The Royal Institute of Technology
CSC

Robotics and Autonomous Systems
2D1426

1337!!

The Robot

Behrang Masoumi 761226 bema02@nada.kth.se
Daniel Norberg 820421 dano02@nada.kth.se
Erik Hartwig 810304 erha02@nada.kth.se
Mathilde Nord 820423 mathilde@nada.kth.se

Abstract

In the course 2D1426, Robotics and Autonomous systems at the Royal Institute of Technology (KTH) the objective was to build a robot capable of playing a variant of soccer. In this report we explain how we created the most simple robot that qualified for the competition.

Explained herein are details such as the drive system we used to control the robot, the Bezier trajectory schema employed to guide the robot and the vision system that we did not implement in the end for the competition.

Index

1 Introduction

This is a project report for the course 2D1426, Robotics and Autonomous Systems, taken by the authors in the spring of 2006 at the Royal Institute of Technology. The project in this course was to build an autonomous mobile robot capable of playing robot football. The specific rules for this robot football are similar to those of the international *Robot Cup* soccer rules, but adapted to local conditions.

The most central parts of the robot were handed out to the participants, to assure equal terms of competition and the possibility of actually finishing the project. Those parts were a computer board outfitted with a Motorola 68332 microcontroller, a CMOS camera, two wheels and motors with encoders.

This report aims to shed some light on how we implemented the robot and what our thoughts were. We should mention that in the end we did not use the camera, we qualified for the competition anyway and our robot was able to make several goals on the field unopposed.

2 The Plan

Our plan was simple; to win the competition. Due to time issues we did not have the opportunity to invest as much time as we would have liked in this project and plan B took form. This was traditional engineering problem solving, qualify for the competition.

From the start we decided that simplicity would give us the best results. We approached every problem with a simplistic straight forward way of thinking. This paradigm is in our opinion the best way to approach all problems in robotics, how ever complex they may look.

We decided to implement movement with Bezier trajectories and connect this to vision information. In the end we did not use the vision system and modelled the movement in such a way that it would be guaranteed to score at least one goal. This was the requirement to qualify to the competition.

2.1 Plan B

What was plan B? It was simply to make it to the competition. We knew that the ball would be placed somewhere on the centre line but not at the centre point. So if we made a trajectory that swept the entire midline and then made it to the other goal, we would qualify. We implemented such a trajectory and hade good enough precision in the model to make three goals before the slight deviations would set the robot too much out of the desired trajectory.

There were however two major weaknesses to this model for scoring several goals. The first was the velocity; to ensure good precision we lowered the speed of the robot. This meant that the robot did not have enough time to score more than one goal during the required two minutes. The second weakness was that the robot could hit the ball on its way back to its own goal if the ball was placed near the centre point after the first goal. If this occurred the ball would be lost forever and the robot would be unable to score any further goals.

The trajectory of the robot was as follows; start at the own goal, make a nice curvature to left side of the centre line, rotate on place, and sweep the centre line to its right side. Rotate towards the opponet´s goal and drive in the same curvature as before to it. Then drive backwards to the own goal. The second run the robot would drive to the other side of the centre line (Right side) and then alter again to the left side. In this way some of the deviations of each run would exclude each other.

This may not be the fanciest solution to the problem, but it worked!

3 Physical design

We were required to build a robot that it would fit inside a cylinder with a diameter of 18 cm. We decided to make the point of gravity as low as possible and to place the wheels in the front of the most part of the robot. This would place the ball between the wheels and give us good control of the ball in curvature movements. The camera was placed on a pole between the wheels, also in the front of the greater part of the robot.

Our design proved to be a very simplistic but robust construction. We also decided to elevate the camera somewhat to give us the opportunity for good distance measurements. The battery was placed right behind the wheels and gave the robot a very good stability with its weight.

4 Drive system

4.1 Goals

The use of the robot as a soccer player puts special requirements on the drive system as it is the principal method for interaction with the environment.

4.1.1 Speed

In order to be able to carry out any plan, such as pushing the ball into the opponents goal, driving must be done with sufficient speed that the plan stays valid throughout. I.e. the ball is still there when the robot arrives. If the drive system is too slow, the ball might have been pushed away by the opponent when the robot arrives at its designated position.

4.1.2 Accuracy

Speed is not enough, the robot must also consistently also be able to arrive at a designated position with some accuracy, otherwise any plan becomes impossible to carry out. Also, the drive system must be able to consistently provide good enough position estimates in order for new plans to be made.

4.1.3 Path Planning

As the environment contains several geometrical constraints such as walls, the opponent, the positioning of the goals, etc., it is of utmost importance that intelligent driving plans that satisfy named constraints can be produced. I.e. if the current task is pushing the ball into the goal, it is often a good idea to position the robot by the ball, facing both the ball and the goal.

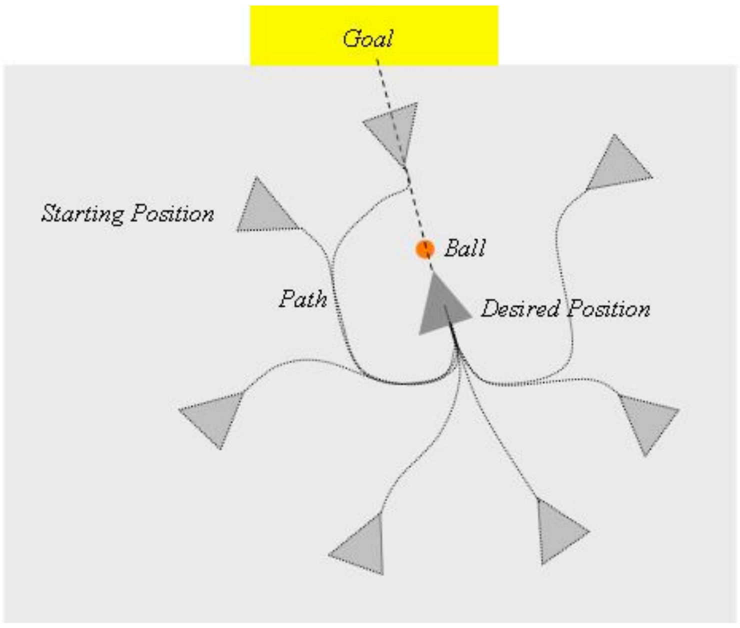

Figure *1*: Ex. Advantageous path planning and positioning of the robot

4.1.4 Simplicity & Performance

In order to be usable, a tentative drive system must be simple enough that it can be implemented in C as well as have modest CPU demands. This rules out many of the fancier driving algorithms.

4.1.5 Abstraction

The implementation of the drive system must provide a good abstraction, that it may be efficiently used by a higher-level decision system.

4.2 Theory & Implementation

4.2.1 Differential Drive

The robot makes use of a two-wheel differential drive system. This means that the two wheels of the robot are used to provide both locomotion and steering by rotating the wheels at different speeds as required. This enables many driving patterns to be performed, such as driving straight, following curves and turning on the spot.

4.2.2 Odometry

In order to position itself, the robot uses odometry and quadrature encoders built into the two motors. These continually count the number of revolutions the wheels have made. Using these values and data about the positioning and size of the wheels, the position of the robot can be calculated. When this is combined with the knowledge of the starting position, this allows for complete dead reckoning.

4.2.3 VW-System

The RoBiOs contains a drive system that provides odometry based dead reckoning positioning as well as elementary drive control. This system is called the VW-Drive system, taking its name from its two driving parameters, velocity, v, and turning rate, w. Using the VW-Drive system allows us to concentrate on higher level path planning.

4.2.4 Bezier Path Planning

In order to drive efficiently and be able to push the ball ahead of the robot, it is useful to be able to drive along smooth paths. One of the simplest, well known and useful mathematical models of smooth trajectories are Cubic Bezier curves.

Given a starting point, ending point and headings at the two, Cubic Bezier curves can provide a smooth path between the two. Additionally, the parameters of a Cubic Bezier curve can be chosen in a manner that makes them stable, i.e. if the path is recalculated at any point on the original path, the new path is identical to the remaining part of the original. This simplifies the low-level path planning as new driving parameters, i.e. velocity and turn rate, can be calculated at all times using actual position data.

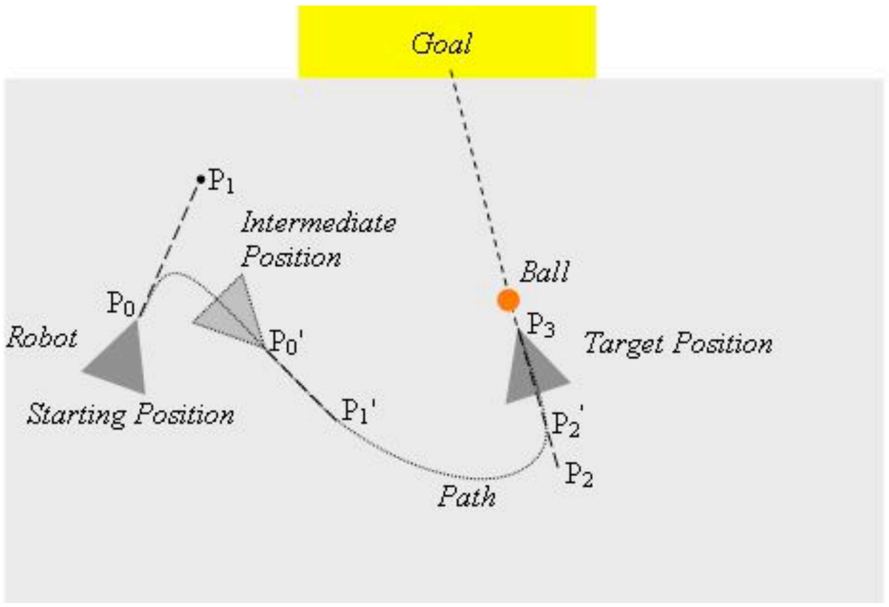

Figure *2*: **Stable Bezier low-level path planning. Px are control points of the Bezier curve. Px' indicates the new positions of the control points at the intermediate position.**

A Bezier path is defined by four vectors, $P_{0..3}$, where P_0 is the starting point, P_1 controls the initial velocity and heading, P_2 controls the target velocity and heading, and P_3 is the target position.

When calculating new driving parameters, P_0 is the current position of the robot and P_1 is P_0 offset in the direction of the robot. P_3 is the target position and P_2 is P_3 offset in the reverse target direction. By setting the magnitudes of the offsets of P_1 and P_2 to an equal amount, *d*, the stability of the Bezier Curve is ensured.

$$d = \frac{2}{3}\sqrt{2}\left(\sqrt{2}-1\right)\left\|\mathbf{P}_3 - \mathbf{P}_0\right\|$$

This magnitude can be used to calculate the relation between the velocity and the turn rate needed at this instant in order to effectively follow a predefined Bezier path. The derivation of the formula is not covered here, its result is merely stated below. Complete derivation of mathematics can be found in the paper *Vision-based Robot Formations with Bezier Trajectories* [7].

$$\omega = v\frac{2(y - d\sin\theta)}{3d^2}$$

θ = {Angle between robot direction and target direction}

Using this formula and a predefined suitable velocity the turn rate can be calculated and thus all the parameters needed by the VW-Drive are available. Following a Bezier path is now merely a matter of constantly recalculating the turn rate based on the current position, and perhaps a modified velocity, and updating the VW-Drive parameters.

The precision of the following of path is mainly limited by the precision of the dead-reckoning positioning system. This becomes the main factor in deciding the velocity of the robot as greater velocity drastically reduces the accuracy of the quadrature odometry dead-reckoning system.

4.3 Results

The drive system worked very well, providing the necessary accuracy to predictably and consistently make several goals blindfolded. Unfortunately, it was necessary to drive very slowly in order to obtain the necessary positioning accuracy, which caused the robot to only manage to make one goal in the seeding.

In order to improve positioning accuracy, attempts were made to calibrate the quadrature odometers using the ticks-per-meter values in the HDT. Suitable values were produced by repeatedly pushing the robot a predefined distance, in a straight line, and reading the odometer values. The mean of these values were then used in the HDT. Unfortunately these values proved to be unsuitable – the VW-Drive refused to drive accurately with calibrated odometer values, probably due to the implementation of the pi-controllers of the VW-Drive. Therefore the measured values were abandoned in favour of empiric values that were produced by simply testing which values gave good positioning accuracy. The measured odometer values were used only as a hint as to a suitable difference between the left and right odometer values in the HDT.

The motors of the robot are clearly much too high geared. The full speed of the motors is never used and the motors become too weak when run at low speed. If an external gearbox were used to gear down the motors, both driving control and dead-reckoning accuracy would improve as the motors would become stronger and the quadrature odometers would produce more ticks per meter.

4.4 Source Code

See appendix 1.

5 Vision

Even though the final implementation of the control software for the robot did not utilize the camera several computer vision solutions were implemented and tested. The purpose of the vision system is to process raw imaged data and output coordinates for the objects in the field of view. Even though our implementation consisted of several algorithms the process can divided into two major parts.

The first part consists of extracting features form the raw image by sub sampling the picture and classifying the pixels. The second part is to triangulate the actual object position in the real world from the extracted features.

5.1 Classification

In this step speed is crucial. The goal is to get as many frames processed correctly as fast as possible. To do this we have to scan through enough pixels in the image to find all objects in the image that are of interest. To select what pixels to scan we tried some different algorithms. The simplest one scanned all pixels, it was accurate but slow. We then tried to skip every other row and column in the image and the increase in speed outweighed the loss in performance. We also tried a dual scan algorithm which initially did a very coarse scan of the image and then a finer scan in areas where the initial scan had indicated that there might be a feature of interest. Neither the accuracy nor the speed was satisfactory for the algorithm.

Since our camera was tilted 45 degrees towards the plane objects in the top of the image became smaller and objects in the bottom became bigger. The algorithm we finally decided on utilized this fact and had different coarseness depending on the vertical position in the image. This gave a significant speedup with almost no performance regression.

To classify what object an image pixel represented we used a HUE lookup table. The reason for this was speed and simplicity. To reduce the lookup table size only the four most significant bits for each colour was used. This dramatically reduced the lookup table size with very little impact on classification result. The reason we decided to use HUE to represent colours was to be able to specify a centre colour and then a tolerance radius to initiate the lookup table. We could now take pictures in the correct light conditions of the objects we wanted to classify and then extract the mean colour and variance. With these parameters constructing and tweaking the lookup table was easy.

5.2 Triangulation

To get 3D coordinates from a perspective projection image we used the following correspondence:

$$p = \lambda RK(I \mid P_0)P$$

Where p is the image coordinates in homogenous form:

$$p = \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

R the rotation matrix for rotation around the X-axis:

$$R = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\omega) & \sin(\omega) \\ 0 & -\sin(\omega) & \cos(\omega) \end{pmatrix}$$

K the camera matrix specifying internal camera parameters:

$$K = \begin{pmatrix} \sigma & \gamma & x_0 \\ 0 & \sigma & y_0 \\ 0 & 0 & 1 \end{pmatrix}$$

$(I \mid P_0)$ is the identity matrix with the camera offset vector appended:

$$(I \mid P_0) = \begin{pmatrix} 1 & 0 & 0 & X_0 \\ 0 & 1 & 0 & Y_0 \\ 0 & 0 & 1 & Z_0 \end{pmatrix}$$

P the world coordinates in homogenous form:

$$P = \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

λ is an arbitrary scale factor.

Since many of these parameters are constant the expressions can be simplified. First we define our centre for the system as the point on the playing field where a line, that passes through the camera centre and is perpendicular to the field, intersects the field. This means that our camera only is translated in the Y direction.

If we extract X and Z from the equations we get a new equation system:

$$\begin{cases} X = \dfrac{(Y_0 - Y)x\sigma\sin(\omega) - Z\,x\,y_0\sin(\omega) + Z\,x\cos(\omega) + (Y_0 - Y)\gamma - Z\,x_0 + \gamma Y_0}{\sigma} \\ Z = \dfrac{(Y_0 - Y)(\sigma\cos(\omega) + y\sigma\sin(\omega))}{\cos(\omega)\,y_0 + \sin(\omega) + y(\sin(\omega)\,y_0 + \cos(\omega))} \end{cases}$$

As we can see this equation system has three unknown (X, Y and Z) and every point in the image only satisfies two. So we would need at least two different 2D points for each 3D point, this is only feasible using two images or adding more constraints to the equations. Since stereo vision not was an option due to lack of equipment and computational power we added more constraints by removing one dimension of freedom from the objects in 3D space. Since all objects we are trying to triangulate moves parallel to the playing field we can instead calculate positions on a plane. We now have the following equation system:

$$\begin{cases} X = \dfrac{Y_0 x \sigma \sin(\omega) - Z x y_0 \sin(\omega) + Z x \cos(\omega) + Y_0 \gamma - Z x_0 + \gamma Y_0}{\sigma} \\ Z = \dfrac{Y_0 (\sigma \cos(\omega) + y \sigma \sin(\omega))}{\cos(\omega) y_0 + \sin(\omega) + y(\sin(\omega) y_0 + \cos(\omega))} \end{cases}$$

To get an accurate position we need to know the distance of the perpendicular offset from the plane to the point we are trying to position. This poses a problem, even if we knew the exact shape and dimensions of an object, it is still the offset to the plane based on the image point we need to know. So somehow we have to be able to know what point on the object corresponds to the point in the image.

For the ball this is easy since it is rotationally invariant and we know the offset. The centre of the ball object in the image can be used as the centre in the world. The rest of the objects are harder but can still be triangulated with some estimations. What we tried was to first find the horizontal centre of the object in the image and the lowest point on that line.

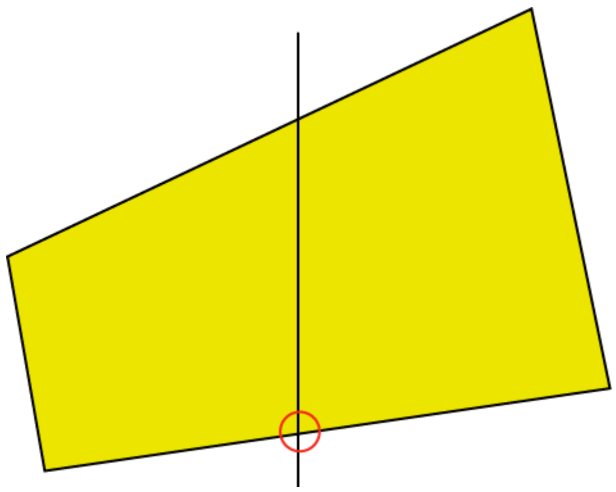

Figure *3*: The estimated center of an object with the centre line and intersection displayed.

Then based on the two assumptions that the centre of the object in the image is centre in the world and that the object is connected to the field we can then position it. The object that causes the most difficulties to triangulate is the opponent. The purple coloured sheet is not connected to field and we do not know what height it is at. Therefore we only estimated the angle to the opponent but not the distance.

5.3 Conclusions

The reason the vision system never made it into the final control system was primarily due to the problems with the classification. The reason there was problems with the classification had two causes. The first was the auto brightness on the camera and the other the glossy material used to create the ball. These two manifested themselves in the fact that large areas of the ball, if illuminated, were either yellow like one of the goals or while like the walls.

6 Conclusions

We managed to build a robot that was able to perform several goals independently without using the camera, though admittedly, depending on a loop-hole in the game rules specifying predictable placement of the ball.

Our method alleviated our lack of working vision and planning algorithms which was mainly due to shortage of time. Still, we take some pride in having procured and implemented Plan B in a matter of hours.

The reasons for our shortage of time were twofold; Firstly, we started building our robot too late. Secondly, we aimed too high in the implementation of the control software.

A lesson we learned was that instead of building really fancy vision algorithms and sophisticated trajectory control and planning algorithms we should have stuck to the basics. As a result of our priorities, a lot of good work and code went to waste and the resulting robot became "below basic".

In the competition our robot did not fare well. It qualifies but lost every match against other robots, naturally, as our robot became helpless when the ball was moved or it collided with its opponent.

7 References

1. M. Bratt. *Project lecture notes A & B*. For course 2D1426 Robotics and Autonomous Systems, KTH, 2006.
2. T. Bräunl. *EyeBot - Online documentation*. http://robotics.ee.uwa.edu.au/eyebot/.
3. T. Bräunl. *Embedded Robotics - Mobile Robot Design and Applications with Embedded Systems*. Springer Verlag, 2003. ISBN: 3-540-03436-6.
4. H. I. Christensen. *Lecture notes 1–6*. For course 2D1426 Robotics and Autonomous Systems, KTH, 2006.
5. R. Siegwart and I.R. Nourbakhsh. *Introduction to Autonomous Mobile Robots*. MIT Press, 2004. ISBN: 0-262-19502-X.
6. http://www.nada.kth.se/kurser/kth/2D1426/aktuellt.html
7. S. Chiem, E. Cervera. *Vision-based Robot Formations with Bézier Trajectories*, http://www.robot.uji.es/people/ecervera/papers/ias04.pdf

Appendix

Appendix 1

drive.h

```
#ifndef DRIVE_H_
#define DRIVE_H_

#include <eyebot.h>

/* Use magic bezier path? */
#define USE_MAGIC 1

/* Parameter Defaults */

#define DEFAULT_ARRIVAL_RADIUS 0.03 /* 3cm */
#define DEFAULT_SPEED_LIMIT 0.2
#define DEFAULT_ANGULAR_SPEED_LIMIT 0.1

/* Destination Flags */
#define DST_GOAL        1
#define DST_BALL        2
#define DST_ARBITRARY   3

/* Path Types */
#define DST_STRAIGHT    0x10000
#define DST_BEZIER      0x20000

/* Specifies that the drive should stop on destination arrival */
#define DST_ARRIVAL_STOP        0x1000
#define DST_ARRIVAL_DECELERATE  0x2000

#define DST_REVERSE     0x100000

#define DEFAULT_DECELERATE_DESTINATION_RADIUS 0.5f
#define DEFAULT_DECELERATE_MIN_SPEED_MULTIPLIER 0.2f

/* Usage:
   1. call initializeDrive() once.
   2. call setDestination(), then
   3. call updateDriveSpeed() until remainingDriveDistance() is
satisfactory
*/

void driveStop();
void initializeDrive();
void setDriveDestination(PositionType dst, int destinationFlags);
void updateDriveSpeed();
```

```
float remainingDriveDistance();
int driveHasArrived();
void setDrivePosition(PositionType pos);
void driveUntilArrival(float radius);
void driveAlign(float direction);

void setDriveSpeed(SpeedType speed);
SpeedType getDriveSpeed(void);
PositionType getDrivePosition();

void setDriveDeceleration(float destinationRadius, float
minSpeedMultiplier);
SpeedType getLimitedDriveSpeed(float sl);
void setDriveAngularSpeedLimit(float w);
void setDriveSpeedLimit(float v);
int getDriveVwHandle();
void setDriveArrivalPrecision(float radius);

#endif /*DRIVE_H_*/
```

drive.c

```
#include "drive.h"
#include "vmath.h"

#include "eyebot.h"

typedef struct
{
    Point2D ball, robotControl, ballEntryRobotControl, ballEntry,
ballEntryBallControl, ballControl;
} MagicBezierPath;

static float angularSpeedLimit;
static float speedLimit;
static float arrivalRadius;
static PositionType destination;
static int destinationType;
static int vw;
static float decelerationDestinationRadius;
static float decelerationMinSpeedMultiplier;

void getMagicBezierPath(
    Point2D ball, Point2D robot, Point2D goal, Point2D
robotDirection,
    Point2D *robotControl, Point2D *ballEntryRobotControl,
    Point2D *ballEntry, Point2D *ballEntryBallControl, Point2D
*ballControl)
{
    float magic_number = 0.390524292f; /* 2*(2-sqrt(2))/3 */
    float robotBallGoalAngle = angle(direction(robot, ball),
direction(ball, goal));

    Point2D goalBall = direction(goal, ball);

    Point2D ballEntryOrigin = translate(ball,scale(goalBall,
         normalizedSigmoid((0.5f +
normalizedSigmoid(fabs(robotBallGoalAngle) / M_PI)) *
sigmoid(distance(ball, robot), 0.2f, 0.4f)) * 0.1f));
    Point2D ballEntryDirection = rotate(goalBall, -
robotBallGoalAngle / 2);
    Point2D ballEntryOffset = scale(ballEntryDirection,
        normalizedSigmoid((0.5f +
normalizedSigmoid(fabs(robotBallGoalAngle) / M_PI)) *
sigmoid(distance(ball, robot), 0.2f, 0.4f)) * 0.1f);

    *ballEntry = translate(ballEntryOrigin, ballEntryOffset);

    *ballEntryRobotControl = translate(*ballEntry,
scale(ballEntryDirection, magic_number * distance(*ballEntry,
robot)));
    *ballEntryBallControl = translate(*ballEntry,
scale(ballEntryDirection, - magic_number * distance(*ballEntry,
ball)));

    *ballControl = translate(ball, scale(direction(goal, ball),
magic_number * distance(*ballEntry, ball)));
    *robotControl = translate(robot, scale(direction(robot,
*ballEntry), magic_number * distance(robot, *ballEntry)));
}

SpeedType getBezierSpeeds(PositionType origin, PositionType
destination)
{
    const float magic_number = 0.390524292; /* 2*(2-sqrt(2))/3 */

    SpeedType speed;

    float direction;
    float dx;
    float dy;
    float dt;
    float l;
    float theta;
    float d;
    float k;

    direction = (destinationType & DST_REVERSE ? -1.0f : 1.0f);
    dx = destination.x - origin.x;
    dy = destination.y - origin.y;
    dt = destination.phi - origin.phi;
    l = distancept(origin, destination);
    theta = atan2f(dy, dx) - origin.phi;

    if (dt > M_PI) dt -= 2*M_PI;
    if (dt <-M_PI) dt += 2*M_PI;

    if (theta > M_PI) theta -= 2*M_PI;
    if (theta <-M_PI) theta += 2*M_PI;

    d = direction * magic_number * l;
    k = (l*sinf(theta) - d * sinf(dt))/(d*d);
    speed.v = direction * speedLimit;
    speed.w = k * speed.v;


    if(fabs(speed.w) > angularSpeedLimit)
    {
        /* Rotate on the spot */
        speed.w = angularSpeedLimit * (speed.w > 0 ? 1 : -1);
        speed.v = 0;
    }
    return speed;
}

void driveUntilArrival(float radius)
{
    PositionType pos;
    float oldRadius = arrivalRadius;
    setDriveArrivalPrecision(radius);

    while(remainingDriveDistance() > radius)
    {
        pos = getDrivePosition();
        updateDriveSpeed();
    }

    updateDriveSpeed();
    arrivalRadius = oldRadius;
}

void driveAlign(float direction)
{
    PositionType pos;
    float theta;
    VWSetSpeed(vw, 0, 0);
    pos = getDrivePosition();
    theta = direction - pos.phi;
    if (theta > M_PI) theta -= 2*M_PI;
    if (theta <-M_PI) theta += 2*M_PI;
    VWSetSpeed(vw, 0, angularSpeedLimit * (theta > 0 ? 1 : -1));
    do
    {
        pos = getDrivePosition();
        theta = direction - pos.phi;
        if (theta > M_PI) theta -= 2*M_PI;
        if (theta <-M_PI) theta += 2*M_PI;
        LCDPrintf("t=%2.3f\n", theta);
        LCDPrintf("phi=%2.3f\n", pos.phi);

    } while(fabs(theta) > 0.1f);
    VWSetSpeed(vw, 0, 0);
//  VWDriveTurn(vw, theta, angularSpeedLimit);
//  VWDriveWait(vw);

}

SpeedType getStraightSpeeds(PositionType origin, PositionType
destination)
{
    SpeedType speed;

    float dx = destination.x - origin.x;
    float dy = destination.y - origin.y;
    float theta = atan2f(dy, dx) - origin.phi;

    if (theta > M_PI) theta -= 2*M_PI;
    if (theta <-M_PI) theta += 2*M_PI;

    if(fabs(theta) > M_PI_2 / 10)
    {
        speed.w = angularSpeedLimit * ((theta > 0) ? 1 : -1);
        speed.v = 0;
    }
    else
    {
        speed.w = theta;
        speed.v = speedLimit;
    }

    return speed;
}

void setDriveDeceleration(float destinationRadius, float
minSpeedMultiplier)
{
    decelerationDestinationRadius = destinationRadius;
    decelerationMinSpeedMultiplier = minSpeedMultiplier;
}

void setDrivePosition(PositionType pos)
{
    VWSetPosition(vw, pos.x, pos.y, pos.phi);
}

int driveHasArrived()
{
    return remainingDriveDistance() < arrivalRadius;
}

void setDriveArrivalPrecision(float radius)
{
    arrivalRadius = radius;
}

PositionType getDrivePosition()
{
    PositionType position;
    VWGetPosition(vw, &position);
    return position;
}

float remainingDriveDistance()
{
    return distancept(getDrivePosition(), destination);
}

int getDriveVwHandle()
{
    return vw;
}

void initializeDrive()
{
    vw = VWInit(VW_DRIVE, 1);
//  VWStartControl(vw,2, 0.5, 7, 0.007);
//  VWStartControl(vw, 3, 0.3, 7, 0.1);
    VWStartControl(vw, 7.0, 0.3, 7.0, 0.1);
    VWSetPosition(vw,0,0,0);
    setDriveArrivalPrecision(DEFAULT_ARRIVAL_RADIUS);
    setDriveSpeedLimit(DEFAULT_SPEED_LIMIT);
    setDriveDeceleration(DEFAULT_DECELERATE_DESTINATION_RADIUS,
DEFAULT_DECELERATE_MIN_SPEED_MULTIPLIER);
    setDriveAngularSpeedLimit(DEFAULT_ANGULAR_SPEED_LIMIT);
}

void setDriveSpeedLimit(float v)
{
    speedLimit = v;
}

void setDriveAngularSpeedLimit(float w)
{
    angularSpeedLimit = w;
}

void setDriveDestination(PositionType dst, int dstType)
{
    destination = dst;
    destinationType = dstType;
}

void driveStop()
{
    VWSetSpeed(vw, 0, 0);
}

void updateDriveSpeed()
{
    PositionType pos;
    pos =  getDrivePosition();
    if( destinationType & DST_ARRIVAL_STOP &&
        distancept(destination,pos) < arrivalRadius)
    {
        SpeedType speed = {0.0f, 0.0f};
        setDriveSpeed(speed);
    }
    else
    {
        SpeedType speed;
        if(destinationType & DST_ARRIVAL_DECELERATE)
        {
            float deceleratedSpeedLimit = speedLimit *
customSigmoid(
                distancept(getDrivePosition(), destination),
                decelerationMinSpeedMultiplier, 1.0f,
                0.0f, decelerationDestinationRadius);

            speed = getLimitedDriveSpeed(deceleratedSpeedLimit);
        }
        else
        {
            speed = getDriveSpeed();
        }

        setDriveSpeed(speed);
    }
}

void setDriveSpeed(SpeedType speed)
{
    VWSetSpeed(vw, speed.v, speed.w);
}

SpeedType getLimitedDriveSpeed(float sl)
{
    float originalSl = speedLimit;
    speedLimit = sl;
    SpeedType speed = getDriveSpeed();
    speedLimit = originalSl;
    return speed;
}

SpeedType getDriveSpeed(void)
{
    PositionType origin;
    VWGetPosition(vw, &origin);
    if(destinationType & DST_STRAIGHT)
    {
        return getStraightSpeeds(origin, destination);
    }
    else
    {
        return getBezierSpeeds(origin, destination);
    }
}
```

```
#ifdef DRIVE_TEST

int main()
{
    initializeDrive();

    PositionType start, end;

    start.x = 0;
    start.y = 0;
    start.phi = 0;

    end.x = 1.0;
    end.y = -1.0;
    end.phi = 0;

    LCDMenu("GO", "", "", "");
    KEYWait(KEY1);

    setDriveDestination(end, DST_ARBITRARY | DST_ARRIVAL_STOP |
DST_ARRIVAL_DECELERATE | DST_STRAIGHT);

    while(1)
    {
        updateDriveSpeed();
        OSWait(1);

        if(driveHasArrived())
        {
            PositionType pos = {0,0,0};
            setDrivePosition(pos);
        }
    }

    VWSetSpeed(vw, -1, 0);

    OSWait(10);

    VWSetSpeed(vw, 0, 0);

    return 0;

}

#endif
```

vmath.h

```
#ifndef __VECTOR_MATH_H
#define __VECTOR_MATH_H

#include <math.h>

typedef struct
{
    float x;
    float y;
}
Point2D;

typedef union
{
    struct
    {
        Point2D p1, p2, b, c;
    };
    Point2D ps[4];
} Bezier;

inline static float distancept(PositionType a, PositionType b)
{
    float dx;
    float dy;
    dx = a.x - b.x;
    dy = a.y - b.y;
    return sqrtf(dx * dx + dy * dy);
}

inline static float distance(Point2D a, Point2D b)
{
    float dx = a.x - b.x;
    float dy = a.y - b.y;
    return sqrtf(dx * dx + dy * dy);
}

inline static float dotproduct(Point2D a, Point2D b)
{
    return a.x * b.x + a.y * b.y;
}

inline static float length(Point2D a)
{
    return sqrtf(a.x * a.x + a.y * a.y);
}


inline static Point2D normalize(Point2D a)
{
    Point2D result;
    float l = length(a);
    if(l == 0)
    {
        result.x = 0;
        result.y = 0;
    }
    else
    {
        result.x = a.x / l;
        result.y = a.y / l;
    }
    return result;
}

inline static float pangle(Point2D a)
{
    return atan2f(a.y, a.x);
}

inline static float angle(Point2D a, Point2D b)
{
    return pangle(b) - pangle(a);
}

inline static float radToDeg(float rad)
{
    return (rad / 2.0f * M_PI) * 360.0f;
}

inline static Point2D direction(Point2D a, Point2D b)
{
    Point2D delta = {b.x-a.x, b.y-a.y};
    return normalize(delta);
}

inline static Point2D scale(Point2D a, float s)
{
    Point2D result = {a.x * s, a.y * s};
    return result;
}

inline static Point2D rotate(Point2D a, float v)
{
    //x' = cos(theta)*x - sin(theta)*y
    //y' = sin(theta)*x + cos(theta)*y

    Point2D result = {
        cosf(v) * a.x - cosf(v) * a.y,
        sinf(v) * a.x + sinf(v) * a.y};
    return result;
}

inline static Point2D translate(Point2D a, Point2D b)
{
    Point2D result = {a.x+b.x, a.y + b.y};
    return result;
}

inline static float regularSigmoid(float x)
{
    return 1.0f / (1.0f + (float)expf(-x));
}

inline static float normalizedSigmoid(float x)
{
    return regularSigmoid(x * 12.0f - 6.0f);
}

inline static float sigmoid(float x, float a, float b)
{
    return normalizedSigmoid((x - a) / (b - a));
}

inline static float customSigmoid(float x, float min, float max,
float a, float b)
{
    return normalizedSigmoid((x - a) / (b - a)) * (max - min) +
min;
}


#endif // __VECTOR_MATH_H
```

1337!!1.c

```
#include <eyebot.h>
#include "../drive/drive.h"
#include <math.h>

#define ROBOT_RADIUS 0.09f

#define FIELD_WIDTH 2.4f
#define FIELD_HEIGHT 1.2f

#define WALL_OFFSET (ROBOT_RADIUS + ROBOT_RADIUS * 0.3f)

#define FRONT_OFFSET 0.037f
#define GOAL_OFFSET 0.1f

//#define CALIBRATE 1
#define SWEEP_WALLS 0
#define SWEEP_MIDDLE 1

#define MODE_SEED 1
#define MODE_MATCH 2

static PositionType home;
static PositionType ball;
static PositionType sweep1;
static PositionType sweep2;
static PositionType goal;

static int calibrate;

static void playSeed()
{
    int i, j;

    j = -1;

    while(1)
    {
        if(calibrate)
        {
            VWDriveStraight(getDriveVwHandle(), 0.25f, 0.3f);
            VWDriveWait(getDriveVwHandle());
            VWSetSpeed(getDriveVwHandle(), 0, 0);
            OSWait(100);

            VWDriveTurn(getDriveVwHandle(), -M_PI_2, 0.3f);
            VWDriveWait(getDriveVwHandle());
            VWSetSpeed(getDriveVwHandle(), 0, 0);
            OSWait(100);
            VWDriveStraight(getDriveVwHandle(), FIELD_HEIGHT / 2 +
0.1f, 0.1f);
            VWDriveWait(getDriveVwHandle());
            VWSetSpeed(getDriveVwHandle(), 0, 0);
            OSWait(100);

            VWGetPosition(getDriveVwHandle(), &sweep1);
            VWSetPosition(getDriveVwHandle(), sweep1.x, -
FIELD_HEIGHT / 2 + FRONT_OFFSET, -M_PI_2);

            VWDriveStraight(getDriveVwHandle(), -0.15f, 0.3f);
            VWDriveWait(getDriveVwHandle());
            VWSetSpeed(getDriveVwHandle(), 0, 0);
            OSWait(100);


            VWDriveTurn(getDriveVwHandle(), -M_PI_2, 0.3f);
            VWDriveWait(getDriveVwHandle());
            VWSetSpeed(getDriveVwHandle(), 0, 0);
            OSWait(100);


            VWDriveStraight(getDriveVwHandle(), 0.5f, 0.3f);
            VWDriveWait(getDriveVwHandle());
            VWSetSpeed(getDriveVwHandle(), 0, 0);
            OSWait(100);

            VWGetPosition(getDriveVwHandle(), &sweep1);
            VWSetPosition(getDriveVwHandle(), FRONT_OFFSET,
sweep1.y, -M_PI);

            VWDriveStraight(getDriveVwHandle(), -0.15f, 0.3f);
            VWDriveWait(getDriveVwHandle());
            VWSetSpeed(getDriveVwHandle(), 0, 0);
            OSWait(100);

            VWDriveTurn(getDriveVwHandle(), -M_PI_2, 0.3f);
            VWDriveWait(getDriveVwHandle());
            VWSetSpeed(getDriveVwHandle(), 0, 0);
            OSWait(100);


        }

        if(SWEEP_WALLS)
        {
            goal.y = 0.0f;

            for(i = 0; i < 2; ++i){

                ball.x = FIELD_WIDTH / 2;
                ball.y = j * (FIELD_HEIGHT / 2 - WALL_OFFSET);
                ball.phi = 0;

                // Go to ball
                setDriveSpeedLimit(0.1f);
                setDriveAngularSpeedLimit(0.4f);
                setDriveDestination(ball, DST_ARBITRARY |
DST_ARRIVAL_DECELERATE);
                driveUntilArrival(0.03f);

                // Go to goal

    setDriveDeceleration(DEFAULT_DECELERATE_DESTINATION_RADIUS,
0.3f);
                setDriveDestination(goal, DST_ARBITRARY |
DST_ARRIVAL_DECELERATE);
                setDriveSpeedLimit(0.1f);
                driveUntilArrival(0.03f);

                // Return to own goal

    setDriveDeceleration(DEFAULT_DECELERATE_DESTINATION_RADIUS,
DEFAULT_DECELERATE_MIN_SPEED_MULTIPLIER);
                setDriveSpeedLimit(0.1f);
                setDriveDestination(ball, DST_ARBITRARY |
DST_REVERSE);
                driveUntilArrival(0.10f);
                setDriveDestination(home, DST_ARBITRARY |
DST_ARRIVAL_DECELERATE | DST_REVERSE);
                driveUntilArrival(0.05f);

                // Other side
                j = -j;
            }
        }

        if(SWEEP_MIDDLE)
        {
            goal.y = j * 0.1f;
            setDriveSpeedLimit(0.1f);
            setDriveAngularSpeedLimit(0.3f);

            sweep1.x = FIELD_WIDTH / 2 - 0.05f;
            sweep1.y = j * (FIELD_HEIGHT / 2 - WALL_OFFSET);
            sweep1.phi = 0;

            setDriveDestination(sweep1, DST_ARBITRARY |
DST_ARRIVAL_DECELERATE);
            driveUntilArrival(0.03f);
            driveStop();
            OSWait(100);

            VWDriveTurn(getDriveVwHandle(), j*(-M_PI_2), 0.2f);
            VWDriveWait(getDriveVwHandle());

            /*sweep2.x = FIELD_WIDTH / 2 - 0.05f;
            sweep2.y = 0;
            sweep2.phi = -M_PI_2;

            setDriveAngularSpeedLimit(1.0f);
            setDriveDestination(sweep2, DST_ARBITRARY);
            driveUntilArrival(0.03f);                                    */

            setDriveSpeedLimit(0.05f);
            sweep2.x = FIELD_WIDTH / 2 - 0.05f;
            sweep2.y = j*(-(FIELD_HEIGHT / 2 - WALL_OFFSET *
1.5));
            sweep2.phi = j*(-M_PI_2);

            setDriveAngularSpeedLimit(0.1f);
            setDriveDestination(sweep2, DST_ARBITRARY);
            driveUntilArrival(0.05f);

/*          sweep2.x = FIELD_WIDTH / 2 + 0.7f;
            sweep2.y = -(FIELD_HEIGHT / 2 - WALL_OFFSET * 2.0f);
            sweep2.phi = 0.2f;

            setDriveDestination(sweep2, DST_ARBITRARY);
            driveUntilArrival(0.03f);*/

            setDriveDestination(goal, DST_ARBITRARY);
            driveUntilArrival(0.05f);

            setDriveSpeedLimit(0.1f);
            setDriveDestination(home, DST_ARBITRARY |
DST_REVERSE);
            driveUntilArrival(0.05f);

            j = -j;
        }
    }
}

static void playAttackMatch()
{
    sweep1.x = FIELD_WIDTH / 2;
    sweep1.y = 0;
    sweep1.phi = 0;

    setDriveSpeedLimit(0.5f);
    setDriveAngularSpeedLimit(4.0f);
    setDriveDestination(sweep1, DST_ARBITRARY);
    driveUntilArrival(0.03f);

    sweep1.x = FIELD_WIDTH * 0.75;
    sweep1.y = FIELD_HEIGHT / 2 - WALL_OFFSET * 2;
    sweep1.phi = 0;

    setDriveDestination(sweep1, DST_ARBITRARY);
    driveUntilArrival(0.03f);

    setDriveDestination(goal, DST_ARBITRARY);
    driveUntilArrival(0.05f);

    sweep1.x = FIELD_WIDTH / 2;
    sweep1.y = FIELD_HEIGHT / 2 - WALL_OFFSET;
    sweep1.phi = 0;

    setDriveDestination(sweep1, DST_ARBITRARY |
DST_ARRIVAL_DECELERATE | DST_REVERSE);
    driveUntilArrival(0.05f);

    sweep1.x = WALL_OFFSET * 2;
    sweep1.y = -0.4f;
    sweep1.phi = M_PI_2;

    sweep2.x = WALL_OFFSET * 2;
    sweep2.y = 0.4f;
    sweep2.phi = M_PI_2;

    while(1)
    {
        setDriveDestination(sweep1, DST_ARRIVAL_DECELERATE |
DST_REVERSE);
        driveUntilArrival(0.05f);

        setDriveDestination(sweep2, DST_ARRIVAL_DECELERATE);
        driveUntilArrival(0.05f);
    }
}

static void playAmokMatch()
{
    while(1)
    {
        VWSetSpeed(getDriveVwHandle(), 4, 0);
        OSWait(500);

        VWSetSpeed(getDriveVwHandle(), -4, 0);
        OSWait(200);
    }
```

```
}

static void playDefenseMatch()
{
    setDriveSpeedLimit(0.5f);
    setDriveAngularSpeedLimit(0.5f);

    sweep1.x = WALL_OFFSET * 2;
    sweep1.y = 0.4f;
    sweep1.phi = M_PI_2;

    sweep2.x = WALL_OFFSET * 2;
    sweep2.y = -0.4f;
    sweep2.phi = M_PI_2;

    while(1)
    {
        setDriveDestination(sweep1, DST_ARRIVAL_DECELERATE);
        driveUntilArrival(0.05f);

        setDriveDestination(sweep2, DST_ARRIVAL_DECELERATE |
DST_REVERSE);
        driveUntilArrival(0.05f);
    }
}

int main()
{


    home.x = 0;
    home.y = 0;
    home.phi = 0;
    goal.x = FIELD_WIDTH;
    goal.y = 0.1f;
    goal.phi = 0;

    LCDMenu("CL", "NCL", "NCL", "NCL");
    switch(KEYGet())
    {
    case KEY1:
        calibrate = 1;
        break;
    default:
        calibrate = 0;
        break;
    }

    while(1)
    {
        LCDMenu("Sd", "Dfns", "Atck", "Amk");
        switch(KEYGet())
        {
        case KEY1:
            OSWait(100);
            initializeDrive();
            playSeed();
            return 0;
        case KEY2:
            OSWait(100);
            initializeDrive();
            playDefenseMatch();
            return 0;
        case KEY3:
            OSWait(100);
            initializeDrive();
            playAttackMatch();
            return 0;
        case KEY4:
            OSWait(100);
            initializeDrive();
            playAmokMatch();
            return 0;
        default:
            break;
        }
    }

    return 0;
```