Développement d'opérateurs de traitement d'images en langage C

Emmanuel TROUVÉ et Lionel VALET

Résumé

Ce document présente un ensemble de formats de données et de règles de programmation adoptés au LISTIC pour développer une bibliothèque commune de traitement de l'information (donnée 1D, 2D et 3D). Il est destiné aux personnes qui souhaitent utiliser les programmes existants et/ou contribuer à enrichir l'ensemble. Du point de vue programmation, ces règles garantissent la cohérence des structures de données et des opérateurs présents dans la bibliothèque. Elles nécessitent simplement une bonne connaissance du langage C pour développer de nouveaux opérateurs. Du point de vue utilisation, les formats de fichiers actuellement reconnus couvrent déjà une grande variété de données. Les trois modes de passage des paramètres offrent une grande souplesse pour le lancement des exécutables.

Mars 2005

Table des matières

1 Objectifs

Ces quelques pages présentent les principes de programmation adoptés au LISTIC pour créer une bibliothèque en C contenant un ensemble d'opérateurs de traitement d'images (lecture et écriture de données, filtrage, segmentation,...). Cette bibliothèque répond à quatre objectifs :

1. faciliter le démarrage des nouveaux arrivants en leur donnant des exemples documentés et un cadre de programmation qui permette ensuite d'intégrer leur travail à la bibliothèque,
2. traiter des données de dimensions 1D, 2D et 3D et de formats très différents allant du 8 bit aux données complexes en imagerie radar,
3. générer *facilement* des exécutables correspondant à des chaînes de traitements sans avoir à réécrire les opérateurs les plus classiques,
4. être compatible avec une interface graphique de lancer et paramétrer graphiquement les exécutables.

Ces objectifs nécessitent de définir quelques règles concernant :

1. les formats des fichiers pour la lecture et l'écriture des images (cf. section 2.1),
2. les types C pour la manipulation des images dans les divers fonctions (cf. section 2.2),
3. la séparation des opérateurs en trois fonctions distinctes : lecture de paramètres, initialisation et calcul, la fonction calcul pouvant être appelée de façon itérative pour les traitements bloc par bloc (cf. section 3.1.1),
4. le passage des paramètres pour être compatible avec 3 modes de lancement de programmes : *MANUEL, FICHIER* ou *AUTOMATIQUE* (cf. section 3.1.2).

L'adoption de ces règles assure la cohérence des opérateurs et des programmes mis en commun. A l'usage elles simplifient également l'écriture de programmes appelants (cf. section 3.2.1) et la création de nouveaux opérateurs (cf. section 3.2.3).

2 Formats et types associés aux images

2.1 Formats d'images (fichiers)

Deux formats d'images sont actuellement reconnus en lecture et en écriture :

1. le format *raster* (extension **.ras**) pour les images 8 bit et les images couleurs 24 bit RVB. Pour les images 8 bit, la présence de *colormap* est gérée en lecture de deux manières :
   - s'il s'agit de fausses couleurs, les tables R, V et B sont conservées pour une éventuelle recopie avec une image résultat,
   - s'il s'agit des niveaux de gris initiaux qui ont été modifiés lors de la sauvegarde au format *raster*, les tables R=V=B sont utilisées pour restaurer les vrais niveaux de gris des pixels. Elles sont ensuite ignorées.

   Sous Unix/Linux, plusieurs outils dont **xv** permettent de visualiser le format *raster*.
2. le format "ENST" qui consiste à stocker l'image en 2 fichiers (éventuellement 3) :
   - un fichier binaire qui contient uniquement les pixels rangés classiquement (ligne par ligne), avec comme extension **.ima, .imw, .imf, .cxs** et **.cxf** respectivement pour les images de type 8 bit, 16 bit, réel sur 4 octets, complexe entier 2×16 bit et complexe réel 2×4 octets (cf. tableau 1).

- un fichier ASCII avec l'extension **.dim** qui contient le nombre de colonnes puis le nombre de lignes, séparés par un espace.
- un fichier ASCII avec l'extension **.3lt** qui permet d'associer aux images 8 bit une *colormap* de type fausses couleurs. Ce fichier facultatif contient 3 séries de 256 entiers (entre 0 et 255) pour spécifier respectivement les niveaux R, V et B. En lecture, si un fichier **.3lt** accompagne les fichiers **.ima, .dim**, les tables R V et B sont lues et conservées. En écriture, si l'image comporte une *colormap*, un fichier **.3lt** est créé avec les fichiers **.ima, .dim**. En revanche, actuellement aucun outil de visualisation ne prend en compte les **.3lt**.

L'outil **xima** développé à l'ENST permet de visualiser l'ensemble de ces images (notamment les 16 bit, les float...). Pour les images 8 bit (**toto.ima, toto.dim**), la visualisation est également possible avec l'outil **xv**, en utilisant la commande **xvima toto**.

2.2 Types d'images (langage C)

Un type d'image est défini pour chaque type de variable classique, du 8 bit au complexe 16 octets. Par soucis d'homogénéité, les pixels, les structures associées aux images et les fonctions utilisent les mêmes identificateurs `u1, rvb, u2, s2, u4, s4, fl, db, cx4, cx8, cx16` pour les 11 types actuellement utilisés (cf. tableau 1). Les images sont manipulées au travers de structures déclarées dans le fichier d'*include* **image.h** sous la forme :

```
/* image entiers 16 bit non-signes */
typedef unsigned short pixu2;
typedef struct{
    pixu2  **p;          /* pointeur tableau 2D     */
    int   nc;            /* nombre de colonne       */
    int   nr;            /* nombre de lignes (rows) */
    char  nom[200];      /* nom eventuel            */
} imau2;
```

Cette structure est la même pour tous les types de pixels : `pixu1`, ...`pixcx16` et pour les structures images associées : `imau1`, ...`imacx16`, avec deux exceptions :

- le type `imau1` pour les images 8 bit comporte en plus trois pointeurs pour les tables de couleurs : `char *lutr, *lutv, *lutb ;`
- le type `imarvb` pour les images couleur 24 bit comprend 3 pointeurs de tableaux 2D : `pixu1 **r, **v, **b ;`

Données	Type C	Identificateur	Extension fichier
entier 8 bit non-signé	*unsigned char*	`u1`	**.ima** | **.ras**
couleur 3×8 bit	*3 unsigned char*	`rvb`	**.ras**
entier 16 bit non-signé	*unsigned short*	`u2`	**.imw**
entier 16 bit signé	*short*	`s2`	-
entier 32 bit non-signé	*unsigned int*	`u4`	**-**
entier 32 bit signé	*int*	`s4`	**-**
réel 4 octets	*float*	`fl`	**.imf**
réel 8 octets	*double*	`db`	**-**
complexe entier	2 × *short*	`cx4`	**.cxs**
complexe réel 8 octets	2 × *float*	`cx8`	**.cxf**
complexe réel 16 octets	2 × *double*	`cx16`	**-**

TAB. 1 – Récapitulatif des types et des formats d'images

Chaque type possède une fonction d'allocation et de libération dans **routines.c** de la forme :

```
int alloc_imaXX( imaXX *im);
int free_imau1( imaXX * im)
```

où `XX = u1 | rvb | u2 | s2 | u4 | s4 | fl | db | cx4 | cx8 | cx16`. Les champs `nr` et `nc` de la structure pointée par `im` doivent contenir le nombre de lignes et de colonnes de l'image à allouer. Le(s) tableau(x) 2D est(sont) alloué(s) en une fois et les pointeurs `im->p[j]` sont placés au début de chaque ligne pour pouvoir utiliser l'adressage 2D (`im->p[j][i]`) ou l'adressage 1D (`im->p[0] + j*im->nc + i`).

3 Développement

3.1 Principes

Une chaîne de traitement peut être vue comme un ensemble d'opérateurs tels que la lecture et l'écriture d'une ou plusieurs images, en totalité ou bloc par bloc, des filtres, des détecteurs...Le principe est d'adopter un même formalisme pour chaque opérateur de manière à d'une part créer une bibliothèque d'opérateurs réutilisables par des appels de fonctions, et d'autre part pouvoir les enchaîner "les yeux fermés" ou de façon quasi-automatique comme le ferait un générateur de programme à partir d'une description simple de la chaîne.

3.1.1 Modularité d'un opérateur

Pour chaque opérateur `oper`, on définit dans un fichier *include* (**proto2D.h** pour les opérateurs les plus classiques) :

- un descripteur sous la forme d'une structure `oper_t` qui contient l'ensemble des variables relatives à l'opérateur (paramètres, tableau de coefficients, mémoire tampon...),
- les prototypes de trois fonctions qui sont généralement de la forme :

1. `param *oper_lect(oper_t *ope, param *ptp, char *debq)` :
   lecture des paramètres liés à l'opérateur. Les questions posées par cette fonction peuvent être personnalisées à l'aide de la chaîne de caractère `debq` qui sera placée au début de chaque question. L'utilisation d'un pointeur de paramètre `ptp` assure la compatibilité avec les trois modes de lancement de programme (cf. section 3.1.2).
2. `int operXX_init(oper_t *ope, imaXX im0, imaXX *imres)` :
   initialisation des variables en fonction des paramètres précédents et des dimensions de l'image(s) entrant (`im0`) ; allocation de l'image(s) sortant (`imres`).
3. `int operXX_calc(oper_t *ope, imaXX im0, imaXX *imres)` :
   calculs de l'image(s) sortant. Cette fonction est susceptible d'être appelée de façon itérative dans le cas de traitement bloc par bloc.

Les trois fonctions sont écrites dans un fichier source `loper.c` (`l` comme librairie). Un programme faisant appel à cet opérateur est généralement créé (ne serait-ce que pour le tester) dans un fichier source `moper.c` (`m` comme `main`). A titre d'exemple, un opérateur *essai* est disponible avec son descripteur dans **/prog-listic/MethodesFusion/BATI/pour_demarrer/essai.h**, les fonctions `essai_lect()`, `essai_init()` et `essai_calc()` dans **/prog-listic/MethodesFusion/BATI/pour_demarrer/les** et un programme appelant dans **/prog-listic/MethodesFusion/BATI/pour_demarrer/messai_u1.c**. L'exécutable **messai_u1** et un exemple de fichier de contrôle **messai_u1.ctrl** sont dans **/prog-listic/MethodesFusion/BATI/pour_demarrer/**.

3.1.2 Modes de passage des paramètres

Trois modes de lancement de programme ont été retenus pour le passage des paramètres :

1. *MANUEL* :
   lancement sans arguments avec saisie manuelle des paramètres et création d'un fichier de contrôle d'extension **.ctrl** contenant les questions et les paramètres entrés,
2. *FICHIER* :
   lancement avec 1 seul argument sur la ligne de commande : le nom d'un fichier de contrôle reconnu par son extension **.ctrl**. Les questions sont posées avec les valeurs par défaut lues dans ce fichier. Il suffit de taper *enter* pour accepter la valeur par défaut ou entrer une nouvelle valeur pour modifier un paramètre. Les nouveaux paramètres sont sauvegardés dans un fichier **.ctrl** (le même ou un autre).
3. *AUTOMATIQUE* :
   lancement avec les paramètres passés dans la ligne de commande par `argv`. Ce mode pourra être notamment utilisé par une interface graphique qui composera la ligne de commande à partir d'un fichier **.ctrl** et des paramètres modifiés par l'utilisateur. L'interface assurera elle-même la sauvegarde des paramètres utilisés.

Ces trois modes coexistent grâce à l'utilisation d'une liste chaînée de structures contenant questions et réponses sous forme ASCII :

```
struct parametre{
    char qst[200];  /* question  */
    char rep[200];  /* reponse   */
    struct parametre *next;
};
typedef struct parametre param;
```

Cette liste et les trois modes de passage de paramètres sont gérés de façon quasi-transparente avec les règles suivantes :

- déclarer dans le `main` deux variables : `param par0, *ptp ;`
  `par0` pour le début de la liste chaînée,
  `ptp`, de type pointeur, pour la position courante dans la chaîne.
- démarrer et terminer la lecture de paramètres en appelant les fonctions `param_debut()` et `param_fin()` (cf. section 3.2.1 et la doc HTML),
- lors de l'appel des fonctions de lecture des opérateurs, transmettre `ptp` déjà alloué et pointant sur le prochain paramètre à lire et le récupérer dans sa nouvelle position. Par exemple pour l'opérateur *bidon* : `ptp = bidon_lect(&bid, ptp, debq) ;`
- dans les fonctions de lecture des opérateurs et dans le `main`, lire les paramètres par la fonction `lec_param()`, récupérer le résultat par les fonctions `atoi(), atof()` ou `strcpy()` et avancer le pointeur de paramètre sur la structure suivante. Les lectures de paramètres se font donc en trois instructions, par exemple pour un entier `dimX` avec une question dans la chaîne `qst` :

```
lec_param(qst, ptp);
dimX = atoi(ptp->rep);
ptp = ptp->next;
```

3.2 Utilisation

On peut distinguer trois niveaux d'utilisation selon si l'on souhaite simplement faire appel à des opérateurs existants, développer rapidement un nouvel algorithme ou intégrer dans la bibliothèque un nouvel opérateur.

3.2.1 Appel d'opérateurs existants

Il s'agit ici d'écrire un programme principal qui enchaîne un ensemble d'opérateurs existants. Un `main` se décompose généralement en 5 parties, avec des opérateurs de lecture et d'écriture différents selon si l'image(s) est traitée(s) en une fois : `read_ima, write_ima` ou bloc par bloc (bpb) : `select_ima, save_ima`.

1. les déclarations :
   - les images (ou blocs) où transitent les données :
     ```
     imaXX_t im0, im1, im2, ...imres ;
     ```
   - les descripteurs associés aux opérateurs :
     ```
     read_ima_t rea ; /* bpb : select_ima_t sel ; */
     opeB_t opB ;
     opeC_t opC ;
     ...
     write_ima_t wri ; /* bpb : save_ima_t sav ; */
     ```
   - pour la liste chaînée des paramètres les variables :
     ```
     param par0, *ptp ;
     ```
2. la lecture des paramètres, encadrée par les fonctions `param_debut()` et `param_fin()`, avec passage et retour du pointeur de paramètre `ptp` et personnalisation des débuts de question (``Q'') :
   ```
   param_debut(argc, argv, &par0) ;
   ptp = &par0 ;
   ptp = read_ima_lect(&rea, ptp, ``Q'') ; /*bpb : select_ima_lect(&sel, ptp, ``Q'')*/
   ptp = opeB_lect(&opB, ptp, ``Q'') ;
   ptp = opeC_lect(&opC, ptp, ``Q'') ;
   ...
   ptp = write_ima_lect(&wri, ptp, ``Q'') ; /*bpb : save_ima_lect(&sav, ptp, ``Q'')*/
   param_fin(argc, argv, &par0) ;
   ```
3. l'initialisation : (les fonctions peuvent dépendre du type `XX`)
   ```
   read_imaXX_init(&rea, &im0) ; /* bpb : select_imaXX_init(&sel, &im0) ; */
   opeBXX_init(&opB, im0, &im1) ;
   opeCXX_init(&opC, im1, &im2) ;
   ...
   write_ima_init(&wri) ; /* bpb : save_ima_init(&sav) ; */
   ```
4. le calcul : (itératif si traitement bloc par bloc)
   ```
   /* bpb : while( select_imaxx_calc(&sel, &im0) ){ */
         opeBXX_calc(&opB, im0, &im1) ;
         opeCXX_calc(&opC, im1, &im2) ;
         ...
   ```

```
        /* bpb : save_imaXX_calc(&sav, imres) ; */
    /* bpb : } */
```

5. la fermeture :
   `write_imaXX_ferm(&wri, imres) ; /* bpb : save_ima_ferm(sel, sav) ; */`

3.2.2 Développement rapide

Pour le développement rapide d'un algorithme, le plus simple est de partir d'un programme existant, par exemple **mbidon_u1.c** pour traiter une image 8 bit en une fois. On peut utiliser des opérateurs présents dans la bibliothèque (cf. section 3.2.1) et insérer directement les parties propres à l'algorithme développé : variables, lecture de paramètres par `lec_param()`, calculs...L'algorithme peut ensuite être repris pour être mis sous la forme d'opérateur (cf. section 3.2.3) ou laissé sous la forme actuelle. Ce programme peut venir enrichir l'ensemble des exécutables de la bibliothèque Une petite description doit être placée au début du fichier source pour l'extraction automatique de la documentation (cf. section 4.1).

3.2.3 Création d'un opérateur

Si un algorithme est susceptible d'être appelé par d'autres chaînes de traitement, il est intéressant de l'écrire (ou le réécrire) sous la forme d'un opérateur pour l'inclure dans la bibliothèque. Il faut alors regrouper ses variables dans une structure et séparer les parties lecture de paramètres, initialisation et calcul. Les étapes sont les suivantes pour un opérateur `toto` :

- fichier *include* (**toto.h** ou un **.h** collectif) : définition de la structure liée à l'opérateur contenant toutes les variables utiles (paramètres, mémoire de travail ...) et prototypage des fonctions `toto_lect()`, `toto_init()`, `toto_calc()`,
- fichier *librairie* **ltoto.c** : corps des fonctions `toto_lect()`, `toto_init()`, `toto_calc()`, avec une documentation globale pour l'opérateur incluant la structure commentée du descripteur et des sous-parties pour les 3 fonctions (cf. **lbidon.c**).
- fichier `main` **mtoto.c** : chaîne de traitement minimale permettant de tester les fonctions et de générer un exécutable autonome **mtoto**. Prévoir une petite documentation en début de fichier source pour l'extraction automatique et un fichier paramètres **mtoto.ctrl** pour accompagner l'exécutable.
- selon l'intérêt (et le temps disponible !), dupliquer pour différents types d'images les fonctions qui varient : `totou1_calc()`, `totou2_calc()`...et les programmes appelants : **mtoto_u1.c**, **mtoto_u2.c**...Faire apparaître les différents types dans la documentation sous la forme : `totoXX_init, XX = u1 | u2 | etc` et `mtoto_XX, XX = u1 | u2 | etc`.

4 Documentation des programmes.

La documentation de la librairie au format html se trouve dans **/prog-listic/MethodesFusion/BATI/doc**. Elle est générée automatiquement à partir des fichiers sources grâce à un logiciel libre, nommé Doxygen.

4.1 Doxygen

Doxygen (http ://www.stack.nl/~dimitri/doxygen/index.html) est un système de documentation fonctionnant aussi bien avec des programmes C, C++ ou JAVA. Les commentaires reconnus par Doxygen sont délimités par

```
/**
...
*/
```

ou

```
/*!
...
*/
```

Les commentaires du type `/* ... */` sont ignorés ce qui permet de placer des informations utiles aux programmeurs sans pour autant que celles-ci soient récupérées par Doxygen.

De plus, Doxygen reconnaît un certain nombre de mots clés permettant de rendre la documentation plus agréable à lire :
- \`brief` : brève description de la fonction.
- \`author` : nom des développeurs.
- \`param` : paramètres d'entrée.
- \`return` : valeur retournée.
- \`version` : peut contenir la date.
- \`see` : référence à d'autres programmes, matérialisés ou non par un lien hypertexte.

Le caractère \ permet d'indiquer à Doxygen que le mot suivant est une commande, c'est donc un caractère réservé. Cette liste n'est pas exhaustive mais correspond aux éléments de base utilisés par la suite.

Un commentaire correspondant à une entrée dans la page html, la liste des entrées peut vite devenir volumineuse. Il est parfois intéressant de définir des sous-entrées à une entrée principale. Ceci est réalisé à l'aide des commandes \`defgroup` et \`ingroup`. Leur utilisation est illustré ci-après notamment pour la documentation des opérateurs.

Il est également possible de mettre des équations mathématiques au format LaTeX dans la documentation en utilisant les balises \f$ (insertion dans le texte) ou \f[(insertion en dehors du texte). Les équations apparaissent dans la sortie html sous forme d'images aux formats gif.

4.2 Application à BATI

L'utilisation de Doxygen aux sources de BATI va distinguer 2 cas :
- les opérateurs.
- les exécutables.

Pour chacun d'eux, le format de documentation à respecter sera différent

4.2.1 Opérateurs

Les opérateurs sont formés d'un ensemble de fonctions (lecture, initialisation, calcul, cf section 3.1.1). La documentation d'un opérateur comprend :

- Une documentation générale située au début du fichier source ayant la forme suivante :

```
/** \defgroup essai

Cet opérateur ne fait rien.\n
Inclure tout de même le fichier bidon.h

@author toto
@version 1999
@see
*/
```

  La commande defgroup permet de définir un nouveau groupe
- Des sous-entrées pour chacunes des fonctions devant apparaître dans la page html, avec la commande ingroup pour les rattacher au groupe créé précédemment.

```
/** \ingroup essai

    XX = u1 | etc : initialisation de l'operateur essai.

    allocation image resultat et images de travail,
    calcul des coeff du masque

    @param des  pointeur descripteur
    @param im0  image initiale (taille connue)
    @param imres  pointeur image resultat
    @author toto
    @version 1.0 (19/10/99); Include : proto2D.h, essai.h
*/
int essaiu1_init(essai_t *des, imau1 im0, imau1 *imres){
```

4.2.2 Exécutables

Pour le moment, les exécutables seront vus comme une fonction particulière de l'opérateur. Ils auront donc une documentation de la forme suivante :

```
/**
\ingroup essai

messaiXX = u1 | etc : executable(s) associe(s) a l'operateur "essai" - programme type.\n
Entree : une image type XX = u1 | etc\n
Sortie : une image type XX = u1 | etc\n
Description : programme type illustrant la saisie de parametres
et l'utilisation des operateurs read_ima, essai et write_ima.\n
Effectue une simple recopie de l'image entrante dans l'image sortante.

\author toto
\version 1.0 (19/10/99)
\see lread_ima, lessai et lwrite_ima
*/
```

En guise d'exemple, il est possible de consulter (accès en lecture seule) les sources qui se trouvent dans **/prog-listic/MethodesFusion/BATI/pour_demarrer**. Il est recommandé de compiler la documentation chez soi avant tout ajout dans la librairie commune afin de s'assurer du résultat. Pour cela utiliser la commande suivante : `doxygen -g DocConfigFile` pour générer un fichier de configuration (Editer le fichier pour le paramètrer selon vos attentes). Puis `doxygen DocConfigFile` dans le répertoire où se trouve les fichiers sources . L'exécutable `doxygen` est disponible sur toutes les distributions Fedora. Le résultat se présente sous la forme d'un répertoire html contenant la documentation au format html. Il ne reste plus qu'à visualiser le fichier **index.html**.