# Lab 2: Sense And Avoid

## Learning Outcomes

- Read sonar and encoder sensors
- Move Robot
- Use state-machines and state diagrams

## Section 1: Requirements and Design

In this lab, you will build a rosnode that senses objects and avoids them by backing up, turning, and then continuing forward.

At all times the robot must maintain a safe speed. You must be able to stop the robot remotely if it is going to collide with an object, robot, or person.

The robot should start by moving forward until it senses an object is less than 30 cm in front of it. Then the robot should reverse and go backwards until the object is more than 60 cm away from the object. Now the robot can turn 90 degrees and proceed moving forward again. This process should continue until you kill the rosnode.

#### State Machines

A useful technique for building robotic systems is a finite-state machine. A finite-state machine has multiple states that the robot can be in. When an event occurs (i.e. a sensor reads a specific value, the user inputs something, or a timer expires) the robot can transition from one state to another. When implementing robot behavior, modeling the system as a finite-state machine allows you to develop each state independently so that you can isolate that logic for testing. Finite-state machines are modeled with state diagrams. See Figure 1 is a state diagram for the Sense and Avoid lab. Subsequent labs may require you to construct a state diagram prior to implementing the code.

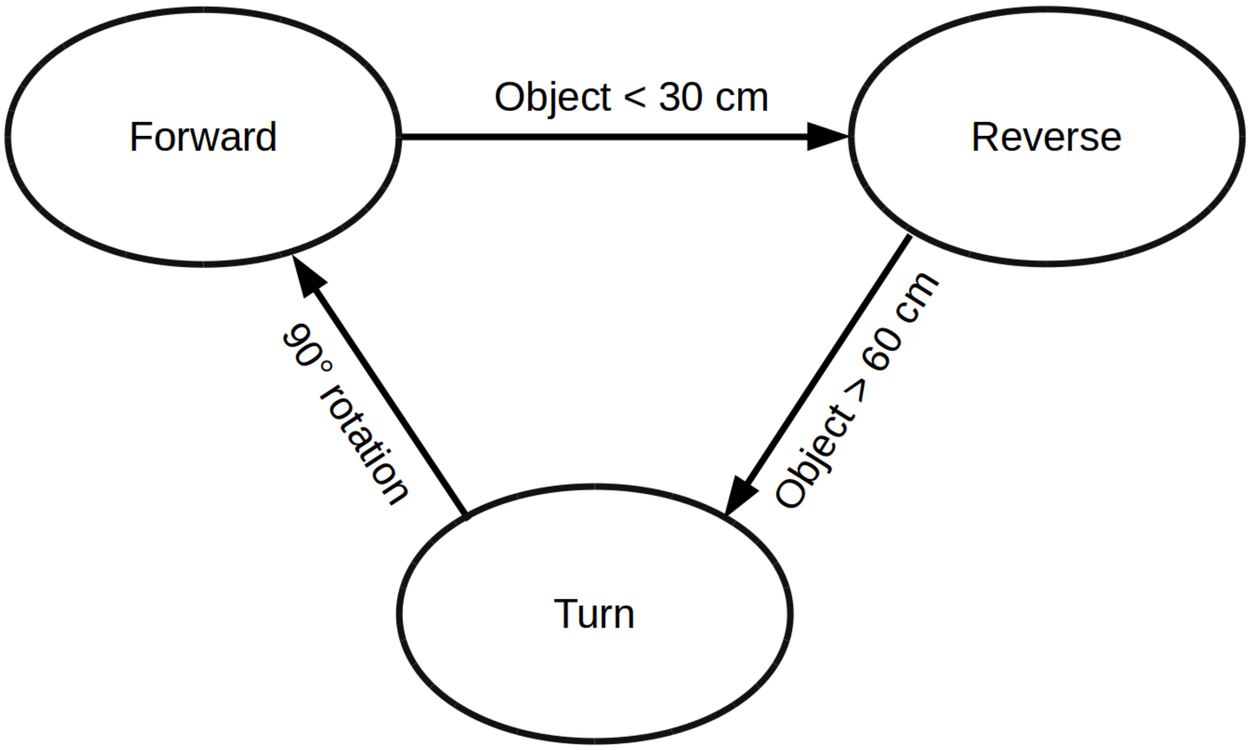

Figure 1: Sense And Avoid State Diagram

## Section 2: Installing the Lab

To perform this lab, you will need to get the sense_and_avoid template into your catkin workspace. First ensure that your Jet has internet access by connecting it using WiFi or ethernet. Next ssh into Jet and enter the following command:

```
wget http://http://nvidiagputeachingkit.github.io/rosjet/static/labs/lab2/lab2_sense_and_avoid-code.zip
```

Now unzip the lab:

```
unzip lab2_sense_and_avoid-code.zip -d ~/catkin_ws/src/jetlabs/lab2_sense_and_avoid
```

Delete the zip file:

```
rm lab2_sense_and_avoid-code.zip
```

To build the code, use the following command when you are in `~/catkin_ws/`:

```
catkin_make --pkg lab2_sense_and_avoid && source devel/setup.sh
```

Run the program with roslaunch:

```
roslaunch lab2_sense_and_avoid lab2.launch
```

## Section 3: Reading Sensor Values

Sensor values are published to specific ROS topics that we can subscribe to. In the second lab on ROS, you used `rostopic echo` to display the messages in the terminal. Now we will listen to these messages with functions. Any interaction with the ROS environment requires the initialization of a NodeHandle. A

NodeHandle refers to the node that we are creating. With the NodeHandle, you can publish your own topics or services, subscribe to topics or services, and read/set ROS paramters. A NodeHandle can be constructed with no arguments or a string. If you use a string in the constructor, then all new topics or services or parameters created by the node will be under the namespace of the string. We will use the default NodeHandle constructor with no namespace, but you can refer to the ROS Docs for more advanced usage (http://wiki.ros.org/roscpp/Overview/NodeHandles). The line below creates a new NodeHandle for our ROS node:

```
ros::NodeHandle nh;
```

Now that the NodeHandle has been initialized, the node is ready to subscribe to topics. Each topic that we want to listen to has it's own subscriber. The method to subscribe to a topic is NodeHandle::subscribe. This method is a generic method, so we can specify the type of the topic between angle braces (this is often not necessary because the compiler can infer the type, but it makes the code more readable). The arguments to subscribe are as follows: - topic (string): name of the topic - queue_size (unsigned int): number of messages to store before messages get discarded - callback (function): function/method that is called when a message is received - instance (pointer to object): a pointer an instance of the class that the callback method belongs to.

Subscribing to the second sonar (middle) sensor is shown below:

```
ros::Subscriber sonar_sub;
sonar_sub = nh.subscribe<std_msgs::Int16>("/arduino/sonar_2",
  10,
  &SenseAndAvoid::sonarCallback,
  this);
```

Notice that 10 was chosen as the queue_size. This number will not be significant for our node because the sonar_1 topic publishes regularly and not rapidly; for topics that can burst many messages, a longer queue length could be beneficial so messages are not discarded.

Here is a stub for the `sonarCallback` method that displays the distance:

```
void SenseAndAvoid::sonarCallback(const std_msgs::Int16::ConstPtr& msg)
{
  ROS_INFO("sonar_1: %d", msg->data);
}
```

The method accepts a pointer to a message of type Int16. To access the integer stored in the message, you can read the `data` field as shown. Other message types can have more complicated message data, so you will need to look up the documentation on those types before writing callbacks.

A similar process can be used for reading messages from the encoders.

### Task: Using Encoder Data

The encoders publish integer messages that correspond to the number of ticks since the robot started.

The SenseAndAvoid class already has two fields (left_count and right_count) that can be used to store the number encoder ticks. Replace the contents of `leftEncoderCallback` and `rightEncoderCallback` with code that updates the values of left_count and right_count.

### Task: Responding to Sonar

In this task we will use the sonar messages to detect an object that is within 30 cm and change the state of the robot if this occurs. The sonar messages returns the distance to an object in centimeters. The 0 value means that the sonar was not able to determine the distance since the sonar echo never returned, so this generally happens when the closest object is more than 200 cm away.

Rewrite the sonarCallback to print “REVERSE” if the robot is in the FORWARD state and it detects an object that is within 30 cm. Make sure you exclude cases when the sonar message is 0. When the object is detected, you should also change the state of the robot to REVERSE.

## Section 4: Publishing to Motors

Instead of publishing directly to the motor topics, we will publish to another topic that specifies the velocity of the robot. Another node converts these velocity messages into motor commands. This abstraction makes it easier to operate the motors and ensures that our code could be ported to another robot platform with minimal modifications since we are only specifying velocities.

The velocities are published to the cmd_vel topic, which has the type of Twist. Twist is a geometric message with a linear and angular component. We will use the linear component’s x value to specify the forward and reverse speed of the robot, and we will use the angular components’s z value to specify the rotation of the robot.

### Task: Moving Forward

Add an `else if` conditional to the `sonarCallback` to check if the state is FORWARD. In this conditional, add the following code:

```
vel_msg.linear.x = LINEAR_SPEED;
vel_msg.angular.z = 0;
```

This code sets the linear speed and maintains the rotational velocity at 0. Both the linear.x and angular.z values are floating point numbers. The value of linear.x is the velocity is m/s of the robot (a negative values corresponds to going in reverse). The value of angular.z is the rotational speed of the robot in rad/s. Setting both values to 0 will stop the robot.

Inside the conditional that checks for an object within 30cm, implement the code that stops the robot (set the velocities to zero). Now the robot should continue forward until it detects an object then stop. Also add the line:

```
last_count = left_count;
```

This code will let us determine how long the robot has been in the reverse state.

To test this code, make sure that you have a way of stopping the robot if it does not operate as expected. You can enter this command in the terminal to stop the robot:

```
rostopic pub /cmd_vel geometry_msgs/Twist \
'{linear: {x: 0, y: 0, z: 0}, angular: {x: 0,y: 0,z: 0}}' --once
```

You can change the LINEAR_SPEED and ANGULAR_SPEED values at the top of sense_and_avoid.cpp to make the robot faster or slower.

## Section 5: Finishing Sense and Avoid

### Task: Turning

Add another `else if` conditional inside `sonarCallback` that checks whether the state is REVERSE, the object is more than OBJECT_DIST_SAFE, and the left_count has registered more than ticks than REVERSE_DURATION since the robot began moving backwards. Inside this conditional, print “TURN” and set the state of the robot to turn. Set the angular.z velocity to -ANGULAR_SPEED and the linear.x velocity to LINEAR_SPEED/2. Also, reset the last_count variable to left_count because we will use them to track how far we have turned.

**Task: Forward Again**

Add a final `else if` conditional inside `sonarCallback` that is executed when the state is TURN and the left_count is greater than `last_count + TURN_COUNT`. When this occurs, the robot should print “FORWARD”, change its state to FORWARD, and begin moving forward again.

Now modify the conditional that changes the state to REVERSE by setting the linear.x value to -LINEAR_SPEED. Feel free to modify the TURN_COUNT constant to obtain an appropriate turn angle. Demonstrate successful operation of your robot by launching this node and holding a folder or piece of cardboard in front of the robot to see it reverse, turn, and continue forward again.

## Challenge:

Add callbacks for the other two sonar sensors and make the robot turn away from objects that are detected on the sides of the robot.