SLAM overview

The state-of-the-art SLAM methods are somewhat difficult to grasp due to their complexity and unexpected tight couplings / leaky abstractions when trying to find out what the main parts are. How they should play with visual-inertial tracking - what we already have - is also unclear. This document attempts to list the essential ingredients in SLAM without making strong assumptions on how they should be assembled into an implementation.

Inputs

Main input: Frames

The main input of a Visual SLAM system is a sequence of camera frames. In more detail, the frames consists of data from one (monocular) or more cameras, for each camera, we have

- **Timestamp**
- **Bitmap image**, usually only grayscale data is used
- Intrinsic **camera parameters**: focal length, principal point, non-linear calibration parameters. May change from frame to frame or be assumed constant. Either known (most relevant for our case) or unknown and calibrated online
- Also an intrinsic parameter: Relative pose of the camera w.r.t. the other cameras (or the IMU). Can be assumed to be know

Input from VIO

Let us assume that we have access to a VIO system provides for each frame

- **VIO track**: 6DoF poses in an arbitrary coordinate system, drifts over time
- (optional) Feature tracks: list of ID, x, y (for each camera in case of stereo)
- (optional) Triangulated feature 3D positions
- ...

It is possible to have different levels of coupling between SLAM and VIO. In the extreme case, there would be no separate VIO and the “VIO input” would be just raw IMU. On the other hand, it is also possible to do visual-only SLAM without any IMU or VIO input. In practice, visual-only SLAM systems (internally) compute something very similar to the VIO track in that case.

Outputs

The goal of SLAM is to convert the sequence of frames to a sequence 6-DoF poses and a visual map (in an abstract sense), both defined in an arbitrary coordinate system.

6-DoF pose sequence

Even though the main output is the pose of the latest frame, poses of historical frames can be relevant too and they do not stay fixed but their poses are refined as the algorithm progresses.

Additional poses: Anchors

Since “everything moves”, including historical positions, AR systems use the concept of “anchors” to refer to user-defined poses that the SLAM algorithm may move around to keep them consistent with the visual map and current pose. Internally, they might use the historical camera poses, the visual map (or both?) as reference. In practice, they represent the positions of the “AR objects”.

Visual map

The visual map has three purposes

- **Visual relocalization**: Allows basically converting an isolated frame to a pose (location) without any context of previous frames. This feature can be used, at least,
  - Internally by the SLAM algorithm (loop closures)
  - Restarting tracking on purpose (new session, e.g, next day) or on failure
  - To share the coordinate system between multiple devices (called “cloud anchors” in ARCore)
- Can help / improve / enable **visual tracking** / local tracking - also during the “happy path” with no major drift
- **Structure-from-Motion** (SfM): reconstructing 3D geometry of the environment, e.g.,
  - Point clouds
  - Depth maps
  - Plane tracking
  - Dense representations / voxels

Visual positioning systems

It is also possible to create the visual maps offline / beforehand in a mapping phase and not update them online. Pre-generated visual maps can also be anchored to global coordinates (e.g., WGS). Such an approach is part of a wider class of methods known as visual positioning system (VPS). SLAM means that the map can be created from scratch without ground truth data (or is at least updated while the algorithm is running).

On the other hand, it could be possible to anchor SLAM-generated maps from one or more sessions to a global map afterwards and this map could be used with new SLAM sessions. This is why the line between SLAM and VPS could get blurred.

Internal state & problem description

Let us restrict further discussion to sparse / feature-based methods. This rules out approaches like LSD-SLAM and Kinect Fusion (a related method for accurate RGB+Depth data).

The SLAM problem description is, based on the input frame sequence

- creating and **continuously updating the visual map**
- **local tracking**: knowing the pose of the most recent frame
- (optional) providing a visual relocalization capability

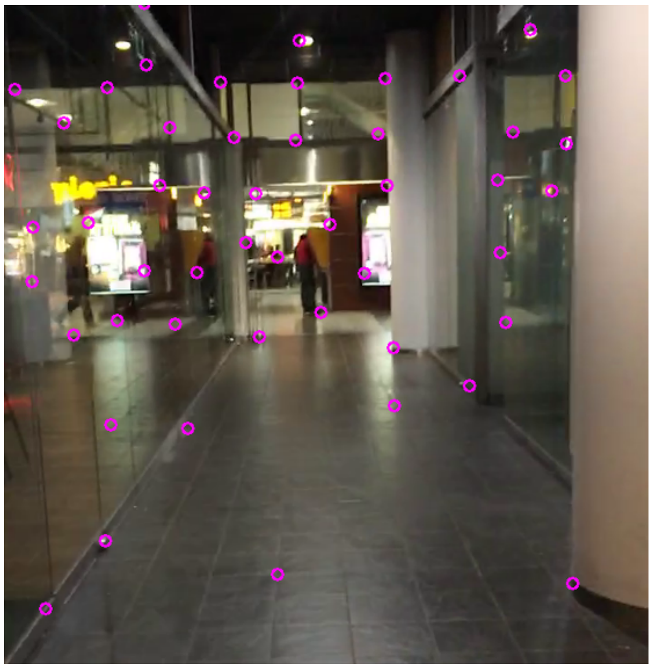
Keyframes

In online SLAM methods, input frames appear at a high frame rate and keeping all of them in memory is not practical over any reasonably long time period. Instead, most frames are immediately discarded once a newer frame appears. The frames that are kept (for any longer) are called *keyframes* and the information kept about them consist of

- **List of 2D features** (see below), some may be discarded as the algorithm proceeds
- For each feature, the 3D feature point it is associated with, if any (initially all unset)
- **6DoF-pose**, which is updated as the algorithm proceeds
- Camera parameters & timestamp (static)

2D features / keypoints

Like in visual tracking, features correspond to 2D image points that can appear in multiple frames. In addition to the (possibly sub-pixel accurate) 2D coordinates, the pixel neighbourhood of the points is somehow used to identify the feature.

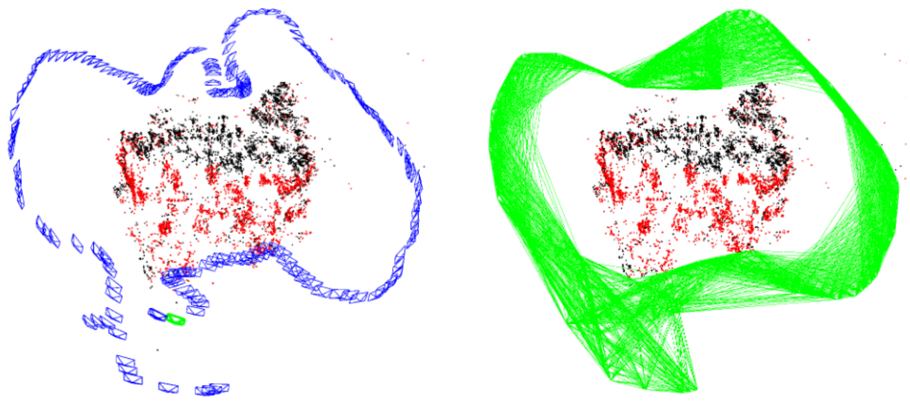

Example 2D features / keypoints

3D features / map points

When a 2D feature is seen from multiple frames (i.e., matched) or, in the stereo case, just with both cameras in the same frame, they can be **triangulated**, which results in (initial) 3D positions associated with the feature. Triangulated feature positions can be later refined in bundle adjustment.

Pose graph

The keyframes are connected via a *pose graph* or *co-visibility graph*, where an edge between keyframes means they share (enough) common matched visual features. The algorithms can also use various subgraphs of the pose graph for performance reasons



(a) KeyFrames (blue), Current Camera (green), MapPoints (black, red), Current Local MapPoints (red)

(b) Covisibility Graph

Pose graph (co-visibility graph) from the ORB-SLAM paper © IEEE

Visual map

The visual map of sparse SLAM methods consists of

- **The list all 3D features** (= the point full cloud), which may also include information from the 2D features (like ORB descriptors) and keyframes based on which they were triangulated. This may include restrictions on viewing angle and distance at which they are "valid". These restrictions are related to the properties of ORB (or other) feature descriptors and respecting them should the accuracy of the method
- **The pose graph**, which connects the 3D features to **keyframe poses** and the 2D feature descriptors

Local tracking

The current frame is usually not a keyframe, but we would still like to know its pose relative to the keyframes. We may want to keep track of its visual features to make the *keyframe decision*, e.g., promoting a frame to a keyframe only if its pose or features are different enough from other keyframes. When creating the new keyframe, its initial pose (and initially matched 2D features) are based on local tracking.

Note that the most similar keyframe, the *reference keyframe* of local tracking, may or may not be the latest one added in the map. Local tracking may utilize the *local map*, which consist of the map points that are currently likely to be visible (based on the neighbourhood of the reference keyframe in the pose graph).

Local tracking is similar to VIO tracking and can use the VIO track as one input.

Techniques

The lego blocks used in solving the SLAM problem

2D feature matching & tracking

Descriptors

One way to identify the features is to explicitly compute a descriptor of its pixel neighbourhood (this can also utilize multiple scales in an *image pyramid* representation of the frame). For example, an ORB descriptor is a sequence of (256?) bits computed from the neighbourhood of a point using a clever algorithm (which we need not touch).

Feature matching

With ORB, or other, descriptors, one can match 2D features between any two images by comparing the descriptors (e.g., computing the hamming distance of ORB bits). There are various approaches in doing the comparison: brute force (all pairs?), FLANN, DBoW tree tricks (used by ORB-SLAM). We can also have different prior knowledge about the expected 2D locations of some features that may also help with the matching problem.

Feature tracking

As an alternative to descriptor matching, the Lucas-Kanade sparse optical flow algorithm (also used in VIO) can accurately match 2D features between consecutive frames, especially if the movement between the features is small. It does that without using explicitly formed descriptors but instead looks at the raw images.

RANSAC / filtering

With both feature matching and tracking approaches, the methods will produce some bad matches. It is possible to filter out some bad matches (only based on the 2D feature information) with RANSAC-methods (e.g., RANSAC-2 / RANSAC-5). Bad matches / features can also be excluded based on outlier detection in other algorithm phases (e.g. PIVO blacklisting).

Bundle adjustment

Bundle adjustment means the simultaneous adjustments of keyframe poses and 3D map points. It’s a well-studied and understood optimization problem and efficient open source implementations (such as g2o & Ceres) are available. The question is, what parts of the visual map to include in the bundle adjustment at which time.

For example, ORB-SLAM & OpenVSLAM use it in two ways at different phases

- **Local bundle adjustment**: Fix the poses of the 2nd (or 3rd) neighbourhood of the current keyframe in the pose graph and only update the poses & map points in the immediate neighbourhood. This is done often
- **Global bundle adjustment.** Include the whole graph or a sparser, but global, subset (the “essential graph”) in the bundle adjustment problem. Done less regularly. Mainly on loop closures (?)

Loop closures

Loop closure candidates

Loop closures are needed when local tracking is not enough. They are similar to visual relocalization. In ORB-SLAM, the visual map is searched for (ORB) matches with the current frame. If a neighbourhood of keyframes all containing good matches are found, then the method attempts to find the pose based on these features, using a certain RANSAC-style method. If it looks good enough, it can be refined (with, e.g., local bundle adjustment?).

DBoW

If there is a large collection of frames from which to find ORB match candidates, the DBoW tree approach can be used to speed up the search of finding the good candidate frames.

Deduplication

In the loop closure, certain map points associated with the current frame will be identified with map points in the loop closure target keyframes, which leads to deduplication of map points (and new edges in the pose graph). Deduplication could also happen when normal new keyframes are added (without loop closures).

Global history correction

After a loop closure, there is an error in the recent history that needs to be corrected (before actual bundle adjustment?)

Pose graph culling

Redundant keyframes can be dropped from the pose graph to increase the performance (and accuracy) of the method.

3D outlier detection

Bundle adjustment, triangulation, deduplication, loop closures etc. can also discard 3D points after they have been triangulated.

Open implementation questions

TODO… This part should discuss the actual implementation options
(Not up to date as of 2020-03-25)

SLAM-VIO coupling

It is possible to have different levels of coupling between SLAM and VIO. Examples:

- No SLAM -> VIO feedback: SLAM only looks at deltas in VIO poses and assumes the tracking eventually recovers if “lost” or behaves badly (or e.g. the whole system is reset if a failure is detected OR just do not handle such failure cases first)
- Loose coupling: SLAM updates the VIO state with the latest pose.
- Tight coupling: VIO shares things like the visual map with SLAM
- Very tight coupling: SLAM inputs raw IMU, no separate VIO

Local tracking implementation

The simplest option is just using the VIO track for the pose (and VIO features for keyframe decisions). Then any feedback to VIO would be tied to keyframes only.

Keyframe addition process

TODO: What exactly happens when a keyframe is added, in what order. We had some plans & ideas about this, list them here

Currently

1. ORB-features extracted evenly over image
2. ORB-features associated with VIO-features (*NOTE: this only used in matching ORB features with each other, and does not change the pixel coordinates of ORBs to LK-tracker-based coordinates from VIO results*)
3. Descriptors computed
4. ORB-features with VIO tracks matched with previous features from tracks, triangulated if not already -> mappoints created
5. Features matched between new keyframe and neighborhood -> new mappoints created / features associated with already existing
6. Mappoints deduplicated:
    a. Project mappoints into keyframe
    b. Match with feature with best hamming distance

    c. Do for both
7. Local BA

Planned

1. Same
2. Same
3. Same
4. Feature matching: current frame <-> reference keyframe
    a. 2D: ORB features in the current frame matched with previous/reference keyframe based on tracker results & ORB descriptor check
    b. 3D reprojection: already triangulated features / map points in the reference keyframe are reprojected to the current frame & ORB matched
    c. 2D: untriangulated ORB features in the reference keyframe are matched with the current frame
5. Feature matching between each neighbour keyframe *K* of the reference keyframe in the pose graph:
    a. same as 4b & 4c, but for *K*.
    b. If enough matches are found, a **new edge is added** between the new keyframe and *K* in the **pose graph**
6. Deduplication: Step 5 may match two different features, both of which have already been independently triangulated, which are then deduplicated/identified at this step
7. Triangulation
    a. Find *untriangulated* features in the current keyframe with enough baseline (or other criteria): Two-camera triangulation for those features using the pair of keyframes with the longest baseline (if there are multiple choices). Use, e.g., the midpoint method. On success, promote the status *triangulated*.
    b. Find *triangulated* but unmapped points in the current frame and triangulate with all available keyframes where these points are visible (using “bundle adjustment with fixed cameras”). If the result looks good in terms of reprojection error, promote add to map (i.e., promote to *mapped*). Otherwise permanently discard the feature (?)
8. Local BA: with all map points in the current keyframe. Outlier features are permanently discarded (Note: could need a status DELETED if these are VIO tracker features)

Local tracking & reference keyframe change

(Optional feature at first but good to keep in mind.). It is possible that a new keyframe is not connected to the latest keyframe but another keyframe in the pose graph.

This could be achieved by running steps 1 - 5a for each frame and, changing the reference keyframe to the frame with most matches.

This should obviously not modify anything in the actual pose graph / keyframes but just the data stored for the current frame, which is a “candidate keyframe” that may or may not be promoted to a keyframe.

A newly added keyframe automatically also becomes the new reference keyframe.

Note that changing the reference keyframe also changes the which pose graph neighbourhood is considered so if you would run steps 1-5a again after changing the reference keyframe, you might get a different result, which may lead to rapid switching of the reference keyframe back and forth between good matches on each frame, but we should first check if this is an actual problem and then deal with it in that case