Assignment 1

1. (30%) Creating a random walk algorithm

Now that you are familiar with the fundamentals of ROS and the stage simulator, you will learn to create a simple ROS node that will drive the robot around, much like a Roomba robot vaccum cleaner. More specifically, the robot should move forward until it reaches an obstacle, then rotate in place for a random amount of time, then move forward again and repeat.

First type the following in a terminal: (for your own benefits, make sure you understand what each of these lines is doing)

```
> cd ~/ros_workspace
> roscreate-pkg random_walk roscpp geometry_msgs sensor_msgs
> cd random_walk
> echo 'rosbuild_add_executable(${PROJECT_NAME} src/random_walk.cpp)' >>
CMakeLists.txt
> touch src/random_walk.cpp
```

Now go into the random_walk/src folder and paste the following code into random_walk.cpp:

<random_walk.cpp>

To build and run this code, type the following in a terminal:

```
> rosmake random_walk
> rosrun random_walk random_walk
```

First you have to have run roscore and the stage simulator in different terminal; for example:

```
> roscore
> rosrun stage stageros cave_single.world
Or
> rosrun stage stageros autolab_single.world
```

Different world files can be found in Worlds.tgz.

You should start seeing range values being printed back to you in the terminal. If you use your mouse to drag the blue square robot in the Stage window, you will see the range values change.

Your task is now to implement a simple random walk algorithm: if the robot is moving sufficiently close to an obstacle in front of it, then rotate in-place for a specific duration;

otherwise move forward by default. Do this by filling in the 3 sections labelled TODO in random_walk.cpp.

_____ <random_walk.cpp> _____

```
#include "ros/ros.h"
#include "geometry_msgs/Twist.h"
#include "sensor_msgs/LaserScan.h"
#include <cstdlib> // Needed for rand()
#include <ctime> // Needed to seed random number generator with a time value


class RandomWalk {
public:
  // Construst a new RandomWalk object and hook up this ROS node
  // to the simulated robot's velocity control and laser topics
  RandomWalk(ros::NodeHandle& nh) :
      fsm(FSM_MOVE_FORWARD),
      rotateStartTime(ros::Time::now()),
      rotateDuration(0.f) {
    // Initialize random time generator
    srand(time(NULL));

    // Advertise a new publisher for the simulated robot's velocity command topic
    // (the second argument indicates that if multiple command messages are in
    //  the queue to be sent, only the last command will be sent)
    commandPub = nh.advertise<geometry_msgs::Twist>("cmd_vel", 1);

    // Subscribe to the simulated robot's laser scan topic and tell ROS to call
    // this->commandCallback() whenever a new message is published on that topic
    laserSub = nh.subscribe("base_scan", 1, &RandomWalk::commandCallback, this);
  };


  // Send a velocity command
  void move(double linearVelMPS, double angularVelRadPS) {
    geometry_msgs::Twist msg; // The default constructor will set all commands to 0
    msg.linear.x = linearVelMPS;
    msg.angular.z = angularVelRadPS;
    commandPub.publish(msg);
  };


  // Process the incoming laser scan message
```

```
  void commandCallback(const sensor_msgs::LaserScan::ConstPtr& msg) {
    if (fsm == FSM_MOVE_FORWARD) {
      // Compute the average range value between MIN_SCAN_ANGLE and
MAX_SCAN_ANGLE
      //
      // NOTE: ideally, the following loop should have additional checks to ensure
      //       that indices are not out of bounds, by computing:
      //
      //       - currAngle = msg->angle_min + msg->angle_increment*currIndex
      //
      //       and then ensuring that currAngle <= msg->angle_max
      unsigned int minIndex = ceil((MIN_SCAN_ANGLE_RAD - msg->angle_min) / msg-
>angle_increment);
      unsigned int maxIndex = ceil((MAX_SCAN_ANGLE_RAD - msg->angle_min) / msg-
>angle_increment);
      float closestRange = msg->ranges[minIndex];
      for (unsigned int currIndex = minIndex + 1; currIndex < maxIndex; currIndex++) {
        if (msg->ranges[currIndex] < closestRange) {
          closestRange = msg->ranges[currIndex];
        }
      }
      ROS_INFO_STREAM("Range: " << closestRange);

      // TODO: if range is smaller than PROXIMITY_RANGE_M, update fsm and rotateStartTime,
      //       and also choose a reasonable rotateDuration (keeping in mind of the value
      //       of ROTATE_SPEED_RADPS)
      //
      // HINT: you can obtain the current time by calling:
      //
      //       - ros::Time::now()
      //
      // HINT: you can set a ros::Duration by calling:
      //
      //       - ros::Duration(DURATION_IN_SECONDS_FLOATING_POINT)
      //
      // HINT: you can generate a random number between 0 and 99 by calling:
      //
      //       - rand() % 100
      //
      //       see http://www.cplusplus.com/reference/clibrary/cstdlib/rand/ for more details
      ////////////////////// ANSWER CODE BEGIN ////////////////////

      ////////////////////// ANSWER CODE END ////////////////////
    }
```

```
  };


  // Main FSM loop for ensuring that ROS messages are
  // processed in a timely manner, and also for sending
  // velocity controls to the simulated robot based on the FSM state
  void spin() {
    ros::Rate rate(10); // Specify the FSM loop rate in Hz

    while (ros::ok()) { // Keep spinning loop until user presses Ctrl+C
      // TODO: Either call:
      //
      //       - move(0, ROTATE_SPEED_RADPS); // Rotate right
      //
      //       or
      //
      //       - move(FORWARD_SPEED_MPS, 0); // Move foward
      //
      //       depending on the FSM state; also change the FSM state when appropriate
      ////////////////////// ANSWER CODE BEGIN ////////////////////

      ////////////////////// ANSWER CODE END ////////////////////

      ros::spinOnce(); // Need to call this function often to allow ROS to process incoming
messages
      rate.sleep(); // Sleep for the rest of the cycle, to enforce the FSM loop rate
    }
  };

  enum FSM {FSM_MOVE_FORWARD, FSM_ROTATE};

  // Tunable parameters
  // TODO: tune parameters as you see fit
  const static double MIN_SCAN_ANGLE_RAD = -10.0/180*M_PI;
  const static double MAX_SCAN_ANGLE_RAD = +10.0/180*M_PI;
  const static float PROXIMITY_RANGE_M = 2.0; // Should be smaller than
sensor_msgs::LaserScan::range_max
  const static double FORWARD_SPEED_MPS = 1.0;
  const static double ROTATE_SPEED_RADPS = M_PI/2;


protected:
  ros::Publisher commandPub; // Publisher to the simulated robot's velocity command topic
  ros::Subscriber laserSub; // Subscriber to the simulated robot's laser scan topic
```

```
  enum FSM fsm; // Finite state machine for the random walk algorithm
  ros::Time rotateStartTime; // Start time of the rotation
  ros::Duration rotateDuration; // Duration of the rotation
};


int main(int argc, char **argv) {
  ros::init(argc, argv, "random_walk"); // Initiate new ROS node named "random_walk"
  ros::NodeHandle n;
  RandomWalk walker(n); // Create new random walk object
  walker.spin(); // Execute FSM loop

  return 0;
};
```

2. (50%) Occupancy grid mapping

Create a program that would implement Occupancy Grid Mapping. Use the random walk implemented in question 1 to have a single robot wander around the environment. Collect the laser data and use them to implement an occupancy grid. Use the provided code as a guideline on how to store and display an occupancy grid in the form of an image. In addition display the path of the robot. Remember to consider *Occupied, Unoccupied* and *Unknown* Cells.

Please find the complete package under grid_mapper.zip. Compile the code using rosmake. The grid_mapper code can be run as:

```
>rosrun grid_mapper grid_mapper 60 60
```

The last two parameters indicate the dimensions of the world.

3. (20%) Occupancy grid mapping using real data

Use the provided logged data publisher instead of a stageros simulation to construct and display an occupancy grid. The data are from an early experiment with a hokuyo laser range finder mounted on top of a TurtleBot 2 robot. The robot pose as published comes also from real data and thus is corrupted by odometric error. In addition display the path of the robot.

The real data are store in a bag file. Run:

```
> rosbag play left-corridor-mapping.bag
```

Use the rostopic list command to see the topics published during the play of the bag. The topics of interest for the grid mapper are:

> \odom

or

> \odom_combined
> \scan

Run the mapper for the two different odometry measurements and comment on the differences.

Note:

There is a difference between image coordinates and Stage coordinates. See sample code and implement accordingly.
You need to have opencv 2.x and some boost packages installed. For the sample code:

- press SPACEBAR to save snapshot of occupancy grid canvas into folder where user opened the executable in.
- press X or x to quit.
- read over the code, and pay attention to the 2 TODO comments especially

Other new things in the project, i.e. why did I have to include a zip of the entire project:

- CMakeLists.txt has some lines needed to link project with Boost and OpenCV libraries
- manifest.xml has ros_dep line that asks project to depend on EXTERNALLY INSTALLED OpenCV library (recommended by ROS, see http://www.ros.org/wiki/opencv2)

For all questions prepare a written report with a screen (images saved in your program; see attached code) that document the progress of the mapping. Compare the two approaches

What to submit:

Prepare a report presenting screenshots of your work, provide a brief overview on how the program works and also discuss the choices made. Email the report together with the source code to the TA.