69156 Simultaneous Localization and Mapping

Lab 3: Visual SLAM: Camera Tracking

1 Problem

Visual SLAM systems are complex pieces of software composed by many components. The goal of this assignment is that you put your hands on specific parts of a minimalist visual SLAM system to better understand its key components, without taking care of other, more complex, implementation details. For that purpose, we provide you the code of Mini-SLAM, a "simple" visual SLAM software with some missing components that you will need to fill up to get it working.

In this first assignment, your task is to implement some basic operations of camera tracking to allow Mini-SLAM to initialize a map from two monocular images, and track the camera against it.

2 Mini-SLAM

Mini-SLAM is KeyFrame-based visual SLAM system implemented in C++, that uses ORB features. In essence, it is a simplified version of ORB-SLAM, without place recognition, and without threads. It is prepared to perform two main tasks:

- Camera Tracking, that processes each new image and computes its camera pose. It also decides whenever we need to insert a new KeyFrame into the map.
- Local Mapping, that processes each new KeyFrame and performs the main mapping operations: point triangulation, observation fusion, local bundle adjustment, etc.

As most visual SLAM systems, the code has some complexity and many components, but it's quite modular, so it is easy to modify specific parts without understanding the full code. In the remaining of the section, we will show its main components, dependencies and installation guide.

2.1 Code structure

The code is structured in three main directories:

- `Apps`: includes some demos showing the use of Mini-SLAM with several datasets.
- `Data`: includes the settings files for several datasets. These files include system parameters and camera calibration.
- `Evaluation`: tools to evaluate the camera trajectory computed by Mini-SLAM.
- `Modules`: the Mini-SLAM library source files.
- `Thirdparty`: third party libraries used by the system.

Inside the `Modules` directory, you will find all the files that compose Mini-SLAM, organized in sub-directories, sorted by topic. As a summary, you can find below all of them:

It is not necessary that you understand all the code, we will remark which parts you need to look at to complete the assignments.

- `Calibration`: implementation of the camera models, its projection and unprojection functions and their Jacobians.
- `DatasetLoader`: tools to load datasets from disk.
- `Features`: implementation of feature/descriptor extractors.
- `Map`: the map of the SLAM system.
- `Mapping`: implementation of the mapping operations.
- `Matching`: implementation of feature matching using their descriptors.
- `Optimization`: implementation of Bundle Adjustment and pose-only optimizations.
- `System`: the Mini-SLAM system itself.
- `Tracking`: implementation of the camera tracking operations.
- `Utils`: some useful functions like geometric operations needed all along the code.
- `Visualization`: tools for displaying the map and the current image.

3 Tasks

In this assignment, you will implement some key operations for camera tracking. You will start with a version of Mini-SLAM that does nothing and in an incremental way you will make it to track the camera pose, better as you complete each task. We strongly recommend that you complete the tasks in order as it will be much easier for you to debug your implementation.

3.1 Task 0 - Download the dataset

For this assignment we are going to use the TUM-RGBD dataset. This dataset is composed by several sequences recorded with a hand-held RGB-D sensor in indoor configurations.

For this task, we ask you to install Mini-SLAM and check it. For that, download the sequences `fr1_xyz` and `fr1_floor` from here. Uncompress the downloaded sequences into a directory (for example, `/home/user/datasets`).

Now install Mini-SLAM, compile it and execute the program `mono_tumrgb` to check that everything is OK. For that, assuming that you are at the root folder of Mini-SLAM, run the following command:

```
./Apps/mono_tumrgbd <path_to_uncompressed_dataset>
```

In our previous example, you should call the program as follows:

```
./Apps/mono_tumrgbd /home/user/datasets/fr1_xyz
```

The program should show in a window all the images of the dataset with the extracted features.

3.2 Task 1 – Monocular Map Initialization

The first step in a monocular SLAM system is to initialize a map from two views. From previous courses, you have already learnt the basics of epipolar geometry and Structure-from-Motion algorithms. For that, we give you an algorithm that from a set of feature matches coming from 2 images, reconstructs the camera poses and the 3D point positions using epipolar geometry.

```
int searchForInitializaion(Frame& refFrame, Frame& currFrame, int th,
                           std::vector<int>& vMatches,
                           std::vector<cv::Point2f>& vPrevMatched){
}
```

Listing 1: Function skeleton for task 1

We ask you in this first step to compute those feature matches. For that, you have to implement the function `searchForInitialization` at `Modules/Matching/DescriptorMatching.h/cc` files. Inside of this function you will find the KeyPoints and descriptors coming from the reference and the current image and your task is to find the feature matches between them. These matches must be stored at the vector `vMatches` with the following format:

- `vMatches[i] = j` means that feature i in the reference frame is matched with feature j in the current frame.
- `vMatches[i] = -1` means that no match for feature i in the reference frame has been found in the current frame.

The parameter `th` indicates the minimum Hamming distance between 2 descriptors to be considered a matching candidate. You can do a brute force matches with all features but it would be more efficient if you implement some kind of guided matching. We recommend you to look at the following .h/.cc code files:

- `Modules/Tracking/Frame.h/.cc`: more specifically, the method `getFeaturesInArea` can help you a lot!
- `Modules/Matching/DescriptorMatching.h/.cc`: more specifically, the method `guidedMatching` is an example of feature matching using geometric information.

We require you to draw the matches between both images to check that they are correct. After this task, the SLAM system will try to compute an initial map but will fail.

3.3 Task 2 – Pinhole camera model

All SLAM systems need to know the calibration and the projection model of the camera used. This is crucial as many operations in a SLAM system involve projecting 3D points into a camera or unprojecting an image feature into a 3D direction vector. In this task, we ask you to implement the projection (and its jacobian) and unprojection functions of the pinhole camera model. This is implemented in `Modules/Calibration/PinHole.cc`.

```
void PinHole::project(const Eigen::Vector3f& p3D, Eigen::Vector2f& p2D){
    /*
     * Your code for task2 here!
     */
```

```
}

void PinHole::unproject(const Eigen::Vector2f& p2D, Eigen::Vector3f& p3D) {
    /*
     * Your code for task2 here!
     */
}

void PinHole::projectJac(const Eigen::Vector3f& p3D, Eigen::Matrix<float,2,3>& Jac) {
    /*
     * Your code for task2 here!
     */
}
```

Listing 2: Function skeletons for task 2

In the code above, both function receive a C++ array that stores the 3D and image points. Remember that C++ indexes for arrays start at 0.

After implementing both functions, your system should be able to initialize a map from two images but the camera tracking will still fail. We ask you to take one initialization and observe the relative camera pose computed and explain if the result makes sense with the images used for the initialization. We also want you to discuss why, for several executions, the initialization may be different even though you use the same pair of images, specially in the translation component.

3.4 Task 3 – Reprojection error

The core of visual SLAM is the Bundle Adjustment algorithm, that needs to define an error to optimize. For feature-based systems as Mini-SLAM, we are going to minimize the reprojection error defined as:

$$e_{ij} = \mathbf{u}_{ij} - \pi_i \left(\mathbf{R}_{iw} \mathbf{x}_{wj} + \mathbf{t}_{iw} \right) \tag{1}$$

for a camera i whose pose is $\mathbf{T}_{iw} = [\mathbf{R}_{iw}, \mathbf{t}_{iw}]$ and a given point $\mathbf{x}_{wj}$ that has been matched with the feature $\mathbf{u}_{ij}$. We use the `g2o` library to implement the Bundle Adjustment.

Your task is to implement the reprojection error at `Modules/Optimization/g2oTypes.h`. In this file we define the types needed by `g2o` to run the optimization algorithm.

```
class EdgeSE3ProjectXYZ: public  g2o::BaseBinaryEdge<2, Eigen::Vector2d,
VertexSBAPointXYZ, g2o::VertexSE3Expmap>{
public:
    EIGEN_MAKE_ALIGNED_OPERATOR_NEW

    EdgeSE3ProjectXYZ();

    bool read(std::istream& is);

    bool write(std::ostream& os) const;

    void computeError()  {
        const g2o::VertexSE3Expmap* v1 = static_cast
            <const g2o::VertexSE3Expmap*>(_vertices[1]);
        const VertexSBAPointXYZ* v2 = static_cast
            <const VertexSBAPointXYZ*>(_vertices[0]);
        //Observed point in the image
        Eigen::Vector2d obs(_measurement);
```

```
        //Predicted 3D world position  of the point
        Eigen::Vector3d p3Dw = v2->estimate();
        //Predicted camera pose
        g2o::SE3Quat Tcw = v1->estimate();

        //Implement here the reprojection error
        //Do it also at the EdgeSE3ProjectXYZOnlyPose class!!
    }
```

Listing 3: Function skeletons for task 3

With this task, the system should be able to perform pose-only Bundle Adjustment minimizing the reprojection error, and track the camera pose for a few images after initialization. As we are still not adding new keyframes and points to the map, the system will get lost soon, as it looses view of the initial points.

We ask you in this task to evaluate the estimated trajectory computing the RMSE Absolute Translation Error (ATE) using the provided scripts, with the following command:

```
python3 Evaluation/evaluate_ate_scale.py <ground_truth_file>
<predicted_trajectory_file> --plot plot.png
```

Listing 4: Script for trajectory evaluation for tasks 3 and 4

3.5 Task 4 – Tracking the local map

The main difference between SLAM and Visual Odometry (VO) is that SLAM computes a map that can be used to increase the accuracy of the estimations. This can be exploited in the camera tracking to get more matches from the map. This is done by taking MapPoints that should be visible in the current image but for some reason have not been already matched.

For that purpose, in this task, we ask you to implement the `track local map` functionality. More specifically, we want you to implement a method to project MapPoints from the local map and match them to features in the current frame that has not a MapPoint already matched.

We give you some code for that at the `searchWithProjection` function at the `Modules/Matching/DescriptorMatching.h/cc` files but you need to finish the implementation. In this function the `th` input argument is the Hamming distance for descriptor matching, the set `vMapPointsIds` stores the MapPoints IDs to be matched and pMap allows you to recover a MapPoint given its ID.

```
int searchWithProjection(Frame& currFrame, int th,
std::unordered_set<ID>& vMapPointsIds, Map* pMap){
    //Your implementation here
}
```

Listing 5: Function skeleton for task 4

After you implement this task, the system should be able to track the camera pose while the camera stays close to the initialization pose, where the initial map points are still visible. For this task, we ask you to evaluate the trajectory you get, and compare with the RMSE ATE you obtained in task 3.

3.6 Task 5 – Fish-eye camera model

As a final task, you should extend the current Mini-SLAM system to add support for a new type of camera. For that, we will use the TUM-VI dataset which uses a fish-eye camera. You can download the

`room1_512` sequence from here. For simplicity, we already provide a `DataLoader` for this dataset.

First, have a look at `Modules/Calibration/Pinhole.h/.cc` as a reference for the implementation of the new camera module. Use the Kannala-Brandt model in Appendix A to fill `project`, `unproject` and `projectJac`. Leave the `unprojectJac` method empty.

Then create a new application in the `Apps` folder that uses a `TUMVILoader` and runs Mini-SLAM with the `room1` sequence. Have a look at `mono_tumrgbd.cc` for reference. Finally, you will need to update the `Modules/System/Settings.cc` and `Data/*.yaml` files to support the new camera.

You should now be able to triangulate a new map and track the camera for a few frames. Try to choose a start frame with some camera movement to get better results and calculate the RMSE ATE as in the previous tasks.

4 Results

You should write a short report presenting and discussing the results of each task and submit it to Moodle, along with the code you have written, by 23:59 on the Monday following the Lab 3.2 session.

Prepare your report using the LaTeX typesetting system with a single column. Follow the *recommendations for writing your reports* that you can find in Moodle and explain how each participant in the group contributed to the lab. The report must answer at least the following questions:

Task 1 (2 pts.): What is your strategy for finding matches? Will it always work? Do you need to make assumptions? – How do you decide if a match is correct? Is your strategy robust? – How do you select the best match for each point? – What is the average number of matches that you get? Show some examples of your final matches between two images.

Task 2 (2 pt.): Write down the mathematical formulae for the pinhole projection and its Jacobian. How did you get the expression for `unproject`? – Is the initialized map correct? And why? Show an example. – Why is the initialization not always the same? Check the relative pose matrix for several executions. Is it always the same, even for the same pair of frames? Why is this?

Task 3 (2 pts.): What is the projection error? What would happen if some of the points in Task 1 were mismatched? – Report the length of the trajectory and the absolute translation error (ATE) for several runs of Mini-SLAM (at least 10).

Task 4 (2 pts.): What strategy did you use to find the matches? Evaluate the trajectory again and compare it with the results you obtained in Task 3.

Task 5 (2 pts.): Write a step-by-step derivation of the Jacobian of the projection. – Briefly discuss the changes you had to make to the code to complete this task and evaluate the performance of the new system as in the previous sections.

References

[1] J. Kannala and S.S. Brandt. "A generic camera model and calibration method for conventional, wide-angle, and fish-eye lenses". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 28.8 (2006), pp. 1335–1340. DOI: 10.1109/TPAMI.2006.153.

A The Kannala-Brandt model [1]

The Kannala-Brandt (KB) camera model is a generic fisheye lens model that extends the conventional pinhole model to handle extreme wide-angle lenses. Unlike standard models that assume rectilinear projection, the KB model expresses the projection using a radial distortion function.

A.1 Projection model

Given a 3D point $\mathbf{x} = (x, y, z)$ in the camera coordinate system, we first compute:

$$r = \sqrt{x^2 + y^2}, \quad \theta = \arctan\left(\frac{r}{z}\right) \tag{2}$$

The distorted radius in the image plane is given by a polynomial function:

$$d(\theta) = \theta + k_1\theta^3 + k_2\theta^5 + k_3\theta^7 + k_4\theta^9 \tag{3}$$

where k_i are distortion coefficients. The final pixel coordinates (u, v) are:

$$u = f_x d(\theta)\frac{x}{r} + c_x, \quad v = f_y d(\theta)\frac{y}{r} + c_y \tag{4}$$

where (f_x, f_y) are the focal lengths and (c_x, c_y) is the principal point in pixels.

A.2 Backprojection model

To recover the direction of a 3D point from an image pixel (u, v):

$$m_x = \frac{u - c_x}{f_x}, \quad m_y = \frac{v - c_y}{f_y}, \quad r' = \sqrt{m_x^2 + m_y^2} \tag{5}$$

The inverse distortion function is applied to obtain θ:

$$\theta = d^{-1}(r') \tag{6}$$

The inverse function can be obtained by fitting an inverse polynomial, by using a precomputed look-up table, or by numerical approximation using Newton's method to obtain the roots.
The direction vector in the camera coordinate system is then computed as:

$$\mathbf{d} = \left(\sin\theta\frac{m_x}{r'}, \sin\theta\frac{m_y}{r'}, \cos\theta\right) \tag{7}$$

which gives the unit direction of the 3D point relative to the camera.