Robust Visual SLAM From Humble Beginnings

Nathan Litzinger[1] and Sidharth Suravarapu[2]

***Abstract*— Visual SLAM (or VSLAM) has proven itself as a low-cost means for simultaneous localization and mapping for mobile robots in a variety of environments and conditions ([4], [5]). However, there is a high barrier to entry in understanding how to tailor or create VSLAM systems for different applications, as they can be complex to describe and implement. In an effort to make visual SLAM more approachable for students and practitioners alike, we set out to outline the process of building up to a modern visual SLAM system starting from the problem's motivation and classical solutions. In particular, we begin with visual odometry, discuss relocalization / loop closure detection, introduce optimization-based smoothing methods to improve odometry, and finally reformulating the entire inference task as an optimization problem.**

Our implementation is available on GitHub at `https://github.com/nlitz88/vslam`

I. Introduction

Simultaneous localization and mapping (SLAM) is a technique that aims to answer the questions "where have I gone, and what did I see along the way?" I.e., the aim is to not only figure out where a robot has traveled in its in environment, but to also figure out what the environment "looks like." SLAM can also more broadly be considered as a state estimation problem, where your state is not only composed of your robot's position and orientation (pose) with respect to some frame, but also contains the positions and orientations of different observed environmental landmarks. SLAM techniques answer these questions by solving for the positions and orientations of the robot and environmental landmarks that best explain the measurements taken by the robot along the way.

The *type* (or types) of measurements that a SLAM technique uses to *infer* the most likely robot and environmental landmark poses largely defines the "type" of SLAM system. In this work, we direct our focus specifically on the scenario where a robot's trajectory and environmental features are being *estimated* or *inferred* based on *visual measurements*. These visual measurements usually consist of *interesting* points in an image, like corners or edges. SLAM with visual measurements is generally considered as visual SLAM, or **VSLAM**.

While the most robust SLAM systems [7] are *multimodal* (incorporate multiple kinds of measurements from different types of sensors), SLAM based on visual features alone has a proven track record of being effective for a variety of use cases [4] [5]. This is a particularly important result for for low-cost robots, as cameras are considerably less expensive than LiDAR sensors (the other most common sensor modality used for SLAM).

However, visual SLAM implementations (and SLAM systems in general) are often difficult to understand and derive for students and practictioners. I.e., it is very difficult to debug and tune a SLAM system when you are not familiar with the key components or design decisions/choices that went into it.

Therefore, we set out to outline the process of incrementally building up to a modern, optimization-based 3D visual SLAM system, hopefully rivaling the capabilities of [4]. This will involve:

1) Introducing visual odometry as a key first aspect in visual SLAM.
2) Building a naive "incremental" visual odometry implementation based on classical computer vision techniques.
3) Using classical CV techniques to detect previously visited locations in order to correct a drifted visual odometry trajectory.
4) Using an optimization technique to *smooth out* the incremental trajectory created by the naive visual odometry system.
5) Replacing/reformulate the incremental visual odometry and loop closure correction (motion estimation **back end**) as an optimization problem using a library like GTSAM. I.e., estimating the motion as an optimization problem over multiple steps, rather than frame-to-frame motion estimation.
6) Finally, estimating the position of observed visual landmarks as a part of the robot's state, in addition to its trajectory (trajectory == sequence of poses, with each pose associated with a particular timestep/stamp).

Our goal is to first describe the visual SLAM problem clearly and intuitively using a factor graph. The goal is to present the problem in a way that is easy to both communicate *and* implement. Further, we aim to provide a rudimentary implementation that can be clearly traced back to the factor graph the problem was modeled by. In particular, we aim to use the GTSAM library [3], as it natively solves inference problems provided in the form of factor graphs. Additionally, we aim to explore current loop closure techniques and identify which are most effective visual SLAM but do not introduce significant complexity. It is our hope that this implementation can serve as an intuitive

[1]Nathan Litzinger is with the Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA 15213, nlitzing@andrew.cmu.edu

[2]Sidharth Suravarapu is with the Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA 15213, ssuravar@andrew.cmu.edu

reference for future students and SLAM practictioners.

II. Related Works

Visual SLAM is a well-studied problem in the robotics and computer vision community alike. Works like [4] [5] have demonstrated the test-of-time in practical usability of visual-SLAM algorithms. However, it is important to first consider works like [6] that detail key techniques that inform the approaches used in Visual SLAM (like the ideas used in visual odometry, for example).

Feature extraction is a key step in the visual odometry pipeline,

State of the art approaches such as SuperPoint [1] rely on deep learning to detect and describe interest points in images, which can be more robust than traditional feature detection methods, especially in challenging lighting conditions or with repetitive textures. For example, Superpoint consists of a fully-convolutional neural network that operates on full-sized images and produces both interest point detections and their associated descriptors in a single forward pass. This differs from other methods, which separate these into multiple steps.

III. Methods

This section details the evolution of the visual SLAM system developed from infancy to completion. We begin with a short overview of the hardware and software that all the data was collected with, followed by an introduction to visual odometry, and incrementally introduce features and enhancements to the system towards creating a modern 3D visual slam implementation.

A. Data Collection

Before beginning development of any visual odometry or visual SLAM capabilities, we first set out to collect data sequences that would be sufficiently representative of common 3D environments that a robot navigating in a three-dimensional environment (like a multi-rotor aircraft or quadrupedal robot) might encounter. This means that we would have to collect data sequences that are not only based inside or outside a building, but also sequences where the robot/camera travels along a trajectory that varies in Z (I.e., through different elevations/heights), as opposed to a trajectory where the position of the camera only varies in two dimensions.

1) Hardware: Figure 1 depicts the hardware setup used to collect the data sequences described below. The system is comprised by an NVIDIA Jetson Orin Nano 8GB and an Intel RealSense D435i depth camera. Note that while visual SLAM does not strictly need an RGB-D capable camera, it can improve the result and/or reduce the number of steps needed to arrive at a motion estimate, depending on how your implementation works. In the case of the initial visual odometry approach we'll describe below, we choose to take advantage of the measured depth to make our implementation more simple.

The Intel Realsense has a pair of infrared cameras for capturing simplified stereo imagery, as well as an RGB camera. For this simple approach, we chose to only record and use the depth produced by the realsense and the left infrared camera image, as the left infrared camera and depth frames coincide.

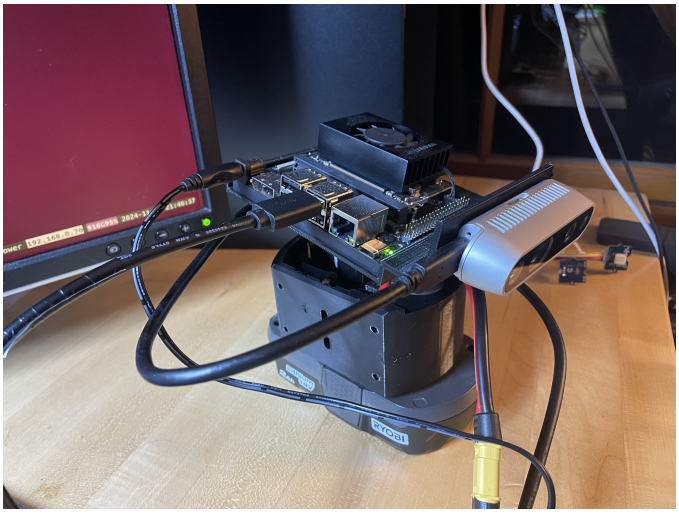

Fig. 1. Data collection rig composed of an NVIDIA Jetson Orin Nano and an Intel RealSense D435i.

2) Indoor House Sequence: Figure 2 shows a preview of the indoor sequence that was collected. The indoor sequence was created by carrying the data collection rig between and around the floors of a house. Carrying the camera between floors was a key decision in order to create a 3D trajectory for the camera to follow–and eventually, for the visual SLAM (or just visual odometry) system to accurately estimate or "infer" from the measurements it makes along the way.

Fig. 2. Preview of the indoor image sequence. The frame on the left is from the left infrared camera and the right frame is the depth provided by the RealSense the IR projector disabled.

B. Visual Odometry

Understanding visual odometry (often abreviated **VO**) is a key first step in implementing and understanding visual SLAM systems. Odometry (generally speaking) is a motion estimate. Computing odometry is how we answer the question "how far have I gone?" using measurements taken while moving. A classic example of odometry that most people are familiar with is the odometer in a car. Using some type of rotation sensor, the car can measure how many rotations its tires have made, and can approximate the distance it has travelled over its lifetime. *Visual* odometry is

no fundamentally no different–but with visual odometry, the odometry (motion estimate) is computed using images from a camera.

In general, there are two major approaches to using camera images for visual odometry:

1) Dense or "Direct" methods
2) Sparse of "Feature-based" methods

Both methods use a sequence of consecutive images to figure out how far the camera *must have moved* based on the observed changes in consecutive image frames. Dense methods use the **entire** image contents (I.e., every pixel), whereas feature based extract key points of interest (keypoints) and track those throughout frames to estimate how the camera must have moved. While the jury is still out on which method is superior / preferable, feature-based methods like [5] have shown great robustness and have been used in practice for many years now. For this reason, we focused our attention on feature-baesd visual odometry.

In order to motivate the need for more advanced/involved optimization based visual odometry methods, we first prototyped a basic visual odometry pipeline grounded in classical computer vision techniques, largely based on the diagram shown in 3 from [6]. Additionally, we created a version of this pipeline that relies on 3D depth data provided from an RGB-D camera like the Intel RealSense we are using. Visual odometry can absolutely still be implemented to work solely on a monocular camera, but will only be able to estimate the camera's motion (odometry) up to some scale factor. We mainly chose to use depth information as it can help simplify the motion estimation step (however, only when detected keypoints are within range of the stereo camera).

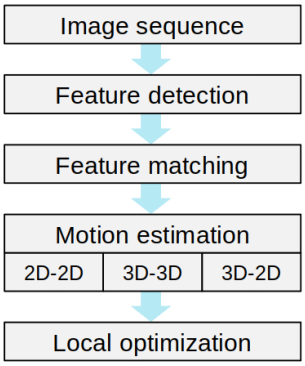

Fig. 3. Outline of the fundamental steps carried out for visual odometry from [6].

We begin the discussion of our classical visual odometry approach beginning from the second step shown in 3. We used the image sequences described in III-A.

1) Feature Extraction: The first step in the visual odometry pipeline is to identify interesting features in the current image. The feature detection method used is **ORB**, (Oriented FAST and Rotated BRIEF). The detector builds off of FAST, a corner detector. A "corner" is defined as a pixel that has a high contrast compared to its surrounding pixels. It looks at the intensity of N contiguous pixels in a circle around the point. FAST is extremely fast because it avoids computationally expensive operations like gradients or matrix calculations, like other methods such the Harris detector. Instead, it relies on simple intensity comparisons. Orientation is assigned to each keypoint using the intensity centroid method, ensuring rotation invariance.

BRIEF lets us encode the image patches we find as binary strings by sampling in a predefined pattern around each of our points. This allows us to perform future feature matching far more efficiently because we're using a binary description to store a description of the image patch rather than having to compare the image path itself.

ORB introduces scale and rotation invariance, which allow it to perform reliably in dynamic environments where the camera may experience rapid rotations or zoom effects.

2) Feature Matching: We can use the binary descriptors previously computed by BRIEF to match detected features in images.

The approach we currently use is the brute force method, which is to compute the Hamming distance between each descriptor in one image and every descriptor in the other image, selecting the match with the smallest distance.

OpenCV provides a number of more efficient algorithms optimized for fast nearest neighbor search in large datasets and for high dimensional features. An example is KNN (K-Nearest Neighbors) matching, where each descriptor in one image finds its top k nearest neighbors (typically 2) in the second image based on Hamming distance. For our future purposes, it will be vital to use more efficient feature matchers.

For the example of the stereo camera at a single timestep, what this gives us is a set of points in the 2D plane of one camera, and a set of matched points in the 2D plane of the other camera. With enough points, we can determine the relative transform between these camera views. This can then be extended to multiple camera views over time.

Computed features for two stereo cameras and their matches can be seen below.

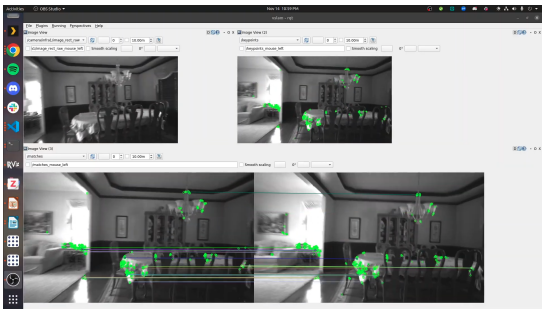

Fig. 4. Detected features and their matches from the indoor image sequence presented in III-A.

3) Motion Estimation: [6] summarizes the three major techniques used to estimate the motion of the camera between different images with corresponding keypoints matched between them:

1) 2D-2D
2) 3D-3D
3) 3D-2D

Type of correspondences	Monocular	Stereo
2D-2D	X	X
3D-3D		X
3D-2D	X	X

Fig. 5. Summary of the different motion estimation methods from [6].

In the case where only a monocular camera is available, motion estimates will have to be computed via 2D-2D correspondences–*unless* you use those 2D correspondences to first triangulate the 3D position of certain keypoints and then use the 3D-2D (PnP) approach. Note that while this is mainly considered an approach that is strictly for the monocular case, it could also become quite useful *even when* using an RGB-D camera–specifically in the case where the detected keypoints are too far away (not within the depth camera's range).

However, if you *do* have depth information available (when using a depth camera and keypoints are in range), life can be made just a little bit simpler using either the 3D-3D or 3D-2D approach. Figure 6 depicts a simple example (not from one of the image sequences) where the 3D position of ORB keypoints is trivially "triangulated" / computed using the depth value found in the depth map at the keypoint's 2D position.

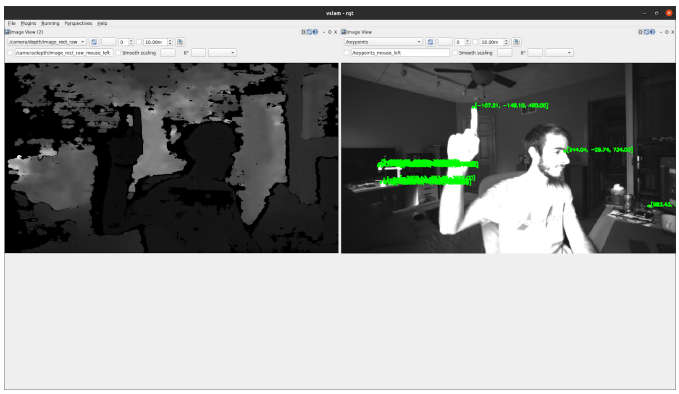

Fig. 6. Computing the 3D positions of ORB keypoints using the corresponding stereo depth map shown on the left.

The 3D-3D approach is perhaps the most intuitive. For the first frame at timestep k, you detect keypoints and measure the 3D position of each of those keypoints using the depth value provided by the stereo camera at that keypoint's position in the depth map. You then repeat these exact same steps for the frame that follows (taken at timestep $k+1$). Additionally, once you have receievd the $k+1$ image, you find a match for each of the keypoints with a keypoint from the timestep k image. Then, intuitively, if you consider a simple case where the camera is *only translated* (no rotation), then you can estimate the camera's motion between those two frames simply as the difference between the 3D positions of all the keypoints. In theory, this works great–but this is rarely done in practice. While this method can certainly work, it simply tends to be inferior to the 3D-2D method discussed next. This comes down to the fact that more noise from the keypoint and depth measurements makes it into each motion estimate between frames. This is because the motion estimate is computed using not only a noisy/imperfect 2D keypoint in each image, but also a noisy/imperfect depth measurement for each of them. Therefore, more error is introduced at each step and the overall estimated trajectory and the motion estimate rapidly drifts away from reality.

The 3D-2D approach, on the other hand, is able to estimate the camera's motion while accruing less error at each step. This method basically boils down to solving the Perspective-n-point (PnP) algorithm. To understand PnP, first imagine you have a handful of 2D keypoints and their corresponding 3D positions expressed in the timestep k camera's frame. Now, you move the camera and take a picture at timestep $k+1$. You identify a bunch of keypoints and then try to find the 2D keypoints in the timestep k image that match these the ones found in your newest $k+1$ image. If successful, this means you have a bunch of 2D points in the timestep $k+1$ image, and the 3D position of those points w.r.t. the timestep k camera frame is known. This means that, if you know rotation and translation of the timestep $k+1$ camera w.r.t. the one at timestep k, then you could (using the camera projection matrix in Figure 7) solve for the 2D location that those 3D points would be projected into.

Of course, the key problem is that **we do not yet know $R|t$ is yet!** The PnP problem solves this problem for us by *estimating what the $R|t$ must be* such that the difference between the 2D position that the projection matrix says the 3D points project to and the **measured** position of those 2D keypoints is minimized (ideally zero). The difference between where the 3D point is projected to and where the keypoint is actually measured to be at is known as the **reprojection error** in computer vision. From the SLAM perspective/literature, one could (as a mental model) think of the camera projection matrix as being the *generative sensor model* that tells us "based on this R—t, we should measure the 2D point to be here" – and then that predicted/hypothesis value is subtracted from the actual measured value. *This intuition will likely be useful later for when this problem is generalized to an optimization problem over measurements made across multiple frames/timesteps.*

On a specific implementation note, a version of PnP from OpenCV was used that also used the RANSAC algorithm [2] to filter out outlier 3D-2D correspondences.

Once either of the above methods detailed above have been used to estimate the 3D motion of the camera (rotation and translation) between image frames, this transformation can be "appended" to previous transformations to get the overall *composed* transformation from the camera frame at the first timestep to the camera frame at the current timestep. Technically speaking, the transformation computed belongs to the SE(3) group (special euclidean group), whose operator

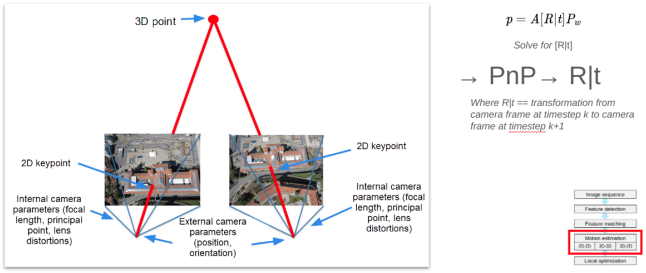

Fig. 7. Summary the 3D-2D motion estimation method based on the Perspective-n-point algorithm (PnP).

for addition / combination is matrix multiplication. All that is to say: to chain together two transformations, just multiply together their transformation matrices.

IV. Results

A. Visual Odometry Results

Using the above rudimentary visual odometry approach described above, the pipeline was applied to the indoor image sequence presented in III-A. The resulting trajectory can be seen in 8. Unfortunately, it appears that some stage of the pipeline thus far is causing a significant amount of error in the resulting trajectory. This could be caused by issues with bad 2D correspondences or bad 3D measurements being made using noisy / inaccurate depth measurements. Further tuning can be performed to make performance more robust and less likely to endure such extreme errors.

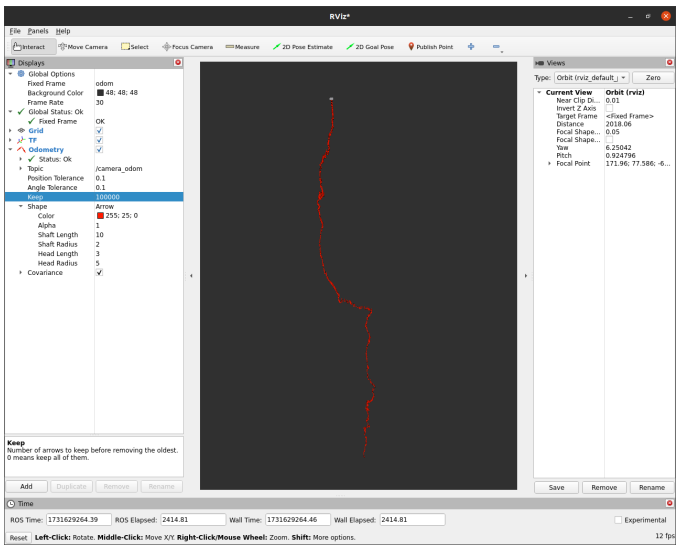

Fig. 8. The (very inaccurate) resulting trajectory from the rudimentary visual odometry pipeline.

V. Conclusion

This work introduces and motivates visual SLAM from the ground up, and (thus far) outlines the implementation of a simple visual odometry system based on classical computer vision techniques and stereo imagery.

References

[1] Daniel DeTone, Tomasz Malisiewicz, and Andrew Rabinovich. Superpoint: Self-supervised interest point detection and description. *arXiv preprint arXiv:1712.07629*, 2017.

[2] Martin A. Fischler and Robert C. Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM*, 24:381–395, 1981.

[3] Michael Kaess. Gtsam library, July 2015.

[4] Raul Mur-Artal, J. M. M. Montiel, and Juan D. Tardos. Orb-slam: A versatile and accurate monocular slam system. *IEEE Transactions on Robotics*, 31(5):1147–1163, October 2015.

[5] Raul Mur-Artal and Juan D. Tardos. Orb-slam2: An open-source slam system for monocular, stereo, and rgb-d cameras. *IEEE Transactions on Robotics*, 33(5):1255–1262, October 2017.

[6] Davide Scaramuzza and Friedrich Fraundorfer. Visual odometry [tutorial]. *IEEE Robotics Automation Magazine*, 18(4):80–92, 2011.

[7] Shibo Zhao, Hengrui Zhang, Peng Wang, Lucas Nogueira, and Sebastian Scherer. Super odometry: Imu-centric lidar-visual-inertial estimator for challenging environments, 2021.