Term Project #1: AI Soccer

Term Project #1

- AI Soccer
  - RED Team (5 robots) vs BLUE Team (5 robots)
  - Webots simulator + Simulation management program (C++) are given.
- Goal
  - Provide control commands (strategy) for 5 robots to win the game
  - The controller program can either be C++ or Python.
  - The commands from controller are transferred to robots at regular intervals.
  - Use what you will learn during the class or implement it along with other algorithms, if needed
- Team composition
  - 2 or 3 students per team
  - Team members are freely decided.
- Submission
  - Source code for competition along with a technical report
- The competition will be held with a full-league style.
  - Each team will have 5-minute games against all other teams.
  - Total # of wins in the competition will be used as one of the grading criteria.

How to Install

- Ubuntu (16.04 or 18.04, recommend using 16.04) should be used.

- Download Webots R2019b from (.deb file)
  - https://cyberbotics.com/download

- Type following lines in Ubuntu terminal to install Webots
  - sudo dpkg -i webots_2019b_amd64.deb
  - sudo apt-get install -f –y
  - Refer to https://www.cyberbotics.com/doc/guide/installation-procedure

- Install required packages
  - https://github.com/aiwc/test_world/wiki/How-to-use-AI-World-Cup-simulation-program

- Download EE682_AI_Soccer.zip from klms
  - Go to /EE682_AI_Soccer/worlds and type
    - webots aiwc.wbt

Basic Information

- Several data are provided from the simulator to the player controller program
  - Constants: Data that are constant throughout the game (field dimensions, robot specifications, etc.)
  - Variables: Data that change throughout the game (scores, field image, robot positions, etc.)
  - The list of data provided is on this ppt.
    - For detailed usage, refer to the provided sample source codes (mainly refer to /EE682_AI_Soccer/examples/player_rulebased_{cpp, py}/)

- Usable data for players (constants throughout a game)
  - game_time: game time duration (sec)
  - goal: width, height of goal area (m)
  - number_of_robots: number of robots
  - penalty_area: width, height of penalty area (m)
  - codewords: 9-bit identity of each robot
  - robot_height: height of robot (m)
  - robot_radius: radius of robot (m)
  - max_linear_velocity: maximum velocity (m/s)
  - field: width, height of field (m)
  - team_info {rating, name}: team name (rating has no information)
  - axle_length: distance between two wheels on each robot (m)
  - resolution: resolution of upper image
  - ball_radius: radius of ball (m)
  - max_meters_run: maximum distance that each robot can run (m)
  - Refer to https://github.com/aiwc/test_world/wiki/Auxiliary-Information for graphical representation

- time: time passed since the beginning of the game (sec)
  - f.time (C++), received_frame.time (Python)

- score: current score of two teams
  - [0]: my team's score, [1]: opponent team's score
  - f.score[TEAM] (C++), received_frame.score[TEAM] (Python)

- reset_reason: signal giving information about current game status
  - (NONE, GAME_START, SCORE_MYTEAM, SCORE_OPPONENT, GAME_END, DEADLOCK)
  - f.reset_reason (C++), received_frame.reset_reason (Python)

- subimages: current frame's image segment data
  - Refer to general_image-fetch_cpp sample source code for the usage

- coordinates: other numeric information (robots and ball coordinates, orientation, etc.)
  - Refer to (player_rulebased_{cpp, py} source code for detailed usage)

- (*f.opt_coordinates).ball (C++), received_frame.coordinates[BALL] (Python)
  - Ball coordinates
  - x, y: current (x, y) position of the ball (m)

- (*f.opt_coordinates).robots[team_id][robot_id] (C++), received_frame.coordinates[team_id][robot_id] (Python)
  - Robot status
  - x, y: current (x, y) position of the robot (m)
  - th: current orientation of the robot (rad)
  - active: whether the robot is currently active or inactive (bool)
    (a robot is inactive when it is sent out of game due to fouls or it has moved for max_meters_run meters)
  - touch: whether the robot is currently in contact with the ball or not (bool)
  - meters_run: total meters the robot run since the beginning of the game (m)

- The team controller program can set wheel speeds of 5 robots in the team to move the robots
  - Wheel speed is in m/s (linear velocity).
  - set_wheel(wheels) method is used to send wheel speeds to the simulator.
    - wheels: 1-D array of 10 wheel speeds (id: 0, 1, 2, 3, 4)*(left, right)

- Game rules
  - Basic rules: https://github.com/aiwc/test_world/wiki/Game-Rules

Sample Code

- Samples are provided on /EE682_AI_Soccer/examples/ directory
- player_rulebased.{cpp, py}
  - A sample code which makes robots move with a simple rule-based strategy
  - You can modify this code to make a rule-based controller.

- play_deep-learning-train.py
  - A sample code which trains only one robot with DQN
  - You can modify this code to train a team controller.

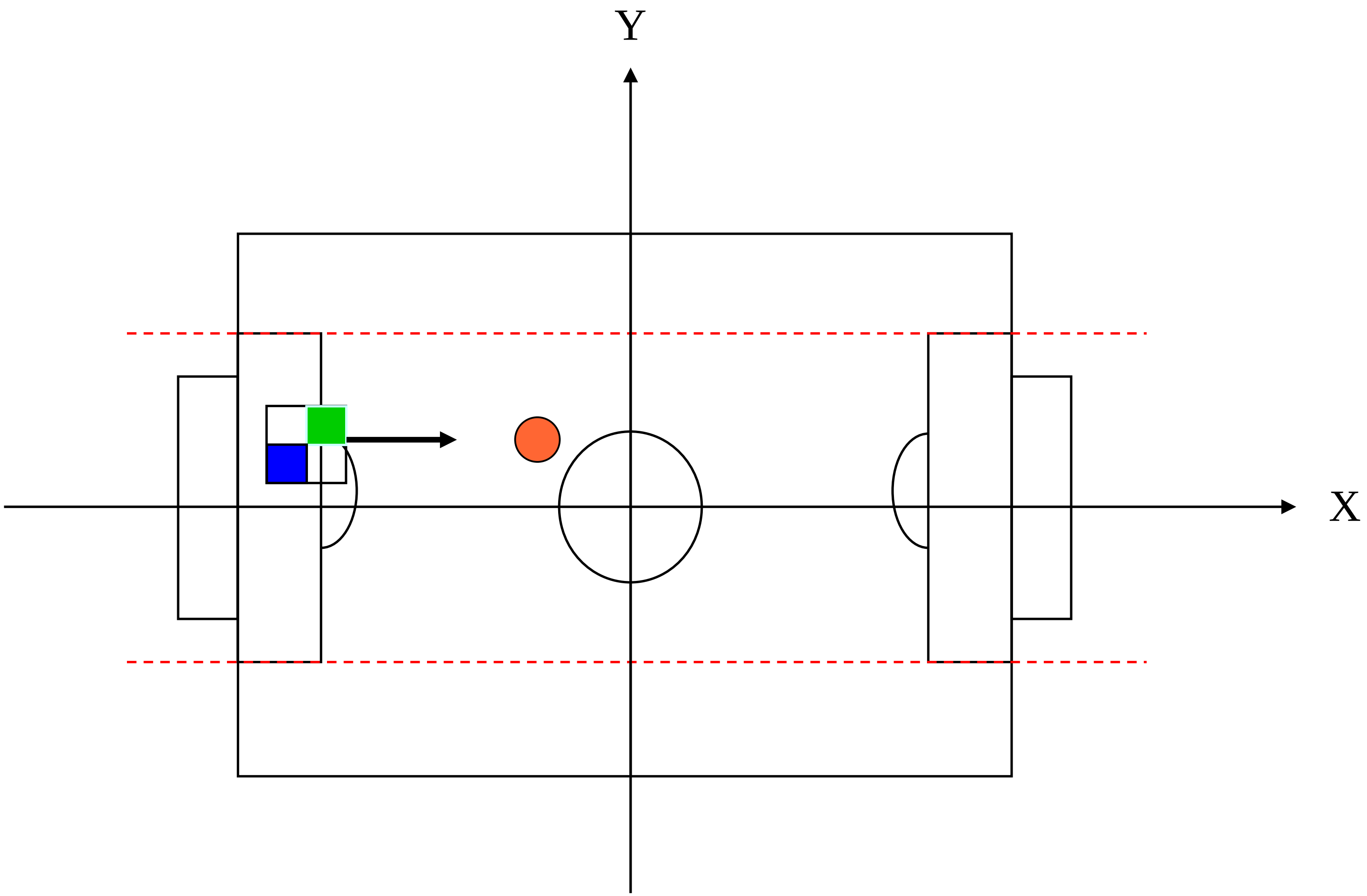
- play_deep-learning-play.py
  - A sample code which makes your robot move with the trained DQN

- Modify /EE682_AI_Soccer/config.json to test different sample programs or your own program

- Example strategy: player_rulebased.py

Goalie

- The goalie only tracks the y-coordinate of the ball position
  - It will keep its x-coordinate fixed within the goal area.

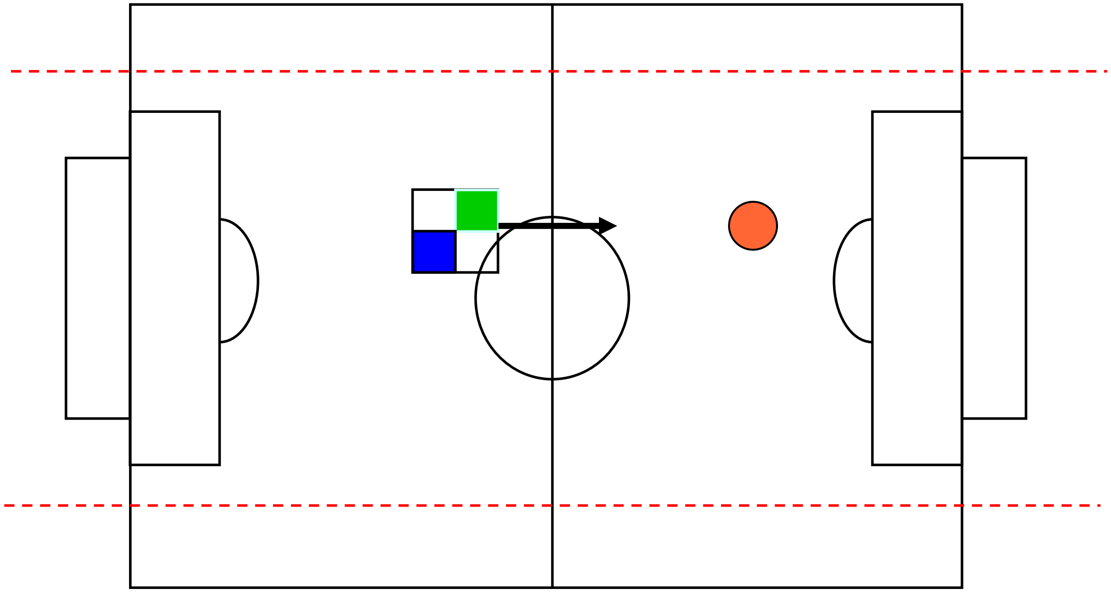


- The goalie only tracks the y-coordinate of the ball position
  - It will keep its x-coordinate fixed within the goal area.

```python
def goalie(self, robot_id):
    # Goalie just track the ball[Y] position at a fixed position on the X axis
    x = (-self.field[X]/2) + (self.robot_size/2) + 0.05
    y = max(min(self.cur_ball[Y], (self.goal[Y]/2 - self.robot_size/2)), -self.goal[Y]/2 + self.robot_size/2)
    self.position(robot_id, x, y)
```

1) Defender

- If the ball is in the opponent side of the field
  - x: the robot locates itself at the middle of the ball and ally goalpost's positions.
  - y: the robot tracks y position of the ball, but only up to 1/3 vertically away from the center (with small offset)



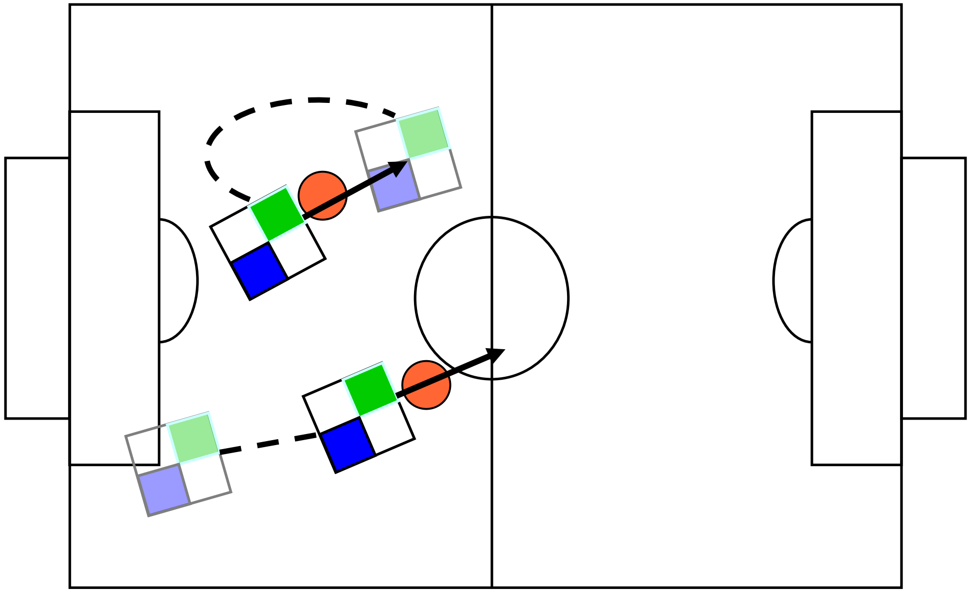
- If the ball is in the opponent side of the field
  - x: the robot locates itself at the middle of the ball and ally goalpost's positions.
  - y: the robot tracks y position of the ball, but only up to 1/3 vertically away from the center (with small offset)

```
def defender(self, robot_id, idx, offset_y):
    ox = 0.1
    oy = 0.075
    min_x = (-self.field[X]/2) + (self.robot_size/2) + 0.05

    # If ball is on offense
    if (self.cur_ball[X] > 0):
        # If ball is in the upper part of the field (y>0)
        if (self.cur_ball[Y] > 0):
            self.position(robot_id,
                          (self.cur_ball[X]-self.field[X]/2)/2,
                          (min(self.cur_ball[Y],self.field[Y]/3))+offset_y)
        # If ball is in the lower part of the field (y<0)
        else:
            self.position(robot_id,
                          (self.cur_ball[X]-self.field[X]/2)/2,
                          (max(self.cur_ball[Y],-self.field[Y]/3))+offset_y)
```

2) Defender

- If the ball is in the ally side of the field
  - If the robot is in front of the ball, the robot goes around the ball
  - If the robot is behind the ball, the robot pushes the ball

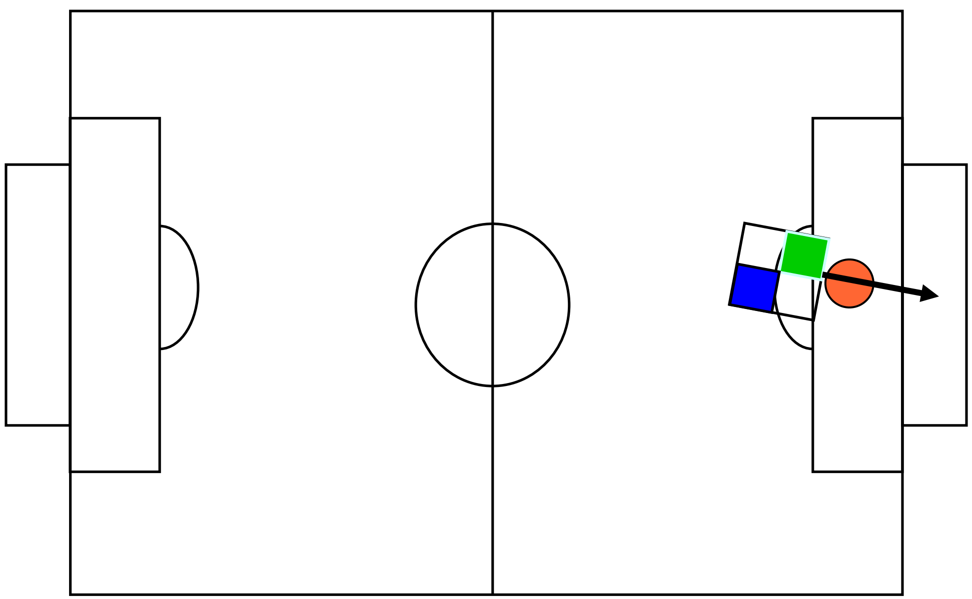


- If the ball is in the ally side of the field
  - If the robot is in front of the ball, the robot goes around the ball
  - If the robot is behind the ball, the robot pushes the ball

```
else:
    # If robot is in front of the ball
    if (self.cur_posture[robot_id][X] > self.cur_ball[X] - ox):
        # If this defender is the nearest defender from the ball
        if (robot_id == idx):
            self.position(robot_id,
                          (self.cur_ball[X]-ox),
                          ((self.cur_ball[Y]+oy) if (self.cur_posture[robot_id][Y]<0) else (self.cur_ball[Y]-oy)))
        else:
            self.position(robot_id,
                          (max(self.cur_ball[X]-0.03, min_x)),
                          ((self.cur_posture[robot_id][Y]+0.03) if (self.cur_posture[robot_id][Y]<0) else (self.cur_posture[robot_id][Y]-0.03)))
    # If robot is behind the ball
    else:
        if (robot_id == idx):
            self.position(robot_id,
                          self.cur_ball[X],
                          self.cur_ball[Y])
        else:
            self.position(robot_id,
                          (max(self.cur_ball[X]-0.03, min_x)),
                          ((self.cur_posture[robot_id][Y]+0.03) if (self.cur_posture[robot_id][Y]<0) else (self.cur_posture[robot_id][Y]-0.03)))
```

1) Midfielder

- The robot closest to the ball
  - If the robot is the closest to the ball and the opponent goalpost is near,
    - The robot moves towards the goalpost.



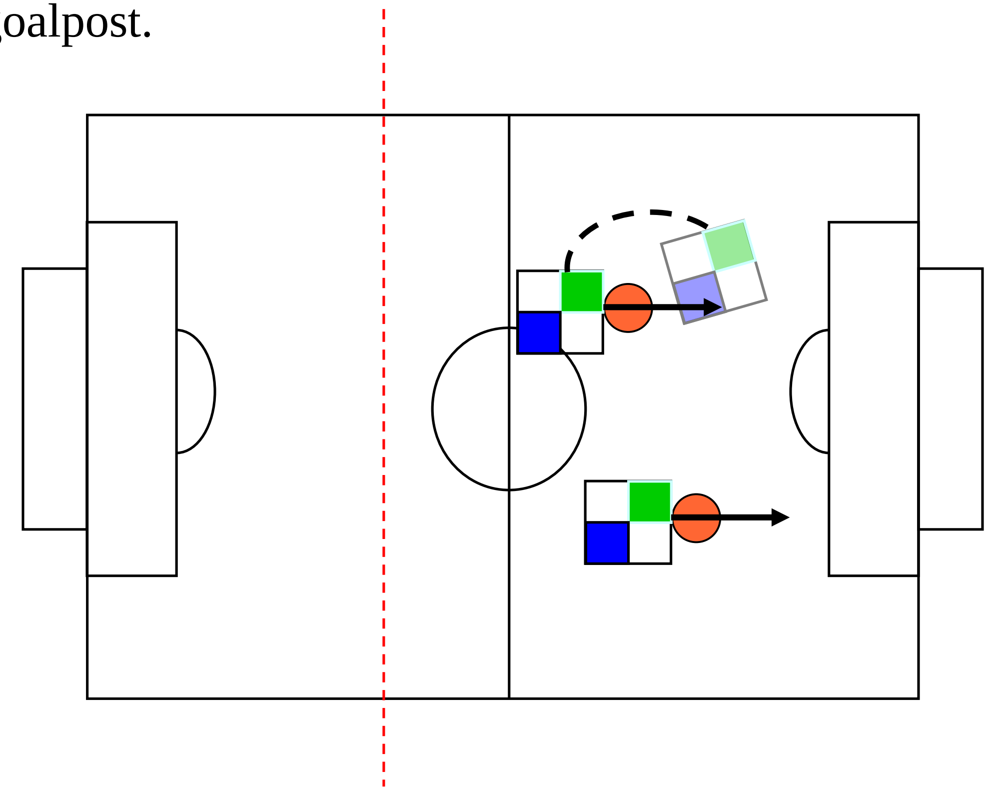
- The robot closest to the ball
  - If the robot is the closest to the ball and the opponent goalpost is near,
    - The robot moves towards the goalpost.

```
def midfielder(self, robot_id, idx, offset_y):
    ox = 0.1
    oy = 0.075
    ball_dist = helper.distance(self.cur_posture[robot_id][X], self.cur_ball[X], self.cur_posture[robot_id][Y], self.cur_ball[Y])
    goal_dist = helper.distance(self.cur_posture[robot_id][X], self.field[X]/2, self.cur_posture[robot_id][Y], 0)

    if (robot_id == idx):
        if (ball_dist < 0.04):
            # if near the ball and near the opposite team goal
            if (goal_dist < 1.0):
                self.position(robot_id, self.field[X]/2, 0)
```

2) Midfielder

- The robot closest to the ball
  - If the robot is close only to the ball,
    - If the robot is in front of the ball, the robot goes around the ball.
    - If the robot is behind the ball, the robot moves towards the opponent goalpost.

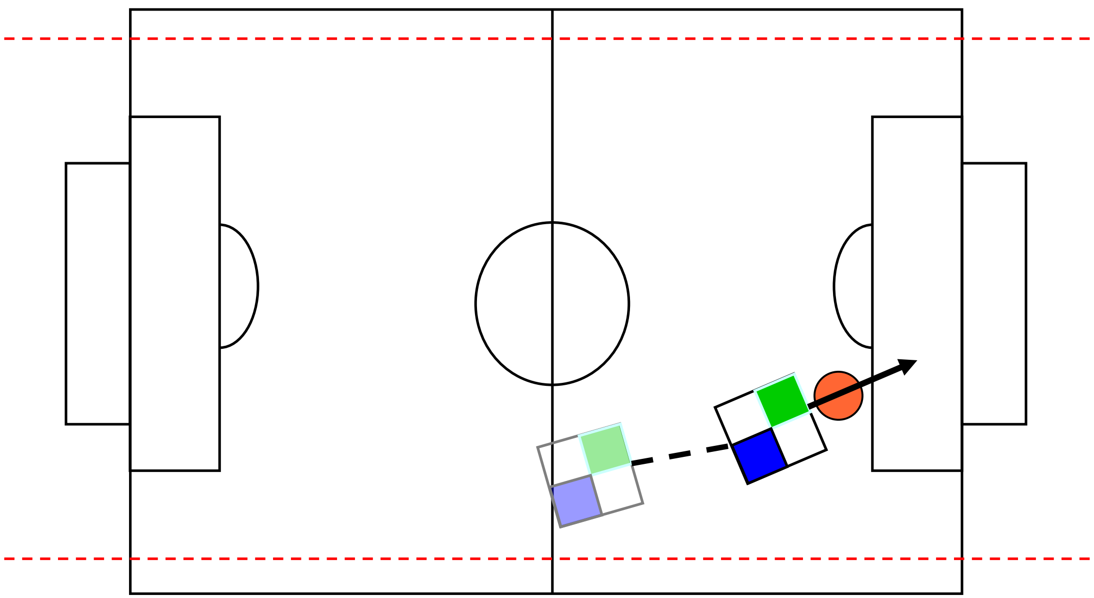


- Nearest robot to the ball
  - If the robot is close only to the ball
    - If the robot is in front of the ball, make the robot go behind the ball
    - If the robot is behind the ball, move towards the goalpost of its opposite team

```
else:
    # if near and in front of the ball
    if (self.cur_ball[X] < self.cur_posture[robot_id][X] - 0.044):
        x_suggest = max(self.cur_ball[X] - 0.044, -self.field[X]/6)
        self.position(robot_id, x_suggest, self.cur_ball[Y])
    # if near and behind the ball
    else:
        self.position(robot_id, self.field[X] + self.goal[X], -self.goal[Y]/2)
```

3) Midfielder

- Robots that are not the closest to the ball
  - The robot only tracks the ball roughly without trying to kick the ball



- Robots that are not the closest to the ball
  - The robot only tracks the ball roughly without trying to kick the ball

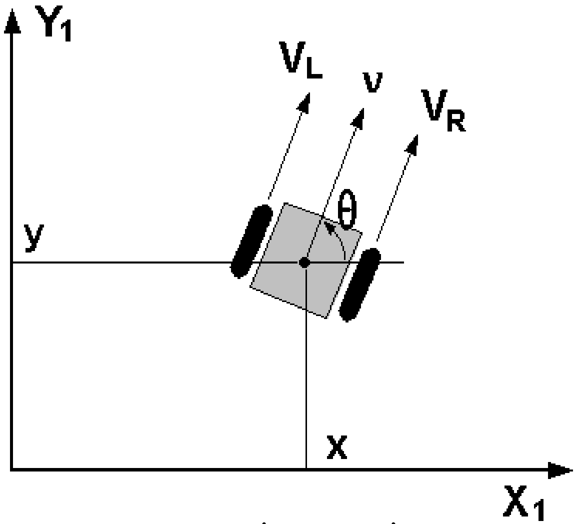
```
else:
    self.position(robot_id, max(self.cur_ball[X]-0.1, -0.3*self.field[Y]), self.cur_ball[Y]+offset_y)
```

Appendix

Robot Posture

Robot Posture

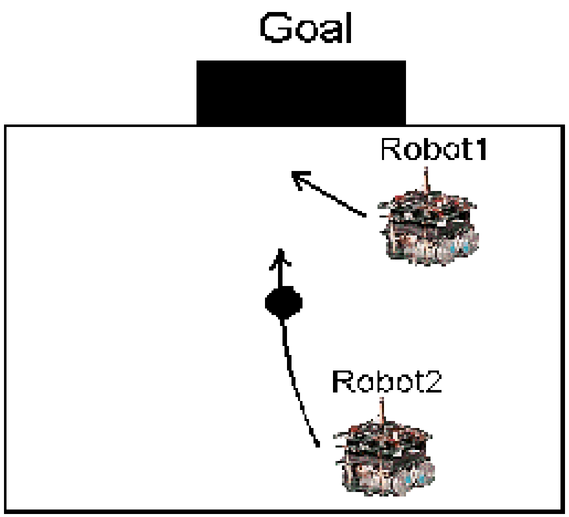
- Posture = (position, heading angle)
  - Position (x, y): center of the robot
  - Heading angle θ: the angle from X_1 to the robot head in CCW direction



Posture (x, y ,q)

Robot Motion Control

- Approaching a target point (destination)
  - Blocking the ball or opponent robots
- Following a trajectory (path)
  - Shooting, kicking, dribbling, etc.

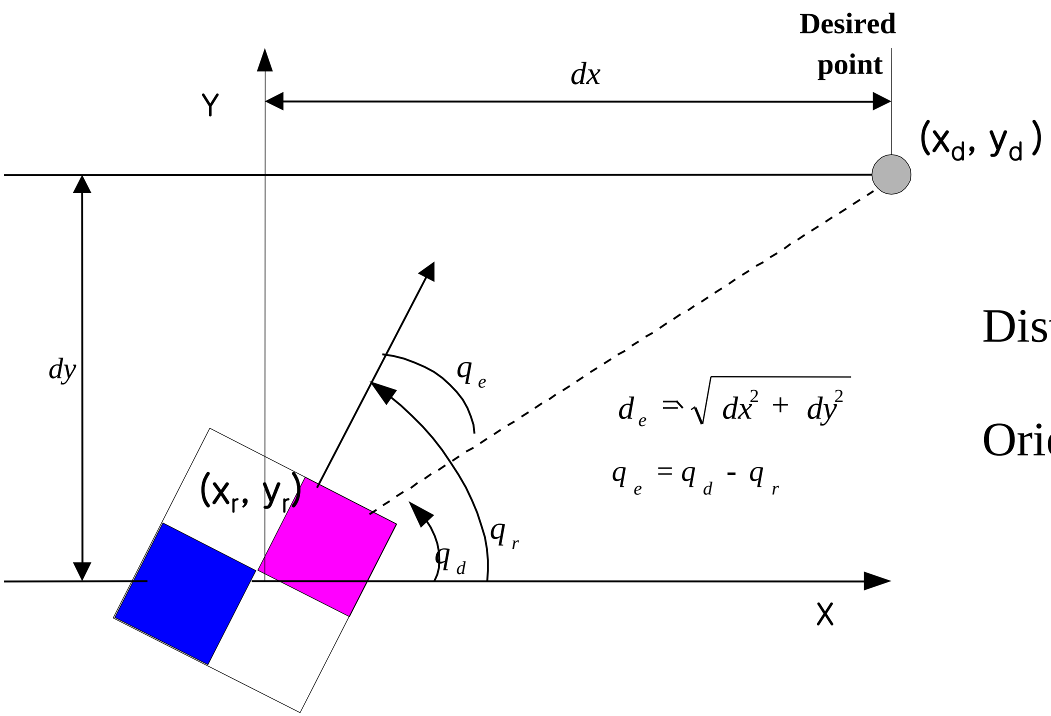


Robot1: Blocking the ball (point)

Robot2: Shooting (path)

Control to a Target Point

- Posture error is used
  - Posture error = desired posture - current posture
  - Desired posture: (x_d, y_d, θ_d)
  - Current posture: (x_r, y_r, θ_r)



Distance error: $d_e = \sqrt{d_x^2 + d_y^2}$

Orientation error: $\theta_e = \theta_d - \theta_r$

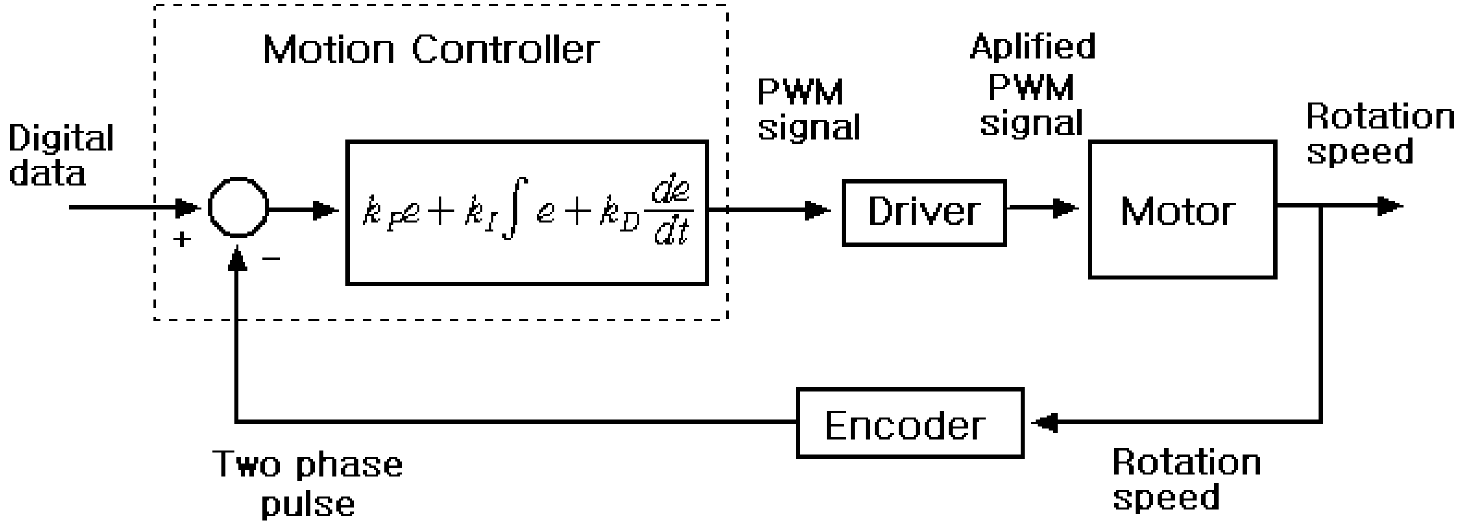
- Control objective:
  - To generate v_L, v_R in such a way that the posture error vanishes
  - We are interested only in distance error in this case,

$$d_e = \sqrt{d_x^2 + d_y^2}$$

- Most popular control algorithms:
  - PID control
  - Fuzzy control

P(ID) Controller

- PID control



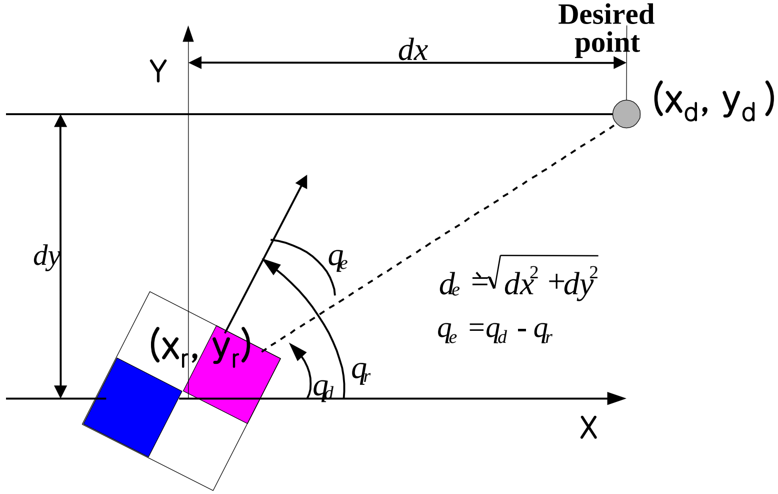
- Basic idea
  - If error is big, control input should be big.
  - If error is small, control input should be small.

- **Posture error of a mobile robot**
  - **Distance error, d_e**
  - **Orientation error, q_e**

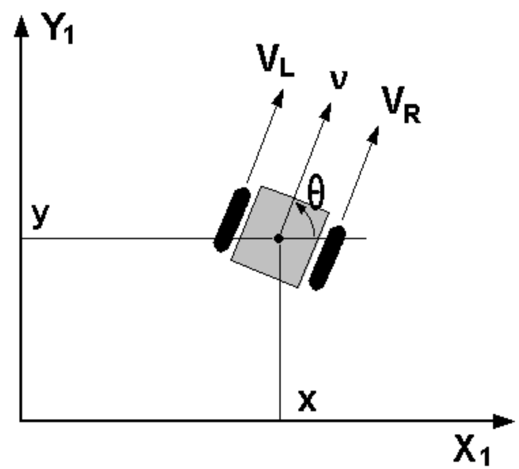
- **Control input *u* of a mobile robot**
  - **Linear velocity, *v***
  - **Angular velocity, *w***

$$d_e = \sqrt{d_x^2 + d_y^2} \qquad \theta_e = \theta_d - \theta_r$$

$$(x_r, y_r, \theta_r)$$



$$\omega = \frac{V_R - V_L}{L} \qquad v = \frac{V_R + V_L}{2}$$



- Proportional (P) control:

$$v = k_d d_e,$$
$$\omega = k_a \theta_e$$

- By the relation of (v, w) and (V_L, V_R),

$$v_L = v - \frac{L}{2}\omega \;= k_d d_e - \frac{L}{2} k_a \theta_e$$
$$v_R = v + \frac{L}{2}\omega \;= k_d d_e + \frac{L}{2} k_a \theta_e$$

- By replacing k_d by K_d and $L/2 \times k_a$ by K_a,

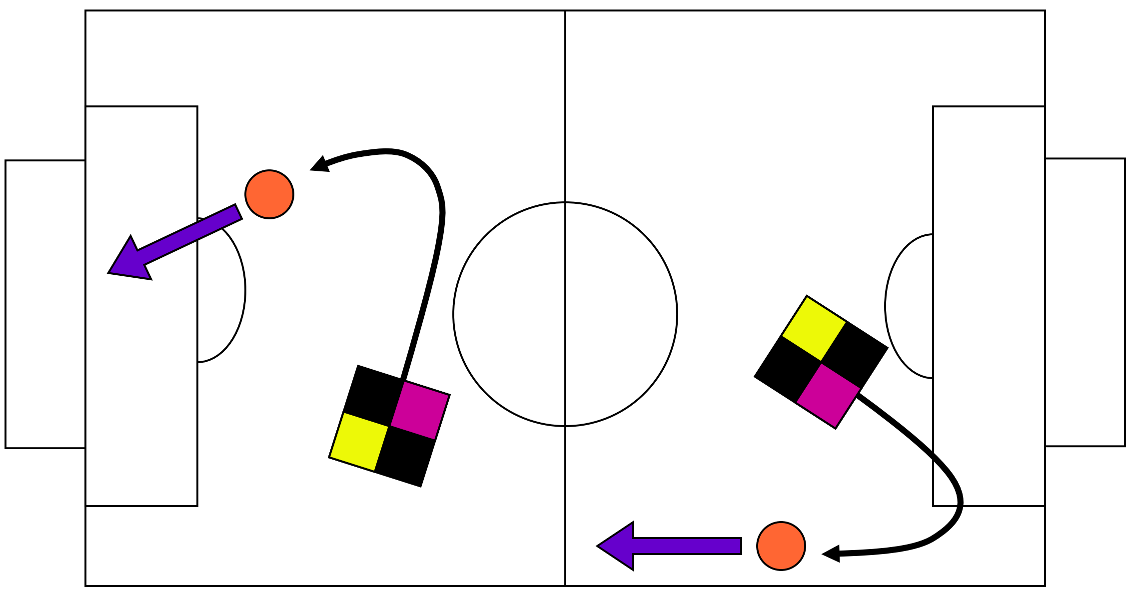
$$v_L = K_d d_e - K_a \theta_e$$
$$v_R = K_d d_e + K_a \theta_e$$

- Selection of P gains (K_d, K_a)
  - First, select K_a :
    - Set $K_d = 0$,
    - Find K_a such that the robot turns smoothly without oscillation, if angle error exists.
  - Then, select K_d
    - Find such K_d that the robot moves to the target point smoothly with a proper speed

Kick

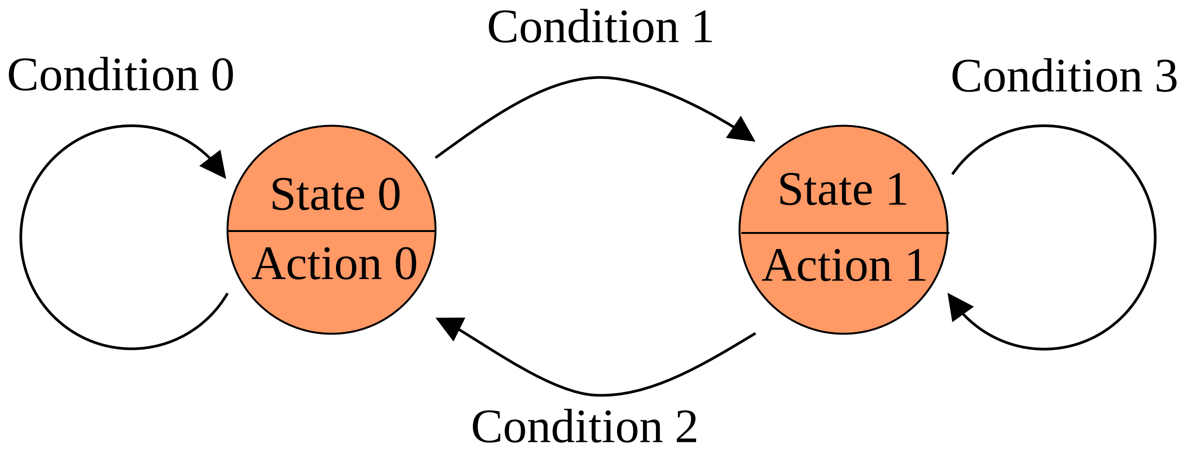
Kick Function

- Kick function
  - To make the robot kick the ball towards the goal
  - Trade-off between speed and accuracy
  - Playfield boarder should be considered.

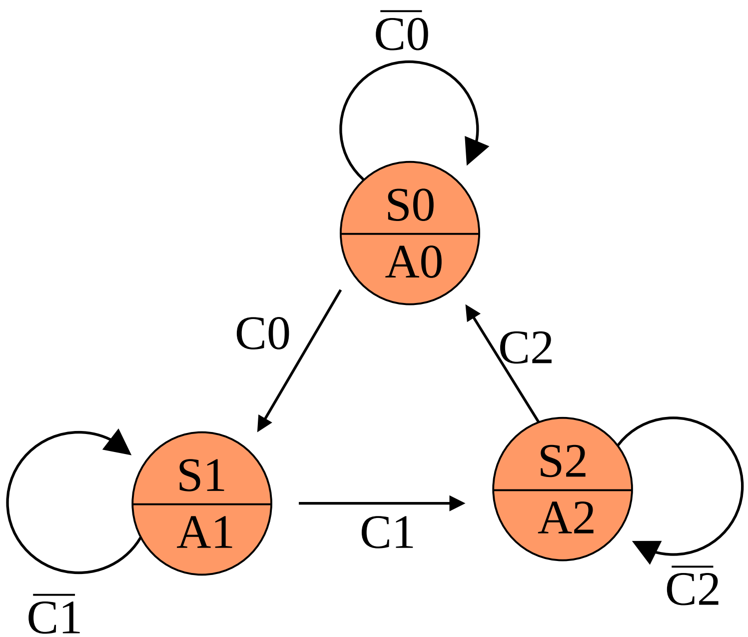


State Diagram

- State machine
  - An imaginary machine that stores the status at a given time and can operate on input or condition to change the status and/or cause an action or output.
  - It can be represented by a state diagram.



- Kick function using a state machine




```
switch(flag){
    case 0: //S0 state
        Do A0;
        If C0, then move to S1;
        break;
    case 1: //S1 state
        Do A1;
        If C1, then move to S2;
        break;
    case 2: //S2 state
        Do A2;
        If C2, then move to S0;
        break;
}
```

S0: Far from the ball
S1: Behind the ball
S2: Kicking the ball

A0: Move behind the ball
A1: Turn to the destination
A2: Kick the ball

C0: When robot has arrived at the position behind the ball
C1: When robot directs to the destination
C2: When robot is out of the kickable region

Kick Function Using a State Machine

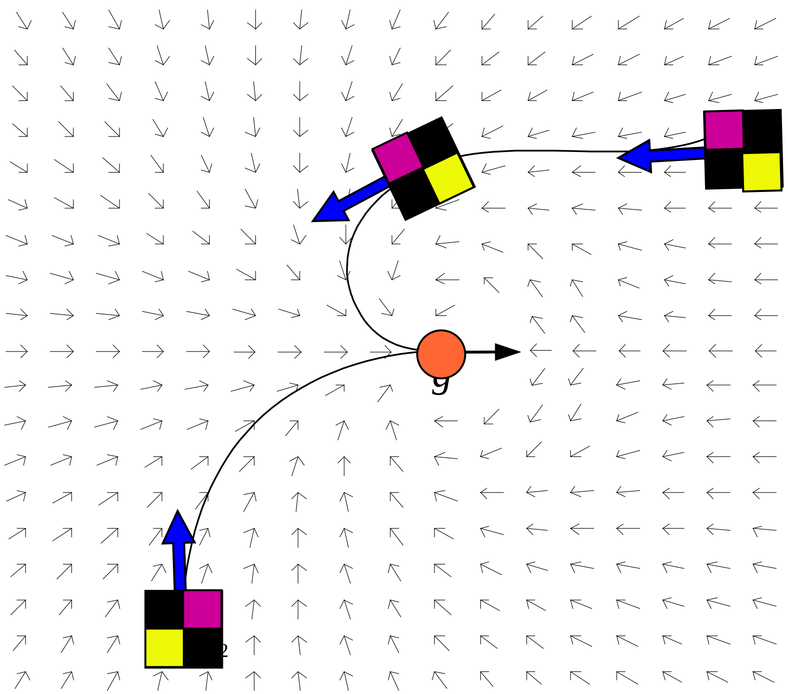
- Pseudo code of Kick()

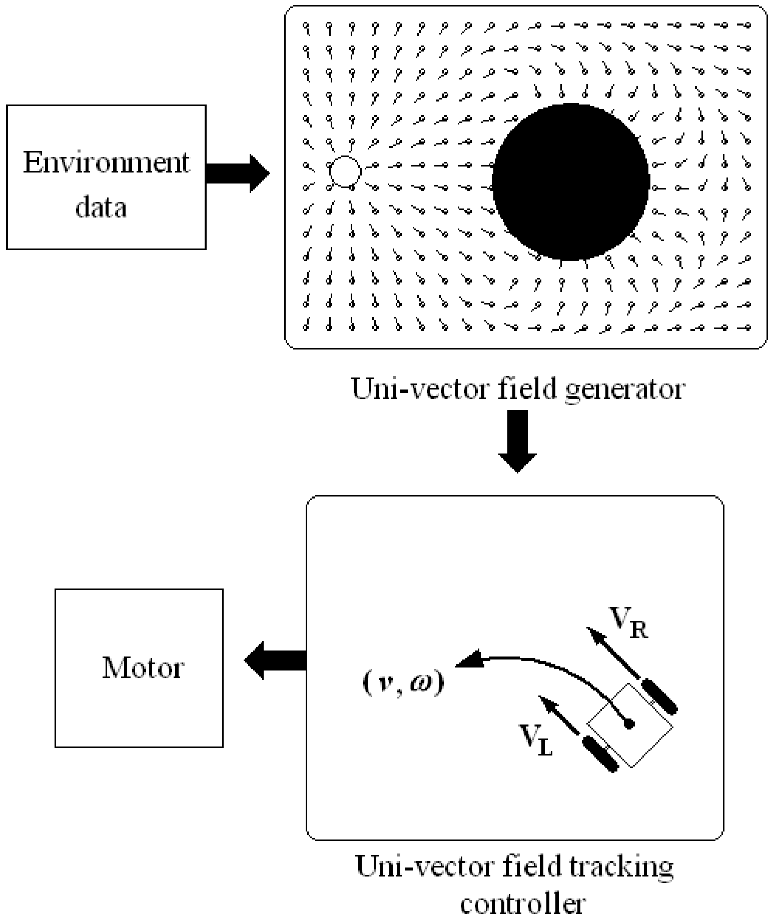
```
void Kick(int whichrobot)
{
        static int flag;
        //Kicking direction
        Set the kicking direction to the goal;
        Near the field boarder, change the direction to avoid collision;


        switch(flag){
        case 0: //S0 state
                Go behind the ball by  using "Position()";
                If the robot has arrived, move to State1;
                break;
        case 1: //S1 state
                Rotate to the destination by using "Angle()";
                If the robot directs to the destination, move to State2;
                break;
        case 2: //S2 state
                Kick the ball by using "Position()";
                If the robot is out of the kickable region, move to State0;
        break;
}
```

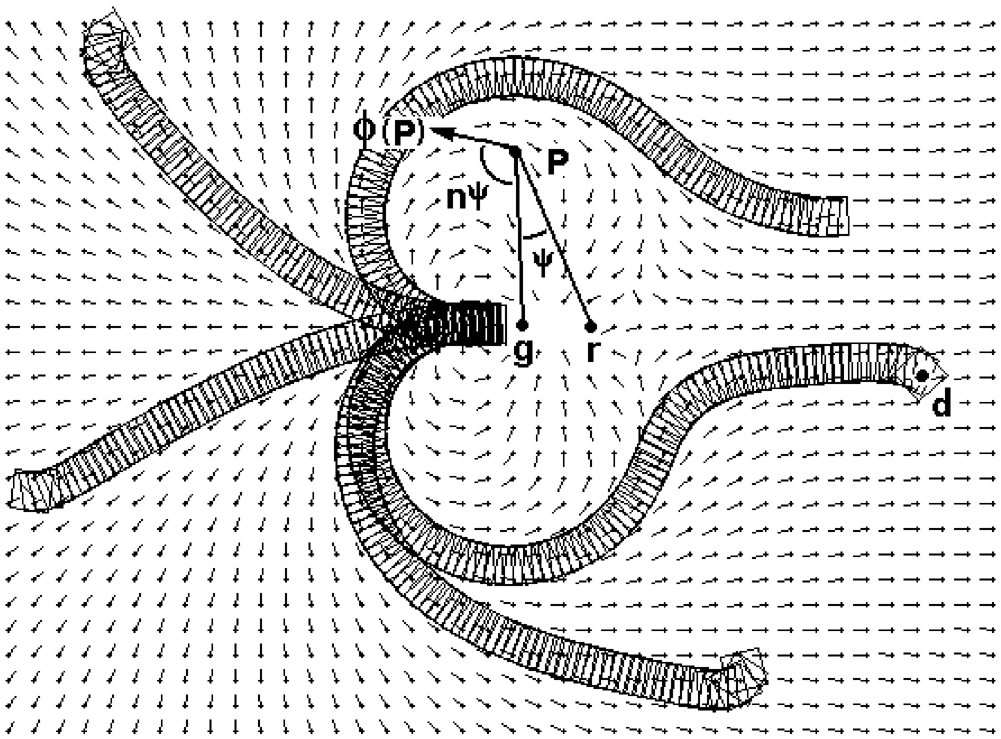
Other Kick Functions

- Univector field method





- Univector field method (Cont'd)



$$\phi(p) = \angle \overrightarrow{pg} - n\psi$$
$$\psi = \angle \overrightarrow{pr} - \angle \overrightarrow{pg}$$

n: positive integer number

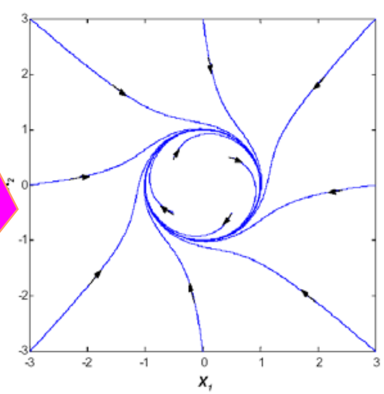
Y.-J. Kim, J.-H. Kim and D.-S. Kwon, ``Evolutionary Programming- based Univector Field Navigation Method for Fast Mobile Robots," *IEEE Trans. Syst. Man. Cybern.*, Part B, Vol. 31, pp. 450-458, June 2001.

- Limit cycle method

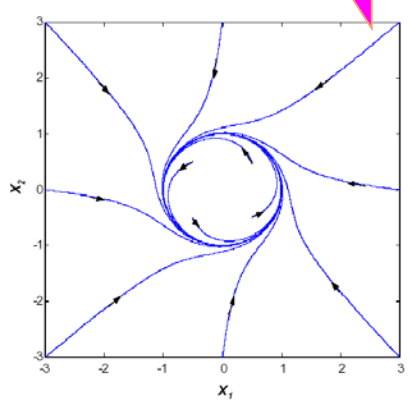
$$\dot{x}_1 = x_2 + x_1(1 - x_1^2 - x_2^2)$$
$$\dot{x}_2 = -x_1 - x_2(1 - x_1^2 - x_2^2)$$

$$\dot{x}_1 = x_2 + x_1(r^2 - x_1^2 - x_2^2)$$
$$\dot{x}_2 = -x_1 + x_2(r^2 - x_1^2 - x_2^2)$$

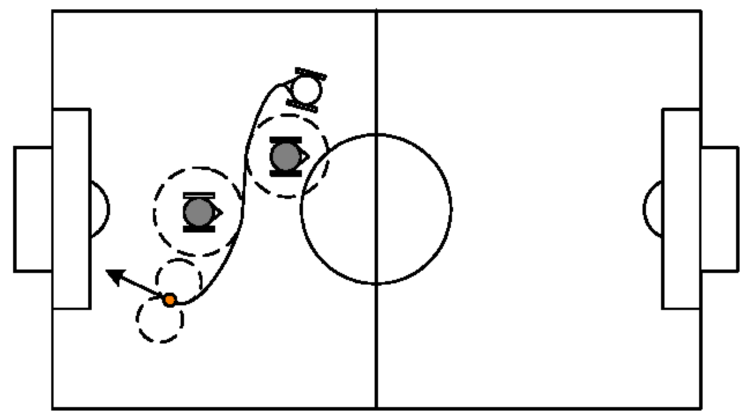
2nd order non-linear equation



Clockwise

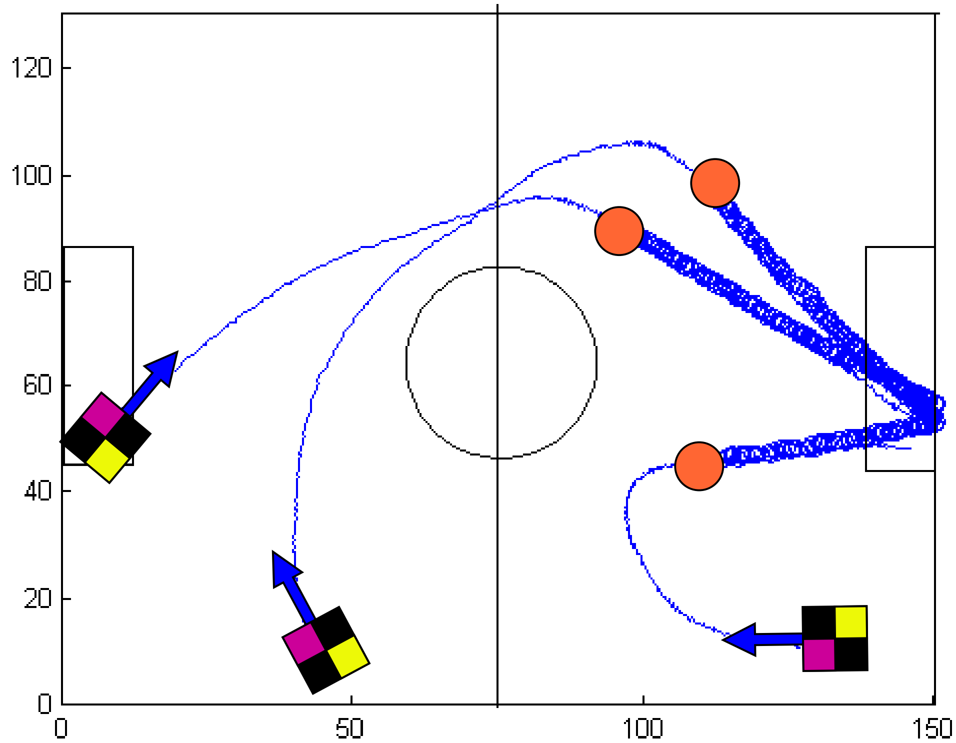


Counter-clockwise



D.-H. Kim and J.-H. Kim, "Limit-cycle Navigation Method for Robot Soccer," *in the Robotics and Autonomous Systems*, Jan. 2003.

- Kick function using a fuzzy controller

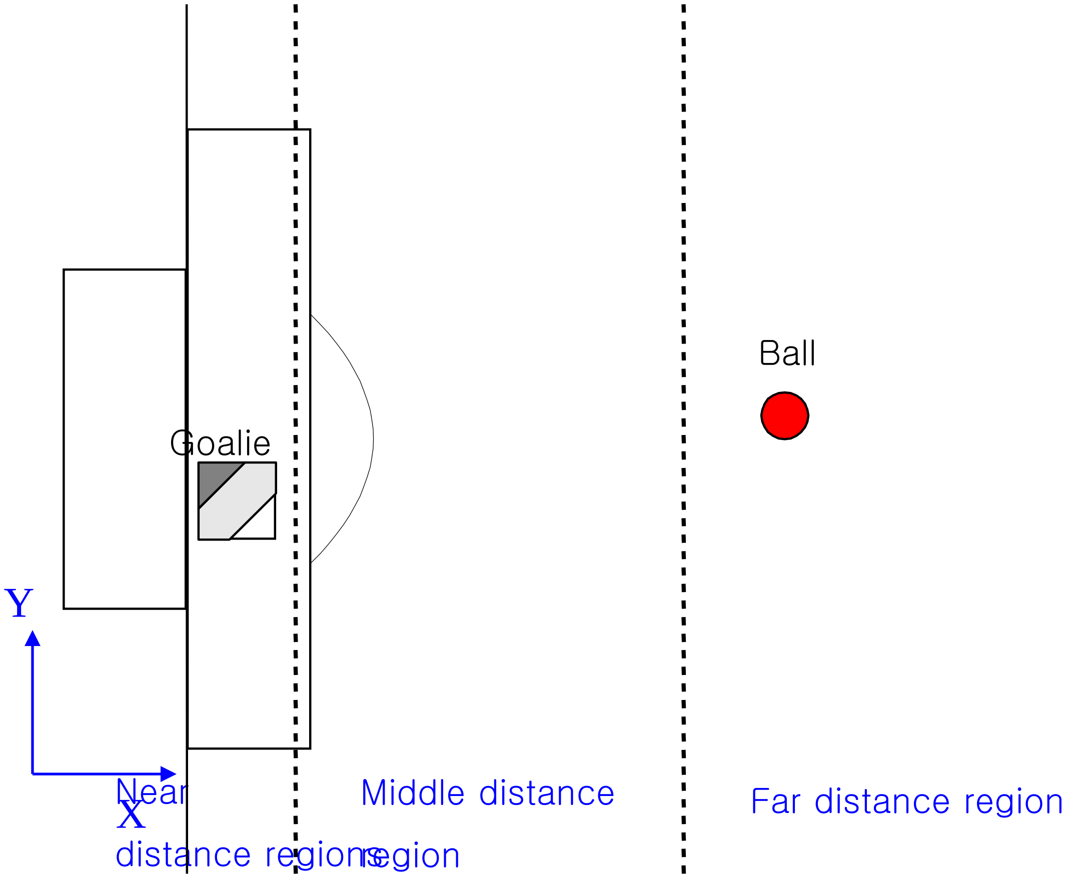


M.-S. Lee, M.-J. Jung and J.-H. Kim, "Evolutionary Programming-based Fuzzy Logic Path Planner and Follower for Mobile Robots," *International Conf. of CEC 2000*, San Diego, USA, 2000, 7 .

Goalie

Regions

- Consider three different lengths of distance between the goalie and the ball



- Goalie motion
  - Move along a line which is parallel to the goal-line
- Three cases based on distance between the goal area and the ball
  - Far distance region
    - Move to Calculated position, (G_OFFSET, y)
  - Middle distance region
    - Move to Y-coordinate of the ball, (G_OFFSET, ball_y)
    - Predict the ball trajectory
  - Near distance region
    - Move to Y-coordinate of the ball, (G_OFFSET, ball_y)

Structure of Goalie Function

- void Goalie(int whichrobot)

Calculate the position that goalie has to move to based on the distance from the goal area to the ball

↓

Assign the boundary constraint value

↓

Move to the calculated position

↓

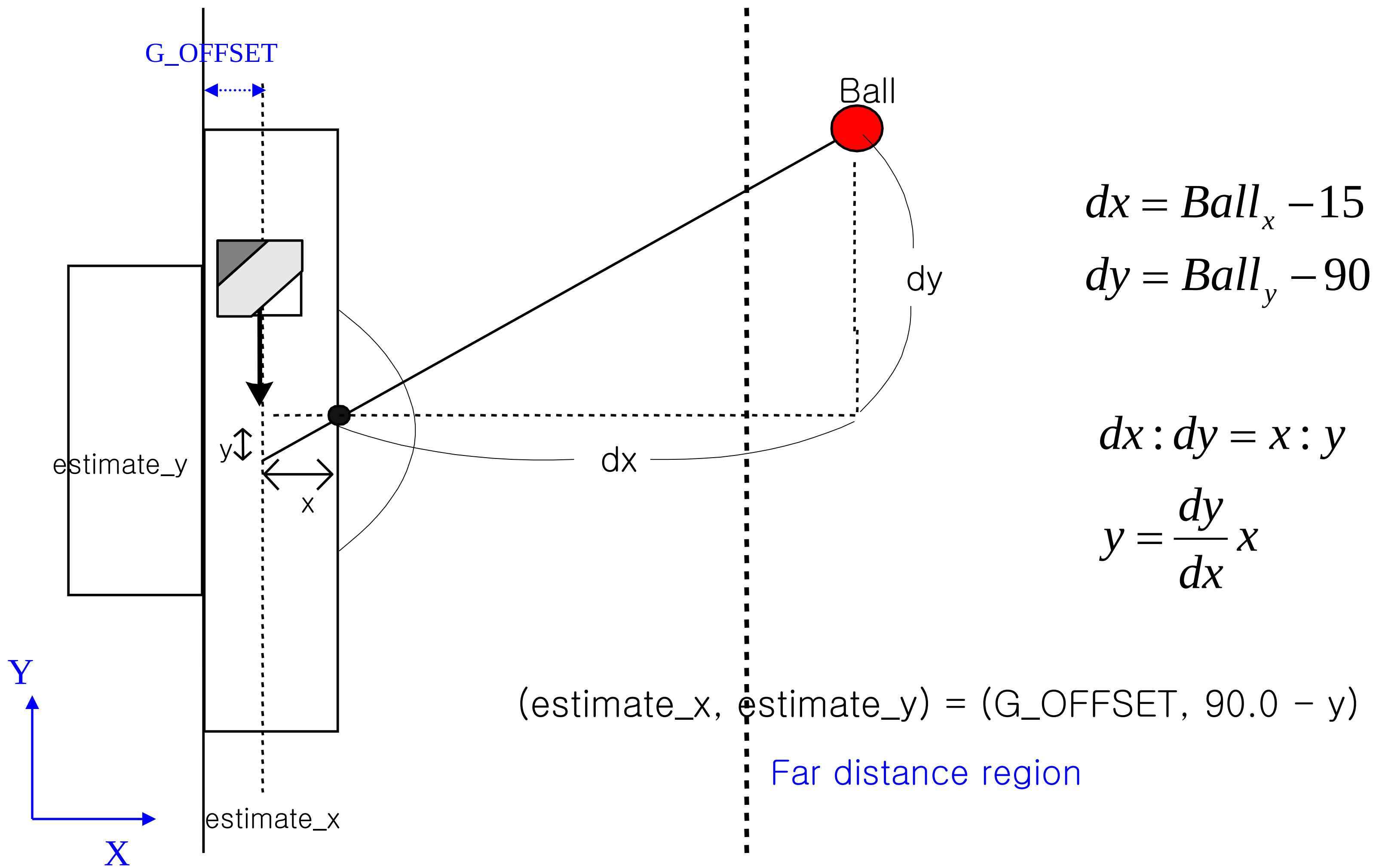
Rotate the goalie angle if distance error is small

↓

Exception routine for goalie robot

Far Distance Case

Ex)



$$dx = Ball_x - 15$$

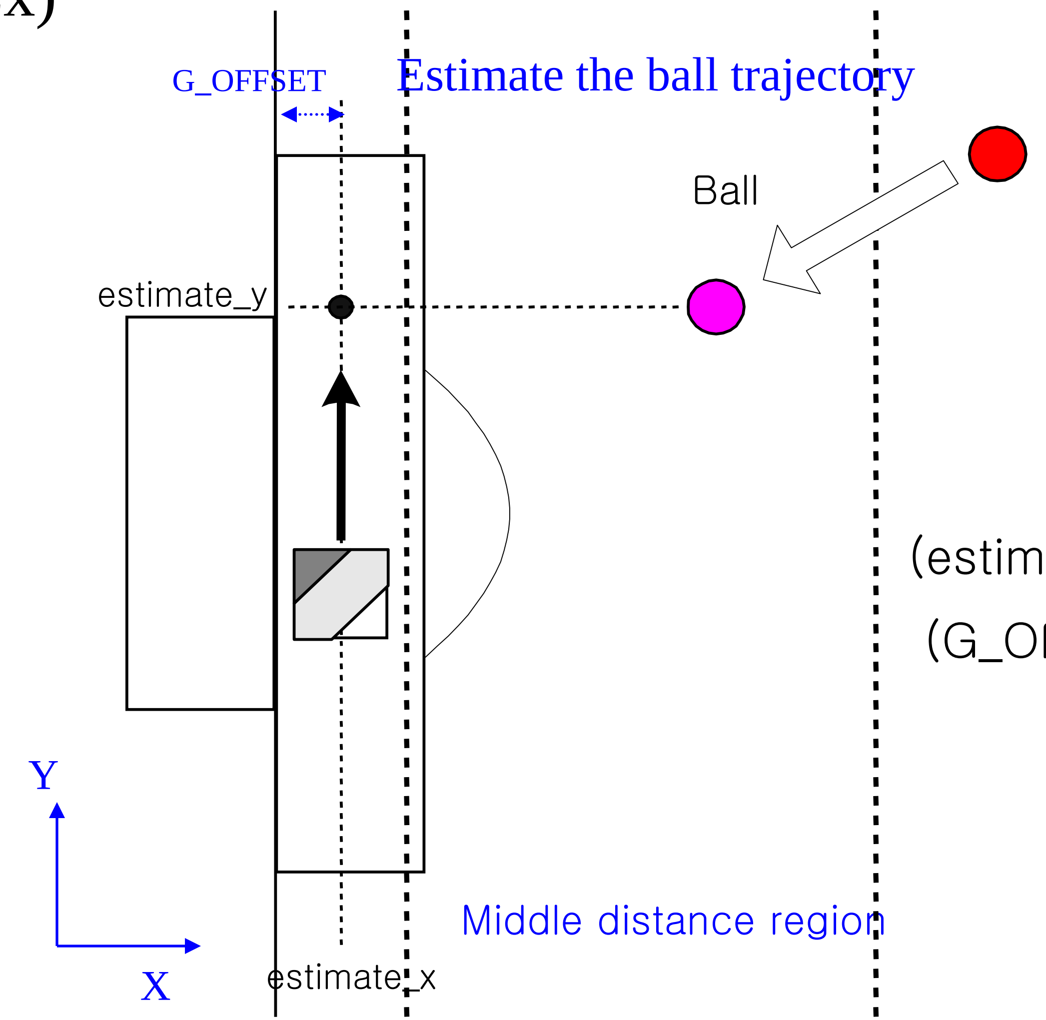
$$dy = Ball_y - 90$$

$$dx : dy = x : y$$

$$y = \frac{dy}{dx} x$$

Middle Distance Case

Ex)

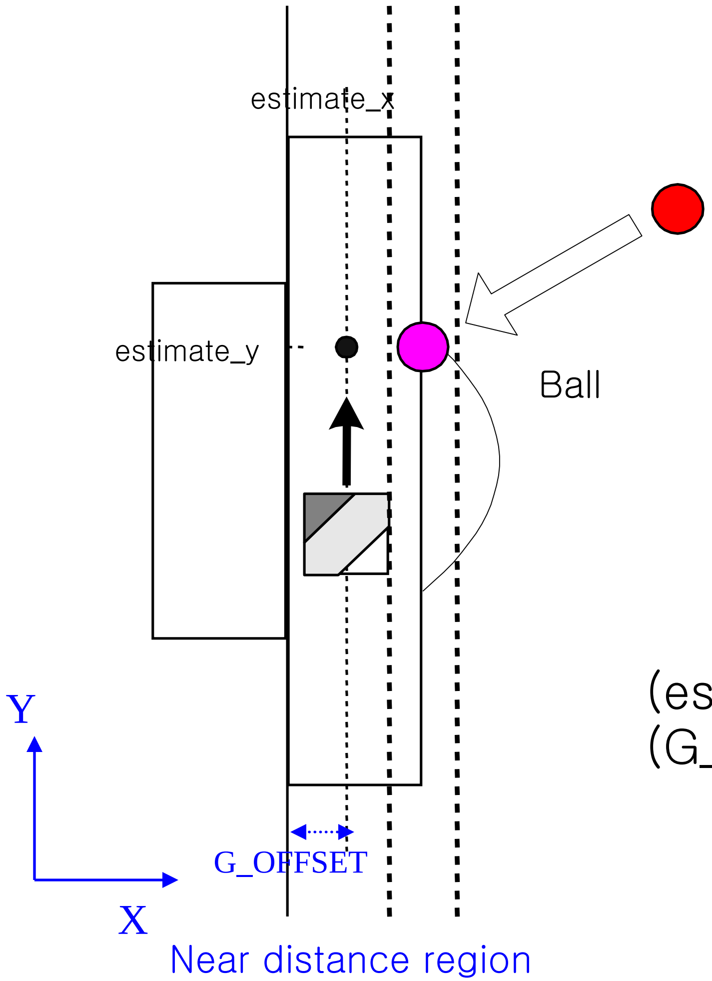


(estimate_x, estimate_y) =

(G_OFFSET, Ball.position.Y)

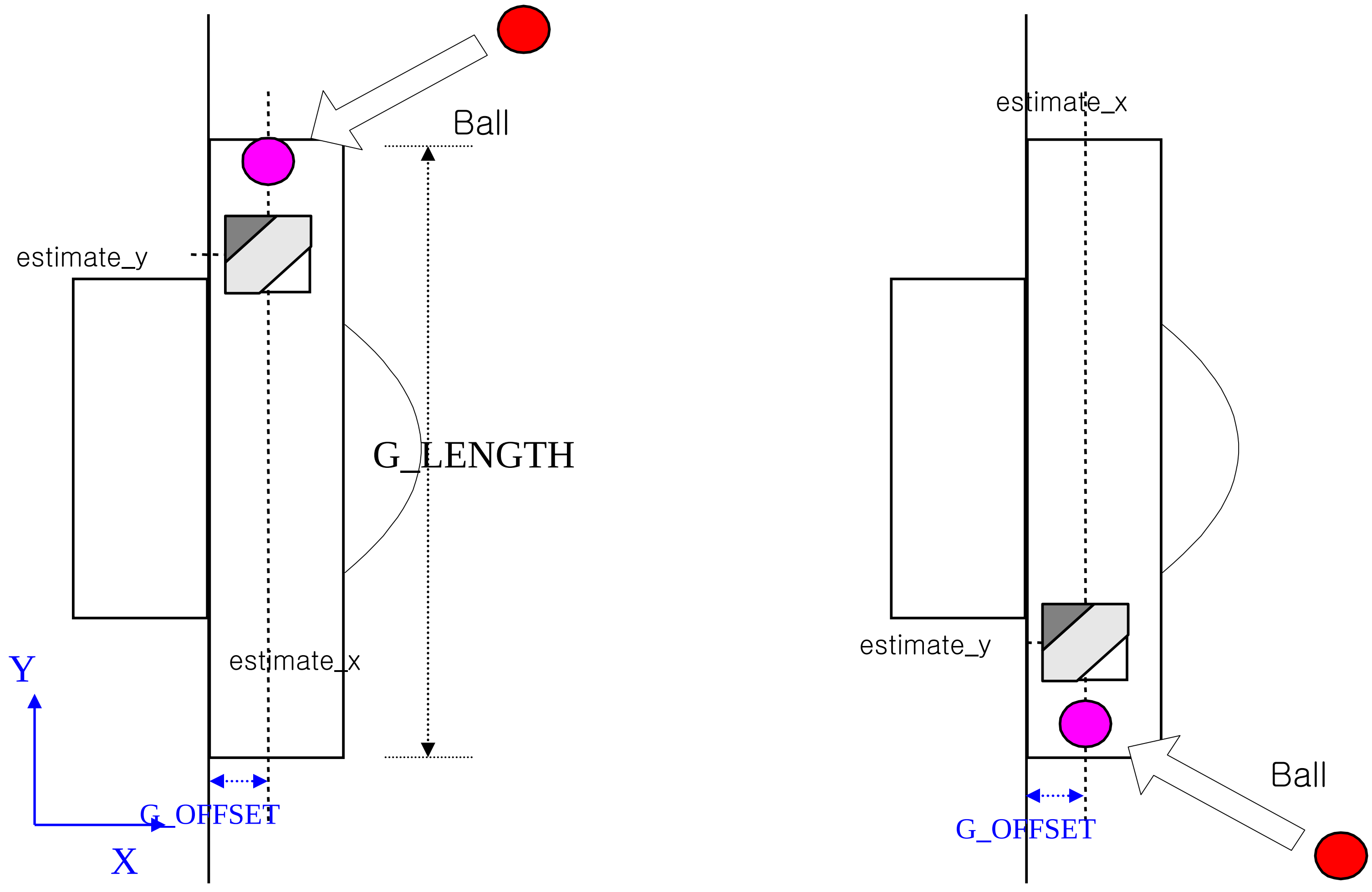
Near Distance Case

Ex)

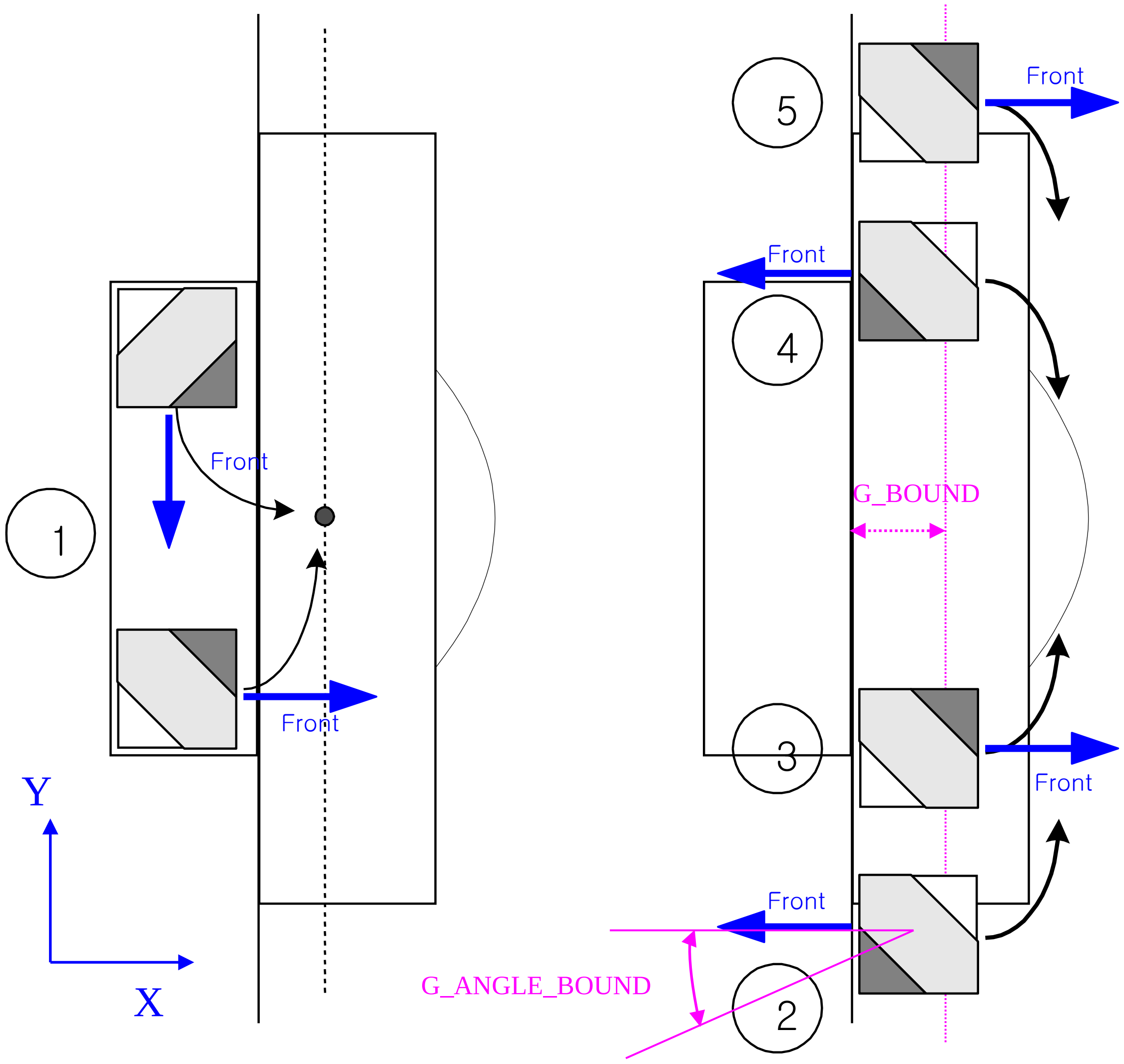


(estimate_x, estimate_y) =
(G_OFFSET, Ball.position.Y)

Boundary Constraint



Exception Routine

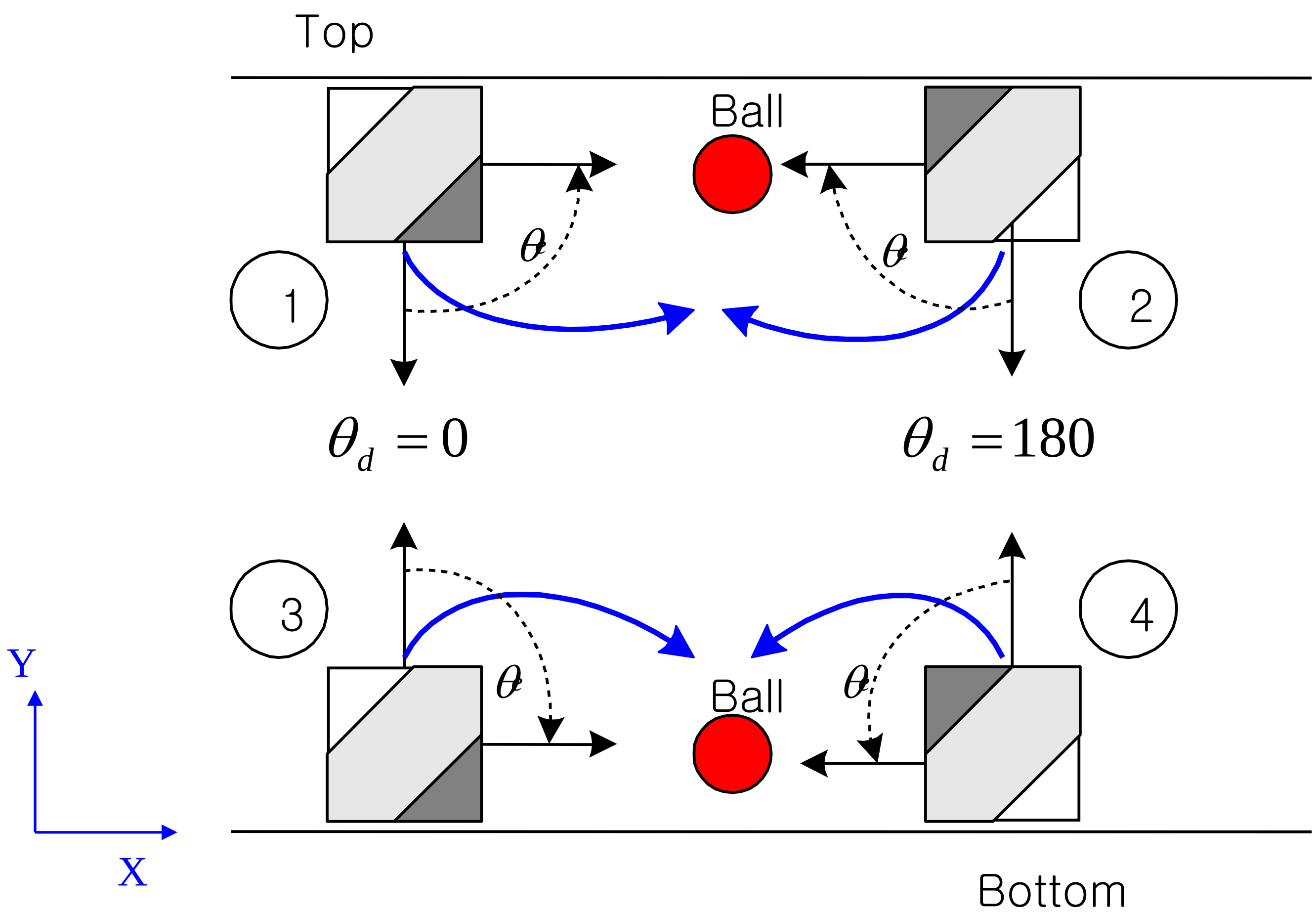


AvoidBound

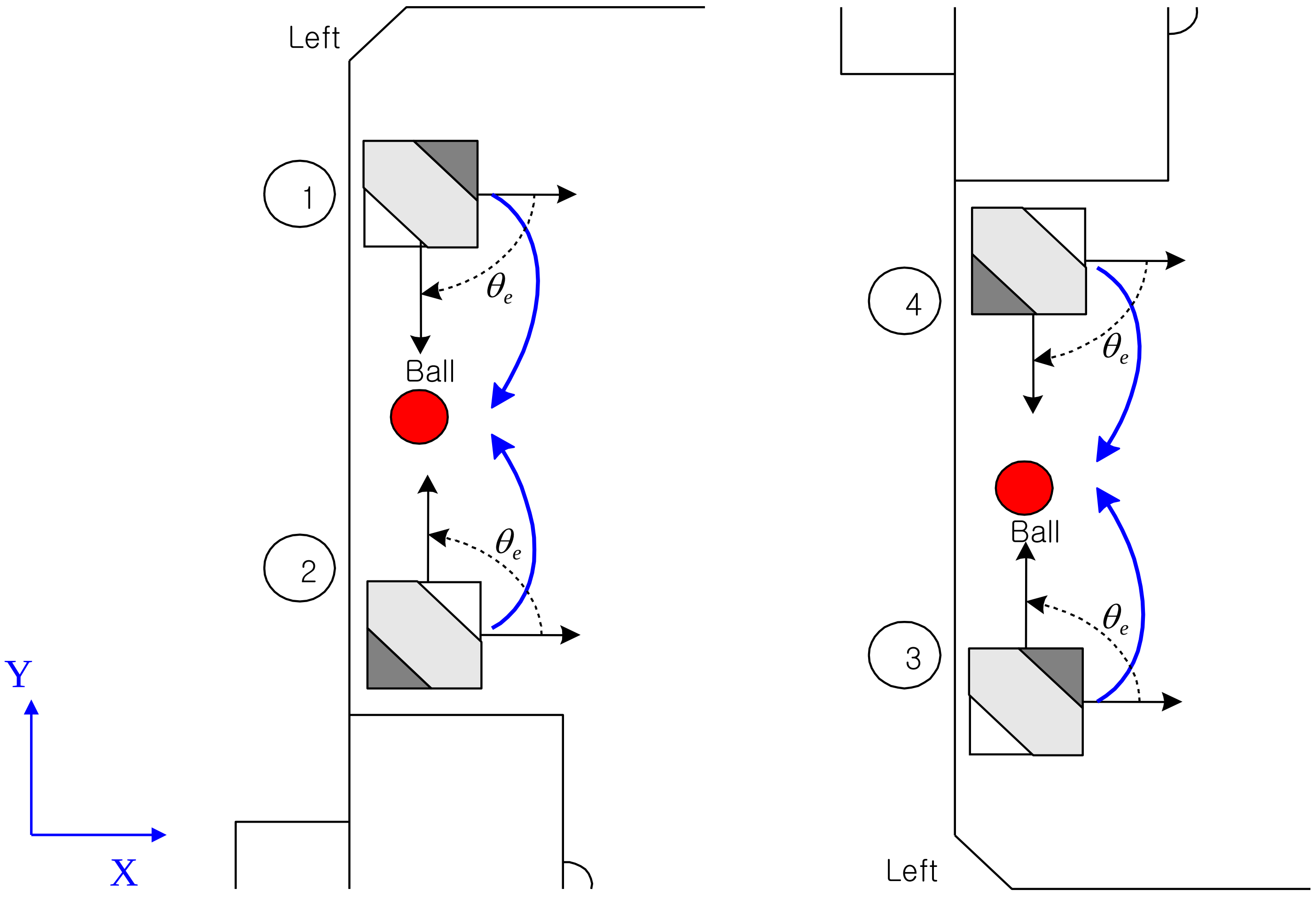
AvoidBound Function

- To consider the situation in which the robot is blocked by the wall of the field
  - Top and bottom walls
  - Left wall
  - Right wall

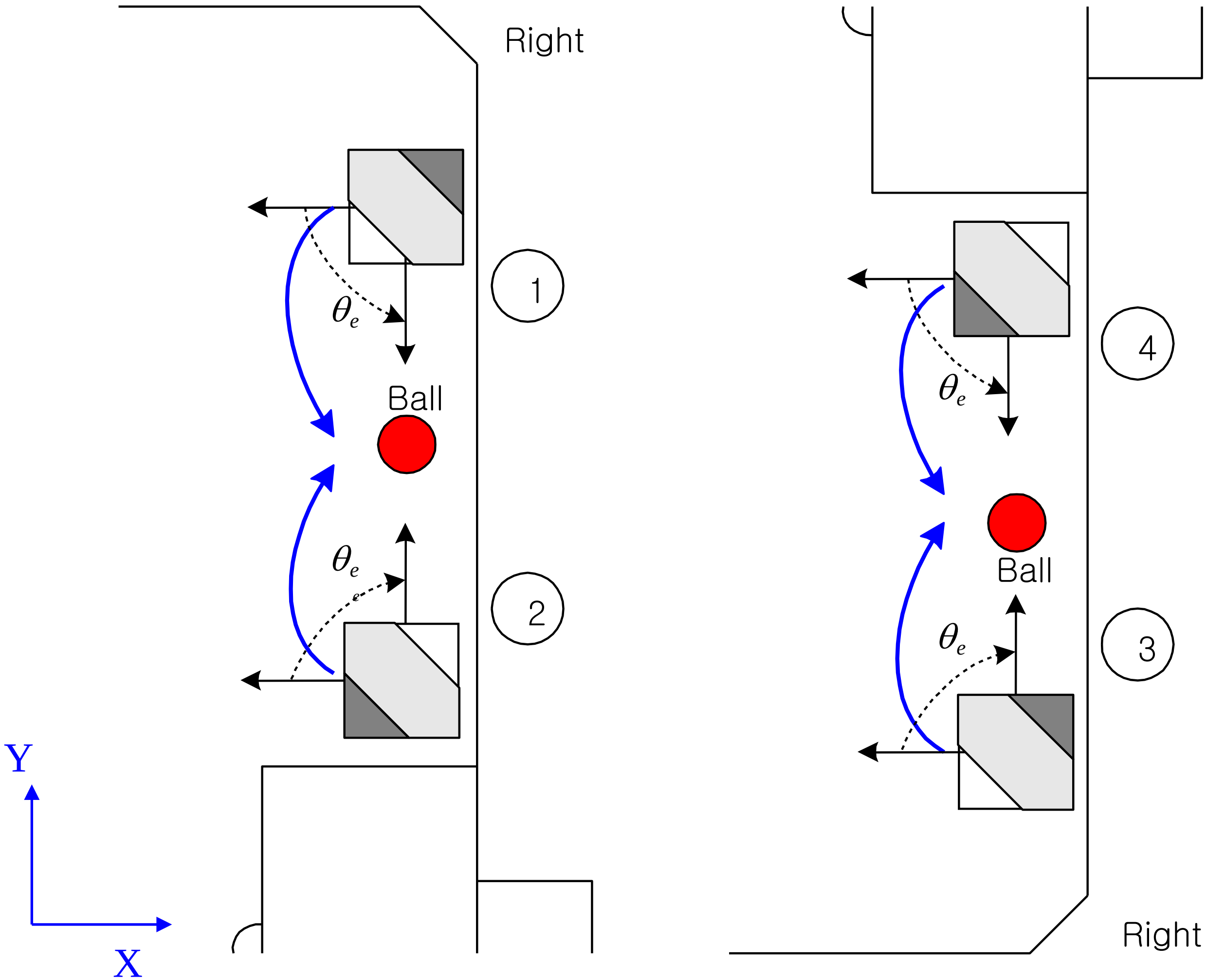
Top and Bottom Walls



Left Wall

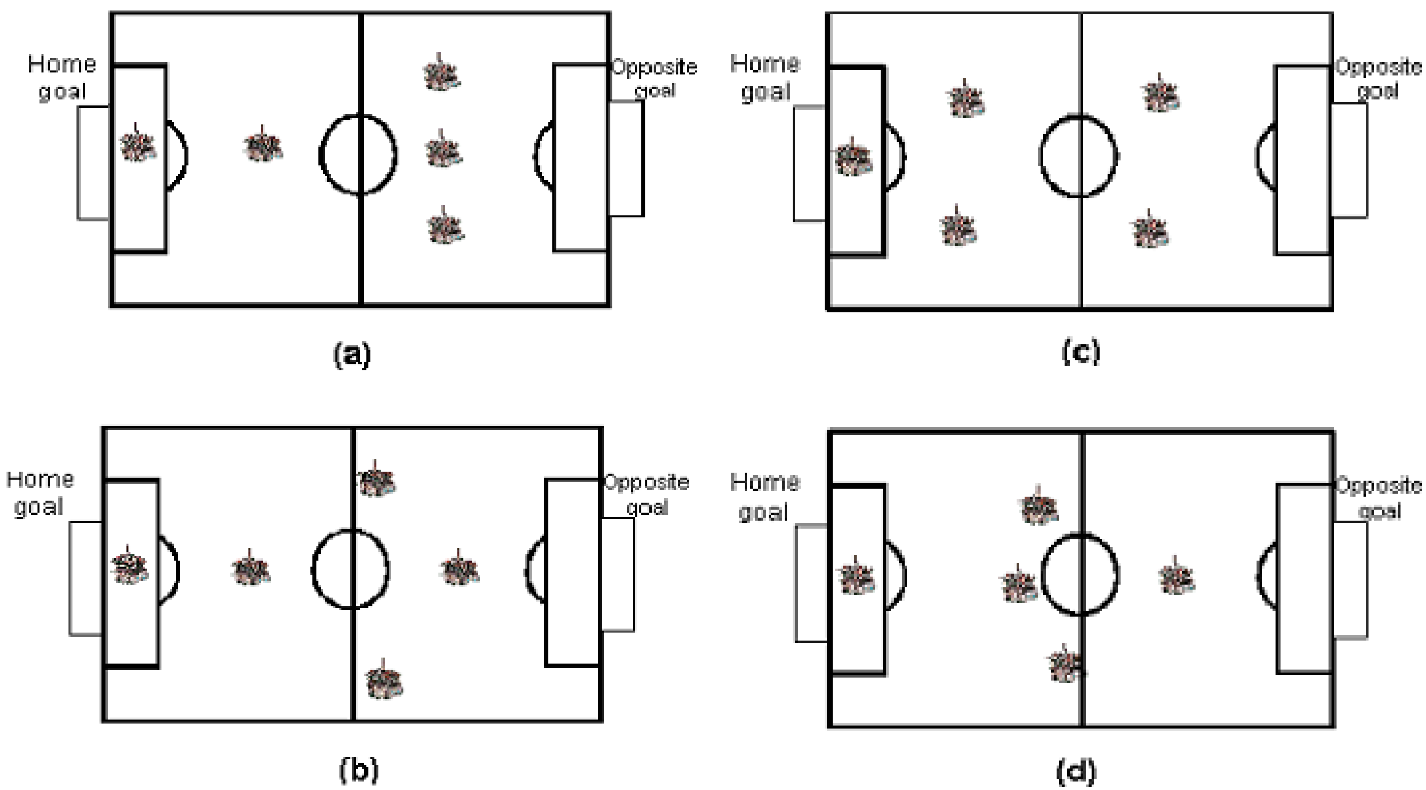


Right Wall



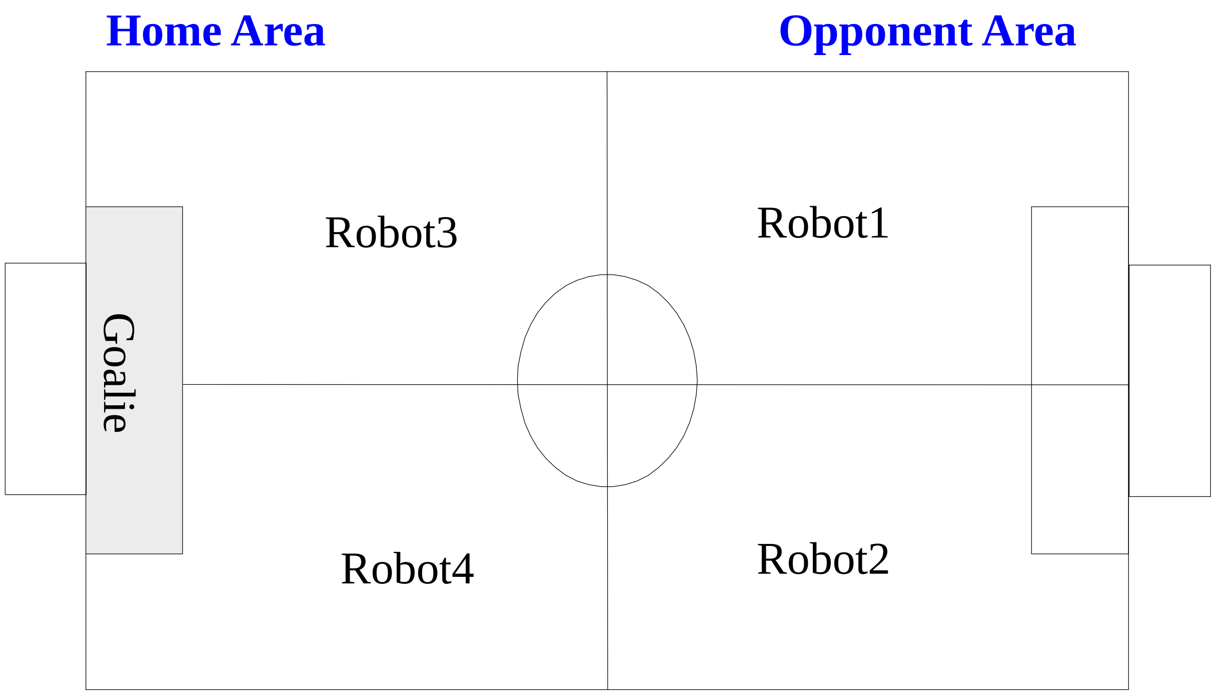
Strategy

Basic Formations



(a) 1-1-3 (b) 1-1-2-1 (c)1-2-2 (d) 1-3-1

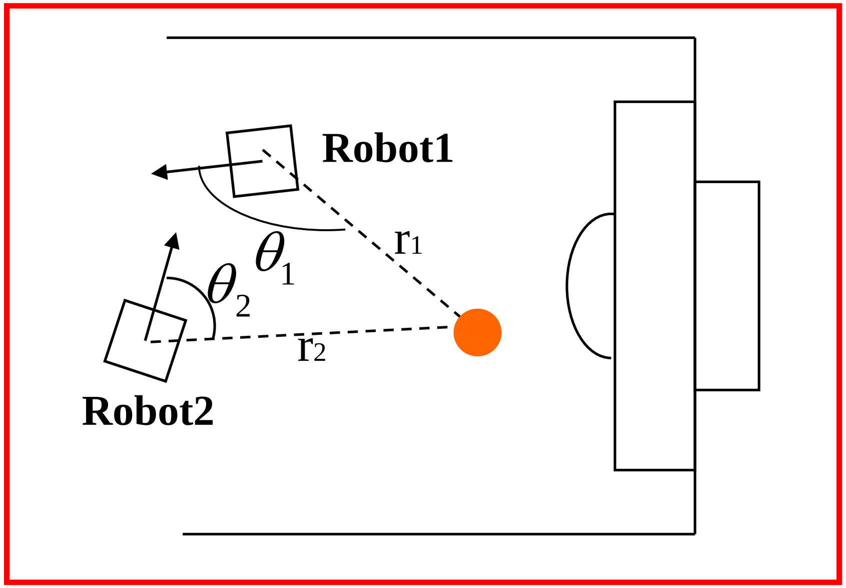
Zone Defense



1 – 2 – 2 formation

Role Selection

- Supervisor
  - To decide when to change the roles of robots.
  - Used conditions: $r_2 < 2r_1,\ -45^o < \theta_2 < 45^o \ \& \ \theta_2 < \theta_1$



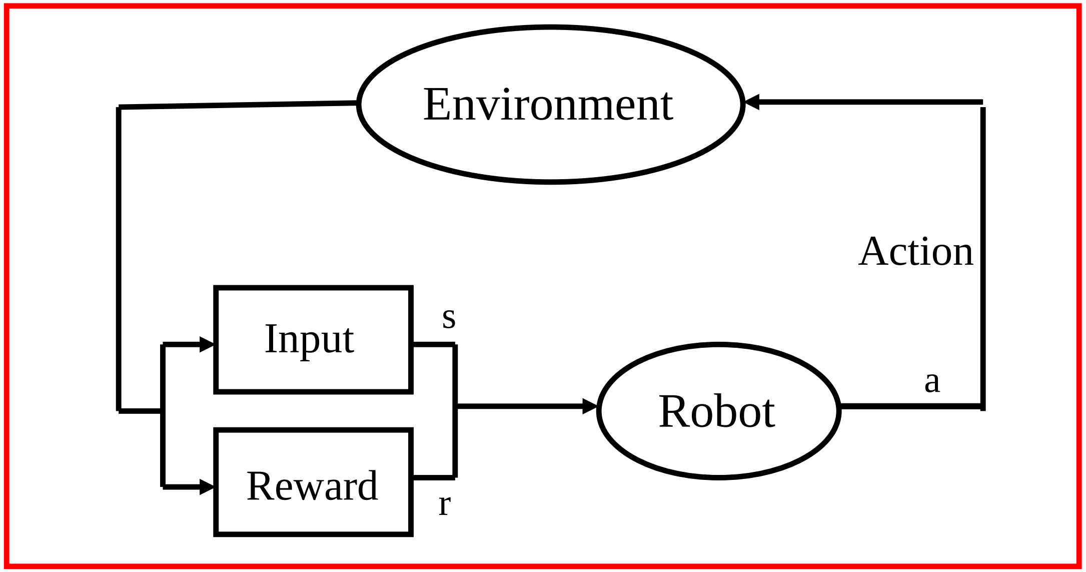
K.-H. Park, Y.-J. Kim, and J.-H. Kim. "Modular Q-learning based multi-agent cooperation for robot soccer." *Robotics and Autonomous Systems* 35.2 (2001): 109-122.

Q-Learning

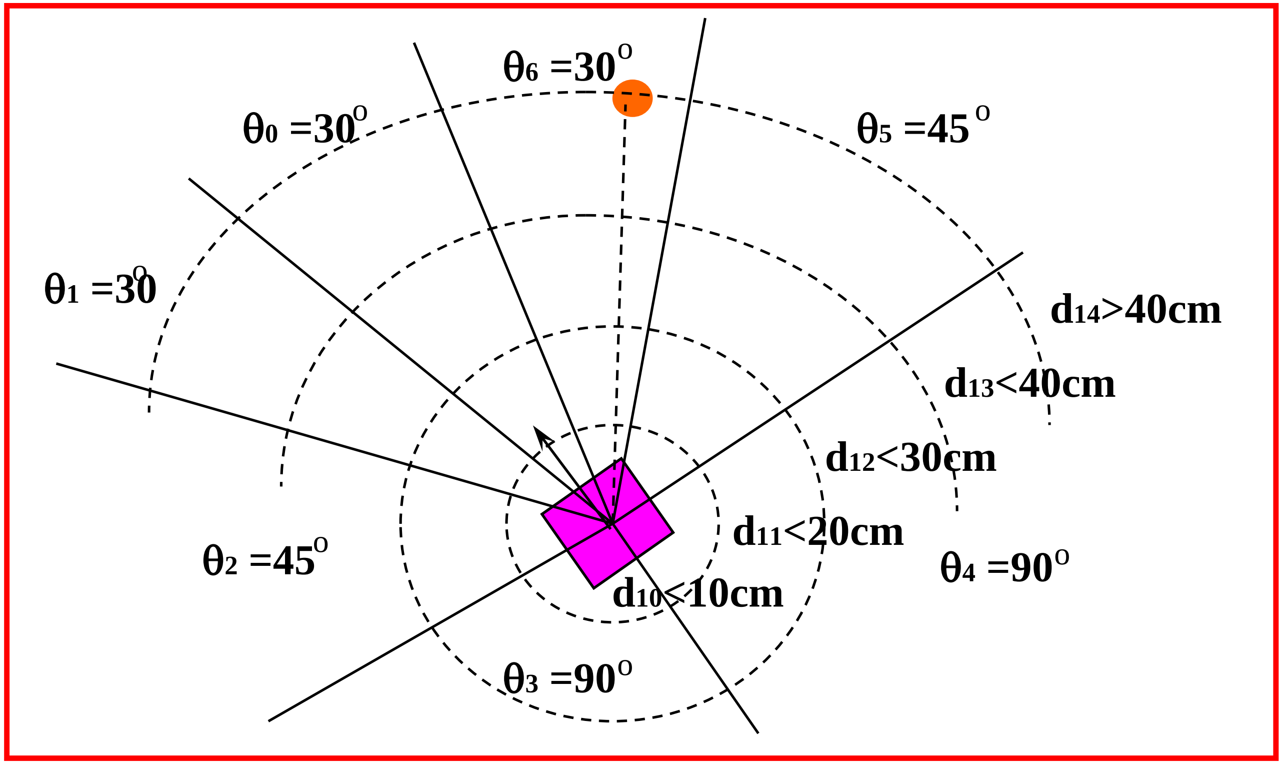
Action-value function for (s,a):

$$Q_n(s,a) = \begin{cases} (1-\alpha_n)Q_{n-1}(s,a) + \alpha_n[r_n + \gamma V_{n-1}(y_n)] & \text{, if } s_n\text{=s \& } a_n\text{=a} \\ Q_{n-1}(s,a) & \text{, otherwise} \end{cases}$$

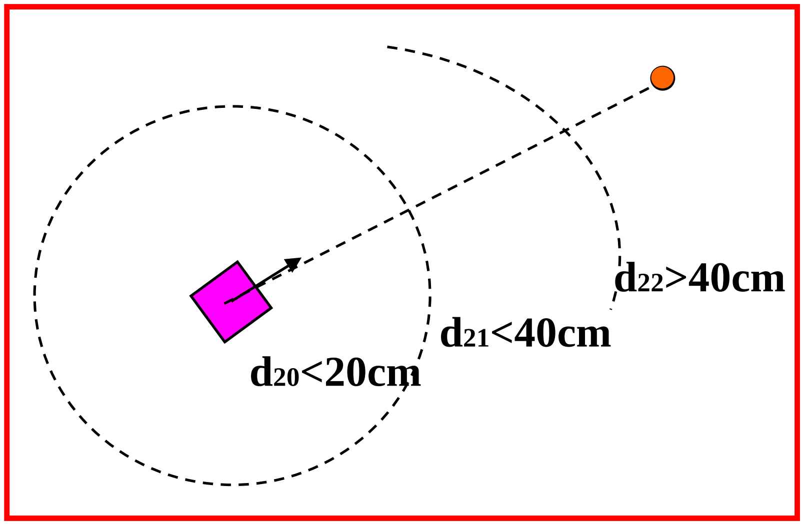
Where $V_{n-1}(y) \equiv \max_b \{Q_{n-1}(y,b)\}$, α_n : the learning ratio.

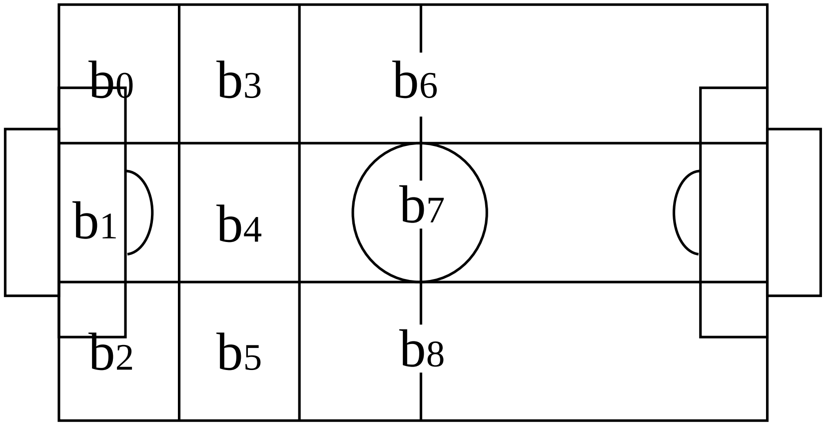


- Robot1: States

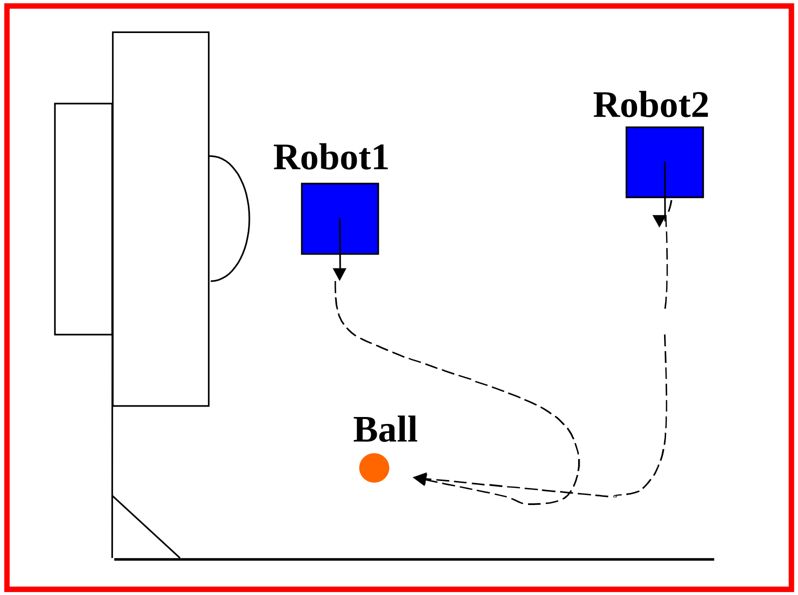


- Robot2 and Ball: States





Total states = 5 (Robot1: distance) x 7 (Robot1: angle) x 3 (Robot2: distance) x 9 (Ball: location) = <u>945 states</u>



- State: $(d_{13}, \theta_1, d_{22}, b_2)$
- Robot1: attacker
- Robot2: defender

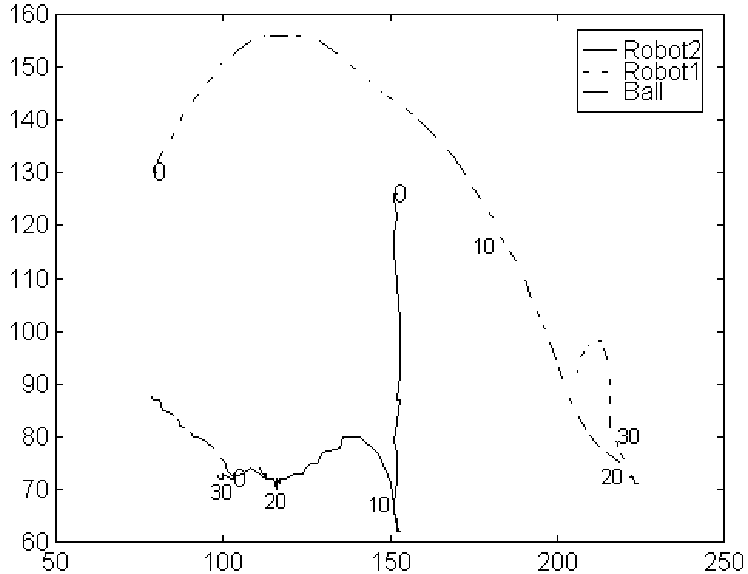
$Q_0(s,a_0) = 0.5$: Initial Q value for action_0 (role change)

$Q_0(s,a_1) = 0.6$: Initial Q value for action_1 (no role change)

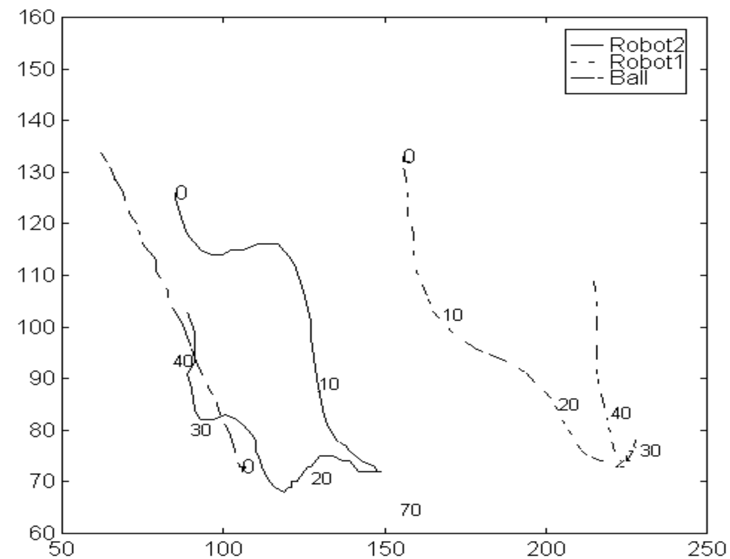
t = sampling time (33ms) x number of samples

(from the current position to the kick position)

Result



$r(s,a_0) = a/(t + b) = 1.137$



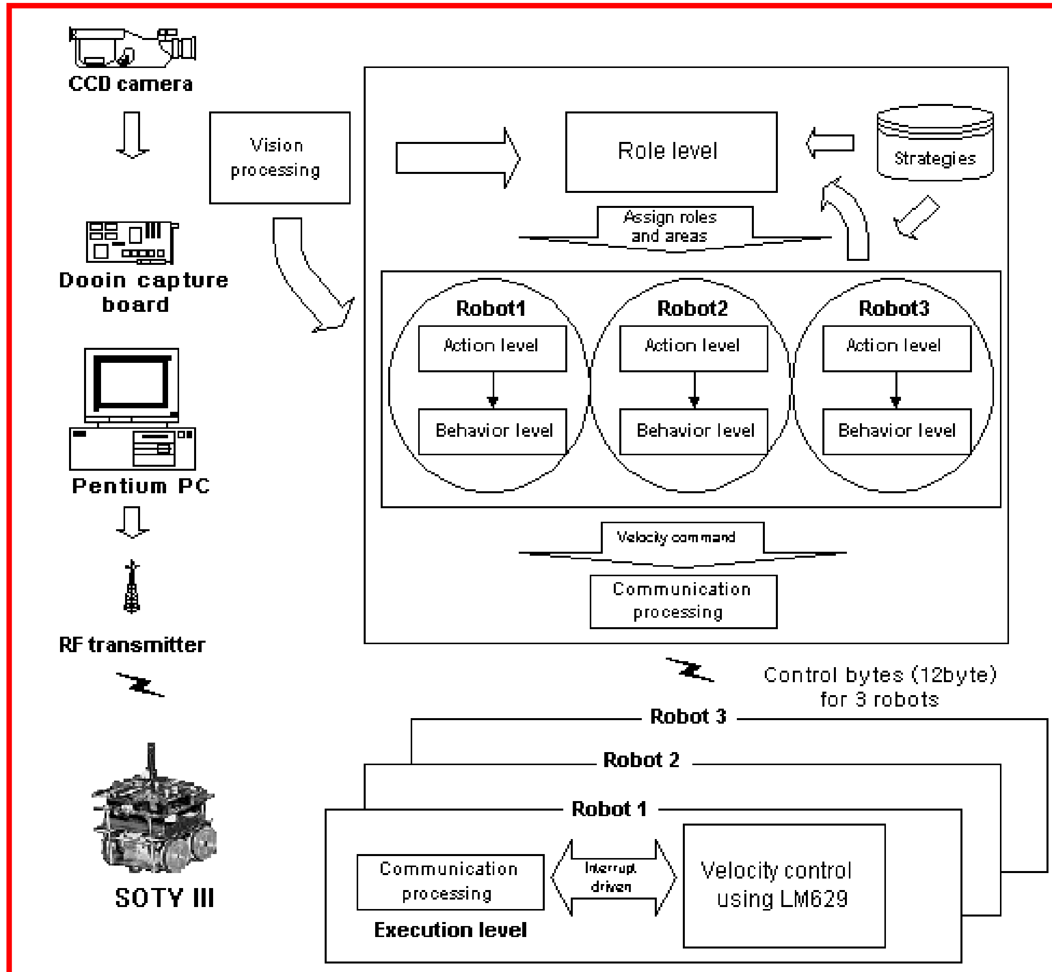
$r(s,a_1) = a/(t + b) = 1.058$

$Q_f(s,a_0) = 1.616$: Q value for action_0 (role change)

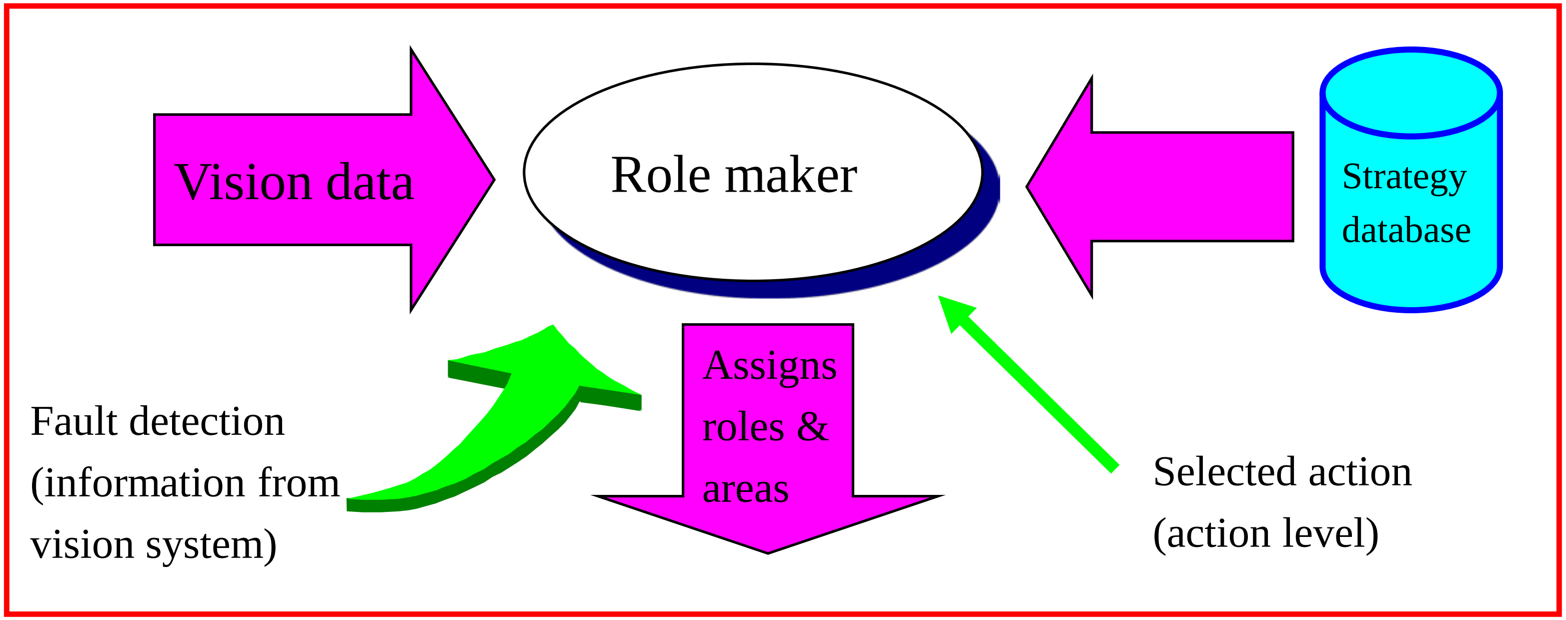
$Q_f(s,a_1) = 1.024$: Q value for action_1 (no role change)

Hybrid Control Architecture

- Combination of hierarchical and behavioral architectures
- Hierarchical architecture
  - Role level
  - Action level
  - Behavior level
  - Execution level
- Behavior-based architecture
  - Behaviors

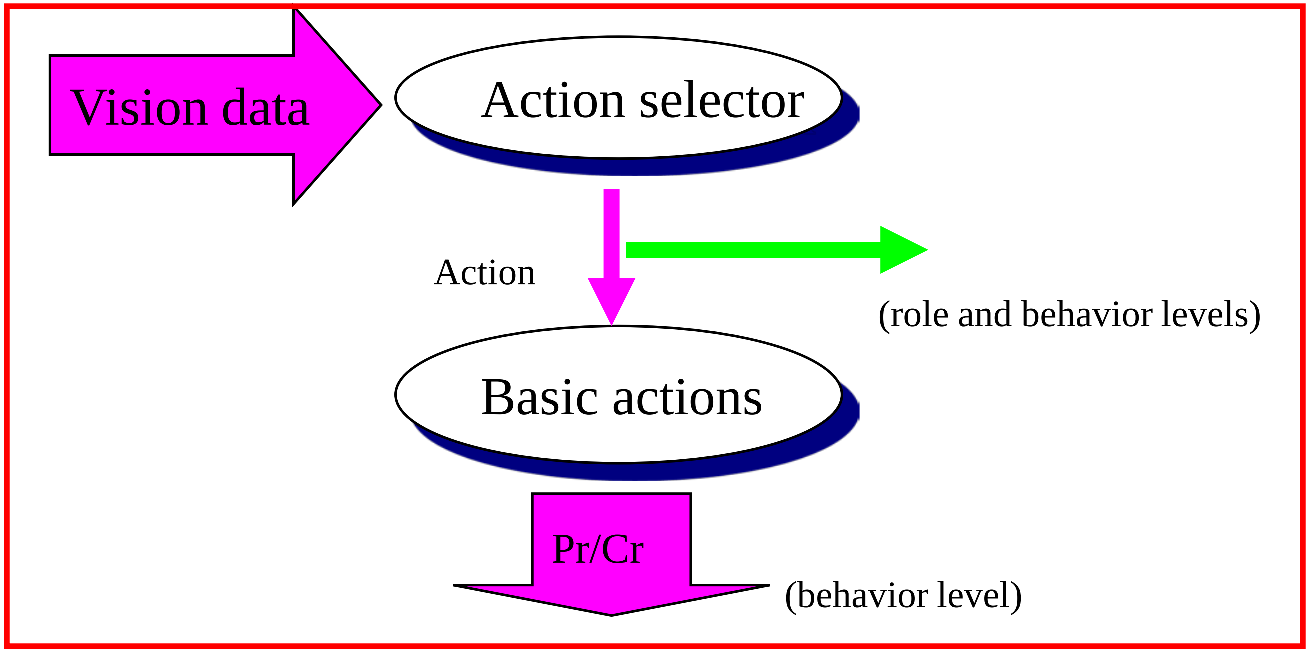


Role Level



- Highest level
- For efficient cooperation among (equivalent) agents
- Easy to model other agents’ actions
    - can dispense with communication among robots

Action Level

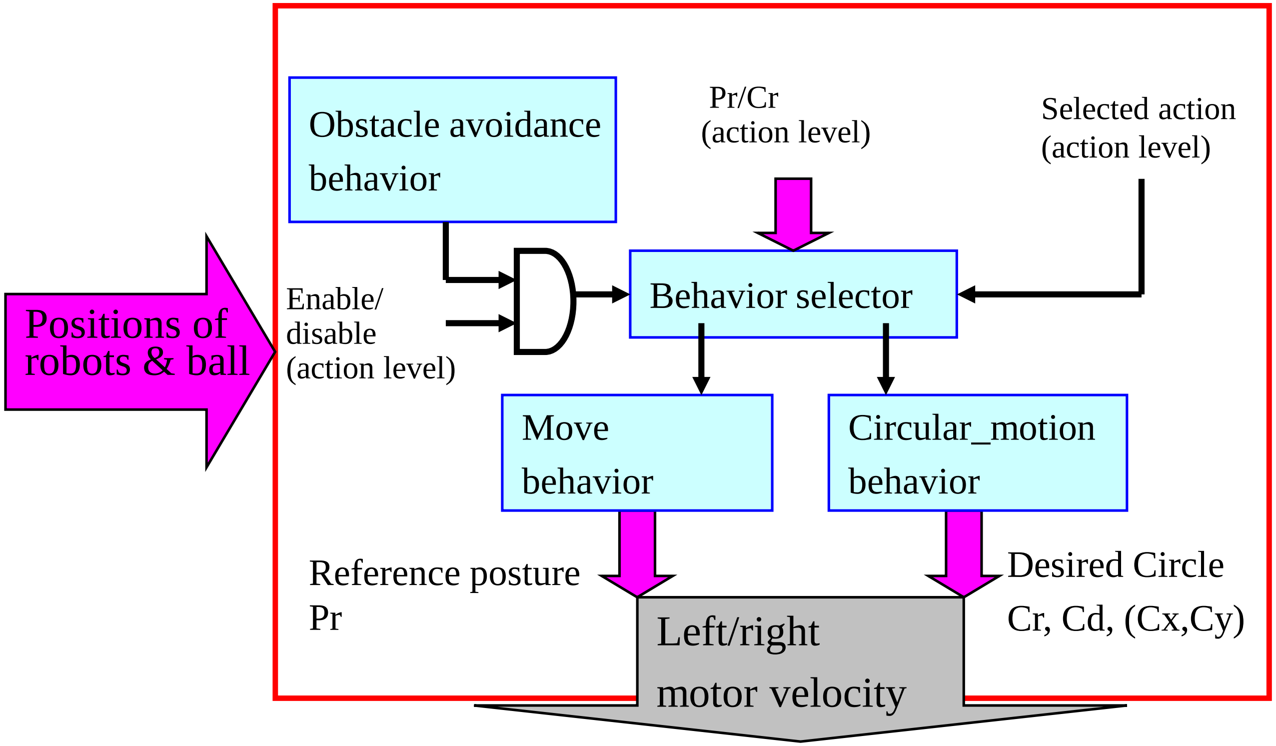


- Ensures reliability
- Degree of achievement: proper action assignment among agents
- In-built strategy (different from that of role level)

Basic Actions (#17)

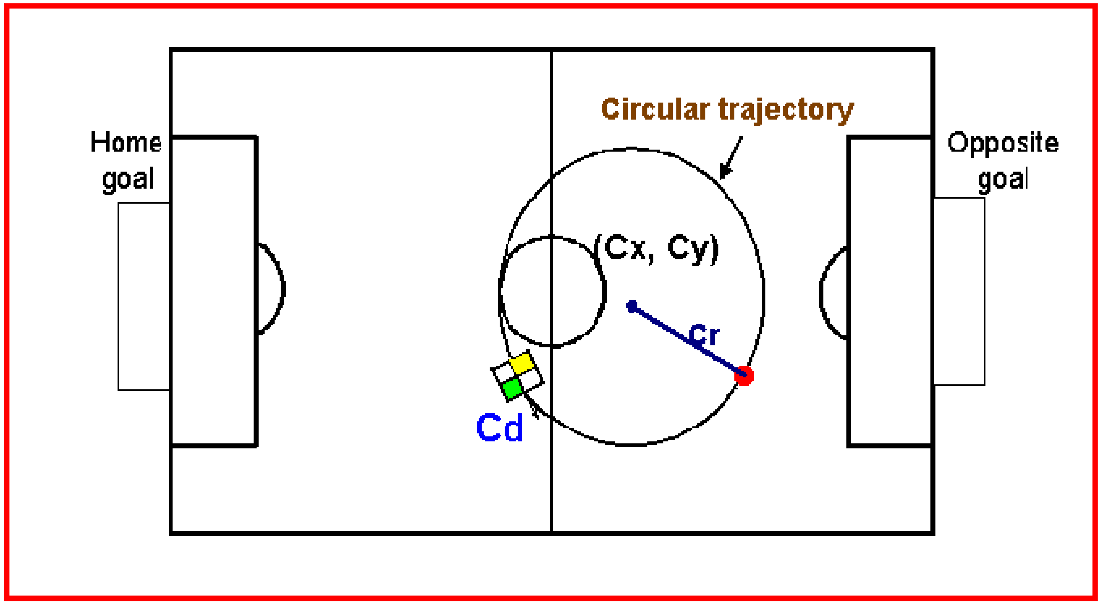
- Primitive actions
  - Stop
  - Wandering
  - Sweep_Ball
  - Ball_Find
- Defending actions
  - Push_Ball
  - Position_To_Push_Ball
  - Screen_Out_Ball
- Attacking actions
  - Shoot
  - Cannon_Shoot
  - Turning_Shoot
  - Spin_Shoot
  - Position_To_Shoot
- Goalkeeper actions
  - Block_Ball
  - K_Defense_Goal
  - K_Default_Position
  - K_Attack
  - K_Need_Escape_Goal

Behavior Level



- Obstacle avoidance (reactive), circular_motion and move behaviors
- Uses reactive behavior to adapt to a varying environment.

Circular_motion Behavior



- Useful for Spin_Shoot and Turning_Shoot actions.
- Generates circular trajectory
  - centered at (C_x, C_y) with a radius of C_r and a direction in C_d.

Execution Level

- Lowest level
- Velocity control of robot actuators (2 DC motors)
- Motion control