

ELSe : la doc

MPD

23 mars 2007



# Table des matières

<b>I</b>	<b>Introduction</b>	<b>7</b>
<b>1</b>	<b>Exemples en dimension 1</b>	<b>9</b>
1.1	Présentation générale . . . . .	9
1.1.1	Objectifs de la bibliothèque . . . . .	9
1.1.2	Flux_Pts, Fonc_Num, Output et ELISE_COPY. . . . .	9
1.2	Des exemples en dimension 1 . . . . .	10
1.3	Exemples sur le type abstrait Fonc_Num . . . . .	10
1.3.1	Fonc_Num primitive . . . . .	10
1.3.2	Opérateurs sur les Fonc_Num . . . . .	13
1.4	Le type abstrait Flux_Pts . . . . .	14
1.4.1	Flux_Pts primitifs . . . . .	14
1.4.2	Opérateur sur les Flux_Pts . . . . .	14
1.5	Le type abstrait Output . . . . .	16
1.5.1	Output primitifs . . . . .	16
1.5.2	Opérateurs sur les Output . . . . .	16
1.6	Gestions des erreurs . . . . .	18
1.6.1	Détection des erreurs . . . . .	18
1.6.2	Prévision des débordements . . . . .	19
1.7	Un premier bilan . . . . .	19
<b>2</b>	<b>Manipulation d’images</b>	<b>21</b>
2.1	premiers essais . . . . .	21
2.2	Images, Images! Nous voulons des images! . . . . .	23
2.3	Palette . . . . .	24
2.4	Opérateur “,” sur les Fonc_Num, changement de coordonnées . . . . .	25
2.5	Transformation radiométrique . . . . .	27
2.6	Palette-RGB, fonctions de projections . . . . .	29
2.7	Flux en dimension 2 . . . . .	31
2.8	Virgule sur Out . . . . .	31
2.9	chc sur les Flux_Pts . . . . .	35
<b>3</b>	<b>Exemples, traitement d’images</b>	<b>39</b>
3.1	Filtre élémentaires, Opérateur trans . . . . .	39
3.2	Filtre prédéfini . . . . .	41
3.3	Interface C++/ELISE, ajout de filtre prédéfini . . . . .	45
3.4	Output de “réduction associative” . . . . .	48
3.5	Filtres “linéaires” . . . . .	50
<b>4</b>	<b>Exemples, analyse d’images</b>	<b>53</b>
4.1	Liste de points . . . . .	53
4.2	Relation de voisinage, dilate . . . . .	56
4.3	Composantes connexes conc . . . . .	60
4.4	Images sur 1,2 et 4 bits . . . . .	64
4.5	Erosion et autres “réduction associative sur relation” . . . . .	65
4.6	Random . . . . .	65

4.7	Principaux services offerts . . . . .	65
4.8	Avantages/inconvénients . . . . .	66
<b>5</b>	<b>bilan</b>	<b>67</b>
<b>6</b>	<b>exemples complets</b>	<b>69</b>
6.1	Egalisation d'histogramme . . . . .	69
6.2	Mandelbrot . . . . .	69
6.3	Morphing aléatoire . . . . .	69
6.4	Correlation sur image de synthèse aléatoire . . . . .	69
<b>7</b>	<b>Comment ça marche</b>	<b>71</b>
7.1	Les trois types de Flux_Pts : RLE, integer et real . . . . .	71
<b>II</b>	<b>Documentation de référence</b>	<b>73</b>
<b>8</b>	<b>Classes Utilitaires</b>	<b>75</b>
8.1	Les points 2 – $D$ (Pt2d<Type>) . . . . .	75
8.2	Les point 3 – $D$ (Pt3d<Type>) . . . . .	75
8.3	Les listes (ElList<Type>) . . . . .	75
8.4	Les piles ElFifo<Type> . . . . .	76
<b>9</b>	<b>Fenêtres graphique</b>	<b>79</b>
9.1	Création de couleur, la classe Elise_couleur . . . . .	79
9.1.1	Création . . . . .	79
9.1.2	Opération . . . . .	79
9.1.3	Conversion . . . . .	79
9.1.4	Couleur prédéfinie . . . . .	79
9.2	Palettes de couleur . . . . .	80
9.2.1	la classe de base Elise_Palette . . . . .	80
9.2.2	la classe Lin1Col_Pal . . . . .	80
9.2.3	la classe Gray_Pal . . . . .	80
9.2.4	la classe BiCol_Pal . . . . .	80
9.2.5	la classe TriCol_Pal . . . . .	80
9.2.6	la classe RGB_Pal . . . . .	80
9.2.7	la classe Circ_Pal . . . . .	81
9.2.8	la classe Disc_Pal . . . . .	81
9.2.8.1	Fonction habituelles . . . . .	81
9.2.8.2	Inspection . . . . .	81
9.2.8.3	La palette discrète “standard” P8COL . . . . .	81
9.2.8.4	Réduction de couleur . . . . .	81
9.3	Gérer les couleur 8-bits la classe Elise_Set_Of_Palette . . . . .	82
9.4	Les styles graphiques . . . . .	82
9.4.1	La classe Col_Pal . . . . .	82
9.4.2	La classe Line_St . . . . .	82
9.4.3	La classe Fill_St . . . . .	82
9.5	La classe de base El_Window . . . . .	82
9.5.1	Utilisation comme Output mode mailé . . . . .	82
9.5.2	Les ordres de dessin vecteurs . . . . .	82
9.5.3	Visualisation de graphes raster . . . . .	82
9.5.4	Changement de géométrie, la fonctionchc . . . . .	82
9.5.5	Mise en parallèle de fenêtre operator   . . . . .	82
9.5.6	Diverses fonctions utilitaires . . . . .	82
9.6	Les fenêtres sur la sortie vidéo, la classe Video_Win . . . . .	82
9.6.1	La classe Video_Display . . . . .	82
9.6.2	Construction et fonctions utilitaires . . . . .	83

9.6.3	Récupération d'évènement . . . . .	83
9.7	Les fenêtres postscript <code>PS_Window</code> . . . . .	83
9.7.1	Fichier postscript, la classe <code>PS_Display</code> . . . . .	83
9.7.2	Fenêtre postscript, la classe <code>PS_Window</code> . . . . .	83
9.7.3	Mosaïque de fenêtre, la classe <code>Mat_PS_Window</code> . . . . .	84
9.8	Les fenêtres raster <code>Bitm_Win</code> . . . . .	84
<b>10</b>	<b>Images et Liste de Points</b>	<b>85</b>
10.1	Généralités sur classes images . . . . .	85
10.1.1	Organisation des classes images . . . . .	85
10.1.2	méthode de lecture-écriture de <code>GenIm</code> . . . . .	85
10.1.3	L'énumération <code>GenIm : :type_el</code> . . . . .	86
10.1.4	Autre fonctionnalité des <code>GenIm</code> . . . . .	86
10.2	Images "standard" . . . . .	86
10.2.1	Propriété commune aux image stantdard . . . . .	86
10.2.2	Image "standard" de dimension 2 . . . . .	87
10.2.3	Image "standard" de dimension 1 et 3 . . . . .	87
10.3	Image de dimension 2 sur 1, 2 et 4 bits . . . . .	87
10.4	Liste de points . . . . .	88
10.4.1	la classe <code>Lin1Col_Pal</code> . . . . .	88
<b>11</b>	<b>Fichiers Images</b>	<b>89</b>
11.1	la classe <code>ElGenFileIm</code> . . . . .	89
11.1.1	Organisation des classes fichier-images . . . . .	89
11.1.2	Lecture-Écriture sur les <code>ElGenFileIm</code> . . . . .	90
11.1.3	Information sur les fichiers <code>ElGenFileIm</code> . . . . .	90
11.2	la classe <code>Elise_File_Im</code> . . . . .	90
11.2.1	Constructeur . . . . .	90
11.2.2	Support du format <code>pnm</code> . . . . .	91
11.3	Le format <code>GIF</code> . . . . .	92
11.3.1	Fonctions spécifiques à la classe <code>Gif_Im</code> . . . . .	92
11.3.2	Image multiples, la classe <code>Gif_File</code> . . . . .	92
11.4	Le format <code>Tiff</code> . . . . .	92
11.4.1	Généralités . . . . .	92
11.4.1.1	Principales caractéristiques supportées . . . . .	93
11.4.1.2	Enumeration codant les propriétés des images <code>TIFF</code> . . . . .	93
11.4.2	Manipulation de fichiers existants . . . . .	94
11.4.3	Création de fichiers . . . . .	94
11.4.3.1	Constructeurs pour fichier en couleur indexées . . . . .	94
11.4.3.2	Constructeur pour les autre type . . . . .	95
11.4.3.3	Arguments optionnels . . . . .	95
11.4.4	Ecriture et formats compressés . . . . .	96
<b>12</b>	<b>Opérateur arithmétiques</b>	<b>97</b>
12.1	Opérateur associatif mixte . . . . .	97
12.2	Opérateur binaire mixte . . . . .	97
12.3	Opérateur unaire entier . . . . .	97
12.4	Opérateur de comparaison . . . . .	97
12.5	Opérateur unaire mixte . . . . .	97
12.6	Opérateur unaire entier . . . . .	97
12.7	Opérateur mathématique . . . . .	97
12.8	Opérateur de conversion . . . . .	97
12.9	Opérateur sur les complexes . . . . .	97
12.10	Opérateur liés à la photogramétrie . . . . .	98
12.11	Opérateur liés à la colorimétrie . . . . .	98
12.12	Définition d'opérateur par l'utilisateur . . . . .	98

12.13	Quelques bizareries personnelles . . . . .	98
<b>13</b>	<b>Filtres prédéfinis</b>	<b>99</b>
<b>14</b>	<b>Vectorisation</b>	<b>101</b>
14.1	Squeletisation . . . . .	101
14.1.1	Squeletisation de petites images . . . . .	101
14.1.2	Squeletisation de grandes images . . . . .	105
14.2	Chainage . . . . .	106
14.3	Approximation polygonale . . . . .	108
14.3.1	description de l’algorithme “pur” . . . . .	108
14.3.2	Options . . . . .	110
14.3.3	La class <code>ArgAPP</code> pour spécifier les options . . . . .	111
14.3.4	La fonction d’approximation polygonale <code>approx_poly</code> . . . . .	111
14.3.5	exemples . . . . .	112
<b>III</b>	<b>Utilisation “avancée”</b>	<b>115</b>
<b>IV</b>	<b>Appendices</b>	<b>117</b>
<b>A</b>	<b>Référence bibliographique</b>	<b>119</b>
<b>B</b>	<b>Listing des programmes d’exemples</b>	<b>123</b>
B.1	intro0.cpp . . . . .	124
B.2	introd2.cpp . . . . .	126
B.3	introfiltr.cpp . . . . .	128
B.4	introanalyse.cpp . . . . .	130
B.5	ddrvecto.cpp . . . . .	133

Première partie

Introduction





# Chapitre 1

## Exemples en dimension 1

### 1.1 Présentation générale

#### 1.1.1 Objectifs de la bibliothèque

La bibliothèque `Elise` est une bibliothèque `C++` de manipulation de données maillées et plus spécifiquement de manipulation d'images (= donnée de dimension 2). Par opposition à d'autres bibliothèques plus "classiques", `Elise` est une assez petite bibliothèque offrant un nombre limité de fonctions. La puissance d'expression d'`Elise` vient alors du haut niveau d'abstraction qu'elle offre et de la possibilité d'assembler de manière simple les objets élémentaires de la bibliothèque pour effectuer en une seule "instruction" des opérations arbitrairement compliquées.

`Elise` a été conçue dans un contexte de recherche appliquée en analyse d'images. On a cherché à faire un outil qui permette de tester facilement des algorithmes et de développer rapidement des petits prototypes d'application. `Elise` vise donc plus à être un outil adapté à la *R&D* ou à l'enseignement du traitement et analyse d'images qu'à permettre de développer des applications de production.

Ce chapitre effectue une présentation générale et relativement informelle de l'utilisation d'`Elise` : d'une part il n'est pas exhaustif quant aux opérations présentes dans `Elise` et d'autre part il ne constitue pas une doc de référence pour les opérations présentées. Son objectif premier est de présenter le "style" de programmation qu'encourage `Elise`.

#### 1.1.2 Flux\_Pts, Fonc\_Num, Output et ELISE\_COPY.

La fonction `ELISE_COPY`<sup>1</sup> est (presque) l'unique fonction de "top level" offerte par la bibliothèque. Cette fonction prend en argument trois paramètres qui sont de type respectifs `Flux_Pts`, `Fonc_Num` et `Output`. Ces 3 types sont des types abstraits de données et les objets qui peuvent être représentés par un `Flux_Pts`, `Fonc_Num` et `Output` sont très variables. La signification de ces trois types est la suivante :

1. les `Flux_Pts`, ou *flux de points*, permettent de décrire des ensembles de points de  $\mathcal{Z}, \mathcal{Z}^2, \mathcal{Z}^3 \dots$  ou  $\mathcal{R}, \mathcal{R}^2 \dots$  ;
2. les `Fonc_Num`, ou *fonctions numériques*, permettent de décrire des fonctions de  $\mathcal{Z}^k$  dans  $\mathcal{Z}^p$ , ou  $\mathcal{R}^k$  dans  $\mathcal{Z}^p$  ou  $\dots$  ;
3. les `Output`, permettent de décrire tous les objets qui, d'une manière ou d'une autre, permettent de mémoriser ou de visualiser le contenu d'une action.

De manière très simplifiée, on peut décrire ainsi comment sont implantés ces 3 types abstraits :

1. les `Flux_Pts` sont des itérateurs permettant de parcourir des ensembles ; chaque flux est caractérisé par la façon dont il redéfinit la méthode virtuelle `bool next_pt(Pt &)` qui renvoie, s'il y a lieu, le prochain point du flux ;
2. les `Fonc_Num` sont des fonctions, donc des objets qui permettent de calculer pour chaque point la valeur de la fonction en ce point ; chaque `Fonc_Num` est caractérisée par la façon dont elle redéfinit la méthode virtuelle `Pt val(Pt pt)` qui renvoie la valeur que la fonction en `pt` ;

---

<sup>1</sup>Il s'agit en fait d'une macro

3. les **Output** permettent d’avoir un certain effet de bord associé à chaque action ; chaque **Output** est caractérisé par la façon dont il redéfinit la méthode virtuelle `void update(Pt pt, Pt val)` qui indique quelle est l’action de l’objet quand on lui demande de se “modifier” au point `pt` avec la valeur `val` ; par exemple, pour une fenêtre graphique, `update(Pt pt, Pt val)` pourra signifier “colorier le pixel `pt` en la couleur correspondant à `val`”.

La signification de l’ordre `ELISE_COPY(flux, fonc, out)` est alors la suivante : *pour tous les points `p` de `flux`, soit `v` la valeur de `fonc` en `p`, modifier `out` en `p` avec la valeur `v`*. La figure 1.1 donne un pseudo-code correspondant à une description simplifiée de la fonction `ELISE_COPY`.

```
void ELISE_COPY(Flux_Pts flux, Fonc_Num fonc, Output out)
{
    Pt pt;
    while (flux.next(pt))
        out.update(pt, fonc(pt));
}
```

FIG. 1.1 – pseudo-code de la fonction `ELISE_COPY`

## 1.2 Des exemples en dimension 1

Bien qu’`Elise` soit surtout orientée traitement d’images 2-D, on introduit les premiers exemples avec des opérations sur des signaux 1-D. L’objectif est de se familiariser avec les principaux concepts d’`Elise` dans un contexte déjà connu ; en effet, la manipulation des signaux 1-D sous `Elise` rappellera ce que l’on peut rencontrer dans la plupart des outils de type “plotter”.

Les exemples de cette section se trouvent tous dans le fichier “`applis/doc_ex/intro0.cpp`” ; on invite le lecteur à se référer à ce fichier et surtout à le compiler puis à l’exécuter. Le listing complet se trouve en section B.1 page 124. Le programme `main` commence par les lignes de la figure 1.2. Les appels à `Elise` faits dans ces premières lignes seront détaillés dans les chapitres suivants, pour la compréhension des exemples, il suffit de retenir que :

- on a ouvert un écran `Ecr` sur le terminal standard en réservant un certain nombre de ressources couleur ;
- on a ouvert une fenêtre `Wv` sur cet écran `Ecr` ;
- on a créé un plotter `Plot1` sur la fenêtre `Wv` en donnant une liste assez longue de paramètres d’initialisation ; un plotter est un objet qui est associé à une fenêtre graphique et a pour objectif de visualiser les signaux  $1 - D$  ;

A la suite de cette initialisation, on trouve une série de lignes qui sont toutes sur le modèle du code de la figure 1.3 :

- on réinitialise le contenu du plotter par `Plot1.clear()` ;
- on fait appel à une des petites fonctions, que nous allons détailler dans la section suivante, et qui auront pour but d’illustrer les trois types abstraits (la fonction est ici `TEST_plot_FX(Plot1)`) ;
- on affiche les axes et on met un point d’arrêt ;

## 1.3 Exemples sur le type abstrait `Fonc_Num`

### 1.3.1 `Fonc_Num` primitive

#### ◊ Fonctions coordonnées

Commençons par la fonction `TEST_plot_FX`, dont le code est donné figure 1.4. Comme il s’agit du tout premier exemple, commentons les 3 arguments passés :

- `plot.all_pts()` renvoie un `Flux_Pts` correspondant à l’ensemble de tous les points sur lesquels est défini `plot` ; dans ce contexte, il s’agit simplement de l’intervalle entier  $[-50 \ 50]$  ;

<pre> // sz of images we will use  Pt2di SZ(512,512);  // palette allocation  Disc_Pal Pdisc = Disc_Pal::P8COL(); Elise_Set_Of_Palette SOP(newl(Pdisc));  // Creation of video windows  Video_Display Ecr((char *) NULL); Ecr.load(SOP); Video_Win Wv (Ecr,SOP,Pt2di(50,50),                Pt2di(SZ.x,SZ.y)); </pre>	<pre> // define a plotter  Plot_1d Plot1 (     Wv,     Line_St(Pdisc(P8COL::green),3),     Line_St(Pdisc(P8COL::black),2),     Interval(-50,50),     newl(P1Box(Pt2di(3,3),                SZ-Pt2di(3,3))         )     + P1ScaleY(1.0)     + P1BoxSty(Pdisc(P8COL::blue),3)     + P1ClipY(true)     + P1ModePl(Plots::draw_fill_box)     + P1ClearSty(Pdisc(P8COL::white))     + PlotFilSty(Pdisc(P8COL::red)) ); </pre>
---	---

FIG. 1.2 – code initialisant certains objets (palette, écran, fenêtre, plotter)

<pre> Plot1.clear(); TEST_plot_FX(Plot1); </pre>	<pre> Plot1.show_axes(); Plot1.show_box(); getchar(); </pre>
--	--

FIG. 1.3 – Exemple des appels effectués aux fonctions test.

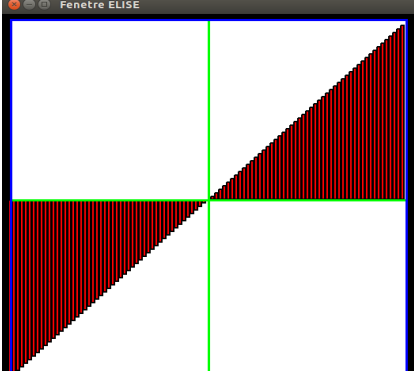
<pre> void TEST_plot_FX (Plot_1d plot) {     ELISE_COPY     (         plot.all_pts(),         FX,         plot.out()     ); } </pre>	
--	--

FIG. 1.4 – code pour plotter la fonction identité  $x \rightarrow x$  et aspect du plotter après exécution.

- `FX` est une variable globale définie par `Elise`, dans le cas de la dimension 1 elle correspond à la fonction identité  $x \rightarrow x$  (de manière plus générale elle permet de référencer la fonction “première coordonnée”  $x, y \rightarrow x$  en dim 2,  $x, y, z \rightarrow x$  en dim 3 ...);
- `plot.out()` renvoie la “conversion” du plotter `plot` en `Output`; pour un plotter converti en `Output` `P1Out`, soit `x` un point et `y = fonc(x)` la valeur d’une fonction en ce point, la signification de `P1Out.update(y,x)` est simplement *plotter sur plot le point (x,y)* (ici, compte-tenu des paramètres

avec lesquels le plotter a été initialisé, plotter le point  $(x, y)$  se fait en dessinant un rectangle rouge, centré en  $x$  et de hauteur  $y$ .

On voit donc que `ELISE_COPY(plot.all_pts(), FX, plot.out())` signifie *visualiser dans plot, sur l'intervalle  $[-50, 50[$ , le graphe de la fonction  $y = x$* . La `Fonc_Num` “FX” est dite `Fonc_Num` primitive car il n’est pas nécessaire pour la construire de disposer d’autre `Fonc_Num`. Introduisons, maintenant deux autres exemples de `Fonc_Num` primitives : les constantes et les tableaux.

#### ◊ Fonctions constantes

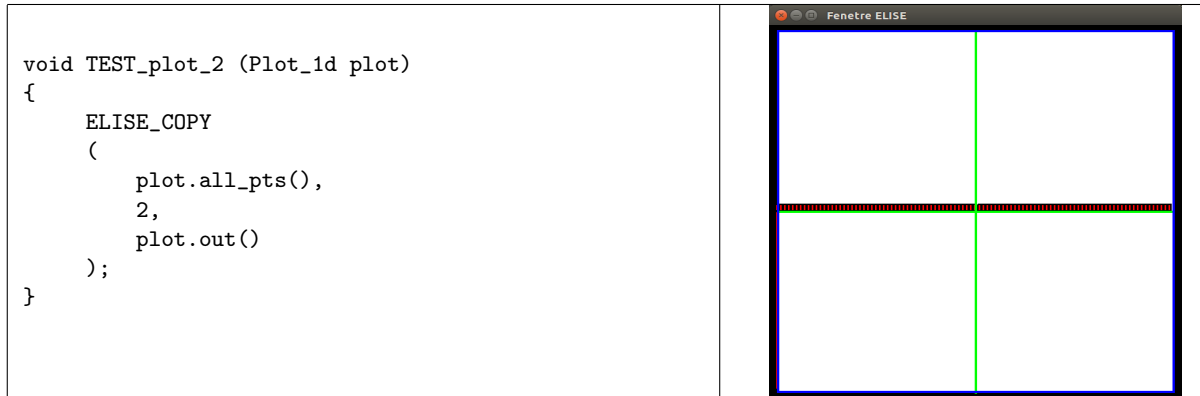


FIG. 1.5 – code pour plotter la fonction constante  $x \rightarrow 2$  et aspect du plotter après exécution.

La fonction `TEST_plot_2`, dont le code est donné figure 1.5, introduit l’utilisation d’une fonction constante; ici,  $\mathcal{E}\mathcal{P}\mathcal{S}\mathcal{E}$  “comprend”<sup>2</sup> que, dans le contexte, 2 signifie la fonction constante  $f : x \rightarrow f(x) = 2$ .

#### ◊ “Tableaux” et `Fonc_Num`.

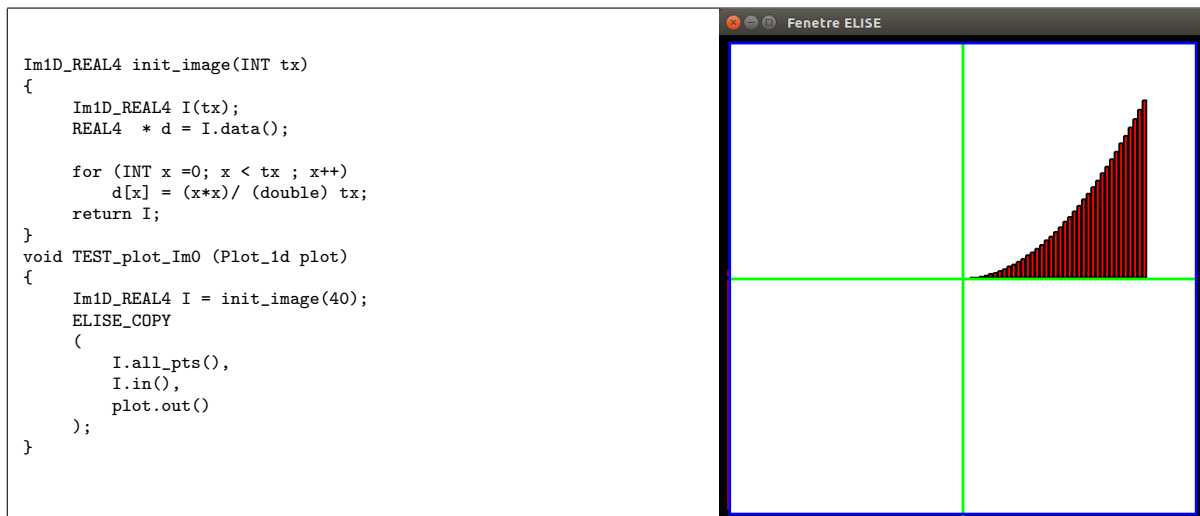


FIG. 1.6 – code pour créer une image en RAM puis pour la plotter, aspect du plotter après exécution.

Le code de la figure 1.6 définit deux fonctions :

- la fonction `init_image` alloue, initialise et renvoie un tableau  $\mathcal{E}\mathcal{P}\mathcal{S}\mathcal{E}$  de dimension 1, de taille `tx` dont les éléments sont stockés en virgule flottante sur 4 octets; les tableaux  $\mathcal{E}\mathcal{P}\mathcal{S}\mathcal{E}$  sont une encapsulation de tableaux “classiques” et on peut, soit les manipuler à “haut niveau” par l’intermédiaire de fonctions membres (voir fonction `TEST_plot_Im0`) pour communiquer avec  $\mathcal{E}\mathcal{P}\mathcal{S}\mathcal{E}$ , soit récupérer l’adresse de leur zone de données (`REAL4 * d = I.data()`) et leur dimension pour les manipuler avec du code C “classique” et communiquer avec d’autres bibliothèques;

<sup>2</sup>ceci est possible car il existe, dans la classe `Fonc_Num` des constructeurs permettant de convertir les entiers ou réels en `Fonc_Num`

- la fonction `TEST_plot_Im0` appelle `init_image` pour créer un tableau `I` de taille 40, puis elle visualise son contenu dans le plotter ; pour ceci, on convertit le tableau en `Fonc_Num` grâce à `I.in()`, on obtient ainsi la fonction  $x \rightarrow d[x]$  (si  $d$  est l'adresse du premier élément de  $I$ ) ;

### 1.3.2 Opérateurs sur les Fonc\_Num

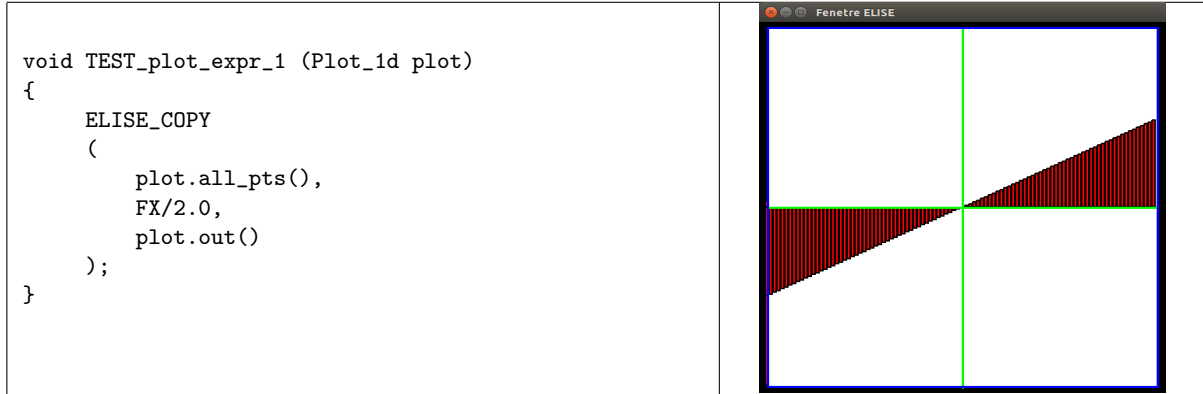


FIG. 1.7 – code pour plotter  $y = \frac{x}{2}$

Le code de la fonction `TEST_plot_expr_1` (figure 1.7) introduit notre premier exemple d'opérateur sur les `Fonc_Num` : “l'opérateur binaire `/`”. Donnons quelques commentaires :

- si  $f_1$  et  $f_2$  sont deux `Fonc_Num` alors  $f_1/f_2$  permet de décrire la fonction  $x \rightarrow \frac{f_1(x)}{f_2(x)}$
- ici par exemple, `FX/2.0` est compris par `ELISE` comme la fonction  $x \rightarrow \frac{x}{2.0}$  ;
- ce qui est vrai pour `/` l'est aussi pour la plupart des opérateurs arithmétiques du C++ (`+`, `-`, `*`, `%`, `&`, `<`, `==`, ... le détail est donné dans les chapitres suivants) :  $f_1 + f_2$  définit la fonction  $x \rightarrow f_1(x) + f_2(x)$ ,  $f_1 * f_2$  définit la fonction  $x \rightarrow f_1(x) * f_2(x)$  ...
- ceci est aussi vrai pour un certain nombre d'opérateurs mathématiques ; par exemple `cos(f)` définit la fonction  $x \rightarrow \cos(f(x))$  ...

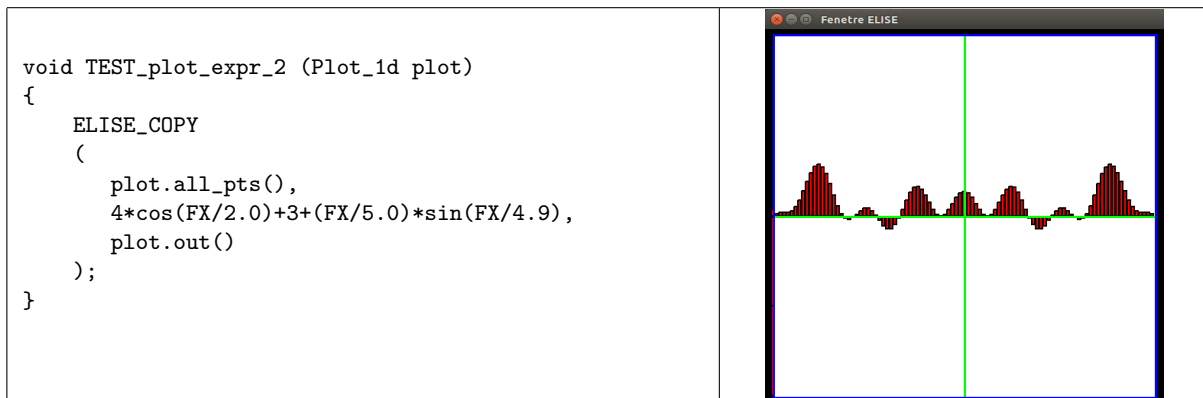
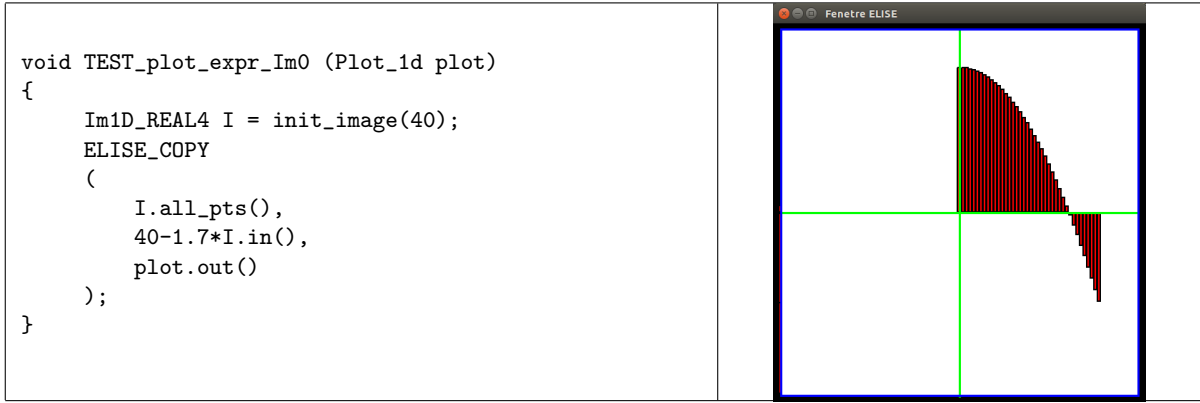
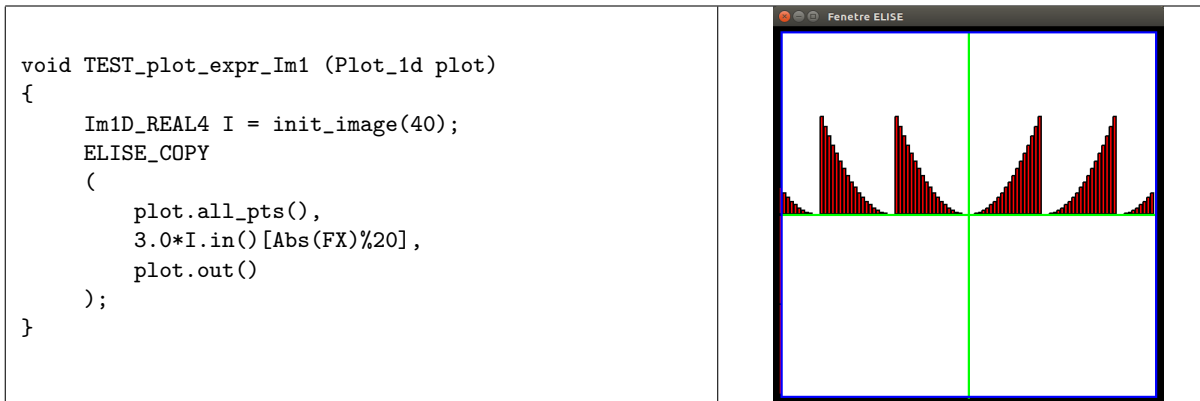


FIG. 1.8 – code pour plotter  $y = 4 * \cos(\frac{x}{2}) + 3 + \frac{x}{5} * \sin(\frac{x}{4.9})$

Comme les résultats de ces opérations arithmétiques sont des `Fonc_Num` au même titre que leurs arguments, ils peuvent sans problème être utilisés eux même comme arguments d'autres opérateurs pour former des expressions arbitrairement compliquées. Par exemple, dans la fonction `TEST_plot_expr_2` de la figure 1.8, l'expression `4*cos(FX/2.0)+ 3+ (FX/5.0) * sin(FX/4.9)` définit la fonction  $x \rightarrow 4 * \cos(\frac{x}{2}) + 3 + \frac{x}{5} * \sin(\frac{x}{4.9})$

Si `I` est un tableau `ELISE`, alors `I.in()` est une `Fonc_Num` et peut aussi être utilisée dans des opérations arithmétiques comme le montre la fonction `TEST_plot_expr_Im0` de la figure 1.9.

En plus des opérateurs arithmétiques, `ELISE` offre de nombreux opérateurs de filtrage dont nous donnerons les premiers exemples dans la section de ce chapitre consacrée aux images.

FIG. 1.9 – code pour plotter  $x \rightarrow 40 - 1.7 * I$ FIG. 1.10 – code pour plotter  $x \rightarrow 3 * I[|x| \% 20]$ 

La fonction `TEST_plot_expr_Im1` de la figure 1.10 introduit un autre opérateur : l’opérateur de composition des `Fonc_Num`; si  $f_1$  et  $f_2$  sont des `Fonc_Num`, alors  $f_1[f_2]$  définit la fonction composée, c’est à dire la fonction  $x \rightarrow f_1(f_2(x))$ . Donc, ici, `I.in() [Abs(FX)%20]` désigne la fonction qui, pour chaque  $x$ , renvoie l’élément de  $I$  d’indice  $|x| \% 20$ . Comme on le verra en 2, cet opérateur est couramment utilisé pour effectuer des transformations géométriques (ou anamorphose ou “morphing”) sur les images et pour certaines transformations radiométriques.

## 1.4 Le type abstrait `Flux_Pts`

### 1.4.1 `Flux_Pts` primitifs

En dimension 1, il existe essentiellement un `Flux_Pts` primitif : l’intervalle connexe  $[x_0 \ x_1[$  que l’on appelle `rectangle(x0,x1)`<sup>3</sup> et qui désigne l’ensemble des points  $x \in \mathbb{Z} / x_0 \leq x < x_1$ . La fonction `TEST_plot_rects` de la figure 1.11 donne un exemple d’utilisation.

Dans tous les exemples précédents on avait utilisé `I.all_pts()` ou `plot.all_pts()`, en fait la fonction membre `all_pts` se contente de faire un appel à la primitive `rectangle` avec les “bons” paramètres (par exemple, pour un tableau de dimension 1,  $x_0 = 0$  et  $x_1 = \text{taille de l'image}$ ).

### 1.4.2 Opérateur sur les `Flux_Pts`

Comme avec les `Fonc_Num`, il existe des opérateurs permettant de créer des `Flux_Pts` complexes à partir de `Flux_Pts` élémentaires.

<sup>3</sup>par généralisation du cas  $d = 2$ , ELISE appelle `rectangle` tous les pavés de  $\mathbb{Z}^d$

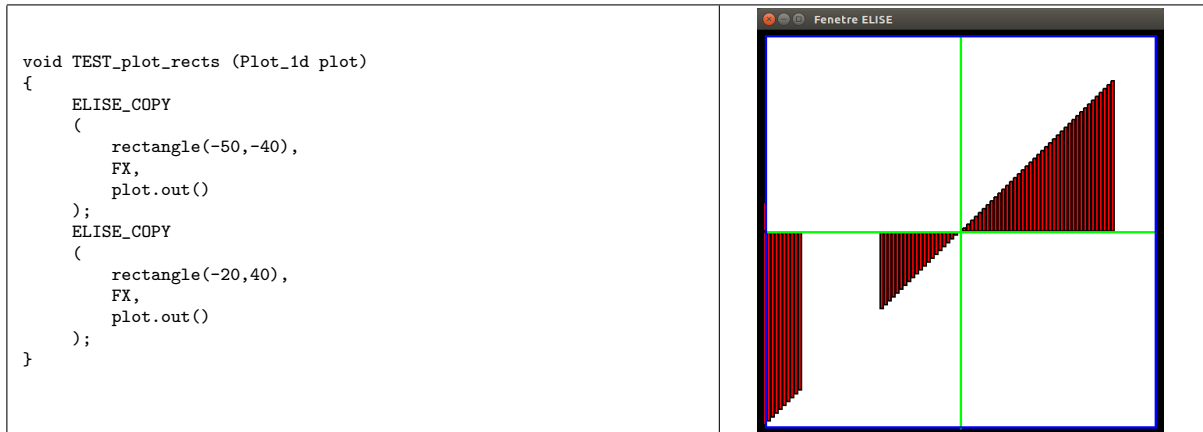


FIG. 1.11 – Utilisation de deux rectangles.

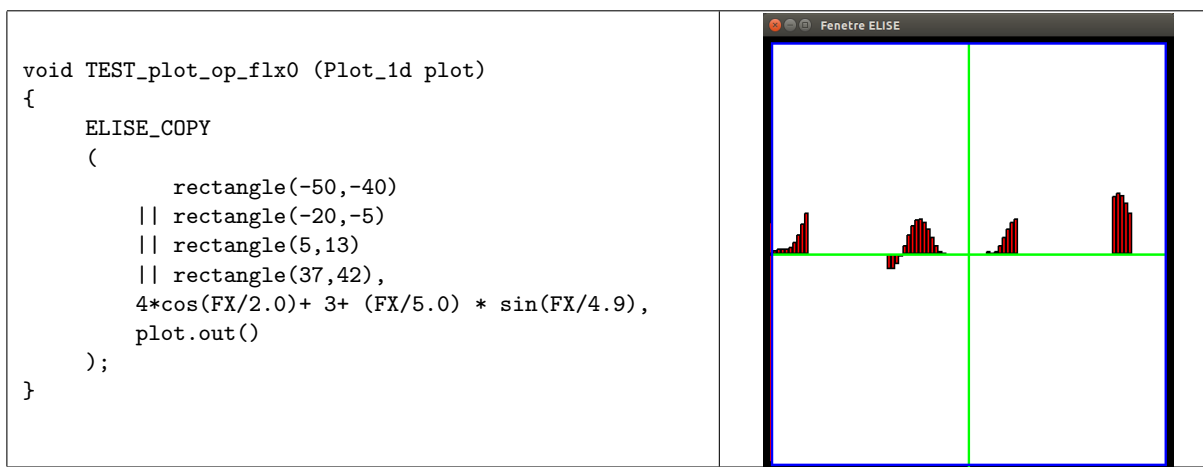


FIG. 1.12 – Opérateur (noté ||) de concaténation sur les flux.

La fonction `TEST_plot_op_flx0` (figure 1.12) utilise l’opérateur de “concaténation” de `Flux_Pts`; si  $fl_1$  et  $fl_2$  sont des `Flux_Pts`, alors  $fl_1 || fl_2$  est le `Flux_Pts` obtenu en parcourant successivement les points de  $fl_1$  puis ceux de  $fl_2$ ; en termes ensemblistes, cette opération correspond donc à peu près à l’opération d’union; dans cet exemple, l’ensemble parcouru est donc  $[-50 - 40 \cup [-20 - 5 \cup [5 13 \cup [37 42[$ .

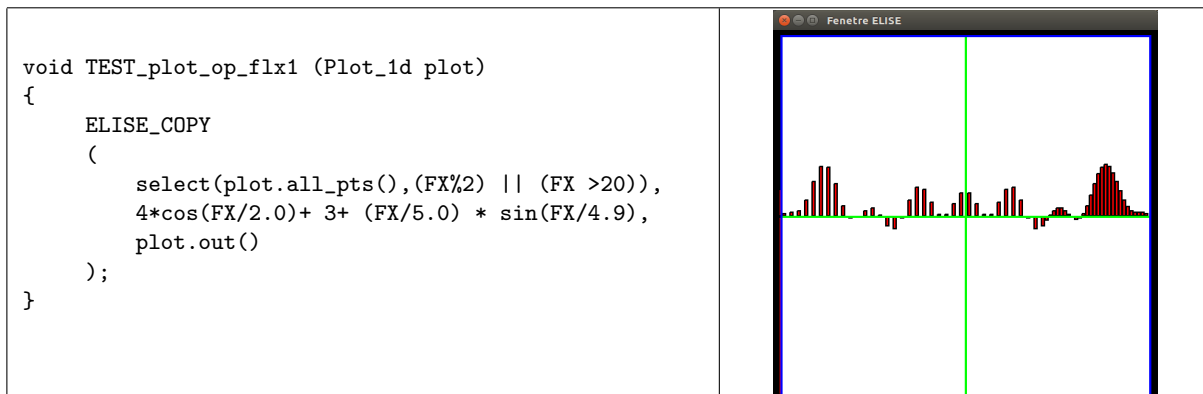


FIG. 1.13 – Opérateur de sélection sur les flux.

La fonction `TEST_plot_op_flx1` (figure 1.13) utilise un opérateur très courant sur les flux : l’opérateur `select`; cet opérateur prend en argument un `Flux_Pts`  $flx$  et une `Fonc_Num`  $fonc$  et crée un

flux qui contiendra les points de  $flx$  tels que  $fonc$  soit vraie <sup>4</sup>; du point de vue ensembliste `select(plot.all_pts(),(FX%2) || (FX >20))`, signifie donc  $\{x \in [-50 \ 50] / (x\%2 \neq 0) \text{ ou } (x > 20)\}$  ou encore en français *l'ensemble des  $x$  t.q.  $-50 \leq x < 50$  et  $x$  est impair ou supérieur à 20.*

## 1.5 Le type abstrait Output

### 1.5.1 Output primitifs

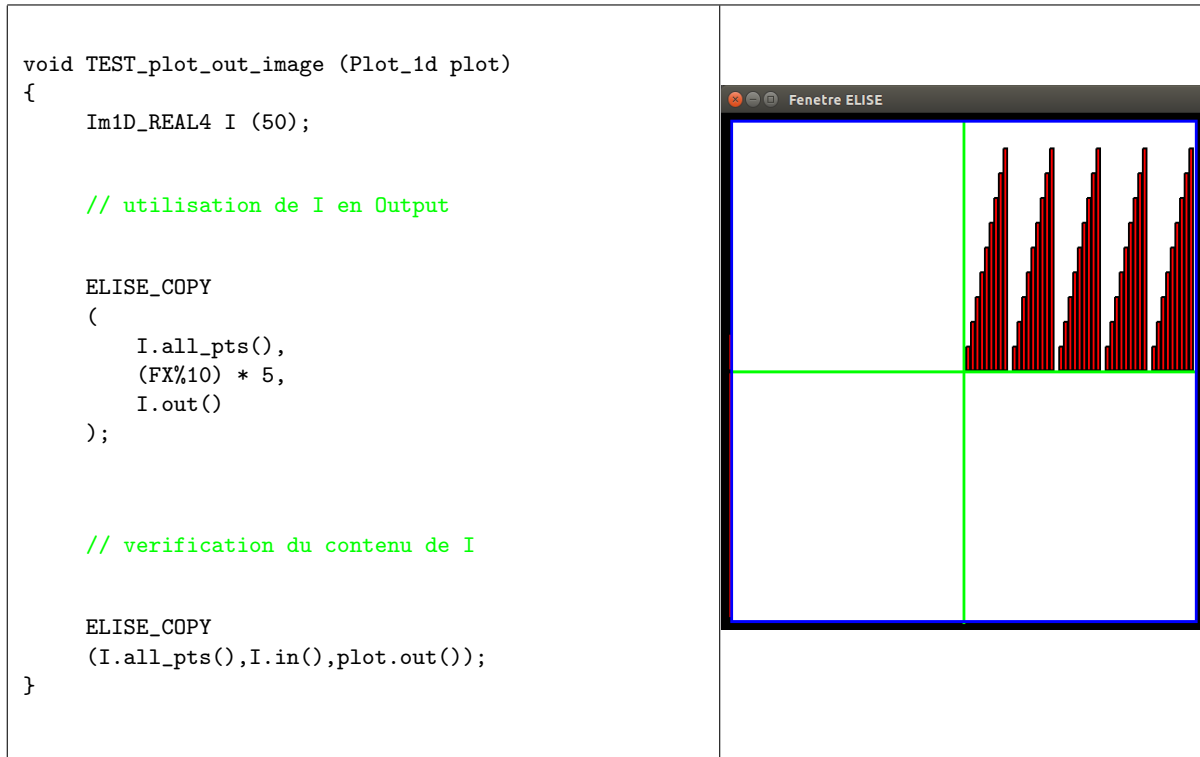


FIG. 1.14 – Output primitifs : écriture dans une image ; ici on écrit  $y = (x\%10) * 5$  dans le tableau  $I$  puis l’on visualise  $I$  avec le plotter.

De manière générale, les **Output** ont deux grands types d’utilité : visualiser et mémoriser. Le seul **Output** que nous ayons vu jusqu’à présent est le **plotter** qui correspond à la fonction de visualisation (pour les signaux 1 – D). Pour la fonction de mémorisation, on introduit maintenant les tableaux considérés comme **Output**.

Pour modifier le contenu d’un tableau **Tab**, dont  $d$  est l’adresse du 1<sup>er</sup> élément, on peut soit attaquer directement les données “physiques” comme dans l’exemple de la figure 1.6, soit passer par la fonction membre `out`. Si **TabOut** est l’**Output** renvoyé par `Tab.out()`, alors l’effet de la méthode `update(INT x, INT v)` sur **TabOut** est d’aller écrire la valeur  $v$  dans le  $x^{\text{ème}}$  élément de  $d$ .

La fonction `TEST_plot_out_image` de la figure 1.14 illustre cette utilisation des tableaux comme **Output** :

- dans un premier temps on va écrire dans le tableau  $I$ , grâce à `I.out()`, la fonction  $(x\%10) * 5$ ;
- ensuite, dans un but “didactique”, on visualise le contenu de  $I$  dans le plotter pour vérifier que l’écriture dans tableau a bien eu pour effet de modifier ses éléments.

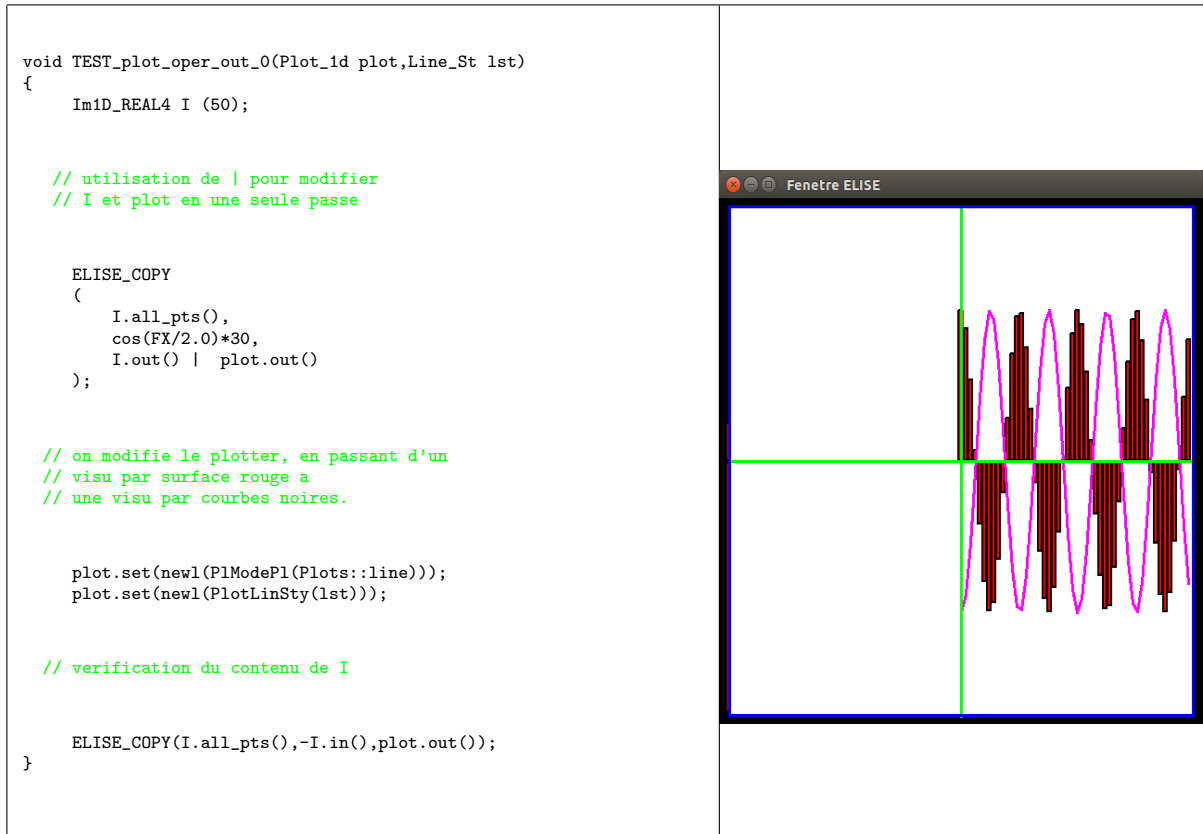
### 1.5.2 Opérateurs sur les Output

#### ◊ Mise en parallèle

Comme avec les **Fonc\_Num** et les **Flux\_Pts** il existe en **Elise** de nombreux opérateurs sur les **Output**. On introduit ici celui qui est sans doute d’usage le plus fréquent : l’opérateur de mise en parallèle (noté  $\parallel$ ).

<sup>4</sup>c.a.d  $\neq 0$  selon une convention du **C++** utilisée systématiquement dans **Elise**, et sur laquelle on reviendra



FIG. 1.15 – Opérateur (noté `|`) de mise en parallèle des Output.

Dans l'exemple de la figure 1.14, on a d'abord copié une fonction dans l'image  $I$  puis ensuite copié le contenu de  $I$  dans le plotter. Dans une application réelle, il serait plus simple et plus efficace de pouvoir, en une seule instruction, copier la fonction *à la fois* dans l'image et le plotter. Ceci est possible, sans rajouter de fonctions de “top-level” ni de types fondamentaux grâce à l'opérateur de mise en parallèle.

Si  $o_1$  et  $o_2$  sont deux Output, alors l'Output  $o_1|o_2$  est un output qui se contente de répercuter les messages de modification à  $o_1$  puis à  $o_2$ . Autrement dit la méthode `update(p,v)` de  $o_1|o_2$  se contente de faire appel séquentiellement à `o1.update(p,v)` puis à `o2.update(p,v)`. La figure 1.15 illustre l'utilisation de cet opérateur :

- tout d'abord on copie la fonction  $\cos(\frac{x}{2}) \cdot 30$  simultanément dans  $I$  et `plot` grâce à l'expression `I.out() | plot.out()` ;
- ensuite, dans un but “didactique”, on vérifie que  $I$  a été correctement modifié ; pour ceci on modifie la visualisation du plotter (on met une visu par courbes noires) et on copie  $I$  en négatif dans le plotter. Bien sûr, comme `|` est un opérateur associatif du C++ , il est possible d'effectuer la mise en parallèle d'autant d'Output que nécessaire en manipulant des expressions telles que  $o_1|o_2|\dots|o_n$ . Ceci va d'ailleurs être illustré dans la section suivante.

#### ◊ Exemples d'autres opérateurs

L'exemple de la figure 1.16 illustre l'associativité de l'opérateur “`|`” sur les Output et donne 2 exemples de nouveaux opérateurs sur les Output.

L'opérateur (fonction membre) `chc` :

- si  $o$  est un Output et  $f$  une `Fonc_Num` alors `o.chc(f)` effectue un changement de coordonnées sur  $o$  : les “messages” de modification, au lieu d'être envoyés au point  $p$  généré par le `Flux_Pts`, sont envoyés au point  $f(p)$  (cet opérateur est à peu près analogue au morphing sur les images) ;
- autrement dit, la fonction `update(p,v)` de `o.chc(f)` se contente de faire appel à `o.update(f(p),v)` ;
- donc dans l'exemple 1.16, `plot.out().chc(FX-50)` a pour effet de dessiner la partie de surface rouge correspondant aux  $x < 0$  ; en effet au lieu d'aller écrire en  $x$ , on va écrire en  $x - 50$ , donc tout le dessin est translaté de 50 vers la gauche ;

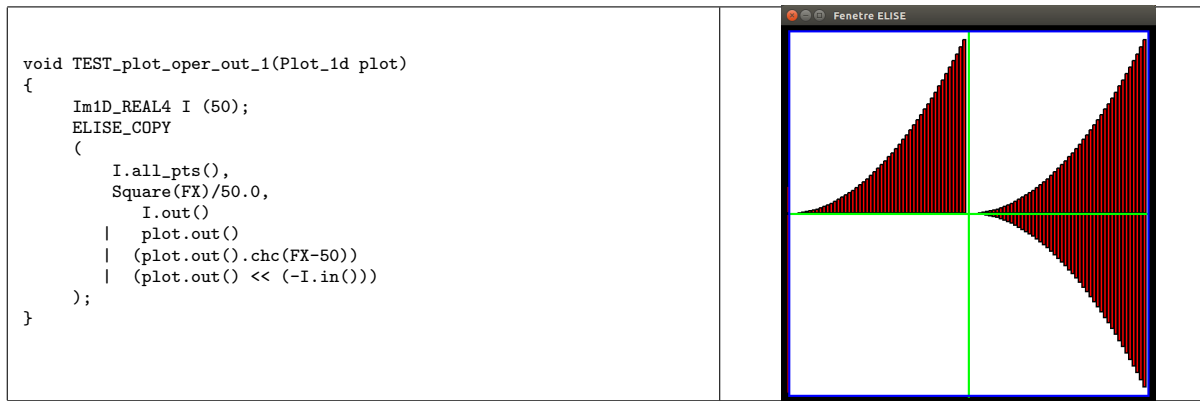


FIG. 1.16 – Deux opérateurs sur les Output : changement de choordonnées (chc) et “redirection” (<<).

L’opérateur <<;

- si  $o$  est un Output et  $f$  une Fonc\_Num alors  $o \ll f$  effectue une redirection sur  $o$ ; au lieu de demander à l’objet de se modifier avec les valeurs calculées pour la Fonc\_Num argument de ELISE\_COPY, on va lui demander de se modifier avec les valeur calculcée au moyen de la fonction  $f$ ;
- autrement dit, la fonction `update(p,v)` de  $o \ll f$  se contente de faire appel à `o.update(p,f(p))`;
- donc dans l’exemple 1.16, `plot.out()<<(-I.in())` a pour effet de dessiner la partie de surface rouge correspondant aux  $y < 0$ ;

## 1.6 Gestions des erreurs

### 1.6.1 Détection des erreurs

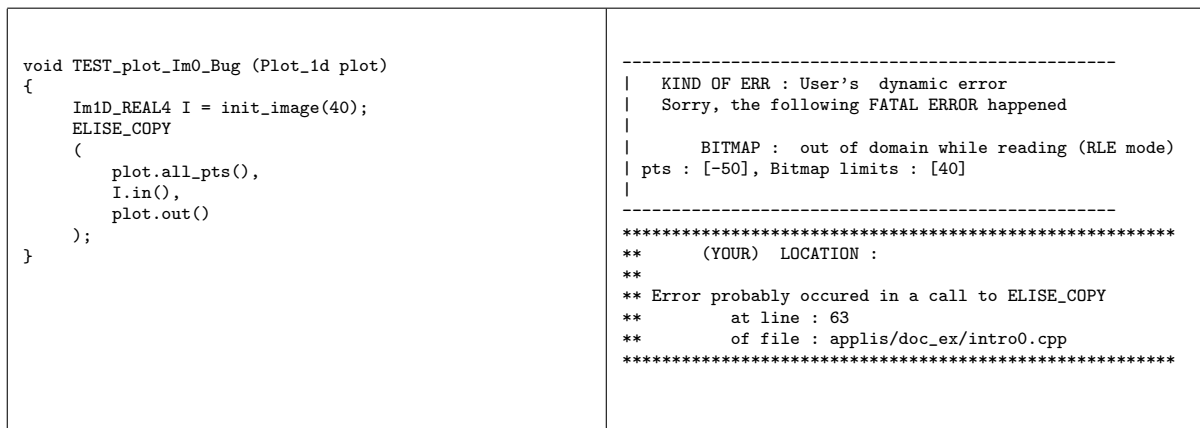


FIG. 1.17 – Exemple de message d’erreur; le code de la colone de gauche génère un débordement de tableau et conduit au message de la colone de droite.

Dans l’exemple de la figure 1.6, nous avons utilisé le Flux\_Pts “`I.all_pts()`” (c.a.d. l’intervalle  $[0 \ 40]$ ) pour plotter le tableau. Si nous utilisons le Flux\_Pts “habituel” `plot.all_pts()` comme dans la fonction `TEST_plot_Im0_Bug` de la figure 1.17, ELISE va détecter que, par exemple pour  $x = -50$ , l’on tente d’accéder à des éléments hors du tableau et générer une erreur à l’exécution.

Une des caractéristiques importantes d’ELISE est que, sauf mention explicite <sup>5</sup> contraire de l’utilisateur (ou bug dans ELISE!), toutes les exécutions qui pourraient potentiellement rendre le programme incohérent (indexation hors limite, division par zéro etc...), sont systématiquement vérifiées. Si par exemple, dans le fichier “`applis/doc_ex/intro0.cpp`”, on supprime les commentaires qui encadrent l’appel à `TEST_plot_Im0_Bug`, on obtiendra un message comme celui de la figure 1.17.

<sup>5</sup>par exemple, pour gagner du temps, une fois le programme mis au point

### 1.6.2 Pr vision des d bordements

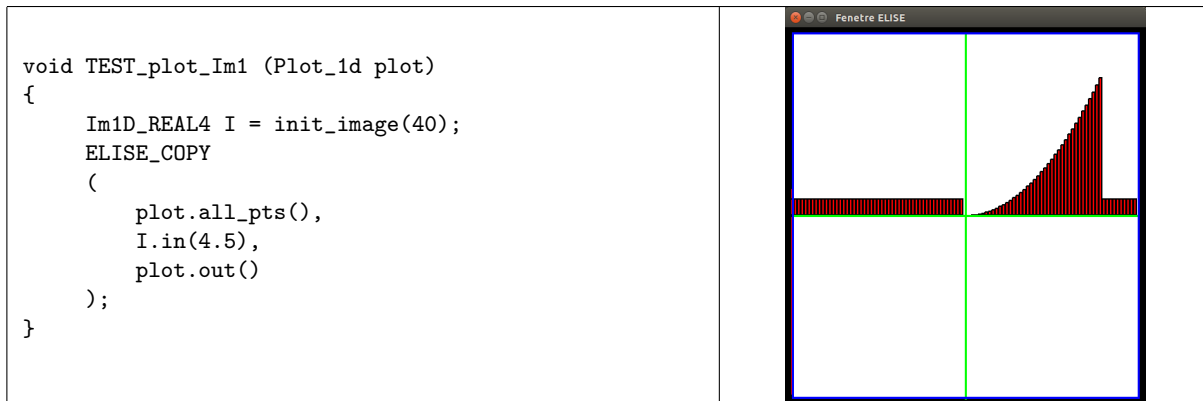


FIG. 1.18 – Utilisation d'un tableau avec prolongement par une constante en dehors de son domaine de d finition.

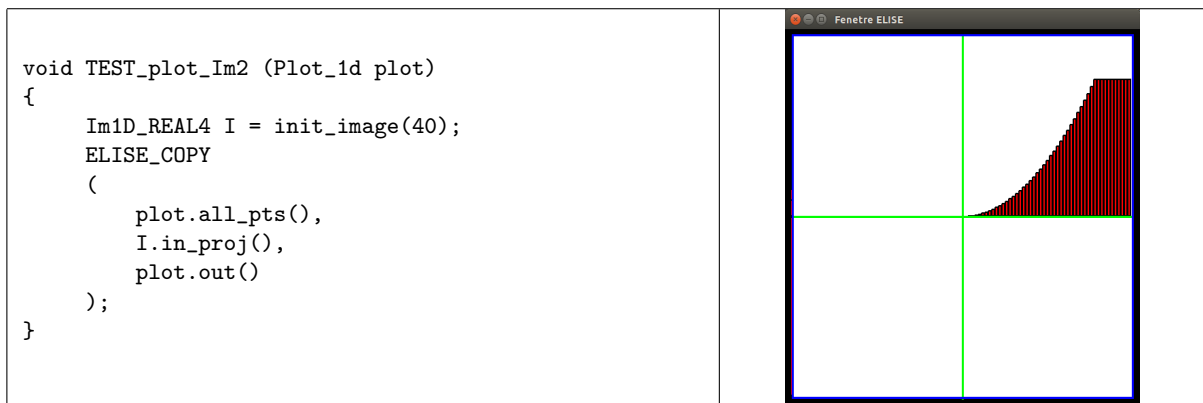


FIG. 1.19 – Utilisation d'un tableau avec prolongement par continuit  en dehors de son domaine de d finition.

Parfois, on souhaite sciemment manipuler un tableau avec un `Flux_Pts` qui d borde de la zone sur laquelle il est d fini. Pour ceci, `ELISE` offre la possibilit  de donner des comportements coh rents en sp cifiant que l'on veut une `Fonc_Num` qui vaille les  l ments du tableau l  o  il est d fini et un certain prolongement ailleurs. Les deux exemples de code des figures 1.18 et 1.19 illustrent ceci ; on appelle `d` l'adresse du premier  l ment de `I` :

- dans `TEST_plot_Im1`, l'expression `I.in(4.5)` d crit une fonction qui vaut `d[x]` pour  $0 \leq x < 40$  et la valeur par d faut 4.5 sinon ;
- dans `TEST_plot_Im2`, l'expression `I.in_proj()` d finit un prolongement par continuit  de `I.in()` ; dans ce cas de dimension 1, il s'agit de la fonction qui vaut `d[0]` si  $x < 0$ , `d[x]` si  $0 \leq x < 40$  et `d[39]` si  $x \geq 40$  ;

## 1.7 Un premier bilan

- \* visualiseur de donn es : tous `Output`.
- \* tableau auquel on acc de par `input output` ;
- \* toutes les erreurs dynamiques v rifi es.
- \* `ELISE` offre : une fonction `ELISE.COPY`, qq objets primitifs, bcp d'op rateur, puissance vient surtout de l'homog n  (le r sultat des op rateur pouvant servir d'argument en entree aux autres op rateurs) ;



## Chapitre 2

# Manipulation d'images

La section 1.2 a donné un premier aperçu du style de programmation sous *Elise* en se limitant à des traitements sur des signaux de dimension 1. Cette section s'intéresse aux images qui sont les données pour lesquelles *Elise* a été développée. Si certains des exemples présentés introduisent de nouveaux concepts, on verra aussi qu'un certain nombre ne sont qu'un prolongement naturel à la dimension 2 de ce que l'on a vu en 1.2.

Contrairement à la section 1.2 précédente qui se voulait un exposé un peu minimaliste sur les principaux concepts, on trouvera ici des exemples de codes qui n'introduisent aucune nouvelle fonctionnalité et ont pour but d'illustrer les possibilités offertes par le modèle.

Le code complet des exemples de cette section se trouve dans le fichier "**applis/doc\_ex/introd2.cpp**". Le programme **main** commence par les lignes de la figure 2.1. Les deux différences par rapport à la section 1.2 sont :

- d'une part le fait que l'on crée plusieurs palettes de couleurs; on reviendra là dessus dans la suite de cette section;
- d'autre part le fait que l'on ne crée pas de plotter car pour les images c'est directement la fenêtre W qui va être manipulée.

<pre>// sz of images we will use  Pt2di SZ(256,256);  // palette allocation  Disc_Pal Pdisc = Disc_Pal::P8COL(); Gray_Pal Pgr (30); Circ_Pal Pcirc = Circ_Pal::PCIRC6(30); RGB_Pal  Prgb (5,5,5);</pre>	<pre>Elise_Set_Of_Palette SOP(newl(Pdisc)+Pgr+Prgb+Pcirc);  // Creation of video windows  Video_Display Ecr((char *) NULL); Ecr.load(SOP); Video_Win W (Ecr,SOP,Pt2di(50,50),               Pt2di(SZ.x,SZ.y));</pre>
---	--

FIG. 2.1 – code initialisant certains objets

## 2.1 premiers essais

Le code de la figure 2.2 est notre premier exemple de manipulation “d'image”, commentons donc l'ensemble de ces arguments :

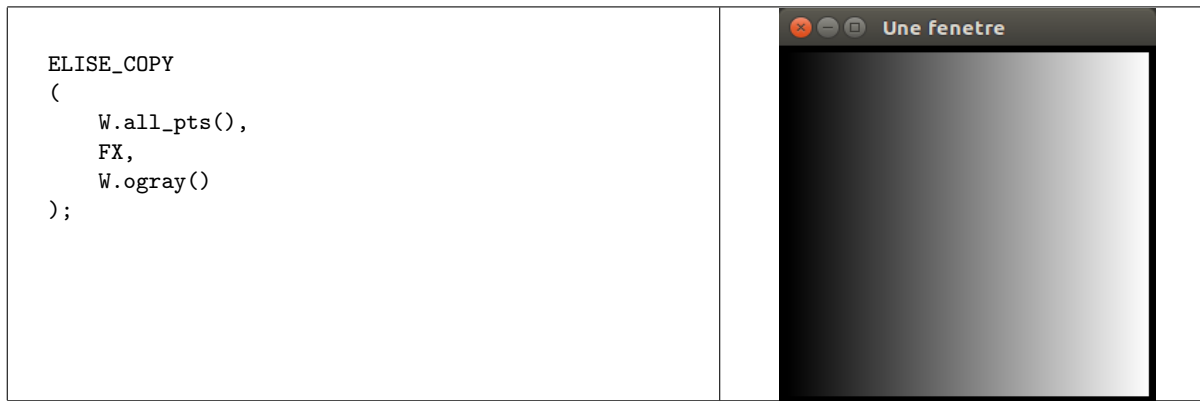


FIG. 2.2 – Code pour visualiser, en mode raster en niveaux de gris, la fonction  $(x, y) \rightarrow x$  sur le carré  $[0\ 256[ \times [0\ 256[$ . Aspect de  $W$  à l'issue de l'exécution de ce code.

- `W.all_pts()` décrit un flux de points correspondant à l'ensemble des pixels de la fenêtre  $W$ ; il s'agit donc du rectangle  $[0\ 256[ \times [0\ 256[$ ;
- `FX` est toujours la fonction première coordonnée; dans ce contexte où les points sont des éléments de  $\mathbb{Z}^2$ , il s'agit donc de la fonction  $(x, y) \rightarrow x$ ;
- `W.ogray()` renvoie un `Output` associé à la fenêtre; de manière générale, quand on considère une fenêtre en `Output`, sa fonction `update(pt, v)` a pour comportement de colorier le pixel `pt` en la couleur associée à `v`; la fonction membre `ogray` indique “classiquement” que la couleur associée à une valeur `v` est un niveau de gris d'intensité proportionnelle à `v` (avec  $0 \rightarrow \text{noir}$  et  $255 \rightarrow \text{blanc}$ ).

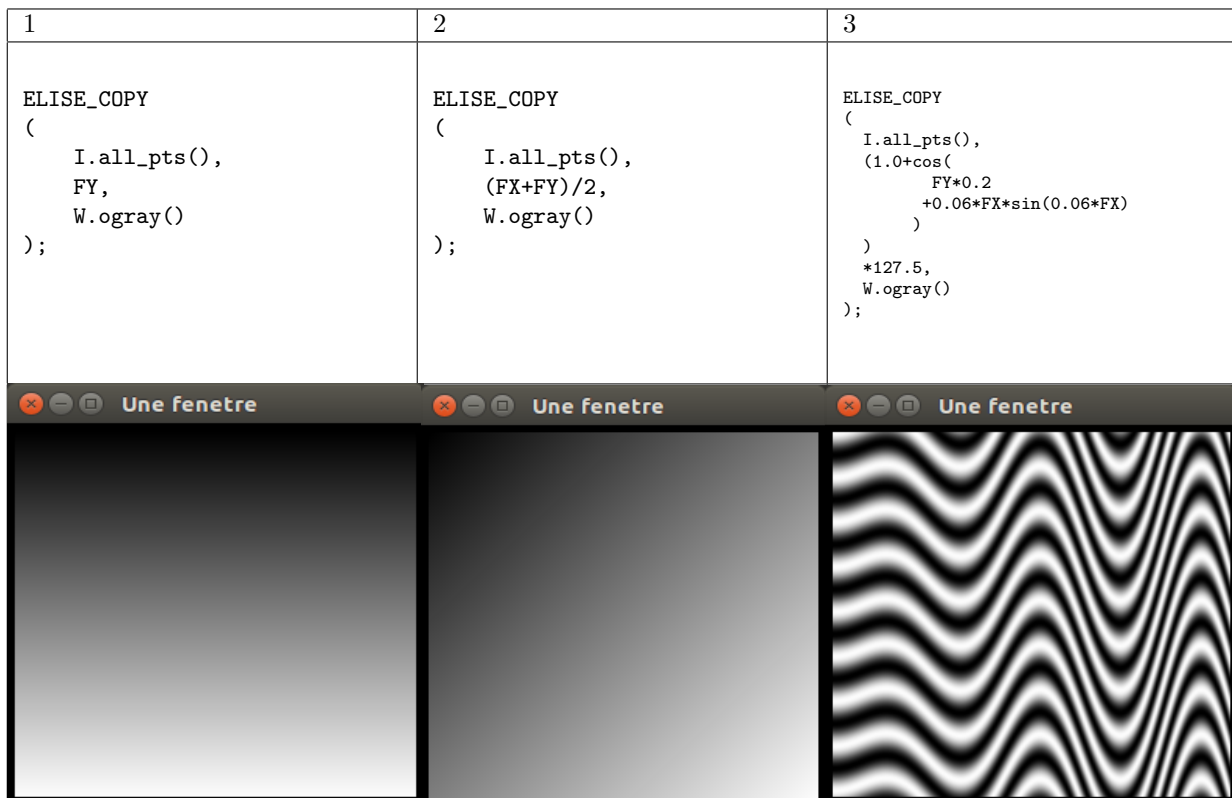


FIG. 2.3 – Exemple utilisant `FX`, `FY` et des opérateurs arithmétiques. Ligne du bas : aspect de  $W$  à l'issue.

La figure 2.3 représente le résultat de l'exécution de `ELISE_COPY(W.all_pts(),fonc,W.ogray())` pour trois valeurs différentes de `fonc`. Commentons brièvement :

1. on utilise la variable globale `FY` définie par `ELISE`; ceci permet de référencer la fonction deuxième

- coordonnée  $(x, y) \rightarrow y$  ;
- les opérateurs arithmétiques sur les `Fonc_Num` vus en dimension 1 restent évidemment valables et l'expression  $(FX+FY)/2$  définit ici la fonction  $(x, y) \rightarrow \frac{x+y}{2}$  ;
  - on peut toujours créer des expressions arbitrairement compliquées et ici  $127.5 * (1.0 + \cos(FY*0.2 + 0.06*FX*\sin(0.06*FX)))$  désigne la fonction  $(x, y) \rightarrow 127.5 * (1.0 + \cos(y * 0.2 + 0.06 * x * \sin(0.06 * x)))$ .

## 2.2 Images, Images ! Nous voulons des images !



FIG. 2.4 – Declaration d'une image tiff, creation d'un tableau 2 – D, chargement de l'image tiff dans le tableau et la fenêtre.

Étant voulant être une bibliothèque de manipulation d'images, il est temps, enfin, de commencer à introduire nos premières "vraies" images. Le code de la figure 2.4, présente l'affichage de notre première image.

Commentons d'abord la ligne `Tiff_Im FLena("DOC/mini_lena.tif");` ;

- on initialise un objet `FLena` de type "Image Tiff" (en prenant en paramètre son nom sur le disque) ;
- nous reportons aux chapitres adéquats les détails sur la manipulation de fichier ; l'essentiel à retenir est que sous ÉLISE un objet fichier image se manipule à peu près comme un tableau ; en première approximation, si l'on sait manipuler un image-tableau sous ÉLISE alors on sait manipuler un fichier image ;
- cette similarité fonctionnelle n'est d'ailleurs pas fortuite car conceptuellement une image (en RAM) et un fichier image correspondent au même type d'objet : des données maillées stockées dans une zone mémoire (RAM ou disque) et accessible en lecture-écriture ;
- les fichiers images possèdent une fonction membre `in()` qui renvoie une `Fonc_Num` ; cette `Fonc_Num` renvoie en chaque point la valeur de l'image en ce point ;
- les fichiers images possèdent <sup>1</sup> une fonction membre `out()` ; l'effet de `update(pv)` sur cet `Output` est de mettre dans le fichier la valeur `v` au point `p` ;

<sup>1</sup>sous réserve qu'ils ne soient pas compressés

Le reste du code est essentiellement une généralisation de ce qui a été vu en dimension 1 :

- `Im2D_U_INT1 I(256,256)` crée un tableau  $2 - D$  de taille  $256 \times 256$  dont les éléments sont des entiers non signés représentés sur 1 octet ;
- par généralisation immédiate du cas  $1 - D$ , soit  $I$  un tableau  $2 - D$ , si  $d$  est la zone mémoire où sont stockées les valeurs, alors `I.in()` décrit la fonction  $(x, y) \rightarrow d[y][x]$  et `I.out().update(p,v)` effectue  $d[p.y][p.x] = v$  ;
- `ELISE_COPY(I.all_pts(), FLena.in(), I.out() | W.ogray())` ; a pour effet de charger le contenu du fichier image à la fois dans le tableau `I` et dans la fenêtre `W`.

Remarquons que cette utilisation de l'opérateur `|` pour, en parallèle, transférer le contenu d'un fichier dans un tableau et visualiser dans une fenêtre est d'usage très courant dans la pratique.

L'image de Lena est maintenant stockée dans le tableau `I` et nous l'utiliserons dans les exemples qui suivent.

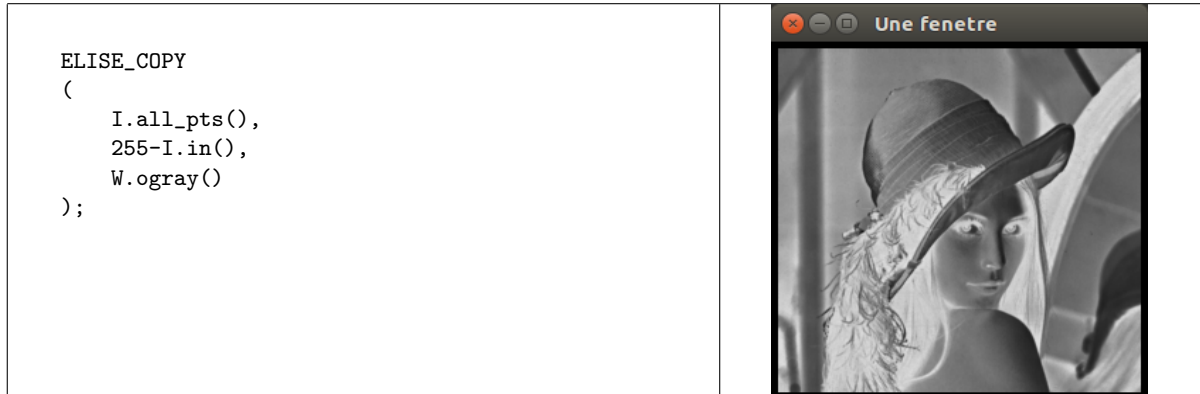


FIG. 2.5 – Expression arithmétique sur les images, affichage de Lena en négatif.

La figure 2.5 est notre premier exemple d'utilisation d'une image dans une expression arithmétique. Dans l'expression `255-I.in()` les valeurs sont inversées par rapport à l'image initiale, ce qui donne cette image en négatif (`255-I.in()` représente la fonction  $(x, y) \rightarrow 255 - d[y][x]$ )

## 2.3 Palette

Quand on écrit dans une fenêtre `W`, c'est par des objets de type `Elise.Palette`<sup>2</sup> que l'on peut contrôler la façon dont les valeurs numériques sont converties en couleur. Il suffit pour cela d'appeler la fonction membre `out` avec un paramètre de type palette ; la figure 2.6 donne trois exemples d'utilisation de palettes :

1. on écrit dans `W.out(Pgr)`, où `Pgr` est une palette en niveau de gris (elle a été créée par `Gray_Pal Pgr (30)`<sup>3</sup> ; on obtient la même image que sur la figure 2.4 (quand on écrit `W.ogray()` `Elise` va rechercher la palette niveau de gris active) ;
2. on écrit, la même fonction que précédemment, dans `W.out(Pcirc)`, où `Pcirc` est une palette de "teinte" : les valeurs sont interprétées comme des angles définissant une couleur sur le "cercle de couleur de l'arc ciel" (voir figure 9.1).
3. on écrit dans `W.out(Pdisc)` où `Pdisc` est une palette discrète (`Disc_Pal Pdisc = Disc_Pal : :P8COL()`) ; dans une palette discrète, on a indiqué de manière explicite quelle couleur est associée à chaque valeur numérique (ici  $0 \rightarrow$  blanc,  $1 \rightarrow$  noir,  $2 \rightarrow$  rouge,  $3 \rightarrow$  blanc ...).

La figure 2.7 donne plusieurs exemples de binarisation de lena, dans tous ces exemples on utilise une palette discrète pour voir le résultat en noir et blanc. Commentons rapidement :

1. on effectue une binarisation classique à seuil fixe (ici 128) ;
2. on effectue une binarisation avec un seuil variable, on obtient ainsi une image qui s'assombrit vers la droite ;

<sup>2</sup>ou plutôt des objets de classes dérivées

<sup>3</sup>le rôle du paramètre 30 sera décrit dans au chapitre 9.2.7, pour l'instant c'est secondaire et on peut ignorer tous les paramètres de création de palette



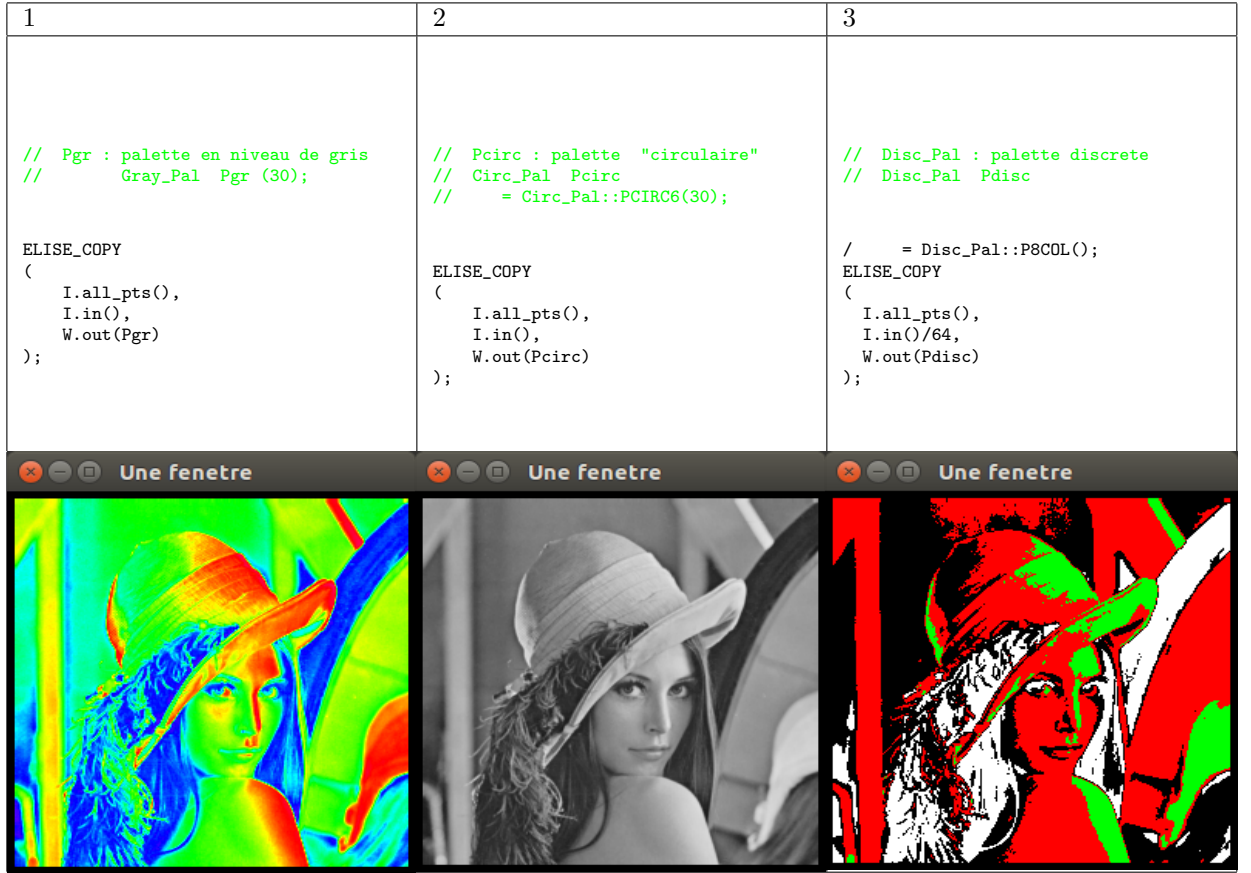


FIG. 2.6 – Utilisation de plusieurs palettes.

3. on effectue une binarisation en comparant l'image avec une fonction périodique, on obtient ainsi un effet de tramage;

## 2.4 Opérateur “,” sur les Fonc\_Num, changement de coordonnées

Toutes les fonctions que nous avons vues pour l'instant sont du type  $\mathcal{Z}^k \rightarrow \mathcal{Z}$ . Pour créer des fonctions de  $\mathcal{Z}^k$  dans  $\mathcal{Z}^p$  sous  $\mathcal{E}\mathfrak{L}\mathfrak{S}\mathfrak{E}$ , on peut se servir de l'opérateur “,” (opérateur virgule). Informellement,  $(f_1, f_2)(x)$  est le point obtenu par simple concaténation des coordonnées de  $f_1(x)$  et  $f_2(x)$ . Plus précisément :

- soit  $f_1$  une fonction de  $\mathcal{Z}^k$  dans  $\mathcal{Z}^{p_1} : (x_1 \dots x_k) \rightarrow (y_1 \dots y_{p_1})$ ;
- soit  $f_2$  une fonction de  $\mathcal{Z}^k$  dans  $\mathcal{Z}^{p_2} : (x_1 \dots x_k) \rightarrow (z_1 \dots z_{p_2})$ ;
- alors  $(f_1, f_2)$  est la fonction de  $\mathcal{Z}^k$  dans  $\mathcal{Z}^{p_1+p_2} : (x_1 \dots x_k) \rightarrow (y_1 \dots y_{p_1} z_1 \dots z_{p_2})$
- on dira que  $p + q$  est la dimension de sortie de la **Fonc\_Num**  $(f_1, f_2)$ .

**Définition 2.1** *dimension de sortie* On appelle dimension de sortie d'une **Fonc\_Num** la dimension de son espace d'arrivée.

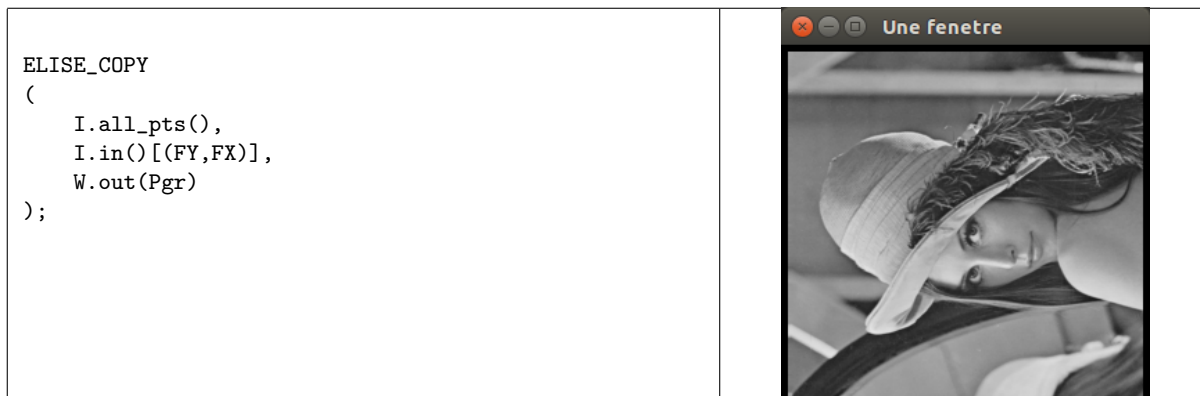
La figure 2.8 donne un exemple d'utilisation de l'opérateur “,” pour effectuer une transformation géométrique simple ; détaillons en rappelant un certains nombres d'éléments déjà vus :

- **FX** est la fonction  $(x, y) \rightarrow (x)$  et **FY** est la fonction  $(x, y) \rightarrow (y)$  ;
- **(FY,FX)** est donc la fonction  $(x, y) \rightarrow (y, x)$  ;
- **I.in()** est la fonction qui renvoie *Lena* en  $(x, y)$
- **I.in()[(FY,FX)]** désigne la composition des fonctions **I.in()** et **(FY,FX)** ; il s'agit donc de la fonction qui en  $(x, y)$  renvoie *Lena* en  $(y, x)$  ;

Sans cet opérateur, les transformations géométriques telles que celles de la figure 2.8 nous auraient été inaccessibles sous  $\mathcal{E}\mathfrak{L}\mathfrak{S}\mathfrak{E}$ . Si par exemple, on essayait d'effectuer **ELISE\_COPY** avec **I.in() [FY]**,  $\mathcal{E}\mathfrak{L}\mathfrak{S}\mathfrak{E}$



FIG. 2.7 – Quelques exemples de binarisation (fixe, progressive et tramage).

FIG. 2.8 – Opérateur “,” utilisé pour créer un fonction  $\mathbb{Z}^2 \rightarrow \mathbb{Z}^2$  et utilisation pour effectuer un changement de coordonnées.

génèrerait (légitimement !) une erreur : `I.in()` nécessite des points de dimension 2 tandis que `FY` génère des points de dimension 1.

La figure 2.9 donne plusieurs exemples d'utilisation de l'opérateur “,” pour effectuer des transformations géométriques :

1. un exemple d'homothétie-rotation ; noter l'utilisation de `I.in(0)` qui permet “de ne pas se soucier des problèmes de débordement” ;
2. un exemple de mosaïque ;
3. un exemple de morphing, les paramètres ont été choisis un peu au hasard ;

Pas de commentaire particulier sur la figure 2.10. Il s'agit d'un exemple conjuguant différentes opérations déjà vues. Maintenant que nous commençons à avoir vu suffisamment de manipulation non triviales sous `Elise`, on rajoutera de plus en plus de petits exemples de ce type. On suggère au lecteur de s'arrêter quelques secondes pour vérifier qu'il comprend pourquoi l'exécution du code fournit l'image présentée.

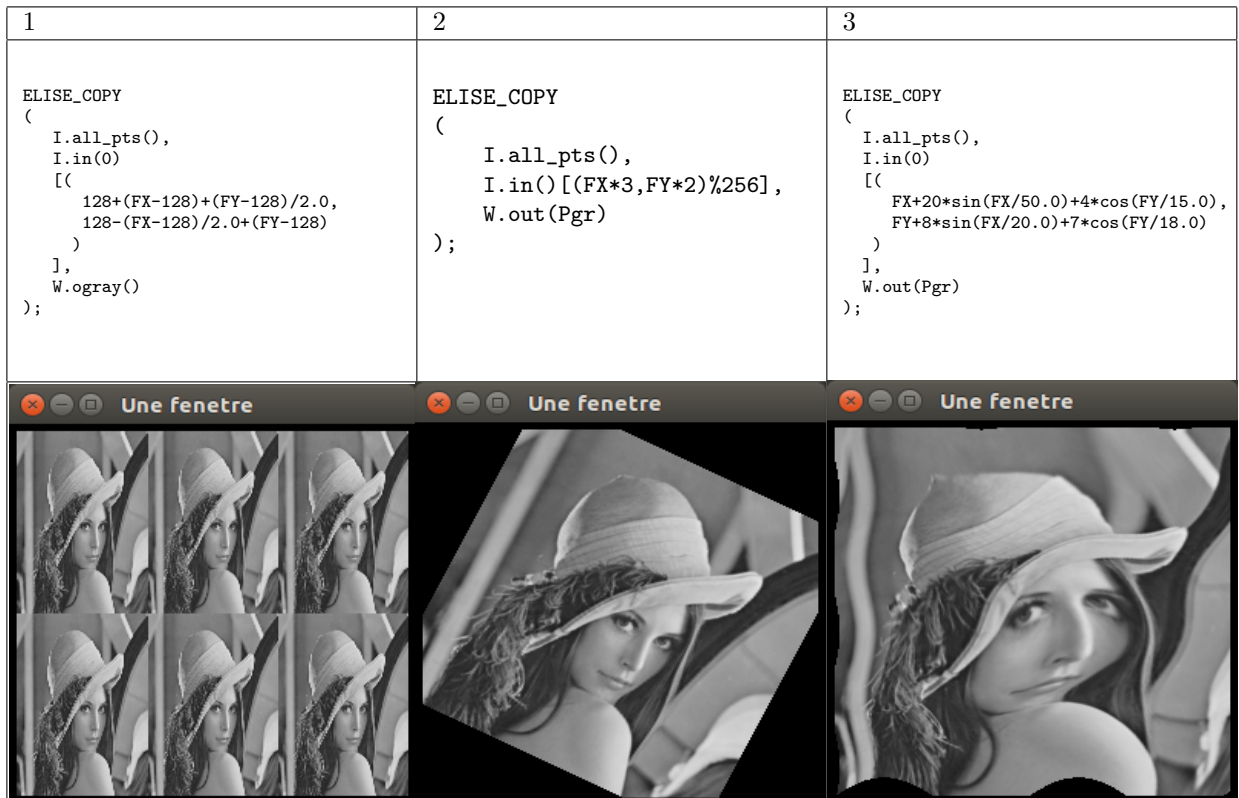


FIG. 2.9 – Exemple de changement de coordonnées.



FIG. 2.10 – Exo.

## 2.5 Transformation radiométrique

En section 2.4 nous avons utilisé l'opérateur de composition pour effectuer des transformations géométriques. Dans cette section, nous allons voir que cet opérateur peut aussi être utilisé pour effectuer des transformations radiométriques.

Cette utilisation de `[]` peut correspondre à des soucis d'optimisation de la vitesse d'exécution comme illustré par les 2 premières colonnes de la figure 2.11 :

- en colonne 1 de la figure 2.11, on effectue, sans aide de l'opérateur `[]`, une gamma-correction  $\frac{5}{3}$  de *Lena* ; cela fonctionne parfaitement mais on voit qu'il est fait appel à l'opérateur `pow` autant de fois qu'il y a de points dans l'image ;
- si on ne veut pas payer le coût d'un appel à `pow` en chaque point de l'image, une technique classique est de remarquer que, *ici*, les valeurs de l'image sont entières et comprises entre 0 et 256 ; il suffit donc de mémoriser les 256 valeurs possibles de la gamma correction dans un tableau et d'utiliser ce




1	2	3
<pre>// gamma-correction "directe" // (sans utiliser de look-up-table)  ELISE_COPY (   I.all_pts(),   pow(I.in()/255.0,5.0/3.0)*255.0,   W.ogray() );</pre>	<pre>// Initialisation de la look- //up-table (table de transition)  Im1D_U_INT1 lut(256); ELISE_COPY (   lut.all_pts(),   pow(FX/255.0,3.0/5.0)*255.0,   lut.out() );  // Utilisation de lut pour // tabuler la gamma-correction  ELISE_COPY (   I.all_pts(),   lut.in()[I.in()],   W.ogray() );</pre>	<pre>// deuxieme parametre (0) pour //initialiser tous les elements de lut  Im1D_U_INT1 lut(256,0); U_INT1 * d = lut.data(); d[1] = d[2] = d[5] = 1; d[10] = d[13] = d[14] = 2; ELISE_COPY (   I.all_pts(),   lut.in()[I.in()/16],   W.odisc() );</pre>
		

FIG. 2.11 – Utilisation de l'opérateur de composition `[]` pour effectuer rapidement ("tabulation") des transformations radiométriques.

tableau pour effectuer la correction de l'image; on remplace ainsi le coût d'un appel à `pow` par un simple déréférencement de tableau;

- cette technique de tabulation est utilisée en colonne 2 de la figure 2.11 pour effectuer une gamma-correction de  $\frac{3}{5}$ ;  
`ELISE_COPY(lut.all_pts(),pow(FX/255.0,3.0/5.0)*255.0,lut.out())` effectuée pour  $0 \leq x < 256$ , on effectue  $lut[x] = \frac{x}{255}^{\frac{3}{5}} * 255$ ; ensuite on utilise `lut` pour corriger l'image.

Il existe aussi des cas où, indépendamment de considérations de temps de calcul, l'utilisation de `[]` est la façon la plus simple de spécifier une correction radiométrique qui ne peut pas s'exprimer naturellement par une formule mathématique; c'est, par exemple, le cas pour les égalisation d'histogrammes (voir 6.1). La colonne 3 de la figure 2.11 donne un exemple où `[]` est la façon la plus simple de spécifier la fonction : "1 si  $\frac{I}{16} \in \{1, 2, 5\}$ ; 2 si  $\frac{I}{16} \in \{10, 13, 14\}$ ; 0 sinon".

Remarquer aussi dans l'exemple de la troisième colonne que l'on donne un deuxième argument au constructeur de `lut`, ici il permet d'initialiser tous les éléments à 0 (par défaut `ELISE` n'effectue pas d'initialisation des tableaux). De la même manière les tableaux 2 –  $D$  prennent en paramètre un troisième argument optionnel, les tableaux 3 –  $D$  un quatrième ...

## 2.6 Palette-RGB, fonctions de projections

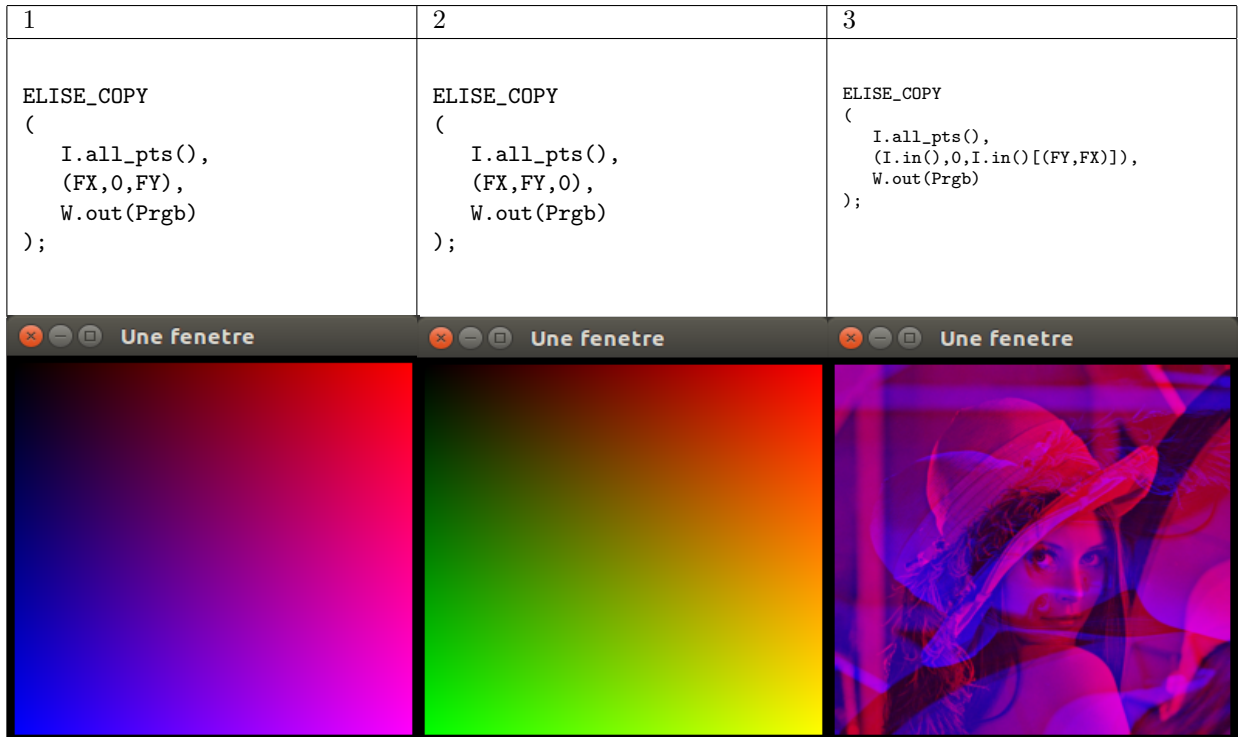


FIG. 2.12 – Utilisation de palettes en RVB.

La section 2.3 a donné trois exemples d'utilisation de palettes (niveau de gris, circulaire et discrète). Ces 3 palettes différentes possèdent un point commun, elles ne nécessitent que un paramètre pour déterminer une couleur. Pour beaucoup de visualisation c'est insuffisant et de manière générale, une palette  $\mathcal{E}\mathcal{L}\mathcal{S}\mathcal{E}$  spécifie un "mapping" de  $\mathcal{R}^k$  vers l'espace des couleurs (où  $k$  est la dimension de la palette, jusqu'à présent  $k = 1$ ). Si  $W$  est une fenêtre et  $Pal$  une palette de dimension  $k$ , alors lorsque l'on écrit dans  $W.out(Pal)$  une fonction  $f$  :

- $\mathcal{E}\mathcal{L}\mathcal{S}\mathcal{E}$  exige que  $f$  soit aussi de dimension de sortie  $k$  (ou plus exactement de dimension de sortie au moins égale à  $k$ )
- pour chaque pixel  $p$ , les  $k$  composantes de  $f(p)$  sont utilisées pour calculer la couleur de  $Pal$  en laquelle il faut colorier  $p$ .

**Définition 2.2** *dimension d'une palette* Une palette de dimension  $k$  spécifie un "mapping" de  $\mathcal{R}^k$  vers l'espace des couleurs.

Par exemple les palette RGB sont de dimension 3, les palettes niveaux de gris et les palette discrètes (ou indexée) sont de dimension 1.

L'exemple le plus classique est celui des palette RVB où un triplet de valeur spécifie une couleur par son intensité dans les trois canaux rouge, vert et bleu. La figure 2.12 illustre l'utilisation d'une palette  $Prgb$  de type  $RGB\_Pal$ . Quelques commentaires :

- dans l'exemple de gauche on écrit la fonction  $(FX,0,FY)$  ; en chaque pixel  $(x,y)$ , on obtient une couleur dont la codification RVB est  $(x,0,y)$  ;
- notons que si l'on avait cherché à écrire la fonction  $(FX,FY)$ ,  $\mathcal{E}\mathcal{L}\mathcal{S}\mathcal{E}$  aurait (légitimement) généré une erreur car un couple de valeur ce n'est pas suffisant pour déterminer une valeur en RVB ;
- l'exemple du milieu et celui de gauche sont laissés comme exercice.

$\mathcal{E}\mathcal{L}\mathcal{S}\mathcal{E}$  fournit des palettes de dimension 1, 2 et 3.

Les exemples de la figure 2.12 correspondaient à une utilisation un peu artificielle des palettes RGB. En pratique les palettes RGB sont surtout utilisées pour visualiser des images directement acquise en RGB. Dans l'exemple de la figure 2.13, le fichier "DOC/lena.col.tif" contient une image tiff RGB que l'on copie directement dans  $W.out(Prgb)$  ; l'exemple devrait être assez naturel et "techniquement" la seule





FIG. 2.13 – Utilisation d’une palette RGB pour visualiser Lena en couleur.

chose à retenir est qu’il n’y a pas de problème de cohérence, car lorsqu’un fichier image en RGB est converti en `Fonc_Num` par `Elise`, sa dimension de sortie est 3 (de manière générale sa dimension de sortie est égale au nombre de canaux du fichier).

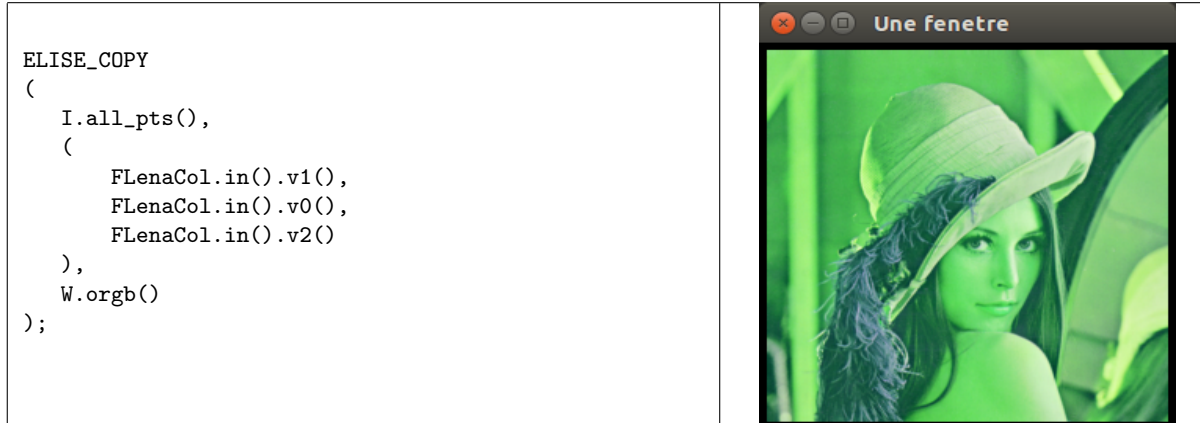


FIG. 2.14 – Utilisation des opérateurs de “projection” pour manipuler les canaux d’une image indépendamment les uns des autres.

En section 2.4, l’opérateur virgule nous a permis de créer des fonctions de dimension de sortie quelconque à partir de fonction de dimension de sortie valant 1. On a besoin parfois de l’opération “inverse” afin de pouvoir isoler telle ou telle composante d’une fonction. Pour ceci, `Elise` fournit les opérateur `kth_proj(int)`, `v0()`, `v1()` et `v2()` :

- si  $f$  est une fonction de dimension de sortie  $n : p \rightarrow f(p) = (v_0, \dots, v_{n-1})$  alors `f.kth_proj(k)` est la fonction de dimension de sortie 1  $p \rightarrow f(p) = (v_k)$ ; en d’autre terme `f.kth_proj(k)` est la projection de  $f$  sur le  $k^{\text{ème}}$  sous-espace; naturellement le résultat ne peut être cohérent que si  $k < n$ , sinon une erreur est déclenchée;
- `f.v0()`, `f.v1()` et `f.v2()` sont simplement des “raccourcis” pour `f.kth_proj(0)` (`f.kth_proj(1)` et `f.kth_proj(2)`) correspondant aux cas les plus fréquents;

- l'exemple de la figure 2.14 utilise ces opérations pour afficher *Lena* en intervertissant les canaux rouge et vert.

## 2.7 Flux en dimension 2

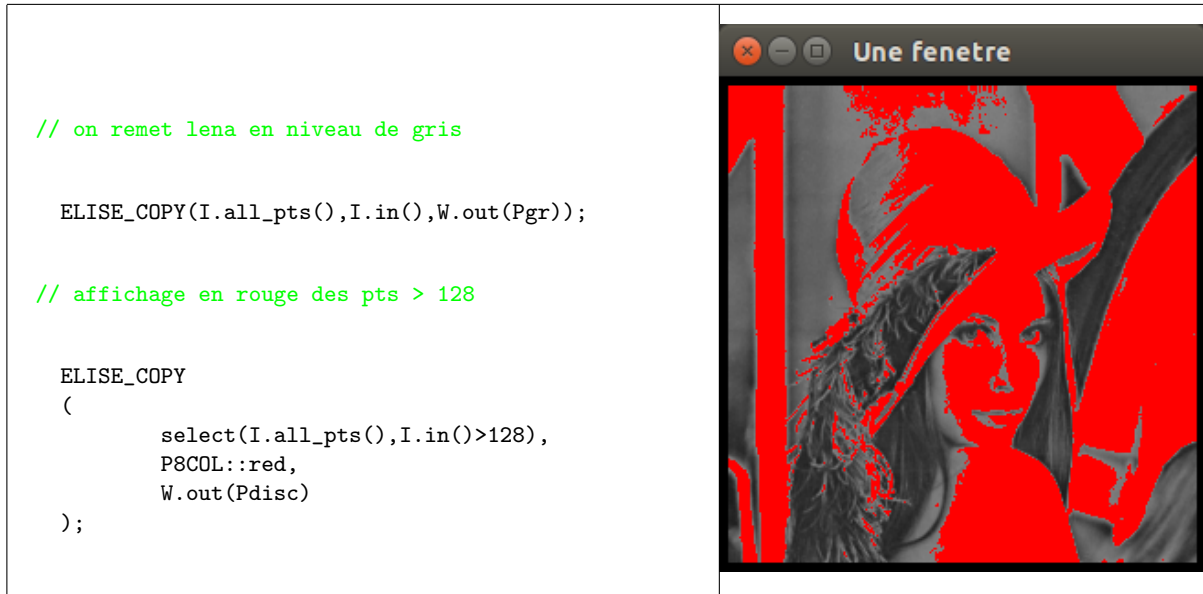


FIG. 2.15 – Illustration de l'opérateur `select`.

La figure 2.15 donne une illustration de l'opérateur `select` en dimension 2. Tous les opérateurs vus en  $1 - D$  se généralisent au  $2 - D$ .

En  $2 - d$ , `Elise` offre une certaine variété de flux. Le code de la figure 2.16 (principales étapes visualisées sur la figure 2.17) en donne quelques exemples. Compte-tenu de ce que l'on a déjà vu, la correspondance entre le code et les images obtenues devrait être assez naturelle. Quelques commentaires :

- il existe des fonctions permettant de décrire, sous forme de flux, des formes géométriques de type “surfaces” (ici rectangles, disques, ellipses, polygones, secteurs angulaires) ;
- il existe des fonction permettant de décrire sous forme de flux des formes géométrique de type courbes (ici droites, contour d'ellipse, contours de rectangle) ;
- la fonction `border_rect(int ep)` permet de décrire sous forme de `Flux_Pts` les bords d'un rectangle ; le paramètre `ep` permet de spécifier l'épaisseur de ce bord de rectangle ;
- la fonction membre `border(int ep)` permet de décrire sous forme de `Flux_Pts` les bords d'un objets tels que fenêtres, images ...elle se contente de rappeler la fonction `border_rect(int ep)` avec les paramètres adéquats (de même que `all_pts` rappelle `rectangle`).
- dans `(I.in()/64)*64` comme la division est entière, on obtient simplement un arrondi à 64 ;

## 2.8 Virgule sur Out

*Cette section aborde un des points les plus délicats vus jusqu'à présent.*

En section 2.6 nous avons affiché directement un fichier RGB dans une fenêtre associée à une palette en RGB ; cela ne posait pas de problème particulier car les dimensions des deux objets étaient “compatibles” au sens où un fichier image RGB renvoie des valeurs sur 3 canaux et la fenêtre associée à une palette RGB attend des valeurs sur 3 canaux .

Supposons maintenant que nous disposions de trois fenêtres  $Wr$ ,  $Wg$  et  $Wb$  et que nous voulions, en une seule instruction, visualiser en niveau de gris, dans chacune de ces trois fenêtres, le contenu des trois canaux de l'image (pour obtenir les trois images de la figure 2.18) ; nous avons déjà vu suffisamment d'opérateur `Elise` pour le faire et le code de gauche de la figure 2.19 donne une solution possible en

<pre> ELISE_COPY(I.all_pts(),P8COL::blue,W.out(Pdisc)); ELISE_COPY (     disc(Pt2di(128,128),100),     I.in(),     W.out(Pgr) );  // FIN Etape 1  ELISE_COPY (     ell_fill(Pt2di(128,128),135,70,1.2),     255-I.in(0),     W.out(Pgr) ); ELISE_COPY (     sector_ang(Pt2di(128,128),100,1.0,3.0),     (I.in() &lt; 128),     W.out(Pdisc) );  // FIN Etape 2  for (INT x = 0; x &lt; 256; x+= 4)     ELISE_COPY     (         line(Pt2di(x,0),Pt2di(128,128)),         I.in(),         W.out(Pgr)     ); ELISE_COPY (     polygone     (         newl(Pt2di(5,250))+Pt2di(128,5)         + Pt2di(250,250)+Pt2di(128,128)     ),     (I.in()/64)*64,     W.out(Pgr) );  // FIN Etape 3 </pre>	<pre> for (INT x = 1; x&lt; 5; x++)     ELISE_COPY     (         border_rect         (             Pt2di(10,10),             Pt2di(15+10*x,15+10*x),             5-x         ),         P8COL::white,         W.out(Pdisc)     );  ELISE_COPY (     W.border(8),     P8COL::green,     W.out(Pdisc) ); ELISE_COPY (     ellipse(Pt2di(128,128),135,70,1.2),     P8COL::red,     W.out(Pdisc) );  // FIN Etape 4 </pre>
--	--

FIG. 2.16 – Code illustrant quelques fonctions permettant de décrire sous forme de **Flux\_Pts** des primitives géométriques.

utilisant les opérateurs `v0()` ... `v2()` (section 2.6) pour accéder aux différents canaux et l'opérateur `<<` (section 1.5.2) pour rediriger les fonctions sur la bonne fenêtre.

Cependant la solution de gauche la figure 2.19 n'est pas très satisfaisante :

- c'est un peu lourd à écrire pour une manœuvre que l'on peut supposer assez courante ;
- c'est assez inefficace car le fichier est en fait lu 3 fois (ce dernier inconvénient peut être contourné si on utilise les **Symb\_FNum** qui seront décrits plus loin) ;

La colonne de droite de la figure 2.19 donne le même effet que la colonne de gauche mais de manière plus concise et plus efficace grâce à l'utilisation de l'opérateur “,” sur les **Output**. Voyons informellement comment ça marche :

- `(Wr.out(Pgr), Wg.out(Pgr), Wb.out(Pgr))` est un **Output** qui attend des triplets de valeurs ;
- sa méthode `update(p,v)` avec `v = (a,b,c)` redistribue le message en appelant successivement :
  - `(Wr.out(Pgr).update(p,a),`
  - `(Wg.out(Pgr).update(p,b),`
  - `(Wb.out(Pgr).update(p,c),`





FIG. 2.17 – Résultat de Lena après le code de la figure précédente.



FIG. 2.18 – Visualisation en niveaux de gris des 3 canaux de lena en couleur

– chaque triplet  $(r, g, b)$  généré par `FLenaCol.in()` est donc éclaté en 3 valeurs qui sont envoyées sur

<pre> Video_Win Wr = W; Video_Win Wg (Ecr,SOP,Pt2di(50,50),SZ); Video_Win Wb (Ecr,SOP,Pt2di(50,50),SZ);  // VERSION "BOVINE"  ELISE_COPY (   W.all_pts(),   0,   (Wr.out(Pgr) &lt;&lt; FLenaCol.in().v0())   (Wg.out(Pgr) &lt;&lt; FLenaCol.in().v1())   (Wb.out(Pgr) &lt;&lt; FLenaCol.in().v2()) ); </pre>	<pre> // VERSION OPERATEUR "," sur OUTPUT  ELISE_COPY (   W.all_pts(),   FLenaCol.in(),   (Wr.out(Pgr),Wg.out(Pgr),Wb.out(Pgr)) ); </pre>
--	---

FIG. 2.19 – Visualisation de chaque canal de *Lena-couleur* en niveau de gris, sans (colonne gauche) et avec (colonne droite) l'opérateur “,” sur les **Output**.

chacune des 3 fenêtres.

Pour étudier de manière précise le comportement de l'opérateur “,” sur les **Output** nous allons devoir introduire un peu de formalisme.

**Définition 2.3** *dimension consommée.*

Chaque **Output**  $O$  possède une dimension consommée  $D^c(O)$ . Cette valeur  $D^c(O)$  correspond au nombre de “canaux” nécessaires pour écrire dans cet **Output**.

Lorsque l'on écrit avec une fonction  $f$  dans  $O$ , la dimension de sortie de  $f$  doit être supérieure ou égale à la dimension consommée de  $O$ .

Donnons quelques exemples sur des **Output** primitifs :

- $D^c(O)$  vaut 1 pour tous les tableaux `Ejse`;
- soit une fenêtre  $W$  et une palette  $Pal$ , alors pour  $W.out(Pal)$   $D^c(O)$  est égale à la dimension de la palette (voir définition 2.2 page 29) ;
- pour un fichier image en écriture  $D^c(O)$  est égale aux nombre de canaux de l'image.

On peut maintenant définir précisément le comportement de l'opérateur “,” :

- soit  $O_1$  et  $O_2$  deux **Output** avec  $d_1 = D^c(O_1)$  et  $d_2 = D^c(O_2)$  ;
- $(O_1, O_2)$  est un **Output** de dimension consommée  $d_1 + d_2$  ;
- soit  $p$  un point et  $v = (x_0, \dots, x_{n-1})$  une valeur avec  $n \geq d_1 + d_2$  ;
- soient  $v_1 = (x_0, \dots, x_{d_1-1})$  et  $v_2 = (x_{d_1}, \dots, x_{d_1+d_2-1})$  ;
- la fonction `update(p,v)` de  $(O_1, O_2)$  se contente alors de faire appelle successivement à  $O_1.update(p, v_1)$  et à  $O_2.update(p, v_2)$  ;
- autrement dit  $O_1$  “consomme” les  $d_1$  premières composantes et  $O_2$  les  $d_2$  composante suivantes ;

L'utilisation d'une expression telle que  $(O_1, O_2, O_3)$  (dans un code comme celui de la figure 2.19) est alors une simple conséquence de la définition précédente et des règles d'associativité :

- on note  $d_1 = D^c(O_1)$ ,  $d_2 = D^c(O_2)$  et  $d_3 = D^c(O_3)$  ;
- d'après les règles du **C++**  $(O_1, O_2, O_3)$  est équivalent à  $((O_1, O_2), O_3)$  ;
- $(O_1, O_2)$  est de dimension consommée  $d_1 + d_2$  ;
- en appliquant la règle on voit donc que  $O_1$  “consomme” les  $d_1$  premières composantes,  $O_2$  les  $d_2$  composante suivantes et  $O_3$  les  $d_3$  suivantes ;
- et ça marche pareille avec  $(O_1, O_2, O_3, O_4, O_5, O_6 \dots)$  !;

Pour spécifier complètement le comportement de l'opérateur “,”, il nous reste à définir les règles permettant de calculer la dimension consommée pour les opérateurs sur les **Output** :

- $D^c(O_1, O_2) = D^c(O_1) + D^c(O_2)$  ; comme on vient de le voir ;
- $D^c(O_1|O_2) = \max(D^c(O_1), D^c(O_2))$  ;  
en effet soit  $n$  le nombre de “canaux” nécessaire pour écrire dans  $O_1|O_2$ , il suffit que  $n$  permette d'écrire dans  $O_1$  et que  $n$  permette d'écrire dans  $O_2$

- $D^c(O \ll f) = 0$ ;  
en effet  $O \ll f$  ne “consomme” aucune des valeurs qui lui sont envoyées (en terme imagés, il est autonome car il génère ses propres valeurs via  $f$ );

#### Remarques

- on voit dans la définition 2.3, que la dimension de sortie de  $f$  n’est pas nécessairement égale à celle de  $O$ . Lorsque elle est supérieure, les coordonnées “excédentaires” sont simplement ignorées; cela peut être pratique parfois;
- pour des raisons liées aux priorités des opérateurs en C++, il est prudent (et en fait quasi obligatoire) de systématiquement mettre entre parenthèses tous les usages de “,” que ce soit sur les **Output** ou sur les **Fonc\_Num**; si par exemple on tentait `ELISE_COPY(flx,func,o1,o2)` le parser du C++ ne verrait pas une application de “,” sur `o1` et `o2` mais une tentative pour appeler `ELISE_COPY` avec 4 arguments. Bien qu’il puisse sembler déconcertant, l’opérateur “,” sur les **Output** est loin d’être seulement une curiosité et il se révèle en fait assez utile. Un cas pratique très courant est celui où l’on veut charger le contenu d’un fichier image RVB dans trois tableaux 2 – D.

Le code de la figure 2.20 donne deux exemples d’emploi simultané des opérateurs “,” et “|”. Ils se trouvent à la fin du fichier “`applis/doc_ex/introd2.cpp`”. Ils effectuent de deux manières différentes la même opération :

- on charge le contenu du fichier `FLenaCol` dans 3 images `R,G` et `B`; en même temps on visualise le résultat dans les 3 fenêtres `Wr,Wg` et `Wb`;
- on vérifie que `R,G` et `B` contiennent ce que l’on attend en visualisant `(R.in(),G.in(),B.in())` avec une palette RGB.

<pre> Im2D_U_INT1 R(256,256); Im2D_U_INT1 G(256,256); Im2D_U_INT1 B(256,256);  // affichage des trois canaux dans Wr, Wg et Wb // et en meme temps dans R,G et B  ELISE_COPY (     W.all_pts(),     FLenaCol.in(),     (Wr.out(Pgr) Wg.out(Pgr) Wb.out(Pgr))       (R.out(),G.out(),B.out()) );  // verif que R,G,B contiennent les bonnes valeurs  ELISE_COPY (     W.all_pts(),     (R.in(),G.in(),B.in()),     W.out(Prgb) ); </pre>	<pre> // une autre facon de faire : // affichage des trois canaux dans Wr, Wg et Wb // et memo en meme temps dans R,G et B  ELISE_COPY (     W.all_pts(),     FLenaCol.in(),     (         Wr.out(Pgr) R.out(),         Wg.out(Pgr) G.out(),         Wb.out(Pgr) B.out()     ) ); ELISE_COPY (     W.all_pts(),     (R.in(),G.in(),B.in()),     W.out(Prgb) ); </pre>
---	---

FIG. 2.20 – Deux façons équivalentes de mémoriser une image couleur dans `R,G` et `B` tout en visualisant les différents canaux.

## 2.9 chc sur les Flux\_Pts

La figure 2.21 illustre l’opérateur `chc`, opérateur de changement de coordonnées sur les **Flux\_Pts**. Donnons la définition de cet opérateur :

- soit  $E$  est un flux de points et  $f$  une fonction de dimension de sortie  $k$ ;
- soient  $p_0, p_1 \dots$ ;

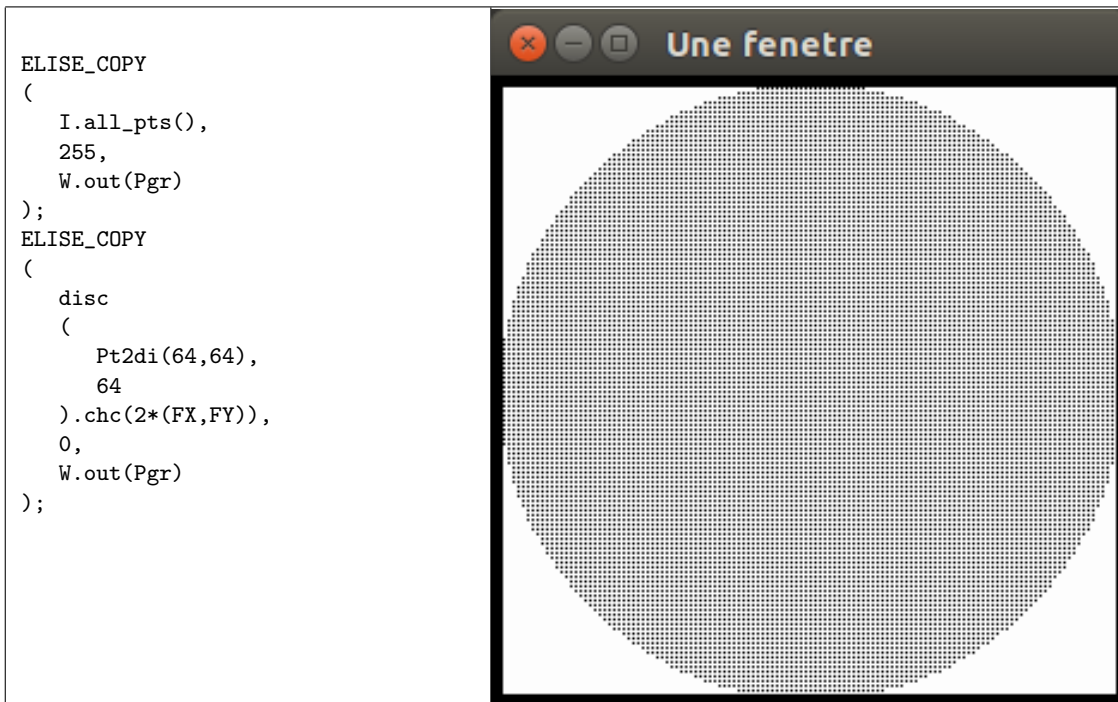


FIG. 2.21 – Illustration de l'opérateur `chc` (changement de coordonnées) sur les `Flux_Pts`

- $E.chc(f)$  est un flux de points de dimension  $k$  qui génère les point  $f(p_0), f(p_1) \dots$ ;
- en terme ensembliste,  $E.chc(f)$  est simplement l'image de  $E$  par la fonction  $f$ ;

Commentons maintenant la figure 2.21 :

- on part du disque de centre  $(64, 64)$  et de rayon  $64$ ;
  - on prend son image par le changement de coordonnées  $(x, y) \rightarrow (2 * x, 2 * y)$
  - on obtient donc les point de coordonnées paires du disque de centre  $(128, 128)$  et de rayon  $128$ ;
- Les exemples des figures 2.22 et 2.23 sont laissés à titre d'exercice.



FIG. 2.22 – Une autre illustration de chc sur les Flux.Pts.



FIG. 2.23 – Encore une autre illustration de chc sur les Flux.Pts.



## Chapitre 3

# Exemples, traitement d'images

Au chapitre 2 nous avons les principales fonctionnalités d'Elise permettant de visualiser, lire, écrire et manipuler (transformations géométriques ou radiométriques) des images. Dans ce chapitre nous allons voir quelques un des mécanismes qu'offrent Elise pour le traitement et l'analyse d'images.

Le fichier contenant les codes de ce chapitre est "applis/doc\_ex/introfiltr.cpp". Comme d'habitude, on commence par initialiser un certain nombre d'objets qui nous serviront au cours de ce chapitre :

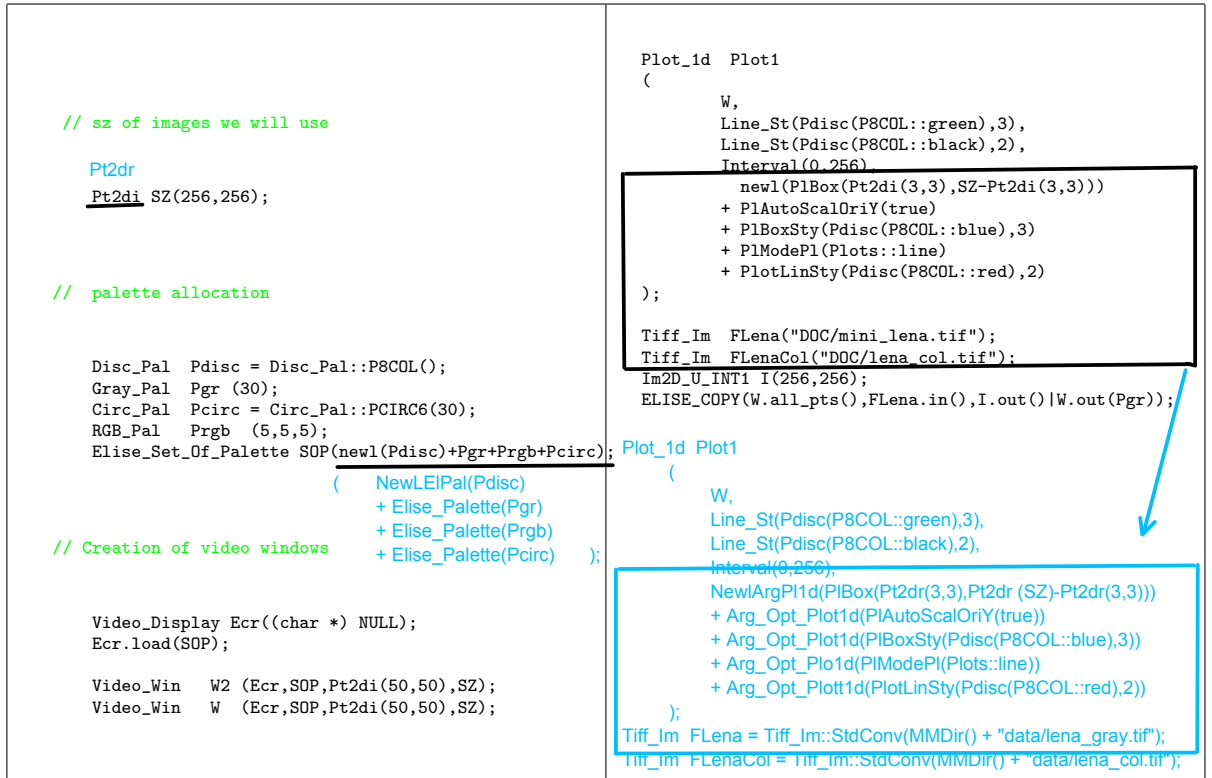


FIG. 3.1 – Initialisation de palettes, fenêtres, ploters, fichier et images.

### 3.1 Filtre élémentaires, Opérateur trans

L'exemple de la figure 3.2 montre qu'en assemblant les opérateurs du chapitre 2, on est déjà capable de définir des petits filtres élémentaires, ici un gradient horizontal. Commentons rapidement (où  $d$  désigne toujours la zone de données) :

- $I.in(0)[(FX+1,FY)] - I.in(0)$  est la fonction  $G_x(x,y) \rightarrow G_x(x,y) = d[y][x+1] - d[y][x]$ ; il s'agit donc d'un schéma de gradient en  $x$ ;

<pre> ELISE_COPY (   W.all_pts(),   Min   (     255,     Abs(I.in(0)[(FX+1,FY)]-I.in(0))*3   ),   W.out(Pgr) ); </pre>	
--	--

FIG. 3.2 – Un exemple de filtre élémentaires défini à partir des opérateur de base.

- on affiche  $\min(|G_x| * 3, 255)$  : on prend la valeur absolue c'est plus simple à visualiser, on multiplie par 3 pour rehausser la dynamique, on tronque à 255 pour éviter des problèmes de débordement.

<pre> ELISE_COPY (   W.all_pts(),   Min   (     255,     Abs(I.in(0)-trans(I.in(0),Pt2di(0,1)))*3   ),   W.out(Pgr) ); </pre>	
---	---

FIG. 3.3 – Un exemple de gradient en  $y$  utilisant l'opérateur **trans**.

Lorsque l'on définit des opérateurs on effectue couramment, comme dans l'expression `I.in(0)[(FX+1,FY)]`, des changement de coordonnées qui sont en fait de simple translation. Pour ceci, *Elise* offre l'opérateur **trans**(Fonc\_Num f,Pt2di p0) qui définit la fonction  $x, y \rightarrow f(x + p_0.x, y + p_0.y)$ . La figure 3.3 donne un exemple d'utilisation de **trans** pour définir un filtre de gradient vertical. Lorsque le changement de coordonnées est une translation, on toujours intérêt à utiliser **trans** plutôt que `[]` pour les raisons suivantes ;

- l'exécution du code par *Elise* sera plus rapide ;
- il existe des propriétés des objets (voir chapitre 7) que l'opérateur **trans** préserve et pas l'opérateur `[]` ; il existe donc des opération qui sont licite avec **trans** et pas avec `[]` ;
- enfin, c'est plus clair quand on relit le code de pouvoir faire immédiatement la différence entre une translation et un changement de coordonnées générique.

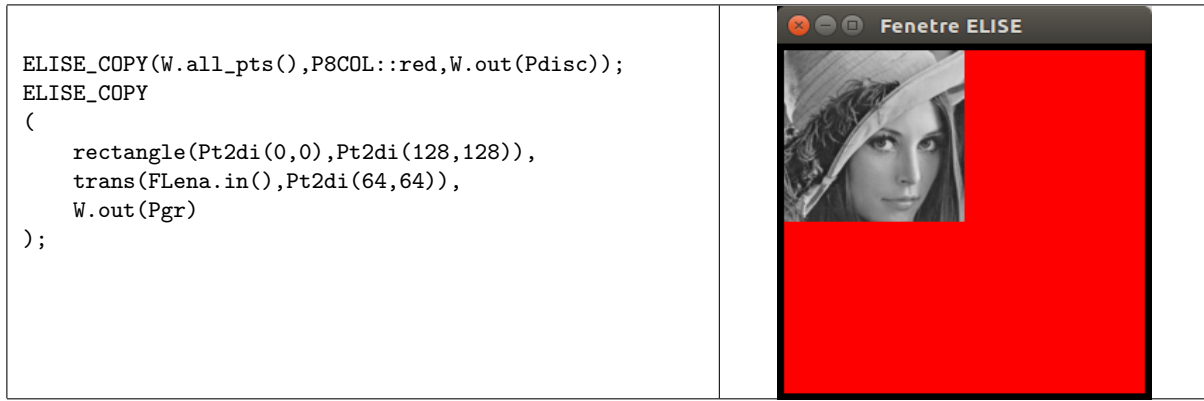
En pratique, une utilisation très courante de **trans** est le cas où l'on dispose d'un “grand” fichier image et où l'on désire en charger une “petite” partie en mémoire. La figure 3.4 donne un exemple d'une telle utilisation de **trans** :

- ici le grand fichier est de taille  $256 \times 256$  ;
- on désire charger une zone de taille  $128 \times 128$  commençant au point (64,64) ;

Les objets **Fonc\_Num**, **Flux\_Ptset** **Output** sont des objets C++ “comme les autres”, à ce titre il peuvent être manipulés, passés en paramètres, affectés dans des variables, servir de valeur de retour à des fonctions ... Cela permet de définir des petits filtres utilisateur en écrivant un code qui va calculer une **Fonc\_Num** à partir d'une autre.

Un exemple de tel code est donné sur la figure 3.5. On définit une fonction **moy** qui prend en parmètre une **Fonc\_Num**  $f$  et un entier  $n$  et renvoie la **Fonc\_Num** moyenne de  $f$  sur le carré  $[-n, n] \times [-n, n]$ , on



FIG. 3.4 – Utilisation de `trans` pour charger une portion de fichier image.FIG. 3.5 – Exemple de code définissant un filtre utilisateur `moy(f,nb)` : moyenne de  $f$  sur le carré  $[-n\ n] \times [-n\ n]$ .

utilise ensuite cette fonction pour faire afficher la moyenne de *Lena* sur  $[-3\ 3] \times [-3\ 3]$ .

## 3.2 Filtre prédéfini

Élise offre un certain nombre d'opérateur de filtrage prédéfinis. On se reportera au chapitre 13 pour un exposé précis et exhaustif du fonctionnement de ces filtres. Pour l'instant nous allons nous contenter de donner des exemples et, auparavant, de donner leur caractéristiques générales :

- pour des raisons d'optimisation, ÉLISE n'autorise l'usage de ces filtres que si l'on parcourt l'image avec un flux de type rectangle  $2-D$  ; sinon il y a génération d'erreur ; c'est, en toute modestie, le seul inconvénient de ces filtres ;
- leur implémentation est en général optimisée tant du point de vue du temps de calcul que de la mémoire qu'ils allouent provisoirement ;
- il opère sur des `Fonc_Num` et renvoie des `Fonc_Num`, on peut donc sans inconvénient les utiliser en cascade ;

1	2	3
<pre> ELISE_COPY (   W.all_pts(),   rect_max(I.in(0),12),   W.out(Pgr) ); </pre>	<pre> ELISE_COPY (   W.all_pts(),   rect_min   (     rect_max(I.in(0),Pt2di(7,7)),     Box2di(Pt2di(-7,-7),Pt2di(7,7))   ),   W.out(Pgr) ); </pre>	<pre> REAL f= 0.9; ELISE_COPY (   W.all_pts(),     canny_exp_filt(I.in(0),f,f)   / canny_exp_filt(I.inside(),f,f),   W.out(Pgr) ); </pre>
		

FIG. 3.6 – **1**Dilatation en niveaux de gris, **2** Fermeture en niveaux de gris et **3** filtrage exponentiel utilisant quelques uns des opérateur de filtrage prédéfinis.

- pour le temps de calcul, on notera notamment que tous les opérateurs de max,min, somme sur une rectangle ont une complexité indépendante de la taille du rectangle;
- pour la mémoire, si on les utilise avec un rectangle de taille  $T_x \times T_y$ , et que le filtre nécessite un voisinage de taille  $v_x \times v_y$ , la taille mémoire allouée provisoirement est typiquement de la forme  $T_x * v_y$ ; on peut donc, par exemple, les appliquer pour aller écrire directement de fichiers à fichiers sur des fichiers dépassant largement la taille mémoire disponible sur la machine;
- il mettent en place un système de bufferisation qui permet, sans effet de bord, d'utiliser le même objets (fichier, tableaux) comme entrées du filtre et pour stocker son résultat;
- il existe une interface utilisateur assez simple pour rajouter de tels filtres;
- assez de publicité, passons aux actes ...

La figure 3.6 donne les premiers exemples d'utilisation de ces filtres, commentons;

1. on utilise l'opérateur **rect\_max** pour effectuer une dilatation en niveau de gris, le masque de dilatation est le rectangle  $[-12 \ 12] \times [-12 \ 12]$ ; notons qu'il est obligatoire ici de donner une valeur par défaut à l'image, en effet pour calculer les valeur du filtre sur le rectangle  $[-256 \ 256] \times [-256 \ 256]$ , Elise va avoir besoin de connaître les valeur de la fonction qui lui est passée en entrée sur  $[-268 \ 268] \times [-268 \ 268]$  (car  $256 + 12 = 268$ !);
2. on effectue une fermeture en niveau de gris en emboîtant dilatation (**rect\_max**) et érosion (**rect\_min**); pour spécifier le rectangle sur lequel le filtre opère, **rect\_max**, **rect\_min** et **rect\_som** peuvent prendre en paramètre soit une boîte, soit un point  $p$  spécifiant le rectangle  $[-p.x \ p.x] \times [-p.y \ p.y]$  soit un entier  $xy$  spécifiant le rectangle  $[-xy \ xy] \times [-xy \ xy]$
3. on utilise l'opérateur **canny\_exp\_filt** pour effectuer une convolution par un filtre exponentiel (si  $f_x, f_y$  sont les paramètres passés à **canny\_exp\_filt**, sa réponse impulsionnelle est  $f_x^{|x|} * f_y^{|y|}$ ); remarquons l'utilisation d'un "truc" assez courant : comme on a utilisé **I.in(0)** pour donner une valeur à l'image en dehors de sont domaine, pour avoir une moyenne non biaisée, on divise le résultat par l'application du filtre à la fonction qui vaut 1 dans l'image et 0 en dehors (cette **Fonc\_Num** est renvoyée par **I.inside()**);

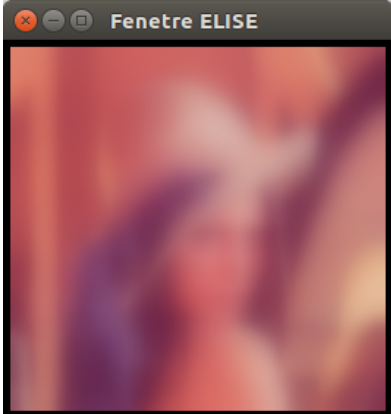
1	2	3
<pre> REAL f = 0.9; ELISE_COPY (   W.all_pts(),   canny_exp_filt(FLenaCol.in(0),f,f) / canny_exp_filt(I.inside(),f,f), W.out(Prgb) ); </pre>	<pre> ELISE_COPY (   W.all_pts(),   Min   (     polar(deriche(I.in(0),1.0),0)     * Fonc_Num(1.0, 256/(2*PI)),     255   ),   (W.out(Pgr), W2.out(Pcirc)) ); </pre>	<pre> // pas de code </pre>
		

FIG. 3.7 – **1** Application d'un filtre à une fonction multi-canal, **2** filtre de deriche donnant un résultat sur 2 canaux (visualisation sur les fenêtre **2** et **3**)

La figure 3.7 donne des de filtres prenant en paramètres ou renvoyant des images multi-canal :

- la première colonne illustre une propriété générale des filtres dont la définition “naturelle” prend en paramètre une fonction mono-canal et renvoie une fonction mono-canal <sup>1</sup> : il peuvent être utilisés sur des images à  $n$  canaux, le résultat sera une fonction à  $n$  canaux correspondant à l'application du filtre dans chaque canal ; ici, on lisse donc à la fois les canaux rouge, vert et bleu ;
- la deuxième colonne (visualisation sur les fenêtre **2** et **3**) apporte plusieurs éléments nouveaux :
  - **deriche** est simplement un opérateur de filtrage qui attend une **Fonc\_Num** mono-canal et renvoie une **Fonc\_Num** bi-canal correspondant au filtre de deriche (les deux canaux correspondent naturellement aux composantes du gradients en  $x$  et  $y$ ) ;
  - **polar** est un opérateur qui prend en paramètre une **Fonc\_Num** sur 2 canaux et renvoie une **Fonc\_Num** sur 2 canaux correspondant à une transformation en coordonnées polaire ; le paramètre 0 indique simplement un angle à renvoyé quand les 2 composantes sont nulles ;
  - **Fonc\_Num(a,b)** (ici  $a = 1.0$ ,  $b = 256/(2*PI)$ ) permet de créer une fonction constante de dimension de sortie 2, qui renvoie toujours le points  $a, b$  ; en effet, selon une règle du C++ l'opérateur “,” est défini sur tous les types natifs (il renvoie son second argument après avoir exécuté son premier) et on ne peut pas changer cette signification ; donc si si on écrivait  $(a,b)$  le compilateur verrait l'application de “,” aux entier et créerait donc une fonction constante à une seule coordonnées ;
  - ici on multiplie par **Fonc\_Num(1.0, 256/(2\*PI))** pour faire rentrer l'angle dans une dynamique  $[-128\ 128]$  sans modifier le module ;
  - le **Min** avec 255 est prudent ;
  - enfin on utilise “,” sur les **Output** pour simultanément afficher le module en niveaux de gris et l'angle en palette circulaire ;

La figure 3.8 ne devrait pas poser de problème particulier :

- on crée une image binaire que l'on stocke dans **Ibin** ; on réutilisera plusieurs fois **Ibin** dans cette section ;

<sup>1</sup>comme c'est, par exemple, le cas pour **canny\_exp\_filt rect\_max, rect\_min, rect\_som** et tous les opérateur morpho (dilatation, extinction ...)



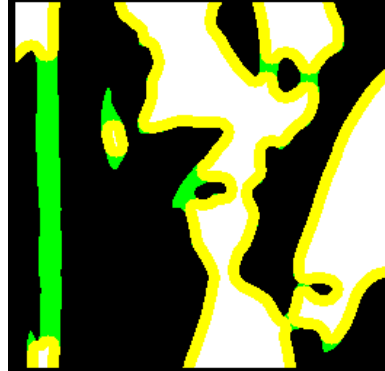
1	2	3
<pre> Im2D_U_INT1 Ibin(256,256); REAL f = 0.8; ELISE_COPY (   W.all_pts(),   canny_exp_filt(I.in(0),f,f) / canny_exp_filt(I.inside(),f,f) &lt; 128, W.out(Pdisc)   Ibin.out() ); </pre>	<pre> ELISE_COPY (   W.all_pts(),   Ibin.in(0),   W.out(Pdisc) ); ELISE_COPY (   select   (     W.all_pts(),     open_5711(Ibin.in(0),40)   ),   P8COL::cyan,   W.out(Pdisc) ); ELISE_COPY (   select   (     W.all_pts(),     erod_5711(Ibin.in(0),40)   ),   P8COL::blue,   W.out(Pdisc) ); </pre>	<pre> ELISE_COPY (   W.all_pts(),   dilat_5711(Ibin.in(0),40) * P8COL::yellow, W.out(Pdisc) ); ELISE_COPY (   select   (     W.all_pts(),     close_5711(Ibin.in(0),40)   ),   P8COL::green,   W.out(Pdisc) ); ELISE_COPY (   select(W.all_pts(),Ibin.in(0)),   P8COL::black,   W.out(Pdisc) ); </pre>
		

FIG. 3.8 – Création d'une image binaire, utilisation de cette image pour tester les opérateur de morpho-math avec le chamfrein 5 – 7 – 11.

- utilisation de `Ibin` pour illustrer l'appel aux filtre d'ouverture et d'érosion (ici en chamfrein 5 – 7 – 11); comme les autres opérateurs de morpho-math opérant sur des ensembles, il prennent en paramètre une fonction entière et considère que les points sont dans l'ensemble ssi la la valeur est  $\neq 0$ ;
- utilisation de `Ibin` pour illustrer l'appel aux filtre de fermeture et de dilatation (ici en chamfrein 5 – 7 – 11);

En attendant l'écriture du chapitre 13, donnons la liste, classées par grande famille, des filtres existant aujourd'hui :

- `rect_som`, `rect_max`, `rect_min` : somme, minimum et maximum sur un voisinage rectangulaire;
- `rect_var_som` su un voisinage variable pour chaque point;
- `som_masq` convolution par un filtre utilisateur;
- `rle_som_masq_binaire` convolution par un filtre utilisateur binaire (rapide si filtre fortement auto-corrélé);
- `canny_exp_filt` `semi_cef` filtre exponentiel, et filtre exponentiel tronqué aux  $x \geq 0, y \geq 0$ ;
- `deriche`, `bobs_grad`, `grad_crois`, `sec_deriv` : gradient de deriche, gradient de robert, gradient croisé, dérivées seconde;
- `extinc_32` `extinc_d8` `extinc_d4` `extinc_5711` `erod_32` `erod_d8` `erod_d4` `erod_5711` `dilat_32` `dilat_d8` `dilat_d4` `dilat_5711` `open_32` `open_d8` `open_d4` `open_5711` `close` `close_32` `close_d8`

- `close_d4 close_5711` : fonction d'extinction, érosion, dilatation, ouverture et fermeture pour les disants des 4 et 8 voisins et des chamfreins 2 – 3 et 5 – 7 – 11 ;
- `EnvKlipshcitz EnvKlipshcitz_5711 EnvKlipshcitz_32 EnvKlipshcitz_d4 EnvKlipshcitz_d8`
- `rect_kth rect_median`  $k^{ème}$  valeur et médian sur un voisinage rectangulaire ;
- `rect_egal_histo` et `rect_rank` égalisation d'histogramme local et fonction de rang sur un voisinage rectangulaire ;
- `label_maj dilate_label` filtre d'étiquette majoritaire et dilatation d'étiquettes ;
- `erod_8_hom, skeleton` érosion homotopique et squelette ; `skeleton_and_dist`
- `flag_vois` flagage d'une relation de voisinage ;
- `nflag_sym nflag_close_sym nflag_open_sym` opération sur des graphes de pixels (symétrie, fermeture symétrique et ouverture symétrique) ;
- `fonc_a_trou`

D'ici la rédaction du chapitre 13, le plus simple est de consulter MPD si vous pensez avoir besoin d'un de ceux non traité en exemple.

### 3.3 Interface C++/ELISE, ajout de filtre prédéfini

Examinons le cas où, dans une application, nous avons besoin d'un filtre qui ne soit pas offert par *Elise* sous forme de filtre prédéfini. Pour la simplicité de l'exposé dans ce chapitre d'introduction nous supposons qu'il s'agit du gradient de *sobel*. Dans une phase de prototypage, on est en général peu sûr du fait que le filtre que l'on a imaginé satisfasse réellement nos besoins et on ne sait pas non plus très bien comment l'application souhaitera s'interfacer avec l'implémentation ; on souhaitera alors d'abord disposer, au moindre coût de programmation, d'une implémentation du filtre fonctionnellement correcte et souple d'emploi, même si elle n'est pas optimale du point de vue de la vitesse d'exécution.

<pre> Fonc_Num sobel_0(Fonc_Num f) {  // creation+initialisation de tableaux      Im2D_REAL8 Fx         ( 3,3,           " -1 0 1 "           " -2 0 2 "           " -1 0 1 "         ); </pre>	<pre>     Im2D_REAL8 Fy         ( 3,3,           " -1 -2 -1 "           "  0  0  0 "           "  1  2  1 "         );      return         Abs(som_masq(f,Fx,Pt2di(-1,-1)))         + Abs(som_masq(f,Fy)); } </pre>
---	---

FIG. 3.9 – Définition du filtre de sobel en utilisant uniquement des opérateurs prédéfinis d'Elise.

Dans ce cas, on aura intérêt à regarder si le filtre peut s'implémenter en assemblant les briques élémentaires (filtres+ opérateurs divers) offertes par *Elise*. S'il s'agit d'une opération aussi simple que le gradient de sobel, cela ne pose évidemment aucun problème et une solution est proposée en figure 3.9. Commentons rapidement les quelques éléments nouveaux qui sont illustrés dans cet exemple :

- On voit qu'il est possible d'initialiser de "petits" tableaux en donnant comme dernier argument au constructeur une chaîne de caractères<sup>2</sup> contenant (sous forme "ASCII") les valeurs initiales ;
- `som_masq(Fonc_Num fonc, Im2D_REAL8 Im, Pt2di pt)` renvoie une `Fonc_Num` qui est l'application du masque de convolution `Im` à `fonc` ; le point `pt` spécifie l'origine du masque ;
- lorsque `pt` est omis, *Elise* considère que le masque est centré ;

Dans un deuxième temps, s'il s'avère que le gradient de sobel correspond effectivement aux besoins de notre application et que l'implémentation sous forme `sobel_0` de la figure 3.9 induit un ralentissement non négligeable, on souhaitera l'implémenter de manière "classique" sous forme d'une fonction opérant directement sur des tableaux. Cela ne pose pas de problème particulier et sous *Elise*, comme

<sup>2</sup>On rappelle, une règle du C++ : la suite de chaînes littérales "A" "B" "C" "D" ... est interprétée par le compilateur comme "ABCD..."

<pre> template &lt;class Type,class TyBase&gt; class Filters { public :  static inline TyBase sobel (Type ** im,INT x,INT y) {     return     Abs     (         im[y-1][x-1]+2*im[y][x-1]+im[y+1][x-1]         - im[y-1][x+1]-2*im[y][x+1]-im[y+1][x+1]     )     + Abs     (         im[y-1][x-1]+2*im[y-1][x]+im[y-1][x+1]         - im[y+1][x-1]-2*im[y+1][x]-im[y+1][x+1]     ); } }; </pre>	<pre> template &lt;class Type,class TyBase&gt; void std_sobel (     Im2D&lt;Type,TyBase&gt; Iout,     Im2D&lt;Type,TyBase&gt; Iin,     Pt2di p0,     Pt2di p1 ) {     Type ** out = Iout.data();     Type ** in = Iin.data();     INT x1 = ElMin(Iout.tx()-1,Iin.tx()-1,p1.x);     INT y1 = ElMin(Iout.ty()-1,Iin.ty()-1,p1.y);     INT x0 = ElMax(1,p0.x);     INT y0 = ElMax(1,p0.y);      // pour eviter les overflow      TyBase vmax = Iout.vmax()-1;     for (INT x=x0; x&lt;x1 ; x++)         for (INT y=y0; y&lt;y1 ; y++)             out[y][x] = ElMin             (                 vmax,                 Filters&lt;Type,TyBase&gt;::sobel(in,x,y)             ); } </pre>
--	--

FIG. 3.10 – Définition “classique” d’une fonction `std_sobel` opérant directement sur des tableaux.

on l’a déjà vu, l’utilisateur a toujours la liberté d’accéder aux zone de données des tableaux pour les manipuler de manière classique. La figure 3.10 donne un exemple possible d’une telle fonction `std_sobel`, commentons les éléments nouveaux de cette implantation :

- on voit que les classes de tableau `Elise` sont des classe `template`, on implémente donc la fonction `std_sobel` sous forme d’un template de fonction afin de pouvoir éventuellement l’utiliser avec différent type de tableau ;
- tous les types d’image que nous avons utilisé jusqu’à présent ne sont que des instanciations particulières de ces classes `template` ; par exemple, `Im2D_U_INT1` est strictement équivalent à `Im2D<U_INT1,INT>` ou encore à `Im2D<unsigned char,int>` ;
- `Elise` fournit des définition `U_INT1`, `U_INT2`, `REAL4` ... pour tout les types numérique utilisés, il vaut mieux utiliser ces définition que les définition habituelles du `C++` pour des raisons de portabilité bien connues (est-ce qu’un `char` est signé, est-ce qu’un `int` est sur 2 ou 4 bytes ...) ;
- les template de classe tableaux sous `Elise` possèdent deux arguments (appelé ici `Type` et `TyBase`) ; le premier argument `Type` est l’argument “important”, il correspond au type sur lequel sont effectivement stocké les éléments du tableaux ;
- le deuxième argument, `TyBase`, est en fait totalement conditionné par le premier, il doit valoir `INT` pour tout les type entiers (`U_INT1`, `INT1` ...) et `REAL` pour les types en virgules flottantes ; cet argument de template est utile pour l’écriture de certaine fonction template, par exemple pour spécifier le type d’un variable intermédiaire et le type de retour d’une fonction ;  
ici par exemple, dans la classe template `Filters`, c’est grâce à `TyBase` que l’on est capable de spécifier que la fonction élémentaire `sobel` doit renvoyer des des `int` si elle opère sur un type intégral et des `double` si elle opère sur un type en virgule flottante ;

Supposons maintenant qu’après validation, le filtre de sobel soit considéré comme un opérateur essentiel dans notre environnement, on souhaitera alors disposer d’une implémentation qui offre à la fois la souplesse de la solution de la figure 3.9 et une efficacité proche de la solution de la figure 3.10. Pour ceci, `Elise` offre une interface permettant de créer des filtres prédéfini à partir d’un minimum de code utilisateur. La code de la figure 3.11 correspond à l’implémentation du filtre de sobel comme opérateur prédéfini ;

- `create_op_buf_simple_tpl(cbI,cbR,fonc,dimout,box)` est une des interface d’`Elise` permettant de rajouter des filtres ;

<pre> template &lt;class Type&gt; void sobel_buf (     Type **          out,     Type ***         in,     const Simple_OPBuf_Gen &amp; arg ) {     for (int d = 0; d &lt; arg.dim_out(); d++)         for (int x = arg.x0(); x &lt; arg.x1(); x++)             out[d][x] =                 Filters&lt;Type, Type&gt;::sobel(in[d], x, 0); } </pre>	<pre> Fonc_Num sobel(Fonc_Num f) {     return create_op_buf_simple_tpl         (             sobel_buf,             sobel_buf,             f,             f.dimf_out(),             Box2di(Pt2di(-1,-1), Pt2di(1,1))         ); } </pre>
--	--

FIG. 3.11 – Code utilisateur pour créer un nouveau filtre “prédéfini” correspondant au gradient de sobel.

- ses deux premiers argument **cbI** et **cbR** sont des call-back que nous allons détailler plus bas ;
- l’argument **fonc** est la **Fonc\_Num** sur laquelle doit s’appliquer le filtre ;
- **dimout** est un entier spécifiant quelle est la dimension de sortie de la **Fonc\_Num** renvoyée par l’opérateur ; on indique que cette dimension sera la même que celle de **fonc** (ici, en accord avec les autres filtres  $\mathcal{E}\mathcal{L}\mathcal{S}\mathcal{E}$  prédéfinis, on implante une version de sobel où, si la fonction en entrée est à  $n$  canaux, la sortie est aussi une fonction à  $n$  canaux correspondant à l’application du filtre dans chaque canal) ;
- **box** est une boîte indiquant quel est le voisinage nécessaire pour calculer le résultat du filtre ; ici on indique que pour connaître la valeur de sobel de  $I$  en un point  $(x_0, y_0)$ , il suffit de connaître  $I(x, y)$  pour  $\{(x, y) / x_0 - 1 \leq x \leq x_0 + 1 \text{ et } y_0 - 1 \leq y \leq y_0 + 1\}$

Détaillons maintenant la signification de ces deux call-back, on notera  $[\delta_{x,0} \ \delta_{x,1}] \times [\delta_{y,0} \ \delta_{y,1}]$  la boîte **box** :

- le premier call-back sera appelé dans le cas où la fonction est entière et le second dans celui où la fonction est réelle ; en pratique, quand l’on voudra comme ici gérer ces deux cas, on passera toujours un template de fonction ;
- le call-back sera appelé avec les paramètres (**out**, **in**, **arg**) pour chaque ligne d’image ; il devra remplir la sortie **out** en fonction de l’entrée **in** ;
- la sortie **out** est à remplir sous la forme **out[d][x]** où  $0 \leq d < d_{out}$  et  $arg.x_0() \leq x < arg.x_1()$
- pour calculer **out**, l’entrée **in** est adressable sous la forme **in[d][y][x]** avec  $0 \leq d < d_{in}$ ,  $\delta_{y,0} \leq y \leq \delta_{y,1}$  et  $\delta_{x,0} + x_0 \leq x < \delta_{x,1} + x_1$  (où  $d_{in}$  = dimension de sortie de la fonction passée en entrée au filtre) ;
- $arg.x_0()$  et  $arg.x_1()$  désignent l’intervalle d’abscisse sur lequel il faut remplir **out** ;
- **arg** contient tout un tas d’information sur le contexte dans lequel opère le filtre ; par exemple ici **arg.dim\_out()** permet de connaître la dimension de sortie ;

La figure 3.12 montre que, une fois écrit par l’utilisateur, les filtres tels que **sobel** ; s’utilisent exactement comme les filtre prédéfinis :

1. utilisation simple de **sobel** ;
2. utilisation de **sobel** sur une fonction de dimension de sortie 3 (ok, le sobel multi-dimensionnel ça n’a pas grande signification, mais c’est un autre problème) ;
3. utilisation emboîtée avec d’autres opérateurs ;

Dans ce chapitre d’introduction, nous n’avons exposé que les caractéristiques les plus simples de l’interface  $\mathcal{E}\mathcal{L}\mathcal{S}\mathcal{E}$  - C++ . Décrivons rapidement quelques unes des caractéristiques que nous verrons en partie II ;

- souvent un filtre doit être paramétré, par exemple les filtres exponentiels doivent avoir accès au facteur “d’amortissement” ; pour ces filtres paramétrés, au lieu de fournir à  $\mathcal{E}\mathcal{L}\mathcal{S}\mathcal{E}$  le call-back sous forme de fonction, on lui passera des objets (de manière plus précise, l’adresse d’objets dérivant de la bonne classe et redéfinissant les méthode virtuelle de calcul) ;
- si le filtre n’a d’intérêt que pour des fonctions entières (ou réelles), on peut toujours passer à  $\mathcal{E}\mathcal{L}\mathcal{S}\mathcal{E}$  la valeur 0 pour le call-back réel ;
- $\mathcal{E}\mathcal{L}\mathcal{S}\mathcal{E}$  fournit aussi une interface pour définir de nouveaux opérateurs arithmétiques ;
- on verra au chapitre 13 comment, tout en respectant cette interface, implémenter des filtre “rapide” (par exemple moyenne sur un voisinage rectangulaire en temps de calcul indépendant de la taille du voisinage) ;



1	2	3
<pre> ELISE_COPY (   W.all_pts(),   Min(255,sobel(I.in(0))),   W.out(Pgr) ); </pre>	<pre> ELISE_COPY (   W.all_pts(),   Min(255,sobel(FLenaCol.in(0))),   W.out(Prgb) ); </pre>	<pre> REAL f = 0.9; Fonc_Num Fonc =   canny_exp_filt(I.in(0),f,f)   / canny_exp_filt(I.inside(),f,f); ELISE_COPY (   W.all_pts(),   Min(255,sobel(Fonc)*8),   W.out(Pgr) ); </pre>
		

FIG. 3.12 – Exemples d'utilisation de l'opérateur `sobel` défini par l'utilisateur.

### 3.4 Output de “réduction associative”

La figure 3.13 utilise deux nouvelles fonctions `extinc_32` et `VMax`; `extinc_32` est simplement un filtre prédéfini calculant la fonction d'extinction en distance 3 – 2; `VMax` appartient à une nouvelle famille de fonction assez importante dont le fonctionnement est le suivant :

- sa signature est `VMax(int & v)` (ou `VMax(double & v)`), il renvoie un `Output` qui a pour effet de mémoriser dans `v` la valeur max atteint par la fonction;
- il existe de même des fonctions `VMin(int &)` et `sigma(int &)` pour mémoriser le min et la somme d'une fonction;
- ces `Output` sont de dimension consommée 1;
- il existe, pour toutes ces fonctions, des surcharges du type `VMax(int * v,int nb)` (ou `VMax(double * v,int nb)`), pour mémoriser le max (min, somme) de `nb` coordonnées; les `Output` sont alors de dimension consommée `nb`;
- il existe des variantes qui sont fonctions membres des classes de points (voir 4.3 page 64);

Explicitons maintenant ce que fait le code la figure 3.13. On veut afficher la fonction d'extinction de l'image `Ibin` avec une dynamique comprise dans [0 255] :

- on calcule cette fonction en mémorisant sa valeur dans `Idist` et le maximum qu'elle atteint dans `vmax`;
- on affiche le résultat dans la fenêtre en niveau de gris en le multipliant par  $\frac{255}{vmax}$ ;
- on remet en blanc les point valant 0 dans `Ibin`;

Le code de la figure 3.14 présente un calcul d'histogramme. Avant de détailler ce code, nous devons introduire la fonction membre `histo` :

- si `I` est un tableau et `d` sa zone de données;
- alors `I.histo()` est un `Output` de dimension consommée 1;
- si `p` est un point et `v` une valeur, alors la méthode `update` de `I.histo()` effectuer simplement `d[p] = d[p]+v`;
- cela fonctionne pareil avec des tableau `1 - D`, `2 - D` ...;
- il existe des fonction membre équivalente `max_eg`, `min_eg` et `mul_eg` pour effectuer `d[p] = max(d[p],v)`, `d[p] = min(d[p],v)` et `d[p] = d[p]*v` (ces fonction sont d'usage moins courant que



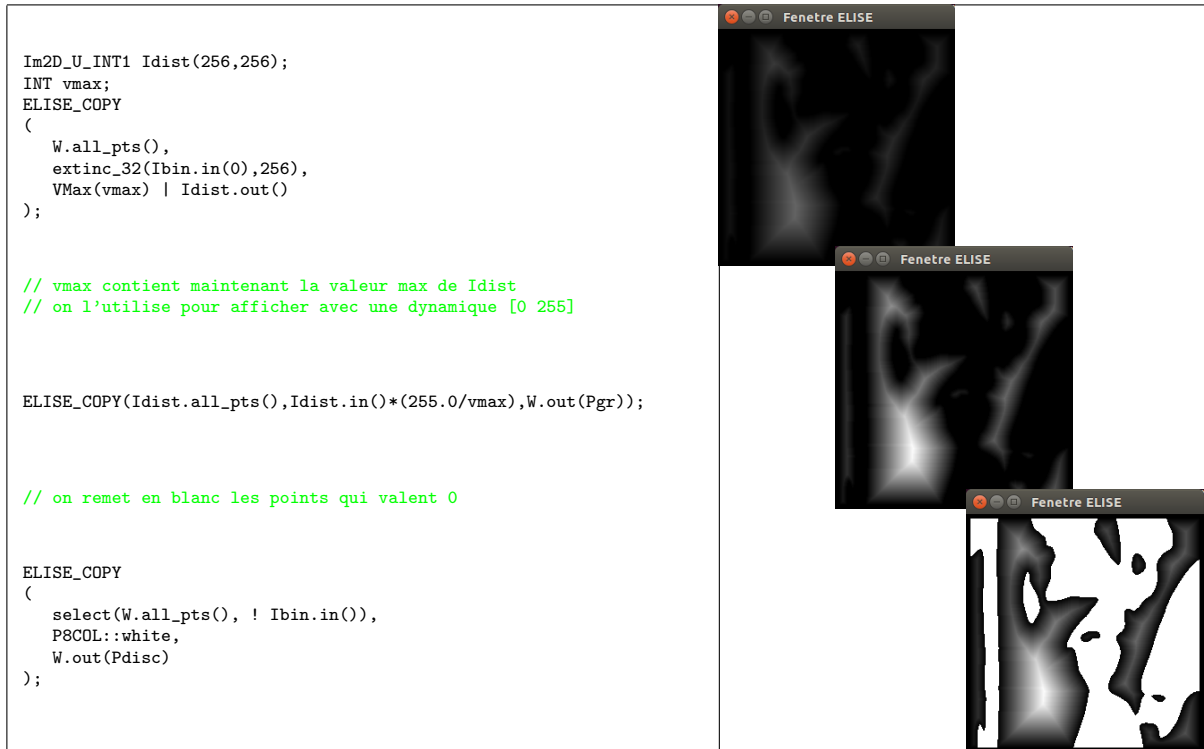


FIG. 3.13 – Utilisation d'un fonction d'extinction. Utilisation de l'opérateur de réduction associative **VMax** : réduction

`histo);`

On peut maintenant voir pourquoi ce code effectue le calcul de l'histogramme :

- on déclare un tableau **H** de dimension 1, de taille 256, codant des entiers sur 4 octets et dont tous les éléments valent initialement 0; doit  $d_H$  sa zone de données de  $H$  ;
- **W.all\_pts().chc(I.in())** utilise l'opérateur de changement de coordonnées sur les flux (vu en 2.9); on parcourt donc l'image en générant le flux  $1 - D$  des niveaux de gris de *Lena*; soit  $d_L$  la zone de donnée de l'image;
- comme c'est la fonction 1 qui est copiée dans **H.histo()**, on effectue finalement pour chaque pixel de l'image  $d_H[d_L[y][x]] += 1$ , ce qui calcule bien l'histogramme;
- finalement on utilise le `plotter` pour visualiser l'histogramme;

Voyons maintenant un point un peu délicat. La fonction `histo` peut sembler non indispensable et on pourrait tenter d'écrire simplement **H+1** dans **H** comme cela le code de la figure 3.15. En fait, cela ne fonctionne pas et la figure 3.15 montre le résultat obtenu; de manière générale, il faut (sous *ELISE* comme ailleurs) faire attention aux opérations qui utilisent le même objet en lecture et en écriture dans la même opération. Le chapitre 7 exposera de manière précise les règles permettant quand ces opérations sont licites. D'ici là, il faut par défaut considérer que ces opérations sont dangereuses et se limiter aux types d'opérations montrées en exemple.

Les fonctions de la famille `histo` ne sont évidemment pas limitées aux tableaux de dimension 1. La figure 3.16 présente un exemple correspondant à un calcul de matrice de cooccurrence. Commentons rapidement :

- **interior(INT ep)** est une fonction membre qui renvoie un flux de point correspondant à “tout les points de l'objet moins un bord d'épaisseur **ep**” ;
- **W.interior(1).chc((I.in(),trans(I.in(),Pt2di(1,0))))** génère le flux des point de la forme  $(d_L[y][x], d_L[y][x+1])$ ; on va donc pouvoir compter les “cooccurrences horizontale” ;
- on visualise le résultat avec un dynamique logarithmique puisque, classiquement, ces matrices sont très concentrées sur la diagonale ;

```

// remet Lena en fonds

ELISE_COPY(W.all_pts(),I.in(),W.out(Pgr));

// cree un tableau pour y stocker l'histogramme

Im1D_INT4 H(256,0);

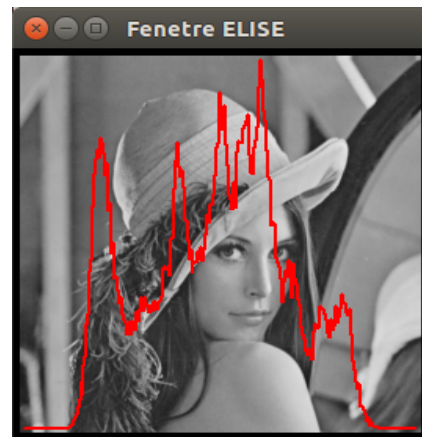
// calcul de l'histogramme

ELISE_COPY
(
  W.all_pts().chc(I.in()),
  1,
  H.histo()
);

// visualisation

ELISE_COPY(Plot1.all_pts(),H.in(),Plot1.out());

```

FIG. 3.14 – Calcul de l'histogramme de *Lena*.

```

// .... tout comme avant

ELISE_COPY
(
  W.all_pts().chc(I.in()),
  1+H.in(),
  H.out()
);

```

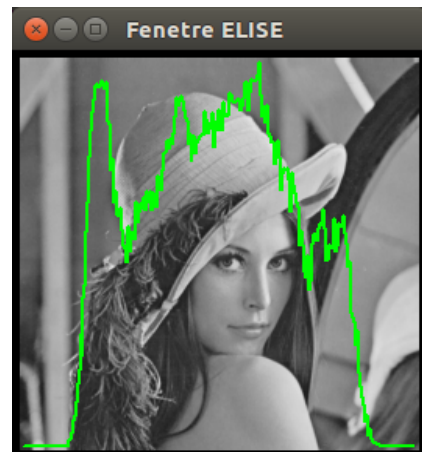


FIG. 3.15 – Calcul "bugué" d'histogramme, en vert résultat erroné obtenu, en rouge histogramme réel.

### 3.5 Filtres "linéaires"

```
Im2D_INT4 Cooc(256,256,0);
INT cmax;
ELISE_COPY
(
  W.interior(1).chc
  ((
    I.in(),
    trans(I.in(),Pt2di(1,0))
  )),
  1,
  Cooc.histo()
  | (VMax(cmax) << Cooc.in())
);
ELISE_COPY
(
  Cooc.all_pts(),
  255 -log(Cooc.in()+1)
  * (255.0/log(cmax+1)),
  W.out(Pgr)
);
```

FIG. 3.16 – Calcul de matrice de coocurrence.



## Chapitre 4

# Exemples, analyse d'images

*En fait je ne sais pas très bien s'il y a une différence réelle entre le traitement et l'analyse d'image, ni où devrait se situer l'éventuelle frontière. Cela me fournit surtout un prétexte pour couper en 2 un gros chapitre.*

Le code des exemples de ce chapitre se trouve dans le fichier "applis/doc\_ex/introanalyse.cpp".

### 4.1 Liste de points

Les outils vus au chapitre précédent étaient essentiellement destinés à effectuer des opérations globales sur les images. En analyse d'image, il apparaît souvent des situations où, après un traitement préalable, on souhaite focaliser son attention sur tel ou tel ensemble de points "remarquables". Pour ceci, on devra pouvoir mémoriser quelque part ces ensembles de points puis les reparcourir rapidement (ie. sans avoir à reparcourir toute l'image). Les listes de points offertes par ELISE ont pour objectif de répondre à ces situations.

1	2	3
<pre>Im2D_U_INT1 Im(256,256); ELISE_COPY (     W.all_pts(),     rect_median(I.in_proj(),5,256),     Im.out()   W.out(Pgr) );</pre>	<pre>Liste_Pts_INT2 l2(2); ELISE_COPY (     select(W.all_pts(),Im.in() &lt; 80),     P8COL::red,     W.out(Pdisc)   l2 );</pre>	<pre>ELISE_COPY (     l2.all_pts(),     P8COL::blue,     W.out(Pdisc) );</pre>

FIG. 4.1 – Premier exemple d'utilisation de liste de points.

La figure 4.1 donne un premier exemple d'utilisation de liste de points, commentons les différentes colonnes :

1. la première colonne a seulement pour objectif de créer une version régularisée de *lena* que l'on mémorise dans `Im`; notons quand même :
  - l'utilisation de la fonction `rect_median` qui est un filtre prédéfini correspondant au médian sur un voisinage rectangulaire;
  - les deux premiers arguments ont la même sémantique que `rect_max` et apparentés (= fonction sur laquelle appliquer le filtre et taille du voisinage); le troisième argument spécifie un majorant des valeurs qu'atteindra l'image, il permet à `ELISE` de dimensionner des tables lui permettant d'optimiser le calcul;
  - notons que l'on prolonge l'image par `in_proj` (prolongement par projection sur la valeur aux bords) déjà vu en 1.6.2; en effet, pour un filtre médian, il n'existe pas de valeur constante correspondant à un prolongement "naturel";
2. dans la deuxième colonne, `Liste.Pts_INT2 12(2)` crée une liste de points `12`; cette liste permettra de stocker des points de dimension 2 et chaque coordonnée sera stockée sur deux octets non signés; cette liste est initialement vide; elle est ensuite utilisée comme `Output` dans l'expression `W.out(Pdisc) | 12`, détaillons comment se comporte une liste de points quand elle est utilisée comme `Output` :
  - sa méthode `update(pt, val)` a pour effet de rajouter `pt` à la fin de la liste;
  - la dimension de `pt` doit être la même que celle de la liste (sinon une erreur est générée);
  - l'`Output` a une dimension consommée de 0 (ce qui est logique compte-tenu du fait que `val` est ignorée);
3. dans la troisième colonne, on utilise la fonction membre `all_pts()`; si `Lpt` est une liste de dimension  $d$ , alors `Lpt.all_pts()` renvoie un `Flux_Pts` de dimension  $d$  qui va parcourir tous les points contenus dans la liste; on regénère donc tous les points mémorisés par le code de la colonne précédente;

1	2	3
<pre>ELISE_COPY (   select(W.all_pts(), Im.in() &gt; 160),   P8COL::green,   W.out(Pdisc)   12 );</pre>	<pre>ELISE_COPY (   12.all_pts(),   P8COL::yellow,   W.out(Pdisc) );</pre>	<pre>ELISE_COPY (   12.all_pts().chc((FY, FX)),   P8COL::cyan,   W.out(Pdisc) );</pre>

FIG. 4.2 – Utilisation de liste de points.

Les deux premières colonnes de la figure 4.2 illustrent une caractéristique des listes de points sous `ELISE` : quand on réutilise plusieurs fois une liste comme `Output`, celle-ci n'est pas réinitialisée à chaque fois mais, au contraire, les nouveaux points sont concaténés à ceux que contient déjà la liste. Ici à l'issue du code de la figure 4.1, la liste `12` contient les points tels que `Im.in() < 80`; ensuite, dans la première colonne du code de la figure 4.2, on rajoute les points tels que `Im.in() > 160`, et on vérifie dans la deuxième colonne de la figure 4.2 que finalement `12` contient bien les points tels que `Im.in() < 80` ou `Im.in() > 160`.

La dernière colone de la figure 4.1 illustre simplement le fait que, quand elles sont utilisées en lecture, les listes de points sont des `Flux_Pts` comme les autres auxquels on peut appliquer les opérateurs sur les `Flux_Pts`.

<pre> ELISE_COPY(W.all_pts(),Im.in(),W.out(Pgr)); Liste_Pts_INT2 l3(3); ELISE_COPY (     select     (         W.all_pts(),         Im.in()&gt;128     ).chc((FX,FY,I.in())),     1,     l3 ); ELISE_COPY(l3.all_pts(),FZ,W.out(Pgr).chc((FX,FY))); </pre>	
---	--

FIG. 4.3 – Liste de points pour stocker des associations “point/valeur”.

Le code de la figure 4.3 illustre (dans un contexte un peu artificiel) une utilisation un peu plus compliquée des listes de points :

- parfois ce ne sont pas uniquement des points que l’on souhaite mémoriser mais plutôt des points plus la valeur qu’ils prennent pour certaines fonctions ;
- soit  $k$  la dimension des points et  $n$  la dimension de sortie de la fonction ;
- pour ceci, il suffit de créer une liste de points de dimension  $n + k$  et d’aller copier dedans des points de  $\mathcal{Z}^{n+k}$  obtenus par concaténation des coordonnées des points et des valeurs de la fonctions (ici par exemple : `Flux.chc((FX,FY,I.in()))`) ;
- ensuite, en lecture, on utilise les fonctions coordonnées `FX,FY,FZ` pour séparer points et valeurs ;
- on a noté ici l’utilisation de `FZ` dont la signification devrait être évidente après `FX` et `FY` ;
- pour les dimensions supérieures, on utilisera `Fonc.Num kth_coord(int k)` pour avoir accès à la  $k^{\text{ème}}$  coordonnée (en fait `FX,FY` et `FZ` sont strictement équivalents à `kth_coord(0)`, `kth_coord(1)` et `kth_coord(2)`) ;
- on notera que pour écrire dans la fenêtre on utilise `W.out(Pgr).chc((FX,FY))`, en effet une fenêtre refuse que l’on écrive dedans avec un `Flux_Pts` de dimension  $\neq 2$  ;

Effectuons quelques remarques complémentaires concernant les listes de points sous `Elise` :

- tout comme les images, les listes de points sont des classes template ; par exemple `Liste_Pts_INT2` est strictement équivalent à `Liste_Pts<INT2,INT>` ;
- ces listes sont implémentées comme des liste chaînées de “petits tableaux” ; cette implémentation permet d’économiser la mémoire (le chaînage ne représente typiquement que 10% de la taille) et d’accélérer certaines opérations ;

<pre> Im2D_INT2 I13 = l3.image(); INT2 ** d = I13.data(); INT nb = I13.tx(); INT2 * tx = d[0]; INT2 * ty = d[1]; INT2 * gray = d[2]; U_INT1 ** im = Im.data(); for (INT k=0 ; k&lt;nb ; k++)     im[ty[k]][tx[k]] = 255-gray[k]; ELISE_COPY(Im.all_pts(),Im.in(),W.out(Pgr)); </pre>	
--	--

FIG. 4.4 – Conversion d’une liste de points en image pour utiliser son contenu dans du code utilisateur.

Parfois on souhaitera récupérer le contenu d'une liste de points pour l'utiliser dans du code utilisateur. Comme la structure de liste de points retenue est un peu compliquée, `Elise` n'offre pas d'accès direct aux données (ça viendra peut-être sous forme d'itérateurs à la STL) mais on permet de convertir ces listes en tableaux qui eux sont faciles à manipuler. Ceci se fait de la façon suivante :

- soit  $L$  une liste de type `Liste_Pts<T1,T2>` de dimension  $d$  contenant  $n$  points ;
- alors `L.image()` renvoie une image  $I$  de type `Im2D<T1,T2>`, dont la taille en  $x$  vaut  $n$  et dont la taille en  $y$  vaut  $d$  ;
- $\forall k, 0 \leq k < d$ , la  $k^{\text{ème}}$  ligne  $I$  contient les coordonnées de  $L$  dans la  $k^{\text{ème}}$  dimension (ou, dit de manière équivalente :  $\forall k, 0 \leq k < n$  la  $k^{\text{ème}}$  colonne de  $I$  contient le  $k^{\text{ème}}$  point de  $L$ ) ;
- ceci est illustré par le code de la figure 4.4 où l'on convertit d'abord la liste `l3` en une image `Il3`, puis on utilise cette image pour aller modifier directement l'image `Im` ;

## 4.2 Relation de voisinage, dilate

En 3.2, nous avons introduit une famille de fonctions (`open_5711 ...`) correspondant aux opérations de morphologie mathématique les plus courantes. Ces fonctionnalités sont plutôt adaptées à des manipulations globales de l'image : elles imposent qu'on les utilise avec un `Flux_Pts` de type rectangle  $2-D$  et, en contrepartie, elles sont optimisées spécialement pour ce type de manipulations. Ces fonctions prennent en paramètre une `Fonc_Num` et renvoient une `Fonc_Num` qui sont les fonctions caractéristiques de l'ensemble avant et après l'opération morphologique.

Lorsque l'on veut effectuer des opérations de type morpho en se focalisant sur un nombre limité de points d'intérêt, les fonctions de la famille `open_5711` peuvent être inadaptées, notamment parce qu'elles nécessitent de parcourir à chaque fois un rectangle englobant tout les points (ce qui d'une part peut être lent et d'autre part conduit à une programmation peu naturelle). `Elise` offre un certain nombre de fonctionnalités permettant d'effectuer du filtrage en se focalisant sur quelques points d'intérêt. La plupart des fonctionnalités utilisent un nouveau type abstrait : le type `Neigh_Rel` (relation de voisinage).

Dans cette section nous allons introduire les types `Neighbourhood` (voisinage) et `Neigh_Rel` (toutes les `Neigh_Rel` contiennent un `Neighbourhood`) et la fonction `dilate` qui est une des fonctions les plus courantes utilisant les `Neigh_Rel`.

Le code de la figure 4.5 illustre la création et l'utilisation de `Neighbourhood`, `dilate` et implicitement de `Neigh_Rel`. Les images des colonnes 2 et 3 ne représentent qu'une vue partielle du résultat afin que l'on puisse observer "au niveau du pixel" l'effet des différentes opérations.

La première colonne crée une image binaire que nous réutiliserons plusieurs fois. On remarquera l'utilisation de `W.border(1)`, pour marquer avec une couleur spécifique les points du bord de l'image, c'est une technique que l'on utilisera souvent avec les opérations que nous allons étudier pour éviter les problèmes de débordement.

La deuxième colonne crée deux objets `V4` et `V8` de types `Neighbourhood` correspondant à ce que l'on dénomme souvent par 4-*Voisinage* et 8-*Voisinage*. Décrivons rapidement le type `Neighbourhood` :

- logiquement un voisinage est simplement un ensemble de points de  $\mathbb{Z}^n$  ; en général c'est un ensemble de point relativement petit (rarement plus de quelques dizaines) et dont les coordonnées sont proches du point origine ; cependant `Elise` n'impose aucune limitation à ce sujet ; la classe `Neighbourhood` permet de représenter les voisinages ;
- il existe un constructeur `Neighbourhood (Pt2di * pt, INT nb)` permettant de créer des voisinages de dimension 2 à  $nb$  points spécifiés par le tableau `pt` ; ici, c'est avec ce constructeur que l'on a créé le voisinage `V4` ;
- il existe deux fonctions `static ,v4()` et `v8()`, de la classe `Neighbourhood` renvoyant le 4-*Voisinage* et 8-*Voisinage* ; c'est avec `Neighbourhood : :v8()` que l'on a créé le voisinage `V8` ;
- il existe un constructeur général `Neighbourhood (Im2D<INT4,INT> im)` permettant de créer un voisinage de dimension `im.ty()` contenant `im.tx()` points (`im[0][k], ... im[d-1][k]` correspondant aux coordonnées du  $k^{\text{ème}}$  point) ;

L'objectif de la class `Neigh_Rel` est de décrire de manière générale la notion de "voisinage conditionnel" afin d'effectuer, par exemple, des opérations de dilatation conditionnelle (au sens habituel de la morpho-math<sup>1</sup>). Détaillons le type abstrait correspondant à la classe `Neigh_Rel` :

- chaque `Neigh_Rel` contient un `Neighbourhood` ;

<sup>1</sup>et éventuellement de manière plus générale avec des opérateurs que l'on ne décrira pas dans ce chapitre



1	2	3
<pre> Im2D_U_INT1 Ibin(256,256);  ELISE_COPY (     W.all_pts(),     I.in() &lt; 128,     W.out(Pdisc)   Ibin.out() ); ELISE_COPY (     W.border(1),     P8COL::red,     W.out(Pdisc)   Ibin.out() ); </pre>	<pre> // Creation de 2 Voisinages  Pt2di Tv4[4] = {     Pt2di(1,0),Pt2di(0,1),     Pt2di(-1,0),Pt2di(0,-1)}; Neighbourhood V4 (Tv4,4); Neighbourhood V8 = Neighbourhood::v8();  // dilatation selon V4  ELISE_COPY (     dilate     (         select(W.all_pts(),Ibin.in()==1),         V4     ),     P8COL::cyan,     W.out(Pdisc) ); </pre>	<pre> // On remet l'ensemble initial // pour visualiser l'effet de // la dilatation  ELISE_COPY (     select     (         Ibin.all_pts(),         Ibin.in() == 1     ),     P8COL::black,     W.out(Pdisc) ); </pre>

FIG. 4.5 – Creation de voisinage (**Neighbourhood**) et utilisation pour dilater un flux (colonnes 2 et 3, image agrandies d'un facteur 3).

- soit  $R^{el}$  une **Neigh\_Rel** et  $V^{ois}$  le voisinage qu'elle contient avec  $V^{ois} = \{v_0, v_1, \dots, v_{n-1}\}$ ;
- soit  $p$  un point, on note  $V^{ois}(p)$  le “dilaté” de  $p$  par  $V^{ois}$  :  

$$V^{ois}(p) = \{p + v_0, p + v_1, \dots, p + v_{n-1}\}$$
- chaque **Neigh\_Rel** est alors caractérisée par la façon dont elle redéfinit la méthode *vois* qui, pour chaque point  $p$ , renvoie un *sous-ensemble* de  $V^{ois}(p)$ ;
- $R^{el}.vois(p)$  est l'ensemble des voisins de  $p$  au sens de la relation  $R^{el}$ ;
- le type le plus simple de relation de voisinage offert par **ELISE** est celui de la relation de voisinage triviale où  $R^{el}.vois(p)$  est *inconditionnellement* égal à  $V^{ois}(p)$ ;
- la classe **Neigh\_Rel** admet un constructeur à un seul paramètre  $V^{ois}$  de type **Neighbourhood** et renvoyant la relation triviale sur  $V^{ois}$ ; donc, selon les règles du C++ , à toutes les fonctions qui attendent des paramètres de type **Neigh\_Rel** on peut passer des **Neighbourhood**, la conversion étant implicitement effectuée par le compilateur;

Nous devons expliquer maintenant la fonction **dilate** utilisée dans la colonne 2 de la figure 4.5 :

- **Flux\_Pts dilate(Flux\_Pts Flx, Neigh\_Rel Rel)** renvoie un **Flux\_Pts** qui est la dilatation de **flx** par **Rel**;

- si  $Flx = \{p_1, \dots, p_n\}$ , `dilate` renvoie un flux de points correspondant à la concaténation de  $R^{el}.vois(p_1), R^{el}.vois(p_2) \dots R^{el}.vois(p_n)$ ;
- ici on utilise `dilate` avec un `Neighbourhood`, donc, comme on l'a vu précédemment, le système “comprend” qu'il s'agit d'une dilatation inconditionnelle selon le 4-*Voisinage*;
- `dilate(select(W.all_pts(),Ibin.in()==1),V4)` décrit donc l'ensemble des points de l'image qui sont 4-*voisin* d'un point noir;
- comme en général les points noirs sont eux même voisins d'un point noir, les points qui sont en bleu, dans l'image de la colonne 2, sont à la fois les points noir de l'image initiale (sauf ceux qui sont “4-isolés” et leur voisins non noirs);
- dans l'image de la colonne 3, on a remis en noir les points initiaux, afin de visualiser distinctement les points “rajoutés” par la dilatation;

1	2	3
<pre> INT nb_pts; ELISE_COPY (   W.all_pts(),   Ibin.in(),   W.out(Pdisc) ); ELISE_COPY (   dilate   (     select(W.all_pts(),Ibin.in()==1),     sel_func(V8,Ibin.in()==0)   ),   P8COL::red,   W.out(Pdisc)   (sigma(nb_pts)&lt;&lt; 1) ); cout &lt;&lt; "found " &lt;&lt; nb_pts &lt;&lt; "\n";  // imprime "found 16560" </pre>	<pre> ELISE_COPY (   W.all_pts(),   Ibin.in(),   W.out(Pdisc) ); Liste_Pts_INT2 l2(2); ELISE_COPY (   dilate   (     select(W.all_pts(),Ibin.in()==1),     sel_func(V8,Ibin.in() == 0)   ),   P8COL::yellow,   W.out(Pdisc)   Ibin.out()   l2 ); cout &lt;&lt; "found " &lt;&lt; l2.card() &lt;&lt; "\n";  // imprime "found 5675" </pre>	<pre> for (int k = 0; k &lt; 5 ; k++) {   Liste_Pts_INT2 newl(2);   ELISE_COPY   (     dilate     (       l2.all_pts(),       Ibin.neigh_test_and_set       (         V8,         P8COL::white,         P8COL::yellow,         20       )     ),     10000,     newl   );   l2 = newl ; } ELISE_COPY(Ibin.all_pts(),   Ibin.in(),W.out(Pdisc)); </pre>

FIG. 4.6 – Exemple de relation de voisinage, dilatations conditionnelles.

Les codes de la figure 4.6 introduisent l'utilisation de relations de voisinage conditionnelles. Commençons par commenter le code de la colonne 1 :

- on réaffiche d'abord l'image binaire initiale :
- on introduit la fonction `sel_func` permettant de “filtrer” une relation par une fonction ;
- sa signature est `Neigh_Rel sel_func(Neigh_Rel,Fonc_Num)` ;
- soit  $R^{el}$  une relation de voisinage,  $F^{onc}$  une fonction, et  $R_{F^{onc}}^{el}$  le résultat de `sel_func( $R^{el}$ ,  $F^{onc}$ )`, alors  $R_{F^{onc}}^{el}$  correspond à la définition :

$$R_{F^{onc}}^{el}.vois(p) \rightarrow \{q \in R^{el}.vois(p) / F^{onc}(p) \text{ est vraie}\}$$

- en termes imagés, `sel_func` est aux `Neigh_Rel` ce que `select` est aux `Flux.Pts` ;

- `dilate(select(W.all_pts(),Ibin.in()==1),sel_func(V8,Ibin.in()== 0))` spécifie donc “l’ensemble des 8-voisins des points noirs qui sont blanc” ;
- l’opérateur `sel_func` peut sembler redondant avec `select` au lecteur attentif ; en l’occurrence il l’est pour cette utilisation avec `dilate` et on aurait obtenu un résultat strictement identique en effectuant une sélection de flux sur le résultat d’une dilatation inconditionnelle ; c’est à dire quelques chose comme :  
`select(dilate(select(W.all_pts(),Ibin.in()==1),V8),Ibin.in()== 0)`  
en fait il existe des opérations (voir `conc` en 4.3) où `sel_func` n’est absolument pas redondant ; je l’ai introduit maintenant car il me semblait que l’utilisation avec `dilate` permettait d’illustrer assez intuitivement son comportement ;
- enfin `(sigma(nb_pts)<< 1)` permet de compter, dans `nb_pts` le nombre de points contenus dans le flux (pour une raison que l’on va voir immédiatement).

Supposons maintenant que l’on veuille mémoriser dans une liste de points l’ensemble des points blancs qui sont 8-voisins d’un point noir. Ce ne serait pas une “bonne idée” de rajouter simplement une liste de point en `Output` au code la colonne 1 de la figure 4.6. En effet, `ELISE` n’apporte aucune garantie au fait que le même point n’apparaîtra pas plusieurs fois dans le résultat de `dilate` ; au contraire, on a la “garantie” que chaque point blanc apparaîtra autant de fois qu’il a de 8-voisin noirs. Dans le code la colonne 1, la valeur imprimée de `nb_pts` est 16560 alors que le nombre réel de points de la frontière est 5675 (le bon compte étant donné par le code la colonne 2).

Pour obtenir l’ensemble des points de la frontière, sans duplicata, il faut effectuer une programmation un peu plus “tendue” illustrée par le code la colonne 2 :

- pour ce qui nous préoccupe, la différence essentielle avec le code de la colonne 1, est que l’on a rajouté `Ibin.out()` comme résultat au `ELISE_COPY` utilisant la dilatation ;
- donc la première fois qu’un point est atteint par la dilatation, on le colorie en jaune dans `Ibin` ; si ce point est 8-voisin d’un autre point noir, il ne sera plus blanc <sup>2</sup>, il ne sera donc pas considéré comme un voisin au sens de la relation `sel_func(V8,Ibin.in() == 0)` ;
- ce style de programmation est tendu au sens où l’on utilise sciemment le même objet en lecture et en écriture ; or, comme on l’avait vu en 3.4 ceci peut parfois créer des effets de bord désagréable ; on donnera au chapitre 7 les règles qui permettent de savoir que ici cette opération est parfaitement saine ;
- enfin, on voit que l’on mémorise le résultat de la dilatation dans une liste de points et que l’on imprime le nombre de points de cette liste ; le nombre de points de la 8-frontière blanche des points noir est donc de 5675.

La colonne 3 de la figure 4.6 introduit une nouvelle fonction `neigh_test_and_set` et implémente avec les outils déjà vus une technique classique en analyse d’image pour effectuer des dilatations itérées. Commentons d’abord la fonction `neigh_test_and_set` ;

- les manipulations du type de la colonne 2 de la figure 4.6, où l’on utilise la même image en lecture et écriture pour éviter de parcourir plusieurs fois les mêmes points, sont très courantes quand on utilise les opérateurs `dilate` et `conc` (voir 4.3) ;
- l’objectif de la fonction membre `neigh_test_and_set` est d’offrir un service permettant d’effectuer cette opération en une seule instruction `ELISE` ; ceci offre d’une part l’avantage d’optimiser le calcul et d’autre part de rendre le code plus clair (dès que l’on voit `neigh_test_and_set` on sait que l’image est utilisée en tant que “marqueur”) ;
- `neigh_test_and_set` est une fonction membre définie pour toutes les image des classes template `Im2D<T1,T2>` ; sa signature est :

`Neigh_Rel neigh_test_and_set(Neighbourhood Vois,INT ValSsel,INT ValUdpate,INT v_max) ;`

- Soit  $I$  l’image utilisée et  $R^{nts}$  la relation renvoyée, elle est caractérisée par :

$$R^{nts}.vois(p) \rightarrow \{q \in V^{ois}(p) / I[q.y][q.x] == Val^{Sel}\}$$

De plus, chaque fois qu’un point est sélectionné on effectue l’action :

$$I[q.y][q.x] \leftarrow Val^{update};$$

enfin la valeur  $v^{max}$  indique à `ELISE` quelle sera la valeur maximale atteinte par l’image afin de pouvoir dimensionner un certain nombre de tables ;

- donc `Ibin.neigh_test_and_set(V8,P8COL : :white,P8COL : :yellow,20)` signifie : “une relation où les voisins d’un point  $p$  sont les 8-voisins qui sont blancs dans `Ibin` sachant que dès qu’un point est sélectionné il doit être colorié en jaune dans `Ibin`, par ailleurs j’offre la garantie qu’aucune valeur testée ne dépassera 20” <sup>3</sup> ;

<sup>2</sup>puisqu’il est jaune maintenant, merci La Palice

<sup>3</sup>Je me demande si le fait de le dire par une phrase rend les chose plus claires ?

- `neigh_test_and_set(Neighbourhood Vois,INT,INT,INT)` est une interface simplifiée correspondant à l'utilisation la plus courante; il existe d'autres surcharges de `neigh_test_and_set` permettant de spécifier de manière plus générale la sélection et l'action à effectuer sur les points sélectionnés;

On peut maintenant commenter le code complet de la colonne 3 de la figure 4.6 :

- ce code effectue une dilatation, selon le 8-voisinage, de taille  $5 + 1$ , des points noirs sur les points blanc en les coloriant en jaune;
- on utilise une propriété très classique des dilaté, pour connaître le  $n + 1^{\text{ème}}$  dilaté il suffit de partir du  $n^{\text{ème}}$  dilaté;
- donc ici, la colonne 2 ayant stocké le premier dilaté dans une liste de points, il suffit de mémoriser à chaque étape le nouveau dilaté dans la liste `new1` qui sera réutilisée comme germe à l'étape d'après (par `l2 = new1`);
- on remarque que ici la fonction à copier (i.e. le  $2^{\text{ème}}$  argument de `ELISE_COPY`) n'a aucune importance puisque le marquage est effectué par `neigh_test_and_set` et que, quand la liste `new1` est utilisée en `Output`, elle ne regarde pas les valeurs qui lui sont passées; on a donc mis la valeur arbitraire 10000;

### 4.3 Composantes connexes conc

Dans cette section nous introduisons la fonction `conc` qui permet de calculer des composantes connexes. On commence aussi à introduire quelques uns des ordres de dessins vecteurs. Le code de la figure 4.7 introduit l'utilisation de `conc`.

Le code de la colonne 1 de la figure 4.7 n'utilise pas encore `conc`, il binarise l'image et attend que l'on clique sur un point noir de l'image binaire au centre duquel on affiche un cercle rouge, commentons les fonctionnalités qu'introduit ce code :

- après avoir binarisé l'image, on met le bord à 0 selon une technique déjà vue pour éviter les problème de débordements ;
- `Col_Pal red = Pdisc(P8COL : :red)` construit un objet `red` de type `Col_Pal` (= couleur de palette); les objets de types `Col_Pal` sont les objets permettant de communiquer à `ELISE` une couleur particulière pour effectuer un affichage vecteur ;
- logiquement une `Col_Pal` est formée de l'association entre une palette et une certaine entrée dans cette palette; les `Col_Pal` sont créées par un appel de fonction sur une palette, les arguments attendus sont tous entiers et le nombre des arguments attendu est égal à la dimension de la palette ;
- `pt = Ecr.clik().pt`, la fonction membre `clik` de la classe `Video_Display` attend un "clique" de l'utilisateur sur une des fenêtres ouvertes par `ELISE` et renvoie un objet de la classe `Clik` qui contient différentes informations sur le "clique" : fenêtre, bouton, point sur lequel on cliqué; *il s'agit d'un processus bloquant*, le programme ne continuera pas tant que l'évènement ne s'est pas produit ;
- enfin on utilise la fonction membre des fenêtres, `draw_circle_loc`, pour dessiner un cercle rouge de rayon 3 centré sur le point de clique ;

Le code de la colonne 2 de la figure 4.7 colorie en magenta la composante 8-connexe des points noirs du points sur lequel on a cliqué, expliquons le fonctionnement de `conc` :

- `Flux_Pts conc(Flux_Pts flx,Neigh_Rel Rel)` prend en paramètre un flux  $F_{lx}$ , une relation  $R_{el}$  et renvoie un flux  $F_{lx}^{Rel}$  qui est la fermeture transitive reflexive <sup>4</sup> de  $F_{lx}$  par  $R_{el}$  ;
- autrement dit  $F_{lx}^{Rel}$  contient  $F_{lx}$ , et les voisins de  $F_{lx}$  selon  $R_{el}$ , et les voisins des voisins de  $F_{lx}$  selon  $R_{el}$ , et les voisins des voisins des voisins ...
- ou encore de manière plus formelle, en posant  $F_{lx}^0 = F_{lx}$  et  $F_{lx}^{n+1} = R_{el}.vois(F_{lx}^n)$   

$$F_{lx}^{Rel} = F_{lx}^0 \cup F_{lx}^{1,Rel} \cup F_{lx}^{2,Rel} \cup \dots \cup F_{lx}^{k,Rel} \cup \dots$$
le produit infini s'arrêtant évidemment à stabilité ;
- du point de vue de l'implémentation, `ELISE` mémorise les points obtenus à chaque étape et les réutilise à l'étape d'après pour calculer leur voisins selon  $R_{el}$  qui deviennent les points de l'étape courante ;
- notons que `ELISE` n'a aucun moyen de savoir comment éviter de mémoriser plusieurs fois les mêmes points, il est donc impératif quand on appelle `conc` d'utiliser une relation qui filtre les points et de gérer un système de marquage permettant de savoir quels points ont déjà été rencontrés ;
- ici l'argument passé à `conc` n'est pas un `Flux_Pts` mais est un point entier (`Ptd2i`), `conc` accepte cet argument parce qu'il existe dans la classe `Flux_Pts` un constructeur `Flux_Pts(Ptd2i pt)` (qui renvoie le `Flux_Pts` réduit à l'unique `pt`) ;

<sup>4</sup>en fait la réflexivité est un peu plus complexe que ça, voir partie II

1	2	3
<pre> // creer une image binaire  Im2D_U_INT1 Ibin(256,256); ELISE_COPY (   W.all_pts(),   rect_median(I.in_proj(),2,256)/8&lt;8,   W.out(Pdisc)   Ibin.out() ); ELISE_COPY (   W.border(1),   0,   Ibin.out()   W.out(Pdisc) );  Pt2di pt; Col_Pal red =Pdisc(P8COL::red);  // Attend que l'on clique sur // un point noir  for (;;) {   cout&lt;&lt;"CLIKER SUR UN POINT NOIR\n";   pt = Ecr.clik()._pt;   if (Ibin.data()[pt.y][pt.x] == 1)   {      // visualise de point      W.draw_circle_loc(pt,3,red);     break;   } } </pre>	<pre> // Affiche la composante connexe // du point sur lequel on a clique  Neighbourhood V8=Neighbourhood::v8(); ELISE_COPY (   conc   (     pt,     Ibin.neigh_test_and_set     (       V8,       P8COL::black,       P8COL::magenta,       20     )   ),   P8COL::magenta,   W.out(Pdisc) ); </pre>	<pre> // Affiche les composantes connexe // passant par la diagonale  Pt2di p1(1,254),p2(254,1); ELISE_COPY (   conc   (     line(p1,p2),     Ibin.neigh_test_and_set     (       V8,       P8COL::black,       P8COL::green,       20     )   ),   P8COL::green,   W.out(Pdisc) );  // visualise la diagonale  ELISE_COPY (   line(p1,p2),   P8COL::red,   W.out(Pdisc) ); </pre>

FIG. 4.7 – Saisie d'un point, calcul de sa composante connexe, calcul de la composante connexe d'une droite.

Dans la majorité des utilisations courantes, le `Flux.Pts` passé à `conc` est réduit à un point ; cependant `Elise` n'impose aucune restriction à ce sujet et le code 3 donne un exemple où le `Flux.Pts` qui sert de germe est une ligne (que l'on dessine en fin de traitement).

La possibilité d'utiliser des germes non ponctuels est parfois assez pratique en analyse d'images, la fi-

1	2
<pre> // Creation d'une image rouge-noir-blanc  Im2D_U_INT1 I2(256,256); ELISE_COPY (   Ibin.all_pts(),   Ibin.in()!=0,   Ibin.out() I2.out() W.out(Pdisc) ); ELISE_COPY (   select   (     Ibin.all_pts(),     Ibin.in()[(FY,FX)]&amp;&amp; (FX&lt;FY) &amp;&amp; (!Ibin.in())   ),   P8COL::red,   I2.out() W.out(Pdisc) ); </pre>	<pre> // composante connexe des points noirs // avec comme germe les points rouges  ELISE_COPY (   conc   (     select(I2.all_pts(),I2.in()==P8COL::red),     I2.neigh_test_and_set     (       V8,       P8COL::black,       P8COL::blue,       20     )   ),   P8COL::blue,   W.out(Pdisc) ); </pre>

FIG. 4.8 – Une utilisation possible de `conc` avec un germe non ponctuel.

figure 4.8 en donne un exemple assez “réaliste” (== mettre en bleu toutes les particules connexes noires qui touchent le rouge).

L'utilisation, de très loin la plus courante de `conc` en analyse d'image est l'analyse de toutes les particules connexes pour des applications de reconnaissances de forme. Le code de la figure 4.10 (avec visualisation sur la figure 4.9) donne une mini-application possible d'analyse en particule connexe utilisant `conc` et un certain nombre d'opérateurs déjà vus. Ce code correspond aux spécifications suivantes :

- parcourir toutes les composantes 8-connexes noires de `Ibin` en les coloriant en vert,
- pour chaque composante possédant plus de 200 points :

1. mettre la particule en cyan
2. calculer son centre de gravité et l'afficher en orange ;
3. calculer sa boîte englobante et l'afficher en bleu ;

L'analyse en particules connexes est un type d'opération avec lequel on mélangera très souvent du code C++ (...for (INT y=0 ; y < 256 ; y++) if (d[y][x] == 1) ...) avec des appels `Elise`. On laisse la compréhension détaillée du code à titre d'exercice, commentons cependant quelques petites fonctionnalités utilisées pour la première fois :

FIG. 4.9 – Résultat du code d’analyse en particules connexes (code sur la figure suivante)

<pre> ELISE_COPY (     Ibin.all_pts(),     Ibin.in()!=0,     Ibin.out() W.out(Pdisc) ); U_INT1 ** d = Ibin.data();  for (INT x=0; x &lt; 256; x++) {     for (INT y=0; y &lt; 256; y++)     {         if (d[y][x] == 1)         {             Liste_Pts_INT2 cc(2);             ELISE_COPY             (                 conc                 (                     Pt2di(x,y),                     Ibin.neigh_test_and_set                     (                         V8,                         P8COL::black,                         P8COL::green,                         20                     )                 ),                 P8COL::green,                 W.out(Pdisc)   cc             );         }     } } </pre>	<pre>         if (cc.card() &gt; 200)         {              // Boite englobante :              Pt2di pmax,pmin;              // centre de gravite :              Pt2di cdg;              // Styles de lignes :              Line_St lstbox(Prgb(0,0,255),2);             Line_St lstcdg(Prgb(255,128,0),3)             ELISE_COPY             (                 cc.all_pts(),                 (FX,FY),                 (pmax.VMax())                   (pmin.VMin())                   (cdg.sigma())                   (W.out(Pdisc) &lt;&lt; P8COL::cyan)             );             W.draw_circle_loc(cdg/cc.card(),3,lstcdg);             W.draw_rect(pmin,pmax,lstbox);         }          // fin if (d[y][x] == 1)      } } </pre>
---	--

FIG. 4.10 – Un code simple d’analyse en particules connexes.

- `Line_St lstbox(Prgb(0,0,255),2)` ; crée un objet de type `Line_St` (style de ligne); les objets `Line_St` sont utilisés pour communiquer à `Élise` le style dans afficher les ordre vecteur de type “dessin au trait” (droite, cercle);

- `Line_St(Col_Pal,REAL)` est un constructeur prenant en paramètre une couleur et une épaisseur; il existe aussi un constructeur `Line_St(Col_Pal)` pour lequel l'épaisseur est spécifiée implicitement à 1.0; ainsi toutes les fonction qui attendent un paramètre de type `Line_St` admettront un paramètre de type `Col_Pal` (ainsi sur la figure 4.7 on a écrit `W.draw_circle_loc(pt,3,red);`);
- on remarque que puisque `Prgb` est de dimension 3, elle prend 3 paramètres, quand elle est utilisée comme fonction pour renvoyer une `Col_Pal`; ainsi `Line_St lstcdg(Prgb(255,128,0),3)` spécifie pour le centre de gravité un style de ligne “orange + épaisseur 3”;
- les fonctions membres de la classe fenêtre graphique `W.draw_circle_loc(Pt2dr,REAL,Line_St)` et `W.draw_rect (Pt2dr,Pt2dr,Line_St)` dessinent un cercle et un rectangle;
- les fonctions membre `Output VMax()`, `Output VMin()` et `Output sigma()` des classes `Ptd2i` et `Pt2dr` renvoient des `Output` de dimension consommées 2 qui ont pour effet de mémoriser en  $x$  et  $y$  les maximum, minimum et sommes des 2 premières coordonnées qui leur sont passées.

## 4.4 Images sur 1,2 et 4 bits

1	2	3
<pre> Im2D_Bits&lt;2&gt; Ibin(256,256); ELISE_COPY (     W.all_pts(),     rect_median(I.in_proj(),2,256)/8&lt;8,     W.out(Pdisc)   Ibin.out() ); </pre>	<pre> for (INT x = 0 ; x&lt; 256; x++)     for (INT y = 0 ; y&lt; 256; y++)     {         INT v = Ibin.get(x,y);         Ibin.set(x,y,v+2);     }  ELISE_COPY (     W.all_pts(),     Ibin.in(),     W.out(Pdisc) ); </pre>	<pre> for (INT y = 0 ; y&lt; 256; y++) {     U_INT1 * d = Ibin.data()[y];     for (INT x = 0 ; x&lt; 64; x++)         d[x] = ~d[x]; }  ELISE_COPY (     W.all_pts(),     Ibin.in(),     W.out(Pdisc) ); </pre>

FIG. 4.11 – Manipulation d’images sur 2 bit.

La figure 4.11 introduit l’utilisation des images de bits; décrivons les caractéristiques générales de ces images :

- ces classes d’images sont des template de classe où l’argument est un entier qui spécifie le nombre de bits sur lequel on code chaque pixels; cet entier peut valoir 1,2 ou 4;
- en interne chaque ligne d’image est “packée”, donc approximativement une image de taille  $t_x, t_y$  sur  $N$  bits consomme  $\frac{t_x * t_y * N}{8}$  octets;
- ce type d’image peut être utile dans des application de type morpho (ou plus généralement manipulation d’images détiquettes);
- à l’intérieur d’un `ELISE_COPY`, ces images s’utilisent comme les autres;

La colonne 1 de la figure 4.11 devrait être claire avec ce qui précède.



La colonne 2 introduit les fonction membre `get(INT x,INT y)` et `set(INT x,INT y,INT val)` permettant de lire et de modifier un par un les pixels de l'image. On offre ces méthodes car le “package” des valeurs complique les manipulations que l'on puet faire sur les image.

Si on veut vraiment optimiser les accès élémentaires, on peut utiliser la fonction `data`, `I.data()[y]` est un tableau packé, en MSBF (most significant bit first), permettant d'accéder à la ligne `y`. Un exemple possible est donné sur la colonne 3 de la figure 4.11, le code devrait être clair pour ceux qui aiment manipuler les bits.

4.5 Erosion et autres “réduction associative sur relation”

1	2
<pre>ELISE_COPY(W.border(1),1,Ibin.out() W.out(Pdisc)); Neighbourhood V8 = Neighbourhood::v8(); ELISE_COPY (   select   (     select(W.all_pts(), Ibin.in()==0),     Neigh_Rel(V8).red_max(Ibin.in())   ),   P8COL::red,   W.out(Pdisc) );</pre>	<pre>// P8COL::red      2 // P8COL::green    3 // P8COL::blue     4 // P8COL::cyan     5 // P8COL::magenta  6  Neighbourhood V4 = Neighbourhood::v4(); ELISE_COPY (   select(W.all_pts(), Ibin.in()==0),   2+Neigh_Rel(V4).red_sum(Ibin.in()),   W.out(Pdisc) );</pre>

FIG. 4.12 –

4.6 Random

4.7 Principaux services offerts

Le type `Rel_Vois`  
PS + raster `Winddw`, `X11` (`WNT`), `Tiff`, `GIF`, `TGA`, `VECTO Image 1,2,3` sur `U.INT1 ...`, `Image 2-D` sur `1,2,4` bits. `Liste.Pts` `Plotters`. Interface pour rajouter ces propres opérateurs.

Caractéristique :

\* Garbage collecté \* tous les accès vérifiés (tableaux etc..)

## 4.8 Avantages/inconvénients

Par rapport à des bibliothèques “classiques”, on peut distinguer les avantages et les inconvénients suivants dans  $\mathcal{E}\mathbb{P}\mathcal{S}\mathcal{e}$  :

- ⊖ il est *relativement* difficile d'effectuer des opérations simples avec  $\mathcal{E}\mathbb{P}\mathcal{S}\mathcal{e}$  car on ne peut pas faire grand chose tant que l'on n'a pas compris le mécanisme d'abstraction de données sur lequel repose  $\mathcal{E}\mathbb{P}\mathcal{S}\mathcal{e}$ .
- ⊕ il est *relativement* facile d'effectuer des opérations complexes avec  $\mathcal{E}\mathbb{P}\mathcal{S}\mathcal{e}$  car, une fois que l'on compris les mécanismes d'abstraction sur lesquels elle repose, il y a peu de chose à savoir pour effectuer ces opérations complexes ;
- ⊖ le temps d'exécution d'un programme développé entièrement sous  $\mathcal{E}\mathbb{P}\mathcal{S}\mathcal{e}$  sera plus long que celui d'un programme écrit en C++ et manipulant directement les données maillées ; à titre d'ordre de grandeur, on peut considérer que l'exécution sous  $\mathcal{E}\mathbb{P}\mathcal{S}\mathcal{e}$  doublera le temps de calcul (mais c'est évidemment très variable suivant les contextes) ; cet inconvénient doit être modulé par le fait qu'il n'y a aucun problème pour mélanger dans une même application du code “traditionnel” avec du code  $\mathcal{E}\mathbb{P}\mathcal{S}\mathcal{e}$  ; à titre d'ordre de grandeur, on peut considérer qu'en réécrivant *a posteriori* les 10% du code critique en terme de temps d'exécution, on peut ramener à 10% le surplus de CPU.

## Chapitre 5

### bilan



## Chapitre 6

# exemples complets

6.1 Égalisation d'histogramme

6.2 Mandelbrot

6.3 Morphing aléatoire

6.4 Correlation sur image de synthèse aléatoire



## Chapitre 7

# Comment ça marche

### 7.1 Les trois types de Flux\_Pts : RLE, integer et real

Les Flux\_Pts RLE sont :

- les rectangles (qu'ils aient été créés par `rectangle`, par `all_pts` ou par `interior`);
- les primitives 2 –  $D$  surfacique (disque, polygone ...);





Deuxième partie

Documentation de référence



# Chapitre 8

## Classes Utilitaires

Ce chapitre décrit plusieurs “petites” classes (ou template de classes) nécessaires pour appeler certaines fonction de plus haut niveaux proposées par `ELISE`.

### 8.1 Les points 2 – $D$ (`Pt2d<Type>`)

Fichier : `"include/general/ptxd.h"`

Pour passer ou récupérer des point de dimension 2 à `ELISE` on utilisera des objets de la classe template `Pt2d<Type>` où `Type` vaudra couramment `INT` ou `REAL`. De manière générale, on préférera utiliser les définitions `Pt2di` et `Pt2dr` plutôt que `Pt2d<INT>` et `Pt2d<REAL>`.

Ce template de classe étant relativement simple, le mieux semble de donner directement la définition de la classe en ne commentant que les quelques fonctions ayant une sémantique non totalement évidente (Ces commentaires viendront dans une version plus achevées de la DOC). Ce code se trouve sur la figure 8.1.

### 8.2 Les point 3 – $D$ (`Pt3d<Type>`)

### 8.3 Les listes (`ElList<Type>`)

Fichier : `"include/general/garb_coll_pub.h"`

Les liste de la classe template `ElList<Type>` sont garbage-collectés (ce qui signifie que l'utilisateur n'a pas à s'occuper de l'allocation ou la libération de la mémoire). Ces listes sont destinées à manipuler de manière aisée de “petites listes”, typiquement passer des argument optionnels à une fonction ou représenter les coins d'un polygone (ou de manière plus générale des liste de vecteurs). Il est déconseiller de les utiliser pour représenter de “grandes” listes d'objets (genre des liste de points raster) car le garbage-collecting impose un sur-coût important en mémoire et en temps de calcul.

Ces liste sont implémentées sous forme de maillon simplement chaînés.

On dispose des méthodes (ou fonctions) suivantes :

- `ElList<Type>()` constructeur pour créer une liste vide;
- `friend ElList<Type> newl(Type)` “constructeur externe” pour créer une liste à un élément.
- `bool empty() const` indique si la liste est vide;
- `INT card() const` renvoie le nombre d'éléments;
- `Type car() const` retourne le premier élément (erreur si vide);
- `Type cdr() const` retourne une liste privée du premier élément(erreur si vide);
- `Type last() const` retourne le dernier élément (erreur si vide); cette opération à un coût proportionnel à la taille de la liste;
- `Type pop()` retourne le `car` et positionne la liste au `cdr` (erreur si vide);
- `ElList<Type> reverse()` renvoie une liste inversée;
- `friend ElList <Type> operator + (ElList<Type> l1, Type val)` renvoie une liste constituée de `val` comme `car` et `l1` comme `cdr`;

**Achtung !!:** une expression telle que `newl(Pt2di(1,1))+Pt2di(2,2)+Pt2di(3,3)` renvoie une liste à 3 éléments dont le *première* élément est `Pt2di(3,3)`.

## 8.4 Les piles `ElFifo<Type>`

Fichier : `"include/ext_stl/fifo.h"`

La classe `ElFifo<Type>` est assez proche de la classe `vector` de la STL (tant du point de vue implémentation que fonctionnalité, mis à part les iterator). Notamment l'implémentation est faite sous forme de tableau dont la taille double juste avant chaque éventuel débordement <sup>1</sup>. Les fonctionnalités qu'offrent les `ElFifo` en plus des `vector` sont :

- la possibilité de rajouter ou supprimer des éléments en tête ;
- la possibilité de les indexer de manière circulaire ; quand cette possibilité est activée, soit  $F$  une `ElFifo` contenant  $n$  éléments, alors  $F[-1]$  est équivalent à  $F[n-1]$  ou  $F[2n-1]$  ...

L'interface publique des `ElFifo<Type>` est composée des méthodes suivantes :

- `ElFifo(int capa = 10, bool circ = false)` pour construire une `ElFifo` initialement vide ; `capa` indique la capacité initiale de la pile, à utiliser si on a une idée suffisamment précise du nombre d'éléments qu'elle contiendra ; `circ` indique si on autorise l'indexation circulaire ;
- `bool circ() const` ; indique si l'indexation circulaire est autorisée ;
- `void set_circ(bool)` positionne l'autorisation d'indexation circulaire ;
- `bool empty() const` indique si elle est vide ;
- `void clear()` ; vide le contenu ;
- `int nb() const` renvoie le nombre d'éléments ;
- `Type operator [] (int k) const` ; renvoie le  $k^{\text{ème}}$  élément ; on doit avoir  $0 \leq k < nb()$  si la liste est non circulaire ; si la liste est circulaire,  $k$  peut être quelconque et est interprété modulo `nb()` ;
- `Type & operator [] (int k)` ; idem précédent pour une `ElFifo` non constante ;
- `Type popfirst()` ; renvoie le premier élément et le supprime ;
- `Type poplast()` ; renvoie le dernier élément et le supprime ;
- `void pushlast(Type)` ; rajoute un élément à la fin ;
- `void pushfirst(Type)` ; rajoute un élément au début ;
- `Type top() const` ; renvoie le dernier élément ;
- `INT capa() const` ; renvoie la capacité (pour info, éventuellement debugage de problèmes mémoire) ;

**Achtung !!:** on ne peut pas faire de copie des `ElFifo<Type>` (le constructeur de copie est volontairement décalré privé et non instancié).

**Achtung !!:** dans l'implémentation actuelle, les `ElFifo` ne doivent être utilisées que pour des objets sans destructeur (ie dont le destructeur ne fait rien d'intéressant) et dont la copie peut se faire par `memcpy` <sup>2</sup>.

<sup>1</sup>il est d'ailleurs possible qu'un de ces jours je base mon implémentation sur la STL

<sup>2</sup>ça va changer

<pre> template &lt;class Type&gt; class Pt2d : public ElTypeNum&lt;Type&gt; { public :      typedef Type (&amp; t2)[2] ;     Type    x;     Type    y;  // Constructeur      Pt2d&lt;Type&gt;() : x (0), y (0) {} ;     Pt2d&lt;Type&gt;(Type X,Type Y) ;     Pt2d&lt;Type&gt;(const Pt2d&lt;INT&gt;&amp; p) ;     Pt2d&lt;Type&gt;(const Pt2d&lt;REAL&gt;&amp; p) ;  // Pseudo-Constructeur, a partir de coordonnees polaires      static Pt2d&lt;Type&gt; FromPolar(REAL rho,REAL teta);  // Operateurs  // unaires, Pt =&gt; Pt      Pt2d&lt;Type&gt; operator - () const ;     Pt2d&lt;Type&gt; yx() const { return Pt2d(y,x);};  // binaires, PtxPt =&gt; Pt      Pt2d&lt;Type&gt; operator + (const Pt2d&lt;Type&gt; &amp; p2) const;     Pt2d&lt;Type&gt; operator * (const Pt2d&lt;Type&gt; &amp; p2) const;     Pt2d&lt;Type&gt; operator - (const Pt2d&lt;Type&gt; &amp; p2) const;  // * et / coordonnee par coordonnees; par ex ==&gt; Pt2d(x1*x2,y1*y2)      Pt2d&lt;Type&gt; mcbyc(const Pt2d&lt;Type&gt; &amp; p2) const;     Pt2d&lt;Type&gt; dcbyc(const Pt2d&lt;Type&gt; &amp; p2) const;  // par ex Sup ==&gt; (Max(x1,x2),Max(y1,y2))      friend Pt2d&lt;Type&gt; Sup ( Pt2d&lt;Type&gt; p1, Pt2d&lt;Type&gt; p2);     friend Pt2d&lt;Type&gt; Inf ( Pt2d&lt;Type&gt; p1, Pt2d&lt;Type&gt; p2);  // binaire, affectation composee      Pt2d&lt;Type&gt; &amp; operator += (const Pt2d&lt;Type&gt; &amp; p2);  // binaire, comparaison, PtxPt ==&gt; bool      bool operator == (const Pt2d&lt;Type&gt; &amp; p2) const;     bool operator != (const Pt2d&lt;Type&gt; &amp; p2) const; </pre>	<pre> // binaires, PtxScalaire ==&gt; Pt      Pt2d&lt;Type&gt; operator * (Type lambda) const;     Pt2d&lt;Type&gt; operator / (Type lambda) const;  // binaires, PtxPt ==&gt; scalaire; produitx scalaire et vectoriel      friend Type scal(const Pt2d&lt;Type&gt; &amp; p1,const Pt2d&lt;Type&gt; &amp; p2);     Type operator ~ (const Pt2d&lt;Type&gt; &amp; p2) const;  // lies a une distance      friend Type dist4(const Pt2d&lt;Type&gt; &amp; p);     friend Type dist8(const Pt2d&lt;Type&gt; &amp; p);     friend REAL euclid(const Pt2d&lt;Type&gt; &amp; p);     friend REAL euclid(const Pt2d&lt;Type&gt; &amp; p1,const Pt2d&lt;Type&gt; &amp; p2);      bool in_box(const Pt2d&lt;Type&gt; &amp; p0, const Pt2d&lt;Type&gt; &amp; p1);     friend void pt_set_min_max(Pt2d&lt;Type&gt; &amp; p0,Pt2d&lt;Type&gt; &amp; p1);  // vus dans les chapitre d'INTRO      Output sigma();     Output VMax();     Output VMin();     Output WhichMax();     Output WhichMin();  // pluto a usage interne ELISE      void to_tab(Type (&amp; t)[2] ) const;  private :     void Verif_adr_xy(); };  // Fonctions specifiques a un des types de points  // points entiers      REAL average_euclid_line_seed (Pt2di);     Pt2di best_4_approx(const Pt2di &amp; p);     Pt2di second_freeman_approx(Pt2di u, bool conx_8,Pt2di ui);     INT num_4_freeman(Pt2di);      Pt2di corner_box_included(Pt2di pmin,Pt2di pmax,bool left,bool down);  // points reels      inline Pt2di round_ni(Pt2dr p);     inline Pt2dr rot90(Pt2dr p);     inline Pt2dr vunit(Pt2dr p); </pre>
---	---

FIG. 8.1 – Header de la classe Pt2d (les définition des méthodes inline ont été supprimées, sauf quand elle correspondait au commentaire le plus simple).



## Chapitre 9

# Fenêtres graphique

### 9.1 Création de couleur, la classe `Elise_colour`

Fichier "include/general/colour.h".

#### 9.1.1 Création

```
static Elise_colour rgb(REAL rr,REAL gg,REAL bb);

static Elise_colour cmy(REAL,REAL,REAL);

static Elise_colour gray(REAL);

Elise_colour();
```

Inspection :

```
REAL r(), REAL g(), REAL b()
```

#### 9.1.2 Opération

```
REAL eucl_dist (const Elise_colour &);

friend Elise_colour operator - (Elise_colour,Elise_colour);

friend Elise_colour operator + (Elise_colour,Elise_colour);

friend Elise_colour operator * (REAL,Elise_colour);

friend Elise_colour som_pond(Elise_colour C1,REAL pds,Elise_colour C2);
```

#### 9.1.3 Conversion

```
void to_its(REAL & i,REAL & t, REAL & s);

static Elise_colour its(REAL i,REAL t,REAL s);
```

#### 9.1.4 Couleur prédéfinie

Les 8 couleurs "primaires" :

red, green, blue, cyan, magenta, yellow, black, white.

Mais aussi :

medium\_gray, brown, orange, pink, kaki, golfgreen, coterotie, cobalt, caramel, bishop, sky, salmon, emerald.

## 9.2 Palettes de couleur

### 9.2.1 la classe de base Elise\_Palette

```
INT nb();
Fonc_Num to_rgb(Fonc_Num);
```

### 9.2.2 la classe Lin1Col\_Pal

```
Lin1Col_Pal(Elise_colour,Elise_colour,INT nb);

Col_Pal operator () (INT);
```

### 9.2.3 la classe Gray\_Pal

```
Gray_Pal(INT nb);

Col_Pal operator () (INT);
```

### 9.2.4 la classe BiCol\_Pal

```
BiCol_Pal (
    Elise_colour c0,
    Elise_colour c1,
    Elise_colour c2,
    INT nb1,
    INT nb2
);

Col_Pal operator () (INT,INT);
```

### 9.2.5 la classe TriCol\_Pal

```
TriCol_Pal (
    Elise_colour c0,
    Elise_colour c1,
    Elise_colour c2,
    Elise_colour c3,
    INT nb1,
    INT nb2,
    INT nb3
);

operator () (INT,INT,INT);
```

### 9.2.6 la classe RGB\_Pal

```
RGB_Pal (INT nb1, INT nb2, INT nb3);

operator () (INT,INT,INT);
```



### 9.2.7 la classe Circ\_Pal

```

Circ_Pal (
    L_El_Col,
    INT NB,
    bool reverse = false
);

static Circ_Pal PCIRC6(INT NB);
class Col_Pal operator () (INT);

```

<pre> Fonc_Num f = polar((FX-128,FY-128),0); ELISE_COPY(I.all_pts(),P8COL::black,W.odisc()); ELISE_COPY (     select(I.all_pts(),(f.v0(&gt;30)&amp;&amp;(f.v0(&lt;100)),     f.v1()*256.0/(2*PI),     W.ocirc() ); </pre>	
---	--

FIG. 9.1 – Code pour générer le cercle des couleur de l’arc en ciel.

### 9.2.8 la classe Disc\_Pal

#### 9.2.8.1 Fonction habituelles

Les constructeurs :

```

Disc_Pal(Elise_colour *,INT nb);

Disc_Pal ( L_El_Col, bool reverse = false);

```

Col\_Pal operator () (INT).

#### 9.2.8.2 Inspection

```

void getcolors(Elise_colour *);

Elise_colour * create_tab_c();

```

#### 9.2.8.3 La palette discrète “standard” P8COL

La fonction membre static Disc\_Pal P8COL();  
La classe P8COL.

#### 9.2.8.4 Réduction de couleur

```

Disc_Pal reduce_col(Im1D_INT4 lut,INT nb_cible);

```

### 9.3 Gérer les couleur 8-bits la classe `Elise_Set_Of_Palette`

### 9.4 Les styles graphiques

Fichier "include/general/graphics.h".

#### 9.4.1 La classe `Col_Pal`

#### 9.4.2 La classe `Line_St`

#### 9.4.3 La classe `Fill_St`

### 9.5 La classe de base `El_Window`

Fichier "include/general/window.h".

#### 9.5.1 Utilisation comme Output mode mailé

Output `out(Elise_Palette)` ;

Les raccourcis :

- `ogray()` ;
- `orgb()` ;
- `odisc()` ;
- `obicol()` ;
- `ocirc()` ;
- `olin1()` ;

#### 9.5.2 Les ordres de dessin vecteurs

- `void draw_circle_loc(Pt2dr,REAL,Line_St)` ;
- `void draw_circle_abs(Pt2dr,REAL,Line_St)` ;
- `void draw_seg(Pt2dr,Pt2dr,Line_St)` ;
- `void draw_rect(Pt2dr,Pt2dr,Line_St)` ;
- `void fill_rect(Pt2dr,Pt2dr,Fill_St)` ;

#### 9.5.3 Visualisation de graphes raster

Output `out_graph(Line_St,bool sym = true)` ;

#### 9.5.4 Changement de géométrie, la fonction `chc`

#### 9.5.5 Mise en parallèle de fenêtre operator `|`

#### 9.5.6 Diverses fonctions utilitaires

La fonction `Disc_Pal pdisc()` (et celles à rajouter).

`Pt2di sz() const` ;

### 9.6 Les fenêtres sur la sortie vidéo, la classe `Video_Win`

#### 9.6.1 La classe `Video_Display`

Construction `Video_Display(const char * name)` ;.

Chargement de palette `void load(Elise_Set_Of_Palette)` ;

### 9.6.2 Construction et fonctions utilitaires

```

Video_Win
(
    Video_Display      ,
    Elise_Set_Of_Palette ,
    Pt2di              ,
    Pt2di              ,
    INT                border_witdh = 5
);

void set_title(char * name);
void set_cl_coord(Pt2dr,Pt2dr);
Video_Win chc(Pt2dr,Pt2dr);
void clear();

```

### 9.6.3 Récupération d'événement

La classe Klik.

La fonction Video.Display : :klik.

## 9.7 Les fenêtres postscript PS\_Window

### 9.7.1 Fichier postscript, la classe PS\_Display

Construction :

```

PS_Display
(
    const char * name,
    const char * title,
    Elise_Set_Of_Palette,
    bool        auth_lzw = true,
    Pt2dr        sz_page = A4 // in cm
);

void comment(const char *);
PS_Window w_centered_max(Pt2di sz,Pt2dr margin);

```

### 9.7.2 Fenêtre postscript, la classe PS\_Window

Construction :

```

PS_Window
(
    PS_Display,
    Pt2di sz,
    Pt2dr p0,
    Pt2dr p1
);

PS_Window chc(Pt2dr,Pt2dr);

```

### 9.7.3 Mosaïque de fenêtre, la classe Mat\_PS\_Window

Construction

```
Mat_PS_Window
(
    PS_Display,
    Pt2di sz,
    Pt2dr margin,
    Pt2di nb,
    Pt2dr inside_margin
);
```

## 9.8 Les fenêtres raster Bitm\_Win

Constuction :

```
Bitm_Win
(
    const char *,
    Elise_Set_Of_Palette,
    Pt2di sz
);

Im2D_U_INT1 im() const;
Bitm_Win chc(Pt2dr,Pt2dr);
```

# Chapitre 10

## Images et Liste de Points

Fichier : `"include/general/bitm.h"`

### 10.1 Généralités sur classes images

#### 10.1.1 Organisation des classes images

FIG. 10.1 – Arbre d’héritage des classe image

#### 10.1.2 méthode de lecture-écriture de GenIm

En lecture trois fonctions membre :

- `Fonc.Num in(void)` ; pas de valeur par défaut, si on sort, génère une erreur (ou “segmentation violation-core dumped” si vérif-`ELSE` inhibée) ;
- `Fonc.Num in(REAL def_out val)` ; valeur par défaut = `val` (elle est déclarée `REAL` pour simplifier l’interface, mais si l’image est sur un type entier le résultat sera une fonction entière) ;
- `Fonc.Num in_proj()` ; prolongement avec continuité aux bord ;

Pour ces trois fonctions :

- si on les utilise avec des flux de points entier, la `Fonc.Num` retournée sera entière si le type de l’image est un type intégral et réel si le type de l’image est un type flottant ;
- si on les utilise avec des flux de points réels, la `Fonc.Num` retournée sera toujours réelle, `ELSE` utilise l’interpolation bilinéaire pour calculer des valeurs entre les pixels ;

En écriture, deux fonctions :

- `Output out(void)` ; pour écrire dedans, si on déborde ça génère une erreur (sauf pour les flux surfaciques ou c’est pas cher de vérifier) ;

- `Output oclip(void)` ; les ordre d’écriture sont clippés, ne génère pas d’erreur si on déborde ;
- En lecture-écriture, fonctions membres de la famille histogramme :
- `Output histo(bool auto_clip = false)` pour écrire dans une image en accumulant par somme (du genre `b[y][x] += val` voir 3.4-page 48 et 3.4-page 49) ; si `auto_clip` vaut `true` les débordement sont correctement gérés, sinon une erreur ou un cor-dump apparaît ;
  - Les 4 fonction membre suivante font une accumulation par somme, maximum, minimum et multiplié (`sum_eg` est strictement équivalente à `histo`) :
    - `Output sum_eg(bool auto_clip = false) ;`
    - `Output max_eg(bool auto_clip = false) ;`
    - `Output min_eg(bool auto_clip = false) ;`
    - `Output mul_eg(bool auto_clip = false) ;`
  - `Output oper_ass_eg(const OperAssocMixte & op, bool auto_clip) ;`, par exemple `oper_ass_eg(OpMin, auto_clip)` est strictement équivalent à `min_eg(auto_clip)` ;
- Pour les opérations d’écriture (et de lecture-écriture) les `Flux.Pts` ne peuvent pas être réels.

### 10.1.3 L’énumération `GenIm : :type_el`

L’énumération `GenIm : :type_el` est rarement utilisée dans directement dans les classes images (car typage static dans les classe template), utilisée pour récupérer ou spécifier dynamiquement un type d’élément pour la manipulation des fichiers d’image (surtout utile pour certaines manipulation “fine” de fichier Tiff).

Un `GenIm : :type_el` peut prendre les valeurs suivantes :

- `u_int1, int1, u_int2, int2, int4, real4, real8` pour les types normaux ;
- `bits1_msb, bits2_msb, bits4_msb, bits1_lsb, bits2_lsb, bits4_lsb`, pour les types sur 1, 2 ou 4 bits ; `msb` signifie “most significant bit first” (`lsb` = “last ...”) ; l’intérêt de distinguer `msb` et `lsb` ne se justifie que pour la gestion de fichier ; en accord avec les standar Tiff, on recommande de toujours utiliser `msb` quand c’est possible ;

Quelques petites fonction globales permettant de manipuler ou créer des `GenIm : :type_el` :

- `INT nbb_type_num(GenIm : :type_el type_el) ;` nombre de bits ;
- `bool msb_type_num(GenIm : :type_el type_el) msb ou lsb`, génère une erreur pour les types normaux ;
- `bool signed_type_num(GenIm : :type_el type_el) ;` est-ce signé, génère une erreur pour les types en virgule flottante ;
- `bool type_im_integral(GenIm : :type_el type_el) ;` est-ce un type en virgule flottante ;
- `GenIm : :type_el type_u_int_of_nbb(INT nbb, bool msb = true) ;` revoie un type correspondant
- `void min_max_type_num(GenIm : :type_el, INT &v_min, INT &v_max) ;` renvoie les valeur max et min codable sur le type, conventionnellement `v_max = v_min + 1` pour les types où le résultat ne peut être codé correctement (`int4, real4, real8`) ;

Deux petites fonction pour créer des images de type variable :

- `GenIm alloc_im1d(GenIm : :type_el type_el, int tx, void * data = 0) ;`
- `GenIm alloc_im2d(GenIm : :type_el type_el, int tx, int ty) ;`

### 10.1.4 Autre fonctionnalité des `GenIm`

Quelques petites fonctions membre :

- `bool same_dim_and_sz(GenIm) ;` est-ce que deux images ont la même dimension et la même taille dans toutes les dimensions ;
- `load_file(class Elise_File_Im)` le fichier et l’image doivent avoir la même taille ; si ils sont du même type, `Elise` optimise le chargement sinon elle rapelle `ELISE_COPY` ;

## 10.2 Images “standard”

### 10.2.1 Propriété commune aux image standard

Soit  $k$  un entier représentant la dimension d’une image (pour l’instant  $k = 1, 2$  ou  $3$ ) :

- les classes permettant de représenter des images de dimension  $k$  sont des template de classe : `ImkD<T1, T2>` (voir 3.3) ;

- le premier argument `T1` est l’argument “important”, il correspond au type sur lequel sont effectivement stocké les éléments du tableaux ;
- le deuxième argument, `T2`, est en fait totalement conditionné par le premier, il doit valoir `INT` pour tout les type entiers (`U_INT1, INT1 ...`) et `REAL` pour les types en virgules flottantes ;

Ces classe possèdent les constructeurs suivants :

- `ImkD<T1,T2>(INT t1, ..., INT tk)` pour créer une image dont les élément ne seront pas initialisés (mais évidemment la mémoire est allouée) ; par exemple `Im1D<T1,T2>(INT tx)` ou `Im3D<T1,T2>(INT tx, INT ty, INT tz)` ;
- `ImkD<T1,T2>(INT t1, ..., INT tk, T2 vinit)` cette fois ci tous les éléments sont initialisés à `vinit` ;
- `ImkD<T1,T2>(INT t1, ..., INT tk, const char * StrInits)` les éléments sont initialisés en allant lire la chaîne `StrInits` ; `EiSe` utilise `sprintf` pour interpréter `StrInits`, se conformer à la doc du C++ pour savoir quels sont les formats admissible ; voir figure 3.9 page 45 un exemple d’utilisation ; utilisée pour l’initialisation de petits tableaux genres paramètres de filtrage ;

On dispose des fonctions membres suivantes :

- `INT tx() const` return la taille en  $x$  ;
- `INT ty() const` (si  $k \geq 2$ ) return la taille en  $y$  ;
- `INT tz() const` (si  $k \geq 3$ ) return la taille en  $z$  ;
- `INT vmax() const` la valeur max du type (`INT vmin() const` est à rajouter) ;
- `T1 * data() (k = 1)`, `T1 ** data() (k = 2)` `T1 *** data() (k = 3)` renvoie la zone de donnée de l’image utilisable, par exemple pour  $k = 3$ , sous la forme `d[z][y][x]` ;

### 10.2.2 Image “standard” de dimension 2

Les images de dimension 2, possèdent aussi les fonctions membres suivantes :

- `T1 * data_lin()`, la zone de donnée sous forme de tableau  $1 - D$ , utilisable pour des manipulation “à l’ancienne” ; explicitons :
  - `Im2D<U_INT1,INT> i2(10,20)` ;
  - `U_INT1 dl * = i2. data_lin()` ;
  - `U_INT1 d * = i2. data()` ;
  - alors `&dl[x+10*y]` est toujours égal à `&d[y][x]` ;
- trois fonctions permettant de créer des relation de voisinage conditionnelles avec filtrage intégré (voir colonne 3, figure 4.6, page 58) :

1. `Neigh_Rel neigh_test_and_set(Neighbourhood, Im1D<INT4,INT> sel, Im1D<INT4,INT> update)` ;
2. `Neigh_Rel neigh_test_and_set(Neighbourhood, INT sel, INT udapte, INT v_max)` ;
3. `Neigh_Rel neigh_test_and_set(Neighbourhood, EList<Pt2di>, INT v_max)` ;

La version 1 est la plus générale (les deux autres ne sont qu’une interface á cette version) ; plutôt qu’un long discours, commentons l’exemple de la figure 10.2 :

- `r1` désigne une relation où les voisins d’un points sont les 4-voisins qui valent 4, 6, 7 ou 8 (coefficient non nul de `sel`) dans `i2`, chaque fois qu’un point est atteint il est mis à 2 si sa valeur dans `i2` était de 4 et à 3 si sa valeur dans `i2` était de 6, 7 ou 8 (coefficients de `update`) ; par ailleurs on garantie que les éléments de `i2` ne seont jamais supérieurs à 10 ; si `sel` et `update` n’avaient pas la même taille, une erreur serait générée ;
- `r2` désigne exactement la même relation avec les mêmes effet de mise à jour que `r1` : les points de liste passée en paramètre à la création de `r2` sont interprété comme des couples “valeur à sélectionné (p.x) - valeur d’update (p.y)” ;

### 10.2.3 Image “standard” de dimension 1 et 3

Pour l’instant rien de plus que ce qui est commun aux images standard ( 10.2.1).

## 10.3 Image de dimension 2 sur 1, 2 et 4 bits

`EiSe` fournit des types d’images sur 1, 2 ou 4 bits. Pour l’instant ces images n’existent qu’en dimension 2 ; on verra plus tard si il y a un besoin pour les autres dimensions. Ces images s’utilisent donc comme

```

Im1D_INT4 sel (10,"0 0 0 0 1 0 1 1 1 0");
Im1D_INT4 update(10,"0 0 0 0 2 0 3 3 3 0");

Im2D_U_INT1 i2(20,20);

// selectionne les (x,y) tq i2[y][x] = 4, 6 , 7 ou 8
// met a 2 deux tels que i2[y][x] = 4 et a 3 les autres

Neigh_Rel r1 = i2.neigh_test_and_set(Neighbourhood::v4(),sel,update);

Neigh_Rel r2 = i2.neigh_test_and_set
(
    Neighbourhood::v4(),
    newl(Pt2di(4,2))+Pt2di(6,3)+Pt2di(7,3)+Pt2di(8,3),
    10
);

```

FIG. 10.2 – Exemple d'utilisation de `neigh_test_and_set`.

les autres tant qu'on les manipule avec `ELISE_COPY`.

Ces classes d'images sont des template dont les argument sont `const INT` et non des classes. En plus des méthodes héritées de la classe `GenIm`, on a les méthodes suivantes qui ont la même signification que pour les images standar :

- `Im2D_Bits(INT tx, INT ty)` et `Im2D_Bits(INT tx, INT ty, INT v_init)`, les deux constructeurs (pas de construction avec des `const char *`);
- `INT tx() const INT ty() const INT vmax() const`;

Il existe une méthode `U_INT1 ** data()` qui retournent la zone de données. Si on écrit `U_INT1 ** d= i.data()` alors `d[y]` représente la ligne d'image  $y$ , il s'agit d'une image compactée en `msbf`; par exemple avec une image sur 2 bits, `(d[99][0]>>4)&3` représente la valeur de l'image pour  $x = 2, y = 99$ , à utiliser si on est vraiment pressé.

Il existe deux méthode, un peu plus lentes, mais plus simples pour manipuler les imgs de bits élément par élément :

- `INT get(INT x, INT y)` pour récupérer la valeur d'un pixel;
- `void set(INT x, INT y, INT v)` pour modifier la valeur d'un pixel;

## 10.4 Liste de points

### 10.4.1 la classe `Lin1Col_Pal`



# Chapitre 11

## Fichiers Images

Fichier : `"include/general/file_im.h"`

Fichier : `"include/general/tiff_file_im.h"`

### 11.1 la classe ElGenFileIm

#### 11.1.1 Organisation des classes fichier-images

FIG. 11.1 – Arbre d’héritage des classe fichiers-image

La figure 11.1 représente l’arbre d’héritage des classes permettant de manipuler des fichiers images sous `Elise`. La classe `ElGenFileIm` a vocation à décrire les fonctionnalités communes à tous les formats d’image supportés par `Elise`.

On remarque que les format `TGA` et `BMP` ne dérivent pas de `ElGenFileIm`, en fait je n’ai pas eu le temps de réorganiser ces 2 classes ; ça viendra peut-être plus tard, sachant que ce n’est pas une priorité pour ce que je considère comme des “mauvais” format (`TGA` n’apporte pas grand chose, `BMP` encore moins et les spécification ne sont pas très claires). Je ne détaillerai pas pour l’instant ces format, en cas de nécessité, consulter les fichiers `"src/bench/tga.cpp"` et `"src/bench/bmp.cpp"` pour avoir des exemples d’utilisation.

Les 3 formats “pleinement” supportés sont :

- le format `GIF` que j’ai conservé, malgré la controverse `LZW`, parce que c’est un petit format bien spécifié et très répandu pour les images circulant sur le WEB ;
- le format `TIF` parce qu’il s’agit du seul format standard qui soit suffisamment riche pour pouvoir être (presque) considéré comme un format universel ;
- un format interne `Elise_File_Im` ; l’intérêt de rajouter encore un nouveau format n’est pas forcément évident, cependant :

- il n’existait pas à ma connaissance de standard permettant de stocker des images de dimensions quelconques ;
- ce format me permet d’émuler d’autres formats simples tels que les `pgm`, `ppm`, `pbm` (en fait à peu près tous les formats non comprimés) ;

Les formats que j’envisage de rajouter sont les suivants :

- le format PNG, qui est assez bien fait, clairement spécifié et pourrait, à terme, remplacer GIF pour les échanges sur le NET (l’algorithme de compression utilisé étant libre de toute protection) ;
- un des formats JPEG dès que j’aurai le temps de trouver une bonne librairie freeware ;

### 11.1.2 Lecture-Écriture sur les `ElGenFileIm`

Il existent trois fonctions-membre de la classe `ElGenFileIm` permettant de lire ou écrire dans un fichier-image :

- `Fonc_Num in()` permet de lire un fichier-image ; tout débordement génère une erreur ;
- `Fonc_Num in(REAL)` permet de lire une image en donnant une valeur par défaut au point qui sortent de l’image ; il n’existe pas (du moins pas encore) de prolongement par projection comme avec les images en RAM (voir `Fonc_Num GenIm : :in_proj()` en 10.1.2) ;
- `Output out()` permet d’écrire dans un fichier-image ; si le fichier image est comprimé, une erreur peut se déclencher (voir les détails dans les sections spécifiques à chaque format) ;

**Achtung !!** : pour des raisons d’efficacité, ces 3 fonctions ne peuvent être utilisées qu’avec un `Flux_Pts` de type `RLE` (voir 7.1).

Les fichiers images héritent de la classe `Rectang_Object` les fonctions membres `Fonc_Num inside()` `const`, `Flux_Pts all_pts()` `const`, `Flux_Pts interior(INT)` `const`, `Flux_Pts border(INT)` `const`.

### 11.1.3 Information sur les fichiers `ElGenFileIm`

Les fonctions membres suivantes permettent d’inspecter le format des fichiers images :

- fonction liées aux caractéristiques logiques de l’image :
  - `INT Dim()` `const` renvoie la dimension (c.a.d. 2 pour les formats standards tels que GIF ou TIF) ;
  - `const int * Sz()` renvoie un tableau contenant la taille de l’image (par exemple, `Sz()[0]` est la taille en  $x$ ) ;
  - `INT NbChannel()` `const` renvoie le nombre de canaux ;
- fonction liées à la représentation des valeurs numériques :
  - `bool SigneType()` `const` est-ce que les valeurs sont signées ;
  - `bool IntegralType()` `const` est-ce que les valeurs sont entières ou flottantes ;
  - `int NbBits()` `const` nombre de bits sur lequel est représentée chaque valeur ;
- fonctions liées à l’organisation physique du fichier sur le disque :
  - `const int * SzTile()` `const` taille du dallage ; avec les formats supportés actuellement, toujours égales à `const int * Sz()` sauf pour les fichiers TIFF ;
  - `bool Compressed()` `const` est-ce que le fichier est comprimé (auquel cas il y aura des limites supplémentaires sur les opérations d’écritures) ;

## 11.2 la classe `Elise_File_Im`

### 11.2.1 Constructeur

Physiquement les fichiers au format `Elise_File_Im` sont organisés de la façon suivante :

- le début de fichier contient un nombre arbitraire d’octets non lus ;
- ensuite les données non comprimées sont stockées d’un seul bloc ; dans le cas d’un fichier de dimension 2, les pixels d’une même ligne sont consécutifs ; dans le cas de la dimension 3 les pixels d’un même plan horizontal sont consécutifs (et à l’intérieur d’un plan les pixels sont codés comme en dimension 2) ; dans le cas de la dimension 4 ...
- quand il y a plusieurs canaux, les valeurs du même pixel sont consécutives ;
- pour les images sur moins de 1,2,4 bits, on stocke en `MSBF`, les lignes sont paddées ;

```

Elise_File_Im::Elise_File_Im
(
    const char *    name,
    INT             dim,
    INT *           sz,
    GenIm::type_el  type_el,
    INT             nb_channel,
    INT             offset_0,
    INT             _szd0 = -1,
    bool            create = false
);

```

FIG. 11.2 – Création d'un fichier `Elise_File_Im`.

Le constructeur “principal” est représenté sur la figure 11.2. La sémantique des paramètres `name`, `dim`, `sz`, `type_el`, `nb_channel` devrait être assez évidente. Commentons donc les 3 derniers paramètres

- `offset_0` indique le nombre d'octet à sauter avant d'arriver à la zone de données ;
- `_szd0`, le plus simple est de toujours lui donner la valeur `-1` (en fait il permet d'avoir une taille physique de ligne plus grande que la valeur logique, ce qui me permet de gérer simplement le padding effectué par la plupart des formats) ;
- `create` indique si il faut créer le fichier ;

<pre> // Fichier de dimension 1  Elise_File_Im (     const char *    name,     INT             sz,     GenIm::type_el  type_el,     INT             offset_0 = 0,     bool            create = false ); </pre>	<pre> // Fichier de dimension 2  Elise_File_Im (     const char *    name,     Pt2di           sz,     GenIm::type_el  type_el,     INT             offset_0 = 0,     bool            create = false ); </pre>	<pre> // Fichier de dimension 3  Elise_File_Im (     const char *    name,     Pt3di           sz,     GenIm::type_el  type_el,     INT             offset_0 = 0,     bool            create = false ); </pre>
--	--	--

FIG. 11.3 – Interface simplifiée pour les fichiers `Elise_File_Im` de dimension 1, 2 ou 3.

Pour les fichiers de dimension 1, 2 ou 3 il existe une interface simplifiée où la taille est donnée par un `INT`, un `Pt2di` ou un `Pt3di`. La figure 11.3 donne la signature de ces 3 constructeurs simplifiés.

Les formats étant non comprimés, il n'y a aucune restriction pour l'écriture dans ces fichiers (autre que l'utilisation de `Flux_Pts RLE`).

### 11.2.2 Support du format `pnm`

Il n'existe pas de classe particulière au format `pnm`, par contre `Elise` offre plusieurs fonctions membres `static` dans la classe `Elise_File_Im` pour manipuler ces formats en les décrivant comme des `Elise_File_Im` particuliers.

La fonction `static Elise_File_Im pnm(const char *)` permet de décrire un fichier `ppm`, `pgm` ou `pbm` existant comme un `Elise_File_Im` (`Elise` se contente de lire l'entête puis de faire appel au constructeur général avec les bons paramètres).

Il existe trois fonctions permettant de créer ces fichiers, elles sont données sur la figure 11.4. À noter

<pre>static Elise_File_Im pbm (     const char *,     Pt2di sz,     char ** comment = 0 );</pre>	<pre>static Elise_File_Im pgm (     const char *,     Pt2di sz,     char ** comment = 0 );</pre>	<pre>static Elise_File_Im ppm (     const char *,     Pt2di sz,     char ** comment = 0 );</pre>
--	--	--

FIG. 11.4 – Fonctions permettant de créer des fichier `pnm`.

qu'il est possible de rajouter un commentaire personnel en entête de fichier grâce au paramètre `char ** comment`.

## 11.3 Le format GIF

Voir [1].

### 11.3.1 Fonctions spécifiques à la classe `Gif_Im`

La classe `Gif_Im` possède un unique constructeur `Gif_Im(char * name)`; le fichier doit exister (et être un fichier GIF cohérent).

Les fichier images de la class `Gif_Im` possèdent les fonctions membre spécifiques suivantes :

- `Im2D_U_INT1 im()` pour renvoyer directement le contenu d'un fichier dans une image; cette fonction peut être un peu plus rapide que de passer par un `ELISE_COPY` et `Fonc_num Gif_Im : :in()`, la différence de vitesse n'est significative que pour les fichier GIF entrelacés;
- `Disc_Pal pal()` pour obtenir la palette discrète associée au fichier;
- `Pt2di sz()` pour obtenir la taille du fichier;

**Achtung !!:** la méthode `Output Gif_Im : :out()` génère une erreur, en effet le format de fichier GIF n'est pas fait pour l'édition incrémentale. Il s'agit d'un format de fichier comprimé et sans découpage par bloc, la seule opération efficace que l'on peut effectuer sur ce type de fichier est de les créer d'un seul bloc <sup>1</sup>. Pour créer un fichier GIF on utilise la méthode statique `create` :

- `static Output create(char * name, Pt2di sz, Elise_colour * tec, INT nbb;`
- l'`Output` doit être utilisé avec un rectangle 2-D dont correspondant exactement au rectangle `[0 sz.x[ × [0 sz.y[`;
- sinon le comportement est assez naturel, le résultat de la fonction est écrit dans un fichier GIF de nom `name` où `nbb` spécifie le nombre de bits de la palette de couleur et `tec` spécifie les entrées dans cette palette;

### 11.3.2 Image multiples, la classe `Gif_File`

Le format GIF permet de stocker plusieurs images dans un seul fichier. Cette possibilité est notamment utilisée pour représenter et télécharger sur le WEB de petites séquences animées (les "ANIMAGIF"). `ELISE` n'offre (aujourd'hui) aucune facilité pour la création de ces ANIMAGIF, par contre il est possible de les récupérer image par image à l'aide de la classe `Gif_File` dont les fonctions membres sont les suivantes :

- `Gif_File(char * name)` pour construire un objet à partir d'un fichier existant;
- `INT nb_im () const` pour obtenir le nombre d'images incluses dans un fichier GIF;
- `Gif_Im kth_im (INT k) const` pour obtenir la  $k^{ème}$  sous-image du fichier;

## 11.4 Le format Tiff

### 11.4.1 Généralités

On pourra consulter [4] pour une description détaillée du format TIFF et de ses options.

<sup>1</sup> ceci devrait peut être se moduler avec la notion d'image multiple, mais bof

#### 11.4.1.1 Principales caractéristiques supportées

Les caractéristiques supportées par `ExifSe` incluent entre autres <sup>2</sup> toutes les caractéristiques des “basic-tiff reader” (selon la classification [4]). De manière synthétiques sont supportés :

- les fichiers issus du monde MAC comme du monde PC (option `MSByteF/LSByteF`)<sup>1</sup>
- les images non comprimées et les formats de compression : `LZW`, `PackBit`, `FAX4i`, `CCITT 1 – D`; le prédicteur “différence horizontal” est supporté;
- les images en niveaux de gris, en palette indexées et en RGB;
- la représentation des images multi-canaux en mode “planaire” et “chunky”;
- les images sur 1, 2, 4, 8, 16, 64 bits;
- l’interprétation `Unsigned_int`, `Signed_int` et `IEEE_float` pour les valeurs numériques (à condition que cela conduise à un type numérique connu d’`ExifSe`, par exemple pas de flottant sur 16 bits);
- l’indexation des images (gestion par dalle, par bande ou en un seul bloc);
- la gestion des images multiples en lecture;

Ne sont pas supportés :

- les formats de compression `JPEG` et `FAX3`;
- les espaces colorimétriques `CMYK`, `YCbCr`, `CIELab` et `TranspMask`;
- les résolution variables par canal (utilisées notamment dans les espace de couleur contenant une composant achromatique à plus haute résolution que les composante chromatique);
- la création des fichiers multiples;

#### 11.4.1.2 Enumeration codant les propriétés des images TIFF

Dans la classe `Tiff_Im` sont définies plusieurs énumérations codant certaines propriétés des images TIFF. Ces types énumérés sont utilisés soit pour inspecter le contenu d’un fichier existant soit pour spécifier la création d’un nouveau fichier. Celles à connaître sont :

- `Tiff_Im : :PH_INTER_TYPE` (tag décrit [4, page-37]), décrit l’interprétation photogrammétrique de l’image, cette énumération peut prendre les valeurs :
  - `WhiteIsZero`;
  - `BlackIsZero`;
  - `RGB`;
  - `RGBPalette`;
  - `TranspMask`;
  - `CMYK`;
  - `YCbCr`;
  - `CIELab`.
- `Tiff_Im : :COMPR_TYPE` (tag décrit [4, page-30]), décrit le mode compression, cette énumération peut prendre les valeurs :
  - `No_Compr`;
  - `CCITT_G3_1D_Compr`;
  - `Group_3FAX_Compr`;
  - `Group_4FAX_Compr`;
  - `LZW_Compr`;
  - `JPEG_Compr`;
  - `MPD_T6` (variante personnelle, non documentée, à ne pas utiliser);
  - `PackBits_Compr`.
- `Tiff_Im : :RESOLUTION_UNIT` (tag décrit [4, page-38]), décrit l’unité dans laquelle est représentée la résolution de l’image, cette énumération peut prendre les valeurs :
  - `No_Unit`
  - `Inch_Unit`
  - `Cm_Unit`
- `Tiff_Im : :PLANAR_CONFIG` (tag décrit [4, page-38]), décrit l’organisation physique des images multi-canaux (par plan séparés ou entrelacés); pas hyper passionnant, il faut cependant savoir que le mode planaire est déconseillé par la doc officielle (certains lecteurs ne le reconnaissent pas) mais qu’il

---

<sup>2</sup>sauf erreur ou omission de ma part

peut conduire à des compressions légèrement meilleurs (notamment en LZW + prédicteur); cette énumération peut prendre les valeurs :

- `Chunky_conf` (= entrelacé);
- `Planar_conf`.
- `Tiff_Im` : `:PREDICTOR` (tag décrit [4, page-64]), décrit quel prédicteur est utilisé en amont de la compression;  
cette énumération peut prendre les valeurs :
  - `No_Predic`;
  - `Hor_Diff`.

### 11.4.2 Manipulation de fichiers existants

Pour créer un objet `Tiff_Im` à partir d'un fichier existant, on utilisera simplement le constructeur :

- `Tiff_Im(const char *)`

Pour lire le contenu d'un fichier on utilisera les fonctions membres `Fonc_Num in()` et `Fonc_Num in(REAL)` comme pour tous les objets de la classe `ElGenFileIm`. Pour modifier le contenu d'un fichier on utilisera la fonction membre `Output out()`, comme avec les autres `ElGenFileIm`, en tenant éventuellement comptes des restriction décrite en 11.4.4 pour la fichiers compressés.

Pour inspecter le contenu d'un fichier TIFF on utilisera les fonctions membres suivantes :

- `Tiff_Im` : `:PH_INTER_TYPE phot_interp()` ;
- `Tiff_Im` : `:RESOLUTION_UNIT resunit()` ;
- `Pt2dr resol()` ; renvoie la résolution (dans l'unité du fichier), il s'agit d'un point car la norme TIFF prévoit la possibilité d'une résolution différente en  $x$  et en  $y$ ;
- `Disc_Pal pal()` ; renvoie la palette si l'interprétation photométrique du fichier est `RGBPalette`, génère une erreur sinon;
- `Tiff_Im` : `:COMPR_TYPE mode_compr()` ;
- `Pt2di sz()` ;
- `INT nb_chan()` ;
- `INT bitpp()` ;
- `GenIm` : `:type_el type_el()` ; renvoie un type numérique tenant compte des valeurs `SignedType`, `IntegralType` et `NbBits`.

Certaines de ces fonctions sont redondantes avec les fonction de la classe `ElGenFileIm` (elle sont conservées pour compatibilité). A des fins de diagnostic un fichier tiff, on utilisera les fonction membres suivantes :

- `bool can_elise_use()` renvoie vrai si le fichier ne contient que des options supportées par `ElSe`;
- `const char * why_elise_cant_use()` renvoie (`char *`) `NULL` si le fichier ne contient que des options supportées par `ElSe`; sinon renvoie une chaîne de caractère indiquant quelle est l'option non supportée;
- `void show()`, imprime sur la sortie standard différentes info;

### 11.4.3 Création de fichiers

#### 11.4.3.1 Constructeurs pour fichier en couleur indexées

Pour créer un fichier `Tiff` avec une palette de couleur indexée, on utilisera le constructeur :

```
Tiff_Im
(
    const char      * name,
    Pt2di           sz,
    GenIm::type_el  type,
    COMPR_TYPE      compr,
    Disc_Pal        palette,
    L_Arg_Opt_Tiff  liste_arg_opt = Empty_ARG
);
```

Commentons :

- avec ce constructeur, l'interprétation photogramétrique est implicitement `Tiff_Im` : `:RGBPalette` (la dimension de sortie du fichier sera 1);

- la taille de la palette des couleur est directement conditionnée par `type`, par exemple si `type` vaut `GenIm : :bits4_msbf`, la palette des couleur `palette`, doit être de 16 (car 4 bits), si ce n'est pas le cas `ËÏSe` générera une erreur ;
- voir 11.4.3.3 pour la sinification de l'argument optionnel `liste_arg_opt` ;

### 11.4.3.2 Constructeur pour les autre type

#### 11.4.3.3 Arguments optionnels

Détaillons maintenant l'utilisation de l'argument `liste_arg_opt` décrit dans les sections précédentes.

Le format `Tiff` prévoye un nombre important d'option dont une grande partie rarement utiles sans être pour autant totalement inutiles (e.q. on peut leur donner des valeur par défaut qui satisferont 90% des utilisateurs). Pour donner accès à ces options aux utilisateurs qui en auraient besoins sans pénaliser, via des constructeur à 50 arguments, on utilise le mécanisme de liste d'arguments optionnels utilisé plusieurs fois dans `ËÏSe` (par exemple pour la squelettisation, voir 14.1.1).

La classe `Arg_Tiff` est la classe de base de toutes les classe dérivées définissant des option de création du format `Tiff` (`Arg_Tiff` est définie au niveau global alors que ses dérivées sont définies comme des classe encapsulées de `Arg_Tiff`). Le type `L_Arg_Opt_Tiff`, est simplement un `typedef` pour spécifier la classe `Ellist<Arg_Tiff>`.

Les classes dérivées de `Arg_Tiff` sont :

- `Tiff_Im : :AResol` permet de spécifier la résolution du fichier ; deux constructeurs :
  - `AResol(REAL value, RESOLUTION_UNIT unit)` ; spécifie une résolution de `value` dans l'unité `unit` ; par exemple pour spécifier 300dpi :  
`Tiff_Im : :AResol(300, Tiff_Im : :Inch_Unit)`
  - `AResol(Pt2dr, RESOLUTION_UNIT)` ;, pareil que le précédent mais permt spécifier une résolution différente en *x* et en *y* ;  
 si ce paramètre est omis, "l'unité" de résolution est `No_Unit` et la résolution est de 1 en *x* et en *y* ;
- `Tiff_Im : :APred` spécifie un prédicteur pour la compression ; constructeur :
  - `Tiff_Im : :APred(Tiff_Im : :PREDICTOR)` ;  
 si ce paramètre est omis, le prédicteur utilisé est `Tiff_Im : :No_Predic` sauf si le mode de compression est `LZW` et que le nombre de bits par pixels est  $\geq 8$  auquel cas le prédicteur par défaut est `Tiff_Im : :Hor_Diff` ;
- `Tiff_Im : :ATiles` spécifie que la fichier est dallé (= tile selon la terminologie de [4]) et fixe la taille des dalles ; un constructeur :
  - `Tiff_Im : :ATiles(Pt2di SzTile)` ;  
 on rappelle que selon la norme [4], les taille des dalles en *x* et *y* doivent être multiples de 16 ; si ce paramètre est omis (ainsi que les paramètre `AStrip` et `ANoStrip`) , `ËÏSe` fixe une taille par défaut, variable selon le nombre de bits de l'image, de manière à ce que, approximativement, chaque dalle non décompressée est une taille 65Ko ;
- `Tiff_Im : :AStrip` spécifie que le fichier est indexé par bandes et fixe le nombre de lignes par bandes, un constructeur :
  - `Tiff_Im : :AStrip(INT row_per_strip)` ;  
 cette option est contradictoire avec `Tiff_Im : :ATiles` ;
- `Tiff_Im : :ANoStrip` spécifie que le fichier n'est pas indexé (toutes les données image en un seul bloc), un constructeur :
  - `Tiff_Im : :ANoStrip()` ;  
 cette option est contradictoire avec `Tiff_Im : :ATiles` et `Tiff_Im : :AStrip` ;
- `Tiff_Im : :APlanConf` spécifie l'organisation physique des différents plan image dans le cas d'une image multi-canal (voir [4, page-38]) ; un constructeur :
  - `Tiff_Im : :APlanConf(Tiff_Im : :PLANAR_CONFIG config)` ;  
 valeur par défaut `Tiff_Im : :Chunky_conf` qui correspond à l'option la mieux supportée, fixer la valeur `Tiff_Im : :Planar_conf` uniquement si on de bonnes raisons de penser que cela améliore la compression ;
- `Tiff_Im : :AMinMax` non documentée pour l'instant car non testée, corrspond au "tags" `MaxSampleValue` et `MinSampleValue` décrits en [4, page-35] ;

Notons que certaines options sont contradictoires (par exemples spécifier à la fois un mode de *tiles* et un mode de *strip*). Si on passe simultanément de telles option, `ËÏSe` ne se plaint pas mais son comportement

d'Élise est indéfini (je crois que c'est sans danger, en général l'une écrase l'autre et c'est tout, mais je n'ai pas vérifié et n'ai pas l'intention de gérer ceci à court ou moyen terme).

Donnons un petit exemple :

<pre> // creer une fichier tiff, compresse en pack-bit, // de taille 200x300, avec des // dalles 64x32, ou les elements sont stocke sur un // octet non signe, fixe la resolution a 400 dpi,  Tiff_Im F1 (   "f1.tif",   Pt2di(200,300),   GenIm::u_int1,   Tiff_Im::PackBits_Compr,   Tiff_Im::BlackIsZero,   newl(Tiff_Im::AResol(400,Tiff_Im::Inch_Unit)) +   Tiff_Im::ATiles(Pt2di(64,32)) ); </pre>	<pre> // creer une fichier tiff, compresse en LZW avec // utilisation du predicteur difference horizontale, // de taille 200x300, avec des // bandes de 10 lignes, ou les elements sont stocke sur un // octet non signe, fixe la resolution a 400 dpi,  Tiff_Im F2 (   "f2.tif",   Pt2di(200,300),   GenIm::u_int1,   Tiff_Im::LZW_Compr,   Tiff_Im::BlackIsZero,   newl(Tiff_Im::AResol(400,Tiff_Im::Inch_Unit)) +   Tiff_Im::APred(Tiff_Im::Hor_Diff) +   Tiff_Im::AStrip(10) ); </pre>
--	--

FIG. 11.5 – Exemples de creation de fichier Tiff.

#### 11.4.4 Ecriture et formats compressés



## Chapitre 12

# Opérateur arithmétiques

### 12.1 Opérateur associatif mixte

+ \* Max Min

### 12.2 Opérateur binaire mixte

/ - pow

### 12.3 Opérateur unaire entier

& && | || xor % mod << >>

### 12.4 Opérateur de comparaison

== != < <= > >=

### 12.5 Opérateur unaire mixte

- Abs square

### 12.6 Opérateur unaire entier

~ !

### 12.7 Opérateur mathématique

sqrt cos sin tan atan erfcc log log2 exp

### 12.8 Opérateur de conversion

Rconv Iconv round\_up round\_down round\_ni round\_ni\_inf

### 12.9 Opérateur sur les complexes

mulc divc squarec polar divc

## 12.10 Opérateur liés à la photogramétrie

`Ori3D_Std : :photo_et_z_to_terrain Ori3D_Std : :photo_et_z_to_terrain`

## 12.11 Opérateur liés à la colorimétrie

`its_to_rgb rgb_to_its mpeg_rgb_to_yuv`

## 12.12 Définition d'opérateur par l'utilisateur

## 12.13 Quelques bizareries personnelles

`diag_m2_sym ecart_circ grad_bilin`

## Chapitre 13

# Filtres prédéfinis



# Chapitre 14

## Vectorisation

Ce chapitre regroupe les fonctionnalités proposées par `XiSe` pour passer du mode maillé au mode vecteur.

### 14.1 Squelettisation

L'algorithme de squelettisation utilisé dans `XiSe` est l'algorithme de "squelettisation par veineirization" décrit dans [2]. Le lecteur souhaitant effectuer un paramétrage fin de cet algorithme est invité à se référer à [2]. Décrivons les principales caractéristiques de cet algorithme :

- le résultat n'est pas un sous ensemble de la forme (= image raster valant 0 ou 1) mais un graphe de pixels (= image raster valant entre 0 et 255 avec les convention habituelles de représentation des graphes de pixels sous `XiSe`) ;
- le fait que le résultat du squelette raster soit un graphe, est complètement transparent pour les cas où ce squelette est ensuite utilisé avec le fonction de chaînage proposées par `XiSe` ;
- l'algorithme peut être paramétré par un certain nombre de grandeurs afin de choisir entre l'alternative : "squelette proche de la forme mais bruité" ou "squelette robuste mais caricatural" ;

#### 14.1.1 Squelettisation de petites images

Fichier : "[applis/doc\\_ex/ddrvecto.cpp](#)"

Pour squelettiser une "petite" image on pourra utiliser la fonction `Skeleton` dont la signature est la suivante :

```
Liste_Pts_U_INT2  Skeleton
(
    Im2D_U_INT1    skel,
    Im2D_U_INT1    image,
    L_ArgSkeleton  = ArgSkeleton::L_empty
);
```

Commentons :

- le premier argument est l'image qui contiendra le squelette ;
- le deuxième argument est l'image à squelettiser ; cette image sera modifiée par l'appel à `Skeleton` : elle contiendra en sortie la fonction d'extinction en distance 3 – 2 ;
- ces deux images doivent impérativement avoir la même taille en  $x$  et en  $y$  ;
- le troisième argument est une liste de paramètre permettant de contrôler le type de squelette obtenu ; cette liste est optionnelle et si elle est omise tous les paramètres prendront leur valeur par défaut ; les élément de cette liste doivent tous être d'une des classe dérivées de `ArgSkeleton` (on trouvera la déclaration de ces classes dans "`include/general/morpho.h`").

Nous allons maintenant détailler l'utilisation des paramètres de squelettisation et l'interprétation de la valeur retournée par la fonction `Skeleton` (valeur rarement utilisée). Le fichier "[applis/doc\\_ex/ddrvecto.cpp](#)" illustre l'utilisation de la fonction `Skeleton` et de ses différents paramètres. Sur la figure 14.1, la colonne de gauche montre le corps de la fonction `test_skel` qui va être

appelée plusieurs fois pour illustrer les différentes options ; la colonne de droite montre le début du programme `main`, on notera ici l'utilisation de `W.chc(...)` pour obtenir une fenêtre avec de "gros" pixels (nécessaire pour voir clairement la structure de graphe du squelette créée).

<pre> void test_skel (     Tiff_Im      I0,     Video_Win    W,     L_ArgSkeleton larg ) {     INT tx = I0.sz().x;     INT ty = I0.sz().y;      Im2D_U_INT1 Iskel(tx,ty);     Im2D_U_INT1 ImIn (tx,ty);      // copie dan ImIn et visualise     // l'image en blanc sur fonds jaune      ELISE_COPY     (         I0.all_pts(),         ! I0.in(),         ImIn.out()           (W.odisc() &lt;&lt; (P8COL::yellow*(ImIn.in()==0)))     );      // appelle le squelette      Liste_Pts_U_INT2 l = Skeleton(Iskel,ImIn,large);      // visualise la valeur retournee par Skeleton     // (utile uniquement au 7eme exemple)      ELISE_COPY(l.all_pts(),P8COL::blue,W.odisc());      // visualise Iskel comme un graphe de pixel     // en appelant "out_graph"      ELISE_COPY     (         Iskel.all_pts(),         Iskel.in(),         W.out_graph(Line_St(W.pdisc() (P8COL::black),2))     );      getchar(); } </pre>	<pre> int main(int,char **) {     INT ZOOM = 8;     Tiff_Im FeLiSe("DOC/eLiSe.tif");      // sz of images we will use      Pt2di SZ = FeLiSe.sz();      // palette allocation      Disc_Pal Pdisc = Disc_Pal::P8COL();     Elise_Set_Of_Palette SOP(newl(Pdisc));      // Creation of video windows      Video_Display Ecr((char *) NULL);     Ecr.load(SOP);      // creer une fenetre avec un pxiel de taille 8      Video_Win W (Ecr,SOP,Pt2di(50,50),SZ*ZOOM);     W = W.chc(Pt2dr(-0.5,-0.5),Pt2dr(ZOOM,ZOOM));     W.set_title("eLiSe dans une fenetre ELISE");      test_skel     (         FeLiSe,         W,         L_ArgSkeleton()     );      test_skel     (         FeLiSe,         W,         L_ArgSkeleton()         + SurfSkel(10)         + AngSkel(4.2)     );     ..... } </pre>
--	--

FIG. 14.1 – Code de la fonction `test_skel` illustrant les différents paramètres de `Skeleton`.

Commentons rapidement les différents appels effectués à `test_skel` :

1. la ligne 1 de la figure 14.2 correspond à un appel avec une liste vide, donc tous les paramètres prennent leur valeur par défaut ; notons que la valeur par défaut du seuil surfacique, `SurfSkel`, est 6 et celle du seuil angulaire `AngSkel` est  $\pi$  ;
2. les lignes 2 et 3 de la figure 14.2 fixent la valeur des paramètres angulaires et surfacique pour obtenir

1		<pre> // Skelette avec toutes // les options par défaut // SurfSkel =&gt; 6 // AngSkel  =&gt; 3.14  test_skel (   FeLiSe,   W,   L_ArgSkeleton() ); </pre>
2		<pre> // Seuil surfaciques et angulaire // + eleves : skelette + caricatural  test_skel (   FeLiSe,   W,   L_ArgSkeleton()   + SurfSkel(10)   + AngSkel(4.2) ); </pre>
3		<pre> // Seuil surfaciques et angulaire // + faibles : skelette + detaille  test_skel (   FeLiSe,   W,   L_ArgSkeleton()   + SurfSkel(3)   + AngSkel(2.2) ); </pre>

FIG. 14.2 – Exemple de skelette plus ou moins caricaturaux obtenus en faisant varier les seuils angulaires et surfaciques.

des squelettes plus caricaturaux ou plus détaillés ;

- la ligne 1 de la figure 14.3 illustre l'utilisation de **ProlgtSkel** (valeur par défaut **false**) ; le comportement “naturel” des algorithmes de squeletisation est de “ronger” les extrémités du squelette ; en ajoutant **ProlgtSkel(true)** à la liste de paramètre on demande un squelette dans lequel les extrémités sont prolongées jusqu’au bord de la forme ;
- la ligne 2 de la figure 14.3 illustre l'utilisation **Cx8Skel** (valeur par défaut **true**) ; dans tous les cas le squelette sera un graphe du 8-voisinage, mais si **Cx8Skel** est positionné à **false**, les composantes connexes du squelette seront des composantes 4-connexes de la forme (ici on voit par exemple

1		<pre> // Avec option de prolongement // des extremités  test_skel (   FeLiSe,   W,   L_ArgSkeleton() + ProlgtSkel(true) ); </pre>
2		<pre> // Avec option de 4-connexité  test_skel (   FeLiSe,   W,   L_ArgSkeleton() + Cx8Skel(false) + ProlgtSkel(true) ); </pre>
3		<pre> // Avec option squelette des disques  test_skel (   FeLiSe,   W,   L_ArgSkeleton() + Cx8Skel(false) + ProlgtSkel(true) + SkelOfDisk(true) ); </pre>

FIG. 14.3 – Illustration des options de prolongement des extrémité, 4-connexité et squelette des disques.

que le squelette du “S” est coupé en 2 au niveau d’un coin joignant deux de ses composantes 4 connexes);

- la ligne 3 de la figure 14.3 illustre l’utilisation de `SkelOfDisk` (valeur par défaut `false`); par défaut, avec des formes suffisamment petites ou suffisamment compactes (proches d’un cercle) le squelette peut complètement disparaître<sup>1</sup>, le seuil de disparition dépendant des valeurs de `SurfSkel` et `AngSkel` (car `SurfSkel` quantifie “suffisamment petites” et `AngSkel` quantifie “suffisamment compactes”);

<sup>1</sup>en effet, avec les squelette continus par exemple, le squelette d’un disque est réduit à un point; hors comme ici le squelette est un graphe, pour les formes assez compacte, on obtient un point isolé, donc aucun arc



1		<pre> // Avec option "resultat"  test_skel (   FeLiSe,   W,   L_ArgSkeleton() + SurfSkel(10) + AngSkel(4.2) + ProlgtSkel(true) + ResultSkel(true) ); </pre>
---	--	---

FIG. 14.4 – Illustration de l’option “avec résultat”.

en fixant le paramètre `SkelOfDisk` à `true` on demande de toujours conserver au moins un arc dans chaque composante; notons que, quelque soit la valeur des différents paramètres, les pixels complètement isolés seront toujours des composantes sans squelette;

- la ligne 1 de la figure 14.4 illustre l’utilisation de `ResultSkel` (valeur par défaut `false`); par défaut la fonction `Skeleton` renvoie une liste de points vide; cependant si `ResultSkel` vaut `true` alors `Skeleton` renvoie une liste qui contient les “centres” de toutes les composantes connexes sans squelette (y compris les éventuels pixels complètement isolés);
- notons enfin l’existence d’un dernier argument optionnel `TmpSkel(Im2D.U_INT2 ImTmp)`; si on utilise ce paramètre `ImTmp` doit être une de même taille que les deux premiers arguments passés à `Skeleton`; `ImTmp` sera utilisée par l’algorithme comme zone mémoire temporaire et permettra de diminuer un peu le temps de calcul;

### 14.1.2 Squeletisation de grandes images

Pour squeletiser de grandes images on utilisera la fonction :

```

Fonc_Num skeleton
(
  Fonc_Num f,
  INT max_d = 256,
  L_ArgSkeleton = ArgSkeleton::L_empty
);

```

Il s’agit d’un filtre bufferisé comme ceux vus en section 3.2 ou au chapitre 13; il requiert donc beaucoup moins de mémoire que la version vue en 14.1.1 et peut être utilisé directement sur des fichier de grande taille.

Le paramètre `max_d` doit indiquer un majorant de la valeur maximum atteinte par la fonction d’extinction, en  $d_{3-2}$ , de l’image à squelettiser. Si on a une idée assez précise de cette valeur il vaut mieux l’indiquer car ça accélérera éventuellement un peu le calcul, mais dans le doute il vaut mieux passer un très fort majorant car le résultat sera erroné si les valeurs effectivement atteintes par la fonction d’extinction dépassait `max_d`.

Sinon la liste d’arguments optionnels a la même rôle qu’avec la version vue en 13. La seule différence est qu’il est complètement inutile d’utiliser le paramètre `TmpSkel(Im2D.U_INT2 ImTmp)` (`ELSe` en fournira un de lui même). La figure 14.5 illustre une utilisation de ce squelette version filtre sur les `Fonc_Num` (histoire de changer on squelettise le complémentaire des caractères).

On remarquera sur la figure 14.5 que le squelette 4-connecté semble plutôt plus naturel que le 8-connecté (alors que c’était plutôt le contraire en prenant le squelette des caractères).

1		<pre> ELISE_COPY (     FeLiSe.all_pts(),     FeLiSe.in(),     W.odisc() ); Line_St lst(Pdisc(P8COL::red),2); ELISE_COPY (     FeLiSe.all_pts(),     skeleton     (         FeLiSe.in(0),         30,         newl(SurfSkel(8))         + ProlgtSkel(true)         + Cx8Skel(false)     ),     W.out_graph(lst) ); </pre>
2		<pre> // idem avec // Cx8Skel(true) </pre>

FIG. 14.5 – Squelette version filtre sur fonction numérique.

## 14.2 Chaînage

Les figures 14.6 et 14.7 donnent un exemple de la fonction de chaînage de squelette, `sk_vect`, proposée par `ELISE`<sup>2</sup>. Commentons d’abord le code la figure 14.6 :

- la fonction `sk_vect` a pour signature :  
`Fonc_Num sk_vect(Fonc_Num fonc, Br_Vect_Action * act) ;`
- le premier argument est une `Fonc_Num` qui décrit le graphe de pixel sur lequel il faut effectuer le chaînage ;
- cette `Fonc_Num` peut être de dimension quelconque ; lorsqu’elle est de dimension supérieure à 1, la première coordonnée est considérée comme le graphe de pixels et les autres coordonnées sont considérées comme des attributs des sommets de ce graphe ;
- le deuxième argument est un pointeur sur un objet d’une classe dérivée de la classe `Br_Vect_Action` ; cet objet définit le “call-back” qui sera appelé pour chaque nouvelle chaîne complète ; le code de la figure 14.7 contient la définition de la classe `Test_Chainage` et il en est commenté ci-dessous ;
- on voit que `sk_vect` renvoie une `Fonc_Num`, cette `Fonc_Num` n’a aucun intérêt particulier, elle renvoie la fonction qui vaut toujours 0 ; c’est uniquement par l’intermédiaire de l’action associée à l’objet dérivé de `Br_Vect_Action` que `sk_vect` effectue quelque chose d’intéressant (c’est par essence par économie de développement, pour réutiliser les services génériques offerts par les opérateurs bufferisés, que `sk_vect` renvoie une `Fonc_Num`) ;

<sup>2</sup>En fait la fonction `sk_vect` peut être utilisée pour chaîner n’importe quel graphe de pixel, mais il est certain qu’en pratique, dans 99% des cas elle sera utilisée pour du chaînage de graphes de pixels issus d’une squeletisation

1		<pre> ELISE_COPY (     FeLiSe.all_pts(),     ! FeLiSe.in(),     W.odisc() ); ELISE_COPY (     FeLiSe.all_pts(),     sk_vect     (         skeleton(! FeLiSe.in(1)),         new Test_Chainage         (             W,             Pdisc(P8COL::yellow),             Pdisc(P8COL::blue),             Pdisc(P8COL::green),             Pdisc(P8COL::red)         )     ),     Output::onul() ); </pre>
---	--	---

FIG. 14.6 – Utilisation de la fonction `sk_vect` pour effectuer le chaînage d'un squelette.

<pre> class Test_Chainage : public Br_Vect_Action { private :     virtual void action     (         const ElFifo&lt;Pt2di&gt; &amp; pts,         const ElFifo&lt;INT&gt; *,         INT     )     {         INT nb = pts.nb();          for (INT k = 0; k&lt; nb + pts.circ()-1; k++)             _W.draw_seg(pts[k],pts[k+1],Line_St(_c4,2));          if ( pts.circ())         {             for (INT k = 0; k&lt; nb; k++)                 _W.draw_circle_loc(pts[k],0.5,_c3);         }         else         {             for (INT k = 0; k&lt; nb; k++)                 _W.draw_circle_loc(pts[k],0.5,_c1);             _W.draw_circle_loc(pts[0],0.5,_c2);             _W.draw_circle_loc(pts[nb-1],0.5,_c2);         }     } } </pre>	<pre> Video_Win _W; Col_Pal _c1; Col_Pal _c2; Col_Pal _c3; Col_Pal _c4;  public :     Test_Chainage     (         Video_Win W,         Col_Pal c1,         Col_Pal c2,         Col_Pal c3,         Col_Pal c4     ) : _W (W),_c1 (c1),_c2 (c2),_c3 (c3),_c4 (c4)     {} }; </pre>
---	---

FIG. 14.7 – Classe `Test_Chainage` utilisée pour illustrer la fonction de chaînage de squelette `sk_vect`.

- `sk_vect` appartient à la classe des filtre prédéfinis rectangulaires et doit impérativement être utilisé avec un flux de points de type rectangle  $2 - D$ ;
- on remarque l'utilisation de la fonction membre statique `onul` de la classe `Output`; cette fonction a la signature :  
`Output Output : :onul(INT dim = 1); /* static */`  
elle renvoie un `Output` de dimension consommée *dim*; l'effet de cet `Output` est justement de ne *rien faire* (c'est plus ou moins l'équivalent du `/dev/null` d'UNIX); cette fonction est utilisée dans les circonstance où l'on doit créer un `Output` uniquement pour satisfaire aux règles de typage; cette fonction peut être

aussi utilisée en conjonction avec l'opérateur “,” sur les **Output** pour certaine redirection “fine” de message;

Commentons maintenant la définition de la classe **Test\_Chainage** donnée sur le code la figure 14.7;

- `void action(const ElFifo<Pt2di> &,const ElFifo<INT> *,INT nb_atr)` est une fonction membre virtuelle qui doit être définie dans les classes dérivée de la classe **Br\_Vect\_Action** (qui est une classe abstraite pure);
- la fonction **action** est appelée par **ELISE** à chaque fois qu'une nouvelle chaîne de pixel a été extraite de manière complète;
- le premier argument passé à **action** est une pile de point contenant, dans un ordre cohérent, les points de la chaîne;
- le deuxième argument contient un tableau de pile d'attributs et le troisième le nombre d'attributs (égal à la dimension de la fonction moins un);

La possibilité de récupérer des attributs associés aux points du squelette est d'usage assez rare; les figures 14.8 et 14.9 illustrent, sur un cas assez artificiel, la récupération par la fonction **action** des attributs associés aux point du squelette.

1	<pre> ELISE_COPY (     FeLiSe.all_pts(),     ! FeLiSe.in(),     W.odisc() ); ELISE_COPY (     FeLiSe.all_pts(),     sk_vect     (         (             skeleton(! FeLiSe.in(1)),             FX%2 +2,             (10+FY)/7         ),         new Test_Attr_Chainage (W,W.pdisc())     ),     Output::onul() ); </pre>
---	--

FIG. 14.8 – Utilisation de la fonction **sk\_vect** pour effectuer le chaînage d'un squelette en passant des attributs.

## 14.3 Approximation polygonale

Dans la majorité des cas, le chaînage du squelette est utilisé en amont d'une vectorisation où l'on effectue une approximation polygonale de chaque chaîne.

### 14.3.1 description de l'algorithme “pur”

Décrivons d'abord brièvement l'algorithme d'approximation polygonale utilisé par **ELISE** dans sa version “pure” :

- soit  $l = p_0 \dots p_n$  un suite de points;
- soit  $L = p_{s_0} \dots p_{s_M}$  un sous suite de  $l$  ( $L$  est une approximation “polygonale” de  $l$ );
- soit  $D(p_{s_k}, p_{s_{k+1}})$  un fonction de coût élémentaire mesurant l'écart entre l'approximation  $[p_{s_k} \ p_{s_{k+1}}]$  et la ligne initiale  $[p_{s_k} p_{s_{k+1}+1} p_{s_{k+2}} \dots p_{s_{k+1}}]$  (par exemple sur la figure 14.10, les points en gris correspondent à la ligne initiale du segment  $[P_2 \ P_3]$ )  $D$  peut par exemple être la moyenne des distance entre le segment  $[p_{s_k} \ p_{s_{k+1}}]$  et la ligne initiale;
- soit  $C^{ste}$  une constante indiquant le degré de précision souhaité de l'approximation;  $C^{ste}$  permet de gérer le compromis entre approximation fidèle mais avec beaucoup de points ou approximation caricaturale mais avec peu de points; plus  $C^{ste}$  est grande, moins l'approximation résultante possèdera de points;

<pre> class Test_Attr_Chainage : public Br_Vect_Action { private :     virtual void action     (         const ElFifo&lt;Pt2di&gt; &amp; pts,         const ElFifo&lt;INT&gt; * attr,         INT     )     {         INT nb = pts.nb();         Pt2dr d(0.5,0.5);          for (INT k = 0; k&lt; nb; k++)         {             _W.draw_circle_loc             (                 d+pts[k],                 attr[1][k]/10.0,                 _pal(attr[0][k])             );         }     } } </pre>	<pre>         PS_Window _W;         Disc_Pal _pal;  public :     Test_Attr_Chainage     (         PS_Window W,         Disc_Pal pal     ) : _W (W) , _pal (pal)     {} }; </pre>
---	--

FIG. 14.9 – Définition d’une classe de chaînage utilisant les attributs pour représenter les points du squelette par des cercles de couleur et de rayons variables.

FIG. 14.10 – Notation pour l’algorithme d’approximation polygonale.

- on définit le coût de l’approximation  $L$  par :  

$$C(L) = M * C^{ste} + \sum_{0 \leq m < M} D(p_{s_m}, p_{s_{m+1}})$$
- $\mathcal{E}$  se calcule alors la sous-suite  $L$  qui minimise le coût  $C(L)$ ; cette minimisation est faite par programmation dynamique (ce qui nécessite un temps de calcul en  $Tn^2$  ou  $T$  est le coût moyen de calcul des  $D(p_k, p_{k'})$ );

FIG. 14.11 – Points (en gris) non étudiés avec l’option de suppression des triplets alignés

FIG. 14.12 – Différentes itérations de l'approximation polygonale

FIG. 14.13 – Les deux options actuellement implantées pour le segment permettant de mesurer un écart avec la polyligne initiale

### 14.3.2 Options

Sur cette base algorithmique,  $\mathcal{E}\mathcal{P}\mathcal{S}\mathcal{e}$  offre plusieurs options :

1. la possibilité d'éliminer *a priori* comme points possibles de l'approximation polygonale les points qui forment un triplet alignés avec leur prédécesseur et leur successeur ; par exemple, sur la figure 14.11, les points grisés seront exclus d'emblée des points possibles de l'approximation résultante (mais il seront quand même utilisé pour calculer le l'écart entre une approximation  $[p_{s_k} p_{s_{k+1}}]$  et sa polyligne initiale) ; cette option est a priori sans danger et permet de gagner du temps (typiquement un facteur 2) ;
2. la possibilité de fixer un seuil maximal sur la valeur  $|s_k - s_{k+1}|$ , ceci peut être utile si ont veut par exemple limiter la taille des sauts résultant de l'approximation ; elle sert surtout à accélérer l'exécution de l'algorithme ;
3. la possibilité, en conjonction avec l'option précédente, d'itérer plusieurs fois le processus, à chaque étape seuls sont considérés comme points possibles de l'approximation polygonale les points de

l'étape précédente (mais cependant, on conserve toujours tous les points pour le calcul des écarts avec les lignes initiales); cette option permet d'avoir une approximation franchissant des sauts arbitrairement grands tout en conservant une vitesse de calcul acceptable; la figure 14.12 illustre cette option :

- on suppose que l'on a fixé un seuil de saut maximal à 3 ;
  - après l'étape 1, on obtient la liste  $P_0, P_1 \dots P_7$  entre lesquels les écarts maximum sont de 3 ;
  - à l'étape suivante, on relance l'algorithme en ne recherchant les points de l'approximation que dans la liste  $P_0 \dots P_7$ , on obtient donc une liste  $Q_0 \dots Q_4$  où les écarts théoriques peuvent atteindre 9 ( $= 3^2$ ) ;
  - si on relance encore l'algorithme, la solution trouvée n'évolue plus ; dans ce cas, `Elise` s'arrêtera quelque soit la valeur donnée au nombre d'itération (en pratique on fixera donc souvent un nombre d'itération très grand, genre  $+\infty$ , pour signifier "jusqu'à stabilité") ;
4. la possibilité de choisir entre plusieurs fonction pour mesurer l'écart entre un segment et la ligne initiale (en fait une seule est implantée aujourd'hui mais, structurellement tout est prévu pour gérer cette option) ;
  5. la possibilité de choisir entre plusieurs mode pour définir le segment avec lequel on calcule un écart à la polygône initiale ; le choix le plus évident, sur la figure 14.13, est de définir que l'écart doit être mesuré entre les points en gris et le segment joignant  $P_2$  à  $P_3$  ; il existe cependant d'autre choix raisonnable, `Elise` offre par exemple la possibilité de définir que cet écart doit être calculé entre la droite d'ajustement aux moindres carrés et la polygône initiale ;

### 14.3.3 La class ArgAPP pour spécifier les options

Pour spécifier ces différentes options, on utilisera un objet de la class `ArgAPP`, Le constructeur de la classe `ArgAPP` est :

```
ArgAPP
(
    REAL prec,
    INT nb_jump,
    ModeCout mcout,
    ModeSeg mseg,
    bool   freem_sup = true,
    INT nb_step = 100000
);
```

La signification des différents paramètres est la suivante :

- `prec` indique la précision souhaitée ;
- `nb_jump` correspond à l'option 2 (taille des saut) ;
- `mcout` correspond à l'option 4 (fonction d'écart entre un segment et une polygône), pour l'instant ce paramètre ne peut valoir que `ArgAPP : :D2_droite` (l'écart est calculé en prenant la somme des carrés des distance des points initiaux au segment) ;
- `mseg` correspond à l'option 5 (mode de calcul du segment), ce paramètre peut valoir `ArgAPP : :Extre` (segment joignant les extrémités) ou `ArgAPP : :MeanSquare` (segment d'ajustement aux moindres carrés) ;
- `freem_sup` correspond à l'option 1 (suppression des triplets alignés), s'il est vrai (valeur par défaut), les points qui forment des triplets alignés ne pourront pas faire partie de l'approximation ;
- `nb_step` correspond à l'option 3 (nombre d'itérations), la valeur par défaut correspond à  $\infty$  (itération jusqu'à stabilité).

### 14.3.4 La fonction d'approximation polygonale approx\_poly

Nous pouvons maintenant décrire la fonction d'approximation polygonale `approx_poly` offerte par `Elise`. Sa signature est :

```
void approx_poly
```

```
(
    ElFifo<INT> &          res,
    const ElFifo<Pt2di> & pts,
    ArgAPP              arg
);
```

De manière évidente `arg` spécifie les options de l'approximation polygonale et `pts` la liste de points à approximer. Détaillons rapidement le paramètre `res` :

- `res` contient en sortie la liste des indices de l'approximation polygnale ;
- si la liste est `pts` est non circulaire (au sens où `pts.circ()` renvoie `false`, voir 8.4, page 8.4) `ÉjSe` comprend qu'il s'agit de faire l'approximation polygonale d'une courbe ouverte ; les extrémité de la courbe doivent donc faire partie de l'approximation et on aura `res[0] == 0` ; et `res.top() == pts.nb()-1` ;
- par contre si `pts` est circulaire, alors `ÉjSe` comprend qu'il s'agit de faire l'approximation polygonale d'une courbe fermée où il n'y a pas de notion d'extrémité et pour laquelle il n'y a aucune raison que le premier points fasse partie de l'approximation ; `ÉjSe` cherche alors a déterminer le début de l'approximation selon des critères géométriques (recherche d'un point "anguleux") ; notons que quoiqu'il arrive les indice seront toujours croissants, ce qui signifie que certains indices dépasseront `pts.nb()`, cela dit comme la liste est circulaire il n'y a aucune précaution particulière à prendre pour indexer `pts` ; explicitons sur un petite exemple :
  - supposons que `pts` soit circulaire et contiennent 20 points ;
  - supposons que, selon les critères utilisés par `ÉjSe` le point le plus anguleux soit le 10<sup>ème</sup> ;
  - le résultat de l'approximation polygnale, contiendra par exemple [1015222630] ;
  - `pts[26], pts[27]...pts[30]` est équivalent à `pts[6], pts[7]...pts[10]` ; on voit donc qu'il n'y a pas de précaution particulière à prendre pour indexer `pts` ;

### 14.3.5 exemples

Les figures 14.14 et 14.15 illustrent une utilisation de la fonction d'approximation polygonale offerte par `ÉjSe`. La figure 14.14 fait des appels à la fonction de vectorisation `sk_vect` avec des objets de la classe `Test_Approx`, à chaque appels on intialise un `Test_Approx` avec une valeur différente du paramètre qui règlera la précision. La figure 14.15 définit la classe `Test_Approx` qui fait appel à la fonction `approx_poly`.



1		<pre> // tableau contenant les differents // parametres réglant la precision // de l'approximation polygonale  REAL prec[3] = {1.0,3.0,10.0}; for (INT k = 0; k &lt; 3 ; k++) {     ELISE_COPY     (         FeLiSe.all_pts(),         ! FeLiSe.in(),         W.odisc()     ); } </pre>
2		<pre> ELISE_COPY (     FeLiSe.all_pts(),     sk_vect     (         skeleton(! FeLiSe.in(1)),         new Test_Approx         (             W,             Pdisc(P8COL::blue),             Pdisc(P8COL::red),             prec[k]         )     ),     Output::onul() ); getchar(); } </pre>
3		

FIG. 14.14 – Utilisation de la classe `Test_Approx` (voir ci-dessous) pour effectuer l'approximation polygonale du squelette. On fait le test avec differents paramètres de précision.

```

class Test_Approx : public Br_Vect_Action
{
private :

virtual void action
(
    const ElFifo<Pt2di> & pts,
    const ElFifo<INT> *,
    INT
)
{
    Pt2dr d(0.5,0.5);
    ElFifo<INT> app;
    approx_poly(app,pts,_arg);
    INT nb = app.nb();

    for (INT k = 0; k< nb-1; k++)
        _W.draw_seg(d+pts[app[k]],d+pts[app[k+1]],_c1);
    for (INT k = 0; k< nb; k++)
        _W.draw_circle_loc(d+pts[app[k]],0.5,_c2);
}

PS_Window _W;
Col_Pal _c1;
Col_Pal _c2;
ArgAPP _arg;

```

```

public :

Test_Approx
(
    PS_Window W,
    Col_Pal c1,
    Col_Pal c2,
    REAL prec
) : _W (W) ,
    _c1 (c1) ,
    _c2(c2),
    _arg (
        prec,
        20,
        ArgAPP::D2_droite,
        ArgAPP::Extre
    )
{}
};

```

FIG. 14.15 – Définition d'une classe `Test_Approx` utilisant la fonction d'approximation polygonale `approx_poly`.

Troisième partie

Utilisation “avancée”



Quatrième partie

Appendices



## Annexe A

# Référence bibliographique





# Bibliographie

- [1] Gif89a
- [2] Marc Pierrot Deseilligny, Georges Stamon et C.Y. Suen, “*Veineirization a new shape descriptor for flexible skeletonization*”, IEEE Transaction on Pattern Analysis and Machine Intelligence, 20(5) : 505 – – – 521, Mai 1998.
- [3] Bjarne Stroustrup, *The C++ programming language*, 3<sup>rd</sup> edition
- [4] Aldus Developper Desc “*Tiff 6.0 Revision Final*”, Juin 1992.



## Annexe B

### Listing des programmes d'exemples

## B.1 intro0.cpp

```

#include "general/all.h"

//**** Fonctions Numerique

// Fonc_Num primitives

void TEST_plot_FX (Plot_id plot)
{
    ELISE_COPY(plot.all_pts(),FX, plot.out());
}

void TEST_plot_2 (Plot_id plot)
{
    ELISE_COPY(plot.all_pts(),2,plot.out());
}

Im1D_REAL4 init_image(INT tx)
{
    Im1D_REAL4 I(tx);
    REAL4 * d = I.data();
    for (INT x =0; x < tx ; x++)
        d[x] = (x*x)/ (double) tx;
    return I;
}

void TEST_plot_Im0 (Plot_id plot)
{
    Im1D_REAL4 I = init_image(40);
    ELISE_COPY(I.all_pts(),I.in(),plot.out());
}

// Operator sur Fonc_Num

void TEST_plot_expr_1 (Plot_id plot)
{
    ELISE_COPY (plot.all_pts(),FX/2.0,plot.out());
}

void TEST_plot_expr_2 (Plot_id plot)
{
    ELISE_COPY
    (
        plot.all_pts(),
        4*cos(FX/2.0)+ 3+ (FX/5.0) * sin(FX/4.9),
        plot.out()
    );
}

void TEST_plot_expr_Im0 (Plot_id plot)
{
    Im1D_REAL4 I = init_image(40);
    ELISE_COPY(I.all_pts(),40-1.7*I.in(),plot.out());
}

void TEST_plot_expr_Im1 (Plot_id plot)
{
    Im1D_REAL4 I = init_image(40);
    ELISE_COPY
    (
        plot.all_pts(),
        3.0*I.in()[Abs(FX)%20],
        plot.out()
    );
}

//**** Flux_Pts

//==== Flux_Pts primitifs

void TEST_plot_rects (Plot_id plot)
{
    ELISE_COPY(rectangle(-50,-40),FX,plot.out());
    ELISE_COPY(rectangle(-20,40),FX,plot.out());
}

//==== operateur sur Flux_Pts

void TEST_plot_op_flx0 (Plot_id plot)
{
    ELISE_COPY
    (
        rectangle(-50,-40)
        || rectangle(-20,-5)
        || rectangle(5,13)
        || rectangle(37,42),
        4*cos(FX/2.0)+ 3+ (FX/5.0) * sin(FX/4.9),
        plot.out()
    );
}

void TEST_plot_op_flx1 (Plot_id plot)
{
    ELISE_COPY
    (
        select(plot.all_pts(),(FX%2) || (FX >20)),
        4*cos(FX/2.0)+ 3+ (FX/5.0) * sin(FX/4.9),
        plot.out()
    );
}

//*** Output

//== Output primitifs

void TEST_plot_out_image (Plot_id plot)
{
    Im1D_REAL4 I (50);
    ELISE_COPY(I.all_pts(),(FX%10)*5,I.out());
    ELISE_COPY(I.all_pts(),I.in(),plot.out());
}

//=== Operateur sur Output

void TEST_plot_oper_out_0(Plot_id plot,Line_St lst)
{
    Im1D_REAL4 I (50);
    ELISE_COPY(I.all_pts(),cos(FX/2.0)*30,I.out()|plot.out());

    plot.set(NewlArgP1ld(P1ModeP1(Plots::line)));
    plot.set(NewlArgP1ld(PlotLinSty(lst)));

    ELISE_COPY
    (
        I.all_pts(),
        -I.in(),
        plot.out()
    );
}

void TEST_plot_oper_out_1(Plot_id plot)
{
    Im1D_REAL4 I (50);

    ELISE_COPY
    (
        I.all_pts(),
        Square(FX)/50.0,
        I.out()
    | plot.out()
    | (plot.out().chc(FX-50))
    | (plot.out() << (-I.in()))
    );
}

//*** Output

void TEST_plot_Im0_Bug (Plot_id plot)
{
    Im1D_REAL4 I = init_image(40);
    ELISE_COPY(plot.all_pts(),I.in(),plot.out());
}

void TEST_plot_Im1 (Plot_id plot)
{
    Im1D_REAL4 I = init_image(40);
    ELISE_COPY(plot.all_pts(),I.in(4.5),plot.out());
}

void TEST_plot_Im2 (Plot_id plot)
{
    Im1D_REAL4 I = init_image(40);
    ELISE_COPY(plot.all_pts(),I.in_proj(),plot.out());
}

int main(int,char **)
{

```

```

// sz of images we will use

Pt2di SZ(512,512);

// palette allocation

Disc_Pal Pdisc = Disc_Pal::P8COL();
Elise_Set_Of_Palette SOP(NewLEIPal(Pdisc));

// Creation of video windows

Video_Display Ecr((char *) NULL);
Ecr.load(SOP);
Video_Win Wv (Ecr,SOP,Pt2di(50,50),Pt2di(SZ.x,SZ.y));

// define a window to draw simultaneously in

Plot_id Plot1
(
    Wv,
    Line_St(Pdisc(P8COL::green),3),
    Line_St(Pdisc(P8COL::black),2),
    Interval(-50,50),
    NewlArgPlid(P1Box(Pt2di(3,3),SZ-Pt2di(3,3)))
    + Arg_Opt_PlotId(P1ScaleY(1.0))
    + Arg_Opt_PlotId(P1BoxSty(Pdisc(P8COL::blue),3))
    + Arg_Opt_PlotId(P1ClipY(true))
    + Arg_Opt_PlotId(P1ModeP1(Plots::draw_fill_box))
    + Arg_Opt_PlotId(P1ClearSty(Pdisc(P8COL::white)))
    + Arg_Opt_PlotId(P1PlotFilSty(Pdisc(P8COL::red)))
);

// NOW TRY SOME PLOTS

// 1- VARIATION ON FUNCTIONS

// 1-1 "primitives" functions

Plot1.clear();
TEST_plot_FX(Plot1);
Plot1.show_axes();
Plot1.show_box();
getchar();

Plot1.clear();
TEST_plot_2(Plot1);
Plot1.show_axes();
Plot1.show_box();
getchar();

Plot1.clear();
TEST_plot_Im0(Plot1);
Plot1.show_axes();
Plot1.show_box();
getchar();

// 1-2 Arithmetic operator + composition

Plot1.clear();
TEST_plot_expr_1(Plot1);
Plot1.show_axes();
Plot1.show_box();
getchar();

Plot1.clear();
TEST_plot_expr_2(Plot1);
Plot1.show_axes();
Plot1.show_box();
getchar();

Plot1.clear();
TEST_plot_expr_Im0(Plot1);
Plot1.show_axes();
Plot1.show_box();
getchar();

Plot1.clear();
TEST_plot_expr_Im1(Plot1);
Plot1.show_axes();
Plot1.show_box();
getchar();

// 2- VARIATION ON FLUX

// 2-1 "primitives" flux

Plot1.clear();
TEST_plot_rects(Plot1);
Plot1.show_axes();
Plot1.show_box();
getchar();

// 2-2 "operator on flux

Plot1.clear();
TEST_plot_op_flx0(Plot1);
Plot1.show_axes();
Plot1.show_box();
getchar();

Plot1.clear();
TEST_plot_op_flx1(Plot1);
Plot1.show_axes();
Plot1.show_box();
getchar();

// 3- VARIATION ON FLUX

// 3-1 "primitives" output

Plot1.clear();
TEST_plot_out_image(Plot1);
Plot1.show_axes();
Plot1.show_box();
getchar();

// 3-2 "primitives" output

Plot1.clear();
TEST_plot_oper_out_0
(
    Plot1,
    Line_St(Pdisc(P8COL::magenta),3)
);
Plot1.set(NewlArgPlid(PlotLinSty(Pdisc(P8COL::black),2)));
Plot1.set(NewlArgPlid(P1ModeP1(Plots::draw_fill_box)));
Plot1.show_axes();
Plot1.show_box();
getchar();

Plot1.clear();
TEST_plot_oper_out_1(Plot1);
Plot1.show_axes();
Plot1.show_box();
getchar();

```

```
// 4 ERROR/ handling over
```

```
/*
    Plot1.clear();
    TEST_plot_Im0_Bug(Plot1);
    Plot1.show_axes();
    Plot1.show_box();
    getchar();
*/

Plot1.clear();
TEST_plot_Im1(Plot1);
Plot1.show_axes();
Plot1.show_box();
getchar();

Plot1.clear();
TEST_plot_Im2(Plot1);
Plot1.show_axes();
Plot1.show_box();
getchar();
}
```

## B.2 introd2.cpp

```
#include "general/all.h"
```

```
int main(int,char **)
{
```

```
// sz of images we will use
```

```
Pt2di SZ(256,256);
```

```
// palette allocation
```

```
Disc_Pal Pdisc = Disc_Pal::P8COL();
Gray_Pal Pgr (30);
Circ_Pal Pcirc = Circ_Pal::PCIRC6(30);
RGB_Pal Prgb (5,5,5);
Elise_Set_Of_Palette SOP(NewLEIPal(Pdisc)+Pgr+Prgb+Pcirc);
```

```
// Creation of video windows
```

```
Video_Display Ecr((char *) NULL);
Ecr.load(SOP);
Video_Win W (Ecr,SOP,Pt2di(50,50),Pt2di(SZ.x,SZ.y));
W.set_title("Une fenetre");
```

```
// show_FX
```

```
ELISE_COPY(W.all_pts(),FX,W.ogray());
getchar();
```

```
// 2-3-4
```

```
ELISE_COPY(W.all_pts(),FY,W.ogray());
getchar();

ELISE_COPY(W.all_pts(),(FX+FY)/2,W.ogray());
getchar();

ELISE_COPY
(
    W.all_pts(),
    127.5 * (1.0 + cos(FY*0.2+0.06*FX*sin(0.06*FX))),
    W.ogray()
);
getchar();
```

```
// 5
```

```
Tiff_Im FLena("../DOC_ELISE/mini_lena.tif");
Im2D_U_INT1 I(256,256);
ELISE_COPY
(
    I.all_pts(),
    FLena.in(),
    I.out() | W.ogray()
);
getchar();
```

```
// 6
```

```
ELISE_COPY
(
    I.all_pts(),
    255-I.in(),
    W.ogray()
);
getchar();
```

```
// 7
```

```
ELISE_COPY
(
    I.all_pts(),
    I.in(),
    W.out(Pcirc)
);
getchar();
```

```
ELISE_COPY
(
    I.all_pts(),
    I.in(),
    W.out(Pgr)
);
getchar();
```

```
ELISE_COPY
(
    I.all_pts(),
    I.in()/64,
    W.out(Pdisc)
);
getchar();
```

```
//
```

```
ELISE_COPY
(
    I.all_pts(),
    I.in()<128,
    W.out(Pdisc)
);
getchar();
ELISE_COPY
(
    I.all_pts(),
    I.in()<FX,
    W.out(Pdisc)
);
getchar();
ELISE_COPY
(
    I.all_pts(),
    I.in()<127.5*(1+sin(FX)*sin(FY)),
    W.out(Pdisc)
);
getchar();
```

```
// operateur ,
```

```
ELISE_COPY
(
    I.all_pts(),
    I.in()[FY,FX],
    W.out(Pgr)
);
getchar();
```

```

ELISE_COPY
(
    I.all_pts(),
    I.in()[(FX*3,FY*2)%256],
    W.out(Pgr)
);
getchar();
ELISE_COPY
(
    I.all_pts(),
    I.in(0)
    [(
        128+(FX-128)+(FY-128)/2.0,
        128-(FX-128)/2.0+(FY-128)
    )],
    W.out(Pgr)
);
getchar();

ELISE_COPY
(
    I.all_pts(),
    I.in(0)
    [(
        FX+20*sin(FX/50.0)+4*cos(FY/15.0),
        FY+8*sin(FX/20.0)+7*cos(FY/18.0)
    )],
    W.out(Pgr)
);
getchar();

ELISE_COPY
(
    I.all_pts(),
    (I.in()+I.in()[(FY,FX)])/2,
    W.out(Pgr)
);
getchar();

//

ELISE_COPY
(
    I.all_pts(),
    pow(I.in()/255.0,5.0/3.0)*255.0,
    W.out(Pgr)
);
getchar();

{
    Im1D_U_INT1 lut(256);
    ELISE_COPY
    (
        lut.all_pts(),
        pow(FX/255.0,3.0/5.0)*255.0,
        lut.out()
    );
    ELISE_COPY
    (
        I.all_pts(),
        lut.in()[I.in()],
        W.out(Pgr)
    );
    getchar();
}

{
    Im1D_U_INT1 lut(256,0);
    U_INT1 * d = lut.data();
    d[1] = d[2] = d[5] = 1;
    d[10] = d[13] = d[14] = 2;
    ELISE_COPY(I.all_pts(),I.in(),W.out(Pgr));
    ELISE_COPY
    (
        I.all_pts(),
        lut.in()[I.in()/16],
        W.out(Pdisc)
    );
    getchar();
}

//

ELISE_COPY
(
    I.all_pts(),
    (FX,0,FY),
    W.out(Prgb)
);
getchar();

ELISE_COPY
(
    I.all_pts(),
    (FX,FY,0),
    W.out(Prgb)
);
getchar();

ELISE_COPY
(
    I.all_pts(),
    (I.in(),0,I.in()[(FY,FX)]),
    W.out(Prgb)
);
getchar();

Tiff_Im FLenaCol("../DOC_ELISE/lena_col.tif");
ELISE_COPY
(
    I.all_pts(),
    FLenaCol.in(),
    W.out(Prgb)
);
getchar();

ELISE_COPY
(
    I.all_pts(),
    (
        FLenaCol.in().v1(),
        FLenaCol.in().v0(),
        FLenaCol.in().v2()
    ),
    W.out(Prgb)
);
getchar();

// 8

ELISE_COPY(I.all_pts(),I.in(),W.out(Pgr));
ELISE_COPY
(
    select(I.all_pts(),I.in()>128),
    P8COL::red,
    W.out(Pdisc)
);
getchar();

ELISE_COPY
(
    I.all_pts(),
    P8COL::blue,
    W.out(Pdisc)
);
ELISE_COPY
(
    disc(Pt2di(128,128),100),
    I.in(),
    W.out(Pgr)
);
getchar();

ELISE_COPY
(
    ell_fill(Pt2di(128,128),135,70,1.2),
    255-I.in(0),
    W.out(Pgr)
);
getchar();

ELISE_COPY
(
    sector_ang(Pt2di(128,128),100,1.0,3.0),
    (I.in() < 128),
    W.out(Pdisc)
);
getchar();

for (INT x = 0; x < 256; x+= 4)
    ELISE_COPY
    (
        line(Pt2di(x,0),Pt2di(128,128)),
        I.in(),
        W.out(Pgr)
    );
    getchar();

ELISE_COPY
(
    polygone
    (
        NewLPt2di(Pt2di(5,250))+Pt2di(128,5)
        + Pt2di(250,250)+Pt2di(128,128)
    ),
    (I.in()/64)*64,
    W.out(Pgr)
);
getchar();

for (INT x = 1; x< 5; x++)
    ELISE_COPY
    (
        border_rect(Pt2di(10,10),Pt2di(15+10*x,15+10*x),5-x),

```

```

        P8COL::white,
        W.out(Pdisc)
    );

ELISE_COPY
(
    W.border(8),
    P8COL::green,
    W.out(Pdisc)
);

ELISE_COPY
(
    ellipse(Pt2di(128,128),135,70,1.2),
    P8COL::red,
    W.out(Pdisc)
);
getchar();

ELISE_COPY(I.all_pts(),255,W.out(Pgr));
ELISE_COPY
(
    disc(Pt2di(64,64),64).chc(2*(FX,FY)),
    0,
    W.out(Pgr)
);
getchar();

ELISE_COPY(I.all_pts(),I.in(),W.out(Pgr));
ELISE_COPY
(
    rectangle(Pt2di(0,0),Pt2di(256,128)).chc((FX,FY*2)),
    I.in()[FX,255-FY],
    W.out(Pgr)
);
getchar();

ELISE_COPY(I.all_pts(),I.in(),W.out(Pgr));
ELISE_COPY
(
    rectangle(Pt2di(0,0),Pt2di(256,128)).chc((FX,2*FY-FY*2)),
    FY,
    W.out(Pgr)
);
getchar();

// On declare trois fenetre Wr, Wg et Wb

Video_Win  Wr = W;
Video_Win  Wg (Ecr,SUP,Pt2di(50,50),SZ);
Video_Win  Wb (Ecr,SUP,Pt2di(50,50),SZ);

Wr.set_title("red chanel");
Wg.set_title("green chanel");
Wb.set_title("blue chanel");

// affichage des 3 canaux en niveau de gris, version "bovine"

ELISE_COPY
(
    W.all_pts(),
    0,
    (Wr.out(Pgr) << FLenaCol.in().v0())
    | (Wg.out(Pgr) << FLenaCol.in().v1())
    | (Wb.out(Pgr) << FLenaCol.in().v2())
);
getchar();

Wr.clear(); Wg.clear(); Wb.clear();

// affichage des 3 canaux en niveau de gris,
// version operateur ", " sur les Output

ELISE_COPY
(
    W.all_pts(),
    FLenaCol.in(),
    (Wr.out(Pgr),Wg.out(Pgr),Wb.out(Pgr))
);
getchar();

Wr.clear(); Wg.clear(); Wb.clear();

Im2D_U_INT1 R(256,256);
Im2D_U_INT1 G(256,256);
Im2D_U_INT1 B(256,256);

// affichage des trois canaux dans Wr, Wg et Wb
// et memo en meme temps dans R,G et B

ELISE_COPY
(
    W.all_pts(),
    FLenaCol.in(),
    (Wr.out(Pgr),Wg.out(Pgr),Wb.out(Pgr))
    | (R.out(),G.out(),B.out())
);

// verif que R,G,B contiennent les bonnes valeurs

ELISE_COPY
(
    W.all_pts(),
    (R.in(),G.in(),B.in()),
    W.out(Prgb)
);
getchar();

// une autre facon de faire
// affichage des trois canaux dans Wr, Wg et Wb
// et memo en meme temps dans R,G et B

ELISE_COPY
(
    W.all_pts(),
    FLenaCol.in(),
    (
        Wr.out(Pgr)|R.out(),
        Wg.out(Pgr)|G.out(),
        Wb.out(Pgr)|B.out()
    )
);
ELISE_COPY
(
    W.all_pts(),
    (R.in(),G.in(),B.in()),
    W.out(Prgb)
);
getchar();
}

template <class T> void f (T * d, INT nb)
{
    for (INT i=0;i<nb;i++) cout<<d[i]<<"\n";
}

Fonc_Num moy(Fonc_Num f,INT nb)
{
    Fonc_Num res = 0;
    for (INT x = -nb; x <= nb; x++)
        for (INT y = -nb; y <= nb; y++)
            res = res + trans(f,Pt2di(x,y));
    return res/ElSquare(2*nb+1);
}

Fonc_Num sobel_0(Fonc_Num f)
{
    Im2D_REAL8 Fx
    (
        3,3,
        " -1 0 1 "
        " -2 0 2 "
        " -1 0 1 "
    );
    Im2D_REAL8 Fy
    (
        3,3,
        " -1 -2 -1 "
        " 0 0 0 "
        " 1 2 1 "
    );
    return
        Abs(som_masq(f,Fx,Pt2di(-1,-1)))
        + Abs(som_masq(f,Fy));
}

template <class Type,class TyBase> class Filters
{
public :

```

## B.3 introfiltr.cpp



```

static inline TyBase sobel (Type ** im,INT x,INT y)
{
    return
    ElAbs
    (
        im[y-1][x-1]+2*im[y][x-1]+im[y+1][x-1]
        - im[y-1][x+1]-2*im[y][x+1]-im[y+1][x+1]
    )
    + ElAbs
    (
        im[y-1][x-1]+2*im[y-1][x]+im[y-1][x+1]
        - im[y+1][x-1]-2*im[y+1][x]-im[y+1][x+1]
    );
}
};

template <class Type,class TyBase>
void std_sobel
(
    Im2D<Type,TyBase> Iout,
    Im2D<Type,TyBase> Iin,
    Pt2di p0,
    Pt2di p1
)
{
    Type ** out = Iout.data();
    Type ** in = Iin.data();
    INT x1 = ElMin3(Iout.tx()-1,Iin.tx()-1,p1.x);
    INT y1 = ElMin3(Iout.ty()-1,Iin.ty()-1,p1.y);
    INT x0 = ElMax(1,p0.x);
    INT y0 = ElMax(1,p0.y);

    // pour eviter les overflow

    TyBase vmax = Iout.vmax()-1;
    for (INT x=x0; x<x1 ; x++)
        for (INT y=y0; y<y1 ; y++)
        {
            out[y][x] = ElMin(vmax,Filters<Type,TyBase>::sobel(in,x,y));
        }
}

template <class Type> void
sobel_buf
(
    Type ** out,
    Type *** in,
    const Simple_OpBuf_Gen & arg
)
{
    for (int d =0; d<arg.dim_out(); d++)
        for (int x = arg.x0(); x<arg.x1() ; x++)
            out[d][x] = Filters<Type,Type>::sobel(in[d],x,0);
}

Fonc_Num sobel(Fonc_Num f)
{
    return create_op_buf_simple_tpl
    (
        sobel_buf,
        sobel_buf,
        f,
        f.dimf_out(),
        Box2di(Pt2di(-1,-1),Pt2di(1,1))
    );
}

int main(int,char **)
{
    // sz of images we will use

    Pt2di SZ(256,256);

    // palette allocation

    Disc_Pal Pdisc = Disc_Pal::P8COL();
    Gray_Pal Pgr (30);
    Circ_Pal Pcirc = Circ_Pal::PCIRC6(30);
    RGB_Pal Prgb (5,5,5);
    Elise_Set_Of_Palette SOP(NewLElPal(Pdisc)+Pgr+Prgb+Pcirc);

    // Creation of video windows

    Video_Display Ecr((char *) NULL);
    Ecr.load(SOP);

    Video_Win W2 (Ecr,SOP,Pt2di(50,50),Pt2di(SZ.x,SZ.y));
    Video_Win W (Ecr,SOP,Pt2di(50,50),Pt2di(SZ.x,SZ.y));

    Plot_id Plot1
    (
        W,
        Line_St(Pdisc(P8COL::green),3),
        Line_St(Pdisc(P8COL::black),2),
        Interval(0,256),
        NewlArgPl1d(P1Box(Pt2di(3,3),SZ-Pt2di(3,3)))
        + P1AutoScalOriY(true)
        + P1BoxSty(Pdisc(P8COL::blue),3)
        + P1ModePl(Plots::line)
        + PlotLinSty(Pdisc(P8COL::red),2)
    );

    Tiff_Im FLena("../DOC_ELISE/mini_lena.tif");
    Tiff_Im FLenaCol("../DOC_ELISE/lena_col.tif");
    Im2D_U_INT1 I(256,256);
    ELISE_COPY
    (
        W.all_pts(),
        FLena.in(),
        I.out() | W.out(Pgr)
    );
    getchar();

    ELISE_COPY
    (
        W.all_pts(),
        Min(255,3*Abs(I.in(0)- I.in(0)[(FX+1,FY)])),
        W.out(Pgr)
    );
    getchar();

    ELISE_COPY
    (
        W.all_pts(),
        Min(255,3*Abs(I.in(0)-trans(I.in(0),Pt2di(0,1)))),
        W.out(Pgr)
    );
    getchar();

    ELISE_COPY(W.all_pts(),P8COL::red,W.out(Pdisc));
    ELISE_COPY
    (
        rectangle(Pt2di(0,0),Pt2di(128,128)),
        trans(FLena.in(),Pt2di(64,64)),
        W.out(Pgr)
    );
    getchar();

    // voir la deinition de moy en tete
    // de fichier

    ELISE_COPY
    (
        W.all_pts(),
        moy(I.in(0),3),
        W.out(Pgr)
    );
    getchar();

    ELISE_COPY
    (
        W.all_pts(),
        rect_max(I.in(0),12),
        W.out(Pgr)
    );
    getchar();

    ELISE_COPY
    (
        W.all_pts(),
        rect_min
        (
            rect_max(I.in(0),Pt2di(7,7)),
            Box2di(Pt2di(-7,-7),Pt2di(7,7))
        ),
        W.out(Pgr)
    );
    getchar();

    {
        REAL fact = 0.9;
        ELISE_COPY
        (
            W.all_pts(),
            canny_exp_filt(I.in(0),fact,fact)
            / canny_exp_filt(I.inside(0),fact,fact),
            W.out(Pgr)
        );
    }
    getchar();

    {

```

```

REAL fact = 0.9;
ELISE_COPY
(
    W.all_pts(),
    canny_exp_filt(FLenaCol.in(0),fact,fact)
    / canny_exp_filt(I.in(0),fact,fact),
    W.out(Prgb)
);
}
getchar();

ELISE_COPY
(
    W.all_pts(),
    Min
    (
        polar(deriche(I.in(0),1.0),0)
        * Fonc_Num(1.0, 256/(2*PI)),
        255
    ),
    (W.out(Pgr), W2.out(Pcirc))
);
getchar();

Im2D_U_INT1 Ibin(256,256);
{
    REAL fact = 0.8;
    ELISE_COPY
    (
        W.all_pts(),
        canny_exp_filt(I.in(0),fact,fact)
        / canny_exp_filt(I.in(0),fact,fact) < 128,
        W.out(Pdisc)
        | Ibin.out()
    );
    getchar();
}

ELISE_COPY
(
    W.all_pts(),
    Ibin.in(0),
    W.out(Pdisc)
);
ELISE_COPY
(
    select(W.all_pts(),open_5711(Ibin.in(0),40)),
    P8COL::cyan,
    W.out(Pdisc)
);
ELISE_COPY
(
    select(W.all_pts(),erod_5711(Ibin.in(0),40)),
    P8COL::blue,
    W.out(Pdisc)
);
getchar();

ELISE_COPY
(
    W.all_pts(),
    dilat_5711(Ibin.in(0),40)
    * P8COL::yellow,
    W.out(Pdisc)
);
ELISE_COPY
(
    select(W.all_pts(),close_5711(Ibin.in(0),40)),
    P8COL::green,
    W.out(Pdisc)
);
ELISE_COPY
(
    select(W.all_pts(),Ibin.in(0)),
    P8COL::black,
    W.out(Pdisc)
);
getchar();

{
    W.clear();
    Im2D_U_INT1 Is(256,256,0);
    std_sobel(Is,I,Pt2di(0,0),Pt2di(256,256));
    ELISE_COPY(Is.all_pts(),Is.in(),W.out(Pgr));
    getchar();
}

ELISE_COPY
(
    W.all_pts(),
    Min(255,sobel_0(I.in(0))),
    W.out(Pgr)
);
getchar();

W.clear();
ELISE_COPY
(
    W.all_pts(),
    Min(255,sobel(I.in(0))),
    W.out(Pgr)
);
getchar();
ELISE_COPY
(
    W.all_pts(),
    Min(255,sobel(FLenaCol.in(0))),
    W.out(Prgb)
);
};

```

```

getchar();

{
    REAL f = 0.9;
    Fonc_Num Fonc =
        canny_exp_filt(I.in(0),f,f)
        / canny_exp_filt(I.in(0),f,f);
    ELISE_COPY
    (
        W.all_pts(),
        Min(255,sobel(Fonc)*8),
        W.out(Pgr)
    );
}
getchar();

Im2D_U_INT1 Idist(256,256);
INT vmax;
ELISE_COPY
(
    W.all_pts(),
    extinc_32(Ibin.in(0)),
    VMax(vmax) | Idist.out()
);
ELISE_COPY(Idist.all_pts(),Idist.in()*(255.0/vmax),W.out(Pgr));
ELISE_COPY
(
    select(W.all_pts(), ! Ibin.in()),
    P8COL::white,
    W.out(Pdisc)
);
getchar();

Im1D_INT4 H(256,0);
ELISE_COPY(W.all_pts(),I.in(),W.out(Pgr));
ELISE_COPY
(
    W.all_pts().chc(I.in()),
    1,
    H.histo()
);
ELISE_COPY(Plot1.all_pts(),H.in(),Plot1.out());
getchar();

Im2D_INT4 Cooc(256,256,0);
INT cmax;
ELISE_COPY
(
    W.interior(1).chc
    ((
        I.in(),
        trans(I.in(),Pt2di(1,0))
    )),
    1,
    Cooc.histo()
    | (VMax(cmax) << Cooc.in())
);
ELISE_COPY
(
    Cooc.all_pts(),
    255 -log(Cooc.in()+1)
    * (255.0/log(cmax+1)),
    W.out(Pgr)
);
getchar();
}

```

## B.4 introanalyse.cpp

```

#include "general/all.h"

int main(int,char **)
{

    // sz of images we will use

    Pt2di SZ(256,256);

    // palette allocation

    Disc_Pal Pdisc = Disc_Pal::P8COL();
    Gray_Pal Pgr (30);
    Circ_Pal Pcirc = Circ_Pal::PCIRC6(30);
    RGB_Pal Prgb (5,5,5);
    Elise_Set_Of_Palette SOP(NewLEIPal(Pdisc)+Pgr+Prgb+Pcirc);
}

```

```

// Creation of video windows

Video_Display Ecr((char *) NULL);
Ecr.load(SOP);

Video_Win W (Ecr,SOP,Pt2di(50,50),Pt2di(SZ.x,SZ.y));

Tiff_Im FLena("../DOC_ELISE/mini_lena.tif");
Tiff_Im FLenaCol("../DOC_ELISE/lena_col.tif");
Im2D_U_INT1 I(256,256);

ELISE_COPY
(
    W.all_pts(),
    FLena.in(),
    I.out() | W.out(Pgr)
);
getchar();

//=====
//
// LISTE DE POINTS
//=====

{
    Im2D_U_INT1 Im(256,256);
    ELISE_COPY
    (
        W.all_pts(),
        rect_median(I.in_proj(),9,256),
        Im.out() | W.out(Pgr)
    );
    getchar();

    Liste_Pts_INT2 l2(2);
    ELISE_COPY
    (
        select(W.all_pts(),Im.in() < 80),
        P8COL::red,
        W.out(Pdisc) | l2
    );
    getchar();
    ELISE_COPY
    (
        l2.all_pts(),
        P8COL::blue,
        W.out(Pdisc)
    );
    getchar();

    ELISE_COPY
    (
        select(W.all_pts(),Im.in() > 160),
        P8COL::green,
        W.out(Pdisc) | l2
    );
    getchar();
    ELISE_COPY
    (
        l2.all_pts(),
        P8COL::yellow,
        W.out(Pdisc)
    );
    getchar();
    ELISE_COPY
    (
        l2.all_pts().chc((FY,FX)),
        P8COL::cyan,
        W.out(Pdisc)
    );
    getchar();

    ELISE_COPY(W.all_pts(),Im.in(),W.out(Pgr));
    Liste_Pts_INT2 l3(3);
    ELISE_COPY
    (
        select
        (
            W.all_pts(),
            Im.in()>128
        ).chc((FX,FY,I.in())),
        l3
    );
    ELISE_COPY(l3.all_pts(),FZ,W.out(Pgr).chc((FX,FY)));
    getchar();

    Im2D_INT2 I13 = l3.image();
    INT2 ** d = I13.data();
    INT nb = I13.tx();
    INT2 * tx = d[0];
    INT2 * ty = d[1];
    INT2 * gray = d[2];
    U_INT1 ** im = Im.data();
    for (INT k=0 ; k<nb ; k++)
        im[ty[k]][tx[k]] = 255-gray[k];
    ELISE_COPY(Im.all_pts(),Im.in(),W.out(Pgr));
    getchar();
}

//=====
//
// RELATION DE VOSINAGES ET DILATATION
//=====

{
    Im2D_U_INT1 Ibin(256,256);

    ELISE_COPY
    (
        W.all_pts(),
        I.in() < 128,
        W.out(Pdisc) | Ibin.out()
    );
    ELISE_COPY(W.border(1),P8COL::red, W.out(Pdisc) | Ibin.out());

    Pt2di Tv4[4] = {Pt2di(1,0),Pt2di(0,1),Pt2di(-1,0),Pt2di(0,-1)};
    Neighbourhood V4 (Tv4,4);
    Neighbourhood V8 = Neighbourhood::v8();

    ELISE_COPY
    (
        dilate
        (
            select(Ibin.all_pts(),Ibin.in() == 1),
            V4
        ),
        P8COL::cyan,
        W.out(Pdisc)
    );
    getchar();

    ELISE_COPY
    (
        select(Ibin.all_pts(),Ibin.in() == 1),
        P8COL::black,
        W.out(Pdisc)
    );
    getchar();

    INT nb_pts;
    ELISE_COPY(W.all_pts(),Ibin.in(),W.out(Pdisc));
    ELISE_COPY
    (
        dilate
        (
            select(Ibin.all_pts(),Ibin.in() == 1),
            sel_func(V8,Ibin.in() == 0)
        ),
        P8COL::red,
        W.out(Pdisc) | (sigma(nb_pts)<< 1)
    );
    cout << "found " << nb_pts << "\n";
    getchar();

    ELISE_COPY(W.all_pts(),Ibin.in(),W.out(Pdisc));
    Liste_Pts_INT2 l2(2);
    ELISE_COPY
    (
        dilate
        (
            select(Ibin.all_pts(),Ibin.in() == 1),
            sel_func(V8,Ibin.in() == 0)
        ),
        P8COL::yellow,
        W.out(Pdisc) | Ibin.out() | l2
    );
    cout << "found " << l2.card() << "\n";
    getchar();

    for (int k = 0; k < 5 ; k++)
    {
        Liste_Pts_INT2 aNewL(2);
        ELISE_COPY
        (
            dilate
            (
                l2.all_pts(),
                Ibin.neigh_test_and_set
                (
                    V8,
                    P8COL::white,
                    P8COL::yellow,
                    20
                )
            ),
            10000,
            aNewL
        );
        l2 = aNewL ;
    }
    ELISE_COPY(Ibin.all_pts(),Ibin.in(),W.out(Pdisc));
    getchar();
}

//=====
//
// COMPOSANTES CONNEXES
//=====

```

```

{
    Im2D_U_INT1 Ibin(256,256);
    ELISE_COPY
    (
        W.all_pts(),
        rect_median(I.in_proj(),2,256)/8 < 8,
        W.out(Pdisc) | Ibin.out()
    );
    ELISE_COPY(W.border(1),0,Ibin.out()|W.out(Pdisc));

    Pt2di pt;
    Col_Pal red =Pdisc(P8COL::red);
    for (;;)
    {
        cout << "CLIKER SUR UN POINT NOIR \n";
        pt = Ecr.clik()._pt;
        if (Ibin.data()[pt.y][pt.x] == 1)
        {
            W.draw_circle_loc(pt,3,red);
            break;
        }
    }
    getchar();

    Neighbourhood V8 = Neighbourhood::v8();
    ELISE_COPY
    (
        conc
        (
            pt,
            Ibin.neigh_test_and_set
            (
                V8,
                P8COL::black,
                P8COL::magenta,
                20
            )
        ),
        P8COL::magenta,
        W.out(Pdisc)
    );
    getchar();

    Pt2di p1(1,254),p2(254,1);
    ELISE_COPY
    (
        conc
        (
            line(p1,p2),
            Ibin.neigh_test_and_set
            (
                V8,
                P8COL::black,
                P8COL::green,
                20
            )
        ),
        P8COL::green,
        W.out(Pdisc)
    );
    ELISE_COPY(line(p1,p2),P8COL::red,W.out(Pdisc));
    getchar();

    Im2D_U_INT1 I2(256,256);
    ELISE_COPY
    (
        Ibin.all_pts(),
        Ibin.in()!=0,
        Ibin.out()|I2.out()|W.out(Pdisc)
    );
    ELISE_COPY
    (
        select
        (
            Ibin.all_pts(),
            Ibin.in()[(FY,FX)]&& (FX<FY) && (!Ibin.in())
        ),
        P8COL::red,
        I2.out()|W.out(Pdisc)
    );
    getchar();
    ELISE_COPY
    (
        conc
        (
            select(I2.all_pts(),I2.in()==P8COL::red),
            I2.neigh_test_and_set
            (
                V8,
                P8COL::black,
                P8COL::blue,
                20
            )
        ),
        P8COL::blue,
        W.out(Pdisc)
    );
    getchar();

    ELISE_COPY(Ibin.all_pts(),Ibin.in()!=0,Ibin.out()|W.out(Pdisc));
    U_INT1 ** d = Ibin.data();

    // et les met en vert, si elles ont + de 200 point:
    // * les mets en cyan
    // * affiche leur boite englobant et centre de gravite

    for (INT x=0; x < 256; x++)
        for (INT y=0; y < 256; y++)
            if (d[y][x] == 1)
            {
                Liste_Pts_INT2 cc(2);
                ELISE_COPY
                (
                    conc
                    (
                        Pt2di(x,y),
                        Ibin.neigh_test_and_set
                        (
                            V8,
                            P8COL::black,
                            P8COL::green,
                            20
                        )
                    ),
                    P8COL::green,
                    W.out(Pdisc) | cc
                );
                if (cc.card() > 200)
                {
                    Line_St lstbox(Prgb(0,0,255),2);
                    Line_St lstcdg(Prgb(255,128,0),3);
                    Pt2di pmax,pmin,cdg;
                    ELISE_COPY
                    (
                        cc.all_pts(),
                        (FX,FY),
                        (pmax.VMax())
                        | (pmin.VMin())
                        | (cdg.sigma())
                        | (W.out(Pdisc) << P8COL::cyan)
                    );
                    W.draw_circle_loc(cdg/cc.card(),5,lstcdg);
                    W.draw_rect(pmin,pmax,lstbox);
                }
            }
    }
    getchar();

    {
        Im2D_Bits<2> Ibin(256,256);
        ELISE_COPY
        (
            W.all_pts(),
            rect_median(I.in_proj(),2,256)/8 < 8,
            W.out(Pdisc) | Ibin.out()
        );
        Col_Pal red =Pdisc(P8COL::red);
        getchar();

        for (INT x = 0 ; x< 256; x++)
            for (INT y = 0 ; y< 256; y++)
            {
                INT v = Ibin.get(x,y);
                Ibin.set(x,y,v+2);
            }

        ELISE_COPY(W.all_pts(),Ibin.in(),W.out(Pdisc));
        getchar();

        for (INT y = 0 ; y< 256; y++)
        {
            U_INT1 * d = Ibin.data()[y];
            for (INT x = 0 ; x< 64; x++)
                d[x] = ~d[x];
        }
        ELISE_COPY(W.all_pts(),Ibin.in(),W.out(Pdisc));
        getchar();

        ELISE_COPY(W.border(1),1,Ibin.out()|W.out(Pdisc));
        Neighbourhood V8 = Neighbourhood::v8();
        ELISE_COPY
        (
            select
            (
                select(W.all_pts(), Ibin.in()==0),
                Neigh_Rel(V8).red_max(Ibin.in())
            ),
            P8COL::red,
            W.out(Pdisc)
        );
        getchar();

        Neighbourhood V4 = Neighbourhood::v4();
        ELISE_COPY
        (
            select(W.all_pts(), Ibin.in()==0),
            2+Neigh_Rel(V4).red_sum(Ibin.in()),
            W.out(Pdisc)
        );
        getchar();
    }
}

```

```

// Parcourt toute les composant connexes de l'image

```

## B.5 ddrvecto.cpp

```

#include "general/all.h"
#include "ext_stl/fifo.h"

void test_skel
(
    Tiff_Im      IO,
    Video_Win    W,
    L_ArgSkeleton larg
)
{
    INT tx = IO.sz().x;
    INT ty = IO.sz().y;

    Im2D_U_INT1 Iskel(tx,ty);
    Im2D_U_INT1 ImIn (tx,ty);

    ELISE_COPY
    (
        IO.all_pts(),
        ! IO.in(),
        ImIn.out()
        | (W.odisc() << (P8COL::yellow * (ImIn.in()==0)))
    );

    Liste_Pts_U_INT2 l = Skeleton(Iskel,ImIn,large);

    ELISE_COPY(l.all_pts(),P8COL::blue,W.odisc());

    ELISE_COPY
    (
        Iskel.all_pts(),
        Iskel.in(),
        W.out_graph(Line_St(W.pdisc() (P8COL::black),2))
    );

    getchar();
}

class Test_Chainage : public Br_Vect_Action
{
private :

    virtual void action
    (
        const ElFifo<Pt2di> & pts,
        const ElFifo<INT> * attr,
        INT
    )
    {
        INT nb = pts.nb();

        for (INT k = 0; k< nb + pts.circ()-1; k++)
            _W.draw_seg(pts[k],pts[k+1],Line_St(_c4,2));

        if ( pts.circ() )
        {
            for (INT k = 0; k< nb; k++)
                _W.draw_circle_loc(pts[k],0.5,_c3);
        }
        else
        {
            for (INT k = 0; k< nb; k++)
                _W.draw_circle_loc(pts[k],0.5,_c1);
            _W.draw_circle_loc(pts[0] ,0.5,_c2);
            _W.draw_circle_loc(pts[nb-1] ,0.5,_c2);
        }
    }

    Video_Win _W;
    Col_Pal _c1;
    Col_Pal _c2;
    Col_Pal _c3;
    Col_Pal _c4;

public :
    Test_Chainage
    (
        Video_Win W,
        Col_Pal c1,
        Col_Pal c2,
        Col_Pal c3,
        Col_Pal c4
    ) : _W (W) , _c1 (c1) , _c2(c2) , _c3(c3),_c4(c4)
    {}
};

class Test_Attr_Chainage : public Br_Vect_Action
{
private :

    virtual void action
    (
        const ElFifo<Pt2di> & pts,
        const ElFifo<INT> * attr,
        INT
    )
    {
        INT nb = pts.nb();

        for (INT k = 0; k< nb; k++)
        {
            _W.draw_circle_loc
            (
                pts[k],
                attr[1][k]/10.0,
                _pal(attr[0][k])
            );
        }

        Video_Win _W;
        Disc_Pal _pal;

    public :
        Test_Attr_Chainage
        (
            Video_Win W,
            Disc_Pal pal
        ) : _W (W) , _pal (pal)
        {}
    };

class Test_Approx : public Br_Vect_Action
{
private :

    virtual void action
    (
        const ElFifo<Pt2di> & pts,
        const ElFifo<INT> * attr,
        INT
    )
    {
        ElFifo<INT> app;
        approx_poly(app,pts,_arg);
        INT nb = app.nb();

        for (INT k = 0; k< nb-1; k++)
            _W.draw_seg(pts[app[k]],pts[app[k+1]],_c1);
        for (INT k = 0; k< nb; k++)
            _W.draw_circle_loc(pts[app[k]],0.5,_c2);
    }

    Video_Win _W;
    Col_Pal _c1;
    Col_Pal _c2;
    ArgAPP _arg;

    public :
        Test_Approx
        (
            Video_Win W,
            Col_Pal c1,
            Col_Pal c2,
            REAL prec
        ) : _W (W) ,
            _c1 (c1) ,
            _c2(c2),
            _arg (
                prec,
                20,
                ArgAPP::D2_droite,
                ArgAPP::Extre
            )
        {}
    };

int main(int,char **)
{
    INT ZOUM = 8;
    Tiff_Im FeLiSe("../DOC_ELISE/eLiSe.tif");

    // sz of images we will use

    Pt2di SZ = FeLiSe.sz();

    // palette allocation

```

```

Disc_Pal Pdsc = Disc_Pal::P8COL();
Elise_Set_Of_Palette SOP(NewLEIPal(Pdsc));

// Creation of video windows

Video_Display Ecr((char *) NULL);
Ecr.load(SOP);
Video_Win W (Ecr,SOP,Pt2di(50,50),SZ*ZOOM);
W = W.chc(Pt2dr(-0.5,-0.5),Pt2dr(ZOOM,ZOOM));
W.set_title("FeLiSe dans une fenetre ELISE");

#if (0)

test_skel
(
    FeLiSe,
    W,
    L_ArgSkeleton()
);

test_skel
(
    FeLiSe,
    W,
    L_ArgSkeleton()
    + SurfSkel(10)
    + AngSkel(4.2)
);

test_skel
(
    FeLiSe,
    W,
    L_ArgSkeleton()
    + SurfSkel(3)
    + AngSkel(2.2)
);

test_skel
(
    FeLiSe,
    W,
    L_ArgSkeleton()
    + ProlgtSkel(true)
);

test_skel
(
    FeLiSe,
    W,
    L_ArgSkeleton()
    + Cx8Skel(false)
    + ProlgtSkel(true)
);

test_skel
(
    FeLiSe,
    W,
    L_ArgSkeleton()
    + Cx8Skel(false)
    + ProlgtSkel(true)
    + SkelOffDisk(true)
);

test_skel
(
    FeLiSe,
    W,
    L_ArgSkeleton()
    + SurfSkel(10)
    + AngSkel(4.2)
    + ProlgtSkel(true)
    + ResultSkel(true)
);

for (INT i = 0; i < 2 ; i++)
{
    ELISE_COPY
    (
        FeLiSe.all_pts(),
        FeLiSe.in(),
        W.odisc()
    );
    Line_St lst(Pdisc(P8COL::red),2);
    ELISE_COPY
    (
        FeLiSe.all_pts(),
        skeleton
        (
            FeLiSe.in(0),
            30,
            new1(SurfSkel(8))
            + ProlgtSkel(true)
            + Cx8Skel(i==1)
        ),
        W.out_graph(lst)
    );
    getchar();
}
#endif

ELISE_COPY
(
    FeLiSe.all_pts(),
    ! FeLiSe.in(),
    W.odisc()
);
ELISE_COPY
(
    FeLiSe.all_pts(),
    sk_vect
    (
        skeleton(! FeLiSe.in(1)),
        new Test_Chainage
        (
            W,
            Pdisc(P8COL::yellow),
            Pdisc(P8COL::blue),
            Pdisc(P8COL::green),
            Pdisc(P8COL::red)
        )
    ),
    Output::onul()
);
getchar();

ELISE_COPY
(
    FeLiSe.all_pts(),
    ! FeLiSe.in(),
    W.odisc()
);
ELISE_COPY
(
    FeLiSe.all_pts(),
    sk_vect
    (
        (
            skeleton(! FeLiSe.in(1)),
            FX/2 +2,
            (10+FY)/7
        ),
        new Test_Attr_Chainage (W,Pdisc)
    ),
    Output::onul()
);
getchar();

REAL prec[3] = {1.0,3.0,10.0};
for (INT k = 0; k < 3 ; k++)
{
    ELISE_COPY
    (
        FeLiSe.all_pts(),
        ! FeLiSe.in(),
        W.odisc()
    );
    ELISE_COPY
    (
        FeLiSe.all_pts(),
        sk_vect
        (
            skeleton(! FeLiSe.in(1)),
            new Test_Approx
            (
                W,
                Pdisc(P8COL::blue),
                Pdisc(P8COL::red),
                prec[k]
            )
        ),
        Output::onul()
    );
    getchar();
}
}

```

# Table des figures

1.1	pseudo-code de la fonction <code>ELISE_COPY</code> . . . . .	10
1.2	code initialisant certains objets (palette, écran, fenêtre, plotter) . . . . .	11
1.3	Exemple des appels effectués aux fonctions test. . . . .	11
1.4	code pour plotter la fonction identité $x \rightarrow x$ et aspect du plotter après exécution. . . . .	11
1.5	code pour plotter la fonction constante $x \rightarrow 2$ et aspect du plotter après exécution. . . . .	12
1.6	code pour créer une image en RAM puis pour la plotter, aspect du plotter après exécution. . . . .	12
1.7	code pour plotter $y = \frac{x}{2}$ . . . . .	13
1.8	code pour plotter $y = 4 * \cos(\frac{x}{2}) + 3 + \frac{x}{5} * \sin(\frac{x}{4.9})$ . . . . .	13
1.9	code pour plotter $x \rightarrow 40 - 1.7 * I$ . . . . .	14
1.10	code pour plotter $x \rightarrow 3 * I[ x \%20]$ . . . . .	14
1.11	Utilisation de deux rectangles. . . . .	15
1.12	Opérateur (noté <code>  </code> ) de concaténation sur les flux. . . . .	15
1.13	Opérateur de sélection sur les flux. . . . .	15
1.14	Output primitifs : écriture dans une image; ici on écrit $y = (x\%10) * 5$ dans le tableau $I$ puis l'on visualise $I$ avec le plotter. . . . .	16
1.15	Opérateur (noté <code> </code> ) de mise en parallèle des <code>Output</code> . . . . .	17
1.16	Deux opérateurs sur les <code>Output</code> : changement de choordonnées ( <code>chc</code> ) et “redirection” ( <code>&lt;&lt;</code> ). . . . .	18
1.17	Exemple de message d’erreur; le code de la colone de gauche génère un débordement de tableau et conduit au message de la colone de droite. . . . .	18
1.18	Utilisation d’un tableau avec prolongement par une constante en dehors de son domaine de définition. . . . .	19
1.19	Utilisation d’un tableau avec prolongement par continuité en dehors de son domaine de définition. . . . .	19
2.1	code initialisant certains objets . . . . .	21
2.2	Code pour visualiser, en mode raster en niveaux de gris, la fonction $(x, y) \rightarrow x$ sur le carré $[0\ 256[ \times [0\ 256[$ . Aspect de $W$ à l’issue de l’exécution de ce code. . . . .	22
2.3	Exemple utilisant <code>FX</code> , <code>FY</code> et des opérateurs arithmétiques. Ligne du bas : aspect de $W$ à l’issue. . . . .	22
2.4	Declaration d’une image tiff, creation d’un tableau $2 - D$ , chargement de l’image tiff dans le tableau et la fenêtre. . . . .	23
2.5	Expression arithmétique sur les images, affichage de Lena en négatif. . . . .	24
2.6	Utilisation de plusieurs palettes. . . . .	25
2.7	Quelques exemples de binarisation (fixe, progressive et tramage). . . . .	26
2.8	Opérateur <code>,”</code> utilisé pour créer un fonction $\mathcal{Z}^2 \rightarrow \mathcal{Z}^2$ et utilisation pour effectuer un changement de coordonnées. . . . .	26
2.9	Exemple de changement de coordonnées. . . . .	27
2.10	<b>Exo.</b> . . . . .	27
2.11	Utilisation de l’opérateur de composition <code>[]</code> pour effectuer rapidement (“tabulation”) des transformations radiométriques. . . . .	28
2.12	Utilisation de palettes en RVB. . . . .	29
2.13	Utilisation d’un palette RGB pour visualiser Lena en couleur. . . . .	30
2.14	Utilisation des opérateurs de “projection” pour manipuler les cannaux d’une image indépendamment les uns des autres. . . . .	30

2.15	Illustration de l'opérateur <b>select</b> . . . . .	31
2.16	Code illustrant quelques fonctions permettant de décrire sous forme de <b>Flux_Pts</b> des primitives géométriques. . . . .	32
2.17	Résultat de Lena après le code de la figure précédente. . . . .	33
2.18	Visualisation en niveaux de gris des 3 canaux de lena en couleur . . . . .	33
2.19	Visualisation de chaque canal de <i>Lena-couleur</i> en niveau de gris, sans (colone gauche) et avec (colone droite) l'opérateur “,” sur les <b>Output</b> . . . . .	34
2.20	Deux façons équivalentes de mémoriser une image couleur dans R,G et B tout en visualisant les différents canaux. . . . .	35
2.21	Illustration de l'opérateur <b>chc</b> (changement de coordonnées) sur les <b>Flux_Pts</b> . . . . .	36
2.22	Une autre illustration de <b>chc</b> sur les <b>Flux_Pts</b> . . . . .	37
2.23	Encore une autre illustration de <b>chc</b> sur les <b>Flux_Pts</b> . . . . .	37
3.1	Initialisation de palettes, fenêtres, plotters, fichier et images. . . . .	39
3.2	Un exemple de filtre élémentaires défini à partir des opérateur de base. . . . .	40
3.3	Un exemple de gradient en <i>y</i> utilisant l'opérateur <b>trans</b> . . . . .	40
3.4	Utilisation de <b>trans</b> pour charger une portion de fichier image. . . . .	41
3.5	Exemple de code définissant un filtre utilisateur <b>moy(f,nb)</b> : moyenne de <i>f</i> sur le carré $[-n\ n] \times [-n\ n]$ . . . . .	41
3.6	<b>1</b> Dilatation en niveaux de gris, <b>2</b> Fermeture en niveaux de gris et <b>3</b> filtrage exponentiel utilisant quelques uns des opérateur de filtrage prédéfinis. . . . .	42
3.7	<b>1</b> Application d'un filtre à une fonction multi-cannal, <b>2</b> filtre de deriche donnant un résultat sur 2 canaux (visualisation sur les fenêtre <b>2</b> et <b>3</b> ) . . . . .	43
3.8	Création d'une image binaire, utilisation de cette image pour tester les opérateur de morpho-math avec le chamfrein 5 – 7 – 11. . . . .	44
3.9	Définition du filtre de sobel en utilisant uniquement des opérateurs prédéfinis d'Elise. . . . .	45
3.10	Définition “classique” d'une fonction <b>std_sobel</b> opérant directement sur des tableaux. . . . .	46
3.11	Code utilisateur pour créer un nouveau filtre “prédéfini” correspondant au gradient de sobel. . . . .	47
3.12	Exemples d'utilisation de l'opérateur <b>sobel</b> défini par l'utilisateur. . . . .	48
3.13	Utilisation d'un fonction d'extinction. Utilisation de l'opérateur de réduction associative <b>VMax</b> : réduction . . . . .	49
3.14	Calcul de l'histogramme de <i>Lena</i> . . . . .	50
3.15	Calcul “bugué” d'histogramme, en vert résultat erroné obtenu, en rouge histogramme réel. . . . .	50
3.16	Calcul de matrice de coocurrence. . . . .	51
4.1	Premier exemple d'utilisation de liste de points. . . . .	53
4.2	Utilisation de liste de points. . . . .	54
4.3	Liste de points pour stocker des associations “point/valeur”. . . . .	55
4.4	Conversion d'une liste de points en image pour utiliser son contenu dans du code utilisateur. . . . .	55
4.5	Creation de voisinage ( <b>Neighbourhood</b> ) et utilisation pour dilater un flux (colonnes 2 et 3, image aggrandies d'un facteur 3). . . . .	57
4.6	Exemple de relation de voisinage, dilatations conditionnelles. . . . .	58
4.7	Saisie d'un point, calcul de sa composante connexe, calcul de la composante connexe d'une droite. . . . .	61
4.8	Une utilisation possible de <b>conc</b> avec un germe non ponctuel. . . . .	62
4.9	Résultat du code d'analyse en particules connexes (code sur la figure suivante) . . . . .	63
4.10	Un code simple d'analyse en particules connexes. . . . .	63
4.11	Manipulation d'images sur 2 bit. . . . .	64
4.12	. . . . .	65
8.1	Header de la classe <b>Pt2d</b> (les définition des méthodes inline ont été supprimées, sauf quand elle correspondait au commentaire le plus simple). . . . .	77
9.1	Code pour générer le cercle des couleur de l'arc en ciel. . . . .	81
10.1	Arbre d'héritage des classe image . . . . .	85



10.2 Exemple d'utilisation de <code>neigh_test_and_set</code> . . . . .	88
11.1 Arbre d'héritage des classe fichiers-image . . . . .	89
11.2 Création d'un fichier <code>Elise_File_Im</code> . . . . .	91
11.3 Interface simplifiée pour les fichier <code>Elise_File_Im</code> de dimension 1, 2 ou 3. . . . .	91
11.4 Fonctions permettant de créer des fichier <code>pnm</code> . . . . .	92
11.5 Exemples de creation de fichier Tiff. . . . .	96
14.1 Code de la fonction <code>test_skel</code> illustrant les différents paramètres de <code>Skeleton</code> . . . . .	102
14.2 Exemple de squelette plus ou moins caricaturaux obtenus en faisant varier les seuils angulaires et surfaciques. . . . .	103
14.3 Illustration des options de prolongement des extrémité, 4-connexité et squelette des disques. . . . .	104
14.4 Illustration de l'option "avec résultat". . . . .	105
14.5 Squelette version filtre sur fonction numérique. . . . .	106
14.6 Utilisation de la fonction <code>sk_vect</code> pour effectuer le chaînage d'un squelette. . . . .	107
14.7 Classe <code>Test_Chainage</code> utilisée pour illustrer la fonction de chaînage de squelette <code>sk_vect</code> . . . . .	107
14.8 Utilisation de la fonction <code>sk_vect</code> pour effectuer le chaînage d'un squelette en passant des attributs. . . . .	108
14.9 Définition d'une classe de chaînage utilisant les attributs pour représenter les points du squelette par des cercles de couleur et de rayons variables. . . . .	109
14.10 Notation pour l'algorithme d'approximation polygonale. . . . .	109
14.11 Points (en gris) non étudiés avec l'option de suppression des triplets alignés . . . . .	109
14.12 Différentes itérations de l'approximation polygonale . . . . .	110
14.13 Les deux options actuellement implantées pour le segment permettant de mesurer un écart avec la polygline initiale . . . . .	110
14.14 Utilisation de la classe <code>Test_Approx</code> (voir ci-dessous) pour effectuer l'approximation polygonale du squelette. On fait le test avec differents paramètres de précision. . . . .	113
14.15 Définition d'un classe <code>Test_Approx</code> utilisant la fonction d'approximation polygonale <code>approx_poly</code> . . . . .	114

# Index

- +, 13
- ,
  - opérateur Fonc\_Num, 25
  - opérateur Output, 34
- , 13
- /, 13
- <<
  - Output, 17
- <=, 13
- ==, 13
- []
  - composition des Fonc\_Num, 13
  - transfo géométrique, 25
  - transfo radiométrique, 27
- all\_pts
  - liste de points, 54
  - plotter, 10
  - tableau, 12
- AngSkel, 102
- ArgSkeleton, 101
- border, 31
- border\_rect, 31
- Br\_Vect\_Action, 106
- canny\_exp\_filt, 41
- chc
  - Flux\_Pts, 35
  - Output, 17
- Clik, 60
- clik, 60
- close\_5711, 43
- Col\_Pal, 60
- conc, 60
- constante
  - Fonc\_Num, 12
- Cx8Skel, 102
- deriche, 42
- dilat\_5711, 43
- dilate, 56
- dimension
  - consommée par un Output, 34
  - d'une palette, 29
  - de sortie d'une Fonc\_Num, 25
- disc, 31
- draw\_circle, 60
- draw\_circle\_loc, 64
- draw\_rect, 64
- ell\_fill, 31
- ellipse, 31
- erod\_5711, 43
- Flux\_Pts, 9
- Fonc\_Num, 9
- FX, 10
- FY, 22
- FZ, 55
- GenIm
  - GenIm : :type\_el, 86
- image
  - fonction membre de Liste\_Pts, 55
- kth\_coord, 55
- kth\_proj(int), 30
- L\_ArgSkeleton, 101
- line, 31
- Line\_St, 63
- Liste\_Pts
  - classe template, 55
  - comme Flux\_Pts, 53
  - comme Output, 53
- Liste\_Pts\_INT2, 53
- mise en parallèle
  - Output, 16
- Neigh\_Rel, 56
- neigh\_test\_and\_set, 59
- Neighbourhood, 56
  - constructeur, (Im2D<INT4,INT>), 56
  - constructeur, (Pt2di \* pt,INT nb), 56
  - fonction statique v4, 56
  - fonction statique v8, 56
- onul
  - Output : :onul, 106
- open\_5711, 43
- out
  - plotter, 10
- Output, 9

- palette
  - circulaire, 24
  - indexée, 24
  - niveau de gris, 24
  - rgb, 29
- plotter, 10
- polar, 42
- polygone, 31
- ProlgtSkel, 102
- rect\_max, 41
- rect\_median, 53
- rect\_min, 41
- rectangle
  - 1-D, 14
- ResultSkel, 102
- sector\_ang, 31
- sel\_func, 58
- select
  - Flux\_Pts, 15, 31
- sigma, 48
  - fonction membre de Ptd2i et Pt2dr, 64
- sk\_vect, 106
- Skeleton, 101
- skeleton, 105
- SkelOfDisk, 102
- SurfSkel, 102
- tableaux
  - 1-D Fonc\_Num, 12, 19
  - 1-D Ouptut, 16
- Tiff\_Im
  - lena en gris, 23
- trans
  - Fonc\_Num, 40
- type\_el
  - GenIm : :type\_el, 86
- v0(), 30
- v1(), 30
- v2(), 30
- v4
  - fonction statique de Neighbourhood, 56
- v8
  - fonction statique de Neighbourhood, 56
- virgule
  - voir “,”, 25
- VMax, 48
  - fonction membre de Ptd2i et Pt2dr, 64
- VMin, 48
  - fonction membre de Ptd2i et Pt2dr, 64