Module « Contrôle Avancé »
Rapport

Éric Abouaf
avril 2006

Table des matières

1 Le Jeu de la vie....3
1.1 Introduction....3
1.2 Programmation....3
1.3 Résultats et conclusions....5
2 Algorithmes génétiques....6
2.1 Le problème du voyageur de commerce....6
2.1.1 Introduction....6
2.1.2 Mise en oeuvre 6
2.1.3 Différentes reproductions et mutations....8
2.1.4 Résultats....9
2.1.5 Conclusion.... 10
2.2 Approximation d'ellipses.... 11
2.2.1 Le problème.... 11
2.2.2 Le codage et l'évaluation....11
2.2.3 Algorithme et résultats.... 11
3 Logique Floue....13
3.1 Le problème.... 13
3.2 Le programme....13
3.3 Résultats....17

1 Le Jeu de la vie

1.1 Introduction

Le jeu de la vie est un automate cellulaire inventé par John Horton Conway. Cet automate est régit par quelques règles très simples.

L'aire de jeu est formée d'une matrice contenant des cellules vivantes ou mortes. À chaque étape, l'évolution d'une cellule est entièrement déterminée par l'état de ses huit voisines de la façon suivante:

- Une cellule morte possédant exactement trois voisines vivantes devient vivante (elle naît).
- Une cellule vivante possédant deux ou trois voisines vivantes le reste, sinon elle meurt.

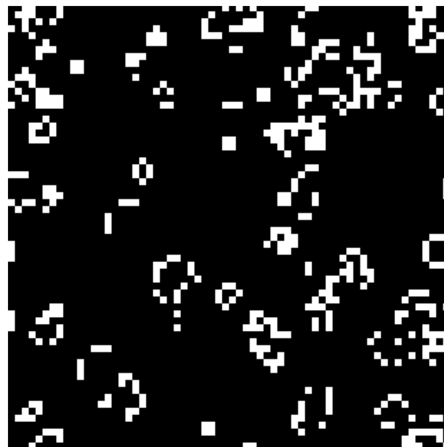


fig. 1
Exemple d'une aire de jeu

1.2 Programmation

Le programme permettant de simuler cet automate cellulaire est très simple. Voici une version pour Matlab qui permet d'enregistrer le résultat sous forme d'un fichier vidéo.

Fichier jeudelavie.m :

```
% Jeu de la vie
% Module Controle Avancé
% Eric Abouaf <neyric@via.ecp.fr>
% 7 Mars 2006

% Paramètres
taille = 64;
steps = 100;

% Programme
```

```
% Génération aléatoire de la première matrice :
matrice = rand(taille,taille);
for i=1:taille
   for j=1:taille
      if matrice(i,j) < 0.5
         matrice(i,j) = 0;
       else
          matrice(i,j) = 1;
       end
    end
end
matrice = logical(matrice);


aviobj = avifile('jeudelavie.avi','fps',5);

for step=1:steps

    % Enregistrement de l'historique
    resultat(:,:,step) = matrice;

    imshow(matrice);  %,'InitialMagnification',500);
    aviobj = addframe(aviobj,getframe);

    % Un pas :
    newmat = logical(zeros(taille, taille));
    for i=1:taille
        for j=1:taille
            n = NombreVoisins(matrice, i,j);

            % Naissance
            if n == 3
                newmat(i,j)=1;
            % Survie
            elseif matrice(i,j) == 1 & ( n == 2 | n == 3 )
                newmat(i,j)=1;
            % Mort
            else
                newmat(i,j)=0;
            end
        end
    end
    matrice = newmat;
end

aviobj = close(aviobj);
```

Fichier NombreVoisins.m :

```
% Renvoie le nombre de voisins (8-connexité) vivants :

function [ n_voisins ] = NombreVoisins( matrice , i , j )

[ width height ] = size(matrice);

top = mod(j-1,height);
bottom = mod(j,height)+1;
left = mod(i-1,width);
right = mod(i,width)+1;

% L'espace est torique :
if top == 0
    top = height;
end
if left == 0
    left = width;
end
```

```
n_voisins = 0;
if matrice(left,top)    n_voisins = n_voisins+1;  end
if matrice(left,j)      n_voisins = n_voisins+1;  end
if matrice(left,bottom) n_voisins = n_voisins+1;  end
if matrice(i,top)       n_voisins = n_voisins+1;  end
if matrice(i,bottom)    n_voisins = n_voisins+1;  end
if matrice(right,top)   n_voisins = n_voisins+1;  end
if matrice(right,j)     n_voisins = n_voisins+1;  end
if matrice(right,bottom)n_voisins = n_voisins+1;  end

end
```

1.3 Résultats et conclusions

Les tests de simulation révèlent très vite la présence de structures stables et de structures oscillantes. Voici un exemple de structure stable et un exemple de structure oscillante :

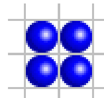


Fig 2. Structure stable

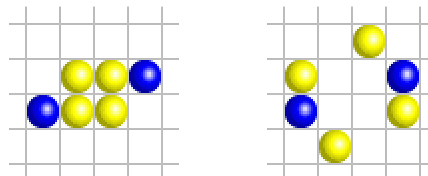


fig 3. Structure oscillante

Il existe également d'autres figures appellées *Vaisseaux* tels que les *planeurs* ou *méduses* , qui sont des structures capables de se reproduire elles-mêmes, translatées dans l'aire de jeu.

Notons un résultat mathématique intéressant: le jeu de la vie constitue *une machine de Turing universelle*, c'est à dire qu'il est possible d'effectuer n'importe quel algorithme si l'aire de jeu est suffisamment grande.

La réalisation du jeu de la vie nous a permis de prendre en main Matlab et de nous familiariser avec des notions communes aux algorithmes génétiques tels que les *générations.*

2 Algorithmes génétiques

2.1 Le problème du voyageur de commerce

2.1.1 Introduction

Le problème du voyageur de commerce est un classique des algorithmes génétiques. En effet, on ne connaît pas d'algorithme qui soit capable de donner un résultat exact en un temps raisonnable. Par conséquent, on doit se replier sur des méthodes permettant de trouver de « bonnes » solutions, comme les algorithmes génétiques.

L'énoncé du problème est le suivant : Etant donné *n* villes représentées par des points, et connaissanr les distances séparant chacune de ces villes, il s'agit de trouver un chemin de longueur totale minimale qui passe par toutes les villes une et une seule fois, puis revienne à la ville de départ. Voici un résultat trouvé sur Internet obtenu par algorithme génétique sur les plus grandes villes de France :

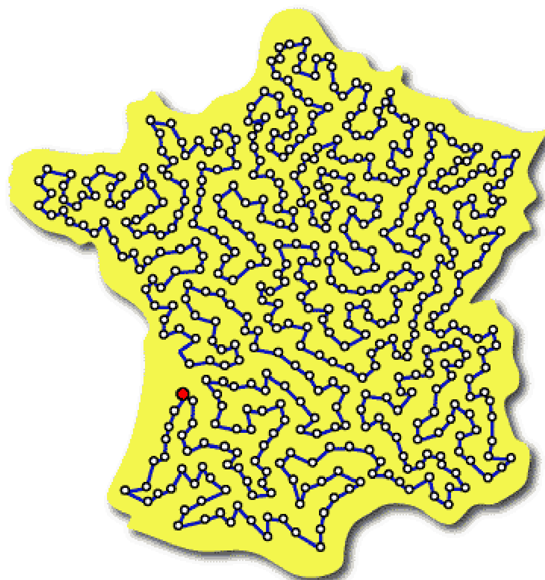


fig 4. Problème du voyageur de commerce

2.1.2 Mise en oeuvre

Les algorithmes génétiques reposent sur un processus calqué sur le principe de « séléction naturelle ». On crée une population de voyageurs de commerce qui vont voyager aléatoirement entre toutes les villes. On calcule pour chacun d'entre eux la distance totale parcourue. Une fois les voyageurs les plus performants « séléctionnés », on essaie de générer une nouvelle population de voyageurs, si possible plus performante que la précédent.

L'algorithme suit alors le plan suivant :

a) Création de la population initiale

b) Evaluation de l'adaptation de la population

c) Classement des individus suivant leur adaptation

d) Reproduction par croisement

e) Mutations de quelques gènes sur quelques individus

On réitère ensuite les étapes de b) à e) jusqu'à obtenir un résultat satisfant.

Les étapes de a) à c) représentent peu d'interêt. La seule optimisation à y apporter réside dans la rapidité du code. Voici la façon dont j'ai procédé :

Début du Fichier voyageur.m :

```
% Paramètres
n_individus = 50;
n_generation = 1000;
taux_mutation = 0.5;


% Calcul les positions des villes à partir d'une image
% ainsi que les distances entre chaque ville (à vol d'oiseau bien sûr)
[coords_villes distances_villes n_villes] = CoordonneesVilles('carte40.png');


% Génération de la population initiale
trajets=zeros(n_individus,n_villes-1);
for i=1:n_individus
   % On part toujours de la ville 1 et on y arrive
   trajets(i,:) = randperm(n_villes-1)+1;          % Utilisation d'une permutation d'un n-uplet
end

% Sauvegarde des distances parcourues par tous les individus
distances_parcourues = zeros( n_individus, n_generation ) ;



% Boucle principale :
for step=1:n_generation

   % Evaluation de l'adaptation des individus
   individus = zeros(n_individus,1);
   for i=1:n_individus
      trajet = [ 1 , trajets(i,:) , 1 ];
      for k=1:n_villes
         individus(i,1)=individus(i,1)+distances_villes(  trajet(k), trajet(k+1) );
      end
      individus(i,2)=i;
   end

   % Classement selon le niveau d'adaptation
   individus=sortrows(individus);
```

Les étapes de d) et e) sont de loin les plus délicates et différentes techniques ont été utilisées pour étudier leur influence sur les résultats.

2.1.3 Différentes reproductions et mutations

- **La reproduction classique**

La reproduction simple entre deux individus consiste à créer un troisième individu qui passera en k-ème position par la k-ème ville de son premier parent ou la k-ème ville de son second parent.

Le problème est que la plupart des trajets ainsi générés ne sont pas valides et doivent être « corrigés » par un autre algorithme. A cause de cette correction, une telle reproduction tient donc plus de la mutation que de la reproduction au sens génétique du terme. Pire, suivant l'algorithme de correction, ces mutations sont souvent les mêmes et enferment les populations dans des minimas locaux.

- **L'élitisme**

Afin de ne jamais faire redescendre la meilleur valeur d'adaptation de la population, on a recours à l'élitisme. Il s'agit juste de conserver le meilleur individu d'une population dans la génération suivante.

Il est parfois interessant de faire de l'élitisme sur les k premiers individus si ceux-ci empruntent des trajets assez différents afin de les faire se reproduire.

- **Les mutations**

De nombreux types de mutations peuvent être utilisés. On peut tout simplement intervertir deux villes mais également des suites de villes (méthode semblable au *décroisement* mais moins efficace.)

- **La création de nouveaux individus**

Afin de continuer l'exploration de l'espace des solutions et pour ne pas rester bloquer dans des *minimas locaux,* la création d'indivus complètement aléatoires à chaque nouvelle génération s'est révélé parfois très efficace. (Elle introduit parfois, au même titre que les mutations, des gros « sauts » dans les valeurs d'adaptations).

2.1.4 Résultats

fig 5. Un trajet trouvé par l'algorithme pour 50 individus sur 800 générations

fig 6. Distance parcourue par le meilleur individu en fonction des générations

2.1.5 Conclusion

Au final, les meilleurs résultats ont été obtenus à l'aide d'un très grand nombre de mutations. Les mutations permettent en effet de créer un individu « éloigné » de l'individu de départ ce qui permet d'explorer une plus grande partie de l'espace des solutions. On obtient également de bons résultats en peu de générations à l'aide d'une grande taille de population, ce qui correspond à une approche « aléatoire » similaire.

Ces résultats ne reproduisent donc pas l'approche initiale qui était surtout basée sur la reproduction mais l'approche du « tir de test » pour explorer le plus grand champ de solutions est très efficace, surtout lorsque elle est couplée à des méthodes d'optimisations locales telles que le « décroisement ». On arrive alors à un algorithme semblable au *« branch and bound »* (algorithme par séparation et évaluation, en français).

fig 7. Voyageur de commerce aux Etats-unis

2.2 Approximation d'ellipses

2.2.1 Le problème

Le problème est cette fois de trouver la plus proche ellipse passant par huits points de mesure de distance effectués à partir d'un même point dans les directions indiquées par la figure ci-dessous:

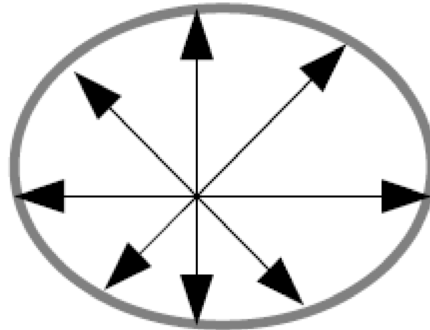


fig 8. Approximation d'ellipse à partir de 8 mesures

2.2.2 Le codage et l'évaluation

Le codage choisit ici est de la forme:*[a b x y]* où a et b représentent respectivement le grand axe et le petit axe de l'ellipse et où (x,y) sont les coordonnées du centre de l'ellipse.

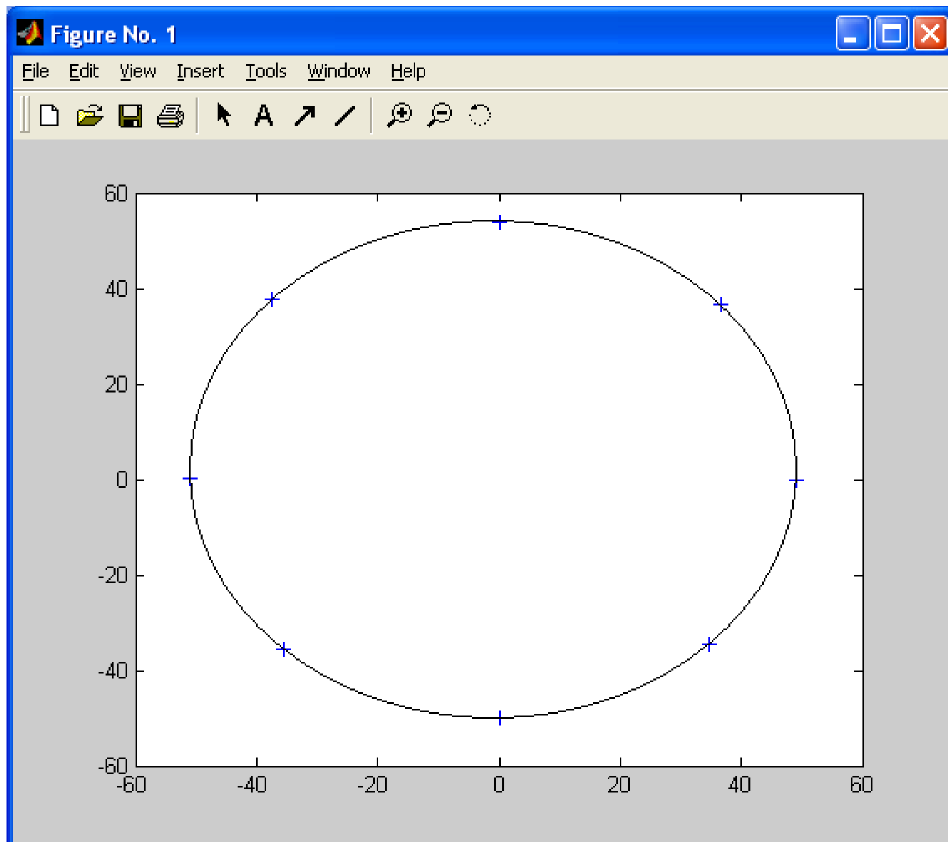
L'évaluation et la valeur d'adaptation de chaque « individu ellipse » est directement donné par son l'équation de l'ellipse. La valeur du terme suivant doit être la plus petite possible :

$$\frac{(x-x0)}{a^2}+\frac{(y-y0)}{b^2}$$

2.2.3 Algorithme et résultats

Chaque population est réalisée à partir de :

- Élitisme sur 1 individu
- ½ de croisement
- ¼ de mutations
- ¼ de nouveaux individus (aléatoires)



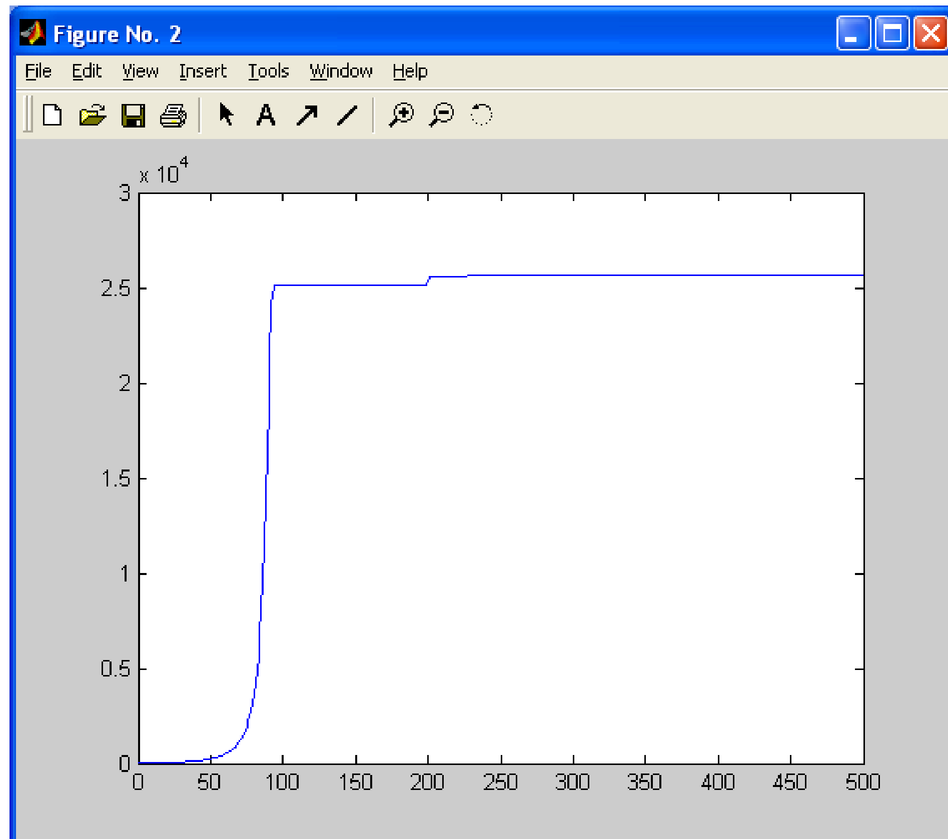


fig 9. Résultats pour 60 individus et 500 générations

3 Logique Floue

3.1 Le problème

Le problème est ici de contrôler un robot mobile (simulé dans notre cas) à l'aide de deux capteurs de distance infrarouges afin d'éviter qu'il se cogne dans les murs.

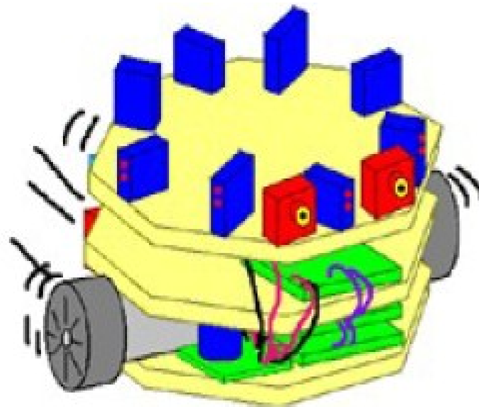


fig 10. Le robot mobile et ses capteurs infrarouges

3.2 Le programme

```
%Paramètres :
n_steps = 1000;
distance_fuzz = 3;

% Modélisation du robot
pos_robot = [ 0 ; 0 ];
angle_robot = 2*pi/3+pi/4;
position = [];
vitesse_robot=0.5;

% Définition des fonctions d'appartenances :
TP=[];MO=[];TG=[];
for i=0:.01:distance_fuzz
   TP=[ TP , 1-i/distance_fuzz ];
   MO=[ MO , i/distance_fuzz ];
   TG=[TG,0];
end
```

```
for i=(distance_fuzz+0.01):.01:distance_fuzz*2
   TP=[TP,0];
   MO=[ MO , 1-(i-distance_fuzz)/distance_fuzz ];
   TG=[TG,(i-distance_fuzz)/distance_fuzz];
end
for i=(2*distance_fuzz+0.01):.01:10
   TP=[TP,0];
   MO=[ MO , 0 ];
   TG=[TG,1];
end

% Définition des sorties :
A=[];G=[];D=[];
for i=-5:.01:-1
   A=[ A, 0];
   G=[ G, 1];
   D=[ D, 0];
end
for i=-1+0.01:.01:0
   A=[ A, 1+i];
   G=[ G, -i];
   D=[ D, 0];
end
for i=0.01:.01:1
   A=[ A, 1-i];
   G=[ G, 0];
   D=[ D, i];
end
for i=1.01:.01:5
   A=[ A, 0];
   G=[ G, 0];
   D=[ D, 1];
end
% Boucle :
for step=1:n_steps

   position = [ position , pos_robot ];

   % Fuzzification
   if angle_robot > pi/2 & angle_robot < 3*pi/2
    distances = [ abs( 15-pos_robot(2) )/abs(sin(angle_robot-pi/2)) , abs(-15-pos_robot(2))/abs(sin(3*pi/4-angle_robot)) ]
   else
    distances = [ abs( -15-pos_robot(2) )/abs(sin(angle_robot-pi/2)) , abs(15-pos_robot(2))/abs(sin(3*pi/4-angle_robot)) ]
   end
```

```
% On fuzz les deux distances:
% On calcul les degrés d'appartenance
muTP1 = TP( min(  [floor(100*distances(1)),size(TP,2)] ) );
muMO1 = MO( min(  [floor(100*distances(1)),size(MO,2)] ) );
muTG1 = TG( min(  [floor(100*distances(1)),size(TG,2)] ) );

muTP2 = TP( min(  [floor(100*distances(2)),size(TP,2)] ) );
muMO2 = MO( min(  [floor(100*distances(2)),size(MO,2)] ) );
muTG2 = TG( min(  [floor(100*distances(2)),size(TG,2)] ) );

% Inférence

%     ----------------------
%    Angle |  TP  | MO | TG |  -> capteur gauche
%     ----------------------
%     TP  |  A    G    G        1 2 3
%     MO  |  D    A    A        4 5 6
%     TG  |  D    A    A        7 8 9
regles = zeros(9,size(A,2));
regles(1,:) = A;    regles(2,:) = G;    regles(3,:) = G;
regles(4,:) = D;    regles(5,:) = A;    regles(6,:) = A;
regles(7,:) = D;    regles(8,:) = A;    regles(9,:) = A;

% On calcul les mini :
mini = zeros(9,1);
mini(1) = min(muTP1, muTP2);    mini(2) = min(muTP1, muMO2);
mini(3) = min(muTP1, muTG2);    mini(4) = min(muMO1, muTP2);
mini(5) = min(muMO1, muMO2);    mini(6) = min(muMO1, muTG2);
mini(7) = min(muTG1, muTP2);    mini(8) = min(muTG1, muMO2);
mini(9) = min(muTG1, muTG2);

maxi = zeros(9,1);
for i=1:9
   minimum = mini(i);
   for k=1:size(regles,2)
      if regles(i,k) > minimum
         regles(i,k) = minimum;
      end
   end
   maxi(i) = max( regles(i,:) );
end
```

```
    % Defuzzification
    maxi(:,2) = maxi(:,1)/sum(maxi(:,1));
    perc_A = maxi(1,2)+maxi(5,2)+maxi(6,2)+maxi(8,2)+maxi(9,2);
    perc_D = maxi(4,2)+maxi(7,2);
    perc_G = maxi(2,2)+maxi(3,2);

    angle_deplacement = (pi/100*perc_G-pi/100*perc_D)/2

    % Moteur "physique"
    angle_robot = angle_robot+angle_deplacement;
    pos_robot = pos_robot + vitesse_robot*[ cos(angle_robot) ; sin(angle_robot) ];

end

subplot(2,2,1);
hold on
plot(position(1,:),position(2,:),'b-');
plot([-500,30],[15,15],'r-');
plot([-500,30],[-15,-15],'g-');
hold off

subplot(2,2,2);
hold on
plot(0:.01:10,TP,'r-');
plot(0:.01:10,MO,'g-');
plot(0:.01:10,TG,'b-');
hold off

subplot(2,2,3);
hold on
plot(-5:.01:5,A,'r-');
plot(-5:.01:5,G,'g-');
plot(-5:.01:5,D,'b-');
hold off

subplot(2,2,4);
hold on
for i=1:9
    plot(-5:.01:5,regles(i,:),'-');
end
hold off
```

3.3 Résultats

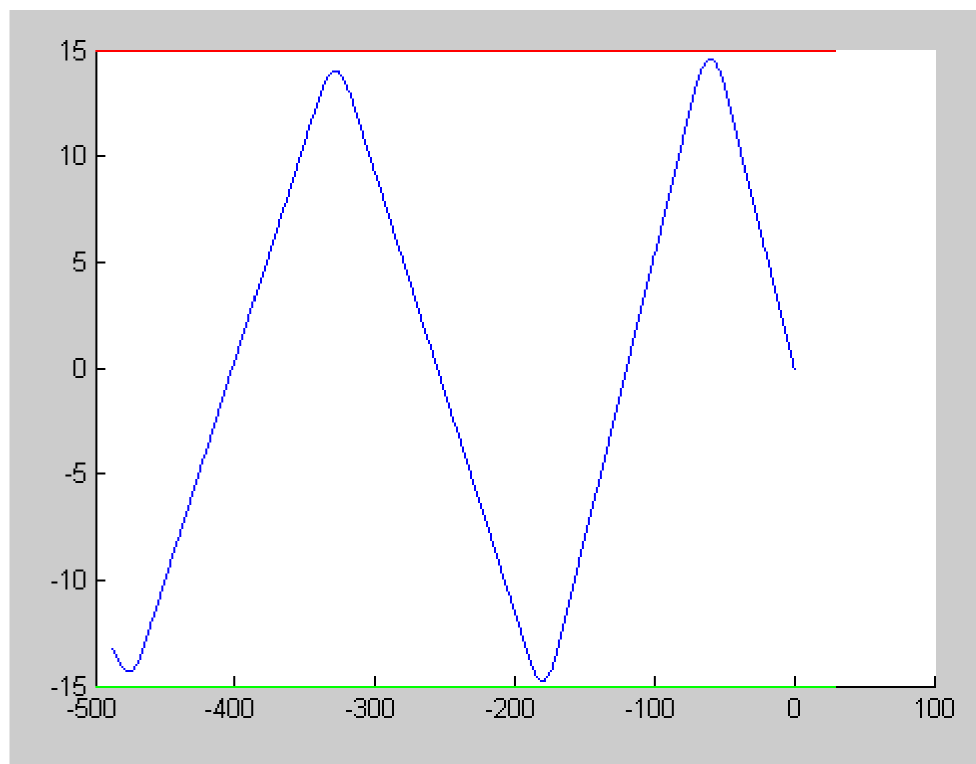


fig 11. Trajet du robot

(les murs sont en haut et en bas)

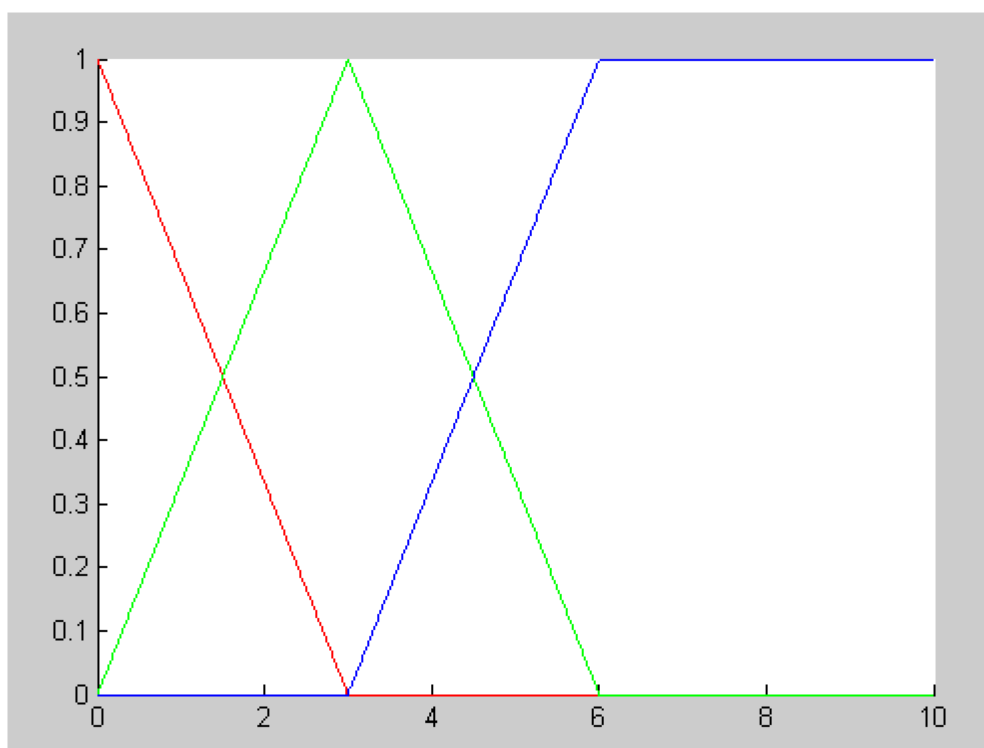


fig 12. Représentations des degrès d'appartenance

(TP: très petit, MO: moyen, TG: très grand)