Clicks per unit distance (handy when writing navigation code)	7.8 click/cm (19.8 click/in.)
Distance between wheels, center-to-center, (this value is the parameter *b* used in the equations of motion below)	19.75 cm, 7.7 in, 154 clicks
Difference of right and left encoder values when making a 90-degree turn	242
Maximum motor speed	270 RPM
Maximum forward robot speed	9.2 cm/sec (3.6 in/sec, 72 clicks/sec)
Mass (weight)	638 grams (1.4 lb.)

Table 1. Vital statistics.

Differential Steering and PID

The two gearmotors that came with the Mindstorms kit that was used to build Peeves are not perfectly matched. Due to variations in manufacture, one runs about 5 percent slower than the other. An imbalance in motor output is important in a differential steering system because even small variations in wheel speed can result in the robot straying from a straight-line path. The default tendency for Peeves was to curve to its left.

Unfortunately, a differentially steered vehicle is not an inherently stable system. Left to its own devices, it will not travel in a straight line, or even along a simple curve. But whenever an author says that a differentially steered vehicle is not an inherently stable system, it is useful to point out that neither is a bicycle. Yet, somehow, a bicycle manages to operate just fine. Despite its lack of inherent stability, a bike has no trouble getting around. It can do so, of course, because it is coupled with an active controller unit... in this case, the rider.

The Bicycle Metaphor

The bicycle metaphor is borrowed from John Lienhard's *The Engines of Our Ingenuity* broadcasts on National Public Radio (episode 106). You may read Dr. Lienhard's commentaries on the history of science and technology at http://www.uh.edu/engines/ If you have a chance, be sure to listen to some of his audio's. They are a real treat.

A robot also has an active controller unit. And, given the right software, it can guide a differential-steering system with smooth precision. In the Peeves implementation, the basis of that software is a technique known as PID.

The acronym PID is short for *Proportional, Integral, and Derivative.* In the PID approach, a controller monitors the error in the system (its deviation from some desired value, or *set point*) and makes corrections based on three criteria. The *Proportional* response is based on the magnitude of the observed error, the *Integral* of that error (error accumulated over time), and the *Derivative* of the error (the rate at which the error changes over time). The PID approach was the basis of the system used to control Peeves and keep it following a straight line.

To illustrate the idea behind the PID technique, let's apply it to the familiar problem of steering an automobile on highway. We desire to keep the car in the center of its lane. If the car begins to drift to the right, we apply a *correction* by turning the steering wheel to the left to stop the movement away from the centerline. In this case, the *set point* is simply for the car to travel in a straight line. We define the speed of the car's lateral movement as our *observed rate of error.* The faster it moves away from the centerline, the

higher the error. Naturally, a competent driver does not overcorrect. So, if the error is small, our response is small. If the error is large, our response is large. In other words, our initial correction is *Proportional* to the error rate (giving us the 'P' in PID).

If we turn the wheel enough, the car will stop moving to the right and drive in a straight line. Although the car will have drifted from the center of the lane, we will have achieved an error rate of zero because we define the error in terms of the lateral speed, not distance off-center. Still, we do care about the car not being in the center of its lane. So we define a second kind of error, the *accumulated error*. In mathematical terms, this value is simply the *Integral* of the error over time (giving us the 'I' in PID). And, referring back to our driving example, we realize that the correction applied to the steering is often proportional to values: the instantaneous drift rate, and the integrated error.

If all goes well, the car will reverse its rightward drift and return to the center of its lane fairly quickly. Perhaps a little too quickly... If we do not back off from the correction as we approach the center of the lane (zero accumulated error), the car will overshoot the centerline and begin to drift to the left. If that happens, we need to make a second, counter-adjustment so that the car returns to the right. But if we over-correct, we may need another counter-adjustment to the left. If this continues, the system will oscillate. To damp that oscillation, the PID technique adds a third factor based on the rate-of-change for the error function, that is its *Derivative* (giving us the 'D' in PID).

The car analogy illustrates the three basic components of the PID technique. In general, we consider the instantaneous and integrated (accumulated) error and make a proportional adjustment. Although this adjustment is often enough, it sometimes leads to a system that tends to overcorrect and oscillate as it returns back to the set point. In such cases, we add the third correction to help settle the system.

For purposes of this discussion, we will let *e(t)* represent a general error function and use the following variation of the PID equation:

$$c(t) = P_P e(t) + P_I \int e(t)dt + P_D \frac{de}{dt} \qquad [1]$$

where

$c(t)$ is the correction factor to be applied to the system;
P_P is the adjustment coefficient for the observed error;
P_I is the adjustment coefficient for the integrated error;
P_D is the adjustment coefficient for the derivative of the error.

This is a non-standard treatment of the PID equation. Most texts place a negative sign behind the derivative term to reflect its role as a damping factor. I use the plus sign because that is how I ultimately coded the term in the Peeves logic. Because of the way I defined my error functions, the value of derivative term is often the opposite sign of the error term (see below). Note also, that some implementations omit the derivative term. Such an approach is suited to cases where the derivative has a small magnitude or where a significant lag in sampling and processing measurements makes it impossible to compute a derivative in a timely manner. In such cases, authors often refer to a *PI solution* rather than a *PID solution*.

There are several web sites describing the PID technique. A good place to start can be found at the Control Engineering Tutorials web site or at the joint Carnegie-Mellon/University of Michigan Control Tutorials for Matlab pages. Many books on control theory also discuss the PID technique. At least one of these is available on line: Lee, Newell, and Cameron's Process Management and Control (the discussion of PID tuning techniques in chapter 5 is especially useful).

PID for Peeves: How one robot implements the algorithm

One common use of the PID technique is in controlling motor speed. If I had taken this approach in the Peeves implementation, I could have used two sets of PID calculations to maintain a constant speed for each motor. Because the Lego rotation sensors are coupled directly to the motors, they form a closed-loop system. Feedback from each sensor can be used as input into the PID calculation for the corresponding motor. Additional calculations can combine sensor inputs to track data such as orientation and position, coordinating the motors and adjusting their respective speeds as needed. Unfortunately, implementing such an approach turned out to be more complicated than I could accomplish given my short development time and the coding limitations of the RCX/NQC environment.

Instead, I took a less ambitious approach. Rather than looking at the problem as a matter of controlling two motors, I looked at the robot as a whole. For the CRS contest, it didn't matter how fast the robot traveled. So controlling the absolute speed of each motor was not as important as controlling their speed relative to each other. The Peeves implementation focuses on the robot's path, and disregards its speed. The important thing is simply that the vehicle goes straight.

The implementation uses a *single* PID calculation based on its observed lateral drift rate to set PWM output levels (and thus, the relative speed) for *the pair* of motors. Essentially, this implementation attempts to control a secondary effect (the overall behavior of the robot), rather than a primary effect (individual motor speed). It was a design choice that worked, but may not have been the best way to get the job done. In the future, I hope to explore the twin-controller implementation. Perhaps that will be the subject of another article. For now, let's look at how Peeves implements its PID controller.

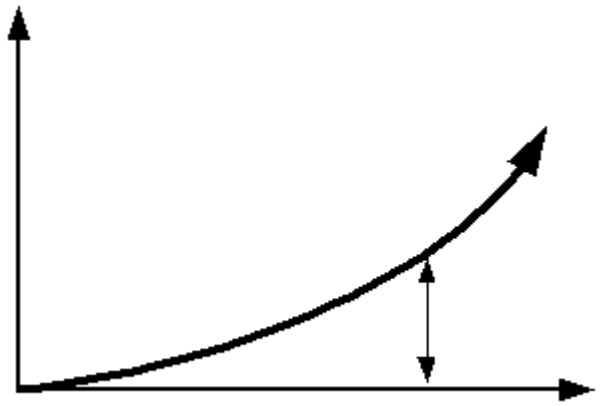

Figure 4. The straight-line path is along the x-axis. The error function is defined to be the rates of change of the y coordinate over time (a straight line being zero error). The integrated error is treated as being the distance from the x-axis, and is sometimes referred to as the *accumulated error* or the *lateral offset from center*.

Figure 4 shows the track of a robot on a Cartesian coordinate plane. If the robot is to follow a straight-line course, it should move along the x-axis. As error accumulates, it tends to move off the axis. There were a couple of different ways that I could have defined the error functions for the PID equation. For example, I could have focussed on the robot's orientation. But, after a bit of trial and error, I settled on the definitions used in the automobile example used above.

Thus, the error function *e(t)* is given by

$$e(t) = \frac{dy}{dt} \qquad [2]$$

The equivalent to the PID equation in [1] is derived by substituting the results from [2] and performing the appropriate integration and differentiation.

$$c(t) = P_P \frac{dy}{dt} + P_I y(t) + P_D \frac{d^2 y}{dt^2} \quad [3]$$

As I was coding, this approach got a bit confusing at times. Because my error function, *e(t)*, is actually a derivative of another function, *y(t)*, the positions of the operators change in two forms of the PID equations [1] and [3]. So when I was referring to "the derivative term in the PID equation", I had to remind myself that I meant the derivative of *e(t)*, not *y(t)*. As you read the notes below, you may find it useful to refer to the following table.

$P_P e(t)$	$P_P \frac{dy}{dt}$	Error term of PID equation: how fast are we drifting from the center?
$P_I \int e(t)dt$	$P_I y(t)$	Integral term of PID equation: how far are we from centerline?
$P_D \frac{de}{dt}$	$P_D \frac{d^2 y}{dt^2}$	Derivative term of PID equation: is the drift rate accelerating or decelerating?

Table 2. Relationships between error functions.

Once I decided how I was going to define the error function for the PID equation, I needed to complete the following steps:

1. Find equations for the error function e(t) = dy/dt, its integral, and its derivative.
2. Find workable parameters for the PID equation.
3. Write code using numerical techniques to solve the PID equation for a differential steering (which involves sine and cosine) while working in the 2-byte integer data types available in the Mindstorms system.

Equations for Differential Steering

Using the PID technique requires that we have some way to compute the error function, its integral, and its derivative. How do we convert data collected by the encoders into a description of the robot's movements? To do so, we need to know something about the behavior of a differential steering system. Fortunately, I had already tackled the details for an earlier article I had written for the Rossum Project (see An Elementary Trajectory Model for the Differential Steering System). From this source, and others, I was able to obtain equations that described the orientation of the robot, and its y-coordinate as functions of wheel velocities and time. These equations made the simplifying assumption that the wheel velocities were held constant. This assumption seemed reasonable since the large reduction ratio of Peeves' gear train provided a high torque relative to the robot's mass. When motor power levels were changed, the wheels would spin "up to speed" quite quickly. So as long as the robot stayed close to its desired straight-line course, the simplifying assumptions would not result in too much error. The Rossum Project web site provided an applet that allowed me to experiment with the consequences of different settings and test this assumption (see MotionApplet).

The following equations were used in the Peeves project to describe the motion of the robot. They can be applied to any differential-steering system as long as you are careful about the simplifying assumptions mentioned above.

$$\theta(t) = (v_R - v_L)t/b + \theta_0 \quad [4]$$

$$r = \frac{b(v_R + v_L)}{2(v_R - v_L)} \quad [5]$$

$$\frac{dy}{dt} = \frac{(v_R + v_L)}{2}\sin((v_R - v_L)t/b + \theta_0) \quad [6]$$

$$y(t) = y_0 - \frac{b(v_R + v_L)}{2(v_R - v_L)}\left[\cos((v_R - v_L)t/b + \theta_0) - \cos(\theta_0)\right] \quad [7]$$

$$\frac{d^2y}{dt^2} = \frac{(v_R + v_L)(v_R - v_L)}{2b}\cos((v_R - v_L)t/b + \theta_0) \quad [8]$$

The following equation is not used in the dead-reckoning implementation, but is included for completeness:

$$x(t) = x_0 + \frac{b(v_R + v_L)}{2(v_R - v_L)}\left[\sin((v_R - v_L)t/b + \theta_0) - \sin(\theta_0)\right] \quad [9]$$

where

x_0, y_0	is the robot's initial position,
$x(t), y(t)$	are the robot's position as a function of time,
$\theta_0, \theta(t)$	are the robot's initial and computed orientation, respectively, (always given in radians),
v_R, v_L	are the velocities of the right and left wheels, respectively,
b	is the distance between wheels, center-to-center,
r	is the robot's turn radius,
t	is time, in seconds.

Tuning the PID Parameters

Once I had equations for the PID component functions, the next step was to obtain parameters for the PID equation. Because the selection of parameters depends on the physical characteristics of the system, there is no stock set of values that can be applied to every implementation. Instead, the parameters for the equations must be *tuned* to the particular platform they are intended to control.

The selection of PID parameters is the aspect of the Peeves Project with which I am least satisfied. Although I eventually arrived at values that work, they do not represent an optimal solution.

In general, there are three approaches that can be used to find values for the PID parameters:

Analytical	Based on theory and accurate measurements of the physical parameters of the apparatus, derive a *transfer function* that describes the response of the system to changes in its input values. Chose parameters using standard techniques from control theory.
Empirical	Using bench-test results for the overall system, apply general-purpose procedures such as the Zeigler-Nichols method which compute parameters based on observed system behavior.
Trial and Error	Starting with an "initial guess", observe the behavior of the system and tweak the parameters until a stable solution is found.

Table 3. Three approaches to tuning PID parameters.

The analytical treatment is probably the best, but requires a thorough understanding of the physics behind the system under consideration and a basic understanding of techniques from control theory. Although my description might seem a bit intimidating, the subject is well covered in many undergraduate-level textbooks, including the on-line version of Process Management and Control referenced above.

Unfortunately, the two-month lead time for the CRS odometry contest did not allow enough time for me to get "up to speed" on the material. In the absence of a good theoretical model of the system, I attempted to use an empirical technique. There are a number of well-known empirical techniques that can be used to tune PID parameters. One of the oldest, and most straightforward, of these is a method promulgated by Zeigler and Nichols in the 1940's. In the Zeigler-Nichols open-loop technique, a developer runs the system "on-the-bench" and observes its response to changes in input level. Test results are then plugged into an series of simple equations that produce estimates for the PID coefficients. Zeigler-Nichols and related empirical methods often yield better results than a simple trial and error approach. They also have the advantage of not requiring knowledge of the process dynamics of a system.

You may read a short article about the Zeigler-Nichols technique in the Control Engineering Archives. The on-line text Process Motion and Control, Chapter 5, discusses PID tuning techniques at length and offers several more recent, but computationally intensive, alternatives to Zeigler-Nichols.

Although I attempted to use the Zeigler-Nichols method to develop PID parameters for my own system, I'm afraid I struck out. My bench tests yielded unsuitable results. I concluded that either I was misapplying the technique or that the dynamics of the differential-steering system did not match some of the basic assumptions of Zeigler-Nichols. With no time left for further research, I resorted to trial and error.

Thanks to tuning guidelines in the Carnegie-Mellon PID Tutorial, I was able to develop a set of parameters that worked reasonably well for Peeves. While the values I obtained were not optimal, an optimal set of parameters was not absolutely required for the odometry contest. Because the robot was expected to stick close to a straight-line path, the error values would be quite small and well within the capabilities of the Peeves implementation. It was only when I tried to stress the system in non-competition testing that shortcomings became apparent.

To see the effect of the sub-optimal selection of parameters, consider Figure 5. The figure shows the results from an experiment where the initial values of the accumulated error (integral error) was set to 40 clicks

(51.2 mm, or about 2 inches). The robot was instructed to run a straight-line path and store its accumulated and instantaneous error values in its datalog. Recall that the integrated error is the robot's lateral offset from the straight-line path, or its y-coordinate, and is given by the function *y(t)* (a distance here measured in millimeters). And, again, we define its observed error, *e(t)*, to be the rate of change of its y-coordinate with respect to time, or *dy/dt* (expressed as lateral movement in millimeters per second). Note that the PID algorithm corrects for the integrated (accumulated) error, an offset of 50 millimeters to the left, by steering to the right. The integrated error drops quite quickly, with relatively little oscillation. As the integrated error gets smaller, so does the magnitude of the controller's response. Ordinarily, this behavior is exactly what we would hope to see. Unfortunately, when the integrated error becomes less than 7 millimeters, the magnitude of the system's response is rather too small and it becomes slow to correct for the remaining offset.

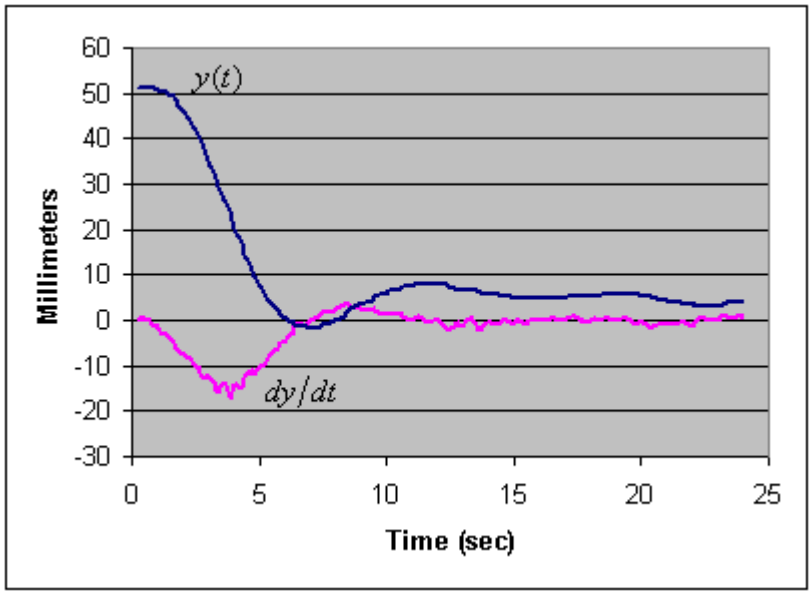

Figure 5. Measured data from straight-line test.

While I was disappointed by Peeves' inability to correct for the remaining offset in a timely manner, it was not that serious an issue in the contest. At most, it degrades the final accuracy of the robot by half a centimeter. Still, in a competition where millimeters count, it could become a serious issue. I plan on continuing my research on the subject and hope to offer more useful information in a future article.

Approximation Issues: Scaling Distance, Time, and Angle to Integers

Once the PID parameters were established, it was time to code the equations for the error functions in the NQC (Mindstorms) programming language. As noted above, the only data type supported by the RCX firmware is the 2-byte signed integer. So not only were floating-point variables unavailable, but any integer arithmetic would have to remain within the range of -32768 to 32767. I would also have to work out approximations for the sine and cosine functions used in the equations shown in [6], [7], and [8].

Perhaps the most common way of handling fractional values on a system that supports only integers is to "scale" the values by multiplying them by some constant. The trick in choosing a scalar is to pick one large enough to avoid a loss of precision while finding one small enough that integer overflow doesn't become a problem. In the case of measures of distance, I was lucky enough to avoid the problem of scaling and round-off altogether.

As noted above, each click of the encoder corresponds to a distance of 1.28 mm. Rather than trying to find a way to scale conventional units of distance, I simply defined the "encoder click" as my fundamental unit of distance. By treating all distances in terms of "clicks", the Peeves software was left free of any need to perform conversions or account for loss of precision.

The treatment of time was also easy. For the version of the RCX firmware that I was running, system timers had the resolution of 0.1 second, a fact that presented a natural value for scaling (a newer version of the firmware provides 0.01-second resolution). In cases where the error functions required time values in their denominators, I simply multiplied the numerators by appropriate powers of 10.

Angles were only a little harder. The equations used to describe the differential-steering system treat angles in radians, not degrees. Although it was tempting to scale them by simply converting them to degrees, a better solution suggested itself. In the equations used for Peeves, angles are always used in close association with the parameter *b* (the distance between wheels, center-to-center), which in Peeves case has a value of 154 "clicks". So, internally, angles were multiplied by *b* to scale them to integers. For example, to specify a right angle, the NQC code for Peeves uses the value $154(\pi/2)$, or about 242 clicks.

NQC does not provide utilities for computing the sine and cosine functions that occur in the differential-steering equations. Fortunately, the trig functions in equations [6], [7], and [8] are used to operate on the angle deviation from the straight-line path. Since the whole point of the PID implementation was to keep the robot traveling in a straight line, I could assume that these angles would be quite small. And, for angles sufficiently close to zero, the Taylor series provides the convenient approximations $\sin(\theta) \approx \theta$ and $\cos\theta \approx 1 - \theta^2/2$. A little algebra and a willingness to disregard terms that were small contributors to the overall value, yielded the following approximations for [6], [7], and [8]:

$$\frac{dy}{dt} \approx (w_R + w_L)(w_R - w_L + b\theta_0)/2tb \quad [10]$$

$$y(t) \approx y_0 + (w_R + w_L)(w_R - w_L + 2b\theta_0)/4b \quad [11]$$

$$\frac{d^2y}{dt^2} \approx (w_R + w_L)(w_R - w_L)/2bt^2 \quad [12]$$

where

w_R, w_L are the displacements of the right and left wheels over the time interval *t*.

The only thing left to do was to add the PID coefficients into the equations and express them in code. A bit more work was required to obtain accurate calculations within the restrictions of 2-byte integers. You may see the results in the source code at peeves.html.

Keeping to the straight and narrow

The CRS dead-reckoning course can be treated as a sequence of maneuvers, alternating between straight-line and cornering operations. Competing robots begin the course traveling in a straight line down a *corridor*. Initially, the Peeves code sets to motors to default "go-straight" values that were determined through experimentation. Due to motor-related issues mentioned above, even the best choice of available parameters resulted in the robot traveling a slightly curved path. So as soon as the motors are activated, the program enters a PID-control loop.

During the straight-line operation, the program monitors its sensors to check for deviations from its standard course. Every 0.2 seconds, it performs the PID-related calculations and determines if a correction is required (although the RCX clock is calibrated in 0.1-second intervals, the larger 0.2-second value was chosen to reduce the effects of timing errors). If the PID calculation indicates the need for a correction, the software adjusts the motor output as appropriate. Refer to the source code at peeves.html for more detail.

The robot also measures distance-traveled along the straight-line path to determine when it reaches the end of a corridor. Once the corridor maneuver is completed, it switches to a turning operation.

Turning the Corner

The PID technique described above kept Peeves traveling in a straight line. Although I may have been able to work out an analogous PID methodology for rounding the corners, I elected to use a simpler approach. In the turning phase the course, the robot simply sets the PWM level for its outer wheel at higher value than for the inner wheel. The resulting difference in wheel speed causes the system to follow a nearly circular path as described in the equations in [7] and [9]. During the turn phase, Peeves simply monitors the angle of its orientation and breaks out of the turn when it reaches 90 degrees. Upon completion of the turn the robot immediately sets its wheels to pre-determined, "go-straight" settings and then performs calculations for the next phase of its navigation.

These calculations involve correcting for errors that may have accumulated during the turn. First, there is a certain amount of delay in the robot's response when it completes the turn. It may not sample its sensors until after it has crossed the 90-degree threshold and even when it sets its PWM output to "go-straight" it will continue to rotate until the motors can overcome the robot's angular momentum. This error in orientation is typically small and can be corrected by the PID logic when the robot enters its next straight-line maneuver. To do so, the program simply notes the deviation from a perfect right-angle turn and treats it as an initial condition for the algorithm to process.

A more substantial error results when the turn radius of the robot deviates from the *expected turn radius*. Refer to Figure 6. If the robot were to execute a turn with a radius equal to half the width of the course (9 inches, 22.9 cm), it could begin the turn right at the end of one corridor and complete the turn at the beginning of the next. But if the actual turn exceeded the turn radius by some deviation factor, ε, the robot would finish its turn outside the centerline and forward of the beginning of the corridor. The control logic needs to account for this deviation and apply a correction.

We can compute the actual turn radius using data from the rotation sensors and equation [5] above. To compensate for how far the robot has advanced into the corridor, we simply reduce the length of the straight-line maneuver it is about to execute.

Unfortunately, the lateral deviation (to the left and right of the centerline) is not so easily corrected. Although we could use the PID logic to adjust for the lateral deviation (treating it as the integral error term in the PID equation), the magnitude of the error might be too large to be absorbed by the compensation logic. Instead, we defer the

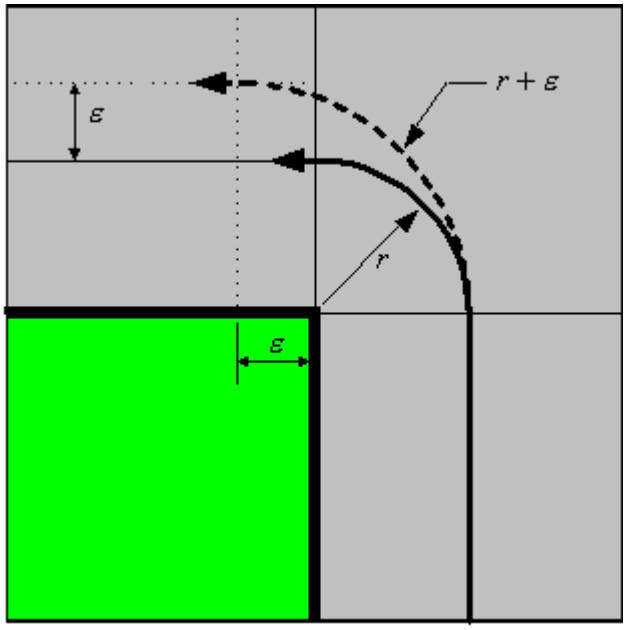

Figure 6. The deviation from central path when the turn radius is larger than expected. The vehicle finishes the turn both forward and outside of its intended position. Fortunately the error can be computed accurately. A symmetrical case exists when the turn radius is smaller than expected.

correction until *after* the next turn. We can compensate for a lateral error on the current leg of the course by simply shortening or lengthening the distance the robot travels on the next.

Unfortunately, after the last turn in the course – when the robot is in the home stretch – there is no *next* leg to adjust. While the obvious solution is to simply plug this lateral error into the PID logic (treating it as the integral error component), I have not yet been able to make it work properly. For now, the robot knows that it is off the centerline, but does not have a method to correct for the difference.

As you can see, the ability to predict the radius of the next turn is an important part of keeping the robot near the centerline of the course. For example, if we estimate that the turn radius is going to be larger than the standard 9-inches (half width of the course), we instruct the robot to begin its turn before it reaches the end of the corridor. Unfortunately, the actual turn radius depends on many different factors including battery age, ambient temperature, and the kind of flooring. That last factor was relevant in the Connecticut Robotics Society event because the competition took place on a different kind of flooring (linoleum) than had been used for developing the Peeves logic (concrete). The program did not have a good turn-radius estimate for the first corner and deviated substantially from the centerline. As it progressed, however, it was able to keep track of its performance on each turn and improve its estimates until it had corrected for the early deviation.

In addition to the offset due to turn radius, the robot also experiences a certain amount of orientation error due to "not-quite-90-degree" turns. Timing and control issues make it very difficult to execute a perfect right-angle turn. Fortunately, the deviation from a 90-degree turn can be measured using the wheel encoders and plugged back into the PID calculations when the robot reenters its straight-line phase. Any error in orientation is quickly absorbed by the PID algorithm.

Things Left Undone

In recent test runs on a concrete floor, Peeves consistently finished within 3 centimeters of its starting position... typically, about 1/2 cm short of the start/finish line and 2 cm to the right of center. While such accuracy is pretty good for a Lego-based robot, there is still room for improvement. Due to the sub-optimal selection of PID parameters mentioned above, I have not successfully implemented a correction for the lateral error on the last turn. Nor have I been completely successful in predicting the radius for the last turn (which would reduce the need for a lateral correction). Even if I addressed these issues, Peeves' internal position tracking can only account for about half the error. The robot knows that it is not dead-on the start/finish position, but consistently underestimates its error. The discrepancy is probably due to a combination of mechanical inaccuracies and effects such as acceleration that are not correctly handled by the software. An imperfect transition from the cornering phase of its operation to the straight-line traversal is also a likely source of inaccuracies.

Conclusion

I believe that the competition in next year's CRS odometry and dead reckoning contest will be far more intense than it was this June. I sincerely hope that Peeves will be beaten by another robot that benefited from the information I've presented in this article. If the success of Peeves illustrates anything, it is the value of harnessing the large body of information available to the roboticist on the web and in the library. I hope that some of the notes that I have presented in this article will lead other robot builders to the resources they need for their own projects.

Acknowledgments

This article would not have been possible without the assistance of many people. I would like to thank Charles Olsen, a talented and knowledgeable engineer who patiently answered my incessant questions about electronics and control theory. Thanks also go to the the members of the Connecticut Robotics Society, in particular Bill Ruehl and Jake Mendelssohn whose good sense and extensive knowledge of robotics are always an inspiration, and to John Gallichotte who organized the CRS odometry event and made important suggestions regarding this article. I'd also like to thank Kenneth Maxon whose earlier *Encoder* article Reaching the Next Step in Motion Control and Sensor-Software Fusion for Mobile Robots helped jump start the Peeves development effort... without the information he provided, I wouldn't have known where to begin.

The author gratefully acknowledges the assistance of Sonalysts, Inc. in the production of the graphics and photographs used in this article. Sonalysts is a Connecticut-based company providing services including software engineering for systems ranging from embedded processors to corporate intranets, from rapid prototyping to life-cycle maintenance.

And, finally, I would like to thank my wife, Kathy Lucas, for her assistance in the preparation of this document and long-suffering support throughout the Peeves project.

Comments? Write to the author gwlucas@connix.com.