A Primer on Odometry and Motor Control

Edwin Olson
eolson@mit.edu

May 3, 2006

1   Introduction

Most robotics problems ultimately reduce to the question: where am I? How can you go somewhere else without some notion of where you are now? You could proceed naively, stumbling around, hoping that your goal will appear in range of your robot's sensors, but a better plan is to navigate. A basic method of navigation, used by virtually all robots, is *odometry*, using knowledge of your wheel's motion to estimate your vehicle's motion.

The goal of this primer is to show you how to use odometry data. A real dataset will be used extensively to show how well (and how badly) these ideas can work.

Suppose your robot starts at the origin, pointed down the x-axis. Its state is $(x, y, \theta) = (0, 0, 0)$. If the robot travels (roughly) straight for three seconds at 1 m/s, a good guess of your state will be $(3, 0, 0)$. That's the essence of odometry.

We'll assume that the vehicle is differentially driven: it has a motor on the left side of the robot, and another motor on the right side. If both motors rotate forward, the robot goes (roughly) straight. If the right motor turns faster than the left motor, the robot will move left.

Our goal is to measure how fast our left and right motors are turning. From this, we can measure our velocity and rate of turn, and then integrate these quantities to obtain our position.

2   Derivation

The speeds of our motors give us two quantities: the rate at which the vehicle is turning, and the rate at which the vehicle is moving forward. All we have to do is integrate these two quantities, and we'll have our robot's state (x, y, θ).

That sounds a bit scary, but the mathematics end up being very simple. If we had analytic expressions for the angular rate and forward rate functions, their integral probably would be scary. But in a real system, our data comes from real sensors that are *sampled* periodically. Every few milliseconds, we get a new measurement of our motor velocities. We will numerically integrate our solution, which is of course, just a fancy way of saying that we'll divide time up into little chunks and just add up all the little pieces.

Suppose our robot is at (x, y, θ). Depending on what kind of sensors we have, we might get measurements of how much the motors have rotated (in radians), or an estimate of how fast they're rotating (angular rate, radians/sec.) We'll consider the first case, but of course the second case can be handled by just multiplying the angular rate by the amount of time that has elapsed since our last iteration. Given the amount of rotation of the motor and the diameter of the wheel, we can compute the actual distance that the wheel has covered.

Suppose the left wheel has moved by a distance of d_{left} and the right wheel has moved d_{right}. For some small period of time (such that d_{left} and d_{right} are short), we can reasonably assume that the vehicle trajectory was an arc (see Fig. 1).
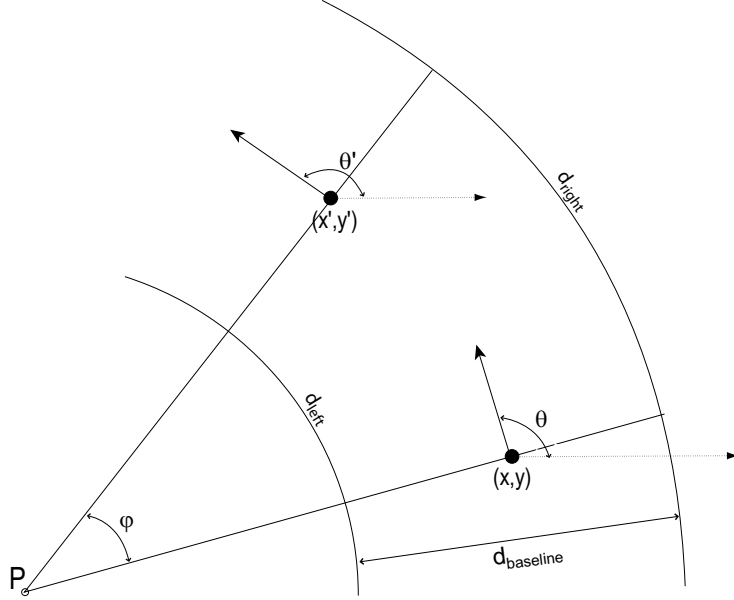

Figure 1: Odometry geometry. Over a small time period, the robot's motion can be approximated by an arc. The odometry problem is to solve for (x', y', θ') given (x, y, θ) and $d_{baseline}$. In the figure, the robot is moving counter-clockwise.

The initial state (x, y, θ) defines our starting point, with θ representing the vehicle's heading. After our vehicle has moved by d_{left} and d_{right}, we want to compute the new position, (x', y', θ').

The center of the robot (the spot immediately between the two wheels that define's the robot's location), travels along an arc as well. Remembering that arc length is equal to the radius times the inner angle, the length of this arc is:

$$d_{center} = \frac{d_{left} + d_{right}}{2} \tag{1}$$

Given basic geometry, we know that:

$$\phi r_{left} = d_{left} \tag{2}$$
$$\phi r_{right} = d_{right} \tag{3}$$

If $d_{baseline}$ is the distance between the left and right wheels, we can write:

$$r_{left} + d_{baseline} = r_{right} \tag{4}$$

Subtracting (2) from (3), we see:

$$\phi r_{right} - \phi r_{left} = d_{right} - d_{left}$$
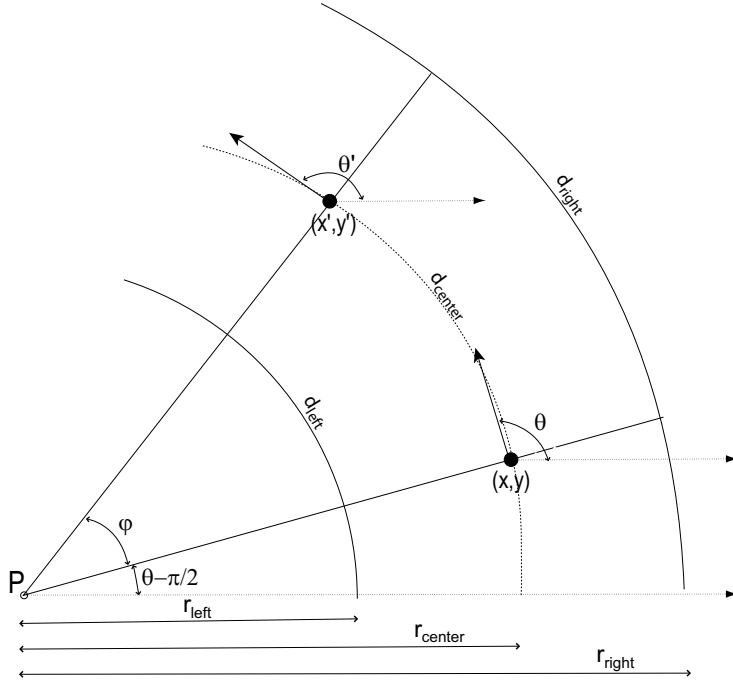$$\phi(r_{right} - r_{left}) = d_{right} - d_{left}$$

Figure 2: Detailed odometry geometry. We compute the center of the circle P by forcing d_{left} and d_{right} to be two arcs with the same inner angle ϕ. From this, we can compute the new robot position (x', y', θ').

$$\phi d_{baseline} = d_{right} - d_{left}$$
$$\phi = \frac{d_{right} - d_{left}}{d_{baseline}} \tag{5}$$

At the risk of making the math a bit uglier than really necessary, let's very carefully compute the new robot state. All of our arcs have a common origin at point P. Note that the angle of the robot's baseline with respect to the x axis is $\theta - \pi/2$. We now compute the coordinates of P:

$$
\begin{aligned}
P_x &= x - r_{center} cos(\theta - \pi/2) \\
&= x - r_{center} sin(\theta) \\
P_y &= y - r_{center} sin(\theta - \pi/2) \\
&= y + r_{center} cos(\theta)
\end{aligned}
$$

Now we can compute x' and y':

$$
\begin{aligned}
x' &= P_x + r_{center} cos(\phi + \theta - \pi/2) \\
&= x - r_{center} sin(\theta) + r_{center} * sin(\phi + \theta) \\
&= x + r_{center}[-sin(\theta) + sin(\phi)cos(\theta) + sin(\theta)cos(\phi)]
\end{aligned}
\tag{6}
$$

and

$$
\begin{aligned}
y' &= P_y + r_{center}sin(\phi + \theta - \pi/2) \\
&= y + r_{center}cos(\theta) - r_{center}cos(\phi + \theta) \\
&= y + r_{center}[cos(\theta) - cos(\phi)cos(\theta) + sin(\theta)sin(\phi)] \quad (7)
\end{aligned}
$$

If ϕ is small (as is usually the case for small time steps), we can approximate $sin(\phi) = \phi$ and $cos(\phi) = 1$. This now gives us:

$$
\begin{aligned}
x' &= x + r_{center}[-sin(\theta) + \phi cos(\theta) + sin(\theta)] \\
&= x + r_{center}\phi cos(\theta) \\
&= x + d_{center}cos(\theta) \quad (8)
\end{aligned}
$$

and

$$
\begin{aligned}
y' &= y + r_{center}[cos(\theta) - cos(\theta) + \phi sin(\theta)] \\
&= y + r_{center}[\phi sin(\theta)] \\
&= y + d_{center}sin(\theta) \quad (9)
\end{aligned}
$$

In summary, our odometry equations for (x', y', θ') reduce to:

$$
\begin{aligned}
d_{center} &= \frac{d_{left} + d_{right}}{2} \quad &(10) \\
\phi &= \frac{d_{right} - d_{left}}{d_{baseline}} \quad &(11) \\
\theta' &= \theta + \phi \quad &(12) \\
x' &= x + d_{center}cos(\theta) \quad &(13) \\
y' &= y + d_{center}sin(\theta) \quad &(14)
\end{aligned}
$$

3 Basic Motor Theory

Subsequent sections will discuss how we can measure the performance of our motors. We'll need to put this into context by talking first about some basic motor physics.

Motors are devices which manipulate magnetic fields in order to convert electrical energy into rotational energy. An electric current flows through coils of wire, causing a magnetic field to form. This magnetic field interacts with permanent magnets (like those on your refrigerator), causing the shaft to rotate.

We also know that motors can be used in reverse; if you rotate the shaft of the motor, the permanent magnets induce a current in the rotating coils of wire. This, of course, is known as a generator.

It is important to realize that motors are *always* behaving as generators. If the output shaft is rotating, a current is being induced in the wires. Usually, this current opposes the current resulting from an externally applied voltage.

3.1 Motor model

A motor can be simply modelled by the resistance of its coil of wire and a voltage source that produces a voltage proportional to rotational speed (see Fig. 3). The "generator" voltage
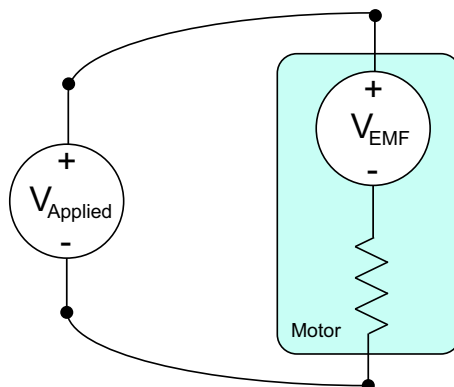
Figure 3: Simple motor model. The simplest motor model is a resistor (corresponding to the resistance of the motor's wire coil), and a voltage source corresponding to the Back EMF.

is known as the Electro-Motive Force (EMF). It's often called the "Back EMF", since it's the voltage coming back out of the motor.

When we interface with a motor, we typically control the *voltage* applied to the motor. This applied voltage can cause current to flow into the motor. At any given moment, Ohm's law can be applied. However, the voltage that we have to consider isn't the applied voltage, but the *net* voltage: $V_{applied} - V_{emf}$. In other words, the current can be written:

$$I = (V_{applied} - V_{emf})/R_{winding} \tag{15}$$

The current of a motor is extremely important: it is proportional to the *torque*. A motor only does work when its producing torque, and since torque is proportional to current, this only occurs when $V_{applied} \neq V_{emf}$.

When a voltage is first applied to a motor, $V_{applied} - V_{emf} > 0$, and so current will flow. This will produce torque and the output shaft will begin to rotate (provided the torque exceeds the static friction of the load.) As time goes on, the motor will slowly accelerate. As the motor gains speed, V_{emf} increases, and the current drops. At some point, we'll reach an equilibrium, where the motor's torque (and current) exactly matches the load.

Here's an important thought experiment: suppose we take a motor and short the two leads together. Will it be easier or harder to turn the output shaft? If we turn the output shaft, the motor will act as a generator, producing a voltage of V_{emf}. But the applied voltage is zero, since the leads are shorted. All of the back EMF voltage is applied across the motor winding, inducing a current. This current will generate a torque that will *oppose* the external torque. Setting both terminals of a motor to the same voltage *brakes* the motor, without the need for any mechanical mechanism.

3.2 Applied Voltage and PWM

As we've seen, the applied voltage is intimately involved in the behavior of a motor. Special circuitry is required to drive motors; a typical microcontroller can only provide a very small amount of power on each of its outputs– far too little to power a motor. We'll need some additional circuitry to drive the motors.

It turns out that building a circuit that can output an arbitrary applied voltage at high current is quite difficult. It is much easier to build a simple high current switch, whose output is either V_{max} or zero. In fact, we can build a switch that can change state very fast, upwards of 100kHz.

Suppose you're trying to build a dimmable light switch; can you build one from just an on-off switch? Suppose you start flipping the light on and off repeatedly, at first fairly slowly. The room is bright, then dark, then bright again. Not a very good dimmer! But you can probably imagine that if you could flip the switch fast enough, that you *could* achieve intermediate brightness levels. This is what we're going to do with motors.

If the switch is "on" 50% of the time, you might expect the motor to run about the same as it does at 50% the voltage. This does, in fact, work... provided that the switching is fast enough. The criteria for the switching rate is that the voltage has to change faster than the motor can react; the motor has an "electrical time constant", which reflects how rapidly it can react to changing electrical conditions.
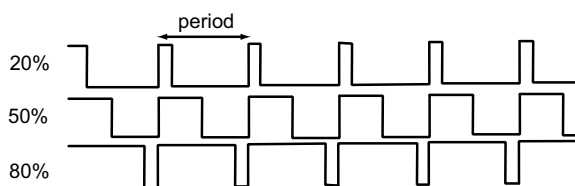

Figure 4: PWM Waveforms. We can approximate different applied voltages by using a pulse-width-modulated signal whose period is less than the motor's electrical time constant.

This technique is called Pulse Width Modulation (PWM) (see Fig. 4), since the width of each pulse is modulated in order to approximate different values of $V_{applied}$.

In general, motor electrical time constants are on the order of a few milliseconds, but PWM frequencies in the ¡20kHz range can cause audible vibrations. So often, PWM frequencies are increased well above the human range of hearing.

3.3   Deficiencies of our model

This is a pretty good model for a motor, but it ignores one often important detail: electrical transients. When the motor is in steady state (i.e., it's been rotating under a uniform load for a long time), our model is accurate. But when conditions are changing (the load is changing, or the applied voltage is changing), the model can be pretty lousy.

The main issue we've overlooked is the transient behavior of the motor. Not only is the coil of wire an electromagnet, it's also an *inductor*. Recall that the voltage across an inductor with inductance L is:

$$V = L\frac{dI}{dt} \tag{16}$$

If the current is changing quickly, the inductor can develop a very large voltage! Many motor controllers support a controllable "slew rate"; this is a way of limiting the rate at which the applied voltage changes (and hopefully the current as well!) Large voltage transients are not good for circuits, and large motors can develope enormous transients on the order of tens or hundreds of volts.

3.4   Summary

We'll continue to use our simple model for the rest of this article. Here's what you need to remember:

- The Back EMF of a motor (V_{emf}) is proportional to the speed of the motor.

- Motor current is proportional to torque.

- We drive motors with PWM, which allows us to approximately control $V_{applied}$.

4 Dead-reckoning Sensors

In order to build an odometry system, you must have a way of measuring the angular velocity of the motor. This can be accomplished in a number of ways:

- Mono Phase Encoders. A sensor is placed on the motor shaft or wheel such that when the shaft rotates, the circuitry generates alternating 1's and 0's. One way of implementing a simple encoder is by attaching a disc with holes to the shaft, and using a breakbeam sensor to detect when a hole passes by. A mono phase encoder can not determine the direction of motion. A serious failure mode occurs when the motor is essentially stationary with a hole half-way in front of the sensor: environmental noise can cause the encoder to trigger and untrigger, making it appear as though the shaft is rotating.
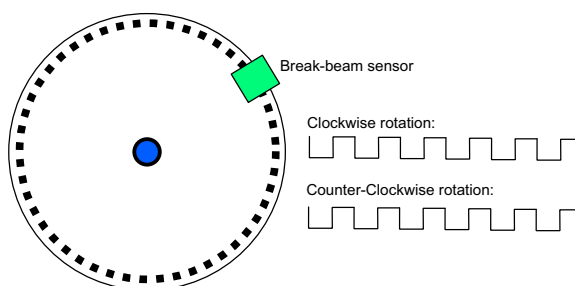

Figure 5: Simple Encoder. As a perforated disk rotates, a break-beam sensor alternates between 1 and 0. The direction of motion is ambiguous.

- Quadrature Phase Encoders. An improvement upon mono phase encoders, quadrature phase encoders use two simple encoders arranged so that they produce waveforms 90 degrees out-of-phase. When the shaft rotates in one direction, signal A will "lead" signal B; when rotating in the opposite direction, signal "B" will "lead" signal A. Quadrature phase encoders are generally considered the most precise form of motor feedback, and can be quite expensive. A simple state machine can decode the quadrature phase signals into "up" and "down" pulses. The two out-of-phase signals provide immunity from spurious transitions due to a hole being half-way in front of one of the sensors: only one signal can be transitioning at any given time, and if this signal jumps back and forth, it maps to alternating (and cancelling) forward and backward motion.

- Tachometer. A tachometer is essentially a small motor attached to the output shaft of the main motor. It operates as a generator, where the output voltage is proportional to the angular velocity of the main motor.

- Direct Back EMF measurement. With a little extra circuitry, we can measure the back EMF of a motor directly. We set $V_{applied} = 0$, wait for the electrical transients to die off, and then measure the voltage produced by the motor. Since we're not applying any power to the motor, the motor will tend to slow down while we're performing the measurement. Fortunately, it doesn't take long for electrical transients to die off; we can measure the back EMF quickly, and then restore power.
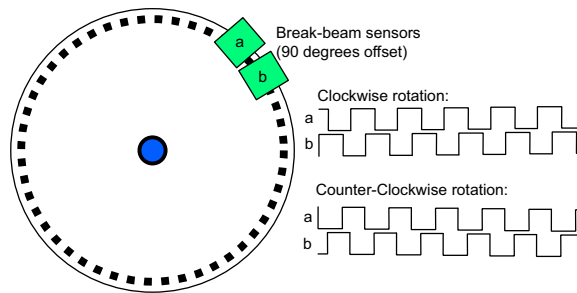
Figure 6: Quadrature Phase Encoder. Two simple encoders, arranged so that they pick up the same perforated holes, but 90 degrees out of phase with each other, allow the direction of rotation to be determined.

- Indirect Back EMF estimation via current sense. We can measure the amount of current flowing through the motor by putting a small-valued resistor in the path between the motor and ground. Ohm's law tells us that the voltage across that resistor will be proportional to the current.

  If we know the motor current, the applied voltage (which *we're* controlling), and the resistance of the motor winding, we can estimate the back EMF. Solving Eqn 15 for V_{emf}

$$V_{emf} = V_{applied} - IR_{winding} \tag{17}$$

- Gyro heading adjustment. Some vehicles have a gyroscope, which can provide an estimate of the vehicle's angular rate $d\theta/dt$. Gyros drift over time, but can be quite accurate over small periods of time. They can be used to improve dead reckoning accuracy, particularly when a poor motor feedback sensor is used (e.g., back EMF.)

5 Odometry in Practice

Odometry can provide good dead-reckoning over short distances, but error accumulates very rapidly. At every step, we inject error (noise) into not just x and y, but also θ. The error in θ is the killer, since every error in θ will be amplified by future iterations.

There are a number of basic noise sources.

- Sensor error. If we're using anything other than quadrature phase encoders for our feedback, then our estimates of the d_{left} and d_{right} will be noisy.

- Slippage. When the robot turns a corner, one wheel will, in general, slip a little bit. This means that even if the odometry data was perfect, the robot's path would not recoverable from it.

- Error in estimate of $d_{baseline}$ or in wheel diameters. We must measure the baseline of the robot and the scale factor relating angular rate and linear velocity (a function of the wheel diameter.) Even small errors in these constants can cause problems– typically, the odometry result will have a consistent veer in one direction or the other. In fact, often the wheels on the robot are slightly different sizes. Even small differences lead to large navigation errors with time.
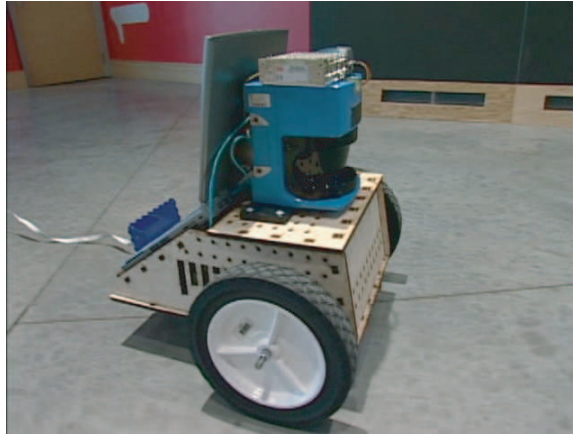
Figure 7: Splinter. This is the robot that collected the dataset. The vehicle, known as "Splinter", has high quality quadrature phase encoders on each drive motor. A laptop on the back of the vehicle logged the data. The large blue device in the front of the vehicle is a laser range finder, not used in this article.

You should be itching to try out odometry now, and fortunately, there's a dataset waiting for you! In this section, we'll walk through a few examples in Matlab.

The dataset consists of 200 meters of odometry and gyro data from an experiment that lasted about 10 minutes. The vehicle that collected the data is shown in Fig. 7. The vehicle left the lab, travelled around a large open space in a clockwise (as viewed from the top) direction three times, then reversed direction and travelled around the same loop once in the opposite direction. Finally, it returned to lab.

A good odometry result is shown in Fig. 8. While the robot started and stopped in (approximately) the same space, notice how the odometry estimate is quite different at the beginning and end. A small error in θ causes all subsequent navigation to be rotated out of alignment.

The dataset is available from /mit/6.186/2005/doc/odomtutorial. Copy it into a directory and start matlab. The data is contained in a single array called "data". Each row in the dataset is a set of observations made at a single point in time. A helper script, "setup.m" is provided which loads the dataset and sets up a few constants. These constants are TIME, LEFTTICKS, RIGHTTICKS, LEFTCURRENT, RIGHTCURRENT, LEFTPWM, RIGHTPWM, and GYRO, and correspond to the column numbers in which data of different types is stored. To get started, type:

```
>> setup
```

Note that the odometry data is provided in units of *ticks*: the amount of forward (or backward) motion that occured since the last sample. Ticks are related to distance (and angular rate) by constant scale factors, which are characteristics of the robot's geometry. We mentioned before that the *baseline*, the distance between the left and right wheels, is also a characteristic of the robot.

These measurements are made using a ruler, and are inexact. One of our tasks will be to find better values of the parameters.

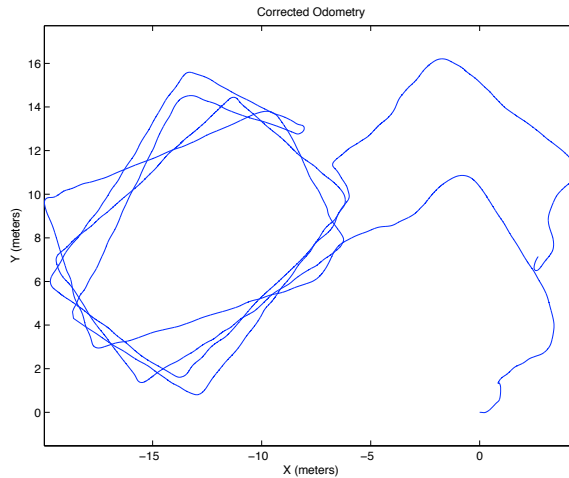| Parameter	Value
baseline	0.45 meters
metersPerTick	0.000045 meters

9

Figure 8: Good odometry result. This result was obtained by manually optimizing the rigid parameters of the vehicle until the path best resembled the actual trajectory. Note that the trajectory, in fact, started and ended at approximately the same position; even good odometry results can contain significant errors.

Problem 1. Examining the raw odometry data

Let's get familiar with the data by plotting the left versus right wheel odometry ticks; we'll be able to see how fast the left and right wheels are moving.

```
>> clf;
>> hold on;
>> plot(data(:,LEFTTICKS),'r');
>> plot(data(:,RIGHTTICKS),'b');
```

The command "clf" clears the figure, "hold on" allows multiple plots to be displayed on top of each other, and the two plot commands plot the vector of data in either red ('r') or blue('b').

Compare the instantaneous velocity of the left and right wheels to the odometry results in Fig. 8. Identify in the velocity plot where the robot was turning.

Problem 2. Plotting the gyro data

Now let's take a look at the gyro's reported angular position over time. Plot the gyro data over time (the commands should look much like a previous example.) Your figure should look something like Fig. 9.

The shape of this graph is characteristic of a robot path which was mostly piecewise-linear; the robot goes straight for a while (the graph is horizontal), turns (the graph is vertical), then goes straight again.

Gyros tend to drift (a roughly uniform linear deviation over time.) We try to calibrate our gyros as carefully as we can so that the drift is small, but often our final data still contains drift. Look at the horizontal segments of the graph: they aren't quite horizontal. Compute a better estimate of the gyro by finding a drift rate which makes the graph look more like Fig. 9.
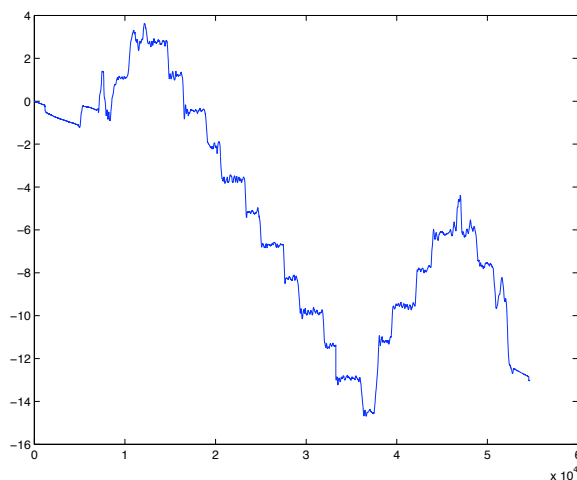
You can do this with code like this:

Figure 9: Adjusted gyro data. The robot traveled in straight line segments, punctuated by (roughly) 90 degree turns. This is readily evident from the gyro data.

```
>> gyro = unwrap(data(:,GYRO));
>> driftrate = <fill in this value>;
>> gyrocorrected = gyro + data(:,TIME)*driftrate;
```

Make sure you understand how this code works. Try it with and without the "unwrap" command. Obviously, you'll need to experiment with different values of driftrate. Dealing with 2π wrap-around is tricky, and *many* bugs can be introduced by neglecting it!

A fancy way of doing the calibration is to find some metric that "measures" the goodness of the drift rate. Suppose you assume that the starting orientation and end orientation were different by $\pi/2$; could you compute the correct drift rate from this?

Problem 3. Basic odometry

Your next task is to replicate the odometry result in Fig. 8. This is the most "key" problem in this whole tutorial, so don't skip it!

Use the equations derived in this section to compute the *change* in θ at each time step. You can integrate this by using the *cumsum* command. You'll now have a vector of the value of θ for every time step which you can use to compute the change in x and y. Using *cumsum* again, compute the actual x and y position. Plot the result.

Your first attempt will look like Fig. 10. These results are not terribly good, but that's largely because the metersPerTick and baseline parameters were only estimates. Try to improve the odometry result by adjusting the parameters. Remember that metersPerTick might actually be different for the left and right wheel!

It may seem a bit tedious to optimize the parameters, and to a degree, it is. However, it is possible (and desirable) to develop an intuition for how the parameters affect the odometry result. You should be able to look at a result and say to yourself, "Hmm, I think the baseline distance is too short."

When optimizing parameters by hand, we often use the *shape* of the robot path as our metric, mostly ignoring the scale. Can multiple sets of values for the baseline and metersPerTick result in the same path shape?

You should also be aware of overfitting. The more "knobs" we have to adjust the size and shape of our path, the more likely we are to be able to coax it into a shape we want–
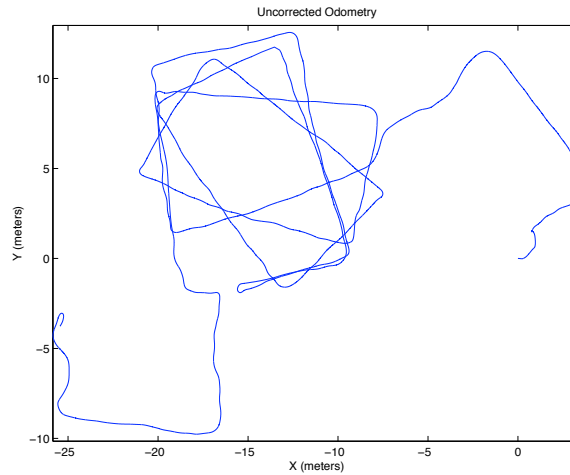
Figure 10: Uncorrected Odometry. Your first attempt at odometry should match this figure.

regardless of what the data actually tells us. Our goal is to get the best *general purpose* values of our parameters– ones that will be useful on future navigation missions, not get the absolute best result on a single, controlled (and perhaps contrived!) experiment. Remember that on any actual experimental run, it is much more likely that there was some error, than that there was no error.

Even with only three parameters (leftMetersPerTick, rightMetersPerTick, and baseline) it is quite possible to overfit, particularly when the dataset is relatively short. A general strategy for dealing with overfitting is to use cross-validation: fit parameters to a subset of the data, and test it on a disjoint subset of the data. We can't readily do that here, since we only have one relatively short dataset.

Problem 4. Gyro-assisted odometry

Suppose your robot is a tricycle, with only the center wheel having an optical encoder. You no longer have d_{left} and d_{right}– you only have d_{center}. This means that you cannot compute changes in θ.

If you have a gyro, you could use the gyro's estimate of heading instead! That is your challenge in this problem. Using the gyro data that you adjusted in Problem 1, recompute the odometry using only d_{center}. Your graph should look similar to Fig. 11.

Is the path better or worse?

If you have d_{left}, d_{right}, and gyro data, how could you fuse the two independent estimates of θ to get a better value of θ? If you're interested in this, you should read about Kalman filters. I recommend Chapter 1 of Peter Maybeck's book, "Stochastic Models, Estimation, and Control"; a PDF is available on the MASLab website.

Problem 5. Estimating velocity with back EMF

In the last problem, we assumed we didn't have complete odometry data. Now we're going to assume we don't have *any*!

The dataset contains a log of the PWM values given to the motor controller, as well as the current consumption. A PWM value of -255 corresponds to about -12V, while +255 corresponds to +12V. The current consumption is given in amps, and the motor winding has a resistance of 3.5 Ohms.
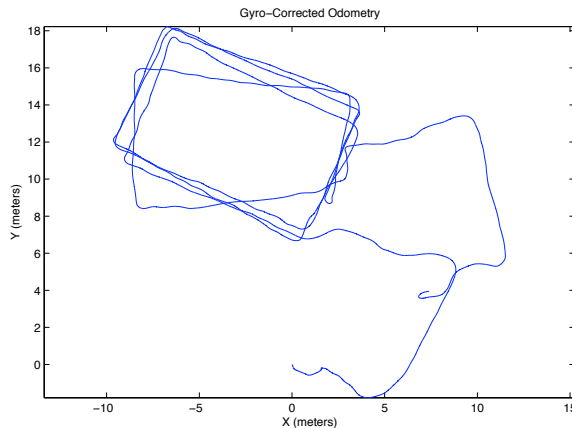
Figure 11: Gyro-derived heading odometry. The (drift-corrected) gyro was used exclusively for heading. You should replicate these results, and try to improve them further by *fusing* the odometry-derived heading and the gyro-derived heading.

Using Eqn. 17, estimate V_{emf} of the motor from these parameters. We know that V_{emf} is proportional to angular velocity, but what is the scale factor? We could look it up on the motor's datasheet, but it's probably not terribly accurate anyway.

Let's consider a single wheel. We have it's V_{emf} and its actual velocity as measured by the quadrature phase encoders. To estimate velocity, you'll need to consider the amount of time that elapsed during every time step. You can compute this using:

```
>> dtime = [ 0; diff(data(:,TIME))]
```

One of the irritating things about working with real systems is that data doesn't always cooperate. It's entirely possible (in fact, *likely*) that you may encounter intervals in which zero time elapsed. When you compute the velocity during this interval, you must be careful to avoid a divide-by-zero! A good way of dealing with this is to estimate the velocity by looking at a window spanning several time samples. This can also serve to filter out some of the noise!

We know that V_{emf} and velocity should be proportional to each other. Time for an upsetting fact: there's probably some nonlinear offset as well. In other words, let's model the system as:

$$velocity = x_1 * V_{emf} + x_2 \tag{18}$$

We want to solve for x_1 and x_2, and we know the other two quantities.

This is (hopefully) true at every timestep, so let's write this in matrix form. Let B be the matrix of actual velocities, and let A be a matrix with V_{emf} in the first column and 1s in the second column (in matlabeese, this is A=[vemf ones(size(vemf))]). We can then write:

$$Ax = B \tag{19}$$

This of course is the most basic of all Linear Algebra problems. Remember that this system is hugely overdetermined: we have *thousands* of simultaneous equations, and only *two* unknown variables. This is a good opportunity to compute a least-squares fit. (For a reminder of the mathematics, see Strang's 18.06 textbook, "Introduction to Linear Algebra".)

$$x = (A^T A)^{-1} A^T B \qquad (20)$$

We can now estimate the velocity of the motor using our coefficients x_1 and x_2, and compare it to our measured values using odometry. For the left wheel, you should get a graph like that in Fig. 17.
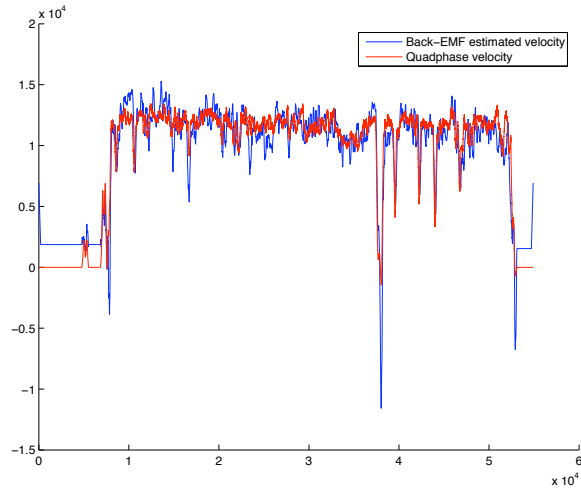

Figure 12: Estimating velocity from Current Sense. Using Eqn. 17, we can estimate back-emf (and thus velocity) using the motor's current and applied voltage.

Now let's put it all together. Estimate the velocity for the left and right wheels, and compute a trajectory without using the quadrature phase data (except for calibrating your models.)

How does it look? When does it perform well/poorly? How does it look if you use the gyro data for your θ estimate? You should be able to achieve something like Fig. 13. Keep in mind that this approach doesn't require any optical encoders at all!
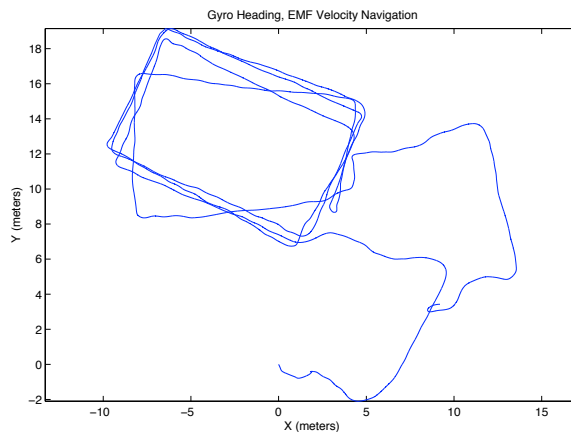

Figure 13: Navigation from gyro and EMF-based velocity.

6   Conclusion

You now have a pretty good idea about what odometry is, what it can do, and what it can't. In Maslab, you won't have optical encoders as good as those used in this problem. But you will have a gyro, current sense, and the ability to build your own quadrature phase optical encoders.

What approaches do you think will work? Is there some other way of navigating? Maslab robots have cameras. If you pick out a feature and follow it around, can you use that to compute changes in robot orientation? If you find a feature of known size, can you estimate how far away you are?

You can purchase other sensors too– perhaps an electronic compass. Inexpensive electronic compasses might be low resolution, but they won't drift like a gyro. Can an electronic compass plus a gyro be a powerful navigation aid?

If you have any comments or corrections on this tutorial, please send me an email at eolson@mit.edu.