Algorithmique et Programmation

—

Examen sur machine

—

G1: keriven*(at)*certis.enpc.fr G2: juan*(at)*certis.enpc.fr
G3: gmellier*(at)*melix.org G4: pierre.maurel*(at)*ens.fr
G5: pierre*(at)*senellart.com G6: adde*(at)*certis.enpc.fr

—

07/01/05

—

1 Calcul de l'exponentielle d'un nombre complexe

Le but de cet exercice est de calculer l'exponentielle d'un nombre complexe et de s'en servir pour calculer le sinus et le cosinus d'un angle.

1. Partir d'un projet "console" `Win 32 Basic Console`
2. Ajouter au projet deux fichiers `complexe.cpp` et `complexe.h` dans lesquels vous déclarerez et définirez à l'endroit approprié :
   (a) une structure `complexe` (et pas `complex` qui existe déjà) représentant un nombre complexe sous forme cartésienne (partie réelle, partie imaginaire)
   (b) les opérateurs `+`, `*` entre deux `complexe`
   (c) l'opérateur `/` définissant la division d'un `complexe` par un `double`
3. On souhaite approximer la fonction exponentielle en utilisant son développement en série entière :

$$e^z = \sum_{i=0}^{+\infty} \frac{z^i}{i!}$$

   Écrire une fonction `exponentielle`, prenant en argument un `complexe z` et un `int n`, et qui retourne la somme :

$$\sum_{i=0}^{n} \frac{z^i}{i!}$$

4. Écrire une fonction `cos_sin` qui renvoie le cosinus et le sinus d'un angle θ en utilisant le développement limité de $e^{i\,\theta}$ à l'ordre n (on passera donc, en plus de l'angle `theta`, l'entier `n` en argument). On rappelle que :

$$e^{i\,\theta} = \cos\theta + i\,\sin\theta$$

5. Tester la fonction `cos_sin` pour différentes valeurs de `theta` et `n`. Vérifier qu'avec $n = 15$ et $\theta = \frac{\pi}{6}$, on obtient une bonne approximation des valeurs du cosinus ($\frac{\sqrt{3}}{2} \approx 0.866025404$) et du sinus ($\frac{1}{2}$).

2 Compression RLE

Dans cet exercice nous allons implémenter l'une des plus anciennes méthodes de compression : le codage RLE (Run Length Encoding). Le principe consiste à détecter une donnée ayant un nombre d'apparitions consécutives qui dépasse un seuil fixe, puis à remplacer cette séquence par deux informations : un chiffre indiquant le nombre de répétitions et l'information à répéter. Aussi, cette méthode remplace une séquence par une autre beaucoup plus courte moyennant le respect du seuil (que nous fixerons, par simplicité, à 0). Elle nécessite la présence de répétitions relativement fréquentes dans l'information source à compresser.

Cette méthode présente peu d'avantages pour la compression de fichier texte. Par contre, sur une image, on rencontre régulièrement une succession de données de même valeur : des pixels de même couleur.
Sur une image monochrome (un fax par exemple), l'ensemble à compresser est une succession de symboles dans un ensemble à deux éléments. Les éléments peuvent être soit des 255 (pixel allumé=blanc), soit des 0 (pixel éteint=noir) ; il est relativement facile de compter alternativement une succession de 0 et de 255, et de la sauvegarder telle quelle.

$$\begin{aligned} \textit{source} &: 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 255\ 255\ 255\ 255\ 255\ 255\ 255\ 255\ 0\ 0\ 0\ 0\ 0\ 255\ 255\ 255\ 255 \\ \textit{compression} &: 10\ 8\ 5\ 4 \end{aligned}$$

Cependant, une convention reste à prendre : savoir par quel pixel commencera la succession de chiffres. Nous considèrerons que le premier nombre représente une succession de 0. Si la source ne commence pas par des pixels noirs (un zéro) il faut alors commencer la chaine codante par un 0 pour indiquer l'absence de pixel noir.

Dans ce qui suit nous allons encoder une image binaire (constituée de 0 et de 255 stockés dans un tableau de `byte`) dans un tableau de `int`.

1. Partir d'un projet "Winlib" (`Winlib5 Project`).
2. Dans la fonction `main`,
   (a) Déclarer les variables de type `const int` w et h representant la largeur et la hauteur de l'image à encoder. Ces variables pourront être initialisées à la valeur 256.
   (b) Déclarer le tableau de `byte` *image_source* de taille $w \times h$ (rappel : le pixel (i, j) est l'élément $i + j * w$ du tableau).
   (c) Déclarer le tableau de `byte` *image_decodee* de taille $w \times h$ (rappel : le pixel (i, j) est l'élément $i + j * w$ du tableau).
   (d) Déclarer le tableau de `int` *image_encodee* de taille $w \times h + 1$
3. Créer une fonction $affiche\_image$ de déclaration :
   `void affiche_image( byte image[], int w, int h );`
   Cette fonction affiche l'image stockée dans *image* de largeur w et de hauteur h à l'aide de la fonction *PutGreyImage* de la Winlib.
4. Dans la fonction *main*, tester la fonction $affiche\_image$ avec *image_source*, sans l'avoir initialisée.
5. Créer une fonction *remplir_rectangle* de déclaration :

```
void remplir_rectangle(
       byte image[], int w, int h,
       int xul, int yul, int height, int width );
```

   Cette fonction dessine un rectangle blanc dans une image en mettant à 255 tous les pixels de l'image contenus dans le rectangle. L'image dans laquelle est dessiné le rectangle plein est donnée par le tableau *image* et les dimensions w et h de l'image. Le rectangle à remplir est donné par les coordonnées *xul* et *yul* de son coin supérieur gauche ainsi que par ses dimensions *height* et *width*.
6. Nous allons maintenant tester la fonction *remplir_rectangle*. Pour ce faire :
   (a) Remplir l'image avec des 0.
   (b) Utiliser la fonction *remplir_rectangle* sur cette image pour y placer un rectangle de votre choix.
   (c) Afficher cette image à l'aide de $affiche\_image$
7. Créer une fonction `RLE_encode` de déclaration :

```
void RLE_encode(
       byte source_image[], int w, int h,
       int compression[], int &comp_size );
```

   L'image à compresser est donnée par sa largeur w, sa hauteur h et par un tableau *source_image*. Le résultat de la compression est stocké dans le tableau *compression* et le nombre d'éléments utilisés dans ce tableau *comp_size*. (pour rappel, le tableau prévu pour recevoir l'image comprimée est déclarée de manière statique dans *main* et $w \times h + 1$ n'est qu'un majorant de la taille de l'image comprimée.)
8. Créer une fonction `RLE_decode` de déclaration :

```
void RLE_decode( int compression[], int comp_size,
       byte decomp_image[] );
```

   L'image à décompresser est donnée par un tableau *compression* et la taille des données dans ce tableau *comp_size*. Le resultat de la décompression est stocké dans le tableau *decomp_image*.

9. Dans la fonction `main` et à l'aide des fonctions précédemment définies :
   (a) Remplir dans le tableau `image_source` deux rectangles de votre choix.
   (b) Afficher cette image
   (c) Encoder *image_source* dans *image_encodee*
   (d) Decoder *image_encodee* dans *image_decodee*
   (e) A l'aide d'une boucle, vérifier que les deux images *image_source* et *image_decodee* sont identiques.
   (f) Afficher l'image décodée et valider la vérification précédente de manière visuelle.
10. Créer une fonction *remplir_diagonale* de déclaration :

    ```
    void remplir_diagonale( byte image[], int w, int h );
    ```

    Cette fonction met tous les pixels (i, j) tels que $i == j$ de *image* à 255.
11. Tester la fonction *remplir_diagonale* de la même façon que la fonction *remplir_rectangle* a été testée.
12. On fixe $w = 256$ et $h = 128$
    (a) Remplir l'image source de zéros.
    (b) Remplir dans l'image source un rectangle dont le coin supérieur gauche est en $(20, 20)$ de dimensions $(80, 80)$
    (c) Remplir dans l'image source un second rectangle dont le coin supérieur gauche est en $(120, 10)$ de dimensions $(100, 100)$
    (d) Compresser l'image source et afficher la taille de l'image compressée *comp_size*
    (e) Remplir à nouveau l'image source de zéros.
    (f) Remplir dans l'image source la diagonale.
    (g) Compresser l'image source et afficher la taille de l'image compressée *comp_size*
13. On fixe $w = 1024$ et $h = 1024$
    (a) Un 'plantage' se produit.
    (b) Identifier le problème et modifier le programme pour qu'il fonctionne avec les valeurs de w et h données.