Annexe A

Réflexions sur le développement logiciel et la recherche

Les réflexions présentées ici ne font pas partie intégrante de la contribution scientifique de ce travail de thèse. Néanmoins, il nous paraît important de s'interroger sur le fonctionnement de la discipline à laquelle nous participons et de proposer des initiatives pour contribuer à son évolution. C'est dans ce cadre que nous proposons à la communauté de traitements d'images médicales un système logiciel complet `ImLib3D`, librement disponible. Avant de décrire `ImLib3D` au paragraphe B.1, p. 156 nous allons présenter, ici, les réflexions qui ont motivé ce travail.

Le traitement d'images est une discipline scientifique comprenant à la fois des aspects théoriques, des aspects expérimentaux et des applications pratiques. L'expérimentation et l'application reposent, en traitement d'images, sur des systèmes logiciels dont la complexité croît avec l'avancement de la discipline. Cette complexité pose des défis fondamentaux.

Après une description des menaces que cette complexité fait peser sur le traitement d'images, nous argumenterons sur la nécessité d'une ouverture et d'une large diffusion des développements logiciels associés à la recherche.

A.1 Reproductibilité et démarche scientifique

Le système des publications, avec évaluation par des pairs, est, aujourd'hui, un des principaux garants de la validité de la démarche scientifique. Mais ce système, bien établi, est aussi utilisé par les institutions (universités, centres de recherche, organismes de financement) pour juger le travail des chercheurs. Publier devient alors un enjeu déterminant pour l'attribution de ressources et pour les carrières de la recherche. La forte concurrence induite, si elle peut être stimulante dans certains cas, suscite une surenchère dans les publications. Les résultats publiés sont alors souvent exposés de manière très favorable, et les problèmes occultés. Dans ce cadre, le travail d'évaluation devient particulièrement délicat, et repose principalement sur deux axes : l'analyse critique et la reproductibilité des résultats.

Évaluation par analyse critique

Par analyse critique, nous entendons le processus d'évaluation qui consiste à évaluer un article sans expérimentation. Si cette approche permet de repérer des erreurs de méthodologie, elle reste fortement subjective. Le passage d'un cadre théorique à un cadre expérimental soulève très souvent de nombreux problèmes, imprévisibles, subtils, mais rédhibitoires. Il est très difficile de prévoir comment une méthode peut réagir à diverse formes de perturbations dans

les données. Il n'est pas aisé d'envisager tous les cas particuliers pouvant dérouter une méthode. Sauf pour des articles purement théoriques (comme la démonstration d'un théorème), il paraît donc peu fiable de baser l'évaluation sur la seule analyse critique.

Évaluation par reproduction expérimentale

La reproductibilité des résultats est un des fondements de la méthodologie scientifique. Dans de nombreuses disciplines, cela implique l'utilisation de matériel onéreux et de procédures lourdes, elle peut même y faire l'objet d'une publication en soi. En traitement d'images, la reproduction expérimentale nécessite la copie du système logiciel, ce qui n'a qu'un coût négligeable. Malheureusement, le logiciel implémentant une méthode décrite dans une publication n'est que rarement librement disponible. La reproduction des résultats nécessite donc, le plus souvent, de ré-implémenter les méthodes décrites. Elle se heurte alors sur plusieurs obstacles majeurs :

Ré-implémentation : un effort insurmontable L'obstacle le plus important à la ré-implémentation d'un logiciel est son coût, en terme de quantité de travail. Un système de traitement d'images moderne peut être constitué de plusieurs dizaines de milliers de lignes de code. En intégrant les divers outils nécessaires, il peut même s'agir de plus d'un million de lignes. Ceci peut représenter plusieurs mois de travail et, en pratique, aucun relecteur n'est prêt à fournir un pareil effort.

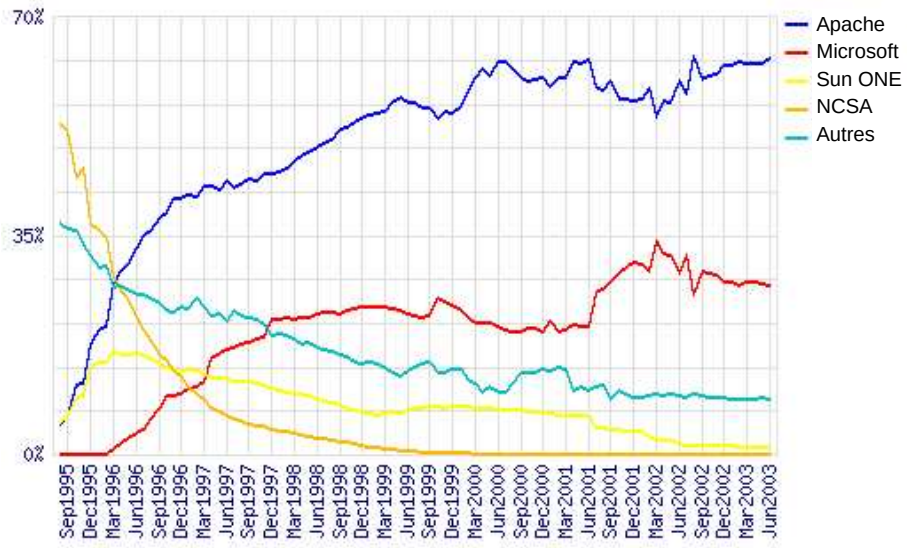
Ré-implémentation : manque de détails Une publication, même si elle décrit de manière détaillée une méthode, ne peut pas en être une description complète. L'expérience montre qu'un grand nombre de petits détails doivent être résolus pour en assurer le fonctionnement effectif. Ces détails, réunis, forment une partie essentielle d'un travail, mais n'ont pas leur place dans le cadre synthétique et l'espace limité d'une publication.

Le jeu de données/images Même si le relecteur réussit à obtenir ou à ré-implémenter le logiciel associé à un article, il lui manque encore les données nécessaires pour l'évaluer et surtout pour reproduire les résultats publiés. En imagerie médicale, par exemple, la création d'une base de donnée d'images d'évaluation (comme celle utilisée dans le travail de cette thèse, voir § 8.1, p. 134) représente un travail considérable et un investissement financier non négligeable. Un autre facteur limitant la reproductibilité des résultats est l'utilisation pratique des méthodes. Souvent, les méthodes requièrent le réglage d'un nombre considérable de paramètres, difficiles à déterminer.

Conclusion partielle

Les considérations précédentes tendent à montrer que dans son état actuel, la discipline de traitement d'images peut être ouverte à des dérives inquiétantes, et que la validité des publications y est sujette à caution. Une illustration en est la disparité entre la simplicité des techniques utilisées dans des applications réelles et l'évolution spectaculaire de la complexité des techniques apparaissant dans les publications.

Face à ces dérives, une solution serait de considérer l'implémentation logicielle comme partie indissociable d'une publication et du travail scientifique. Le paradigme du logiciel libre offre un cadre adapté, dynamique et éprouvé, présenté au paragraphe A.2. Les freins à son extension sont analysés au paragraphe suivant (A.3).



TAB. A.1 – *Les logiciels libres ont pris une position dominante dans l'infrastructure logicielle d'Internet. Par exemple, à partir de 1996, Apache devient le premier serveur Web (source : netcraft.com)*

Si la diffusion libre des logiciels associés à un travail paraît nécessaire, elle n'est pas suffisante. Le partage des données ainsi que l'organisation de cadres d'évaluation standardisés pour des méthodes ayant les mêmes objectifs, sont d'autres aspects essentiels.

A.2 Logiciels libres

« Logiciel libre » est un terme générique pour désigner des logiciels dont le code source est redistribué librement, avec divers types de licences destinées à empêcher toute ré-appropriation susceptible de les priver de leur liberté[1]. Les logiciels libres ne sont plus une simple curiosité, ils sont devenus un des principaux acteurs de l'informatique professionnelle actuelle. Des logiciels libres tels BIND (DNS, $\simeq$ 100% de parts de marché), Apache (serveur Web, voir tableau A.1) et SendMail (serveur mail, $\simeq$ 80%) forment les briques de base de l'infrastructure logicielle de l'Internet. Des systèmes d'exploitation libres, tels GNU-Linux, gagnent en importance[2] et commencent même à s'introduire dans le marché des ordinateurs personnels. Le moteur de recherche Google, par exemple, repose sur un cluster de plus de 10 000 serveurs sous Linux[3]. Divers pays (Brésil, Pérou, Chine ...) ont adopté, ou sont en voie d'adopter, des mesures législatives incitant ou imposant aux institutions, et parfois aux entreprises d'Etat l'utilisation de logiciels libres[4] [66].

L'essor des logiciels libres a intrigué de nombreux observateurs. Les motivations économiques et sociologiques des divers acteurs ont fait l'objet de nombreuses études (références dans [60]). Les mécanismes sociologiques (économie de la réputation, capital symbolique) en œuvre dans les communautés de logiciel libres [56, 91] présentent de fortes ressemblances avec ceux agissant dans le champ de la recherche scientifique [54].

[1] `http://www.gnu.org/philosophy/free-sw.fr.html`
[2] En 2002, Linux possèdait 49% des parts sur le marché des serveurs contre 25% pour MS Windows [43].
[3] `http://www.google.fr/press/highlights.html`
[4] `http://news.com.com/2100-1001-272299.html?legacy=cnet`

Accessibilité et flexibilité

Contrairement à une idée reçue, la gratuité des logiciels libres n'est pas leur principal attrait. De nombreux utilisateurs professionnels, particulièrement dans l'industrie, et souvent dans la recherche, sont prêts à payer le prix de logiciels tels que MATLAB.

Les avantages majeurs des logiciels libres sont leur accessibilité et leur flexibilité. Pour évaluer ou utiliser un logiciel libre, il suffit de le télécharger et de l'installer. Lorsqu'un chercheur ou un ingénieur a ponctuellement besoin d'un outil, il n'a pas besoin de demander une autorisation, de faire un bon de commande, et d'attendre la livraison. Une fois le logiciel libre téléchargé, il peut aisément l'intégrer en partie ou en totalité au sein de ses propres développements. De plus, il peut le décortiquer, l'analyser et s'en inspirer. Ces aspects sont très importants. Il est fréquent, en informatique, d'apprendre par l'exemple, cela n'étant possible qu'avec le code source à disposition. Par ailleurs, si le logiciel libre ne répond pas exactement aux besoins de l'ingénieur, des modifications / corrections, souvent très simples, peuvent aisément être apportées au logiciel afin de mieux l'adapter. Par exemple, un logiciel propriétaire est disponible uniquement sous forme binaire, on ne peut donc l'installer que sur les plates-formes (systèmes d'exploitation) prévues par le vendeur. Dans le cas du libre, il est possible de les recompiler, quitte à y faire de légères modifications.

Qualité logicielle

Contrairement à une autre idée reçue, les logiciels libres sont souvent à la fois plus fiables et plus performants que leurs équivalents non libres. Les études quantitatives précises se multiplient dans ce sens[5], certaines comparant, à l'aide de protocoles expérimentaux rigoureux, les taux de pannes survenant sur des configurations équivalentes, dans des environnements réels.

L'univers du logiciel libre est en foisonnement permanent. De nouveaux projets se créent puis meurent en permanence. Les logiciels que nous évoquons ici sont ceux ayant atteint une certaine maturité, par la persévérance de leurs développeurs, et la sélection sévère imposée par les utilisateurs.

Une des explications de la qualité des logiciels libres est à chercher dans le fonctionnement dynamique des communautés « Open Source ». Les utilisateurs, communiquant par e-mail ou au travers de forums ouverts, signalent les défaillances et participent à leur réparation [91, 123]. L'enthousiasme généré par l'environnement coopératif ouvert est un facteur important de cette dynamique.

Les développeurs d'un produit libre sont engagés dans une concurrence basée sur la réputation [54], et accordent une grande importance à la fiabilité de leur produit. Leur code source étant ouvert et visible par tous, il est exposé à la critique (tout comme une publication scientifique est évaluée par des pairs). Le développeur peut donc difficilement se permettre de produire du code de mauvaise qualité. La situation est toute autre dans certains environnements non libres, ou la pression commerciale peut pousser à la production rapide de code de mauvaise qualité.

Dans le cas particulier de la recherche, la publication de leurs développements en « Open Source » incite les chercheurs à faire attention à la qualité et à la documentation de leur production logicielle. Si cela demande un effort supplémentaire, le résultat est une plus grande ré-utilisabilité. Au lieu de produire du code expérimental « jetable », le chercheur pose alors des briques sur lesquelles d'autres pourront construire.

[5] `http://www.dwheeler.com/oss_fs_why.html`

Éthique

Un autre argument majeur en faveur du logiciel libre est l'éthique. Au lieu de développer pour leur bénéfice propre ou celui de l'institution dans laquelle ils travaillent, les producteurs de logiciels libres diffusent au plus grand nombre le fruit de leur travail[6]. Tout comme le chercheur peut être motivé par l'avancement de la science, la large diffusion d'un logiciel libre participe au progrès technique. Les chercheurs du secteur public sont rémunérés par le travail du contribuable, qui attend d'eux la plus grande efficacité dans l'avancement de la recherche. Restreindre la diffusion de leur production est un frein important à l'avancement de la recherche, ce qui est contraire à la mission qui leur est implicitement confiée.

Par ailleurs, l'accroissement des inégalités au niveau mondial, maintient une majorité de la population à l'écart du progrès technique. Les institutions de recherche des pays pauvres n'ont pas les moyens de payer les licences souvent coûteuses des logiciels de la recherche. Partager la production logicielle scientifique contribue à moins les marginaliser sur le plan scientifique, et leur permet d'accéder plus facilement aux résultats de la recherche en termes d'applications.

A.3 Freins à l'extension des logiciels libres dans la recherche

Si une grande partie des logiciels libres trouvent leur origine dans le milieu universitaire ou de la recherche, il reste que de nombreux laboratoires sont réticents à la diffusion libre de leur production logicielle.

Les équipes de recherche sont souvent en compétition pour les mêmes ressources limitées (financements, publications), et ne souhaitent alors pas diffuser leur travail de peur de favoriser les concurrents. Mais inversement, la diffusion libre d'un logiciel performant peut aussi contribuer fortement au rayonnement du laboratoire.

Les restrictions budgétaires et les incitations de la part des pouvoir publics, poussent les laboratoires à vouloir générer des ressources propres en commercialisant leur production logicielle. Cependant, il n'est pas certain qu'un groupe de recherche dispose des moyens humains[7] nécessaires à la réussite commerciale d'un logiciel. La diffusion de la production logicielle, dans un cadre libre, demande aussi un effort non négligeable, même s'il est bien moindre que dans le cadre commercial. Il s'agit de mettre en forme le code source, de le documenter, de le maintenir et d'interagir avec les utilisateurs. Comme il a été dit précédemment, une partie de ce travail est très utile pour le laboratoire lui-même, en ce qu'il permet de pérenniser les développements.

[6] Ceci n'est pas toujours entièrement désintéressé : le capital symbolique accumulé en participant au monde du libre peut souvent être utilisé pour obtenir des carrières bien rémunérés dans le secteur marchand.

[7] Ingénieurs, commerciaux et service après vente.

Annexe B

ImLib3D

ImLib3D est une plate-forme de traitement d'images volumiques composée d'une librairie et d'un logiciel de visualisation interactif séparé. Elle est distribuée librement (http://imlib3d.sourceforge.net), en « Open Source », et est placée dans un cadre de développement distribué coopératif (sourceforge.net). Elle a été crée et est gérée par l'auteur, avec des contributions de T.Vik. ImLib3D a été conçue pour la recherche et le développement en traitement d'images volumiques (3D, et 3D+t). Elle vise à être facile à utiliser par des scientifiques n'ayant pas nécessairement une connaissance pointue en programmation[1]. En revanche, elle ne vise pas à fournir un outil adapté au clinicien[2]. L'ensemble du travail de cette thèse a été implémenté à l'aide de ImLib3D.

ImLib3D repose sur une conception orientée objet et utilise des concepts modernes s'inspirant de la librairie standard du C++. Ces fondements conceptuels seront brièvement présentés aux paragraphes suivants, avant d'aborder la présentation technique de ImLib3D.

Dans la communauté de traitement d'images cérébrales, plusieurs initiatives de logiciels librement disponibles sont à noter. Statistical Parametric Mapping[3] est un logiciel basé sur Matlab®[4], très répandu, en particulier dans l'imagerie fonctionnelle. Matlab est un environement propriétaire proposant un langage de programmation interpreté, orienté vers le prototypage rapide d'applications scientifiques. Il est moins adapté au développement d'applications complexes, en termes de performances et de conception logicielle, qu'un langage de programmation moderne, comme le C++. Une autre initiative, très récente est ITK[5]. La première version, développée par des industriels sous contrat du NLM[6], est apparue en 2002. Elle est fondée sur des concepts similaires à ceux de ImLib3D (généricité, orientée objet, ... voir ci-après). Il s'agit d'un projet de taille conséquente, conçu avec une méthodologie rigoureuse. Le financement de ITK prend fin en 2003 et la gestion de cette initiative, dans l'avenir, parait incertaine. Si cette initiative est prolongée et si elle reste « Open Source », elle pourrait prendre une ampleur considérable dans le milieu du traitement d'image médicales. Dans le domaine

[1] L'utilisation exige néanmoins la connaissance de la syntaxe du C++. Il a été choisi dans ImLib3D d'utiliser un C++ conforme au standard. La modification du C++ (par l'utilisation de macros, ou d'un préprocesseur) est une possibilité intéressante permettant de simplifier d'avantage la syntaxe[19], mais exige aussi l'apprentissage d'un « nouveau » langage de programmation.

[2] Un outil destiné au clinicien pourrait être conçu comme une couche logicielle s'appuyant sur ImLib3D. Les besoins, les concepts, et la terminologie utilisés par les cliniciens sont très éloignés de ceux utilisés dans la communauté de traitement d'images. Un outil commun ne paraît donc pas satisfaisant.

[3] SPM[30] : http://www.fil.ion.ucl.ac.uk/spm

[4] http://mathworks.com

[5] Insight Toolkit : http://itk.org

[6] National Library of Medicine du National Institutes of Health

non médical, remarquons Vigra[7] (traitement 2D) et Olena[8] [19] (traitement N-D), ayant des approches génériques comparables à la nôtre.

B.1 Fondements conceptuels

La conception de `ImLib3D` repose sur trois fondements : la généricité, les itérateurs et la conception orientée objet. Ces concepts sont brièvement présentés dans les paragraphes suivants. Le concept de l'extensibilité et son implémentation dans `ImLib3D` seront décrits plus loin (§ B.5, p. 169). D'autres concepts de génie logiciel employés dans `ImLib3D`, tels que les batteries de tests automatisées, ne seront pas décrits ici.

B.1.1 Généricité

Motivations Le stockage et la manipulation de données sont des aspects fondamentaux de la programmation. Le programmeur structure ses données dans des tableaux (1D, 2D ...), des listes chaînées, des tableaux associatifs, etc., selon ses exigences en termes de performances et de place mémoire. La gestion de ces structures de données, appelées conteneurs, a longtemps été laissée à la charge du programmeur (librairies *ad hoc* dans certains langages compilés)[9]. Le programmeur avait alors le choix entre des conteneurs spécifiquement typés (en C/C++ : un tableau *d'entiers*, une liste chaînée *de flottants*), ou des conteneurs non-typés (un tableau de pointeurs `void`, une liste chaînée contenant des pointeurs `void`). Ces deux solutions ne sont pas satisfaisantes. La première implique la réécriture des conteneurs pour chaque nouveau type de données à stocker. Dans le cas de la seconde solution (les conteneurs non-typés) le trans-typage doit être effectué explicitement par l'utilisateur et il n'y a, en général, pas de vérification de type[10], ce qui est extrêmement dangereux. De surcroît, cette solution laisse à l'utilisateur la gestion de la mémoire (allocation, désallocation des données stockées), ce qui, en pratique, est très fastidieux et peu fiable. Les problèmes que nous venons d'illustrer pour les conteneurs, sont plus généraux, ils existent aussi pour les traitements. D'une manière plus générale, la dépendance à des types (ou des constantes) fixes pose d'importants problèmes pour la conception d'un logiciel.

Description Dans la programmation générique, les classes, structures ou fonctions génériques sont paramétrées par des types[11] qui ne seront précisés qu'à l'utilisation. Par exemple, le concepteur crée une classe représentant une liste chaînée dont le type de données reste indéfini. Ensuite, l'utilisateur instancie cette classe générique pour créer une liste chaînée de flottants (par exemple). Le concept de la généricité[12], et son introduction dans le C++ par

[7] `http://kogs-www.informatik.uni-hamburg.de/~koethe/vigra`

[8] `http://www.lrde.epita.fr/olena`

[9] Pour les langages non compilés, des vérifications de type sont possibles à l'exécution, et les remarques qui suivent ne s'appliquent pas.

[10] La vérification de type peut être très coûteuse à la fois en mémoire et en temps de calcul.

[11] La généricité peut porter sur des types, mais aussi sur des valeurs constantes. Par exemple, si on définit un vecteur de dimension fixe, la valeur constante de cette dimension peut être un argument template.

[12] La généricité ne doit pas être confondue avec le polymorphisme. Dans la conception objet, le polymorphisme permet effectivement de manipuler de manière homogène des objets d'une classe dérivant d'un même parent. La généricité permet, elle, de manipuler des objets de familles totalement distinctes, sans avoir à faire de transtypage (up-cast, down-cast), et sans avoir à créer des hiérarchies artificielles (cf. conteneurs en Java). Les deux concepts sont distincts mais complémentaires.

les templates et STL[13], ont marqué une évolution importante dans la conception de la programmation. STL fournit une librairie de classes génériques avec, entre autres, un cadre unifié, extrêmement bien conçu, de gestion des différentes formes de conteneurs (tableaux, listes chaînées, tableaux associatifs, etc.), et les moyens de parcourir ces conteneurs (itérateurs, voir paragraphe suivant). Par ailleurs, la programmation générique a montré de nombreux potentiels[14], comme en témoigne le foisonnement d'applications dans boost[15]. Si la conception et l'implémentation de librairies génériques restent des tâches complexes[16], leur utilisation, par le programmeur est très simple et transparente.

Généricité dans ImLib3D Une image volumique à support discret est naturellement interprétée comme un conteneur à support tridimensionnel. En traitement d'images il est courant de manipuler des images pouvant avoir des valeurs réelles, de valeurs complexes (domaine de Fourier), de valeurs entières (images de label), de valeurs binaires, etc. Dans ImLib3D, ces images sont implémentées dans la même philosophie que les conteneurs génériques de STL. Les fonctions (processeurs) de traitement d'image sont aussi génériques, et peuvent donc être réutilisées pour plusieurs types d'images différentes.

B.1.2 Itérateurs

Itérateurs : Introduction Lors de sa conception, un conteneur peut être vu sous deux angles : le stockage des données (évoqué précédemment), et le parcours des données. Dans la conception classique, chaque conteneur était parcouru par une approche spécifique. Par exemple, un tableau était parcouru en balayant ses indices i et en indexant le tableau (`tab[i]`), une liste chaînée était parcourue, elle, en étudiant successivement le nœud courant puis en passant au nœud suivant. Le concept d'itérateur permet d'unifier le parcours des différents types de conteneurs avec une même syntaxe[17]. Un itérateur est un objet indiquant la position courante dans un conteneur. Il permet d'accéder au contenu à cette position (déréférencement) et fournit les méthodes permettant de modifier sa position (avancer, reculer, avancer de plusieurs éléments, etc.). Chaque conteneur, et ses itérateurs associés, ont des capacités et une complexité algorithmique spécifiques associées à chaque opération. Par exemple, avancer un itérateur de n éléments a une complexité algorithmique de $O(1)$ pour un tableau, de $O(n)$

[13] La Standard Template Library (STL), est une librairie de conteneurs génériques initialement proposée par A. Stepanov en 1992 et qui a été incorporée au standard ANSI/ISO définissant le C++ en 1994. Voir [105, 79].

[14] À titre anecdotique, on peut démontrer que l'exécution par le compilateur de l'instanciation récursive de templates est équivalente à une machine de Turing. Il est alors théoriquement possible de faire exécuter par le compilateur un programme arbitraire. Ceci montre les potentiels mais aussi les dangers de l'utilisation de templates. Il peut devenir tentant, dans certaines situations, d'utiliser les templates dans des contextes pour lesquels il n'ont pas été prévus. Ceci peut aboutir à du code excessivement complexe, ou bien à une modification potentiellement déroutante de la syntaxe du C++.

[15] Boost `http://www.boost.org` est un répertoire de librairies C++, soumises à une révision par pairs stricte. Ces librairies ont pour vocation d'être intégrées dans des versions futures du standard définissant le langage C++.

[16] Il s'agit de concevoir du code qui puisse fonctionner pour un grand nombre de types différents, ce qui exige beaucoup de soin.

[17] En plus de permettre une syntaxe unifiée, le concept d'itérateur permet également de dissocier les algorithmes des conteneurs. Par exemple, une fonction qui trie (quicksort) une suite de données pourra travailler uniquement sur les itérateurs, sans se soucier du type de conteneur associé. Cette approche n'est cependant pas utilisée dans ImLib3D, car la plupart des processeurs de traitement d'image nécessitent des informations sur le conteneur (l'image) qui ne sont pas présentes dans des itérateurs.

pour une liste chaînée et de $O(\log_2(n))$ pour un tableau associatif (implémenté en STL sous forme d'un arbre binaire).

Syntaxe en C++/STL Un exemple simple d'itérateur est le pointeur en C/C++. Considérons un pointeur p sur un élément d'un tableau. L'accès au contenu à la position indiquée par p se fait par le déréférencement *p. Le passage à l'élément suivant se fait par l'incrémentation p++. Cette syntaxe est reprise et généralisée dans STL pour les itérateurs sur toutes les formes de conteneurs (tableaux, listes chaînées, etc.). La généralisation de cette syntaxe est rendue possible grâce à la redéfinition des opérateurs du C++.

B.1.3 Conception orientée objet

Motivations Le génie logiciel, dont la conception orientée objet est un des aspects importants, est né d'un constat pratique. La gestion de la complexité d'un logiciel est le principal enjeu du développement logiciel. En d'autres termes, s'il est facile d'écrire un programme implémentant une fonctionnalité précise, il est très difficile d'écrire un système logiciel de plus grande taille qui puisse évoluer. Un petit programme ($< 3\,000$ lignes) peut être appréhendé dans son ensemble par un individu, et aucune méthodologie particulière n'est nécessaire à sa conception[18]. Lorsque la taille et le nombre de développeurs croissent, on atteint rapidement un point de blocage. Le temps alors passé à comprendre, reconcevoir, et réimplémenter, dépasse de loin le temps passé à ajouter de nouvelles fonctionnalités. Une méthodologie devient donc indispensable.

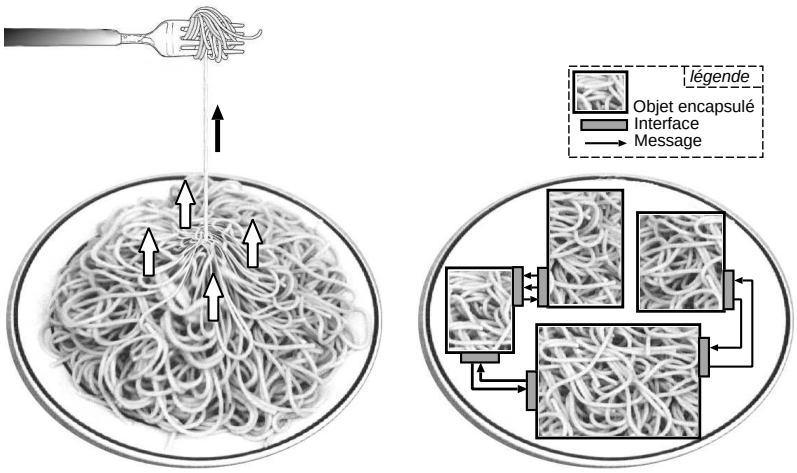
Méthodologie objet La méthodologie objet repose, entre autres, sur le concept de modularité. Un logiciel est vu comme un ensemble de boîtes noires (objets) communiquant entre elles. La communication entre boîtes noires doit être réduite à son strict minimum, et se fait au travers d'interfaces bien définies et documentées (figure B.1). Néanmoins, l'approche orientée objet n'est pas, en soi, une garantie de bonne conception. Des hiérarchies de classes trop complexes, et trop imbriquées (over-engineering), sont des erreurs fréquentes. La reconception et la réécriture font donc toujours partie du cycle de vie d'un logiciel. Une bonne conception permet d'allonger la durée de vie avant reconception, et de diminuer la quantité de code devant être réécrit à chaque cycle.

B.2 La librairie ImLib3D

ImLib3D est composée d'une librairie et d'un visualiseur décrit au paragraphe B.3, p. 167. Deux autres éléments, les extensions utilisateur (§ B.5, p. 169) et l'interface ligne de commande (§ B.4, p. 169), seront aussi décrits ultérieurement (voir figure B.2).

La librairie, décrite ici, fournit tous les composants nécessaires pour écrire des programmes de traitement d'images volumiques. Le **premier** composant est formé des classes d'images et toute l'infrastructure logicielle qui facilite leur gestion. Nous décrirons d'abord les fonctionnalités communes à toutes les images (§ B.2.1), puis les différentes classes d'images proposées (§ B.2.2), et finalement les itérateurs facilitant le parcours des images (§ B.2.3). Le **deuxième** composant est le système de gestion des processeurs de traitement d'images qui fournit une

[18] La difficulté d'enseigner la programmation orientée objet provient de là : les étudiants ont rarement eu à gérer des logiciels dépassant quelques centaines de lignes.



a : complexité mal gérée b : orienté objet

FIG. B.1 – *Métaphore du plat de spaghetti. (a) Un logiciel peut rapidement devenir très complexe (comme un plat de spaghetti collant). Lorsqu'on cherche à modifier un élément (retirer un spaghetti) tout est si imbriqué, que cela a des répercussions sur l'ensemble du logiciel. Dans la conception orientée objet (b) on encapsule la complexité dans des boîtes noires communiquant par des interfaces bien définies. Les répercussions d'une modification sont alors mieux contrôlées.*

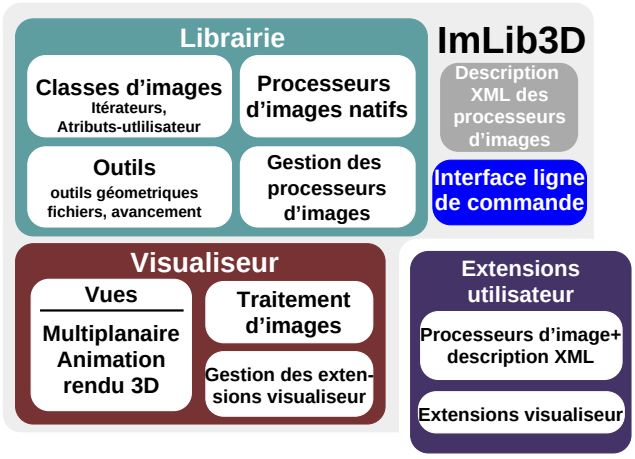


FIG. B.2 – *Vue d'ensemble des différents éléments de* `ImLib3D`

interface unifiée et documentée à toutes les opérations de traitement d'images. Il sera décrit au paragraphe B.2.4. Le **troisième** composant, les processeurs de traitement d'images natifs fournis avec ImLib3D, seront décrits au paragraphe B.2.5. Nous finirons la description (§ B.2.6) par le **quatrième** composant de la librairie, constitué d'un ensemble d'outils divers mis à la disposition des utilisateurs de ImLib3D.

B.2.1 Les fonctionnalités communes à toutes les images

ImLib3D propose un grand nombre de classes d'images partageant certaines fonctionnalités communes. Une partie de ces fonctionnalités, comme la gestion de la taille de l'image, les accès fichier, les masques et les propriétés utilisateur est décrite par la classe Image3D. Toutes les images dans ImLib3D dériveront (§ B.2.2) de cette classe (non générique).

Manipulation élémentaire des images La création d'images et l'accès à leurs éléments se fait très simplement à l'aide d'opérateurs intuitifs. Voici un exemple :

```
// création d'une image à valeurs flottantes de taille 100x100x100
Image3Df monImage(100,100,100);
monImage(5,2,3)=100;     // change  la valeur au point (5,2,3)
cout << monImage(5,2,3); // affiche la valeur au point (5,2,3)
```

Fichiers Les formats de fichiers d'images médicales volumiques présents dans le commerce (DICOM, ANALYZE [AVW]) ont un codage trop rigide pour permettre l'enregistrement de tous les types différents d'images pouvant apparaître dans ImLib3D (et les informations associées). Un format de fichier spécifique basé sur le XML est donc proposé. Le XML est un langage de description extensible, hiérarchique, et très souple qui permet d'ajouter simplement des informations structurées. ImLib3D permet également à l'utilisateur, s'il le souhaite, de lire et d'écrire des fichiers au format AVW.

```
Image3Df monImage("fichier.im3D");
monImage.WriteToFile("fichier2.im3D");
Image3Df monImage2;
monImage2.ReadFromFile("fichier2.im3D");
```

Masques Il est fréquent en traitement d'images de vouloir associer des régions géométriques à une image. On voudra, par exemple, définir des régions d'intérêt (ROI) sur lesquelles pourront se restreindre certains traitements. Dans ImLib3D, toute image est optionnellement associée à un masque. Le masque Mask3D est une image à valeurs entières (0-255).

Propriétés dynamiques utilisateur Souvent, un utilisateur souhaite associer à une image des informations telles que des données sur un patient ou des préférences d'affichage. La solution de l'héritage proposée par l'approche objet est parfois insuffisante[19], ou trop lourde à mettre en œuvre. La classe Image3D permet à l'utilisateur d'ajouter dynamiquement des attributs nommés de n'importe quel type à une image. Elle fournit un mécanisme pour gérer ces propriétés automatiquement (entrée/sorties fichier, opérateurs de recopie etc.) et vérifie la cohérence des types :

[19]Lorsque des propriétés doivent être ajoutées dynamiquement, par exemple à l'exécution dans une interface graphique, l'héritage n'est pas envisageable.

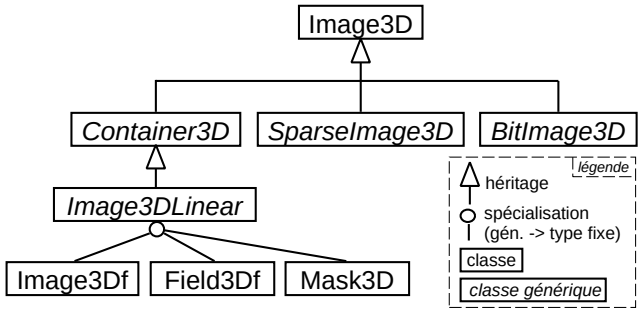


FIG. B.3 – *Hiérarchie simplifiée des principales classes d'images.*

```
Image3Df image(100,100,100);
image.SetProperty("nom","Toto");
image.SetProperty("age",35);
image.SetProperty("position tumeur",Vect3Df(53,25,55));
cout << image.Property<Vect3Df>("position tumeur");
```

Interpolation Dans de nombreuses situations, particulièrement lors de transformations géométriques (rotation, transformation affine ...), on souhaite prendre la valeur d'une image sur des points qui ne sont pas situés exactement sur la grille discrète. Les interpolateurs implémentés sont de type plus proches voisins, linéaire, sinus cardinal [61] et B-spline. Ce dernier est particulièrement performant [107] (voir aussi § 7.3.5, p. 122). Les images et les interpolateurs étant génériques, il est possible d'associer un interpolateur à toute image supportant les opérations linéaires (par exemple, des images à valeurs flottantes, des images à valeurs complexes, des champs de vecteurs, ou tout autre type défini par l'utilisateur). Voici un exemple pour un champ de vecteurs :

```
  Field3Df champ(10,10,10);
...
  champ.SetInterpolator(new SplineInterpolator3D<Vect3Df> >(3));
  cout << champ.Value(1.1, 2.2, 3.3);
```

B.2.2 Les différentes classes d'images

ImLib3D utilise des conteneurs génériques, semblables à ceux de STL, pour représenter des images de types différents. Nous allons commencer leur description par une vue d'ensemble de la hiérarchie des principales classes (figure B.3).

De cette classe parent Image3D dérivent trois sous-classes. La plus utilisée, Container3D, décrit des images classiques, où les données sont stockées dans un tableau tridimensionnel d'éléments[20]. Container3D est une classe générique. Les deux autres sous-classes, les images creuses SparseImage3D, et les images compactes BitImage3D, ont des modes de stockage des données différents, peu encombrants en mémoire.

Images classiques

Les images classiques dérivées de Container3D peuvent être instanciées pour des types de données très divers. Des exemples courants sont les images à valeurs flottantes Image3Df et les

[20] Après expérimentation, le stockage des données image en une seule région contiguë de mémoire est apparue optimale.

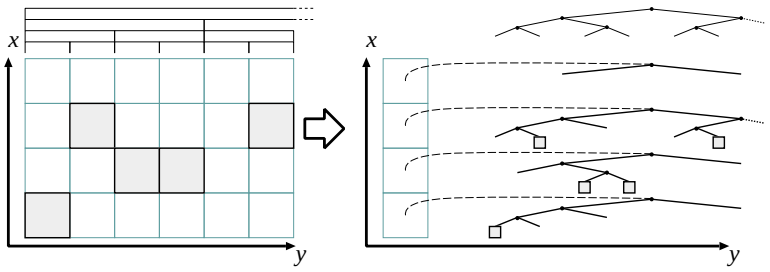


FIG. B.4 – *Image creuse (en 2D pour l'illustration). Seuls les éléments non nuls de l'image sont stockés. Un tableau d'arbres binaires (conteneur associatif* `std::map` *dans* `STL`*) permet un accès à un point aléatoire en un temps* $O\big(\log_2(n)\big)$ *où* n *est la taille de l'image dans une direction (ici y). En 3D il s'agit d'un tableau bidimensionnel d'arbres binaires. L'accès séquentiel aux éléments non nuls de l'image se fait en un temps linéaire.*

images à valeurs entières `Mask3D`[21]. Parmi les images classiques, nous pouvons distinguer celles supportant des opérations linéaires, images flottantes, images complexes (`Image3Dcomplex`), champ de vecteurs (`Field3Df`), etc. D'autres types d'images sont simples à dériver de `Container3D`[22]. L'utilisateur peut aussi très facilement spécialiser `Container3D` pour ses propres besoins. Par exemple, une image, où chaque élément serait un modèle statistique (moyenne, variance) pourrait être utilisé comme suit :

```
  struct ModeleStatistique {float moyenne,ecartType;};
...
  Container3D<ModeleStatistique> atlas(50,50,50);
  atlas(5,2,3).moyenne=10;
  atlas(5,2,3).ecartType=2;
```

Images creuses

Les images classiques peuvent devenir très volumineuses. Dans certaines applications, comme dans la segmentation proposée dans la partie II, seul un nombre limité de points sont non nuls. Il est alors possible de réduire fortement l'encombrement mémoire en stockant uniquement les valeurs non nulles des images. Les images creuses `SparseImage3D` dans `ImLib3D` ont une structure creuse (disposition d'éléments nuls et non nuls) fixe, qui peut être partagée par plusieurs images creuses. Chaque image creuse ne contient plus (pour simplifier) qu'un tableau des valeurs non nulles. L'accès séquentiel aux éléments non nuls est donc très rapide. La structure creuse (figure B.4) permet aussi l'accès à une position arbitraire de l'image. Pour un point où la valeur de l'image est nulle, l'accès se fait en un temps constant $O(1)$; pour les points non nuls l'accès se fait en un temps logarithmique. Ces opérations sont transparentes pour l'utilisateur, qui manipule l'image de la même manière que les images ordinaires. La classe `SparseImage3D` est générique permettant la création d'images creuses ayant des éléments de tout type.

```
  Mask3D elementsNonNuls("elementsNonNuls.im3D");
  SparseStructure3D structureCreuse=SparseStructure3D::Create(elementsNonNuls);
```

[21] Les valeurs de `Mask3D`, sur un octet, sont limitées entre 0 à 255.

[22] Par exemple, dans `ImLib3D` il existe une image représentant une subdivision régulière de l'espace utile pour résoudre le problème des k plus proches voisins `Neighbors3D`. Chaque élément de cette image est un tableau redimensionnable (`std::vector`) contenant des points tridimensionnels.

```
SparseImage3D<float> imageCreuse1(structureCreuse);
SparseImage3D<int  > imageCreuse2(structureCreuse);
```

Images compactes

Les images génériques classiques permettent de créer des images de tout type. Cependant, la taille minimale d'un élément est un octet. Lorsque l'utilisateur souhaite manipuler des images à valeurs entières ayant un nombre restreint de valeurs (images binaires par exemple), il est possible d'avoir une représentation plus compacte. La classe `BitImage3D` permet de stoker chaque voxel sur un nombre arbitraire de bits (identique pour toute l'image).

```
// création d'une image dont les valeurs sont représentées sur 2 bits
// les groupes de 2 bits sont arrangés sur des octets
BitImage3D<byte,2,byte> image(50,50,50);
image(5,2,3)=2;
```

B.2.3 Itérateurs : le parcours des images

En traitement d'images, une opération essentielle est le parcours des images. Les itérateurs fournissent un cadre simple et standardisé dans ce but. Ils allègent le code, améliorent sa lisibilité et peuvent permettre des gains de performance. Au lieu d'écrire une triple boucle imbriquée pour parcourir l'image, une seule boucle suffit :

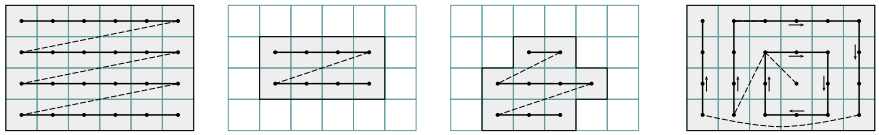
```
Image3Df image(100,100,100);

// parcours d'une image par la méthode classique
for(int z=0;z<image.Depth() ;++z)
  for(int y=0;y<image.Height();++y)
    for(int x=0;x<image.Width() ;++x)
      image(x,y,z)=rand();

// parcours d'une image à l'aide d'un itérateur
for(Image3Df::iterator p=image.begin();p!=image.end();++p)
  *p=rand();
```

Comme l'illustre cet exemple, l'utilisation des itérateurs permet aussi d'encapsuler (cacher les détails de l'implémentation) le fonctionnement de l'image. Dans la méthode classique, l'utilisateur doit avoir des connaissances sur des détails du fonctionnement interne de la classe afin de savoir dans quel ordre il est plus efficace d'écrire les trois boucles[23] (x, y, z ou z, y, x).

`ImLib3D` fournit différents types d'itérateurs pour parcourir les images de diverses manières (figure B.5). Il existe deux versions de l'itérateur permettant le parcours lexicographique de toute l'image. L'une `iteratorFast` est très rapide (pas d'indice à gérer). L'autre `iteratorXYZ`, légèrement plus lente, permet de suivre la position (x, y, z) sur laquelle se trouve l'itérateur. D'autres itérateurs permettent de parcourir des portions restreintes d'une image (régions parallélépipédiques, ou définies par un masque), ou de parcourir l'image dans un ordre spécial (cubes concentriques).

Les itérateurs de `ImLib3D` sont compatibles avec `STL`. Il est donc possible de les utiliser avec les algorithmes `STL` standards (`sort`, `find`, `for_each`, `mismatch`...).

[23] L'ordre des boucles a un impact considérable sur la performance dû à l'utilisation de la mémoire cache.



a : normal b : zone c : masqué d : concentrique

FIG. B.5 – *Trois types d'itérateurs parcourant des régions différentes d'une image (en 2D pour l'illustration). (a) Itérateur parcourant l'ensemble de l'image dans l'ordre lexicographique. (b) Itérateur parcourant une portion parallélépipédique de l'image. (c) Itérateur parcourant une région définie par le masque de l'image. (c) Itérateur parcourant l'image par cubes concentriques successifs, à partir d'un point.*

B.2.4 Gestion des processeurs de traitement d'images

Nous avons décrit, dans les paragraphes précédents, les différentes opérations de base permettant de manipuler des images dans `ImLib3D`. Ces opérations peuvent être composées pour implémenter des algorithmes de traitement d'image. Il s'agit alors de gérer le nombre important d'algorithmes différents qui peuvent exister dans une boîte à outils de traitement d'images. Cette gestion permettra de manipuler, et d'étendre, ces processeurs au travers de diverses interfaces.

Nous définissons le concept de « processeur de traitement d'image » (ou pour raccourcir « processeur d'image ») pour désigner l'implémentation d'un algorithme. Concrètement, un processeur d'image est composé d'une fonction C++ (habitant le `namespace IP3D`) et d'une description complète de cette fonction[24]. La description, structurée dans un répertoire hiérarchique `XML`, fournit des informations détaillées sur tous les aspects de la fonction : son nom, sa généricité, une documentation courte, une documentation plus longue, une description complète de chaque argument (nom, type, valeur par défaut, ...). Cette description est utilisée dans différents contextes (figure B.6). Les processeurs d'images respectent des conventions de syntaxe strictes, garantissant l'uniformité de leur utilisation. Ils pourront être appelés dans différents contextes (visualiseur, ligne de commande, programme C++). Le mécanisme précèdent permet à l'utilisateur d'ajouter très simplement ses propres processeurs d'images (voir § B.5).

B.2.5 Processeurs natifs

`ImLib3D` fournit un nombre important de processeurs de traitement d'image. Ils sont énumérés au tableau B.1, leur description détaillée dépasse le cadre de cette introduction. Certains processeurs (`ConvenienceProcessors`) correspondent à des manipulations qui se font très simplement en C++, mais qui sont aussi nécessaires dans les autres interfaces (graphique, ligne de commande).

[24]Nous avons expérimenté diverses approches pour la gestion des processeurs d'images, dont certaines utilisant des hiérarchies de classes pour représenter les processeurs. Finalement, il apparaît que ces approches ajoutent beaucoup de complexité et ne résolvent pas les problèmes de manière satisfaisante. Notre conclusion est qu'un processeur d'image est finalement un concept très large. Il est l'équivalent conceptuel d'une fonction. Il est cependant apparu qu'il était important de disposer d'une description permettant à d'autres modules de gérer ces fonctions convenablement.

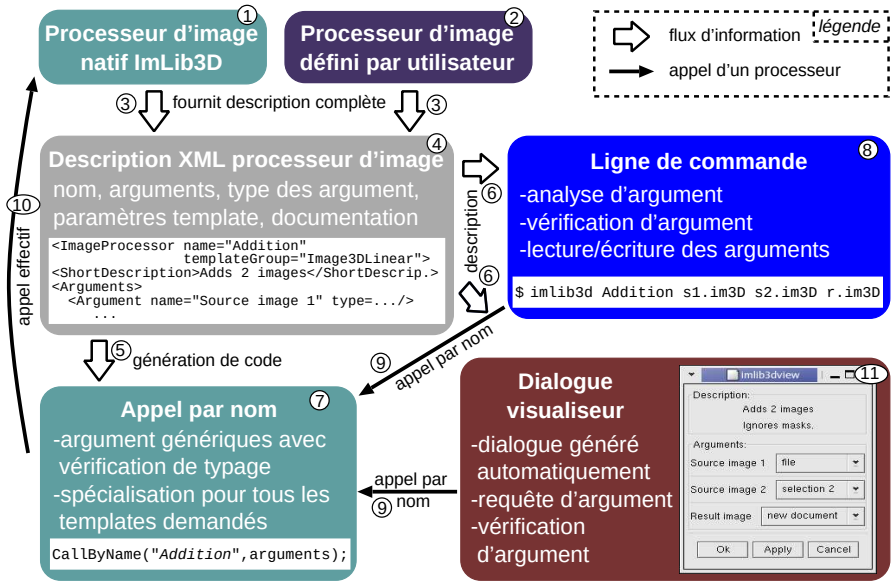


FIG. B.6 – *Chaque processeur de traitement d'image, qu'il soit natif (1), ou une extension de l'utilisateur (2), est décrit (3) de manière détaillée. La description (4), en XML, fournit toutes les informations nécessaires pour permettre de générer automatiquement (5) le code nécessaire pour appeler le processeur. Ces informations sont aussi suffisantes pour générer automatiquement (6) des dialogues interactifs du visualiseur et une interface ligne de commande. L'appel d'un processeur par son nom, en fournissant une liste d'arguments génériques (7), nécessite d'avoir préalablement spécialisé (5) tous les processeurs génériques pour tous les types souhaités par le concepteur du processeur. Par exemple, lorsque l'utilisateur demande, sur la ligne de commande (8) d'additionner deux images complexes ($\mathbb{N}^3 \rightarrow \mathbb{C}$), il est nécessaire que le processeur* `Addition` *ait été préalablement compilé (5,7) pour le type complexe (la compilation à la volée poserait trop de problèmes). L'appel en ligne de commande (8) utilise (6) la description XML du processeur pour gérer les arguments fournis par l'utilisateur et vérifier leur cohérence. Les arguments saisis sont ensuite mis dans une liste générique, et le processeur d'image est appelé par son nom (9). Le processeur correspondant, avec la bonne spécialisation est ensuite effectivement appelé (10). Le même schéma est utilisé pour l'appel d'un processeur à partir d'un dialogue interactif du visualiseur (11).*

Arithmetic
- Addition
- AdditionWithConstant
- Difference
- DifferenceWithConstant
- Multiplication
- MultiplicationWithConstant
- Division
- MaxImage
- Abs

ImageStats
- Average
- AverageAndVariance
- RobustAverageAndVariance
- Median

MorphologicalOperators
- MedianFilter
- SharpeningFilter
- Erosion
- Dilation
- Opening
- Closing
- DistanceTransform
- ConnectedComponentLabelling
- FillHoles
- MakeValueList
- LargestConnectedComponent
- **VoxelCoding**
  - VoxelCode
  - SSCode
  - BSCode
  - ShortestPathExtraction
  - Skeleton

Estimator
- NoiseVarianceEstimation

Threshold
- SimpleThreshold
- SimpleThresholds2
- LimitThreshold
- OtsuThresholds
- UniModalThreshold

Transform
- TransformBSpline3D*
- TransformWithInverseField
- TransformAffine
- Scale
- Flip
- SwapAxes
- WrapTranslate

LinearFiltering
- Convolution
- SeparableConvolution
- ConvolutionFFT
- FFTLowPassFilterApodizedIdeal
- FFTLowPassFilterButterworth
- FFTLowPassFilterChebyscheff
- PartialSum
- CumulativeSum
- BoxFilter
- GaussianApproxFilter
- SymmetricBinomialFilter

ImageAs1DVector
- ScalarProduct

ShapeAnalysis
- CenterOfGravity
- PrincipalAxes
- BoundingBox

VisualizationAids
- TransformRectangularGrid
- AnimateField

ConvenienceProcessors
- ImageTypeConversion
- Crop
- Pad
- InsertImage
- ResizeImageSupport
- FindFirstDifference
- FindMinMax
- SetAllVoxels
- AddMask
- GetMask
- ApplyMask
- SetDefaultInterpolator
- SetProperty
- ExtractFromComplexImage

FileConversion
- ReadFromFileIPB*
- WriteToFileIPB*
- ReadTransfoFromFileIPB*
- ImageFromRawData
- RawDataFromImage
- WriteToFileAVW
- ReadFromFileAVW
- ReadFromFileBruker
- ExportImageSlice

TestPatterns
- Sphere
- Cone
- NoiseUniform
- NoiseGaussian
- Explode
- Target
- Bump
- Ramp
- Parallellogram
- RectangularGrid
- ColoredGrid

Normalization
- NormalizeAverageAndVariance
- NormalizeJointHistogram
- ComputeJointHistogramImage

FFTOperators
- FFTDouble
- FFTFloat
- FFTInverseDouble
- FFTInverseFloat
- FFTDoubleToComplex
- FFTFloatToComplex
- FFTInverseComplexToDouble
- FFTInverseComplexToFloat

BrainProcessors*
- ExtractHeadMask*
- SegmentationFromAtlas*
- SegmentHeadAndBrain*
- CenterCenterOfGravity*
- ChangeDetectionConstantPlusNoise*

Registration*
- RigidRegistration*
- AffineRegistration*
- DeformableRegistration*

TAB. B.1 – *Processeurs de traitement d'image existants regroupés hiérarchiquement. Les processeurs marqués d'un astérisque sont des extensions utilisateur. Les autres processeurs sont natifs à ImLib3D.*

B.2.6 Outils

Surveillance de l'avancement d'une tâche Le traitement de certaines images tridimensionnelles peut être très long. L'utilisateur s'inquiète alors, et souhaite connaître l'état d'avancement du traitement, et le temps restant. `ImLib3D` fournit une approche de surveillance à base de fils[25] très simple à utiliser et n'ayant pas d'impact sur la performance du programme[26]. L'utilisateur doit simplement indiquer une variable qu'il souhaite surveiller. Généralement il s'agira d'un itérateur :

```
Image3Df::iterator p;
ImageProgress avancement("état d'avancement:",p);
for(p=image.begin();p!=image.end();++p)
{
   ...
}
```

Cet exemple imprimera à l'écran le message « état d'avancement : xx% », toutes les secondes (valeur paramétré), en fonction de la position de l'itérateur dans l'image.

Outils divers `ImLib3D` fournit une série de fonctions utilitaires pour la manipulation des chaînes de caractères (utilisant `std::string`), pour la manipulation de fichiers et pour l'affichage de courbes.

Outils géométriques `ImLib3D` fournit aussi un ensemble de classes facilitant les manipulations géométriques : des vecteurs bidimensionnels et tridimensionnels, une classe pour les transformations affines, des régions parallélépipédiques et quelques autres outils simples.

B.3 Le visualiseur

`ImLib3D` propose un visualiseur extensible permettant d'afficher, de traiter et d'éditer des images volumiques. Les images sont représentées dans une classe `Document` ayant des spécialisations génériques permettant de manipuler un grand nombre de types d'images différentes.

L'espace d'affichage est composé d'un nombre variable de vues (`View`). Trois grands types de vues sont actuellement implémentés. Les vues multiplanaires, les animations multiplanaires et le rendu 3D surfacique. L'utilisateur peut lier certaines vues entre elles (avec différents adaptateurs, si les tailles des images sont différentes). La navigation dans une vue sera alors synchronisée avec certaines autres vues.

B.3.1 Vues

Vue multiplanaire L'affichage multiplanaire permet de naviguer dans un volume tridimensionnel en visualisant trois coupes perpendiculaires en un point désigné par le curseur. Elles sont re-dimensionables, et l'utilisateur peut facilement zoomer sur le point désigné. Lorsque le type de l'image le permet (type linéaire), les vues multiplanaires permettent d'afficher les images en utilisant l'interpolation linéaire, améliorant ainsi nettement la qualité de l'affichage.

[25] Les fils (en anglais : threads) sont des processus légers qui s'exécutent parallèlement, et qui partagent l'espace d'adressage mémoire du processus principal.

[26] L'approche classique exige d'ajouter des tests à l'intérieur d'une boucle potentiellement critique.

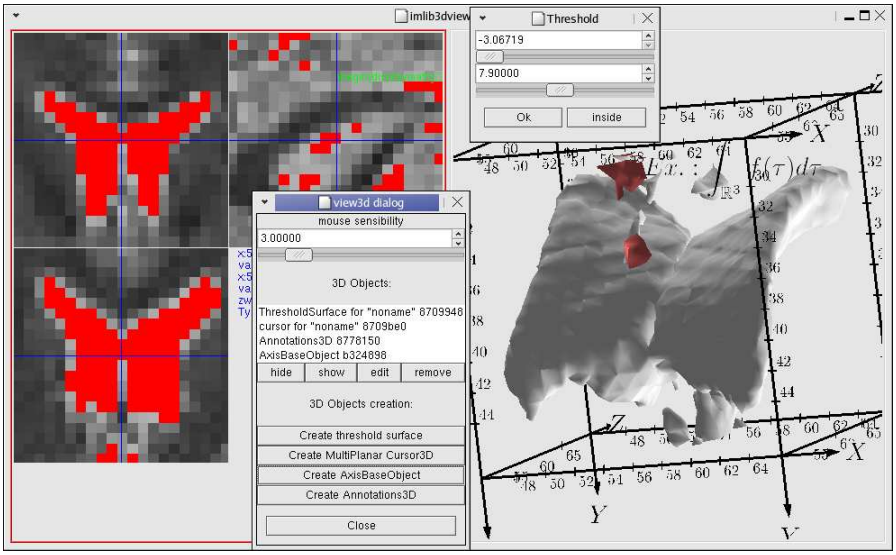


FIG. B.7 – *À gauche une vue multiplanaire agrandie des ventricules. À droite une vue 3D du volume seuillé (rouge) dans la vue multiplanaire.*

Parmi de nombreuses fonctionnalités signalons l'affichage avec des palettes de couleurs extensibles, le seuillage interactif, le réglage de contraste, l'affichage par superposition du masque associé, l'affichage d'histogrammes, l'affichage de coupes 1D de l'image et la sélection interactive de régions parallélépipédiques.

Animations multiplanaires L'affichage d'animations à partir d'une série d'images est particulièrement utile pour l'analyse de séries temporelles et pour visualiser des changements entre deux images. L'animation se présente sous la forme d'une vue multiplanaire où défilent les images successives. La plupart des opérations des vues multiplanaires décrites au paragraphe précèdent sont accessibles.

Vue 3D Cette vue fournit une visualisation surfacique en 3D isométrique simple. Elle ne prétend pas être un système de visualisation 3D complet. Elle permet d'afficher des surfaces correspondant à des images seuillées interactivement dans des vues multiplanaires[27]. Elle permet aussi d'afficher en 3D des annotations (y compris du texte avec de formules en LaTeX), une représentation du curseur correspondant aux trois plans de coupe d'une vue multiplanaire et des axes gradués.

B.3.2 Autres fonctionnalités

Traitement d'image Tous les processeurs de traitement d'images sont accessibles à partir du visualiseur. Cela comprend les processeurs définis dans `ImLib3D` et ceux définis comme des extensions par l'utilisateur. Un dialogue complet, comprenant une documentation et une saisie interactive des arguments, est généré automatiquement pour chaque processeur.

Annotations Un système souple d'annotations permet d'ajouter des commentaires et des dessins (texte, sphères de couleur ...) à toute position de l'espace. Ces annotations pourront

[27] Nous utilisons optionnellement la librairie `gts` (GNU Triangulated Surface Library : `http://gts.sourceforge.net`) pour la simplification et la manipulation de maillages.

être enregistrées avec l'image.

Dessin Un système simple de dessin interactif volumique est proposé dans les vues multi-planaires. L'utilisateur peut choisir une plume de taille, d'aspect et d'intensité variables et peut défaire (undo multiple) ses modifications. Cet outil permet de segmenter manuellement des structures anatomiques.

B.4 Interface ligne de commande

`ImLib3D` permet l'appel de tous les processeurs de traitement d'image à partir de la ligne de commande. Cette interface est utile pour expérimenter, pour effectuer des opérations simples, pour enchaîner des séquences simples d'opérations (scripts) et pour interagir avec d'autres logiciels.

```
imlib3d Addition image1.m3D image2.im3D resultat.im3D
imlib3d ExportImageSlice resultat.im3D  2 50 resultat.ppm
```

Une liste de tous les processeurs est obtenue par `imlib3d --list`. La documentation d'un processeur est affichée par `imlib3d NomDuProcesseur`.

B.5 Extensibilité d'`ImLib3D`

Pour maintenir la cohérence et la propreté d'un système logiciel, il est important que ses utilisateurs puissent aisément ajouter des fonctionnalités sans avoir à modifier le logiciel lui-même. Ces fonctionnalités peuvent répondre aux besoins particuliers de l'utilisateur. Il peut aussi s'agir des fonctionnalités expérimentales ayant pour vocation d'être intégrées ultérieurement dans le logiciel lorsqu'elles seront validées et stabilisées. Dans `ImLib3D` deux types d'extensions sont possibles. L'ajout de nouveaux processeurs de traitement d'image (§ B.5.1) reposant sur une documentation `XML` détaillée des fonctionnalités, et l'extension du système de visualisation.

B.5.1 Extensions des processeurs de traitement d'image

Pour ajouter un processeur de traitement d'image, l'utilisateur écrit les fonctions correspondantes ainsi que la description `XML` associée. Un utilitaire de `ImLib3D` permet alors de générer un programme exécutable[28], qui sera placé dans un chemin déterminé par une variable d'environnement. À l'exécution, les différentes interfaces de `ImLib3D` reconnaîtront automatiquement les nouveaux processeurs.

B.5.2 Extensions du visualiseur

Le travail de recherche en traitement d'images est grandement facilité par des outils de visualisation spécifiques au problème considéré. L'ajout dans le visualiseur de `ImLib3D` de systèmes de visualisation avec des applications très spécifiques n'est pas souhaitable. La conception orientée objet du visualiseur permet à l'utilisateur de créer de nouvelles classes de visualisation (vues) répondant à ses besoins spécifiques, en les dérivant des classes existantes.

[28]Il est prévu de permettre aussi l'extension par librairie dynamique, plus performante, mais moins portable.

Le visualiseur permet d'ajouter ces vues, au moment de l'exécution, en s'appuyant sur des librairies dynamiques (DLL). Les extensions du visualiseur ont été utilisées dans ce travail de thèse dans la partie segmentation (§ 5.1.1, p. 90, et figure 5.1.1) et dans la partie détection (§ 8.4, p. 135, et figure 8.1).

B.6 Perspectives et conclusions

Dans cette partie, nous avons décrit un cadre logiciel reposant sur des fondements conceptuels solides. Ce logiciel est proposé librement à la communauté de traitement d'images volumiques qui lui a montré son intérêt[29]. L'ensemble du travail décrit dans cette thèse, ainsi que d'autres travaux dans notre laboratoire reposent sur `ImLib3D`. Les principaux concepts de `ImLib3D` ont donc été modelés et validés par la pratique. La librairie a atteint un niveau de maturité permettant de passer à une étape de plus ample diffusion, et à une interaction accrue avec d'autres initiatives semblables dans le domaine. Les bases paraissent établies pour la recherche de nouveaux contributeurs sur le modèle proposé par E. Raymond [91].

Plusieurs perspectives s'offrent alors. La construction progressive d'une bibliothèque comprenant les principaux algorithmes de traitement d'images médicales utilisés par la communauté. Ceci permettrait la création d'une plate-forme commune pour l'évaluation et la comparaison des algorithmes proposés.

La consolidation de l'infrastructure logicielle existante par une modularisation accrue paraît souhaitable. En particulier, le système de gestion de processeurs d'image est un concept qui paraît être suffisamment général pour devenir un module séparé. Par ailleurs, une documentation plus technique donnant une vue d'ensemble aiderait des contributeurs potentiels à s'insérer dans le code.

[29] Plus de 7500 téléchargements à l'écriture de ce manuscrit.