Programming for Failure- Enabling Robots to Recover from the Inevitable
Annaleise Kealiher
The Bots Who Say Ni

Programming for Failure

Enabling Robots to Recover from the Inevitable

1. Introduction

Robots often go astray. Claws drop things. Sensors misread surroundings. Reliability is indispensable for an achieving robot and hard to attain. This paper will provide guidance on how to correct a bot's course, retrieve abandoned game pieces, and improve a bot's dependability.

2. Self-Correcting Navigation

A massive problem with autonomous navigation is that small amounts of drift compound over time to devastating errors. To alleviate this, a robot needs to use sensors to evaluate its position and correct it accordingly. The advantages of creating a robot that can correct its own navigational failures are staggering. The testing and editing of code is a much shorter process with a sensor-driven robot. Small variances in hardware or battery levels can be mitigated by smart code.

There are multiple ways to achieve a course-correcting robot. One option is to use the internal encoders to measure the rotations on each motor. Unfortunately, these encoders are often not accurate enough, and the code has to be tweaked each time weight is redistributed, or motors are exchanged. Ideally, code should be able to work with whichever robot it is loaded to. Another option is the top-hat color sensor. When placed over a line, color sensors are extremely handy for helping your robot stay on the line and on track. However, most often, a robot will need to navigate to a place that doesn't have lines, so color sensors are of limited use. Finally, there is the ET sensor. ET distance sensors are an exceptional choice for eliminating navigational drift and improving robot reliability.

3. Wall following

Because walls are commonplace on a Botball game board, distance sensors can be an excellent aid to navigation. Using an ET's strengths in distance reading to follow a wall is much like using a color sensor to follow a line, except that wall following can be used to stay on track almost anywhere on the board. To follow a wall, the only coding logic needed for a robot is "Turn away from the wall if I am are too close; turn towards the wall I am are too far; and go straight ahead while in the 'Goldilocks Zone'".

```
int main()
{
    while(1){
        if(analog(1) > 1500){       //If the ET sensor sees you are closer than 1500 distance...
            turnright(50);          //turn away from the wall.

        }
        else if (analog(1) < 1300){ //Otherwise, if the ET sensor sees you are farther
                                    //than the 1300 distance...
            turnleft(50);           //turn towards the wall.
        }
        else{                       //Otherwise, if the ET sensor sees you are in the
                                    //correct distance zone...
            goForward(1000);        //go straight.
        }

    }
    return 0;
}
```

Figure 1: Code demonstrating wall following with ET sensors

Despite the code being so simple, there are several challenges which are inherent to wall following. ET sensors themselves are prone to flukes, and have distance limitations. Constantly wiggling and adjusting as it follows a wall, a robot's direction is never certain, which can cause problems later. Once these challenges are overcome, ETs can be an indispensable aid.

4. Solving problems with wall-following

Luckily, many of these problems have surprisingly simple solutions. If a sensor is giving irrational distances as occasional flukes, it is possible to reduce these errors by double checking the distance value.

```
    //this program will go forward until an ET sensor reads the wall as closer than 2000.

    int wallClose = 0; //initialize variable for wheather or not the wall is close
    while (wallClose == 0){ //while the wall is not close.....

        if(analog(1) > 2000){    //if the ET sensor sees a value larger than 2000...
            msleep(10);          //wait a teensy bit
            if(analog(1) > 2000){   //then look again to make sure the first time wasn't a fluke
                wallClose = 1;    //if it is still true, than the wall is close.
            }
        }
        goForward(100);
    }
    return 0;
}
```

Figure 2: Code sample demonstrating double checking a sensor's value

Another problem with the ET sensors are their inconsistencies at close distances that can limit their close-range usability. As a sensor approaches a wall, at some point about 5cm close to the wall, the sensor begins to see the wall as being farther away instead of closer.

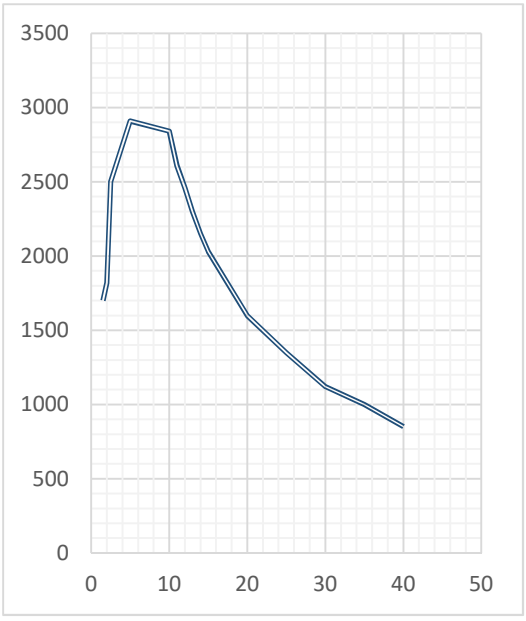

Table 1: Graph of data I collected on the distance reading from an ET sensor versus the actual distance the sensor is from the wall. The X axis shows the sensor's distance from a wall in centimeters, and the Y axis shows the analog output from the sensor. Note that at 5cm, the distance reading from the ET sensor actually gets smaller with decreasing distance unlike the rest of the graph.

This is because when the sensor is too close to the wall, the IR light it emits bounces back to the receiver on the sensor at such an extreme angle that a portion of the light starts to miss the

receiver altogether, messing up the distance reading. The best way to fix this is to mount the ET farther away from the wall on your robot so that it is never in a position to get an inaccurate reading.

To adjust its course, a wall-following robot has to turn back and forth to respond to the wall. This creates issues because, later, when a bot goes straight ahead after wall following, it will have lost track of which way "straight" is. This too can be resolved. Use of paired sensors, such as 2 ETs, 2 buttons, or 2 top-hats, can "square up" a robot.

If buttons are mounted on the front of a robot, for example, a robot just has to continue turning once the first button hits the wall, until the second button makes contact. Then the robot is perpendicular to the wall, and it can turn 90 degrees to be parallel again. By employing these adjustments to wall following, the robot is almost equipped to begin self-corrected navigation.

5. Pom-circling

One formidable challenge remains with wall-following: pom-circling. Diligently following what it recognizes as a wall, a robot can be led astray from its course by a single, stray pom. When an ET sensor takes a distance measurement, it shines IR light onto a surface and measures the angle of return of the reflected light. Upon hitting a pom, the IR light is scattered by the fluffy surface, making the pom seem father from the sensor than it really is. This will provide readings consistent with a distant wall. Once this robot sees this phantom wall, it will turn towards the pom, and begin a circling path around it.

Fortunately, using a combination of sensors can solve this. Although ETs are blind to color, cameras are quite capable at discerning colors from a distance. If the robot takes an occasional picture while wall-following, it can detect poms by their color in the picture, and know to drive past them before resuming wall-following. The fusion of ET sensors and cameras together can enable the bot to escape the pom circling trap, and keep on course.

```
int main()
{

    int timeCounter = seconds();    //Initialize a variable to keep track of the time
    camera_open();
    while(1){
        if(seconds() > timeCounter + 2){    //if it has been 2 seconds...
            printf("it has been 2 seconds");

            camera_update(); //enable the camera and take a picture.
            if(get_object_count(0) > 0){    //if the camera channel 0 sees one or more objects...
                printf("look out, I see a pom!");
                goForward(1000);    //drive past pesky pom
            }
            timeCounter = timeCounter + 2;  // resets time counter to proper time
        }
        //now we will start the wall following...
```

Figure 3: Code sample demonstrating how to avoid pom-circling with cameras

6. Reliability with game objects

Once it has successfully navigated to some game objects, a robot will need to score them, and again, reliability is critical. Even perfectly tuned programing will err occasionally. The most elegantly constructed claws will drop things. Things on the field will be missed. Game objects will be set up differently each time and eventually something will be outside of a robot's tolerances and inevitably be missed. When a robot misses or drops something, it wastes both time and points, and can create other problems such as pom-circling, or a physical barrier. The best way to make sure no game objects are left behind is to use sensors to detect the deserted pieces. If time allows, a robot can backtrack and recover lost pieces. If the game object is solid, such as a foam block, ET sensors are optimal. If the object has an oddly textured surface, or is brightly colored like a pom, a camera can be used instead. No matter the challenge, a system for recovery can provide contingencies to increase reliability and prevent unnecessary points waste.

7. Pom recovery with camera

The following is an example of using a camera to recover poms.

```
while(gotPom == 0){
    set_servo_position(0, 900); //open claw

    while(1)
    {
        camera_update(); //take a picture
        if(get_object_count(0) == 0 || get_object_center_x(0, 0) < 60)  //if there are no poms seen...
        {
            printf("there is no pom or pom is on the left\n");
            turnleft(10);
        }
        else if(get_object_center_x(0, 0) > 100)// if the pom is seen on the right, turn right.
        {
            printf("the pom is on the right \n");
            turnright(10);
        }
        else{ //if the pom is in the middle range of the camera's view, continue to the next part of the code.
            printf("the pom is dead ahead \n");
            break;
        }
    }
```

Figure 4: Code Sample using a camera to locate a pom

For this code, the camera must face forward on the robot. The code will take pictures using *camera_update();*, and evaluate the location of the pom based on these pictures. The camera should be set to a blob-tracking channel, and configured to notice the color blobs in pom colors. If the pom-colored blob in the picture is on the left or right, the robot turns accordingly until the blob is centered. Once the blob is straight ahead, the robot can continue on to the next portion of the code: grabbing the pom.

```
while(gotPom == 0){
    camera_update();
    if(get_object_count(0) == 0){ //if the camera has lost the pom...
        break;  //go back to other code.
    }

    if(get_object_area(0,0) < 1000){ //if the object is too small on the screen (far)...

        printf("the pom is too far away \n");
        goForward(10);

    }

    else{ //if the pom is large and in the middle of the screen, it is perfect for grabbing.

        printf("the pom is close enough to grab");

        set_servo_position(0,1330); // close claw around pom.
        gotPom = 1;// we have the pom now.

    }
}
```

Figure 5: Code Sample that gets closer to and grabs a pom

After centering the pom, the bot can begin moving closer. Occasionally, the camera can lose a blob, so to make the code more reliable, this code section should also make sure a pom-colored blob still exists before moving in. By evaluating the size of the pom-blob, the robot can then determine how far away it is, and move closer if necessary.

Because the robot is using a camera to navigate, the placement of the poms on the field is no longer critical to the program's success. This program can be used to locate other things too, such as a Botguy, or any other colored goal. However, once the object has been captured, the robot will be disoriented, and possibly off track. Using this method at the end of a program should make no additional issues, but often a robot will need to deliver the object instead of just grabbing it. Using ET sensor wall-following, a bot can reestablish its previous path to success.

8. Conclusion

Without sensors, robots are blind and will take wrong turns, fumble objects, and make mistakes. Through clever use of ET distance sensors, cameras, and other sensors, a robot will be able to see, touch, and make up a picture of the world around it. Most importantly, a robot with senses can consistently adapt to challenges and avoid failures.

Special thanks to the members of the Dead Robot Society team who offered training at the workshop and answered questions throughout the season.