Navigation Using Position Tracking
Wesley Y. Myers
Ethan Y. Myers
Cedar Brook Academy

Navigation Using Position Tracking

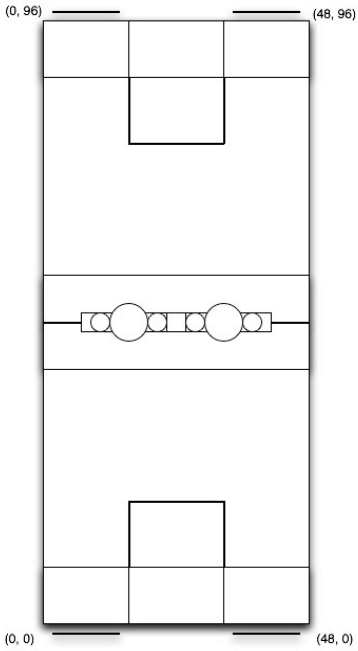
1. Introduction

This paper presents how to navigate using position tracking. Position tracking is where the robot keeps track of its position by monitoring its motion. We described our implementation of position tracking at last year's conference[1] and that paper is summarized in the next section.

2. Background – Position Tracking

Position tracking is where the robot uses measurements of the wheel motion to keep track of its location and heading. To do position tracking, a coordinate system is first established. In this case, the game table is overlaid with a coordinate system, which has its origin at the bottom left corner. Since the table's width is 48 inches, the domain becomes [0,48]. The length of the table is 96 inches, so the range is [0,96]. Thus, the coordinate of the top right corner of the table is (48,96). The direction is in radians where 0 radians is parallel to the x-axis going from left to right. (See Figure 1)

Figure 1 – 2006 Botball Table



When a robot moves, its movement can be represented as a vector consisting of direction and distance. In a given amount of time, the robot will move some distance along a direction from a known position. This movement has to be translated into a new position in the table's coordinate system. The average movement of both of the robot's wheels provides the measurement for the distance that the robot has traveled, and the difference between the right and left wheels' movement can be used to calculate the change in direction.

The XBC uses Back EMF to record how much the motors have moved the wheels[2]. This information is obtained by using the get_motor_position_counter function of the XBC firmware. The distance traveled by each wheel is calculated by subtracting the previous counters from the current counters to get the number of

ticks traveled by each motor. The result is then divided by a conversion factor that changes the units into inches to get Δright_distance and Δleft_distance. Each motor has a different conversion factor because each motor is slightly different.

Δleft_distance = (left_counts – previous_left_counts) / left_counts_inch
Δright_distance = (right_counts – previous_right_counts) / right_counts_inch
Δdistance = (Δright_distance + Δleft_distance)/2

Given the direction and the distance, trigonometry is used to find Δx and Δy. So:

Δx = Δdistance * cos(direction)
Δy = Δdistance * sin(direction)

To get the new location, we add Δx and Δy to the original location values. To calculate the change in heading, we recognize that any difference between Δright_distance and Δleft_distance will cause the robot to change direction. The direction change is equal to:

(Δright_distance - Δleft_distance)/(the distance between the wheels)

Figure 2

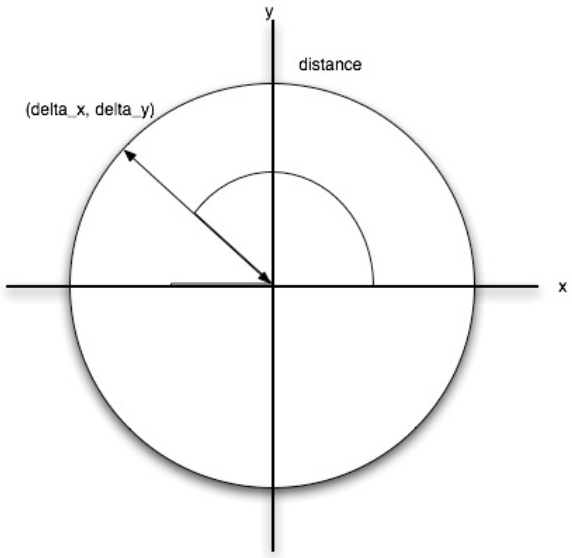
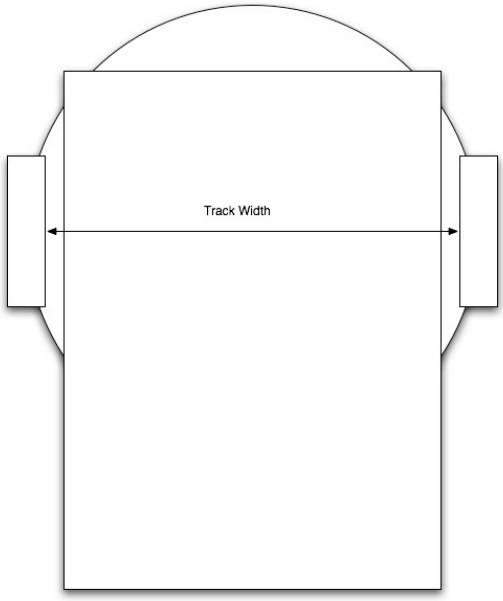


Figure 3



Where Track Width is equal to the distance between the wheels.

3.0 Navigation

Our navigation functions are based on getting and keeping the robot pointed in the proper direction as the robot moves. For rotation, the robot is turned in place until it points to the correct direction, while in go_to_point the robot is kept pointed at the

destination until the destination is reached.

3.1 Navigation – Rotation

One of the most important things when it comes to navigation is turning in the direction you want. The first step in turning to the correct direction is getting the current heading. The position tracking code provides the heading information. To find how far the robot needs to turn, the difference between the current and desired direction has to be calculated (1.1). To keep the result within the range of [-2π, 2π], the result is normalized (1.2). An interesting result from normalization is that the shortest rotation direction is also found. We know from trigonometry that -3π/2 is equivalent to π/2 and 4π is equivalent to 2π (Figure 4).

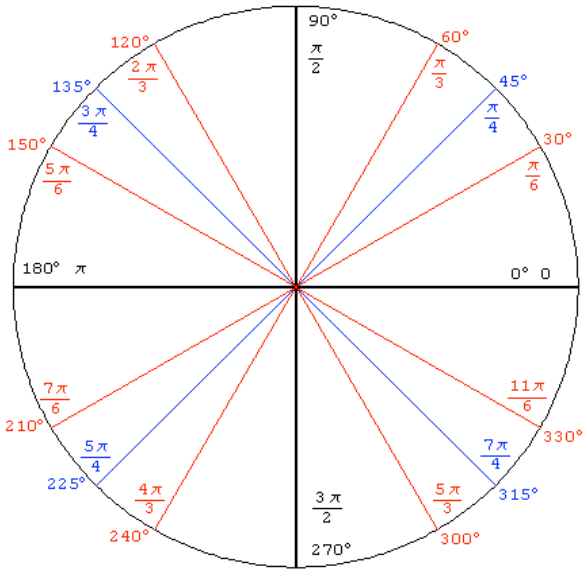
It is important to note that at the completion of this function, the robot will not be exactly pointed in the desired direction, or no zero error. Because obtaining zero error is impossible because of imprecision in controlling the motors, a small error such as .1 radian is suitable for our use. So, as the robot turns, the absolute error is continuously checked until it is within the desired limits of error (1.3).

(1.1) error = target_heading – current_heading;

(1.2) error = normalize_heading(error);

(1.3) while(absolute_error < absolute_value(rotation_error))
turn();

Figure 4



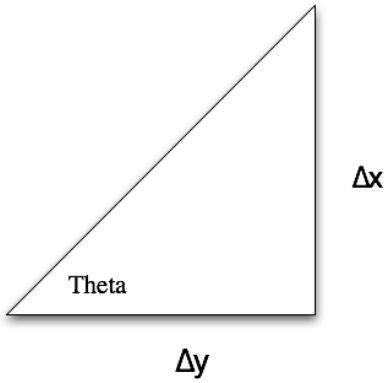
3.2 Navigation – Go_to_Point

Another feature of the navigation library is to move the robot to a point on the table. The go_to_point function does this by keeping the robot pointed at the destination while the robot is moving and stopping when the robot has reached the destination. This is based on RidgeSoft's paper[3].

Go_to_point has to continuously deal with any directional error as the robot moves

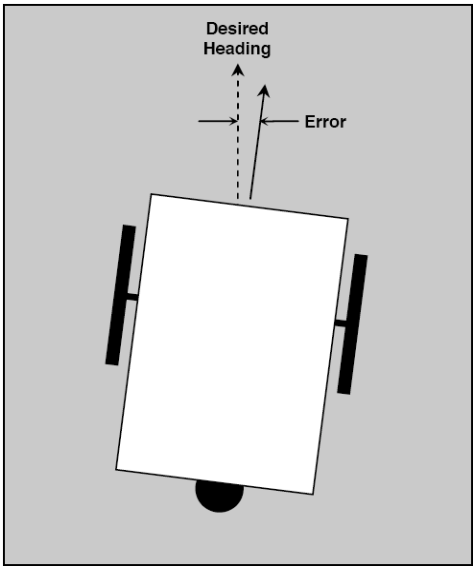
towards the destination. If the robot is traveling in the correct direction, then it will go straight to the point. However, in the beginning, as the robot starts moving, there is usually error in the robot's direction, which needs to be corrected. Also, the robot needs to periodically deal with error as it moves to the destination. To do this, the directional error is determined by determining the difference between the current direction and the direction the robot needs to take to get to the end point. (Figure 6) To find the new direction to the destination, the inverse tangent of Δx and Δy is used to find the angle, theta. (Figure 5)

Figure 5



The error is multiplied by a factor to determine the power levels supplied to the motors. Generally, the power is reduced to the motor on the side where we want the robot to turn. By doing this, it not only keeps the robot pointed in the right direction, but it also gets it to the destination. If the robot is also pointed in the wrong direction in the beginning, it will realize that and vary the wheel speed so that it will turn in the correct direction and then start moving. Over the course of the robot traveling to the point, it periodically checks and corrects its alignment on the target point. This keeps the robot lined up on the target.

Figure 6[3]



As in the rotate function, we also need to find how far off the robot is from where it needs to go. So the difference is taken between the current and desired locations, and take the absolute value of them (1.4). When the sum of the x and y errors are less than the allowed error, then the robot is at the desired location. We recognize that the proper way is to use Pythagorean's Theorem. It provides a more correct answer, but the current method gets the robot close enough and saves processing time.

(1.4) x_error = go_to_x – current_x;
y_error = go_to_y – current_y;
absolute_x_error = absolute_value(x_error);
absolute_y_error = absolute_value(y_error);

Since the robot sometimes needs to get to tricky places, go_to_point can also make the robot go backwards instead of forwards. This is done by adding π to the current direction when calculating the directional error. This, in effect, reverses the direction that the robot will be moving. The other difference is that the wheel powers are reversed so that it can drive backwards.

3.3 Navigation – Going Straight

A basic capability that every robot needs is the ability to go straight, without moving in a curve to one side. The Go to Point function discussed in Section 3.2 can be used to do this by calculating the endpoint based on the desired distance. Trigonometry is used to calculate the endpoint.

There are a couple things already known: the robot's current position, heading, and the distance the robot is to travel. From Trigonometry we know a couple of rules about right triangles. The cosine of the angle is really the x length divided by the hypotenuse and the sine of the angle is the y length divided by the hypotenuse.

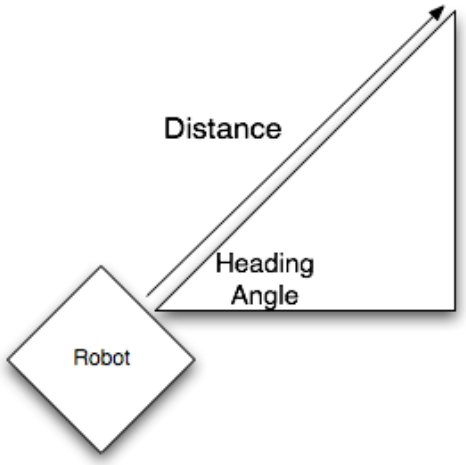
(1.5) $\cos(\theta) = x/z$, where z is the hypotenuse
(1.6) $\sin(\theta) = y/z$, where z is the hypotenuse

Thus:

Δx = distance * cos(direction)
Δy = distance * sin(direction)

The destination is determined by adding Δx and Δy to our current location. Then go_to_point is called with the destination coordinates as the parameters. Since the robot is already pointing at the destination, it will go straight there.

Figure 7



4.0 Status

Most of the Navigation Library was implemented between last year's (2006) regional and national tournaments. One of our robots successfully used the navigation library in conjunction with the position tracking library at the last year's nationals. At this year's regional tournament, both of our robots successfully used these libraries. We have found that this library, in particular go_to_point function, has greatly simplified our

navigation. Most of our dead reckoning instructions use go_to_point instead of the motor commands. For example, if we need to add an intermediate turn or point, we just add a call to go_to_point instead of having to recalculate a whole sequence of motor commands. Finally, even though the position library gains error over time, our strategies are such that we move to within a couple of inches of the destination and use sensors to get us the rest of the way. In summary, our navigation library is a very useful library.

References

[1] Myers, Wesley, Myers, Ethan. "Position Tracking Using the XBC." Proceedings of the 5th Annual Conference on Educational Robotics 7 July 2006: 157-163.

[2] R. LeGrand, K. Machulis, D. Miller, R. Sargent and A. Wright, "The XBC: a Modern Low-Cost Mobile Robot Controller," Proceeding of IROS 2005, IEEE Press, 2005.

[3] RidgeSoft. "Programming Your Robot To Navigate." RidgeSoft, LLC. (2005): 1-27.