité de formation à distance, 245 étudiants y sont inscrits (ce chiffre intègre les étudiants des universités partenaires) ; par ailleurs 35 enseignants-chercheurs d'autres établissements de l'enseignement supérieur et 9 personnels divers travaillent dans les laboratoires de l'établissement ; l'établissement emploie 42 contractuels dont 11 sur postes vacants, 19 sur contrat CNRS, le reste sur budget de l'établissement ; son budget annuel hors salaires est de 20 M€ et sa masse salariale est estimée à 35 M€.

		GEPI	LERMA	LESIA	LUTH	SYRTE	IMCCE	Présidence	USN	HORS LES MURS	TOTAL
CNAP	Astronomes	10	4	17	5	4	3	1	0	2	46
	Astronomes-adjoints	9	5	15	1	5	5	0	1	2	43
Sous total CNAP		19	9	32	6	9	8	1	1	4	89
CNRS	Directeurs de recherche	2	8	13	8	3	1	0	0	0	35
	Chargés de recherche	3	3	11	9	11	4	0	0	0	41
Sous total CNRS		5	11	24	17	14	5	0	0	0	76
CNU	Professeurs	1	1	4	2	0	2	0	0	0	10
	Maîtres de conférence	2	7	8	3	2	5	0	0	0	27
Sous total CNU		3	8	12	5	2	7	0	0	0	37
TOTAL		27	28	68	28	25	20	1	1	4	202

FIGURE 1 – Répartition des effectifs (chercheurs et EC) par composante et par corps

Cette institution représente le plus grand pôle national de recherche en astronomie. Il a été fondé en 1667. Près d'un tiers (30 %) des astronomes français y poursuivent leurs recherches au sein de sept laboratoires situés sur trois campus (Paris fondé en 1667 par Louis XIV, Meudon fondé en 1876 par Jules Janssen dans des bâtiments du château de Meudon et Nançay construit en 1953 sur un terrain de l'ENS).

Il existe un partenariat fort entre l'Observatoire et le CNRS/INSU car les laboratoires de recherches sont tous des "Unités mixtes de recherche" et, souvent, en rattachement secondaire à d'autres universités scientifiques.

Les activités de recherche de l'Observatoire de Paris sont structurées autour de :

- 5 UMR associées au CNRS :
  - GEPI : Galaxies, Étoiles, Physique et Instrumentation
  - LERMA : Laboratoire d'Études du Rayonnement et de la Matière en Astrophysique
  - LESIA : Laboratoire d'Études Spatiales et d'Instrumentation en Astrophysique
  - LUTH : Laboratoire Univers et Théories
  - SYRTE : SYstèmes de Référence Temps Espace
- 1 institut qui est aussi une UMR CNRS :
  - IMCCE : Institut de mécanique céleste et de calcul des éphémérides
- 1 unité de service et de recherche :
  - La station radioastronomique de Nançay
- 1 laboratoire de recherche associé ayant pour tutelle principale l'Université de Paris-Diderot, Paris 7 :
  - APC : AstroParticule et Cosmologie
- 1 unité mixte de service, créée en 2002, qui regroupe les services communs et centraux.
- 1 unité de Formation et Enseignement

1.2 Matériels utilisés

La DIO (Division Informatique de l'Observatoire) est un service chargé de tous les aspects informatiques mutualisés de l'Observatoire de Paris : système et réseau, calcul, stockage ainsi que du système d'information, de la bureautique des services centraux et de l'infrastructure technique de l'Observatoire virtuel.

Pour accéder aux services numérique géré par la DIO un compte LDAP m'a été créé. Via ce compte je pourrais aussi accéder au courriel electronique (@obsmp.fr) et au réseau sans-fils interne. L'accès à cluster est possible aussi bien d'extérieur de réseau que d'intérieur. Ce compte LDAP permet aussi d'atteindre le réseau interne via un des deux "firewall" (physiquement 1 à Meuden et 1 à Pais).

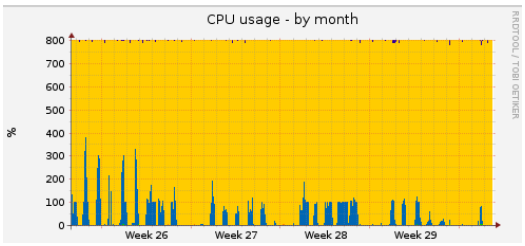
Le travail à été distribué entre une machine personnel et les serveurs LERMA :

- ARAMIS
  un serveur sous CentOS Linux version 5.9,
  avec 8 cœurs (deux Intel® Xeon® CPU X5450 @ 3.00GHz) 64 bits
  et 16 Go de mémoire vive.

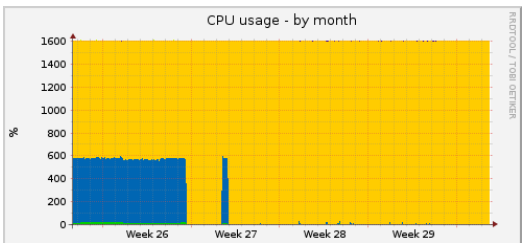
- M2PARIS
  un serveur sous Debian Linux version 7,
  avec 16 cœurs (deux Intel® Xeon® CPU L5520 @ 2.27GHz) 64 bits
  et 12 Go de mémoire vive.

- MEDUSA
  un serveur sous CentOS Linux version 5.9,
  avec 48 cœurs (quatre AMD Opteron® 6176 SE @ 2.3GHz) 64 bits
  et 128 Go de mémoire vive.

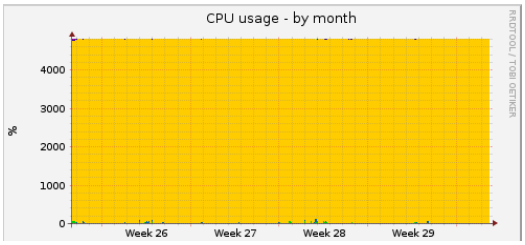
- HAKA
  une machine personnelle sous Ubuntu Linux version 12.04.2,
  avec Intel® Core®2 CPU 6400 @ 2.13GHz 32 bits
  et 2 Go de mémoire vive.



(a) ARAMIS



(b) M2PARIS



(c) MEDUSA

FIGURE 2 – La charge des serveurs pour le mois de juillet 2013 vu par l'outil Mumin

La machine personnelle (HAKA), je l'utilisais pour connecter aux autres machines plus puissantes. Compiler un projet, tel que GDL qui contient plus de 120 000 lignes de codes sur un machine comme HAKA va prendre environ 10 minutes, quand sur un serveur comme ARAMIS la compilation va prendre au maximum 2 minutes quand la machine est partiellement chargée.

La charge des serveurs a été un des raisons d'avoir plusieurs machines à disposition, sur la Figure 2 on voit que la machine ARAMIS est partiellement chargé (Figure 2a), la machine M2PARIS est chargé de manier plus permanent (Figure 2b) et la charge de la machine MEDUSA est négligeable (Figure 2c). Le fait d'avoir en disposition des machines qui sont chargé a ses avantages, bien sur la majorité de travail est fait sur une machine avec la charge insignifiant pour diminuer le temps de compilation. Dès que la correction de code est terminé il faut tester le logiciel dans environnements variants pour savoir son comportement. Dans un environnement chargé a été trouvé le problème avec les nombres des threads de OpenMP défini dans GDL (plus de détailles dans le chapitre 3.3 Nombre de threads sur page 20).

L'autre a été l'exigence de tester son compatibilité avec systèmes d'exploitations et architectures variés.

1.3 Logiciels utilisés

Les systèmes d'exploitation que j'ai utilisés pendant le stage c'etais les différents distributions de Linux et parfois Mac OS X. Pendent le stage je travaillais uniquement en mode commande. Les plus utilisés sont cités ci-dessus :

GNU Compiler Collection (GCC)

Crée en 1987 **GCC** est un ensemble de compilateurs portés sur les majorités de systèmes d'exploitation (Linux, Unix, Windows etc...) et de microprocesseurs (x86, AMD64, ARM, SPARC etc...). Il a été créé par le projet GNU comme un compilateur de langage C, mais après quelques ans de développement il est devenu beaucoup plus fonctionnel que juste un compilateur.

Aujourd'hui **GCC** est très extensible et adaptable aux besoins des utilisateurs, il contient aussi les compilateurs pour nombreux langages de programmation, tels que C, C++, Objective-C, Java, Ada, Fortran et d'autres langages, plus ou moins connus. **GCC** est le compilateur fourni avec nombreux systèmes d'exploitation comme Linux, BSD, Mac OS X ou BeOS. La plupart des logiciels libres sont compilée avec **GCC**, car en plus d'être performant, il est libre au sens de la licence GNU GPL.

D'après tout ça on comprend pourquoi on a choisi **GCC** comme compilateur pour le projet GDL.

GNU Debugger (GDB)

GDB a été créé en 1988 par Richard Stallman comme le débogueur standard du projet GNU. Comme pour **GCC** il existe des ports de **GDB** sur différent types de microprocesseurs et systèmes d'exploitation, il supporte différents langages de programmation et il est distribué sous les termes de la licence GNU GPL.

GDB propose de nombreuses fonctionnalités pour tracer et contrôler l'exécution d'un programme informatique pas à pas. On peut surveiller ou modifier (si il y a besoin) une variable interne, on peut aussi faire un appel à fonction, indépendant du comportement prévu dans le programme. **GDB** permet le débogage distant selon gdbserver, qui est lancé sur la même machine que le programme et le débogage est effectuer d'une autre. Cette technique est souvent utilisée pour déboguer programme sur un système embarqué.

(a) GCC

(b) GDB

(c) Valgrind

Figure 3 – Les logos

Valgrind

L'une de plus grandes difficultés de C++ est de gérer correctement la mémoire, alors pour diminuer les problèmes liés à la fuite de mémoire (l'absence de désallocation de l'espace utilisé, double désallocation de l'espace utilisé etc...) j'ai utilisé **Valgrind**.

A l'origine il a été crée comme un débogueur de mémoire pour Linux sur architecture x86, mais depuis il est devenue un plate-forme d'analyse dynamique pour les contrôleurs de la mémoire et "profilers". Il est disponible pour les systèmes d'exploitation Linux, Mac OS X et Android sur les microprocesseurs x86, x86-64, PowerPC et ARM. Grâce à sa licence GNU GPL v2, il existe des ports non-officiels sur d'autres systèmes et architectures.

Essentiellement **Valgrind** est un machine virtuelle utilisant la technique de compilation à la volée, le programme d'origine ne s'exécute pas directement sur un processeur, avant de ce procédure il est traduit dans une représentation intermédiaire simplifié, les instructions sont ajoutées par un outil ("memcheck", "profiler" ou un outil externe) et finalement ces instructions sont traduites dans un code machine. A cause des procédures de traduction sous **Valgrind** l'exécution d'un programme est au moins 5 fois plus lent.

Il est possible de combiner **Valgrind** avec **GDB** pour profiter des avantages de ce deux excellents outils.

2 Détails du sujet de stage

2.1 Présentation d'IDL

IDL - Interactive Data Language est un langage interprété sous licence propriétaire. Créé en 1977 il a attiré l'attention des astronomes, médecins et d'autres scientifiques qui ont eu besoin d'un langage assez simple à apprendre et assez puissant pour traiter les données massives. La popularité d'**IDL** augmentait très vite à cause de ses fonctionnalités très pratiques pour les chercheurs, qui n'ont pas été programmeurs, mais ils ont eu besoin d'un outil pour faire des calculs scientifique. Entre ces fonctionnalités on trouve :

- typage dynamique, le type est attribué automatiquement pendant la définition de sa valeur (il ya a une quinzaine de types) ;
- capacités d'un langage de programmation orientée objet ;
- interprétation par machine virtuelle ;
- affichage de tracés (1D, 2D, 3D) sans utilisation des bibliothèques graphiques externe ;
- est un langage vectorisé, les opérations sur scalaires sont généralisée pour pouvoir les utiliser sur les vecteurs, matrices et les tableaux à une dimension supérieure ;
- multitâche, certaines fonctions et procédures sont parallélisés.

Grâce à ces avantages et un disponibilité sur les systèmes d'exploitations les plus utilisé y compris Windows, Mac OS X, Linux et Solaris, aujourd'hui il existe de nombreuses bibliothèques écrites complètement en syntaxe **IDL**. Ce sont les modules spécifique à un domaine, qui ont été crée par les scientifiques pour simplifier l'utilisation de langage **IDL** dans ses domaines. Par exemple : *IDL Astronomy User's Library* ou plus court *AstroLib* crée par Wayne Landsman. Cette bibliothèque contient des procédures fréquemment utilisée par les astronomes, notamment pour relire certaines formats de données (FITS).

IDL est un langage de programmation dont l'interpréteur du langage (ou compilateur) est propriétaire, ce qui signifie qu'il n'est pas libre. Cela signifie aussi que les scientifiques n'ont pas accès au code source et qu'il ne peuvent ouvrir qu'une quantité limitée de session au travers d'un système de jetons.

Le système de jetons dans **IDL** est contraignant. Déjà, l'abonnement pour un jeton par an coute plus de 1000 €. En plus, il y a un nombre limité d'ouverture d'**IDL**, car à chaque démarrage du logiciel, ce dernier va réserver des jetons disponibles sur un serveur. Cela implique aussi que l'utilisateur doit rester connecté au serveur détenteur de jetons, ce qui peut très vite s'avérer

compliqué en déplacement professionnel durant des présentations ou lors de démonstration pendant des conférences.

Mais ce qui pose encore plus de problèmes aux scientifiques, c'est la pérennité du logiciel. En effet, ne pas avoir les codes source à disposition implique le fait que si la société en charge de garder le logiciel à jour décide de cesser ses activités, personne ne pourra continuer le développement. Dans le passé, la NASA et l'ESA ont contraint l'éditeur, qui voulait ne garder que Windows, à poursuivre le developpement sous Mac OS X et Linux. Or, la technique évolue tous les jours, et les logiciels tels qu'est **IDL** se doivent de la suivre. De plus, s'il n'est plus possible d'utiliser **IDL** dans le future, les nombreuse bibliothèques écrit en syntaxe **IDL** deviendront inutile et il faudra recoder tous les algorithmes dans un autre langage de programmation, ce qui serait très long et extrêmement cher.

2.2 Présentation de GDL

GDL - GNU Data Language est un compilateur libre compatible avec la syntaxe **IDL**. Il a été créé par Marc Schellens en 2004 et il est développé par une communauté internationalle de volontaires. **GDL** a été créé pour bénéficier des avantages d'**IDL** en ajoutant la liberté de compilateur pour assurer la pérennité de logiciel lui même et les bibliothèques tierce écrit en syntaxe **IDL**. L'interpretation de syntaxe est basé sur ANTLR.

GDL est déjà une alternative viable couvrant la plupart des fonctionnalités d'**IDL** et peut être utilisé pour compiler large "pipelines" de codes et bibliothèques en syntaxe **IDL**, lire les donnée de divers formats utilisées par les scientifiques (comme FITS, HDF, NetCDF, XDR), faire des calculs complexes et retourner un résultat exacte. Grâce aux packagers les versions précompilées et pré-configurées sont mise à disposition pour les distribution de Linux les plus populaires (Debian, Ubuntu, Fedora, Gentoo, ArchLinux...), FreeBSD et Mac OS X. Pour les utilisateurs plus expérimentés une version la plus récente est disponible par CVS, qui peut être compilée presque sur tous les systèmes basés sur Unix, la compatibilité avec Windows n'est pas assurée, mais tous les nouveau fonctionnalités prennent en compte la compatibilité avec ce système et les fonctionnalités déjà présent dans **GDL** vont évoluer pour être compatible avec tous les systèmes possibles.

Le choix de bonnes librairies externes et de bons algorithmes internes, l'usage de OpenMP, donnent une bonne performance globale à GDL. Sur les multi-cœurs, les gaines attendus sont généralement constatés. Les tests

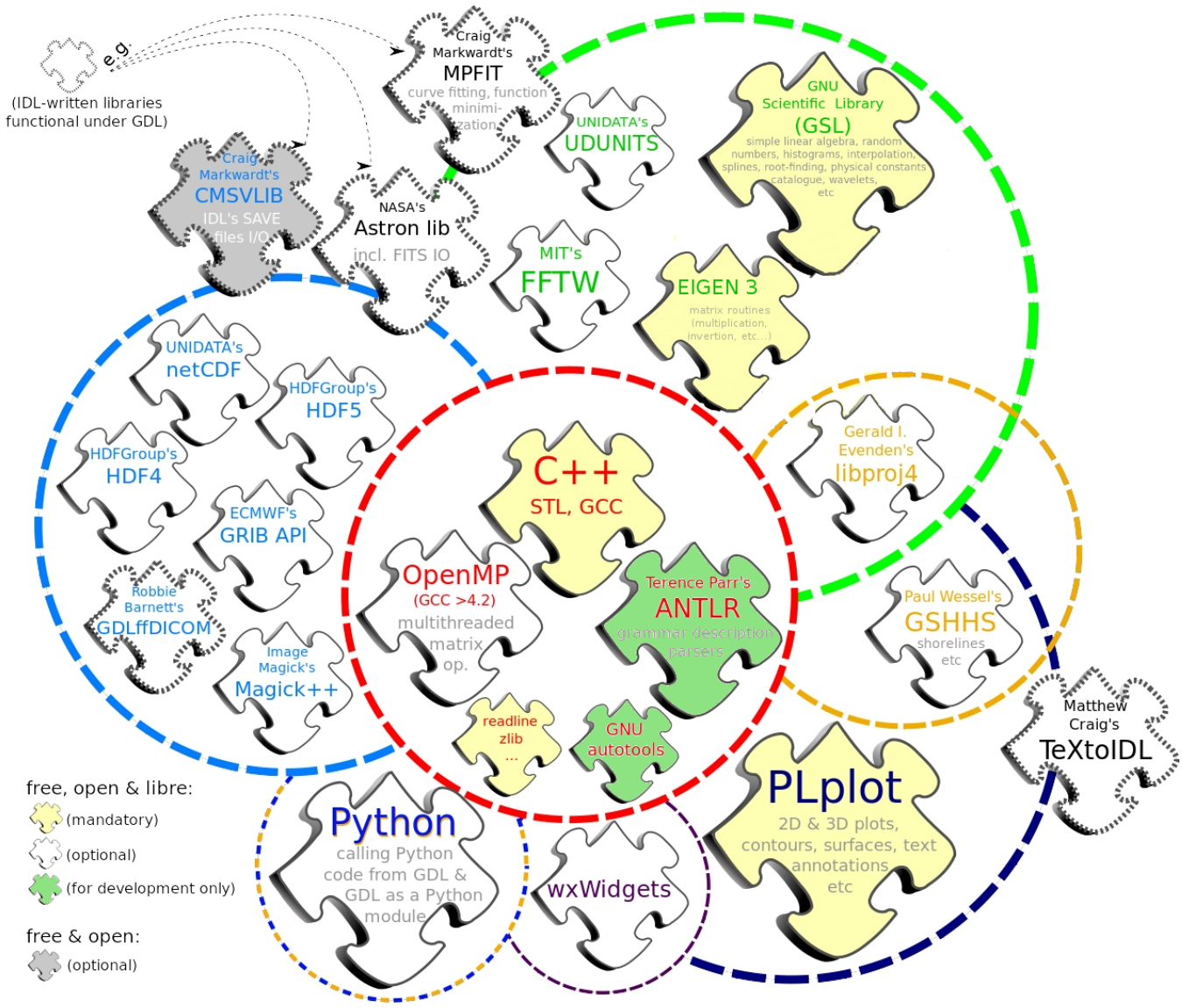


FIGURE 4 – Dependences de GDL

de performances actuels montrent que les calcules avec **GDL** sont aussi vite (parfois plus vite), que les calculs avec son adversaire propriétaire et ce, en respectant le gain par nombre de cœurs. L'ultime déficience connue dans le benchmarck de référence (IDL TIME_TEST_4) est SMOOTH, sujet de seconde stagiaire.

Le plus grand inconvénient de **GDL** est l'absence de plusieurs fonction-s/procédures présentes dans **IDL**, alors ma mission a été de diminuer le

nombre de fonctions/procédures manquantes et de tester ces nouvelles fonctionnalités en comparant les performances de **GDL** et d'**IDL**.

Le projet **GDL** utilise la licence GNU GPL v2/v3. Les bibliothèques tierces utilisée dans ce projet sont uniquement des bibliothèques libres (la notion d'un logiciel libre/licence libre est définie dans le chapitre 2.3 Logiciel libre sur page 15) avec la licence GPL ou un autre, compatible avec GPL. Une exception unique concerne une fonctionnalité importante d'un format propriétaire de fichier interne à **IDL**. Il se trouve qu'une librairie externe open source à la licence incompatible avec la GNU GPL permet de gérer ce format. Il revient à l'utilisateur d'ajouter lui même ce code.

2.3 Logiciel libre

La notion de Logiciel Libre (Free Software) est inventée par Richard Stallman dans le début des années 1980. Une première définition est publiée en 1986 par FSF (Free Software Foundation), une organisation américaine à but non lucratif, dont la mission mondiale est la promotion du logiciel libre et la défense des utilisateurs. Cette définition se traduit par une tente d'une licence définissant les droits et les devoirs des utilisateurs en s'appuyant sur le droit d'auteur (Copyright). Rarement, FSF modifie la notion de Logiciel libre afin de la clarifier ou pour résoudre des questions portant sur des points difficiles. Les deux principales licences GNU GPL sont la version 2 et version 3.

Selon la dernière définition, un programme est un logiciel libre si, en tant qu'utilisateur de ce programme, vous avez les quatre libertés essentielles :

liberté 0
: la liberté d'exécuter le programme, pour tous les usages ;

liberté 1
: la liberté d'étudier le fonctionnement du programme, et de le modifier pour qu'il effectue vos tâches informatiques comme vous le souhaitez ; l'accès au code source est une condition nécessaire ;

liberté 2
: la liberté de redistribuer des copies, donc d'aider votre voisin ;

liberté 3
: la liberté de distribuer aux autres des copies de vos versions modifiées ; en faisant cela, vous donnez à toute la communauté une possibilité de profiter de vos changements ; l'accès au code source est une condition nécessaire.

Pour protéger la [non]liberté d'un logiciel et les droit d'utilisateur il existe plusieurs type de licences. La FSF classe les licences en trois catégories :

- les licences libres compatibles avec GPL ;
- les licences libres non compatibles avec GPL ;
- les licences non libres/propriétaires.

FIGURE 5 – Logo de GPL version 3

GNU General Public License

La Licence Publique Générale GNU (GNU GPL ou GPL) est une licence des logiciels libres. Elle a été créé par Richard Stallman en 1989 pour permettre au grand nombre de projets de partager leur code source. C'est la licence le plus utilisée dans le monde de logiciels libres. La dernière version est "GNU GPL version 3" publiée en 2007. Ses termes autorisent l'utilisateur de logiciel sous licence GPL à : distribuer ce logiciel, le modifier, distribuer la version modifié. L'utilisateur n'est pas autorisée à : changer la licence, distribuer ce logiciel sans code source, commercialiser ce logiciel.

Ce ne signifie pas qu'on peut pas faire de l'argent avec les logiciels libres. L'entreprise multi-nationale Red Hat avec chiffre d'affaire d'un milliard de dollars est dédiée aux logiciels libre et Open Source, qui a basée son modèle d'affaires sur la vente des services et supportes pour les logiciels libres, notamment le noyau Linux et le système d'exploitation RHEL (Red Hat Enterprise Linux).

Berkeley software distribution license

La licence BSD est une licence libre compatible avec la GPL. Ses termes sont moins restrictives que celles de GPL. Un logiciel sous licence BSD en différence de GPL peut être utiliser dans un autre projet commercial.

End User License Agreement

Contrat de Licence Utilisateur Final est un licence non libre/propriétaire, ses termes sont définis par l'entreprise publiant le logiciel. Il est utilisé pour mettre les différent restrictions (restriction de distribution ; restriction sur le nombre d'utilisateurs, qui peuvent utiliser ce logiciel).

3 Travail réalisé

3.1 Les premiers pas

Mon stage a commencé avec l'insertion dans le projet GDL. Ma première tâche a consisté à compiler le code source de GDL sur les différentes machines (HAKA, ARAMIS...) avec les différents options (en activant ou non certaines bibliothèques externes facultatives). Pour chaque machine la procédure de compilation a été différente, car les versions de logiciels/bibliothèques installés ont été différentes ou sur certaines machines les logiciels/bibliothèques nécessaires pour la compilation n'ont pas été installé du tout. Les deux cas veulent dire que ça va prendre encore plus du temps de compiler un projet sur de telles machines. Trouver une version de logiciel qui va être compatible avec des logiciels/bibliothèques déjà installé ce n'est pas un tâche triviale à fin que le projet GDL contient 5 bibliothèques obligatoires et plus d'un dizaine de bibliothèques optionnelles.

Le build de projet GDL est effectué par *Autotools*, mais dans la version suivante 0.9.4 *CMake* va remplacer *Autotools*, ce changement est déjà accompli dans la version CVS du projet. Pour mieux comprendre la raison, pourquoi le projet a abandonné *Autotools* voici le bref description de ces deux outils :

Autotools
: Autotools (ou GNU build system) est un ensemble des projets GNU utilisés pour le build de projet sur différents systèmes d'exploitations basés sur Unix ; ces outils peuvent être utilisés sur Windows, mais c'est une procédure très restrictive (compilation est possible seulement avec GCC) et prend plus de temps. Il est basé sur shell script, alors il ne nécessite pas l'installation des outils *Autotools* pour faire un build. Ces outils sont très populaire dans le monde Linux. Les outils *Autotools* ont de nombreux inconvénients à cause de quoi les développeurs cherchent d'autres solutions. Les problèmes qui repoussent des programmeurs sont :
  - la syntaxe très compliquée, qui rend difficile l'écriture d'un script de configuration ;
  - large nombre des outils composant avec les syntaxes différents ;
  - les messages d'erreurs difficiles à comprendre ;
  - création de larges scripts de configuration même pour un projet basique ;
  - difficilement extensible, difficulté à rajouter des fonctionnalités non

standard.

CMake

Cross Platform Make ou simplement *CMake* est un moteur de production créé dans le début des années 2000 pour être compatible avec divers plate-formes (à peu près tous les systèmes basée sur Unix, Windows : Borland, Visual C++, cygwin...) et divers environnements de développement (KDevelop, XCode, Visual Studio...). Il devient un outils de plus en plus utilisé à cause de ces nombreux avantages :

- ne dépend que d'un compilateur C++ quelconque ;
- syntaxe très simple à apprendre ;
- il génère un seul Makefile pour touts les plate-formes supportées ;
- les messages d'erreurs aident à comprendre le problème ;
- supporte le plate-forme des tests ;
- la performance améliorée par rapport aux *Autotools*.

Au début de projet GDL l'utilisation de *CMake* a été impossible à cause de sa licence non compatible avec GPL (GDL utilise la licence GPL), mais la migration sur la licence BSD a aidé à surmonter l'obstacle de non compatibilité. Le point le plus faible de *CMake* représente son documentation.

Par exemple, si on veut compiler sur ARAMIS la version CVS du projet, on peut utiliser la suite des commandes suivants (au debut on suppose qu'on est dans la repertoire *~/sources/gnudatalanguage/gdl/*) :

```
mkdir compil build
cd compil
cmake .. -DHDF=OFF -DPSLIB=OFF
   -DPLPLOTDIR=~/sources/plplot-5.9.6/build/
-DCMAKE_INSTALL_PREFIX=~/sources/gnudatalanguage/gdl/build/
make -j 8
make check
```

3.2 Fonction d'inversion de matrice

3.2.1 Contexte

Ma première mission d'écriture de code a été le développement de fonction d'inversion de matrice. Pour ce moment la fonction d'inversion de matrice (INVERT) était déjà présente dans le projet GDL, mais elle a été écrite en utilisant la librairie GSL (GSL représente des outils de calculs numériques en

mathématiques appliquées, elle fait partie du projet GNU et est distribuée selon les termes de la licence GNU GPL), alors j'ai dû le réécrire en utilisant une autre librairie qui s'appelle Eigen (librairie de C++ contenant des templates qui implémentent l'algèbre linaire et des opérations sur les matrices, sous la licence libre MPL2, qui représente l'hybride de BSD et GPL).

3.2.2 Realisation

La philosophie de GDL est directement inspirée du monde libre, ainsi, si un code existe déjà et qu'il est possible de l'intégrer dans le projet grâce à sa licence, il est inutile de perdre du temps à le réécrire. On peut aussi choisir entre plusieurs librairies exterieurs en fonction de criters : simplicité, taille, performance, évolution. C'est pourquoi, plutôt que de coder l'algorithme d'inversion de matrice j'ai préféré utiliser les routines existantes et largement testées.

Dans la fonction d'inversion, la matrice est donné sous les types du langage C/C++. Pour mieux traiter les données Eigen a ses propres types de données et pour convertir les donnée existant de type non-Eigen à un type interne d'Eigen on a le mapping. La classe Map de bibliotheque Eigen contient des routines de mapping. Ces routines sont trés simple à utiliser et ils sont tres performant.

Exemple de mapping :

```
float array[rows*cols];
Map<MatrixXf> m(array,rows,cols);
```

3.2.3 Tests

Avant de publier dans le CVS la nouvelle fonction d'inversion d'une matrice il faut le bien tester. Les tests sont nombreaux et divers :

- tests de justess,
- tests de types,
- tests de correspondance avec la syntaxe IDL,
- tests de performance.

Ces tests sont necessaire pour assurer la qualité de projet et d'eviter les problemes en future.

3.3 Nombre de threads

3.3.1 Context

Quand la fonction d'inversion de matrice utilisant la librairie Eigen (INVERT_EIGEN) a été prête, ses benchmarcks sur une machine chargé ont montré un résultat complètement différent de celui de la machine non chargé. Par rapport à IDL la même fonction d'inversion de matrice sur une machine non chargé a été plus vite pour touts les types, mais sur la machine qui a été chargé (même s'il y a eu un seul cœur occupé et 15 disponibles) pour certaines types la performance a baissé à 10 fois ou même plus. D'après l'analyse de documentation de la librairie Eigen et la consultation (par courriel électronique) avec les autres membres de la communauté (Marc Schellens, Giles Duvert...) on a trouvé que le problème a causé le nombre de threads définie statiquement et indépendant de la charge de machine pour OpenMP (Eigen utilise OpenMP pour bénéficier de multi-cœurs).

Cette occasion a produit un devoir inattendu : j'ai dû écrire une fonction qui va définir le nombre de threads dynamiquement avant chaque exécution d'un procédure/une fonction et cette fonction va prendre en considération la charge de la machine sur laquelle elle est exécutée.

3.3.2 Realisation

Sous les systèmes d'exploitations basées sur Unix on calcul le nombre de threads optimale (*suggested_num_threads*) par la formule suivante :

$$suggested\_num\_threads = nbofproc - avload \qquad (1)$$

Le nombre de processeurs sur la machine (*nbofproc*) est connu pour OpenMP et c'est un simple appelle à sa fonction interne. Alors, il nous reste de trouver la charge moyenne de la machine (*avload*). La charge moyenne de la machine est définie comme un nombre réel de type X.XX (par exemple : *avload*=2.64 sur une machine avec 8 processeurs virtuelles, où 2 signifie, que deux processeurs sont entièrement occupés et .64 signifie, que le troisième est chargé partiellement ; les cinq processeurs sont libres), de dernière(s) 1, 5 ou 15 minutes. Dans notre cas on a décidé d'utiliser le moyen de 5 derniers minutes, car une minute c'est un intervalle très court pour faire conclusion sur la charge de machine et 15 minutes c'est trop longue (on veut pas prendre en compte la charge de la machine par un logiciel qui a déjà terminé son fonctionnement). Puisque le nombre de processeurs et un nombre entier

et la charge moyenne est un nombre réel, la valeur de la charge moyenne est arrondi au plus proche.

Sous le système d'exploitation Windows le nombre de threads optimale est calculé par la formule un peu différent :

$$suggested\_num\_threads = nbofproc - avload * nbofproc/100 \quad (2)$$

La différence est à cause de format de présentation de charge de la machine, qui est représenté sous Windows comme un pourcentage d'occupation des tous les processeurs virtuelles (par exemple : $avload$=50 sur une machine avec 8 processeurs virtuelles signifie, que 4 processeurs sont occupés et les 4 restantes sont libres).

Code source de la fonction *set_ num_ threads* est disponible dans Annexe A sur page i.

3.3.3 Etude des performances

L'aspect très important dans la concurrence entre GDL et IDL c'est la performance, afin qu'ils sont utilisées pour les calcules scientifique. Signifiant, que les données à traiter sont assez large (par exemple : les matrices avec les dimensions supérieurs à 200). La performance de majorités des fonctions de GDL est très proche ou même meilleur que la performance d'IDL, mais pour attirer plus d'utilisateurs il faut avoir un meilleur indice de performance que IDL pour tous les fonctions sur tous les systèmes d'exploitations.

Pour trouver les indices de performance des différents fonctions d'inversion de matrice une suite de testes a été écrit, cette teste se compose de calcul du temps d'inversion des matrices aux très larges dimensions (255, 256, 300, 500). Les tests ont été faites sur ARAMIS, qui pour le moment de tests a eu 2 cœurs occupés en permanence par des autres processus. Cette condition a été idéale pour tester la performance de nouvelle caractéristique, ce qu'a été introduit par la fonction de définition de nombre de threads.

Type	INVERT[1]	INVERT_EIGEN[1]	IDL	INVERT[2]	INVERT_EIGEN[2]
BYTE	7.383	6.249	0.543	0.663	0.397
INT	5.242	6.125	0.530	0.657	0.374
LONG	5.108	5.605	0.537	0.656	0.365
FLOAT	7.768	5.283	0.533	0.630	0.339
DOUBLE	5.645	6.580	1.161	0.690	0.466
COMPLEX	10.218	9.467	3.991	2.004	1.651
DCOMPLEX	13.543	2.675	4.365	2.160	2.089
UINT	9.204	7.416	0.520	0.671	0.399
ULONG	5.546	5.238	0.522	0.666	0.397

TABLE 1 – Résultats des tests de performance

D'après les tests, la fonction d'inversion de matrice utilisant la bibliothèque Eigen a meilleur performance que celui utilisant la bibliothèque GSL. Mais sur une machine chargé les deux fonctions deviennent incomparable avec la fonction de IDL, qui est largement plus vite. Alors, après la fonction *set_num_threads* la performance de la fonction d'inversion de matrice avec GSL (qui est utilisé dans la plupart de fonctions codés sous C++ pour les calcules scientifiques) est proche de la fonction présent dans IDL et parfois est meilleur.

Ce qui concerne la fonction d'inversion de matrice avec Eigen, il a les meilleurs résultats pour touts les types (Table 1 montre les temps de calcule pour les multiples fonctions). En plus d'un bon performance, Eigen est simple à utiliser et il se charge de la gestion de mémoire, qui dispense l'utilisateur de fuite de mémoire. C'est pourquoi Eigen a été choisi comme un bibliothèque pour les calcules matricielles pour les nouvelles fonctionnalités, qui vont faire la partie de GSL version 9.4.

En conséquence de la définition dynamique de nombre de threads on a obtenu meilleur indice de performance pour l'inversion de matrice aussi bien que pour tous les fonctions utilisant la parallélisation avec OpenMP.

1. sans *set_num_threads* sur une machine chargé
2. avec *set_num_threads* ou sans *set_num_threads*, mais sur une machine non chargé

3.4 Factorisation de Cholesky

La tâche suivante a été de coder sous C++ la factorisation (décomposition) de Cholesky d'une matrice. Dans IDL on a quatre procédures/-fonctions concernant la factorisation de Cholesky (CHOLDC, CHOLSOL, LA_CHOLDC, LA_CHOLSOL). Tous ces procédures/fonctions étaient absents dans GDL et des utilisateurs souhaient en beneficier. Certaines codes en syntaxe IDL/GDL (PlanckSkyModel, iCosmo) utilisent CHOLDC et ne pouvaient etre totallement audites. Ma mission a été de les rajouter en utilisant la bibliothèque Eigen.

Cette factorisation est utilisée pour la résolution des systèmes d'équations linéaires, simulations avec méthode de Monte-Carlo, filtre de Kalman, l'inversion d'une matrice hermitienne.

3.4.1 Description de factorisation de Cholesky

La factorisation de Cholesky, nommée d'après André-Louis Cholesky, consiste, pour une matrice symétrique définie positive **A**, à déterminer une matrice triangulaire inférieure **L** telle que :

$$A = LL^T \tag{3}$$

Où $\mathbf{L^T}$ est la transposée de matrice triangulaire inférieure **L**. On peut imposer que les éléments diagonaux de la matrice **L** soient tous positifs, et la factorisation correspondante est alors unique.

$$\begin{pmatrix} 4 & 12 & -16 \\ 12 & 37 & -43 \\ -16 & -43 & 98 \end{pmatrix} = \begin{pmatrix} 2 & & \\ 6 & 1 & \\ -8 & 5 & 3 \end{pmatrix} \begin{pmatrix} 2 & 6 & -8 \\ & 1 & 5 \\ & & 3 \end{pmatrix}$$

FIGURE 6 – Exemple de factorisation de Cholesky

Pour la résolution d'un système d'équations linéaires, la factorisation de Cholesky est approximativement deux fois plus efficace que la décomposition LU. Alors c'est très important d'avoir dans GDL les routines concernant la factorisation de Cholesky.

3.4.2 La réalisation de CHOLDC, CHOLSOL

La première d'ensemble de factorisation de Cholesky que j'ai codé a été la procédure **CHOLDC**. Cette procédure est utilisé pour trouver une matrice triangulaire inférieure d'une matrice symétrique définie positive donnée. Coder cette procédure a été une tâche sans peine, car j'ai déjà eu l'expérience de codage avec librairie Eigen, que j'ai obtenu pendent ma mission précédent (la fonction d'inversion de matrice). En plus Eigen contient tous les algorithmes de suite de Cholesky, alors j'ai dû à faire la répartition des appels des algorithmes par les types appropriés.

Ensuite j'ai implémenté la fonction **CHOLSOL**. Cette fonction calcule la solution pour un système d'équations linéaires **Ax=B**. Cette fonction prend comme premier argument la matrice calculée par **CHOLDC**, mais GDL en différence d'IDL utilise la matrice triangulaire supérieure qui contient les valeurs initiales. La différence est à cause de librairie Eigen, qui ne support pas la résolution d'un système d'équations linéaires d'après les valeurs intermédiaires (la matrice décomposée).

```
GDL> A = [[ 6.0, 15.0, 55.0], $
GDL>    [15.0, 55.0, 225.0], $
GDL>    [55.0, 225.0, 979.0]]
GDL>
GDL> B = [9.5, 50.0, 237.0]
GDL>
GDL> CHOLDC, A, P
GDL>
GDL> PRINT, CHOLSOL(A, P, B)
     -0.50000      -1.00000       0.50000
GDL>
```

(a) GDL

```
IDL> A = [[ 6.0, 15.0, 55.0], $
IDL>    [15.0, 55.0, 225.0], $
IDL>    [55.0, 225.0, 979.0]]
IDL>
IDL> B = [9.5, 50.0, 237.0]
IDL>
IDL> CHOLDC, A, P
IDL>
IDL> PRINT, CHOLSOL(A, P, B)
    -0.499998     -1.00000     0.500000
IDL>
```

(b) IDL

FIGURE 7 – Exemple de résolution d'un système d'équations linéaires dans différents interpréteurs

La Figure 7 montre un exemple de résolution d'un système d'équations linéaires **Ax=B** dans différents interpréteurs en utilisant la procédure **CHOLDC** et ensuite la fonction **CHOLSOL**. Pour résoudre un système d'équations linéaires dans IDL on est obligé d'exécuter les deux procédures, mais dans GDL on peut le résolve sans utilisation de **CHOLDC**. Puisque dans la fonctions **CHOLSOL** on fait la décomposition (c'est obligatoire en raison d'utilisation de bibliothèque Eigen). Pour assurer la compatibilité on peut résoudre **Ax=B** à la manière d'IDL (suite de deux commandes).

La besoin de calculer deux fois la même factorisation baisse la performance de GDL dans résolution d'un système d'équations linéaires, mais GDL a un autre avantage par rapport à IDL : c'est une solution plus exacte. Dans un exemple qui est représenté sur la Figure 7 la solution exacte est un vecteur avec les valeurs : { -0.500000, -1.00000, 0.500000 } - ce que retourne GDL, mais le vecteur calculé par IDL est un peu différent, la première valeur contient un erreur de -0.000002.

3.4.3 La réalisation de LA_CHOLDC, LA_CHOLSOL

La procédure **LA_CHOLDC** est la procédure **CHOLDC** généralisé pour trouver la décomposition d'une matrice à coefficients complexes (matrice hermitienne). Pour la factorisation d'une matrice à coefficients réels, en plus de la Formule (3) (page 23) qui est utilisé par la procédure **CHOLDC**, **LA_CHOLDC** peut utiliser la Formule (4), où **U** est une matrice triangulaire supérieur et $\mathbf{U^T}$ est sa transposée :

$$A = U^T U \tag{4}$$

Pour une matrice hermitienne la procédure **LA_CHOLDC** peut utiliser l'une des formules suivantes :

$$A = U^H U \tag{5}$$

$$A = L L^H \tag{6}$$

H signifie une matrice adjointe (aussi appelée matrice transconjuguée).

La réalisation de **LA_CHOLDC** a été similaire à la réalisation de **CHOLDC**, car la bibliothèque Eigen gère les types complexes aussi bien que les autres types.

3.5 Convolution

La procedure **COVOL** convolve la matrice avec noyau et retourne la matrice modifié

Conclusion

Pour conclure, avec LaTeX on obtient un rendu impeccable mais il faut s'investir pour le prendre en main.

Références

[REF] auteur. *titre*. édition, année.

[LPP] Rolland. *LaTeX par la pratique*. O'Reilly, 1999.

Table des figures

Abréviations

ANTLR ANother Tool for Language Recognition

BSD Berkeley Software Distribution

CLI Command Line Interface

CNAP Conseil National des Astronomes et Physiciens

CNRS Centre National de la Recherche Scientifique

CPU Central Processing Unit

CVS Concurrent Versions System

DIO Division Informatique de l'Observatoire

EC Enseignant-Chercheur

ESA European Space Agency

ENS École Normale Supérieure

EPSCP Établissement Public à caractère Scientifique, Culturel et Professionnel

FITS Flexible Image Transport System

FSF Free Software Foundation

HDF Hierarchical Data Format

IDL Interactive Data Language

INSU Institut National des Sciences de l'Univers

GDL GNU Data Language

GPL General Public License

GSL GNU Scientific Library

LDAP Lightweight Directory Access Protocol

MESR Ministère de l'Enseignement Supérieur et de la Recherche

MPL Mozilla Public License

NASA National Aeronautics and Space Administration

NetCDF Network Common Data Form

PRAG PRofesseur AGrégé

UMR Unité Mixte de Recherche

XDR External Data Representation

Annexes

A Code de *set_num_threads*

```
#ifndef _OPENMP
int get_suggested_omp_num_threads() {
  return 1;
}
#endif

#if defined _OPENMP
int get_suggested_omp_num_threads() {

  int default_num_threads=1, suggested_num_threads=1;

  char* env_var_c;
  env_var_c = getenv ("OMP_NUM_THREADS");
  if(env_var_c)
    {
      return atoi(env_var_c);
    }
  //    cout<<"OMP_NUM_THREADS is not defined"<<endl;

  //set number of threads for appropriate OS
  int avload, nbofproc=omp_get_num_procs();
  FILE *iff;

#if defined(__APPLE__) || defined(__MACH__) || defined(__FreeBSD__)
    || defined(__NetBSD__) || defined(__OpenBSD__) ||
    defined(__bsdi__) || defined(__DragonFly__)
  cout<<"is MAC/*BSD"<<endl;
  iff= popen("echo $(sysctl -n vm.loadavg|cut -d\" \" -f2)", "r");
  if (!iff)
    {
      return default_num_threads;
    }

#elif defined(__linux__) || defined(__gnu_linux__) || defined(linux)
  iff= popen("cat /proc/loadavg |cut -d\" \" -f2", "r");
  if (!iff)
    {
      return default_num_threads;
```

```
    }

#elif defined (__unix) || (__unix__)
  iff=freopen("/proc/loadavg","r",stderr);
  fclose(stderr);
  if(!iff)
    {
      cout<<"your OS is not supported"<<endl;
      return default_num_threads;
    }
  iff= popen("cat /proc/loadavg 2>/dev/null|cut -d\" \" -f2", "r");
  if (!iff)
    {
      return default_num_threads;
    }

#elif defined(_WIN32) || defined(__WIN32__) || defined(__WINDOWS__)
  iff= popen("wmic cpu get loadpercentage|more +1", "r");
  if (!iff)
    {
      return default_num_threads;
    }
  char buffer[4];
  char* c;
  c=fgets(buffer, sizeof(buffer), iff);
  if(!c)
    {
      return default_num_threads;
    }
  pclose(iff);
  int cout=0;
  while(buffer[count]!='\0' && buffer[count]!=' ')count++;
  for(int i=1,j=1;i<=count;i++,j*=10)
    avload+=(buffer[count-i]-'0')*j;
  suggested_num_threads=nbofproc-(int)(avload*((float)nbofproc/100)+0.5);
  return suggested_num_threads;

#else
  cout<<"Can't define your OS"<<endl;
  return default_num_threads;
#endif

  char buffer[4];
```

```
80   char* c;
81   c=fgets(buffer, sizeof(buffer), iff);
82   if(!c)
83     {
84       return default_num_threads;
85     }
86   pclose(iff);
87   avload=(buffer[0]-'0')+((buffer[2]-'0')>5?1:0);
88
89   suggested_num_threads=nbofproc-avload;
90   return suggested_num_threads;
91 }
92 #endif
```

B Annexes2

C Annexes3