AI Robotics
Project 1
Demonstration of
Time-based and Dead Reckoning
Navigation Techniques

Karan M. Gupta
gupta@cs.ttu.edu

Rajat Goyal
goyal@cs.ttu.edu

Abstract

This report discusses the implementation of Dead Reckoning and Time-based techniques for autonomous robot navigation. It includes the various design issues encountered at different phases of this project. The project was implemented using Lego MindStorms Robotics Invention System 2.0 and the Not Quite C progamming language construct.

1 Introduction

When designing a robot it is necessary to know about the different techniques that can be used to navigate the robot. Here we explore two basic methods of robot navigation:

Dead Reckoning Prior to the development of celestial navigation, sailors navigated by ”deduced” (or ”dead”) reckoning. In Dead Reckoning, the navigator finds his position by measuring the course and distance he has travelled from some known point. Starting from a known point the navigator measures out his course and distance from that point.

Similar to the odometer in a car, estimating a robot’s position is accomplished by summing its incremental movements relative to a starting point.

In order for this method to work, the navigator needs a way to measure his course, and a way to measure the distance. This can be implemented in robotics by a variety of encoders like, mechanical shaft encoder, light sensing encoder, visual input, etc.

Time-based Navigation Dead Reckoning uses the distance as a measure for navigation. An alternative to this method is measuring the time the robot has travelled. The displacement of time, relative to a pre determined distance the robot can travel under various conditions in unit time, will give the exact location. This is Time-based Navigation.

2 Implementation

This project was implemented on a Lego MindStorms Robotics Invention System 2.0 kit. It was running the RCX 2.0 firmware. The Not Quite C (NQC) programming language was used to implement the Time-based and Dead Reckoning navigation algorithms. The project was implemented on the Windows XP platform with the BricX NQC Development Environment.

2.1 Construction

The robot was designed bearing in mind the job that is required for it to do, i.e. follow a particular predefined path. The robot needed to go in a straight line. It was required that it make several, moderately accurate, 90 degree turns. For such a task we decided to develop a robot that had a simple but effective design. The most important of element of our design is symmetry: every part on the robot’s body is attached as a pair. For every part on a side of the robot, there is another similar part on the opposite side.

The key components of the robot are:

- 1 RCX 2.0
- 2 9V Motors
- 2 Mid-size wide wheels
- 1 Mid-size wheel rim
- 2 Touch sensors
- Several gears

The main parts in the body of the robot are the two motors. These two motors form the mid-section of the robot and give width to the robot's body. These two motors are joined back-to-back and rest upon long beams that extend to the length of the robot's body. All of the other body parts are built upon this basic structure. The robot implements the mechanism of a front-wheel driven car. Two mid-size wide-body wheels run by the power given by the motors. Each motor runs one wheel only. The wheels are wide-bodied and were selected primarily to curtail the drift that could cause the robot to steer away from its defined straight line path. The wheels are connected to the motors by gears. Both the front wheels have a independent axle that runs from the wheel to the center of the body of the robot. Since the axles are free to rotate, there will also be some lateral movement of the axles, and not just a rotatory motion. This could cause some errors in the steering. If both the wheels are not looking at the exact (well, almost) same view ahead, one of them is likely to pull the robot away from a straight-line path. Therefore, the construction is such that each axle passes through two beams on the body. This minimizes the lateral movement of the axles keeping the wheels as much in synchronicity as possible with these parts.

Each motor has an eight-tooth small gear on its shaft. This passes the motor's motion to a crown gear, which rotates on a free axle, below the motor. The crown gear serves two important functions. Firstly, it passes the power from the motor to the wheel. Secondly, it passes the motion to another gear combination which drives an egg-shaped part. This is the bump-sensor activator. The combination of this activator and a touch sensor serves the purpose of a shaft encoder, counting the rotations of the wheel. The gear combination that drives it has the ratio 3:1 thereby giving the shaft encoder a high refresh rate. The shaft encoder is used only for the Dead Reckoning navigation method.

The mid-size wheel rim is the rear-wheel of the robot. It is attached to the body of the robot by a long freely rotatable axle, that passes through two beams. We have added a few bricks and plates to the body to make it stronger. The RCX is perched right on top of the robot allowing an easy view of the LCD panel and access to the buttons. The weight of the RCX lowers the centre of gravity of the robot, keeping it stable.

This construction gives us a reasonably straight-moving robot. Since, the rubber has been removed from the rear-wheel, excess friction between it and the floor due to the swaying of the body during turns has also been minimized.

2.2 Dead Reckoning

The hardware design consists of 2 bump-sensor activators attached to the axle. They are used to nudge the Touch Sensor whenever the wheel is turning. The Touch Sensor sends a message to the RCX code whenever it is activated (depressed or released). This system intuitively gives rise to a notion of an event driven mechanism. Every activation of the Touch Sensor can be treated as an event. The NQC language provides for a very flexible event handling system. There are 2 customizable events which are used, one for each sensor. MYEVENT1 plugs into SENSOR1 for "listening" to events on Motor A and similarly MYEVENT2 plugs into SENSOR3 for Motor C. The power settings for the motors for both, going forward and turning, were arrived at after extensive calibration.

The forward function uses Motor A to determine the distance travelled.(For purposes of calibration, the bump-sensor activator has to be aligned to be just on top of the Touch Sensor when starting for the first time). The distance to be travelled is passed to the function as a parameter. Everytime the event (MYEVENT1) is triggered the value of counter1 is incremented by 1. The value of counter1 is checked against a pre-determined value (number of clicks gained in 1 feet) multiplied by the distance to be travelled. Again, the number of "clicks" of Touch Sensor that the robot travels in one feet was determined after calibration. The robot is made to come to a complete

halt (motor is switched ”off” as opposed to coasting) after counter1 becomes equal to the designated value.

The turn function works in a similar way. The direction in which to turn (left/right) is passed as a parameter to the turn function. The number of clicks to be counted were calibrated against the motor that would turn the greater distance. (For example, while turning right, Motor C would travel the greater distance). The direction of both the motors is reversed so that the robot makes the turn taking its center as the turning pivot. The motor that is turning the greater distance gets more power than the other one. The robot is made to come to a complete halt after the turn maneuver has been completed.

2.3 Time-based Navigation

The design of the program is pretty simple for time-based navigation. The time taken to travel 1 foot was calibrated and hard coded. The power settings for both the motors were once again determined by calibration.

The distance to be travelled is passsed to the forward function as a parameter. The motors are switched on for the time period equal to the time required to travel 1 foot multiplied by the distance to be travelled. Both the motors are switched off after the robot has completed the maneuver.

The turn function takes the direction in which to turn as a parameter. The time taken to turn 90 degrees, and the power settings for each motor were calibrated and hard coded. The direction of both the motors is reversed so that the robot makes the turn taking its center as the turning pivot. The motor that is turning the greater distance gets more power than the other one. The robot is made to come to a complete halt after the turn maneuver has been completed.

In both the programs the robot is made to wait for a period of 200ms after the completion of every maneuver.

3 Design Issues

As with any project, there were several bumps along the way that we had to face. However, we managed to solve most of the problems we were having, in the time that was given. Here’s a short description of the route we took to reach the current design.

Being overly ambitious in the beginning as well as being armed with the knowledge that wheels tend to slip, we decided to make a humanoid robot. Our robot was going to have legs and arms and knees and a pelvis, and it was going to walk. So we started by drawing out as much of the design as we could on a board and then set about to implement the design. The first lesson learnt was: *Things are not as easy as they seem*. No matter how much we tried, we were unable to copy the action of the pelvis in walking. It seemed as though we needed a third motor. So, that idea was abandoned due to lack of resources.

The next design was a two-wheeler. This was a very small robot, devoid of any bricks except the RCX and two motors. It was extremely light and the wheels were directly conncted to the motor’s shafts which in turn were directly attached to the RCX. It ran beautifully, however, there was no control over the direction. That was when we realised that having at least one more wheel, or some sort of rear-rest is absolutely necessary.

Knowing that it was going to be difficult to make accurate turns (we did not have a robot then which ran straight), we decided on a holomonic architecture. The robot was to be made up of two bodies: a lower body which had the wheels and a motor which was responsible for giving forward motion to the wheels. There was a differential gear which helped keep both the wheels move by the same amount. There was another big part which rotated freely over the lower part which had another motor for rotating the wheels-base below it. The two bodies were connected by a shaft on which both the bodies could rotate freely. The RCX was placed on the higher set to make it heavy. However, this design grossly misfired because it was easier for the higher body to rotate itself around the shaft than to make four wheels rotate over the ground. Once again we were at a loss for another motor which could have been used to build a hydraulic system to ease the turning motion.

Finally we came upon the current design: simple but effective. However, we faced some problems here too. But one-by-one we got rid of most of them. The first problem was making the robot go in a straight line. That was solved by using long axles

that went into the center of the body and through 2 bricks at least. This prevented lateral movement of the axles. The straight-moving problem was also solved by keeping the body as narrow as possible, curtailing drag in any one direction. Moreover, every lego part on one side was placed at exactly the same spot on the other side, to maintain weight balance(exactly like weight-balancing is done in wheels on cars). By this time, the robot was performing fairly well, except for a few random quirks during turns. After several hours we understood what could be a probable reason for this. We added a *Wait(200 msec)* statement in our programs after every task (*goFwd* and *turn*). This allowed the motors to come to the correct electronic potential for the next maneuver. However, there was still some quirkiness in the turns. This we figured to be from the rubber ring on the rear-wheel. When the robot turned the front wheels moved and the rear-wheel simply dragged along the ground. This led to a lot of friction from the rear-end of the robot's body. So, we removed the ring and the turns were considerably improved.

Apart from the hardware issues there were several software problems that we encountered. We initially wanted to develop the code for this project using legOS running on a Linux machine. We made several attempts to compile the necessary legOS tools for USB transfer, but eventually gave it up due to compilation problems. After that we tried installing the same tools using CygWin on Windows XP and once again faced compilation problems, however, this time in the legOS source code itself. Then we began RCX Brick Language but had to give it up very soon, because it was simply not powerful enough and neither was it comfortable enough for us to code in. It does not support structural programming. Finally we settled with NQC and the BricX NQC Development Environment which we found to be extremely conducive towards efficient coding.

The biggest problem in both dead-reckoning and time-based navigation is that errors keep on accumulating. If the robot were to mess up on the first turn (say, by 5 degrees), there is no saying how far it could go away from the finish point even if all the other turns are perfect.

4 Conclusion

These methods suffer from lack of goal specification. The robot does not have a goal to achieve, except to follow the path as best as it can. The path has been pre coded into the robot. It cannot generate its own path, in case of a stochastic environment for example. The introduction of a random variable in the environment that changes the path parameters renders the robot useless. A navigation method that allows the robot to identify/specify a goal would work much better. An example of such a method is Potential Fields. Once the goal has been specified the robot can generate its own path to the goal.

Another "roadblock" in these methods is obstacle avoidance. If a robot, following either of these navigation techniques, was to encounter a random obstacle then it would certainly mess up the hard coded values. For example, if a robot running on time-based navigation were to get stuck in a rut for 2 seconds and then got out and continued on its original path, then it would certainly fall short of the final destination by the distance travelled in unit time * 2 (taking an optimistic estimate). Once again potential fields will be able to come up with a better solution to such situations. The robot can still carry on its job, delayed by 2 seconds, but still able to reach the goal.