Assignment 1

The objective of this assignment is for you to become familiar with the Robot Operating System (ROS) software, which you will be using in future assignments to program various algorithms for robots.

1. Installing and learning ROS

ROS is officially supported on Ubuntu Linux, so if you would like to install Ubuntu on your own computer, we highly recommend that you install it inside an Ubuntu 11.04 environment. From this point onwards, we will assume that you are installing ROS on Ubuntu.

Please follow these instructions to install ROS; you will want to install the ros-diamondback-desktop-full version. You should have created a folder called “ros_workspace” in your home folder during the installation process.

The Ubuntu Linux computers in Trottier 3rd floor should all have ROS installed already, so you may choose to use those machines to do your assignments on.

The previous ROS installation page is in fact the first item in a comprehensive tutorial on learning ROS. Please follow through all of the **Beginner level** tutorials on this page. You may skip items 11 and 14, as we will not be using Python in this course.

2. Installing and learning ROS stage package

In the first few assignments we will be working with a simulated robot equipped with a horizontal sweeping laser scanner. Therefore, in addition to installing the core ROS packages, you will need to install the stage robot simulator. Depending on what version of ROS you’ve installed previously, you may already have this stage package already. To find out, type the following in a terminal:

```
> rosmake stage
```

If you have stage installed, it should report something along the lines of:

```
> [ rosmake ] Built 37 packages with 0 failures.
```

If you do not see this line, then you must install the stage package. If you have super-user permissions on your machine, type the following in a terminal:

```
> sudo apt-get install ros-diamondback-simulator-stage
> rosmake stage
```

If you do not have super-user permissions, then type the following in a terminal instead:

```
> cd ~/ros_workspace
> svn co https://code.ros.org/svn/ros-pkg/stacks/stage/trunk
```

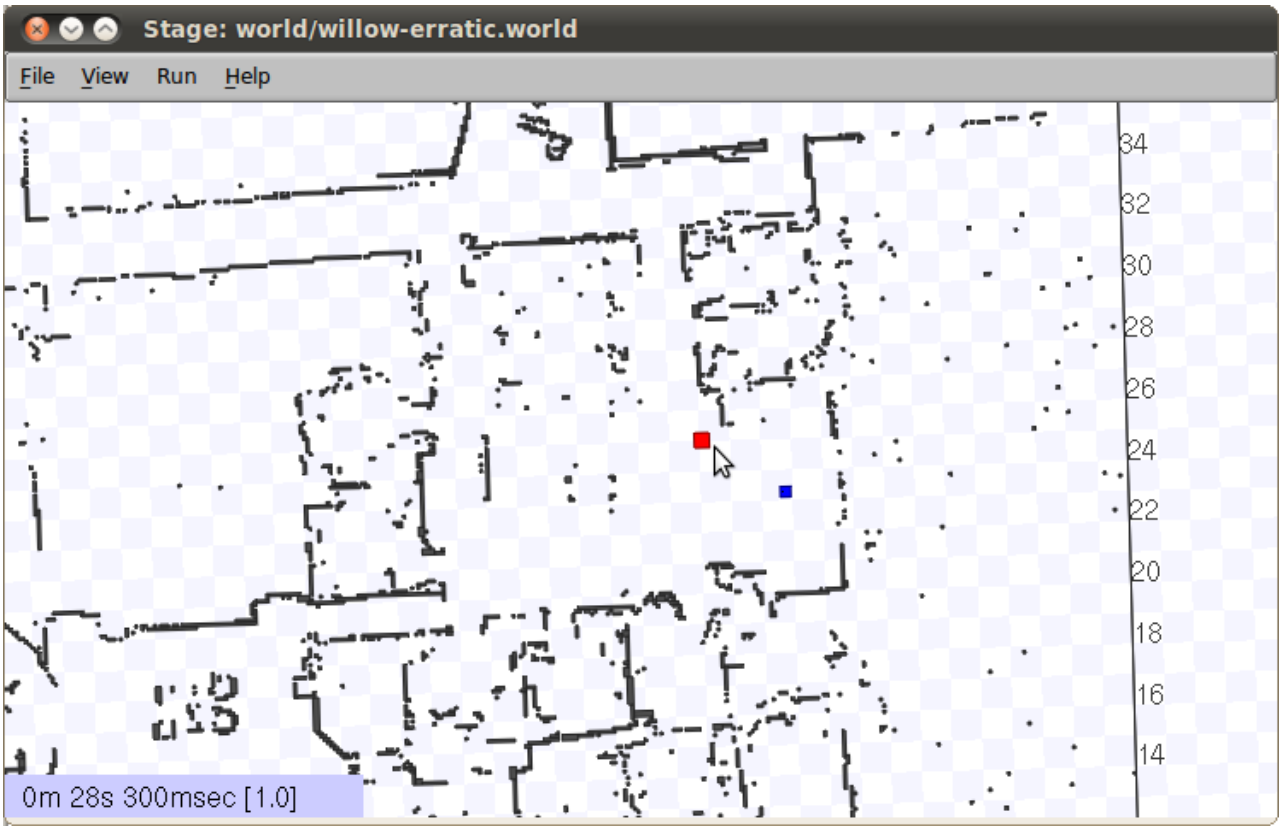
> rosmake stage

After installing and building the ROS stage package, you will need to install a few more custom packages that you will use to learn how ROS stage works. Type the following in a terminal:

> cd ~/ros_workspace
> svn co https://code.ros.org/svn/wg-ros-pkg/stacks/pr2_controllers/branches/pr2_controllers-1.4/control_toolbox
> rosmake control_toolbox
> svn co https://code.ros.org/svn/wg-ros-pkg/branches/trunk_cturtle/sandbox/teleop_base teleop_base
> rosmake teleop_base

Now follow these instructions (from step 3 onwards) to run stage visualization and keyboard control. Keep in mind that you will need to execute roscore, stageros, teleop_base, and rviz simultaneously using separate terminals.

You will see two robots on the Stage window: a red square and a blue square. Your commands will affect the blue robot only.



3. 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 are 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
```

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;
};
```