Autonomous Robot: Obstacle following

Guillaume Lemaître - Sophia Bano
Heriot-Watt University, Universitat de Girona, Université de Bourgogne
g.lemaitre58@gmail.com - sophiabano@hotmail.com

I. Objective

The main objective of this laboratory exercise was to program an obstacle following behaviour on an e-puck Robot using Webots Environment. There are two main stages for developing Obstacle following behaviour.

- Sensors Calibration
- Obstacle Following

II. Calibration sensors

E-puck has eights infra-red sensors as shown in figure 1, which has been used in this exercise to develop obstacle following behaviour. These sensors must be calibrated before developing obstacle following behaviour.

In order to do calibration, the robot head is placed perpendicular to an obstacle such that IR0 and IR7 faces obstacle and are in contact with obstacle as shown in figure 2. Then robot is moved away from obstacle at a linear speed equal to -10 until it covers a distance of 6 cm.

The code developed to do this is shown below, where *displacement* comes from odometry which tells the total distance cover by robot. When the value of displacement becomes equal to 6 (cm), robot motion stops.

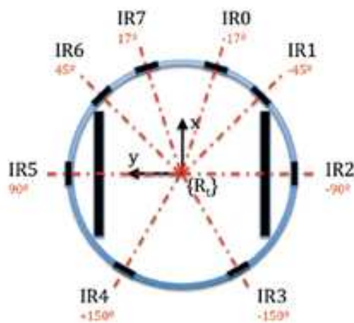


Figure 1. IR sensors location of e-puck

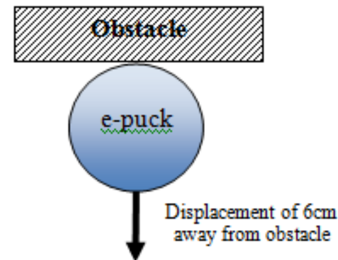


Figure 2. Robot configuration for calibrating sensors

```
if(displacement>-6)
{
    linear_speed=-10;
    angular_speed=0;
    left_speed=linear_speed-angular_speed;
    right_speed=linear_speed+angular_speed
        ;
}
else
{
    linear_speed=0;
    angular_speed=0;
    left_speed=linear_speed-angular_speed;
    right_speed=linear_speed+angular_speed
        ;
}
```

The logFile.mat of this algorithm is opened in Matlab, and information about column 3 (displacement along x-axis) and column 6 (value of IR0) is extracted. The x-displacement versus IR0 value are plotted and a polynomial of 8th order is fitted over this using Matlab Basic Fitting tools. Result is shown in figure 3.

Thus the coefficients of 8th order polynomial come out to be the following:

```
p0=1.8658e-27
p1=-3.0795e-23
p2=2.1084e-19
p3=-7.7546e-16
p4=1.6596e-12
p5=-2.0957e-9
p6=1.5174e-6
```

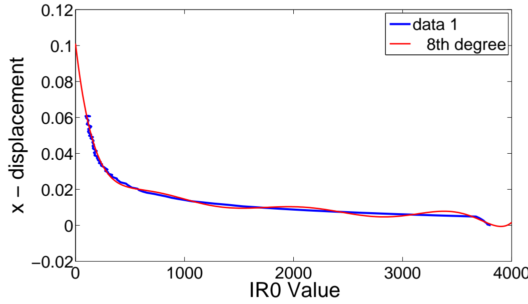


Figure 3. Fitting 8th order polynomial over sensor data

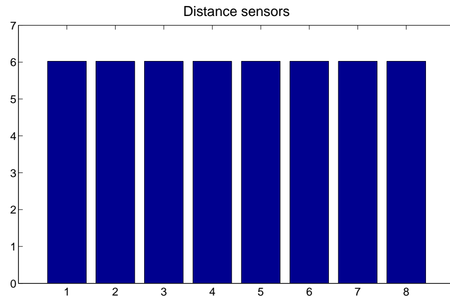


Figure 4. Calibrated distance sensors with no obstacles close to them

```
    p7=−0.00058968
    p8=0.1154
```

As a result of calibration, now if there is no obstacle near robot, sensor will give a maximum value which is 6 cm. And whenever a close object is detected, respective sensor will give the correct distance of object from sensor. Value of distance sensors with no close obstacle detected is shown in figure 4. In case of no close object, all sensors will give value equal to 6cm.

```
for (i = 0; i < 8; i++)
{
 value = wb_distance_sensor_get_value(
     distance_sensor[i]);
  if (value >4000)
  {
  value=4000;
  }
  else if(value<130)
  {
  value=130;
  }
sensors_value[i] = coef[8]*pow(value,8)+
    coef[7]*pow(value,7)+coef[6]*pow(value
    ,6)+coef[5]*pow(value,5)+coef[4]*pow(
    value,4)+coef[3]*pow(value,3)+coef[2]*
    pow(value,2)+coef[1]*pow(value,1)+coef
    [0];
```

```
sensors_value[i]=sensors_value[i]*100;
```

The coefficients of 8th order polynomial are used in algorithm to get distance of obstacle from robot. The code above shows the code for using polynomial. Note that depending on the graph shown in figure 3, upper and lower bound of sensor is set. If sensor value comes out to be greater than 4000(means obstacle is close to robot), value is limited to 4000 and if sensor value comes out to be less than 130(means no close obstacle), value is limited to 130. This helps in proper fitting of polynomial.

III. Obstacle following

Three states have been implemented in order for the robot to follow obstacle properly. These states are explained in detail below:

A. State 0

This state implements the condition that when there is no close obstacle near robot, it will keep on moving straight. This is done by checking IR0, IR1, IR6 and IR7. All these sensors are on the front side of robot. If any of these sensor detects close obstacle near them, the robot will jump to state 1 else it will keep on moving straight with a linear speed of 100. The threshold value is set to 1.5 which defines close obstacle distance from robot. The code for developing this condition is shown below.

```
//////////////////////////////////////
// Initialisation state
// Go in ahead until to
// find an object
// When an object is detected
// go to the state 1
//////////////////////////////////////
if(state==0)
{
  // If an object is detected
  if((sensors_value[0]<thres)||(
      sensors_value[1]<thres) ||(
      sensors_value[6]<thres) ||(
      sensors_value[7] < thres))
  {
    //// Go to the next state
    printf("Detected object -> state 1 \n"
        );
    state=1;
  }
  // Otherwise
  else
  {
    // Go ahead
```

```
    printf("Going ahead \n");
    state=0;
    linear_speed=100.0;
    angular_speed=0.0;
  }
}
```

B. State 1

The second state allows to the robot to satisfy these two following conditions:

- The robot has to be perpendicular to the consider obstacle
- No obstacle have to be in the front of the robot.

The code below shows the implementation of the state 1.

```
////////////////////////////////////
// State to put the robot
// perpendicular to an obstacle
// and in the same time that no
// obstacle are on the
// front of the robot
///////////////////////////////////
if(state == 1)
{
  // If the robot is perpendicular to one
       obstacle
  // And that the front of the robot is
       free
  if((sensors_value[2] <(thres+0.3)) && ((
      sensors_value[0]>thres)||(
      sensors_value[7]>thres)))
  {
    // Go to the state 2
    printf("\n Perpendicular to the object
         and free field at the front -> 
         state 2 \n");
    state=2;
  }
  // Otherwise
  else
  {
    // Turn left
    printf("\n Turning left \n");
    state=1;
    linear_speed=0.0;
    angular_speed=100.0;
  }
}
```

1) Perpendicular criterion: After detected an obstacle, the robot have to rotate until to be perpendicular to follow the obstacle.

Figure 5(a) shows the possible arrival position of the robot near of the obstacle. The robot will turn to the left (anti-clockwise) until to be perpendicular to the obstacle. The position of the infra-red IR2 sensor is at -90 degrees as shown on the figure 5(a). The robot will be considered perpendicular when the distance return by the sensor IR2 will be weak. The final position is shown on the figure 5(b).

2) Free front field criterion: For some complex case as shown on the figure 6, the perpendicular criterion is not sufficient.

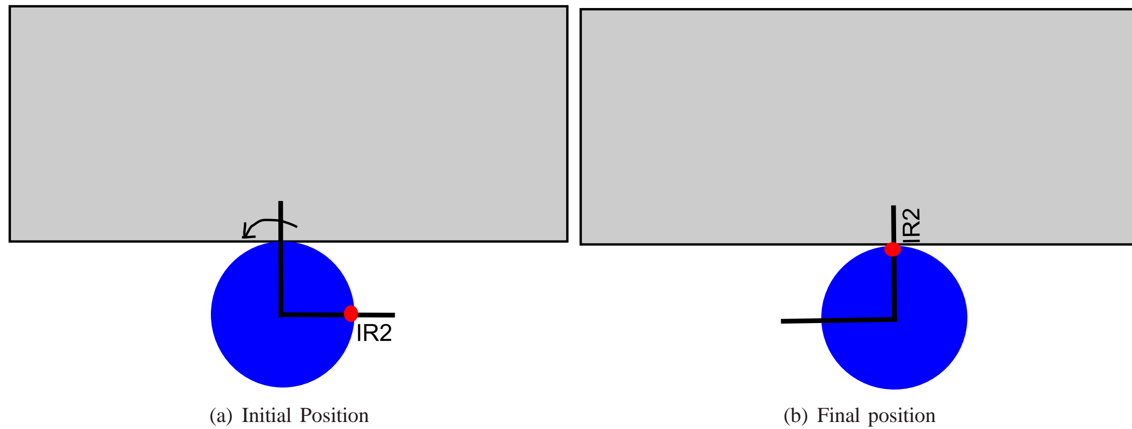
In the previous section, we explained how the robot can set up to be perpendicular to the obstacle. However, as shown on the figure 6(b), the robot can meet another obstacle in the same time than to be perpendicular. In order to fix this problem, the sensors IR0 and IR1 will give the information concerning the field at the front of the robot. Moreover to be perpendicular, the robot will turn until the moment than the sensor IR0 indicate than the field at the front of the robot is free and the sensor IR1 will be enough far of the edge as shown on the figure 6(c). When the robot is perpendicular and have free field at the front, the state 2 can be activated.

C. State 2

The state 2 allows to the robot to follow the object. In order to check if the robot is as in the condition shown on the figure 7(a), the difference of distances between the sensors IR1 and IR2 is computed.

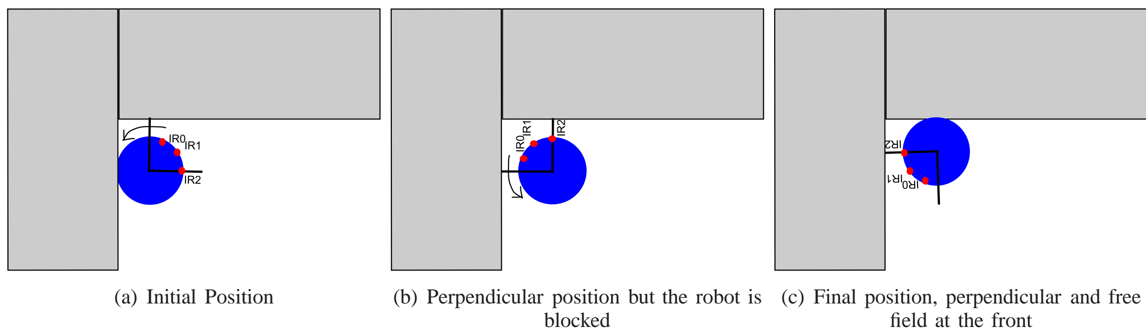
1) Difference of distances small: The case on the figure 7(a) shows the moment where the robot will follow the edge and the difference of distances of sensors will be small. The code which implements this part is presenting below:

```
////////////////////////
// If the robot follow
// the edge but is to near
// of
// the edge -> turn left
////////////////////////
else if((d<=1) && (sensors_value[2]<thres)
    )
{
  printf("\n Too near of the edge -> turn 
       left \n");
  linear_speed=80.0;
  angular_speed=20.0;
}
//////////////////////////
// Otherwise
// Go ahead
/////////////////////////
else
{
```



(a) Initial Position (b) Final position

Figure 5. Representation of the perpendicular criterion



(a) Initial Position (b) Perpendicular position but the robot is blocked (c) Final position, perpendicular and free field at the front

Figure 6. Representation of the perpendicular criterion

```
  printf("\n Going ahead \n");
  linear_speed=80.0;
  angular_speed=0.0;
}
```

Two cases were implemented:

- If the robot is too near of the edge, the robot will escape this edge.
- Otherwise, the robot will go ahead.

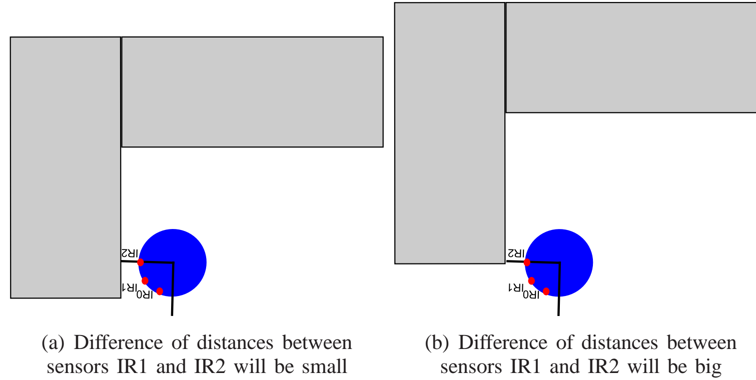
In the first case, if the distance returns by the sensor IR2, the robot will rotate a little to the right to escape the edge. Otherwise, if the robot is not to near of the edge, it will go ahead.

2) Difference of distances important: A typical case is presented on the figure 7(b). The difference of the distances between sensors IR1 and IR2 will be very important. In order to have the adequate value of angle speed, we implemented three different case based on the minimum distance return by the sensors IR1 or IR2. The code implementing this task is shown below:

```
////////////////////////////////
// Definition of variables
// Definition of the slope
double m=10;
//  Parameter for the linear
// function to use the angle
double p1 = 5;
double p2 = 25;
double p3 = 10;
////////////////////////////////

////////////////////////////////
// We have to compute the
// angular speed depending
// of the distance of where
// the robot is from
// the border of the obstacle
////////////////////////////////

/////////////////////////////////
// if the robot is near
// of the corner
// turn the softest as possible
/////////////////////////////////
if (min_sensor<thres - 0.5)
{
  linear_speed=80.0;
  angular_speed=-((m*min_sensor)+p1);
```



(a) Difference of distances between sensors IR1 and IR2 will be small

(b) Difference of distances between sensors IR1 and IR2 will be big

Figure 7. Introduction of the distance sensors criteria

```
  printf("\n Corner near: Turn soft right 
      \n");
}
////////////////////////////////////
// if the robot is going
// to far of the corner
// Turn more to the right
////////////////////////////////////
else if (min_sensor > thres)
{
  linear_speed=80.0;
  angular_speed=-((m*min_sensor)+p2);
  printf("\n Corner far: Turn hard right \
      n");
}
////////////////////////////
// Otherwise
////////////////////////////
else
{
  linear_speed=80.0;
  angular_speed=-((m*d)+p3);
  printf("\n Turn right \n");
}
```

The three different cases are:

- The robot is far of the obstacle and have to turn hard to the right.
- The robot is near of the obstacle and have to turn soft to the right.
- Otherwise turn right normally.

The angle speed is defined in each case by three different linear functions shown on figure 8.

3) State changement: It is possible that the robot looses the obstacle or meets another obstacle. In this case, we have to change the state. The following code presents the implementation of state change.

```
/////////////////////////////////
```

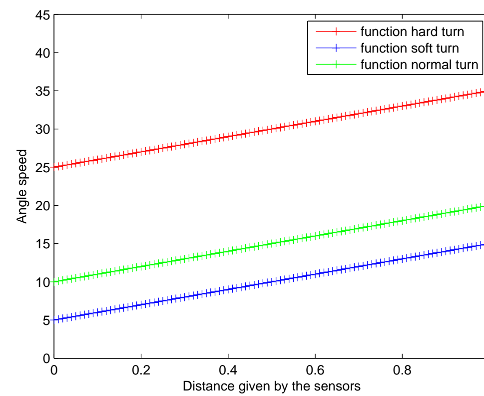


Figure 8. Three different function to select the angle speed

```
// If we detect an object at
// the front of the robot
// -> state 1 to be
// perpendicular and have
// free front range
///////////////////////////////
if((sensors_value[0]<thres)||(
    sensors_value[7]<thres))
{
  printf("\n Object at the front -> state 
      1 \n");
  state = 1;
}
```

```
////////////////////////////////////
// If the sensors of the side
// have big values, it means
// that the robot doesn't
// follow anymore an obstacle
// -> Go to the state 0
// to find a new obstacle
////////////////////////////////////
```

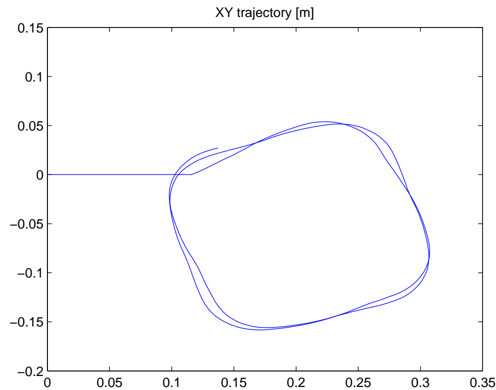


Figure 9. Simple example

```
else if((sensors_value[1]>=3.5) && (
    sensors_value[2]>=3.5))
{
  linear_speed=80.0;
  angular_speed=0.0;
  state=0;
  printf("\n Loose the object -> state 0 \
      n");
}
```

If an object is detected at the front using the sensors IR0 and IR7, the state 1 is activated. If the robot does not have any object on the right (IR2 and IR1), the state 0 is activated.

IV. Results and conclusion

In this part, we will present to different example with different complexities.

A. Simple example

A simulation was performing with one cube. The robot has to follow the object and the trajectory was recorded and shown on the figure 9.

We can observe that the motion carried out is almost cubic. The edge parts are flat without big discontinuity. The corners are not taken in an abrupt way. The repeatability is good. Moreover the robot does not lose the obstacle in any case.

B. Complex example

A simulation was performing with three cubes in ”L” position. The robot has to follow the object and the trajectory was recorded and shown on the figure 10.

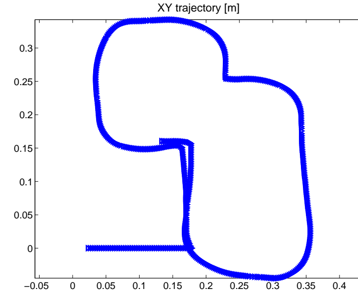


Figure 10. Complex example

Simulating a complex example, the results are the same than a simple example. The trajectory is not far of the object. Moreover, the robot does not lose the robot in any case.

APPENDIX A
CODE

```
#include <webots/robot.h>
#include <webots/differential_wheels.h>
#include <webots/distance_sensor.h>
#include <webots/light_sensor.h>
#include <webots/camera.h>
#include <webots/accelerometer.h>
#include <math.h>
#include <time.h>
#include <sys/time.h>
#include <stdio.h>

#define TIME_STEP 256
#define WHEEL_RADIUS 0.0205
#define AXLE_LENGTH 0.053


/* GLOBAL VARIABLES */

double sim_time,real_time;
double l_past,r_past;
double dl,dr,d,da;
double theta,x,y;
double sensors_value[8];
double linear_speed,angular_speed;
double left_speed, right_speed;
double displacement;
double IR2dist=0;
int count=0;

double state, iniX, iniY, iniTheta;

static void compute_odometry() {
  double l = wb_differential_wheels_get_left_encoder();
  double r = wb_differential_wheels_get_right_encoder();

  dl = ((l-l_past) / 1000.0) * 2 * 3.141592* WHEEL_RADIUS; // distance covered by left wheel
       in meter
  dr = ((r-r_past) / 1000.0) * 2 * 3.141592* WHEEL_RADIUS;  // distance covered by right
       wheel in meter
  d  = (dr + dl) / 2 ;
  da = (dr - dl) / AXLE_LENGTH; // delta orientation
  r_past=r;
  l_past=l;
  displacement=displacement+(d*100);
  //printf("estimated distance covered by left wheel: %g m.\n",dl);
  //printf("estimated distance covered by right wheel: %g m.\n",dr);
  printf("estimated distance covered by the robot: %g m.\n",d);
  //printf("estimated change of orientation: %g rad.\n",da);

  //printf("Wheel distances: l= %f; r= %f; dl= %f; dr= %f; d= %f; da= %f;\n",l,r,dl,dr,d,da)
       ;


  theta=theta+da;
  x=x+d*cos(theta);
  y=y+d*sin(theta);

  printf("displacement: %f \n",displacement);
```

```
}

static void high_level_controller() {


//////////////////////////////////////////////
//// Definition of parameters
//////////////////////////////////////////////
double thres=1.5;


//////////////////////////////////////////////////////
///// Initialisation state
///// Go in ahead until to find an object
///// When an object is detected go to the state 1
//////////////////////////////////////////////////////
if(state==0)
{
  //// If ab object is detected
  if((sensors_value[0]<thres)||(sensors_value[1]<thres) ||(sensors_value[6]<thres) ||(
      sensors_value[7] < thres))
  {
    //// Go to the next state
    printf("Detected object -> state 1 \n");
    state=1;
  }
  //// Otherwise
  else
  {
    //// Go ahead
    printf("Going ahead \n");
    state=0;
    linear_speed=80.0;
    angular_speed=0.0;
  }
}

//////////////////////////////////////////////////////////////
///// State to put the robot perpendicular to an obstacle
///// and in the same time that no obstacle are on the
///// front of the robot
//////////////////////////////////////////////////////////////
if(state == 1)
{
  ///// If the robot is perpendicular to one obstacle
  ///// And that the front of the robot is free
  if((sensors_value[2] <(thres+0.3)) && ((sensors_value[1]>thres-0.3)&&(sensors_value[0]>
      thres)))
  {
    //// Go to the state 2
    printf("\n Perpendicular to the object and free field at the front -> state 2 \n");
    state=2;
  }
  ///// Otherwise
  else
  {
    //// Turn left
    printf("\n Turning left \n");
    state=1;
    linear_speed=0.0;
```

```
    angular_speed=100.0;
  }
}

// ////////////////////////////////////////////////////////////////
///// Following the obstacle the nearest possible
///// Until the moment that the robot detect something at
///// the front -> Return to the state 1 to be perpendicular
///// and have a free front range
// ////////////////////////////////////////////////////////////////

if(state==2)
{
  //// If the robot is perpendicular and near of the obstacle
  if(sensors_value[2]<thres+0.5)
  {
    // ////////////////////////////////////////////////////////////
    ///// Definition of variables
    ///// Definition of the slope
    double m=10;
    /////  Parameter for the linear function to use the angle
    double p1 = 5;
    double p2 = 25;
    double p3 = 10;
    // ////////////////////////////////////////////////////////////
    ///// Compute the distance between the sensor at 90
    ///// and the sensors at 45
    double d=sensors_value[1]-sensors_value[2];
    // /////////////////////////////////////////////////////////////
    // /////////////////////////////////////////////////////////////
    ///// Function to find the minimum distance between
    ///// the sensor at 90  and 45
    double min_sensor = 0;
    if (sensors_value[1]>sensors_value[2])
    {
      min_sensor = sensors_value[2];
    }
    else
    {
      min_sensor = sensors_value[1];
    }
    // ////////////////////////////////////////////////////////////

    // ////////////////////////////////////////////////////////////
    ///// If the distance between both sensor is so important
    ///// It seems that the robot is an corner of the obtacle
    ///// and has to turn right
    // ////////////////////////////////////////////////////////////
    if(d>1)
    {
      // //////////////////////////////////////////////////////////
      //// We have to compute the angular speed depending
      //// of the distance of where the robot is from
      //// the border of the obstacle
      // //////////////////////////////////////////////////////////

      // //////////////////////////////////////////////////////////
      //// if the robot is near of the corner
      //// turn the softest as possible
      // /////////////////////////////////////////////////////////
      if (min_sensor<thres - 0.5)
```

```
      {
        linear_speed=80.0;
        angular_speed=-((m*min_sensor)+p1);
        printf("\n Corner near: Turn soft right \n");
      }
      ////////////////////////////////////////////////////////////
      //// if the robot is going to far of the corner
      //// Turn more to the right
      ////////////////////////////////////////////////////////////
      else if (min_sensor > thres)
      {
        linear_speed=80.0;
        angular_speed=-((m*min_sensor)+p2);
        printf("\n Corner far: Turn hard right \n");
      }
      //////////////////////////////////////////////////////////////
      //// Otherwise
      //////////////////////////////////////////////////////////////
      else
      {
        linear_speed=80.0;
        angular_speed=-((m*d)+p3);
        printf("\n Turn right \n");
      }
    }
    //////////////////////////////////////////////////////////////
    //// If the robot follow the edge but is to near of
    //// the edge -> turn left
    //////////////////////////////////////////////////////////////
    else if((d<=1) && (sensors_value[2]<thres))
    {
      printf("\n Too near of the edge -> turn left \n");
      linear_speed=80.0;
      angular_speed=20.0;
    }
    ///////////////////////////////////////////////////////////////
    ///// Otherwise
    ///// Go ahead
    ///////////////////////////////////////////////////////////////
    else
    {
      printf("\n Going ahead \n");
      linear_speed=80.0;
      angular_speed=0.0;
    }


    //////////////////////////////////////////////////////////////
    //// If we detect an object at the front of the robot
    //// -> state 1 to be perpendicular and have
    //// free front range
    /////////////////////////////////////////////////////////////
    if((sensors_value[0]<thres)||(sensors_value[7]<thres))
    {
      printf("\n Object at the front -> state 1 \n");
      state = 1;
    }
  }

  //////////////////////////////////////////////////////////////////
  //// If the sensors of the side have big values, it means
```

```
    //// that the robot doesn't follow anymore an obstacle
    //// -> Go to the state 0 to find a new obstacle
    ////////////////////////////////////////////////////////////////
    else if((sensors_value[1]>=3.5) && (sensors_value[2]>=3.5))
    {
      linear_speed=80.0;
      angular_speed=0.0;
      state=0;
      printf("\n Loose the object -> state 0 \n");
    }
  }


  printf("\n State=%.1lf\n",state);


}


static void low_level_controller() {

  left_speed=linear_speed-angular_speed;
  right_speed=linear_speed+angular_speed;

}

static void update_log_file() {

  int i;
  FILE * logFile;

  logFile = fopen("logFile.txt","a");
  fprintf(logFile, "%f %f %f %f %f",real_time,sim_time,x,y,theta);

  for (i = 0; i < 8; i++) {
      fprintf(logFile, " %f",sensors_value[i]);
    }
  fprintf(logFile, " %f %f %f %f \n",linear_speed,angular_speed,left_speed,right_speed);

  fclose ( logFile );
  printf("Time: real=%0.3fs; sim=%0.3fs. \nPosition: x= %0.3fm; y= %0.3fm; theta= %0.3
      fdegree; \n",real_time,sim_time,x,y,theta*180/3.141592);
  printf("Distance sensors: %0.1f / %0.1f / %0.1f / %0.1f / %0.1f / %0.1f / %0.1f / %0.1f\n"
      ,sensors_value[0],sensors_value[1],sensors_value[2],sensors_value[3],sensors_value[4],
      sensors_value[5],sensors_value[6],sensors_value[7]);
  printf("Linear speed: %0.1f; Angular speed: %0.1f;\n\n",linear_speed,angular_speed);


}


int main(int argc, char *argv[]) {

  /* define variables */
  WbDeviceTag distance_sensor[8];
  double coef[8]={0};
  double value=0;
  int i;

  clock_t start, end;
```

```
  /* initialize Webots */
  wb_robot_init();

  /* get and enable devices */
  wb_differential_wheels_enable_encoders(TIME_STEP);


  for (i = 0; i < 8; i++) {
    char device_name[4];

    /* get distance sensors */
    sprintf(device_name, "ps%d", i);
    distance_sensor[i] = wb_robot_get_device(device_name);
    wb_distance_sensor_enable(distance_sensor[i],TIME_STEP);
  }

  /*initialize global variables*/
  sim_time=0.0;
  real_time=0.0;
  end = clock();
  theta=0.0;
  x=0.0;
  y=0.0;
  wb_robot_step(TIME_STEP);
  l_past = wb_differential_wheels_get_left_encoder();
  r_past = wb_differential_wheels_get_right_encoder();

  state=0;
  iniX = x;
  iniY = y;
  iniTheta = theta;
  count=0;
  displacement=0;
  state=0;
  /* main loop */
  for (;;) {
coef[8]=1.8658*pow(10,-27);
coef[7]=-3.0795*pow(10,-23);
coef[6]=2.1084*pow(10,-19);
coef[5]=-7.7546*pow(10,-16);
coef[4]=1.6596*pow(10,-12);
coef[3]=-2.0957*pow(10,-9);
coef[2]=1.5174*pow(10,-6);
coef[1]=-0.00058968*pow(10,0);
coef[0]=0.1154*pow(10,0);  // real p0=[-2.8796e-24,4.0641e-20,-2.309e-16,6.7637e-13,-1.0882e
    -09,9.5095e-07,-0.0004,0.0951]
  // sim p0 = [-3.5638e-24, 4.3924e-20, -2.1927e-16, 5.6962e-13, -8.2424e-10,6.6189e-07,
      -0.0003,0.06797]
  for (i = 0; i < 8; i++) {

  value = wb_distance_sensor_get_value(distance_sensor[i]);

  if (value >4000)
  {
  value=4000;
  }
  else if(value<130)
  {
  value=130;
  }
  sensors_value[i] = coef[8]*pow(value,8)+coef[7]*pow(value,7)+coef[6]*pow(value,6)+coef[5]*
```

```
      pow(value,5)+coef[4]*pow(value,4)+coef[3]*pow(value,3)+coef[2]*pow(value,2)+coef[1]*
      pow(value,1)+coef[0];
  sensors_value[i]=sensors_value[i]*100; //m to cm
  if (sensors_value[i]<0)
  {
   sensors_value[i]=0;
  }
  printf("sensors_values= %lf \n",sensors_value[i]);
  }


      /* get sensors values */
   // for (i = 0; i < 8; i++) {
   //   sensors_value[i] = wb_distance_sensor_get_value(distance_sensor[i]);
      //printf("distance %f \n",sensors_value[i]);
   // }

    /* compute odometry*/
    compute_odometry();

    /* compute high-level control */
    high_level_controller();

    /* compute low-level control */
    low_level_controller();

    /* set speed values */
    wb_differential_wheels_set_speed(left_speed,right_speed);

    /* save data in a log file */
    update_log_file();

    /* perform a simulation step */
    wb_robot_step(TIME_STEP);


    /* updating the real and simulating time */
    double cpu_time_used;
    start = clock();
    cpu_time_used = ((double) (start-end)) / CLOCKS_PER_SEC;
    end = start;
    real_time = real_time + cpu_time_used;
    sim_time = sim_time + TIME_STEP/1000.0;
  }

  return 0;
}
```