Rapport de stage :
Méthodes algébriques pour le calcul de l'espace de configurations d'un robot parallèle.

Matthieu Finiasz

Stage du 15 septembre au 18 décembre 1998

Table des matières

1 Le projet SAGA

Le projet SAGA (Systèmes Algébriques, Géométrie et Applications) est un petit pré-projet (qui doit encore être accepté par une commision pour obtenir le statut de projet) de l'INRIA de Sophia-Antipolis. Il se compose essentiellement de deux chercheurs Jean-Pierre Merlet et Bernard Mourrain (les autres n'étaient pas présent au moment de mon stage), qui m'ont encadré et guidé tout au long de mes recherches. Comme dans tous les projets de l'INRIA il y a aussi beaucoup de doctorants, ici ils étaient trois : David Daney, Didier Bondyfalat et Laurent Busé. J'ai à plusieurs reprises travaillé en collaboration avec David qui travaille lui aussi sur des problèmes de résolution de systèmes de polynômes à plusieurs variables, mais lui dans le cadre de l'étalonnage de robots parallèles, et non du calcul d'espace de travail.

J'ai été en stage au sein de ce projet entre le 15 septembre et le 18 décembre 1998. Au début j'ai du commencer par lire quelques articles afin de me mettre un peu à niveau sur le plan des méthodes de résolution de systèmes polynômiaux, puis après j'ai tout de suite commencé à chercher ce que je pouvais faire pour le calcul d'espace de travails de robots parallèles. J'ai d'abord cherché dans la direction la plus naturelle : une résolution exacte du problème général, mais voyant vite que c'était illusoire de vouloir résoudre ce problème je me suis orienté vers des méthodes de résolution approchée, mais cela n'a pas réellement abouti : mes algorithmes ne marchaient pas bien, étaient lents, et avaient comme seul intérêt de faire de jolis dessins, d'autant plus qu'il existe déjà un algorithme de résolution approchée du problème général de

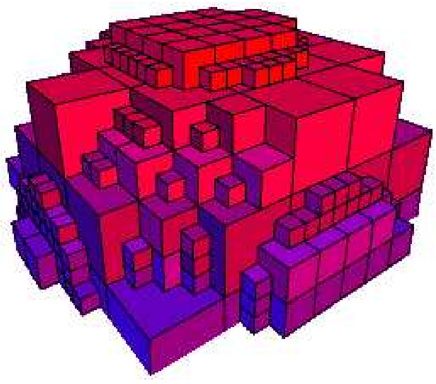

FIG. 1: *Un exemple de joli dessin*

l'espace de travail d'un robot parallèle (programmè par J.P. Merlet) et qui est bien plus efficace que tout ce que j'aurais pu faire. Je me suis donc ensuite orienté (ou plutôt Bernard Mourrain m'a orienté) vers le calcul du bord de cet espace de travail : en étudiant le cas de la dimension 2 et du robot parallèle plan je me suis rendu compte qu'il était facile d'éliminer certaines variables du problème et donc ainsi d'obtenir des résultats moins forts (on a moins d'information à la fin) mais en un temps beaucoup plus rapide. C'est l'algorithme qui découle de ces constatations que j'ai programmé et que j'ai décidé de vous présenter ici.

2 Les robots parallèles

2.1 Présentation

Un robot parallèle est un robot constitué de deux plates-formes (l'une fixe et l'autre mobile), reliées entre elles par des segments de longueurs variables et fixés à chacune des plates-formes par une liaison sphérique. Ainsi, quand on fait varier la longueur des segments la plate-forme mobile se déplace. On rencontre par exemple cette structure de robots dans les simulateurs de vol en taille réelles : la plate-forme mobile est la cabine, la plate-forme fixe est le sol, et les deux sont reliées entre elles par plusieurs vérins qui permettent de simuler les déplacements de l'avion.

Le nom de ce type de robot vient du fait que ses segments sont fixés en parallèle entre les deux plates-formes, par opposition au robot série qui n'a qu'un seul bras, constitué de plusieurs segments articulés entre eux.

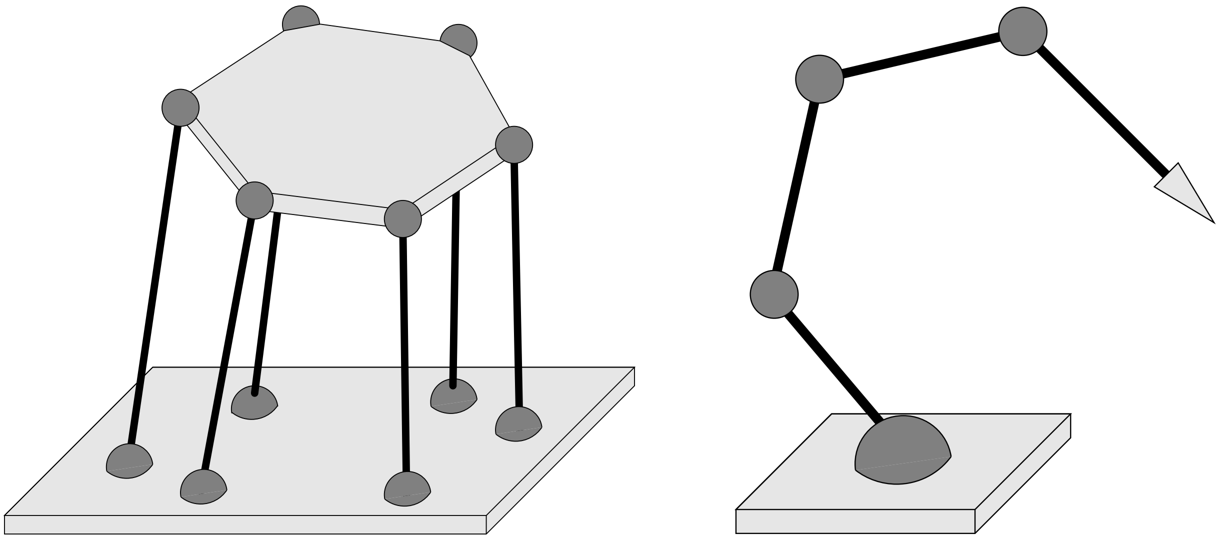

FIG. 2: *Schémas d'un robot parallèle et d'un robot série*

Ces types de robots sont tous les deux utilisés dans l'industrie et ont chacun leurs avantages : le robot série va pouvoir travailler en un point tout en contournant un obstacle et est donc beaucoup utilisé en industrie automobile, par exemple pour fixer un pare-brise en passant par le trou de la portière, le robot parallèle a lui l'avantage de pouvoir manipuler des charges plus importantes (il y a beaucoup moins d'effet de levier sur des segments en parallèle que sur des segments en série).

2.2 Les calculs ...

En revanche, un des problèmes du robot parallèle est que son utilisation demande beaucoup de calculs. En effet, pour simplement connaître la position de la plate-forme mobile en fonction de la longueur des jambes (c'est ce que l'on appelle le *modèle géométrique direct*) il faut effectuer un calcul qui n'a rien de facile. Les premières méthodes de résolution exactes de ce problème commencent tout juste à apparaître et on utilise encore le plus souvent des méthodes approchées (par optimisation).

De plus, les robots parallèles sont très sensibles à la position des points d'attache des segments : une petite variation de la position d'attache peut entraîner un grand déplacement de la plate-forme mobile. Donc, pour pouvoir utiliser le robot avec une bonne précision il faut commencer par l'étalonner[1] : recalculer avec précision les coordonnées des points d'attache (qui ne coïncident bien sur pas exactement avec celles fournies par le fabricant). Là encore les méthodes employées par les industriels (ceux qui vont utiliser le robot parallèle) ne sont pas toujours les meilleurs, et dans tous les cas sont assez compliquées. Il n'existe d'ailleurs toujours pas de méthode de résolution exacte pour le problème de l'étalonnage des robots parallèles.

Pour la plus part des calculs que l'on veut effectuer sur les robots parallèles on utilise donc encore des méthodes approchées, et pas toujours très efficaces. C'est aussi le cas pour le calcul de l'espace de configuration (ou espace de travail) du robot.

2.3 Espace de travail

L'espace de travail d'un robot parallèle est l'ensemble des configurations qu'il peut atteindre (chaque configuration est en fait un couple $(\mathcal{P}, \mathcal{R})$ où $\mathcal{P}$ est la position dans l'espace d'un point de la plate-forme mobile et $\mathcal{R}$ son orientation par rapport à un repère fixe) quand on met des bornes sur les longueurs de ses segments. En effet un vérin a une allonge maximale, et une longueur minimale (quand il est entièrement rentré dans lui-même) et cela limite donc l'ensemble des points de l'espace que la plate-forme peut atteindre. Cependant, quand un industriel veut utiliser un robot parallèle pour effectuer une tâche dans une chaîne de montage il peut être utile pour lui de savoir jusqu'où et avec quel orientation l'outil fixé sur la plate-forme mobile peut aller travailler. Il faut donc pour cela calculer l'espace de travail du robot, mais selon le contexte cet espace de travail ne sera pas le même. En effet plusieurs cas sont possibles : pour un simulateur de vol on peut chercher à connaître l'ensemble des points de l'espace que la cabine peut atteindre (avec une orientation quelconque) afin de savoir par exemple la taille minimale de la pièce dans laquelle on peut le mettre, pour un outil dans une chaîne de montage il est plus utile de connaître l'espace que l'on peut atteindre avec toutes les orientations possibles et donc l'espace où l'on peut amener les pièces sans avoir à se soucier de leur orientation.

C'est là que commencent à diverger les méthodes utilisées : selon les cas la méthode la mieux

1. Voir par exemple : **Daney D.** *Mobility constraints on the legs of a parallel robot to improve the kinematic calibration.* In *New machine concepts for handling and manufacturing devices on the basis of parallel structures*, Braunschweig, 10-11 Novembre 1998

adaptée n'est pas la même. Il existe un algorithme général (développé par Jean-Pierre Merlet) qui permet de calculer l'ensemble de toutes les configurations que peut atteindre le robot, basé sur un découpage de l'espace $(\mathcal{P}, \mathcal{R})$ en cubes de plus en plus petits. Le problème est qu'à chaque étape de l'algorithme on teste pour chaque cube s'il est entièrement à l'intérieur ou entièrement à l'extérieur de l'espace de travail, et quand il est entre les deux ou que l'on ne sait pas dire on le découpe en deux selon chaque dimension, ce qui fait à chaque fois 64 cubes (car on est dans un espace de dimension 6). Si on veut une bonne précision c'est algorithme devient donc vite très lent. Cependant, une fois que l'on connaît toutes les configurations possibles on peut connaître tous les types d'espace de travail que l'on veut, aussi bien avec un angle fixé qu'avec un angle quelconque.

Mais on n'a pas toujours besoin de calculer l'espace en dimension 6 (position et orientation de la plate-forme), et pour aller plus vite on peut chercher à éliminer des variables : les variables d'orientation par exemple.

C'est ici qu'interviennent les méthodes algébriques ...

3 Utilisation de méthodes algébriques pour l'élimination de variables

3.1 La matrice de Bezout

Commençons par regarder un exemple : quand on veut trouver les zéros communs d'un couple de polynômes en deux variables X et Y, une méthode efficace est de considérer ces deux polynômes comme des polynômes en une variable X dépendants d'un paramètre Y. On calcule alors la matrice du résultant de ces deux polynômes puis son déterminant et obtient un polynôme en Y dont l'ensemble des racines contient toutes les valeurs possibles de Y dans les couples (X, Y) solution du problème. Il suffit après de remplacer Y dans les équations du début pour avoir les solutions.

L'intérêt de cet exemple est qu'il montre bien que pour résoudre un problème de nature polynômial, le plus simple est bien souvent de commencer par éliminer des variables : en construisant la matrice du résultant on supprime la variable X et on se retrouve avec une seule variable, ce qui rend le problème beaucoup plus simple.

Cela se passe de la même façon quand on considère des polynômes en n variables : en éliminant une équation on peut éliminer une variable. Donc quand on a un système de n équations polynômiales en n inconnues il suffit de choisir une des inconnues comme paramètre pour obtenir un système de n équations à $n-1$ inconnues dont on va calculer le résultant. Il n'y a donc qu'à construire la matrice de résultant de ces n équations, puis à calculer son déterminant. Il y a alors plusieurs choix possibles : selon la forme des équations la matrice la mieux adaptées n'est pas la même. La matrice de Macaulay est celle qui ressemble le plus à la matrice du résultant de deux polynômes, mais avec n polynôme elle est vite très grande. C'est pour cela que l'on y préfère souvent la matrice de Bezout, qui a l'avantage d'être beaucoup plus compacte. Je ne donne pas ici les méthodes pour construire ces matrices mais dans les deux cas leur déterminant est un polynôme en la $n^{\text{ème}}$ variable (celle que l'on avait choisi comme paramètre au début) dont l'ensemble des racines contient encore une fois l'ensemble des solutions du problèmes, projetées sur leur $n^{\text{ème}}$ coordonnée. Mais malheureusement il n'y a qu'une inclusion de l'ensemble des solutions dans l'ensemble trouvé, et cela par ce que la construction des matrices de Macaulay ou de Bezout introduit un terme parasite (un facteur multiplicatif sur le pgcd des n polynômes) qui rajoute des solutions que l'on est obligé d'éliminer par la suite.

3.2 Un exemple d'utilisation : le modèle géométrique direct

Pour mieux se rendre compte de l'utilité de l'élimination de variable regardons ce que cela donne pour la résolution du modèle géométrique direct (la détermination de la configuration de la plate-forme mobile quand on fixe les longueurs des six segments) dans le cas particulier où deux segments sont attachés à un même point de la plate-forme mobile.

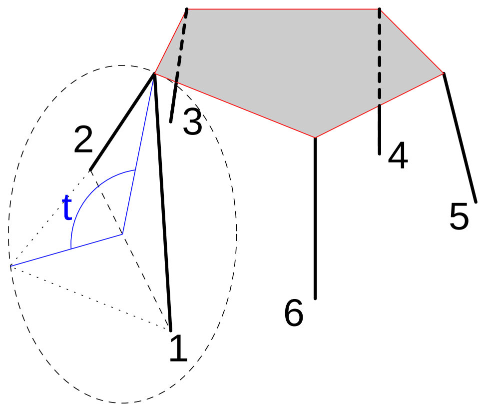

FIG. 3: *le sommet des segments 1 et 2 parcours un cercle*

Dans le cas général la résolution du problème géométrique direct revient à résoudre le système des six équations de longueur de jambe. C'est un système de six équations de degré 2 à six inconnues (si on écrit l'orientation de la plate-forme avec une matrice de rotation à trois paramètres (a, b, c)), mais dans ce cas particulier le problème est grandement simplifié : le point de la plate-forme mobile où se rejoignent les deux segments se trouve nécessairement sur un cercle (que l'on peut déterminer avec les longueurs des deux jambes) et donc on peut paramètrer la position de la plate-forme avec une seule variable t au lieu de trois. On élimine donc dès le début deux variables grâce à deux équations de longueur de jambe.

Il nous reste alors un système de quatre équations à quatre inconnues : a, b, c et t. Pour le résoudre on construit une matrice de Dixon. C'est aussi une matrice de résultant mais elle a l'avantage d'avoir des termes de plus bas degré (le calcul de son déterminant est donc plus rapide et introduit un terme parasite plus petit). En revanche elle ne peut être utilisée que si les variables à éliminer apparaissent avec un degré 2. C'est ici le cas pour a, b et c. On construit donc la matrice de Dixon et sont déterminant est un polynôme en t. Il ne reste plus qu'à trouver ses racines et à remplacer dans les quatre équations pour finir la résolution. On peut toutefois améliorer la précision numérique de cette résolution en résolvant un problème de valeurs propres généralisées (trouver les t tels que la matrice $A - tB$ ne soit pas inversible) sur la matrice de Dixon au lieu de chercher les racines de son déterminant.

4 Le calcul d'espaces de travail

L'espace de travail est l'ensemble des configurations que la plate-forme mobile peut atteindre quand ses segments ont des longueurs comprises entre deux bornes. C'est donc l'ensemble des points d'un espace de dimension six qui vérifient 12 inégalités. Mais cet ensemble est un peu compliqué à calculer. Nous allons donc essayer de simplifier le problème. Pour cela nous considérerons seulement un espace de travail en dimension 3 (la projection de l'espace de dimension six sur l'espace (X, Y, Z) par rapport aux angles a, b et c), et afin d'avoir des égalités plutôt que des inégalités nous regarderons le bord de cet espace. Voyons d'abord comment cela se passe pour un robot parallèle plan.

4.1 Espace de travail d'un robot parallèle plan

Un robot parallèle plan est constitué d'une plate-forme mobile attachée par trois segments, le tout placé dans un même plan. On peut donc paramètrer la position de cette plate-forme par trois variables (X, Y) pour la position et t pour l'orientation.

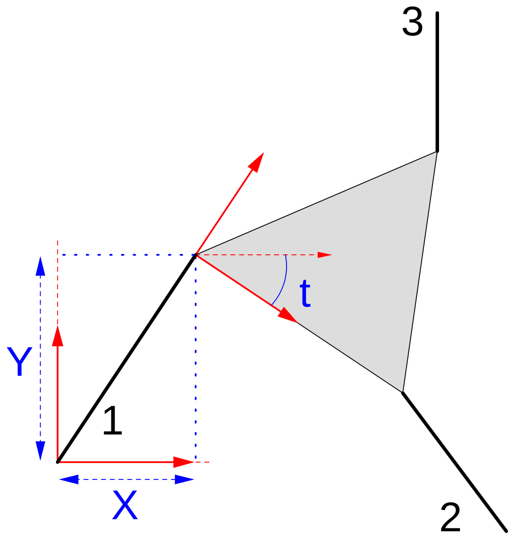

FIG. 4: *Schéma d'un robot parallèle plan*

Son espace de travail est donc la projection d'un espace de dimension 3 dans la plan. Mais quelles sont les équations qui nous donnent le bord de cet espace ? Dans l'espace de travail de dimension 3 (avant d'avoir projeté selon t) le robot est sur le bord de l'espace quand il est à l'intérieur de l'espace (toutes ses jambes ont des longueurs comprises entre les bornes) et quand une variation infinitésimale de la longueur de l'une des jambes peut l'en faire sortir : c'est donc qu'il y a au moins une jambe qui a pour longueur l'une de ses bornes.

En fait l'espace de travail a plus ou moins la forme d'un cube déformé, dont les faces sont les points de l'espace où une jambe est à sa longueur maximum, les arêtes les points où deux jambes sont au maximum, et les sommets les points où les trois jambes sont au maximum.

Mais à quelle condition un point du bord de cet espace de dimension 3 va-t-il se retrouver sur le bord de l'espace projeté ? Pour mieux voir cela regardons un dessin.

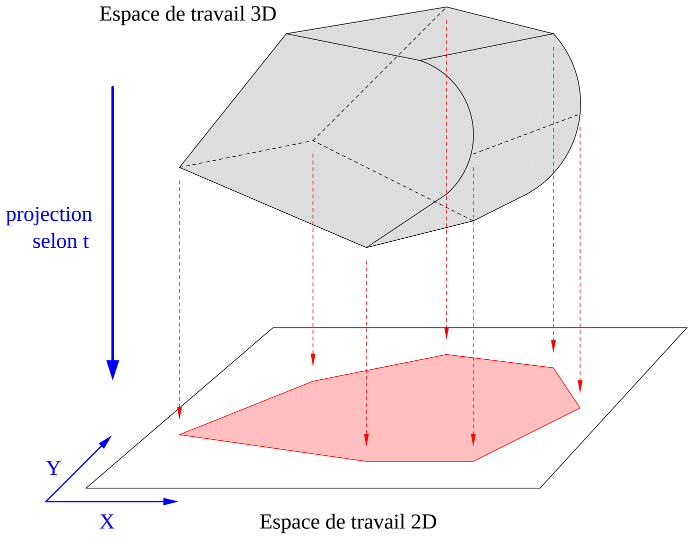

Fig. 5: *La projection de l'espace de dimension 3 par rapport à t*

On remarque facilement qu'il y a deux façons d'être sur le bord de l'espace projeté : soit on se trouve sur une arête du "cube", soit on se trouve sur une face du cube, en un point où l'espace tangent contient la direction t.

Le premier cas correspond aux points où le robot a deux jambes à des longueurs maximales (ou minimales), le deuxième aux points où une jambes est au maximum et où la dérivée de sa longueur par rapport à t (avec X et Y fixés) est nulle. Ce sont en fait les points où la plate-forme du robot est alignée avec le segment qui est au maximum.

Dans les deux cas on obtient deux équations en X, Y et t dans lesquelles il est facile d'éliminer t pour obtenir une seule équation en X et Y. En fait on obtient 12 sextiques et 12 cercles : une sextique pour chaque couple de jambes et chaque longueur (maximale ou minimale), et deux cercles pour chaque jambe et chaque longueur (car la plate-forme peut être alignée avec la jambe dans les deux sens). Ces sextiques et ces cercles découpent le plan en plusieurs morceaux qui sont chacun soit entièrement à l'intérieur de l'espace de travail, soit entièrement à l'extérieur. Il suffit donc de tester un point de chaque morceau pour déterminer de quel côté de la bordure il se trouve. Cela permet ensuite de reconstituer l'espace de travail, simplement en recollant les morceaux.

Voila donc comment on peut calculer de manière relativement simple et en un temps assez court l'espace de travail d'un robot parallèle plan (ou plutôt sa projection sur le plan). Cela se passe de façon similaire pour un robot qui n'est pas plan, mais avec des calculs un peu plus longs. C'est ce que nous allons voir.

4.2 Le cas de la plate-forme de Gough

4.2.1 Description

Dans le cas de la plate-forme de Gough la position du robot est représentée par le couple position/orientation $(\mathcal{P}, \mathcal{R})$: il faut donc que l'on projette son espace de travail de dimension 6 sur l'espace des positions $\mathcal{P}$ (qui est de dimension 3). Là encore on va chercher à trouver le bord de cet espace de dimension 3, et pour cela nous allons regarder de quoi il peut se composer.

Etre sur le bord de l'espace de travail de dimension 3 c'est en fait être en un point où l'*espace tangent*[2] à cet espace est de dimension inférieure ou égale à deux (on va en fait uniquement chercher les points auxquels l'espace tangent est de dimension exactement deux car cela suffit à avoir tout le bord, simplement en prolongent). Or la projection sur l'espace des positions conserve la tangence des espace : l'espace tangent en l'image d'un point est l'image de l'espace tangent en ce point. Pour trouver le bord de l'espace de travail en dimension 3 il faut donc trouver tous les points de l'espace de travail de dimension 6 dont l'espace tangent se projette en un espace de dimension deux.

Pour cela il y a quatre possibilités différentes :

- soit l'espace tangent est déjà de dimension deux
- soit l'espace est de dimension trois et la projection lui enlève une dimension
- soit il est de dimension quatre et il en perd deux
- soit il est de dimension cinq et il en perd trois

4.2.2 Résolution

Nous allons maintenant chercher à savoir comment nous pouvons trouver de tels points ...

- Le premier cas correspond aux points qui se situent sur la zone où quatre segments du robot sont à des longueurs extrêmes.

2. avec la convention que pour un cube, par exemple, l'espace tangent à un point d'une arête est la droite support de cette arête, celui tangent à un point d'une face est le plan qui contient cette face, et l'espace tangent à un point de l'intérieur est l'espace entier

- Le deuxième, aux points où trois segments ont des longueurs extrêmes, et où la matrice des dérivées partielles des longueurs de ces trois segments par rapport aux variables d'orientation (a, b et c) a un déterminant nul.
- Le troisième, aux points où deux segments ont des longueurs extrêmes et deux mineurs de la matrice des dérivées partielles sont nuls.
- Le quatrième, aux points où un segment est à une longueur extrême, et les trois dérivées partielles sont nulles. Ceci correspond en fait (comme en dimension 2) aux points où la plate-forme mobile du robot est alignée avec le segment : ce sont donc quatre sphères pour chaque segment.

Pour trouver ces ensembles de points il suffit donc de résoudre à chaque fois un système de quatre équations en 6 inconnues : X, Y, Z pour l'espace des positions et a, b, c pour les orientations. Plus précisément il faut trouver la projection sur $\mathcal{P}$ de ces solutions. Pour cela on va donc résoudre le système d'équations en éliminant les variables d'orientation. Comme on a à chaque fois quatre équations cela va juste bien pour pouvoir éliminer les trois variables a, b et c en calculant le bezoutien (déterminant de la matrice de Bezout) de ces quatre équations. C'est parmi les zéros de ce bezoutien qu'il faut chercher les bords de l'espace de travail du robot.

S'il n'y avait qu'un seul système d'équations le problème serait alors résolu : il suffirait de trouver quelle moitié de l'espace est celle qui correspond à l'espace de travail recherché. Cependant on obtient en fait, comme dans le cas bidimensionnel, un système pour chaque combinaison de segments et pour chacun des quatre cas. Dans chacun des quatre cas il faut donc choisir n jambes parmi 6, et pour chaque série de n jambes il y a 2^n combinaisons entre longueur minimale et maximale possibles. On trouve donc :

- 1 segment $\Rightarrow C_6^1 \times 2 = 12$
- 2 segment $\Rightarrow C_6^2 \times 2^2 = 60$
- 3 segment $\Rightarrow C_6^3 \times 2^3 = 160$
- 4 segment $\Rightarrow C_6^4 \times 2^4 = 240$

Soit un total de 472 bezoutiens à calculer dont il faut ensuite calculer les zéros pour enfin sélectionner les bons morceaux de chaque surface ...

4.2.3 Implémentation

L'implémentation se passe en plusieurs étapes : il faut tout d'abord calculer les matrices de Bezout, puis ensuite calculer les zéros de leurs déterminants, et enfin sélectionner les zones de l'espace qui sont à l'interieur de l'espace de travail.

Pour la première étape j'ai utilisé Maple pour générer du code C. Le calcul des 472 matrices (en fait on n'en calcule que 460 car le cas où un seul segment est à un extrème correspond en fait à une sphère) et la génération du code C prend environ 2 jours (sur une sparc 20). C'est beaucoup plus long que si j'avais utilisé un programme entièrement en C (d'autant plus que la librairie maple pour générer le C est très lente), mais au moins c'est rapide à programmer.

Il y a alors deux méthodes possibles. L'une consiste à faire un découpage de l'espace selon X et Y le long d'un quadrillage et de chercher les zéros des bezoutiens sur des droites verticales. L'autre consiste a découper l'espace en petits cubes et pour chaque cube regarder si l'un des bezoutiens à un zéro dans ce cube, et si oui de le redécouper en 8 cubes plus petits. Dans chacun de ses cas le code C généré à la première étape doit être different.

Pour la première méthode on commence donc par fixer des valeurs pour X et Y. Les matrices de Bezout que nous avons calculé ont alors pour coefficients des polynômes à une seule variable Z. On pourrait alors se contenter de calculer les déterminants des matrices, puis de trouver leurs zéros par une méthode classique, mais cela est assez lent et numériquement moins précis que la méthode de calcul de valeurs propres généralisées (implémentée dans la librairie ALP) qui permet de faire la même chose. La résolution de cette étape se fait donc en C++ (le langage utilisé pour ALP) et est en fait assez rapide (quelques heures pour les 400 matrices).

La deuxième méthode (avec des découpages récursifs en cubes) n'a pas été implémentée et nécessite l'utilisation d'arithmétique d'intervalles (pas encore implémentée dans ALP à l'époque du stage) pour ses calculs. Le principe est en fait de fixer des bornes sur les valeurs de X, Y et Z (chaque valeur reste dans un intervalle) et de regarder si les determinants des matrices de Bezout prennent pour valeurs des intervalles qui contiennent zéros. Si aucun intervalle (il y en a un pour chacune des 400 matrices) ne contient zéro c'est que le cube est entièrement à l'interieur (ou à l'exterieur) de l'espace de travail, sinon on redécoupe le cube jusqu'à avoir des cubes suffisamment petits (de la taille de la précision qui est recherchée).

Ensuite, il ne reste plus qu'à tester un point de chaque sous-espace (soit un petit segment vertical, soit un petit cube) qui est soit entièrement dedans soit entièrement dehors et de reconstituer un espace avec ceux qui sont dedans.

5 Conclusion

L'utilisation de méthodes algébriques nous permet donc de suffisamment simplifier certains problèmes pour pouvoir les résoudre (au moins de manière partielle) en des temps raisonnables et avec une précision acceptable. Dans notre cas la résolution prend un temps assez important (il y a deux jours de calculs avec Maple), mais après cela on peut obtenir une précision aussi bonne que l'on veut (il suffit de prendre des découpages plus fins) en un temps assez rapide.

Cependant il y a encore beaucoup d'autres problèmes de même nature (basés sur la résolution de systèmes polynômiaux) qui n'ont pas été résolus (par exemple en chimie : l'optimisation de la forme de certaines molécules dans le but d'augmenter leur efficacité). Le développement de la librairie ALP (http://www-sop.inria.fr/saga/Bernard.Mourrain/ALP/) devrait donc contribuer à la réalisation de ces algorithmes, en permettant de les programmer plus vite, et avec des méthodes plus efficaces.

A Code Maple pour la génération des matrices

```
with(linalg):
libname:="/users/97/maths/finiasz/maple/",libname:
# On change libname pour utiliser une librairie C améliorée.
readlib(C):
read("genBezoutMat1"): # Ce sont des formes générales de la matrice de Bezout
read("genBezoutMat2"): # pour un bon nombre de polynômes.

#= Procedures ===============================================================

Beq:=proc(liste)
local i,j,v,eq:
j:=0:
for i from 1 to nops(liste) do
  j:=j+1:
  eq[j]:=simplify(evalm(([X,Y,Z]+R&*y[liste[i][1]]-x[liste[i][1]])&*
   ([X,Y,Z]+R&*y[liste[i][1]]-x[liste[i][1]])-liste[i][2]^2)
   *(a^2+b^2+c^2+d^2));
od:
if nops(liste)=2 then
  eq[3]:=diff(eq[1],a)*diff(eq[2],b)-diff(eq[2],a)*diff(eq[1],b);
  eq[4]:=diff(eq[1],a)*diff(eq[2],c)-diff(eq[2],a)*diff(eq[1],c);
elif nops(liste)=3 then
  v:=[a,b,c]:
  eq[4]:=det(matrix(3,3,(i,j)->diff(eq[i],v[j])));
fi:
RETURN(subs(d=1,[seq(eq[i],i=1..4)]));
end:

#= Begining of program =====================================================

gobo:=readdata("Coords.dat",3):

for j from 1 to 6 do
 x[j]:=map(convert,gobo[j],rational):
 y[j]:=map(convert,gobo[j+6],rational):
od:
l[1][0]:=convert(gobo[13][1],rational):
l[1][1]:=convert(gobo[13][2],rational):
l[2][0]:=convert(gobo[14][1],rational):
l[2][1]:=convert(gobo[14][2],rational):
l[3][0]:=convert(gobo[15][1],rational):
l[3][1]:=convert(gobo[15][2],rational):
l[4][0]:=convert(gobo[16][1],rational):
l[4][1]:=convert(gobo[16][2],rational):
l[5][0]:=convert(gobo[17][1],rational):
l[5][1]:=convert(gobo[17][2],rational):
l[6][0]:=convert(gobo[18][1],rational):
```

```
l[6][1]:=convert(gobo[18][2],rational):

R:=evalm(1/(a^2+b^2+c^2+d^2)*matrix(3,3,[a^2-b^2-c^2+d^2,2*(a*b-c*d),
2*(a*c+b*d),2*(a*b+c*d),-a^2+b^2-c^2+d^2,2*(b*c-a*d),2*(a*c-b*d),
2*(a*d+b*c),-a^2-b^2+c^2+d^2])):

lnumpat[2]:=[[1,2],[1,3],[1,4],[1,5],[1,6],[2,3],[2,4],[2,5],[2,6],[3,4],
        [3,5],[3,6],[4,5],[4,6],[5,6]]:
lnumpat[3]:=[[1,2,3],[1,2,4],[1,2,5],[1,2,6],[1,3,4],[1,3,5],[1,3,6],[1,4,5],
        [1,4,6],[1,5,6],[2,3,4],[2,3,5],[2,3,6],[2,4,5],[2,4,6],[2,5,6],
        [3,4,5],[3,4,6],[3,5,6],[4,5,6]]:
lnumpat[4]:=[[1,2,3,4],[1,2,3,5],[1,2,3,6],[1,2,4,5],[1,2,4,6],[1,2,5,6],
        [1,3,4,5],[1,3,4,6],[1,3,5,6],[1,4,5,6],[2,3,4,5],[2,3,4,6],[2,3,5,6],
        [2,4,5,6],[3,4,5,6]]:

monoms1:=[1,a,b,c,a*b,a*c,b*c,a^2,b^2,c^2]:
Ld1:=[[0,0,0],[1,0,0],[0,1,0],[0,0,1],[1,1,0],[1,0,1],[0,1,1],[2,0,0],[0,2,0],
        [0,0,2]]:

for i from 1 to 6 do
  lex[i]:=x[i][3]:
  lex[i+6]:=(l[i][1]+sqrt(y[i][1]^2+y[i][2]^2+y[i][3]^2))^2
                         -(x[i][1]-X)^2-(x[i][2]-Y)^2:
  lex[i+12]:=(l[i][0]-sqrt(y[i][1]^2+y[i][2]^2+y[i][3]^2))^2
                         -(x[i][1]-X)^2-(x[i][2]-Y)^2:
od:


Nom:=`GiveBornes`:
rv := []:
param := []:
glob := ["extern double X,Y; extern double hut[18]"]:
loc := []:
ass := [seq(hut[i-1]=lex[i],i=1..18)]:
pb := ProgramBlock(loc,ass):
np := NamedProgram(Nom,rv,param,glob,pb):
writeto("GiveBornes.c"):
  lprint(`#include <math.h>`):
  C(np,optimized):
writeto(terminal):

for i from 1 to 6 do
  lex[i]:=evalm([X,Y,Z]+R&*y[i]-x[i]):
  lex[i]:=subs(d=1,lex[i][1]^2+lex[i][2]^2+lex[i][3]^2):
od:

Nom:=`EvalLength`:
rv := []:
param := ["double Z, double a, double b, double c"]:
```

```
glob := ["extern double X,Y; extern double l[6]"]:
loc := []:
ass := [seq(l[i-1]=lex[i],i=1..6)]:
pb := ProgramBlock(loc,ass):
np := NamedProgram(Nom,rv,param,glob,pb):
writeto("EvalLength.c"):
  lprint(`#include <math.h>`):
  C(np,optimized):
writeto(terminal):


for k from 1 to nops(lnumpat[2]) do
  for i from 0 to 3 do
    bob:=Beq([[lnumpat[2][k][1],l[lnumpat[2][k][1]][iquo(i,2)]],
        [lnumpat[2][k][2],l[lnumpat[2][k][2]][irem(i,2)]]]);
    for ic from 1 to 4 do
      for j from 1 to 10 do
        V[ic][j]:=coeff(coeff(coeff(bob[ic],a,Ld1[j][1]),b,Ld1[j][2]),
                    c,Ld1[j][3]);
      od:
    od:
    gre:=map(eval,subs(Vc=V,evalm(GenBez))):
    limatgre:=NULL;
    fprintf(terminal,"Building Matrix %d\n",k*4+i-4);
    for lgh from 1 to 14 do
      for cgh from 1 to 14 do
        limatgre:=limatgre,expand(gre[cgh,lgh]):
      od:
    od:
    for d from 0 to 9 do
      N[d]:=map(coeff,[limatgre],Z,d):
    od:
    fprintf(terminal,"   Writing C output for Matrix %d...\n",k*4+i-4);
    for d from 0 to 9 do
      Nom:=`A`.(k*4+i-4).d:
      rv := []:
      param := ["double * tm"]:
      glob := ["X,Y"]:
      loc := []:
      ass := [seq(tm[ygh-1]= N[d][ygh],ygh=1..196)]:
      pb := ProgramBlock(loc,ass):
      np := NamedProgram(Nom,rv,param,glob,pb):
      writeto("matrixDir/A".(k*4+i-4).d.".c"):
        C(np,optimized):
      writeto(terminal):
    od:
  od:
od:
```

```
for k from 1 to nops(lnumpat[4]) do
  for i from 0 to 15 do
    bob:=Beq([[lnumpat[4][k][1],l[lnumpat[4][k][1]][iquo(i,8)]],
   [lnumpat[4][k][2],l[lnumpat[4][k][2]][iquo(irem(i,8),4)]],
           [lnumpat[4][k][3],l[lnumpat[4][k][3]][iquo(irem(i,4),2)]],
           [lnumpat[4][k][4],l[lnumpat[4][k][4]][irem(i,2)]]]);
    for ic from 1 to 4 do
      for j from 1 to 10 do
        V[ic][j]:=coeff(coeff(coeff(bob[ic],a,Ld1[j][1]),b,Ld1[j][2]),
                    c,Ld1[j][3]);
      od:
    od:
    gre:=map(eval,subs(Vc=V,evalm(GenBez))):
    limatgre:=NULL;
    fprintf(terminal,"Building Matrix %d\n",k*16+i+44);
    for lgh from 1 to 14 do
      for cgh from 1 to 14 do
        limatgre:=limatgre,expand(gre[cgh,lgh]):
      od:
    od:
    for d from 0 to 4 do
      N[d]:=map(coeff,[limatgre],Z,d):
    od:
    fprintf(terminal,"   Writing C output for Matrix %d...\n",k*16+i+44);
    for d from 0 to 4 do
      Nom:=`A`.(k*16+i+44).d:
      rv := []:
      param := ["double * tm"]:
      glob := ["X,Y"]:
      loc := []:
      ass := [seq(tm[ygh-1]= N[d][ygh],ygh=1..196)]:
      pb := ProgramBlock(loc,ass):
      np := NamedProgram(Nom,rv,param,glob,pb):
      writeto("matrixDir/A".(k*16+i+44).d.".c"):
        C(np,optimized):
      writeto(terminal):
    od:
  od:
od:


for k from 1 to nops(lnumpat[3]) do
  for i from 0 to 7 do
    bob:=Beq([[lnumpat[3][k][1],l[lnumpat[3][k][1]][iquo(i,4)]],
           [lnumpat[3][k][2],l[lnumpat[3][k][2]][iquo(irem(i,4),2)]],
           [lnumpat[3][k][3],l[lnumpat[3][k][3]][irem(i,2)]]]);
    for ic from 1 to 3 do
      for j from 1 to 10 do
        V[ic][j]:=coeff(coeff(coeff(bob[ic],a,Ld1[j][1]),b,Ld1[j][2]),
```

```
                   c,Ld1[j][3]);
      od:
    od:
    for j from 1 to 20 do
      V[4][j]:=coeff(coeff(coeff(bob[4],a,Ld2[j][1]),b,Ld2[j][2]),c,
                 Ld2[j][3]);
    od:
    gre:=map(eval,subs(Vc=V,evalm(GenBez2))):
    limatgre:=NULL;
    fprintf(terminal,"Building Matrix %d\n",k*8+i+292);
    for lgh from 1 to 28 do
      for cgh from 1 to 28 do
        limatgre:=limatgre,expand(gre[cgh,lgh]):
      od:
    od:
    for d from 0 to 7 do
      N[d]:=map(coeff,[limatgre],Z,d):
    od:
    fprintf(terminal,"   Writing C output for Matrix %d...\n",k*8+i+292);
    for d from 0 to 7 do
      Nom:=`A`.(k*8+i+292).d:
      rv := []:
      param := ["double * tm"]:
      glob := ["X,Y"]:
      loc := []:
      ass := [seq(tm[ygh-1]= N[d][ygh],ygh=1..784)]:
      pb := ProgramBlock(loc,ass):
      np := NamedProgram(Nom,rv,param,glob,pb):
      writeto("matrixDir/A".(k*8+i+292).d.".c"):
        C(np,optimized):
      writeto(terminal):
    od:
  od:
od:
```

B Code Maple pour calculer une forme générale de la matrice de Bezout

```
with(linalg):
read("bezout"):

R:=evalm(1/(a^2+b^2+c^2+d^2)*matrix(3,3,[a^2-b^2-c^2+d^2,2*(a*b-c*d),
     2*(a*c+b*d),2*(a*b+c*d),-a^2+b^2-c^2+d^2,2*(b*c-a*d),2*(a*c-b*d),
     2*(a*d+b*c),-a^2-b^2+c^2+d^2]));

#= Calcul de la matrice générale 1 ========================================
#
# C'est la matrice que l'on utilise quand on a quatre polynômes en a,b et c
# de degré inférieur à 2.

monoms1:=[1,a,b,c,a*b,a*c,b*c,a^2,b^2,c^2]:
Ld1:=[[0,0,0],[1,0,0],[0,1,0],[0,0,1],[1,1,0],[1,0,1],[0,1,1],[2,0,0],
       [0,2,0],[0,0,2]]:

for i from 1 to 4 do
  eq[i]:=evalm(monoms1&*[seq(Vc[i][j],j=1..10)]);
od:

GenBez:=mbezout([eq[1],eq[2],eq[3],eq[4]],[a,b,c]):

save monoms1,Ld1,GenBez,"genBezoutMat1";

#= Calcul de la matrice générale 2 ========================================
#
# Le cas où trois équations sont de degré 2 et une de degré trois :
# trois segments à des longueurs extrèmes et le déterminant de la matrice
# des dérivées partielles nul (c'est cette équation qui est de degré trois).

monoms2:=[1,a,b,c,a*b,a*c,b*c,a^2,b^2,c^2,a*b*c,a^2*b,a*b^2,a^2*c,a*c^2,b^2*c,
           b*c^2,a^3,b^3,c^3];
Ld2:=[[0,0,0],[1,0,0],[0,1,0],[0,0,1],[1,1,0],[1,0,1],[0,1,1],[2,0,0],[0,2,0],
       [0,0,2],[1,1,1],[2,1,0],[1,2,0],[2,0,1],[1,0,2],[0,2,1],[0,1,2],
       [3,0,0],[0,3,0],[0,0,3]];

for i from 1 to 3 do
 Vc[i]:=vector(10):
 eq[i]:=evalm(Vc[i]&*monoms1);
od:
Vc[4]:=vector(20):
eq[4]:=evalm(Vc[4]&*monoms2):

GenBez2:=mbezout([eq[1],eq[2],eq[3],eq[4]],[a,b,c]):
save monoms2,Ld2,GenBez2,"genBezoutMat2":
```

C La “matrice de Bezout” du package Elimination

```
##
##    Title:  Elimination
##    Created: Mon Jun 27 15:23:33 1998
##    Author:  Bernard Mourrain
## <mourrain@sophia.inria.fr>
##
## Description: Elimination of variables, multivariate bezoutian,
##              multivariate resultant, ...
##---------------------------------------------------------------
with(linalg):
##---------------------------------------------------------------
Jd := proc(lp::list)
# calcule de la matrice de Phi([p0,..,pn]) avec division par x[i]-xi[i]
# avant le determinant.
# meme option
local N,i,j,k,vx,vxi,mtr,s;
  if nargs <2 then
    vx := sort(convert(indets(lp),list));
  else
    vx := args[2]
  fi;
  N := nops(lp);
  if nops(vx) <> N-1 then
    ERROR(`the number of polynomials must be the number of variables +1 `)
  fi;
  mtr := array(1..N,1..N);
  for i from 1 to N do mtr[i,1] := lp[i] od;
  for j from 2 to N do
     for i from 1 to N do
       mtr[i,j]:= subs(vx[j-1]=xi[j-1],mtr[i,j-1])
     od;
  od;
  for j from N to 2 by -1 do
     for i from 1 to N do
       mtr[i,j] := quo(mtr[i,j]-mtr[i,j-1], xi[j-1] - vx[j-1], xi[j-1]);
     od;
  od;
  #dt :=det(convert(mtr,matrix));
  #collect(expand(dt),vx,distributed);
  evalm(mtr);
end:
#-----------------------------------------------------
Theta := proc(lp::list)
# calcule  Phi([p0,..,pn]) avec division par x[i]-xi[i] avant le determinant.
  Jd(args):
  det(%);
end:
```

```
#----------------------------------------------------------
Delta := proc(P,var,lx,lxi)
# dev de P dans la base des monomes en X et en xi;
local i,j,vx,vxi,c,nbase,lbase,lcx,nx,mxi,nxi,mtr,mr,lambda,v,lr,dt;
  if nargs <2 then
    vx  := sort([op(indets(P))]);
  else
    vx := args[2];
  fi;
  for i from 1 to nops(vx) do vx := subs(xi[i]=NULL,vx) od;
  vxi := [xi[j]$j=1..nops(vx)];
  dt :=collect(expand(P),vx,distributed);
  lcx:= [coeffs(dt,vx,'ll')];
  if nargs>2 then lx:=[ll] fi;
  nx := nops(lcx);
  nxi := 0;
  mr := [];
  mxi := NULL;
  for j from 1 to nops(lcx) do
    c := lcx[j];
    lambda := [coeffs(c,vxi,'v')];
    v := [v];
    if  nxi=0
        then lr :=[]
        else lr := convert(vector(nxi,0),list)
    fi;
    for i from 1 to nops(v) do
       if assigned(lbase[v[i]])
          then
       lr  := subsop(lbase[v[i]] = lambda[i], lr);
          else
       lr  := [op(lr),lambda[i]];
       nxi := nxi+1;
       lbase[v[i]] := nxi;
       mxi := mxi,v[i];
       fi;
    od;
    mr := [op(mr),lr];
  od;
  if nargs >3 then lxi := [mxi] fi;
  for  i from 1 to nops(mr) do
    if nops(mr[i]) <> nxi then
       mr := subsop(i= [op(mr[i]),
        op(convert(vector(nxi-nops(mr[i]), 0),list))],mr);
    fi;
  od;
  matrix(nx,nxi,mr);
end:
#----------------------------------------------------------
```

```
mbezout := proc(lpol::list)
# lpol := [p1, ..., pn];
# calcule PHi puis la matrice puis triangularise;
local var,M;
  if nargs <2 then
    var  := sort([op(indets(lpol))]);
  else
    var := args[2];
  fi;
  Theta(lpol,var);
  if nargs>3 then
     Delta(%,var,args[3],args[4]);
  elif nargs>2 then
     Delta(%,var,args[3]);
  else
     Delta(%,var);
  fi;
end:
#-------------------------------------------------------
```

La suite à été tronquée (car elle ne m'a pas servie) mais contient encore beaucoup d'autres algorithmes pour le calcul de résultants et l'élimination de variables.

D Le code C++ pour le calcul de l'espace avec des petits segments verticaux

```
#include "require.H"       // Ce sont des packages de la librairie
#include "linear/Solve.H"  // ALP, développée par le projet SAGA.
#include "linear/Eigen.H"
#include <math.h>
#include <list>
#include <fstream.h>

#define VCT Vector<Std<array1d<complex<double> > > >
#define MAT Matrix<Spl<lapack<double> > >
void (***A)(double * tm);

double X,Y,hmin1,hmin2,hmax1,hmax2;
double hut[18];
double l[6];
double extrl[12];

extern "C" void ConstArrFunc();
extern "C" void GiveBornes();
extern "C" void EvalLength(const double,const double,const double,
                             const double);

void BlockCompagnon(MAT * M, int n, MAT& Aa, MAT& Bb) {
 int i,j,k,d;
 d = M[0].nbrow();
 Aa.rep.rebuild(d*(n-1),d*(n-1));
 Bb.rep.rebuild(d*(n-1),d*(n-1));
 for (i=0;i<d*(n-2);i++) {
   Bb(i,i) = 1;
   Aa(i,i+d) = -1;
 }
 for (i=0;i<d;i++) {
   for (j=0;j<d;j++) {
     Bb(i+d*(n-2),j+d*(n-2)) = M[n-1](i,j);
    }
   for (j=0;j<d;j++) {
     for (k=0;k<n-1;k++) {
       Aa(i+d*(n-2),j+d*k) = M[k](i,j);
     }
   }
 }
}


double HeightInFunc(int i,int j){
  if (hut[i+6*j]<0) {
```

```
    return(0);
  } else {
    return(sqrt(hut[i+6*j]));
  }
}


double MakeSum(){
  extern const double l[6];
  double p,sum=0;
  for (int i=0;i<6;i++){
    p=sqrt(l[i])-(extrl[i]+extrl[i+6])/2;
    if (p<0){
      p=-p;
    }
    p+=(extrl[i]-extrl[i+6])/2;
    if (p<0){
      p=0;
    }
    sum+=p;
  }
  return(sum);
}


int TestIn(const double Z){
  cout << "Strating TestIn Z="<< Z <<endl;
  if (Z<hmax2 && Z>hmin2){
    return(0);
  } else {
    double vec[3];
    double at=-2,bt=-2,ct=2;
    EvalLength(Z,0,0,0);
    double old = MakeSum();
    cout << old << endl;
    for (int it=1;it<100;it++){
      if (old==0){
        return(1);
      } else {
        EvalLength(Z,at+0.001,bt,ct);
        vec[0] = MakeSum()-old;
        EvalLength(Z,at,bt+0.001,ct);
        vec[1] = MakeSum()-old;
        EvalLength(Z,at,bt,ct+0.001);
        vec[2] = MakeSum()-old;
        double N = vec[0]*vec[0]+vec[1]*vec[1]+vec[2]*vec[2];
        vec[0] /= N;
        vec[1] /= N;
        vec[2] /= N;
```

```
        at -= vec[0]/100;
        bt -= vec[1]/100;
        ct -= vec[2]/100;
        EvalLength(Z,at,bt,ct);
        old = MakeSum();
      }
    }
    return(0);
  }
}

void SelectInt(list<double>& V, list<double>& InterRemain){
  int prec=0;
  for (list<double>::const_iterator li=V.begin();li!=V.end();++li){
    if (++li==V.end()) {
      --li;
      if (prec==1){
        InterRemain.push_back(*li);
      }
    } else {
      --li;
      if (TestIn(0.5*((*li)+(*(++li))))==1){
        --li;
        if (prec==0){
          InterRemain.push_back(*li);
        }
        prec=1;
      } else {
        --li;
        if (prec==1){
          InterRemain.push_back(*li);
        }
        prec=0;
      }
    }
  }
}


void SaveInt(list<double>& InterRemain){
  ofstream DataFile("SolWork.dat", ios::app);
  DataFile << "+ " << X << " " << Y << endl;
  for (list<double>::const_iterator li=InterRemain.begin();
         li!=InterRemain.end();li++){
    DataFile << *li << " ";
  }
  DataFile << '\n' <<endl;
}
```

```
//- Main ----------------------------------------------------------------------

int main(int argc, char** argv) {
  ConstArrFunc();
  fstream Coords("Coords.dat", ios::in);
  for (int i=0;i<36;i++){
    Coords >> extrl[0];
  }
  for (int i=0;i<6;i++){
    Coords >> extrl[i] >> extrl[i+6];
  }
    if (argc==3) {
    X = atof(argv[1]);
    Y = atof(argv[2]);
  } else {
    cout << "Donnez les valeurs de X et Y :" << endl;
    cin >> X >> Y;
  }
  GiveBornes();
  hmax1=hut[0]+HeightInFunc(0,1);
  hmin1=hut[0]-HeightInFunc(0,1);
  hmax2=hut[0]+HeightInFunc(0,2);
  hmin2=hut[0]-HeightInFunc(0,2);
  for (int k=1;k<6;k++){
    double val;
    val=HeightInFunc(k,1);
    if (hut[k]+val<hmax1){
      hmax1=hut[k]+val;
    }
    if (hut[k]-val>hmin1){
      hmin1=hut[k]-val;
    }
    val=HeightInFunc(k,2);
    if (hut[k]+val>hmax2){
      hmax2=hut[k]+val;
    }
    if (hut[k]-val<hmin2){
      hmin2=hut[k]-val;
    }
  }
  cout << "inside : [" << hmin1 << "," << hmax1 << "]   outside : ["
            << hmin2 << "," << hmax2 << "]" << endl;
  if (hmin1>=hmax1 || (hmin2<=hmin1 && hmax2>=hmax1)){
    // on est trop loin ou alors le robot est bugg'e...
  } else {
    list<double> V,InterRemain;
    V.push_front(hmax1);
    V.push_front(hmax2);
    V.push_front(hmin1);
```

```
V.push_front(hmin2);

for (int k=0;k<1;k++){
  MAT Bb;
  MAT Aa;
  MAT * Ht =new MAT[10](14,14);
  for (int e=0;e<10;e++) {
     A[k][e](Ht[e].rep.tab_);
  }
  BlockCompagnon(Ht,10,Aa,Bb);
  VCT rout =  Eigenval(Aa,Bb);
  for (int i=0;i<rout.size();i++) {
    if (rout[i].imag()==0 && rout[i].real()<hmax1 &&
          rout[i].real()>hmin1){
      if (rout[i].real()>hmax2 || rout[i].real()<hmin2){
        V.push_front(rout[i].real());
      }
    }
  }
  delete[](Aa.rep.tab_);
  delete[](Bb.rep.tab_);
  for(int i=0;i<10;i++){
    delete[](Ht[i].rep.tab_);
  }
}

for (int k=60;k<300;k++){
  MAT Bb;
  MAT Aa;
  MAT * Ht =new MAT[5](14,14);
  for (int e=0;e<5;e++) {
     A[k][e](Ht[e].rep.tab_);
  }
  BlockCompagnon(Ht,5,Aa,Bb);
  VCT rout =  Eigenval(Aa,Bb);
  for (int i=0;i<rout.size();i++) {
    if (rout[i].imag()==0 && rout[i].real()<hmax1 &&
          rout[i].real()>hmin1){
      if (rout[i].real()>hmax2 || rout[i].real()<hmin2){
        V[n]=rout[i].real();
        n++;
      }
    }
  }
  delete[](Aa.rep.tab_);
  delete[](Bb.rep.tab_);
  for(int i=0;i<5;i++){
    delete[](Ht[i].rep.tab_);
  }
```

```
    }

    for (int k=300;k<460;k++){
      MAT Bb;
      MAT Aa;
      MAT * Ht =new MAT[8](28,28);
      for (int e=0;e<8;e++) {
         A[k][e](Ht[e].rep.tab_);
      }
      BlockCompagnon(Ht,8,Aa,Bb);
      VCT rout =  Eigenval(Aa,Bb);
      for (int i=0;i<rout.size();i++) {
        if (rout[i].imag()==0 && rout[i].real()<hmax1 &&
              rout[i].real()>hmin1){
          if (rout[i].real()>hmax2 || rout[i].real()<hmin2){
            V[n]=rout[i].real();
            n++;
          }
        }
      }
      delete[](Aa.rep.tab_);
      delete[](Bb.rep.tab_);
      for(int i=0;i<8;i++){
        delete[](Ht[i].rep.tab_);
      }
    }
    V.sort();
    SelectInt(V,InterRemain);
    SaveInt(InterRemain);
  }
  return(0);
}
```

Il manque à la fin du programme la phase où l'on doit tester pour chaque segment si il est à l'interieur ou à l'exterieur de l'espace de travail, mais ce n'est pas la partie interessante de l'agorithme.