Langage C pour le TSI

Programme

Marc Donias

Introduction

Le Langage C est l'un des meilleurs choix possibles pour l'implémentation efficace d'algorithmes. Il permet tout à la fois d'écrire un code « bas niveau », très proche des modes de fonctionnement voire des spécificités des matériels (processeurs, mémoires, etc.), et d'utiliser des structures de données complexes permettant d'atteindre un niveau d'abstraction élevé. La vitesse d'exécution des programmes obtenus est presque sans équivalent, au prix toutefois d'un temps de développement relativement important du fait de nombreuses erreurs possibles.

Une programmation avancée en Langage C nécessite une parfaite maîtrise de la notion de pointeurs. Dans le cas du traitement du signal, une des premières questions qui se pose concerne le codage d'un signal (cf. annexe) qu'il soit monodimensionnel, bidimensionnel (images) ou de dimension supérieure. Les algorithmes opérant sur les signaux requièrent également l'utilisation de structures de données informatiques classiques (listes chaînées, piles, files, arbres, graphes, etc.) qui font un usage intensif des pointeurs.

Dans l'objectif d'une programmation efficace, un intérêt tout particulier doit être porté à l'optimisation au sens large : ceci concerne aussi bien le génie logiciel (cohérence d'écriture, pertinence de la structuration, etc.) que l'amélioration de la vitesse d'exécution.

Le but recherché ici est avant tout pédagogique : explorer les différentes possibilités offertes par le Langage C et les illustrer à travers un exemple concret de traitement du signal. Nous opérons sur des images qui permettent une validation généralement plus immédiate. Afin de s'affranchir de soucis de visualisation et d'accès aux données, les algorithmes seront développés sous la forme de librairies dynamiques pour N'D, une interface de visualisation et de traitement d'images et de vidéos.

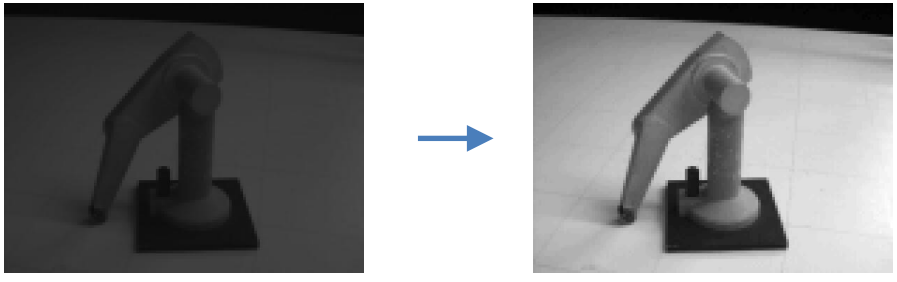
Pointeurs de A à Z

- Notions de base
  - Passage par référence
  - Tableaux
  - Allocation
- Types composés
- Arithmétique de pointeurs
- Notions avancées
  - Pointeurs génériques
  - Pointeurs de pointeurs
  - Pointeurs de fonctions
- Codage d'un signal
  - Signal
  - Image
  - Multidimensionnel

Passage par référence (1/2)

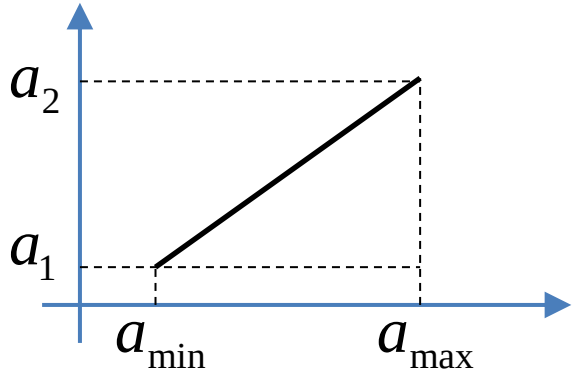
Changement de dynamique

Un changement de dynamique permet souvent d'améliorer l'aspect visuel d'une image. Il s'agit de remplacer l'intervalle $[a_{min}, a_{max}]$ des intensités initiales par un nouvel intervalle $[a_1, a_2]$ plus adapté à la visualisation (classiquement, a_1=0 et a_2=255).



La mise en œuvre nécessite de calculer les valeurs extrémales a_{min} et a_{max} puis d'appliquer une transformation linéaire.

$$I'_{i,j} = a_1 + \frac{a_2 - a_1}{a_{\max} - a_{\min}} \left(I_{i,j} - a_{\min} \right)$$



Passage par référence (2/2)

Ecrire la procédure qui calcule les valeurs extrémales v_{min} et v_{max} d'un vecteur v_{in} :

```
void v_compute_min_max(double * v_in, long size,
                       double * v_min, double * v_max);
```

Ecrire ensuite une procédure de changement de dynamique d'une image :

```
void img_stretch_intensity(double * img_in,
                           long width, long height,
                           double * img_out);
```

Tableaux (1/2)

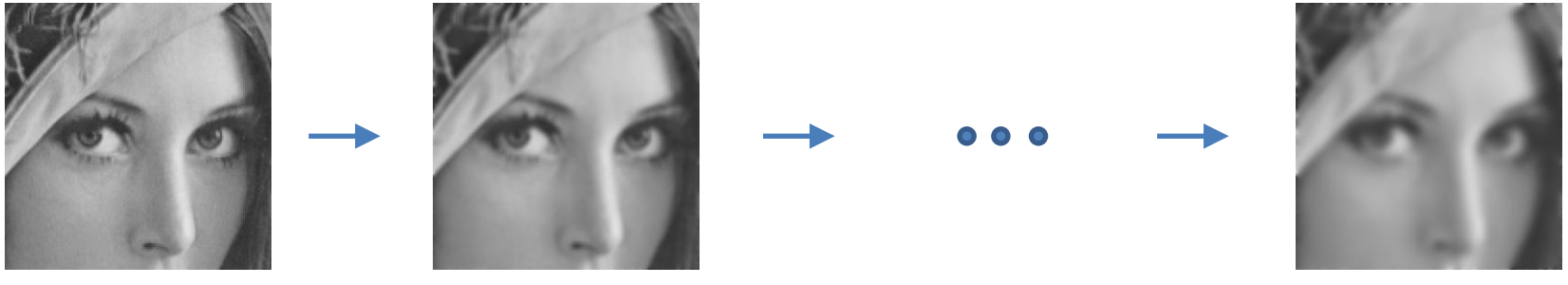
Filtres linéaires 2D

L'application d'un filtre linéaire H de taille $(2M+1) \times (2N+1)$ s'obtient par une équation de convolution :

$$I'_{i,j} = \sum_{m=-M}^{M} \sum_{n=-N}^{N} I_{i-m,j-n} H_{m,n}$$

Au moyen d'un tableau 2D statique (sans allocation dynamique), écrire une procédure de filtrage passe-bas élémentaire par un « moyenneur rectangle 3×3 » :

```
void filter2d_mean3(double * img_in, long width, long height,
                    double * img_out);
```

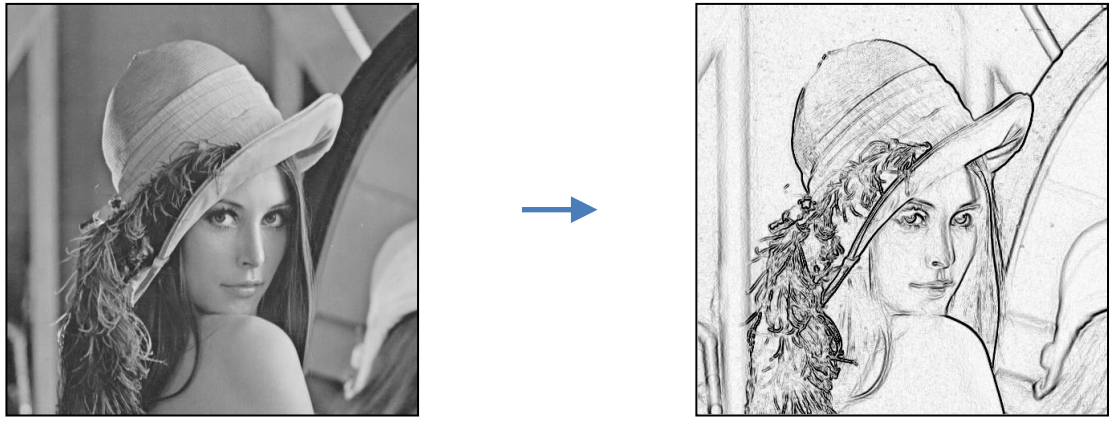


Tableaux (2/2)

De manière similaire, écrire une procédure qui calcule la norme du gradient de l'opérateur de Sobel :

```
void sobel_norm(double * img_in, long width, long height,
                double * img_out);
```

$$S_x = \frac{1}{8}\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \qquad S_y = \frac{1}{8}\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$



C Allocation et transtypage (1/5)

Filtres linéaires 2D

Ecrire une fonction qui construit (allocation et calcul des coefficients) un filtre passe-bas 2D de type gaussien :

```
double * gaussian2d_create(double sigma, long * size);
```

Nécessairement tronqué, sa taille M sera automatiquement déterminée en fonction de son écart-type σ (par exemple, $\mathrm{M} = 1 + 2 \times C(3\sigma)$).

Ecrire une procédure de convolution 2D :

```
void convolution2d(double * img_in, long width, long height
                   double * mask2d, long tx, long ty,
                   double * img_out);
```

Ecrire ensuite une procédure de lissage gaussien 2D :

```
void filter2d_gaussian(double * img_in, long width, long height,
                       double sigma,
                       double * img_out);
```

Allocation et transtypage (2/5)

Réécrire cette procédure en tenant compte de la séparabilité du filtre gaussien :

$$H_{2D}(x,y) = H_{1D}(x)H_{1D}(y)$$

au moyen des différentes fonctions et procédures suivantes :

- ✓ une fonction qui construit un filtre passe-bas 1D de type gaussien

```
double * gaussian1d_create(double sigma, long * size);
```

- ✓ une procédure « d'extraction » de ligne

```
void img_get_raw(double * img, long width, long height,
                 long no,
                 double * v);
```

- ✓ une procédure « d'insertion » de ligne

```
void img_set_raw(double * img, long width, long height,
                 long no,
                 double * v);
```

Allocation et transtypage (3/5)

- ✓ une procédure « d'extraction » de colonne

```
void img_get_column(double * img, long width, long height,
                    long no,
                    double * v);
```

- ✓ une procédure « d'insertion » de colonne

```
void img_set_column(double * img, long width, long height,
                    long no,
                    double * v);
```

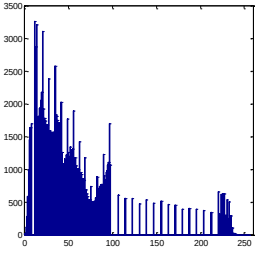
- ✓ une procédure de convolution 1D

```
void convolution1d(double * v_in, long size,
                   double * mask1d, long t,
                   double * v_out);
```

Allocation et transtypage (4/5)

Egalisation d'histogramme

Il s'agit de modifier une image de sorte que son histogramme calculé sur N classes tende vers une répartition uniforme.



I_e
image d'entrée

$(a_1,\ldots,a_{N+1})$ classes

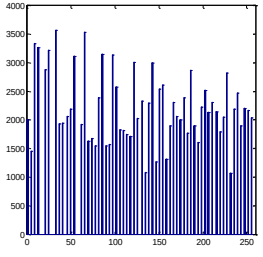
Calcul d'histogramme

H

Calcul du changement de classe

k

Affectation d'une nouvelle intensité

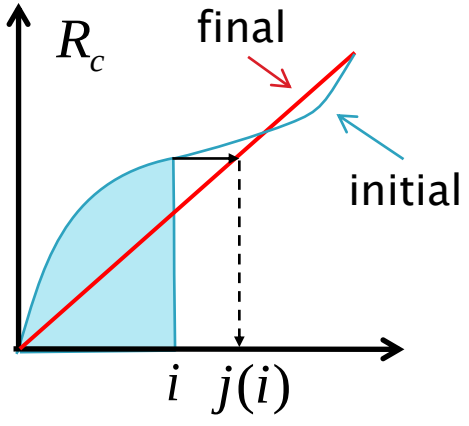


I_s
image de sortie

Allocation et transtypage (5/5)

Soient a_{min} et a_{max} les valeurs extrémales de l'image initiale, et R son histogramme ou tableau d'occurrences. Chaque classe i de R est associée à une nouvelle classe j :

$$j(i) = \underset{k}{ArgMax}\left(\sum_{n=0}^{k} R(n) \leq \frac{i}{N-1}\sum_{n=0}^{N-1} R(n)\right)$$



dont l'indice j et le représentant en valeur $v(j)$ peuvent être directement obtenus par :

$$j(i) = E\left[(N-1)\frac{\sum_{n=0}^{i} R(n)}{\sum_{n=0}^{N-1} R(n)}\right] \qquad v(j) = a_{\min} + \frac{j+\frac{1}{2}}{N}(a_{\max} - a_{\min})$$

Ecrire une procédure d'égalisation d'histogramme de prototype :

```
void img_equalize_histogram(double * img_in,
                            long width, long height,
                            long nb_levels, double * img_out);
```

Types composés (1/2)

Quantification de couleurs

A des fins d'affichage ou de compression, il peut être nécessaire de réduire le nombre de couleurs différentes d'une image. Dans le cas d'un codage RVB sur 24 bits, il s'agit d'identifier *N* couleurs, parmi 16 777 216 possibles, susceptibles de représenter au mieux l'image traitée.
Une méthode statistique simple consiste à analyser le nuage colorimétrique d'une image : les populations (ou amas) de couleurs très proches, isolées puis réduites à un représentant, permettent un codage sur un nombre restreint de couleurs.

Types composés (2/2)

L'extension réelle de l'espace RVB délimité par les pixels de l'image traitée est décomposé en 216 régions de taille (ou extension) identique. Les différents amas obtenus sont représentés par leur barycentre.

Ecrire une procédure de réduction de nombre de couleurs à 216, de prototype :

```
void img_reduce_colors_to_216(unsigned char * img_color_in,
                              long width, long height,
                              unsigned char * img_color_out);
```

Les différentes caractéristiques des régions seront regroupées au sein de la structure :

```
struct AREA_216
  {
    unsigned char bary_r, bary_g, bary_b;
    long nb_points;
    long sum_r, sum_g, sum_b;
  };
```

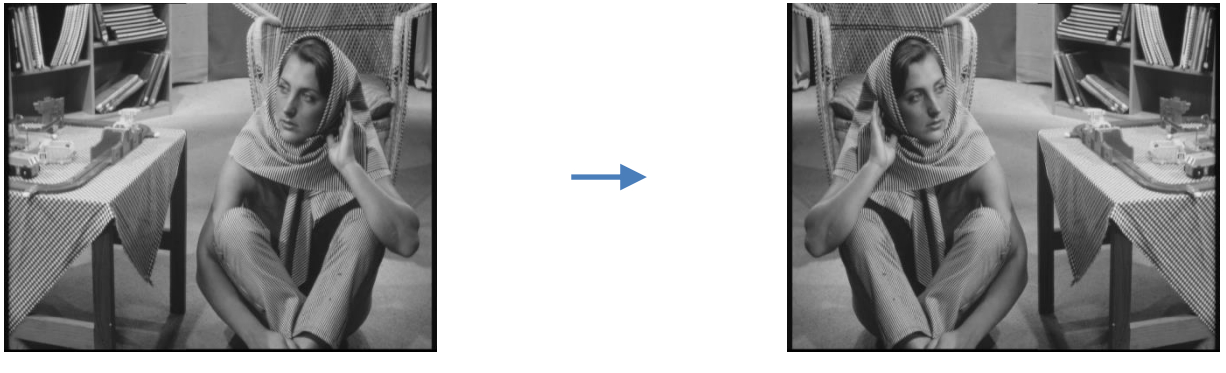
C Arithmétique des pointeurs (1/2)

Types simples – Miroir horizontal

Ecrire une procédure réalisant un miroir horizontal selon le prototype :

```
void img_mirror_horizontal(double * img_in,
                           long width, long height,
                           double * img_out);
```

Deux boucles imbriquées, l'une sur les lignes et l'autre sur les colonnes, ainsi que deux pointeurs (de début et de fin de ligne) seront mis en œuvre afin d'éviter des calculs d'adresse et d'obtenir une exécution « rapide ».



C Arithmétique des pointeurs (2/2)

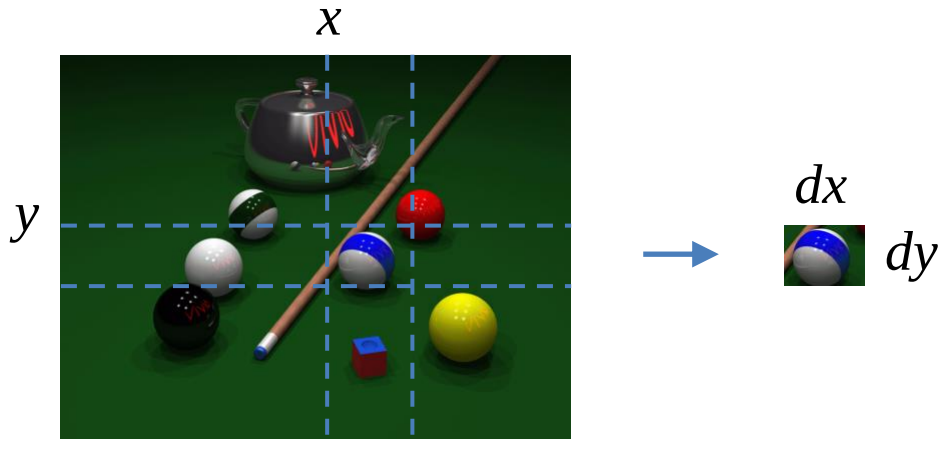
Types composés – Découpe

Similairement, écrire une procédure qui extrait une région rectangulaire, de largeur dx et de hauteur dy, d'une image en couleurs et de prototype :

```
void img_extract_area(unsigned char * img_color_in,
                      long width, long height,
                      long x, long y, long dx, long dy,
                      unsigned char * img_color_out);
```

à l'aide de la structure

```
struct RGBA
  {
    unsigned char r;
    unsigned char g;
    unsigned char b;
    unsigned char a;
  };
```



Pointeurs génériques

Rotation

Ecrire une procédure de rotation d'un quart de tour d'une image qui traite indifféremment le cas d'intensités scalaires ou de vraies couleurs selon le prototype :

```
void img_rotate(void * img_in, long width, long height,
                long type_data,
                void * img_out);
```

Par convention, pour le paramètre `type_data`, la valeur 0 indiquera une intensité scalaire en précision flottante sur 64 bits tandis que la valeur 1 indiquera une image en vraie couleur sur 24 bits.

Pointeurs de pointeurs

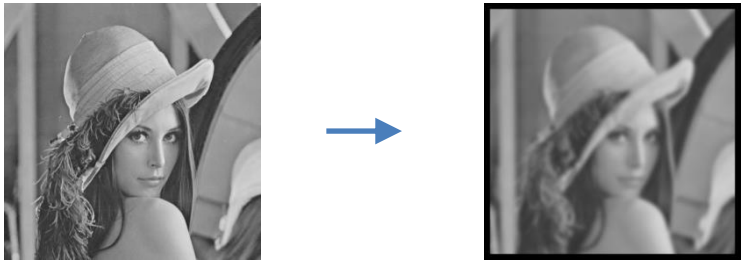
Filtres linéaires 2D

Ecrire une fonction qui construit (allocation et calcul des coefficients) un filtre passe-bas 2D de type gaussien sous la forme d'une matrice (tableau bidimensionnel) selon le prototype :

```
double ** gaussian2d_create_matrix(double sigma,
                                   long * width, long * height);
```

Ecrire une procédure de convolution 2D de prototype :

```
void convolution2d_by_matrix(double * img_in,
                             long width, long height
                             double ** mask2d, long tx, long ty
                             double * img_out);
```



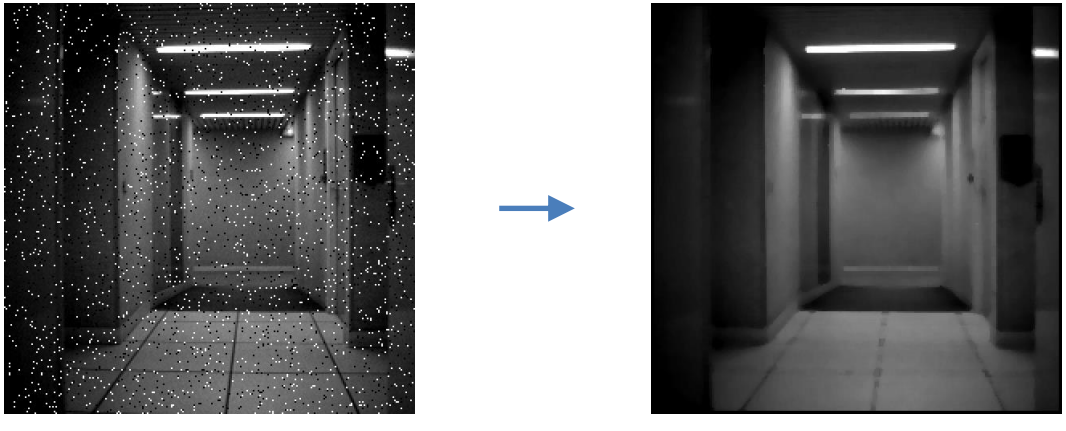
Pointeurs de fonctions (1/3)

Filtre médian 2D

Utiliser la procédure `qsort`, de la librairie C ANSI, afin d'effectuer un filtrage non linéaire 2D de type médian sur une image selon le prototype :

```
void filter2d_median(double * img_in, long width, long height,
                     long tx, long ty,
                     double * img_out);
```

Il sera nécessaire d'écrire une fonction de comparaison passée en paramètre de la procédure `qsort`.



Pointeurs de fonctions (2/3)

Diffusion de couleur dominante

La méthode consiste à remplacer la couleur de chaque pixel par celle de son voisinage, en y incluant le pixel courant, qui se caractérise par la plus forte saturation d'une des composantes R, V ou B.

Le prototype de la procédure à implémenter est le suivant :

```
void img_diffuse_hot_color(unsigned char * img_color_in,
                           long width, long height,
                           long tx, long ty,
                           unsigned char * img_color_out);
```

La procédure `qsort` sera mise en oeuvre avec la structure RGBA déjà utilisée et une fonction de comparaison opérant sur des triplets (R, V, B).

Pointeurs de fonctions (3/3)

Filtrage 2D « libre »

Il s'agit ici d'écrire une procédure de filtrage local 2D opérant sur des fenêtres glissantes à l'aide d'une méthode passée en paramètre. Nous ne considérons que des techniques sans paramètres et indifférentes aux positions des pixels traités. Chaque fenêtre glissante, centrée sur un pixel de l'image initiale, donne lieu à l'appel d'une fonction de filtrage 1D dont le prototype est :

```
double * v_filter1d(double * v_in, long size);
```

La procédure de filtrage 2D a pour prototype :

```
void filter2d_by_method(double * img_in,
                        long width, long height,
                        double (* method)(double *, long));
```

Les méthodes à implémenter sont respectivement la moyenne, le minimum, le maximum et la médiane.

Codage d'un signal (1/4)

Signal

Un signal discret monodimensionnel $S = \{S_1, \ldots, S_N\}$ de N échantillons se code simplement sous la forme d'un tableau à une seule dimension ou vecteur V de longueur N et dont chaque cellule contient la valeur d'un échantillon.

$V[0] = S_1$	$\cdots$	$V[i] = S_{i+1}$	$\cdots$	$V[N-1] = S_N$

Contrairement à la convention généralement utilisée en algèbre, les indices de numérotation de cellule en Langage C commencent à 0.

La taille en octets de chaque cellule dépend de l'information codée :

- ✓ types simples (char, int, short, long, float, double, etc),
- ✓ types composés (structures).

Codage d'un signal (2/4)

Image

Codage d'un signal (3/4)

Codage d'un signal (4/4)

Multidimensionnel