Robotics - Project

Brandon Reiss

Tuesday, May 8th, 2012

Project Description

1 Overview

This document refers to experience working with the TurtleBot robotics package from Willow Garage. The package consists of an iRobot Create drivetrain on top of which is mounted a series of platforms up to approximately 0.5m tall. The package contains an ASUS 1215N netbook as an onboard computer. The configuration used for this project supports an XBox Kinect sensor as well as a robotic arm.

An extensive suite of software tools based on the Robot Operating System, or ROS, are provided and maintained by Willow Garage. ROS is a publish/subscribe architecture designed to flow data on-demand to system components expressed as nodes. A topic is a single stream of data. Topics are addressed using paths beginning with the Unix path separator character "/". For example, the Kinect RGB image data are published by the `openni_camera` node on `/camera/rgb/image_color`.

All code relevant to this project is stored on the nyu-robotics github available at `https://github.com/ylecun/nyu-robotics`. Special thanks to Prof. Lecun for sharing literally hundreds of thousands of lines of code from Lush and other projects as resources for this project.

2 Scope

The main focus of this project is to perform visual odometry as a basis for building a coherent map of the environment surrounding the robot. The work completed for this project falls into three major categories:

1. RosLush – a software library interfacing ROS with Lush
2. TurtleBot driver – a simple driving program
3. Visual odometry – odometry correction tools and procedures

I RosLush

A critical component of this project is the infrastructure required to control the robot systems. Due to the abundance of packages existing in ROS, it was chosen as the underlying method of interacting with the robot hardware. However, since a great deal of resources for machine learning, computer vision, and linear algebra were available in Lush, it was also very beneficial to have a library to facilitate interaction between the two programming environments.

1 Design

RosLush is a C^{++} library that supports a variety of essential topics and message types used to control the TurtleBot. Its generic C^{++} template-based design allows clients to define new interfaces to ROS topics. By utilizing the `ros::spinonce()` API, the library allows Lush to retain control flow by polling ROS topics for

the newest data available. The `turtlebot` Lush package is built on RosLush. A typical main loop using `turtlebot` performs the following steps:

1. Start subscriptions to all desired topics

```
(libload "local/turtlebot")
(de main-loop ()
  (let ((tb (new turtlebot)))
    ;; Subscribe to topics.
    (==> tb start-sub-camera-rgb-image-color)
    (==> tb start-sub-camera-depth-points)
```

2. Enter a main loop

```
    ;; Loop forever.
    (while (t)
```

3. Wait for new data

```
      ;; Get new data from topics.
      (while (< (==> tb update-camera-rgb-image-color) 0))
      (while (< (==> tb update-camera-depth-points) 0))
```

4. Process data

```
      (make-robot-do-amazing-stuff
        :tb:camera-rgb-image-color :tb:camera-depth-points))))
```

5. Repeat $3 - 4$

Each `turtlebot` instance in Lush is a unique C^{++} object that holds state for the current message id grabbed from each topic as well as objects that facilitate direct, no-copy access to the data received by ROS. The library benefits from the fact that many parameters are configurable using ROS launch scripts, and the `openni_camera` node performs automatic rectification of the XYZRGB data.

For more documentation, see `lush/local/ros/roslush.h` from the nyu-robotics github. This file defines the C^{++} templates required in order to create a ROS subscriber or publisher. Example usages abound in `lush/local/turtlebot/turlebot.h`, where each RosLush topic is a brief traits class that provides to RosLush the topic name, data types, and routines for sharing data between ROS and Lush. A topic requires less than a single screen of code to implement.

2 Improvements

RosLush does not provide any interfaces to control ROS parameters. This is okay for most application since parameters may be entered into launch scripts, but could be a useful feature. In terms of code, there are two trivial Lush wrapper function definitions required per per topic. This is not a heavy burden, but there may be designs that could eliminate such duplication. However, since the data returned to Lush is expressed as a slot in the `turtlebot` object, it is not sensible to eliminate these wrappers without a corresponding change in how data are accessed since the slot must be accessed by its full name anyway.

II TurtleBot driver

The first live demonstration component for this project is based on RosLush. It is contained in the file `demos/blr/turtlebot-driver-0.lsh` on the nyu-robotics github. The purpose of the *driver-0* program is to explore a room autonomously while avoiding obstacles. Since this application makes no use of planning, it tries to use naive approach to increase its coverage of the explored space.

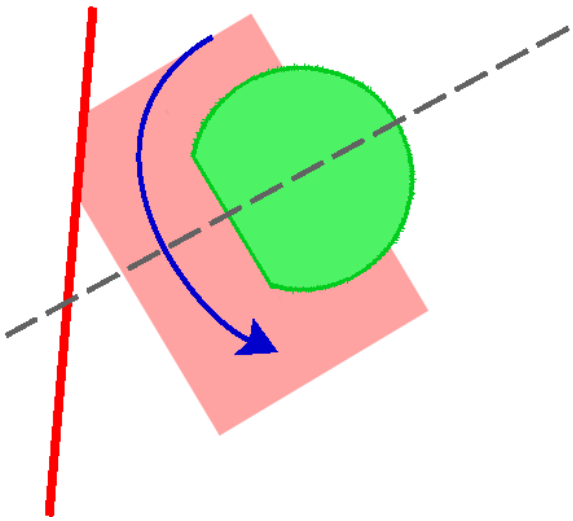

Figure 1: Description of the *driver-0* program function. The robot is drawn in green, the "near field" in light red, and the current straight-line path as a dashed gray line. The "near field" intersects the wall obstacle. Due to the bias in how the obstacle and the "near field" intersect, a torque is generated that instructs the robot to scan for the next clear path in the counter-clockwise direction.

1 Algorithm

In order to assess the surrounding environment, *driver-0* uses a 2d grid discretization of the environment. This rectangular map stores instantaneous data regarding the traversability of the terrain in front of the robot. The approach uses the following steps depicted in Figure 1:

1. Obtain an up-to-date point cloud.
2. Process the point cloud into a cost map and assign the average point height plus the variance in height of all points that fall in a particular 2d grid cell as the cost for that cell.
3. Observe the "near field" of the map for cells with a high cost where the "near field" is simply a region of interest near the robot. Compute for each scanline of the "near field" a torque based on the lever from the robot center to the cell. Threshold all cells below a certain threshold to have zero torque.
4. Check for a sudden drop in point cloud density. This may indicate an obstruction within 1m of the robot.
5. Update the driver state based on the near-field state and point cloud density. If there is no torque and high density, continue straight. When there is an obstacle to the left of the robot, a torque indicating that the robot should scan right will result. The opposite scan direction is emitted for an obstacle to the right. A low point cloud density will force the robot to enter a scanning mode.
6. Send a velocity message to `/cmd_vel` based on the current robot state. For straight driving, use a constant linear velocity only. For a scan state, use a constant angular velocity only.
7. Goto step 1.

2 Observations

The behavior of this simple program models loosely a particle under an elastic collision with a wall. The robot will move in straight-line paths throughout the environment, scanning for the next unobstructed direction in the most efficient direction upon detection of a near collision.

When the robot encounters a wall, it is very sensitive to the size of the "near field" in terms of when the scanning state terminates and straight-line motion resumes. It would be best to encourage paths that drive parallel to environment boundaries in order to explore the limits of the space. For this reason, an enhancement to *driver-0* is to make the "near field" ROI a trapezoid shape so that the robot will tend to exit a scan next to a wall sooner and travel in a direction that is more parallel.

A very essential feedback mechanism that is not implemented in *driver-0* is the iRobot Create bump sensor. For such a naive driving program, the bump sensor can initiate a scan state and avoid behavior that causes the robot to get stuck driving into an obstacle that it cannot see. Thin chair legs, thick wires, or furniture with legs set on relatively thin bases can all become obstructions that will be subtle in the cost map. In the absence of planning, these features disappear due to the range limitations of the Kinect. Therefore, the event of hitting them is often the only way to discern their presence.

The most severe limitation of this approach is that the Kinect provides no data for depths less than 1m, and so the forgetful nature of using only instantaneous maps causes many common obstacles to disappear. Since the intention of *driver-0* was to explore as much of the space as possible, it often came too close to obstacles and then fell victim to the near-range blindness of the Kinect. A more conservative threshold when defining a "near field" would improve this behavior in the future with a corresponding loss in "adventurous" behavior since many smaller spaces will not be traversable using a larger "near field".

III Visual odometry

An instantaneous map alone cannot provide enough data to perform planning functions with the robot since anything not immediately within the field-of-view of a sensor is completely ignored. There are many algorithms that can operate on a 2d grid to produce paths between two points in a map. Examples include

1. Brushfire
2. Medial axis using distance operator
3. Astar and variants

All code relevant to visual odometry and mapping is contained in the files `demos/blr/feature-map.lsh` and `demos/blr/prototype.lsh`. The function `xyz2global-feature-map-test` in `prototype.lsh` implements the main loop used to test visual odometry.

1 Mapping

A robust and coherent map of the environment is required to perform planning. Each instantaneous viewpoint must be integrated into the global map using some kind of matching. While the robot provides its own odometry, the inexpensive IMU sensors and unreliable wheel speed measurements lead to odometry measurements that have a substantial error.

Figure 2 shows that a typical 360° sweep of a room with a smooth floor surface will result in a rotational error of nearly 90° despite the sensor fusion capabilities of `/robot_pose_ekf/odom`. This is not sufficient to do planning since even a very small space cannot be mapped accurately.

2 Feature correspondence

Given two successive viewpoints V_i, V_{i+1}, and a mapping where for a given interest point m_i in V_i, the mapping gives the corresponding location of the interest point in V_{i+1}, one can compute parameters that describe a rigid transformation taking V_i to V_{i+1}.

Therefore, the task in visual odometry is to find such correspondences using features that can identify uniquely regions in one viewpoint and the same region in the next viewpoint.

Many techniques exist for this problem including

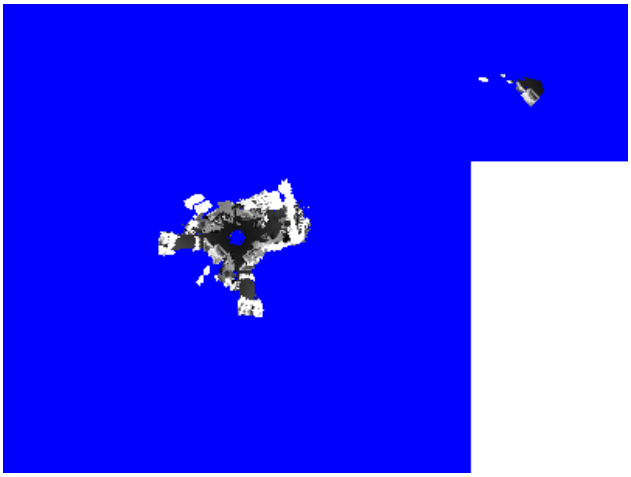

Figure 2: A full 360° scan of the robotics lab on the 12th floor of NYU 719 Broadway. Error on the range of 90° is common for a scan that uses only data from `/robot_pose_ekf/odom` in order to register successive viewpoints into the global map.

- SIFT/SURF, image space feature correspondence
- SLAM, probabilistic point cloud matching
- Kinect Fusion, voxel-based hardware accelerated mapping

Since the goal was to use the correspondence for planning, the more sophisticated methods listed above were not used for this project since they can have increasing data storage costs and implementation hurdles. Instead, features extracted from the XYZRGB point cloud directly were used.

Features were collected at each cell of the 2d grid describing the map of the environment, where each cell integrated data from all points whose x and y coordinates fell within the axis-aligned bounding box of the cell grid. In other words, cells took features from all points whose projection to the ground plane fell within that cell's 2d coordinate bounds. Both depth and RGB features were extracted to build up the local and global feature maps.

The first feature based on depth was simply a histogram whose N bins each describe a range of z values. By counting the number of points at a given cell that fall in each of the height ranges, a histogram is constructed that approximates the distribution of heights at a given cell. A vertical wall would correspond to a fairly uniformly distributed histogram whereas a table would have samples only in the bin near the table surface.

In conjunction with the height histogram, the median RGB color for all points falling within a particular height histogram bin was computed in order to differentiate cell appearance. The justification for using the median is that it is better suited to expressing a unique color without blending in samples that are likely to be shared in common between cells such as a wall color or other background data. Using a mean would tend to make cells look similar, even when they have a dominant unique color.

Euclidean distance was used to score features as being unique and also as being a match between the local and global maps. A threshold on the ratio of the euclidean distance of the match to the length of the originating feature was used rather than an absolute threshold since features in the space extracted have no unit length.

3 Experiments

Using 15 bins and a grid cell size of 10cm provided optimal results. The fact that the test environment was cluttered was ideal, and this approach would fail very badly in a uniform environment. On average, scans

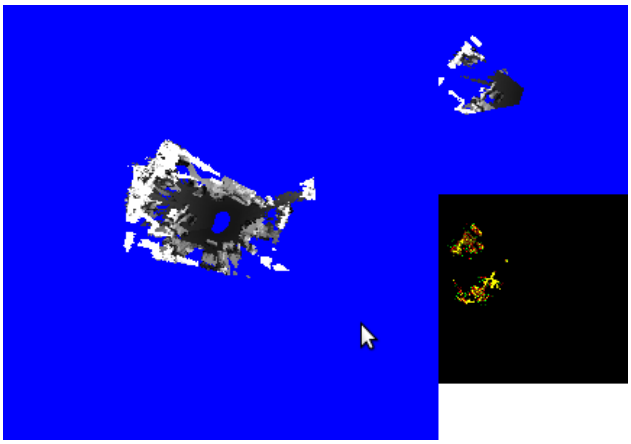

Figure 3: A full 360° scan of the robotics lab on the 12th floor of NYU 719 Broadway. Error on the range of 30° is common for a scan that using both odometry data from `/robot_pose_ekf/odom` as well as inter-frame visual odometry from feature correspondences.

using visual odometry had error on the range of 30° compared to 90° for those using odometry only. In general, error accumulates in a very non-linear fashion. While there is error inherent to the discretization of the space itself, the major cause of error is when the viewpoint passes over regions for which the features lose their discriminative power. Either erroneous correspondences will result, leading to a rapid jump in the map, or the map will appear artificially similar and "smear" in place rather than update properly in accordance with the robot motion.

Figure 3 shows a typical result for mapping with visual odometry using the method outlined in this document. Figure 4 shows that perseverance can be rewarded with a coherent scan. However, for a proper visual odometry system we would expect results like Figure 4 to be repeatable rather than exceptional.

4 Observations

The biggest issue with the system implemented and the biggest question for such systems in general is whether or not the features used possess sufficient discriminative power to provide robust estimates of camera pose. In this case, the pursuit of computational efficiency and compact representation were favored, but at the expense of a feature that was robust enough to capture a variety of environments.

Most of the variation in results was obtained by changing the grid size and altering the permissiveness of the feature matching step. The grid size acts similarly to an image pyramid by changing the scale of the features. This could be implemented explicitly in order to try to improve the results. The best results were obtained by computing two iterations on the same scale map. The first iteration would provide a larger search window to match the global map. The output from the first iteration was then used to reprocess the local map from the source point cloud before computing a second iteration using a limited search window. A pyramid would replace the iterations on the same scale map and allow features to emerge at different scales.

Another possible enhancement is to subtract the mode color of the current local map in order to bias the search for colors that are unique. For indoor environments, this would attempt to suppress the wall color, perhaps allowing a signal for some other color in the cell to emerge instead.

Ultimately, it is very possible that the features selected in this method are not unique enough to provide stable correspondences. In that case, using a proven feature extraction algorithm like SIFT or SURF in the camera RGB image would suffice, followed by storing the feature in its corresponding grid cell in the map. Using such features would also be better suited to a different data structure than the map since such features would be very sparse in comparison to the dense map constructed for this project.

Still, a bare hallway that lacks edges would also fail when using SIFT. In this case, plane fitting and

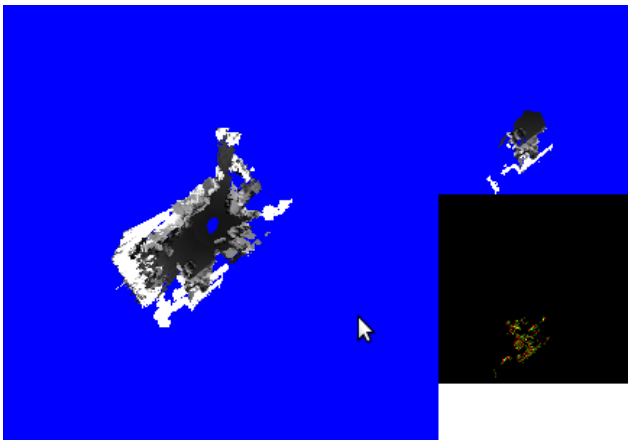

Figure 4: A full 360° scan of the robotics lab on the 12th floor of NYU 719 Broadway. An exceptional result was obtained using using both odometry data from `/robot_pose_ekf/odom` as well as inter-frame visual odometry from feature correspondences.

odometry may be the only viable options.