[title]

Hi, I’m Katrina and I’m going to tell you a bit about my project.

[slide 1] ~1m

My project investigates mapping for UAVs. UAVs can be used to map areas which ground-based robots cannot easily traverse, such as ones with rough ground or steep surfaces.

However, in order to create a map, a robot needs to know where it is so that it can add its sensor measurements into a global frame. In order to do this, the robot needs to be able to estimate how it is moving. Most robots have a gyroscope which allows them to estimate how their orientation is changing. For ground-based robots with wheels, a position estimate can generally be obtained using rotary encoders. For UAVs GPS is generally used. However this is not always a possibility as GPS can be blocked by things like trees and tall buildings.

Another common method that doesn’t have this issue, and the one that my work focuses on, is visual odometry. That is, attaching a camera to a UAV and using the images it captures to estimate the change in the position and orientation of the camera between frames.

[slide 2] ~1:43

By images I don’t just mean regular coloured ones. Instead I am working with RGB-D data, which is a colour image and an associated depth map.

These images can be used together to form a pointcloud, or the images can be worked with directly. As you can probably see, these depth images, and thus the resulting point cloud, are fairly noisy. This will be the case for most RGB-D sensors, especially ones that are small enough to be mounted on a quadcopter.

We can use this data to estimate the change in pose of the camera by finding the transform that aligns the data captured in each frame. This is known as data registration. There are a number of different methods which can be used for data registration, which I will discuss later.

[slide 3] ~ 2:05

Before we can get to registering the data, first we need to have some data to register. So I began my project by building a quadcopter. It was important that I built the quadcopter myself so that I would know how to fix it when it inevitably crashed.

The camera I used is a RealSense D415, which you can see pictured here.

Another thing I want to draw your attention to is the Vicon markers.

[slide 4] ~2:19

The Vicon system is made up of IR sensors that detect and track these markers. Which means that by attaching markers to my quadcopter I can use the Vicon system to get it's pose as it flies. This is what I use to get my ground truth data.

[Slide 5] ~

As well as physically assembling the quadcopter, I also needed to set up a bunch of software and connections. Here’s an overview of my system for actually flying the quadcopter and acquiring data.

The computational units are shown in blue. The first is the TX2, this is the computer mounted on the quadcopter. Second is the Pixhawk, which is the flight controller for the quadcopter. Third is the vicon computer. Fourth is the workstation, this is used for interfacing with both the vicon computer and TX2.

I ended up flying the quadcopter using automated trajectories, using data from the Vicon to allow the quadcopter to keep track of where it is. This data is transmitted from the Vicon computer to the workstation and then to the TX2, all using the robot operating system, or ROS for short.

The trajectories are programmed on the TX2 using ROS. The commands from this program are then sent to the Pixhawk which controls the speed of the rotors.

Even though the trajectories were automated, I still needed the transmitter in order to arm the quadcopter, land it, and stop it if necessary.

The RealSense camera was used to capture the RGB-D data. This was controlled via the TX2 using the official ROS package, with the data being saved as a rosbag.

In terms of what data to actually capture, there are two important things to consider.

[slide 6] ~3:44

The first thing to consider is the trajectory. Ideally the trajectory should be fairly smooth and be slow enough that the images captured have a high degree of overlap and are not blurred. The trajectories I ended up investigating were a circle, a rectangle and a lawnmower trajectory.

The circle is a relatively simple trajectory that has smooth motion, and a high level of overlap. However for each frame both the orientation and translation of the camera change.

With the rectangle and lawnmower trajectory, the rotation and translation are performed sequentially. This allows us to see how well the algorithms perform for straight line motion and 90 degree rotation.

Note that for the square the camera is looking inwards, perpendicular to the direction of motion. Whereas for the lawnmower the camera is looking forwards, towards the direction of motion. This means that these trajectories will have different levels of overlap.

The circle and lawnmower are trajectories that are often used in practice, whereas the rectangle is not, although something similar could be used if the trajectory is using way-points.

[slide 7] ~4:49

The second thing to consider is the scene. There needs to be enough texture so that the RGB images have enough distinct features. There also need to be objects so that the depth data is useful.

I used two different scenes, the one on the left was only used for the circle trajectory, and the one on the right was used with all three trajectories.

[slide 8] ~5:32

Before I can get into the registration algorithms I used, we need to talk a bit about frames. Here’s a photo of the scene with the quadcopter frame marked in red and the world frame marked in white.

Note that the world frames remains fixed relative to the ground, whereas the quadcopter frames remains fixed to the quadcopter, which is moving relative to the world frame. Thus there are actually multiple quadcopter frames, one for each timestep.

[slide 9] ~6:00

In addition to the world and quadcopter frames, I also have a camera and Vicon frame. The camera is mounted to the quadcopter, so like the quadcopter frame there is also a camera frame for each timestamp.

The Vicon frame is the frame that my ground truth data is reported in. It is fixed relative to the world frame.

[slide 10] ~6:33

So I have one more frame, this is the image frame, my only 2D frame. As you can probably guess it also moves around so there is a different one for each timestamp.

What I’m showing here is the pinhole camera model, which I what I used the transform between the image and camera frames. Note that the RealSense is calibrated so I know the intrinsic parameters such as the focal lengths and principle points.

[slide 11] ~7:00

Here’s an example of some 3D points in a scene and how they are viewed in the camera and image frame for two different camera poses.

I drew a box around the image frame to make it more obvious which is which.

[slide 12] ~7:13

OK, now we can talk about registration algorithms. I ended up investigating three methods –the Essential Matrix method, the Kabsch algorithm and a Perspective-n-Point algorithm, or PnP for short.

These are all methods that take two sets of points where the correspondences between them are known. The easiest way to work out point correspondences is by using a feature detector and matcher on the RGB image, so that’s what I did. Note that this ends up giving quite a few false matches, so the inputs to these algorithms has many outliers. Thus all three methods used the Random Sample Consensus algorithm (RANSAC for short) in try to filter out the outliers.

The Essential Matrix method uses only RGB data. I used Nister’s five-point algorithm for calibrated cameras to both find the Essential Matrix and then recover the camera’s pose from it.

Note that the Essential Matrix method can only find transforms up to some scale factor, which I estimated manually.

The Kabsch algorithm uses only 3D data.

It first estimates the translation between the set of points as the translations which causes their centroids to coincide. Then it seeks to minimize the average Euclidean distance between the points. There’s a nice way of doing this using singular value decomposition which I won’t go into here.

The PnP algorithm generally takes 2D data from an image frame with corresponding 3D points in the other frame.

PnP seeks to minimize reprojection error. There are a few different algorithms for doing this, some iterative and some closed-form. I tried an iterative method and a closed form method but didn't find any significant differences between them.

[slide 13] ~9:09

Here are some results. This slide is just for the circle trajectory for the scene with only two boxes.

One of the things I found was that the methods are very sensitive to the amount of movement between frames. I varied this by recording data at a high frequency and then just skipping a certain number of frames when doing the registration.

The essential matrix method and PnP performed better for less movement, and Kabsch performed better for larger movements.

As you can see, Kabsch still wasn't great even with a good processing frequency, but it produces nonsense with a bad one. The reason for it's poor performance is probably at least partially due to the noise in the depth image.

The Essential matrix method and PnP both perform relatively well.

[Looking at the trajectory with the better processing frequency, we can see that they both have different sorts of errors. The essential matrix method has a tilt in the trajectory, indicating that the rotation is wrong. PnP has a few sections where the trajectory is too far out but the shape of that section is right, indicating the translation is wrong just before that section.]

I also found that PnP took much longer than both the Essential matrix method and Kabsch. This might be due to the noisy depth map, which makes it harder for RANSAC to find enough inliers. If this is the reason the fact that it didn't happen for Kabsch could indicate that the RANSAC thresholds were not strict enough.

[slide 14] ~10:32

Here's the results for the different trajectories with the other scene. I’m only displaying the best trajectories here due to space constraints.

PnP performed better the Essential Matrix method for the rectangular and lawnmower trajectories, while the Essential Matrix was best for the circle trajectory.

For the rectangle and lawnmower trajectories, PnP seems to work pretty well for the straight-line motion, but sometimes fails at the corners.

[slide 15] ~10:59

So in summary, the Essential Matrix and PnP methods has fairly similar accuracy for the circle trajectory, but PnP was better for the other two.

However, PnP is also quite a bit slower than the Essential matrix method, and so should not be used for time-critical tasks.

Both PnP and the Essential Matrix method performed better for smaller motions between frames. For larger motions Kabsch can be superior.

Note that I investigated just one part of the mapping pipeline. In order to produce state of the art results you cannot just rely on a good registration algorithm.

Most mapping algorithms have methods to automatically choose which frames to register, allowing them to discard blurry frames and deal with varying speeds. The importance of this step can be seen from how differently the methods perform for different processing frequencies.

Mapping algorithms generally also have ways to deal with drift by “closing the loop”, that is, detecting when they return to a location or see an object again and updating the map accordingly.

Thus if I wanted to extend this into a full mapping algorithm I would start by adding one or both of these things.

~12:14