RidgeSoft™

Enabling Your Robot to Keep Track of its Position

www.ridgesoft.com

Revision 1.2

Introduction

If you are like many other robot developers, you have probably wondered, “How can my robot keep track of its location?” Solving this problem is often a fundamental requirement when developing a mobile robot. Even a simple robot that needs to return to its home base – for example, a charging station – can benefit from having the ability to know where it is as it carries out its tasks.

This tutorial will teach you how to program your robot to use odometry and dead reckoning to enable your robot to keep track of its location.

Before You Get Started

This tutorial builds on topics covered in the following tutorials:

Creating Your First IntelliBrain Program
Programming Your Robot to Perform Basic Maneuvers
Creating a User Interface for Your Robot
Creating Shaft Encoders for Wheel Position Sensing

If you are not already familiar with the concepts covered in these tutorials, you should complete them first, before attempting this tutorial. This and other tutorials are available from the RidgeSoft web site, www.ridgesoft.com.

The programming steps in this tutorial build on the MyBot program developed in the *Creating Shaft Encoders for Wheel Position Sensing* tutorial.

You will need an IntelliBrain™-Bot educational robot kit to complete this tutorial.

How can my robot keep track of its location?

Knowing where you are is a problem that has challenged and fascinated humans since the beginning of time! Our solutions to this problem range from relying on our basic senses of sights, sounds and smells to using sophisticated technologies, such as a global positioning system (GPS).

Similarly, there are a multitude of approaches to enable your robot to determine its current location, a process referred to as “localization.” These approaches range from sensing landmarks, to “dead reckoning,” to using GPS receivers.

Which method of localization should my robot use?

Rather than attempting to pick an ideal localization method, you can, instead, design your software in a way that will allow you to experiment with a variety of localization methods. You can do this by designing a generic interface in your Java program that will be suitable for interfacing to a variety of localization methods. Making use of Java’s interface mechanism will allow you to use different localization methods interchangeably, rather than locking your program into a single solution to the localization problem.

Creating a Localizer Interface

Figure 1 shows a class diagram depicting navigation and localization classes that you can develop to enable your robot to keep track of its position as well as navigate from place to place. In this design, the class that implements the "Localizer" interface uses sensors to keep track of your robot's current location and the direction it is heading. The Localizer monitors the sensors and determines your robot's current position, it's "pose," and makes the position data accessible to other classes through a "getter" method. The getter method returns the current pose data as an instance of the class named "Pose."

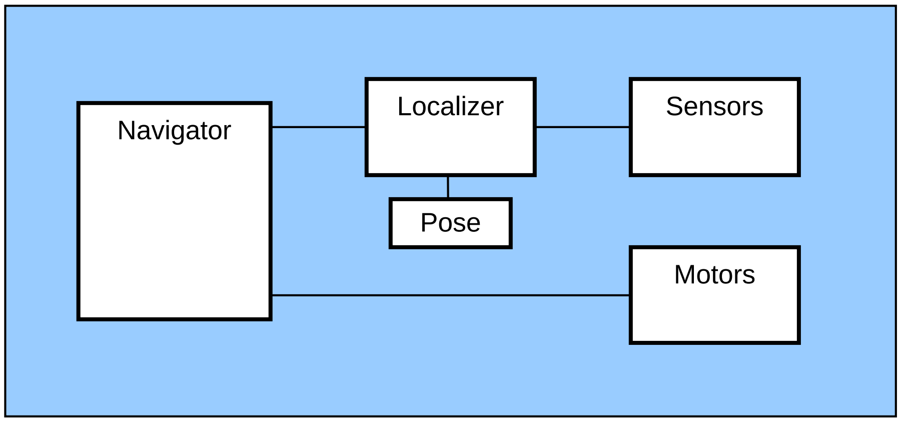

Figure 1 – Navigation and Localization Class Diagram

The Navigator relies on the Localizer to keep track of your robot's current position, using this information to calculate how to navigate to its next destination. The Navigator then powers the motors such that the robot heads toward the destination. Although the navigator and motors are shown here to provide context, navigation has been left as a topic for a separate tutorial.

In this tutorial you will extend the MyBot program you developed in the *Creating Shaft Encoders for Wheel Position Sensing* tutorial. Using the File->New Class menu item in RoboJDE, add a new interface named "Localizer."

Your Localizer interface will need to declare a method to get the current Pose (x, y position and heading). In addition, it will also be a good idea to declare methods to set these values so your robot's position can be set when the program starts and so it can be corrected from time to time. Edit the new class to contain the following code:

```
public interface Localizer {
    public Pose getPose();
    public void setPose(Pose pose);
    public void setPosition(float x, float y);
    public void setHeading(float heading);
    }
```

Again using RoboJDE, create the "Pose" class. This class will need member variables to keep track of the position coordinates, x and y, and the direction your robot is heading. Since the Pose class will be very simple you can make its member variables, x, y and heading, directly accessible by other classes by declaring them "public." This will allow slightly faster access to these variables than if they had to be read through get methods such as pose.getX() and pose.getY(), saving computing power for other tasks. You will want to prevent Pose objects from changing while they are being used for navigation calculations. You can do this by declaring them "final." This will make all Pose objects immutable. Member variables declared final can only be assigned values when an instance of a class is constructed.

Use the following code for the Pose class:

```
public class Pose {
    public final float x;
    public final float y;
    public final float heading;

    public Pose(float x, float y, float heading) {
        this.x = x;
        this.y = y;
        this.heading = heading;
    }
}
```

By defining the Localizer interface and the Pose class, other portions of your program will not need to know anything about the specifics of one localization technology or another. This will allow the rest of your program to work interchangeably with a variety of localization methods.

Position Tracking using Dead Reckoning

Dead reckoning dates back to the days when sailors used measurements of speed, heading and time to deduce the location of their sailing ships. You can use dead reckoning as a means for your robot to keep track of its position. By closely monitoring the movement of its wheels using shaft encoders, your robot can track its movements. Using basic geometry it can calculate – "deduce" – changes in its position caused by each small movement.

Implementing a Localizer

Your next step is to create a class that implements the Localizer interface. Using two shaft encoders and dead reckoning, this class will use odometry to perform localization.

Create a new class named "OdometricLocalizer" and include the following code:

```
import com.ridgesoft.robotics.ShaftEncoder;

public class OdometricLocalizer extends Thread
             implements Localizer {
    private static final float PI = 3.14159f;
    private static final float TWO_PI = PI * 2.0f;

    private ShaftEncoder mLeftEncoder;
    private ShaftEncoder mRightEncoder;
    private float mDistancePerCount;
    private float mRadiansPerCount;
    private float mX = 0.0f;
    private float mY = 0.0f;
    private float mHeading = 0.0f;
    private int mPeriod;
    private int mPreviousLeftCounts;
    private int mPreviousRightCounts;

    public synchronized void setPose(Pose pose) {
        mX = pose.x;
        mY = pose.y;
        mHeading = pose.heading;
    }

    public synchronized void setPose(float x, float y,
                                     float heading) {
        mX = x;
        mY = y;
        mHeading = heading;
    }

    public synchronized void setPosition(float x, float y) {
        mX = x;
        mY = y;
    }

    public synchronized void setHeading(float heading) {
        mHeading = heading;
    }

    public synchronized Pose getPose() {
        return new Pose(mX, mY, mHeading);
    }
}
```

This code implements the straight-forward aspects of the OdometricLocalizer class, leaving the most interesting portions of this class, those that implement dead reckoning, for you to add as you complete the following sections.

Understanding Dead Reckoning

The key to implementing dead reckoning is to understand the geometry of your robot and how it moves. In this case, your robot is an IntelliBrain-Bot, a robot that uses two independently powered wheels to move and steer. This is a frequently used design for mobile robots that is commonly referred to as "differential drive."

If you have completed the *Creating Shaft Encoders for Wheel Position Sensing* tutorial, each wheel on your robot will have a shaft encoder that maintains a counter indicating the angular position of the wheel. As the wheel rotates forward, the counter counts up. As it rotates backwards, the counter counts down. Each up or down count of the encoder corresponds to a fraction of a turn of the wheel. The distance the wheel travels across the floor with each count can be calculated by dividing the circumference of the wheel by the number of counts the encoder increments or decrements with one full revolution of the wheel. The equation for this is:

```
distancePerCount = Pi * diameterWheel / countsPerRevolution
```

When the robot moves in approximately a straight line, the distance it travels is simply the average number of encoder counts the two wheels turn times the distance the robot travels per encoder count:

```
deltaDistance = (leftCounts + rightCounts) / 2.0 * distancePerCount
```

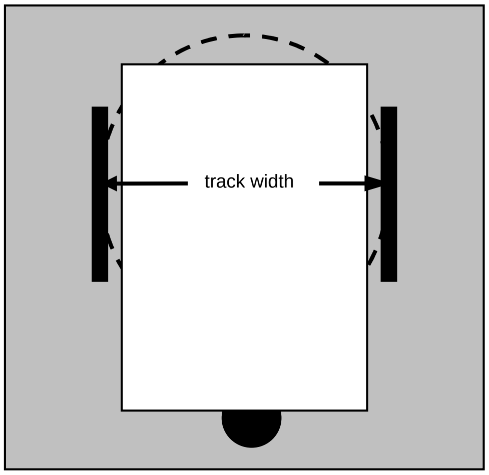

Figure 2 - Circle Traced by Robot Turning in Place

If your robot rotates in place by turning its wheels in opposite directions its wheels will trace a circle whose diameter is equal to its track width, as shown in Figure 2. Your robot's heading changes as the wheels make their way around this circle. Your robot will rotate 2 x Pi radians (360 degrees) when the wheels have traversed the circumference of this circle. Therefore, the number of encoder counts to rotate a full circle depends on the geometry of your robot, and can be calculated by the equation:

```
countsPerRotation = (trackWidth / wheelDiameter) * countsPerRevolution
```

This equation gives the number of counts for a single wheel. If you take the difference in counts between the two wheels and note that each rotation is 2 x Pi radians, you can rearrange the equation into the following two equations:

```
radiansPerCount = Pi * (wheelDiameter/trackWidth) / countsPerRevolution

deltaHeading = (rightCounts – leftCounts) * radiansPerCount
```

where deltaHeading is the angle in radians your robot rotates.

Given the left and right encoder counts, these equations enable your program to estimate your robot's position provided it moves straight ahead or rotates in place, but what if the robot moves along an arbitrary path?

Your localizer class can estimate your robot's position along an arbitrary path by treating its motion as many small, discrete movements. By adding all of the discrete movements together, the localizer can deduce your robot's position regardless of the path it follows.

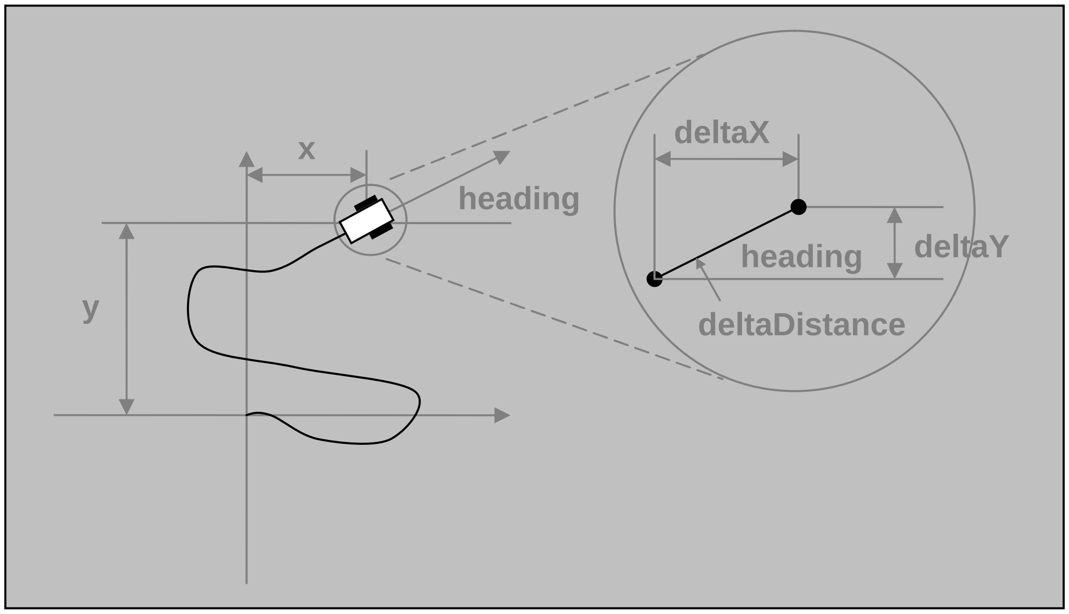

Figure 3 – Calculating Discrete Position Changes

As shown in Figure 3, imagine your robot moves a small distance, deltaDistance, while it travels forward in the heading direction. Assuming the direction your robot is heading doesn't change significantly during each small movement, the change in the position along the x axis, deltaX, and change in position along the y axis, deltaY, can be calculated using the following trigonometric calculations:

```
deltaX = deltaDistance * cos(heading)

deltaY = deltaDistance * sin(heading)
```

The deltaX and deltaY values can then be added to the previous position estimate to obtain a new position estimate.

Implementing Dead Reckoning

In order to keep accurate tabs on your robot's position, your localizer class will need to sample the shaft encoders frequently so it can perform the dead reckoning calculations for small movements that can be assumed to be in a straight line. One way you can do this is by implementing the class as an extension of the Thread class. This will allow your dead reckoning calculations to be performed at regular intervals without undue interaction with other tasks your software needs to perform at the same time. By using a separate thread, you don't need to come up with an intricate design that schedules periodic dead reckoning calculations in between other tasks. Instead, you can leave it up to the underlying Java virtual machine to take care of this.

Your OdometricLocalizer class will need to declare it extends the Thread class and it will need to include a run method. The run method must periodically perform the dead reckoning calculations.

Implement the run method according to the following outline:

```
public void run() {
    try {
        // periodically sample the encoder and
        // perform dead reckoning calculations
            :
    }
    catch (Throwable t) {
        t.printStackTrace();
    }
}
```

It is a good idea to always include a try-catch block in the run method of any sub-class of Thread you create, as you have done in this run method. If any unexpected exceptions occur in your program, the catch block will print a stack trace that will help you debug the problem. Without this, the tread will simply terminate without giving you an indication something has gone wrong.

Your run method will need to include a loop to periodically perform the necessary calculations. It will also need to follow this with a call to the Thread.sleep method to suspend it until it is time to calculate the next position update.

Place the following code in the try block in the code you previously added to the run method, above:

```
long nextTime = System.currentTimeMillis();
while(true) {
    // read encoders
```

```
            :
        // calculate change in pose
            :
        // update position and heading estimates
            :
        nextTime += mPeriod;
        Thread.sleep(nextTime - System.currentTimeMillis());
    }
```

Reading the encoders is simply a matter of calling each encoder's getCounts method:

```
    int leftCounts = mLeftEncoder.getCounts();
    int rightCounts = mRightEncoder.getCounts();
```

Conveniently, this code does not depend on the specific type of shaft encoder sensor your robot uses. The ShaftEncoder interface has enabled loose coupling between the encoder classes and your localizer class, facilitating interchangeability of encoders. For example, if you need to improve the accuracy of your robot's localizer you can easily switch to higher precision encoders such as Nubotics WheelWatcher WW-01 encoders.

After your code has obtained updates from the encoders, it must perform the dead reckoning calculations discussed previously. Add the following code to do this:

```
    int deltaLeft = leftCounts - mPreviousLeftCounts;
    int deltaRight = rightCounts - mPreviousRightCounts;
    float deltaDistance = 0.5f * (float)(deltaLeft + deltaRight)
                               * mDistancePerCount;
    float deltaX = deltaDistance * (float)Math.cos(mHeading);
    float deltaY = deltaDistance * (float)Math.sin(mHeading);
    float deltaHeading = (float)(deltaRight - deltaLeft)
                              * mRadiansPerCount;
```

Updating the absolute position and heading variables is just a matter of adding the deltaX, deltaY and deltaHeading to the previous values. However, because there are multiple threads accessing the pose data, you must be careful to ensure that one thread doesn't read the pose data while the localizer thread is updating it, otherwise, your program will generate erroneous results.

You can use Java's built-in synchronization mechanism to coordinate access to the variables that are accessed by several threads by putting the update code in a synchronized code block and adding the "synchronized" modifier to all methods that allow other threads to access the pose data.

Add the following code to update the position variables with the incremental changes in position:

```
    synchronized(this) {
        mX += deltaX;
        mY += deltaY;
        mHeading += deltaHeading;

        // limit heading to -Pi <= heading < Pi
        if (mHeading > PI)
            mHeading -= TWO_PI;
        else if (mHeading <= -PI)
            mHeading += TWO_PI;
    }
```

Following this, you must update the previous counts variables for the next iteration:

```
    mPreviousLeftCounts = leftCounts;
    mPreviousRightCounts = rightCounts;
```

Finally, you will need to include a constructor in the OdometricLocalizer class, as follows:

```
    public OdometricLocalizer(ShaftEncoder leftEncoder,
                              ShaftEncoder rightEncoder,
                              float wheelDiameter,
                              float trackWidth,
                              int countsPerRevolution,
                              int threadPriority,
                              int period) {
        mLeftEncoder = leftEncoder;
        mRightEncoder = rightEncoder;
        mDistancePerCount = (PI * wheelDiameter)
                             / (float)countsPerRevolution;
        mRadiansPerCount = PI * (wheelDiameter / trackWidth)
                             / countsPerRevolution;
        mPeriod = period;
        mPreviousLeftCounts = leftEncoder.getCounts();
        mPreviousRightCounts = rightEncoder.getCounts();
        setDaemon(true);
        setPriority(threadPriority);
        start();
    }
```

Making Use of Your New Classes

Your new classes are now ready to be put to use. You will need to construct an OdometricLocalizer object in the main method of your program's main class, MyBot. The constructor for the OdometricLocalizer class requires that you provide your robot's wheel diameter and track width measurements. For the IntelliBrain-Bot, these are approximately 2.65 inches and 4.55 inches, respectively.

Add the following code to the MyBot class just after the lines where the shaft encoder objects are constructed:

```
Localizer localizer = new OdometricLocalizer(
                                 leftEncoder,
                                 rightEncoder,
                                 2.65f,
                                 4.55f,
                                 16,
                                 Thread.MAX_PRIORITY-1, 30);
```

Testing

You should also implement a few test classes that will enable you to verify and debug your localizer classes. First, you will want to implement a class to display the current pose on the LCD display. You can do this by creating a PoseScreen class and inserting the following code:

```
import com.ridgesoft.io.Display;

public class PoseScreen implements Screen {
    private Localizer mLocalizer;

    public PoseScreen(Localizer localizer) {
        mLocalizer = localizer;
    }

    public void update(Display display) {
        Pose pose = mLocalizer.getPose();
        display.print(0, Integer.toString((int)pose.x) +
                         ", " + (int)pose.y);
        display.print(1, Integer.toString(
                         (int)Math.toDegrees(pose.heading)));
    }
}
```

You will also need to add a line to the MyBot class to add this screen to the list of screens, as follows:

```
new PoseScreen(localizer),
```

Lastly, you need to add a couple of functions to make your robot move. For now, a function that moves the robot forward for a specified period of time and a similar function that rotates the robot in place for a specified period of time will be sufficient. These two classes will allow you to compare your robot’s actual position after a simple move to the position deduced by the localizer. Create two classes TimedForward and TimedRotate to add these new functions to your program. Use the following code for these classes:

<u>TimedForward</u>

```
import com.ridgesoft.robotics.Servo;

public class TimedForward implements Runnable {
    private Servo mLeftServo;
    private Servo mRightServo;
    private DirectionListener mLeftListener;
    private DirectionListener mRightListener;
    private int mDuration;

    public TimedForward(Servo leftServo, Servo rightServo,
                        DirectionListener leftListener,
                        DirectionListener rightListener,
                        int duration) {
        mLeftServo = leftServo;
        mRightServo = rightServo;
        mLeftListener = leftListener;
        mRightListener = rightListener;
        mDuration = duration;
    }

    public void run() {
        try {
            // forward
            mLeftServo.setPosition(100);
            if (mLeftListener != null)
                mLeftListener.updateDirection(true);
            mRightServo.setPosition(0);
            if (mRightListener != null)
                mRightListener.updateDirection(true);
            Thread.sleep(mDuration);

            // stop
            mLeftServo.setPosition(50);
            mRightServo.setPosition(50);
        }
        catch (Throwable t) {
            t.printStackTrace();
        }
    }

    public String toString() {
        return "Timed Forward";
    }
}
```

<u>TimedRotate</u>

```
import com.ridgesoft.robotics.Servo;

public class TimedRotate implements Runnable {
    private Servo mLeftServo;
    private Servo mRightServo;
    private DirectionListener mLeftListener;
    private DirectionListener mRightListener;
    private int mDuration;

    public TimedRotate(Servo leftServo, Servo rightServo,
                       DirectionListener leftListener,
                       DirectionListener rightListener,
                       int duration) {
        mLeftServo = leftServo;
```

```
            mRightServo = rightServo;
            mLeftListener = leftListener;
            mRightListener = rightListener;
            mDuration = duration;
        }

        public void run() {
            try {
                // rotate
                mLeftServo.setPosition(45);
                if (mLeftListener != null)
                    mLeftListener.updateDirection(false);
                mRightServo.setPosition(45);
                if (mRightListener != null)
                    mRightListener.updateDirection(true);
                Thread.sleep(mDuration);

                // stop
                mLeftServo.setPosition(50);
                mRightServo.setPosition(50);
            }
            catch (Throwable t) {
                t.printStackTrace();
            }
        }

        public String toString() {
            return "Timed Rotate";
        }
    }
```

You must add these new functions to the function list in the MyBot class, as follows:

```
    new TimedForward(leftServo, rightServo,
                     (DirectionListener)leftEncoder,
                     (DirectionListener)rightEncoder,
                     10000),
    new TimedRotate(leftServo, rightServo,
                     (DirectionListener)leftEncoder,
                     (DirectionListener)rightEncoder,
                     1800),
```

Now you can build and download your program to your robot. When you run the “Timed Forward” function, your robot will move straight forward for 10 seconds. When you run the “Timed Rotate” function your robot will rotate in place for 1.8 seconds. When you run these functions, you can scroll the thumbwheel to display the PoseScreen. This will allow you to see the position calculated by your localizer class. You can test the accuracy of your localizer class by comparing the displayed values to the actual measurement you make with a ruler and a protractor.

As you test your robot, you will likely see that the position calculated by the localizer tends to drift. As the robot travels further, the error will normally continue to increase. Much of the error can be attributed to the lack of precision of the shaft encoders. However, regardless of the precision of the encoders, errors will tend to accumulate as your robot moves. This is because these measurements and calculations are based on self-centric measurements, not external references, which do not drift. The dead reckoning algorithm has no means to detect or correct accumulated errors.

The AnalogShaftEncoder objects from the *Creating Shaft Encoders for Wheel Position Sensing* tutorial can measure the wheel position to the accuracy of 1/16 of a rotation, which is 22.5 degrees. While this is quite reasonable for measuring straight distances, it is more problematic when calculating your robot's heading. The precision with which your localizer can track your robot's heading using these sensors is low, which can result in errors accumulating quickly.

There are several ways you can improve your robot's localization accuracy. The most obvious way is to use higher precision encoders. Nubotics WheelWatcher WW-01 encoders have eight times more precision. These encoders can track the wheel position to 2.8 degrees. Upgrading your robot with WheelWatcher encoders will significantly improve the accuracy of your localizer. However, as mentioned previously, since the position calculations are strictly based on self-centric measurements rather than fixed references external to your robot, errors will still accumulate, but the magnitude of the accumulated error will be significantly reduced.

Another way to improve the accuracy of the localizer is to increase the track width of your robot or reduce the size of its wheels. Either of these changes will increase the precision to which the heading can be calculated, reducing errors that accumulate due to lack of precision of heading estimates.

One additional method of improving the accuracy of your localizer is adding a compass sensor such as the Devantech CMPS03. This will allow your robot to track its heading relative to a fixed external reference, the Earth's magnetic field. Because the Earth's magnetic field is an external reference, the heading estimate will not be subject to accumulated errors, as is the case with dead reckoning-based heading estimates. However, a magnetic compass is subject to other errors, such as local disruption of the Earth's magnetic field caused by ferrous metals and electrical equipment.

Even though your robot cannot calculate its position with great accuracy, you will find as you implement navigation functions that incorporate position feedback from the localizer it will, nevertheless, be able to navigate surprisingly well.

Conclusion

Localization is a difficult problem. Choosing an effective method will depend on the requirements for a particular robot. Combining several methods may be effective. For example, your robot could use dead reckoning to navigate to the vicinity of a landmark then switch to landmark-based navigation, just as a ship's captain might use dead reckoning while out of sight of land and then switch to landmark-based navigation when land is in sight.

By completing this tutorial you have learned how to use dead reckoning to enable your robot to keep track of its position. However, dead reckoning is dependent on accurate measurement of incremental movements of your robot. The precision of measurements will greatly impact the accuracy to which your robot can track its position. Furthermore, with dead reckoning, errors accumulate as your robot moves. This will result in the position calculated by the localizer continually drifting from your robot's actual position.

You can counter the problem of accumulated errors by enhancing your robot to sense external references such as landmarks or the Earth's magnetic field. By using fixed external references, your robot can detect and remove accumulated errors. You can also use higher precision encoders or change the geometry of your robot to reduce, but not eliminate, accumulated errors.

In addition to implementing dead reckoning-based localization, you have taken advantage of Java's multi-threading and interface capabilities. These capabilities have enabled you to define a generic interface that supports a wide range of localization technologies. Because this interface is very generic, you can easily substitute classes that implement other methods of localization. The use of an interface allows you to do this without impacting other portions of your program. In summary, the Localizer interface defines a cohesive function, localization, in such a way that is loosely coupled to other classes that make up your robot's control program. By designing your Java classes to provide cohesive functionality and loose coupling to other classes you will make your programs easier to maintain, extend and reuse.

Exercises

1. Measure and record the diameter of your robot's wheels and the track width of your robot.
2. How far will your robot travel if both wheels rotate one revolution at the same speed?
3. How many degrees will your robot's heading change if both wheels rotate one revolution in opposite directions?
4. If you put 25% larger wheels on your robot, how will it affect the number of degrees your robot's heading will change when the wheels rotate one revolution in opposite directions?

5. If you increase the track width of your robot by 50% how will it affect the number of degrees your robot's heading will change when the wheels rotate one revolution in opposite directions?
6. If both wheels on your robot turn at the same speed, how far will your robot move with each encoder count?
7. How many counts do the encoders on your robot increase or decrease with each full revolution of the wheel?
8. What is the maximum distance your robot can move without incrementing or decrementing either encoder counter value?
9. What would be the maximum distance your robot could move without incrementing or decrementing either encoder counter value be if you were to switch to using encoders that count 128 counts per wheel revolution?
10. If the wheels on your robot each rotate one full encoder count in opposite directions, how many degrees will your robot's heading change?
11. What is the maximum possible angle your robot can rotate in place without the encoder counter values changing? Hint: Each wheel can rotate a maximum of just under one encoder count in opposite directions without incrementing or decrementing either encoder's counter value.
12. What effect will errors in your measurements of wheel diameter and track width have on dead reckoning position calculations?
13. How will switching to encoders that count 128 counts per revolution affect the precision to which the OdometricLocalizer can measure the heading of your robot?
14. How would doubling the track width of your robot affect the precision to which the OdometricLocalizer can measure the heading of your robot?
15. How would decreasing the diameter of the wheels by 25% affect the precision to which the OdometricLocalizer can measure the heading of your robot?
16. Upgrade your robot to use Nubotics WheelWatcher WW-01 encoders. Run several tests to compare the accuracy of position measurements of your robot with the original encoders to the accuracy with the WW-01 encoders. What differences in accuracy did you observe?
17. Add a Devantech CMPS03 sensor to your robot. Create a new Localizer class that uses the compass to determine the heading and the encoders to measure distance traveled. Run several tests and compare the accuracy of your robot's position estimates with and without the compass.
18. Investigate several other methods of localization. What are the pros and cons of each method?

RidgeSoft, LLC
PO Box 482
Pleasanton, CA 94566
www.ridgesoft.com