MAP401-Projet logiciel

Le but de ce projet est de nous familiariser avec la manipulation des images et à apprendre à bien gérer son temps et ses tâches à l'aide d' outils comme le diagramme de Gantt et le journal de bord. On va garder l'image lettre-L-cursive tout au long de notre rapport pour montrer comment l'image change avec l'exécution de chaque tâche. On va aussi garder notre variable d(distance seuil) à 12.

Tâche 1 - Image Bitmap

Paquetage image(image.h, image.c)

image.h

Dépendances: types_macros.h
Structures:
Pixel: Peut être soit BLANC(0) ou NOIR(1)
Image: consiste de L et H (Largeur et hauteur de l'image respectivement) et un tableau de pixels.

image.c

Dépendances: image.h
La fonction **ecrire_image(Image I)**

- parcours
- utilise la fonction get_pixel_image afin de stocker chaque pixel dans la variable p.
- successivement affiche toutes les valeurs de p à chaque itération.

La fonction **negatif_image(Image I)**

- utilise la fonction *creer_image* afin de créer une image(inew) de dimension L*H où tous les pixels sont initialement initialisés à 0.
- accède à chaque pixel de l'image passé en paramètre avec la fonction *get_pixel_image*.
- convertit les pixels noir en blanc et les pixels blancs en noirs.La nouvelle valeur du pixel convertit est stockée dans la variable *p*.
- La fonction set_pixel_image est appelée afin d'attribuer la valeur p au pixel à l'adresse (i,j) de inew.
- La fonction retourne l'image inew.

Pour tester les deux fonctions écrites, on les appelle dans le main qui se trouve dans le fichier *test_image.c*.

Tâche 2 - Géométrie 2D

Dans la tâche 2, on a défini 2 structures (dans geom2d.h) :

1. type Vector qui consiste de 2 chiffres x,y de type double.
2. type Point qui consiste aussi de 2 chiffres x,y de type double.

Pour les tâches suivantes, on modifie geom2d.h en ajoutant les structures suivantes:

3. type Segment qui consiste de deux points P1 et P2.
4. type CourbeBezier_2 qui consiste de 3 points de contrôle C0,C1,C2.
5. type CourbeBezier_3 qui consiste de 4 points de contrôle C0,C1,C2,C3.

geom2d.c (avec les fonction utilisées dans les autres tâches aussi**)**:

La fonction

- set_point(double x, double y)
  Créer un point p avec 2 paramètres de type double.
  retourne le point p
- add_point(Point P1, Point P2)
  Additionne les 2 points P1 et P2 en additionnant les p.x's et p.y's ou p est un point quelconque.
  Utilise *set_point* afin d'établir le point de retour et de point est retourné.
- substract_point(Point P1, Point P2)
  Utiliser le même principe que *add_point* pour l'opération de soustraction.
- int_division(Point A , double a)
  Définit un Point P dont les valeurs de x et y sont A.x/a et A.y/a respectivement.
  Retourne le point P
- vect_bipoint(Point A , Point B)
  Trouve le point P qui est le bipoint des deux points passés en paramètre.
  Retourne le point P.
- printPoint(Point A)
  Affiche le point P
- printVector(Vector V)
  Retourne le vecteur V.
- int_product(Point A , double a)
  Définit un Point P dont les valeurs de x et y sont A.x*a et A.y*a respectivement.
  Retourne le point P
- scalar_product(Vector A, Vector B)
  Retourne le produit scalaire de deux vecteurs.
- euclidean_norm(Vector A)
  Retourne la norme euclidienne du vecteur A.
- distance_points(Point A, Point B)
  Retourne la distance entre les deux points A et B.
- find_line(Vector A, Vector B)

Affiche l'équation de la ligne passant à travers les deux vecteurs A et B

- comp_Points(Point A, Point B)
  Retourne 1 si les deux points sont égaux.
  Sinon retourne 0.

Tâche 3 - Extraction d'un contour externe

La partie 1 de la tâche 3 consiste à déterminer le contour d'une image.
La partie 2 consiste à écrire le contour dans un fichier.

Paquetage robot (robot.h, robot.c)

robot.h

Dépendance: *image.h*
Structures et types:

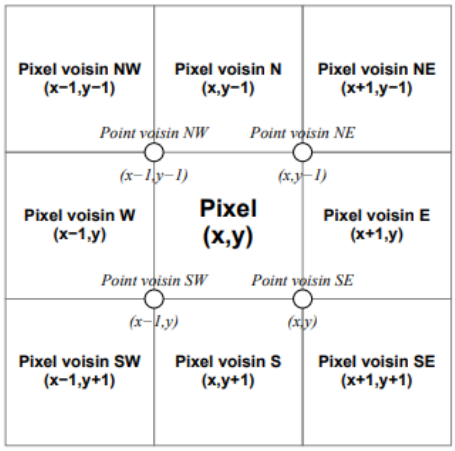
- Point : consiste des deux doubles x et y.
- Pixel_point : consiste d'un point p et d'une pixel color.
- Orientation: Peut être North, East, South, West
- Robot: a un point p et une orientation o

robot.c

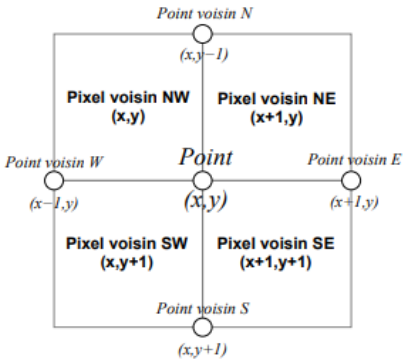
Dépendance: *robot.h*
Ce fichier .c contient toutes les fonctions pour définir un robot qui va nous permettre d' identifier le contour d'une image.

- **initialize_robot** permet de placer le robot en position (x,y) et en orientation o.
- **move(Robot *r)** gère les différentes façons pour que le robot puisse avancer d'une case. Chaque cas de *o* est géré, comme indiqué dans cette figure du poly.

- **void turn_left(Robot *r)** fait tourner le robot à gauche, c'est-à-dire que cette fonction change l'orientation courant du robot de telle façon que le robot ait tourner à gauche par rapport à sa position et orientation initiale.
- **void turn_right(Robot *r)** utilise le même principe que la fonction précédente sauf que maintenant on gère la situation où le robot tourne à droite.
- **void position(Robot *r, Point *p)** stocke la position de la case courante du robot dans le point p.
- **void value_of_pixel(Image I, Robot *r, Pixel *p_r, Pixel *p_l, Point p)** gère les différents cas de *o* afin d'avoir la valeur du pixel de la case qui se trouve à gauche (p_l) et à droite(p_r) de la case courante du robot.



Voisins d'un pixel $(\mathbf{x}, \mathbf{y})$



Voisins d'un point (x, y)

- **Orientation new_orientation(Point p, Image I, Robot *r).** On appelle la fonction *value_of_pixel* afin d'obtenir les valeurs des pixels p_r et p_l. Si le pixel gauche est noir, le robot tourne à gauche (*turn_left(r)*) sinon si le pixel droite est blanc le robot tourne à droit(*turn_right(r)*). La nouvelle orientation du robot est retournée.

Paquetage contour (contour.h, contour.c)

Les fonctions pour trouver le contour de l'image

contour.h

Dépendance: robot.h, image.h, sequence_point.h

contour.c

- **Point find_initial_pixel(Image I):** Parcourt l'image jusqu'à trouver le pixel initial qui est définit comme étant le premier pixel noir avec un pixel blanc à son Nord. Utilise la fonction *get_pixel_image* afin de retenir la couleur du pixel courant p2 et du pixel p1 qui soit à son Nord. Ainsi en dehors de la boucle on retient la valeur de p1 comme étant initialement la valeur du pixel à l'indice (0,0) qui se trouve juste en haut de l'image. Ainsi, dans la boucle les valeurs de p1 et p2 changent jusqu'à trouver le pixel qui répond aux conditions indiquées. Ce point p est ensuite retourné.
- **void find_contour(Image I, FILE *file, FILE *file_eps)**
  - Récupère le point du pixel initial de l'image I (*find_initial_pixel(I)*)
  - Le robot est initialisé à cette position avec l'orientation East(*Initialize_Robot)*
  - Puis on définit une liste contour de type List_point qui va cotonnier tous les points du contour. La première position est ajoutée à cette liste.
  - Puis on initialise un flag à True qui devient faux que quand la position du robot est de nouveau le point initial.
  - Le robot avance d'une case et on ajoute ce point à la liste contour et on appelle la fonction *new_orientation(position, I,&r)* afin de déterminer la nouvelle orientation du robot pour suivre le contour de l'image.

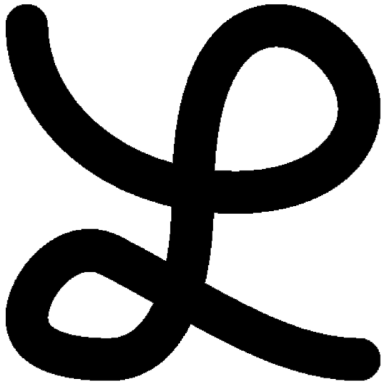
- On affiche le nombre total de segments qui est contour.size-1.
- On écrit ensuite les points du contour dans un fichier avec la fonction *write_contour_in_file(contour,file).*

Tâche 4- Sortie au format PostScript encapsulé

Dans le paquetage *sequence.c*:
Afin de convertir cette liste de contour en un fichier .eps on a écrit la fonction *convert_to_eps* qui prend comme paramètres: List_Point L, FILE *fileout, UINT lenght et UINT Height.
Les deux premières lignes du ficher sont %!PS−Adobe−3.0 EPSF−3.0 et %%BoundingBox: xmin -ymin xmax ymax. Puis le premier point du contour suivi de l'instruction moveto Puis on parcourt tous les éléments de la liste et les prochaines lignes contiennent respectivement les valeurs de x et y de chaque point suivis l'instruction lineto. Quand on arrive à la fin de la liste, on sort de la boucle et on vérifie si les dernières valeurs de x et y correspondent au point initial. Si oui, on utilise l'option fill (remplissage) sinon on utilise l'option stroke(pour la tracé du contour).

Tâche 5- Extraction des contours d'une image

Exemple d'exécution:



Le nombre total de segments des contours de l'image initiale: 4228

image.c

- On crée la fonction *Image image_mask(Image I).* On initialise une nouvelle image inew de même dimension que l'image I et on la stocke dans la variable inew. On initialise tous les pixels de inew à 0 et on retourne cette image.

Paquetage sequence_point

sequence_point.h

Dépendance:
Nouvelles Structures:

- **Node_Multi_Contour**: Contient une variable *data* du type *Contour* et *next_contour de type *Node_Multi_Contour*.

- **List_Multiple_Contour:** Contient une variable count de type *integer*, *head de type *List_Multiple_Contour* et *tail de type *List_Multiple_Contour*.
  (le head est ici la tête de la liste et tail la queue de la liste)

sequence_point.c

On introduit les nouvelles fonctions suivantes dans *sequence.c*.

- **List_Multiple_Contours create_List_Multi_empty()** crée une liste L vide de type *List_Multiple_Contours* en initialisant l'entier *contour* de la liste à 0 et le pointeur *head* à NULL.
- **List_Multiple_Contours add_next_Contour(List_Multiple_Contours *List, Contour c)**
  - Crée un noeud *node* de type *Node_Multi_Contour .*
  - Alloue le contour c à node->data.
  - Si la liste *List* est vide, node devient l'unique élément de *List* et ainsi aussi le *head* de List.
  - Sinon, on ajoute la nouvelle contour (contenue dans node) à la fin de List et cette dernière devient le *tail* de List.
  - On retourne *List.

contour.c

Dépendance: *contour.h*

- **Image create_image_mask(Image I, int *count)**
  - créer l'image M en appelant la fonction *image_mask(I)*,
  - On définit ensuite deux Pixel_Point p1 et p2.
  - On parcourt l'image afin d'avoir la couleur du pixel courant et celle de son pixel voisin N. Si celui-ci est BLANC et que la couleur de p1 soit NOIR on marque le pixel à l'indice (x,y) de l'image M comme NOIR et on incremente le count passé en paramètre.
  - L'image masque M est retourné.
- On modifie la fonction **find_contour** (afin de trouver plusieurs contours dans la même image) tel qu'elle prend maintenant comme paramètres: Image I, Image *M, int *countToWhites , Point initial_pixel, List_Multiple_Contours* multi_contours.
  - Le début de la fonction est la même que celle de la tâche précédente.
  - Ce qui change dans cette fonction est le pixel qu'on change dans l'image masque.
  - Si l'orientation du robot est East, on alloue la couleur BLANC au pixel NE du pixel courant et on incrémente countToWhites.
  - Puis on fait avancer le robot et le point position est maintenant la position du robot.
  - On ajoute ce point à notre liste contour et on récupère l'orientation du robot.
  - Si cette orientation est East et que le robot soit de nouveau à son point de départ, on sort de la boucle.

- On stocke le nombre de segments du contour dans countsegmets et on ajoute ce contour à la liste multi_contours.

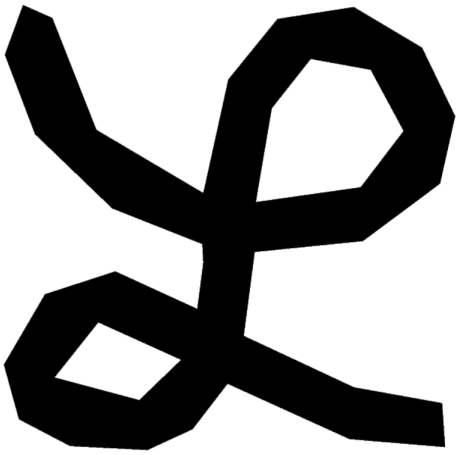
  Difficultés: Gérer les différentes structures dans la fonction.

- **void find_multiple_contour(Image I, FILE *file, FILE *file_eps)**
  - Initialisation des entiers *countBlacks, countToWhites, countContour,allSegments, currSegmentsCount et totalPoints* à 0.
  - Création d'une liste multi_contours de type *List_Multiple_Contours* et de l'image masque M.
  - On parcourt l'image I.
  - On vérifie la couleur du pixel courant. S'il est NOIR, on définit ce le point du pixel comme p et on appelle la fonction *find_contour* pour trouver le contour avec point initial p.
  - On incrémente la valeur de countContours et on ajoute le nombre de segment du contour courant à la valeur allSegments.
  - Si le nombre countToWhites est égale à countBlacks(qu'on obtient avec la fonction *create_image_mask,* on passe au calcul du prochain contour.
  - Ensuite on affiche le nombre total de segments et le nombre de contours qu'on obtient de *multi_contours.count*. On appelle ensuite la fonction *write_in_files* pour écrire les contours dans un fichier et le convertir en fichier .eps.
- **void write_in_files(Image I, List_Multiple_Contours multi_contours, FILE* file, FILE* file_eps)**
  - Comme les fonctions décrites dans les tâches précédentes, on écrit chaque contour dans un fichier .contour et on convertit celui-là en un fichier .eps.

Pour tester la tâche 5, on écrit le paquetage test_robot ou on lit le fichier qu'on teste et on appelle la fonction *find_multiple_contours*.

Tâche 6- Simplification de contour par segment

Le but de la tâche 6 est de simplifier la séquence de contours suivant une distance-seuil d.



Le nombre total de segments des contours de la simplification avec d = 12.0 : 41
Exécution avec d=12

Paquetage geom2d

geom2d.c

Ecriture de la fonction:

- distance_point_to_line(Point P, Segment S)
  - compare les deux points P1 et P2 du segment S avec *comp_Points.* S'ils sont égaux, on calcule la distance entre le point P et le point du segment avec la fonction *distance_points*.
  - Sinon on calcule la valeur x (comme indiqué dans le poly) en faisant appel aux fonctions *scalar_product* et *vect_bipoint.*
  - On a trois cas de x à gérer:
    - x<0 : distance entre le point S.P1 et P
    - x>1 : distance entre le point S.P2 et P
    - sinon on calcule le point Q=`add_point(int_product(substract_point(S.P1,S.P2),x),S.P1)` et la distance entre les points Q et P.
  - Retourne la distance.

Paquetage contour

contour.c

- On modifie la fonction ***find_multiple_contour*** pour qu'elle retourne la liste multi_contours.
- Ecriture de la fonction `List_Segments simplification_douglas_peucker(Tableau_Point cont,int j1,int j2,double distance)` comme décrite dans le poly. (On utilise les fonctions décrites dans les tâches précédentes).

- **write_List_Segments_to_eps(FILE *file_eps,List_segments L)** écrit une liste L de segments dans le fichier file_eps, avec le même principe décrit dans les tâches précédentes.
- **write_all_simple_segments_eps(Image I,List_Multiple_Contours contours,FILE *file_simple_eps_1,double d)** utilise la fonction *simplification_douglas_peucker* afin de simplifier la liste de contours. Puis on écrit cette liste de segments simplifiés avec la fonction, *write_List_Segments_to_eps* dans un fichier eps (`_simple1.ps`)

test_robot.c

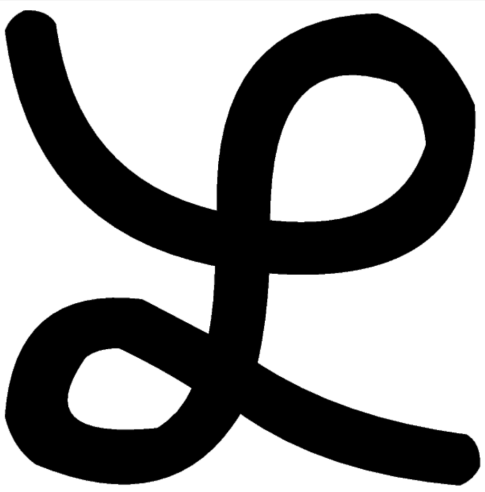
Dépendances:robot.h, image.h,contour.h

On teste toutes les fonctions écrites en appelant d'abord la fonction find_multiple_contours (On stocke le retour de cette fonction dans une variable contour de type List_Multi_Contours) puis la fonction *write_List_Segments_to_eps* avec ce même contour comme paramètre.

Les difficultés de cette tâche: La récursion dans *simplification_douglas_peucker* (Il faut creer une liste L1 et L2 vide à chaque récursion)

Tâche 7-Simplification de contours par Bézier

Partie 1 - Simplification par courbes de Bézier de degré 2



Nombre de courbe de bézier est: 25

Paquetage courbe_bezier

courbe_bezier.h

Dépendance: robot.h, geom2d.h, contour.h
Structures définies:

- ***CourbeBezier_2_Node*** : consiste d'une **courbe** de type *CourbeBezier2* et ***next_courbe** de type *CourbeBezier_2_Node*
- ***List_CourbeBezier_2:*** consiste d'un entier **count** et d'un **head** et **tail** de type *CourbeBezier_2_Node.*
- **CourbeBezier_3_Node:** consiste d'une **courbe** de type *CourbeBezier3* et ***next_courbe** de type *CourbeBezier_3_Node*
- ***List_CourbeBezier_3:*** consiste d'un entier **count** et d'un **head** et **tail** de type *CourbeBezier_3_Node.*

courbe_bezier.c

Les fonctions suivantes permettent la simplification de contours à l'aide de courbes de Bézier de degré 2.

- ***CourbeBezier_2_Node *create_element_Courbe_Bezier_2(CourbeBezier_2 c):*** Créé un nœud de type *CourbeBezier_2_Node.*
- ***List_CourbeBezier_2 create_List_Courbe_Bezier_2_vide()*** crée une liste vide de type List_CourbeBezier_2 en initialisant le count à 0 et le head et le tail à NULL.
- ***List_CourbeBezier_2 add_element_Courbe_Bezier_2(List_CourbeBezier_2* L, CourbeBezier_2 e)*** ajoute le CourbeBezier_2 e à la liste L.
- ***List_CourbeBezier_2 concatener_List_Courbe_Bezier_2(List_CourbeBezier_2 L1, List_CourbeBezier_2 L2)*** concatène les 2 listes de Bezier2 passées en paramètres.
- ***Point calculate_C1_bezier(Tableau_Point*cont,int j1,int j2,int n)*** calcule le point de contrôle C1 obtenue en simplifiant une séquence de points *cont* par une courbe de bézier de degré 2 comme décrite dans cette partie du poly:

  A) Cas $n = 1$ (contour $\{P_{j1}, \ldots, P_{j2}\} = \{P_{j1}, P_{j1+1}\}$ réduit à deux points, soit un seul segment) :

  $$C_1 = \frac{1}{2}(P_{j1} + P_{j2})$$

  B) Cas $n \geq 2$ (contour $\{P_{j1}, \ldots, P_{j2}\}$ avec au moins 3 points) :

  $$C_1 = \alpha \left(\sum_{i=1}^{n-1} P_{i+j1} \right) + \beta(P_{j1} + P_{j2}) \quad \text{avec} \quad \alpha = \frac{3n}{n^2 - 1} \quad \text{et} \quad \beta = \frac{1 - 2n}{2(n+1)}$$

- ***Point calculate_C_t(CourbeBezier_2 B, double t)*** calcule un point c_t quelconque du courbe de bézier avec l'équation :

  $$\left\{ C(t) = C_0\,(1-t)^2 + C_1\,2\,t\,(1-t) + C_2\,t^2,\ t \in [0,1] \right\}$$

- ***double distance_point_Bezier(Point Pj,CourbeBezier_2 courbe,double ti)*** calcule la distance entre un point Pj et la courbe de Bézier2. On appelle la fonction *calculate_C_t* pour trouver le point C5TI) sur la courbe et on calcule la distance entre ce point et Pj avec la fonction *distance_points.*
- ***List_CourbeBezier_2 simplification_douglas_peucker_bezier2(Tableau_Point cont,int j1,int j2,double d)*** simplifie la partie du contour cont compris entre les indices j1 et j2 avec la distance-seuil d et retourne une liste L de courbes de bezier2. On utilise les fonctions définies précédemment pour implémenter cet algorithme comme décrit dans le poly.
  Difficulté: l'écriture de distance_point_Bézier et quoi choisir comme j1 et j2.

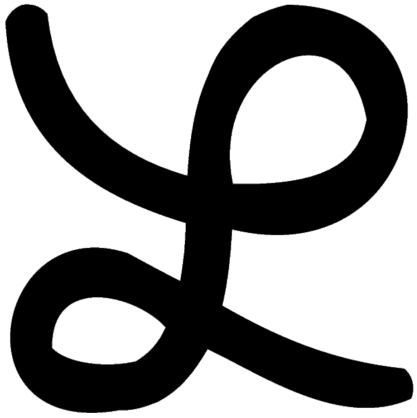
- ***CourbeBezier_3 courbeBezierFrom2to3(CourbeBezier_2 bezier2)*** convertit une courbe de bézier de degré 2 en 3 en utilisant les fonctions définies dans geom2d avec les équations suivantes:

$$\bar{C}_0 = C_0 \qquad \bar{C}_1 = (C_0 + 2C_1)/3 \qquad \bar{C}_2 = (2C_1 + C_2)/3 \qquad \bar{C}_3 = C_2$$

- ***List_CourbeBezier_3 degreeElevationfrom2To3(List_CourbeBezier_2 *list)*** convertit tous les éléments de la liste de courbe de béziers 2 en des courbes de béziers 3 et retourne cette nouvelle liste.
- ***void write_curve_bezier_in_eps(FILE* fileOut,List_CourbeBezier_3 *L,Image I)*** écrit la liste de courbe de béziers de degré 3 dans un fichier eps.
- ***void write_all_courbe_bezier_eps(FILE* f,Image I, List_Multiple_Contours contours,int d)*** écrit toutes les courbes de béziers 3 obtenues en simplifiant tous les éléments de la *list_Multiple_Contours* **contours** avec la fonction *simplification_douglas_peucker_bezier2,* et en élevant le degré de toutes les béziers 2 obtenues avec *degreeElevationfrom2To3* et en écrivant chaque List_CourbeBezier3 obtenu dans le fichier eps.

On teste les fonctions ci-dessus avec test_bezier.c.Le fichier eps produite est de la forme *nomdufichierentréé*.`_curve.ps`.

Partie 2 - Simplification par courbes de Bézier de degré 3



Nombre de courbe de bezier est: 22

- Même principe que pour la partie 1 sauf qu'on travaille maintenant avec les courbes de béziers de degré 3.
- ***double alpha(double n), double beta(double n), double lambda(double n),double gamma(double n)*** sont calculé ainsi ***:***

$$\alpha = \frac{-15\,n^3 + 5\,n^2 + 2\,n + 4}{3(n+2)(3\,n^2+1)} \qquad \beta = \frac{10\,n^3 - 15\,n^2 + n + 2}{3(n+2)(3\,n^2+1)} \qquad \lambda = \frac{70\,n}{3(n^2-1)(n^2-4)(3\,n^2+1)}$$

$$\gamma(k) = 6\,k^4 - 8\,n\,k^3 + 6\,k^2 - 4\,n\,k + n^4 - n^2$$

- ***Point sum_points(Tableau_Point *cont,int n,int j1,bool point)***

- ***Point calculate_Ct_3(CourbeBezier_3 B, double t), double distance_point_bezier3(Point Pj, CourbeBezier_3 B, double ti)*** opère de la meme façon que dans la partie 1.
- ***CourbeBezier_3 approx_bezier3(Tableau_Point *cont, int j1, int j2)*** calcule le point C0,C1,C2 et C3 de la courbe.
- De même pour la fonction
  List_CourbeBezier_3 simplification_douglas_peucker_bezier3(Tableau_Point cont,int j1,int j2,double d)

  On teste les fonctions pour les courbes de béziers 3 avec *test_bezier3.c.*
  Dépendances: contour.h, image.h,robot.h , courbe_bezier.h
  Le fichier eps produite est de la forme *nomdufichierentréé.*`_curve3.ps.`

Tâche 8- Tests de robustesse et performance Comparatif des simplifications

On observe les temps d'exécution de chaque programme avec plusieurs images.
Ci-dessus est un exemple pour nos résultats avec l'image lettre-L-cursive.pbm.

multiple contour	0.03
simplification par segment	0.03
Bezier 2	0.02
Bezier 3	0.04

On observe que plus une image a de contours initialement, plus grand est son temps d'exécution. Par exemple Manara.pbm a 1412 contours et a ainsi un plus grand temps d'exécution que lettre-L-cursive.pbm qui a seulement 3 contours.

Remark:

Vous pouvez trouvez un manuel utilisateur sous format d'une fichier README.md dans le code source