Laboratory 3
Mobile Robot Perception

CSC 232 / ECE 232 / CSC 432 / ECE 437 - Autonomous Mobile Robots

February 22th, 2016 - March 16th, 2016

Academic Honesty

The following laboratory exercises and and the laboratory report must be performed in accordance with the University of Rochester's Academic Honesty Policy. You may discuss the laboratory exercises with others, however all of the code, data, and material contained within the report must be your own in accordance with the laboratory instructions. Please type or write "I affirm that I have not given or received any unauthorized help on this assignment, and that all work is my own." at the beginning of the laboratory report. Please contact Professor Howard at thomas.howard@rochester.edu if you have any questions about the academic honesty policy for this laboratory.

Report Due Date

This document describes a two week long laboratory due on March 16th, 2016. The laboratory report shall be written in LaTeX, Word, Pages, or the text editor of your choice and submitted as a PDF with a tarball containing the source code through the course website on Blackboard. The laboratory report shall contain a description of the activities taken in each step of the laboratory and answer all questions posed at the end of this document.

Overview

Laboratory 3 will build upon the code that you developed in Laboratory 1 and 2 to enhance the robot simulator with perception capabilities and a graphical interface to simulate the depth sensor of the TurtleBot robot. By the end of this laboratory, you should be able to:

- Simulate the uncertain measurements from a beam model
- Construct a graphical user interface (GUI) based on Qt and C++
- Draw lines and points in OpenGL
- Compile a QT-based GUI using CMake
- Modify a ROS launch file
- Visualize the robot pose and the sensor measurements of your simulator

Requirements

This laboratory has four parts. First, you will modify your existing simulator to simulate the response of a depth sensor. Second, you will construct a graphical user interface to visualize your simulated depth sensor and robot pose from Laboratory 2. Third, you will write a new ROS launch file to bring up the depth sensor interface on the robot. Lastly, you will run a series of tests to evaluate the differences between your simulated depth sensor and the physical hardware in the Questions section.

Sensor Simulator

In Chapter 6 of Probabilistic Robotics a method of modeling range finders call the *Beam Model* is discussed. In this laboratory we will implement a simplified form of this model that only assumes Gaussian noise about the real measurement. With respect to the weighted mixture model described in Equation 6.12 in Probabilistic Robotics and again in the quick reference at the end of this laboratory instruction handout, this means that $z_{hit} = 1$ and $z_{short} = z_{max} = z_{rand} = 0$ and that only the measurement model in Equation 6.4 is required to be implemented. The variance of this distribution σ_{hit} should be given as a command-line argument in the *simulator* executable. The sensor must now also calculate the range between the true pose in the simulator and various objects in the simulated world. The geometry of these objects are given in the questions at the end of the laboratory. In this laboratory, the true range of the objects can be calculated geometrically. To complete your sensor model, create a sensor_msgs::LaserScan message by sampling the range calculated between a minimum and maximum sensor angle. The discretization of your LaserScan measurements and minimum and maximum sensor angles shall be command line inputs into your simulator.

Graphical User Interface

In addition to modifying your simulator, Laboratory 3 will require to implement a graphical user interface (GUI) program. The GUI shall be written in Qt/C++ and illustrate the pose of the robot and the range measurements of the robot. The pose of the robot and the range measurements of the robot shall both be report in the world frame. An outline for the code for the header, source, and main programs is seen below. As you will see, we construct a class called “GUI” that inherits from a base class in Qt called “QGLWidget”. This class starts a timer in the class constructor that calls the callback function “timer_callback” at a fixed rate. This callback function calls “ros::spinOnce” to handle incoming messages whose subscriber callbacks have been described as class functions. The “paintEvent” function is where the on-screen drawing of the state held inside the class is performed. The current function for “paintEvent” draws a x-y-z coordinate system (+x is a red line, +y is a green line, +z is a blue line) at the origin. The robot pose shall be drawn as a box, circle, or point at the robot $x - y$ position and the sensor measurement shall be drawn as a series of lines that emanate outwards from the robot origin.

Listing 1: code framework for header (gui.h)

```
#include <iostream>
#include "ros/ros.h"
#include "nav_msgs/Odometry.h"
#include "sensor_msgs/LaserScan.h"
#include <QtGui/QApplication>
#include <QtGui/QWidget>
#include <QtOpenGL/QGLWidget>
#include <QtCore>
#include <QTimer>
#include <GL/glu.h>

class GUI: public QGLWidget {
    Q_OBJECT
  public:
    GUI( QWidget * parent = NULL );
    virtual ~GUI();
        void handle_laserscan( const sensor_msgs::LaserScan::ConstPtr& msg );
        void handle_odom( const nav_msgs::Odometry::ConstPtr& msg );

        QTimer* timer;

  protected slots:
        void timer_callback( void );

  protected:
    virtual void initializeGL();
    virtual void paintGL();
  };
```

Listing 2: code framework for source (gui.cc)

```
#include "gui.h"

GUI::
GUI( QWidget * parent ) : QGLWidget( parent ), timer() {
  setMinimumSize( 600, 600 );

  connect( timer, SIGNAL( timeout() ), this, SLOT( timer_callback() ) );
  timer->start( 100 );  // call timer_callback at 0.1 Hz (period is 100 ms)
}

GUI::
~GUI() {

}

void
GUI::
handle_laserscan( const sensor_msgs::LaserScan::ConstPtr& msg ){
  // implement storing of laserscan message here
  return;
}

void
GUI::
handle_odom( const nav_msgs::Odometry::ConstPtr& msg ){
  // implement storing of robot pose here
  return;
}

void
GUI::
timer_callback( void ){
  ros::spinOnce(); // Process the messages in here
  return;
}

void
GUI::
initializeGL(){
  glClearColor( 1.0, 1.0, 1.0, 1.0 );
  gluOrtho2D(-5,5,-5,5);
  return;
}

void
GUI::
paintGL(){
  glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
  glLoadIdentity();
  // draw a coordinate system at the origin
  glBegin( GL_LINES );
  glColor4f( 1.0, 0.0, 0.0, 1.0 );
  glVertex3f( 0.0, 0.0, 0.0 );
  glVertex3f( 1.0, 0.0, 0.0 );
  glColor4f( 0.0, 1.0, 0.0, 1.0 );
  glVertex3f( 0.0, 0.0, 0.0 );
  glVertex3f( 0.0, 1.0, 0.0 );
  glColor4f( 0.0, 0.0, 1.0, 1.0 );
  glVertex3f( 0.0, 0.0, 0.0 );
  glVertex3f( 0.0, 0.0, 1.0 );
  glEnd();

  // implement drawing of laserscan and robot pose here
  return;
}
```

Listing 3: code framework for main program (gui_process.cc)

```
#include <QtGui/QApplication>
#include "gui.h"
#include "gui_process_cmdline.h"

using namespace std;

int
main( int argc,
      char* argv[] ){
  gengetopt_args_info args;
  cmdline_parser( argc, argv, &args );
  QApplication app( argc, argv );
  ros::init( argc, argv, "gui" );
  ros::NodeHandle node_handle;
  GUI gui;
  ros::Subscriber subscriber_reset_odometry = node_handle.subscribe( "/
     laserscan", 1, &GUI::handle_laserscan, &gui );
  ros::Subscriber subscriber_odom = node_handle.subscribe( "/odom", 1, &GUI::
     handle_odom, &gui );
  gui.show();
  return app.exec();
}
```

To build this code, you will have to modify your CMakeLists.txt file in several ways, which include finding the QT headers and libraries.

Listing 4: modifications to the CMakeLists.txt file

```
# find the QT libraries
find_package(Qt4 COMPONENTS QtCore QtGui QtOpenGL REQUIRED)
find_package( OpenGL REQUIRED )
find_package( GLUT REQUIRED )

include(${QT_USE_FILE})
include_directories(${QT_INCLUDE_DIR} ${OPENGL_INCLUDE_DIR} )
# generate the MOC_SRCS from the QT header
qt_wrap_cpp(gui MOC_SRCS gui.h)
# generate the header and source files from the gengetopt file
execute_process(COMMAND ${GENGETOPT} -i ${CMAKE_CURRENT_SOURCE_DIR}/
   gui_process.ggo --file-name gui_process_cmdline --output-dir=${
   CMAKE_CURRENT_BINARY_DIR} --unamed-opts)
# generate the executable and link the libraries
add_executable(gui-process gui_process.cc gui.cc ${MOC_SRCS} ${
   CMAKE_CURRENT_BINARY_DIR}/gui_process_cmdline.c)
target_link_libraries(gui-process ${ROS_LIBRARIES} ${QT_LIBRARIES} ${
   OPENGL_LIBRARIES} )
install( TARGETS gui-process DESTINATION bin )
```

An illustration of the message passing between the three programs is illustrated in Figure 1.

ROS Launch File

A ROS launch file is a file that starts various processes included in the ROS TurtleBot codebase. To start the depth sensor in a mode that simulates a planar laser rangefinder, we need to modify the base “minimal.launch” ROS launch file that was used in Laboratory 1. To do this, we write a new ROS launch file called “lab3.launch” as follows:

```
<launch>
<include file="$(find␣turtlebot_bringup)/launch/minimal.launch"/>
<node pkg="depthimage_to_laserscan" type="depthimage_to_laserscan" name="
   depthimage_to_laserscan">
  <param name="image" value="/camera/depth/image_raw"/>
</node>
</launch>
```

To initialize the depth sensor on the robot, run the following command:

```
roslaunch lab3 lab3.launch
```

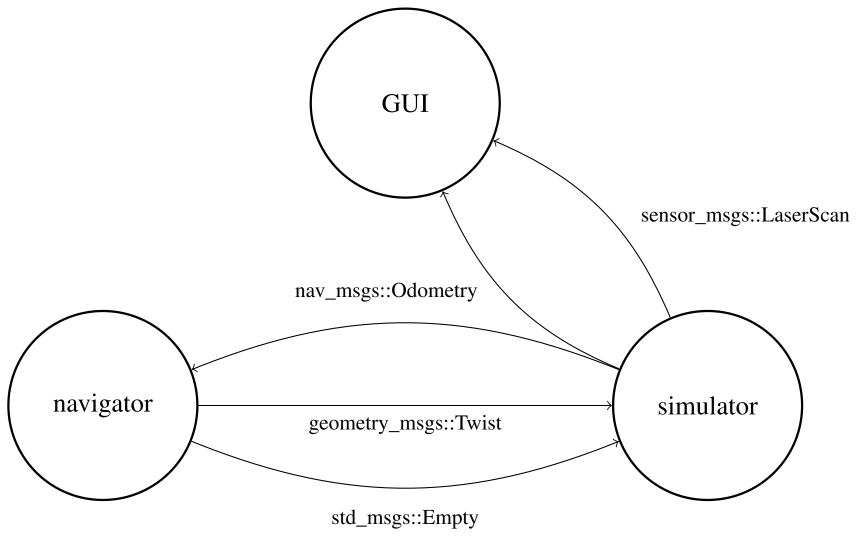

Figure 1: A diagram of the message passing between the *navigator*, *simulator*, and *GUI* programs in Laboratory 3

Code Checklist

- ☐ ***simulator* function to calculate the range measurements for a given geometry**
- ☐ ***simulator* function to sample from beam model for the range measurements**
- ☐ ***simulator* sensor_msgs::LaserScan publisher for the "/scan" channel**
- ☐ ***simulator* paintEvent function to draw the incoming range measurements and pose**

- ☐ ***GUI* nav_msgs::Odometry subscriber for the "/odom" channel**
- ☐ ***GUI* sensor_msgs::LaserScan subscriber for the "/scan" channel**
- ☐ ***GUI* handle_laserscan function to save the incoming range measurements**
- ☐ ***GUI* handle_odom function to save the incoming odometry measurements**

Questions

1. Modify your simulator to reflect the geometry illustrated in Figure 2. In your GUI, collect 10.0 seconds of data recorded at 10.0 Hz. Assume a $\sigma_{hit} = 0.1$ for your beam model, the minimum and maximum laserscan sensor angle is $\pm 45°$ and resolution of the sensor is 64 beams, $\alpha_1, \alpha_2, \ldots, \alpha_6 = 0.01$ for your velocity motion model. In Matlab, plot the range measurement as a function of time and the beam angle.

2. Now test your GUI with a physical setup in the laboratory that reflects the same environment as in Figure 2. Again, collect 10.0 seconds of data recorded at 10.0 Hz and plot the range measurement as a function of time and the beam angle in Matlab.

3. Discuss the differences you observed between the simulated and physical environments, in particular the noisiness of the range measurements and the size of the observed code.

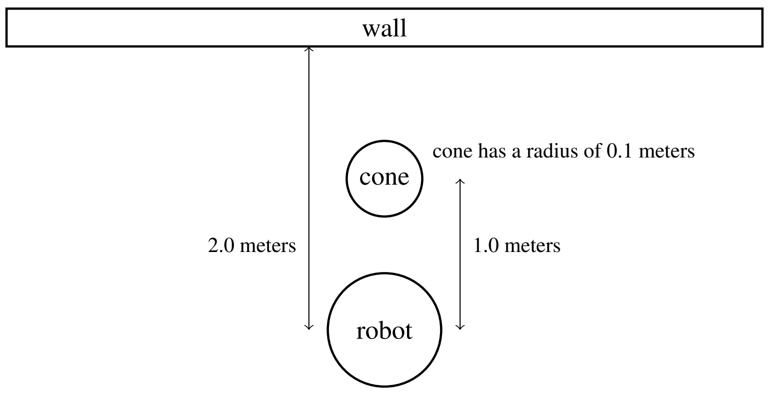

Figure 2: A diagram showing the simulated and physical test environments

Quick Reference

This section provides a quick reference for some of the mathematics that are necessary to complete the laboratory. These equations may also be found in Chapter 6 of Probabilistic Robotics or the books held on reserve in the library.

The hit mode for the beam model is:

$$p_{hit}\left(z_t^k | x_t, m\right) = \begin{cases} \eta \mathcal{N}\left(z_t^k; z_t^{k*}, \sigma_{hit}^2\right) & \text{if} \quad 0 \le z_t^k \le z_{max} \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

The short mode for the beam model is:

$$p_{short}\left(z_t^k | x_t, m\right) = \begin{cases} \eta \lambda_{short} e^{-\lambda_{short} z_t^k} & \text{if} \quad 0 \le z_t^k \le z_t^{k*} \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

The failure mode for the beam model is:

$$p_{failure}\left(z_t^k | x_t, m\right) = \begin{cases} 1 & \text{if} \quad z_t^k = z_{max} \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

The random error mode for the beam model is:

$$p_{rand}\left(z_t^k | x_t, m\right) = \begin{cases} \frac{1}{z_{max}} & \text{if} \quad 0 \le z_t^k \le z_{max} \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

The weighted mixture model for the beam model is:

$$p\left(z_t^k | x_t, m\right) = \begin{pmatrix} z_{hit} \\ z_{short} \\ z_{max} \\ z_{rand} \end{pmatrix}^T \begin{pmatrix} p_{hit}\left(z_t^k | x_t, m\right) \\ p_{short}\left(z_t^k | x_t, m\right) \\ p_{max}\left(z_t^k | x_t, m\right) \\ p_{rand}\left(z_t^k | x_t, m\right) \end{pmatrix} \tag{5}$$