A Better Dead-Reckoning System

Spencer Fishman
spencer@hurricanedevelopment.com

Derek Huang
17huangd@millburn.org

Brian Jimenez
brianjimenez869@gmail.com

Emily Liu
emliu@ctemc.org

Sudharshan Venkatesh
venkatesh.sudharshan@gmail.com

22 July 2016
The New Jersey Governor’s School of Engineering and Technology

Abstract

Dead-reckoning is a self-contained method of navigation that uses on-board sensors such as rotary encoders, accelerometers, and gyroscopes to estimate the position of a system. However, dead-reckoning traditionally does not take into account errors that accumulate in measurements, and dead-reckoning calculations can become extremely inaccurate very quickly. The problem is compounded when on-board system sensors are noisy, as the noise makes it difficult to determine the true state of the system, and when the system’s propulsion is inaccurate, where large drift results even when the system should be traveling in a straight line.

Through experimentation with a robot constructed with inexpensive materials, the implementation of a proportional-integral-derivative (PID) controller and various sensor data filters rectify hardware deficiencies enough to confirm the viability of dead-reckoning as a method of navigation. Filtering methods such as the Kalman filter and the particle filter serve to significantly decrease the noise level in sensor readings and deliver more consistent sensor readings to the PID control loop, which mostly eliminates drift and error during the robot’s movement.

Even with relatively inexpensive and inaccurate hardware, it is clear that a well-implemented dead-reckoning system will still function at an acceptable level of precision. For those unable to use GPS or celestial cues for navigation, dead-reckoning can provide an inexpensive, relatively accurate, and reliable alternative.

1 Introduction

In navigation, external aids are not always available. In such cases, dead-reckoning offers a solution. Dead-reckoning is the determination of one’s relative position using only system sensors; a pure dead-reckoning system does not use out-of-system assistants such as GPS or celestial bearings [1]. Dead-reckoning as a navigation component can be used to approximate the location of a vehicle, whether air, land, or sea. This type of system is useful when there are no external sources to provide positional data, such as when there is no GPS signal or when the clouds are obscuring the stars.

Today, dead-reckoning is most often accomplished through a mechanical or computerized system, with positional data estimated using values read from sensors such as gyroscopes, accelerometers, or rotary encoders. Most dead-reckoning systems today include a full inertial navigation system, which combines gyroscopes and accelerometers with a guidance computer, possibly even with electronic or optical sensing units [4]. Through the use of a dead-reckoning system achievable on a compact robot with gyroscope, accelerometer, and rotary encoders, the robot gains access to an accurate mobility system without relying on GPS.

Before the advent of GPS, dead-reckoning, short for “deduced-reckoning,” was primarily used in nautical navigation. It was helpful in determining sunrise and sunset, predicting landfall, and conducive to predicting the future path of the ship and avoiding hazardous obstacles and otherwise dangerous areas.

Current examples of dead-reckoning systems in use are found in older GPS units, which are still
able to reliably navigate for a few miles using an on-board gyroscope when no GPS signal is available, and in inertial navigation units themselves, which are inherently capable of dead-reckoning and are used in general aviation, guided munitions, and commercial and military navigation. The greatest advantage and disadvantage that a dead-reckoning system has over a guidance system relying on external positional data is that it is completely self-contained, which makes it a viable method for determining position in conditions where GPS or celestial cues are unavailable but also makes it liable to additional, cumulative error. However, because the process of dead-reckoning estimates one’s position through the extrapolation of prior knowledge and sensor data, the dead-reckoning position may only approximate the true position [6]. Further, the estimated position becomes increasingly inaccurate as errors, both from sensor input and calculations, accumulate. Fortunately, some algorithms have been developed for the purpose of minimizing this error, some of which have proved practical and accurate for navigation purposes.

Thus, with a reliable dead-reckoning system, applications for which relative position is needed but access to GPS and celestial cues are unavailable or unattainable become possible [5]. This enables further integration and the advancement of modern day technologies in non-optimal conditions.

2 Background

2.1 PID

2.1.1 PID Control

One method to control output error is the three-term PID (proportional-integral-derivative) controller, a control feedback loop first designed in 1911 by Elmer Sperry, on which Nicholas Minorsky would publish a theoretical analysis on in 1922 based upon his observations of a helmsman’s actions when steering a ship [7]. As illustrated in Figure 1, the three controls that the PID controller takes into account are in its name: proportional, integral, and derivative control.

Proportional control is the most straightforward to understand; again using a nautical example, proportional (P) control entails adjusting the ship’s steering back towards the desired course in proportion to how far off course the ship currently is. The further off course the ship is, the more a helmsman will turn the wheel, and vice versa.

However, proportional control is not always enough to combat a accumulating error, known as integral error. A nautical example of such error would be long-term drift that occurs as a helmsman is attempting to steer back on course. Even as the helmsman is steering the wheel proportionally with regards to how far off course the ship currently is, integral error in the form of long-term drift, such as a persisting lateral current, will still throw the ship far off course. Therefore, the integral (I) control in a PID controller will adjust how the controller is settling, limiting large inaccuracies resulting from the development of cumulative error.

The last control that the PID controller implements is derivative (D) control, which takes into account the rate of change in error. Again using a nautical example, the error that results as the ship travels along may increase or decrease in magnitude, and by determining the rate of change of the error, it is possible to approximately predict the future error behavior of the ship and use this knowledge to potentially improve the stability of the ship’s path. However, since an idealized derivative is not causal, i.e. independent of past and current inputs and therefore impossible to discretize, PID controllers often have a low-pass filter for the derivative term, which limits higher frequency gain and noise [8].

With \( u(t) \) defined as controller output, equivalent to the manipulated variable \( MV(t) \), the PID control algorithm is:

\[
  u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de(t)}{dt} \tag{1}
\]
Here $K_p$ is the proportional gain of the controller, $K_i$ is the integral gain, and $K_d$ is the derivative gain, with all three gains being constants that must be tuned in order to generate the desired control response. $e(t)$ is a continuous function representing process error, and can be calculated discretely by subtracting a measured process variable output, $PV(t)$, from an ideal output value $SP$, or set point, so that $e_t = SP - PV_t$. The variable of integration is $\tau$, which takes on time variables from 0 to a current instantaneous time $t$, and the expression $\frac{de(t)}{dt}$ is the instantaneous current rate of change of the error $e(t)$.

The informal representation of the equation is:

$$u(t) = P_{out} + I_{out} + D_{out} \quad (2)$$

This form clearly demonstrates the controller output as the sum of the proportional term $P_{out}$, the integral term $I_{out}$, and the derivative term $D_{out}$.

The proportional term, as explained earlier, produces an output proportional to the current error $e(t)$, with the magnitude of the output and therefore the sensitivity of the response adjusted by the proportional gain constant, $K_p$.

Reconciling the two representations of the PID equation, the proportional term in (2) is illustrated as:

$$P_{out} = K_p e(t) \quad (3)$$

Therefore, the higher the proportional gain $K_p$, the greater the change in the controller’s output for a given error, and the more responsive or sensitive the controller will be. However, if the proportional gain is too high, such that the controller consistently overshoots the desired set point $SP$, the system may become unstable, as the controller’s output will consistently oscillate above and below the set point.

Conversely, lowering the proportional gain will decrease the controller’s output and responsiveness for a given error. However, if the gain is too low, the controller’s response to disturbances will be too small to be effective, and the system will accrue error.

The integral term’s value, in contrast to the value of the proportional term, depends not just on how large the present error $e(t)$ is but also on the duration of the current error and previous errors. A sum of the accumulated instantaneous error over the duration that the controller has been active, the integral term delivers correction for accumulated errors that should have been corrected at previous times. Like the proportional term $P_{out}$, the value of $I_{out}$ is adjusted by a gain constant, in this case $K_i$, the integral gain constant.

Here the integral term in (2) is represented as:

$$I_{out} = K_i \int_0^t e(\tau)d\tau \quad (4)$$

The integral term decreases the time for the desired set point $SP$ to be reached and seeks to counter the effects of cumulative error that result from the use of a proportional, or P controller. However, because the integral term attempts to correct past accumulated errors, it can also result in the controller delivering output that overshoots the desired set point $SP$. If the integral gain $K_i$ is set too high, the effect of the integral term will also lead to the oscillation of the controller’s output, increasing in severity until the controller’s output oscillates between its maximum and minimum values. This behavior effectively turns the PID controller into a “bang-bang,” or hysteresis controller, which quickly switches back and forth between two extreme and opposite outputs.

The derivative term is based on error’s rate of change and is a preemptive form of control that seeks to predict the next state of the system. Determined by the instantaneous slope of the error over time, the process error derivative is then multiplied by a gain constant, the derivative gain constant $K_d$, to adjust the value of the derivative term $D_{out}$ and its effect on the overall output of the controller $u(t)$.

The derivative term in (2) is represented as:

$$D_{out} = K_d \frac{de(t)}{dt} \quad (5)$$

By predicting system behavior based on the current rate of change of the error, derivative action is able to improve the stability of the system and decrease the time spent before reaching stability. However, if the derivative gain $K_d$ is set too high, the controller’s output will exhibit oscillatory behavior, reducing the stability of the system.

### 2.1.2 PID Tuning

In order for the PID controller to function optimally, it must be tuned, which involves adjusting
the values of $K_p$, $K_i$, and $K_d$. These values vary based on the specifics of each system, and if input values are scaled up or down due to programming in a new library, the constants have to be scaled up or down accordingly. Manual tuning is generally very slow and tedious, and without simulation software such as PIDEasy to model PID controller behaviors, manual tuning becomes completely empirical and an arduous and imprecise chore.

However, there is a well known method of tuning that uses the Ziegler-Nichols method, which uses two parameters based on measurements. Following this method, the integral and derivative gains $K_i$ and $K_d$ are both set to zero, with the proportional gain $K_p$ being increased until it reaches the ultimate gain $K_u$, where the output of the controller begins to oscillate in a stable and consistent manner. The period of oscillation, $T_u$, is then used to set the integral and derivative gains depending on the type of controller that is being implemented.

Figure 2.1.2 is a table displaying several methods of PID tuning following the Ziegler-Nichols method or other methods to produce specialized controller behaviors.

<table>
<thead>
<tr>
<th>Rule Name</th>
<th>$K_p$</th>
<th>$K_i$</th>
<th>$K_d$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classic Ziegler-Nichols</td>
<td>$0.6 \times K_u$</td>
<td>$0.5 \times T_u$</td>
<td>$0.125 \times T_u$</td>
</tr>
<tr>
<td>Pessen Integral Rule</td>
<td>$0.7 \times K_u$</td>
<td>$0.4 \times T_u$</td>
<td>$0.15 \times T_u$</td>
</tr>
<tr>
<td>Some Overshoot</td>
<td>$0.33 \times K_u$</td>
<td>$0.5 \times T_u$</td>
<td>$0.33 \times T_u$</td>
</tr>
<tr>
<td>No Overshoot</td>
<td>$0.2 \times K_u$</td>
<td>$0.5 \times T_u$</td>
<td>$0.33 \times T_u$</td>
</tr>
</tbody>
</table>

Figure 2: Parameters from Ziegler-Nichols Rule [9]

The Ziegler-Nichols method is designed to give PID controllers the best disturbance rejection and produces a controller with an aggressive gain and overshoot [9]. However, in some instances, it is more desirable to minimize or eliminate overshoot, and for these specialized scenarios, the Ziegler-Nichols method is not desirable.

There are variations from the original Ziegler-Nichols proportions, as some yield a smoother transition, while others provide better optimal disturbance rejection. The different rules and their respective proportions shown in Figure 2.1.2 reflect this tunable range [9].

However, PID constants can also be manually tuned. Ziegler-Nichols or other rules are not always practical to use in all cases, and in some cases manual tuning is the better option. Manual tuning involves setting all the constants to zero, then increasing $K_p$ in small increments until the output oscillates about the set point. Then $K_d$ is increased in small increments until this oscillation stops. These two steps are repeated until $K_p$ reaches a gain so high that no value of $K_d$ will stop the oscillation. Once this happens, $K_i$ is increased until steady-state error is as small as possible and the loops runs smoothly [12].

![Figure 3: Effects of independently increasing each gain](image)

### 2.1.3 Limitations of PID Control

Despite the versatility of the PID algorithm in its application to many control problems, as well as its ability to work decently even with very rough tuning, PID control can sometimes exhibit poor performance, and it is never perfect. Of course, due to the nature of the controller as a feedback loop with constant gain parameters, the PID controller knows nothing about real-time occurrences in the physical system. Therefore, control is always reactive, and some level of compromise is inherent.

As touched on before, one of the main difficulties regarding a PID controller is its tendency to overshoot or oscillate around its set point value, especially when the gains are too high. For example, such behavior can be found in amateur line-following robots that implement imperfectly and manually tuned PID control algorithms. These robots will constantly swing back and forth over the edge of the line they are following as their control algorithms constantly overshoots the set point of the light sensor value. The output to the motors exhibits an oscillatory pattern, as each overshooting of the set point is corrected with another overshooting of the set point, in the reverse direction. Such behavior is very similar to the actions of a line-following robot programmed with a much
simpler hysteresis control algorithm and provides suboptimal performance.

Another issue with PID controllers is a result of their linear and symmetric nature, i.e. the fact that they exhibit the same control behavior whether the process variable input $PV(t)$ is above or below a set point $SP$. In other words, given some magnitude $k$ of an error $e_t$ for a discrete system at a current time $t$, a PID algorithm will provide an equivalent proportional change in output magnitude regardless of whether $PV_t = SP + e_t$ or $PV_t = SP - e_t$, as presented in the Figure 4.

![Linear Symmetric System Response](image1)

**Figure 4:** System responds in the same way, regardless of state

When applied to a system with asymmetric response, where the response of a system to an applied control may be different depending on the direction that the controller is correcting, a standard PID controller will generate a large cumulative error as it attempts to apply symmetric control to a system that responds asymmetrically. An example of such an asymmetric system would be a water storage system that can be filled very quickly but can only drain very slowly. The response graph of the tank to a control algorithm that manages the flow of water in and out of the storage system to maintain a consistent water level inside the system would be similar to the graph in Figure 5 illustrating nonlinear asymmetric system response, where positive control elicits a steep response from the system as water can flow in quickly, but negative control elicits a slow response as the system drains water quite slowly.

![Nonlinear Asymmetric System Response](image2)

**Figure 5:** System responds more if positive control is applied, but responds much less to negative control

In this scenario, a control system implementing PID would not be a good choice. PID control would likely overshoot the set point water level due to the fast flow of water into the system, while being slow to correct the overshoot; as the flow of water is slower out of the system, the PID control would cause a significantly slower negative response by the water storage system. Thus, using PID control in this situation would cause the storage to have a higher water level than is desired most of the time.

Though it is possible to tune a PID control scheme for an asymmetric system by severely damping its output in order to reduce or even prevent overshoot, doing so would severely affect the performance of the control loop. Such a damped PID controller would cause a sluggish reaction and be extremely slow in dealing with even moderate error introduced into the system.

### 2.2 Noise and Filtering

#### 2.2.1 Signal Noise

As touched on earlier, a dead-reckoning system requires data input from on-board sensors that can deliver the required initial data to the system and then calculate an approximate estimated position, $EP$. In an ideal world, a sensor of any kind, whether an accelerometer, gyroscope, light sensor, or radar receiver, should read exactly the value that should be currently measured by the sensor. In
other words, if a free body is accelerating at 0.5 m/s² in the x direction only, an ideal accelerometer would read only that value exactly for any time t that the free body is accelerating in such a manner.

Unfortunately, this is never true in practice. No matter how accurate a sensor is purported to be, there are always factors that will cause measurement inaccuracies. Sometimes it is a fault in the sensor itself, but often times, signal variation from the sensor is often due to electronic noise such as Johnson-Nyquist and shot noise, small variations in the sensor’s measurement of an experimental value, and other natural sources of noise. Johnson-Nyquist noise is unavoidable, caused by random thermal motion of charge carriers, typically electrons, in an electrical conductor; shot noise is similar, arising from the quantized nature of electron flow. These two types of electrical noise affect signal frequencies on a wide band, and along with other sources of noise such as small vibrations, give unwanted variations in sensor readings [11].

For a full PID controller, signal noise is highly undesirable, as it causes the system to exhibit oscillatory and jerky behavior. This behavior stems from the derivative term in a full PID; when there is significant signal noise and constant variations in the rate of change of error, the derivative term will amplify the effects of the signal noise and cause large fluctuations in the controller output. This unwanted amplification of signal noise by the derivative term can be attenuated by using a low-pass filter to reduce noise at higher frequencies, which create the greatest fluctuations in output, but low-pass filtering can cancel out derivative control, effectively turning a PID controller into a PI controller [8]. Additionally, since sensor values are being read across a wide range, filters with cutoff frequencies, like the high- and low-pass filters, may not be suitable for use.

2.2.2 Kalman Filtering

The approach is to reduce overall sensor output noise in real time and try to estimate close to the true value that is being measured. One such method of reducing overall sensor noise is to implement Kalman filtering, also known as linear quadratic estimation, a recursive algorithm that produces successive estimates of the true measured value of a system corrupted by white noise [13]. The filter is most commonly used as part of the control of dynamic and complex systems, such as aircraft, ships, and continuous manufacturing processes [13].

In order to apply control to one of these complex systems, it is imperative to first understand the system status, which is usually accomplished by reading data in from sensors embedded in the system. Again, there is always persistent noise from the measured data, and the Kalman filter provides a method for inferring the required data from noisy measurements. By using a system’s state transition model, known control input to the system, and an iterative measurement process of data (typically received from various sensors), the Kalman filter can then form an estimate of the system’s state that is far more accurate than by estimating off of any single measurement at some time t in the system.

The mathematical model of the Kalman filter assumes a multidimensional model of the filter, which is shown below:

\[
x_k = F_k x_{k-1} + B_k u_k + w_k
\]  

In the equation above, the term \(x_k\) is the predicted true state of the system, and is the output of the filter. The term \(F_k\) is the state transition model of the system, which models the changes between each \(x_k\) term and for a current term \(x_k\) is multiplied by \(x_{k-1}\). \(B_k\) is a known control-input model which is multiplied by the control vector \(u_k\), which represents the control input into the system at the current time \(t_k\). \(w_k\) is the process noise at the time \(t_k\) that has been introduced into the system. In this generalized model of the Kalman filter, \(w_k\) is assumed to be part of a multivariate Gaussian, or normal, distribution with a zero mean and a covariance of \(Q_k\), and would be represented as such:

\[
w_k \sim \mathcal{N}(0, Q_k)
\]  

At the same time \(t_k\), an observation \(z_k\) made is modeled in the form below:

\[
z_k = H_k x_k + v_k
\]  

In the observation model, the term \(H_k\) is the observation model of the system which relates the true state of the system \(x_k\), with an added observation noise \(v_k\), which like the process noise \(w_k\) is also a zero mean multivariate Gaussian distribution with a covariance of \(R_k\), represented as such:
\( v_k \sim N(0, R_k) \)  \hspace{1cm} (9)

An important point to keep in mind is that all noise vectors, including \( w_k \) and \( v_k \), are assumed to be mutually independent, i.e. none of the noise values influences the value of any other noise in the Kalman model. Equally important is the simple fact that most dynamic systems in real life will not exactly fit the theoretical Kalman model of the linear dynamic system.

In its essence, the Kalman filter creates an average prediction of the particular state of a system by reporting a weighted average of each new measurement with the previous state values. Weighting allows values to have more or less influence on the filter’s final output, depending on their estimated uncertainty. In other words, values that are estimated to be less uncertain carry more weight, and the converse is also true. The weighting is determined from the calculation of covariance, a measure of how much two random variables change together; the resulting weighted average is a new state estimate in between the predicted and measured state, and it is therefore less uncertain than any state reported from either value alone. This process is then iterated for each discrete time interval, with each new estimate and covariance influencing the next iteration, with the filter’s output arriving closer and closer to the true measured value. An extra benefit of this iterative process is that the Kalman filter only requires the last set of data from the previous iteration, and therefore can be implemented recursively.

However, the Kalman filter’s behavior is not monolithic in its every implementation. The Kalman gain, which is calculated in every iteration of the filtering process and is therefore not constant, is a product of the current state estimate and the relative certainty of the current measurements and can therefore be tuned to change the behavior of the Kalman filter. The higher the gain, the closer the filter adheres to the measurement data, while the lower the gain, the more the filter follows the system’s state transition model. Using a lower gain allows the filter to reduce noise, but it also reduces how responsive the filter is when measurements begin to change overall. A maximum gain of one makes the filter ignore the state transition model, while a minimum gain of 0 makes the filter ignore all measurements.

Following the established model, the theoretical implementation of the Kalman filter can be generalized into two major steps, a predict step and an update step. Mathematically, the steps are represented below, with notation \( \hat{x}_{n|m} \) representing the estimation of \( x \) at a time \( n \) with observations up to and including time \( m \), where \( m \leq n \):

**Predict**

\[
\hat{x}_{k|k-1} = F_k \hat{x}_{k-1|k-1} + B_k u_k \quad (10)
\]

\[
P_{k|k-1} = F_k P_{k-1|k-1} F_k^T + Q_k \quad (11)
\]

**Update**

\[
\hat{y}_k = z_k - H_k \hat{x}_{k|k-1} \quad (12)
\]

\[
S_k = H_k P_{k|k-1} H_k^T + R_k \quad (13)
\]

\[
K_k = P_{k|k-1} H_k^T S_k^{-1} \quad (14)
\]

\[
\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k \hat{y}_k \quad (15)
\]

\[
P_{k|k} = (I - K_k H_k) P_{k|k-1} \quad (16)
\]

Here the term \( \hat{x}_{k|k-1} \) represents the current predicted state of the system \( x_k \) given all previous observations up to and including time \( k-1 \), a product of the state transitional model \( F_k \) and the previous predicted value \( \hat{x}_{k-1|k-1} \) given all previous observations up to and including time \( k-1 \), and the control vector \( u_k \). The current predicted covariance \( P_{k|k-1} \) is multiplied by the transposed state transition model \( F_k \), added to the process noise covariance \( Q_k \).

The term \( \hat{y}_k \) is innovation, the difference between the observation \( z_k \) and the product of the observation model \( H_k \) and the current predicted state of the system \( \hat{x}_{k|k-1} \), given all previous observations up to and including time \( k-1 \). In other words, \( \hat{y}_k \) is the error between the observed state of the system and the past predicted state of the system. The term \( S_k \) is the innovation covariance, the
product of the observation model \( \mathbf{H}_k \), the predicted covariance \( \mathbf{P}_{k|k-1} \) given all previous observations up to and including time \( k - 1 \), and the transposed observational model \( \mathbf{H}^T_k \), added to the observation noise covariance \( \mathbf{R}_k \). \( \mathbf{K}_k \) is the Kalman gain, the product of the predicted covariance \( \mathbf{P}_{k|k-1} \) given all previous observations up to and including time \( k - 1 \), the transposed observation model \( \mathbf{H}^T_k \), and the inverse of the innovation covariance \( \mathbf{S}_k \), \( \mathbf{S}_k^{-1} \).

\( \hat{\mathbf{x}}_{k|k} \) is the updated predicted state of the system given all observations up to and including the current time \( k \), and is the sum of the predicted state of the system \( \hat{\mathbf{x}}_{k|k-1} \) given all previous observations up to and including time \( k - 1 \) and the product of the calculated Kalman gain \( \mathbf{K}_k \) and the innovation \( \mathbf{y}_k \). Finally, the term \( \mathbf{P}_{k|k} \) is the updated predicted covariance and is the difference between the identity matrix \( \mathbf{I} \) and the product of the Kalman gain \( \mathbf{K}_k \) and the observation model \( \mathbf{H}_k \) multiplied by the predicted covariance \( \mathbf{P}_{k|k-1} \) given all previous observations up to and including time \( k - 1 \).

This generalized version of the Kalman filter assumes a multidimensional application of the filter with vectors represented as matrices. The prediction steps in the filtering process calculate a new estimated state of the system \( \hat{\mathbf{x}}_{k|k-1} \) and a new predicted covariance \( \mathbf{P}_{k|k-1} \) each iteration using values calculated from the previous iteration, taking into account noise and control input into the system as well as its state model \( \mathbf{F}_k \). Then, the update steps use \( \hat{\mathbf{x}}_{k|k-1} \) and \( \mathbf{P}_{k|k-1} \) calculated in the prediction steps to find the innovation \( \mathbf{y}_k \) and the innovation covariance \( \mathbf{S}_k \) in order to calculate the Kalman gain \( \mathbf{K}_k \) for the current iteration. Following, \( \hat{\mathbf{x}}_{k|k-1}, \mathbf{P}_{k|k-1}, \mathbf{y}_k, \mathbf{K}_k, \) and the observational model of the system \( \mathbf{H}_k \) are used to update the estimated state of the system to \( \hat{\mathbf{x}}_{k|k} \) and calculate a new predicted covariance \( \mathbf{P}_{k|k} \).

### 2.2.3 The Particle Filter

However, despite how straightforward the Kalman filter theoretically is to implement in discrete time, actual implementation runs into tricky mathematical complications involving multiple calculations of covariance and state matrices, which is very tedious. There is, however, an alternative to the Kalman filter: the particle filter.

The particle filter is a tool for tracking the state of a dynamic system modeled by a Bayesian network, in our case robot localization [15]. A Bayesian network is a graphical model of joint multivariate probability distributions, which are characterized by \( N \) random variables that are conditionally codependent [16]. The principle behind the particle filter is to use a complex model to get an approximate solution [15].

The robot’s state \( s \) is its position in the \( x \)- and \( y \)- directions and its angle \( \theta \). This can be depicted as \( s = (x, y, \theta) \). The particle filter is represented by a probability density function of a set of samples for possible values of each of the variables of the state. Each sample value set is considered a “particle,” from which the filter gets its name. The more particles with the same set of values, the more likely that this set of values is the robot’s current state [15].

The filter improves and narrows down the possible states of the robot by taking multiple posterior samples. Posterior probability is the probability of an event \( A \) occurring given that event \( B \) has already occurred, or \( P(A \mid B) \) [17]. This is calculated by Bayes’ Theorem [18]:

\[
P(A \mid B) = \frac{P(B \mid A) P(A)}{P(B)} \tag{17}
\]

Samples are taken, and based on the observations they are each assigned an “importance weight,” which factors into the probability of each respective particle. The total number of particles remains the same, so the particles are redistributed in the sample space based on their weights. Higher weighted particles are densely clustered together, which is statistically where the real state is more likely to be [15].

The algorithm starts with the initial belief state, \( P(s_0) \), which is the initial guess of the probability function. Typically, the position of the robot is initially unknown so the particles are evenly scattered throughout the possible state space. To narrow down the location, the algorithm loops through three steps: resample, update, and prediction [15].

The belief state \( P(s_t \mid d_{0\ldots t}) \) is calculated by evaluating

\[
\eta P(z_t \mid s_t) \int P(x_t \mid u_{t-1}, s_{t-1}) P(s_{t-1} \mid d_{0\ldots t-1}) dx_{t-1}
\]

\( P(s_t \mid d_{0\ldots t}) \) is the resample part of the loop, which is our belief on the variables of the state.
Here a new sample of particles is chosen to represent the weight of each. $P(z_t \mid s_t)$ is the update, which assigns importance weights to the particles based on the sample. This is multiplied by the normalization constant $\eta$, which ensures that all the weighted particles add up to 1 [19]. The prediction is $\int P(x_t \mid u_{t-1}, s_{t-1})P(s_{t-1} \mid d_{0 \ldots t-1})dx_{t-1}$, which takes each particle and adds a random sample to result in a new distribution of particles [15].

3 Procedure

3.1 Starting Tools and Materials

The basic robot provided is a skid-steer drive controlled by an Arduino-compatible SainSmart UNO R3 microcontroller, with an attached Adafruit v2.3 motor shield and Adafruit LCD shield.

Wired into the breadboard as well are a pair of VEX 2-wire 393 motors, a pair of VEX optical shaft encoders, and a Kootek GY-521 breakout board containing an InvenSense MPU-6050 six-axis combined accelerometer/gyroscope MEM. Power is provided by a mounted KMASHI 10000 mAh external battery pack. The robot’s wheels and frame are composed of standard VEX parts, and the robot’s overall dimensions are approximately 25 cm (w) by 20.5 cm (l) by 17.5 cm (h).
Software initially used during the programming and the development of the robot were the Arduino IDE, Git command-line tools, and the GitHub web-based Git repository hosting service. The Arduino IDE facilitated the development of C/C++ source code to be compiled for the SainSmart UNO, GitHub provided a way to create a centralized repository for storage of all relevant files and libraries, while Git command-line tools provided a way to update content in the GitHub repository and provide a means of version control.

3.2 Using the SainSmart UNO

3.2.1 First Steps

The SainSmart UNO is a less expensive alternative to the original Arduino UNO and shares the same layout and features. Programmable through the Arduino IDE, the SainSmart UNO performs comparably to the Arduino UNO and can be treated similarly to a genuine UNO.

After downloading and installing the necessary software and libraries, the team spent some time getting familiar with the Arduino IDE, the Adafruit software libraries that provided programming functionality for the motors and the LCD screen and leverage an existing, and Jeff Rowberg’s I2C device libraries, which provide an interface and methods for using I2C to communicate with the MPU-6050 and the VEX optical shaft encoders.

3.2.2 Writing a PID Controller

It was quickly established that for effective control of the robot’s travel, some kind of feedback control loop would need to be written. The VEX motors are standard DC motors, and since they contain no motor controllers, they are very inaccurate without good control software. Of course, there are many excellent and viable methods of control, but the team settled on using PID, a widely used control algorithm generally held in high regard.

The approach was to write a PID controller to adjust the power output of the two motors based on the readings taken from both the quadrature encoders and the MPU-6050 accelerometer and gyroscope. While the method as to how encoder and MPU values would be read was being determined, some team members went ahead to write a set of generic PID methods; the robot would require more than one implementation of PID, so the generic methods would prove useful in more than one respect.

3.2.3 Reading Values From the MPU-6050

After familiarization with the software libraries stated above, one of the first tasks to accomplish was to figure out how to read values from the MPU-6050 accelerometer/gyroscope. Lack of well-commented example code and obscurity of documentation led to certain confusion in terms of how to write methods that would interface with the MPU-6050 and retrieve the accelerometer and gyroscope data that were required.

However, after doing some research, a relatively low-level solution was found. The MPU-6050 has a FIFO (first-in, first-out) buffer, and like any microchip, stores data in registers. It soon became clear that in order for the UNO to read the values measured by the MPU-6050, the values would have to be pulled directly from the MPU-6050’s data registers.

More research uncovered a listing of the registers inside the MPU-6050 that store data ranging from I2C control, user control, power management, and most importantly, the accelerometer and gyroscope measurements needed. The relevant registers are reported in Figure 14.

It then became quite clear the appropriate method of approaching the problem. An I2C connection would have to be opened with the MPU-6050, and from it the program would have to request fourteen 8-bit registers from the MPU-6050, starting with the address of the first accelerometer data register ACCEL_XOUT_H, reassemble the single byte data chunks into seven 16-bit integers, and finally make use of the six relevant data points: x, y, and z accelerometer values, and roll, pitch, yaw...
gyroscope values. The simple test program created for this purpose is shown in Figure 15.

### 3.3 Switching to the Cypress

#### 3.3.1 Advantages of Cypress

To provide an accurate picture of a system’s state, measurements must be read from sensors with only very short delays between each read cycle and low loss. This often requires the use of interrupt pins. Input received through interrupt pins will briefly halt the execution of the main program loop in order to handle incoming data from sensors, which means that sensor data is always read and processed as soon as it is available. If the sensors are not connected to interrupt pins, then the microcontroller will only read the data from the sensors when it has the time to do so between process loops. Loops can vary in their execution speed, and they may not be running fast enough to read in data reliably, causing data points to be missed.

For example, imagine looking at the passenger side of a car driving by on the freeway. Intuitively, the wheels must be rotating clockwise in order for the car to go forward, but sometimes the wheels might appear to be rotating counterclockwise. This is because the human brain does not actually receive a moving picture from the eyes. Rather, as in a movie, the brain receives a series of pictures, updated many times per second. The eyes fail to “refresh” fast enough, and when the images are processed by the brain, the wheels appear to rotate the wrong way.

Interrupts are necessary to get good sensor data, but the SainSmart UNO only has two interrupt pins while the encoders and MPU combined require five. This means that, with the UNO, the sensors did not read steady data streams and would report inaccurate values; at times, the sensor data showed that the robot was moving backward when in reality it was moving forward. In contrast, the Cypress PSoC 4 has thirty-two interrupt pins, so data can be read and updated as soon as it is available. Inaccurate data due to too few interrupt pins becomes a non-issue.

Cypress PSoC 4 has programmable hardware, meaning that not all tasks are run on the microprocessor. As a result, the code runs much faster, and more calculations can be done in the same amount of time. Dead-reckoning involves performing computations, sometimes complex, on sensor data in order to calculate current location. Using the Cypress PSoC 4 greatly improves this endeavor. Hardware is configured in Cypress software PSoC 4 Creator 3.3 by adding hardware components to a schematic, customizing those components properties, and connecting the components together.

---

<table>
<thead>
<tr>
<th>Addr (Hex)</th>
<th>Addr (Dec.)</th>
<th>Register Name</th>
<th>Serial Name</th>
<th>Bit 7</th>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>3B</td>
<td>59</td>
<td>ACCEL_XOUT_H</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3C</td>
<td>60</td>
<td>ACCEL_XOUT_L</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3D</td>
<td>61</td>
<td>ACCEL_YOUT_H</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3E</td>
<td>62</td>
<td>ACCEL_YOUT_L</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3F</td>
<td>63</td>
<td>ACCEL_ZOUT_H</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>64</td>
<td>ACCEL_ZOUT_L</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>65</td>
<td>TEMP_OUT_H</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>66</td>
<td>TEMP_OUT_L</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>67</td>
<td>GYRO_XOUT_H</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>68</td>
<td>GYRO_XOUT_L</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>69</td>
<td>GYRO_YOUT_H</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>70</td>
<td>GYRO_YOUT_L</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>71</td>
<td>GYRO_ZOUT_H</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>72</td>
<td>GYRO_ZOUT_L</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 14: The registers responsible for storing accelerometer, temperature, and gyroscope data.
3.3.2 Changing from C++ to C

The SainSmart UNO was programmed primarily using C++ in the Arduino IDE. However, Cypress PSoC 4 runs exclusively on C. Alternative Motor Shield and MPU 6050 libraries written in C had to be located and implemented. This was also used as an opportunity to reorganize code into separate files and write wrapper functions to handle things like reading and writing encoder values.

3.3.3 Getting Relative Time

Accurate and precise relative time is necessary for the calculations involved in Dead-reckoning. The Arduino library includes the `millis()` and `micros()` functions, which return time in milliseconds and microseconds respectively from the initial execution of the program. However, the Cypress has no built-in functions that handle relative time. Hardware needs to be programmed to accomplish this task. A clock is wired to the counter, and the counter’s value is entered into a status register. Status registers are set by hardware and can be read by software. Given the hardware restrictions on the Cypress PSoC 4, CY8CKIT-042-BLE specifically, the clock had to be limited at 10 kHz and the counter space capped at 8 bits. This means that the status register can only hold 0.0255 seconds at a time, nowhere close to the duration of even a relatively short program. Only relative time is necessary, so a control register was wired to the counter reset. Control registers are the opposite of a status register, in that the former are set by software and read by hardware and vice versa with the latter. Control registers in combination with status registers allow for two-way communication between hardware and software. With the hardware setup diagrammed in Figure 16, the clock becomes able to be read and reset at the same time, yielding the time taken to execute a single loop iteration. No loop should ever take more than a few milliseconds, so the time cap is never an issue.

3.3.4 Configure Hardware for Motors

The two motors are configured in PSoC 4 Creator and are connected to a clock that will update the motor values at 24 MHz. The outputs of the motor controller are wired to output pins, which then direct the Adafruit v1 Motor Shield to supply power to the motors. The hardware component on the top of the diagram is for communicating the direction of the motors. [21]
3.3.5 Configure Hardware for Encoders

The two quadrature encoders must also be configured on the hardware side of PSoC 4 in order to be accessed in software. Each encoder is connected to a clock that instructs them to update their values at 24MHz and to a pin that is configured for interrupts.

3.4 Adjusting Raw MPU-6050 Data

The MPU-6050 can be initialized to measure acceleration in Gs at any of the following sensitivities: ±2g, ±4g, ±8g, or ±16g. The gyroscope can also be configured to measure angular velocity in degrees per second at one of the following sensitivities: ±250, ±500, ±1000, ±2000. As explained in section 3.2.3, the MPU-6050 stores individual sensor outputs as 16 bit signed integers. This means that all values, from the accelerometer or gyroscope, can be anywhere on the domain [−32768, 32768]. This means that the data must be scaled in order for the raw data to have the proper units.

\[
Scalar = \frac{|Range|}{2^{16}}
\]  

A ±2g and ±250deg/sec configuration was used:

\[
S_{\text{Acceleration}} = \frac{4 \times 9.80665}{2^{16}} \tag{19}
\]

\[
S_{\text{Gyroscope}} = \frac{500}{2^{16}} \tag{20}
\]

Note: 1 g = 9.80665 m/s\(^2\) is used to convert g to m/s\(^2\)

Now that the data can be interpreted correctly, the MPU-6050 needs to be calibrated. If the sensor was oriented perfectly on the robot, then the resting values for acceleration and radial velocity should be close to zero. However, placing the sensor correctly is exceedingly difficult, and sensor placement could shift when transporting the robot. Instead of relying on perfect placement, the sensor is calibrated in software before it is used. This is done by averaging \(n\) values of accelerometer and gyroscope data, and using that to offset the values read from the MPU-6050. For example, if \(Z\) is measured acceleration in the \(x\) direction, \(A\) will be the acceleration after it has been adjusted:

\[
Offset = \sum_{i=0}^{n} \frac{S_{\text{acceleration}} \cdot Z_i}{n} \tag{21}
\]

\[
A = S_{\text{acceleration}} \cdot Z - Offset \tag{22}
\]

3.5 Reading Precise Encoder Values

Quadrature encoders return an unsigned 16 bit integer, which is not very useful considering that
encoders can move forwards and backwards. In order to capture motion in the positive and negative directions, encoder values are offset by $-2^{15} - 1$ resulting in $2^{15} - 1$ being treated as the “zero” point.

Encoder Value $= \text{Raw Value} - 2^{15}$ \hspace{1cm} (23)

In addition, integers have maximum and minimum values that they can hold before they overpass those limits and wrap around. Once encoder values go above $2^{16} - 1$ or below 0, the value will overflow or underflow respectively and will wrap back to $2^{15} - 1$. When wrapping like this happens, the new encoder reading will have changed by at least $2^{15} - 1$ ticks. To counteract this, the program keeps track of the amount of overflows and underflows.

\[
\begin{align*}
\text{if } & \text{Current Value} - \text{Previous Value} \leq -2^{13} \\
& \text{Increment overflow counter by 1} \\
\text{if } & \text{Current Value} - \text{Previous Value} \geq 2^{13} \\
& \text{Increment underflow counter by 1}
\end{align*}
\]

The underflow and overflow counters are stored separately, as an alternative to using only one variable and increasing it or decreasing it by one depending on either underflow or overflow. This is done because an underflow accounts is $2^{15}$ away from the zero value, but an overflow is 1 less at $2^{15} - 1$. So storing separate values allows for more accurate readings.

Encoder Value $= \text{Overflows} \times (2^{15} - 1) - \text{Underflows} \times (2^{15}) + \text{Encoder Value}$

### 3.6 Motor Calibration

As explained in section 2.1, the goal of PID is to calculate what the motors should do in order to move the robot to the correct position and orientation. It is necessary that the robot moves in a direct relationship to the PID output; however, many motor-robot combinations do not have directly proportional power-speed relationships. To determine the power vs. speed relationship, trials were executed at each motor power, incrementing by 5, from 65 (25.5% power) to 255 (100% power), for a total of 39 trials. Encoder readings in degrees per second, time taken to complete each trial in milliseconds, and wheel circumference were used to calculate an averaged meters per second speed between two trials at each motor power. A linear regression $\text{speed} = 0.0025 \times \text{power} - 0.0196$ was computed with a coefficient of determination $R^2 = 0.99837$.

![Power-Speed Relationship](image)

The $R^2$ close to 1 shows that the relationship is almost direct, with changes in PID output eliciting proportional changes in motor speed. If the relationship was not proportional, a logarithmic regression’s inverse would be necessary to calculate the appropriate motor power.

### 3.7 Filters

#### 3.7.1 Kalman Filter

In order to improve the data read from the various sensors, the team implemented two separate multidimensional extended Kalman filters (EKF). The first EKF uses the average value between left and right encoders in conjunction with accelerometer data to estimate distance traveled (position), velocity, and acceleration. The second KF uses the difference between left and right encoders in addition to gyroscope values to estimate angle and angular velocity. Working together, these two filters will be able to provide more accurate estimations on the robot’s current location on the $y$ axis and its yaw.

**Single-Axis Kalman Filter** The implementation is somewhat simplified from the standard Kalman filter described by equations 10 to 16. The first step is to determine the state transition model, $F$, which needs to relate previous position, velocity,
and acceleration to the new states. Taking the first and second antiderivatives of acceleration, with respect to time, yields the basic physics equations that relate position, velocity, and acceleration.

\[ a = a \]  
\[ v = \int a \, dt = a \Delta t + v_i \]  
\[ y = \int \int a \, dt = \frac{a \Delta t^2}{2} + v_i \Delta t + y_i \]  

Transition Matrix

\[
\begin{bmatrix}
1 & \Delta t \\
0 & 1 \\
0 & 0
\end{bmatrix}
\]  

(27)

Next, the observation model, \( H \), must be created to relate system states to observations. In this case, there are two sensors and three states. The accelerometer sensor relates directly to the acceleration state and the average encoder value relates directly to the position state. There are no sensors that relate to velocity.

\[
\begin{bmatrix}
\text{encoders} \\
\text{accelerometer}
\end{bmatrix} =
\begin{bmatrix}
1 & 0 & 0 \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
\text{position} \\
\text{velocity} \\
\text{acceleration}
\end{bmatrix}
\]  

(28)

The last thing to set is the variances for sensors, \( R \), and prediction estimations, \( Q \). This is how the Kalman filter will determine the weights in the weighted average. Accelerometer data was sampled 100 times while the robot was stationary and the variance was calculated.

\[ \sigma^2 = \frac{1}{n} \sum_{i=0}^{n} (x_i - \bar{x})^2 \]  

(29)

Accelerometer \( \sigma^2 = 0.000774 \, m^2/s^4 \)  

(30)

There is no simple and accurate solution to measure encoder variance, so a low value is assumed.

Encoder \( \sigma^2 = 0.0001 \, m^2 \)  

(31)

The final measurement variance is as follows:

\[
\begin{bmatrix}
\text{encoder} & \text{accelerometer}
\end{bmatrix} =
\begin{bmatrix}
0.0001 & 0 \\
0 & 0.000774
\end{bmatrix}
\begin{bmatrix}
\text{encoder} \\
\text{accelerometer}
\end{bmatrix}
\]

The variance for prediction is assumed as follows:

\[
\begin{bmatrix}
pos & \text{vel} & \text{accel}
\end{bmatrix} =
\begin{bmatrix}
0.0001 & 0 & 0 \\
0 & 0.0001 & 0 \\
0 & 0 & 0.0001
\end{bmatrix}
\begin{bmatrix}
pos \\
\text{vel} \\
\text{accel}
\end{bmatrix}
\]

These constants are necessary to define in order to implement a Kalman filter. An extended Kalman filter library, TinyEKF, was then used to handle the matrix manipulation in each step [22].

**Radial Kalman Filter** The transition model, \( F \), for the radial Kalman filter is as follows:

\[
\begin{bmatrix}
1 & \Delta t \\
0 & 1
\end{bmatrix}
\]  

(32)

The observation model, \( H \), is as follows:

\[
\begin{bmatrix}
(\text{left encoder} - \text{right encoder}) \times 180^\circ \\
\text{gyroscope}
\end{bmatrix} =
\begin{bmatrix}
1 & 0 \\
0 & 1
\end{bmatrix}
\begin{bmatrix}
\theta \\
\omega
\end{bmatrix}
\]  

(33)

The sensor variance, \( R \), is

\[
\begin{bmatrix}
\text{encoder} & \text{gyroscope}
\end{bmatrix} =
\begin{bmatrix}
0.0001 & 0 \\
0 & 0.007352
\end{bmatrix}
\begin{bmatrix}
\text{encoder} \\
\text{gyroscope}
\end{bmatrix}
\]

and the prediction variance, \( Q \), is

\[
\begin{bmatrix}
\theta & \omega \\
0 & 0.0001
\end{bmatrix}
\]

(34)

This filter is also implemented using the TinyEKF library.

**3.7.2 Particle Filter**

The model for the particle filter comprises a state vector of three states: \( x \)-position, \( y \)-position, and heading (angular position). Each iteration of the filter, the living particles are incremented using data from the various sensors, according to the following equations:

\[ x_n = d \cos \theta_a + x_{n-1} \]  
\[ y_n = d \sin \theta_a + y_{n-1} \]  

In these equations, \( d \) represents the path length traveled by the robot, and \( \theta_a \) represents the average heading of the robot in a given time interval. These equations represent a linear simplification of
the robot’s motion by breaking it down into infinitesimal chunks in time, where $d$ is approximated to a straight-line motion.

Due to time limitations, only encoder data was able to be implemented and tested. In this implementation, $d$ is the average encoder distance and $\theta$ is calculated using the difference between the left and right encoder values and the width of the robot.

An implementation incorporating accelerometer and gyroscope data was also created but was unable to be tested. In this implementation, accelerometer data is twice-integrated and the infinitesimal changes in $x$- and $y$-position are combined with encoder data in a weighted average to become $d$, and gyroscope data is integrated once and the infinitesimal changes in heading data are combined with the angles obtained using encoder data in a weighted average to become $\theta$.

4 Results

4.1 Kalman Filtering

Both the single-axis and rotational Kalman filters did a great job of removing noise from the sensors with the highest variances, the accelerometer and gyroscope. Figure 20 in Appendix B shows single axis robot data and its filtered counterparts. The robot motion over this 16-second interval consisted of traveling forward 2 meters, backward 2 meters, and then forward 1 meter. The next 6 oscillations are caused by intentional human interference, and then the robot moving back to 1 meter once the interference has ceased.

The encoder data, in blue, is very accurate and does not need much filtering in this short term displacement of 2 meters. However, the accelerometer data, in orange, has a lot of noise and significant drift towards the end of the data set. If the second integral of raw acceleration were used to calculate position, it would be represented by the Position Double Integral, in purple. This calculation is incredibly inaccurate and cannot be used for dead reckoning. The Kalman filtered acceleration, in red, is significantly less noisy, and the Kalman filtered position, in green, is very accurate. The Kalman filter works very well along this axis to filter accelerometer data.

Figure 21 in Appendix B illustrates the robot’s raw and filtered rotational data. This data is from the robot driving on a straight line. When the radial displacement, $\theta$ departs from 0, the robot adjusts to continue on a straight path. Once again the encoder data, in blue, is very accurate, but the gyroscope data, in orange, has a lot of noise and jumps to over 20 degrees/second multiple times during the trial. Once filtered with encoder data, the Kalman $\omega$ has considerably less noise and is now usable.

4.2 Particle Filtering

The particle filter implemented provided an excellent estimation of current position and heading of the robot.
Unfortunately, due to time constraints, the particle filter was unable to be as thoroughly tested as the Kalman filter, though it appears to have similar potential. Overall, the particle filter is highly accurate using purely encoder data, which has low variance, but the introduction of gyroscope and accelerometer data caused some additional variance in the filter output.

One problem with the particle filter came due to the stochastic nature of the filter; its need of a large sample size of particles to be accurate in noisy data resulted in relatively slow and memory-heavy calculations. This limited the effectiveness of the linear approximation used in calculations and thus the accuracy of the filter output.

An illustration of limited sample data from the particle filter is demonstrated in Figure 19. The green dots represent the reported position of the robot, and the red dots represent a random sample of the particles; not all particles are shown, in order to improve image visual quality. The sequence of graphs shows the robot’s movement in the positive $x$-direction and negative $y$-direction.

5 Conclusions

After successfully implementing a PID controller and both a Kalman and particle filter in the robot, the team created a functioning dead-reckoning system with the capability to accurately control and estimate position and travel, despite the numerous difficulties faced regarding the quality, accuracy, and reliability of the hardware used in the robot’s construction. Filtering attenuated the accelerometer’s noisy readings enough to produce usable readings, while the PID controller managed the robot’s movements precisely enough to avoid continuous large accumulations of error.

Even though the concept of dead-reckoning is nothing new and revolutionary, and has been implemented successfully throughout history, the team has proven that even with inexpensive and low-quality equipment, the theory and implementation of a usable dead-reckoning system are both possible. For individuals without access to GPS or celestial cues, dead-reckoning can be a reliable and accurate alternative method of navigation.

Acknowledgements

The authors of this paper would like to thank everyone who was essential to the completion of this project, namely their Residential Teaching Assistant Anthony Yang and their mentors, Rupesh Chinta and Matthew Tantoy. Anthony Yang was instrumental in the guidance and organization of the project, while Rupesh Chinta and Matthew Tantoy were indispensable in providing access to hardware and technical advice. Many thanks go to the the individuals and organizations who organized and sponsored the Governor’s School of Engineering and Technology, in particular Dean Jean Patrick Antoine, Dean Ilene Rosen, Rutgers the State University of NJ, Rutgers School of Engineering, Lockheed Martin, Printrbot, and South Jersey Industries. Sincere gratitude goes to Jeff Rowberg and his excellent PC device library, to Adafruit Industries, to Simon D. Levy and his TinyEKF Kalman implementation, and to Cypress Semiconductor. Without their contributions this research endeavour would not have been possible.

References


Appendix A

Single Axis Kalman Filter

Predict

\[
\hat{x}_{k|k-1} = F_k x_{k-1|k-1}
\]

\[
\begin{bmatrix}
y_{k|k-1} \\
v_{k|k-1} \\
a_{k|k-1}
\end{bmatrix} = \begin{bmatrix}
1 & \Delta t & \frac{\Delta t^2}{2} \\
0 & 1 & \Delta t \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
y_{k-1|k-1} \\
v_{k-1|k-1} \\
a_{k-1|k-1}
\end{bmatrix}
\]

\[P_{k|k-1} = F_k P_{k-1|k-1} F_k^T + Q_k\]

Update

\[
K_k = P_{k|k-1} H_k^T (H_k P_{k|k-1} H_k^T + R_k)^{-1}
\]

\[\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k (Z_k - H_k \hat{x}_{k|k-1})\]

\[
\begin{bmatrix}
y_{k|k} \\
v_{k|k} \\
a_{k|k}
\end{bmatrix} = \begin{bmatrix}
y_{k|k-1} \\
v_{k|k-1} \\
a_{k|k-1}
\end{bmatrix} + \begin{bmatrix}
\text{enc} \\
\text{acc}
\end{bmatrix} - \begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
y_{k|k-1} \\
v_{k|k-1} \\
a_{k|k-1}
\end{bmatrix}
\]

\[P_{k|k} = (I - K_k H_k) P_{k|k-1}\]

Rotational Kalman Filter

Predict

\[
\hat{x}_{k|k-1} = F_k x_{k-1|k-1}
\]

\[
\begin{bmatrix}
\theta_{k|k-1} \\
\omega_{k|k-1}
\end{bmatrix} = \begin{bmatrix}
1 & \Delta t \\
0 & 1
\end{bmatrix}
\begin{bmatrix}
\theta_{k-1|k-1} \\
\omega_{k-1|k-1}
\end{bmatrix}
\]

\[P_{k|k-1} = F_k P_{k-1|k-1} F_k^T + Q_k\]

Update

\[
K_k = P_{k|k-1} H_k^T (H_k P_{k|k-1} H_k^T + R_k)^{-1}
\]

\[\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k (Z_k - H_k \hat{x}_{k|k-1})\]

\[
\begin{bmatrix}
\theta_{k|k} \\
\omega_{k|k}
\end{bmatrix} = \begin{bmatrix}
\theta_{k|k-1} \\
\omega_{k|k-1}
\end{bmatrix} + \begin{bmatrix}
\text{enc} \\
\text{gyro}
\end{bmatrix} - \begin{bmatrix}
1 & 0 \\
0 & 1
\end{bmatrix}
\begin{bmatrix}
\theta_{k|k} \\
\omega_{k|k}
\end{bmatrix}
\]

\[P_{k|k} = (I - K_k H_k) P_{k|k-1}\]

\[\begin{bmatrix}
\theta_{k} \\
\omega_{k}
\end{bmatrix} = \begin{bmatrix}
\text{enc} \\
\text{gyro}
\end{bmatrix} - \begin{bmatrix}
1 & 0 \\
0 & 1
\end{bmatrix}
\begin{bmatrix}
\theta_{k} \\
\omega_{k}
\end{bmatrix}\]
Appendix B

Figure 20: Single-Axis Robot Data

Figure 21: Rotational Robot Data