Programming Your Robot to Navigate

www.ridgesoft.com

Revision 1.0

Introduction

Whether you are building a robot to explore another planet, compete in the DARPA Grand Challenge, rescue victims of a disaster or just satisfy your own curiosity, it's likely you'll want it to have the ability to navigate from place to place on its own. This tutorial will show you how to do that by building on concepts and software components from several other tutorials.

The tutorial, *Programming Your Robot to Perform Basic Maneuvers,* demonstrates how you can program your robot to move straight forward and rotate in place. By programming timed sequences of these simple maneuvers you can make your robot move in more complex patterns; however, you are likely to find your robot does not maneuver as predictably as you would like. Unfortunately, this technique – simply programming your robot to turn its motors on and off in predetermined timed sequences – fails to account for varying conditions your robot will encounter. Your robot's actual performance will vary significantly as battery level, floor texture, and other factors change.

This tutorial will demonstrate how you can use position feedback to improve your robot's ability to navigate. You will integrate shaft encoding and localization classes from the *Creating Shaft Encoders for Wheel Position Sensing* and *Enabling Your Robot to Keep Track of its Position* tutorials to take advantage of position feedback. By incorporating position feedback, your robot will be able to act on real-time measurements of its actual performance rather than blindly following an inflexible sequence of operations.

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
Enabling Your Robot to Keep Track of its Position

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 tutorial, *Enabling Your Robot to Keep Track of its Position*.

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

Understanding How to Program Your Robot to Navigate

Before you embark on programming your robot to navigate, you will first need to think about how your program will go about solving the navigation "problem." You can do this by breaking down the problem into smaller problems. In order to navigate effectively your robot must:

1. know where it wants to go,
2. know where it is and what direction it is facing,
3. determine the heading (direction to) its destination,
4. steer to and maintain the heading to its destination, and
5. stop when it has reached its destination.

Knowing Where to Go

Knowing where to go is easy, at least for your navigator. It is the responsibility of higher level software to specify the course to take. Your navigator will only need to provide a means for it to be told where your robot should go next. Therefore, it will need to provide methods that allow higher level software to set the next goal. The following four methods will serve this purpose:

1. moveTo - move to a specified location,
2. turnTo - turn to face a particular direction,
3. go - move continuously in one direction, and
4. stop.

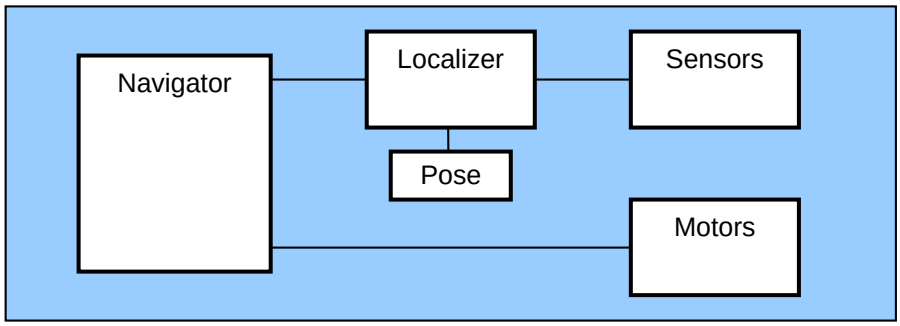

Figure 1 – Navigation and Localization Class Diagram

Knowing Where Your Robot Is

Enabling your robot to keep track of where it is isn't particularly easy; however, if you have completed the tutorial *Enabling Your Robot to Keep Track of its Position* you will know how to create classes that solve this problem. With two shaft encoder sensors (see the *Creating Shaft Encoders for Wheel Position Sensing* tutorial) and the OdometricLocalizer class, your robot will be able to keep track of its position. You can feed position data back into your navigator by interfacing it to an instance of your OdometricLocalizer class, as shown in Figure 1.

Determining Where to Head

Using the output of an OdometricLocalizer object, your navigator can use trigonometry to calculate the heading to its next destination.

Figure 2 depicts your robot heading toward its destination. The location of the destination relative to your robot's current position in Cartesian coordinates is (xError, yError), the x and y components of the "error" between where it is and where it is trying to go. The heading is the angle to the destination.

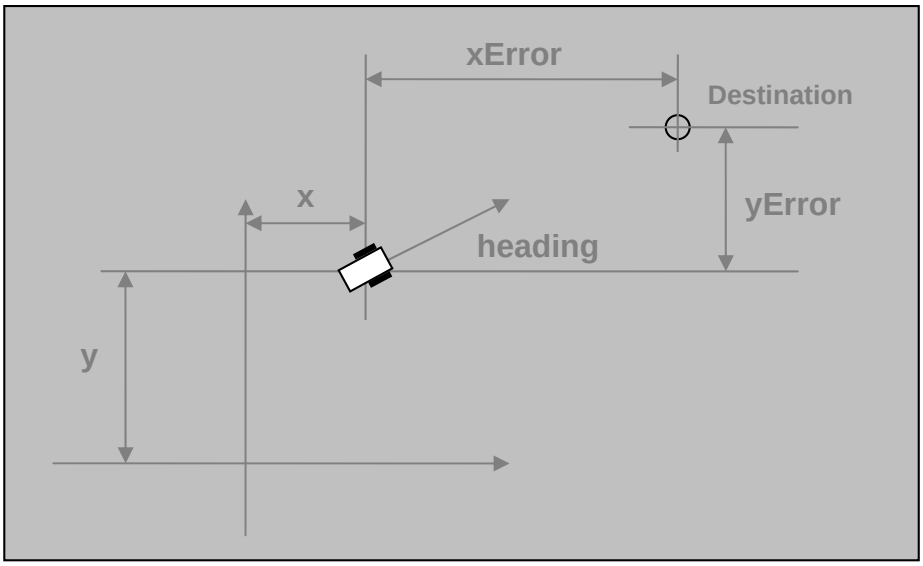

Figure 2 - Navigating to a Point

Your navigator can calculate xError and yError by subtracting its current x and y coordinates from those of the destination:

xError = destinationX – x
yError = destinationY – y

Recalling from trigonometry, the heading is the arc tangent of xError divided by yError:

heading = arctan(xError / yError)

Steering to and Maintaining a Heading

Once your navigator has calculated the direction your robot needs to head, it must apply power to the motors to steer your robot such that it maintains the heading and moves forward. This may sound hard to accomplish, but it turns out it is fairly easy to accomplish.

The key to keeping your robot heading toward its destination is recognizing that it will move approximately straight ahead when your navigator applies the same power to both wheels. Your navigator can steer your robot left or right as it

moves forward by applying slightly more power to one wheel than the other. The further off course your robot is, the harder your navigator will need to steer to quickly return to the desired heading; therefore, the larger the error, the larger the power differential it will need to apply to steer back on course.

This technique is an application of "proportional control," an extremely widely used method for controlling dynamic systems. It gets its name because the output of the controller – in this case, your navigator – is proportional to the error between the actual value and the target value of the variable being controlled – in this case, your robot's heading.

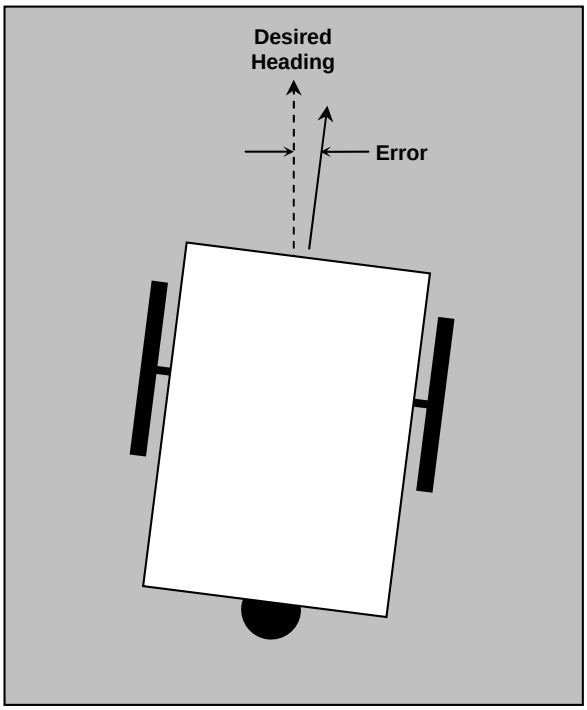

Figure 3 - Heading Error

Consider the situation shown in Figure 3. The robot is heading forward, but slightly off its desired heading. The navigator can correct for the error in heading by applying slightly more power to the right wheel than the left wheel. The amount of power to apply to each wheel can be calculated according to the following equations:

leftWheelPower = drivePower – differential
rightWheelPower = driverPower + differential

The drivePower is the base level of power applied to the wheels to move straight ahead. The differential variable controls the difference in power applied to the two wheels. The power differential can be made proportional to the heading error using the equation:

differential = gain * error

Hence, the difference in power applied to the wheels will become larger as the error increases, causing the robot to respond more forcefully to get back on course. When the error is zero both wheels will receive the same power.

The gain value in the previous equation is a constant value that controls how aggressively your navigator will respond to error. A large gain constant will result in an aggressive response to error. However, too large a gain will cause your robot to over-steer and oscillate wildly. Setting the gain too small will result in a sluggish response to error or no response at all.

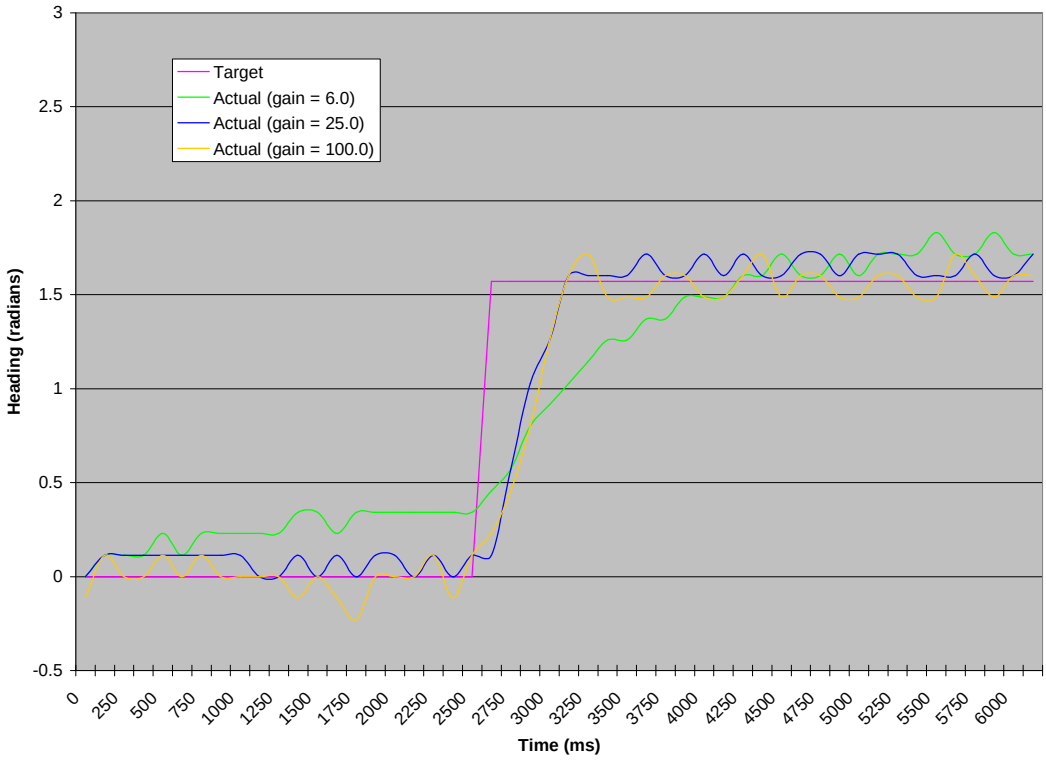

Figure 4 - Navigator Response to Heading Change

Figure 4 shows measurements of the response of a real robot to a sudden change of the target heading. These measurements were taken using the TestNavigatorResponse class included in Appendix A. With the gain set to 6.0 the robot was slow to respond to the change and the robot drifted off course. With the gain set to 100.0 the robot over steered and displayed erratic motion. Setting the gain to 25.0 yielded a quick response and good stability.

Finally, in order to keep your robot heading to its destination, your navigator will frequently need to readjust the power it applies to the wheels; otherwise, it will quickly wander off course. It can do this by periodically taking into account its current position and repeating all of the navigation calculations.

A convenient way of implementing this is to use a separate thread for navigation. This thread can perform the navigation calculations then sleep for a short period of time before repeating the process.

Determining When to Stop

Your navigator will need to stop your robot when it reaches its destination. You might think all your navigator has to do is turn off the motors when its current position is the destination. Unfortunately, it isn't quite that simple!

It is easy to forget your robot is not a high precision system. It will not navigate with perfect accuracy. Although it can navigate to close proximity of its destination, if you insist on extreme precision you will discover that your robot will get close to its destination and then begin to flounder around attempting to reach the exact destination. In robotics, you usually have to accept close as being good enough! Instead of insisting on perfect navigation, your robot will perform better if you relax your accuracy requirements and program it to stop within reasonable proximity of its destination.

Your robot could use the Pythagorean Theorem to determine its proximity to the next destination. However, this calculation requires calculating a square root, which is takes a lot of time to compute. Instead, you can use the sum of the absolute values of the two error terms, xError and yError, as a rough approximation of how far your robot is from its destination. This will not yield the exact distance, but it will be good enough and far easier to calculate.

Controlling Servo Speed

Before you begin implementing your navigator you will first need to develop a class to provide finer control of the servos that power your robot's wheels. In other tutorials you learned how to maneuver your robot by simply turning the servo motors on and off. However, steering your robot while it drives forward requires more precise control of the power it applies to each wheel. Simply turning the motors on or off will not provide sufficient control.

The servos included with the IntelliBrain-Bot kit incorporate modifications that enable them to rotate continuously so they function like motors, rather than as conventional servos. Modified servos are commonly called "continuous rotation servos." Even though continuous rotation servos function mechanically like motors, your program still controls them using servo positioning commands provided by the setPosition method of the Servo interface. When used with an unmodified servo, this method allows your program to set the shaft position between 0% and 100% of its range of motion.

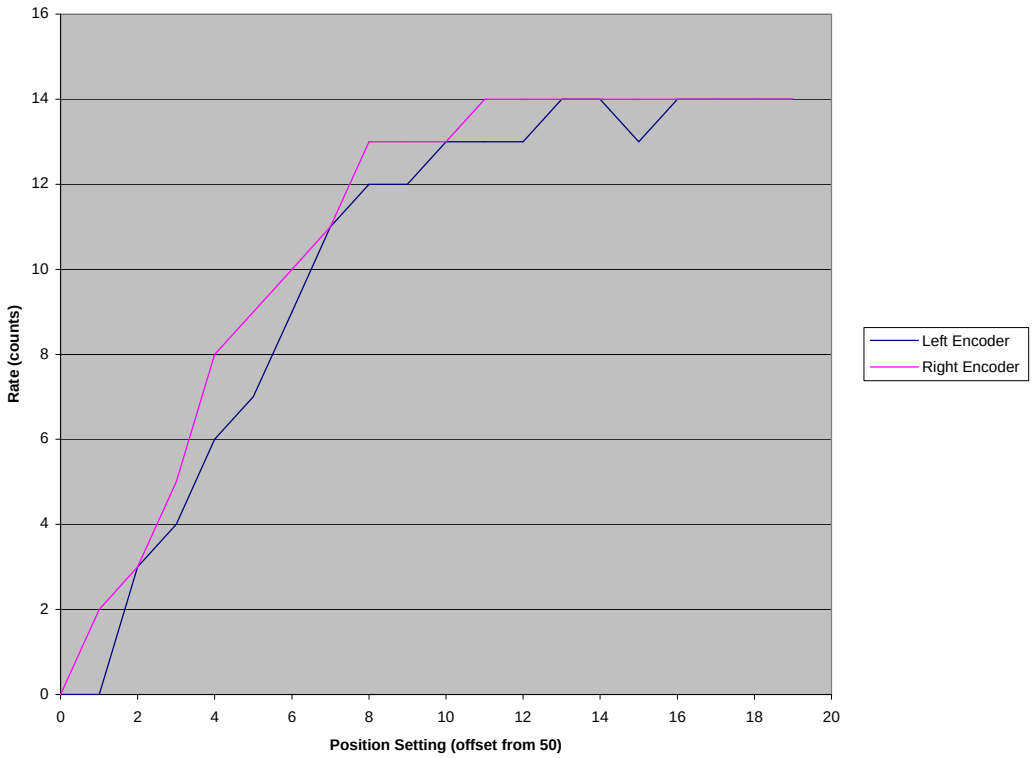

Figure 5 - Servo Speed Verses Position Input Setting

With a continuous rotation servo, setting the position to the midpoint, setPosition(50), reduces the power to zero. Hence, this is the neutral point because there is no power applied to the servo shaft. Setting the position to a value greater than 50 causes the servo shaft to turn in the forward direction. Setting the position to a value less than 50 causes the servo shaft to turn in the reverse direction.

Unfortunately, the power output by the servos included with the IntelliBrain-Bot kit is not linearly related to the value you provide in the setPosition command. The servos reach their maximum power and speed long before the position value reaches the extremes of the input range, 0 and 100. Figure 5 shows this.

The data charted in Figure 5 was collected using the TestServoResponse class, which is listed in Appendix B. This chart shows that the wheels reach maximum speed when the position value reaches 14 units above or below the neutral point, which is 50 percent. Therefore, the effective range of the position setting is roughly 50±14 (36 to 64). Varying the position setting within this range will affect the wheel speed. Additional variation beyond this range will have no effect.

The RoboJDE class library defines a generic motor interface named “Motor.” You will need to create a Motor façade around the Servo class by creating a new class that implements Motor interface and translates setPower commands to

setPosition commands. This will allow your navigator to control the servos as if they were motors. And, more importantly it will allow your navigator class to be used with robots that use conventional DC motors as well as modified servos.

Using RoboJDE, create a new class named ContinuousRotationServo with the following code:

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

public class ContinuousRotationServo implements Motor {
    private Servo mServo;
    private boolean mReverse;
    private int mRange;
    private DirectionListener mDirectionListener;

    public ContinuousRotationServo(Servo servo,
                        boolean reverse, int range) {
        mServo = servo;
        mReverse = reverse;
        mRange = range;
        mDirectionListener = null;
    }

    public ContinuousRotationServo(Servo servo,
                        boolean reverse, int range,
                        DirectionListener directionListener) {
        mServo = servo;
        mReverse = reverse;
        mRange = range;
        mDirectionListener = directionListener;
    }

    public void setDirectionListener(
                        DirectionListener directionListener) {
        mDirectionListener = directionListener;
    }

    public void brake() {
        // servos don't provide braking, just turn the power off
        setPower(Motor.STOP);
    }

    public void setPower(int power) {
        if (mDirectionListener != null)
            if (power != 0)
                mDirectionListener.updateDirection(power > 0);
        if (mReverse)
            power = -power;

        if (power == 0) {
            mServo.off();
            return;
        }
        else if (power > Motor.MAX_FORWARD)
```

```
            power = Motor.MAX_FORWARD;
        else if (power < Motor.MAX_REVERSE)
            power = Motor.MAX_REVERSE;

        mServo.setPosition(
                    (power * mRange) / Motor.MAX_FORWARD + 50);
    }
}
```

The setPower method contains the most interesting code in this class. This method sets the power using the following line of code:

```
mServo.setPosition((power * mRange) / Motor.MAX_FORWARD + 50);
```

The mRange variable is the effective range above and below the neutral value, 50. According to the previous analysis, mRange should be 14 for the servos included in the IntelliBrain-Bot kit. Rather than hard coding the value 14, it is preferable to provide for it to be specified in the ContinuousRotationServo class's constructor. This will allow the class to be reused with other continuous rotation servos that may have somewhat different characteristics.

If a DirectionListener has been configured, the setPower method updates it with the motor direction. You will use this to communicate the motor direction to your AnalogShaftEncoder objects. The following lines provide this function:

```
if (mDirectionListener != null)
    if (power != 0)
        mDirectionListener.updateDirection(power > 0);
```

The two servos mount in opposite directions on your robot's chassis. The sense of direction of rotation of the left servo's shaft is opposite of that of the right servo. The following lines of code allow the sense to direction to be reversed:

```
if (mReverse)
    power = -power;
```

By reversing the sense of rotation of the right servo, your program can use positive power values to rotate either wheel forward or negative power values to rotate either wheel backward.

The setPower method handles the special case of turning the servo off when the power is set to zero. It also limits the range of the power variable with the following code:

```
if (power == 0) {
    mServo.off();
    return;
}
else if (power > Motor.MAX_FORWARD)
    power = Motor.MAX_FORWARD;
```

```
else if (power < Motor.MAX_REVERSE)
    power = Motor.MAX_REVERSE;
```

Implementing Navigation Classes

As discussed previously, you can incorporate navigation into your robot's software by implementing a navigator class that provides the following methods:

1. moveTo - move to a specified location,
2. turnTo - turn to face a particular direction,
3. go - move continuously in one direction, and
4. stop.

With these four methods you will be able to program your robot to navigate freely on a flat surface. Since there are many ways you could implement a class that provides these basic navigation functions, it's a good idea to define a "Navigator" interface. This will allow you to create interchangeable navigation classes that use different techniques to provide basic navigation functions. By using an interface, you will help ensure the coupling between your navigation class and rest of your program will be loose. This will enable you to experiment with other methods of navigation without having to rework large portions of your program. It will even allow you to fundamentally change the mechanical design of your robot – for example, switch to using a four wheeled vehicle which steers like a car – without having to undertake a total rewrite of your program.

Use RoboJDE to create the Navigator interface as follows:

```
public interface Navigator {
    public void moveTo(float x, float y, boolean wait);
    public void moveTo(float x, float y,
                       NavigatorListener listener);
    public void turnTo(float heading, boolean wait);
    public void turnTo(float heading,
                       NavigatorListener listener);
    public void go(float heading);
    public void stop();
}
```

One variation of the moveTo and rotateTo methods in the Navigator interface provides a "wait" argument. This argument indicates whether the method should return immediately or wait for the operation to complete. The second variation allows for the methods to return immediately and later notify to a NavigatorListener when the operation completes or is cancelled.

Use RoboJDE to create the NavigatorListener interface using the following code:

```
public interface NavigatorListener {
    public void navigationOperationTerminated(
                                    boolean completed);
}
```

The navigationOperationTerminated method of the listener will be called when the navigation operation completes or is cancelled. The completed argument will be true if the operation completed or false if the operation was cancelled.

Implementing a Differential Drive Robot Navigator Class

You must take into account the mechanics of your robot when you design your navigator. The mechanical design determines how your navigator steers your robot. Designing a navigator for a robot that has four wheels and steers like a car is significantly different from designing a navigator for a differential drive robot. Fortunately, the Navigator interface isn't tied to one specific mechanical design or another. It is generic and defines methods that are equally relevant to navigating robots that use significantly different mechanical designs.

Since your IntelliBrain-Bot robot is based on the very common differential drive design, it is preferable to implement a navigator that is suitable to any differential drive robot rather than restrict it to just your IntelliBrain-Bot robot.

Use RoboJDE to create a DifferentialDriveNavigator class. This class should extend the Thread class and implement the Navigator interface, as follows:

```
public class DifferentialDriveNavigator extends Thread
                                        implements Navigator {
}
```

Your navigator needs to support four states: 1) moving to a particular point, 2) rotating to face a certain direction, 3) going straight ahead and 4) stopped. Your navigator will also need to repeatedly execute the navigation calculations, allowing it to make frequent adjustments required to keep it on course. The following code accomplishes these things:

```
public void run() {
    try {
        while (true) {
            switch (mState) {
            case MOVE_TO:
                goToPoint();
                break;
            case GO:
                goHeading();
                break;
            case ROTATE:
                doRotate();
                break;
            default:    // stopped
                break;
            }
            Thread.sleep(mPeriod);
        }
    }
    catch (Throwable t) {
```

```
            t.printStackTrace();
        }
    }
```

Now you have defined the high-level operation of your navigator. Next, you will need to implement the details by creating the goToPoint, goHeading and doRotate methods.

Going Straight Ahead with Proportional Control

The function of the goHeading method is quite simple. It needs to check the current heading and adjust the power it applies to the motors.

Use the following code to implement the goHeading method:

```
    private synchronized void goHeading() {
        Pose pose = mLocalizer.getPose();
        float error = mTargetHeading - pose.heading;
        if (error > PI)
            error -= TWO_PI;
        else if (error < -PI)
            error += TWO_PI;

        int differential = (int)(mGain * error + 0.5f);

        mLeftMotor.setPower(mDrivePower - differential);
        mRightMotor.setPower(mDrivePower + differential);
    }
```

Note, this code implements the proportional control algorithm discussed previously. In addition, it incorporates checks to constrain the error to be between -Pi and Pi, because the largest magnitude error is Pi or -Pi, when your robot is heading in the opposite direction it should be heading.

Navigating to a Specific Location

Now that your robot has the ability to steer itself in a particular direction, it will not be hard to implement the goToPoint method, allowing it to navigate to a specific location. Going to a specific location is a simple matter of keeping your robot headed toward the destination and then stopping once it arrives.

In the previous section you solved the problem of keeping your robot headed in the right direction. All you need to add are: 1) the calculation to determine the heading to follow from its current position to its destination and 2) a check to stop when your robot reaches its destination.

Your goToPoint method must determine the heading from your robot's current position to its destination. This method needs to calculate the arc tangent of xError / yError to determine the heading. The RoboJDE class library provides the Math.atan2 method, which you can use to calculate the heading given xError and yError.

Once your goToPoint method updates the heading, it can rely on the goHeading method to do the steering.

Finally, your goToPoint method will need to check if the sum of the absolute values of xError and yError is small enough to indicate your robot is in the proximity of its destination.

Implement the goToPoint method as follows:

```
private synchronized void goToPoint() {
    Pose pose = mLocalizer.getPose();
    float xError = mDestinationX - pose.x;
    float yError = mDestinationY - pose.y;

    float absXError = (xError > 0.0f) ? xError : -xError;
    float absYError = (yError > 0.0f) ? yError : -yError;
    if ((absXError + absYError) < mGoToThreshold) {
        // stop
        mLeftMotor.setPower(Motor.STOP);
        mRightMotor.setPower(Motor.STOP);
        mState = STOP;

        // notify listener the operation is complete
        updateListener(true, null);

        // signal waiting thread we are at the destination
        notify();
    }
    else {
        // adjust heading and go that way
        mTargetHeading = (float)Math.atan2(yError, xError);
        goHeading();
    }
}
```

Rotating in Place

The final navigation method you need to implement is doRotate. This method's purpose is to rotate your robot in place. Similar to the goToPoint method, this method will use data from the localizer, as well as the target heading to determine which direction to rotate and when to stop.

Implement the doRotate method as follows:

```
private synchronized void doRotate() {
    Pose pose = mLocalizer.getPose();
    float error = mTargetHeading - pose.heading;
    // choose the direction of rotation that results
    // in the smallest angle
    if (error > PI)
        error -= TWO_PI;
    else if (error < -PI)
```

```
            error += TWO_PI;
        float absError = (error >= 0.0f) ? error : -error;
        if (absError < mRotateThreshold) {
            mLeftMotor.setPower(Motor.STOP);
            mRightMotor.setPower(Motor.STOP);
            mState = STOP;

            // notify listener the operation is complete
            updateListener(true, null);

            // signal waiting thread we are at the destination
            notify();
        }
        else if (error > 0.0f) {
            mLeftMotor.setPower(-mRotatePower);
            mRightMotor.setPower(mRotatePower);
        }
        else {
            mLeftMotor.setPower(mRotatePower);
            mRightMotor.setPower(-mRotatePower);
        }
    }
```

Coordinating Threads

You've kept your control logic simple and avoided undue coupling between the navigator and other components by using a dedicated thread to execute the navigation code. This enables your navigator to pilot your robot without awkward consideration for your robot's other tasks, such as sampling sensors.

If you were to implement your entire program using a single thread, your navigator would need to include logic to take into consideration the priority and timing constraints of other tasks. For example, you would need to consider if the arc tangent calculation takes so much time that it could cause your wheel encoder to miss counts. This issue is easy to address with multi-threading; you simply give the wheel encoder threads a higher priority than the navigator thread. Then the wheel encoder threads will preempt the navigator thread when they are ready to run. This moves the scheduling burden to the underlying RoboJDE virtual machine – simplifying your software. The virtual machine will automatically take care of preempting the navigator thread when a wheel encoder thread needs to take a quick peek at its sensor.

Multi-threading helps simplify scheduling, but in order for it to work properly, you must consider the interaction between threads. Your main thread will provide high-level control of your robot and rely on your navigator to carry out specific operations to accomplish the commands issued via the go, moveTo, turnTo and stop methods. These are just simple methods that set the goal of the navigator. The run method carries out these operations. You can implement the moveTo method as follows:

```
    private synchronized void moveTo(float x, float y, boolean wait,
                                     NavigatorListener listener) {
```

```
        updateListener(false, listener);
        mDestinationX = x;
        mDestinationY = y;
        mState = MOVE_TO;
        if (wait) {
            try {
                wait();
            }
            catch (InterruptedException e) {}
        }
    }
```

Your program will call this method as necessary to give the navigator its next goal. This method is called by a thread other than your navigator's thread; therefore, you must coordinate these threads to control access to the data they share.

The variables that define the goal – mDestinationX, mDestinationY, mTargetHeading and mState – are accessed by the navigator thread and any threads that call the go, moveTo, turnTo and stop methods. These variables must be changed and read as a consistent set; otherwise, the navigator thread could resume execution at an inopportune time and use some values from the new goal and other values from the old goal. For example, the navigator might use mDestinationX from a new goal and mDestinationY from the previous goal. This would result in the navigator steering toward an unintended location, the combination of the new destination's x coordinate and the old destination's y coordinate. This wouldn't be desirable, especially if heading toward this unintended destination happened to lead your robot over a cliff!

Fortunately, Java makes it easy to solve this problem by providing the "synchronized" keyword. Adding this keyword to a method or a block of code allows the virtual machine to act like a traffic cop, only allowing one thread into synchronized code at a time. By adding the synchronized keyword to the methods: go, moveTo, turnTo, stop, goHeading, goToPoint and doRotate you can ensure the shared variables will always be accessed and manipulated as a consistent set.

One other thread coordination caveat you must consider is allowing a calling thread to wait while the navigator carries out the requested command. This will allow a thread calling moveTo or turnTo to wait for the operation to complete before continuing to execute. Once again, Java provides a means to do this, wait and notify methods. The moveTo method makes use of the wait method. This tells the virtual machine the thread must wait until another thread notifies the virtual machine that the waiting thread can continue. The notify method is used by the goToPoint and doRotate methods. This tells the virtual machine to notify a waiting thread, if there is one, so it can resume execution.

You will need to add the remaining navigation methods, which are similar to those already discussed:

```
public void moveTo(float x, float y, boolean wait) {
    moveTo(x, y, wait, null);
}

public void moveTo(float x, float y,
                   NavigatorListener listener) {
    moveTo(x, y, false, listener);
}

public synchronized void turnTo(float heading, boolean wait,
                                NavigatorListener listener) {
    updateListener(false, listener);
    mTargetHeading = normalizeAngle(heading);
    mState = ROTATE;
    if (wait) {
        try {
            wait();
        }
        catch (InterruptedException e) {}
    }
}

public void turnTo(float heading, boolean wait) {
    turnTo(heading, wait, null);
}

public void turnTo(float heading, NavigatorListener listener) {
    turnTo(heading, false, listener);
}

public synchronized void go(float heading) {
    mTargetHeading = normalizeAngle(heading);
    mState = GO;
    updateListener(false, null);
}

public synchronized void stop() {
    mLeftMotor.setPower(Motor.STOP);
    mRightMotor.setPower(Motor.STOP);
    mState = STOP;
    updateListener(false, null);
}
```

In addition, you will need to add an import statement, member variables, a constructor and several utility methods.

Import

```
import com.ridgesoft.robotics.Motor;
```

Member Variables

```
private static final float PI = 3.14159f;
private static final float TWO_PI = PI * 2.0f;
```

```
    private static final float PI_OVER_2 = PI / 2.0f;

    private static final int STOP = 0;
    private static final int GO = 1;
    private static final int MOVE_TO = 2;
    private static final int ROTATE = 3;

    private Motor mLeftMotor;
    private Motor mRightMotor;
    private Localizer mLocalizer;
    private int mDrivePower;
    private int mRotatePower;
    private int mPeriod;
    private int mState;
    private float mDestinationX;
    private float mDestinationY;
    private float mTargetHeading;
    private float mGain;
    private float mGoToThreshold;
    private float mRotateThreshold;
    private NavigatorListener mListener;
```

Constructor

```
    public DifferentialDriveNavigator(
                Motor leftMotor, Motor rightMotor,
                Localizer localizer
                int drivePower, int rotatePower,
                float gain,
                float goToThreshold, float rotateThreshold,
                int threadPriority, int period) {
        mLeftMotor = leftMotor;
        mRightMotor = rightMotor;
        mLocalizer = localizer;
        mDrivePower = drivePower;
        mRotatePower = rotatePower;
        mGain = gain;
        mGoToThreshold = goToThreshold;
        mRotateThreshold = rotateThreshold;
        mPeriod = period;
        mState = STOP;
        mListener = null;
        setPriority(threadPriority);
        setDaemon(true);
        start();
    }
```

Utility Methods

```
    private void updateListener(boolean completed,
                                NavigatorListener newListener) {
        if (mListener != null)
            mListener.navigationOperationTerminated(completed);

        mListener = newListener;
```

```
    }

    private float normalizeAngle(float angle) {
        while (angle < -PI)
            angle += TWO_PI;
        while (angle > PI)
            angle -= TWO_PI;
        return angle;
    }
```

Integrating Navigation Classes into Your Program

You have implemented all of the classes your robot will need to navigate. All that remains to be done is integrating them into your program and writing a few test functions; then you can give your navigator a try.

Constructing ContinuousRotationServo Objects

Picking up where you left off with your MyBot program from the *Enabling Your Robot to Keep Track of its Position* tutorial, your next step is to add a ContinuousRotationServo class for each servo. This will create a façade around each servo enabling your navigator to interface to the servos as if they were conventional motors. Add the following code to the MyBot class just after the construction of the OdometricLocalizer class:

```
ContinuousRotationServo leftMotor =
            new ContinuousRotationServo(leftServo, false, 14,
            (DirectionListener)leftEncoder);
ContinuousRotationServo rightMotor =
            new ContinuousRotationServo(rightServo, true, 14,
            (DirectionListener)rightEncoder);
```

The second argument to your ContinuousRotationServo class is a Boolean value that indicates whether the sense of direction should be reversed. You must reverse the sense of direction for the right servo. The third argument is the effective range of servo position commands, which was determined previously to be 14.

Constructing a Navigator Object

Add the following lines of code to the main method to construct a DifferentialDriveNavigator object:

```
Navigator navigator = new DifferentialDriveNavigator(
                                    leftMotor, rightMotor,
                                    localizer,
                                    8, 6, 25.0f, 0.5f, 0.08f,
                                    Thread.MAX_PRIORITY - 2, 50);
```

The arguments following the localizer argument control the behavior of the navigator. The first argument is the average power the navigator will use when moving forward. The next argument is the power the navigator will use when

rotating in place. This is followed by the gain, which controls how aggressively the navigator responds to heading errors. The next two arguments define the thresholds which control how close is close enough to the goal when moving forward and rotating in place.

Following the control constants is the thread priority argument. It is a good idea to set the priority for the navigator slightly lower than the priority for the shaft encoders, which are highest priority, and the localizer, which is next to highest priority. The navigator relies on the shaft encoders and the localizer to perform its function. It doesn't make sense to give it a higher priority than other threads it depends upon. Furthermore, the encoder objects will miss count if their polling of the sensors gets delayed for too long. Polling the encoder sensors is not a large task, but it does have timing constraints that must be met in order for the encoders to function properly. Therefore, it makes sense to give the encoders the highest priority.

The final argument to the constructor specifies the update period in milliseconds. Fifty milliseconds is a reasonable value to use.

Testing Your Navigator

You can test your navigator by implementing a few small test classes that use the navigator to maneuver your robot in specific patterns. Creating the following four test classes will help verify the function of your navigator:

1) NavigateForward – move your robot straight ahead for a specified distance
2) Rotate – rotate your robot 180 degrees in place
3) NavigateSquare – move your robot in a square pattern
4) NavigateFigureEight – moves your robot in an figure eight pattern

NavigateForward

This class will manuever your robot straight ahead for the distance specified in the constructor.

```
public class NavigateForward implements Runnable {
    private Navigator mNavigator;
    private float mDistance;

    public NavigateForward(Navigator navigator, float distance) {
        mNavigator = navigator;
        mDistance = distance;
    }

    public void run() {
        mNavigator.moveTo(mDistance, 0.0f, true);
    }

    public String toString() {
```

```
            return "Navigate Forward";
        }
    }
```

Rotate

This class will rotate your robot 180 degrees.

```
    public class Rotate implements Runnable {
        private Navigator mNavigator;

        public Rotate(Navigator navigator) {
            mNavigator = navigator;
        }

        public void run() {
            mNavigator.turnTo((float)Math.toRadians(180), true);
        }

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

NavigateSquare

This class will maneuver your robot in a square according to the size specified in the constructor.

```
    public class NavigateSquare implements Runnable {
        private Navigator mNavigator;
        private float mSize;

        public NavigateSquare(Navigator navigator, float size) {
            mNavigator = navigator;
            mSize = size;
        }

        public void run() {
            mNavigator.moveTo(mSize, 0.0f, true);
            mNavigator.moveTo(mSize, -mSize, true);
            mNavigator.moveTo(0.0f, -mSize, true);
            mNavigator.moveTo(0.0f, 0.0f, true);
            mNavigator.turnTo(0.0f, true);
        }

        public String toString() {
            return "Square";
        }
    }
```

NavigateFigureEight

This class will navigate your robot in a figure-eight pattern.

```
public class NavigateFigureEight implements Runnable {
    private Navigator mNavigator;
    private float mHeight;
    private float mWidth;

    public NavigateFigureEight(Navigator navigator,
                               float height, float width) {
        mNavigator = navigator;
        mHeight = height;
        mWidth = width;
    }

    public void run() {
        mNavigator.moveTo(mHeight, mWidth, true);
        mNavigator.moveTo(mHeight, 0.0f, true);
        mNavigator.moveTo(0.0f, mWidth, true);
        mNavigator.moveTo(0.0f, 0.0f, true);
        mNavigator.turnTo(0.0f, true);
    }

    public String toString() {
        return "Figure Eight";
    }
}
```

Extend the list of selectable functions in your MyBot class by adding the following code:

```
new NavigateSquare(navigator, 16.0f),
new NavigateFigureEight(navigator, 48.0f, 32.0f),
new Rotate(navigator),
new NavigateForward(navigator, 100.0f),
```

Testing

You can now test out your navigation software by experimenting with the four classes you just created. When you choose the “Navigate Forward” function your robot should drive straight ahead for 100 inches. When you choose the “Rotate” function your robot should rotate in place 180 degrees. Choosing the “Navigate Square” function should cause your robot to drive in a square which is 16 inches on a side. Finally, the “Figure Eight” function should cause your robot to navigate a shape that is roughly a figure eight.

You should find that your robot is reasonably effective at navigating these preprogrammed patterns. However, you will also find the precision of your robot’s navigation is far from perfect! Table 1 lists a number of the most significant sources of error that affect the precision with which our robot can navigate.

Table 1 – Major Sources of Error

Error Sources	Description
Calibration	The accuracy of the wheel diameter and wheel base measurements provided to the localizer affect its accuracy.
Encoder Quantization	The accuracy to which the wheel encoders are able to measure the position of the wheels directly affects the accuracy of the localizer. Using wheel encoders that provide more counts per revolution, such as the WheelWatcher WW-01 encoder, will reduce the error due to encoder quantization.
Wheel Slippage	Any slippage of the wheels will result in localization errors.
Localization Technique	The dead reckoning localization method you have used is based on self-centric encoder measurements. This results in accumulation of error because the robot has no fixed external reference from which it could recalibrate its position. Incorporating external references such as landmarks, the Earth's magnetic field (compass) or satellites (GPS) would help improve the accuracy of the localizer.

Conclusion

By completing this tutorial you have implemented two new software components, a Motor façade for continuous rotation servos and a Navigator for a differential drive robot. These are both generic classes that you can reuse for other robot projects. They are not specific to just the IntelliBrain-Bot robot. By using the Motor and Navigator interfaces, as well as multi-threading, you have been able to minimize the coupling of these components to other software, hardware and electronics components, making it easy to reuse them for other robot projects.

You also discovered that your robot is less than perfect. This is not just a shortcoming of your robot; imperfection is a fundamental issue all roboticists contend with. While it is tempting to try to produce a robot that is error free, no matter how hard you try, you can only reduce the built-in errors, you can't eliminate them, and you can't do anything to prevent unpredictable random errors. Creating strategies to deal with the real world limitations of robots will provide robotics researchers an assortment of interesting problems to solve that will keep them busy for many years to come!

Exercises

1. Experiment with each of the four navigation test functions you created. Does your robot navigate the pattern you expect it to navigate? Does your robot navigate the pattern perfectly each time?
2. Using the “Navigate Forward” function, measure the average distance your robot moves over several runs. Calibrate the value of the wheel diameter such that your robot moves an average of 100 inches over several runs.
3. Using the “Rotate” function, calibrate the track width value such that your robot is as accurate as possible at rotating an average of 180 degrees over several runs.
4. Modify your program so your robot travels 50 inches instead of 100 inches forward when you choose the Navigate Forward function.
5. Modify your program so your robot travels in a square which is 50 inches on a side when you choose the Navigate Square function. Is your robot more able to get back to its starting point when the square is larger or smaller? Why?
6. Program your robot to navigate a new shape that you choose.
7. Try increasing and decreasing the stop thresholds your navigator uses. How does increasing the stop threshold affect your robot’s navigation? How does decreasing the stop threshold affect your robot’s navigation?
8. Change the drive power. How does this affect your robot’s navigation?
9. Change the rotate power. How does this affect your robot’s navigation?
10. Double the gain constant. How does this affect your robot’s navigation?
11. Halve the gain constant. How does this affect your robot’s navigation?
12. Use the TestNavigatorResponse class in Appendix A to collect data and create a chart similar to Figure 4. Hint: Add an instance of this class as another function in the list of functions in your MyBot class.
13. Use the TestServoResponse class in Appendix B class to collect data and create a chart similar to Figure 5. Hint: Add an instance of this class as another function in the list of functions in your MyBot class.

Appendix A – TestNavigatorResponse Class

```
import com.ridgesoft.robotics.PushButton;

public class TestNavigatorResponse implements Runnable {
    Navigator mNavigator;
    Localizer mLocalizer;
    PushButton mButton;

    public TestNavigatorResponse(Navigator navigator,
                                 Localizer localizer,
                                 PushButton button) {
        mNavigator = navigator;
        mLocalizer = localizer;
        mButton = button;
    }

    public void run() {
        float desiredHeading = 0.0f;
        mNavigator.go(desiredHeading);
        int servoSetting = 0;
        float[] target = new float[50];
        float[] heading = new float[50];
        long nextTime = System.currentTimeMillis();
        for(int i = 0; i < heading.length; ++i) {
            nextTime += 125;

            try {
                Thread.sleep(
                    nextTime - System.currentTimeMillis());
            } catch (InterruptedException e) {}
            Pose pose = mLocalizer.getPose();
            heading[i] = pose.heading;
            target[i] = desiredHeading;
            if (i == 20) {
                desiredHeading = (float)Math.PI / 2.0f;
                mNavigator.go(desiredHeading);
            }
        }
        mNavigator.stop();

        while (!mButton.isPressed());
        for (int i = 0; i < target.length; ++i) {
            System.out.println(Integer.toString(i * 125) + '\t' +
                               target[i] + '\t' + heading[i]);
        }
    }

    public String toString() {
        return "Nav. Response";
    }
}
```

Appendix B – TestServoResponse Class

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

public class TestServoResponse implements Runnable {
    private ShaftEncoder mLeftEncoder;
    private ShaftEncoder mRightEncoder;
    private Servo mLeftServo;
    private Servo mRightServo;
    private PushButton mButton;

    public TestServoResponse(Servo leftServo,
                             Servo rightServo,
                             ShaftEncoder leftEncoder,
                             ShaftEncoder rightEncoder,
                             PushButton button) {
        mLeftServo = leftServo;
        mRightServo = rightServo;
        mLeftEncoder = leftEncoder;
        mRightEncoder = rightEncoder;
        mButton = button;
    }

    public void run() {
        int servoSetting = 0;
        int leftPrevious = mLeftEncoder.getCounts();
        int rightPrevious = mRightEncoder.getCounts();
        int[] leftSamples = new int[20];
        int[] rightSamples = new int[20];
        int[] powerSamples = new int[20];
        long nextTime = System.currentTimeMillis();
        for(int i = 0; i < powerSamples.length; ++i) {
            nextTime += 1000;
            try {
                Thread.sleep(
                    nextTime - System.currentTimeMillis());
            } catch (InterruptedException e) {}
            int leftCounts = mLeftEncoder.getCounts();
            int rightCounts = mRightEncoder.getCounts();
            powerSamples[i] = servoSetting;
            leftSamples[i] = leftCounts - leftPrevious;
            leftPrevious = leftCounts;
            rightSamples[i] = rightCounts - rightPrevious;
            rightPrevious = rightCounts;
            servoSetting += 1;
            mLeftServo.setPosition(50 + servoSetting);
            mRightServo.setPosition(50 - servoSetting);
        }
        mLeftServo.setPosition(50);
        mRightServo.setPosition(50);
        while (!mButton.isPressed());
        for (int i = 0; i < powerSamples.length; ++i) {
            System.out.println(Integer.toString(i) +
                               '\t' + powerSamples[i] +
```

```
                                        '\t' + leftSamples[i] +
                                        '\t' + rightSamples[i]);
            }
        }

        public String toString() {
            return "Servo Response";
        }
    }
```


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