CompRobo Spring 2017

Sublinear monocular localization

Eric Miller and Dhash Shrivathsa

Introduction & Background

Monocular localization in three dimensional space with a map is the task we intend to solve, presenting a computationally efficient approach to handle frame-to-frame keypoint transitions and localization within a known map volume. The map is generated via rtabmap (https://github.com/introlab/rtabmap_ros), an application for ROS and a companion application for the Google Project Tango, and the localization is built for any device with a camera.

Basic Architecture

The project adopts a client-server model with a tightly optimized inner loop of keypoint localization given some previous location to attain high-framerate high-accuracy localization.

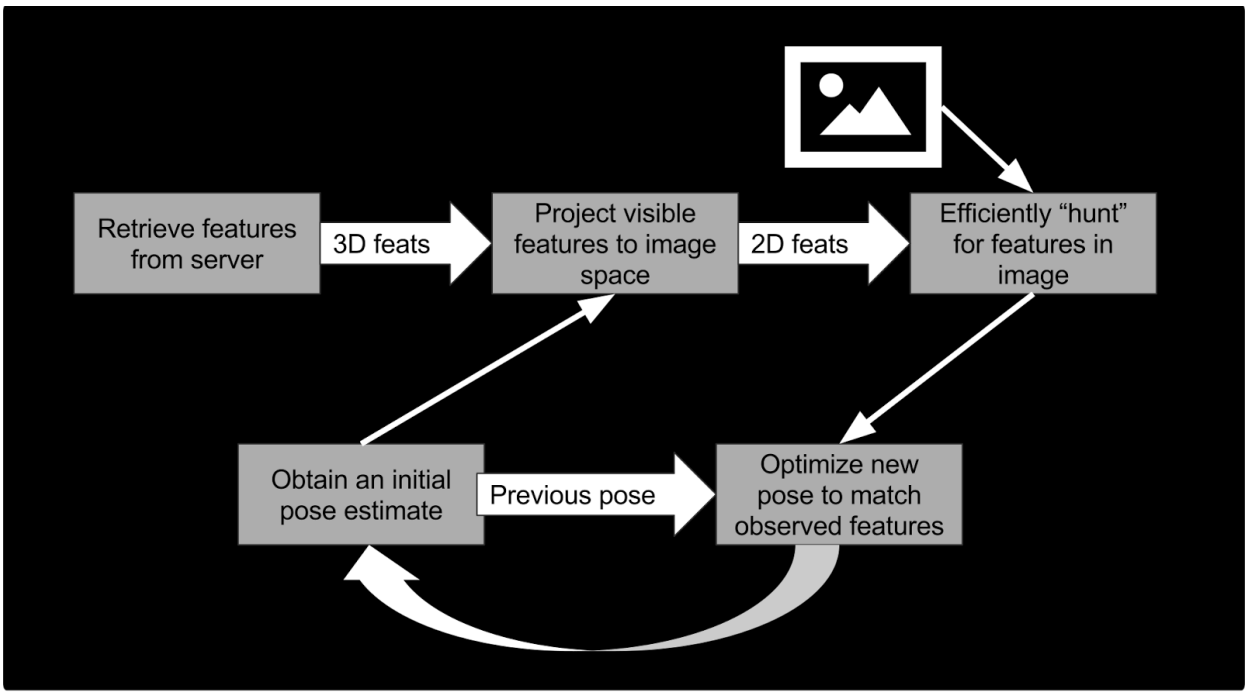
The server is responsible for holding the full map and answering queries regarding location within and visible keypoints, and the client is responsible for handling frame-to-frame updates of key points to extract pose.

This separation of concerns is due to network latency being prohibitively expensive at high-framerates, i.e. it is impossible to query a non-local map instance at 240FPS.

Client software

Because the server is responsible for making the map, the client is able to do it's part of the work much more efficiently than conventional approaches allow. In particular, because the client begins each iteration with a very good initial pose estimate coming from the high

frame rate and (potentially in the future) IMU data, the problem of localizing is reduced to the problem of slightly updating a pose estimate that is already quite accurate.



To take advantage of this, the client is structured as two pipelines, a feature pipeline that processes and refines features by using image data from the camera feed, and a pose pipeline that estimates the location of the camera based on those detected features. These two pipelines communicate with each other, allowing the client to maintain its own internal state.

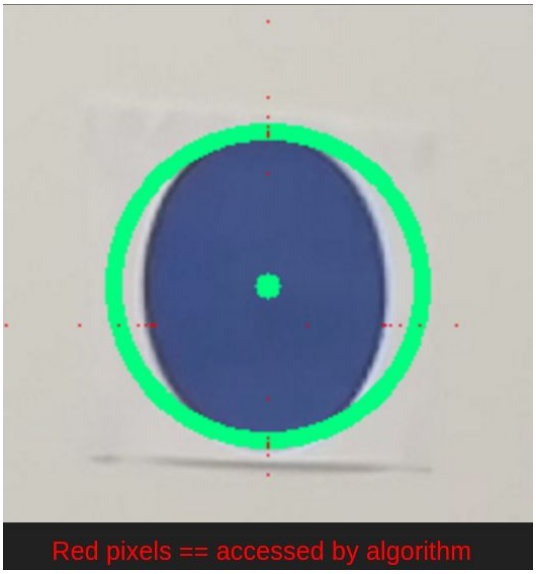
Feature search algorithms

```
def do_binary_search(image, color, start, end,
                     numSteps=float('inf'), verbose=False):
```

```
def do_linear_search(image, color, start, end,
                     num_steps=None, tol=0, verbose=False):
```

Because the camera is assumed to be reasonably well localized from the previous frame, the projected pixel coordinates of each feature in the image is actually a reasonable descriptor, particularly for large-scale features like circles and edges. This means that the

feature search process can start from the predicted location of that feature in the image, and search efficiently from there.



To do that, we implemented two basic building blocks: `do_linear_search` and `do_binary_search`. The two methods have very similar signatures, and perform the same basic task of scanning along a linear image vector to find a specific color, but they prioritize different properties. Binary searching is extremely efficient, but can be fragile if the image has very high noise levels. In practice, modern cameras seem to produce images with little enough noise that this isn't an issue. Linear search accepts an additional argument allowing it to tolerate random pixel noise or image errors up to *tol* pixels in size. This ability comes with increased algorithmic complexity, operating in O(sqrt(n)) with the size of the image, rather than the O(log(n)) achievable by binary search.

Server software

The server in our model provides a single endpoint, aptly named HelpMe. This takes a single frame and orientation, finding keypoints across the whole frame using algorithms that are O(n^2) in complexity (SIFT/HoG methods require a kernel convolution) to query the whole map at once to localize within the map frame. This route outputs the most likely place within the map that the current frame was sent from, as well as the features that will be visible from that point in real coordinates (R^3 + associated information). This bootstraps the client with a seed to start iterative refinement with.

Server-side software exists to do this task since we cannot hold the map in RAM on a mobile device, and since we can run the SIFT keypoint generation and matching offline with no latency bounds. This lets us have huge maps without having to solve the best way to query this data.

Results

Ultimately, we didn't accomplish the goal of an integrated end-to-end map generation and localization system. We did generate several pieces of functioning code, including almost all of the client-side software and robust bundle relaxation and map building from the Tango data. The video below demonstrates the client software working on a manually-inputted map in a constructed environment. Additionally, our git repo includes several Android apps, including a partial port of the client software into Android and a tool for capturing high-speed video with synchronized IMU data.

https://youtu.be/dy0ZJJRrMes

https://github.com/HALtheWise/comprobo-fast-localizer

Next Steps

Integrate client and server together

Although all of the major pieces of our software exist, we have yet to demonstrate a functioning, integrated system. Based on preliminary results, this doesn't seem like it will be too hard, but integration is always tricky...

Support edge and line features

The code is structured such that it is very easy to add additional feature types, with the choice of circles as a starting feature largely coming down to development convenience. Implementing other features, particularly edge and/or line features, would allow operation of the system in generic indoor environments.

Make use of IMU data

Integrating IMU data into the pose pipeline of the client would allow the system to continue tracking through substantially more dramatic motions.

Reflections

Dhash

Primary learning for this project was in teaming aspects for me, with a failure to integrate being the primary stopping point that caused this project to fail. I hope to learn and improve from this.

I now understand the applications of RANSAC and Gauss-Newton iterative pose refinement in a way I did not before, and better understand the movement of keypoints and how some assumptions we can make about their locality can lead to optimizations.

On a wholly other note, I now, more than before, understand how to write client-server code when the client is an Android phone. Building real Android applications is something I've done halfway before, and this was the furthest foray. I hit multiple API's and classes and learned a bunch about the platform.

Eric

This project was also a very interesting teaming and development experience for me, especially because it never fully worked. There are a number of lessons I'm going to take out of this project, both from things that worked well and things that worked badly.

- **Be very explicit about what work has been accomplished.** Particularly after I left campus for the last weekend for a scheduled trip, it became difficult for Dhash to know what had been completed, what behavior to expect from the code that did exist, and what still needed to be done. Simply having a Trello board isn't enough to resolve this, face-to-face time is critical.
- **Finish an MVP early.** It doesn't need to be fancy, just integrated. Having the luxury of iterating on some integrated product is inestimably valuable. While we generated lots of code, failing to integrate early in the process has prevented us from having a good, sharable demo I'm proud of.
- **Jupyter notebooks make for a great development environment.** Having never taken SoftDes, this project was the most intensely I have ever used Jupyter notebooks to date, and they made for a really nice development environment. Even when developing performance-limited code, getting the bugs ironed out in Python before moving to a faster language is a good workflow.

Ultimately, while I'm not happy with the way this project went at a technical level, the amount of learning about teaming and software development more than makes up for it, and the value of that learning is not to be underestimated.