Randy Lewis

Cooperative Turtlebot Room Mapping

Abstract

The goal of this project was to write an algorithm to coordinate multiple Turtlebots using ROS to map a room quickly and efficiently. This quarter, besides minimally contributing to basic hardware and software setup, I wrote two nodes, one that makes the Turtlebots move, and another one that accesses their visual data. Both nodes work in similar ways, creating ROS objects to fulfill the functions of publishing and subscribing, and then using these objects over and over again in a loop until the user ends the program. The main problems that have not yet been solved are operating system related failures, and errors in example programs. Overall, this quarter we managed to get most of the basics set up; we next need to write the actual core of the program.

Big Picture

The Cortes and Martinez lab is using ROS (a library written within C++ and Python to give robot developers a generic tool for programming their robots) and Turtlebots (commercially available ROS compatible robots that are essentially upside down buckets with wheels, a laptop, and a variety of sensors) to explore cooperative coordination algorithms.

This quarter I worked on a project with Daniel Heideman to use ROS to coordinate multiple Turtlebots to cooperatively map a room with their Kinect sensors and then to localize themselves in that map. An algorithm already exists to map a room with one Turtlebot (SLAM Gmapping), so the focus of the project was to harness the power of multiple robots to make the mapping process faster and more accurate.

A parallel project, pursued by Katherine Liu, aimed to write a similar algorithm to map a room and localize Turtlebots in it, using one fixed overhead camera rather than the various Kinect sensors on the Turtlebots. Although initially, both of these projects are intended to stand alone, in the end they may each be merged in order to improve each other's accuracy and ease of execution.

My Contributions

My contribution to the project this quarter included writing two nodes: one to make multiple Turtlebots simultaneously drive in a circle, and one to access the data from multiple Turlebots' Kinect sensors.

The first node worked by publishing name specific velocity commands to all the robots, causing them to drive in circles. Although the set of commands to specifically drive in a circle may or may not later prove useful, the idea was to have an example program from which to copy that could programmatically make different Turtlebots move simultaneously (and although the node as is makes all Turtlebots drive in identical circles, it could easily be adapted to different simultaneous movements).

The second node subscribed to some of the Kinect data coming from the Turtlebots. The specific topics chosen were the ones that Rviz (a ROS node built for visualizing robots and their surroundings) subscribes to by default, so as to allow for easy interface with the visual program later in the project. This subscriber node was also written to be robot specific, and thus can distinguish data published from one Turtlebot from another.

In order for both of these nodes to work, the Turtlebot's startup files had to be modified so that the topics they published and subscribed to were name specific. Although Daniel did the majority of the work on this front (by figuring out how to make this modification in the first place, and by doing it for the topics

involving the robot's base (and thus its movement)), I extrapolated from his work to extend the name specific functionality to the 3d sensor data.
Both of these nodes are located on the mencia computer on the "randy" account in the directory /catkin_ws/src/turtlebot_room_mapper/src/, and can be viewed and run from there. Also, the code is included at the end of this report under the "referenced code" section.

Methodology

The circle publisher node works in the following manner:
1) The desired velocity topics are imported for later use in the program.
2) Inside the main class of the program, separate publishing objects are created for each Turtlebot's name specific velocity topic (for example, to create a Giottobot velocity publisher object, the program specifies that the topic the object publishes on is /giottobot/mobile_base/commands/velocity).
3) The program executes a loop wherin it waits until it has at least one subscriber to begin publishing (this is not actually necessary because the program publishes a stream of commands rather than one singular command, but the point of this program is to copy from later, and this sleep loop might be useful later).
4) After at least one robot has begun subscribing to its name specific velocity command, the program executes an infinite loop wherin it publishes a velocity command consisting of a linear and angular velocity (which can be varied depending on the desired radius and speed) at a certain predefined frequency. In order to make a smooth circle, the loop runs at a faster rate than it takes for each command to be executed so that the velocity commands are constantly being overwritten by the beginning of the next one.
5) The loop is exited when the user presses ctrl-c.
The Kinect data subscriber works similarly to the publisher, but with the complication of extra methods outside the main class.
1) All the desired topics are imported (such as, for example, nav_msgs/Odometry, and sensor_msgs/Laser_Scan) for later use in the program.
2) The program defines "callback" methods for each individual robot's data topics that will later be run each time the master computer receives a message over this topic from a robot. Currently, the callback methods just print out the data to the screen with the appropriate robot and topic name, but the next step for this program will be to concatenate/average the different data, then republish it in general form.
3) Inside the main class, subscriber objects are created for each robot for each desired topic (meaning that each subscriber object has a corresponding callback method).
4) An infinite loop is run, which subscribes to each desired topic from each robot, and then calls each individual callback method to print out each message (running this node gets a little messy in the command line, but it is just a proof of concept—the final program won't be printing each and every piece of data out).
5) The loop is exited when the user presses ctrl-c.

Tips

There are several tips I would give to anyone attempting to take over the project where we have left off:
1) Remember to add these lines to end of your bashrc file:
source /opt/ros/groovy/setup.bash //(or hydro or whatever version you're using)
source ~/catkin_ws/devel/setup.bash
If you forget these lines, things don't work, and it is very difficult to figure out what's wrong. Putting them in your bashrc runs them every time you open a new terminal window, so you basically never have to deal with them again.

2) Remember to modify your Cmakelists file to include paths to each node that you write. You would think ROS would automatically know where to look for them in compilation, but if you don't put these two lines in your Cmakelists file, trying to compile with “catkin make” will not work properly:
add_executable(<desired node name> src/<actual c++ file>.cpp)
target_link_libraries(<desired node name> ${catkin_LIBRARIES})
3) Make full use of the ROS tutorials and the ROS answer service contained on the ROS website here: http://wiki.ros.org/ROS/Tutorials and here: http://answers.ros.org/questions/. Although Google has been helpful to me in this project, it has not even come close to the combined power of these two sites to provide me with examples and answers to specific questions.

Pitfalls

Three main problems were encountered this quarter that we have not yet been solved. The first was a mistake contained in one of the Turtlebot software updates. The problem was that the updated software no longer contained the right nodes to publish the data from the Kinect sensors. Of course, since the entire project is based around using these sensors to map a room, this was a major complication. To work around this problem, Mike and Evan reinstalled the operating system on half of the Turtlebots in order to effectively unupdate them (and thus eliminate the update related problem). This worked in the sense that the unupdated Turtlebots published Kinect data, but was difficult to work with because having several Turtlebots and one master computer all in different states of update inevitably leads to some glitches and problems that could otherwise be avoided. This problem was almost solved at the end of the quarter when Evan further updated one of the Turtlebots from Groovy (the second newest version of ROS) to Hydro (the newest version). However, although the Turtlebot running Hydro ran the right nodes to publish Kinect data, it could not for some reason connect to the Kinect (as if it was not plugged in), which has not yet been solved.

The second unsolved problem was the inability to find a launch file command on the central computer that would launch other launch files or nodes on the Turtlebots (such as, most importantly, the name specific base and camera startup launch files). The only way to run these files is currently either to remotely connect to each Turtlebot using ssh, and run the programs from the command line, or to physically open the netbook laptops and run them from there. This is another vexing problem, because, without human setup, it eliminates the possibility of a stand-alone program on the master computer mediating all the Turtlebots and the possibility of dividing the processing power needed to run the final program between the various computers. Again, this problem seems to be due to an error on the part of Clearpath Robotics, since Evan said that he remembered being able to launch launch files on one computer from a launch file on another computer in Fuerte (a version of ROS preceding Groovy and Hydro), but that Groovy seemed to lack this capability. In the end, this problem may also be solved by updating and correctly configuring Hyrdo, but this has not yet been the case.

The final unsolved problem was the inability to reliably run the Gmapping algorithm and other working programs in conjunction with rviz to visualize the Turtlebot Kinect data. This problem was very difficult to deal with because it was very intermittent—one day running Gmapping and rviz with a Turtlebot would crash at one stage, and the next it would crash at another stage, with no apparent difference in usage. This problem was likely due at least in part to the various stages of somewhat incompatible updates on different Turtlebots and the master computer, but was not entirely explained by this phenomenon (for example, one identified glitch was that Gmapping would usually crash at a much

later stage in execution if we waited for about a minute between each startup launch file and between the launch files and running the algorithm).

Referenced Code

Publisher Node

```
#include "ros/ros.h"
//include the types of messages you'll be publishing
#include "std_msgs/String.h"
#include <geometry_msgs/Twist.h>
//include other stuff that's just generally required for publishing things
#include <stdlib.h>
#include <sstream>




int main(int argc, char **argv)
{
ros::init(argc, argv, "publisher");


//object to interact with ROS
ros::NodeHandle n;


//declare pi for easy turns in radians
float pi = 3.14159;


//create publisher object publishing data of type geometry_msgs/Twist on topic name
/mobile_base/commands/velocity, with buffer list size of 1000
ros::Publisher vPub = n.advertise<geometry_msgs::Twist>("/mobile_base/commands/velocity", 1000);
//publisher object specifically for giotto
ros::Publisher giottoPub =
n.advertise<geometry_msgs::Twist>("/giottobot/mobile_base/commands/velocity", 1000);
//publisher object specifically for bellini
ros::Publisher belliniPub =
n.advertise<geometry_msgs::Twist>("/bellinibot/mobile_base/commands/velocity", 1000);
//publisher object specifically for titian
ros::Publisher titianPub = n.advertise<geometry_msgs::Twist>("/titianbot/mobile_base/commands/velocity",
1000);
//publisher object specifically for boticelli
ros::Publisher boticelliPub =
n.advertise<geometry_msgs::Twist>("/boticellibot/mobile_base/commands/velocity", 1000);
```

```
ros::Rate loop_rate(10); //set loop rate to 10 hz


//check every .1 seconds for a subscriber, sleep while waiting, don't proceed until someone has subscribed
//while(vPub.getNumSubscribers() == 0) ros::Duration(0.1).sleep();


int count = 0;
while (ros::ok())
{
//stuff partial circle message with data, then publish it
geometry_msgs::Twist partialCircle;


partialCircle.linear.x=.5;
partialCircle.angular.z=pi/2;


ROS_INFO("partial circle #%d, linear:%f,angular:%f", count, partialCircle.linear.x,partialCircle.angular.z);


vPub.publish(partialCircle);
giottoPub.publish(partialCircle);
belliniPub.publish(partialCircle);
titianPub.publish(partialCircle);
boticelliPub.publish(partialCircle);


ros::spinOnce();


//this makes the loop run at the loop rate, whch is 10 hz, declared earlier
loop_rate.sleep();


++count;
}


return 0;
}
```

Subscriber Node

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <nav_msgs/Odometry.h>
#include <sensor_msgs/LaserScan.h>
#include <sensor_msgs/PointCloud2.h>
#include <sensor_msgs/Image.h>
#include <sensor_msgs/Imu.h>



//subscribing callback functions for specific robots


void giottoOdomCallback(const nav_msgs::Odometry locator)
{
ROS_INFO("Giotto Position: %f,%f", locator.pose.pose.position.x,locator.pose.pose.position.y);
ROS_INFO("Giotto Heading: linear:%f,angular:%f",
locator.twist.twist.linear.x,locator.twist.twist.angular.z);
}
void belliniOdomCallback(const nav_msgs::Odometry locator)
{
ROS_INFO("Bellini Position: %f,%f", locator.pose.pose.position.x,locator.pose.pose.position.y);
ROS_INFO("Bellini Heading: linear:%f,angular:%f",
locator.twist.twist.linear.x,locator.twist.twist.angular.z);
}
void giottoScanCallback(const sensor_msgs::LaserScan& scanner)
{
for (int i=0;i<scanner.ranges.size();i++)
{
ROS_INFO("Giotto Ranges[%d]=%f",i,scanner.ranges[i]);
}
for (int i=0;i<scanner.intensities.size();i++)
{
ROS_INFO("Giotto Intensities[%d]=%f",i,scanner.intensities[i]);
}
}
void belliniScanCallback(const sensor_msgs::LaserScan& scanner)
{
for (int i=0;i<scanner.ranges.size();i++)
{
ROS_INFO("Bellini Ranges[%d]=%f",i,scanner.ranges[i]);
```

```
}
for (int i=0;i<scanner.intensities.size();i++)
{
ROS_INFO("Bellini Intensities[%d]=%f",i,scanner.intensities[i]);
}
}
void giottoPointCloudCallback(const sensor_msgs::PointCloud2& points)
{
for (int p=0;p<points.data.size();p++)
{
ROS_INFO("Giotto PointCloud2 Data[%d]=%d",p,points.data[p]);
}
}
void belliniPointCloudCallback(const sensor_msgs::PointCloud2& points)
{
for (int p=0;p<points.data.size();p++)
{
ROS_INFO("Bellini PointCloud2 Data[%d]=%d",p,points.data[p]);
}
}
void giottoRawImageCallback(const sensor_msgs::Image image)
{
for (int i=0;i<image.data.size();i++)
{
ROS_INFO("Giotto Raw Image Data[%d]=%d",i,image.data[i]);
}
}
void belliniRawImageCallback(const sensor_msgs::Image image)
{
for (int i=0;i<image.data.size();i++)
{
ROS_INFO("Bellini Raw Image Data[%d]=%d",i,image.data[i]);
}
}
void giottoImageColorCallback(const sensor_msgs::Image& image)
{
for (int i=0;i<image.data.size();i++)
{
ROS_INFO("Giotto Image Color Data[%d]=%d",i,image.data[i]);
}
}
void belliniImageColorCallback(const sensor_msgs::Image& image)
{
```

```
for (int i=0;i<image.data.size();i++)
{
ROS_INFO("Bellini Image Color Data[%d]=%d",i,image.data[i]);
}
}
//this one seems to be really similar to odom, but for now I'm keeping it here anyway
void giottoImuDataCallback(const sensor_msgs::Imu& locator)
{
ROS_INFO("Giotto Position: %f,%f", locator.orientation.x,locator.orientation.y);
ROS_INFO("Giotto Velocity: linear:%f,angular:%f", locator.angular_velocity.x,locator.angular_velocity.z);
ROS_INFO("Giotto Acceleration: linear:%f", locator.linear_acceleration.x);
}
void belliniImuDataCallback(const sensor_msgs::Imu& locator)
{
ROS_INFO("Bellini Position: %f,%f", locator.orientation.x,locator.orientation.y);
ROS_INFO("Bellini Velocity: linear:%f,angular:%f", locator.angular_velocity.x,locator.angular_velocity.z);
ROS_INFO("Bellini Acceleration: linear:%f", locator.linear_acceleration.x);
}
int main(int argc, char **argv)
{
ros::init(argc, argv, "subscriber");


//object to interact with ROS
ros::NodeHandle n;


//declare pi for easy turns in radians
float pi = 3.14159;


//subscribe to the topics rviz subscribes to (right now just on the unupdated turtlebots, because of the
camera problems with the updated ones)


//odom
//subscribe to /giottobot/odom
ros::Subscriber giottoOdomSub = n.subscribe("/giottobot/odom", 1000, giottoOdomCallback);
//subscribe to /bellinibot/odom
ros::Subscriber belliniOdomSub = n.subscribe("/bellinibot/odom", 1000, belliniOdomCallback);


//scan
```

```
//subscribe to /giottobot/scan
ros::Subscriber giottoScanSub = n.subscribe("/giottobot/scan", 1000, giottoScanCallback);
//subscribe to /bellinibot/scan
ros::Subscriber belliniScanSub = n.subscribe("/bellinibot/scan", 1000, belliniScanCallback);


//point cloud
//subscribe to /giottobot/camera/depth_registered/points
ros::Subscriber giottoDepthSub1 = n.subscribe("/giottobot/camera/depth_registered/points", 1000,
giottoPointCloudCallback);
//subscribe to /bellinibot/camera/depth_registered/points
ros::Subscriber belliniDepthSub1 = n.subscribe("/bellinibot/camera/depth_registered/points", 1000,
belliniPointCloudCallback);


//raw image
//subscribe to /giottobot/camera/depth_registered/image_rect_raw
ros::Subscriber giottoDepthSub2 = n.subscribe("/giottobot/camera_depth_registered/image_rect_raw",
1000, giottoRawImageCallback);
//subscribe to /bellinibot/camera/depth_registered/image_rect_raw
ros::Subscriber belliniDepthSub2 = n.subscribe("/bellinibot/camera_depth_registered/image_rect_raw",
1000, belliniRawImageCallback);


//image color
//subscribe to /giottobot/camera/rgb/image_color
ros::Subscriber giottoImageSub1 = n.subscribe("/giottobot/camera/rgb/image_color", 1000,
giottoImageColorCallback);
//subscribe to /bellinibot/camera/rgb/image_color
ros::Subscriber belliniimageSub1 = n.subscribe("/bellinibot/camera/rgb/image_color", 1000,
belliniImageColorCallback);


//imu data
//subscribe to /giottobot/mobile_base/sensors/imu_data
ros::Subscriber giottoImageSub2 = n.subscribe("/giottobot/mobile_base/sensors/imu_data", 1000,
giottoImuDataCallback);
//subscribe to /bellinibot/mobile_base/sensors/imu_data
ros::Subscriber belliniImageSub2 = n.subscribe("/bellinibot/mobile_base/sensors/imu_data", 1000,
belliniImuDataCallback);


ros::spin(); //constantly execute subscribing stuff
```

```
//somehow combine/average all the camera data (not done yet)
//...


//somehow remap the resulting data to the topics rviz subscribes to (not done yet)
//...


return 0;
}
```

Conclusion/Future Direction

This quarter, we accomplished most of the basic setup requirements to write the actual bulk of the program. The Turtlebots were configured in conjunction with the master in such a way that they can communicate via ROS (which seems basic but did not initially work), and nodes were written to programmatically move the robots and to access their visual data. Also, some progress was made using rviz to visualize the Kinect data coming from individual Turtlebots, although, as previously mentioned, there are still many unsolved problems on this front.

The final program needs to launch the appropriate startup files on the individual Turtlebots, to subscribe to all their Kinect data, and then to control the robots while publishing a map to Rviz. In order to accomplish this, each previously mentioned problem needs to be solved, and the actual algorithmic part of the program needs to be written.