GSoC 2017 Proposal: MRPT

Robust SLAM/localization with artificial visual landmarks and stereo vision

About Me

Name
Shivang Agrawal
University
IIT Kharagpur
Email
shivang1997agrawal@gmail.com
Github
shivangag
Timezone
IST (UTC +5:30)
Contact No
+91 9455538734
Personal Background

I am a second year undergraduate student in the department of chemical engineering at Indian Institute of Technology, Kharagpur. I am a robotics-enthusiast and am currently a member of a student run research group working on an autonomous aerial vehicle that can localize itself in GPS denied environment. The goal is to participate in the International Aerial Robotics Competition (IARC). I have completed these robotics courses Perception (Coursera), Computational Motion Planning (Coursera), Estimation and Learning (Coursera), Introduction to Computer Vision (Udacity) and Artificial Intelligence for Robotics (Udacity), that makes me well versed with all the jargons and techniques in the field of robotics and also comfortable enough to understand most of the research papers from the field. I have 2+ year working experience in C++ and 4+ year working with OOP concepts through java. I am proficient in working with VCS like git. My development environment is primarily Ubuntu 14.04 with sublime text editor. I am a quick learner and I look forward to learning new things that may pop up in between the completion of the project.

Abstract

The simultaneous localization and mapping (SLAM), a problem of constructing or updating a map of an unknown environment while simultaneously keeping track of an agent’s location within it, has been intensively studied in the robotics community in the past. There exist several methods to solve it, at least approximately. One category of these methods is Visual SLAM, which primarily uses visual(camera) sensors to extract landmark information.

The current method of finding feature points in an image and its correspondence in other image using detectors and descriptors like SIFT, SURF, ORB etc, although flexible, are computationally expensive and are prone to errors.
A less flexible but more robust method is to equip the environment with some fiducial markers that can be detected accurately to sub pixel level and identified using some IDs, removing the overhead of computing feature points in an image and finding their correspondence in other image. The method can be used to provide a robust SLAM and localization algorithm to a robot in controlled environment, or to establish ground truth values to test various other SLAM algorithms against it.

Proposal | Algorithm

The algorithm to be followed for the SLAM and localization will be as followed. Some part of it has been directly taken from this paper.

1. Get the initial stereo images from a calibrated stereo camera.
2. Get the marker points vector $p_l$ and $p_r$ for left and right camera.
3. Triangulate the points found in previous step in 3D coordinates with respect to left camera center.
   a. If $K_l$ and $K_r$ are intrinsic camera calibration matrix of Left and Right camera and $R$ and $T$ are the Rotation and Translation of the right camera with respect to left camera then
      $$P_1 = K_L \begin{bmatrix} I_{3x3} & 0 \end{bmatrix}$$
      $$P_2 = K_r \begin{bmatrix} R & T \end{bmatrix}$$
      are the projection matrix for left and right camera, hence
      $$\lambda_1 \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = P_1 \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}.$$
to get the value of $X$, $Y$, and $Z$ we can then solve the equation

$$
\lambda_2 \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = P_2 \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}
$$

using singular value decomposition of $A$ matrix.

4. Add these points to `mrpt::vision::TLandmarkLocationsVec` vector.
5. Add left camera frame pose to `mrpt::vision::TFramePosesVec` vector.
6. Get next stereo image and Repeat step 2 and 3.
7. We now have two sets of rigid 3D-points $X = \{X_1, X_2, \ldots, X_n\}$ and $Y = \{Y_1, Y_2, \ldots, Y_n\}$ where $X_i$ and $Y_i$ are the 3D-coordinates of the $i$-th interest point before and after the motion. To get translation vector $T$ and rotation matrix $R$ between the left camera before and after the motion follow these steps:
   a. Calculate the centroids
\[ \mu_x = \frac{1}{n} \sum_{i=1}^{n} X_i \]
\[ \mu_y = \frac{1}{n} \sum_{i=1}^{n} Y_i \]

b. Calculate the covariance:
\[ \Sigma_{xy} = \frac{1}{n} \sum_{i=1}^{n} (Y_i - \mu_y)(X_i - \mu_x)^T \]

c. Let \( UDV^T \) be the singular value decomposition of \( \Sigma_{xy} \), \( \text{SVD}(\Sigma_{xy}) \)

d. Now \( S = I \), if \( \det(U) \cdot \det(V) = 1 \), else if \( \det(U) \cdot \det(V) = -1 \), \( S = \text{diag}(1, 1, \ldots, -1) \)

e. Then \( R = USV^T \)

f. \( T = \mu_y - R\mu_x \)

8. Add \( R \) and \( T \) transformation to previous frame’s \( R \) and \( T \) to get the rotation and translation of current frame with respect to initial camera frame.

9. Repeat Step 4 and 5.

10. Bundle Adjust the last n frames.

11. Repeat step 6 to 9, while the new frame has at least one tag that has been seen lesser than \( m \) number of times.

12. If all the tags in a frame has been seen greater than a number \( m \), predefined by the user,
Then we can estimate the pose of the camera using 3D to 2D point correspondence.
Proposal | Specifications

The Project can be divided into these specific tasks:

- **Integrate marker detection libraries to mrpt-detector module**

  The goal of the task is to create a generic marker detection and storage class that can detect and store a subset of the currently available markers like apriitags, aruco, chilitags, reactIVision. A brief review of apriitags and aruco done by me can be found [here](#). I propose to add an interface for the apritags and aruco detector libraries to the mrpt marker detector class. Other detector libraries like one mentioned above can also be detected by aruco hence we can skip them for now on in favour of using aruco. As it stands out apritags can also be detected by aruco, But because of widespread use of apritag in robotics community, no external dependencies, and being written in C makes it more portable for multi-platform libraries like mrpt. Anyway, the proposed class will be generic enough to add new detector libs. Now the generic detector class can be implemented in two ways presented here with pros and cons:

  - Implement an **abstract** marker detection class that can be **extended** for different detector **libraries**.
    - Pros: New detector libraries can be added easily.
    - Cons: Increased overhead on user to use specific class for a specific tag system.
  - Implement a generic header file which has opaque pointers to detection parameters, Marker detection, tag family and other objects used by most of the libraries. The actual detection implementation can then be done in one cpp file.
Which library to use will then depend upon the tag family and if we can include that library using #IF guards.

- **Pros:** The user is abstracted from the internal implementation of the library as they only have to use one object for every tag.
- **Cons:** The CMarkerDetection.cpp file will have to be changed every time to add new detector libs.

- As suggested by Nikos Koukis, a more generic approach to implement the marker detector will be to create a detection policy and then create, for each detector-libs, a new implementation for the policy that can be passed to the **CMarkerDetection** class as template argument.

Also as both AprilTags and Aruco can work with multiple tag families we can also pass the tag family as template argument.

The policy interface that should be respected by each detection policy class are:

- **initialize():** To initialize all the parameters required for detection.
- **detect():** To detect the markers in the image.

An example of detection policy for aruco will be as follow:

```cpp
class CArucoDetectionPolicy{
private:
    int iThreshparam1, iThreshParam2;
    MarkerDetector MDetector;
public:
    template <class tag_family>
    void initialize(const mrpt::utils::CConfigFileBase &config){
        //Initialize using config file and tag_family
    }
    void detect(const CObservation *obs, vector<detectable_object>&detected){
        // Detect tags using aruco functions
    }
};
```
A prototype for CMarkerDetection class is then as follows:

```cpp
template <class TAG_FAMILY, DETECTION_POLICY>
class CMarkerDetection: public CObjectDetection{
public:
    void detectObjects_Impl(const mrpt::obs::CObservation *obs,
                            vector< detectable_object &detected>{
        detection_policy::detect(obs, detected);
    }
    void init(const mrpt::utils::CConfigFileBase &cfg){
        detection_policy::initialize<TAG_FAMILY>(cfg);
    }
};
```

This gives user the freedom to use any detector-lib they want by just creating a detection-policy class for it and passing it as a template argument to CMarkerDetection class.

For storing the marker points CDetectableMarker is proposed. After analysing many fiducial marker system, it was clear that all of libraries detected and returned the corner points of the tag in integer type, but to make it as generic as possible the following templatized data structure will be used

```cpp
template<class T>
vector< pair< T, T > > m_points;
```

The template will be useful in handling different image sizes easily.

- **Create a new CMarkerPoints class**

The object of this class will store the 3D poses of each corner point of tags, with the tag ID. The member function of the class, apart from the standard constructor and destructor, will be
● **vector**<**CPoint3D**> **getPoints**(**int** **ID**) to get a vector of **CPoint3D** for all the points within the tag.
● **void** **setPoints**(vector<**CPoint3D**> **new_tag**) to set the new **CPoint3D** for all the points within the tag.
● **void** **saveToFile**(**const** std::string &**file**)  
● **void** **changeCoordinatesReference**(**CPoint3D** **new_reference**)  

- **Create a new CMarkerMap that extends the CMetricMap**

This map will store the **CMarkerPoints** objects in a vector. The objects will be stored in the order of their first appearance. An array, with index corresponding to tag IDs and values equal to tags index in map’s vector, will be stored for quick lookup of a tag with given ID.

Some of the member functions of the class will be

- **void** **push_back**(CMarkerPoints **new_point**): To add new CMarkerPoints object to map.
- **CMarkerPoints** **getMarkerPoints**(int **ID**): To get the CMarkerPoints object with given tag ID.
- **void** **setMarkerPoints**(vector<**CPoint3D**> **new_tag**, int **ID**): To set the **CPoint3D** for all the points of the CMarkerPoints object with given tag ID.
- **void** **saveToFile**(**const** std::string &**file**)  
- **void** **changeCoordinatesReference**(**CPoint3D** **new_reference**)  

- **Create a new CMarkerStereoSLAM class that extends CMetricMapBuilder**
This class will build the CMarkerMap incrementally and at the same time store and improvise the trajectory of the camera, using the stereo images. It will call bundle_adj_full function to bundle adjust the last n frames as mention in the algorithm above.

Some of the member variables of the class will be

- **CMarkerMap map**: To store the map of the markers the robot has seen so far.
- **mrpt::vision::TFramePosesVec trajectory**: To store the trajectory of the camera frames.

The public member functions of the class will be:

- **initialize()**: To initialize the object.
- **addObservation()**: To add new observation to the map.
- **getCurrentPose()**: returns the pose of the newest camera frame
- **saveMapToFile()**: to store the marker map in a file
- **getTrajectory()**: returns the camera pose vector.
- **changeReferenceFrame()**: To change the reference frame of the landmark points and camera frame poses.

The private function of the classes will be:

- **getMarkerPoints()**: To get the CDetectableMarker object from the image observation
- **triangulatePoint()**: To triangulate a point in 3D using stereo image pair as mentioned in the algorithm above, to get an initial estimate of the points pose.
- **estimateRT()**: To get an initial estimate of R matrix and T vector of the given camera frame.

- **Create a new application for SLAM with stereo vision and markers**
The most important deliverable of this project for the end user will be an application that can be run out-of-the-box to provide a robot with SLAM capabilities in an environment with fiducial markers attached to the walls and/or ceiling. The user will have to provide the stereo camera calibration, the family of tag used and some other parameters to control the behavior of the program.

The Flow of the application will be as follows:

- Get Stereo images, either live or from a dataset, with calibration values.
- Initialize CMarkerStereoSLAM object with camera calibration values and tag detection parameters
- add observations to the object
- Get the latest camera frame Pose to localize.
- Integrate gui map viewer to view the camera poses and landmark points.

Timeline

1. **Community Bonding Period [May 5 - 30]**
   a. Learn more about mrpt codebase and get familiar with the coding style followed.

2. **Week 1 [May 30 - June 5]**
   a. CMarkerDetection and CDetectableMarker class.
   b. Add apriltag and aruco support.

3. **Week 2 [June 5 - June 12]**
   a. CMarkerPoints class.

4. **Week 3 [June 13 - June 20]**
   a. CMarkerMap class

5. **Week 5 [June 21 - June 30]**
a. Finishing touches to classes added so far.

b. CMarkerStereoSLAM class header file

c. Get points from stereo observation using CMarkerDetection

6. Week 6 & 7[July 1 - July 14]
   a. triangulatePoint().
   
b. estimateRT().
   
c. Populate CMarkerMap with few initial values

7. Week 8[July 15 - July 22]
   a. Integrate bundle_adj_full()
   
b. getTrajectory()
   
c. saveMarkerMap

8. Week 9 & 10[July 23 - Aug 7]
   a. Stereo SLAM application.

   a. Iron out the wrinkles
   
b. Documentation
   
c. Tests.
Stretch Goals

I also propose to create ROS wrappers for the new SLAM module, so that it can be included in the mrpt_slam ROS package. This can be done in last week of August, if time permits, else I will complete it after GSoC ends.

Plans for summer

I currently don’t have any plans for summer. I’ll be traveling to my hometown at the start of June and will work from there until July 15th, after which I’ll have to return to college for the new semester. I will continue working from my college thereafter. As I only have four courses (four credit each which means 4 hours classes each week) in my next semester, I’ll be able to devote 4-6 hours daily which should be enough to continue the project.