University of Aveiro
Electronics and Telecommunications' Department

# Cyclop

**Autonomous Mobile Robots**
**(Final Report)**

**Professor Keith L. Doty**
**Scott Jantz**

July 95

**Rosa Maria Charneca Pasadas - 6319 ET**
**Rui Jorge Ferreira da Costa - 5691 ET**

**Table of Contents**

**Abstract**

This paper is a **brief description of Cyclop** (presented firstly as **Fly**): an **autonomous mobile robot** that was designed to do simple obstacle avoidance, bump detection and floor absence detection (in order to prevent **Cyclop** from falling into holes or abysses). It **integrates six *IR* sensors**: three in the front for **obstacle avoidance** (used in the first presentation), two in front of Cyclop's two wheels for **abyss detection** and one pointing to a rotating mirror that is supposed to perform a 360º obstacle detection around the robot (this is **Cyclop's most salient characteristic**), **hence its name**. It also integrates three bumpers in the front to perform bump detection.

Cyclop has two actuators (motor drivers) that respond accordingly to the stimulus input at his sensors.

The **obstacle avoidance algorithm** used in the **first presentation** was performed by a **non-linear dynamic system differential equation** for **heading direction** dynamics and a **linear function** for **speed dynamics**.

The **obstacle avoidance algorithm** that will be used in the **second presentation** is performed by a **function that averages and differentiates the readings between the left and right** side of the robot **for heading direction** and a **non-linear dynamic system differential equation** for **speed dynamics**.

**Executive Summary**

In this project the **obstacle avoidance behavior** was **implemented in the first stage** with **three *IR* sensors in Cyclop's front**. **Later** on we tried to do it **with one *IR* led and one *IR* sensor** that were **above a rotating mirror**, and **finally** we used just **one *IR* sensor above the mirror** and **six pointing ahead *IR* leds in Cyclop's front**.

The six pointing ahead leds illuminate the obstacles and the mirror reflects the beams (reflected from the obstacles) to the *IR* sensor that's above it.

To **drive the motor that rotates the mirror**, we used a **PWM signal** that was programmed in the MC68HC11A0FN microprocessor in an adaptive way. For some unknown reason the PWM signal changed in an unusual way during program's execution and we decided to make a circuit that generates the PWM signal.

This circuit compares data from a counter and data that was defined by the user with four switches, giving a PWM signal as a result of this comparison.

This **evolution** will be **explained in more detail in the "Experimental Layout and Results" item**.

## Introduction

This project is integrated in the M. Sc. in Electronics Engineering subject "Autonomous Mobile Robots" that is still taking place in the University of Aveiro (June/July, 95). It consists of designing an autonomous robot, Cyclop, that can perform obstacle avoidance using a rotating *IR* sensor. We tested a particular algorithm for obstacle avoidance with three frontal *IR* sensors that worked fairly well. This last sensor gave us (and still is giving) many practical problems explained in detail in the "Experimental Layout and Results" item.

## Integrated System

The following figure is **Cyclop's** organisational description.

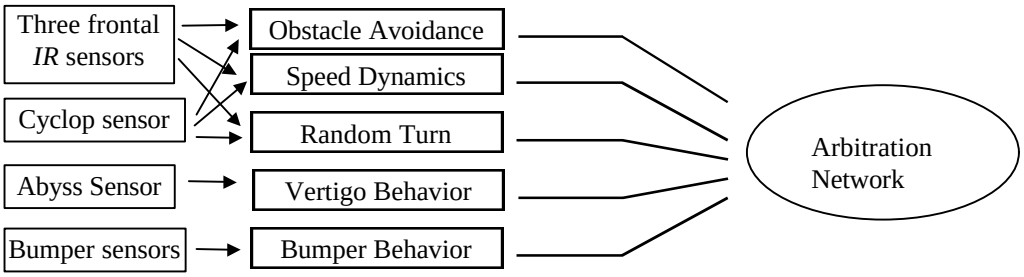
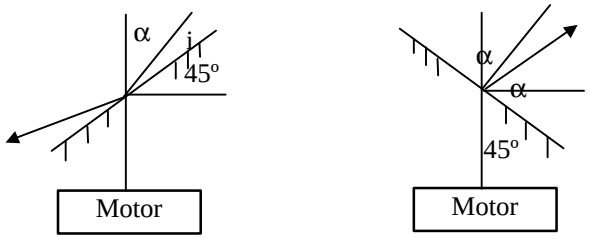



figure 1

**Mobile Platform**

At first we thought about using a 45º mirror and make the adjustment of the angle between the beam and the floor adjusting the angle between the *IR* led and the rotational axis. As one can see in the next figure, this is a basic error, for if in one instant the beam is pointing down, it will point up after the mirror rotates 180º.




I II

figure 2

To achieve a solution we need axial symmetry. The angle of the beam will have to be determined by the angle between the mirror and the floor:

- The *IR* emitter must be placed on the rotational axis
- β must be chosen to design γ

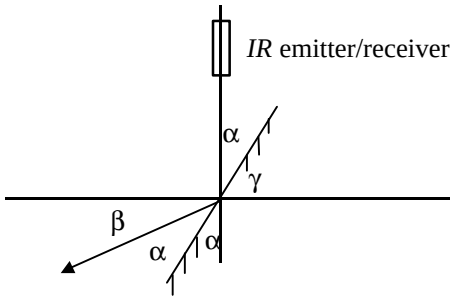



figure 3

$2\alpha+\beta=90^o$ and $\alpha+\gamma=90^o \Rightarrow$ **$\gamma=45+\beta/2$**

The mobile platform was designed as shown in the figure below.

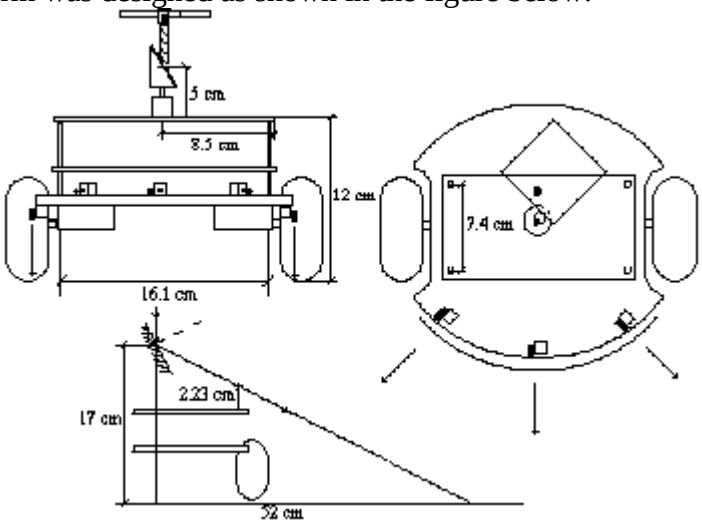



figure 4

**Actuation**

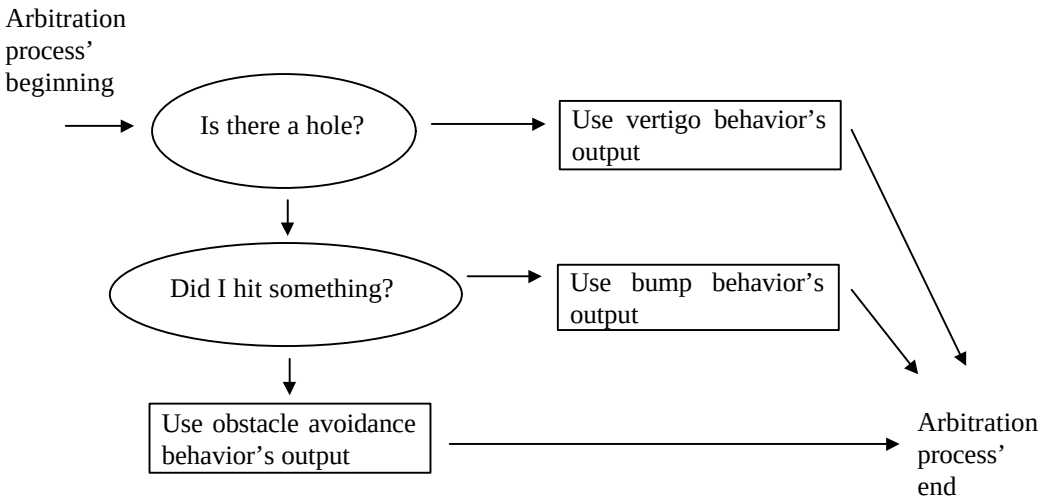



figure 5

**Sensors**

The following table summarises Cyclop's sensors.

| Physical Sensor | Function | Location | Quantity |
|---|---|---|---|
| *IR* | avoid obstacles | front, front left, front right | 3 |
| *IR* | avoid obstacles | up, pointed at the rotating mirror | 1 |
| *IR* | avoid holes | under the robot's board, before the wheels | 2 |
| Bumper | Bump detection | front | 3 |

The bumpers are simple switches that give the robot a digital input. They were designed the same way our colleagues' A. Branco and P. Kulzer.

The rotating *IR* sensor works as follows (see next figure):The illuminating leds beam the obstacle, which reflects radiation to the robot. If the reflected beam has the angle depicted in the "Mobile Platform" item it is felt by the *IR* sensor. The sensor's colimer eliminates other beaming directions. When the mirror (powered by a PWM driven motor) passes by the synchronising led it gives a pulse to the sensor. We can thus know where the sensor is looking at.

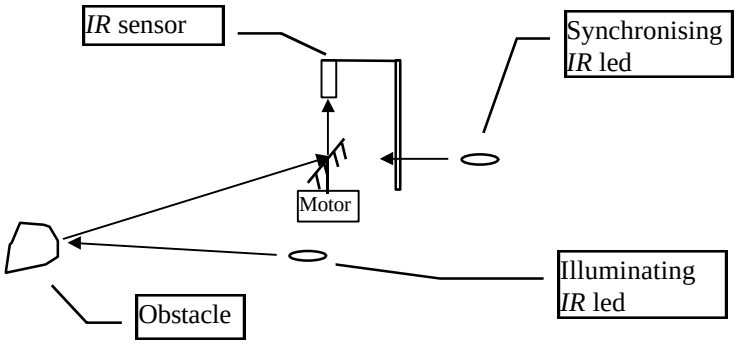



figure 6

**Behaviors**

•Obstacle avoidance with the three frontal *IR* sensors:

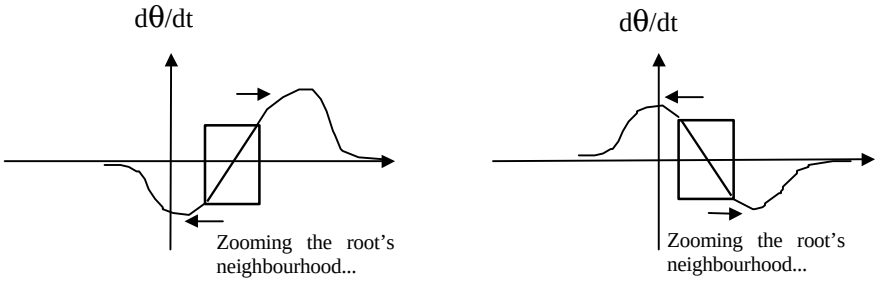
For the **first presentation** we just implemented **two behaviors**: **obstacle avoidance** (using the three frontal *IR* sensors) with controlled speed and **turn back** (when it reached a local minimum that he could not leave if it had only the obstacle avoidance behavior), currently modified to a random turn.

The **obstacle avoidance behavior** was **based on** a non-linear dynamic system differential equation (see **code in appendix** - obavoid file's process "obavoid"), i.e., something like:

$d\theta/dt = f(\theta)$,

where f(.) is a non-linear function [5].

Let us now introduce the concept of **repellor**: a repellor is a zero of the previous equation with negative values of dθ/dt at its left and positive at the right, at some neighbourhood of that point. Like this, if we are close to the zero at its right, dθ/dt >0, so θ increases, getting away from the zero. The same happens at the left of the zero: we get more and more away from the zero, hence the name.



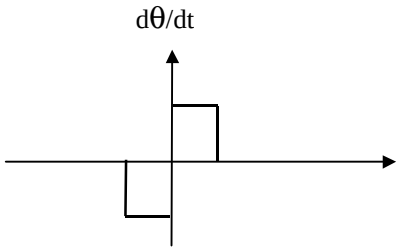

**Repellor** **Attractor**

figure 7

We understand easily the concept of **attractor** under the same ideas.

We can not use simple linear functions like the ones above to implement obstacle avoidance for a repellor has a limited influence around him, and the repellor represented has unlimited influence.

In our task we used the following function as the basic repellor:




**Repellor**

figure 8

As the obstacle gets closer, the height and width of the two pulses become higher. The integration of the three frontal sensors is accomplished by an addition of the resultant repellor functions.

The speed dynamics is supposed to be directly related to the duty-cycle of the motor drivers, and is accomplished by a simple linear function:

V = Vmax -m*(Sensor-S0), where m=(Vmax-Vmin)/(Sat-S0)

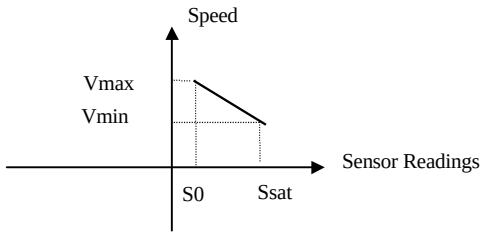



figure 9

When the robot falls in local minima its speed decreases to a certain value. When the robot's arbitration network detects that, it switches to the *turn back* behavior.

Mathematical representation of the world might be seen by some as traditional AI [4]. It must be seen as well under another point of view: mathematical language is a synthetic one. One can say much using few words, thus reducing substantially the code.

•Obstacle avoidance using the rotating *IR* sensor

Cyclop also supports **bump and vertigo behaviour**, although these were not implemented by the time of the first presentation.

After a bump or a dangerous height detection it turns back for a while. Afterwards it turns to the left or to the right if it felt the stimulus at the right or at the left, respectively. If the stimulus is symmetric it turns back for a while and performs a random rotation to avoid the problem.

We weren't able though to use the vertigo behavior integrated with the rotating *IR* sensor. We could use it only with the three frontal *IR* sensors, for we needed a high intensity on the leds used to illuminate the obstacles for the rotating *IR* sensor and this interfered with the vertigo *IR* sensors. We could not switch between the two groups of leds easily because of a timing problem that will be later detailed (Experimental Layout and Conclusion), so we only used this behavior in the three frontal *IR* detector case.

The heading direction dynamics used in the rotating *IR* sensor case is simple: we make a weighted average on the left and the right side of our retina (with greater weights on the

dead-ahead direction), compare the values and decide thus which side to turn, if at all.

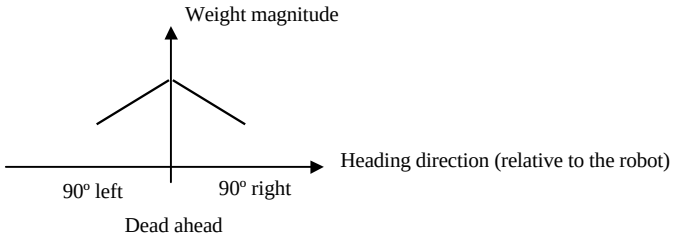



figure 10

For the speed dynamics we used a non-linear dynamic system differential equation:

$$\frac{dv}{dt} = -v*f(R_{max})+\beta*v,$$

where $\mathbf{R_{max}}$ is the retina's maximum sensor reading and $\beta$ is a constant.

This equation is non-linear in the sense that **f(.)** is a non-linear function that depends on the robot's sensor readings, which depend themselves on the robot's cinematic variables, among which is the robot's speed.

When the retina has saturation readings, $f(R_{max})$ is -1, giving an attractor to zero. When there are no obstacles, $f(R_{max})> 0$, giving an attractor to infinity. We added another attractor to infinity ($\beta*v$) to allow the robot to escape from local minima where it would be trapped. In case it hits the obstacle, the bumper switches to another behavior.

**Experimental Layout and Results**

In this item we will describe the problems we experienced and the causes we think that caused them.

In the beginning we used both a led and an *IR* sensor above the mirror, as close as possible to the motor's rotational axis. The synchronising led's pulse is reproduced in situation a).


figure 11

As one can see, the synchronising pulses have a too large width. This can hide obstacles of low magnitude. This could be due to two factors: an *IR* sensor frequency response too low for high frequencies or led or sensor's miscolimming. We changed the SHARP *IR* sensor output *OPAMP*'s capacitor to 4.7nF.

Afterwards we put three colimed leds in the front of the robot, instead of close to the sensor, thus spreading the light to illuminate the obstacles like a car at night. The situation did not improve very much. Finally we uncolimed the three leds and colimed the sensor. The colimed sensor avoids direct radiation from both the synchronising led and the illuminating ones, selecting only the reflected radiation from the obstacles in the angle defined in the "Mobile Platform" section. The results were much improved, as we can see in the next picture.

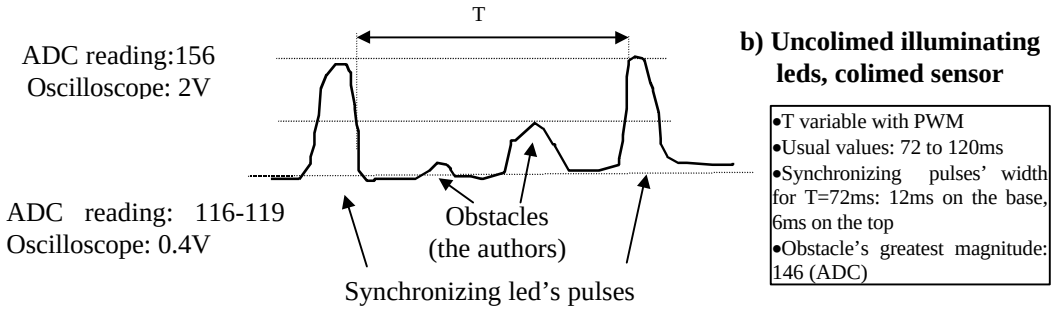



figure 12

For the first test our PWM was generated by the CPU and updated according to a PI algorithm:

```
{...}
 /*Duty cycle processing*/

 error = IrMotorPeriod-DESIRED_IR_MOTOR_PERIOD;

 Integral += error*Ki;/*Determining integral additive...*/

 /*if(Integral>0.8) Integral=0.8;
 if(Integral<-0.8)  Integral=-0.8;/*Limiting  integral
additive...*/

 dutyCycle = Kp*error + Integral;/*PI controller*/
 if (dutyCycle> 1.0) dutyCycle=1.0;
 if (dutyCycle< 0.0) dutyCycle=0.0;
{... }
```

The period read from the board was unstable. So, we programmed a constant PWM.

We simply used the OC1M4 and OC1D4 bits to program port A's pin 4, just like the following process' code extract:

```
...

bit_set(OC1M, 0b00010000);
while(1){
  bit_set(OC1D, 0b00010000);
  wait(0.07);
  bit_clear(OC1D, 0b00010000);
  wait(0.03);}
...
```

The resulting PWM was barely correct, but in some periods there was no signal:

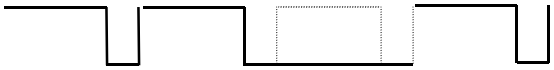


figure 13

Afterwards we tried to use the HC11's *TOC4* register. The PWM was OK, but the system time was not functioning properly. It would measure time delays of 1 second while we measured 5 seconds in our watches.

Finally we used a binary counter, a binary comparator and a 555 to generate our PWM, as can be seen in the electrical scheme. We used a very simple routine to detect the signal that our synchronising led gives when the mirror faces the back of the robot, as one can see in the "vision()" function (see "cyclop.c" file, in the appendix).

The mirror's rotation period read from the board was usually lower than the correct one, read from the oscilloscope, and it varied a lot, while the oscilloscope waveform had a very stable period. There were times, however, when the period read from the board was quite stable. Usually this occurred when the vision routine was the only one running.
When we added all the code, the two periods would only match if their value was above 100ms. In this situation the ADC values read from the sensor were also reasonably stable and coherent with the obstacles' position and distance.
In one experiment we took 240 samples at Ts = 1ms, registering the retina values with their respective instants. The result was a distance between synchronising peaks of 23 samples. That would mean a rotating mirror's period of 23ms. However, reading the

difference between any consecutive samples' instants we measured 4ms! That means a 92ms period, a value that matched the oscilloscope measured period.

In summary, when the period read from the board was too short compared to the one read from the oscilloscope or when it was unstable, the ADC values were inaccurate. When the period was shorter than about 100ms, the retina values became nonsense, taking into account the obstacles' places.

Our last code uses neither processes nor local variables. We decided to do so comparing the two possibilities on their rotating periods and retina's coherence with what we saw on the oscilloscope.

**Conclusion**

Without a stable period it is impossible to relate the samples we are taking to their positions, because we are basing ourselves on the last period to calculate the next sample period. Even if you do a completely asynchronous routine, you still have to decide when are you going to sample again, so the last rotating period must be close to the next one.

We are sure that the smaller period is not due to close obstacles' readings because we took measurements with the motor standing still and they were quite lower than the synchronising led's signal (146, as described in the previous item, against a 150 threshold).

We are also sure that the ADC is working properly. If it wasn't, we would not give coherent values at lower periods.

We suspect that the following factors somehow give problems to the system timer:

- Using processes: we compared just the sample acquisition routine as a process and as a function inside an infinite loop in the main function. The second situation's results were more reasonable.
- Using variables inside the functions: Somehow the results were more logic using all variables as global, at least in some cases.

We have a fair obstacle avoidance and speed dynamic algorithm for the three frontal obstacle avoidance *IR* sensors and operational bump and vertigo behaviors, although there are calibration problems in peculiar situations of the vertigo sensors.

Our robot's main characteristic is the rotating *IR* sensor. Its hardware is OK and the vision routine works correctly for periods lower than 100ms (otherwise the rotating period becomes extremely incoherent and unstable). However, as strange as it may be, our current motor's period is of about 72ms, and the obstacle avoidance behavior works, although we are quite sure that its retina's values are quite inconsistent at that frequency.

All the facts detailed in the experiments point out that the problem is a timing one. So we can only conclude that *Interactive C* cannot handle our task's timings.

If we started our project again we would definitely use assembly language. Another important thing that anyone that plans to build a robot with a similar sensor should know is that the sensor above the mirror should be colimed.

This robot is not *R2D2* or *3CPO* but it worked in spite of all the troubles we had. So perhaps it shouldn't be called *Waterloo*. Perhaps it deserved the *Cyclop* name.

**Documentation**

[1] Tracy L. Anderson and Max Donath, "Animal Behavior as a Paradigm for Developing Robot Autonomy", edited by Pattie Maes, MIT/Elsevier; pp 145-168.
[2] Randall D. Beer, Hillel J. Chiel, and Leon S. Sterling, "A Biological Perspective on Autonomous Agent Design", edited by Pattie Maes, MIT/Elsevier; pp169-186.
[3] Valentino Braintenberg, "Vehicles: Experiments in Synthectic Psychology".

[4] R. A. Brooks, "Elephants don't play chess" - extracted from "Designing Autonomous Agents", edited by Pattie Maes, MIT/Elsevier, pp3-15.
[5] John J, D'Azzo, Constantine H. Houpis, "Linear Control System Analysis and Design Conventional and Modern", McGraw-Hill International Editions, 3rd edition, pp 21, 505-541.
[6] Keith L. Doty and Akram Bou-Ghannam, "Controlling Situated Agent Behaviors with Consistent World Modeling and Reasoning".
[7] Keith L. Doty and Reid R. Harrison, "Sweep Stractegies for a Sensory-Driven, Behavior-Based Vacuum Cleanning Agent", AAAI 1993 Fall Symposium Series Instantiating Real-world Agents.
[8] Keith L. Doty, Stefano Caselli, Reid R. Harrison, Francesco Zanichelli, "Landmark Map Construction and Navigation in Enclosed Environments".

[9] Keith L. Doty, Stefano Caselli, Reid R. Harrison, Francesco Zanichelli, "Mobile Robot Navigation in Enclosed Large-scale Space".
[10] Keith L. Doty, "Position Paper on Lessons Learned from Implemented Software Architectures for Physical Agents".

[11] Keith L. Doty and Steven Louis Seed, "Autonomous Agent Map Construction in Unknown Enclosed Environments", MLC-COLT'94 Robot Learning Workshop, Rutgers, New Brunswick, N.J.

[12] Joseph L. Jones, Anita M. Flynn, "Mobile Robots Inspiration to Implementation", A.K.Peters, Ltd.

[13] K.S. Fu, R.C. Gonzalez, C.S.G. Lee, "Robotics Control, Sensing, Vision, and Intelligence", McGraw-Hill International Editions, pp. 267-293.

[14] Richard D. Klafter, Thomas A. Chmielewski, Michael Negin, "Robotic Enginnering An Integrated Approach", Prentice Hall, pp. 314-508, pp. 665-691.

[15] Fred Martin, "The 6.270 Robot Builder's Guide for the 1992 MIT LEGO Robot design Competition".

[16] Gregor Schoner and Michael Dose, "A dynamical systems approach to task-level system integration used to plan and control autonomous vehicle motion".

## Appendices

**File "obavoid.c": source code for Obstacle avoidance using the 3 frontal *IR* sensors, vertigo and bump behaviors**:

```
/*CONSTANTS*/

/*IR sensor readings when there are no obstacles*/
int Sensor0=105;

/*IR Sensor saturation readings*/
int SensorSat=160;

/*IR Sensor security reading*/              /*######*/
int SensorSec=150;

/*IR left, center and right sensors' positions*/
float L_SENSOR_DEGREE = -45.0;
float C_SENSOR_DEGREE = 0.0;
float R_SENSOR_DEGREE = 45.0;

/*Repellor's width factor*/
float WIDTH_FACTOR = 4.0;

/*Obstacle's weight factor*/
float OBSTACLE_WEIGHT = 1.5;

/*Motor Compensation (one of our motor is slower than the other)*/
float MOTOR_COMPENS = 0.81;
```

```
/*Speed dynamics parameter (slope)*/
float m = 90.0/( (float) (SensorSat-Sensor0));

/*GLOBAL VARIABLES*/

/*IR Sensor readings initializations*/
int lSensor=Sensor0,
    cSensor=Sensor0,
    rSensor=Sensor0,
    lDownSensor=SensorSec,                  /*######*/
    rDownSensor=SensorSec;

/*Bumper readings initializations*/
int lBumpSensor,                /*$$$$*/
    cBumpSensor,
    rBumpSensor;

/*Motor power obstacle avoidance behaviour's output*/
float lMotorObavoid, rMotorObavoid;


/*Vertigo behaviour's output*/          /*#######*/
int vertigoBack,
    vertigoRight,
    vertigoLeft;

/*Bump behaviour's output*/        /*$$$$$$$*/
int bumpBack,
    bumpRight,
    bumpLeft;

/* behavior's output flags */      /*&&&&&&&*/
int bumpFlag=0,
    vertigoFlag=0,
    avoidFlag=0;

/*Speed dynamics' output*/
float Speed_dSpeed;


/*Robot's speed*/
float speed;



/*AUXILIARY FUNCTIONS*/

float abs(float x)
{
 if (x<0.0) return(-x);
 else return x;
}
```

```
float repellor(float x, float w)
/*
  "x" is the obstacle's position
  "w" is the repellor's width (allways positive)
*/
{
 if ( x<-w || x==0.0 || x>w)
   return 0.0;
 else if (-w<=x && x<0.0)
                return (-1.0);
 else if (x>0.0 && x<=w)
                return 1.0;
}


void wait(int milisec)
{
 long timer_a;

 timer_a = mseconds()+ (long) milisec;

 while(timer_a > mseconds())
   {
    defer();
   }
}

int random()
{
 return (int) mseconds() & 0x3FF;
}


void turn_back(float speed)
{
  motor(0, -speed);
  motor(1, -speed);
  wait(random()+500);
}


void turn_left(float speed)
{
  motor(0, -speed);
  motor(1, speed);
  wait(random()+500);
}


void turn_right(float speed)
{
  motor(0, speed);
  motor(1, -speed);
  wait(random()+500);
}
```

```
void random_turn(float speed)
{
 if (random() & 1)
   turn_left(speed);
 else
   turn_right(speed);
}


void sound(float frequency, float length)
{
 pokeword(0x26,(int)(1E6 / frequency));
 bit_set(0x1020,0b10000000);   /*PA7*/
 sleep(length);
 bit_clear(0x1020,0b10000000);
 pokeword(0x26,0);
 bit_clear(0x1000,8);          /*turn power to spkr off*/
}


/*PROCESSES/BEHAVIOURS*/

/*Sensing process*/
void sensing()
{
 while(1)
   {
    lSensor=analog(1);
    cSensor=analog(2);
    rSensor=analog(0);
    lDownSensor=analog(3);                  /*######*/
    rDownSensor=analog(5);
    lBumpSensor=digital(2);             /*$$$$$$*/
    cBumpSensor=digital(1);
    rBumpSensor=digital(0);
   }

}


/*Speed dynamics process*/
void dSpeed()
{
 int maximum;

 while(1)
   {
    maximum = lSensor;
    if (cSensor>maximum) maximum=cSensor;
    if (rSensor>maximum) maximum=rSensor;

    Speed_dSpeed = 100.0-m*( (float) (maximum-Sensor0));
   }
}
```

```
/*Obstacle avoidance behaviour's process*/
void obavoid()
{
 float headDot;
 float left, center, right;

 while(1)
   {
    left= (float) (lSensor-Sensor0);
    center= (float) (cSensor-Sensor0);
    right= (float) (rSensor-Sensor0);

    headDot = left*repellor(-L_SENSOR_DEGREE,
                                                left*WIDTH_FACTOR)+
                        center*repellor(-C_SENSOR_DEGREE,
                                                center*WIDTH_FACTOR)+
                        right*repellor(-R_SENSOR_DEGREE,
                                                right*WIDTH_FACTOR);

    headDot = headDot/( (float) (SensorSat-Sensor0));

    lMotorObavoid = speed*(1.0+OBSTACLE_WEIGHT*headDot);
    rMotorObavoid = speed*(1.0-OBSTACLE_WEIGHT*headDot);
    avoidFlag=1;
   }

}

/*highs avoidance behaviour's process*/          /*####*/
void vertigo()
{
 while(1)
  {
   if (lDownSensor < SensorSec && rDownSensor < SensorSec)
    {
            vertigoFlag=1;
            vertigoBack=1;
            vertigoRight=(random() & 1);
            vertigoLeft=(random() & 1);
    }
   else if (lDownSensor < SensorSec)
    {
            vertigoFlag=1;
            vertigoBack=1;
            vertigoRight=0;
            vertigoLeft=1;
    }
   else if (rDownSensor < SensorSec)
    {
            vertigoFlag=1;
            vertigoBack=1;
            vertigoLeft=0;
            vertigoRight=1;
    }
   else vertigoFlag=0;
  }
}
```

```
/*bump behaviour's process*/          /*$$$$$$$$*/
void bump()
{
 while(1)
   {
            if (!lBumpSensor && !rBumpSensor)
             {
              bumpFlag=1;
              bumpBack=1;
              bumpRight=(random() & 1);
              bumpLeft=(random() & 1);
             }
            else if (!lBumpSensor)
             {
              bumpFlag=1;
              bumpBack=1;
              bumpRight=0;
              bumpLeft=1;
             }
            else if (!rBumpSensor)
             {
              bumpFlag=1;
              bumpBack=1;
              bumpLeft=0;
              bumpRight=1;
             }
            else if (!cBumpSensor)
             {
              bumpFlag=1;
              bumpBack=1;
              bumpRight=(random() & 1);
              bumpLeft=(random() & 1);
             }
            else bumpFlag=0;
   }
}

/*Arbitration/integration process*/
void arbitration()
{
 while(1)
   {
    if (vertigoFlag)            /*######*/
            {
             if (vertigoBack==1)
              {
               turn_back(30.0);
               random_turn(30.0);
              }
             if (vertigoLeft==1)
              {
               turn_left(30.0);
               random_turn(30.0);
              }
```

```
            if (vertigoRight==1)
             {
              turn_right(30.0);
              random_turn(30.0);
             }
/*      sound(1000.,.2);*/
           }
           if (bumpFlag)
           {
            if (bumpBack)  turn_back(25.0);
            if (bumpLeft)  turn_left(25.0);
            if (bumpRight) turn_right(25.0);
/*      sound(1500.,.2);*/
           }
           if (avoidFlag)
           {
            speed=Speed_dSpeed;

            if (speed<26.0 && abs(lMotorObavoid-rMotorObavoid)<1.0)
                    random_turn(40.0);
            else
              {
                    motor(1,lMotorObavoid);
                    motor(0,MOTOR_COMPENS*rMotorObavoid);
              }
           }
  }
}


/*MAIN FUNCTION*/
void main()
{
 poke(0x4000,0x37);

 start_process(sensing());
 start_process(vertigo());
 start_process(bump());
 start_process(dSpeed());
 start_process(obavoid());
 start_process(arbitration());
}
```

**File "cyclop.c": source code for obstacle avoidance and bump behaviors for the rotating sensor processes**

```
/*CONSTANTS*/

/*synchronizing led's thresholds*/
int SYNCHRO_IR_HI=150;
int SYNCHRO_IR_LOW=135;

/*Rotating IR sensor readings when there are no obstacles*/
int rSensor0=117;

/*Rotating IR Sensor saturation readings*/
int rSensorSat=160;


int MAX_CELLS=16;/*Number of visual cells*/
float INV_MAX_CELLS = 1./( (float) MAX_CELLS);

/*Thus we define a matrix of a greater size than we really
need (prevent from getting out of its range*/
int MAX_INDEX=4*MAX_CELLS;


/*Motor Compensation (one of our motor is slower than the other)*/
float MOTOR_COMPENS = 0.81;

float OBSTACLE_WEIGHT = 0.008;

/*Speed dynamics parameter (slope)*/
float rm = 1.0/( (float) (rSensorSat-rSensor0));



/*GLOBAL VARIABLES*/

/*Auxiliary flag to 1st peak synchronization*/
int FirstVision=1;

long IrMotorPeriod= (long) 2000;

/* "MAX_INDEX", to prevent array out of range error*/
int retina[64];

long time[64];/* Instants of the retina samples*/

int weights[64];/* Obstacle Weights*/

int index;

/*Motor power obstacle avoidance behaviors' output from rotating sensor*/
float lMotorObavoidRot, rMotorObavoidRot;

float speed = 55.0;
```

```
/*Speed dynamics' output*/
float Speed_dSpeed;

int lBumpSensor,
    cBumpSensor,
    rBumpSensor;



/*AUXILIARY FUNCTIONS*/

int abs(int x)
{
 if (x<0) return(-x);
 else return x;
}

float fabs(float x)
{
 if (x<0.) return(-x);
 else return x;
}


int random()
{
 return (int) mseconds() & 0x3FF;
}


void wait(int milisec)
{
 long timer_a;

 timer_a = mseconds()+ (long) milisec;

 while(timer_a > mseconds())
   {
    defer();
   }
}


void turn_back(float rSpeed, float lSpeed)
{
  motor(0, -rSpeed);
  motor(1, -lSpeed);
  wait(random()+500);
}
```

```
void random_turn(float rSpeed, float lSpeed)
{
 if (random() & 1)
  {
   motor(0, -rSpeed);
   motor(1, lSpeed);
   wait(random()+500);
  }
 else
  {
   motor(0, rSpeed);
   motor(1, -lSpeed);
   wait(random()+500);
  }
}



/*"PROCESSES"*/

/*Sensing process*/
void sensing()
{
    lBumpSensor=digital(2);
    cBumpSensor=digital(1);
    rBumpSensor=digital(0);

}
```

```
 long tf,t0,t;
 long deltaT; /*Rotating IR sensor's sample period*/
 int indy, ReadingRot;
void vision()
{
 /*
  The following processing is based on the synchronizing LED's signal.
  It gives a narrow pulse when the mirror passes by the back of the
  robot.
 */

      deltaT = (long) (( (float) IrMotorPeriod)*INV_MAX_CELLS);
        /*Determining the rotating IR sensor's sample period*/


      while(analog(4) > SYNCHRO_IR_LOW);
      /*Waits for negative transition*/


      t0=mseconds();
      /*Negative transition instant determination*/
      t= t0;
      indy=0;

      while(SYNCHRO_IR_HI > (ReadingRot=analog(4)) )
      /*Processing until positive transition */
        {
         tf=mseconds();

         if (tf>=t && indy<MAX_INDEX)
           {
            retina[indy] = ReadingRot;
            t += deltaT;
            time[indy] = tf;
            indy++;
           }

        }/*while(SYNCHRO_IR_HI > analog(4))*/

       indy--;
       index = indy;

       IrMotorPeriod = tf-t0;
       /*Updating IR Motor Period*/

}/*void vision()*/
```

```
/*Speed dynamics process*/
int maximum,i;

void dSpeed()
{

   maximum = retina[IMIN];

   for( i = IMIN+1;i < IMAX; i++)
     {
       if (retina[i] > maximum)
         maximum = retina[i];
     }

   Speed_dSpeed = speed*(1.0-rm*( (float) (maximum-rSensor0)));
      /*Dynamics driven by sensorial input*/

   Speed_dSpeed += 0.01*speed;
      /*Adding attractor to infinity...*/

   if (Speed_dSpeed < 15.0) Speed_dSpeed = 15.0;
}/*void dSpeed()*/



/* obstacle avoidance*/
 int IMIN = MAX_CELLS/4,
     IMAX = 3*MAX_CELLS/4;
 int left, right;

void obavoid()
{
   left = 0;
   for (i=IMIN; i<2*IMIN; i++)
     {
      if (retina[i] < rSensor0) break;
      left = left + (retina[i]-rSensor0)*weights[i];
     }

   right = 0;
   for (i=2*IMIN; i<IMAX; i++)
     {
      if (retina[i] < rSensor0) break;
      right = right + (retina[i]-rSensor0)*weights[i];
     }

   lMotorObavoidRot = speed*(1.0+OBSTACLE_WEIGHT*( (float) (left-right)));
   rMotorObavoidRot = speed*(1.0-OBSTACLE_WEIGHT*( (float) (left-right)));

}/*void obavoid()*/
```

```
/*Arbitration/integration process*/
void arbitration()
{
/*bump behaviour's process*/
  if (!lBumpSensor && !rBumpSensor)
       {
        turn_back(27.,27.);
        random_turn(27.,27.);
       }
  else if (!lBumpSensor)
        turn_back(30.,15.);

  else if (!rBumpSensor)
        turn_back(15.,30.);

  else if (!cBumpSensor)
       {
        turn_back(27., 27.);
        random_turn(27.,27.);
       }

  else /*Do simple obstacle avoidance*/
       {
        speed=Speed_dSpeed;

        /*motor(1,lMotorObavoidRot);
        motor(0,MOTOR_COMPENS*rMotorObavoidRot);*/

      }/*simple obstacle avoidance*/

}/*Arbitration*/



void main()
{

   for(i = IMIN;i < 2*IMIN; i++)
      weights[i] = 2*(IMIN-abs(i-2*IMIN+1));

   for(i = 2*IMIN ;i < IMAX; i++)
      weights[i] = 2*(IMIN-abs(i-2*IMIN));

    /*Let's turn on the lights...*/
    poke(0x4000, 120);

   while(1)
    {
     sensing();
     vision();
     dSpeed();
     obavoid();
     arbitration();
    }
}
```