Autonomous Maze Solving Robot

Eric Kaggen
May 2012

Introduction

I have built a robot which will autonomously navigate, learn, and find the goal point in a line maze. The robot can potentially search the entire maze in trying to find the exit because it must learn the structure of the maze as it navigates it. There's no way for the robot to know the best course through the maze before it tries everything. This is interesting it explores robot learning, and combines it with PID control for line following, and path planning for navigating what it already knows. Given the vast amount of different systems that need to work for the robot to work this robot was a huge challenge.

The goal of this project was to design a robot which could be placed on any line maze, and would navigate to the goal. I am pleased with the results of the project.

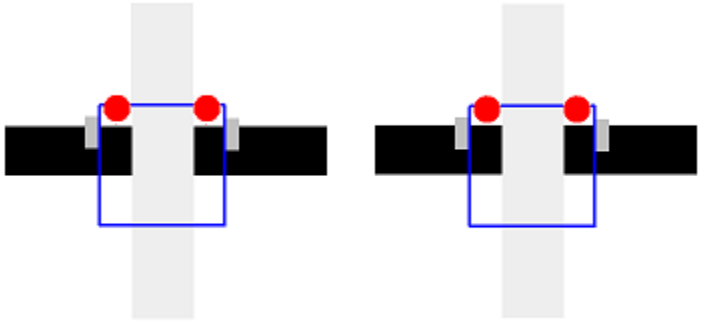
This paper is segmented into several sections. First I will discuss the physical design of the robot, including the types of sensors used, and their placement. After that I will go into the various systems involved in the logical design of the robot, along with the major challenges associated with each. This begins with a discussion about line following and how I accomplished it, including calibration. Then I will discuss how the robot detects intersections, and how it is able to classify them. I will then talk about how the system is split into the computer based master controller, and the on board robot controller. After which I will talk about how the robot actually navigates the maze using all of these systems, including how it knows that it's finished.

Physical Robot Design

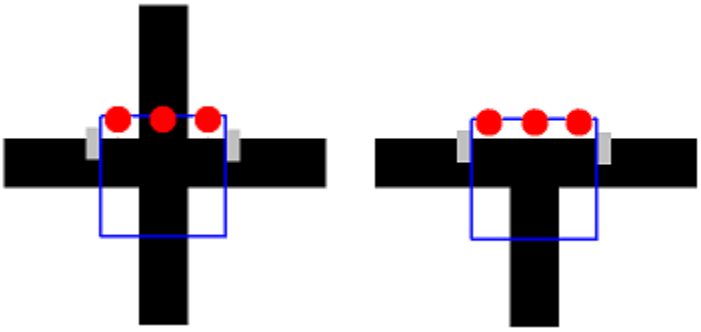
The difficulty in this project was in the software to control the robot. A basic design for the physical aspect of the robot was used. It has two front wheels for locomotion, and a pivoting caster type rear wheel to aid in turning and stability. It is especially good for following a fixed path as the one motor per wheel design allows quick adjustments to be made if the robot starts to fall off the line.

The robot uses three sensors. Two of them are plain NXT light sensors, and the third is a color sensor (which is mostly used as a third light sensor). In order for this robot to work it must have a minimum of three light sensors. While plain line following can be done with a single sensor it is very sensitive to noise. A second sensor adds increased reliability and allows higher speed. This would be fine for a plain line following robot, but the robot would have trouble with telling T-intersections apart from four way intersections. Thus a middle sensor is added. The picture below illustrates this issue.

What the robot sees with 2 light sensors



What the robot sees with 3 light sensors



The middle sensor also serves to detect the goal. The goal is a blue area, so the color sensor is used to detect the blue area so that the robot knows that it hit the goal.

Logical Robot Design

Line Following

The first step to getting my robot to navigate the maze was to make it stay "in the maze". Since this was a line maze this meant staying on the line. If it falls off the line and loses it then it will not be able to complete the rest of the maze, because it relies on having 100% certainty of its location relative to all parts of the maze that its seen in the past.

I implemented a PID controller to keep the robot on the line. The difference between the right sensor and left sensor is used to determine if the robot is moving off the line, and by how much.

$$e(t) = right(t) - left(t)$$

$$out(t) = K_p \cdot e(t) + \frac{K_i}{100} \cdot pastError + K_d \cdot \frac{de}{dt}$$

The K_i is divided by a constant 100 to make it the same order as the other constants (easier to tune).

$$leftPower(t) = basePower + out(t)$$

$$rightPower(t) = basePower - out(t)$$

$$pastError \leftarrow \frac{pastError}{dt} \cdot dampingConstant + error \cdot dt$$

$$dt = currentTicks - previousTicks$$

$$de = error - lastError$$

It can be noticed that the integral part of PID is slightly modified. Each time an update is done to the past error the amount of past errors is reduced. This in effect "ages" the past errors, so that old errors don't affect the present as much. This is because the robot is on a line. If the robot develops some errors which are then corrected it will be following the line again. The errors from this event won't completely go away, so robot will develop a new error in response to errors that may have happened long ago.

Since the NXT sensors are imperfect even the same type of light sensor may report a different value for the same external stimuli. This would throw PID off, since there would be an error detected even when there is none. Therefore I developed a calibration routine. When the robot is started the user invokes the calibration routine. The robot will spin in place and collect values from each sensor. A very simple form of machine learning is done with these values. The highest and lowest value for each sensor is stored. The robot then develops a linear hypothesis based on these values.

$$w_1 = \frac{high(sensor1) - low(sensor1)}{high(sensor2) - low(sensor2)}$$
$$w_0 = high(sensor1) - w_1 \cdot high(sensor2)$$

$$h(sensor2) = w_0 + w_1 \cdot sensor2$$

That hypothesis is used to normalize live sensory data so that the sensors will report equal values for the same stimuli.

$$sensor1 = h(sensor2)$$

Intersection Detection

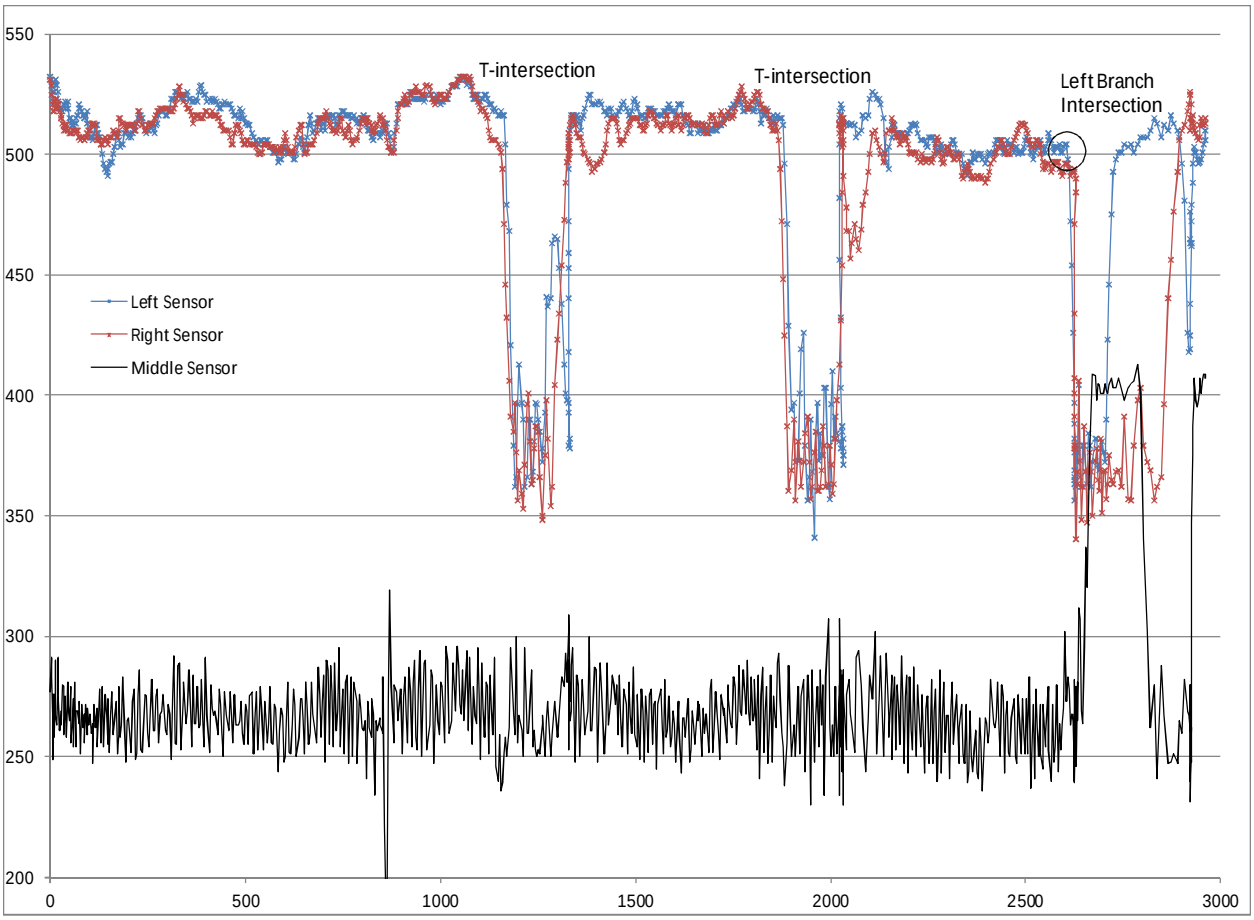
Intersection detection was by far the largest challenge to tackle as far as control goes. Its problem is twofold. First, to maintain control even when an intersection is encountered, and secondly to be able to detect the type of each intersection. Consider the follow types of intersections that may be encountered in a maze. These will be referenced later on.

Four way intersection | Branch intersection | A turn | T-intersection | Dead End

The middle branch intersection and turn are horizontally asymmetrical and come in both left and right forms.

Maintaining Control Through Intersections

When the robot is following a line and comes to an intersection the PID controller can fail and the robot will fly off the line depending on the type of intersection encountered. Specifically this issue especially impacts intersections which are horizontally asymmetrical from the perspective of the robot, such as a left branch intersection. In this case the robot will encounter the intersection and the value for the left sensor will drop. The PID controller will think that a very large error has developed, and will attempt to fix the error. Consult the following graph for a representation of this



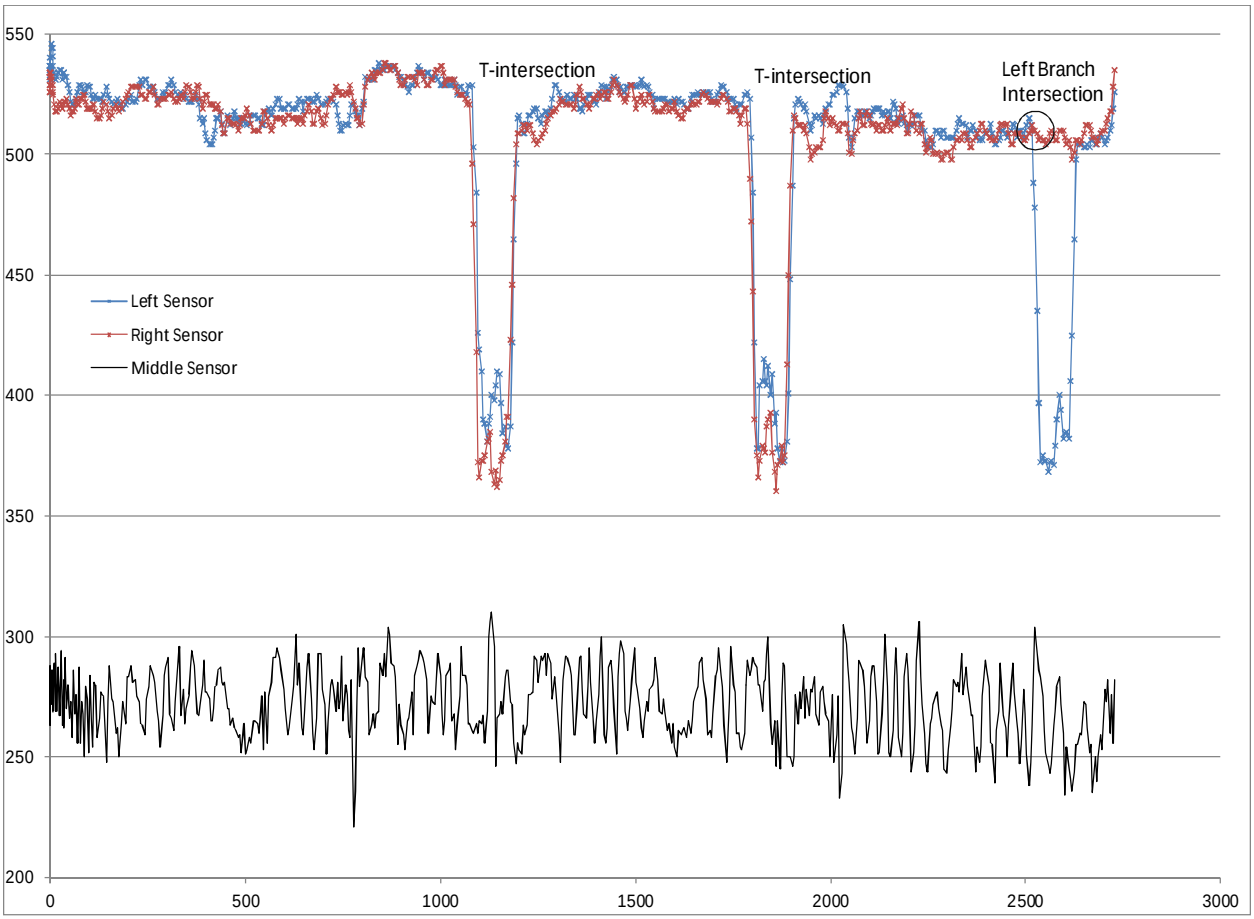
The robot in the graph first encounters two four way intersections. It doesn't have a big problem with these since both sensors stay in check (Although while the robot is on the intersection PID is "blind" since both sensors will report dark even if the robot is off center). The issue occurs when the robot hits a left branch intersection. You can see that the robot hits the left branch intersection, and then the left sensor rapidly drops. The right sensor shortly follows suit (indicates that the robot is now perpendicular to its previous direction). The robot continues to spin and the robot falls off the line (when the middle sensor is high the robot is off the line).

As seen by the circled area of the graph, there is a very short window during which this can be detected, so detection must have a low enough threshold to trigger fast, but must not trigger during normal events. I developed a formula which fits both of these categories. It takes into account the fact that as the robot approaches the intersection one or more sensors will drop in value quickly. This rate of change is the derivative of the sensors. Since both sensors will drop during a symmetrical intersection they both must be taken into account. Additionally, to respond extra quickly to asymmetrical intersections the rate of change of the error is taken into account.

$$intersectionDetect = \left|\frac{dRight}{dt} - \frac{dLeft}{dt}\right| - \frac{dRight}{dt} - \frac{dLeft}{dt}$$

This value will go high when an intersection is detected. It will go even higher faster when an asymmetrical intersection is detected. When this value goes above a certain threshold an intersection is being approached, and the robot enters intersection mode. PID is disabled while in intersection mode, so that the robot doesn't attempt to correct this potential "error". Since the robot is relatively straight on the line before the intersection it won't fall off the line in the short time the robot is over the line. The robot exits intersection mode after a specified distance corresponding to the width of the tape (detected by the rotary encoders in the wheels)

A graph of the sensor values from same series of intersections with intersection detection implemented is shown below. Notice that the robot remains constant throughout the intersection. The right sensor doesn't go crazy and the middle sensor stays low.



Detecting the type of intersection

Another challenge is detecting the type of an intersection. This is done in phases. While the robot is in intersection mode it will collect and store the error between the two sensors. When it exits intersection mode it sums errors up and divides by the time spent in the intersection.

$$classifier = \frac{\sum error}{\Delta t} \cdot threshold$$

The intersection is left asymmetrical if the classifier is less than 1, right asymmetrical if the classifier is greater than 1, otherwise it is symmetrical.

Finally, the robot must determine if the intersection has a straight component (such as a T-intersection vs. four way intersection). After exiting intersection mode the robot checks the value of the middle sensor. If it is still low then the intersection is determined to have a straight component. When no straight component exists the robot will stop so that it doesn't go straight off the line.

A final note, the robot won't enter intersection mode for a dead end since it technically isn't an intersection and therefore can't be detected as such, however it will report a dead end if all 3 sensors go high.

Split System Design

I split the software that controls the robot into two pieces. The first piece runs directly on the robot to take avoid the latency involved with remotely getting sensory data and responding to it. The second piece runs on a computer and communicates with the software on the robot using the Bluetooth communication protocol that I developed. The roles that each part play are similar to organisms in a real world situation. The robot software manages tasks which relate to staying on the line, and the computer manages the robot itself. It can be compared to unconscious primal part of the brain. You walk but you don't need to consciously adjust your balance and move each leg. The computer portion is like the conscious part of the brain. It tells the robot what to do, just like a person does when they tell their legs to walk forward.

The job of the first piece is to handle all micromanagement tasks which require quick responses to sensory data. The robot takes two commands (which are relevant to maze solving), go straight, and rotate. Go straight will cause the robot to go straight until it hits either a dead end, or until it passes a set number of intersections. The entire PID control system for staying on the line is on the robot. Intersection detection is on the robot too, as it must be able to quickly switch into and out of intersection mode. When an intersection is detected the robot will detect its type and tell the computer the type, as well as the time it was encountered over Bluetooth. It is stateless as far as the maze goes, it doesn't remember intersections after it's processed them. Since the time is given, the computer can sequence intersections and determine how far apart they are.

All of the maze solving logic resides in the command and control program (written in C#). The program maintains a map of what has been discovered in the maze so far, and gives the robot individual instructions to visit each one. It receives data about each intersection so that it can update its internal map when new intersections are discovered, and it can keep track of exactly where in the maze the robot is. The computer program also contains the GUI frontend for issuing commands to the entire system.

Maze Solving

The maze solving portion of the project is entirely on the computer program. This can complicate things because it must maintain synchronization with the robot at all times. Some clever tricks are used to accomplish this, most notably a fair amount of multi-threading.

The task of the first thread used for maze solving is to read Bluetooth packets off the pipe and store them. It will scan the packet and wake up any threads which are waiting for a packet which matches the incoming type.

The second thing needed for maze solving synchronization is packet filtering. When a command is sent out it can be assigned a packet filter which will catch all incoming packets matching a specific type which are associated with that outgoing command. A request to get a packet from the filter will block until the filter receives that packet. If the robot has completed the command associated with the filter then the request won't block, but will rather return a null packet. Two threads are needed to do this, one to wait for the "command completed packet" and the other to catch intermediate packets.

The last and most important thread actually does the maze solving task. I solve the maze by performing an implicit depth first search on a graph representation of the maze. However a simple depth first search won't suffice. Firstly, the computer program can move from node to node in its depth first search way faster than the robot can. Therefore the depth first search requires that an intersection detected packet be received each time it traverses from one node to the next. It uses the packet filter to do this, which will block if none are available. This way the depth first search on the computer will be in sync with the robot. As soon as the robot passes an intersection depth first will proceed. Also, unlike in a non-physical situation backtracking doesn't just involve forgetting about what you've just processed and processing the last node you saw. It actually involves physically directing the robot back to that node's location. The computer does a greedy "straight" first algorithm which allows the robot to pass multiple intersections with a single command. Since the robot will stop at dead ends the maze solver just needs to tell it where to turn next.

Results

After tweaking the PID constants and thresholds and getting reliable results I ran the robot through dozens of trials in many different mazes. The robot detected all intersections correctly and remained on the line, although there were varying degrees of smoothness to the line following. When testing I tried to design mazes which made the robot make lots of turns close together to try to break the PID control. I was unable to break the robot.