Particle Swarm Optimization

Contents

1 Swarm Intelligence

1.1 Properties of Swarm Intelligence

Swarm Intelligence (SI) refers to a class of algorithms that simulates natural and artificial systems composed of many individuals that coordinate using decentralized control and self-organization. The algorithm focuses on the collective behaviors that result from the local interactions of the individuals with each other and with the environment where these individuals stay. Some common examples of systems involved in swarm intelligence are colonies of ants and termites, fish schools, bird flock, animal herds.

A typical swarm intelligence system has the following properties:

1. It consists of many individuals.
2. The individuals are relatively homogeneous (i.e. they are either all identical or they belong to a few typologies).
3. The interactions between the individuals are based on simple behavioral rules that exploit only local information that the individuals exchange directly or via the environment.
4. The global behavior of the system results from the local interactions of individuals with each other and with their environment.

The main property of a swarm intelligence system is its ability to act in a coordinated way without the presence of a coordinator or of an external controller. Despite the lack of individuals in charge of the group, the swarm as a whole does show intelligent behavior. This is the result of interactions of spatially located neighboring individuals using simple rules.

The swarm intelligence has already been used in many different fields since it was first introduced. The studies and applications of swarm intelligence mainly focus on the clustering behavior of ants, nest-building behavior of wasps and termites, flocking and schooling in birds and fish, ant colony optimization, particle swarm optimization, swarm-based network management, cooperative behavior in swarms of robots, etc.

1.2 Particle Swarm Optimization

The PSO algorithm is an important member of swarm intelligence algorithms originally developed by Kennedy and Eberhart in 1995 [1]. It was motivated by social behavior of bird flock or fish schooling and the technique shares many similarities with evolutionary computation techniques such as Genetic Algorithms (GA). As in other population-based intelligence systems, PSO requires an initial population of random solutions. The search for optima is obtained by updating generations without evolution operators such as crossover and mutation. The potential solutions are usually called particles in PSO. These particles fly through the solution space by following their own experiences and the current optimum particles. It was shown that the PSO algorithm is comparable in performance with and may be considered as an alternative method to GA.

2 Particle Swarm Optimization

The Particle Swarm Optimization algorithm falls into the category of SI algorithms and is a population-based optimization technique. It was motivated by social behavior (i.e. collective behavior) of bird flocking or fish schooling and shares many similarities with evolutionary computation techniques such as GA. The optimization process of a PSO system begins with an initial population

of random solutions and searches for optima by updating various properties of the individuals in each generation. However, unlike GA, PSO does not have evolution operators such as crossover and mutation. The potential solutions are known as particles which fly through the solution space by following their own experiences and the current best particles.

2.1 Formal Description of PSO

Suppose there are M agents, called particles, used in the PSO algorithm. Each agent is treated as a particle with infinitesimal volume with its properties being described by the current position vector, its velocity vector and the personal best position vector. These vectors are N-dimensional vectors containing properties based on the N decision variables. At the n-th iteration, the three vectors describing the properties of particle $i(1 \leq i \leq M)$ are:

1. The current position vector: $X_{i,n} = (X_{i,n}^1, \cdots, X_{i,n}^j, \cdots, X_{i,n}^N)$, where each component of the vector represents a decision variable of the problem and $1 \leq j \leq N$.

2. The velocity vector: $V_{i,n} = (V_{i,n}^1, \cdots, V_{i,n}^j, \cdots, V_{i,n}^N)$, denoting the increment of the current position where $1 \leq j \leq N$.

3. The personal best position vector: $P_{i,n} = (P_{i,n}^1, \cdots, P_{i,n}^, \cdots, P_{i,n}^N)$ for $1 \leq j \leq N$.

Note that the personal best position vector, which is denoted as p_{best}, returns the best objective function value or fitness value in the subsequent iterative process. The initial approximation of $X_{i,0}$ may be randomly generated within the search domain $[X_{min}^j, X_{max}^j](1 \leq j \leq N)$, where X_{min}^j and X_{max}^j are lower limit and upper limit of particle positions in the j-th dimension. Generally, uniform distribution on $[X_{min}^j, X_{max}^j]$ in the j-th dimension is used to generate the initial current position vector. In a similar way, the initial velocity vector $V_{i,0}$ may be initialized by choosing its j-th component randomly on $[-V_{max}^j, V_{max}^j]$, where V_{max}^j is the upper limit of velocities in the j-th dimension. The initial approximation of $P_{i,n}$ can be set as the initial current position vector.

At the n-th iteration, the particle with its personal best position which returns the best objective function value among all the particles is called the global best particle. This personal best position is denoted as $P_{g,n}$, which is recorded in a position vector $G_n = (G_n^1, \cdots, G_n^N)$ known as the global best position. Here g is the index of the global best particle.

Without loss of generality, we can consider the minimization of an objective function $f(x)$ (the maximization problem is the negation of the minimization problem). The update of $P_{i,n}$ is given by

$$P_{i,n} = \begin{cases} X_{i,n}, & f(x_{i,n}) < f(P_{i,n}) \\ P_{i,n-1}, & f(x_{i,n}) \geq f(P_{i,n}), \end{cases} \tag{1}$$

and thus G_n can be found by $G_n = P_{g,n}$, where $g = \text{argmax}_{1 \leq j \leq M}\{f(P_i, n)\}$.

At the $(n+1)$-th iteration, the component-wise properties of each particle can be stated formally as:

$$V_{i,n+1}^j = V_{i,n}^j + c_1 r_{i,n}^j (P_{i,n}^j - X_{i,n}^j) + c_2 R_{i,n}^j (G_n^j - X_{i,n}^j), \tag{2}$$

$$X_{i,n+1}^j = X_{i,n}^j + V_{i,n+1}^j, \tag{3}$$

for $i = 1, \cdots, M$, $j = 1, \cdots, N$, where c_1 and c_2 are acceleration coefficients. The parameters $r_{i,n}^j$, and $R_{i,n}^j$, form two different sequences of random numbers distributed uniformly over $(0,1)$, that is, $\{r_{i,n}^j : j = 1, \cdots, N\} \sim U(0,1)$ and $\{R_{i,n}^j : j = 1, \cdots, N\} \sim U(0,1)$. Generally, the vector $V_{i,n}$ has each of its components being restricted in the interval $[-V_{max}^j, V_{max}^j]$. The PSO in which the velocities of the particles are updated according to (2) is called the original PSO.

2.2 Qualitative Analysis of the Velocity Updating Equation

The parameters c_1 and c_2 are known as the acceleration coefficients which are two positive constants; $r_{i,n}^j$ and $R_{i,n}^j$ are two random numbers uniformly distributed on $(0,1)$. $P_{i,n}$ is the personal best position of the i-th particle and G_n is the global best position at the n-th iteration. The global best position is selected from the personal best positions of all particles in the swarm at each iteration. From Equation 2, it is clear that the updating velocity of each particle is composed of three parts as:

1. The previous velocity, $V_{i,n}^j$, known as the 'inertia part, providing the necessary momentum for the particles to fly in the search space.

2. The 'cognition' part, $c_1 r_{i,n}^j (P_{i,n}^j - X_{i,n}^j)$, representing the particle's own experience.

3. The 'social' part, $c_2 R_{i,n}^j (G_{i,n}^j - X_{i,n}^j)$, reflecting the information shared among all the particles.

When $c_1 = c_2 = 0$, there are no cognition and social parts in the velocity update. The particle flies uniformly in a straight line until it reaches the boundary of the search space. No optimal solution can be found unless the solution is on its trajectory.

When $c1 = 0$, there is no cognition part in Equation 2. The resulting velocity update equation is known as the social-only model. With the interaction among particles, the PSO algorithm has the ability to reach a new search space. However, the particle swarm converges at a relatively fast speed. The algorithm can be easily trapped into the local optima for complex problems, but it may have good performance in solving some specific problems.

When $c_2 = 0$, there is no social part in the velocity update. This means that there is no information sharing between the particles. The PSO algorithm

is known as running in cognition-only model. Without any interaction amongst the particles, running a swarm with population size M is essentially equivalent to running M individuals independently. As such, the PSO algorithm becomes a multi-start random search algorithm and has slower convergence rate than the algorithm in social-only model.

When there is no inertia part, the particle velocity is determined by its personal best position and the global best position. In this model, the velocity of the particle is said to be memory-less and the particle has better local search ability. Hence, the algorithm can converge more rapidly than the original PSO. However, without the inertia part, particles can only sample their positions around their personal best and global best positions, and have less probability of searching other areas.

3 Variants of Particle Swarm Optimization

3.1 Inertia Weight

In order to balance the local search and global search during an optimization process in the original PSO, the concept of an inertia weight, w, is introduced, leading to the modified velocity update:

$$V_{i,n+1}^{j} = wV_{i,n}^{j} + c_1 r_{i,n}^{j}(P_{i,n}^{j} - X_{i,n}^{j}) + c_2 R_{i,n}^{j}(G_{n}^{j} - X_{i,n}^{j}). \tag{4}$$

The PSO algorithm supplemented with (4) is known as PSO with Inertia Weight (PSO-In). The inertia weight w can be a positive value chosen according to experience or from a linear or non-linear function of the iteration number. When $w = 1$, the PSO-In is equivalent to the original PSO. The values of c_1 and c_2 in Equation 2 are generally set to be 2, which implies that the social and cognition parts have the same influence on the velocity update.

When w is between 0.8 and 1.2, the PSO algorithm has better opportunities to find the global optimum in a reasonable number of iterations. Within the range of $0.9 - 1.2$, Shi and Eberhart [2] introduced a significant improvement in the performance of the PSO method with a time-decreasing inertia weight over the generations. This time-decreasing strategy for w can be written as:

$$w_n = \frac{(w_{initial} - w_{final})(n_{max} - n)}{n_{max}} + w_{final}, \tag{5}$$

where w_n is the value of the inertial weight at the n-th iteration, $w_{initial}$ and w_{final} are the initial and final values of the inertia weight, n is the current iteration number n_{max} is the maximum number of iterations.

Therefore, at the beginning of the search within a run, the PSO with a large inertia weight can provide high diversity in order to use the full range of the search space; and at the end of the search, a small inertia weight can help the PSO converge to the optimal solution with fine-tuning. The commonly used value of w changes from $w_{initial} = 0.9$ at the beginning of the search to $w_{final} = 0.4$ at the end of the search.

3.2 Random Inertia Weight

In order to solve the dynamic nature of real-world applications effectively by PSO, the following random inertia weight was proposed [3]:

$$w_n = \frac{0.5 + rand}{2}, \tag{6}$$

where rand is a random number uniformly distributed within the range $(0, 1)$. The value of the inertia weight ranges between 0.5 and 1.5 with a mean value of 0.75. Several benchmark functions were examined and the results showed that it has rapid convergence in the early stage of the optimization process and can find a reasonably good solution for most of the functions.

In [4], Pant et al. proposed a new dynamic inertia weight by using Gaussian distribution (i.e. normal distribution). This kind of random inertia weight is defined by the equation:

$$w_n = \frac{|randn|}{2}, \tag{7}$$

where *randn* is a random number with the standard normal distribution. Different probability distributions, such as Gaussian distribution and exponential probability distribution, may be used to initialize the velocities and positions of the particles. The Gaussian distribution used in their study has the probability density function:

$$f(x) = \frac{1}{2b} \exp\left\{-\frac{x-a}{b}\right\}, \tag{8}$$

where $a, b > 0$ are two parameters that can be changed to control the mean and the variance. The values of a and b were chosen to be $a = 0.3$ and $b = 1$, which provides a mean value 0.78 with standard deviation 0.93 for the random number with the exponential distribution.

3.3 Non-Linear-Decreasing Inertia Weight

To adjust the search behavior during the initial iterations and later iterations of the search, Chatterjee and Siarry [5] proposed a non-linear adaptive inertia weight for PSO. The non-linear variation of the inertia weight multiplying the old velocity of the particle can improve the speed of convergence and be able to fine-tune the search in the multi-dimensional space. This non-linear inertia weight is given by:

$$w_n = \frac{(n_{max} - n)^r}{n_{max}^r}(w_{initial} - w_{final}) + w_{final}, \tag{9}$$

where $w_{initial}$ and w_{final} are the initial and final values of the inertia weight, n is the current iteration number, n_{max} is the maximum number of iterations, r is the non-linear modulation index.

3.4 Adaptive Inertia Weight

Adaptive inertia weight is based on the influence of a particle on the others according to the effect of attraction towards the global best position, that is, the distance from the particles to the global best position, was proposed by Suresh et al. [6]. At the n-th iteration, the value of inertia weight for particle i is determined by:

$$w_{i,n} = w_0 \left(1 - \frac{d_{i,n}}{D_n}\right), \tag{10}$$

where w_0 is a random number uniformly distributed in the range of $[0.5, 1]$, $d_{i,n}$ is the current Euclidean distance from particle i to the global best position, D_n is the maximum distance from a particle to the global best position at iteration n. That is:

$$D_n = \text{argmax}_i\{d_{i,n}\}. \tag{11}$$

The Euclidean distance $d_{i,n}$ is calculated as follows:

$$d_{i,n} = \sqrt{\sum_{j=1}^{N} \left(G_n^j - X_{i,n}^j\right)^2}. \tag{12}$$

In order to avoid premature convergence in the final stages of the search, Suresh et al. also changed the position update equation to the one given as follows:

$$X_{i,n+1}^j = V_{i,n+1}^j + (1-\rho)X_{i,n}^j, \tag{13}$$

where ρ is a random number uniformly distributed on the range of $(-0.25, 0.25)$.

4 Optimizing Fuzzy Controller for Mobile Robot

Let the mobile robot be initially positioned at pose $x_r(t)$, $\mathrm{y}_r(t)$, $\theta_r(t)$ at time $t = 0$. It is required to move to the target position at x_t, y_t, θ_t, and the target is stationary, i.e, independent of time. The motion command is obtained from two fuzzy inference systems (controller), one for velocity and one for angular turn rate. When the time is advanced from $t\Delta t$ to $(t+1)\Delta t$, the new position can be described by

$$\begin{aligned} x_r(t+1) &= x_t(t) + v(t)\Delta t \cos(\theta_r(t)) \\ y_r(t+1) &= y_t(t) + v(t)\Delta t \sin(\theta_r(t)) \\ \theta_r(t+1) &= \theta(t) + \omega(t)\Delta t \theta_r(t), \end{aligned} \tag{14}$$

where $v(t)$ and $\omega(t)$ are the velocity and angular drive commands respectively.

For illustrative purpose, we consider simple fuzzy controllers. For the velocity controller, it contains one input; the distance between the current car pose and target pose. That is

$$ds = \sqrt{dx^2 + dy^2}, \quad dx = x_t - x_r, \quad dy = y_t - y_r, \tag{15}$$

where the time index is dropped for presentation simplicity. For the angular control, it is obtained from the angular difference

$$d\theta = \Phi(\tan^{-1}(dy, dx) - \theta_r), \tag{16}$$

where $\Phi(\theta)$ is angle wrapping function

$$\Phi(\theta) = \begin{cases} \theta + 2\pi, & \theta < \pi \\ \theta - 2\pi, & \theta > \pi. \end{cases} \tag{17}$$

Let the velocity fuzzy system be 'fis_vel' and angular fuzzy system be 'fis_ang', see Appendix A and B. For a simple exercise, we modify the velocity output membership function 'ZE (zero)', which is a triangular function 'MF1='ZE':'trimf',[0 0 0.2]'. We will try to optimize the upper bound (0.2). For the angular controller, we also modify the output membership 'ZE (zero)', which is a triangular function 'MF2='ZE':'trimf',[-0.7853 0 0.7853]'. We optimize the lower and upper bounds (-0.7853, 0.7853).

Now we consider the application of PSO to this controller optimization problem. Note that the bounds on the angular control is the negation of each other, hence, they can be treated simply as a single variable x_θ. Together with the velocity control bound x_v, we have two variables to be optimized. Therefore, we encode a particle as $\mathbf{x} = [x_v \; x_\theta]^\top$.

In a Matlab program, we first import the fuzzy systems, see Appendix:

```
fis_vel=readfis('MTRN4010_vel.fis');
fis_ang=readfis('MTRN4010_ang.fis');
```

Then from looping through each particle, we modify the corresponding membership bounds by:

```
v=PSO.X(1,n); a=PSO.X(2,n);% velocity and angular bounds
fis_vel.output.mf(1).params(3)=v;% modify vel output Z-MF
fis_ang.output.mf(2).params(3)=a;% modify ang output Z-MF
fis_ang.output.mf(2).params(1)=-a;
```

A simulation run on the mobile robot movement is then conducted. At the end of simulation, the fitness (objective function) is obtained from:

```
err=[car.x-target.x car.y-target.y AngleWrap(car.q-target.q)];% pose error
fit(n)=sqrt(sum(err.^2));% objective function
```

The 'g_{best}' and 'p_{best}' values are determined using:

```
if fit(n)<PSO.gbest,% find global best
  PSO.gbest=fit(n);
  PSO.Gbest=PSO.X(:,n);
end;
if fit(n)<PSO.pbest(n),% find particle best
  PSO.pbest(n)=fit(n);
  PSO.Pbest(:,n)=PSO.X(:,n);
end;
```

Then we calculate particle velocities and positions using the linearly decreasing inertia weight strategy:

```
w=PSO.w2+(1-g/PSO.G)*PSO.dw;% decreasing inertia weight
PSO.V=w*rand(PSO.D,PSO.N).*PSO.V+...
  PSO.cp*rand(PSO.D,PSO.N).*(PSO.Pbest-PSO.X)+...
  PSO.cg*rand(PSO.D,PSO.N).*(repmat(PSO.Gbest,[1,PSO.N])-PSO.X);
PSO.X=PSO.X+PSO.V;
```

Furthermore, we have to repair out-of-bound values:

```
for d=1:PSO.D,
  z=find(PSO.X(d,:)<PSO.BND(d,1));
  PSO.X(d,z)=PSO.BND(d,1)+rand(1,length(z))*diff(PSO.BND(d,:));
  z=find(PSO.X(d,:)>PSO.BND(d,2));
  PSO.X(d,z)=PSO.BND(d,1)+rand(1,length(z))*diff(PSO.BND(d,:));
end;
```

where 'PSO.BND' is the bound. Then the PSO repeats the loop until the stopping iteration is reached.

After obtained the optimal fuzzy system parameters, we can simulate the mobile robot motion using the found parameters. The complete example code is given in Appendix C.

Appendix

A Velocity Fuzzy Control

```
[System]
Name='MTRN4010_vel'
Type='mamdani'
Version=2.0
NumInputs=1
NumOutputs=1
NumRules=3
AndMethod='min'
OrMethod='max'
ImpMethod='min'
AggMethod='max'
DefuzzMethod='mom'

[Input1]
Name='ds'
Range=[0 20]
NumMFs=3
MF1='ZE':'trimf',[-Inf 0 4]
```

```
MF2='PM':'trimf',[4 10 16]
MF3='PB':'trimf',[10 20 200]

[Output1]
Name='vel'
Range=[0 0.5]
NumMFs=3
MF1='ZE':'trimf',[0 0 0.2]
MF2='PM':'trimf',[0.1 0.25 0.4]
MF3='PB':'trimf',[0.3 0.5 0.5]

[Rules]
1, 1 (1) : 1
2, 2 (1) : 1
3, 3 (1) : 1
```

B Angular Fuzzy Control

```
[System]
Name='MTRN4010_ang'
Type='mamdani'
Version=2.0
NumInputs=1
NumOutputs=1
NumRules=3
AndMethod='min'
OrMethod='max'
ImpMethod='min'
AggMethod='max'
DefuzzMethod='centroid'

[Input1]
Name='dq'
Range=[-3.14159265358979 3.14159265358979]
NumMFs=3
MF1='NB':'trimf',[-5.655 -3.142 -0.6283]
MF2='ZE':'trimf',[-2.513 0 2.513]
MF3='PB':'trimf',[0.6283 3.142 5.655]

[Output1]
Name='ang'
Range=[-3.14159265358979 3.14159265358979]
NumMFs=3
MF1='NB':'trimf',[-6.283 -3.142 1.571]
```

```
MF2='ZE':'trimf',[-0.7853 0 0.7853]
MF3='PB':'trimf',[-1.571 3.14 6.28]

[Rules]
2, 2 (1) : 1
1, 1 (1) : 1
3, 3 (1) : 1
```

C Complete Sample Matlab Code

```
function []=MTRN4010_91()
% sim car movement to target
% fuzzy control on velocity/angle
% PSO optimize zero membership function parameters
% modify vel.fis output membership ZERO upper bound
% modify ang.fis output membership ZERO lower/upper bound
% obective function is square rooted sum of pose error squares
% because of many simulations, the programm take a longer time to finish

% program management
clc; clear all; close all; dbstop if error;
set(0,'defaultaxesfontname','times new roman');

% car motion filed, sim time
field.range=50;
time.dt=1; time.T=500;

% car variables
[fig]=FigureNew(field);
[car]=CarNew(field);
[car]=CarNow(car,time,0,0);
[car]=CarShow(fig,car,0);
[target]=TargetNew(field);

% import fuzzy system
fis_vel=readfis('MTRN4010_vel.fis');
fis_ang=readfis('MTRN4010_ang.fis');

% PSO parameters
PSO.VLB=0.1; PSO.VUB=2;% car velocity lower/upper bound
PSO.ALB=0.1*pi; PSO.AUB=0.8*pi;% car angle lower/upper bound
PSO.D=2; PSO.G=20; PSO.N=30;% particle dimension generations, particles
PSO.V=rand(PSO.D,PSO.N);% initial PSO particle velocity
PSO.Gbest=[]; PSO.gbest=realmax;% PSO gbest
```

```
PSO.Pbest=[]; PSO.pbest=ones(1,PSO.N)*realmax;% PSO pbest
PSO.w1=0.9; PSO.w2=0.4; PSO.dw=PSO.w1-PSO.w2;% init and final inertia weight
PSO.cg=2; PSO.cp=2;% social, congnitive factor
PSO.X=rand(PSO.D,PSO.N); PSO.BND=[PSO.VLB PSO.VUB; PSO.ALB PSO.AUB];% PSO bound
for d=1:2,% particles for car vel/angle Z-MF
  PSO.X(d,:)=PSO.BND(d,1)+PSO.X(d,:)*diff(PSO.BND(d,:));
end;

% PSO main loop
carInit=car;% initial car pose
for g=1:PSO.G,% PSO generation
  disp(PSO.X);% particles
  car=carInit;% restore init car pose
  for n=1:PSO.N,% PSO particles
    v=PSO.X(1,n); a=PSO.X(2,n);% velocity and angular bounds
    fis_vel.output.mf(1).params(3)=v;% modify vel output Z-MF
    fis_ang.output.mf(2).params(3)=a;% modify ang output Z-MF
    fis_ang.output.mf(2).params(1)=-a;
    % sim car movement
    for t=0:time.dt:time.T,
      [ds]=FindDistance(car,target);
      vel=evalfis(ds,fis_vel);
      [dq]=FindAngular(car,target);
      ang=evalfis(dq,fis_ang);
      [car]=CarNow(car,time,vel,ang);
    end;
    err=[car.x-target.x car.y-target.y AngleWrap(car.q-target.q)];% pose error
    fit(n)=sqrt(sum(err.^2));% objective function
    if fit(n)<PSO.gbest,% find global best
      PSO.gbest=fit(n);
      PSO.Gbest=PSO.X(:,n);
    end;
    if fit(n)<PSO.pbest(n),% find particle best
      PSO.pbest(n)=fit(n);
      PSO.Pbest(:,n)=PSO.X(:,n);
    end;
  end;
  disp(sprintf('Generation %d Gbest %5.3f %5.3f gbest %5.3f',...
    g,PSO.Gbest.',PSO.gbest));% currently best results
  % PSO update
  w=PSO.w2+(1-g/PSO.G)*PSO.dw;% decreasing inertia weight
  PSO.V=w*rand(PSO.D,PSO.N).*PSO.V+...
    PSO.cp*rand(PSO.D,PSO.N).*(PSO.Pbest-PSO.X)+...
    PSO.cg*rand(PSO.D,PSO.N).*(repmat(PSO.Gbest,[1,PSO.N])-PSO.X);
  PSO.X=PSO.X+PSO.V;
  % repare particles
```

```
  for d=1:PSO.D,
    z=find(PSO.X(d,:)<PSO.BND(d,1));
    PSO.X(d,z)=PSO.BND(d,1)+rand(1,length(z))*diff(PSO.BND(d,:));
    z=find(PSO.X(d,:)>PSO.BND(d,2));
    PSO.X(d,z)=PSO.BND(d,1)+rand(1,length(z))*diff(PSO.BND(d,:));
  end;
end;

% show result with optimized controller
fis_vel.output.mf(1).params(3)=PSO.Gbest(1);% modify vel output Z-MF
fis_ang.output.mf(2).params(3)= PSO.Gbest(2);% modify ang output Z-MF
fis_ang.output.mf(2).params(1)=-PSO.Gbest(2);
car=carInit;% restore initial car pose
for t=0:time.dt:time.T,
  [ds]=FindDistance(car,target);
  vel=evalfis(ds,fis_vel);
  [dq]=FindAngular(car,target);
  ang=evalfis(dq,fis_ang);
  [car]=CarNow(car,time,vel,ang);
  [car]=CarShow(fig,car,t);
end;



function [fig]=FigureNew(field)
fig.fig=figure('units','normalized','position',[0.3 0.2 0.5 0.5]);
axis([-1 1 -1 1]*field.range); hold on; grid on; axis equal;
xlabel('x-direction'); ylabel('y-direction');
fig.ax=axis;

function [target]=TargetNew(field)
target.x=(rand-0.5)*field.range;
target.y=(rand-0.5)*field.range;
target.q=AngleWrap(rand*2*pi);
target.shape=[ 2 0; 1 1; -1 1; -1 -1; 1 -1; 2 0]';
target.hdL.shape=plot(target.shape(1,:),target.shape(2,:),...
  'color','r','linewidth',2);
Rz=[  cos(target.q) -sin(target.q);
      sin(target.q)  cos(target.q)];
shape=Rz*target.shape+repmat([target.x;target.y],1,6);
set(target.hdL.shape,'xdata',shape(1,:),'ydata',shape(2,:));

function [ds]=FindDistance(car,target)
dx=target.x-car.x;
dy=target.y-car.y;
ds=sqrt(dx^2+dy^2);
```

```
function [dq]=FindAngular(car,target)
dx=target.x-car.x;
dy=target.y-car.y;
q=atan2(dy,dx)-car.q;
dq=AngleWrap(q);

function [q]=AngleWrap(q)
while q<pi,
  q=q+2*pi;
end;
while q>pi,
  q=q-2*pi;
end;

function [car]=CarNew(field)
car.x=(rand-0.5)*field.range*3;
car.y=(rand-0.5)*field.range*3;
car.q=AngleWrap(randn*2*pi);
car.trace=[car.x; car.y; car.q];
car.shape=[ 2 0; 1 1; -1 1; -1 -1; 1 -1; 2 0]';
car.hdL.shape=plot(car.shape(1,:),car.shape(2,:),'color','b','linewidth',2);
car.hdL.trace=plot(car.trace(1,:),car.trace(2,:),'color',[0 0.66 0]);

function [car]=CarNow(car,time,v,w)
car.x=car.x+time.dt*v*cos(car.q);
car.y=car.y+time.dt*v*sin(car.q);
car.q=car.q+time.dt*w; car.q=AngleWrap(car.q);
car.trace(:,end+1)=[car.x; car.y; car.q];

function [car]=CarShow(fig,car,t)
Rz=[  cos(car.q) -sin(car.q);
      sin(car.q)  cos(car.q)];
shape=Rz*car.shape+repmat([car.x;car.y],1,6);
set(car.hdL.shape,'xdata',shape(1,:),'ydata',shape(2,:));
set(car.hdL.trace,'xdata',car.trace(1,:),'ydata',car.trace(2,:));
axis(fig.ax); title(sprintf('Time %d',t)); pause(0.001);
```

References

[1] R. Eberhart and J. Kennedy, "A new optimizer using particle swarm theory," in *Micro Machine and Human Science, 1995. MHS'95., Proceedings of the Sixth International Symposium on*, pp. 39–43, IEEE, 1995.

[2] Y. Shi and R. C. Eberhart, "Parameter selection in particle swarm optimization," in *International conference on evolutionary programming*, pp. 591–600, Springer, 1998.

[3] R. C. Eberhart and Y. Shi, "Tracking and optimizing dynamic systems with particle swarms," in *Evolutionary Computation, 2001. Proceedings of the 2001 Congress on*, vol. 1, pp. 94–100, IEEE, 2001.

[4] M. Pant, T. Radha, and V. Singh, "Particle swarm optimization using gaussian inertia weight," in *Conference on Computational Intelligence and Multimedia Applications, 2007. International Conference on*, vol. 1, pp. 97–102, IEEE, 2007.

[5] A. Chatterjee and P. Siarry, "Nonlinear inertia weight variation for dynamic adaptation in particle swarm optimization," *Computers & operations research*, vol. 33, no. 3, pp. 859–871, 2006.

[6] K. Suresh, S. Ghosh, D. Kundu, A. Sen, S. Das, and A. Abraham, "Inertia-adaptive particle swarm optimizer for improved global search," in *Intelligent Systems Design and Applications, 2008. ISDA'08. Eighth International Conference on*, vol. 2, pp. 253–258, IEEE, 2008.