Xlim3D:
Un logiciel de traitement d'image
[Projet RACE]

Documentation éditée par Marc Van Droogenbroeck
Centre de Morphologie Mathématique
January 25, 2002

Avant-propos

Le logiciel Xlim3D est le fruit du travail de plusieurs chercheurs du Centre de Morphologie Mathématique de Fontainebleau. Au départ, chacun développait ces propres fonctions mais, au fil du temps, il devient indispensable d'intégrer ces parties pour permettre à de nouveaux utilisateurs d'en bénéficier. Le manque de cohérence, tant au niveau de la nomenclature qu'au niveau de la programmation, se traduit par une complexité supplémentaire de l'initiation à ce logiciel.

Ce document présente différents travaux et tente d'uniformiser le cadre de travail futur, pour enrayer une éventuelle dérive...

On trouvera à l'intérieur de ce manuel des textes écrits en partie par des personnes qu'il faut citer:

Jean-François Rivest Il lui revient le mérite d'avoir écrit la première librairie étoffée de traitement d'images au Centre et de l'avoir diffusée ;

Christophe Gratin L'initiateur des traitements morphologiques tridimensionnels par logiciel. Sa librairie s'appuie sur l'interpréteur Xlisp-Stat, qui est devenu l'interpréteur de base ;

Hugues Talbot Le concepteur de nombreuses fonctions, y compris celles traitant les graphes ;

Mike Clarkson Le premier à avoir proposé et intégré les travaux des chercheurs.

Il faut également remercier les différentes personnes qui ont contribué au développement du logiciel: M. Bilodeau, B. Marcotegui, F. Meyer, J.-F. Rivest, H. Talbot, M. Van Droogenbroeck

Contents

Part I

L'interpréteur de commandes

Chapter 1

Utilisation de l'interpréteur

Il est bien connu que le langage Lisp est un langage d'intelligence artificielle. La plupart des manuels qui le décrivent ont ce type d'applications à l'esprit. La pratique réserve cependant un autre usage à l'interpréteur; dans certaines applications, il sert de lien avec des librairies de langage compilé. Les raisons sont simples: on a la facilité d'un langage interprété avec la vitesse d'un langage compilé. Cela permet des développements rapides de programmes puisqu'on évite la compilation et parce qu'il est possible de dialoguer aisément avec l'interpréteur, par exemple en affichant des messages.

Le programme Xlim3D est basé sur le langage défini par l'interpréteur Xlisp-Stat qui permet toute une série d'opérations comprenant notamment des manipulations statistiques de données, des opérations vectorielles, un système de programmation orienté objet et un mécanisme d'allocation dynamique. Des librairies de traitement d'image ont ensuite été greffées sur ce noyau en exploitant au mieux les deux derniers atouts du langage interprété, c'est-à-dire qu'une partie des traitements (notamment celui des éléments structurants) se fait dans la philosophie d'une orientation objet et que l'interpréteur charge dynamiquement des librairies suivant les besoins. Cela eut été impossible avec l'interpréteur Xlisp sur lequel était construit Xlim [3]. C'est Christophe Gratin [1] qui a songé à utiliser l'interpréteur Xlisp-Stat pour profiter du potentiel de l'orientation objet. Il faut aussi féliciter Michel Bilodeau qui a opéré la fusion des logiciels existants pour en faire des "packages" que l'on peut à présent charger dynamiquement. Grâce à ce chargement, il est possible d'insérer du code compilé même durant l'utilisation de l'interpréteur pour effectuer une mise au point ou éviter que l'interpréteur ne prenne trop de mémoire au chargement. Remercions enfin Jean-François Rivest dont les rapports ont servi de base au présent document.

Cette partie du manuel décrit l'utilisation du langage interprété. Sans vouloir établir une liste exhaustive des fonctions (voir [4] pour cela), nous présenterons les commandes de base de l'interpréteur en détaillant les quelques particularités du langage Xlisp-Stat qui le différencient d'un autre interpréteur Lisp. Nous expliquerons également comment développer et ajouter son propre code dans l'interpréteur.

1.1 Les premiers pas

1.1.1 Début d'une session

Assurez-vous d'abord que les exécutables `Xlim3d` et `xlisp` sont définies dans votre "PATH". Si le logiciel est bien configuré, il suffit de le lancer en tapant la commande `Xlim3d`. Le programme vérifie alors si le répertoire courant contient un fichier `init.lsp`, sinon il vous demande d'en placer un dans le répertoire. Ce fichier est important parce qu'il définit des variables bien utiles à l'utilisation des librairies de traitement d'image. On peut aussi ajouter des commandes qui configurent l'interpréteur d'une certaine manière, comme en chargeant par défaut l'une ou l'autre librairie, ou encore demander à `Xlim3d` de charger immédiatement un fichier en faisant `Xlim3d fichier.lsp`. L'interpréteur chargera alors `init.lsp` et ensuite `fichier.lsp`. Pour sortir du programme, `Ctrl-C` ne fonctionne pas parce que cette commande est réservée à l'interruption de routines Lisp mais il faut taper `(exit)`.

1.1.2 Quelques explications concernant le langage

Il y a deux types de résultat que l'interpréteur de Xlim3D est capable de fournir. Le premier est l'évaluation d'une expression. Les quelques lignes suivantes définissent une variable et évaluent son contenu:

```
> (setf a 10)
10
> a
10
>
```

La variable `a` vaut bien 10. La deuxième utilité est la réalisation d'une action, ce qui en Lisp s'exprime par un jeu de parenthèses contenant la fonction et ses arguments. Ainsi,

```
> (+ 1 2)
3
>
```

réalise l'addition de 1 et 2. En fait, Lisp utilise une notation fonctionnelle sous la forme `(fonction argument1 argument2)`, évalue la quantité entre parenthèses et retourne le résultat de l'évaluation. Chaque valeur peut alors être utilisée comme argument d'une autre fonction elle-même entre parenthèses et ainsi de suite. La priorité est donnée aux parenthèses intérieures. Les instructions

```
> (+ (* 2 3) 4)
10
>
```

seront traduites respectivement par `(+ 6 4)` et puis par `10`.

Le langage Lisp est basé sur le concept de liste. En d'autres termes, contrairement au langage C, le nombre d'arguments des fonctions ainsi que leur type est plus souple. Le Lisp se chargera des vérifications quant au nombre et à la validité des arguments. Ainsi, `(+ 2 3 6.2)` est correct mais pas `(+ 2 3 "car")` qui tente d'additionner une chaîne de caractères.

Les commentaires commencent toujours par un point virgule (;). Tout ce qui suit sur la même ligne sera ignoré par l'interpréteur. Notons aussi que l'interpréteur convertit tout, hormis les chaînes caractères, en majuscules. Nous conseillons néanmoins d'utiliser des majuscules à certains moments pour augmenter la lisibilité du code, par exemple lors de la définition de constantes.

1.2 Les types d'objets rencontrés

Le Lisp manipule des *nœuds*, sortes d'entité de base qui contiennent des informations comme une valeur, un type, ... Une fonction ou une image est contenue dans un nœud. Cette structure fondamentale est à l'origine de l'implémentation en C de l'interpréteur et quand sont venues les librairies de traitement d'images, il a fallu adapter la structure du nœud pour qu'il traite correctement les images.

1.2.1 Les types de données

Xlisp-Stat gère plusieurs types de données. Voici la liste des principaux types proposés par l'interpréteur à laquelle nous avons ajouté les types *image* et *graphe*:

- Les **entiers**. Ce sont des `long` comme en langage C;
- Les **points flottants**. Ce sont des `double` comme en langage C. Il faut toujours mettre le point décimal dans un nombre afin qu'il n'y ait pas de méprise, comme dans "5.0";
- Les **nombres complexes**. En guise d'exemple, `(setf a (sqrt -1))` crée le nombre complexe `a`;
- Les **tableaux**, qui peuvent contenir tous les autres types, eux-même compris lorsqu'on veut faire des tableaux multidimensionnels;
- Les **chaînes de caractères**;
- Les **objets**;
- Les **listes**;
- Les **variables booléennes**. Elles n'ont que la valeur "true" notée `T` ou `NIL` qui est également le signe d'une liste vide;
- Les **pointeurs de fichier** ("streams");
- Les **images** qui peuvent être 2D ou 3D;
- Les **graphes**.

Tester un type. Parfois, il peut être utile de savoir quel est le type d'un symbole ou d'un argument de fonction. Nous avons dressé la liste des prédicats qui examinent le type de leur argument (ils retournent `T` ou `NIL`):

- `(integerp a)` vérifie si `a` est un entier;

- `(floatp a)` vérifie si `a` est un réel;
- `(complex a)` vérifie si `a` est un nombre complexe;
- `(numberp a)` vérifie si `a` est un nombre (entier, réel ou complexe);
- `(arrayp a)` vérifie si `a` est un tableau;
- `(stringp a)` vérifie si `a` est une chaîne de caractères;
- `(objectp a)` vérifie si `a` est un objet;
- `(listp a)` vérifie si `a` est une liste;
- `(streamp a)` vérifie si `a` est un pointeur de fichiers;
- `(image2dp a)` vérifie si `a` est une image 2D;
- `(image3dp a)` vérifie si `a` est une image 3D;
- `(graphp a)` vérifie si `a` est un graphe.

Il y a une fonction qui, bien que ce ne soit pas un test proprement dit, retourne le type de `a` en toutes lettres: `(type-of a)`.

1.2.2 Évaluations et assignations

On assigne une valeur à une variable en utilisant la fonction `setf` (ou `setq`) ou `def`. Cette dernière instruction mémorise la variable et permet surtout de trouver une trace des constantes en tapant l'instruction `(variables)`. On libère une variable définie avec `(def a)` par `(undef 'a)` (noter la présence d'un "quote" dont l'utilité est décrite ci-après). Voici quelques exemples:

```
> (setf var 5)   ; On cree la variable var contenant 5
5                ; Xlisp repond cela
> var            ; On evalue var
5                ; Xlisp repond cela
> (def GRID "s") ; La constante GRID recoit la chaine "s"
GRID
> (variables)    ; La fonction qui imprime la liste des constantes
(GRID)
> (undef 'GRID)  ; On lib\‘ere GRID
GRID
> (variables)    ; La liste sera vide
NIL
>
```

On peut également faire des assignations multiples:

```
> (setf b 4 c 6 d "foo")
"foo"
>
```

La dernière expression évaluée sera celle qui sera retournée.

Une règle importante à propos des assignations: *ne jamais assigner un symbole déjà réservé par le système* –c'est la raison pour laquelle les fonctions usuelles de Xlisp-Stat se trouvent dans l'index des fonctions et dans le paquet **xlispstat** de l'index des "packages". Ceci ne peut que causer des problèmes. Fréquemment, Xlisp ne se plaindra pas mais écrasera tout simplement la valeur précédemment stockée sous ce nom, ce qui peut avoir des effets désastreux. En particulier, ne jamais assigner le symbole 'T', ou 'NIL'... Ce sont des erreurs qui sont communément faites. Parfois, il arrive qu'on ne désire pas évaluer une expression. Pour ce faire, Lisp met à disposition la fonction `quote` ou `'` pour éviter une évaluation. Ainsi, `(setf plus '(+ a 5))`, équivalente à `(setf plus (quote (+ a 5)))`, est une fonction qui définit l'addition de a et 5. Quelques lignes pour illustrer le comportement:

```
> (setf plus (quote (+ a 5))) ; Definit une fonction "plus"
(+ A 5)
> (setf a 5)
5
> (eval plus)  ; "eval" force l'evaluation de plus (interessant pour une
10             ; fonction)
> (setf a 15)
15
> (eval plus)
20
>
```

En agissant de la sorte, nous avons défini une fonction avant que la variable `a` ne soit définie puisque l'évaluation est repoussée jusqu'au moment de l'appel `(eval plus)`.

La fonction `quote` est couramment utilisée pour obtenir de l'aide sur une fonction. En effet, `(help 'fonction)` fournit une aide en ligne. De même, `(apropos 'name)` liste toutes les fonctions dont le nom contient la chaîne de caractères "name". Cela mérite une illustration:

```
> (help 'def)
DEF                                                    [function-doc]
Syntax: (def var form)
VAR is not evaluated and must be a symbol.  Assigns the value of FORM to
VAR and adds VAR to the list *VARIABLES* of def'ed variables. Returns VAR.
If VAR is already bound and the global variable *ASK-ON-REDEFINE*
is not nil then you are asked if you want to redefine the variable.
NIL
> (apropos 'ero)
HERO
A-SERO
GBIPERO
VEROS
VEROH
A-HERO
SERO
BIPERO
```

```
ZEROP
ERO
NIL
```

En ayant fait `(apropos 'ero)`, l'interpréteur affiche la liste de toutes les fonctions qui permettent de faire une érosion morphologique (sauf `zerop`). L'interpréteur est capable de gérer les informations d'aide par lui-même très simplement à partir d'une chaîne de caractères insérée après la définition du prototype d'une fonction. Cette possibilité est tellement importante que, avant même avoir expliqué comment définir une fonction, nous avons jugé utile de fournir un exemple.

```
> (defun aire (r)
"Calcule l'aire d'un disque a partir de son rayon"
(* r r 3.14))
AIRE       ; La fonction aire existe desormais ainsi qu'une aide en ligne
> (help 'aire)
AIRE                                                    [function-doc]
Calcule l'aire d'un disque a partir de son rayon
NIL
```

1.2.3 L'orientation objet

L'objectif poursuivi par les concepteurs de Xlisp-Stat en introduisant des notions de l'orientation objet est la facilité de mise au point de l'interface. Cela ne signifie pas que l'utilisation des concepts soit limitée à la programmation de l'interface graphique. Xlisp-Stat fournit un type "objet" et une série de fonctions capables de les traiter. Par ailleurs, la partie 3D de Xlim3D utilise une hiérarchie d'objet pour traiter les éléments structurants. Idéalement, les images devraient être des objets mais on est encore loin d'une telle situation. Nous invitons le lecteur à consulter le document [4] pour une description détaillée de l'orientation objet dans le cadre de Xlisp-Stat.

Nous allons à présent décrire une série de fonctions qui s'appliquent aux différents. La liste n'est bien sûr pas exhaustive mais elle comporte les instructions les plus utiles.

1.3 Opérations sur des nombres

Avant d'entamer une description des fonctions qui traitent les nombres, il faut faire une remarque concernant la présentation des résultats. Un chiffre `3` apparaissant tel quel à l'écran peut être un entier ou un réel. Sachant que la fonction `sqrt` retourne toujours un réel, on peut comprendre ce qui se passe quand on exécute ces quelques lignes:

```
> (sqrt 9)             ; On prend la racine carr\'ee de 9
3
> (integerp (sqrt 9))
NIL                    ; Le resultat n'est pas un entier ...
> (floatp (sqrt 9))
T                      ; C'est un r\'eel
>
```

Conversion du type des nombres. Il est fondamental d'être capable de convertir des nombres d'un type dans un autre. Pour convertir un réel en un entier, les opérations sont `(truncate a)`, qui rend entier en supprimant la partie décimale, et `(floor a)`, qui tronque toujours vers le plus petit entier. Pour convertir un entier en réel, il faut faire `(float a)`.

Opérations arithmétiques. Voici les opérations importantes que l'on peut faire sur les nombres: il y a les opérations arithmétiques, qui peuvent prendre un nombre variable d'arguments (2 au minimum):

- `(+ a b c)` Addition de `a`, `b`, `c`;
- `(- a b c)` Soustraction de `a`, `b`, `c`;
- `(* a b c)` Multiplication de `a`, `b`, `c`;
- `(/ a b c)` Division de `a` par `b` et `c`. Calcule $(a/(b \times c))$;
- `(min a b c)` Minimum entre `a`, `b`, `c`;
- `(max a b c)` Maximum entre `a`, `b`, `c`;
- `(rem a b)` Reste de la division de `a` par `b` (Exemple: `(rem 3.5 1.1)` fournit `0.2`).

Si un des arguments des opérations précédentes est réel, le résultat sera réel lui aussi. Une remarque importante sur la division. Avec l'ancien interpréteur Xlisp, si tous les arguments de la division étaient des entiers, la division était entière, i.e. le résultat était tronqué à l'entier. Ce n'est plus le cas dans la version actuelle de `Xlim3d` qui retourne parfois un réel même si les arguments sont entiers. Par exemple,

```
> (/ 5 3)      ; Un division entre nombres entiers
1.66667        ; ... ne produit pas un entier
> (integerp (/ 6 3))
T              ; sauf si a est un multiple de b
>
```

Les opérateurs arithmétiques peuvent aussi s'appliquer à des listes ou à un mélange de listes et de nombres. On dit alors que les fonctions ont été "vectorisées". Quelques commandes illustrent la richesse du langage.

```
> (+ '(1 2 3) '(4 5 6))  ; On additionne deux listes de meme longueur
(5 7 9)
> (+ 5 '(1 2 5))         ; Le nombre est ajoute a chacun des elements
(6 7 10)
>
```

Il y a aussi des opérations à une seule opérande:

- `(1+ a)` retourne `a + 1`.
- `(1- a)` retourne `a - 1`.

Opérations particulières. Voici quelques opérations qui peuvent servir:

- `(abs a)` Valeur absolue de `a`;
- `(sin a)` Sinus de `a`. L'angle est exprimé en radians;
- `(cos a)` Cosinus de `a`;
- `(tan a)` Tangente de `a`;
- `(sqrt a)` Racine carrée de `a` (le résultat est nécessairement un réel);
- `(exp a)` Exponentiation e^a (le résultat est un réel);
- `(expt a b)` ou `(^ a b)` produit a^b où `a` et `b` peuvent être des entiers ou des réels;
- `(log a)` Logarithme népérien;
- `(random a)` retourne un nombre aléatoire entre 0 et `a-1`.

1.4 Opérations sur quelques types de données particulières

1.4.1 Chaînes de caractères

Le Lisp contient tout ce dont on a besoin pour traiter les chaînes. Cette section ne mentionne que ce qui est le plus fréquemment rencontré.

Lorsqu'on veut imprimer des chaînes de caractères, il peut s'avérer utile de rajouter les caractères de contrôle suivants, un peu tels qu'ils sont définis dans le langage C:

- retour de chariot;
- retour de chariot sans saut de ligne;
- caractère de tabulation.

Pour concaténer des chaînes, on peut utiliser la fonction `(strcat a b c ...)` qui concatène les chaînes `a b c`. La fonction `(format)` produit le même genre de résultat mais sa mise en œuvre est plus lourde.

Il y a également tout un ensemble de fonctions de comparaison. Par exemple, la fonction `(string= a b)` qui compare deux chaînes de caractères. Avec cette fonction, on peut également comparer des sous-chaînes en utilisant les clés `:start1 :start2 :end1 :end2` qui indiquent respectivement le numéro du caractère du début de la comparaison pour la chaîne 1, pour la chaîne 2, et les fins de comparaison. Voici quelques exemples:

```
(string= "J Smith" "K Smith" :start1 1 :start2 1)
; supprime les premiers caracteres, et retourne T
(string= "a" "b")     ; retourne NIL
(string= "a" "a")     ; retourne T
```

Pour couper des morceaux de chaînes, il y a aussi une fonction: `(subseq string start end)`. Cette fonction extrait de la chaîne ce qui est entre `start` et `end`.

Les sections suivantes décrivent quelques fonctions pour gérer des listes et des tableaux.

1.4.2 Listes

Presque par définition, Xlisp est capable de travailler sur des listes. D'abord une petite explication sur ce qu'est une liste. Ensuite, nous détaillerons quelques fonctions de traitement de listes.

Une liste est un ensemble de taille variable d'objets quelconques compris entre parenthèses. On peut également avoir des listes de listes, des listes de tableaux, et ainsi de suite. `(12 15 18 22)` et `(12 "a" 3.14)` sont toutes deux des listes.

Il y a une quantité énorme de fonctions qui servent à gérer des listes. Étant donné que ce sont fréquemment des structures imbriquées, il peut être assez difficile de savoir à quoi on a accès tant les nuances, qui font la fierté des lispiens, sont nombreuses.

Voici les fonctions jugées les plus utiles:

- `(list a b c ...)` crée une liste avec les éléments `a b c ...`. On peut aussi mettre une liste à l'intérieur d'une autre, comme dans l'exemple `(list 1 2 (list 3 4))` qui retournera `(1 2 (3 4))`;
- `(first a)` extrait et retourne le premier élément de la liste `a`;
- `second, third, fourth` retournent respectivement le second, le troisième et le quatrième élément de la liste;
- `(nth n liste)` retourne le n-ème élément de la liste;
- `(rest liste)` retourne la liste moins le premier élément;
- `(last liste)` retourne une liste composée du dernier élément de la liste. `(append liste1 liste 2)` ajoute la liste 2 au bout de la liste 1 pour ne former qu'une liste. Par exemple: `(append '(1 2 3) '(4))` retourne `(1 2 3 4)`.

Quelques lignes pour illustrer l'usage des fonctions:

```
> (setf a (list 1 2 3 4))
(1 2 3 4)          ; Voici ce que Xlim3d retourne
> (setf b (list "Hello" "world"))
("Hello" "world")
> (setf c '(1 2)) ; Remarquer la presence du quote qui evite l'evaluation
(1 2)
> (list a b c)
((1 2 3 4) ("Hello" "world") (1 2)) ; Listes imbriquees
> (append a b c)
(1 2 3 4 "Hello" "world" 1 2)
> (rest a)
(2 3 4)
> (last a)
(4)
> (second b)
"world"
```

1.4.3 Tableaux

Les tableaux sont des structures qui ressemblent beaucoup à celles rencontrées dans des langages de programmation plus traditionnels comme le Fortran. On accède à un élément d'un tableau en spécifiant son numéro. L'indice du premier élément est zéro. Les tableaux, comme en Fortran, ne peuvent pas être rallongés aisément. Si on ne connaît pas la taille initiale d'un tableau et que la longueur est susceptible de changer constamment, il est préférable d'utiliser des listes.

La différence notable des tableaux avec les autres langages, c'est qu'il est possible de stocker des données de type différents dans un même tableau. Par exemple, l'élément 0 peut stocker un entier, l'élément 1 une liste, etc. Comme toujours, pour faire des tableaux multidimensionnels, il faut stocker des tableaux dans les tableaux.

Voici quelques fonctions pour gérer des tableaux:

- `(make-array taille)` crée un tableau de la taille `taille`;
- `(vector e1 e2 e3 ...)` crée un tableau et l'initialise avec les expressions `e1 e2 e3 ...` La taille du tableau sera ajustée pour contenir toutes les expressions;
- `(aref tableau no)` accède à l'élément `no` du tableau `tableau`.

Voici un exemple:

```
> (setf tableau (make-array 2))     ; Creation d'un tableau
#(NIL NIL)
> (setf (aref tableau 0) 0)         ; Assignation: tableau[0]<=0
0
> (setf (aref tableau 1) '("a" 1)) ; Assignation: tableau[1]<=(list "a" 1)
("a" 1)
> tableau                           ; Imprime le contenu du tableau
#(0 ("a" 1))
```

1.5 Opérations booléennes et tests

Ces opérations sont des tests afin de manipuler et de générer les variables booléennes vraies `T` et fausses `NIL`.

1.5.1 Opérations logiques

On peut faire des opérations logiques avec les variables booléennes:

- `(and a b)` *et* logique entre `a` et `b`;
- `(or a b)` *ou* logique entre `a` et `b`;
- `(not a)` *non* logique: si la condition `a` était vraie, le résultat serait `NIL`. Il y a certains effets également lorsqu'on donne une liste: si elle est vide `NIL`, le résultat est `T`.

1.5.2 Tests de comparaison

Il y a un grand nombre d'opérateurs de comparaison. Voici ceux qui sont utiles:

- `(= a b)` vérifie si `a` et `b` sont numériquement égaux. Les arguments peuvent être des entiers ou des réels;
- `(/= a b)` vérifie si `a` et `b` ne sont pas égaux. Attention, $(\backslash = ab) est interprété comme (= \texttt{ab}) et non pas comme$
- `(> a b)` vérifie si `a` est plus grand que `b`;
- `(<= a b)` vérifie si `a` est plus petit ou égal à `b`;
- `(>= a b)` vérifie si `a` est plus grand ou égal à `b`;
- `(null a)` vérifie si la liste `a` est vide.

1.6 Contrôle de boucles

Les lispiens ont tendance à ne pas utiliser les structures de contrôle des programmes de la même façon que le reste du monde. Dans cette section, nous montrerons comment utiliser les structures de contrôle de Lisp avec un fort accent langage C.

1.6.1 Blocs

Il y a quelques fonctions pour construire des blocs en Lisp. Les lispiens font généralement la grimace et préfèrent utiliser d'autres constructions mais, comme il n'est pas question d'esthétique ici, elles valent la peine d'être présentées.

- `(prog1 (fa a) (fb a) (fc a) ...)` exécute la séquence de fonctions `(fa a) (fb a) (fc a) ...` La valeur retournée est celle de la première expression à être évaluée;
- `(progn (fa a) (fb a) (fc a) ...)` même chose que `(prog1 ...)`, sauf que c'est maintenant le résultat de la dernière expression à être évaluée qui sera retournée.

Pour retourner une valeur d'un bloc d'instructions, ou d'une boucle, tout en l'interrompant, utiliser la fonction `(return a)`. Cette fonction retourne du bloc en question la valeur de `a`. Ces fonctions sont exactement comme des constructions de type "begin–end" que l'on retrouve en Pascal.

1.6.2 Prise de décision

Voici les fonctions pour prendre des décisions.

- `(when test (fa a) (fb a) (fc a) ... )` quand `test` est égal à T, exécute la suite d'instructions `(fa a) (fb a) (fc a)...` C'est comme un "if" en langage C, sauf qu'il n'y a pas de clause "else";
- `(unless test (fa a) (fb a) (fc a)...)` même chose que la fonction `(when ...)`, sauf que maintenant, la suite d'instructions sera exécutée quand le test sera égal à NIL;

- `(if test (then) (else optionnel))` si l'expression `test` est vraie, exécute l'expression `then`. Sinon, et s'il y a un else, exécute celui-ci. La valeur retournée est celle de `then` quand `test` est vrai, sinon `else`. S'il n'y a pas de `else`, et que `then` n'est pas évaluée, c'est NIL qui est retourné. Il faut noter que si l'on désire faire plusieurs opérations dans un `if`, il faut utiliser des blocs avec `prog1` ou `progn`.

Quelques exemples:

```
(when (= a b)
     (print "a et b egaux")
     (print (* a b)))

(if (/= a b)
    (progn
         (print "a et b differents")
         (print (+ a b))
    )
    (progn
         (print "ceci est la clause else")
         (print (- a b))
    )
)
```

1.6.3 Boucles itératives

Voici maintenant les manières de faire des boucles:

- `(loop (fa a) (fb a) (fc a) ...)` Cette fonction est une boucle infinie. Elle exécute la séquence `(fa a) (fb a) (fc a) ...` jusqu'à ce qu'il y ait interruption d'une façon ou d'une autre. Généralement, on interrompt ce type de boucle en utilisant la fonction `(return)`. Par exemple:

```
> (setf a 0)
0
> (loop
    (print "Hello")
    (setf a (1+ a))
    (if (= a 3) (return "fin de boucle."))
    (print "non, on continue!")
)

"Hello"
"non, on continue!"
"Hello"
"non, on continue!"
"Hello" "fin de boucle."
>
```

- `(dotimes (a stop ret) (fa a) (fb a) (fc a)...)` Cette fonction est une boucle itérative. `a` part à zéro et continue jusqu'à temps qu'elle atteigne `stop` exclusivement. La variable `a` est une *variable locale*, ce qui signifie qu'elle n'est connue qu'à l'intérieur de la boucle. S'il y a un autre symbole `a` à l'extérieur de la boucle, sa valeur sera conservée. A la fin de la boucle, l'expression optionnelle `ret` est retournée. S'il n'y a pas de `ret` la valeur retournée est `NIL`. Par exemple:

```
(dotimes (a 10 "c'est fini!")
    (print a))
```

  Cet exemple imprimera de 0 à 9 et retournera comme valeur finale `"c'est fini!"`.

- `(dolist (sym liste ret) (fa a) (fb a) (fc a) ...)`
  C'est une boucle itérative. Elle décompose la liste "`liste`" élément par élément, qu'elle assigne à `sym`. La variable `liste` est une *variable locale*. Sa valeur n'est connue qu'à l'intérieur de la boucle. Elle exécute le corps de la boucle avec `sym` comme valeur jusqu'à temps que la liste `liste` soit vide. La valeur optionnelle `ret` est retournée à la fin. Par exemple:

```
> (setf liste (list "un" "deux" "trois"))
("un" "deux" "trois")
> (dolist (chaine liste)
     (print chaine))

"un"
"deux"
"trois" NIL
>
```

1.7 Manipulation des fichiers et impression

1.7.1 Manipulation des fichiers

L'entrée de paramètres ou la sauvegarde des résultats requiert la manipulation de fichiers qu'il faut pouvoir lire ou écrire, ce qui dans la plupart des langages n'est guère aisé. Xlisp-Stat n'échappe pas à la règle. Quelques fonctions élémentaires et puissantes sont décrites ci-dessous. Comme en langage C, on manipule des pointeurs de fichiers qui sont passés d'une fonction à une autre. Ces pointeurs sont appelés "streams".

- `(open "file.txt")` ouvre le fichier dont le nom est "file.txt". La fonction retourne un pointeur sur ce fichier qui peut être ouvert en lecture ou en écriture.
  Par exemple:
  `(setf pointeur (open "file.txt" :direction :input)`
  ouvre le fichier "file.txt" en lecture et
  `(setf pointeur (open "file.txt" :direction :output)`
  ouvre ce fichier en écriture;

- `(read pointeur)` retourne un objet lu dans le fichier pointé quelle que soit sa nature (entier, réel, chaîne de caractères, ...) et passe au suivant. Cette fonction détecte le type de l'objet lu en utilisant la même syntaxe que la fonction `(princ )`(cf. page xix). Ainsi, pour une chaîne de caractères, il faut qu'elle soit entre des guillemets (quotes). C'est la principale fonction de lecture disponible. Contrairement à l'usage dans Xlisp, la fonction ne renvoie pas NIL en cas de fin de fichier mais un message d'erreur. Pour obtenir NIL quand on arrive à la fin du fichier, il faut donner un deuxième argument à la fonction, à savoir `NIL`. La syntaxe exacte est alors `(read pointeur nil)`;

- `(read-line pointeur)` lit une ligne entière du fichier pointé par `pointeur` et retourne la chaîne de caractères correspondant à cette ligne. L'interprétation de la ligne est à charge de l'utilisateur avec les fonctions de manipulation de chaînes de caractères;

- `(close pointeur)` ferme le fichier pointé par `pointeur`. Il est conseillé de fermer un fichier après usage.

L'exemple suivant construit une liste avec tout ce que contient un répertoire:

```
; Cree un fichier contenant le resultat de ls
(system "ls > tmp_ls")
(setf file (open "tmp_ls" :direction :input))
; Initialisation d'une liste
(setf liste (list ))
; Boucle jusqu'a la fin du fichier
(loop
      (if (setf el (read-line file))
            (setf liste (append liste (list el)))
            (return)
      )
      (print liste)
)
```

Voici un exemple qui ouvre un fichier et transfère son contenu multiplié par 2 dans un autre:

```
; Ouverture d'un fichier en lecture et d'un autre en ecriture
(setf input (open "input-file.txt" :direction :input))
(setf output (open "output-file.txt" :direction :output))
(loop
      (if (setf number (read input nil))
            (print (* number 2) output)
            (return)
      )
)
; Fermeture des fichiers
(close input)
(close output)
```

1.7.2 Impression

Dans tous les langages, imprimer les résultats n'est pas une mince affaire. Xlisp ne fait pas exception à la règle. Cette section montre comment on peut se débrouiller pour avoir quelque chose rapidement et simplement.

Les fonctions de type PRINT

Ces fonctions impriment les résultats sur le terminal ou dans un fichier.

- `(print a out)` imprime d'abord un retour chariot suivi de la valeur de `a` (Xlisp faisait l'inverse). L'argument `out` est optionnel: c'est le pointeur à un fichier. S'il est absent, la fonction imprime sur le terminal. Elle retourne toujours la valeur de `a`. Par exemple: `(print "hello")` imprimera la chaîne de caractères sur le terminal. A noter que les chaînes de caractères sont entourées de guillemets;
- `(princ a out)` imprime la valeur de `a` sans un retour de chariot. `out` est optionnel: c'est le pointeur à un fichier. En son absence, la fonction imprime sur le terminal. `princ` retourne la valeur de `a`. Par exemple: `(princ "hello")` imprimera la chaîne de caractères sur le terminal. La chaîne ne sera pas entourée de quotes;
- `(prin1 a out)` fait la même chose que `(princ)`, sauf que les chaînes de caractères sont entourées de quotes;
- `(terpri destination)` imprime un retour de chariot sur la destination, qui est optionnelle. S'il elle n'est pas spécifiée, c'est sur le terminal.

La fonction format

L'usage de cette fonction est assez complexe. Elle est également très puissante et polyvalente: on peut aussi bien imprimer sur un terminal que dans un fichier, et on peut également créer une chaîne de caractères à l'aide de cette fonction.

Voici sa syntaxe:

```
(format destination formattage expr1 expr2 ...  )
```

où

- `destination` est soit un pointeur à un fichier, soit T si on veut imprimer sur le terminal, soit NIL si on veut créer une chaîne de caractères;
- `formattage` est une chaîne de caractères spécifiant le format;
- `expr1 expr2 ...` sont les expressions à imprimer.

Les directives de formattage importantes sont:

- A! : imprime la valeur de l'expression. Si c'est une chaîne de caractères, elle est imprimée sans quotes;
-

- ! imprime un tilde;
- ! seul à la fin d'une ligne, c'est un caractère de continuation.

Un exemple typique de ce que nous avons à faire en traitement d'images: sauvegarder des images (dans l'exemple se sont des nombres) dans des fichiers numérotés de 0 à 5.

```
(dotimes (i 6)
  (setf out-name (format NIL "image~a.txt" i))  ; Cree le nom du fichier
  (setf out (open out-name :direction :output)) ; Ouvre le fichier
  (print i out)  ; Imprime un nombre dans le fichier (ou une autre action)
  (close out)
)
```

1.8 Fichier script

Il peut être intéressant de conserver l'historique de toutes les instructions Lisp que l'on a entrées au clavier. Il existe dans Xlisp la fonction `dribble` qui permet de définir un fichier dans lequel seront enregistrées toutes les commandes; ce type de fichier s'appelle un fichier "script".

Xlim3D contient en plus la fonction `getpid` qui retourne le numéro du processus Xlisp en cours. Ceci permet de créer facilement des noms de fichiers uniques pour les fichiers script. Il suffit d'inclure dans le fichier "init.lsp" la commande suivante: `(dribble (format nil "script~a.lsp" (getpid)))`, utile seulement si l'on désire conserver une trace de toutes les sessions.

Chaque instance du programme travaillera ainsi avec un fichier script distinct.

Chapter 2

Les composantes de `Xlim3d`

Pour des raisons historiques, le logiciel Xlim3D résulte de la fusion de deux logiciels `Xlim` et `Xl3d`, le premier étant réservé à la manipulation d'images bidimensionnelles contrairement au second. La principale conséquence de cette fusion est la coexistence de deux structures d'images. Le manuel de référence indique à chaque fois si les arguments sont des images 2D ou 3D entre parenthèses à côté du mot **Description**. L'importante fonction qui fait le passage entre les images est `ImCopy`.

Dans cette version de `Xlim3d`, on s'est contenté de faire communiquer les deux logiciels entre eux pour qu'ils puissent échanger des données, c'est-à-dire des images. La programmation en Lisp apporte un peu plus de souplesse au système. Ainsi, la fonction `(imxv imin)` permet de visualiser les deux types d'images (voir le fichier "Fondamental.lsp"). On remarquera également que, puisque Xl3d était en mesure de traiter des images 2D, un bon nombre de fonctions actuelles sont redondantes. Pour couronner le tout, deux structures indépendantes de graphe sont apparues dans le logiciel: l'une est adaptée aux grands graphes (elle tirée de `Xlim`), l'autre est préférable pour des petits graphes (`Xl3d`). Bien qu'il soit préférable d'unifier le type, ce n'est pas trop gênant parce que ces deux graphes desservent des objectifs différents.

Les sections suivantes décrivent les composantes de `Xlim3d`. C'est un texte basé sur les rapports de C. Gratin, légèrement adapté à la nouvelle version du logiciel. La partie consacrée aux graphes est le fruit du travail de H. Talbot.

2.1 Les images

2.1.1 La structure d'image 2D

Les pixels d'une image 2D sont définis sur un nombre fixe de bits qui dépend de l'installation (généralement 16 bits). Il y aussi moyen de travailler avec des images de réels mais seulement pour quelques fonctions (comme `fft`).

Un exemple de session avec des images 2D.

```
> (setf in (imread "images/cam.ras"))     ; Lit une image d'entree
raster_read: Reading raster image images/cam.ras: [256][256]...
#<Foo: #f1770>
```

```
> (setf out (timmalloc in))    ; Cree une image de sortie de meme taille
#<Foo: #f1608>
> (ero4 in out 2)              ; Erosion en 4-connexite de taille 2
#<Foo: #f1608>
> (imxv out)                   ; Visualisation
WRITE-RASTER: Writing raster image tmp28960.1: [256][256]...
>
```

Si cette dernière instruction n'est pas disponible, il faut lancer

```
> (raster-write out "tmp_xlim.ras")
> (system "xv tmp_xlim.ras &")
```

2.1.2 La structure d'image 3D

La structure d'image 3D est celle qui est generée par une fonction du package 3D (`ImGet`, `ImGet2D` ou `ImGetSame`). Ces images peuvent être bi ou tridimensionnelles. Les images bidimensionnelles ne sont pas compatibles avec ce qui a été appelé "structure d'image 2D", parce qu'ici il y a nécessairement un bord. Les seules fonctions qui peuvent travailler avec les deux types d'image (c'est-à-dire les structures d'image 2D et 3D) sont les instructions de copie (`ImCopy` et `ImIsCopy`).

Une image 2D du package 3D est définie dans le plan XZ (attention, c'est bien XZ et non XY), tandis que le premier plan d'une image 3D est XY. Cette implémentation permet de traiter la parité des images 2D (trame hexagonale) et des images 3D (trame cubique à faces centrées) avec la même structure d'image. Que ce soit le plan XZ ou XY n'a pas d'importance qu'au moment de lire une image. Signalons que le format "visilog" admet des images tant 2D que 3D. Ainsi nous pouvons faire

```
> (setf im2d (ImGet2D 256 256))
> (ImReadVisilog im2d filename)
```

et aussi

```
> (setf im3d (ImGet 256 256 10))
> (ImReadVisilog im3d filename)
```

Pour utiliser le format "raster", qui lui ne lit que des images 2D, il faudra faire

```
> (setf im2d (ImGet2D 256 256))
> (ImReadRaster im2d filename)
```

Mais les autres fonctions de lecture (`ImReadAnyBitMap`, `ImReadText`, `ImReadAscii`) n'acceptent que des images 3D. Dans un bitmap correspondant à une image 3D, les premiers pixels correspondent au premier plan XY, tandis qu'une image 2D est vue comme un plan XZ. Pour rendre les images 2D compatibles avec les instructions de lecture des images 3D, nous devons utiliser l'instruction `ImSwapYZ` qui permute les coordonnées Y et Z ; c'est-à-dire la lecture d'une image 2D au moyen d'une fonction 3D se réalise comme suit:

```
> (setf im2d (ImGet2D 256 256))
> (ImSwapYZ im2d)
> (ImReadAnyBitMap im2d filename 256 256 20 0 0 0 0 256 256 1)
> (ImSwapYZ im2d)
```

Pour simplifier l'usage, nous avons défini les fonctions `ImReadAnyBitMap2D` et `ImReadAscci2D` dont l'usage est:

```
> (setf im2d (ImGet2D 256 256))
> (ImReadAnyBitMap2D im2d filename 256 256 0 0 0 256 256)
> (ImReadAscii2D im2d "im.asc")
```

Dans chaque image est définie une fenêtre active: c'est la partie de la fenêtre sur laquelle s'effectue effectivement le traitement. Elle est donnée par les coordonnées du premier point de la fenêtre et par ses tailles dans les trois dimensions.

Les images sont codées sur 8 bits non signés, 16 bits non signés ou 32 bits signés. Mais la plupart des fonctions sont écrites pour des images 8 bits. Il n'est donc pas assuré qu'une fonction donnée soit applicable à une image 16 ou 32 bits. Un message d'erreur, sous la forme `Bad image type`, apparaîtra à chaque fois que le type de l'image n'est pas supporté par l'algorithme. Il n'existe pas à proprement parler d'images binaires. Une image binaire sera simplement vue comme une image 8 bits dont les pixels ont des valeurs nulles (i.e. pixel éteint) ou non nulles (i.e. pixel allumé).

Certaines conventions ont été adoptées pour les fonctions de traitement d'images, notamment:

- toutes les fonctions de traitement d'images ont un nom qui commence par `Im` –nous conseillons néanmoins d'utiliser une nomenclature où il n'y aurait plus que `I` pour indiquer qu'il s'agit d'une fonction s'appliquant à une image;
- nous n'avons pas surchargé les opérateurs existant (comme par exemple `add`), ce qui aurait pu être une solution;
- les nombres correspondant à des valeurs de pixels passés comme paramètres ou retournés comme résultat sont toujours donnés sous forme de réels, même s'ils s'appliquent à des images entières (par exemple retourne un nombre réel, et dans `(ImSetConstant a value)` value doit être un réel).

Un exemple de session avec des images 3D.

```
> (setf n1 (ImGet2D 256 256))                ; Creation d'une image source
Image (256,1,256), unsigned char, active window [(0,0,0),(256,1,256)]
> (setf n2 (ImGetSame n1))                   ; Creation d'une autre image
Image (256,1,256), unsigned char, active window [(0,0,0),(256,1,256)]
> (ImReadRaster n1 "images/cam.ras")         ; Initialisation de l'image
Image (256,1,256), unsigned char, active window [(0,0,0),(256,1,256)]
> (ImIsCopy n1 n2)                           ; Copie
Image (256,1,256), unsigned char, active window [(0,0,0),(256,1,256)]
> (Imxv n2)                                  ; Affichage
Image (256,1,256), unsigned char, active window [(0,0,0),(256,1,256)]
>
```

2.2 Les graphes

2.2.1 Les graphes en tant que structure supplémentaire

Dans cette section, nous présentons ce qu'il est nécessaire de savoir pour pouvoir manipuler les graphes définis au niveau du compilateur C afin de comprendre leur utilisation et de pouvoir développer des fonctions personnelles. Pour une meilleure compréhension de ce qui va suivre, il peut être bon de se reporter aux travaux de L. Vincent [5, 6].

Introduction sur les graphes morphologiques. Par graphe *morphologique*, nous entendons une structure reliant des sommets ("vertices"), qui correspondent à des points, par des arêtes ("edges"), dont les sommets ou les arêtes peuvent être valués.

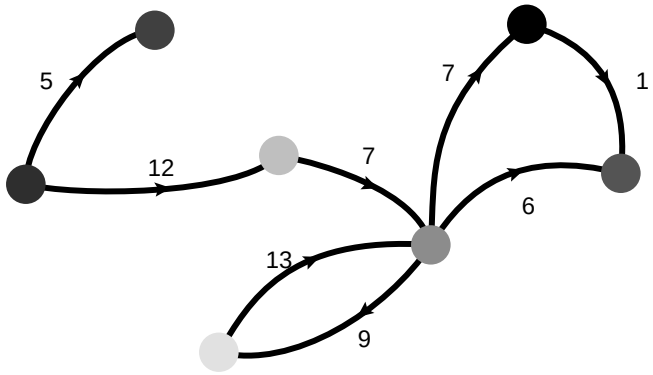

Figure 2.1: Exemple de graphe planaire aux sommets et arêtes valués.

La structure que nous allons décrire permet d'avoir des graphes planaires (dont aucune des arêtes ne se croisent) ou non, des graphes orientés ou non (A est relié a B mais B n'est pas relié a A), connexes ou non, mais d'une manière générale, les graphes que nous allons considérer seront planaires, non orientés et connexes. Vincent s'était lui-même limité aux cas des graphes aux arêtes non valuées. La majorité des opérations que nous comptons faire subir aux graphes ne changent pas leur structure de connexité, mais seulement les valuations des arêtes et des sommets. Les graphes morphologiques sont simplement considérés comme des images à niveaux de gris aux propriétés de connexité variables selon le lieu. Comme l'a montré Vincent, les opérations morphologiques sur les images sont également valables la plupart du temps sur les graphes (érosion, dilatation, ligne de partage des eaux,...). Les images ne sont en général que des graphes aux propriétés de connexité très régulières).

La structure de graphe implémentée est la suivante:

```
typedef struct hgraph {
    int           id;        /* graph kind */
    GWIN         *field;     /* describe a window around graph */
    int           order;     /* number of vertices in a graph */
```

```
    VERTEX       *vertices;  /* array of vertices */
    VALUE        *values;    /* valuation of the vertices */
    int           nb_edges;  /* total number of non-oriented edges */
    int          *neighs;    /* array of back-relation to vertices */
    VALUE        *edval;     /* valuation of the edges */
} HGRAPH;
```

Voila comment cette structure se décompose:

- `id` est un entier qui décrit à quel type de graphe on a affaire:
  - `IDGR_DELAUNAY (1)` : triangulation de Delaunay;
  - `IDGR_GABRIEL (2)` : graphe de Gabriel;
  - `IDGR_RNG (2)` : graphe de voisinage relatif.

  D'autres identifications de graphe sont bien sûr possibles: arbres de recouvrement minimal, par exemple;
- `field` est une petite structure qui décrit les dimensions d'une fenêtre nécessaire pour contenir le graphe;
- `order` est juste le nombre de sommets du graphe;
- `vertices` est un simple tableau de sommets. La structure de sommet contient la position de chaque sommet dans la fenêtre, et un index dans `neighs[]` sur le premier de ses voisins;
- `values` est un simple tableau de même taille que `vertices`, et contient la valeur (niveau de gris) de ce sommet;
- `nb_edges` est le nombre d'arêtes du graphe (asymptotiquement 6 fois le nombre de sommet dans le cas de graphes (triangulation de Delaunay) créés à partir d'images mosaïque);
- `neighs` est un tableau d'assez grande taille. Il contient des index de sommets dans `vertices`, et permet de connaître les relations entre tous les sommets. Comme il est décrit plus haut, chaque structure de sommet contient un indice dans ce tableau, qui correspond au premier de ses voisins. Les voisins suivant du sommet considéré sont rangés dans l'ordre croissant dans ce tableau. Pour connaître l'ensemble des voisins d'un sommet, il suffit de regarder dans `nb_edges` à partir de l'index dans la structure sommet du sommet en cours, jusqu'à l'index de dans la structure sommet du sommet suivant. Les index rencontrés sont ceux des voisins du sommet en cours dans `vertices`. La figure 2.2 permet de mieux comprendre cette organisation;
- `edval` est un tableau de même taille que `neighs`, et contient les valuations des arêtes du graphe.

Cette structure est bien sur un compromis entre plusieurs contraintes. Elle a les avantages suivants:

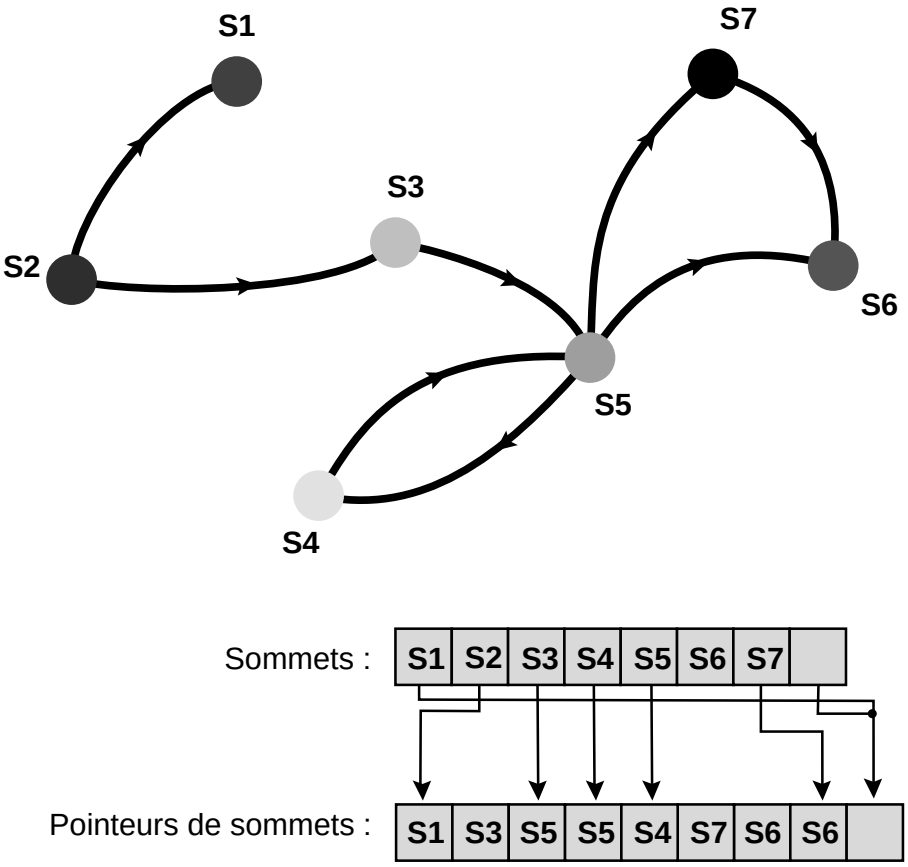

Figure 2.2: Codage par sommets. Chaque sommet pointe vers ses voisins stockés dans un tableau de pointeur de sommets (indices).

1. Elle est relativement souple: elle permet de stocker aussi bien les graphes orientés que non, les graphes planaires ou non, etc;

2. Elle est relativement simple d'accès: les voisins d'un sommet s'obtiennent facilement;

3. Par contre, elle est gourmande en place surtout dans le cas de graphes non orientés, aux arêtes non valués. Chaque arête est immuablement notée deux fois: une fois comme lien entre A et B, une autre fois comme lien entre B et A;

4. De plus, elle n'est pas très souple lorsqu'il s'agit d'ajouter ou d'enlever une arête à un graphe. De manière interne, on aura intérêt à revenir à une structure basée sur des listes chaînées lorsqu'on voudra traiter ce genre de problèmes.

2.2.2 Les graphes relatifs à la partie 3D

Quand on désire parler de connexité, on définit un graphe: un réseau d'arcs reliant les points de l'image entre eux. En réseau hexagonal, on pourra par exemple relier un point aux six points les plus proches. En réseau carré, on le reliera tantôt aux 4 points les plus proches (4-connexité), tantôt aux 8 points les plus proches (8-connexité). Un graphe est défini par une liste (l'objet lispien) de vecteurs, c'est-à-dire une liste de triplets de valeurs réelles représentant les coordonnées relatives (x,y,z) des points reliés par un arc au point considéré.

Une "parité" a été introduite sur les plans (sur les lignes en 2D) de sorte que l'on puisse travailler en réseau cubique à face centrée (hexagonal en 2D). Cette parité apparaît au niveau de la définition du graphe puisqu'alors les coordonnées relatives par rapport au point central peuvent être non entières.

Exemple:

```
(def hexagonal6Gr '((-0.5 0.0 -1.0) (0.5 0.0 -1.0) (-1.0 0.0 0.0) (1.0 0.0
0.0) (-0.5 0.0 1.0) (0.5 0.0 1.0)))
```

Ceci définit les six voisins d'un point en réseau hexagonal. Attention: remarquez que, puisqu'il s'agit d'un graphe s'appliquant à une image 2D, la deuxième coordonnée est toujours nulle (et non pas la troisième).

Un certain nombre de graphes standard ont été créés dans le fichier "Graphs.lsp": `square4Gr` (défini dans le plan XZ), `square4XYGr` (défini dans le plan XY), `square8Gr` (défini dans le plan XZ), `square8XYGr` (défini dans le plan XY), `hexagonal6Gr, cubic6Gr, cubic26Gr, cfc12Gr`. Ces variables pourront être utilisées comme paramètres d'un certain nombre de fonctions Lisp (comme `ImMinima, ImFillHoles, ImWatershed...`).

ATTENTION: les vecteurs définissant un graphe NE DOIVENT PAS posséder de coordonnée strictement supérieure à 1 en valeur absolue. La raison en est que les algorithmes ne font aucun test de présence des points dans l'image. Pourquoi 1 ? Car toutes les images possèdent un bord supplémentaire de 1 pixel (qui n'apparaît qu'au niveau de la programmation en C). Si les coordonnées dépassent 1 vous devrez alors travailler sur une sous-image de vos images en utilisant les fonctions `ImSetWindow` et `ImSetGlobalWindow`. Mais la gestion des bords risque alors de ne pas être assurée convenablement. Prudence donc...

2.3 Noyaux de points

On utilisera aussi dans les fonctions de Xlim3D une structure appelée *noyau* (*Kernel*) qui est en fait identique à la structure *graphe* –c'est une liste de vecteurs– mais dont le rôle est différent. On définira un noyau K par exemple pour déterminer la moyenne, la médiane, ou bien encore la plus petite des valeurs d'une image sur K. A la différence du graphe, le noyau pourra contenir le vecteur nul.

Exemple:

```
(def hexagonK '((-0.5 0.0 -1.0) (0.5 0.0 -1.0) (-1.0 0.0 0.0) (0.0 0.0 0.0)
(1.0 0.0 0.0) (-0.5 0.0 1.0) (0.5 0.0 1.0)))
```

Un certain nombre de noyaux standard ont été créés dans le fichier "Kernels.lsp": `diamondK` (défini dans le plan XZ), `diamondXYK` (défini dans le plan XY), `squareK` (défini dans le plan XZ), `squareXYK` (défini dans le plan XY),`hexagonK, octaedronK, cubeK, cuboctaedronK`. Ces variables pourront être utilisées comme paramètres d'un certain nombre de fonctions Lisp (comme `ImMean, ImInfKernel, ImConvolve...`).

Comme dans le cas des graphes, si les coordonnées dépassent 1, vous devrez travailler sur une sous image de vos images en utilisant les fonctions `ImSetWindow` et `ImSetGlobalWindow`. La gestion des bords risque encore de ne pas être assurée convenablement. Prudence donc, une fois encore.

2.4 Les éléments structurants

Les éléments structurants sont implantés sous la forme d'objets. Ils peuvent être de plusieurs formes:

1. décomposé comme somme d'éléments structurants élémentaires, c'est-à-dire d'éléments structurants définis sur un voisinage de taille 1. Un tel élément structurant B (appelé "Steiner") s'écrit:
$$B = B_1 \oplus B_2 \oplus ... \oplus B_n$$

2. homothétique d'un élément structurant convexe élémentaire. C'est un cas particulier du cas précédent:
$$B^n = B \oplus B \oplus ... \oplus B$$

3. quelconque, défini sous la forme d'une image.

Ces distinctions sont d'origine technique: on n'utilise pas le même algorithme selon qu'un élément structurant peut être décomposé ou pas, pour des questions de vitesse de calcul.

Il existe 4 classes d'éléments structurants:

- La classe de base est la classe **SElement**. L'implantation de l'élément structurant n'est pas connue. Elle sert simplement de classe-mère aux classes qui vont suivre. Elle ne possède ni champ de données ni méthode;

- **DecomposedSE**, sous-classe de la classe `SElement`, est la classe des élément structurants du cas 1. Un élément structurant est alors défini par une liste de noyaux de points (Kernel vu plus haut), chaque noyau étant lui-même une liste de vecteurs;

- **HomoteticSE**, sous-classe de `DecomposedSE`, correspond aux éléments structurants du cas 2. Un élément structurant est alors défini par la donnée d'un noyau de points (Kernel) et d'une taille;
- Enfin **ImageSE**, sous-classe de `SElement`, est une classe d'éléments structurants a priori quelconques. Ils sont définis par une image.

Un certain nombre d'instances de la classe `HomoteticSE` ont été créées (fichier "SElements.lsp"). Elles correspondent aux boules de base associées aux différents graphes: `diamondSE` (défini dans le plan XZ), `diamondXYSE` (défini dans le plan XY), `squareSE` (défini dans le plan XZ), `squareXYSE` (défini dans le plan XY), `hexagonSE`, `octaedronSE`, `cubeSE`, `cuboctaedronSE`. Ces instances ont été initialisées avec une taille égale à 1. Cette taille peut bien entendu être modifiée par l'utilisateur quand il le désire (voir la méthode `:Size`).

Chapter 3

Développement de programmes

Le développement de nouvelles fonctions peut s'effectuer soit au niveau de l'interpréteur, soit au niveau du langage compilé suivant le type de fonctions à réaliser. Reste à savoir comment débogger un programme ou ajouter une fonction à une librairie. Nous allons décrire des solutions pour ce type de problèmes pour les deux niveaux.

3.1 Développement de programmes Lisp

3.1.1 Ajout d'une fonction

Pour qu'elle soit utilisable au niveau Lisp, une fonction doit être définie en suivant un prototype particulier:

```
(defun nom (arguments) "Aide en ligne" (corps de la fonction))
```

Passons en revue les différentes composantes du prototype:

- **nom**. Le nom de la fonction s'écrit en un mot. Il peut être quelconque mais il est conseillé de "calquer" la nomenclature existante. Nous mettons aussi en garde les programmeurs qui redéfinissent des fonctions existantes; ce n'est pas une bonne façon de faire. Pour avoir une liste des fonctions fournies, consultez l'index des fonctions à la fin de ce document;

- **arguments**. La définition des arguments est assez complexe car il y a moyen de définir des arguments optionnels et variables (`&optional`, `&rest`), ainsi que d'introduire des clés (`&key`) dans la définition ou encore de limiter le domaine de validité d'une variable par `&aux`. Avant de passer à l'examen de l'usage de ces mots, il convient de préciser le domaine d'existence d'une variable. Les arguments qui ne sont précédés d'aucun mot-clef sont copiés dans ces variables (muettes) au moment d'entrer dans la fonction. Les arguments déclarés dans le prototype seront locaux à la fonction. Si on utilise des arguments qui ne sont pas déclarés dans la fonction, ils seront considérés comme globaux. Exemple:

```
> (setf x 10)                ; La variable x est mise a 10
10
> (defun f (x) (setf y 4)(setf x 5)) ; Une fonction qui met x a 5
```

```
F                                        ; (et y a 4)
> (f x)                          ; On tente de modifier x
5
> x                              ; La valeur de x n'a pas change
10
```

Par contre, le rôle de `(setf y 4)` est moins clair. Comme telle l'instruction va affecter la valeur d'une variable qui aurait existé précédemment. Pour éviter cet effet de débordement, il faut utiliser le mot `&aux` dans la définition de la fonction. Voici quelques lignes d'illustration:

```
> (setf y 10)
10
> (defun f (x &aux y) (setf y 4) (setf x 5)) ; Noter la presence
F                                            ; de &aux
> (f 5)
5
> y
10                     ; La variable globale y n'est plus modifiee
>
```

On place généralement `&aux` à la fin des arguments. Signalons une erreur fréquente: `&aux` limite l'existence d'une variable à la fonction mais ne libère pas un bloc de donnée qui lui est attaché. Ainsi, si `y` était une image, le nœud `y` serait libéré à la fin mais pas la zone de mémoire contenant les pixels. Pour remédier à cela, il faudra libérer explicitement la zone de mémoire des images (par exemple avec `imfree`) avant de sortir de la fonctions sinon on sera rapidement à cours de mémoire.
Le mot `&rest` prend tous les arguments restants. Voici un exemple d'utilisation qui définit une addition avec un nombre quelconque d'arguments:

```
> (defun add (x &rest y)   ; y est une liste avec tous les arguments
     (loop
        (if (null y) (return))
        (setf x (+ x (first y)))
        (setf y (rest y))
     )
     x
)
ADD
> (add 4)
4
> (add 4 5 68)
77
>
```

Les mots `&optional` et `&key` sont plutôt destinés à traiter des arguments optionnels. Le fonctionnement de `&optional` est trivial sauf en ce qui concerne la valeur par défaut

de l'argument qui lui est attaché. Cet argument vaut `NIL` par défaut. Si on veut modifier cette valeur par défaut, on place l'argument avec la valeur entre parenthèses. L'implémentation de la fonction factorielle sous forme récursive illustre le fonctionnement:

```
> (defun factoriel (n &optional imp)   ; imp dit s'il faut afficher
     (if imp (format T "n = ~a\n" n))  ; les resultats intermediaires
     (if (= n 0) 1 (* (factoriel (- n 1) imp) n)))
FACTORIEL
> (factoriel 3)
6
> (factoriel 3 T)
n = 3
n = 2
n = 1
n = 0
6
>
```

Pour imprimer par défaut, on remplace la première ligne par

```
> (defun factoriel (n &optional (imp T)) ; imp dit s'il faut afficher
```

Comment faire lorsqu'on a plusieurs arguments optionnels? On peut recourir au mot `&key`. La syntaxe de définition est proche de celle de `&optional`; seule la manière de mettre en œuvre la variable diffère.

```
> (defun 5+ (x &key imp (ajouter 5)) ; Definit deux arguments
    (if imp (+ x ajouter)))          ; optionnels. Par defaut, imp
5+                                   ; vaut NIL et ajouter vaut 5.
> (5+ 10)
NIL
> (5+ 10 :imp T)
15
> (5+ 10 :imp T :ajouter 5.1)     ; On a modifie la valeur par defaut
15.1
>
```

Nous conseillons vivement l'usage du mot `&key` au lieu de `&optional` ou de `&rest`;

- **aide en ligne**. Il s'agit d'une chaîne de caractères qui apparaît lorsqu'on lance la commande `(help 'nom)`. La manière habituelle d'écrire l'aide est

```
"Args: (x y)
Voici ce que fait la fonction."
```

- **corps de la fonction**. On place tout ce que l'on veut dans le corps d'une fonction. La dernière expression évaluée est retournée.

3.1.2 Chargement de fichiers

La fonction pour charger des fichiers Xlisp est `(load "fichier")`. Le fichier, qui doit obligatoirement se terminer par ".lsp", sera alors lu et interprété exactement comme s'il avait été tapé au terminal. L'extension par défaut est `.lsp`. Si tout se déroule bien, l'interpréteur retourne la valeur `T`.

3.1.3 Déboggage

Trois outils de déboggage sont disponibles dans l'interpréteur: l'arrêt en cours d'exécution, le traçage des arguments et l'exécution pas à pas. La plupart des langages Lisp les fournissent mais leur comportement diffère sensiblement d'un système à une autre. La description entamée ici est propre à Xlisp-Stat.

Arrêts lors de l'exécution

Le moyen le plus simple d'interrompre l'exécution est l'utilisation de la fonction `break`. Placer l'instruction à l'intérieur du corps d'une fonction permet d'examiner le contenu des variables. Le programme se poursuit dès que l'on tape `(continue)`. Voici un exemple:

```
> (defun fonction (x) (setf x (+ x 5)) (break) x)
FONCTION
> (fonction 10)
break: **BREAK**
if continued: return from BREAK
1> x
15
1> (continue)
[ continue from break loop ]
15
```

Le nombre qui apparaît à gauche de "`>`" indique le niveau dans lequel a lieu l'interruption. `break` accepte une chaîne de caractères comme argument optionnel. Dans ce cas, la chaîne apparaîtra lors de l'arrêt.

Le système génèrera un arrêt de type `break` lors d'une erreur à l'exécution, même si cette instruction n'est pas placée dans le code, à condition que la variable globale `*breakenable*` ne vaille pas `nil`. Il en va de même pour la variable `*tracenable*` qui imposera l'impression d'une "trace" des appels ayant précédé l'interruption. `baktrace` permet d'imprimer les messages si `*tracenable*` vaut `nil` malgré tout. Les commandes `(debug)` et `(nodebug)` permettent de basculer plus aisément la valeur de la variable `*breakenable*`.

Tracer une fonction

Typiquement, un programme plante parce que des mauvais arguments lui ont été fournis. Lorsqu'il est planté quelque part, il indique dans le prompt le numéro du niveau de déboggage. On peut alors examiner les arguments en faisant `(baktrace n)` où n est le nombre de niveaux dans la pile. Voici un exemple:

```
> (debug)          ; Initialisation du mode de deboggage
T
> (defun aire (r)  ; Une fonction qui calcule l'aire d'un disque
(* r r 3.14))
AIRE
> (aire "foo")     ; On donne un mauvais type d'argument
error: not a number - "foo"
1> (baktrace)
Function: #<Subr-BAKTRACE: #b92f8>
Function: #<Subr-*: #d0870>
Arguments:
  "foo"
  "foo"
  3.14
Function: #<Closure-AIRE: #1a3190>
Arguments:
  "foo"
NIL
1>
```

Il faut un peu d'expérience pour déchiffrer de tels messages.

Il est possible au cours de l'exécution d'utiliser une fonction qui montre la trace des appels de fonction. Cette fonction s'appelle `(trace fn)` et elle donne la liste des arguments qui sont soumis à la fonction `fn`. Un exemple:

```
> (trace aire)   ; On veut tracer la fonction aire.
(AIRE)
> (aire 1 2 3 4)
Entering: AIRE, Argument list: (1 2 3 4)
error: too many arguments
1>
```

Pour supprimer une fonction de la "trace", il faut faire `(untrace fn)`. La fonction `fn` sera supprimée de la liste des fonctions à tracer. Ces deux fonctions sont très utiles pour localiser les problèmes.

Exécution pas à pas

Comme dans `gdb`, Xlisp-Stat prétend pouvoir observer le déroulement du programme pas à pas au moyen de l'instruction `step`. Ce mode de fonctionnement semble réservé à Macintosh car il ne marche pas dans la version actuelle de l'interpréteur.

Niveaux de déboggage

Il y a des niveaux en mode debug. Il faut être capable de revenir à un niveau antérieur. Cela se fait au moyen de deux fonctions: `(clean-up)` (qui remonte d'un niveau) et `(top-level)` ou `Ctrl-C` (qui ramène au sommet). Voici un exemple:

```
> (debug)
```

```
T
> (+ "foo" "ree")
error: not a number - "foo"
1> foo
error: unbound variable - FOO
if continued: try evaluating symbol again
2> (clean-up)
[ back to previous break level ]
1> ree
error: unbound variable - REE
if continued: try evaluating symbol again
2> (top-level)
[ back to top level ]
>
```

Quelques subtilités du système

La mémoire est quelque chose de critique en Lisp. On en manque toujours. Voici quelques fonctions pour la gérer.

Un interpréteur Lisp a des structures qu'on appelle "nœuds". Ces nœuds servent à stocker les éléments des listes et à les coller ensemble. Au cours des manipulations, les nœuds sont fréquemment déliés et se retrouvent dans les limbes: il n'est plus possible de les trouver pour les ré-utiliser. Une fonction pour ramasser ces nœuds "déchets" s'appelle un "garbage collector". Il y a eu une quantité considérable de recherches pour optimiser ce garbage collector, et les lispiens en sont fiers. Par ailleurs, le fait d'examiner tous les nœuds existants pour trouver ceux qui sont libres prend du temps qui pourrait être consacré au calcul. La fonction pour forcer la collecte des déchets est `(gc)`. Elle est fréquemment invoquée automatiquement. En effet, chaque fois que Xlisp manque de mémoire, il fait appel à cette fonction. S'il continue à en manquer, il faudra qu'il en demande au système d'exploitation. Ceci est fait par la fonction `(expand n)` où `n` est le nombre de paquets de nœuds qu'il demandera. Habituellement, un paquet de nœuds comprend approximativement 1000 nœuds. Cette fonction est appelée automatiquement, mais on peut le faire aussi manuellement.

On peut voir les statistiques de mémoire de Xlisp en faisant la fonction `(room)`.

Une autre fonction qui n'est pas portable et qui n'a pu entrer dans les catégories de fonctions décrites dans ce rapport: la fonction `(system "commande")` qui exécute la commande demandée. Cette commande dépend évidemment des possibilités du système d'exploitation.

3.2 Développement de librairies compilées

3.2.1 Ajout d'une fonction

Ajouter une fonction d'une librairie compilée n'est pas aussi simple qu'au niveau Lisp. En plus de la définition du niveau Lisp, il faut lier la fonction de sorte que l'interpréteur aille chercher le code au moment de l'appel de la fonction.

Il y a trois sortes de fichiers à modifier ou à créer:

1. Un fichier de définition Lisp. Dans ce fichier se trouveront les noms de nouvelles fonctions créées lors d'un appel dynamique;

2. Les sources de la librairie: le cœur de l'apport au système existant;

3. Un fichier "colle" qui établit une passerelle entre le niveau Lisp et le niveau C. Ces fichiers se terminent généralement par "glue.c".

Pour illustrer la démarche, nous préférons renvoyer à des fichiers existants qui pourront servir de modèle (notamment pour l'écriture de la colle). Examinons le cas de la fonction C `minval()` fournie dans le package `2d`. Cette fonction est contenue dans le fichier "src/-imlib/jfr/imstat.c". Telle quelle, la fonction n'est pas utilisable parce que les arguments de Lisp sont différents. Il faut alors écrire une fonction `xminval` qui fait l'interface et opère le passage des arguments. Cette dernière est comprise dans le fichier "xlimglue.c" du même répertoire. Bien que la fonction puisse être insérée dans le code C de l'interpréteur, ceci ne suffit pas encore parce que l'interpréteur ne trouvera pas de fonction du nom de `minval`. Tout fonctionnera correctement lorsque sera définie la fonction (ou macro) Lisp "minval" comme dans le fichier "2d.lsp" du même répertoire. Notons au passage que cela peut se faire en utilisant de préférence la fonction `defun` qui se chargera elle-même d'examiner les arguments, soulageant de devoir le faire au niveau C. De plus, on peut gérer la création d'une image de sortie au niveau Lisp plutôt qu'au niveau C. Ainsi, au lieu de définir `lpe4` par:

```
(defmacro LPE4 ( &rest z)
    `(call-lfun "xlpe4" ,@z))
```

il vaut mieux écrire

```
(defun LPE4 (ImMinima ImOriginal ImOut)
" Args: ImMinima ImOriginal ImOut
 Watershed with reconstruction to remove minimums ..."
    (call-lfun "xlpe4" ImMinima ImOriginal ImOut)
    ImOut  ; Returns the ouput image
)
```

Après adaptation du "Makefile" local, tout est en place pour que la fonction soit disponible dans le `package` en question.

Si vous cherchez de l'inspiration pour programmer dans la partie 3D, vous pouvez consulter des fichiers source dans le répertoire "src/imlib/3d". Principalement, regarder dans l'ordre les fichiers *.c et *.h suivants: "3dstruc.c, 3dimage.c, 3dmisc.c, 3dio.c" et "3derror.c". Ce sont eux qui contiennent les principales fonctions de création et d'accès aux images. Les autres fichiers contiennent les primitives de traitement d'images.

3.2.2 Exemple: développer une fonction qui traite les graphes

Dans cette partie nous présentons ce qu'il est utile de savoir pour pouvoir ajouter des fonctions personnelles de manipulation de graphe dans Xlisp-Stat. Cet exemple est tiré d'un document de H. Talbot.

La philosophie de la manipulation des graphes au niveau du *C* est extrêmement proche de celle de la manipulation des images dans `Xlim3D`. On a besoin de savoir:

- Comment échanger des données entre le Lisp et le C;
- Comment manipuler les structures de graphes.

De même que pour rajouter des fonctions de traitement d'image à `Xlim3D`, on va ici avoir besoin d'échanger les arguments entre le C et le Lisp, après avoir écrit un programme de traitement de graphe en C. Ceci s'opère au moyen d'une *fonction colle (glue)*, écrite en C, qu'il va falloir déclarer.

Le mieux est sans doute encore de donner un exemple. Imaginons que nous voulions ajouter une fonction de traitement de graphe, par exemple la dilatation simple. Nous appellerons cette fonction `gdil`. Cette fonction va prendre trois arguments : un graphe d'entrée, un graphe de sortie, et une taille, qui correspond à un nombre d'itérations.

Déclaration du nom de la fonction Lisp

Pour être capable d'appeler la fonction C de dilatation à partir du Lisp, il faut déclarer un nom Lisp qui correspond à l'appel de la fonction `xgdil()`. Depuis que Xlim3D permet le chargement dynamique de fonctions, la déclaration se fait dans un fichier Lisp qui, dans ce cas, s'appelle "graphs.lsp" (il est dans les "packages").

Faites bien attention : `GDIL` est le nom Lisp de la dilatation. En clair on appellera la dilatation en Lisp par : `(gdil gin gout size)`. Quand à `xgdil`, c'est le nom de la fonction colle, que nous allons détailler maintenant.

Écriture de la fonction colle

La fonction colle est une fonction en C qui se charge de récupérer les arguments du Lisp, de les convertir en arguments compréhensibles par le C, d'appeler la "vraie" fonction de dilatation, de récupérer les résultat et le cas échéant, de les retourner au Lisp. On pourrait normalement la déduire d'un simple prototype de la fonction C de dilatation parce ce que les deux fonctions sont très proches.

Nous allons détailler plusieurs aspects de l'écriture de la fonction colle mais une remarque importante s'impose avant: avec la nouvelle version de `Xlim3D`, il est possible de déporter une partie de la gestion de la colle au niveau Lisp, c'est-à-dire dans le fichier "graphs.lsp". Cela est préférable tant pour des questions de facilité d'écriture que pour les besoins de gestion du logiciel. Ainsi, vérifier le type d'un argument s'effectuera de préférence au niveau Lisp si cela est possible.

1. Récupérer les arguments et les convertir. On récupère les arguments provenant du Lisp au moyen des macros suivantes (la liste n'est pas exhaustive):

- `xlgetarg()`: permet de lire n'importe quel argument. Des conversions doivent avoir lieu ensuite pour permettre de l'utiliser. Voir la fonction colle de `garith`. Cette macro ne fait pas de vérification de type;
- `xlgafixnum()`: lit un argument entier. Lorsque l'argument lu n'est pas un entier, une erreur est générée;
- `xlgaflonum()`: lit un argument en flottant. Vérifie le type;

- `xlgastring()`: lit une chaîne de caractères. Vérifie le type;
- `xlgaimage2d()`: lit une image 2D. Il faut utiliser `xlgaimage3d()` pour récupérer une image 3D. Le type de l'argument est vérifié;
- `xlgagraph()`: lit un graphe morphologique. Le type est vérifié.

Il existe d'autres macros disponibles, mais celles-ci sont les plus courantes pour gérer des fonctions sur les images ou sur les graphes. Il est assez rare qu'on manipule des listes, des fonctions, des tableaux... Rien ne vous empêche de le faire cependant.

2. Vérification des arguments. Souvent en conjonction avec `xlgetarg()`, on voudra vérifier à la main le type des arguments passé au *C*. Voici une liste de macros qui permettent de faire ce genre de choses.

- `fixp(LVAL node)`: vérifie si l'argument est entier;
- `floatp(LVAL node)`: vérifie si l'argument est un flottant;
- `image2dp(LVAL node)` ou `image3dp(LVAL node)`: vérifient si l'argument est une image;
- `stringp(LVAL node)`: vérifie si l'argument est une chaîne de caractères;
- `graphp(LVAL node)`: vérifie si l'argument est une image.

Nous devons aussi nous assurer que le nombre d'arguments est correct, et générer une erreur le cas échéant. Voici une courte liste de fonctions et macros utiles:

- `moreargs()`: retourne `TRUE` s'il reste des arguments, `FALSE` sinon;
- `xltoomany()`: renvoie le message "il y a trop d'argument sur la ligne de commande";
- `xlerror(char *message, LVAL node)`: permet d'afficher un message d'erreur à propos du nœud `node`.

Cette partie n'est pas nécessaire si la définition de `GDIL` au niveau Lisp est réalisée avec `defun` parce que cette fonction intègre directement une vérification du nombre d'arguments avant l'appel de la fonction colle.

3. Fonctions et macros de conversion de type. Ces macros et fonctions permettent de créer un nœud Lisp à partir d'un type *C* et réciproquement. En plus des macros standard, nous avons ajouté des fonctions pour manipuler les images et les graphes. Voici les fonctions propres aux graphes:

- `LVAL cvgraph(HGRAPH *graph)` retourne un nœud Lisp nouvellement créé à partir d'un graph *C*;
- `HGRAPH getgraph(LVAL nœud)` retourne un pointeur sur un graphe (ATTENTION !) à partir d'un nœud.

Exemple: colle de la fonction de dilatation `xgdil`. Nous voulons programmer la fonction de dilatation sur des graphes. Soit la fonction Lisp `(gdil gin gout size)` où `gin` et `gout` sont deux graphes pas nécessairement identiques, et `size` une taille, donc un entier. Voici alors le code de la fonction colle:

```
/* dilation */
LVAL xgdil()
{
LVAL a, b;
int iter;

a = xlgagraph();
b = xlgagraph();
iter = getfixnum(xlgafixnum());

if (!moreargs()) {
gdil(getgraph(a), getgraph(b), iter);
return b;
} else
xltoomany();

return NIL;
}
```

Manipuler les graphes au niveau du C

Pour reprendre notre exemple, dans le cas de la dilatation, on cherche à parcourir les sommets les uns après les autres. On veut pouvoir accéder éventuellement aux arêtes. Dans le cas où on veut parcourir des arbres ou faire des recherches en largeur d'abord, on peut vouloir gérer une pile FIFO. Nous allons présenter des méthodes pour faire tout cela. Il est bien clair que cette partie ne présente pas la méthode canonique de manipulation de graphes.

Parcourir la liste des sommets du graphe. Cette partie est vraiment simple, puisque les sommets sont simplement mémorisés dans un tableau. Voici quelques lignes qui permettent de rechercher l'élément maximum dans le graphe:

```
    int i, mv;

    mv = gin->values[1];
    for (i = 2 ; i < gin->order ; i++) {
        if (mv > gin->values[i]) mv = gin->values[i];
    }
    return mv;
```

Parcourir les arêtes. Voilà qui est un peu plus complexe. Comme il est indiqué dans [6], on peut agir de la façon suivante:

```
order = gin->order;
for (k=0 ; k<iter ; k++) {
  gcopy(&gwork, gout);
  for (i=1 ; i<order ; i++) { /* forget the last one for now */
      inf = gout->values[i];
      for (j=gout->vertices[i].neigh ; j<gout->vertices[i+1].neigh ;j++) {
      if (gout->values[gout->neighs[j]] < inf)
          inf=gout->values[gout->neighs[j]];
      }
      gwork.values[i]=inf;
  }
}
```

En clair les voisins d'un sommet `i` sont rangés dans `gout->neighs` de `j = gout ->vertices[i].neigh` à `j < gout->vertices[i+1].neigh`. Les numéros de ces voisins sont donc dans `gout->neighs[j]` pour toutes les valeurs de `j` décrites. Les valeurs de ces arêtes sont elles rangées dans `gout->edval[j]`.

On note que cette convention permet de décrire des graphes orientés et de permettre des valuations différentes selon le sens de parcours d'une arête.

Les files d'attentes. Les files d'attentes pour l'instant implémentées sont des files FIFO non-hiérarchiques. Adapter les FAH pour les images au cas des graphes est assez trivial. La liste des fonctions disponibles sur les files d'attentes est la suivante:

- `gFifo_init(GFIFO *theFifo, int n);` initialise la taille de la file d'attente à `n` éléments;
- `gFifo_free(GFIFO *theFifo);` libère la file d'attente;
- `gFifo_empty(GFIFO *theFifo);` teste si la file d'attente est vide ou non;
- `GFIFO_ELT gFifo_first(GFIFO *theFifo);` retourne le premier élément de la file d'attente. `NULL` est retourné si la pile est vide;
- `int gFifo_add(GFIFO *theFifo, GFIFO_ELT elt);` ajoute un élément dans la file d'attente.

Ajouter et enlever des arêtes au graphe. Pour cette partie, la structure de graphe choisie n'est pas très souple. Il vaut mieux revenir à une structure de graphe organisée à l'aide de listes chaînées. La structure suivante est proposée:

```
typedef struct bvertex {
        VALUE value;
        VALUE passval;
        struct bvertex *next;
    } BVERTEX;
```

De façon très simple, il suffit d'organiser les sommets du graphe en un tableau de `BVERTEX` de taille adéquate, et de rajouter à chaque sommet un pointeur vers une liste chaînée. Pour plus de détail, voir la fonction `addnewedge()` dans "hgraph.c".

3.2.3 Déboggage

Utilisation de l'instruction "gdb"

La commande `gdb`, définie sur la plupart des machines où tourne UNIX, permet de débogger les programmes. La manière habituelle de procéder consiste à lancer `gdb` suivi du nom de l'exécutable. Par exemple, `gdb a.out` lance le programme de déboggage pour `a.out`.

Pour débogger Xlim3D, il faut commencer par lancer `gdb xlisp` mais cela ne suffit pas à cause du chargement dynamique des fonctions (`load "package/..."`). En fait, les adresses des fonctions ne sont pas toutes connues de `gdb` au moment du chargement. Par conséquent, si le code pourra être récupéré lors du chargement des librairies, `gdb` sera dans l'impossibilité de situer les fonctions qu'elles contiennent et donc d'examiner leur contenu. Pour déjouer cette situation, l'utilisateur doit communiquer l'adresse des fichiers `*.o` à `gdb` après le chargement dynamique des fonctions. Voici une exemple de session avec `gdb` qui met en lumière la procédure à suivre et l'ordre des instructions:

```
/user/me%gdb xlisp
GDB is free software and you are welcome to distribute copies of it
under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
GDB 4.2, Copyright 1991 Free Software Foundation, Inc...
(gdb) r
(gdb) r
Starting program: /user/me/bin/xlisp
XLisp version 2.1, Copyright (c) 1989, by David Betz
XLIM3D version 1.0 by CMM. 1992.
...
; loading "package/2d.lsp"
; linking /user/me/Xlim3d/lib/xlimglue.o  0x29d3e0
; linking /user/me/Xlim3d/lib/libkernel.a  0x0
; linking /lib/libc.a  0x0
...
; loading "package/3d.lsp"
; linking /user/me/Xlim3d/lib/3dglue.o  0x1e6e78
; linking /user/me/Xlim3d/lib/lib3d.a  0x0
; linking /lib/libc.a  0x0
...
>
Taper Crtl-C
Program received signal 2, Interrupt
0x8f8f4 in read ()
(gdb) add /user/me/Xlim3d/lib/xlimglue.o  0x29d3e0
add symbol table from file "/user/me/Xlim3d/lib/xlimglue.o" at
text_addr = 0x29d3e0?
(y or n) y
(gdb) add /user/me/Xlim3d/lib/3dglue.o  0x1e6e78
add symbol table from file "/user/me/Xlim3d/lib/3dglue.o" at
text_addr = 0x1e6e78?
```

```
(y or n) y
(gdb) continue
Continuing.
```

A partir de ce moment, les fonctions sont connues à l'intérieur de `gdb` et le déboggage peut se poursuivre normalement.

Ce n'est pas toujours commode de débogger dans `gdb` surtout quand on désire voir défiler les lignes de code. `emacs` offre une solution agréable à ce problème mais son emploi est loin d'être très simple. Voici comment démarrer dans `emacs`:

```
/user/me%emacs
Puis, il faut taper dans l'ordre Esc-x, gdb, Xlim3d (ou xlisp).
```

Ceci crée une session normale avec `gdb` à l'intérieur de `emacs`. Imaginons que l'on désire examiner le comportement de la routine `raster_read`. La première étape consiste à spécifier à `gdb` où se trouvent les sources de la fonction en tapant `directory /user/me/Xlim3d/src/imlib/jfr`. Suivant l'installation de `emacs`, cette étape est nécessaire ou non. Pour simplifier la tâche, on peut aussi mettre toutes ces commandes qui spécifient où trouver les sources dans un fichier ".gdbinit" placé dans le répertoire courant. Puis, il suffit de taper `break raster_read` et de poursuivre l'exécution du programme. Ceci fera apparaître une seconde fenêtre avec le contenu du fichier "imio.c" dès le moment où le programme entre dans la fonction `raster_read`. La commande `exls` fournie dans le répertoire "src/xlispstat2.1R2/emacs" facilite quelque peu le travail à condition de parvenir à l'adapter à son environnement `emacs`, ce qui n'est pas si simple.

Bibliography

[1] Gratin Christophe 'XLIM3D = Xlispstat + imlib + Traitement d'images 3D', Document interne au Centre de morphologie mathématique, 92 pages, Avril 1993.

[2] Rivest Jean-François: 'Transformer XLIM en un environnement d'analyse d'image', Document interne au Centre de morphologie mathématique, 1991.

[3] Rivest Jean-François: 'Utiliser XLISP', Document interne au Centre de morphologie mathématique, 31 pages, Avril 1992.

[4] Tierney Luke: 'LISP-STAT; An Object-Oriented Environment for Statistical Computing and Dynamic Graphics', John Wiley & Sons, New York, 1990.

[5] Vincent Luc: 'Graphs and mathematical morphology', *Signal Processing*, Vol. 16, Special issue on mathematical morphology, pp. 365–388, Avril 1989.

[6] Vincent Luc: 'Algorithmes morphologiques à base de files d'attentes et de lacets, extention aux graphes', Thèse de doctorat, Ecole des Mines de Paris, 1990.

Part II

Xlim3D: manuel de référence

Chapter 4

Les images

4.1 La structure d'image

`ImIs2D` *imin* Fonction

imin	Image

Description (3D): Retourne T si l'image est bidimensionnelle, () sinon.

Package: `3d`

`IType` *imin* Fonction

imin	Image

Description (2D): Cette fonction imprime le type des données contenues dans l'image passée en argument. Elle permet surtout de distinguer les images contenant des entiers ou des réels.

Exemple: `(itype in)`

Auteur: Marc Van Droogenbroeck
Package: `real`

`ImType` *imin* Fonction

imin	Image

Description (3D): Retourne la chaîne de caractères correspondant au type de l'image *imin* ("unsigned char", "unsigned short", etc.).

Package: `3d`

4.2 La grille d'image

`image-grid` *imin* Fonction

imin	Image source

Description (2D): Retourne la trame de l'image source. Soit `"s"` ou `"h"` pour une trame carrée ou hexagonale.

Exemple: `(image-grid imin)`

Package: `2d`

`image-new-grid` *imin grid* Fonction

imin	Image source
grid	Chaîne de caractères: grille

Description (2D): Change la trame de l'image source. Soit `"s"` ou `"h"` pour une trame carrée ou hexagonale. Il n'y a pas de conversion proprement dite, mais seulement un changement de type d'images.

Exemple: `(image-new-grid imin "s")`

Package: `2d`

`image-parity` *imin* Fonction

imin	Image source

Description (2D): Retourne la parité de la première ligne de l'image source. Soit 1 pour une ligne impaire ou 0 pour une ligne impaire. Ce champ de la structure image est important pour la trame hexagonale. En effet, il y a une convention qui est variable: on déplace vers la droite de 1/2 pixel les rangées impaires.

Exemple: `(image-parity imin)`

Package: `2d`

`image-new-parity` *imin parity* Fonction

imin	Image source
parity	Entier: la parité de la première ligne de l'image

Description (2D): Change la convention de parité de la première ligne d'une image. En trame hexagonale, les lignes paires ou impaires sont déplacées de 1/2 pixel vers la droite. Cette fonction dit si la ligne 0 de l'image doit être déplacée ou non. Les valeurs sont 1 ou 0, avec respectivement la ligne impaire (1) ou paire (0) .

Exemple: `(image-new-parity imin 1)`

Package: `2d`

4.3 Entrée et sortie

4.3.1 Lecture d'images

`imread` *fileName* Fonction

fileName	Chaîne de caractères: nom de l'image sur disque

Description (2D): Lit une image de format Morpho Pericolor, TIFF, SUN Rasterfile ou Visilog, indifféremment. Le type de l'image est déterminé par l'extension du nom de l'image (la dernière série de caractères après le point). L'extension pour une image visilog est ".vz", pour une image TIFF ".tif" ou ".tiff", pour une image Sun Rasterfile ".ras" et pour une image Morpho Pericolor ".imp". Retourne l'image lue.

Exemple: `(setf ima (imread "test.tiff")) ; lit l'image "test.tiff".`

Auteur: Hugues Talbot

Package: `talbot`

`tfread` *arg* Fonction

arg	Nom du fichier (chaîne de caractères)

Description (2D): `tfread` lit les images au format TIFF. Sont supportés les images 8 bits, un octet par pixel, Noir et Blanc (niveau de gris en fait). Les images palettes ne voient pas leur LUT décodée. Ces limitations se comprennent dans la mesure où Xlim3D n'a pour l'instant aucun moyen de traiter des images couleurs si ce n'est en séparant les canaux à la main. La fonction retourne une poignée sur l'image lue.

Exemple: `(setf input (tfread "test.tiff"))`

Auteur: Hugues Talbot

Package: `talbot`

`mpread` *arg* Fonction

arg	Nom du fichier (chaîne de caractères)

Description (2D): `mpread` lit les images au format Morpho Pericolor. La fonction retourne une poignée sur l'image lue.

Exemple: `(setf input (mpread "test.imp"))`

Auteur: Hugues Talbot

Package: `talbot`

`raster-read` *arg* Fonction

arg	Nom du fichier (chaîne de caractères)

Description (2D): `raster-read` lit des images de format raster de SUN. Sont supportés 8 bits, arithmétique entière.

Exemple: `(setf input (raster-read "image.ras"))`

Package: `2d`

`ImReadRaster` *imout fileName* Fonction

imout	Image résultat
fileName	Chaîne de caractères

Description (3D): Chargement dans l'image *imout* du fichier image *fileName*. Le fichier image doit être au format raster. Ce format est uniquement accepté pour les images 2D.

Exemple: `(setq n1 (ImGet2D 256 256)) (ImReadRaster n1 "ima.ras")`

Package: `3d`

`visiread` *arg* Fonction

arg	Nom du fichier (chaîne de caractères)

Description (2D): `visiread` lit des images de format VISILOG. Sont supportés 8 et 16 bits, arithmétique entière.

Exemple: `(setf input (visiread "image.ima"))`

Package: `2d`

`ImReadVisilog` *imout fileName* Fonction

imout	Image résultat
fileName	Chaîne de caractères

Description (3D): Chargement dans l'image *imout* du fichier image *fileName*. Le fichier image doit être au format visilog. C'est le seul format d'entrée accepté pour les images 3D.

Exemple: `(setq n1 (ImGet 256 256 10)) (ImReadVisilog n1 "ima.img")`

Package: `3d`

`read-all` *name skip x y byte grid order* Fonction

name	Nom du fichier (chaîne de caractères)
skip	Taille de l'en-tête à ne pas lire (nombre d'octets)
x	Dimension en '`x`' de l'image
y	Dimension en '`y`' de l'image
byte	Nombre d'octets par pixel
grid	Type de grille: `"s"` pour trame carrée et `"h"` pour hexagonale
order	Ordre des octets: `"s"` poids fort à gauche et `"b"`
	poids fort à droite (compatible IBM PC)

Description (2D): Cette fonction lit des images de tous formats, à condition que l'on connaisse des caractéristiques fondamentales sur ces images et que les pixels soient stockés les uns à la suite des autres. Retourne une image toute allouée. L'exemple ci-dessous lit une image de format VISILOG (le header est de 76 bytes), à 16 bits par pixel, trame carrée, et les données sont de type non-IBM.

Exemple: `(setf input (read-all "image.ima" 76 128 128 2 "s" "s"))`

Package: `2d`

`ImReadAnyBitMap2D` *imout fileName fileXSize fileYSize headerSize &optional winXStart winYStart winXSizewinYSize* Fonction

imout	Image
fileName	Chaîne de caractères
fileXSize	Entier: taille image en X
fileYSize	Entier: taille image en Y
headerSize	Entier: taille du header (en octets)
winXStart	Entier: coordonnée X du point de départ de la fenêtre
winYStart	Entier: coordonnée Y du point de départ de la fenêtre
winXSize	Entier: taille en X de la fenêtre
winYSize	Entier: taille en Y de la fenêtre

Description (3D): Chargement d'une image 2D d'un format quelconque. Le fichier image *fileName* est supposé contenir un en-tête suivi des pixels. Il est possible de (et il faut) spécifier quelle partie de l'image on désire charger.

Exemple: (setq im (ImGet2D 256 256))
(ImReadAnyBitMap2D im "ima.img" 512 512 120 20 20 256 256)

Auteur: Beatriz Marcotegui
Package: `3d`

`ImReadAnyBitMap` *imout fileName fileXSize fileYSize fileZSize headerSize* Fonction
&optional winXStart winYStart winZStart winXSize winYSize winZSize

imout	Image résultat
fileName	Chaîne de caractères
fileXSize	Entier: taille image en X
fileYSize	Entier: taille image en Y
fileZSize	Entier: taille image en Z
headerSize	Entier: taille du header (en octets)
winXStart	Entier: coordonnée X du point de départ de la fenêtre
winYStart	Entier: coordonnée Y du point de départ de la fenêtre
winZStart	Entier: coordonnée Z du point de départ de la fenêtre
winXSize	Entier: taille en X de la fenêtre
winYSize	Entier: taille en Y de la fenêtre
winZSize	Entier: taille en Z de la fenêtre

Description (3D): Chargement d'une image de format différent de visilog. Le fichier image *fileName* est supposé contenir un en-tête suivi des pixels. Il est possible de (et il faut) spécifier quelle partie de l'image on désire charger.

Exemple: (setq n1 (ImGet 256 256 10))
(ImReadAnyBitMap n1 "ima.img" 256 256 120 16 40 0 0 200 256 80)

Package: `3d`

`readhand` *name* Fonction

name	Chemin indiquant où se trouve le fichier à lire.

Description (2D): La routine `readhand` sert à lire des petites images entrées dans un fichier à la main; elle convient surtout durant la phase de mise au point des algorithmes. Le fichier contiendra la taille horizontale et la taille verticale respectivement sur les deux premières lignes. Puis viennent les données rangées comme dans une matrice, c.-à-d. avec un blanc entre les entiers.

Exemple: `(readhand "/images/element/b2")`

Auteur: Marc Van Droogenbroeck

Package: `vandroog`

`ImReadAscii2D` *imout fileName* Fonction

imout	Image résultat
fileName	Chaîne de caractères

Description (3D): Chargement d'une image enregistrée sous la forme d'un fichier ascii (fichier texte) dans lequel les valeurs des pixels sont codées par des chaînes de caractères séparées par des blancs. Le fichier doit commencer par les tailles de l'image en X et Y (en realité X et Z).

Exemple: Un exemple de fichier
6 6
2 2 2 2 2 2
3 3 3 3 3 3
4 4 4 4 4 4
5 5 5 5 5 5
6 6 6 6 6 6

Auteur: Beatriz Marcotegui
Package: `3d`

`ImReadAscii` *imout fileName* Fonction

imout	Image résultat
fileName	Chaîne de caractères

Description (3D): Chargement d'une image enregistrée sous la forme d'un fichier ascii (fichier texte) dans lequel les valeurs des pixels sont codées par des chaînes de caractères séparées par des blancs. Le fichier doit commencer par les tailles de l'image en X, Y et Z.

Exemple: Un exemple de fichier:
6 7 2
1 1 1 1 1 1 1
1 2 2 1 1 2 3
1 2 2 3 3 4 1
2 2 2 2 3 5 1
3 3 5 8 8 1 1
1 1 2 2 2 1 1
0 0 0 2 0 1 1
0 1 1 3 4 5 0
0 0 0 2 3 4 4
0 0 0 0 3 3 3
0 0 0 3 3 4 4
0 0 5 5 5 6 8

Package: `3d`

`readlum` *dirname expr2 expr3 expr4 expr5 expr6 expr7* Fonction

dirname	Répertoire où se trouvent les deux trames (lum001 et lum002)
expr2	0
expr3	720
expr4	576
expr5	1
expr6	"s"
expr7	"s"

Description (2D): Cette routine sert à lire une image composée de deux trames et stockée sous cette forme; elle est inspirée de la routine `read-all` (format numérique CCIR 601). La chaîne de caractères est le chemin indiquant le répertoire qui contient les deux trames (fichiers lum001 et lum002).

Exemple: `(setf im (readlum "/images/claire" 0 720 576 1 "s" "s"))`

Auteur: Marc Van Droogenbroeck

Package: `vandroog`

`ImReadText` *imout fileName* Fonction

imout	Image résultat
fileName	Chaîne de caractères

Description (3D): Chargement dans l'image *imout* du fichier image *fileName*. Le fichier image doit être un fichier texte décrivant une liste de fichiers constituant les différentes sections en Z d'une image 3D. Ce fichier texte doit contenir les informations suivantes :
1ère ligne : mot clef **emphyrio**
2ème ligne : taille en X
3ème ligne : taille en Y
4ème ligne : taille en Z (i.e. nombre de fichiers images 2D)
5ème ligne : nombre de bits par pixel (8, 16 ou 32)
6ème ligne : taille de l'en-tête precedent la valeur du premier pixel dans les fichiers images 2D.
7ème ligne et suivantes : noms des fichiers images 2D.

Exemple: Un exemple de fichier:
emphyrio
256
256
4
8
128
image/section.0
image/section.1
image/section.2
image/section.3

(setq n1 (ImGet 256 256 10))
(ImReadText n1 "ima.txt")

Package: `3d`

4.3.2 Ecriture d'images

`tfwrite` *im name &optional comp* Fonction

im	Image source
name	Nom du fichier de sortie
comp	Argument optionnel : entier indiquant la compression employée

Description (2D): `tfwrite` écrit sur le disque les images au format TIFF 8 bits, Noir et Blanc (niveau de gris), un octet par pixel. Le Noir est représenté par la valeur 0. La version de TIFF supportée est 5.0. Le dernier argument est optionnel, et s'il est donné, ce doit être un entier de valeur :

- 0 : pas de compression
- 1 : compression de Lempel, Ziv & Welch
- toute autre valeur ou pas de valeur indique : pas de compression.

Exemple:

```
(tfwrite ima "result.tiff" 1) ; ecrit l'image ima sur le disque avec
compression de LZW
```

Auteur: Hugues Talbot

Package: `talbot`

`ImWriteTIFF` *imin fileName* Fonction

imin	Image
fileName	Chaîne de caractères

Description (3D): Enregistrement dans le fichier *fileName* de l'image *imin* selon le format TIFF. Cette fonction n'est applicable qu'à des images 2D.

Package: `3d`

`raster-write` *im name &optional code* Fonction

im	Image source
name	Nom du fichier de sortie
code	Code pour inverser la LUT (optionnel)

Description (2D): Ecrit un fichier de format raster de SUN. Taille supportée: 8 bits. Si on met comme code d'opération `"i"` en option, la LUT sera inversée.
Cette fonction vérifie si le nombre de colonnes dans l'image est pair, à cause de problèmes inhérents aux rasters de SUN. Si ce n'est pas le cas, la fonction ajoute une colonne de zéros à droite de l'image.

Exemple: `(raster-write image "sortie.ima")`

Package: `2d`

`ImWriteRaster` *im fileName* Fonction

im	Image
fileName	Chaîne de caractères

Description (3D): Enregistrement dans le fichier *fileName* de l'image *im* selon le format raster. Cette fonction n'est applicable qu'à des images 2D.

Package: `3d`

`visiwrite` *im arg* Fonction

im	Image source
arg	Nom du fichier de sortie

Description (2D): Ecrit un fichier de format VISILOG. Taille supportée: `short` 16 bits.

Exemple: `(visiwrite image "sortie.ima")`

Package: `2d`

`visiwrite8` *im arg* Fonction

im	Image source
arg	Nom du fichier de sortie

Description (2D): Ecrit un fichier de format VISILOG. Taille supportée: `char` 8 bits.

Exemple: `(visiwrite8 image "sortie.ima")`

Package: `2d`

`ImWriteVisilog` *imin fileName* Fonction

imin	Image
fileName	Chaîne de caractères

Description (3D): Enregistrement dans le fichier *fileName* de l'image *imin* selon le format visilog.

Package: `3d`

`ImWriteAscii2D` *imin fileName* Fonction

imin	Image
fileName	Chaîne de caractères

Description (3D): Enregistrement de l'image imin sous la forme d'un fichier ascii tel que décrit dans la fonction `ImReadAscii2D`.

Auteur: Beatriz Marcotegui
Package: `3d`

`ImWriteAscii` *imin fileName* Fonction

imin	Image
fileName	Chaîne de caractères

Description (3D): Enregistrement de l'image *imin* sous la forme d'un fichier ascii, tel que décrit dans la fonction `ImReadAscii`.

Package: `3d`

`ImWriteEPS` *imin fileName* Fonction

imin	Image
fileName	Chaîne de caractères

Description (3D): Enregistrement dans le fichier *fileName* de l'image *imin* sous la forme d'un fichier PostScript. Cette fonction n'est applicable qu'à des images 2D.

Package: `3d`

`ImWriteVIFF` *imin fileName* Fonction

imin	Image
fileName	Chaîne de caractères

Description (3D): Enregistrement dans le fichier *fileName* de l'image *imin* selon le format VIFF. Cette fonction n'est applicable qu'à des images 2D.

Package: `3d`

`ImWritePGM` *imin fileName* Fonction

imin	Image
fileName	Chaîne de caractères

Description (3D): Enregistrement dans le fichier *fileName* de l'image *imin* selon le format PGM. Cette fonction n'est applicable qu'à des images 2D.

Package: `3d`

`ImWritePNM` *imin fileName* Fonction

imin	Image
fileName	Chaîne de caractères

Description (3D): Enregistrement dans le fichier *fileName* de l'image *imin* selon le format PNM. Cette fonction n'est applicable qu'à des images 2D.

Package: `3d`

`ImWriteVFF` *imin fileName* Fonction

imin	Image
fileName	Chaîne de caractères

Description (3D): Enregistrement dans le fichier *fileName* de l'image *imin* selon le format VFF (Visualisation File Format). Cette fonction permet d'importer des images 3D dans le logiciel SUNVISION.

Package: `3d`

`ImWriteText` *imin fileRadical* Fonction

imin	Image
fileRadical	Chaîne de caractères

Description (3D): Enregistrement de l'image *imin* sous la forme d'une liste de fichiers images visilog 2D et d'un fichier de description, tel que décrit pour la fonction `ImReadText`. Les noms des fichiers images sont obtenus par concaténation de la chaîne de caractères *fileRadical*, d'un point (.) et du numéro du plan (0,1,2 etc.).

Package: `3d`

`ImDump` *imin &optional (format defaultImDumpFormat)* Fonction

imin	Image
format	Chaîne de caractères

Description (3D): Affiche le contenu de l'image *imin* à l'écran sous forme ascii, selon le format décrit par la chaîne de caractères *format*. Ce format est celui qui sera passé à la fonction C *printf* pour afficher la valeur de chacun des pixels (par exemple : "%4d ", "%3x" etc.).

Exemple:

```
(ImDump n1 "%3x")    ; dump des pixels sous forme de valeurs
 hexadécimales sur 3 digits.
```

Package: `3d`

4.3.3 Visualisation d'images

Quelques routines qui permettent la visualisation d'images sur un PC compatible muni d'une carte VGA.

`pcdisp` *image posx posy* Fonction

image	Image à afficher à gauche de l'écran
posx	Coordonnée '`x`' dans l'écran du début de l'image
posy	Coordonnée '`y`' dans l'écran du début de l'image

Description (2D): Cette fonction affiche une image en mode VGA 320 x 200 du PC. Le coin en haut à gauche est placé à la position *(posx, posy)*. Si l'image est trop grande, elle est toujours lue à partir de la position (0,0) et est tronquée.

Exemple: `(pcdisp image1 coordx coordy)`

Auteur: Fernand Meyer

Package: `2d`

`greyview` *image* Fonction

image	Image à afficher

Description (2D): Visualisation d'une image à niveaux de gris avec une table de couleurs comportant 64 niveaux de gris.

Exemple: `(greyview image)`

Auteur: Fernand Meyer
Machine: PC muni d'une carte VGA

Package: `2d`

`colview` *image* Fonction

image	Image à afficher

Description (2D): Visualisation d'une image à niveaux de gris en fausses couleurs.

Exemple: `(colview image)`

Auteur: Fernand Meyer
Machine: PC muni d'une carte VGA

Package: `2d`

`gcview` *image* Fonction

image	Image à afficher

Description (2D): Visualisation d'une image à niveaux de gris avec une table de couleurs comportant 64 niveaux de gris. Les niveaux de gris 1, 2 et 3 sont réservés à des plans graphiques de couleur respectivement bleue, rouge et verte.

Exemple: `(gcview image)`

Auteur: Fernand Meyer
Machine: PC muni d'une carte VGA

Package: `2d`

`gtview` *image name* Fonction

image	Image à afficher à gauche de l'écran
name	Texte à afficher en bas à gauche de l'écran

Description (2D): Visualisation d'une image à niveaux de gris avec une table de couleurs comportant 64 niveaux de gris et affichage d'un texte en bas à gauche de l'écran.

Exemple: `(gtview image text)`

Auteur: Fernand Meyer
Machine: PC muni d'une carte VGA

Package: `2d`

`ctview` *image name* Fonction

image	Image à afficher à gauche de l'écran
name	Texte à afficher en bas à gauche de l'écran

Description (2D): Visualisation d'une image à niveaux de gris avec une table de couleurs comportant 64 niveaux de gris et affichage d'un texte en bas à gauche de l'écran.

Exemple: `(ctview image text)`

Auteur: Fernand Meyer
Machine: PC muni d'une carte VGA

Package: `2d`

`dgview` *image1 image2 name* Fonction

image1	Image à afficher à gauche de l'écran
image2	Image à afficher à droite de l'écran
name	Texte à afficher en bas à gauche de l'écran

Description (2D): Visualisation de deux image à niveaux de gris côte à côte, avec une table de couleurs comportant 64 niveaux de gris. Le texte est affiché en bas à gauche de l'écran.

Exemple: `(dgview image1 image2 text)`

Auteur: Fernand Meyer
Machine: PC muni d'une carte VGA

Package: `2d`

`dcview` *image1 image2 name* Fonction

image1	Image à afficher à gauche de l'écran
image2	Image à afficher à droite de l'écran
name	Texte à afficher en bas à gauche de l'écran

Description (2D): Visualisation de deux image à niveaux de gris côte à côte, avec une table de couleurs de type fausses couleurs. Le texte est affiché en bas à gauche de l'écran.

Exemple: `(dcview image1 image2 text)`

Auteur: Fernand Meyer
Machine: PC muni d'une carte VGA

Package: `2d`

`ImSunDsp expr` Fonction
ou
`ImXv expr` Fonction

`expr`	Image ou liste d'images

Description (3D): Visualise les images 2D specifiées à l'aide du programme *xv*. Fonctionne sous X windows uniquement.

Exemple: `(ImSunDsp n1 n2 n3) ou (ImXv (list n1 n2 n3))`

Machine: SUN

Package: `3d`

`ImXV3D` *image M N &optional filename* Fonction

image	Image
M	Entier, nombre de plans par ligne
N	Entier, nombre de lignes
filename	Nom du fichier à écrire sur le disque

Description (3D): Visualise des images 3D.
Génère une image 2D, dans la quelle les plans de l'image *image* (en Z) sont disposés côte à côte selon un ordre lexicographique avec *M* plans par ligne et *N* lignes. Cette image 2D est visualisée à l'aide du programme *xv*, et liberée après. Pour visualiser une image avec *xv*, il est nécessaire d'écrire un fichier sur le disque. Le nom de ce fichier est géneré automatiquement, sauf s'il est passé comme argument dans l'appel de la fonction.

Exemple:

```
(ImXV3D image 3 2 "mon_image") ; Visualise 6 plans de l'image image
passant par un ficher qui s'appelle "mon_image.tmp"
(ImXV3D image 2 2) ; Visualise 4 plans de l'image image passant par un
fichier de nom géneré automatiquement
```

Machine: SUN

Auteur: Beatriz Marcotegui
Package: `3d`

`ImEdit` *imin* Fonction

imin	Image

Description (3D): Visualise l'image 2D *imin* à l'aide du programme *editimage*. Fonctionne sous X windows.
Machine: SUN

Package: `3d`

`ImXloadimage expr` Fonction

`expr`	Image ou liste d'images

Description (3D): Visualise les images 2D spécifiées à l'aide du programme *xloadimage*.

Exemple: `(ImXloadimage n1 n2 n3) ou (ImXloadimage (list n1 n2 n3))`

Machine: SUN

Package: `3d`

`disp` *image num* Fonction

image	Image à afficher
num	Entier donnant le numéro de la fenêtre dans laquelle l'image doit être affichée.

Description (2D): Cette fonction est spécifique NeXT. Il s'agit en fait d'une macro à écrire dans le '`init.lsp`', décrite comme suit :

```
(defun disp (im sc)
`(display ,im ,sc (symbol-name (quote ,im))))
(defun disp2 (im sc)
(display (eval im) sc (symbol-name im)))
```

Ces deux macros sont indispensables au bon fonctionnement de la frontale NeXT à Xlim3D. L'utilisation de cette fonction est assez évidente: on donne l'image à afficher et un numéro d'écran (ainsi plusieurs images différentes peuvent elles être affichées successivement dans une même fenêtre). Le nom de l'image sera affiché dans le titre de la fenêtre.

Exemple: `(disp im0 0) ; affiche l'image im0 dans l'ecran 0.`

Auteur: Hugues Talbot
Machine: NeXT

Package: `nil`

`m3d` *image* Fonction

image	Image à visualiser en perspective

Description (2D): Cette fonction retourne une poignée vers une image TOUJOURS 512x256 qui est une représentation 3D minimale de l'image d'entrée. On doit donc toujours affecter le résultat de cette fonction à un identificateur de Xlisp-Stat.

Exemple:

```
(setf hop3d (m3d im0))  ; met dans hop3d une nouvelle image,
représentation 3d de im0.
```

Package: `nil`

4.3.4 Agrandissement d'images (zoom direct et inverse)

`zoom` *iminout* Fonction

iminout	Image d'entrée/sortie

Description (2D): Fait un zoom 2X dans le sens horizontal et vertical. L'image d'entrée est zoomée et remplacée.

Exemple: `(zoom input)`

Auteur: Hugues Talbot

Package: `2d`

`unzoom` *iminout* Fonction

iminout	Image d'entrée et de sortie

Description (2D): Cette fonction fait subir à l'image d'entrée un zoom de rapport 1/2 dans les deux directions (abscisse et ordonnée). La transformation s'effectue dans l'image elle-même. L'image de sortie est obtenue en faisant la moyenne des quatre pixels adjacents dans les deux directions du zoom (et non en choisissant un pixel sur quatre). Une grande quantité d'information sur l'image est donc perdue. Il s'agit en fait de l'application successive de unzoomx et de unzoomy. Cette fonction n'est cependant pas une macro.

Exemple:

```
(unzoom im0) ; la taille de im0 est divisée par 2 dans les deux
directions après un filtrage passe-bas bidimensionnel.
```

Auteur: Hugues Talbot

Package: `talbot`

`zoom-x` *iminout* Fonction

iminout	Image d'entrée/sortie

Description (2D): Fait un zoom 2X dans le sens horizontal. L'image d'entrée est zoomée et remplacée.

Exemple: `(zoom-x input)`

Auteur: Hugues Talbot

Package: `2d`

`unzoomx` *iminout* Fonction

iminout	Image d'entrée et de sortie

Description (2D): Cette fonction fait subir à l'image d'entrée un zoom de rapport 1/2 dans la direction de l'axe des abscisses. La transformation s'effectue dans l'image elle-même. L'image de sortie est obtenue en faisant la moyenne des deux pixels adjacents dans la direction du zoom (et non en choisissant un pixel sur deux). Une grande quantité d'information sur l'image est donc perdue.

Exemple:

```
(unzoomx im0) ; la taille de im0 est divisée par 2 après un filtrage
passe-bas.
```

Auteur: Hugues Talbot

Package: `talbot`

`zoom-y` *iminout* Fonction

iminout	Image d'entrée/sortie

Description (2D): Fait un zoom 2X dans le sens vertical. L'image d'entrée est zoomée et remplacée.

Exemple: `(zoom-y input)`

Auteur: Hugues Talbot

Package: `2d`

`unzoomy` *iminout* Fonction

iminout	Image d'entrée et de sortie

Description (2D): Cette fonction fait subir à l'image d'entrée un zoom de rapport 1/2 dans la direction de l'axe des ordonnées. La transformation s'effectue dans l'image elle-même. L'image de sortie est obtenue en faisant la moyenne des deux pixels adjacents dans la direction du zoom (et non en choisissant un pixel sur deux). Une grande quantité d'information sur l'image est donc perdue.

Exemple:

```
(unzoomy im0) ; la taille de im0 est divisée par 2 après un filtrage
passe-bas.
```

Auteur: Hugues Talbot

Package: `talbot`

4.3.5 Impression d'images à l'écran

`printimage` *im* Fonction

im	Image source

Description (2D): Imprime les valeurs de chaque pixel sur la console.

Exemple: `(printimage image)`

Package: `2d`

`printbinimage` *im* Fonction

im	Image binaire à imprimer à l'écran

Description (2D): Cette routine imprime une image binaire à l'écran sans laisser d'intervalle entre les pixels. Une image à plusieurs niveaux de gris est binarisée en valeurs égales ou différentes de 0. Seule la trame carrée est autorisée.

Exemple: `(printbinimage image)`

Auteur: Marc Van Droogenbroeck

Package: `vandroog`

4.4 Conversion de format d'image

`IIntToReal` *imin imout* Fonction

imin	Image à convertir (entière)
imout	Image de sortie (réelle)

Description (2D): Cette routine copie l'image d'entiers dans une image contenant des réels.

Exemple: `(iinttoreal imin imout)`

Auteur: Marc Van Droogenbroeck
Package: `real`

`IRealToInt` *imin imout* Fonction

imin	Image à convertir (réelle)
imout	Image de sortie (entière)

Description (2D): Cette routine copie l'image de réels dans une image d'entiers.

Exemple: `(irealtoint imin imout)`

Auteur: Marc Van Droogenbroeck
Package: `real`

`ImWordToByte` *imin imout* Fonction

imin	Image
imout	Image résultat

Description (3D): Convertit l'image *unsigned short* (i.e. 16 bits) *imin* en l'image *unsigned char* (ie 8 bits) *imout*, les valeurs résultats étant rééchelonnées linéairement de sorte à être comprises entre 0 et 255.

Package: `3d`

4.5 Propriétés d'image

`smlval` Fonction

Description (2D): Retourne la valeur minimale de la machine pour les pixels d'une image.

Exemple: `(smlval)`

Package: `2d`

`bigval` Fonction

Description (2D): Retourne la valeur maximale de la machine pour les pixels d'une image.

Exemple: `(bigval)`

Package: `2d`

`minval` *imin mask* Fonction

imin	Image
mask	Optionel : image-masque

Description (2D): Retourne la valeur minimale dans une image. Si on donne une image-masque, l'opération se fera sur les pixels sous le masque donné.

Exemple: `(minval input)`

Package: `2d`

`ImMinimum` *imin* Fonction

imin	Image

Description (3D): Retourne le minimum de l'image *imin*.

Package: `3d`

`ImMaxValue` *imin* Fonction

imin	Image

Description (3D): Retourne la valeur maximale codable pour les images du type de *imin*. Si *imin* est *"unsigned char"*, ce sera 255; si elle est *"unsigned short"*, ce sera 65535...

Package: `3d`

`maxval` *imin mask* Fonction

imin	Image
mask	Optionnel: image-masque

Description (2D): Retourne la valeur maximale dans une image. Si on donne une image-masque, l'opération se fera sur les pixels sous le masque donné.

Exemple: `(maxval input)`

Package: `2d`

`ImMaximum` *imin* Fonction

imin	Image

Description (3D): Retourne le maximum de l'image *imin*.

Package: `3d`

`minval-h` *imin row* Fonction

imin	Image
row	Numéro de ligne

Description (2D): Retourne la valeur minimale d'une ligne d'une image. Les lignes partent à 0.

Exemple:

```
(minval-h input 3) ; retourne la valeur min de l'image ``input''
sur la ligne 3.
```

Package: `2d`

`maxval-h` *imin row* Fonction

imin	Image
row	Numéro de ligne

Description (2D): Retourne la valeur maximale d'une ligne d'une image. Les lignes partent à 0.

Exemple:

```
(maxval-h input 3) ; retourne la valeur max de l'image ``input''
sur la ligne 3.
```

Package: `2d`

`minval-v` *imin col* Fonction

imin	Image
col	Numéro de colonne

Description (2D): Retourne la valeur minimale d'une colonne d'une image. Les colonnes partent à 0.

Exemple:

```
(minval-v input 3) ; retourne la valeur min de l'image ``input''
sur la colonne 3.
```

Package: `2d`

`maxval-v` *imin col* Fonction

imin	Image
col	Numéro de colonne

Description (2D): Retourne la valeur maximale d'une colonne d'une image. Les lignes partent à 0.

Exemple:

```
(maxval-v input 3) ; retourne la valeur max de l'image ``input''
sur la colonne 3.
```

Package: `2d`

`image-x` *imin* Fonction

imin	Image source

Description (2D): Retourne la taille en '`x`' de l'image source.

Exemple: `(image-x imin)`

Package: `2d`

`ImXSize` *imin* Fonction

imin	Image

Description (3D): Retourne un entier égal à la taille en X de l'image *imin*. Fonctionne sous X windows.

Package: `3d`

`image-y` *imin* Fonction

imin	Image source

Description (2D): Retourne la taille en '`y`' de l'image source.

Exemple: `(image-y imin)`

Package: `2d`

`ImYSize` *imin* Fonction

imin	Image

Description (3D): Retourne un entier égal à la taille en Y de l'image *imin*.

Package: `3d`

`ImZSize` *imin* Fonction

imin	Image

Description (3D): Retourne un entier égal à la taille en Z de l'image *imin*.

Package: `3d`

`ImWXSize` *imin* Fonction

imin	Image

Description (3D): Retourne un entier égal à la taille en X de la fenêtre active de l'image *imin*.

Package: `3d`

`ImWYSize` *imin* Fonction

imin	Image

Description (3D): Retourne un entier égal à la taille en Y de la fenêtre active de l'image *imin*.

Package: `3d`

`ImWZSize` *imin* Fonction

imin	Image

Description (3D): Retourne un entier égal à la taille en Z de la fenêtre active de l'image *imin*.

Package: `3d`

`ImWXStart` *imin* Fonction

imin	Image

Description (3D): Retourne un entier égal à la coordonnée en X du premier point de la fenêtre active de l'image *imin*. (0,0,0) correspond au premier point de l'image.

Package: `3d`

`ImWYStart` *imin* Fonction

imin	Image

Description (3D): Retourne un entier égal à la coordonnée en Y du premier point de la fenêtre active de l'image *imin*. (0,0,0) correspond au premier point de l'image.

Package: `3d`

`ImWZStart` *imin* Fonction

imin	Image

Description (3D): Retourne un entier égal à la coordonnée en Z du premier point de la fenêtre active de l'image *imin*. (0,0,0) correspond au premier point de l'image.

Package: `3d`

4.5.1 Accès aux caractéristiques des images

`setborder` *iminout val* Fonction

iminout	Image source et de sortie
val	Valeur du bord

Description (2D): Met le tour d'une image à une certaine valeur donnée. Les dimensions de l'image ne changent pas. L'image d'entrée et de sortie sont les mêmes.

Exemple: `(setborder image-io 5)`

Package: `2d`

`same-putborder` *iminout val* Fonction

iminout	Image source
val	Valeur du bord

Description (2D): Met un bord à une image. La valeur du bord est donnée. Les dimensions de l'image changent. L'image d'entrée et de sortie sont les mêmes.

Exemple: `(same-putborder image-io 5)`

Package: `2d`

`rmborder` *iminout* Fonction

iminout	Image source et de sortie

Description (2D): Supprime un bord de l'image. L'image d'entrée et de sortie sont les mêmes.

Exemple: `(rmborder image-io)`

Package: `2d`

`new-putborder` *image imout val* Fonction

image	Image source
imout	Image destination
val	Valeur du bord

Description (2D): Met un bord autour d'une image a une valeur donnée. Il faut que l'image de sortie ait une taille nulle parce que le bloc de données est perdu au cours de la reallocation de mémoire.

Exemple: `(new-putborder input output 3)`

Package: `2d`

`putside-u` *image val* Fonction

image	Image source et de sortie
val	Valeur du bord

Description (2D): Met un bord en haut d'une image à une valeur donnée. L'image d'entrée et de sortie sont les mêmes.

Exemple: `(putside-u image-io 5)`

Package: `2d`

`putside-d` *image val* Fonction

image	Image source et de sortie
val	Valeur du bord

Description (2D): Met un bord en bas d'une image à une valeur donnée. L'image d'entrée et de sortie sont les mêmes.

Exemple: `(putside-d image-io 5)`

Package: `2d`

`putside-r` *image val* Fonction

image	Image source et de sortie
val	Valeur du bord

Description (2D): Met un bord à droite d'une image à une valeur donnée. L'image d'entrée et de sortie sont les mêmes.

Exemple: `(putside-r image-io 5)`

Package: `2d`

`putside-l` *image val* Fonction

image	Image source et de sortie
val	Valeur du bord

Description (2D): Met un bord à gauche d'une image à une valeur donnée. L'image d'entrée et de sortie sont les mêmes.

Exemple: `(putside-l image-io 5)`

Package: `2d`

`rmside-u` *iminout* Fonction

iminout	Image source et de sortie

Description (2D): Ote un bord en haut d'une image. L'image d'entrée et de sortie sont les mêmes.

Exemple: `(rmside-u image-io)`

Package: `2d`

`rmside-d` *iminout* Fonction

iminout	Image source et de sortie

Description (2D): Ote un bord en bas d'une image. L'image d'entrée et de sortie sont les mêmes.

Exemple: `(rmside-d image-io)`

Package: `2d`

`rmside-r` *iminout* Fonction

iminout	Image source et de sortie

Description (2D): Ote un bord à droite d'une image. L'image d'entrée et de sortie sont les mêmes.

Exemple: `(rmside-r image-io)`

Package: `2d`

`rmside-l` *iminout* Fonction

iminout	Image source et de sortie

Description (2D): Ote un bord à gauche d'une image. L'image d'entrée et de sortie sont les mêmes.

Exemple: `(rmside-l image-io)`

Package: `2d`

`setside-u` *iminout val* Fonction

iminout	Image source et de sortie
val	Valeur du bord

Description (2D): Met le bord en haut d'une image à une valeur donnée. L'image d'entrée et de sortie sont les mêmes. Les dimensions de l'image ne changent pas.

Exemple: `(setside-u image-io 5)`

Package: `2d`

`setside-d` *iminout val* Fonction

iminout	Image source et de sortie
val	Valeur du bord

Description (2D): Met le bord en bas d'une image à une valeur donnée. L'image d'entrée et de sortie sont les mêmes. Les dimensions de l'image ne changent pas.

Exemple: `(setside-d image-io 5)`

Package: `2d`

`setside-r` *iminout val* Fonction

iminout	Image source et de sortie
val	Valeur du bord

Description (2D): Met le bord à droite d'une image à une valeur donnée. L'image d'entrée et de sortie sont les mêmes. Les dimensions de l'image ne changent pas.

Exemple: `(setside-r image-io 5)`

Package: `2d`

`setside-l` *iminout val* Fonction

iminout	Image source et de sortie
val	Valeur du bord

Description (2D): Met le bord à gauche d'une image à une valeur donnée. L'image d'entrée et de sortie sont les mêmes. Les dimensions de l'image ne changent pas.

Exemple: `(setside-l image-io 5)`

Package: `2d`

4.5.2 Accès aux valeurs des images

`get-pixel` *imin x y* Fonction

imin	Image entrée
x	Entier: coordonnée '`x`' du pixel
y	Entier: coordonnée '`y`' du pixel

Description (2D): Retourne la valeur du pixel situé à la coordonnée `(x,y)`. Les coordonnées commencent à 0. En haut à gauche, c'est le point (0,0).

Exemple: `(get-pixel imin 0 5)`

Package: `2d`

`ImReadPixel` *imin x y z* Fonction

imin	Image
x	Entier, coordonnée en X
y	Entier, coordonnée en Y
z	Entier, coordonnée en Z

Description (3D): Retourne la valeur du pixel de coordonnées spécifiées.

Package: `3d`

`put-pixel` *imout x y val* Fonction

imout	Image de sortie
x	Entier: coordonnée '`x`' du pixel
y	Entier: coordonnée '`y`' du pixel
val	Valeur à mettre dans l'image. (entier)

Description (2D): Met un pixel spécifié par ses coordonnées à une valeur donnée dans une image.

Exemple: `(put-pixel input 4 5 -213)`

Package: `2d`

`ImWritePixel` *iminout x y z value* Fonction

iminout	Image résultat
x	Entier, coordonnée en X
y	Entier, coordonnée en Y
z	Entier, coordonnée en Z
value	Réel, valeur du pixel

Description (3D): Affecte au pixel de coordonnées spécifiées la valeur *value*.

Package: `3d`

`imera` *imin value* Fonction

imin	Image
value	Entier

Description (2D): Met tous les pixels de l'image à une valeur donnée.

Exemple:

```
(imera ima 200) ; Met tous les pixels de l'image ima à 200.
```

Package: `2d`

`ImSetConstant` *imout value* Fonction

imout	Image résultat
value	Réel

Description (3D): Affecte la valeur constante *value* à tous les pixels de l'image *imout*.

Exemple: `(ImSetConstant im 0.0)  ; mise à 0 de l'image im`

Package: `3d`

4.5.3 Translation, rotation et transposition

`rot+` *iminout* Fonction

iminout	Image d'entrée et de sortie.

Description (2D): Cette fonction opère une rotation de 90 degrés sur l'image d'entrée dans le sens trigonométrique positif. Les images d'entrée et de sortie sont les mêmes. Aucune information n'est perdue, l'image de sortie aura ses caractéristiques changées correctement dans le cas où les dimensions en abscisse et en ordonnée de l'image de départ ne sont pas les mêmes.

Exemple: `(rot+ im0) ; un quart de tour.`

Auteur: Hugues Talbot

Package: `talbot`

`rot-` *iminout* Fonction

iminout	Image d'entrée et de sortie.

Description (2D): Cette fonction opère une rotation de 90 degrés sur l'image d'entrée dans le sens trigonométrique négatif. Les images d'entrée et de sortie sont les mêmes. Aucune information n'est perdue, l'image de sortie aura ses caractéristiques changées correctement dans le cas où les dimensions en abscisse et en ordonnée de l'image de départ ne sont pas les mêmes.

Exemple: `(rot- im0) ; un quart de tour.`

Auteur: Hugues Talbot

Package: `talbot`

`rot180` *iminout* Fonction

iminout	Image d'entrée et de sortie.

Description (2D): Cette fonction opère une rotation de 180 degrés (donc une symétrie centrale) sur l'image de sortie et d'entrée (qui est en fait la même). Aucune information sur l'image n'est perdue. Cette fonction n'est pas une macro et n'est pas une combinaison de deux rotations de 90 degrés.

Exemple: `(rot180 im0) ; un demi-tour.`

Auteur: Hugues Talbot

Package: `talbot`

`ImTranspose` *imin imout* Fonction

imin	Image
imout	Image résultat

Description (3D): Transpose l'image *imin*, le transposé étant le symetrique par rapport au centre du parallélipipède que constitue l'image.

Package: `3d`

`ImSwapYZ` *iminout* Fonction

iminout	Image

Description (3D): Transforme l'image bidimensionnelle (X,1,Z) *iminout*, en image (X,Z,1): la taille en Z devient la taille en Y, et la taille en Y (= 1) devient la taille en Z. Réciproquement *ImSwapYZ* transforme également une image (X,Y,1) en image bidimensionnelle (X,1,Y).

Exemple:

```
(def im3d (ImGet 256 256 32)) ; im3d est une image 3D
(def im2d (ImGet2D 256 256))  ; im2d est une image 2D $-->$ (256,1,256)
(ImSwapYZ im2d)               ; im2d est une image (256,256,1)
(ImSetConstantWindow im3d 0 0 10 256 256 1)
                        ; modification de la fenêtre active de im3d
(ImIsCopy im3d im2d)    ; copie de im3d dans im2d $-->$ im2d
                          contient le plan #10 de im3d
(ImSwapYZ im2d)        ; im2d est a nouveau une image 2D (256,1,256)
```

Package: `3d`

`ImRotateAxesYZX` *imin* Fonction

imin	Image

Description (3D): Effectue une rotation des axes de l'image *imin* selon la formule suivante : le point (x, y, z) est deplacé en $(x' = y, y' = z, z' = x)$. La valeur renvoyée est la nouvelle image.

Exemple: `(setq n2 (ImRotateAxesYZX n1))`

Package: `3d`

`ImRotateAxesZXY` *imin* Fonction

imin	Image

Description (3D): Effectue une rotation des axes de l'image *imin* selon la formule suivante : le point (x, y, z) est deplacé en $(x' = z, y' = x, z' = y)$. La valeur renvoyée est la nouvelle image.

Exemple: `(setq n2 (ImRotateAxesZXY n1))`

Package: `3d`

4.6 Opérations élémentaires sur des images

4.6.1 Opérations élémentaires sur des ensembles

4.6.2 Opérations élémentaires sur des images en niveaux de gris

`invertim` *imin imout* Fonction

imin	Image entrée
imout	Image sortie

Description (2D): Inverse l'image tout en la gardant positive (255-*imin*). L'image d'entrée et l'image de sortie peuvent être les mêmes.

Exemple: `(invertim input output)`

Package: `2d`

`ImInvert` *imin imout* Fonction

imin	Image
imout	Image résultat

Description (3D): Inverse l'image *imin* dans l'image *imout*.

Package: `3d`

`ima-min` *imin1 imin2 imout* Fonction

imin1	Image entrée
imin2	Image entrée
imout	Image de sortie

Description (2D): Prend le minimum entre deux images pixel par pixel. Les images d'entrée et de sortie peuvent être les mêmes.

Exemple: `(ima-min a b out)`

Package: `2d`

`ImInf` *imin1 imin2 imout* Fonction

imin1	Image
imin2	Image
imout	Image résultat

Description (3D): Opération $imout = inf(imin1, imin2)$.

Package: `3d`

`ima-max` *imin1 imin2 imout* Fonction

imin1	Image entrée
imin2	Image entrée
imout	Image de sortie

Description (2D): Prend le maximum entre deux images pixel par pixel. Les images d'entrée et de sortie peuvent être les mêmes.

Exemple: `(ima-max a b out)`

Package: `2d`

`ImSup` *imin1 imin2 imout* Fonction

imin1	Image
imin2	Image
imout	Image résultat

Description (3D): Opération $imout = sup(imin1, imin2)$.

Package: `3d`

`ImComplete` *imin imout &key (graph defaultGraph)* Fonction

imin	Image
imout	Image résultat
graph	Graphe

Description (3D): Complétude inférieure de *imin*, résultat dans *imout*.

Package: `3d`

4.6.3 Seuillage

`thresh` *imin val1 val2 imout* Fonction

imin	Image d'entrée
val1	Valeur basse entrée
val2	Valeur haute entrée
imout	Image de sortie

Description (2D): Cette fonction met à 1 tout ce qui est dans la plage donnée comme argument. Plus simple d'utilisation que `icomp`. Evidemment, l'image d'entrée peut être la même que l'image de sortie. L'intervalle est fermé.

Exemple:

```
(thresh input 100 200 out) ; Tout ce qui est dans la plage 100--200 sera
mis à 1.
```

Package: `2d`

`ImThresh` *imin imout lowThresh hiThresh &optional (value (ImMaxValue imout))* Fonction

imin	Image
imout	Image résultat
lowThresh	Réel, seuil bas
hiThresh	Réel, seuil haut
value	Réel

Description (3D): Seuillage de l'image *imin*, résultat dans *imout*, entre les valeurs *lowThresh* et *hiThresh* (bornes comprises). Les pixels dont la valeur est comprise dans cet intervalle sont passés à la valeur *value*, les autres à 0.

Package: `3d`

`ImCutDown` *imin imout* `thresh` Fonction

imin	Image
imout	Image résultat
`thresh`	Réel, seuil

Description (3D): Tous les pixels de *imin* dont la valeur excède `thresh` recoivent cette valeur (`thresh`). Les autres pixels sont inchangés. Résultat dans *imout*.

Package: `3d`

`dthr` *imin val1 val2 imout* Fonction

imin	Image d'entrée
val1	Valeur de seuillage basse
val2	Valeur de seuillage haute
imout	Image de sortie

Description (2D): Cette fonction réalise un double seuillage par reconstruction.

Niveau: LISP

```
; double white thresh with recons
(defun dthr (imin low high imout)
  (let (imiii)
      (setf imiii (timmalloc imin))
      (thresh imin high 256 imout)
      (thresh imin low 256 imiii)
      (recons8 imout imiii)
      (imfree imiii)
  )
)
```

Les meilleurs résultats sont obtenus avec des valeurs de seuils pas très différentes.

Exemple: `(dthr imin 10 15 imout)`

Auteur: Hugues Talbot

Package: `nil`

4.6.4 Fonctions calculées en chaque pixel

`clip` *imin val1 val2 imout val3 val4* Fonction

imin	Image d'entrée
val1	Valeur basse entrée
val2	Valeur haute entrée
imout	Image de sortie
val3	Valeur basse sortie
val4	Valeur haute sortie

Description (2D): Cette fonction étire la plage dynamique de l'image d'entrée. Tout ce qui est en dehors des valeurs basse et haute données en entrées sera ramené aux valeurs basse et haute données pour la sortie. L'image d'entrée peut être la même que l'image de sortie.

Exemple:

```
(clip input 100 200 output 0 255)
```

prend l'image "input", étire linéairement tout ce qui est entre 100 et 200 de façon à ce que l'image "output" soit entre 0 et 255. Tout ce qui dépassera la fourchette 100–200 sera mis soit à 0 soit à 255.

Package: `2d`

`IRealClip` *iminout min max* Fonction

iminout	Image entrée-sortie (réelle)
min	Réel minimum
max	Réel maximum

Description (2D): Cette fonction écrête l'image de sorte à la ramener dans la dynamique complète que forme l'intervalle donné en argument.

Exemple: `(irealclip imin 12.0 255.0)`

Auteur: Marc Van Droogenbroeck

Package: `real`

`stretch` *imin val1 val2 imout val3 val4* Fonction

imin	Image d'entrée
val1	Valeur basse entrée
val2	Valeur haute entrée
imout	Image de sortie
val3	Valeur basse sortie
val4	Valeur haute sortie

Description (2D): Cette fonction est presque semblable à `clip`. La différence est que ce qui ne dépasse pas les plages données n'est pas coupé comme c'est le cas avec `clip`.

Exemple: `(stretch input 100 200 output 0 255)`
prend l'image "input", étire linéairement tout ce qui est entre 100 et 200 de façon à ce que l'image "output" soit entre 0 et 255. Tout ce qui dépassera la fourchette 100–200 sera également "étiré", ce qui fait que l'on peut dépasser les plages données en sortie. L'image d'entrée peut être la même que l'image de sortie.
Package: `2d`

4.6.5 Opérations arithmétiques

Cette section décrit les fonctions qui travaillent non pas sur des voisinages de pixels mais sur les pixels individuellement.

`arith` *imin arg code imout* Fonction

imin	Image d'entrée
arg	Image d'entrée ou constante entière
code	Chaîne: opération arithmétique
imout	Image de sortie

Description (2D): C'est la fonction qui fait des opérations arithmétiques sur les images. Il faut comme premier argument une image. Le second argument est soit une image, soit une constante.Le troisième argument est plus compliqué: on spécifie l'opération à effectuer. En voici la liste (la syntaxe est similaire à celle des opérateurs en langage C):

`"+"` addition
`"-"` soustraction
`"*"` multiplication
`"/"` division
`"/0"` division. Le numérateur est multiplié par 10^0
`"/1"` division. Le numérateur est multiplié par 10^1
`"/2"` division. Le numérateur est multiplié par 10^2
`"/3"` division. Le numérateur est multiplié par 10^3
`"/4"` division. Le numérateur est multiplié par 10^4
`"/5"` division. Le numérateur est multiplié par 10^5
`"/6"` division. Le numérateur est multiplié par 10^6
`"/7"` division. Le numérateur est multiplié par 10^7
`"?"` division inverse: (a ? b) = (b / a) (Pour les entiers)

"&" ET, bit par bit
"|" OU, bit par bit
"%" modulo
"5" modulo inverse: (a 5 b) = (b % a). (Pour les entiers)
"&&" ET logique: même comportement qu'en C
"||" OU logique: même comportement qu'en C
">>" shift des bits vers la droite
"<<" shift des bits vers la gauche
"!" NON logique: même comportement qu'en C
">0" valeur absolue
"+-" donne le signe du pixel: 0 lorsque égal à 0, 1 lorsque positif, et -1 lorsque négatif

Les images d'entrée et de sortie peuvent être les mêmes. La valeur retournée par la fonction est l'image de sortie. S'il y a une division par zéro, le résultat est `bigval`.

Exemple:

```
(arith image-a image-b "+" image-sortie)
Additionne l'image a à l'image b.  Le résultat sur l'image de sortie.
(arith image 5 "-" image)
soustraction de 5 à l'image. Même image en entrée comme en sortie.
(arith image 0 "!" image)
complémentation binaire de l'image. Même image en entrée et en sortie.
```

Package: `2d`

`ImAddConstant` *iminout value* Fonction

iminout	Image résultat
value	Réel

Description (3D): Ajoute la constante *value* à l'image *iminout*. La constante peut bien sûr être négative.

Package: `3d`

`ImAddImage` *imin1 &optional imin2 imout* Fonction

imin1	Image
imin2	Image
imout	Image résultat

Description (3D): Ajout de l'image *imin1* à l'image *imin2*, résultat dans *imout*. Si *imin2* est absent, ajoute *imin1* à *imout*.

Package: `3d`

`ImAddImageCeil` *imin1 &optional imin2 imout* Fonction

imin1	Image
imin2	Image
imout	Image résultat

Description (3D): Ajout de l'image *imin1* à l'image *imin2*, résultat dans *imout*. Le résultat est plafonné à la valeur maximale autorisée pour le type de l'image (255 pour une image 8 bits, 65536 pour une image 16 bits etc.).

Package: `3d`

`ImSubImageFloor` *imin1 &optional imin2 imout* Fonction

imin1	Image
imin2	Image
imout	Image résultat

Description (3D): Si *imin2* est présente, soustraction $imout = max(0, imin1 - imin2)$. Si *imin2* est absente, soustraction $imout = max(0, imout - imin1)$.

Package: `3d`

`ImSubImageAbs` *imin1 &optional imin2 imout* Fonction

imin1	Image
imin2	Image
imout	Image résultat

Description (3D): Valeur absolue de la différence des images *imin1* et *imin2* ($imin1 - imin2$) si *imin2* est présente, de la différence des images *imout* et *imin1* ($imout - imin1$). Résultat dans *imout*.

Package: `3d`

`ImDivConstant` *iminout value* Fonction

iminout	Image résultat
value	Réel

Description (3D): Division de l'image *iminout* par la constante *value*.

Package: `3d`

`ImDivImage` *imin1 &optional imin2 imout* Fonction

imin1	Image
imin2	Image
imout	Image résultat

Description (3D): Division de l'image *imin1* par l'image *imin2*, résultat dans *imout*. Si *imin2* est absente, division de *imout* par *imin1*.

Package: `3d`

`ImMultConstant` *iminout value* Fonction

iminout	Image résultat
value	Réel

Description (3D): Multiplication de l'image *iminout* par la constante *value*.

Package: `3d`

`ImMultImage` *imin1 &optional imin2 imout* Fonction

imin1	Image
imin2	Image
imout	Image résultat

Description (3D): Multiplication de l'image *imin1* par l'image *imin2*, résultat dans *imout*. Si *imin2* est absente, multiplication de *imout* par *imin1*.

Package: `3d`

`ImBitAndConstant` *iminout value* Fonction

iminout	Image
value	Réel

Description (3D): ET logique bit à bit de l'image *iminout* avec la constante *value*.

Package: `3d`

`ImBitAndImage` *imin1 &optional imin2 imout* Fonction

imin1	Image
imin2	Image
imout	Image résultat

Description (3D): ET logique bit à bit des images *imin1* et *imin2*, ou des images *imout* et *imin1* si *imin2* est absente.

Package: `3d`

`ImBitOrImage` *imin1 &optional imin2 imout* Fonction

imin1	Image
imin2	Image
imout	Image résultat

Description (3D): OU logique bit à bit des images *imin1* et *imin2*, ou des images *imout* et *imin1* si *imin2* est absente.

Package: `3d`

4.6.6 Comparaison d'images

`icomp` *imin arg1 code imout arg2 arg3* Fonction

imin	Image
arg1	Image ou nombre entier
code	Chaîne: opération de comparaison
imout	Image de sortie
arg2	Image résultat vrai ou valeur vraie
arg3	Image résultat faux ou valeur fausse

Description (2D): `icomp` est la fonction de comparaison entre deux images ou une image et un entier. Cette fonction a comme argument une image en entrée. Le second argument est soit une image, soit un entier. Le troisième argument est une chaîne de caractères décrivant l'opération de comparaison à effectuer. Elle renvoie l'image de sortie. La syntaxe est similaire à celle du langage C. Voici la liste des opérations disponibles:

`"=="` égal
`"!="` pas égal
`">"` plus grand
`"<"` plus petit
`">="` plus grand ou égal
`"<="` plus petit ou égal

Si le résultat de cette comparaison est vrai ou faux, l'image de sortie prendra des valeurs définies par les arguments de sortie. Ces arguments peuvent être soit des scalaires entiers, soit des images. Dans le cas d'images, le pixel ayant la même localisation que les pixels à comparer aura la valeur correspondante sur l'image de sortie. La valeur retournée par la fonction est le nombre de pixels passant le test de comparaison.
On peut prendre les mêmes images en entrée qu'en sortie.

Exemple:

```
(icomp input 6 ">" out 1 0)
; tous les pixels > 6 auront 1 comme valeur de sortie sur out.
(icomp input test ">" out 10 0)
; tous les pixels plus grands que l'image
; test auront comme valeur de sortie sur out 10.
(icomp input test ">" out vrai faux)
; tous les pixels plus grands que l'image test
; auront comme valeur de sortie sur out les pixels de l'image ``vrai''.
; Les autres auront comme valeur les pixels de l'image ``faux''.
```

Package: `2d`

`ImCompare` *imin im_ou_val cond imout ResVrai ResFaux* Fonction

imin	Image à comparer
im_ou_val	Image ou valeur à comparer
cond	Chaîne: opération de comparaison
imout	Image résultat
ResVrai	Résultat vrai. Une image ou une valeur
ResFaux	Résultat faux. Une image ou une valeur

Description (3D): Comparaison entre images et valeurs réelles. La comparaison est faite pixel par pixel entre *imin* et *im_ou_val* (qui peut être une image ou une valeur réelle). Si le résultat de la comparaison est vrai, le pixel correspondant de *imout* prend la valeur de *ResVrai*, si le résultat est faux il prend la valeur de *ResFaux*.
Voici la liste d'opérations disponibles:

`"=="` égal
`"!="` différent
`">"` plus grand
`"<"` plus petit
`">="` plus grand ou égal
`"<="` plus petit ou égal

Exemple:

```
(imcompare in  6 ">" out 1 0)
; Tous les pixels plus grands que 6 auront comme valeur dans out 1,
; les autres auront la valeur 0.
(imcompare input test ">" out 5 0)
; Tous les pixels plus grands que l'image test auront comme valeur de
; sortie 5 dans out, les autres 0.
(imcompare input test ">" out vrai faux)
; Tous les pixels plus grands que l'image test auront comme valeur
; dans out la valeur de l'image <vrai>.  Les autres auront la
; valeur de l'image <faux>.
```

Auteur: Beatriz Marcotegui

Package: `3d`

`ncomp` *imin arg code* Fonction

imin	Image entrée
arg	Image d'entrée ou entier
code	Chaîne: code d'opération

Description (2D): Fonctionne de la même façon que `icomp`, sauf qu'il n'y a pas d'image de sortie. Il n'y a que la valeur de retour qui donne le nombre de points ayant passé le test. Pour la liste des opérations, voir `icomp`: ce sont les mêmes.

Exemple: `(ncomp input 1 "==") ; Le nombre de points == 1.`

Package: `2d`

4.7 Manipulation des niveaux de gris

`ImLut` *imin imout lut* Fonction

imin	Image
imout	Image résultat
lut	Liste d'entiers

Description (3D): Passage de l'image *imin* à travers la Look Up Table *lut*. Résultat dans *imout*. *lut* est une liste d'entiers dans laquelle *(nth i lut)* est la valeur qui sera affectée à tous les pixels de valeur *i* dans *imin*.

Package: `3d`

`ImSquare` *imin imout* Fonction

imin	Image
imout	Image résultat

Description (3D): Look Up Table "met au carré" de l'image *imin*, résultat dans *imout*.

Package: `3d`

`ImSquareRoot` *imin imout* Fonction

imin	Image
imout	Image résultat

Description (3D): Look Up Table "racine carrée" de l'image *imin*, résultat dans *imout*.

Package: `3d`

`ImLogarithm` *imin imout* Fonction

imin	Image
imout	Image résultat

Description (3D): Look Up Table "logarithme" de l'image *imin*, résultat dans *imout*.

Package: `3d`

`ImBornedLogarithm` *imin imout low high maxval* Fonction

imin	Image d'entrée
imout	Image résultat
low	Borne inférieure
high	Borne supérieure
maxval	Valeur maximal de sortie

Description (3D): *imout* est une anamorphose de *imin*. Les valeurs comprises entre *low* et *high* sont transformées entre zéro et `maxval` à travers une fonction logarithmique. Les valeurs plus petites que *low* sont mises à zéro et les valeurs plus grandes que *high* sont mises a `maxval`.

Exemple:
```
(setf n_plateaux (ImLabelRegions in out :gragh cubic26Gr)) ;
Etiquetage de l'image in dans out.  n_plateaux est le nombre de régions de la
mosaïque in
```

Auteur: Beatriz Marcotegui
Package: `3d`

4.8 Gestion de la mémoire

4.8.1 Création et destruction d'images

`immalloc` *x y frame* Fonction

x	Entier: taille horizontale
y	Entier: taille verticale
frame	Chaîne de caractères. Type de trame:
	`"s"` pour carrée,
	`"h"` pour hexagonale. Attention: minuscules seulement.

Description (2D): Alloue une image. Retourne une poignée à celle-ci. Il faut déclarer la taille horizontale et verticale, ainsi que le type de trame: hexagonale ou carrée.

Exemple:

```
(setf ima (immalloc 10 34 "s"))
; Image 10 x 34 de trame carrée allouée: elle s'appelle ima.
```

Package: `2d`

`RealImmalloc` *hsize vsize trame* Fonction

hsize	Entier: taille horizontale
vsize	Entier: taille verticale
trame	Type de trame: “s” pour carrée, “h” pour hexagonale

Description (2D): Alloue une image de réels. Retourne une poignée à celle-ci. Il faut déclarer la taille horizontale et verticale, ainsi que le type de trame: hexagonale ou carrée.

Exemple: `(setf ima (realimmalloc 10 34 "s")) ; Allocation d'une image 10x34 de trame carrée`

Auteur: Marc Van Droogenbroeck

Package: `real`

`ImGet2D` *xSize ySize &optional (imType "unsigned char")* Fonction

xSize	Taille en X
ySize	Taille en Y
imType	Chaîne de caratères: type de l'image

Description (3D): Création d'une image 2D. La valeur retournée est l'image créée. Les valeurs autorisées pour *imType* sont “unsigned char”, “unsigned short” et “int”. La valeur par défaut est “unsigned char”.

Exemple: `(setq n1 (ImGet2D 256 256)) ; création d'une image 2D de taille 256x256 codée sur 8 bits par pixel`

Package: `3d`

`ImGet` *xSize ySize zSize &optional (imType "unsigned char")* Fonction

xSize	Entier: taille en X
ySize	Entier: taille en Y
zSize	Entier: taille en Z
imType	Chaîne de caractères: type de l'image

Description (3D): Création d'une image. La valeur retournée est l'image créée. Les valeurs autorisées pour *imType* sont "unsigned char", "unsigned short" et "int". La valeur par défaut est "unsigned char".

Exemple: `(setq n1 (ImGet 256 256 32 "unsigned short")) ; création d'une image de taille 256x256x32 codée sur 16 bits par pixel`

Package: `3d`

`timmalloc` *im* Fonction

im	Image source

Description (2D): Alloue de la mémoire pour une image, retourne une poignée à cette image, et initialise la structure de même manière que *im*.

Exemple: `(setf nouveau (timmalloc input))`

Package: `2d`

`Realtimmalloc` *imin* Fonction

imin	Image source

Description (2D): Alloue de la mémoire pour une image de réels, retourne une poignée à cette image, et initialise la structure de même manière que *imin*. L'image servant de modèle peut être composée d'entiers ou de réels; le résultat est le même.

Exemple: `(setf nouveau (realtimmalloc input))`

Auteur: Marc Van Droogenbroeck

Package: `real`

`ImGetSame` *imin &optional (imType (ImType model))* Fonction

imin	Image
`imType`	Chaîne de caractères: type de l'image

Description (3D): Création d'une image ayant les même caractéristiques de taille que l'image *imin*. La valeur retournée est l'image créée. Si le type n'est pas spécifié la nouvelle image est de même type que l'image servant de modèle.

Exemple:

```
(setq n1 (ImGet 256 256 32)) & création d'une image de taille
256x256x32 codée sur 8 bits par pixel
(setq n2 (ImGetSame n1) & création de n2 de mêmes caractéristiques que
n1
(setq n3 (ImGetSame n1 "unsigned short")) & création de n3 de mêmes
caractéristiques que n1, mais de profondeur 16 bits.
```

Package: `3d`

`ImMapMxN` *imin* `M N` Fonction

imin	Image
`M`	Entier, nombre de plans par ligne
`N`	Entier nombre de lignes

Description (3D): Génère, à partir d'une image 3D *imin*, une image 2D dans laquelle les plans (en Z) sont disposés côte à côte selon un ordre lexicographique avec `M` plans par ligne et `N` lignes. Retourne cette nouvelle image.

Exemple:

```
(setq n1 (ImGet 256 128 32))
(ImReadVisilog n1 "ima3D.img")
(setq n2 (ImMapMxN n1 4 8))
(imxv n2)
```

Package: `3d`

`imfree` *imin* Fonction

imin	Image

Description (2D): Libère de la mémoire prise par une image. Cette image est alors détruite.

Exemple: `(imfree imin)`

Package: `2d`

`ImFree` *expr* Fonction

expr	Image ou liste d'images

Description (3D): Libération (i.e. destruction) des images spécifiée par *expr*.

Exemple:

```
(ImFree n1 n2 n3) & destruction des images n1, n2 et n3
(ImFree (list n1 n2 n3)) & autre forme admise
```

Package: `3d`

`imcopy` *imin imout* Fonction

imin	Image source
imout	Image destination

Description (2D): Copie les pixels de l'image source dans l'image destination. Il faut que les deux images soient déjà allouées.

Exemple: `(imcopy input output)`

Package: `2d`

`ImCopy` *imin imout* Fonction

imin	Image
imout	Image résultat

Description (3D): Recopie l'image *imin* dans l'image *imout*. Celle-ci a été créée pour faire le pont entre la partie 2D et la partie 3D. En effet, ses paramètres peuvent être aussi bien des images 2D que des images 3D. Si l'image d'entrée est une image 2D alors *imout* sera une image 3D. Par contre, si *imin* est une image 3D alors *imout* sera une image 2D.

Package: `3d`

`ImIsCopy` *imin imout* Fonction

imin	Image
imout	Image résultat

Description (3D): Recopie l'image *imin* dans l'image *imout*.

Package: `3d`

`col-copy` *imin col1 imout col2* Fonction

imin	Image d'entrée
col1	Entier: numéro de la colonne à copier
imout	Image de sortie
col2	Entier: numéro de la colonne de destination

Description (2D): Copie une colonne donnée d'une image à une autre. L'image d'entrée et de sortie peuvent être les mêmes.

Exemple: `(col-copy input 3 output 6)`

Package: `2d`

`line-copy` *imin line1 imout line2* Fonction

imin	Image d'entrée
line1	Entier: numéro de la ligne à copier
imout	Image de sortie
line2	Entier: numéro de la ligne de destination

Description (2D): Copie une ligne donnée d'une image à une autre. L'image d'entrée et de sortie peuvent être les mêmes.

Exemple: `(line-copy input 3 output 6)`

Package: `2d`

4.8.2 Gestion d'une fenêtre dans l'image

`ImSetWindow` *iminout xStart yStart zStart xSize ySize zSize* Fonction

iminout	Image
xStart	Entier: coordonnée en X du point origine de la fenêtre
yStart	Entier: coordonnée en Y du point origine de la fenêtre
zStart	Entier: coordonnée en Z du point origine de la fenêtre
xSize	Entier: taille en X de la fenêtre
ySize	Entier: taille en Y de la fenêtre
zSize	Entier: taille en Z de la fenêtre

Description (3D): Définition de la fenêtre active de l'image *iminout*.

Exemple:
```
(setq n1 (ImGet 256 256 128)) ; Création de l'image.
La fenêtre par défaut est celle définie a l'aide de
ImSetConstantGlobalWindow.
(ImSetWindow n1 32 60 10 128 128 64) ; Modification de la fenêtre active.
```

Package: `3d`

`ImSetGlobalWindow` *xStart yStart zStart xSize ySize zSize* Fonction

xStart	Entier: coordonnée en X du point origine de la fenêtre
yStart	Entier: coordonnée en Y du point origine de la fenêtre
zStart	Entier: coordonnée en Z du point origine de la fenêtre
xSize	Entier: taille en X de la fenêtre
ySize	Entier: taille en Y de la fenêtre
zSize	Entier: taille en Z de la fenêtre

Description (3D): Définition de la fenêtre active par défaut. Ce sera la fenêtre active de toutes les images allouées par la suite.

Package: `3d`

4.9 Copie–coller et masquage

`cut` *image supx supy infx infy* Fonction

image	Image source
supx	Coordonnée horizontale du rectangle en haut à gauche
supy	Coordonnée verticale du rectangle en haut à gauche
infx	Coordonnée horizontale du rectangle en bas à droite
infx	Coordonnée verticale du rectangle en bas à droite

Description (2D): Retourne une image composée du rectangle défini par les arguments. Coupe l'image d'entrée et crée l'image qui sera retournée. Le rectangle comprend également ses frontières. Par exemple, (1,1)–(1,1) aura comme taille 1. Si une des coordonnée est négative, il y aura une valeur prise par défaut: la taille horizontale ou verticale de l'image si respectivement c'est une coordonnée verticale ou horizontale qui est négative. L'origine est le coin supérieur gauche.

Exemple: `(setf nouveau (cut input 3 4 10 10))`

Package: `2d`

`cutpaste` *image supx supy infx infy imout outsupx outsupy* Fonction

image	Image source
supx	Coordonnée horizontale du rectangle en haut à gauche
supy	Coordonnée verticale du rectangle en haut à gauche
infx	Coordonnée horizontale du rectangle en bas à droite
infx	Coordonnée verticale du rectangle en bas à droite
imout	Image de sortie
outsupx	Coordonnée horizontale du rectangle de sortie en haut à gauche
outsupy	Coordonnée verticale du rectangle de sortie en haut à gauche

Description (2D): Même chose que `cut`, sauf que l'on spécifie ici l'image de sortie et la localisation du rectangle qui sera collé. Si une des coordonnée est négative, il y aura une valeur prise par défaut: la taille horizontale ou verticale de l'image si respectivement c'est une coordonnée verticale ou horizontale qui est négative.

Exemple: `(cutpaste input 3 4 10 10 output 2 2)`

Package: `2d`

4.10 Transformations linéaires

4.10.1 Filtrage par convolution

`conv3` *imin imout expr3 expr4* Fonction

imin	Image entrée
imout	Image 3 × 3 contenant le noyau de convolution
expr3	Image de sortie

Description (2D): Convolution par un noyau carré 3 x 3. Il n'y a pas de normalisation. Ca marche uniquement en trame carrée, bien évidemment. L'image d'entrée et de sortie peuvent être les mêmes. Tout ce qui est hors de l'image est considéré comme étant à zéro.

Exemple: `(conv3 input es output)`

Package: `2d`

`ImConvolve` *imin imout kernel weights)* Fonction

imin	Image
imout	Image résultat
kernel	Noyau
weights	Liste de réels

Description (3D): La valeur moyenne de *imin* est calculée sur l'ensemble des points *kernel* à l'aide des pondérations définies par *weights*. Le résultat est attribué à *imout*.

Exemple: `(ImConvolve n1 n2 diamondK '(0.1 0.1 0.6 0.1 0.1))  ; n2 = moyenne pondérée de n1`

Package: `3d`

4.10.2 Filtrage par passage dans le domaine transformé

`fftidlp` *imin imout fc* Fonction

imin	Image d'entrée
imout	Image de sortie
fc	Entier : fréquence de coupure en pixel.

Description (2D): Cette fonction opère un filtrage passe-bas idéal sur l'image d'entrée et place le résultat sur l'image de sortie. La fréquence de coupure est indiquée en pixel ; elle correspond au rayon du filtre énergie/phase laissant passer les fréquences basses. Cette fonction effectue une fft directe, le filtrage et une fft inverse. On ne peut accéder par cette fonction aux valeurs de phase et d'amplitude de la transformée de Fourier de l'image de départ.

ATTENTION : l'image d'entrée et de sortie doivent avoir les mêmes dimensions, en particulier les images doivent être carrées, et les dimensions en abscisse et en ordonnées doivent être des puissances de 2. Ceci est une limitation théorique de l'algorithme que l'on peut circonvenir en plaçant une image ne convenant pas à ces standards dans une image plus grande qui y convient et en mettant les pixels non définis à 0. Quelques messages de contrôle sont émis sur la sortie standard. On peut les ignorer. Les valeurs de l'image de sortie (qui sont des flottants) sont convertis en entiers entre 0 et 255.

L'algoritme utilisé est normalement plutôt performant grâce à une tabulation des valeurs trigonométriques et un calcul effectué intégralement en entier.

Exemple: `(fftidlp im0 im1 10) ; filtrage passe-bas.`

Auteur: Hugues Talbot

Package: `talbot`

`fftidhp` *imin imout fc* Fonction

imin	Image d'entrée
imout	Image de sortie
fc	Entier: fréquence de coupure en pixel.

Description (2D): Cette fonction opère un filtrage passe-haut idéal sur l'image d'entrée et place le résultat sur l'image de sortie. La fréquence de coupure est indiquée en pixel ; elle correspond au rayon du filtre énergie/phase laissant passer les fréquences hautes. Cette fonction effectue une fft directe, le filtrage et une fft inverse. On ne peut accéder par cette fonction aux valeurs de phase et d'amplitude de la transformée de Fourier de l'image de départ.

ATTENTION: limitation d'utilisation. Voir `fftidlp` pour le détail.

L'algoritme utilisé est normalement plutôt performant grâce à une tabulation des valeurs trigonométriques et un calcul effectué intégralement en entier.

Exemple: `(fftidhp im0 im1 10) ; filtrage passe-haut.`

Auteur: Hugues Talbot

Package: `talbot`

`ffte2lp` *imin imout fc* Fonction

imin	Image d'entrée
imout	Image de sortie
fc	Entier: fréquence de coupure en pixel.

Description (2D): Cette fonction opère un filtrage exponentiel passe-bas d'ordre 2 sur l'image d'entrée et place le résultat sur l'image de sortie. La fréquence de coupure est indiquée en pixel ; elle correspond à l'endroit où la valeur du filtre n'est plus que de 50% de ce qu'elle est à l'origine. Le filtre vaut 0 à l'infini. Cette fonction effectue une fft directe, le filtrage et une fft inverse. On ne peut accéder par cette fonction aux valeurs de phase et d'amplitude de la transformée de Fourier de l'image de départ.

ATTENTION: limitation d'utilisation. Voir `fftidlp` pour le détail.

L'algoritme utilisé est normalement plutôt performant grâce à une tabulation des valeurs trigonométriques et un calcul effectué intégralement en entier.

Exemple: `(ffte2lp im0 im1 10) ; filtrage passe-bas exponentiel.`

Auteur: Hugues Talbot

Package: `talbot`

`fftbtlp` *imin imout fc* Fonction

imin	Image d'entrée
imout	Image de sortie
fc	Entier : fréquence de coupure en pixel.

Description (2D): Cette fonction opère un filtrage Butterworth passe-bas d'ordre 1 sur l'image d'entrée et place le résultat sur l'image de sortie. La fréquence de coupure est indiquée en pixel ; elle correspond à l'endroit où la valeur du filtre n'est plus que de 50% de ce qu'elle est à l'origine. Le filtre vaut 0 à l'infini. Cette fonction effectue une fft directe, le filtrage et une fft inverse. On ne peut accéder par cette fonction aux valeurs de phase et d'amplitude de la transformée de Fourier de l'image de départ.

ATTENTION: limitation d'utilisation. Voir `fftidlp` pour le détail.

L'algoritme utilisé est normalement plutôt performant grâce à une tabulation des valeurs trigonométriques et un calcul effectué intégralement en entier.

Exemple: `(fftbtlp im0 im1 10) ; filtrage passe-bas Butterworth.`

Auteur: Hugues Talbot

Package: `talbot`

`fftbthp` *imin imout fc* Fonction

imin	Image d'entrée
imout	Image de sortie
fc	Entier : fréquence de coupure en pixel.

Description (2D): Cette fonction opère un filtrage Butterworth passe-haut d'ordre 1 sur l'image d'entrée et place le résultat sur l'image de sortie. La fréquence de coupure est indiquée en pixel ; elle correspond à l'endroit où la valeur du filtre n'est plus que de 50% de ce qu'elle est à l'infini. Le filtre vaut 0 à l'origine. Cette fonction effectue une fft directe, le filtrage et une fft inverse. On ne peut accéder par cette fonction aux valeurs de phase et d'amplitude de la transformée de Fourier de l'image de départ.

ATTENTION: limitation d'utilisation. Voir `fftidlp` pour le détail.

L'algoritme utilisé est normalement plutôt performant grâce à une tabulation des valeurs trigonométriques et un calcul effectué intégralement en entier.

Exemple: `(fftbthp im0 im1 10) ; filtrage passe-haut Butterworth.`

Auteur: Hugues Talbot

Package: `talbot`

4.10.3 Transformées spectrales

`fftspect` *imin imout* Fonction

imin	Image d'entrée
imout	Image de sortie

Description (2D): Cette fonction opère une FFT sur l'image d'entrée et place son spectre en énergie (phase/intensité) dans l'image de sortie.

ATTENTION: limitation d'utilisation. Voir `fftidlp` pour le détail.

L'algoritme utilisé est normalement plutôt performant grâce à une tabulation des valeurs trigonométriques et un calcul effectué intégralement en entier.

Exemple: `(fftspect im0 im1) ; spectre en énergie dans im1.`

Auteur: Hugues Talbot

Package: `talbot`

`fftspectl` *imin imout* Fonction

imin	Image d'entrée
imout	Image de sortie

Description (2D): Cette fonction opère une FFT sur l'image d'entrée, et place son spectre en énergie (phase/intensité) dans l'image de sortie. Afin de faire mieux ressortir les détails dans le spectre lui-même, les données de sorties sont passées dans un filtre logarithmique en `Log(1+x)`. Comme '`x`' est toujours positif, on ne crée pas d'artefact. Les zéros de '`x`' sont également conservés. Les données ne sont renormalisées qu'après passage dans le Log.

ATTENTION: limitation d'utilisation. Voir `fftidlp` pour le détail.

L'algoritme utilisé est normalement plutôt performant grâce à une tabulation des valeurs trigonométriques et un calcul effectué intégralement en entier.

Exemple: `(fftspectl im0 im1) ; spectre en énergie logarithmique dans im1.`

Auteur: Hugues Talbot

Package: `talbot`

`fftreal` *imin imout1 imout2* Fonction

imin	Image d'entrée
imout1	Image de sortie (réelle) avec la partie réelle
imout2	Image de sortie (réelle) avec la partie imaginaire

Description (2D): Cette fonction réalise la FFT (Fast Fourier Transform) d'une image (composée d'entiers) dont les tailles horizontale et verticale sont quelconques. Les images de sortie contiennent respectivement la partie réelle et la partie imaginaire de la transformée de Fourier. Il n'est pas inutile de rappeler que le spectre d'un signal purement réel (comme l'image d'entrée) est symétrique conjugué. D'autre part, l'origine du plan transformé est le coin supérieur gauche. La fonction est nettement plus rapide si les dimensions de l'image sont des puissances de 2.

Exemple:

```
(setf out1 (realtimmalloc in))
(setf out2 (realtimmalloc in))
(fftreal in out1 out2)        ; FFT directe
(ifftreal out1 out2 out)      ; FFT inverse
```

Auteur: Marc Van Droogenbroeck

Package: `fft`

`ifftreal` *imin1 imin2 imout* Fonction

imin1	Image d'entrée avec la partie réelle
imin2	Image d'entrée avec la partie imaginaire
imout	Image de sortie (entière)

Description (2D): Cette fonction réalise la IFFT (Inverse Fast Fourier Transform) de deux images de réels contenant la composante réelle et imaginaire du spectre. Il s'agit d'images dont les côtés sont des longueurs quelconques. Néanmois, la fonction est nettement plus rapide si les dimensions de l'image sont des puissances de 2.

Auteur: Marc Van Droogenbroeck

Package: `fft`

`fft` *imin1 imin2 imout1 imout2* Fonction

imin1	Image d'entrée (réelle) avec la partie réelle
imin2	Image d'entrée (réelle) avec la partie imaginaire
imout1	Image de sortie (réelle) avec la partie réelle
imout2	Image de sortie (réelle) avec la partie imaginaire

Description (2D): Cette fonction réalise la FFT (Fast Fourier Transform) d'une image de valeurs complexes dont les tailles horizontale et verticale sont quelconques. Les images de sortie contiennent respectivement la partie réelle et la partie imaginaire de la transformée de Fourier. L'origine du plan transformé est le coin supérieur gauche. La fonction est nettement plus rapide si les dimensions de l'image sont des puissances de 2. Attention, la routine transforme les valeurs des variables d'entrée *imin1* et *imin2*.

Exemple:

```
(setf in1 (realtimmalloc in))
(setf in2 (realtimmalloc in))
(Iinttoreal in in1) ; Copie in dans la matrice contenant les reels
(setf out1 (realtimmalloc in))
(setf out2 (realtimmalloc in))
(fft in1 in2 out1 out2)        ; FFT directe
(ifft out1 out2 aux1 aux2)      ; FFT inverse
```

Auteur: Marc Van Droogenbroeck

Package: `fft`

`ifft` *imin1 imin2 imout1 imout2* Fonction

imin1	Image d'entrée (réelle) avec la partie réelle
imin2	Image d'entrée (réelle) avec la partie imaginaire
imout1	Image de sortie (réelle) avec la partie réelle
imout2	Image de sortie (réelle) avec la partie imaginaire

Description (2D): Cette fonction réalise la IFFT (Inverse Fast Fourier Transform) de deux images de réels contenant la composante réelle et imaginaire du spectre (voir `fft`). Il s'agit d'images dont les côtés sont des longueurs quelconques. Néanmois, la fonction est nettement plus rapide si les dimensions de l'image sont des puissances de 2.

Auteur: Marc Van Droogenbroeck

Package: `fft`

4.10.4 Analyse en sous-bandes

`SubbandAnalysis` *imin imout1 imout2 imout3 imout4* Fonction

imin	Image d'entrée (entière ou réelle)
imout1	Image de sortie contenant un quart du contenu fréquentiel (réelle)
imout2	Image de sortie contenant un quart du contenu fréquentiel (réelle)
imout3	Image de sortie contenant un quart du contenu fréquentiel (réelle)
imout4	Image de sortie contenant un quart du contenu fréquentiel (réelle)

Description (2D): Analyse en sous-bandes de l'image d'entrée avec des filtres de Johnston. Les quatres images de sortie (obligatoirement réelles) reprennent respectivement les composantes de fréquence: PBH(Passe-Bas Horizontal)-PBV, PBH-PHV (Passe-Haut), PHH-PBV, PHH-PHV. Elles ont toutes une taille égale au quart de l'image originale; la fonction détecte une anomalie de cet ordre. Les images de sortie doivent exister préalablement à l'appel de la fonction. La valeur retournée est la moyenne. Il s'agit d'un réel que l'on a soustrait au signal pour éviter une corruption trop importante des signaux sous-bande due à la composante continue. Ces filtres ne sont pas à reconstruction parfaite.

Exemple: `(subbandanalysis imin imout1 imout2 imout3 imout4)`

Auteur: Marc Van Droogenbroeck
Package: `sub`

`SubbandSynthesis` *imin1 imin2 imin3 imin4 imout mean* Fonction

imin1	Image d'entré contenant un quart du contenu fréquentiel (réelle)
imin2	Image d'entré contenant un quart du contenu fréquentiel (réelle)
imin3	Image d'entré contenant un quart du contenu fréquentiel (réelle)
imin4	Image d'entré contenant un quart du contenu fréquentiel (réelle)
imout	Image reconstituée à partir des quatre bandes (entière ou réelle)
mean	Moyenne du signal à reconstruire (réel)

Description (2D): La fonction de synthèse travaille en parallèle avec la routine d'analyse (`SubbandAnalysis`). Les filtres de synthèse aboutissent à reconstruire presque parfaitement l'image. Les quatres images d'entrée (obligatoirement réelles) reprennent respectivement les composantes de fréquence: PBH(Passe-Bas Horizontal)-PBV, PBH-PHV (Passe-Haut), PHH-PBV, PHH-PHV. Elles ont toutes une taille égale au quart de l'image à reconstruire; la fonction détecte une anomalie de cet ordre. Les images de sortie ainsi que l'image de sortie doivent exister préalablement à l'appel de la fonction. La moyenne est le nombre réel obtenu à l'analyse. Il faut garder à l'esprit que les signaux peuvent sortir de la gamme dynamique, nécessitant parfois un écrêtage ou une mise à niveau par ajout d'une composante continue. Ces filtres ne sont pas à reconstruction parfaite.

Exemple: `(subbandsynthesis imin1 imin2 imin3 imin4 imout 141.3)`

Auteur: Marc Van Droogenbroeck
Package: `sub`

Chapter 5

Opérations morphologiques de base

5.1 Erosion et dilatation

`ImErode` *imin imout se* Fonction

imin	Image
imout	Image résultat
se	Elément structurant

Description (3D): Erosion de *imin* dans *imout* par l'élément structurant *se*.

Exemple:

```
(send cuboctaedronSE :Size 5) & Etablir la taille du cuboctaedre à 5

(ImErode n1 n2 cuboctaedronSE) & Erosion par un cuboctaèdre de taille 5
```

Package: `3d`

`ImInfKernel` *imin imout kernel* Fonction

imin	Image
imout	Image résultat
kernel	Noyau

Description (3D): La plus petite valeur de *imin* est calculée sur l'ensemble de points *kernel* et est attribuée à *imout* (c'est une érosion).

Exemple: `(ImInfKernel n1 n2 cuboctaedronK) ; n2 = erodé de n1 par un cuboctaèdre de taille 1`

Package: `3d`

`ImDilate` *imin imout se* Fonction

imin	Image
imout	Image résultat
se	Elément structurant

Description (3D): Dilatation de *imin* dans *imout* par l'élément structurant *se*.

Exemple:

```
(send cuboctaedronSE :Size 5) & Etablir la taille du cuboctaedre à 5
(ImDilate n1 n2 cuboctaedronSE) & dilatation par un cuboctaèdre de
taille 5
```

Package: `3d`

`ImSupKernel` *imin imout kernel* Fonction

imin	Image
imout	Image résultat
kernel	Noyau

Description (3D): La plus grande valeur de *imin* est calculée sur l'ensemble de points *kernel* et est attribuée à *imout* (c'est une dilatation).

Exemple: `(ImSupKernel n1 n2 cuboctaedronK) ; n2 = dilaté de n1 par un cuboctaèdre de taille 1`

Package: `3d`

`ImDilateByImageSE` *imin imout imse* Fonction

imin	Image
imout	Image résultat
imse	Image contenant l'élément structurant

Description (3D): Dilatation de l'image *imin* dans l'image *imout* par l'élément structurant défini par l'image *imse*. *imse* est une image dans laquelle les points de l'élément structurant ont la valeur 1 et les autres la valeur 0, sauf le centre de l'élément structurant qui doit être à la valeur 3 (= 1 + 2) s'il appartient à l'élément structurant, ou à la valeur 1 (= 0 + 2) s'il n'y appartient pas.
Cette fonction est une fonction de bas niveau et ne devrait normalement pas etre invoquée par l'utilisateur. La fonction de haut niveau à utiliser est la fonction `ImDilate` qui invoque `ImDilateByImageSE` quand l'élément structurant est de la classe *ImageSE*.

Package: `3d`

5.1.1 Erosion et dilatation avec des éléments structurants spécifiques

`ero4` *imin imout number* Fonction

imin	Image d'entrée
imout	Image de sortie
number	Nombre d'itérations

Description (2D): Erosion morphologique. Elément structurant: une croix pour ero4, un hexagone pour ero6 et un carré pour ero8. L'image d'entrée et de sortie peuvent être les mêmes.

Exemple: `(ero4 input output 4)`

Package: `2d`

`dil4` *imin imout number* Fonction

imin	Image d'entrée
imout	Image de sortie
number	Nombre d'itérations

Description (2D): Dilatation morphologique. Elément structurant: une croix pour clo4, un hexagone pour clo6 et un carré pour clo8. L'image d'entrée et de sortie peuvent être les mêmes.

Exemple: `(dil4 input output 4)`

Package: `2d`

`bipero` *imin imout x y* Fonction

imin	Image d'entrée
imout	Image de sortie
x	Coordonnée '`x`' du deuxième point de l'élément structurant
y	Coordonnée '`y`' du deuxième point de l'élément structurant

Description (2D): Erosion par un bi-point. On a l'origine et on donne les coordonnées du second point. L'image d'entrée et de sortie peuvent être les mêmes. La transformation n'est pas géodésique. Un point tombant hors de l'image est considéré égal à `SMLVAL`.

Exemple: `(bipero input output 1 1)`

Package: `2d`

`bipdil` *imin imout x y* Fonction

imin	Image d'entrée
imout	Image de sortie
x	Coordonnée '`x`' du deuxième point de l'élément structurant
y	Coordonnée '`y`' du deuxième point de l'élément structurant

Description (2D): Dilatation par un bi-point. On a l'origine et on donne les coordonnées du second point. Attention: l'élément structurant n'est pas transposé (ce qui peut être important pour des opérations composées comme l'ouverture). L'image d'entrée et de sortie peuvent être les mêmes. La transformation n'est pas géodésique. Un point tombant hors de l'image est considéré égal à `bigval`.

Exemple: `(bipdil input output 1 1)`

Package: `2d`

5.1.2 Erosion et dilatation dans des directions arbitraires

`sero` *imin imout dirs number* Fonction

imin	Image d'entrée
imout	Image de sortie
dirs	Entier: les directions dans lesquelles sera faite l'opération
number	Nombre d'itérations

Description (2D): Erosion sur une trame carrée. Dans ce cas-ci, on spécifie les directions dans lesquelles on veut éroder l'image. Les directions sont encodées de la manière suivante:

```
a4  a3  a2
a5  a0  a1
a6  a7  a8
```

Où les indices de '`a`' représentent les numéros de bits de l'entier *dirs*. Ca marche uniquement en trame carrée, bien évidemment. L'image d'entrée et de sortie peuvent être les mêmes.

Exemple: `(sero input output 4 7)`

Package: `2d`

`sdil` *imin imout dirs number* Fonction

imin	Image d'entrée
imout	Image de sortie
dirs	Entier: les directions dans lesquelles sera faite l'opération
number	Nombre d'itérations

Description (2D): Dilatation sur une trame carrée. Dans ce cas-ci, on spécifie les directions dans lesquelles on veut dilater l'image. Les directions sont encodées de la manière suivante:

```
a4  a3  a2
a5  a0  a1
a6  a7  a8
```

Où les indices de 'a' représentent les numéros de bits de l'entier *dirs*. Ca marche uniquement en trame carrée, bien évidemment. L'image d'entrée et de sortie peuvent être les mêmes.

Exemple: `(sdil input output 4 7)`

Package: `2d`

`hero` *imin imout dirs number* Fonction

imin	Image d'entrée
imout	Image de sortie
dirs	Entier: les directions dans lesquelles sera faite l'opération
number	Nombre d'itérations

Description (2D): Erosion sur une trame hexagonale. Dans ce cas-ci, on spécifie les directions dans lesquelles on veut éroder l'image. Les directions sont encodées de la manière suivante:

```
  a3  a2
a4  a0  a1
  a5  a6
```

Où les indices de 'a' représentent les numéros de bits de l'entier *dirs*. Ca marche uniquement en trame hexagonale, bien évidemment. L'image d'entrée et de sortie peuvent être les mêmes.

Exemple: `(hero input output 4 7)`

Package: `2d`

`hdil` *imin imout dirs number* Fonction

imin	Image d'entrée
imout	Image de sortie
dirs	Entier: les directions dans lesquelles sera faite l'opération
number	Nombre d'itérations

Description (2D): Dilatation sur une trame hexagonale. Dans ce cas-ci, on spécifie les directions dans lesquelles on veut dilater l'image. Les directions sont encodées de la manière suivante:

```
  a3  a2
a4  a0  a1
  a5  a6
```

Où les indices de 'a' représentent les numéros de bits de l'entier *dirs*. Ca marche uniquement en trame hexagonale, bien évidemment. L'image d'entrée et de sortie peuvent être les mêmes.

Exemple: `(hdil input output 4 7)`

Package: `2d`

`slinero8` *imin imout dx dy & optionnel method* Fonction

imin	Image d'entrée
imout	Image de sortie
dx	Déplacement en abscisse
dy	Déplacement en ordonnée
method	Optionnel: un entier indiquant la méthode (voir texte)

Description (2D): Cette fonction opère une érosion avec un élément structurant rectiligne centré. Une des extrémités de l'ES est donnée en abcisse et en ordonnée par *dx* et *dy*. On suppose toujours que le centre de l'élément structurant est en $(0, 0)$. Puisque l'élément structurant est symétrique, sa longueur sera donc: $\sqrt{(2dx)^2 + (2dy)^2}$ en théorie. Projetée sur l'axe des x ou celui des y, sa longueur sera cependant toujours impaire, de même que son nombre de pixels, puisque l'ES est un segment de droite en 8-connexité.
On approxime un ES rectiligne par la méthode de Bresenham. Ceci étant dit, il y a plusieurs méthodes pour arriver à faire des érosions avec ce genre d'ES:

- Méthode 1: La méthode la plus banale. L'élément structurant est balladé sur toute l'image, l'Inf est recalculé à chaque fois. Il est géométriquement symétrique. Cette méthode est la plus efficace pour les plus petites tailles d'ES (longueur totale < 7), son temps d'exécution est linéaire avec la taille en pixel de l'ES. On obtient cette méthode si on précise 1 dans l'argument *method*. C'est aussi la méthode employée par défaut.

- Méthode 2: La méthode alternative. On ne ballade plus l'élément structurant, on génère un balayage de l'image ayant la bonne orientation, et en régime permanent on fait toujours entrer un seul pixel et sortir un seul pixel de l'ES, ce qui permet d'obtenir une méthode très efficace, puisque son temps d'exécution est *constant* avec la taille de l'ES en pixel. La génération des balayages est un peu difficile à obtenir, ce qui fait

que pour les petites tailles on perd du temps par rapport à la méthode triviale. On se rattrape très vite dès que la taille est supérieure à environ 7 pixels (longueur totale).

Exemple: `(slinero8 im0 im1 3 2 2) ; érosion dans la direction (3,2), méthode 2`

Ou encore:

```
; on definit l'operation duale
(defun slindil8 (imin imout dx dy &optional (method 1))
    (invertim imin imin)
    (slinero8 imin imout dx dy method)
    (invertim imin imin)
    (invertim imout imout) ; returns imout: OK
)

; op\'eration en soi
(slindil8 im0 im1 25 0 2)  ; attention, l'ES aura une longueur de 50 pixels !
```

Voir aussi: `slinope8, sero`

Auteur: Hugues Talbot

Package: `line`

5.1.3 Erosion et dilatation adaptatives dans des directions quelconques

`a-sero` *imin imout imdirs number* Fonction

imin	Image d'entrée
imout	Image de sortie
imdirs	Image: les directions dans lesquelles sera faite l'opération
number	Nombre d'itérations

Description (2D): Erosion sur une trame carrée. Dans ce cas-ci, on spécifie les directions dans lesquelles on veut éroder l'image. Ces direction sont adaptatives: elles sont spécifiées sur une image, ce qui fait qu'en chaque pixel, on peut avoir des éléments structurants différents Les directions sont encodées de la manière suivante:

```
a4  a3  a2
a5  a0  a1
a6  a7  a8
```

Où les indices de 'a' représentent les numéros de bits de l'entier *dirs*. Ca marche uniquement en trame carrée, bien évidemment. L'image d'entrée et de sortie peuvent être les mêmes.

Exemple: `(a-sero input output dirs 7)`

Package: `2d`

`a-sdil` *imin imout imdirs number* Fonction

imin	Image d'entrée
imout	Image de sortie
imdirs	Image: les directions dans lesquelles sera faite l'opération
number	Nombre d'itérations

Description (2D): Dilatation sur une trame carrée. Dans ce cas-ci, on spécifie les directions dans lesquelles on veut dilater l'image. Ces direction sont adaptatives: elles sont spécifiées sur une image, ce qui fait qu'en chaque pixel, on peut avoir des éléments structurants différents. Les directions sont encodées de la manière suivante:

```
a4  a3  a2
a5  a0  a1
a6  a7  a8
```

Où les indices de 'a' représentent les numéros de bits de l'entier *dirs*. Ca marche uniquement en trame carrée, bien évidemment. L'image d'entrée et de sortie peuvent être les mêmes.

Exemple: `(a-sdil input output dirs 7)`

Package: `2d`

`a-hero` *imin imout imdirs number* Fonction

imin	Image d'entrée
imout	Image de sortie
imdirs	Image: les directions dans lesquelles sera faite l'opération
number	Nombre d'itérations

Description (2D): Erosion sur une trame hexagonale. Dans ce cas-ci, on spécifie les directions dans lesquelles on veut éroder l'image. Ces direction sont adaptatives: elles sont spécifiées sur une image, ce qui fait qu'en chaque pixel, on peut avoir des éléments structurants différents Les directions sont encodées de la manière suivante:

```
  a3  a2
a4  a0  a1
  a5  a6
```

Où les indices de 'a' représentent les numéros de bits de l'entier *dirs*. Ca marche uniquement en trame hexagonale, bien évidemment. L'image d'entrée et de sortie peuvent être les mêmes.

Exemple: `(a-hero input output dirs 7)`

Package: `2d`

`a-hdil` *imin imout imdirs number* Fonction

imin	Image d'entrée
imout	Image de sortie
imdirs	Image: les directions dans lesquelles sera faite l'opération
number	Nombre d'itérations

Description (2D): Dilatation sur une trame hexagonale. Dans ce cas-ci, on spécifie les directions dans lesquelles on veut dilater l'image. Ces direction sont adaptatives: elles sont spécifiées sur une image, ce qui fait qu'en chaque pixel, on peut avoir des éléments structurants différents Les directions sont encodées de la manière suivante:

```
  a3  a2
a4  a0  a1
  a5  a6
```

Où les indices de 'a' représentent les numéros de bits de l'entier *dirs*. Ca marche uniquement en trame hexagonale, bien évidemment. L'image d'entrée et de sortie peuvent être les mêmes.

Exemple: `(a-hdil input output dirs 7)`

Package: `2d`

5.2 Erosion et dilatation de rang–maximum

`gerop` *imin imb rank imout* Fonction

imin	Image d'entrée
imb	Image binaire contenant l'élément structurant plan
rank	Valeur du rang pour le calcul de l'érode
imout	Image de sortie

Description (2D): Cette fonction construit une image où on ne prend pas la plus petite valeur pour l'érodé en niveaux de gris mais la Nième valeur en partant de la plus petite. Avec un seuil nul, c'est la dilaté habituel. L'élément structurant est quelconque (il est fourni dans une image binaire).

Exemple: `(gerop imin imb seuil imout)`

Auteur: Marc Van Droogenbroeck

Package: `vandroog`

`ImRank` *imin imout kernel rank* Fonction

imin	Image
imout	Image résultat
kernel	Voisinage
rank	Entier

Description (3D): La valeur de rang *rank* de l'image *imin* est calculée sur l'ensemble de points *kernel* et est attribuée à *imout*. $rank = 0$ correspond à l'érodé.

Exemple: `(ImRank n1 n2 cuboctaedronK 3) ; n2 = pixels de rang 3 de n1 sur un cuboctaèdre de taille 1`

Package: `3d`

`gdilp` *imin imb rank imout* Fonction

imin	Image d'entrée
imb	Image binaire contenant l'élément structurant plan
rank	Valeur du rang pour le calcul du dilaté
imout	Image de sortie

Description (2D): Cette routine construit le dual de la fonction "gerop". Elle construit une image où on ne prend pas la plus grande valeur pour la dilaté en niveaux de gris mais la Nième valeur en partant de la plus grande. Avec un seuil nul, il s'agit du dilaté usuel. L'élément structurant est quelconque (il est fourni dans une image binaire).

Exemple: `(gdilp imin imb seuil imout)`

Auteur: Marc Van Droogenbroeck
Package: `vandroog`

5.3 Erosion et dilatation avec des éléments structurants en niveaux de gris

`veros` *imin imb imout number* Fonction

imin	Image d'entrée
imb	Image 3×3 contenant l'élément structurant
imout	Image de sortie
number	Nombre d'itérations

Description (2D): Erosion sur une trame carrée. Dans ce cas-ci, on spécifie l'élément structurant volumique avec lequel on veut éroder l'image. Ca marche uniquement en trame carrée, bien évidemment. L'image d'entrée et de sortie peuvent être les mêmes.

Exemple: `(veros input es output 7)`

Package: `2d`

`vdils` *imin imb imout number* Fonction

imin	Image d'entrée
imb	Image 3 × 3 contenant l'élément structurant
imout	Image de sortie
number	Nombre d'itérations

Description (2D): Dilatation sur une trame carrée. Dans ce cas-ci, on spécifie l'élément structurant volumique avec lequel on veut dilater l'image. Ca marche uniquement en trame carrée, bien évidemment. L'image d'entrée et de sortie peuvent être les mêmes.

Exemple: `(vdils input es output 7)`

Package: `2d`

`veroh` *imin imb imout number* Fonction

imin	Image d'entrée
imb	Image 3 × 3 contenant l'élément structurant
imout	Image de sortie
number	Nombre d'itérations

Description (2D): Erosion sur une trame hexagonale. Dans ce cas-ci, on spécifie l'élément structurant volumique avec lequel on veut éroder l'image. Ca marche uniquement en trame hexagonale, bien évidemment. L'image d'entrée et de sortie peuvent être les mêmes.

Exemple: `(veroh input es output 7)`

Package: `2d`

`vdilh` *imin imb imout number* Fonction

imin	Image d'entrée
imb	Image 3 × 3 contenant l'élément structurant
imout	Image de sortie
number	Nombre d'itérations

Description (2D): Dilatation sur une trame hexagonale. Dans ce cas-ci, on spécifie l'élément structurant volumique avec lequel on veut dilater l'image. Ca marche uniquement en trame hexagonale, bien évidemment. L'image d'entrée et de sortie peuvent être les mêmes.

Exemple: `(vdilh input es output 7)`

Package: `2d`

Chapter 6

Procédures élémentaires

6.1 Ouverture et fermeture

`BOpen` *imin imb imout* Fonction

imin	Image d'entrée binaire
imb	Image contenant l'élément structurant plan
imout	Image de sortie binaire

Description (2D): Cette fonction réalise l'ouverture d'une image binaire par un élément structurant de forme quelconque, fourni dans une image. Elle ne traite que les trames carrées pour l'instant.

Exemple: `(bopen imin imb imout)`

Auteur: Marc Van Droogenbroeck
Package: `vandroog`

`ImOpen` *imin imout se* Fonction

imin	Image
imout	Image résultat
se	Elément structurant

Description (3D): Ouverture de *imin* dans *imout* par l'élément structurant *se*.

Exemple:

```
(send cuboctaedronSE :Size 5) & Etablir la taille du cuboctaedre à 5
(ImOpen n1 n2 cuboctaedronSE) & Ouverture par un cuboctaèdre de
taille 5
```

Package: `3d`

`ImClose` *imin imout se* Fonction

imin	Image
imout	Image résultat
se	Elément structurant

Description (3D): Fermeture de *imin* dans *imout* par l'élément structurant *se*.

Exemple:

```
(send cuboctaedronSE :Size 5) & Etablir la taille du cuboctaedre à 5
(ImClose n1 n2 cuboctaedronSE) & Fermeture par un cuboctaèdre de
taille 5
```

Package: `3d`

6.1.1 Ouverture et fermeture avec des éléments structurants spécifiques

`ope4` *imin imout number* Fonction

imin	Image d'entrée
imout	Image de sortie
number	Nombre d'itérations

Description (2D): Ouverture morphologique. Elément structurant: une croix pour `ope4`, un hexagone pour `ope6` et un carré pour `ope8`. L'image d'entrée et de sortie peuvent être les mêmes.

Exemple: `(ope4 input output 4)`

Package: `2d`

`clo4` *imin imout number* Fonction

imin	Image d'entrée
imout	Image de sortie
number	Nombre d'itérations

Description (2D): Fermeture morphologique. Elément structurant: une croix pour `clo4`, un hexagone pour `clo6` et un carré pour `clo8`. L'image d'entrée et de sortie peuvent être les mêmes.

Exemple: `(clo4 input output 4)`

Package: `2d`

`linope` *imin imout size* Fonction

imin	Image d'entrée
imout	Image de sortie
size	Taille

Description (2D): Cette série de fonctions opère sur l'image d'entrée respectivement :

- LINOPEH : une ouverture linéaire horizontale
- LINOPEV : une ouverture linéaire verticale
- LINOPE+ : une ouverture linéaire à +45 degrés (sens trigo)
- LINOPE- : une ouverture linéaire à -45 degrés (sens trigo)

Niveau: LISP
Ces fonctions sont implémentées comme des fonctions LISP.

Exemple: `(linopeh im1 im2 40) ; ouverture linéaire horizontale de taille 40`

Auteur: Hugues Talbot

Package: `line`

`linclo` *imin imout size* Fonction

imin	Image d'entrée
imout	Image de sortie
size	Taille

Description (2D): Cette série de fonctions opère sur l'image d'entrée respectivement :

- LINCLOH : une fermeture linéaire horizontale
- LINCLOV : une fermeture linéaire verticale
- LINCLO+ : une fermeture linéaire à +45 degrés (sens trigo)
- LINCLO- : une fermeture linéaire à -45 degrés (sens trigo)

Niveau: LISP
Ces fonctions sont implémentées comme des fonctions LISP.

Exemple: `(lincloh im1 im2 40) ; fermeture linéaire horizontale de taille 40`

Auteur: Hugues Talbot

Package: `line`

6.1.2 Ouverture et fermeture dans des directions quelconques

`sope` *imin imout dirs number* Fonction

imin	Image d'entrée
imout	Image de sortie
dirs	Entier: les directions dans lesquelles sera faite l'opération
number	Nombre d'itérations

Description (2D): Ouverture sur une trame carrée. Dans ce cas-ci, on spécifie les directions dans lesquelles on veut ouvrir l'image. Les directions sont encodées de la manière suivante:

```
a4  a3  a2
a5  a0  a1
a6  a7  a8
```

Où les indices de 'a' représentent les numéros de bits de l'entier *dirs*. Ca marche uniquement en trame carrée, bien évidemment. L'image d'entrée et de sortie peuvent être les mêmes.

Exemple: `(sope input output 4 7)`

Package: `2d`

`sclo` *imin imout dirs number* Fonction

imin	Image d'entrée
imout	Image de sortie
dirs	Entier: les directions dans lesquelles sera faite l'opération
number	Nombre d'itérations

Description (2D): Fermeture sur une trame carrée. Dans ce cas-ci, on spécifie les directions dans lesquelles on veut fermer l'image. Les directions sont encodées de la manière suivante:

```
a4  a3  a2
a5  a0  a1
a6  a7  a8
```

Où les indices de 'a' représentent les numéros de bits de l'entier *dirs*. Ca marche uniquement en trame carrée, bien évidemment. L'image d'entrée et de sortie peuvent être les mêmes.

Exemple: `(sclo input output 4 7)`

Package: `2d`

`hope` *imin imout dirs number* Fonction

imin	Image d'entrée
imout	Image de sortie
dirs	Entier: les directions dans lesquelles sera faite l'opération
number	Nombre d'itérations

Description (2D): Ouverture sur une trame hexagonale. Dans ce cas-ci, on spécifie les directions dans lesquelles on veut ouvrir l'image. Les directions sont encodées de la manière suivante:

```
  a3  a2
a4  a0  a1
  a5  a6
```

Où les indices de 'a' représentent les numéros de bits de l'entier *dirs*. Ca marche uniquement en trame hexagonale, bien évidemment. L'image d'entrée et de sortie peuvent être les mêmes.

Exemple: `(hope input output 4 7)`

Package: `2d`

`hclo` *imin imout dirs number* Fonction

imin	Image d'entrée
imout	Image de sortie
dirs	Entier: les directions dans lesquelles sera faite l'opération
number	Nombre d'itérations

Description (2D): Fermeture sur une trame hexagonale. Dans ce cas-ci, on spécifie les directions dans lesquelles on veut fermer l'image. Les directions sont encodées de la manière suivante:

```
  a3  a2
a4  a0  a1
  a5  a6
```

Où les indices de 'a' représentent les numéros de bits de l'entier *dirs*. Ca marche uniquement en trame hexagonale, bien évidemment. L'image d'entrée et de sortie peuvent être les mêmes.

Exemple: `(hclo input output 4 7)`

Package: `2d`

`slinope8` *imin imout dx dy & optional method* Fonction

imin	Image d'entrée
imout	Image de sortie
dx	Déplacement en abscisse
dy	Déplacement en ordonnée
method	Optionnel: un entier indiquant la méthode (voir texte)

Description (2D): Cette fonction réalise une érosion suivie d'une dilatation avec un élément structurant rectiligne. Les méthodes de bases sont les mêmes que pour l'érosion seule, voir cette fonction pour plus de détails. ATTENTION, les numéros des méthodes sont différents: la méthode standard est la méthode 0, et la méthode alternative est maintenant la méthode 3 ! De ces nombres, on pourra deviner finement que `slinope8` et `slinero8` appellent en fait la même routine avec des arguments différents. Cependant avec `slinope8` seuls 0 et 3 sont permis, ainsi qu'avec `slinero8` seuls 1 et 2 sont autorisés.

La méthode par défaut est la méthode standard (0).

Exemple: `(slinope8 im0 im1 3 2 2) ; Ouverture rectiligne`

Ou encore:

```
; on definit l'operation duale
(defun slinclo8 (imin imout dx dy &optional (method 1))
    (invertim imin imin)
    (slinope8 imin imout dx dy method)
    (invertim imin imin)
    (invertim imout imout) ; returns imout: OK
)

; operation en soi
(slinclo8 im0 im1 25 0 2)  ; attention, l'ES aura une longueur de 50 pixels !
(slinclo8 im0 im2 0 25 2)  ; meme remarque
(icomp im1 im2 ">" im2 im2 im1) ; inf entre im1 et im2 , resultat dans im2.
```

Voir aussi: `slinero8`, `sope`, `linopev`, `linopeh`, `linope+`, `linope-`

Auteur: Hugues Talbot

Package: `line`

6.1.3 Ouverture et fermeture adaptatives dans des directions quelconques

`a-sope` *imin imout imdirs number* Fonction

imin	Image d'entrée
imout	Image de sortie
imdirs	Image: les directions dans lesquelles sera faite l'opération
number	Nombre d'itérations

Description (2D): Ouverture sur une trame carrée. Dans ce cas-ci, on spécifie les directions dans lesquelles on veut ouvrir l'image. Ces directions sont adaptatives: elles sont spécifiées sur une image, ce qui fait qu'en chaque pixel, on peut avoir des éléments structurants différents. Les directions sont encodées de la manière suivante:

```
a4  a3  a2
a5  a0  a1
a6  a7  a8
```

Où les indices de 'a' représentent les numéros de bits de l'entier *dirs*. Ca marche uniquement en trame carrée, bien évidemment. L'image d'entrée et de sortie peuvent être les mêmes.

Exemple: `(a-sope input output dirs 7)`

Package: `2d`

`a-sclo` *imin imout imdirs number* Fonction

imin	Image d'entrée
imout	Image de sortie
imdirs	Image: les directions dans lesquelles sera faite l'opération
number	Nombre d'itérations

Description (2D): Fermeture sur une trame carrée. Dans ce cas-ci, on spécifie les directions dans lesquelles on veut fermer l'image. Ces directions sont adaptatives: elles sont spécifiées sur une image, ce qui fait qu'en chaque pixel, on peut avoir des éléments structurants différents. Les directions sont encodées de la manière suivante:

```
a4  a3  a2
a5  a0  a1
a6  a7  a8
```

Où les indices de 'a' représentent les numéros de bits de l'entier *dirs*. Ca marche uniquement en trame carrée, bien évidemment. L'image d'entrée et de sortie peuvent être les mêmes.

Exemple: `(a-sclo input output dirs 7)`

Package: `2d`

`a-hope` *imin imout imdirs number* Fonction

imin	Image d'entrée
imout	Image de sortie
imdirs	Image: les directions dans lesquelles sera faite l'opération
number	Nombre d'itérations

Description (2D): Ouverture sur une trame hexagonale. Dans ce cas-ci, on spécifie les directions dans lesquelles on veut ouvrir l'image. Ces directions sont adaptatives: elles sont spécifiées sur une image, ce qui fait qu'en chaque pixel, on peut avoir des éléments structurants différents. Les directions sont encodées de la manière suivante:

```
  a3  a2
a4  a0  a1
  a5  a6
```

Où les indices de 'a' représentent les numéros de bits de l'entier *dirs*. Ca marche uniquement en trame hexagonale, bien évidemment. L'image d'entrée et de sortie peuvent être les mêmes.

Exemple: `(a-hope input output dirs 7)`

Package: `2d`

`a-hclo` *imin imout imdirs number* Fonction

imin	Image d'entrée
imout	Image de sortie
imdirs	Image: les directions dans lesquelles sera faite l'opération
number	Nombre d'itérations

Description (2D): Fermeture sur une trame hexagonale. Dans ce cas-ci, on spécifie les directions dans lesquelles on veut fermer l'image. Ces directions sont adaptatives: elles sont spécifiées sur une image, ce qui fait qu'en chaque pixel, on peut avoir des éléments structurants différents. Les directions sont encodées de la manière suivante:

```
  a3  a2
a4  a0  a1
  a5  a6
```

Où les indices de 'a' représentent les numéros de bits de l'entier *dirs*. Ca marche uniquement en trame hexagonale, bien évidemment. L'image d'entrée et de sortie peuvent être les mêmes.

Exemple: `(a-hclo input output dirs 7)`

Package: `2d`

6.2 Ouverture et fermeture de rang–maximum

`gopep` *imin imb rank1 rank2 imout* Fonction

imin	Image d'entrée
imb	Image binaire contenant l'élément structurant plan
rank1	Valeur du rang pour le calcul de l'érode
rank2	Valeur du rang pour le calcul du dilaté
imout	Image de sortie

Description (2D): Cette routine effectue la mise en cascade d'une érosion et d'une dilatation toutes deux caractérisées par un seuil. Avec deux seuils nuls, on retombe sur la définition habituelle de l'ouverture pour des images. L'élément structurant est quelconque (il est fourni dans une image binaire).

Exemple: `(gopep imin imb seuil1 seuil2 imout)`

Auteur: Marc Van Droogenbroeck

Package: `vandroog`

`gclop` *imin imb rank1 rank2 imout* Fonction

imin	Image d'entrée
imb	Image binaire contenant l'élément structurant plan
rank1	Valeur du rang pour le calcul du dilaté
rank2	Valeur du rang pour le calcul de l'érode
imout	Image de sortie

Description (2D): Cette routine effectue la mise en cascade d'une dilatation et d'une érosion toutes deux caractérisées par un seuil. Avec deux seuils nuls, on retombe sur la définition habituelle de la fermeture pour des images. L'élément structurant est quelconque (il est fourni dans une image binaire).

Exemple: `(gclop imin imb seuil1 seuil2 imout)`

Auteur: Marc Van Droogenbroeck

Package: `vandroog`

6.3 Ouverture et fermeture avec des fonctions structurantes

`vopes` *imin imb imout number* Fonction

imin	Image d'entrée
imb	Image 3 × 3 contenant l'élément structurant
imout	Image de sortie
number	Nombre d'itérations

Description (2D): Ouverture sur une trame carrée. Dans ce cas-ci, on spécifie l'élément structurant volumique avec lequel on veut ouvrir l'image. Ca marche uniquement en trame carrée, bien évidemment. L'image d'entrée et de sortie peuvent être les mêmes.

Exemple: `(vopes input es output 7)`

Package: `2d`

`vclos` *imin imb imout number* Fonction

imin	Image d'entrée
imb	Image 3 × 3 contenant l'élément structurant
imout	Image de sortie
number	Nombre d'itérations

Description (2D): Fermeture sur une trame carrée. Dans ce cas-ci, on spécifie l'élément structurant volumique avec lequel on veut fermer l'image. Ca marche uniquement en trame carrée, bien évidemment. L'image d'entrée et de sortie peuvent être les mêmes.

Exemple: `(vclos input es output 7)`

Package: `2d`

`vopeh` *imin imb imout number* Fonction

imin	Image d'entrée
imb	Image 3 × 3 contenant l'élément structurant
imout	Image de sortie
number	Nombre d'itérations

Description (2D): Ouverture sur une trame hexagonale. Dans ce cas-ci, on spécifie l'élément structurant volumique avec lequel on veut ouvrir l'image. Ca marche uniquement en trame hexagonale, bien évidemment. L'image d'entrée et de sortie peuvent être les mêmes.

Exemple: `(vopeh input es output 7)`

Auteur: Hugues Talbot

Package: `2d`

`vcloh` *imin imb imout number* Fonction

imin	Image d'entrée
imb	Image 3 × 3 contenant l'élément structurant
imout	Image de sortie
number	Nombre d'itérations

Description (2D): Fermeture sur une trame hexagonale. Dans ce cas-ci, on spécifie l'élément structurant volumique avec lequel on veut fermer l'image. Ca marche uniquement en trame hexagonale, bien évidemment. L'image d'entrée et de sortie peuvent être les mêmes.

Exemple: `(vcloh input es output 7)`

Package: `2d`

6.4 Filtrage morphologique

6.4.1 Toggle mapping (contraste)

`Toggle` *im1 im2 im3 imout* Fonction

im1	Image '`a`'
im2	Image '`b`'
im3	Image '`c`'
imout	Image de sortie '`o`'

Description (2D): Cette fonction fait un toggle mapping: si $b - a < c - b, o = a$. Si $b - a > c - b, o = c$. Enfin, si $b - a = c - b, o = b$.

Exemple: `(toggle ima imb imc imout)`

Package: `2d`

6.4.2 Filtres auto-médian

`automed4` *imin imout number* Fonction

imin	Iimage entrée
imout	Image sortie
number	Nombre d'itérations

Description (2D): Filtre automédian. Elément structurant: une croix pour `automed4`, un hexagone pour `automed6` et un carré pour `automed8`. L'image d'entrée et de sortie peuvent être les mêmes.

Exemple: `(automed4 input out 10) ; automédian en losange, taille 10.`

Package: `2d`

6.4.3 Supremum de plusieurs opérateurs

`alinero8` *imin imout size &optional method* Fonction

imin	Image d'entrée
imout	Image de sortie
size	Taille de l'élément structurant
method	Optionnel: un entier indiquant la méthode (voir texte)

Description (2D): Cette fonction réalise une érosion avec un élément structurant rectiligne dans toutes les directions représentatives de la trame pour la taille donnée. Le Sup de toutes les érosions dans toutes ces directions est donné en résultat. La méthode de base est la même que pour `slinero8`. Cette fonction décide despotiquement quelles sont les directions qu'il faut considérer, et affiche ces directions à la console.
La taille donnée correspond au *rayon* du disque dans lequel s'inscrivent tous les éléments structurants. On a à peu près $\pi \times size$ directions considérées. Les méthodes disponibles sont les mêmes que pour `slinero8` à savoir: 1 pour la méthode standard (complexité en carré de la taille) et 2 pour la méthode alternative (complexité linéaire avec la taille).
La méthode par défaut est la méthode standard (1).

Exemple: `(alinope8 im0 im1 10 2) ; sup de toutes les érosions rectilignes de taille 10`

Voir aussi: `alinope8, slinero8, sero`

Auteur: Hugues Talbot

Package: `line`

`alinope8` *imin imout size & optional method* Fonction

imin	Image d'entrée
imout	Image de sortie
size	Taille de l'élément structurant
method	Optionnel: un entier indiquant la méthode (voir texte)

Description (2D): Cette fonction réalise une ouverture avec un élément structurant rectiligne dans toutes les directions représentatives de la trame pour la taille donnée. Le Sup de toutes les ouvertures dans toutes ces directions est donné en résultat. La méthode de base est la même que pour `slinope8`. Cette fonction décide despotiquement quelles sont les directions à considérer, et affiche ces directions à la console.
La taille donnée correspond au *rayon* du disque dans lequel s'inscrivent tous les éléments structurants. On a à peu près $\pi size$ directions considérées. Les méthodes disponibles sont les mêmes que pour `slinope8` à savoir: 0 pour la méthode standard (complexité variant comme le carré de la taille) et 3 pour la méthode alternative (complexité linéaire avec la taille).
La méthode par défaut est la méthode standard (0).

Exemple:

```
>(alinope8 im0 im1 6 3)  ; sup de toutes les ouvertures lineaires de taille 6
Loop#:0, x = 6, y = 0
Loop#:1, x = 0, y = 6
Loop#:2, x = 6, y = 1
Loop#:3, x = 1, y = 6
Loop#:4, x = -1, y = 6
Loop#:5, x = -6, y = 1
Loop#:6, x = 6, y = 2
Loop#:7, x = 2, y = 6
Loop#:8, x = -2, y = 6
Loop#:9, x = -6, y = 2
Loop#:10, x = 5, y = 3
Loop#:11, x = 3, y = 5
Loop#:12, x = -3, y = 5
Loop#:13, x = -5, y = 3
Loop#:14, x = 4, y = 4
Loop#:15, x = -4, y = 4
Square grid image [256x256] #<Foo: #2c9a70>
>
```

Voir aussi: `alinero8`, `slinope8`, `sero`

Auteur: Hugues Talbot

Package: `line`

Chapter 7

Composantes connexes

7.1 Mesures fondamentales

`volume` *imin* Fonction

imin	Image

Description (2D): Retourne la valeur de la somme des pixels d'une image.

Exemple:

```
(volume input)
(setf moyenne (/ volume input) (* (image-x input) (image-y input))) ;
calcule la valeur moyenne de l'image.
```

Package: `2d`

`ImVolume` *imin* Fonction

imin	Image

Description (3D): Retourne le volume (= somme des valeurs de tous les pixels) de l'image *imin*.

Package: `3d`

`SNR` *varimin1 imin2* Fonction

imin1	Image d'entrée (entiers ou réels)
imin2	Image erreur (entiers ou réels)

Description (2D): Cette fonction retourne le rapport signal à bruit (en décibels) entre une image d'entrée et une image erreur. Les deux images peuvent être des images entières ou des images composées de réels. A toutes fins utiles, la fonction imprime aussi les écarts types des deux signaux.

Exemple: `(snr imin imout)`

Auteur: Marc Van Droogenbroeck
Package: `real`

7.2 Mesures de fréquences

7.2.1 Histogrammes

`histo` *imin imout line &optional mask* Fonction

imin	Image d'entrée
imout	Image de sortie
line	Numéro de la ligne de l'image de sortie
mask	Optionnel: image masque

Description (2D): Fait un histogramme de l'image d'entrée, et va la mettre sur une ligne de l'image de sortie. Si on ajoute une image-masque, l'histogramme ne se fera que sur les pixels sous le masque donné.

Exemple: `(histo input output 3)`

Package: `2d`

`ImHistogram` *imin* Fonction

imin	Image

Description (3D): Retourne une liste contenant l'histogramme de *imin*. Le Ième élément de la liste est le nombre de pixel de valeur I. La liste contient (N + 1) entiers, N étant le niveau de gris maximum rencontré dans l'image.

Package: `3d`

7.2.2 Manipulation d'histogramme

`range-histo` *imin val1 val2 imout line* Fonction

imin	Image d'entrée
val1	Valeur basse de l'intervalle considéré
val2	Valeur haute de l'intervalle considéré
imout	Image de sortie
line	Numéro de la ligne de l'image de sortie

Description (2D): Fait un histogramme de l'image d'entrée, et va la mettre sur une ligne de l'image de sortie. On donne en plus des valeurs de bornes afin d'exclure du calcul de l'histogramme les valeurs qui se trouveraient au-dehors de ces bornes.

Exemple: `(range-histo input 40 200 output 3)`

Package: `2d`

`rankval` *imin real* Fonction

imin	Image
real	Nombre point flottant

Description (2D): Retourne la valeur du rang de l'histogramme des pixels d'une image. Examine l'histogramme de l'image, et fait sortir la valeur correspondant au seuil de la distribution cumulée donnée en argument.

Exemple: `(rankval input 0.5) ; valeur médiane de l'image.`

Package: `2d`

`varval` *imin &optional mask* Fonction

imin	Image
mask	Optionnel :image-masque

Description (2D): Retourne la valeur de la variance dans une image. Si on donne une image-masque, l'opération se fera sur les pixels sous le masque donné.

Exemple: `(varval input)`

Package: `2d`

7.3 Mesures statistiques et distributions aléatoires

`modeval` *imin mask* Fonction

imin	Image
mask	Optionnel: image-masque

Description (2D): Retourne la valeur du mode dans une image. Si on donne une image-masque, l'opération se fera sur les pixels sous le masque donné.

Exemple: `(modeval input)`

Package: `2d`

`ImMeanKernel` *imin imout kernel* Fonction

imin	Image
imout	Image résultat
kernel	Noyau

Description (3D): La valeur moyenne de *imin* est calculée sur l'ensemble des points *kernel* et est attribuée à *imout*.

Exemple: `(ImMeanKernel n1 n2 cuboctaedronK) ; n2 = moyenne de n1 sur un cuboctaèdre de taille 1`

Package: `3d`

`recavg` *imin imout mask x y seuil cor* Fonction

imin	Image entrée
imout	Image de sortie
mask	Image masque
x	Taille x: entier
y	Taille y: entier
seuil	Seuil aire valide: flottant
cor	Facteur de correction du gain: flottant

Description (2D): C'est un filtre rectangulaire à moyenne mobile: on donne le rectangle pour la taille du filtre. Par ailleurs, cette moyenne est faite sur un masque: si le masque = 0, les points de l'image d'entrée ne sont pas pris en compte. Si la fraction des points valides i.e. sous un masque = 1 dépasse un certain seuil ("seuil aire valide"), la valeur de sortie sera la somme des pixels dans la fenêtre × le facteur de correction du gain. Si les données dans la fenêtre sont considérées comme invalides, la valeur SMLVAL sera mise au centre de cette fenêtre.
Les images d'entrée et de sorties, ainsi que le masques ne doivent pas être les mêmes. Les tailles des fenêtres sont mises de façons à être impaires, c'est-à-dire que si une taille est paire, la fonction lui additionne 1. Taille minimale: 3.

Exemple: `(recavg input output mask 10 20 0.5 0.1)`
entrée filtrée avec le masque "mask". Si moins de 50 % de la fenêtre comporte des points valides, SMLVAL est mis au centre de la fenêtre.

Package: `2d`

`recmed` *imin imout mask x y seuil cor* Fonction

imin	Image entrée
imout	Image de sortie
mask	Image masque
x	Taille x: entier
y	Taille y: entier
seuil	Seuil aire valide: flottant
cor	Facteur de correction du gain: flottant

Description (2D): Même principe que `recavg`, sauf que c'est la médiane qui est calculée.

Exemple: `(recmed input output mask 10 20 0.5 0.1) ; filtre rectangulaire médian.`

Package: `2d`

`ImMedianKernel` *imin imout kernel* Fonction

imin	Image
imout	Image résultat
kernel	Noyau

Description (3D): La valeur médiane de *imin* est calculée sur l'ensemble de points *kernel* et est attribuée à *imout*.

Exemple: `(ImMedianKernel n1 n2 cuboctaedronK) ; n2 = mediane de n1 sur un cuboctaèdre de taille 1`

Package: `3d`

`recmod` *imin imout mask x y seuil cor* Fonction

imin	Image entrée
imout	Image de sortie
mask	Image masque
x	Taille x: entier
y	Taille y: entier
seuil	Seuil aire valide: flottant
cor	Facteur de correction du gain: flottant

Description (2D): Même principe que `recavg`, sauf que c'est le mode qui est calculé.

Exemple: `(recmod input output mask 10 20 0.5 0.1) ; filtre rectangulaire mode.`

Package: `2d`

`recrank` *imin imout mask x y seuil cor rank* Fonction

imin	Image entrée
imout	Image de sortie
mask	Image masque
x	Taille x: entier
y	Taille y: entier
seuil	Seuil aire valide: flottant
cor	Facteur de correction du gain: flottant
rank	Rang de l'histogramme

Description (2D): Cette fonction est similaire dans l'approche à la fonction `recavg`. La différence est dans un argument supplémentaire: le rang de l'histogramme.

Exemple: `(recrank input output mask 10 20 0.5 0.1 0.5)`

Même chose que `recavg`, sauf que le rang 0.5 est demandé à la fonction. Ce rang 0.5 est la médiane de l'histogramme.

Package: `2d`

`lowpass-v` *imin col1 size imout col2* Fonction

imin	Image d'entrée
col1	Numéro de la colonne d'entrée
size	Taille de la fenêtre
imout	Image de sortie
col2	Numéro de la colonne de sortie

Description (2D): Filtrage par moyenne mobile d'une colonne de l'image d'entrée. C'est remis sur une colonne de l'image de sortie, et on peut contrôler la taille de la fenêtre. La sortie est normalisée par la taille du filtre.

Exemple: `(lowpass-v input 3 200 output 0)`

Package: `2d`

`median-v` *imin col1 size imout col2* Fonction

imin	Image d'entrée
col1	Numéro de la colonne d'entrée
size	Taille de la fenêtre
imout	Image de sortie
col2	Numéro de la colonne de sortie

Description (2D): Filtrage par calcul de médian d'une colonne de l'image d'entrée. C'est remis sur une colonne de l'image de sortie, et on peut contrôler la taille de la fenêtre. La sortie est normalisée par la taille du filtre.

Exemple: `(median-v input 3 200 output 0)`

Package: `2d`

`median-vcol` *imin imout col height real mult mask* Fonction

imin	Image d'entrée
imout	Image de sortie
col	Numéro de la colonne de sortie
height	Hauteur de la fenêtre
real	Flottant: fraction de validité
mult	Facteur par lequel un pixel de sortie va être multiplié
mask	Image-masque

Description (2D): Trace la courbe du médian d'une image sur toute la largeur de celle-ci. La courbe est mise sur une image: on spécifie le numéro de la colonne. Le médian se fait sur une image-masque: seulement les pixels de l'image d'entrée qui sont sous le masque sont entrés dans l'histogramme. La valeur de sortie dépend également d'un seuil de validité: en effet, si la fenêtre n'est pas occupée par un marqueur à une certaine fraction donnée en argument, la valeur de sortie sera SMLVAL. Si l'image-masque n'est pas donnée, elle est considérée comme une image avec tous ses pixels à 1.

Exemple: `(median-vcol input out 3 100 0.75 1.0 masque)`

médian sur input. Image de sortie: out, colonne no. 3. Si il a moins de 75% de la fenêtre sur un marqueur, le centre de la fenêtre sera à SMLVAL.

Package: `2d`

`rank-v` *imin col1 size imout col2 rank* Fonction

imin	Image d'entrée
col1	Numéro de la colonne d'entrée
size	Taille de la fenêtre
imout	Image de sortie
col2	Numéro de la colonne de sortie
rank	Flottant: rang de l'histogramme

Description (2D): Filtrage d'une colonne de l'image d'entrée par calcul du rang. C'est remis sur une colonne de l'image de sortie, et on peut contrôler la taille de la fenêtre. La sortie est normalisée par la taille du filtre. Le rang est donné par le dernier argument de la fonction.

Exemple: `(rank-v input 3 200 output 0 0.5) ; un rang de 0.5 est une médiane.`

Package: `2d`

`rank-vcol` *imin imout col height real mult rank &optional mask* Fonction

imin	Image d'entrée
imout	Image de sortie
col	Numéro de la colonne de sortie
height	Hauteur de la fenêtre
real	Flottant: fraction de validité
mult	Facteur par lequel un pixel de sortie va être multiplié
rank	Flottant: rang
mask	Image-masque: optionnel

Description (2D): Fait une courbe de rang d'une image sur toute la largeur de celle-ci. La courbe est mise sur une image: on spécifie le numéro de la colonne. Le médian se fait sur une image-masque: seulement les pixels de l'image d'entrée qui sont sous le masque sont entrés dans l'histogramme. La valeur de sortie dépend également d'un seuil de validité: en effet, si la fenêtre n'est pas occupée par un marqueur à une certaine fraction donnée en argument, la valeur de sortie sera SMLVAL. Si l'image-masque n'est pas donnée, elle est considérée comme une image avec tous ses pixels à 1.

Exemple: `(rank-vcol input out 3 100 0.75 1.0 0.5 masque)`

rank sur input. Image de sortie: out, colonne no. 3. Si il a moins de 75% de la fenêtre sur un marqueur, le centre de la fenêtre sera à SMLVAL.

Package: `2d`

`scan-h` *imin image &optional image* Fonction

imin	Image d'entrée
image	Image de sortie ou image-masque
image	Image de sortie

Description (2D): Balaie l'image d'entrée et crée des courbes sur l'image de sortie. S'il y a 3 arguments, le second est un masque. S'il n'y a que 2 arguments, toute l'image est balayée horizontalement et les courbes sont faites en conséquence.
Voici ce qu'est chaque courbe:

Colonne 0: nombre de pixels marqueurs
Colonne 1: valeur min sur la ligne
Colonne 2: valeur max sur la ligne
Colonne 3: somme sur la ligne
Colonne 4: moyenne sur la ligne
Colonne 5: médian sur la ligne

Exemple: `(scan-h input out)`

Package: `2d`

`v-log` *imin mask code imout default expr6* Fonction

imin	Image d'entrée
mask	Image-masque
code	Code d'opération
imout	Image de sortie
col	Numéro de la colonne de l'image de sortie
default	Valeur de sortie s'il n'y a pas de pixels masques sur la ligne balayée

Description (2D): Cette fonction génère une colonne contenant une mesure dans l'image. Voici les statistiques extraites:

`"n"`: minimum sur la ligne horizontale.
`"x"`: maximum sur la ligne horizontale.
`"s"`: somme sur la ligne horizontale.
`"a"`: valeur moyenne sur la ligne horizontale.
`"0"`: valeur moyenne $\times 10^0$ sur la ligne horizontale.
`"1"`: valeur moyenne $\times 10^1$ sur la ligne horizontale.
`"2"`: valeur moyenne $\times 10^2$ sur la ligne horizontale.
`"3"`: valeur moyenne $\times 10^3$ sur la ligne horizontale.
`"4"`: valeur moyenne $\times 10^4$ sur la ligne horizontale.
`"5"`: valeur moyenne $\times 10^5$ sur la ligne horizontale.
`"6"`: valeur moyenne $\times 10^6$ sur la ligne horizontale.
`"v"`: variance sur la ligne horizontale.
`"v"`: variance sur la ligne horizontale.
`"m"`: médian sur la ligne horizontale.
`"e"`: mode sur la ligne horizontale: la valeur correspondant au maximum de l'histogramme.

La mesure est prise sous un masque: tout ce qui est sous le masque est pris comme mesure. Ce masque est optionnel. S'il n'y en a pas, ce seront tous les points de l'image d'entrée qui seront pris. Il faut faire attention au fait que le résultat est arrondi pour pour être mis sur une image entière. L'image d'entrée et de sortie peuvent être les mêmes.

Exemple: `(v-log input "x" out 0 (bigval))`

Package: `2d`

7.3.1 Génération de valeurs aléatoires

`ima-ran` *imout val* Fonction

imout	Image de sortie
val	Amplitude

Description (2D): Cette fonction met des nombre aléatoires suivant une distribution uniforme dans l'image donnée. On peut contrôler l'amplitude de l'image générée.

Exemple: `(ima-ran input 30)`

Package: `2d`

7.4 Analyse de particules individuelles

7.4.1 "Labelisation" de particules

`label4` *iminout* Fonction

iminout	Image d'entrée/sortie

Description (2D): Etiquetage des taches binaires sur l'image. L'entrée est la même image que la sortie. La propagation des étiquettes se fait en 4,6,8-connexité. La valeur retournée par la fonction: le nombre de taches.

Exemple: `(label4 image)`

Package: `2d`

`ImLabel` *imin imout &key (graph defaultGraph)* Fonction

imin	Image
imout	Image résultat
graph	Graphe

Description (3D): Etiquetage des composantes connexes de l'image *imin*. La connexité est définie par le graphe *graph*. Résultat dans *imout*. Chaque composante connexe reçoit une valeur différente. La numérotation repart à 1 si le nombre de composante connexes dépasse l'entier maximum représentable dans le type de l'image (par exemple 255 pour une image 8 bits). Il est possible de connaître le nombre de composantes connexes en invoquant la fonction `ImAccumulator`.

Exemple:

```
(ImThresh n1 n2 30.0 255.0)
(ImLabel n2 :graph square4Gr)  ; étiquetage d'une image 2D en 4-connexité
```

Package: `3d`

`ImLabelRegions` *imin imout &key (graph defaultGraph)* Fonction

imin	Image d'entrée
imout	Image
graph	Graphe

Description (3D): Etiquetage des plateaux de *imin*. La connexité est définie par le graphe *graph*. Résultat dans *imout*. Chaque plateau reçoit une valeur différente. La numérotation repart à 1 si le nombre de composante connexes dépasse l'entier maximum représentable dans le type de l'image (par exemple 255 pour une image 8 bits). Retourne le nombre des plateaux existant dans l'image *imin*.

Auteur: Beatriz Marcotegui
Package: `3d`

`firstlabel` *imin immask* Fonction

imin	Image d'entrée
immask	Image servant de masque

Description (2D): Cette fonction renvoie la valeur du premier numéro de région contenue à l'intérieur du masque (l'extérieur du masque vaut 0).

Exemple: `(firstlabel imin immask)`

Package: `vandroog`

`points4` *imin imout* Fonction

imin	Image d'entrée
imout	Image de sortie

Description (2D): Cette fonction prend les taches binaires de l'image d'entrée, et crée une image de sortie contenant un point appartenant à chaque tache. Ces taches sont considérées en 4,6,8-connexité. L'image d'entrée est labellisée à la fin, ainsi que l'image de sortie.

Exemple: `(points4 input out)`

Package: `2d`

`firstpoints4,6,8` *iminout* Fonction

iminout	Image d'entrée et de sortie

Description (3D): Le premier point de chaque région (lorsqu'on balaie l'image dans le sens vidéo) garde la valeur de la région. Le reste des points est mis a zéro. Retourne le nombre de régions.

Auteur: Beatriz Marcotegui
Package: `bea2d`

`ImRegionFirstPoint` *imin imout &key (graph defaultGraph)* Fonction

imin	Image
imout	Image résultat
graph	Graphe

Description (3D): Trouve le premier point de chaque région de *imin* et lui donne la valeur de la région dans *imout*. Ces points sont les premiers points de chaque région lorsqu'on balaie l'image dans le sens vidéo. Le reste des points est mis a zéro.

Auteur: Beatriz Marcotegui
Package: `3d`

`lspro4` *image x y val* Fonction

image	Image de sortie avec taches.
x	Coordonnée '`x`'
y	Coordonnée '`y`'
val	Valeur à propager

Description (2D): Cette fonction propage dans une image-marqueur une valeur donnée. On donne en entrée les coordonnées où un pont de la tache se trouve. La propagation se fait en 4,6,8-connexité. Les taches binaires sont tout ce qui est différent de zéro. On peut répéter l'opération plusieurs fois.

Exemple: `(lspro4 io 4 4 100) ; la tache à (4,4) est mise á 100.`

Package: `2d`

`blobsiz4` *iminout* Fonction

iminout	Image d'entrée/sortie

Description (2D): Propagation de la taille en pixels de chaque tache binaire dans celle-ci. Propagation en 4,6,8-connexité. La valeur retournée par la fonction: le nombre de taches.

Exemple: `(blobsiz4 image)`

Package: `2d`

`dirpro6` *iminout* Fonction

iminout	Image d'entrée/sortie

Description (2D): Cette fonction propage dans les taches de l'image d'entrée/sortie la valeur de l'angle que fait cette tache avec l'axe des '`x`'. La valeur retournée par cette fonction est le nombre de taches qu'il y a dans l'image.
Elle fonctionne en utilisant la régression linéaire: on a une liste des coordonnées '`x`' et '`y`' des points d'une tache donnée. Il est possible en utilisant de la régression linéaire standard de trouver la pente de la droite passant par ce "nuage" de points. En prenant l'arc-tangente de cette pente, on a l'angle. Il y a un petit problème, en particulier avec les nuages de points présentant une tendance à être verticaux. Ceci est corrigé en faisant la régression linéaire en inversant les axes '`x`' et '`y`'. La fonction choisit d'elle-même la bonne régression à faire en examinant la variance selon l'axe des '`x`' et selon l'axe des '`y`'. L'axe ayant la variance la plus grande sera celui qui sera considéré comme l'axe des '`x`'.

Exemple: `(dirpro6 image)`

Package: `2d`

7.4.2 Nombre d'Euler

`ImAccumulator` Fonction

Description (3D): Retourne la valeur numérique stockée dans le compteur C *ImAccumulator*. Ce compteur est utilisé par exemple dans l'algorithme d'étiquetage pour mémoriser le nombre de composantes connexes.

Exemple:

```
(ImLabel n1 n2 :graph cfc12Gr) ; Etiquetage
(format t "Nombre de composantes connexes =   a" (ImAccumulator))
   ; Nombre de CC
```

Package: `3d`

7.4.3 Statistiques d'image sur des domaines définis par des labels

`extpro4` *iminout image code* Fonction

iminout	Image-marqueur d'entrée/sortie
image	Image en niveaux de gris dont on va propager une valeur
code	Code d'opération: chaîne de caractères

Description (2D): Cette fonction propage dans chaque marqueur binaire de l'image-marqueur une valeur donnée par l'image en niveau de gris. La propagation se fait en 4,6,8-connexité.
Les codes d'opération sont les suivants:

- `"n"` minimum de la région
- `"x"` maximum de la région
- `"a"` valeur moyenne de la région
- `"m"` mode de l'histogramme de la région
- `"e"` médian de l'histogramme de la région
- `"s"` somme de la région

Il faut que l'image d'entrée et l'image de sortie soient différentes.

Exemple:

```
(extpro4 marqueurs gris "a") ; propage la valeur moyenne de gris
sous chaque tache de ``marqueurs''.
```

Package: `2d`

`ImLabelWithValue` *imregions imvalues imout method &key (graph defaultGraph)* Fonction

imregions	Image
imvalues	Image
imout	Image résultat
method	Entier
graph	Graphe

Description (3D): Les composantes connexes de l'image *imregions* recoivent une valeur calculée d'après les valeurs des pixels de l'image *imvalues*. Si *method* vaut 0, c'est la moyenne de *imvalues* sur la CC qui sera affectee à cette CC dans *imout*. Si *method* vaut 1, ce sera le maximum. Pour 2 ce sera la moyenne et enfin pour 3 ce sera la surface de la CC. La connexité est définie par le graphe *graph*. Résultat dans *imout*.

Exemple:

```
(ImWatershed n1 n2)   ;  Ligne de partage des eaux et bassins versants
(ImInvert (ImThresh n2 n2 0.0 0.0) n2)  ; Bassins versants dans n2
(ImLabelWithValue n2 n1 2 n2) ;  Moyenne de l'image originale affectée
à chaque bassin versant;
; La ligne de partage des eaux reste a la valeur 0.
```

Package: `3d`

`ImLabelRegionsWithValue` *imregions imvalues imout method &key (graph defaultGraph)* Fonction

imregions	Image
imvalues	Image
imout	Image résultat
method	Entier
graph	Graphe

Description (3D): Les plateaux de l'image *imregions* recoivent une valeur calculée d'après les valeurs des pixels de l'image *imvalues*. Si *method* vaut 0, c'est la moyenne de *imvalues* sur la CC qui sera affectée à cette CC dans *imout*. Si *method* vaut 1, ce sera le maximum. Pour 2 ce sera la moyenne et enfin pour 3 ce sera la surface de la CC. La connexité est définie par le graphe *graph*. Résultat dans *imout*.

Exemple:

```
(ImBasins n1 n2) & Bassins versants (sans ligne de partage des eaux)
(ImLabelRegionsWithValue n2 n1 2 n2) & Moyenne de l'image originale
affectée à chaque bassin versant
```

Package: `3d`

`cextpro4` *iminout image code* Fonction

iminout	Image-marqueur d'entrée/sortie
image	Image en niveaux de gris dont on va propager une valeur
code	Code d'opération: chaîne de caractères

Description (2D): Cette fonction propage dans chaque marqueur binaire de l'image-marqueur une valeur donnée par l'image en niveau de gris. La propagation se fait en 4,6,8-connexité. A la différence de `extpro4`, les valeurs sont prises sur le contour de la tache.
Les codes d'opération sont les suivants:

- `"n"` minimum de la région
- `"x"` maximum de la région
- `"a"` valeur moyenne de la région
- `"m"` mode de l'histogramme de la région
- `"e"` médian de l'histogramme de la région

Il faut que l'image d'entrée et l'image de sortie soient différentes.

Exemple:

```
(cextpro4 marqueurs gris "a") ; propage la valeur moyenne de gris
pris sur le contour des ``marqueurs''.
```

Package: `2d`

`propa6` *iminout imageref code* Fonction

iminout	Image d'entrée dont chaque plateau d'altitude constante sert de masque. Sert aussi d'image de sortie
imageref	Image de référence : image à niveaux de gris dont on va extraire une statistique à l'intérieur de chaque masque de l'image d'entrée.
code	Code d'opération: chaîne de caractères

Description (2D): Cette fonction considère chaque particule connexe de valeur homogène dans l'image d'entrée comme un masque séparé. A l'intérieur de chacun de ces masques, elle établit une valeur caractéristique de la région à partir de l'image de référence et écrit cette valeur dans l'image d'entrée/sortie. La valeur choisie peut être un maximum, minimum, une valeur moyenne, un médian. Elle est sélectionnée par le code d'opération. Voici sa signification:

`"n"`: minimum de la tache.
`"x"`: maximum de la tache.
`"a"`: valeur moyenne de la tache.
`"m"`: médian de la tache.
`"e"`: mode de la tache: la valeur correspondant au maximum de l'histogramme de la tache.

La propagation se fait pour l'instant en 6-connexité seulement.

Exemple: `(propa6 inout ref "a")` propage la valeur moyenne de gris de *iminout* à l'intérieur de chaque plateau de *imageref*. Le résultat est dans *iminout*.

Package: `talbot`

`slice-mes6` *iminout dir* Fonction

iminout	Image d'entrée et de sortie
dir	Direction dans laquelle est prise la mesure

Description (2D): Cette fonction fait un balayage de toute l'image dans la direction donnée en argument. Elle propage dans chaque tranche d'une tache binaire la longueur de cette tranche. C'est comme si on faisait la mesure de chaque coupe d'un objet binaire dans une direction donnée sur toute l'image.
Les directions sont tout simplement le numéro de la direction parce qu'on en fait qu'une à la fois. `slice-mes6` fonctionne sur des trames hexagonales et `slice-mes8` sur des trames carrées.
ATTENTION: il se peut que le fonctionnement de la fonction ne soit pas correct.

Exemple: `(slice-mes6 mark 1)`

Package: `2d`

7.5 Granulométrie

`fgranu4` *imin fileName* Fonction

imin	Image d'entrée
fileName	Fichier de sortie: chaine de caractères

Description (2D): Cette fonction fait l'analyse individuelle de particules considérées 4,6,8-connexes. Les résultats sont en ASCII sur le fichier de sortie. La valeur de retour de cette fonction: nombre de particules.
Format du fichier de sortie: numéro de la tache, taille de la particule en pixels, coordonnées du premier point vu lors d'un balayage sens raster `(x,y)`, centre de gravité `(x,y)`, coordonnées du dernier point vu lors d'un balayage sens raster `(x,y)`, fin de ligne.

Exemple: `(fgranu4 image "granulometrie.out")`

Package: `2d`

7.6 Fonction distance

`dist4` *iminout* Fonction

iminout	Image d'entrée et de sortie

Description (2D): Cette fonction calcule la fonction de distance en utilisant des érodés successifs par une croix pour `dist4`, un hexagone pour `dist6` et un carré pour `dist8`. Ce n'est évidemment pas de cette façon qu'elle à été mise en oeuvre, mais plutôt avec des algorithmes séquentiels.

Exemple: `(dist4 mark)`

Auteur: Hugues Talbot

Package: `2d`

`dist6` *expr1 imout* Fonction

expr1	Image binaire
imout	Image résultat contenant la fonction distance

Description (2D): Ces fonctions implémentent la notion de fonction distance pour une image binaire. Une image binaire est une image qui contient des objets sur un fond. Les objets sont repérés par une valeur positive quelconque et le fond par une valeur nulle.

Exemple: `(dist6 im1 im2)`

Auteur: Hugues Talbot

Package: `2d`

`eudf8` *imin imout* Fonction

imin	Image binaire d'entrée
imout	Image de sortie

Description (2D): Cette fonction donne la fonction distance euclidienne selon la méthode de Danielsson en 8-connexité. L'image d'entrée doit être une image binaire avec le fond à 0 et les objets à une valeur supérieure ou égale à 1. Sur l'image de sortie, les valeurs de la fonction distance sont ramenées à l'entier le plus proche. On n'a donc pas une précision optimale. On sait d'autre part que la fonction distance de Danielsson n'est qu'une approximation de la vraie fonction distance euclidienne. Cependant cette approximation est excellente, et sur une image en entier, comme celle que donne cette fonction, cette différence n'est absolument pas perceptible.

Exemple: `(eudf8 im1 im2) ; im2 contiendra la fonction distance euclidienne de im1.`

Auteur: Hugues Talbot

Package: `talbot`

`eudff` *imin imout* Fonction

imin	Image binaire d'entrée
imout	Image de sortie

Description (2D): Cette fonction donne également la fonction distance euclidienne, inspirée de celle de Danielsson, mais selon un algorithme à base de files d'attente, selon les méthodes proposées par Luc Vincent dans sa thèse, et reprises par Pierre Soille. Le résultat est une fonction distance qui donne les mêmes résultats (cas d'erreur compris !) que Danielsson, mais qui est jusqu'à 10 fois plus rapide selon les cas. Dans certains cas "pathologiques", elle peut cependant être un peu plus lente que la méthode de Danielsson de base (c'est pourquoi nous donnons les deux méthodes). Attention, *imin* doit être une image binaire.

Exemple: `(eudff im1 im2) ; im2 contiendra la fonction distance Euclidienne de im1.`

Auteur: Hugues Talbot

Package: `talbot`

`ImEuDistance` *imin imout* Fonction

imin	Image
imout	Image résultat

Description (3D): Fonction distance euclidienne de l'image binaire *imin*. Résultat dans *imout*. Le calcul est effectué en 2D selon un réseau carré, en 3D selon un réseau cubique.

Package: `3d`

`ImDistanceOnGraph` *imin imout &key (graph defaultGraph)* Fonction

imin	Image
imout	Image résultat
graph	Graphe

Description (3D): Fonction distance discrète sur le graphe défini par *graph* de l'image binaire *imin*. Résultat dans *imout*.

Package: `3d`

Chapter 8

Gradients

8.1 Gradients usuels

`asobel` *imin imout* Fonction

imin	Image d'entrée
imout	Image de sortie

Description (2D): Gradient de Sobel: le module de ce gradient est approximé par la valeur absolue du max entre le gradient vertical et horizontal.

```
                    1 0 -1                      1  2  1
    Horizontal      2 0 -2      Vertical        0  0  0
                    1 0 -1                     -1 -2 -1
```

Exemple: `(asobel input output)`

Package: `2d`

`roberts` *imin imout* Fonction

imin	Image d'entrée
imout	Image de sortie

Description (2D): Gradient de Roberts défini par $max\{abs[f(i,j) - f(i+1,j+1)], abs[f(i,j+1) - f(i+1,j)]\}$.

Exemple: `(roberts input output)`

Package: `2d`

8.2 Résidus

`ImInternalGradient` *imin imout se* Fonction

imin	Image
imout	Image résultat
se	Elément structurant

Description (3D): Gradient interne, i.e. *original* - *érodé*, de *imin* dans *imout* en utilisant l'élément structurant *se*.

Package: `3d`

`ImEXternalGradient` *imin imout se* Fonction

imin	Image
imout	Image résultat
se	Elément structurant

Description (3D): Gradient externe, i.e. *dilaté* - *original*, de *imin* dans *imout* en utilisant l'élément structurant *se*.

Package: `3d`

`ImMorphoGradient` *imin imout se* Fonction

imin	Image
imout	Image résultat
se	Elément structurant

Description (3D): Gradient morphologique, i.e. *dilaté* - *érodé*, de *imin* dans *imout* en utilisant l'élément structurant *se*.

Package: `3d`

`mgrad4` *imin imout* Fonction

imin	Image d'entrée
imout	Image de sortie

Description (2D): Gradient morphologique en utilisant une croix (4-connexité) pour `mgrad4`, un hexagone pour `mgrad6` et un carré pour `mgrad8` comme élément structurant. L'image d'entrée et de sortie peuvent être identiques.

Exemple: `(mgrad4 input output)`

Package: `2d`

8.3 Chapeau haut-de-forme

`wtophatr` *imin imout x* Fonction

imin	Image d'entrée
imout	Image de sortie
x	Taille de l'érosion.

Description (2D): Cette fonction est un top-hat blanc par reconstruction de taille donnée. Elle est implémentée par une fonction LISP dont voici le code:

Niveau: LISP

Exemple: `(wtophatr imin imout 5) ; top-hat de taille 5.`

Auteur: Hugues Talbot

Package: `talbot`

`btophatr` *imin imout x* Fonction

imin	Image d'entrée
imout	Image de sortie
x	Taille de l'érosion.

Description (2D): Cette fonction est un top-hat noir par reconstruction de taille donnée. Elle est implémentée par une fonction XLISP.

Niveau: LISP

Exemple: `(btophatr imin imout 5) ; top-hat noir de taille 5.`

Auteur: Hugues Talbot

Package: `talbot`

`ImWhiteTopHat` *imin imout se* Fonction

imin	Image
imout	Image résultat
se	Elément structurant

Description (3D): Chapeau haut-de-forme blanc (white), i.e. *original* − *ouverture*, de *imin* dans *imout* par l'élément structurant *se*.

Exemple:

```
(send cuboctaedronSE :Size 5) & Etablir la taille du cuboctaedre à 5
(ImWhiteTopHat n1 n2 cuboctaedronSE) & white top hat par un cuboctaèdre
de taille 5
```

Package: `3d`

`ImBlackTopHat` *imin imout se* Fonction

imin	Image
imout	Image résultat
se	Elément structurant

Description (3D): Chapeau haut-de-forme noir (black), i.e. $fermeture - original$, de *imin* dans *imout* par l'élément structurant *se*.

Exemple:

```
(send cuboctaedronSE :Size 5) & Etablir la taille du cuboctaedre à 5
(ImBlackTopHat n1 n2 cuboctaedronSE) & Black top hat par un cuboctaèdre
de taille 5
```

Package: `3d`

`ImWTHContrast` *imin imout se* Fonction

imin	Image
imout	Image résultat
se	Elément structurant

Description (3D): Accentuation du contraste par addition du chapeau haut-de-forme blanc par l'élément structurant *se*.

Exemple:

```
(send cuboctaedronSE :Size 2) & Etablir la taille du cuboctaedre à 2
(ImWTHContrast n1 n2 cuboctaedronSE) & Accentuation du contraste
```

Package: `3d`

`ImBTHContrast` *imin imout se* Fonction

imin	Image
imout	Image résultat
se	Elément structurant

Description (3D): Accentuation du contraste par soustraction du chapeau haut-de-forme noir par l'élément structurant *se*.

Exemple:

```
(send cuboctaedronSE :Size 2) & Etablir la taille du cuboctaedre à 2
(ImBTHContrast n1 n2 cuboctaedronSE) & Accentuation du contraste
```

Package: `3d`

8.4 Gradients de surface

8.4.1 Azimuth du gradient

`azgrad6` *imin imout* Fonction

imin	Image d'entrée
imout	Image de sortie

Description (2D): C'est l'azimuth du gradient. L'image d'entrée est déjà une image de gradient. Ce que fait cette fonction: elle examine dans le voisinage de taille 1 tous les points qui sont maxima. Elle encode sur la sortie les directions de chacun des ces points maxima. Les points sont encodés comme la fonction `arrow6 8 ()`. Il n'y a qu'une chose: si deux directions font opposition, elles sont supprimées.

Exemple: `(azgrad8 input output)`

Package: `2d`

`azsobel` *imin imout* Fonction

imin	Image d'entrée
imout	Image de sortie

Description (2D): Azimuth du gradient de Sobel: l'azimuth de ce gradient est fait en calculant l'arc-tangente du rapport du résultat de la convolution du masque de Sobel vertical et du résultat de la convolution du masque de Sobel horizontal. Le résultat est en degrés et arrondi à l'entier le plus près. Les images d'entrée et de sortie peuvent être les mêmes.

Exemple: `(azsobel input output)`

Package: `2d`

Chapter 9

Squelettes

9.1 Squelettes binaires construits à partir de marqueurs

`skel6` *imin marker imout* Fonction

imin	Image binaire a squelettiser
marker	Image binaire des marqueurs
imout	Image résultat

Description (2D): Ces fonctions sont une implémentation des squelettes binaires avec marqueurs selon le modèle de Luc Vincent (voir sa Thèse de doctorat). On obtient le squelette standard en donnant comme image des marqueurs le squelette par boules maximales (maxima locaux de la fonction distance). On peut obtenir toute une gamme de squelettes différents en donnant comme image des marqueurs quelque chose de différent, par exemple en donnant les érodés ultimes on obtient le squelette minimal, et rien du tout les marqueurs homotopiques des particules de l'image de départ. Les marqueurs doivent être compris dans l'adhérence des particules de départ, et leur forme est conservée au cours du traitement. Le squelette obtenu n'est pas forcément un ensemble de particules d'un pixel de large et une étape d'amincissement peut donc être nécessaire. Une image binaire est de la forme suivante: les pixels du fond ont une valeur nulle, les pixels des objets une valeur positive. Le résultat donne une image binaire dont les objets ont une valeur de 255. Ces algorithmes sont sensés être très rapides.

Exemple:

```
; on a alloué im1, im2, im3, ims images de travail.
(setf im1 (imread "test.tiff")) ; image binaire
(dist8 im1 im2) ; image de la fonction distance
(lmax8 im2 im3) ; maxima locaux (squelette par boules maximales)
(skel8 im1 im3 ims) ; le résultat est dans ims: squelette vrai.
```

Auteur: Hugues Talbot

Package: `talbot`

9.2 Centroïdes

`centroid6` *iminout* Fonction

iminout	Image d'entrée et sortie

Description (2D): Cette fonction construit le centroïde de toutes les particules connexes dans l'image à partir de leur frontière extérieure à valeur 0. Les trous sont entourés par une boucle fermée. Tous les points valant SMLVAL dans l'image gardent cette valeur et servent de points conditionnants. On peut ainsi construire le squelette total, en conditionnant par tous les points sans amont de la fonction distance ; on peut de même construire le squelette minimal en conditionnant par les érodés ultimes.

Exemple: `(centroid input)`

Package: `talbot`

9.3 Transformée par tout ou rien

`shmt` *imin lut imout number* Fonction

imin	Image d'entrée
lut	Image lut
imout	Image sortie
number	Nombre d'itérations.

Description (2D): Recherche des configurations particulières dans une image binaire. Fonctionne en trame carrée. L'idée est de donner un numéro à chaque type de configuration rencontré. La lut donnée sert à spécifier une action lorsque la configuration est rencontrée.
Les directions sont encodées comme ceci:

```
a4  a3  a2
a5  a0  a1
a6  a7  a8
```

Pour la lut, c'est une image de 512 × 1 pixels. Chaque pixel verra sa valeur mise sur l'image de sortie si sur l'image d'entrée on retrouve la configuration spécifiée par son numéro.
On peut tout faire avec ceci (ou presque). Par exemple, on peut faire des amincissements et épaississements.
Le nombre d'itérations peut être fixé. Si égal à 0, les itérations sont faites jusquá idempotence.

Exemple: `(shmt input lut out 0)`

Package: `2d`

`hhmt` *imin lut imout number* Fonction

imin	Image d'entrée
lut	Image lut
imout	Image sortie
number	Nombre d'itérations.

Description (2D): Recherche des configurations particulières dans une image binaire. Fonctionne en trame hexagonale. L'idée est de donner un numéro à chaque type de configuration rencontré. La lut donnée sert à spécifier une action lorsque la configuration est rencontrée.
Les directions sont encodées comme ceci:

```
  a3  a2
a4  a0  a1
  a5  a6
```

Pour la lut, c'est une image de 128 $\times$ 1 pixels. Chaque pixel verra sa valeur mise sur l'image de sortie si sur l'image d'entrée on retrouve la configuration spécifiée par son numéro.
On peut tout faire avec ceci (ou presque). Par exemple, on peut faire des amincissements et épaississements.
Le nombre d'itérations peut être fixé. Si égal à 0, les itérations sont faites jusquá idempotence.

Exemple: `(hhmt input lut out 0)`

Package: `2d`

9.4 Squelettes non-homothétiques

`lskel4,8` *imin imout* Fonction

imin	Image d'entrée
imout	Image de sortie

Description (2D): Ces fonctions donnent les squelettes en 4 et 8 connexité selon la methode proposée par Luc Vincent.

Niveau: XLISP Il s'agit de fonctions XLISP. Voir la documentation sur `skel4,8`.

Auteur: Hugues Talbot

Package: `talbot`

9.5 Amincissement et épaississement

9.5.1 Amincissement homotopique

`thin4` *iminout* Fonction

iminout	Image binaire à amincir

Description (2D): Ces fonctions sont des fonction d'amincissement homotopique d'une image binaire dans elle même. Dans le cas des squelettes de Luc Vincent, on obtient souvent des squelettes "epais" de 2 pixels de large. Une telle étape d'amincissement fait disparaître les zones "grasses". Il est dans ce cas vital que l'image soit amincie dans elle-même, car autrement le squelette disparaît en entier !

Exemple: `(thin4 ims)`

Auteur: Hugues Talbot

Package: `talbot`

9.5.2 Amincissement isotrope

`inthin4` *imin number code* Fonction

imin	Image d'entrée et de sortie
number	Entier: nombre de voisins.
code	Chaîne de caractères: code d'opération.

Description (2D): Cette fonction compte le nombre de voisins d'un point non-nul et fait un test décrit par le code d'opération. Si le test est positif, 0 est mis sur l'image de sortie à la place du pixel central. Cette opération est itérée jusqu'à idempotence. Cette opération est mise en oeuvre par files d'attente, et il faut faire attention: il peut y avoir des effets indésirables causés par l'ordre de traitement des pixels.
Voici la liste des opérations disponibles:

`"=="`: égal

Les tests se font en 4,6,8-connexité.

Exemple: `(inthin6 input 1 "==") ; Pour faire un ébarbulage.`

Package: `2d`

9.5.3 Squelette par fléchage

`amont48` *flba flho iminout* Fonction

flba	Image d'entrée : flèches descendantes
flho	Image d'entrée : flèches montantes
iminout	Image d'entrée/sortie binaire : points crête et amont

Description (2D): Cette fonction accepte comme arguments d'entrée les champs de flèches descendantes `flba` et montantes `flho` ainsi que les points crête associés. Elle effectue une remontée vers l'amont des points crête de manière à construire le squelette associé aux champs de flèches. Les points du squelette apparaissent avec une valeur 1 et les points crête avec une valeur 0 dans l'image de sorti. Pour que les résultats soient corrects, il faut que l'image `flba` soit complètée: tout point possède une flèche descendante à l'exception des minima régionaux qui ont une valeur nulle. La fonction amont6 travaille en trame 6. La fonction amont48 travaille en 4/8 connexité: le fond est en 4-connexité et les particules en 8-connexité. Pour cette raison le champ de flèches `flba` doit être 4-complet: tout point à l'exception des minima régionaux possède une flèche descendante dans les directions 0, 2, 4 ou 6.

Exemple: `(amont6 flbas flho cr)`

Package: `talbot`

9.6 Points critiques, étude du voisinage

9.6.1 Points de terminaison

`endpts4` *imin imout* Fonction

imin	Image d'entrée
imout	Image sortie

Description (2D): Détecteur de points extremité dans une image binaire. Cette détection est faite en 4,6,8-connexité. Les points qui vont ressortir seront ceux qui ont un seul voisin. La fonction retourne le nombre de points extremité.
Il faut que l'image d'entrée et l'image de sortie soient différentes.

Exemple: `(endpts4 input output)`

Package: `2d`

9.6.2 Points multiples

`mulpts4` *imin imout* Fonction

imin	Image d'entrée
imout	Image sortie

Description (2D): Détecteur de points triples et supérieurs dans une image binaire. Cette détection est faite en 4,6,8-connexité. Les points qui vont ressortir seront ceux qui ont plus de 2 voisins.
Il faut que l'image d'entrée et l'image de sortie soient différentes.

Exemple: `(mulpts4 input output)`

Package: `2d`

`ineigb4` *imin number code imout* Fonction

imin	Image d'entrée
number	Entier: nombre de voisins.
code	Chaîne de caractères: code d'opération.
imout	Image sortie

Description (2D): Cette fonction compte le nombre de voisins d'un point non-nul et fait un test décrit par le code d'opération. Si le test est positif, 1 est mis sur l'image de sortie. La fonction retourne une valeur.
Voici la liste des opérations disponibles:

`"=="`: égal
`"!="`: pas égal
`">"`: plus grand
`"<"`: plus petit
`">="`: plus grand ou égal
`"<="`: plus petit ou égal

Il faut que l'image d'entrée et l'image de sortie soient différentes. Les tests se font en 4,6,8-connexité.

Exemple: `(ineigb4 input 2 "==" output)`

Package: `2d`

9.7 Ebarbulage

`2d` *iminout* Fonction

iminout	Image d'entrée et de sortie

Description (2D): Fait un ébarbulage de l'image donnée. Les tests se font en 4,6,8-connexité.

Exemple: `(prune6 input) ; Pour faire un ébarbulage.`

Package: `2d`

`ebarb6` *imin imout length* Fonction

imin	Image d'entrée
imout	Image sortie
length	Longueur de l'ébarbulage - 1 de chaque côté

Description (2D): Routine d'ébarbulage pour images à niveaux de gris. Les images d'entrée sont ou bien des images de type squelette ou alors des images résidus d'une ouverture de taille 1. Dans une telle image résidu, chaque point a au moins un voisin plus petit que lui-même. Cette fonction commence par supprimer toutes les particules de longueur 1 ou 2. Puis, elle détecte les points extrémités, c'est à dire les points ayant 4 voisins consécutifs valant 0. Dans un dernier temps, on calcule la fonction distance à ces points extrémités et on ne garde que les points à une distance supérieure ou égale à la valeur `length`.

Exemple: `(ebarb6 input output length)`

Package: `2d`

Chapter 10

Géodésie

10.1 Erosion et dilatation géodésiques

`ImGeoErode` *imin immask imout se* Fonction

imin	Image
immask	Image
imout	Image résultat
se	Elément structurant

Description (3D): Erosion géodésique de *imin* dans *imout* par l'élément structurant *se*, au dessus (à l'extérieur) du masque *immask*.

Exemple:

```
(send cuboctaedronSE :Size 5) & Etablir la taille du cuboctaedre à 5
(ImGeoErode n1 n3 n2 cuboctaedronSE) & Erosion géodesique dans le
masque n3 par un cuboctaèdre de taille 5
```

Package: `3d`

`ImGeoDilate` *imin immask imout se* Fonction

imin	Image
immask	Image
imout	Image résultat
se	Elément structurant

Description (3D): Dilatation géodésique de *imin* dans *imout* par l'élément structurant *se*, à l'intérieur du masque *immask*.

Exemple:

```
(send cuboctaedronSE :Size 5) & Etablir la taille du cuboctaedre à 5
(ImGeoDilate n1 n3 n2 cuboctaedronSE) & dilatation géodesique dans
le masque n3 par un cuboctaèdre de taille 5
```

Package: `3d`

`gbipero` *imin imout x y* Fonction

imin	Image d'entrée
imout	Image de sortie
x	Coordonnée '`x`' du deuxième point de l'élément structurant
y	Coordonnée '`y`' du deuxième point de l'élément structurant

Description (2D): Erosion par un bi-point. On a l'origine et on donne les coordonnées du second point. Attention: l'élément structurant n'est pas transposé. L'image d'entrée et de sortie peuvent être les mêmes. La transformation est géodésique, ce qui fait qu'un point qui tomberait hors de l'image n'est par considéré dans le résultat.

Exemple: `(gbipero input output 1 1)`

Package: `2d`

`gbipdil` *imin imout x y* Fonction

imin	Image d'entrée
imout	Image de sortie
x	Coordonnée '`x`' du deuxième point de l'élément structurant
y	Coordonnée '`y`' du deuxième point de l'élément structurant

Description (2D): Dilatation par un bi-point. On a l'origine et on donne les coordonnées du second point. Attention: l'élément structurant n'est pas transposé. L'image d'entrée et de sortie peuvent être les mêmes. La transformation est géodésique, ce qui fait qu'un point qui tomberait hors de l'image n'est par considéré dans le résultat.

Exemple: `(gbipdil input output 1 1)`

Package: `2d`

10.2 Reconstructions

`drecons4` *iminout imcon* Fonction

iminout	Image à être érodée
imcon	Image de contrainte

Description (2D): C'est la fonction de reconstruction duale en niveaux de gris et en binaire. La première image donnée en argument est celle qui va être propagée et reconstruite dans la seconde, qui la limite dans son expansion. La propagation se fait en 4,6,8-connexité. Il faut que l'image d'entrée et l'image de sortie soient différentes.

Exemple: `(drecons4 mark limite)`

Package: `2d`

`ImOverBuild` *imin iminout &key (graph defaultGraph)* Fonction

imin	Image
iminout	Image résultat
graph	Graphe

Description (3D): Reconstruction (érosion géodésique jusqu'à idempotence) de *iminout* au dessus de *imin* selon le graphe *graph*, résultat dans *iminout*.

Exemple:
```
(ImIsCopy n1 n2)
(ImAddConstant n2 4.0)
(ImOverBuild n1 n2 :graph cfc12Gr)
```

Package: `3d`

`recons4` *iminout imcon* Fonction

iminout	Image à être dilatée
imcon	Image de contrainte

Description (2D): C'est la fonction de reconstruction en niveaux de gris et en binaire. La première image donnée en argument est celle qui va être propagée et reconstruite dans la seconde, qui la limite dans son expansion. La propagation se fait en 4,6,8-connexité.
Il faut que l'image d'entrée et l'image de sortie soient différentes.

Exemple: `(recons4 mark limite)`

Package: `2d`

`ImUnderBuild` *imin iminout &key (graph defaultGraph)* Fonction

imin	Image
iminout	Image résultat
graph	Graphe

Description (3D): Reconstruction (dilatation géodésique jusqu'à idempotence) de *iminout* sous *imin* selon le graphe *graph*, résultat dans *iminout*.

Exemple:
```
(ImIsCopy n1 n2)
(ImAddConstant n2 -4.0)
(ImUnderBuild n1 n2 :graph cfc12Gr)
```

Package: `3d`

`ImEroBuildOpen` *imin imout se &key (graph defaultGraph)* Fonction

imin	Image
imout	Image résultat
se	Elément structurant
graph	Graphe

Description (3D): Ouverture obtenue par érosion de *imin* par l'élément structurant *se*, suivie d'une reconstruction conditionnellement à *imin* en utilisant la connexité définie par *graph*. Résultat dans *imout*.

Exemple:
```
(ImEroBuildOpen n1 n2 (send cuboctaedronSE :Size 5) :graph
cfc12Gr)    ; ouverture par érosion reconstruction par un cuboctaèdre de
taille 5
```

Package: `3d`

`ImDilBuildClose` *imin imout se &key (graph defaultGraph)* Fonction

imin	Image
imout	Image résultat
se	Elément structurant
graph	Graphe

Description (3D): Fermeture obtenue par dilatation de *imin* par l'élément structurant *se*, suivie d'une reconstruction conditionnellement à *imin* en utilisant la connexité définie par *graph*. Résultat dans *imout*.

Exemple:
```
(ImDilBuildClose n1 n2 (send cuboctaedronSE :Size 5) :graph
cfc12Gr)    ; fermeture par dilatation reconstruction par un cuboctaèdre de
taille 5
```

Package: `3d`

10.2.1 Sélection d'objets par reconstruction

`fhol4` *imin imout* Fonction

imin	Image d'entrée
imout	Image de sortie

Description (2D): Ces fonctions remplissent les trous des objets binaires contenus dans l'image de départ. Les objets sur l'image de sortie ont pour valeur de niveau de gris 255.

Exemple: `(fhol4 im1 im2)`

Auteur: Hugues Talbot

Package: `talbot`

`ImFillHoles` *imin imout &key (graph defaultGraph)* Fonction

imin	Image
imout	Image résultat
graph	Graphe

Description (3D): Bouchage des trous de *imin* selon la connexité définie par le graphe *graph*. Résultat dans *imout*.

Package: `3d`

`remregion4` *imin imout* Fonction

imin	Image d'entrée contenant les labels des régions
imout	Image de sortie

Description (2D): Cette routine supprime les régions de taille unitaire (eu égard un voisinage en 4-connexité) qui se trouvent dans l'image d'entrée; une région de ce type reçoit le numéro de région de son voisin de gauche. Un bord d'épaisseur d'un pixel n'est pas traité. Par contre, les images d'entrée et de sortie peuvent être les mêmes.

Exemple: `(remregion4 imin imout)`

Package: `vandroog`

`elibo4` *imin imout* Fonction

imin	Image d'entrée
imout	Image de sortie

Description (2D): Ces fonctions éliminent les objets touchant les bords de l'image d'entrée. Les images d'entrée et de sortie sont binaires. Les objets sur l'image de sortie ont pour valeur de niveau de gris valant 255.

Exemple: `(elibo4 im1 im2)`

Package: `2d`

10.3 Résidus obtenus après reconstruction

`ImWhiteBuildTopHat` *imin imout se &key (graph defaultGraph)* Fonction

imin	Image
imout	Image résultat
se	Elément structurant
graph	Graphe

Description (3D): Chapeau haut-de-forme associé à l'ouverture par érosion et reconstruction par l'élément structurant *se*. Résultat dans *imout*.

Package: `3d`

`ImBlackBuildTopHat` *imin imout se &key (graph defaultGraph)* Fonction

imin	Image
imout	Image résultat
se	Elément structurant
graph	Graphe

Description (3D): Chapeau haut-de-forme associé à la fermeture par dilatation-reconstruction par l'élément structurant *se*. Résultat dans *imout*.

Package: `3d`

10.4 Fonctions de distance géodésique

`ImGeoDistanceOnGraph` *imin imout immask &key (graph defaultGraph)* Fonction

imin	Image
imout	Image résultat
immask	Image
graph	Graphe

Description (3D): Fonction distance discrète géodésique à l'intérieur du masque *immask* sur le graphe défini par *graph* de l'image binaire *imin*. Résultat dans *imout*.

Package: `3d`

`infdist4` *imin imout* Fonction

imin	Image d'entrée
imout	Image de sortie contenant le résultat.

Description (2D): Fonction distance géodesique aux bords descendants des plateaux pour des images à niveaux de gris. Un bord descendant d'un plateau étant constitué de tous les points frontières voisins d'un point plus bas. La fonction distance utilisée est celle associée à la connexité de la trame: les points à distance 1 d'un point donné sont tous ses voisins. Remarque: lorsque l'image d'entrée est une image binaire, on obtient la fonction distance de cette image binaire.

Exemple: `(infdist4 inim imdist)`

Package: `talbot`

10.5 Zones d'influence géodésique

10.5.1 Squelettes par zones d'influence

`skiz4` *iminout* Fonction

iminout	Image d'entrée et de sortie.

Description (2D): Fait le SKIZ de l'image en 4, 6 ou 8-connexité. Le SKIZ n'est pas euclidien, mais dépend fortement de la trame. Il est fait par files d'attente. La valeur retournée par la fonction correspond au nombre de particules initiales. On obtient des zones d'influence étiquetées séparées par des contours mis à `BIGVAL`. Si on utilise `skiz4`, on aura les zones labellisées en 4-connexité, et les frontières en 8-connexité, et vice-versa pour l'utilisation de `skiz8`. Il n'y a pas de ce type de problèmes pour la 6-connexité.

Exemple: `(skiz4 image)`

Package: `2d`

`gdskiz4` *iminout immask* Fonction

iminout	Image d'entrée et de sortie.
immask	Image du masque géodésique.

Description (2D): Fait le SKIZ géodésique de l'image en 4, 6 ou 8-connexité. Le SKIZ n'est pas euclidien, mais dépend fortement de la trame. Il est fait par files d'attente. La valeur retournée par la fonction correspond au nombre de particules initiales. On obtient des zones d'influence étiquetées séparées par des contours mis à `BIGVAL`. Si on utilise `skiz4`, on aura les zones labelisées en 4-connexité, et les frontières en 8-connexité, et vice-versa pour l'utilisation de `skiz8`. Il n'y a pas de ce type de problèmes pour la 6-connexité.

Exemple: `(gdskiz4 imin masque)`

Package: `2d`

Chapter 11

Ligne de partage des eaux, intégrale et fléchage

11.1 Ligne de partage des eaux

`wsh4` *imin imout* Fonction

imin	Image source
imout	Image sortie

Description (2D): C'est la ligne de partage des eaux brute, sans imposition de minima. Pour `wsh4`, la propagation se fait en 4-connexité, ce qui fait que la ligne de partage des eaux est en 8-connexité, et que les bassins versants sont en 4-connexité. Pour `wsh8`, c'est le contraire, les frontières sont en 4-connexité. Quant à `wsh6`, il n'y a pas de problèmes de connexité.
Il faut que l'image d'entrée et l'image de sortie soient différentes.

Exemple: `(wsh4 grad wagra)`

Package: `2d`

`czpe4` *image imR imG imB* Fonction

image	Image des labels étiquetés. Image de sortie également.
imR	Composante rouge de l'image couleur
imG	Composante verte de l'image couleur
imB	Composante bleue de l'image couleur

Description (2D): Cette fonction permet de segmenter des images couleurs à partir d'un jeu de marqueurs étiquetés. Sont données au départ une image *image* contenant tous les marqueurs étiquetés et 3 composantes de couleur *imR*, *imG*, *imB*. Il s'agit d'un algorithme de croissance de régions. Le jeu consiste à faire conquérir le maximum de territoire à chaque marqueur. Tant que l'algorithme ne s'est pas stabilisé, on affecte pendant chaque cycle un point de l'image à un marqueur voisin: ce point est choisi parce que sa couleur est la plus proche de la couleur d'un point déjà couvert par le marqueur.

Exemple: `(czpe4 lab imR imG imB)`

Package: `talbot`

`ImBasins` *imin imout &key (graph defaultGraph)* Fonction

imin	Image
imout	Image résultat
graph	Graphe

Description (3D): Détermination des bassins versants de *imin* selon le graphe *graph*. Le résultat est écrit dans *imout*. A chaque bassin versant est associée une valeur de gris distincte (sauf s'il y a plus de bassins que la valeur maximale représentable dans le type de l'image). Il n'y pas de ligne de séparation entre les bassins.

Exemple: (ImBasins n1 :graph cfc12Gr)
(ImThresh n1 n2 1.0 1.0 255.0) ; n2 contient le premier bassin versant

Package: `3d`

`ImWatershed` *imin imout &key (graph defaultGraph)* Fonction

imin	Image
imout	Image résultat
graph	Graphe

Description (3D): Ligne de partage des eaux de *imin* selon le graphe *graph*. Résultat dans l'image *imout*. A chaque bassin versant est associée une valeur de gris distincte (sauf s'il y a plus de bassins que la valeur maximale représentable dans le type de l'image) et la ligne de partage est à la valeur 0.

Exemple: `(ImWatershed n1 :graph hexagonal6Gr) (ImThresh n1 n1 0.0 0.0)`

Package: `3d`

`mosa4` *image1 imout image2 code* Fonction

image1	Image dont les minima régionaux serviront de masque.
imout	Image résultat.
image2	Image à niveaux de gris dont une statistique sera extraite sous les masques.
code	Code d'opération: chaine de caractères

Description (2D): Cette fonction détecte d'abord les minima régionaux de l'image *image1*. Puis en se servant de chacun de ces minima régionaux comme d'un masque, elle établit une valeur caractéristique de la région à partir de l'image *image2* et écrit cette valeur dans l'image de sortie *imout*. La valeur choisie peut être un maximum, minimum, une valeur moyenne, un médian. Elle est sélectionnée par le code d'opération. Voici sa signification:

`"n"`: minimum de la tache.
`"x"`: maximum de la tache.
`"a"`: valeur moyenne de la tache.
`"m"`: médian de la tache.
`"e"`: mode de la tache: la valeur correspondant au maximum de l'histogramme de la tache.

La propagation se fait en 4,6,8-connexité. Il faut que les trois images soient différentes. La routine `mosa4,6,8` retourne le nombre de minima rencontrés dans l'image *image1* .

Exemple: `(mosa6 gradient sortie image "a")` propage la valeur moyenne de gris de "image" à l'intérieur de chaque minimum régional de "gradient". Le résultat est dans "sortie".

Package: `2d`

`ImConstrainedWS` *imregions imgradient imout &key (graph defaultGraph)* Fonction

imregions	Image
imgradient	Image
imout	Image résultat
graph	Graphe

Description (3D): L'image *imregions* contient des composantes connexes possédant chacune un niveau de gris (pas nécessairement distincts). `ImConstrainedWS` propage ces valeurs en restant dans les limites des bassins versants de *imgradient*. Ceci revient à faire une ligne de partage des eaux avec les marqueurs imposés que constituent les CC de *imregions* et à affecter à chaque bassin versant la valeur du label de départ dans *imregions*. Le résultat *imout* est la ligne de partage des eaux de *imgradient* dont l'homotopie a été modifiée en imposant comme minima les composantes connexes de *imregions*.

Auteur: Beatriz Marcotegui
Package: `3d`

11.2 Ligne de partage des eaux avec marqueurs

`lpe4` *minima image imout* Fonction

minima	Image minima étiquetés
image	Image dont on veut faire la ligne de partage des eaux
imout	Image de sortie

Description (2D): C'est la ligne de partage des eaux avec reconstruction de façon à éliminer les minima en trop. Ainsi, on a une image de marqueurs étiquetés de façon à imposer les minima marqués.
Le résultat, c'est une image avec les bassins versants qui gardent les mêmes numéros que les minima imposés. Ces bassins versants sont séparés par la ligne de partage des eaux qui a la valeur maximale de l'image. Pour `lpe4`, la propagation se fait en 4-connexité, ce qui fait que la ligne de partage des eaux est en 8-connexité, et que les bassins versants sont en 4-connexité. Pour `lpe8`, c'est le contraire, les frontières sont en 4-connexité. Quant à `lpe6`, il n'y a pas de problèmes de connexité.
Il faut que toutes les images soient différentes (ce que le programme ne vérifie pas!).

Exemple: `(lpe4 lab grad dest)`

Package: `2d`

`ImMosaic` *imregions imgradient imout &key (graph defaultGraph)* Fonction

imregions	Image
imgradients	Image
imout	Image résultat
graph	Graphe

Description (3D): L'image *imregions* contient des composantes connexes (CC) possédant chacune un niveau de gris (pas nécessairement distinct). `ImMosaic` propage ces valeurs en restant dans les limites des bassins versants de *imgradients*. Ceci revient à faire une ligne de partage des eaux avec les marqueurs imposés que constituent les CC de *imregions* et à affecter à chaque bassin versant la valeur du minimum de départ dans *imregions*. Le résultat, *imout*, est une sorte de mosaïque dont les morceaux sont les bassins versants de *imgradients* dont l'homotopie aurait été modifiée.

Exemple:

```
(ImMinima a c)              ; c contient les minima de l'image a
(ImLabelWithValue c b c 2) ; chaque minimum a maintenant pour valeur
                             la moyenne de l'image b sur le minimum
(ImMosaic c a c)    ; extension de ces minima à la totalité
                      de l'image en respectant les bassins versants
```

Package: `3d`

`zpe4` *minima iminout* Fonction

minima	Image des labels étiquetés. Image de sortie également.
iminout	Image dont on veut calculer les zones de partage des eaux

Description (2D): C'est la ligne de partage des eaux avec reconstruction de façon à éliminer les minima en trop. Ainsi, on a une image de marqueurs étiquetés de façon à imposer les minima marqués.
Le résultat, c'est une image avec les bassins versants qui gardent les mêmes numéros que les minima imposés. Ces bassins versants ne sont pas séparés par la ligne de partage des eaux comme pouvait l'être `lpe`.
Il faut que toutes les images soient différentes.

Exemple: `(zpe4 lab grad)`

Package: `2d`

11.3 Intégrale

`integ4` *imlimout image* Fonction

imlimout	Image des conditions limite pour l'intégration et image de sortie
image	Image qu'on cherche à intégrer

Description (2D): Cette fonction construit l'intégrale de l'image *image* (un gradient par exemple) en utilisant comme conditions aux limites l'image *imlimout*. C'est à dire qu'elle construit une image dont le gradient "image - érodé" est identique à l'image *image* en tout point où l'image *imlimout* est nulle. L'image *imlimout* reste inchangée en tout point où elle est non nulle. Il faut que l'image d'entrée et l'image de sortie soient différentes.

Exemple: `(integ4 lim grad)`

Package: `talbot`

`labinteg4` *imlimout image imlabel* Fonction

imlimout	Image des conditions limite pour l'intégration et image de sortie
image	Image qu'on cherche à intégrer
imlabel	Image de labels propagés pendant l'intégration

Description (2D): Cette fonction construit l'intégrale de l'image *image* (une image gradient par exemple) en utilisant comme conditions aux limites l'image *imlimout*. C'est à dire qu'elle construit une image dont le gradient "image - erodé" est identique à l'image *image* en tout point où l'image *imlimout* est nulle. L'image *imlimout* reste inchangée en tout point où elle est non nulle. Chaque point de l'image *imlimout* permet de calculer un certain nombre de points dans l'image résultat. L'image de labels garde la trace de ces filiations. Le label de chaque point de l'image limite est propagé à tous les points qu'il engendre. On peut ainsi construire une image mosaique de type ligne de partage des eaux de l'intégrale du gradient. Il faut que l'image d'entrée et l'image de sortie soient différentes.

Exemple: `(labinteg4 lim grad lab)`

Package: `talbot`

11.4 Fléchage

`arrow4,8` *imin imout code* Fonction

imin	Image d'entrée
imout	Image sortie
code	Chaîne de caractères: code de l'opération

Description (2D): Fait un fléchage en 4- ou 8-connexité. Les directions sont encodées comme ceci:

```
a3  a2  a1
a4      a0
a5  a6  a7
```

Liste des codes d'opération: le centre reçoit une flèche s'il est:

`">"` plus grand que le voisin
`"<"` plus petit que le voisin
`">="` plus grand ou égal que le voisin
`"<="` plus petit ou égal que le voisin
`"=="` égal au voisin
`"!="` pas égal au voisin

L'image d'entrée doit être différente de l'image de sortie. En connexité 4, on ne considère que les voisins dans les directions 0, 2, 4 et 6. La propagation se fait en 4,8-connexité. ATTENTION, il semblerait que `arrow8` ne fonctionne pas correctement.

Exemple: `(arrow8 input out ">")`

Package: `talbot`

`ImArrow` *imin imout relation &key (graph defaultGraph)* Fonction

imin	Image
imout	Image résultat
relation	Chaîne de caractères
graph	Graphe

Description (3D): Génère l'image de fléchage associée à l'image *imin* et à la relation *relation*. *relation* peut prendre les valeurs suivantes: " $>$ ", " $<$ " et " $=$ ". Le bit I d'un pixel x de *imout* (0 correspondant au bit de poids faible) vaut 1 si $imin(x)$ $relation$ $imin(voisin_I(x))$ est verifié. $voisin_I(x)$ est la valeur du Ieme voisin de x au sens du graphe defini par *graph*.

Package: `3d`

`arrow6` *imin imout code* Fonction

imin	Image d'entrée
imout	Image sortie
code	Chaîne de caractères: code de l'opération

Description (2D): Fait un fléchage en 6-connexité. Les directions sont encodées comme ceci:

```
  a2  a1
a3      a0
  a4  a5
```

Liste des codes d'opération: le centre reçoit une flèche s'il est:

`">"` plus grand que le voisin
`"<"` plus petit que le voisin
`">="` plus grand ou égal que le voisin
`"<="` plus petit ou égal que le voisin
`"=="` égal au voisin
`"!="` pas égal au voisin

L'image d'entrée doit être différente de l'image de sortie.

Exemple: `(arrow6 input out ">")`

Package: `talbot`

`fl_inv6` *imin imout* Fonction

imin	Image d'entrée
imout	Image sortie

Description (2D): Cette fonction inverse le champ de flèches de l'image d'entrée *imin*. Si dans l'image d'entrée *imin*, le pixel a possède une flèche dans la direction du pixel b, alors dans l'image de sortie, le pixel b possèdera une flèche vers le pixel a. L'image d'entrée et de sortie doivent être différentes.

Package: `talbot`

`ImInvertArrow` *imin imout &key (graph defaultGraph)* Fonction

imin	Image
imout	Image résultat
graph	Graphe

Description (3D): Génère du fléchage inverse du fléchage *imin*: si x possède une flèche vers y, c'est maintenant y qui possède une flèche vers x.

Package: `3d`

`fl_comp6` *imin imout* Fonction

imin	Image d'entrée
imout	Image sortie

Description (2D): Cette fonction complémente le champ de flèches de chaque pixel. Les flèches de l'image de sortie se trouvent dans les directions où il n'existe pas de flèches dans l'image d'entrée. L'image d'entrée et de sortie peuvent être identiques.

Auteur: Fernand Meyer

Package: `talbot`

`icomplet4,8` *iminout* Fonction

iminout	Image d'entrée et de sortie

Description (2D): Cette fonction complète le champ de flèches *iminout*. Appelons "flècheurs" les points dont part une flèche. L'algorithme est basé sur des files d'attente mais est équivalent à l'algorithme plus simple suivant. Répéter jusqu'à stabilité:

> Soit x le point courant rencontré lors du balayage de l'image: - si x n'est pas flécheur, il devient flécheur vers chacun de ses voisins déjà flécheur à la fin du balayage précédent.

Après convergence tout point est devenu flécheur, à l'exception des minima régionaux de l'image. Remarque : `icomplet4` ne considère que 4 voisins alors que `icomplet8` considère les 8 voisins.

Package: `talbot`

`fl_ero4,6,8` *imin imout* Fonction

imin	Image d'entrée
imout	Image sortie

Description (2D): Cette fonction fait l'érosion d'un champ de flèches *imin*. Pour chaque point on considère la couronne de ses voisins (6 voisins en 6-connexité, 4/8 voisins en 8-connexité). On considère l'ensemble binaire des points vers lesquels arrive une flèche du point central dans le champ de flèchage *imin*. On effectue une érosion géodésique de ces points sur la couronne de voisins. Puis on construit les flèches allant du point central vers les points à valeur 1 de l'image érodée. L'image d'entrée et de sortie peuvent être identiques.

Package: `talbot`

`fl_dil4,6,8` *imin imout* Fonction

imin	Image d'entrée
imout	Image sortie

Description (2D): Cette fonction fait la dilatation d'un champ de flèches *imin*. Pour chaque point on considère la couronne de ses voisins (6 voisins en 6-connexité, 4/8 voisins en 8-connexité). On considère l'ensemble binaire des points vers lesquels arrive une flèche du point central dans le champ de flèchage *imin*. On effectue une dilatation géodésique de ces points sur la couronne de voisins. Puis on construit les flèches allant du point central vers les points à valeur 1 de l'image dilatée. L'image d'entrée et de sortie peuvent être identiques.

Package: `talbot`

`fl_isole6` *imin imout* Fonction

imin	Image d'entrée
imout	Image sortie

Description (2D): Cette fonction analyse le flèchage de chaque point. Si dans le champ de flèches il n'y a aucune flèche ou bien uniquement des flèches isolées, le résultat vaut 1, sinon il vaut 0. L'image d'entrée et de sortie peuvent être identiques.

Package: `talbot`

`fl_unite4` *imin imout* Fonction

imin	Image d'entrée
imout	Image sortie

Description (2D): Cette fonction analyse le flèchage de chaque point. Si dans le champ de flèches il n'y a qu'une seule composante connexe de flèches, le résultat vaut 1, sinon il vaut 0. L'image d'entrée et de sortie peuvent être identiques.

Package: `talbot`

`fl_nb4` *imin imout* Fonction

imin	Image d'entrée
imout	Image sortie

Description (2D): Cette fonction analyse le flèchage de chaque point. L'image de sortie *imout* contient le nombre de composantes connexes de flèches dans le voisinage de chaque point. Ce comptage se fait uniquement en connexité 4 lorsqu'on est en trame carrée. L'image d'entrée et de sortie peuvent être identiques.

Auteur: Fernand Meyer

Package: `talbot`

`crete48` *imin1 imin2 imout* Fonction

imin1	Image d'entrée : flèches descendantes
imin2	Image d'entrée : flèches montantes
imout	Image sortie binaire : points crête valant 0

Description (2D): Cette fonction prend les champs de flèches descendantes *imin1* et montantes *imin2* en entrée et produit les points crête en sortie. Les points non crête apparaissent avec une valeur 1 et les points crête avec une valeur 0 dans l'image de sortie *imout*. Pour que les résultats soient corrects, il faut que l'image *imin1* soit complètée: tout point possède une flèche descendante à l'exception des minima régionaux qui ont une valeur nulle. Ces points de valeur nulle dans *imin1* ne sont pas détectés comme des points crête. La fonction `crete6` travaille en trame 6. La fonction `crete48` travaille en 4/8 connexité : le fond est en 4-connexité et les particules en 8-connexité. Pour cette raison le champ de flèches *imin1* doit être 4-complet : tout point à l'exception des minima régionaux possède une flèche descendante dans les directions 0, 2, 4 ou 6.
En trame carrée les directions sont encodées de la manière suivante:

```
a3  a2  a1
a4      a0
a5  a6  a7
```

Exemple: `(crete6 flbas flho cr)`

Package: `talbot`

`ImCompleteArrow` *imin imout &key (graph defaultGraph)* Fonction

imin	Image
imout	Image résultat
graph	Graphe

Description (3D): Complétude inférieure du fléchage *imin*, résultat dans *imout*.

Package: `3d`

Chapter 12

Détection des extrema et dynamique

12.1 Détection d'extrema régionaux

`minlab4` *imin imout* Fonction

imin	Image d'entrée
imout	Image de sortie

Description (2D): Détecteur de minima dans une image. Chaque minimum porte un label différent des autres sur l'image de sortie. Fonctionne sur une trame carrée ou hexagonale (4,6 et 8-connexité).
Il faut que l'image d'entrée et l'image de sortie soient différentes.

Exemple: `(minlab8 input output)`

Package: `2d`

`ImMinima` *imin imout &key (graph defaultGraph)* Fonction

imin	Image
imout	Image résultat
graph	Graphe

Description (3D): Détermination des minima régionaux de *imin*, résultat dans *imout*. La connexité est définie par le graphe *graph*.

Package: `3d`

`ImExtendedMinima` *imin imout depth &key (graph defaultGraph)* Fonction

imin	Image
imout	Image résultat
depth	Entier
graph	Graphe

Description (3D): Détermination des minima régionaux étendus de profondeur *depth* de *imin*, résultat dans *imout*. La connexité est définie par le graphe *graph*.

Package: `3d`

`ImModifyMinima` *imin iminout &key (graph defaultGraph)* Fonction

imin	Image
iminout	Image résultat
graph	Graphe

Description (3D): Modifie l'homotopie de *imin*, en lui imposant les minima contenus dans l'image "binaire" *iminout*, selon le graphe *graph*. Les minima à imposer doivent être à la valeur `(ImMaxValue iminout)` dans *iminout*.

Exemple:

```
(ImMinima n1 n2 :graph cfc12Gr) ; minima de n1 dans n2
(ImOpen n2 n2 (send cuboctaedronSE :Size 4))
         ; filtrage des minima selon leur taille
(ImModifyMinima n2 n1 :graph cfc12Gr) ; modification de n1 qui ne conserve
plus que des minima d'une certaine taille
```

Package: `3d`

`ImMinimaFirstPoint` *imin imout &key (graph defaultGraph)* Fonction

imin	Image
imout	Image résultat
graph	Graphe

Description (3D): Trouve le premier point de chaque minimum régional de *imin*. A la fin de l'opération *imout* contient autant de points à 1 qu'il y a de minima, le reste étant à 0. Ces points sont les premiers points des minima au sens de l'ordre de lecture (de gauche à droite et de haut en bas).

Package: `3d`

`ImMaxima` *imin imout &key (graph defaultGraph)* Fonction

imin	Image
imout	Image résultat
graph	Graphe

Description (3D): Détermination des maxima régionaux de *imin*, résultat dans *imout*. La connexité est définie par le graphe *graph*.

Package: `3d`

`lmax` *imin imout val* Fonction

imin	Image d'entrée
imout	Image de sortie
val	Dynamique des maxima à détecter.

Cette fonction détecte les maxima regionaux étendus d'une taille donnée (voir thèse de M. Grimaud).

Niveau: XLISP
Elle est implémentée par une fonction XLISP.

Exemple: `(lmax imin imout 10) détecte les maxima régionaux de taille 10.`

Auteur: Hugues Talbot

Package: `talbot`

`ImExtendedMaxima` *imin imout height &key (graph defaultGraph)* Fonction

imin	Image
imout	Image résultat
height	Entier
graph	Graphe

Description (3D): Détermination des maxima régionaux étendus de hauteur *height* de *imin*, résultat dans *imout*. La connexité est définie par le graphe *graph*.

Package: `3d`

`ImModifyMaxima` *imin iminout &key (graph defaultGraph)* Fonction

imin	Image
iminout	Image résultat
graph	Graphe

Description (3D): Modifie l'homotopie de *imin*, en lui imposant les maxima contenus dans l'image "binaire" *iminout*, selon le graphe *graph*. Les maxima à imposer doivent être à la valeur (ImMaxValue iminout) dans *iminout*.

Exemple:

```
(ImMaxima n1 n2 :graph cfc12Gr) ; maxima de n1 dans n2
(ImOpen n2 n2 (send cuboctaedronSE :Size 4))
          ; filtrage des maxima selon leur taille
(ImModifyMaxima n2 n1 :graph cfc12Gr) ; modification de n1
          qui ne conserve plus que des maxima d'une certaine taille
```

Package: `3d`

`lmax6` *imin imout* Fonction

imin	Image de départ
imout	Image des maxima locaux de l'image

Description (2D): Ces fonctions donnent comme résultat les maxima *locaux* de l'image d'origine, à ne pas confondre avec les maxima *régionaux* de cette même image. Un maximum est local si dans le plus petit voisinage possible au sens de la trame de travail, le niveau de gris du pixel considéré est supérieur *ou égal* à celui des pixels dans le voisinage considéré. Un pixel fait partie d'un maximum régional s'il fait partie une région de même valeur de niveau de gris, strictement supérieure à la valeur du niveau de gris dans le dilaté de cet ensemble par le voisinage minimal de la trame considérée. On voit qu'un maximum local n'est pas forcément régional mais que le contraire est vrai. Dans le cas où on prend les maxima locaux de la fonction distance d'un ensemble binaire, on obtient le squelette par boules maximales (non connexe) de cet ensemble.

Exemple: `(lmax6 im1 im2)`

Auteur: Hugues Talbot

Package: `talbot`

12.2 Dynamique

`hdynmin48,6` *imin imout connectivity* Fonction

imin	Input image
imout	Output image
imout	Integer giving the connectivity

Description (2D): This function produces the dynamics of the input image minimums. Note that it depends on the connectivity (an argument is used to precise the connectivity).

Auteur: Hugues Talbot

Package: `talbot`

`ImDynMinima` *imin imout &key (graph defaultGraph)* Fonction

imin	Image
imout	Image résultat
graph	Graphe

Description (3D): Détermination des minima régionaux de *imin* dans *imout*. On attribue aux minima la valeur de leur dynamique. La connexité est définie par le graphe *graph*.

Auteur: Beatriz Marcotegui
Package: `3d`

Chapter 13

Les graphes 3D

13.1 Graphes prédéfinis et fonctions s'y appliquant

13.1.1 Variables

`cfc12Gr` Variable

Description (3D): Graphe constitué des 12 points les plus proches en réseau cubique à face centrée (3D).

Exemple:

```
(ImMaxima n2 n1 :graph cfc12Gr)    ; maxima 3D selon le graphe 12-connexe
```

`cubic26Gr` Variable

Description (3D): Graphe constitué des 26 points les plus proches en réseau cubique (3D).

Exemple:

```
(ImMaxima n2 n1 :graph cubic26Gr) ; maxima 3D selon le graphe 26-connexe
```

`cubic6Gr` Variable

Description (3D): Graphe constitué des 6 points les plus proches en réseau cubique (3D).

Exemple:

```
(ImMaxima n2 n1 :graph cubic6Gr)  ; maxima 3D selon le graphe 6-connexe
```

`hexagonal6Gr` Variable

Description (3D): Graphe constitué des 6 points les plus proches en réseau hexagonal (2D).

Exemple:

```
(ImMaxima n2 n1 :graph hexagonal6Gr) ; maxima 2D selon le graphe
6-connexe
```

`square4Gr` Variable

Description (3D): Graphe constitué des 4 points les plus proches en réseau carrée (2D).

Exemple:

```
(ImMaxima n2 n1 :graph square4Gr)  ; maxima 2D selon le graphe 4-connexe
```

`square8Gr` Variable

Description (3D): Graphe constitué des 8 points les plus proches en réseau carrée (2D).

Exemple:

```
(ImMaxima n2 n1 :graph square8Gr)  ; maxima 2D selon le graphe 8-connexe
```

13.1.2 Fonctions sur les graphes

`GraphSize` *graph* Fonction

graph	Graphe

Description (3D): Retourne le nombre d'éléments du graphe.

Package: `3d`

`DefGraph` *graph* Fonction

graph	Graphe

Description (3D): Initialise la variable *defaultGraph* à la valeur *graphe* et désactive la variable **warnDefaultGraph**.
Dans les fonctions de `xlim3d` faites par propagation (watershed, labélisation, calcul des composantes connexes, ...), il faut préciser quels sont les voisins d'un pixel (le *graphe*). Le graphe au niveau lisp est un paramètre *key*, initialisé par défaut à la valeur de la variable globale *defaultGraph*. En étant un paramètre *key*, il peut être présent ou absent dans la liste d'arguments d'appel à la fonction. S'il est présent, la fonction utilisera comme graphe le paramètre spécifié dans la liste d'arguments. S'il est absent la fonction utilisera le graphe par défaut, c'est-à-dire la variable *defaultGraph*.
Quand on utilise le graphe par défaut, le système nous envoie un message d'alarme: "Warning: default graph used" au cas où on n'a toujours pas initialisé la variable *defaultGraph* selon les besoins (cette variable est initialisée à NIL par le système). Après avoir initialisé correctement cette variable pour éviter le message d'alarme, il faut désactiver la variable **warnDefaultGraph**.

Exemple:

```
(ImWatershed in out :graph cubic26Gr)
; Est la ligne de partage des eaux utilisant comme graphe la trame
; cubique avec 26 voisins.
(ImWatershed in out)
; Est la ligne de partage des eaux utilisant comme graph
;  defaultGraph. Le message d'alarme est apparu.
(DefGraph cubic26Gr)
; Définition du graphe par défaut à cubic26Gr et désactivation
; du message d'alarme
(ImWatershed in out)
; Ligne de partage des eaux utilisant le graphe par défaut
(cubic26Gr)
; Le message d'alarme n'apparaît plus
```

Auteur: Beatriz Marcotegui
Package: `3d`

Chapter 14

Codage

14.1 Chaînes de Freeman

`chain-code8` *imin x y imout* Fonction

imin	Image d'entrée
x	Entier: coordonée '`x`' du point de départ de la chaîne Ce point appartient au contour de l'objet
y	Entier: coordonée '`y`' du point de départ de la chaîne Ce point appartient au contour de l'objet
imout	Image sortie: ne doit pas être initialisée

Description (2D): Cette fonction génère une chaîne en suivant le contour d'une tache sur une image binaire. On donne les coordonnées du pixel appartenant au contour de cette tache, et la fonction retourne une image qui a trois lignes: la première ligne contient le code des directions, la seconde et la troisième contiennent respectivement les coordonnées '`x`' et '`y`' des points du contour. La fin de la chaîne est signalée par la valeur -1 partout. L'image est automatiquement allouée pour contenir la chaîne, ce qui fait qu'il ne faut pas initialiser les champs de la structure-image qui sera donnée comme sortie.
Les directions sont encodées comme ceci:

```
a4   a3  a2
a5   a0  a1
a6   a7  a8
```

L'image d'entrée doit être différente de l'image de sortie. La chaîne est en 8-connexité.

Exemple: `(chain-code8 input out 5 6)`

Package: `2d`

`ring-chain8` *imin x y imout* Fonction

imin	Image d'entrée
x	Entier: coordonée '`x`' du point de départ de la chaîne Ce point appartient au contour de l'objet
y	Entier: coordonée '`y`' du point de départ de la chaîne Ce point appartient au contour de l'objet
imout	Image sortie: ne doit pas être initialisée

Description (2D): Cette fonction génère une chaîne en suivant l'intérieur d'un anneau sur une image binaire. On donne les coordonnées du pixel appartenant à l'intérieur de l'anneau sans toucher celui-ci , et la fonction retourne une image qui a trois lignes: la première ligne contient le code des directions, la seconde et la troisième contiennent respectivement les coordonnées '`x`' et '`y`' des points du contour. La fin de la chaîne est signalée par la valeur -1 partout. L'image est automatiquement allouée pour contenir la chaîne, ce qui fait qu'il ne faut pas initialiser les champs de la structure-image qui sera donnée comme sortie.
Les directions sont encodées comme ceci:

```
a4  a3  a2
a5  a0  a1
a6  a7  a8
```

L'image d'entrée doit être différente de l'image de sortie. La chaîne est en 8-connexité.

Exemple: `(ring-chain8 input out 5 6)`

Package: `2d`

Chapter 15

Analyse de textures et classification

15.1 Traitement de textures

`extend` *imin immask res coeff imout* Fonction

imin	Image source
immask	Image servant de masque
res	Nombre de résolutions dans la hiérarchie
coeff	Nombre de coefficients
imout	Image de sortie

Description (2D): Cette routine réalise une extrapolation de textures par le biais d'une déconvolution spectrale à plusieurs niveaux de résolution. L'image source est connue sur tous les pixels du masque différents de 0. Pour accélérer le traitement, on peut déconvoluer tous les signaux d'une décomposition sous-bande hiérarchique (nombre de résolutions ≥ 0). Pour effectuer une sélection des raies spectrales importantes sans déconvolution, il suffit de prendre un nombre de coefficients négatif. Les trois images (de taille identique) seront créées avant l'appel à cette routine. Limitation: la taille maximale d'une image pour l'extrapolation est fixée par les variables TV et TH (nécessairement égales) dans le fichier src/vandroog/extend.h. Un message d'erreur est renvoyé si la taille de l'image dépasse les limites autorisées.

Exemple: `(extend imin immask 2 500 imout)`

Auteur: Marc Van Droogenbroeck

Package: `vandroog`

Chapter 16

Un langage pour le traitement d'images

16.1 Commande du système d'exploitation

`cd` *directoryName* Fonction

directoryName	Chaîne de caractères : nouveau répertoire de travail

Description (2D): Cette fonction simplissime permet de changer de répertoire de travail, comme la commande "cd" du shell UNIX. La syntaxe est cependant différente.

Exemple: `(cd "/usr/users/guest/work")`

Auteur: Hugues Talbot

Package: `talbot`

`GetPid` Fonction

Description: Retourne le numéro de processus de `Xlim3d`.

Auteur: Christophe Gratin
Package: `xlispstat`

`system` *args* Fonction

args	Chaîne de caractères: commande

Description: Documenté ailleurs dans la documentation standard de XLISP. Lance une commande système.

Exemple:

```
(system "ls *.lsp") ;liste tous les fichiers xlisp.
```

Package: `xlispstat`

`chd` *directoryName* Fonction

directoryName	Chaîne de caractères

Description (3D): Change le répertoire courant.

Auteur: Christophe Gratin

Package: `3d`

`Ls` *&optional args* Fonction

args	Chaîne de caractères

Description (3D): Appel de la fonction ls.

Exemple: `(ls) ou (ls "*.lsp")`

Auteur: Christophe Gratin
Package: `3d`

`Pwd` Fonction

Description (3D): Affiche le répertoire courant.

Auteur: Christophe Gratin
Package: `3d`

`call-lfun` *"ListUndefined"* Fonction

Description (3D): Cette routine permet de connaître les fonctions qui ne sont pas définies après le chargement dynamique des librairies. La liste affichée reprend toutes les fonctions qui ont une "glue" mais pas le code correspondant dans une librairie "*.a" ou dans un fichier objet "*.o". Le mode de fonctionnement est lié au chargeur dynamique "dld".

Chapter 17

Non-classé

`ImDrawBall` *iminout xCenter yCenter zCenter radius value* Fonction

iminout	Image résultat
xCenter	Entier, coordonnée en X
yCenter	Entier, coordonnée en Y
zCenter	Entier, coordonnée en Z
radius	Entier, rayon
value	Réel

Description (3D): Trace une boule de centre et de rayon spécifiés à la valeur *value* dans l'image *iminout*.

Package: `3d`

`ImDrawBox` *iminout xStart yStart zStart xSize ySize zSize value* Fonction

iminout	Image résultat
xStart	Entier, coordonnée en X
yStart	Entier, coordonnée en Y
zStart	Entier, coordonnée en Z
xSize	Entier, taille en X
ySize	Entier, taille en Y
zSize	Entier, taille en Z
value	Réel

Description (3D): Trace un parallélépipède à la valeur *value* dans l'image *iminout*.

Package: `3d`

`ImUntangle` *imin* Fonction

imin	Image

Description (3D): Effectue une séparation des lignes paires et impaires de l'image *imin* qui est en trame entrelacée. Si *imin* est une image (X,Y,Z), l'image retournée sera (X, Y/2, 2Z) et contiendra les lignes paires de *imin* dans un plan et les impaires dans le plan suivant. La valeur renvoyée est la nouvelle image, une image désentrelacée.

Exemple: `(setf out (ImUntangle in)) ; out sera une image de taille X Y/2 2Z`

Auteur: Beatriz Marcotegui
Package: `3d`

`ImInterleave` *imin* Fonction

imin	Image

Description (3D): C'est l'opération inverse de ImUntangle. A partir d'une image (X,Y,Z), elle génère une image (X,2Y, Z/2) que contient comme lignes paires un plan de l'image *imin* et comme lignes impaires le plan suivant de *imin*. La valeur renvoyée est la nouvelle image, une image entrelacée.

Exemple: `(setf out (ImInterleave in)) ; out sera une image de taille X 2Y Z`

Auteur: Beatriz Marcotegui
Package: `3d`

Chapter 18

Les éléments structurants: classes et méthodes

18.1 La classe Sélément

18.1.1 Données

Pas de champ de données.

18.1.2 Méthodes

Pas de méthode spécifique.

18.2 La classe DecomposedSE

18.2.1 Données

- sEList: Liste des éléments structurants (noyaux) élémentaires.

18.2.2 Méthodes

`:SEList` Méthode sur `DecomposedSE`

SYNTAXE
(send *object* :SEList &optional *sEList*)

sEList	Liste de noyaux

Description (3D): Méthode d'accès au champ *sEList*. Si le paramètre optionnel *sEList* est spécifié, la valeur de ce paramètre est affectée au champ *sEList*. Si *sEList* est spécifié, retourne *object*, sinon retourne la valeur du champ.

Exemple: (setq se (send DecomposedSE :new '(((0.0 0.0 0.0) (1.0 0.0 0.0)) (0.0 0.0 0.0) (0.0 0.0 1.0)))))
(send se :SEList) ; retourne la valeur actuelle du champ
(send se :SEList '(((0.0 0.0 0.0) (1.0 0.0 0.0) (0.0 0.0 1.0) (1.0 0.0 1.0)))) ; change la valeur du champ
Package: `nil`

`:New` *sEList* Méthode sur `DecomposedSE`

SYNTAXE
(send DecomposedSE :New *sEList*)

sEList	Liste de noyaux

Description (3D): Retourne une instance de la classe initialisée avec sEList comme liste d'éléments structurants élémentaires.

Exemple: (setq se (send DecomposedSE :new '(((0.0 0.0 0.0) (1.0 0.0 0.0)) ((0.0 0.0 0.0) (0.0 0.0 1.0)))))
Package: `nil`

`:Copy` Méthode sur `DecomposedSE`

SYNTAXE
(send *object* :Copy)
Description (3D): Retourne une copie conforme de l'instance appelante (*object*).

Exemple: (setq se1 (send DecomposedSE :new '(((0.0 0.0 0.0) (1.0 0.0 0.0)) ((0.0 0.0 0.0) (0.0 0.0 1.0)))))
(setq se2 (send se1 :Copy)) ; un autre objet, identique à se1
Package: `nil`

`:Transpose` Méthode sur `DecomposedSE`

SYNTAXE
(send *object* :Transpose)

Description (3D): Transpose l'élément structurant (en prenant le symétrique de chaque élément structurant élémentaire). L'objet est modifié.

Exemple: (setq se (send DecomposedSE :new '(((0.0 0.0 0.0) (1.0 0.0 0.0)) (0.0 0.0 0.0) (0.0 0.0 1.0)))))
(send se :Transpose)
Package: `nil`

`:Erode` Méthode sur `DecomposedSE`

SYNTAXE
(send *object* :Erode *imout imin*)

imout	Image résultat
imin	Image

Description (3D): Erode l'image *imin* par l'élément structurant *object*. Résultat dans *imout*. Retourne *object*.

Exemple: `(send se :Erode n2 n1)`

Package: `nil`

`:Dilate` *imout imin* Méthode sur `DecomposedSE`

SYNTAXE
(send *object* :Dilate *imout imin*)

imout	Image résultat
imin	Image

Description (3D): Dilate l'image *imin* par l'élément structurant *object*. Résultat dans *imout*. Retourne *object*.

Exemple: `(send se :Dilate n2 n1)`

Package: `nil`

`:GeoDilate` Méthode sur `DecomposedSE`

SYNTAXE
(send *object* :GeoDilate *imout imin immask*)

imout	Image résultat
imin	Image
immask	Image

Description (3D): Dilate l'image *imin* par l'élément structurant *object* dans le masque géodésique *immask*. Résultat dans *imout*. Retourne *object*.

Exemple: `(send se :Dilate n2 n1)`

Package: `nil`

18.3 La classe HomoteticSE

18.3.1 Données

- baseKernel: Noyau de base.

- size: Nombre de fois qu'il est additionné avec lui-même.

18.3.2 Méthodes

Cette classe hérite bien entendu de toutes les méthodes de la classe *DecomposedSE*. Certaines methodes ont ete redéfinies mais la documentation de la méthode de la classe mère s'applique

encore. Elles ne sont donc pas documentées une deuxième fois.

`:BaseKernel` Méthode sur `HomoteticSE`

SYNTAXE
(send *object* :BaseKernel &optional *baseKernel*)

baseKernel	Liste: noyau de base

Description (3D): Méthode d'accès au champ *baseKernel*. Si le paramètre optionnel *baseKernel* est spécifié, la valeur de ce paramètre est affectée au champ *baseKernel*. Si *baseKernel* est spécifié, retourne *object*, sinon retourne la valeur du champ.

Exemple:

```
(setq se (send HomoteticSE :new diamondK 6)) ; un "diamant" de taille 6
(send se :BaseKernel) ; retourne la valeur actuelle du champ
(send se :SEList squareK) ; change la valeur du champ $-->$,
                            se est maintenant un carre de taille 6
```

Package: `nil`

`:Size` Méthode sur `HomoteticSE`

SYNTAXE
(send *object* :Size &optional *size*)

size	Liste: noyau de base

Description (3D): Méthode d'accès au champ *size*. Si le paramètre optionnel *size* est spécifié, la valeur de ce paramètre est affectée au champ *size*. Si *size* est spécifié, retourne *object*, sinon retourne la valeur du champ.

Exemple:

```
(setq se (send HomoteticSE :new diamondK 6)) ; un "diamant" de taille 6
(send se :Size) ; retourne la valeur actuelle du champ
(send se :Size 10) ; change la valeur du champ $-->$ se,
                     est maintenant un diamant de taille 10
```

Package: `nil`

`:New` Méthode sur `HomoteticSE`

SYNTAXE
(send HomoteticSE :new *baseKernel size*)

baseKernel	Liste: noyau de base
size	Entier

Description (3D): Retourne une instance de la classe initialisée avec *baseKernel* comme noyau de base et de taille *size*.

Exemple:

```
(setq se (send HomoteticSE :new hexagonK 6)) & un hexagone de taille 6
(send se :Dilate n2 n1) & dilatation par l'hexagone de taille 6 se
```

Package: `nil`

18.4 La classe ImageSE

18.4.1 Données

- image: image de la forme binaire prise comme élément structurant,
- center: liste des coordonnées du centre de l'élément structurant.

18.4.2 Méthodes

`:Image` Méthode sur `ImageSE`

SYNTAXE
(send *object* :Image &optional *image*)

image	Image

Description (3D): Méthode d'accès au champ *image*. Si le paramètre optionnel *image* est spécifié, la valeur de ce paramètre est affectée au champ *image*. Si *image* est spécifié, retourne *object*, sinon retourne la valeur du champ.

Exemple: `(ImXv (send se :Image))  ; visualisation de l'élément structurant`

Package: `nil`

`:Center` Méthode sur `ImageSE`

SYNTAXE
(send *object* :Center &optional *center*)

center	Liste de noyaux

Description (3D): Méthode d'accès au champ *center*. Si le paramètre optionnel *center* est spécifié, la valeur de ce paramètre est affectée au champ *center*. Si *center* est spécifié, retourne *object*, sinon retourne la valeur du champ.

Exemple: `(send se :Center '(40 0 50))  ; modification de l'origine de l'élément structurant se`

Package: `nil`

`:New` Méthode sur `ImageSE`

SYNTAXE
(send ImageSE :new *image center*)

image	Image
center	Liste de trois entiers

Description (3D): Retourne une instance de la classe initialisée avec *image* comme image contenant la forme binaire prise comme élément structurant. La forme binaire doit être à la valeur 1 dans *image* et le reste à 0. *center* est un triplet d'entiers définissant la position du centre de l'élément structurant. Le point ainsi désigne doit se trouver à l'intérieur de l'image *image*.

Exemple:

```
(setq seIm (ImGet2D 64 64)) ; allouer une image pour y dessiner l'es
(ImDrawBall seIm 32 0 32 20 1) ; dessiner un disque
(setq se (send ImageSE :new seIm '(32 0 32))) ; le centre de l'élément
              structurant au centre du disque (très original !)
```

Package: `nil`

`:Copy` Méthode sur `ImageSE`

SYNTAXE
(send *object* :Copy)

Description (3D): Retourne une copie conforme de l'instance appelante.

Exemple: `(setq se2 (send se1 :Copy)) ; un autre objet, identique à se1`

Package: `nil`

`:Transpose` Méthode sur `ImageSE`

SYNTAXE
(send *object* :Transpose)

Description (3D): Transpose l'élément structurant (en prenant le symétrique de chaque élément structurant élémentaire). L'objet est modifié.

Exemple:
```
(send se2 (send se1 :Copy))
(send se2 :Transpose)
```
Package: `nil`

`:Erode` Méthode sur `ImageSE`

SYNTAXE
(send *object* :Erode *imout imin*)

imout	Image résultat
imin	Image

Description (3D): Erode l'image *imin* par l'élément structurant *object*. Résultat dans *imout*. Retourne *object*.

Exemple:
```
(send se :Erode n2 n1)
```
Package: `nil`

`:Dilate` Méthode sur `ImageSE`

SYNTAXE
(send *object* :Dilate *imout imin*)

imout	Image résultat
imin	Image

Description (3D): Dilate l'image *imin* par l'élément structurant *object*. Résultat dans *imout*. Retourne *object*.

Exemple:
```
(send se :Dilate n2 n1)
```
Package: `nil`

Chapter 19

Les graphes: nouvelle structure dans XLispStat

19.1 Les fonctions sur les graphes

19.1.1 Création et destruction de graphes

`grMake` *imMosaic imEdgeVal* Fonction

imMosaic	Image mosaïque
imEdgeVal	Image servant à valuer les arètes

Description (2D): Cette fonction prend deux images comme entrée. La première est une image mosaïque qui permet de décrire les relations de voisinage et la seconde de valuer les arètes. La sortie est un graphe. En clair, dans le graphe résultat seront voisins deux bassins versants qui auront au moins une ligne de partage des eaux en commun (en 8-connexité). La valuation des sommets correspondante sera celle de l'intérieur de chaque bassin versant, et la valuation de l'arête correspondante sera le *minimum* dans l'image *imEdgeVal* le long de l'arête de ligne de partage des eaux commune. Le graphe de sortie est nécessairement une triangulation de Delaunay.

Exemple: `(setf g0 (grMake imMosa imGrad)) ; construction du graphe`

Ou encore:

```
(defun grimmalloc (imin &optional (debug ()))
    (let (imiii0 res)
        (setf imiii0 (timmalloc imin))
        (format t "Getting the mosaic image\n")
        (roberts imin imiii0)
        (mosaic8 imiii0 imin imiii0 "a")
        (setf res (grmake imiii0 imin))
        (if debug
            (imcopy imiii0 debug)
            (progn
                (setf gimdisplay (timmalloc imiii0))
                (imcopy imiii0 gimdisplay)
            )
        )
        (imfree imiii0)
        res
    )
)
```

Cette fonction permet de construire un graphe à partir d'une image standard.

Auteur: Hugues Talbot
Package: `graphs`

`gDup` *graphin* Fonction

graphin	Graphe d'entrée prototype

Description (2D): Cette fonction duplique le graphe d'entrée *graphin* et retourne un nouveau graphe, ayant la même structure, les même valuation d'arêtes et de sommets. Une véritable duplication est effectuée.

Exemple: `(setf g1 (gDup g0)) ; duplique le graphe g0`

Auteur: Hugues Talbot
Package: `graphs`

`gDupL` *graphin* Fonction

graphin	Graphe d'entrée prototype

Description (2D): Cette fonction duplique le graphe d'entrée *graphin* et retourne un nouveau graphe, ayant la même structure, les même valuation d'arêtes et de sommets. Cette fonction n'opère pas une véritable duplication, seules les valuations des arêtes et des sommets sont effectivement dupliquées, la structure du graphe (voisinages) n'est que liée au graphe prototype. Si les relations de voisinages changent dans *graphin*, alors elles changent aussi dans le graphe résultat. Ceci permet pas mal de souplesse, et surtout permet de gagner beaucoup de mémoire.

Exemple: `(setf g1 (gDupL g0)) ; duplique le graphe g0`

Auteur: Hugues Talbot
Package: `graphs`

`gCopy` *graphin graphout* Fonction

graphin	Graphe d'entrée
graphout	Graphe de sortie

Description (2D): Cette fonction copie *graphin* dans *graphout*.

Exemple: `(gcopy g0 g1) ; copie de g0 dans g1`

Auteur: Hugues Talbot
Package: `graphs`

`gFree` *graphin* Fonction

graphin	Graphe à libérer

Description (2D): Cette fonction libère la mémoire occupée par le graphe *graphin*. Celui-ci devient inaccessible par la suite, bien sûr.

Exemple: `(gFree g0) ; libère le graphe g0`

Auteur: Hugues Talbot
Package: `graphs`

19.1.2 Afficher les graphes

`imMakeFG` *imProto graphin* Fonction

imProto	Une image
graphin	Graphe à afficher

Description (2D): Cette fonction permet de remplir une image contenant un graphe résultat. L'image donnée doit avoir les mêmes dimensions que l'image mosaïque ayant servi à sa création. Au retour, elle contient 0 partout sauf un point par sommet, mis à la valeur du sommet en question. La position de chaque sommet est telle que l'intersection avec les bassins versants d'origine n'est pas nulle. Ceci permet de reconstruire une image mosaïque grâce à la fonction `propa8`.

Exemple: `(imMakeFG imSameOrig g0) ; remplit une image résultat`

```
; Voici un exemple de fonction de visualisation:
(defun gdisp (gin screen &key (graphe ()) (noedge ()) )
    (let (imiii0 thegraph)
        (if (and (not graphe) (not gimdisplay))
            (format t "No can do, man: no reference image given\n")
            (progn
                (if (not graphe)
                    (setf graphe gimdisplay))
                (setf imiii0 (timmalloc graphe))
                (setf thegraph (timmalloc graphe))
                (immakefg imiii0 gin)
                (imcopy graphe thegraph) ; graphe must remain untouched
                (propa8 thegraph imiii0 "x")
                (if noedge (dil8 thegraph thegraph 1))
                (disp thegraph screen)
                (imfree imiii0)
                (imfree thegraph)
            )
        )
    )
    graphe  ; returned value (it's an image)
)
```

Auteur: Hugues Talbot
Package: `graphs`

19.1.3 Arithmétique sur les graphes

`gArith` *graphin Operand Operation grResult* Fonction

graphin	Graphe d'entrée
Operand	Soit un graphe, soit une constante
Operation	Chaîne de caractère décrivant l'opération
grResult	Graphe résultat

Description (2D): Cette fonction opère *Operation* sur *graphin* et *Operand*. *Operand* peut être soit un graphe soit une constante entière signée. *grResult* est le graphe dans lequel l'opération est effectuée. Tous les graphes peuvent être différents ou non, mais leur structure de sommet doit être la même; *seule la valuation des sommets est prise en compte.*
Operand peut être :

- `"+"` : addition.
- `"-"` : soustraction.
- `"*"` : multiplication.
- `"/"` : division entière.
- `"%"` : modulo.
- `"&"` : ET logique (bit par bit).
- `"|"` : OU logique (bit par bit).
- `"^` : OU exclusif logique (bit par bit).

Exemple:

```
(gArith g0 g1 "+" g1)  ; additionne g0 et g1 dans g1
(gArith g0 12 "-" g1)  ; retranche 12 à g0 dans g1
```

Auteur: Hugues Talbot
Package: `graphs`

19.1.4 Comparaison de deux graphes

`gComp` *graphin Operand Operation grResult valueResYes valueResNo* Fonction

graphin	Graphe d'entrée
Operand	Soit un graphe, soit une constante
Operation	Chaîne de caractère décrivant l' opération
grResult	Graphe résultat
valueResYes	Valeur à mettre dans *grResult* si la comparaison est VRAIE
valueResNo	Valeur à mettre dans *grResult* si la comparaison est FAUSSE

Description (2D): Cette fonction compare *graphin* et *Operand*. *Operand* peut être soit un graphe soit une constante entière signée. *grResult* est le graphe dans lequel la comparaison est effectuée. *valueResYes* et *valueResNo* peuvent aussi être soit des graphes soit des constantes entières signées. Tous les graphes peuvent être différents ou non, mais bien sûr compatibles en structure. *Seule la valuation des sommets est prise en compte.* *Operand* peut être :

`"=="` : test d'égalité.
`"!="` : test de différence.
`">="` : supérieur ou égal.
`"<="` : inférieur ou égal.
`">"` : strictement supérieur.
`"<"` : strictement inférieur.

Exemple:

```
(gComp g0 g1 "!=" g1 255 0) ; compare g0 et g1 dans g1, binarise
à 255 et 0 (gArith g0 g1 "<" g1 g3 g4) ; toggle-mapping
```

Auteur: Hugues Talbot
Package: `graphs`

19.1.5 Information sur les graphes

`gInfo` *graphin* Fonction

graphin	Graphe sur lequel on veut avoir des informations

Description (2D): Cette fonction donne quelques informations minimales sur le graphe d'entrée: nombre de sommets, d'arêtes...

Exemple: `(gInfo g0) ; décrit g0`

Auteur: Hugues Talbot
Package: `graphs`

`gPeek` *graphin VerticeNb* Fonction

graphin	Graphe sur lequel on veut avoir des informations
VerticeNb	Constante entière positive donnant le numéro du sommet auquel on est intéressé

Description (2D): Cette fonction permet d'avoir des informations sur un sommet du graphe donné: nombre de voisins, valuation de ces voisins, valuation des arêtes...

Exemple: `(gPeek g0 1000) ; informe sur le sommet #1000 de g0`

Auteur: Hugues Talbot
Package: `graphs`

`gMinval` *graphin* Fonction

graphin	Graphe d'entrée

Description (2D): Cette fonction permet de connaître la valuation minimale parmi les sommets de *graphin*.

Exemple: `(gMinval g0) ; Donne la valeur minimale de g0`

Auteur: Hugues Talbot
Package: `graphs`

`gMaxval` *graphin* Fonction

graphin	Graphe d'entrée

Description (2D): Cette fonction permet de connaître la valuation maximale parmi les sommets de *graphin*.

Exemple: `(gMaxval g0) ; Donne la valeur maximale de g0`

Auteur: Hugues Talbot
Package: `graphs`

19.1.6 Opérations morphologiques

`gDil` *graphin graphout size* Fonction

graphin	Graphe d'entrée
graphout	Graphe de sortie
size	Nombre d'itérations

Description (2D): Cette fonction réalise la dilatation de *graphin* dans *graphout*. *graphin* et *graphout* peuvent en fait être les mêmes.

Exemple: `(gDil g0 g0 1) ; Dilatation de g0 dans g0 de taille 1`

Auteur: Hugues Talbot
Package: `graphs`

`gEro` *graphin graphout size* Fonction

graphin	Graphe d'entrée
graphout	Graphe de sortie
size	Nombre d'itérations

Description (2D): Cette fonction réalise l'érosion de *graphin* dans *graphout*. *graphin* et *graphout* peuvent en fait être les mêmes.

Exemple: `(gEro g0 g0 1) ; Erosion de g0 dans g0 de taille 1`

Auteur: Hugues Talbot
Package: `graphs`

`gRecons` *graphout graphMask* Fonction

graphout	Graphe d'entrée reconstruit
graphMask	Graphe masque

Description (2D): Cette fonction réalise l'opération de reconstruction numérique par dilatation géodesique de *graphout* dans lui-même, *graphMask* étant le masque. Permet de réaliser la reconstruction binaire, bien sûr. Notez bien que le graphe d'entrée est modifié.

Exemple: `(gRecons g0 gmask) ; reconstruction numérique de g0 sous gmask`

Auteur: Hugues Talbot
Package: `graphs`

19.1.7 Opération linéaires

`glzc` *graphin graphout* Fonction

graphin	Graphe d'entrée
graphout	Graphe de sortie

Description (2D): Cette fonction effectue l'opération dite de "Laplacian Zero Crossing". C'est un Laplacien sur graphe. *graphout* contient seulement les valeurs positives.

Exemple: `(glzc g0 gout) ; Laplacien`

Auteur: Hugues Talbot
Package: `graphs`

19.1.8 Marquage d'un nœud du graphe

`gmark` *graphin graphout number* Fonction

graphin	Input graph
graphout	Output graph
number	Number of vertice to mark

Description (2D): Marks a vertice in the input graph. The output graph is identical to the input graph but has an extra vertice which points to the marked vertice. This extra vertice has no neighborhood but has a value equal to that of the given number.

Auteur: Ronald Jones
Package: `jones`

`gunmark` *graphin graphout number* Fonction

graphin	Input graph
graphout	Output graph
number	Number of vertice to remove mark from

Description (2D): Takes the mark off a vertice in the input graph. The output graph is identical to the input graph but has one less vertice as the vertice which carried the mark is removed.

Auteur: Ronald Jones
Package: `jones`

19.1.9 Graphes et arbres

`gmst` *graphin graphout* Fonction

graphin	Input graph
graphout	Output graph

Description (2D): Computes the minimum spanning tree of the input graph using Prim's algorithm. The input graph must be connected with or without a marked vertice or disconnected with a marked vertice in each connected sub-graph. The identity of the output graph is set to the value 5.

Auteur: Ronald Jones
Package: `jones`

`gtom` *graphin graphout* Fonction

graphin	Input graph
graphout	Output graph

Description (2D): Computes the tree of minima of a graph using the 'retreat of the sea' algorithm. The input graph must be a minimum spanning tree (with identity set to the value 5) with no marked vertices. The output graph has identity set to the value 6.

Auteur: Ronald Jones
Package: `jones`

`gdyn` *graphin graphout* Fonction

graphin	Input graph
graphout	Output graph

Description (2D): Computes the dynamics for the vertices in the input graph. The leaves are given dynamic values and the lakes are given depth values. The input graph must be a tree of minima (with identity set to the value 6) with no marked vertices. The output graph has identity set to the value 7.

Auteur: Ronald Jones
Package: `jones`

Part III

Race Project

Chapter 20

Function description

20.1 General purpose and system

`file_size` *filename* Fonction

filename	Name of the file on disk

Description (2D): Measure the size in bits of a file on disk.
Package: `upc`

`help` *&optional string* Fonction

string	Name of a function

Description (2D): Print the description of a function. If no parameter is given, it prints a list of all functions. If the transformation ends with a number, do NOT put it; ex: for help on `ero` type (help "ero").
Package: `nil`

20.2 Read & write images

`picread` *string* Fonction

string	Name of the file containing the image

Description (2D): Read an image on disk in pic format. Since it is in a pic format, the image is assumed to be squared.
Package: `upc`

`picwrite` *ima string* Fonction

ima	Image to write
string	Name of the file to be created

Description (2D): Write an image on disk in pic format.
Package: `upc`

20.3 Visualization and general image manipulation

`downsample` *imin imout* Fonction

imin	Input image
imout	Output image

Description (2D): Downsample by a factor of two in horizontal and vertical.
Package: `upc`

`fillreg` *imori imlabel imout model* Fonction

imori	Reference image
imlabel	Label image
imout	Output image
model	Model to be used to fill the regions

Description (2D): Fill each region of the image identified by the label image by a given *model* estimated on the reference image.
The possible *model*s are:
"poly0" Zero order polynomial (MSE optimization)
"poly1" First order polynomial (MSE optimization)
"poly2" Second order polynomial (MSE optimization)
"sqrt_energy" Square root of the energy
Package: `upc`

`upsample` *imin imout value* Fonction

imin	Input image
imout	Output image
value	Value to set the new points of the grid

Description (2D): Upsampling of an image. The new points of the grid are set to *value*.
Package: `upc`

20.4 Filters

`arope4,8` *imin imout area* Fonction

imin	Input image
imout	Output image
area	Area in number of pixels

Description (2D): Area opening: elimination of all bright components whose area is smaller than the parameter *area*. 4 and 8 connectivity are available.
Package: `upc`

`arclo4,8` *imin imout area* Fonction

imin	Input image
imout	Output image
area	Area in number of pixels

Description (2D): Area closing: elimination of all dark components whose area is smaller than the parameter *area*. 4 and 8 connectivity are available.
Package: `upc`

20.5 Reconstruction

`dreconsdir` *immarkout imref dir* Fonction

immarkout	Marker and output image
imref	Reference image
dir	Direction of reconstruction: 1,2,3,4

Description (2D): Dual reconstruction of binary & grey-level images. The first image (marker) is propagated by geodesic erosion into the second one (reference). The reconstruction is done in a direction defined by *dir*: 1=Horizontal, 2=Diagonal 45, 3=Vertical, 4=Diagonal 135.
Package: `upc`

`reconsdir` *immarout imref dir* Fonction

immarout	Marker and output image
imref	Reference image
dir	Direction of reconstruction: 1,2,3,4

Description (2D): Reconstruction of binary & grey-level images. The first image (marker) is propagated by geodesic dilation into the second one (reference). The reconstruction is done in a direction defined by dir: 1=Horizontal, 2=Diagonal 45, 3=Vertical, 4=Diagonal 135.
Package: `upc`

20.6 Skeleton

`anchor4,6,8` *imin imout thres* Fonction

imin	Binary input image
imout	Output image
thres	Threshlod

Description (2D): Anchor points of a binary image, in 4,6 or 8 connectivity, for doing its skeleton (i.e. local maxima of the distance function) by using `skelh`. The "anchor point" image is thresholded at *thres* level (only those points which are >= *thres* will be kept), so that, depending on its value, different skeletons will be obtained:

$= 0$: classical skeletons,

> 0 : smoothed skeletons.

Package: `upc`

`conrgwsh4` *imlabel imori imcon* Fonction

imlabel	Input/output label image
imori	Original image
imcon	Label image defining the constraints

Description (2D): Segment *imori* by using the label defined by *imlabel*. *imcon* is used to constrain the result by a previous segmentation.
Package: `upc`

`hhmtr` *imin imlut imout iter* Fonction

imin	Input image, binary
imlut	Lut image
imout	Output image
iter	Iterations

Description (2D): Hit or miss transform on hexagonal grid. This operation looks for particular pixel configurations in the binary input image. Similar to hhmt but each structuring is rotated to perform efficiently sequential thinning.
Package: `upc`

`rgwsh4,8` *imlabout imori* Fonction

imlabout	Input label and output image
imori	Original image

Description (2D): Region growing formulation of the watershed. The algorithm segment the objects of *imori* that have been identified by *imlabout*. The result is in *imlabout*. The propagation can be done in 4 and 8 connectivity.
Package: `upc`

`shmtr` *imin imlut imout iter* Fonction

imin	Input image, binary
imlut	Lut image
imout	Output image
iter	Iterations

Description (2D): Hit or miss transform on square grid. This operation looks for particular pixel configurations in the binary input image. Similar to `shmt` but each structuring is rotated to perform efficiently sequential thinning.
Package: `upc`

`skelh` *imin imanc imout* Fonction

imin	Input image, binary
imanc	Image containing anchor points
imout	Output image

Description (2D): Skeleton of *imin* (in 6-connectivity) by using anchor points (in 6-connectivity). Depending on the "anchor point" image, different kinds of skeletons can be obtained see Luc Vincent PhD. Thesis). For instance:

max(dist(*imin*)): classical skeletons; this anchor points can be obtained with `anchor6` and $thres = 0$.

threshold(quench(*imin*)): smoothed skeleton, depending on threshold level; this anchor points can be obtained with `anchor6` and $thres > 0$.

last_ero(*imin*): minim skeleton.

none: homotopic markers of the different sets in an image.

...

Package: `upc`

`skels` *imin imanc imout* Fonction

imin	Input image, binary
imanc	Image containing anchor points
imout	Output image

Description (2D): Skeleton of *imin* (in 8-connectivity) by using anchor points (in 4 or 8-connectivity). Depending on the "anchor point" image, different kinds of skeletons can be obtained (see Luc Vincent PhD. Thesis). For instance:

max(dist(*imin*)): classical skeletons; this anchor points can be obtained with `anchor4,8` and $thres = 0$.

threshold(quench(*imin*)): smoothed skeleton, depending on threshold level; this anchor points can be obtained with `anchor4,8` and $thres > 0$.

last_ero(*imin*): minim skeleton.

none: homotopic markers of the different sets in an image.

...

Package: `upc`

`write_quench` *imin buffer* Fonction

imin	Grey-level image (skeleton)
buffer	Single line image (output)

Description (2D): It scans *imin* from left to right, up to down and write each non zero pixel of *imin* in *buffer*. *buffer* has to be allocated but not initialised before.
Package: `upc`

20.7 Labeling

`conlabelfl4` *imin imlabel imout size tolerance* Fonction

imin	Input image
imlabel	Label image defining the constraints
imout	Output image
size	Size limit of the flat zone to be labeled
tolerance	Grey-level tolerance on the flatness

Description (2D): Labels the flat zones of *imin* using *imlabel* as constraint (the labelling process will not allow any crossing of boundaries defined by *imlabel*)
Package: `upc`

`conmarlabelfl4` *imin imark imlabel imout* Fonction

imin	Input image
imark	Marker image
imlabel	Label image defining the constraints
imout	Output image

Description (2D): Labels the flat zones of *imin* that are marked by *imark* using *imlabel* as constraint (the labelling process will not allow any crossing of boundaries defined by *imlabel*)

Package: `upc`

`labelfl4` *imin imout size tolerance* Fonction

imin	Input image
imout	Output image
size	Size limit fo the labelled zones
tolerance	Flatness tolerance

Description (2D): Labelling of flat zones of *imin*. Only the flat zones of size larger than *size* are labelled (if *size*=0 all flat zones are labelled). Non labelled flat zones are merged and the label zero is assigned to them. A flat zone is the largest connected component of quasi constant grey level value. The meaning of "quasi" depends on the tolerance parameter (Strict flat zones => *tolerance* = 0)

Package: `upc`

`marlabelfl4` *iminimark imout* Fonction

imin	Input image
imark	Marker image
imout	Output image

Description (2D): Labels the flat zones of *imin* that are marked by *imark*.

Package: `upc`

`maxloc4,6,8` *imin imout* Fonction

imin	Input image
imout	Output image

Description (2D): Local maxima (points >= all their nearest_maxloc6_neighbours) of *imin* in *imout* (6-connectivity). Original grey values are kept.

Package: `upc`

20.8 Contour

`cont_buff` *imcont buffer1 buffer2* Fonction

imcont	Contour image
buffer1	Buffer with the set of movements
buffer2	Buffer with the set of starting points

Description (2D): Performs the coding of the contour image by means of chain code and triple points. The set of movements is stored in *buffer1* and the triple points marking the position of each cluster in *buffer2*. Returns the number of clusters in the image.
Package: `upc`

`cont_label` *imcont imlabel* Fonction

imcont	Contour image
imlabel	Image of labels

Description (2D): Creates a label image from a contour image. The label #0 is not used in the labelling. The function returns the number of regions in the labelled image.
Package: `upc`

`label_cont` *imlabel imcont* Fonction

imlabel	Image of labels
imcont	Contour image

Description (2D): Creates a contour image from a label image. The contour image has size (2N+1)(2N+1) being N the size of the *imlabel* image. Returns the number of contour points in the image.
Package: `upc`

`polygon` *imin imlabel dataFilename polOrder* Fonction

imin	Original image
imlabel	Original label image resulting from a previous segmentation stage (label range is [1 ... nb_region])
dataFilename	Name of the file which gets the bitstream
polOrder	String specifying the order of the polynomial for texture approximation : "poly0" "poly1"

Description (2D): Contour coding by polygonal approximation.
The general principle is to transform original regions into polygons whose shape can be simply defined by a set of control points. The contour transformation is mastered by an error criterion, assuming a polynomial approximation of the luminance within the regions. In addition, the original topology of the segmentation is preserved. The resulting polygons are subsequently defined in a non redundant manner by transmitting the control points lying on their left sides only. Each polygon can be made of several left sides since they are not necessarily convex. The first point of each left side could be addressed by its absolute coordinates, but it is actually defined by a so-called absolute vector pointing from the previous transmitted left side. The next control points within a given left side are addressed by so-called relative vectors pointing from the previous control points. Variable length entropic coders have been designed for both kinds of vectors. Statistics (probability of vector occurrence) have been computed on a representative set of images of constant size (352x288). Huffman tables files are: abs_x.huf, abs_y.huf, rel_x.huf, rel_y.huf
Package: `lep`

`x_polygon` *dataFilename imout* Fonction

dataFilename	Name of the file containing the bitstream
imout	Label image reconstructed after decoding

Description (2D): Decoding of the bitstream created by function `polygon` (contour coding by polygonal approximation).
The left sides of the polygons are described by absolute and relative vectors read from the bitstream file and written in output buffer associated to a label number. Once the buffer contains the left sides, it is scan downwards from left to right, spreading the label numbers and updating their value everytime a new label is found.
Package: `lep`

20.9 Coding

`vq_sample` *imin imlabel dataFilename sampleSize blockSize* Fonction

imin	Original image
imlabel	Original label image
dataFilename	Name of the file which gets the bitstream
sampleSize	Size of the squared sample which is extracted in each textured region
blockSize	Size of the blocks which are actually the codewords of the VQ dictionnary (codebook). (sampleSize-blockSize+1)*(sampleSize-blockSize+1) is the size of the codebook

Description (2D): Microscopic texture coding by vector quantisation using a texture sample to generate the codebook.
Only the microscopic textured regions (tiled roof, brick wall) are labelled, starting from label 1. Areas of slowly varying luminance are labelled 0. Thus, it is assumed that a function has previouly detected these 2 kinds of regions and performed a new labelling of the textured regions. A texture sample of size *sampleSize* $\times$ *sampleSize* is extracted from each texture and transmitted. All possible blocks of size *blockSize* $\times$ *blockSize* (with *blockSize* $<$ *sampleSize*) are extracted from the sample to build the codebook of the vector quantization. In the coding procedure, each *blockSize* $\times$ *blockSize* block of the textured area is compared to the blocks of the codebook; the address of the most similar block is simply transmitted to the decoder. In addition, the sample is compressed by a DCT based scheme for transmission. Regions which are not large enough to allow the extraction of an entire sample are not encoded.
Package: `lep`

`x_vq_sample` *imlabel dataFilename imout* Fonction

imlabel	Label image specifying the textured regions shapes
dataFilename	Name of the file containing the bitstream
imout	Image in which the textures will be reconstructed by x_vq_sample

Description (2D): Decoding of the bitstream created by function `vq_sample` (microscopic texture coding).
For each specified region, a sample is read from the bitstream, decompressed and used to generate a codebook. each block of the region is reconstructed by taking the codeword (actually a block of the sample) pointed by the address read from the bitstream.
Package: `lep`

`csvq` *imin imlab immap cod bufvar buffilt bufindex bufmean cwn cwx cwy hx hy bits* Fonction

imin	Input image
imlab	Image of labels
immap	Map image of pixels to process
cod	Output coded image
bufvar	Buffer of coded variance factors
buffilt	Buffer of coded filter coefficients
bufindex	Buffer of codebook indexes of blocks
bufmean	Buffer of coded block means
cwn	Number of codewords for codebook
cwx	Horizontal dimension of codeword
cwy	Vertical dimension of codeword
hx	Horizontal dimension of AR filter
hy	Vertical dimension of AR filter
bits	Number of bits used by SVQ codification

Description (2D): `csvq` codes each region of input image *imin*, applying a stochastic codebook to each region (defined by the labels image). This codebook is generated filtering images of gaussian white noise with a filter obtained from the AR analysis of the region.
Package: `upc`

`ortho_basis2` *imori imlabel immap imout bufout type order QQ dpc* Fonction

imori	Image to code
imlabel	Label image
immap	Map image
imout	Output image
bufout	Buffer of symbols of coefficients
type	Model to be used to fill the regions: "pol_ort" or "cos_ort".
order	Order of the model (0,1,2,3,4,...)
QQ	Coeff. quantification ("y" / "n")
dpc	Quantification step

Description (2D): Fill each region of the image identified by the label image by a given model estimated on the reference image. The possible models are:

"pol_ort" orthogonal polynomial (MSE optimization)
"cos_ort" orthogonal harmonics (MSE optimization)

The model coefficients can be quantificated with aquantilication step *dpc*.The number of coefficients is $=(order+1)^2$. The region where map is zero are not filled.
Package: `upc`

`overlap_rec` *imin imout* Fonction

imin	Max-balls skeleton image
imout	Multilevel reconstruction

Description (2D): Dilate each point of the skeleton (8-C), so that the overlaping balls sum. Gives a grey level reconstructed image instead of a binary image. Used to find redundant points in skeletons (points which area of dilation is covered by the dilations of other points)
Package: `upc`

`rm_skel_redun` *imoverlap skel minskel buf* Fonction

imoverlap	Overlap image, multilevel reconstruction (input)
skel	Clasical max-balls skeleton (input)
minskel	Minimal skeleton (output)
buf	Single line image with values of the quench function of *minskel* (output)

Description (2D): Using the multilevel reconstruction of the image (can be obtained with `overlap_rec`), it removes in the clasical skeleton the redundant points which representation area is covered by the dilation of the other points (of the minimal skeleton). The values of the quench function are written in a buffer = a single line image.
Package: `upc`

20.10 Entropy coding

`carith` *buffer filename size* Fonction

buffer	Buffer image
filename	Filename
size	Size of alphabet

Description (2D): Arithmetic coding of the *buffer*. The coding measures the pdf of the source and writes the bit sequence in *filename*.bit and the pdf in *filename*.pdf. The parameter *size* defines the size of the alphabet. The *buffer* should be full (its size defines the symbol sequence length).
Package: `upc`

`celias` *imin fileout* Fonction

imin	Input image to code
fileout	Compacted file

Description (2D): Return size in bits of the created file on disk.
Codes the positions of points in an image, by counting the number of 0 between two points and coding it in a ternary digits system, then it is coverted in bits. suitable for sparse images (skeletons...)
Package: `upc`

`chuffman` *buffer filename size* Fonction

buffer	1D image containing the set of symbols
filename	File where the coding is to be stored
size	Size of the alphabet

Description (2D): Performs the Huffman coding of the set of symbols contained in *buffer*. It produces to files. *Filename*.pdf contains the pdf of these symbols and *filename*.bit the bit stream. It returns the size in bits of *filename*.bit.
Package: `upc`

`crl2D` *bin_in bufout* Fonction

bin_in	Binary input image
bufout	Output buffer

Description (2D): Two-dimensional run-length coding of binary images. The output buffer must be an image structure and the program will allocate it. How to create an image structure: `(setf buf (immalloc 0 0 "s")`
Package: `upc`

`darith` *filename size buffer* Fonction

buffer	Buffer image
filename	Filename
size	Size of alphabet

Description (2D): Arithmetic decoding of *filename*. The decoding reads the bit sequence in *filename*.bit and the pdf in *filename*.pdf. The parameter *size* defines the size of the alphabet. The *buffer* should be allocated.
Package: `upc`

`delias` *imout filename* Fonction

imout	Binary image of the positions of points in coded image
filename	Elias code file

Description (2D): Decodes a file written in Elias code. The output is a binary image corresponding to the points of the coded image. *imout* must be allocated and initalised before.
Package: `upc`

20.11 Buffer manipulation

`group_symbol` *bufin size group bufout* Fonction

bufin	Input buffer
size	Size of the input alphabet
group	Number of symbols to group together
bufout	Output buffer

Description (2D): Group the symbols contained in *bufin* and write them in *bufout*. The size of the input alphabet is *sizealin*, they are grouped by *group* number. The function returns the size of the output alphabet: (*size*) exp *group*.
Package: `upc`

`image_to_buffer` *imin seg map buf* Fonction

imin	Input image
seg	Label image
map	Label image
buf	Image buffer (one line image)

Description (2D): Write into a buffer, that is an image of one line, the pixels of *imin*. The scanning of *imin* is defined by *map* and *seg* which are two label images: Each region where *map* is equal to 0 is skipped. The pixels of a given region defined by *seg* are written one after the other in the buffer.
Package: `upc`

20.12 FLIP, curve manipulation

`append-curve` *filename imin col min max minr maxr chname chuname* Fonction

filename	File name of the file
imin	Image
col	Column to be written in the file
min	Minimum value (input to IMLIB)
max	Maximum value (input to IMLIB)
minr	Minimum output value (real)
maxr	Maximum output value (real)
chname	Channel name
chuname	Name of channel unity

Description (2D): Take a FLIP file and append a column (curve) to it. The filename is the same except for the extension which is automatically added. The column is streched in dynamics as in the clip function.
Package: `nil`

`write-curve` *filename outimage col deep min max minf maxf chname chuname* Fonction

filename	File name string
outimage	Output FLIP image format previously read
col	Column number where curve will be wrote
deep	Origin deep (in feet)
min	Minimum value to put in IMLIB
max	Maximum value to put in IMLIB
minf	Minimum output value (float)
maxf	Maximum output value (float)
chname	Channel name
chuname	Units channel name

Description (2D): Writes a column (FLIP format). The *col* is normalized in its dynamic range, like CLIP function. It reads a FLIP image format before: because it needs the last read image to inheritates characteristics related with the signal acquisition parameters. The file name can be any valid operating system file name (including the file name path expression).

Exemple: `(write-curve "tt.curve" imin 4 1254.7 0 255 0.0 255.0)`

Package: `nil`

20.13 Detail coding

`glsktc` *imin imtmp marker imdecoded imcoded* Fonction

imin	Textured residue image to code
imtmp	Dummy argument (not used at the moment)
marker	Marker image for the zones to code
imdecoded	Image coded by dilation or erosion of k-level skeleton decomposition
imcoded	k-level grey skeleton image

Description (2D): `glsktc` (Grey Level SKeleton Texture Coding) performs the grey level skeleton decomposition of the zones not marked by *marker* of *imin*. The *imcoded* output is the best reconstruction of input image from the k-best-level of skeleton decomposition. The best criteria is a function of the entropy of k-best-level of the skeleton and the absolute reconstruction error. The function returns the best skeleton level (positive: erosion, negative: dilation)

Exemple: `(setf level (glsktc ori ori mask decoded coded))`

Package: `nil`

`detail_step` *imres imori immap imcod imsupport buffer size num (optional) contrast* Fonction

imres	Residual image
imori	Original image
immap	Map image
imcod	Coded details
imsupport	Binary support of details
buffer	Buffer of grey levels
size	Maximum detail size
num	Maximum number of details selected
contrast	Minimum contrast (optional)

Description (2D): Extraction, selection and coding of details from the coding residue *imres*. A ranking is done according to the perceptual significance of each detail in the original image. Only the first *num* details are selected. Areas where *immap*=0 are discarded for detail selection. The details are returned in *imcod* each coded with a constant grey level on a zero valued background. The binary support of the details and its grey levels are also returned.
Package: `nil`

`detail_extraction` *imres imori imdet size (& optional) contrast* Fonction

imres	Residual image
imori	Original image
imdet	Details image
size	Maximum detail size
contrast	Minimum contrast (optional)

Description (2D): Modified Meyers post-it thinning. Obtains details present in residual images. They are returned in *imdet* as isolated blobs on a zero background. Their amplitudes are the differential values over the original image without details. *contrast* is the minimum contrast over the background for a point to be considered as a possible detail. If no value is provided, it is set to the variance of the residual image.
Package: `nil`

`detail_selection` *imdet imori immap imlab num size* Fonction

imdet	Details image
imori	Original image
immap	Map image
imlab	Label image
num	Maximum number of details selected
size	Maximum detail size

Description (2D): Ranking and selection of details. Ranks and selects details from *imdet*. The selected details are labelled in order of importance and returned in *imlab*. The return value of the function is the number of details selected that is smaller or equal than *num*.
Package: `nil`

`detail_coding` *imlab imori imcod imsupport buffer num* Fonction

imlab	Labelled details
imori	Original image
imcod	Coded details
imsupport	Binary support of details
buffer	Buffer of grey levels
num	Maximum number of details to be coded

Description (2D): Fills details with constant values and returns the coded image *imcod* with the details coded on a zero background. Its binary support and a buffer with the. Amplitudes are also returned.

Package: `nil`

Index des constantes

Index des fonctions

Contenu des packages