Cs 4970.001/5973.002

April 30th, 2003

Neelam Chauhan
Josh Guice
Troy Humphrey
Mark Woehrer

Project 3

Deliberating and Acting

Final Report

Group 8

Table of Contents

Hardware Design

General Description

The hardware design chosen for this project is a 4 wheel differential drive model (Figure 1). Standard LEGO pieces are used to form a roughly square chassis atop which the HandyBoard is glued[1]. Two arms mounted in the front extend forward and form a *hatch* to capture and hold cubes. The various sensors are mounted either by hot glue or plastic zip-ties secured through holes in the LEGO technic pieces.

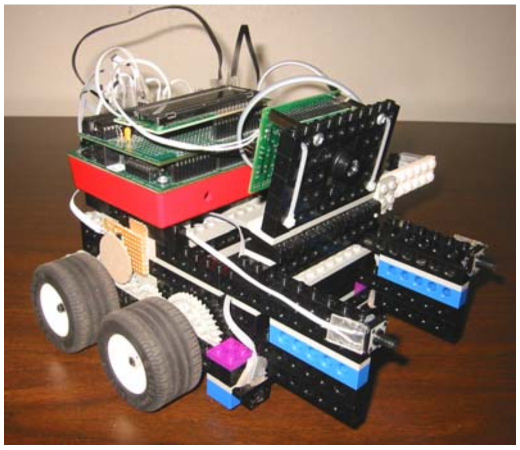


Figure 1

¾ View of Robot

Motors and Gearing

The 2 motors used are standard LEGO Technic 9v gear reduction motors. These motors are positioned perpendicular to the robot's direction of travel on either side of the main chassis in the rear. Each motor independently drives one side of the robot allowing the right/left drives to operate in any combination. The gears used are in an 8:40 tooth ratio from the motor axle to the rear wheel axle respectively. This provides a 5 to 1 reduction to the rear drive wheels increasing the amount of torque available to the robot's drive train. The rear drive axle is connected to the front axle with a 1:1 gear train.

Sensors

Two wheel encoders, two "tophat" sensors, two bump sensors, and a CMUcam vision system are attached to the robot. The wheel encoders (Figure 2) are custom built

[1] Mark discovered that covering a LEGO piece with vinyl tape, then gluing to it made removal of the glue much easier.

using Hamamatsu P5587 photoreflectors and encoding wheels (Figure 3) geared at the same RPM as the motors themselves[2]. This provides a straight driving resolution of 19.75 encoder ticks per inch traveled, and a pivoting resolution of 1.240 ticks per degree turned.

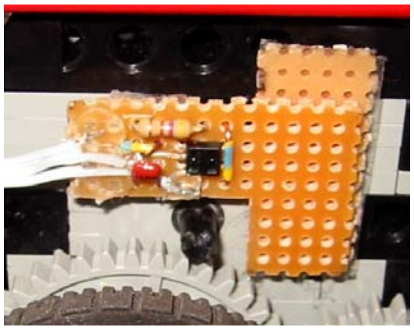


Figure 2

Custom wheel encoder

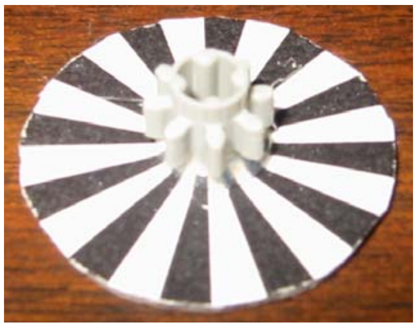


Figure 3

12 stripe encoder wheel

The CMUcam is frontally mounted using hinged LEGO pieces (Figure 4) so that its angle can be adjust as necessary to provide an optimum balance between range and nearsightedness.

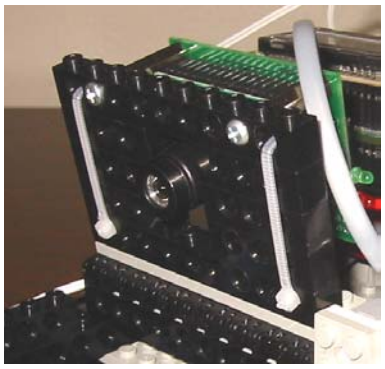


Figure 4

CMUcam on hinged mount

The two tophat style reflectivity sensors are mounted in the front on either side of the robot and point downward at the floor. This allows them to be used to detect the black tape that delineated goals in the arena. The two bump sensors are mounted on the front ends of the cube hatch and face forward relative to the robot. These sensors are positioned in this way to allow them to be used to detect wall collisions[3]. The majority of the hardware is devoted to odometry and calibrating the robot's position relative to its

[2] See Appendix A for encoder parts list and schematic
[3] This capability is used by the robot to reorient itself within the world (see software section)

environment (Figure 5). It was determined that proper calibration was crucial to the success of the robot in accurately moving about the arena for long periods of time.

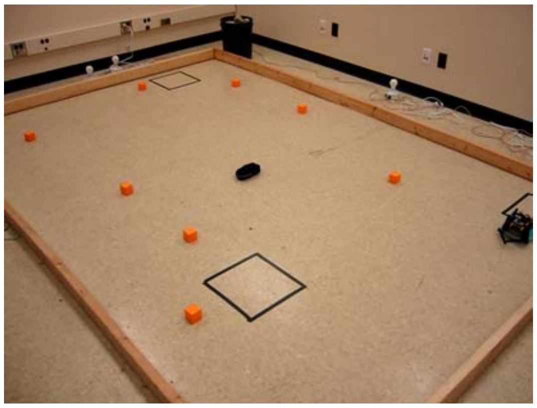


Figure 5

The robot's world

Calibration: Motors

It was found experimentally that the 100 motor power levels for the motor() function degenerate into just 8 levels (excluding zero). For example power levels from 25-37 represent the same power level level of 25. Once the exact values were found the following table was constructed.

```
int my_table[9]={0,13,25,38,50,63,75,88,100};
```

Individual motor tables were used to account for the uneven power output of the two motors used. An initial attempt was made to match the motor but in practice the robot did not drive straight. It was determined that each motor would have its own conversion table. This keeps the left motor from dominating the right motor. The exact values were found experimentally using an inductive method, and the following tables were constructed.

```
int left_motor_table[9] ={0,1,2,2,2,2,2,3,3};
int right_motor_table[9]={0,1,2,3,4,5,6,7,7};
```

Calibration: Turning/Driving

The turn and drive function were calibrated by counting the encoder ticks for a given angle of rotation or distance traveled. More effort was put into the turn function since it had the greatest effect on the positional uncertainty of the robot, i.e. small errors in turn angle could grow into large errors in position and orientation.

Calibration: Camera

Given the information on the CMUcam website we corrected the camera x-offset by modifying trackRaw() function found in cmucam3.ic. Specifically we subtracted 9 pixels for the x value.

Camera color calibration was done in YUV mode, using the floor as white reference (which worked signicantly better than a piece of white paper). We could detect both orange and blue targets with confidence of >150 at 6 foot. Our custom mean color values were derived from procedure described in "Track Window" function, which was much better than our initial attempts using the built-in tracking functions. The exact values are given in the following tables (notice the small deviation values):

# Hougen Neon Orange

Color	Value	Deviation
R	229	3
G	74	2
B	16	0

# Hougen Neon Blue

Color	Value	Deviation
R	74	3
G	58	1
B	148	4

Conclusions

Overall the hardware performed above expectations. The custom wheel encoders provided exceptional dead-reckoning capabilities and the general architecture was adequate for the task, but did not excessive in any regard. The combination of sensors was found to be very appropriate for the particular task and went hand-in-hand with the software design.

Software Design

General Description

The software architecture attempts to apply principals presented in the course material. Specifically materials related to the hybrid deliberate/reactive approach. The deliberative layer consists of a waypoint generator, path planner, and navigation module. While the reactive layer consists of tracking functions using the CMUcam and various odometry/calibration related functions using the encoders, tophat sensors, and bump sensors. There is also a locomotion layer responsible for handling the low-level movement commands and providing abstractions for the higher-level modules

Locomotion Layer

Our drive system commands contained two basic functions -- drive and turn. These commands were called sequentially and were allowed exclusive control of the drive motors. Both functions return the actual angle and distance. There are other related functions for calibration, lining up on a wall, lining up on the tape, measuring the distance traveled, and the angle turned.

File: loco.ic

***Primary functions*:**

void init_loco(void)
The init_loco() function initializes the locomotion code. This function calls the function init_encoders().

float drive(float dist)
The drive() function drives straight a distance given in inches. If the distance is greater than 8 inches the function uses a ramp for the motor power. At the end of the function the my_x,my_y global variables are updated to the robots actual position using the sin() and cos() functions. This function calls reset_dist() and read_dist() helper functions.

float turn(float angle)
The turn() function turns an angle given in degrees. This function uses a fixed power of the turn duration. At the end of the function the global my_theta variable is updated. This function calls read_angle() and read_angle() helper functions

Auxiliary functions:

int enter_goal(float dist)
The enter_goal() function drives the given distance looking for black line of the goal. This function is a modified version of drive() which uses a fixed power and reads the "Top Hat" sensors looking for a particular value above a threshold. This function updates my_x,my_y and returns a value indicating if a goal was found.

void cal_turn(void)
The cal_turn() function is used to calibrate the turn angle and is called at the beginning of a run. This funcion calls helper function turn_ticks() which is simplified version of the turn() function.

Reactive Layer

void stage(void)
The stage() function is used to line the robot up on a black line. This function includes a upper limit to the number of iterations so that an infinite loop is avoided. This function utilizes calls the motor commands and therefore does not update the world model. It calls the helper functions left_hat() and right_hat(). This function is based on code from Team 3, Project 1.

void stage_wall(void)
The stage_wall() function is used to line the robot up on a wall. This function includes a timeout to limit the duration of the procedure. This function directly calls the motor commands and therefore does not update the world model.

File: mycam.ic

Primary functions:

void cam_track_cube(void)
The cam_track_cube() function tracks the position of an orange object by making proportional adjustments to the robot's orientation. This function include a dead-zone which reduces unneeded corrections. This function calls the cam_find_cube() helper function.

Auxiliary functions:

void my_init_cam(void)
The my_init_cam() function performs the functions needed to initialized the camera, set the white balance, and init the color values for orange and blue.

void init_cube_values(void)
void init_blue_values(void)
The init_cube_values() and init_blue_values() functions initialize the min/max values needed for the trackRaw() function. These functions call the fix_color_val() helper function which makes sure the initiliazed min/max color values are within the valid range.

Deliberative Layer

The deliberative layer maintains a world model which consists of the location of all *known* objects[4] including the robot itself. It uses this information to find the nearest

[4] Unexplored areas of the robot's world were not tracked, however there were "slots" in the world model for newly discovered objects.

cube to the robot's current position, then computes 4 waypoints surrounding this cube (in the cardinal directions). The purpose of the waypoint is to place the robot a predetermined, constant distance from either a goal or a cube, at which point the reactive layer is invoked to perform the desired behavior. Once calculated, these waypoints are then evaluated for accessibility (i.e. inside the arena, not too close to a wall, etc.) and the closest remaining waypoint is selected as the secondary destination. The path planning module then plans a path to the waypoint, with the restriction that it may only turn the robot at right angles to the world[5]. The path which leaves the robot facing the destination after execution is generated using world-centric angles and distances. The navigation module then takes this plan and converts the world-centric angles to robot-centric angles and sends the appropriate commands to the actual turn and drive routines. This module also performs some optimization to minimize the amount of turning the robot performs.

File: deliberation.ic

***Primary functions*:**

void plan_path(float x, float y)
The plan_path() function is responsible for generating world-centered path instructions for reaching the x,y destination.

void compute_waypoints(float x, float y)
The compute_waypoints() function calculates four waypoints in orthogonal directions a predetermined distance from an x,y point.

int validate_waypoints()
The validate_waypoints() function sets a validity flag for each waypoint and returns false if no valid waypoint exists.

void navigate()
The navigate() function converts path information to robot-centric angles, optimizes these instructions, and issues the actual motor commands.

Auxiliary functions:

int get_nearest(int start, int end)
The get_nearest() function returns the index of the object in the world model array nearest the robot within the index range given.

File: main.ic

void main()
The main() function specifies the script that the robot will follow to carry out the required task. It first calls the various initialization routines, then starts a 25 minute loop in which the robot attempts to place cubes in the goals. First the script directs the robot to place all

[5] The robot starts facing orthogonal to the world.

known cubes in their nearest goals. Upon success, the cube is removed from the world model and the robot is instructed to realign itself using the arena walls to reset its position. When no more cubes exist in the world model, the script tells the robot to begin searching for cubes to place in goals. After the 25 minutes have elapsed, the second part of the script is executed. In this section, the odometry is turned off and the robot switches into "hunt" mode in which it attempts solely to contact the blue bot.

Conclusions

The software performed somewhat poorly during the demonstration due to issues with the CMUcam, small errors in last minute changes, and code that had been commented out for testing[6]. Overall, however, the software design is very robust and capable of performing the task perfectly during repeated trials (which it actually did in simulation). Behaviors that were conceived but not fully implemented include the following: the ability to predict the next position of the dynamic object based on two previous readings of its position and the robot's knowledge of the dynamic object and its own speeds, the ability to sense obstacles while executing path information and replan accordingly, and the ability to create a large number of intermediate waypoints to a destination and push these onto a stack of navigation commands which are executed from the top. Perhaps when *Advanced* Intelligent Robotics is offered, these concepts can become realities.

[6] This actually turned out to be beneficial, otherwise the robot would have "chimed" occasionally when it thought it had placed a cube in the goal (which it only did once by accident) and caused us to lose a point each time.

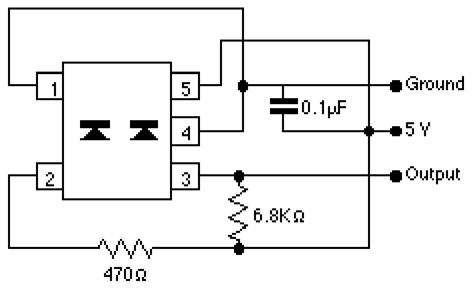
Appendix A – Custom Encoder Information

Parts List

The parts used in construction of the encoders (and their prices) are as follows:

Device Type	Value	Price Each	Catalog #	Supplier
Hamamatsu Photoreflector	P5587	3.25	R64-5587	Acroname
Resistor	470 Ohm	0.198	2711317	RadioShack
Capacitor	0.1uF	0.498	2720109	RadioShack
PC Board		0.845	2760148	RadioShack
	Total	4.791		

Schematic Diagram



Note: The 6.8KΩ resistor was omitted.

Appendix B – Table of Figures

Appendix C – Source Code

main.ic

```
#use "deliberation.ic"
#use "cmucamlib.ic"
#use "mycam.ic"

#define FALSE 0
#define TRUE 1

void main()
{
    int nearest_cube,nearest_goal,got_cube = FALSE,no_more_cubes =
FALSE;

    printf("Press Start\n");
    while(!start_button());

    init_loco();
    cal_turn();
    my_init_cam();
    init_wm();
    reset_time();

    while (get_time() < 20.0)
      {
        // get a cube if we don't already have one
        if (!got_cube)
          {
            // find nearest cube in world model (WM)
            nearest_cube = get_nearest_cube();

            // save cube location as global robot destination
            dest_x = wm[nearest_cube][0];
            dest_y = wm[nearest_cube][1];

            //printf("\nCube:%f,%f\n",dest_x,dest_y);

            // try to go to nearest cube
            while (! go_to_waypoint())
              {
                if (WM_has_cubes())
                  // no valid waypoints to cube exist, remove cube from
WM

                  remove_from_WM(nearest_cube);
                else
                  {
                    no_more_cubes = TRUE;

                    break;
                  }
            }
            if (no_more_cubes == TRUE)
              break;

            cam_track_cube();

            got_cube = TRUE;
        }

        nearest_goal = get_nearest_goal();

        // save goal location as global robot destination
        dest_x = wm[nearest_goal][0];
        dest_y = wm[nearest_goal][1];

        //printf("\nGoal:%f,%f\n",dest_x,dest_y);

        // try to go to nearest goal
        while (! go_to_waypoint())
          // no valid waypoints to goal exist, remove goal from WM
          remove_from_WM(nearest_goal);

        // Drives all the way to goal location
        // enter_goal() should replace this behavior
        //go_to_destination();

        if(enter_goal(W_STANDOFF))
          {
            play_tune();

            remove_from_WM(nearest_cube);

            got_cube = FALSE;

            recalibrate();
        }
        else
          {
            got_cube = TRUE;
        }
    }

    reset_time();

    while(get_time() < 5.0)
      {
        // kill blue bot
        //printf("Hunting...\n");
    }

    printf("Finished!\n");
}
```

loco.ic

```
#define TRUE 1
#define FALSE 0

#define LEFT_MOTOR 1
#define RIGHT_MOTOR 2

#define LEFT_ENCODER 0 /* PIN7 */
#define RIGHT_ENCODER 1 /* PIN8 */


#define INITIAL_THETA 0.0 /* South */
#define INITIAL_X 0.0
#define INITIAL_Y 0.0

#define LEFT_WALL 15
#define RIGHT_WALL 13

#define LEFT_HAT 2
#define RIGHT_HAT 3

#define MY_PI 3.141592654
//float my_x=INITIAL_X,my_y=INITIAL_Y;
float my_x,my_y;

float my_theta=INITIAL_THETA;


int old_right_ticks, old_left_ticks;
float velocity_T=0.1;
#define VELOCITY_SAMPLES 1
float left_vel_array[VELOCITY_SAMPLES];
float right_vel_array[VELOCITY_SAMPLES];
float velocity_const=19.1;
float left_vel,right_vel;
int left_stall,right_stall;

int int_abs (int input)
{
    if(input<0) input = -input;
    return input;
}

float stall_velocity=5.0;
int right_stall_state=0;
int left_stall_state=0;
long left_stall_start;
long right_stall_start;

void detect_stall (void){
    int lstall,rstall;

    for(;;){
        //printf("%d %d %d %d %d
%d\n",(int)left_vel,(int)right_vel,current_left_motor_power,current_rig
ht_motor_power,left_stall_state,right_stall_state);
        lstall=( left_vel < stall_velocity) && (int_abs(
current_left_motor_power) > 0);
        rstall=(right_vel < stall_velocity) &&
(int_abs(current_right_motor_power) > 0);

        //printf("%d %d\n",left_stall_state,right_stall_state);

        if(left_stall_state == 0){
            if(lstall){
                left_stall_state=1;
                left_stall_start=mseconds();
            }
        }
        else
          if(left_stall_state == 1){
              if(!lstall)
                left_stall_state=0;
              else
                if(mseconds()-left_stall_start>100L){
                    left_stall_state=2;
                    left_stall=1;
                }
          }
          else
            if(left_stall_state == 2){
                if(!left_stall){
                    left_stall_state=0;
                    left_stall=0;
                }
            };

        if(right_stall_state == 0){
            if(rstall){
                right_stall_state=1;
                right_stall_start=mseconds();
            }
        }
        else
          if(right_stall_state == 1){
              if(!rstall)
                right_stall_state=0;
              else
                if(mseconds()-right_stall_start>100L){
                    right_stall_state=2;
                    right_stall=1;
                }
          }
          else
            if(right_stall_state == 2){
                if(!right_stall){
                    right_stall_state=0;
                    right_stall=0;
                }
            }
```

```
    }
    /*
    if(right_vel < 0.1 && int_abs(current_right_motor_power) >= 2){
        right_stall=1;
    }else{
        right_stall=0;
    }
    */
    sleep(0.1);
}


void calc_velocity (void){

    int i;
    int right_ticks,left_ticks;
    float left_sum,right_sum;
    float left,right;
    for(;;){

        old_right_ticks=read_encoder(RIGHT_ENCODER);
        old_left_ticks=read_encoder(LEFT_ENCODER);

        sleep(velocity_T);

        right_ticks=read_encoder(RIGHT_ENCODER)-old_right_ticks;
        left_ticks=read_encoder(LEFT_ENCODER)-old_left_ticks;


        i++;
        if(i>=VELOCITY_SAMPLES) i=0;

        left_sum -= left_vel_array[i];
        right_sum -= right_vel_array[i];


left_vel_array[i]=(float)left_ticks/(velocity_const*velocity_T);

right_vel_array[i]=(float)right_ticks/(velocity_const*velocity_T);

        left_sum += left_vel_array[i];
        right_sum += right_vel_array[i];

        left_vel=left_sum/(float)VELOCITY_SAMPLES;
        right_vel=right_sum/(float)VELOCITY_SAMPLES;

    }
}

void init_encoders (void)
{
    enable_encoder(LEFT_ENCODER);
    enable_encoder(RIGHT_ENCODER);
}
```

```c
int my_table[9]={0,13,25,38,50,63,75,88,100};
int left_motor_table[9] ={0,1,2,2,2,2,2,3,3};
int right_motor_table[9]={0,1,2,3,4,5,6,7,7};


int current_left_motor_power;
int current_right_motor_power;

void my_motor(int moto, int input){
    int index;

    if(input<0){
        index=-input;
    }else{
        index=input;
    }


    if(index>8) index=8;

    if(moto == LEFT_MOTOR){
        current_left_motor_power=index; // this has to be first
        index=left_motor_table[index];
    }else
      if(moto == RIGHT_MOTOR){
          current_right_motor_power=index; // this has to be first
          index=right_motor_table[index];
      }else{
          beep();
          printf("mymotor");

      }

    if(input<0){
        motor(moto,-my_table[index]);
    }else{
        motor(moto,my_table[index]);
    }
    //printf("%d %d %d",input,index,my_table[index]);

}

//#define TICKS_PER_INCH 18.31
#define DRIVE_TICKS_PER_INCH 19.75

int old_lticks;
int old_rticks;

void reset_dist(void){
    old_rticks=read_encoder(RIGHT_ENCODER);
    old_lticks=read_encoder(LEFT_ENCODER);
}

float read_dist(void){
    float r_dist;
    float l_dist;
    float ave_dist;

    r_dist=(float)(read_encoder(RIGHT_ENCODER)-
old_rticks)/DRIVE_TICKS_PER_INCH;
    l_dist=(float)(read_encoder(LEFT_ENCODER)-
old_lticks)/DRIVE_TICKS_PER_INCH;

    ave_dist=(r_dist + l_dist)/2.0;


    //return (float)(read_encoder(RIGHT_ENCODER)-
old_ticks)/RIGHT_TICKS_PER_INCH;

    return ave_dist;

}


/* finish
float sign (float input)
{
    if(input>0.0
}
*/

float drive(float dist)
{
    //min 5.72 inch
    int i,pid;
    float actual_dist,abs_dist;

    printf("drive %d\n",(int)dist);

    if(dist > 0.0) abs_dist=dist;
    else abs_dist=-dist;

    reset_dist();

    if(dist>8.0){
        if(dist>0.0){
            for(i=0;i<=8;i+=1){
                my_motor(LEFT_MOTOR,+i);
                my_motor(RIGHT_MOTOR,+i);
                //if(left_stall || right_stall) break;
                sleep(0.1);
            }
        }else{
            for(i=0;i<=8;i+=1){
                my_motor(LEFT_MOTOR,-i);
                my_motor(RIGHT_MOTOR,-i);
                //if(left_stall || right_stall) break;
                sleep(0.1);
            }
        }
    }else{
        if(dist>0.0){
            my_motor(LEFT_MOTOR,+2);
            my_motor(RIGHT_MOTOR,+2);
        }else{
            my_motor(LEFT_MOTOR,-2);
            my_motor(RIGHT_MOTOR,-2);
        }

    }
    beep();
    //pid=start_process(detect_stall());
    while(read_dist()<abs_dist){
        //printf("%d\n",(int)read_dist());
        //if(left_stall || right_stall) break;
    }
    //kill_process(pid);

    // Hit the brake
    my_motor(LEFT_MOTOR,-1);
    my_motor(RIGHT_MOTOR,-1);
    sleep(.05);

    my_motor(LEFT_MOTOR,0);
    my_motor(RIGHT_MOTOR,0);

    sleep(1.0);
    //beep();
    // Update world model


    if(dist >0.0)
      actual_dist=read_dist();
    else
      actual_dist=-read_dist();

    my_x+=actual_dist*cos(my_theta*MY_PI/180.0);
    my_y+=actual_dist*sin(my_theta*MY_PI/180.0);
    //printf("d:%f\n",actual_dist);

    return actual_dist;
}


int enter_goal(float dist)
{
    //min 5.72 inch
    int i,pid;
    float actual_dist,abs_dist;
    int lhat,rhat;
    int found_goal=FALSE;

    printf("drive goal %d\n",(int)dist);

    if(dist > 0.0) abs_dist=dist;
    else abs_dist=-dist;

    reset_dist();
    if(dist>0.0){
        my_motor(LEFT_MOTOR,+2);
        my_motor(RIGHT_MOTOR,+2);
    }else{
        my_motor(LEFT_MOTOR,-2);
        my_motor(RIGHT_MOTOR,-2);
    }

    beep();
    while(read_dist()<abs_dist){
        lhat=analog(LEFT_HAT);
        rhat=analog(RIGHT_HAT);
        if((lhat>190) || (rhat>190)){
            found_goal=TRUE;
            break;
        }


    }

    my_motor(LEFT_MOTOR,0);
    my_motor(RIGHT_MOTOR,0);
    sleep(1.0);
    //beep();
    // Update world model


    if(dist >0.0)
      actual_dist=read_dist();
    else
      actual_dist=-read_dist();

    my_x+=actual_dist*cos(my_theta*MY_PI/180.0);
    my_y+=actual_dist*sin(my_theta*MY_PI/180.0);
    //printf("d:%f\n",actual_dist);
    return found_goal;
}


/*
if(end_angle >= 5.5){
    end_angle -= 5.5;
}//end if cludging
*/

//#define RIGHT_TICKS_PER_DEGREE 1.25
//#define RIGHT_TICKS_PER_DEGREE 1.45 /* increase for more turn */
//#define RIGHT_TICKS_PER_DEGREE 1.375 /* clean wheels clean floor */
//#define RIGHT_TICKS_PER_DEGREE 1.156 /* dirty wheels dirty floor */
//#define RIGHT_TICKS_PER_DEGREE 1.140 /* dirty wheels dirty floor */

//float RIGHT_TICKS_PER_DEGREE;
float DEGREES_PER_TICK;

int r_old_angle_ticks;
int l_old_angle_ticks;
```

```
void reset_angle(void){
    r_old_angle_ticks=read_encoder(RIGHT_ENCODER);
    l_old_angle_ticks=read_encoder(LEFT_ENCODER);
}

float read_angle(void){
    float r_angle;
    float l_angle;
    float angle;

    r_angle=(float)(read_encoder(RIGHT_ENCODER)-
r_old_angle_ticks)*DEGREES_PER_TICK;
    l_angle=(float)(read_encoder(LEFT_ENCODER)-
l_old_angle_ticks)*DEGREES_PER_TICK;

    angle=(r_angle+l_angle)/2.0;
    //printf("%d %d\n",(read_encoder(RIGHT_ENCODER)-
old_angle_ticks),(int)angle);
    return angle;
}

float turn(float angle)
{
    // min 57.24 degrees
    int i;
    float actual_angle;
    float abs_angle;

    printf("turn %d\n",(int)angle);
    reset_angle();

    if(angle>+0.0){
        my_motor(LEFT_MOTOR,+2);
        my_motor(RIGHT_MOTOR,-2);
    }
    else
      if(angle<-0.0){
          my_motor(LEFT_MOTOR,-2);
          my_motor(RIGHT_MOTOR,+2);
      }
      else{
          my_motor(LEFT_MOTOR,0);
          my_motor(RIGHT_MOTOR,0);

      }


    //my_motor(LEFT_MOTOR,-2);
    //my_motor(RIGHT_MOTOR,+2);

    if(angle < 0.0) abs_angle=-angle;
    else abs_angle=angle;
    //printf("%d %d\n",(int)read_angle(),(int)(abs_angle));
    while(read_angle()<abs_angle){
        //while(1){
        //printf("%d %d\n",(int)read_angle(),(int)(abs_angle));
        //sleep(0.1);
    }
    my_motor(LEFT_MOTOR,0);
    my_motor(RIGHT_MOTOR,0);
    sleep(0.5);
    //beep();
    // Update world model

    if(angle>0.0){
        actual_angle=+read_angle();
    }else{
        actual_angle=-read_angle();
    }

    my_theta+=actual_angle;
    while(my_theta >360.0) my_theta-=360.0;
    while(my_theta <-360.0) my_theta+=360.0;

    return actual_angle;

}

void turn_ticks(int ticks)
{
    // min 57.24 degrees
    int i;
    int r_old_ticks,r_new_ticks,r_diff;
    int l_old_ticks,l_new_ticks,l_diff;
    int ave_diff;


    printf("turn %d\n",(int)ticks);
    r_old_ticks=read_encoder(RIGHT_ENCODER);
    l_old_ticks=read_encoder(LEFT_ENCODER);

    if(ticks>0){
        /*
        for(i=0;i<=8;i+=1){
            my_motor(LEFT_MOTOR,+i);
            my_motor(RIGHT_MOTOR,-i);
            sleep(0.1);
        }
*/
        my_motor(LEFT_MOTOR,+2);
        my_motor(RIGHT_MOTOR,-2);

    }
    else{
        my_motor(LEFT_MOTOR,0);
        my_motor(RIGHT_MOTOR,0);

    }


    ave_diff=0;

    while(ave_diff<ticks){
        printf("%d %d ave:%d\n",l_diff,r_diff,ave_diff);
        r_new_ticks=read_encoder(RIGHT_ENCODER);
        l_new_ticks=read_encoder(LEFT_ENCODER);
        r_diff=r_new_ticks-r_old_ticks;
        l_diff=l_new_ticks-l_old_ticks;

        ave_diff=(r_diff+l_diff)/2;
        //sleep(0.1);
    }
    //printf("%d %d\n",(int)read_angle(),(int)(abs_angle));
    my_motor(LEFT_MOTOR,0);
    my_motor(RIGHT_MOTOR,0);
    sleep(0.5);
    //beep();
    // Update world model



}

void cal_turn(void)
{
    int ticks, exit_flag=0;

    for(;;){
        printf("press start cal turn stop to quit\n");
        while(!start_button()){
            if(stop_button()){
                exit_flag=1;
                break;
            }
        }
        if (exit_flag) break;

        sleep(0.5);
        while(!start_button() ){
            ticks=1*200+1*(knob()-128);
            printf("%d press start\n",ticks);
            sleep(0.1);
        }


        sleep(1.0);
        turn_ticks(ticks);
        beep();
    }
    DEGREES_PER_TICK=90.0/(float)ticks;
    printf("DPT %f\n",DEGREES_PER_TICK);
}
void stage_wall()  /* The robot must be staged on the edge of a box to
ensure
                 perpendicluar allignment to drive to the next box.*/
{
    int lw,rw;
    float old_time;

    lw=0;
    rw=0;
    old_time=seconds();
    while(seconds()-old_time<5.0){
        //while(1){
        lw=digital(LEFT_WALL);
        rw=digital(RIGHT_WALL);
        printf("%d %d\n",lw,rw);

        if(lw==1 && rw==1){
            printf("done\n");
            my_motor(LEFT_MOTOR,0);
            my_motor(RIGHT_MOTOR,0);

            //reset_enco();
            break;
        }
        else
          if(lw==1 && rw==0){
              my_motor(LEFT_MOTOR,-2);
              my_motor(RIGHT_MOTOR,2);
              sleep(0.01);

          }
          else
            if(lw==0 && rw==1){
                my_motor(LEFT_MOTOR,2);
                my_motor(RIGHT_MOTOR,-2);
                sleep(0.01);

            }
            else
              if(lw==0 && rw==0){
                  my_motor(LEFT_MOTOR,1);
                  my_motor(RIGHT_MOTOR,1);
              };


    }
    my_motor(LEFT_MOTOR,0);
    my_motor(RIGHT_MOTOR,0);

}

void init_loco(void)
{
    init_encoders();
    DEGREES_PER_TICK=0.8063;
}
```

deliberation.ic

```
#use "loco.ic"
#use "music.ic"

int P_ANGLE = 0;
int P_DISTANCE = 1;

#define FALSE 0
#define TRUE 1

#define FOV 15 /* Robot's field of view (should be even divisor of 360)
*/

int NORTH_WALL = 0;
int SOUTH_WALL = 1;
int EAST_WALL = 2;
int WEST_WALL = 3;

int CUBE_START = 0; // start of cube entries in WM
int CUBE_END = 15; // end of cube entries in WM
int GOAL_START = 16; // start of goal entries in WM
int GOAL_END = 19; // end of goal entries in WM

float X_ARENA_SIZE = 96.0; //arena size in x direction (inches)
float Y_ARENA_SIZE = 144.0; //arena size in y direction (inches)

float EDGE_TOL = 6.0; //edge tolerance in inches for waypoint
validation
// i.e. within this many inches of edge == invalid waypoint

float MAX_DIST = 180.0; //maximum possible inces from robot to another
point

float RECAL_DIST = 5.5; //distance from robot touch sensors to center
of rotation

int W_UPPER = 0;
int W_LOWER = 1;
int W_RIGHT = 2;
int W_LEFT = 3;

int W_X = 0;
int W_Y = 1;
int W_VALID = 2;


float W_STANDOFF = 24.0; //distance between waypoint and target
(inches)

//current data: demonstration


float wm[21][2] = {
    {7.0,3.0},  {2.0,3.5},  {6.0,4.5},    {6.0,6.0}, //Cubes 0 - 3
      {1.0,7.0},  {1.0,9.5},  {6.5,10.0}, {7.5,10.0},//Cubes 4 - 7
      {-1.0,-1.0},{-1.0,-1.0},{-1.0,-1.0},{-1.0,-1.0},//Cubes 8 - 11
      {-1.0,-1.0},{-1.0,-1.0},{-1.0,-1.0},{-1.0,-1.0},//Cubes 12 - 15

      {1.0,1.0},  {6.0,3.0},  {2.0,11.0}, {-1.0,-1.0},//Goals 16 - 19

      {4.0,5.5}};//Robot 20

/*
// Simple Test WM
float wm[21][2] = {
    {3.0,4.0},  {-1.0,-1.0},  {-1.0,-1.0},    {-1.0,-1.0}, //Cubes 0 -
3
      {-1.0,-1.0},  {-1.0,-1.0},  {-1.0,-1.0}, {-1.0,-1.0},//Cubes 4 -
7
      {-1.0,-1.0},{-1.0,-1.0},{-1.0,-1.0},{-1.0,-1.0},//Cubes 8 - 11
      {-1.0,-1.0},{-1.0,-1.0},{-1.0,-1.0},{-1.0,-1.0},//Cubes 12 - 15

      {7.0,4.0},  {-1.0,-1.0},  {-1.0,-1.0}, {-1.0,-1.0},//Goals 16 -
19

      {1.0,1.0}};//Robot 20
*/

//global variables
float path[2][2]; //short sequence of path information used by
navigation function
// path[SECTION][ANGLE], PATH[SECTION][DISTANCE];
float waypoint[4][3]; // possible waypoints near a destination
// waypoint[W_UPPER][W_X][W_VALID];

float real_time; //used in conjuction with seconds for timing functions
float dest_x,dest_y; //actual destination of robot
float way_x,way_y; // coords of last waypoint


/* FUNCTIONS begin here, not methods (it's IC, not Java ;) */

//changes world model to inches
void convert_wm(void)
{
    int loop,inner;
    for(loop=0; loop < 21; loop++){
        for(inner = 0; inner < 2; inner++){
            wm[loop][inner] *= 12.0;
        }//end for inner
    }//end for loop
}//end convert world model

void init_wm(void)
{
    convert_wm();
    //must init my_x, my_y, and my_theta
    my_x=wm[20][0];
    my_y=wm[20][1];
    my_theta = 0.0; // robot points SOUTH initially
```

```
}//end initialize world model



// returns time in minutes from last reset
float get_time()
{
    return (seconds() - real_time)/60.0;
}//end get time



// resets value time is measured from
void reset_time()
{
    real_time = seconds();
}//end reset time



// returns the absolute value of a float
float float_abs(float n)
{
    if (n < 0.0)
      return -n;
    else
      return n;
}//end abs for float



// Returns distance from robot location to destination
float get_distance (float x, float y)
{
    float a = float_abs(x - my_x);
    float b = float_abs(y - my_y);
    float c = sqrt(a*a + b*b);
    return c;
}//end get distance



// Convert x direction to absolute angle
float x_to_angle(float x)
{
    if (x < 0.0)
      // -x dir = 180 degrees absolute heading
      return 180.0;
    else
      // +x dir = 0 degrees absolute heading
      return 0.0;
}

// Convert y direction to absolute angle
float y_to_angle(float y)
{
    if (y < 0.0)
```

```
      // -y dir = 270 degrees absolute heading
      return 270.0;
    else
      // +y dir = 90 degrees absolute heading
      return 90.0;
}


// fills the path array with a sequence of turn and drive orders.
// only provides directions from robot's current location to given
// coords (uses global knowledge of robot's final destination to
// ensure robot is pointing at destination)
void plan_path(float x, float y)
{
    // get distances we need to travel
    float delta_x = x - my_x;
    float delta_y = y - my_y;

    // travel x path first if x coord doesn't change from waypoint to
    // actual destination
    if (x == dest_x)
      {
        // travel in x direction first
        path[0][P_ANGLE] = x_to_angle(delta_x);
        path[0][P_DISTANCE] = float_abs(delta_x);
        path[1][P_ANGLE] = y_to_angle(delta_y);
        path[1][P_DISTANCE] = float_abs(delta_y);
    }
    // otherwise travel y path first
    else
      {
        // travel in y direction first
        path[0][P_ANGLE] = y_to_angle(delta_y);
        path[0][P_DISTANCE] = float_abs(delta_y);
        path[1][P_ANGLE] = x_to_angle(delta_x);
        path[1][P_DISTANCE] = float_abs(delta_x);
    }
}

// populate waypoint global array
void compute_waypoints(float x, float y)
{
    // left waypoint
    waypoint[W_LEFT][W_X] = x + W_STANDOFF;
    waypoint[W_LEFT][W_Y] = y;
    waypoint[W_LEFT][W_VALID] = (float)FALSE;

    // right waypoint
    waypoint[W_RIGHT][W_X] = x - W_STANDOFF;
    waypoint[W_RIGHT][W_Y] = y;
    waypoint[W_RIGHT][W_VALID] = (float)FALSE;

    // upper waypoint
    waypoint[W_UPPER][W_X] = x;
    waypoint[W_UPPER][W_Y] = y + W_STANDOFF;
    waypoint[W_UPPER][W_VALID] = (float)FALSE;
    // lower waypoint
    waypoint[W_LOWER][W_X] = x;
    waypoint[W_LOWER][W_Y] = y - W_STANDOFF;
    waypoint[W_LOWER][W_VALID] = (float)FALSE;
}

int validate_waypoints()
{
    int i,valid_waypoint_exists = FALSE;

    // check each waypoint for "in-bounds"
    for (i=0;i < 4;i++)
      {
        if ((waypoint[i][W_X] < (X_ARENA_SIZE - EDGE_TOL)) &&
            (waypoint[i][W_X] > (0.0 + EDGE_TOL)) &&
            (waypoint[i][W_Y] < (Y_ARENA_SIZE - EDGE_TOL)) &&
            (waypoint[i][W_Y] > (0.0 + EDGE_TOL)))
          {
            waypoint[i][W_VALID] = (float)TRUE;

            // at least one waypoint is valid
            valid_waypoint_exists = TRUE;
        }
    }

    return valid_waypoint_exists;
}

// executes movement commands stored in the path array
void navigate()
{
    int i;
    float angle;

    // Convert absolute angles to relative angles and issue commands
    for (i=0;i < 2;i++)
      {
        angle = path[i][P_ANGLE] - my_theta;

        // optimize angle
        if (angle > 180.0)
          {
            angle -= 360.0;
        }
        else if (angle < -180.0)
            {
              angle += 360.0;
          }
          else
            {
              angle = angle;
          }

        if (angle != 0.0)
          turn(angle);
        if (path[i][P_DISTANCE] != 0.0)
          drive(path[i][P_DISTANCE]);
    }
}

int get_nearest_waypoint()
{
    int i,nearest = -1;
    float tmp,smallest = MAX_DIST;

    for (i=0;i < 4;i++)
      {
        if (waypoint[i][W_VALID] == (float)TRUE)
          {
            tmp = get_distance(waypoint[i][W_X],waypoint[i][W_Y]);
            if (tmp < smallest)
              {
                smallest = tmp;
                nearest = i;
              }
          }
    }

    return nearest;
}

int get_nearest(int start, int end)
{
    int i,nearest = -1;
    float tmp,smallest = MAX_DIST;

    for (i=start;i <= end;i++)
      {
        if (wm[i][0] > -1.0)
          {
            tmp = get_distance(wm[i][0],wm[i][1]);
            if (tmp < smallest)
              {
                smallest = tmp;
                nearest = i;
              }
          }
    }

    return nearest;
}

int get_nearest_cube()
{
    return get_nearest(CUBE_START,CUBE_END);
}

int get_nearest_goal()
{
    return get_nearest(GOAL_START,GOAL_END);
}

void find_cube(void)
{
    /*    int i;

 // turn 360 degress in FOV steps
 for (i=0;i < 360/FOV;i++)
 {
 turn((float)FOV);
 findCube();
 // NOT FINISHED
 }*/ //end for i
}

void remove_from_WM(int index)
{
    wm[index][0] = -1.0;
    wm[index][1] = -1.0;
}

int go_to_waypoint()
{
    int nearest_waypoint;

    compute_waypoints(dest_x,dest_y);

    // check validity of waypoints
    if (validate_waypoints())
      {
        // valid waypoint exists, get nearest waypoint
        nearest_waypoint = get_nearest_waypoint();

        // plan path to nearest waypoint
        plan_path(waypoint[nearest_waypoint][W_X],
                  waypoint[nearest_waypoint][W_Y]);

        // execute path
        navigate();

        return TRUE;
    }
    else
      return FALSE;
}

int validate_destination()
{
    if ((dest_x > (X_ARENA_SIZE - EDGE_TOL)) ||
        (dest_x < (0.0 + EDGE_TOL)) ||
        (dest_y > (Y_ARENA_SIZE - EDGE_TOL)) ||
        (dest_y < (0.0 + EDGE_TOL)))
      {
        return FALSE;
    }
    return TRUE;
}

int go_to_destination()
{
    // check validity of destination
    if (validate_destination())
      {
        // destination is valid, plan path to get there
        plan_path(dest_x,dest_y);

        // execute path
        navigate();

        return TRUE;
    }
    else
      return FALSE;
}

int WM_has_cubes()
{
    int i;

    for (i=CUBE_START;i <= CUBE_END;i++)
      {
        if (wm[i][W_X] > -1.0)
          return TRUE;
    }

    return FALSE;
}

int cube_exists()
{
    // check camera confidence for orange
    return TRUE;
}

int acquire_cube()
{
    // behavior to actually "swallow" cube
    printf("\nGot Cube!!!\n");

    return TRUE;
}

void play_tune()
{
    //Plays Tune
    charge();
    printf("Press Start\n");
    while(!start_button());
    printf("Played tune...\n");
}

void initiate_wall_recal(int wall)
{
    // bump wall & turn in direction of bumped sensor
    // until both touch at same time (or within short time?)
    // then reset my_location
    //printf("*bump* *bump*--*bump*...*bump*...*BING!*\n");
    // Uncomment me
    stage_wall();

    // Assume we are now touching the wall in the x or y direction
    if (wall == NORTH_WALL)
      my_x = RECAL_DIST;
    if (wall == SOUTH_WALL)
      my_x = X_ARENA_SIZE - RECAL_DIST;
    if (wall == EAST_WALL)
      my_y = RECAL_DIST;
    if (wall == WEST_WALL)
      my_y = Y_ARENA_SIZE - RECAL_DIST;

    printf("Recalibrated!\n");
}


void wall_recalibrate_x()
{
    int which_wall;

    // decide which wall is closer
    if (my_x < 48.0)
      {
        // North wall is closer
        dest_x = EDGE_TOL + 1.0;
        which_wall = NORTH_WALL;
    }
    else
      {
        // South wall is closer
        dest_x = X_ARENA_SIZE - EDGE_TOL - 1.0;
        which_wall = SOUTH_WALL;
    }

    dest_y = my_y;
    go_to_destination();
    initiate_wall_recal(which_wall);

    // Backup an inch
    drive(-1.0);
}

void wall_recalibrate_y()
{
    int which_wall;

    // decide which wall is closer
    if (my_y < 72.0)
      {
        // East wall is closer
        dest_y = EDGE_TOL + 1.0;
        which_wall = EAST_WALL;
    }
    else
      {
        // West wall is closer
        dest_y = Y_ARENA_SIZE - EDGE_TOL - 1.0;
        which_wall = WEST_WALL;
    }

    dest_x = my_x;
    go_to_destination();
    initiate_wall_recal(which_wall);

    // Backup an inch
    drive(-1.0);
}

void recalibrate()
{
    // recalibrate robot position
    wall_recalibrate_x();
    wall_recalibrate_y();
}
```

mycam.ic

```
//#use "cmucamlib.ic"
//#use "hb_test.ic"

/*
#define CUBE_R 237
#define CUBE_G 56
#define CUBE_B 18
#define CUBE_TOL 30
*/

// Hougen orange YUV, Floor WB
#define CUBE_R 229
#define CUBE_G 74
#define CUBE_B 16
#define CUBE_TOL 30


int cube_R_min,cube_R_max;
int cube_G_min,cube_G_max;
int cube_B_min,cube_B_max;

void my_init_cam(void)
{
    init_camera(); // initialize the camera in YUV mode
    clamp_camera_yuv(); //clamp camera white balance in YUV mode
    init_cube_values();
    init_blue_values();
}

int fix_color_val (int val)
{
    if(val<0) val=0;
    else if(val>255) val=255;

      return val;
}

void init_cube_values(void)
{
    cube_R_min=fix_color_val(CUBE_R - CUBE_TOL);
    cube_R_max=fix_color_val(CUBE_R + CUBE_TOL);

    cube_G_min=fix_color_val(CUBE_G - CUBE_TOL);
    cube_G_max=fix_color_val(CUBE_G + CUBE_TOL);

    cube_B_min=fix_color_val(CUBE_B - CUBE_TOL);
    cube_B_max=fix_color_val(CUBE_B + CUBE_TOL);

    //printf("%d %d\n",cube_B_min,cube_B_max);
}

int cam_find_cube()
```

```
{
    if
(trackRaw(cube_R_min,cube_R_max,cube_G_min,cube_G_max,cube_B_min,cube_B
_max) > 0) {
        return track_confidence;
    } else return 0;
}

void cam_track_cube(void)
{
    int i;
    int blah_flag;

    blah_flag=0;
    //for(;;){
    for(i=0;i<5;i++){
        if (cam_find_cube() > 150)
          {
            blah_flag=1;
            printf("X:%d Y:%d S:%d
A:%d\n",track_x,track_y,track_size,track_area);

            if (track_x < -3){
                turn(0.5*(float)track_x);
            }
            else
              if (track_x > 3){
                  turn(0.5*(float)track_x);
              }
              else
                drive(12.0);

        }
        else
          {
            printf("Searching...\n");
            if(blah_flag == 1){
                break;
                printf("cubee done\n");
            }
        }
    }
}
/*
#define BLUE_R 96
#define BLUE_G 82
#define BLUE_B 85
#define BLUE_TOL 20
*/
/*
#define BLUE_R 211
#define BLUE_G 48
#define BLUE_B 20
#define BLUE_TOL 30
*/
/*
// These values don't seem to work
```

```
#define BLUE_R 38
#define BLUE_G 67
#define BLUE_B 179
#define BLUE_TOL 30
*/

/*
#define BLUE_R 74
#define BLUE_G 58
#define BLUE_B 148
#define BLUE_TOL 30
*/

// Hougen blue (neon) YUV Floor WB
#define BLUE_R 74
#define BLUE_G 58
#define BLUE_B 148
#define BLUE_TOL 30


int blue_R_min,blue_R_max;
int blue_G_min,blue_G_max;
int blue_B_min,blue_B_max;

void init_blue_values(void)
{
    blue_R_min=fix_color_val(BLUE_R - BLUE_TOL);
    blue_R_max=fix_color_val(BLUE_R + BLUE_TOL);

    blue_G_min=fix_color_val(BLUE_G - BLUE_TOL);
    blue_G_max=fix_color_val(BLUE_G + BLUE_TOL);

    blue_B_min=fix_color_val(BLUE_B - BLUE_TOL);
    blue_B_max=fix_color_val(BLUE_B + BLUE_TOL);

    //printf("%d %d\n",cube_B_min,cube_B_max);
}

int cam_find_blue()
{
    if
(trackRaw(blue_R_min,blue_R_max,blue_G_min,blue_G_max,blue_B_min,blue_B
_max) > 0) {
        return track_confidence;
    } else return 0;
}

#define LEFT_MOTOR 1
#define RIGHT_MOTOR 2

void cam_track_blue(void)
{
    int c;
    while(1){

        if (cam_find_blue() > 150)
          {
            //printf("C:%d X:%d Y:%d S:%d
A:%d\n",track_confidence,track_x,track_y,track_size,track_area);
            printf("C:%d X:%d \n",track_confidence,track_x);

            /*
            if (track_x < -4){
                turn(0.2*(float)track_x);
                //my_motor(LEFT_MOTOR,-2);
                //my_motor(RIGHT_MOTOR,+2);

            }
            else
              if (track_x > 4){
                  turn(0.2*(float)track_x);
                  //my_motor(LEFT_MOTOR,+2);
                  //my_motor(RIGHT_MOTOR,-2);

              }else{
                  //drive(12.0);
                  //my_motor(LEFT_MOTOR,0);
                  //my_motor(RIGHT_MOTOR,0);
              }
  */

            my_motor(LEFT_MOTOR,+(int)(0.1*(float)track_x));
            my_motor(RIGHT_MOTOR,-(int)(0.1*(float)track_x));

        }
        else
          {
            my_motor(LEFT_MOTOR,0);
            my_motor(RIGHT_MOTOR,0);

            printf("Searching...\n");
        }


        //cam_find_blue();
        //printf("C:%d X:%d Y:%d S:%d
A:%d\n",track_confidence,track_x,track_y,track_size,track_area);

    }
}




/*
void main()
{
    #if 0
    init_camera(); // initialize the camera in YUV mode
    clamp_camera_yuv(); //clamp camera white balance in YUV mode
    #endif
    init_encoders();

    printf("hello\n");
    while(!start_button()){

//printf("%f\n",RIGHT_TICKS_PER_DEGREE=1.156+0.25*(((float)knob()/256.0
)-0.5));

//printf("%f\n",RIGHT_TICKS_PER_DEGREE=1.0+0.5*2.0*(((float)knob()/256.
0)-0.5));
        RIGHT_TICKS_PER_DEGREE=1.36;
        sleep(0.1);
    }

    init_cube_values();
    #if 0
    while(1)
      {
        if (trackCube() > 50)
          {
            printf("X:%d Y:%d S:%d
A:%d\n",track_x,track_y,track_size,track_area);

            if (track_x < -5){
                turn(-2.0*(float)track_x);
            }
            else
              if (track_x > 5){
                  turn(+2.0*(float)track_x);
              }
              else
                ao();
            //turn(0.0);
        }
        else
          {
            printf("Searching...\n");
        }
    }
    #endif
    turn(-40.0);
    sleep(1.0);

}
*/
```