Monocular Visual SLAM System

Documentation

Visual SLAM Project Team

December 14, 2024

Contents

1 Introduction

This document provides comprehensive documentation for a Monocular Visual SLAM (Simultaneous Localization and Mapping) system. The system is designed to create both 2D and 3D maps of an environment using only a single camera.

1.1 System Overview

The system implements a feature-based visual SLAM approach that:

- Tracks camera motion using visual features
- Reconstructs 3D points from 2D image correspondences
- Generates both 2D trajectory maps and 3D point clouds
- Supports both recorded datasets and live video input

2 Installation

2.1 Dependencies

The system requires the following dependencies:

```
numpy>=1.19.4
opencv-python>=4.5.0
open3d>=0.13.0
matplotlib>=3.3.3
tqdm>=4.54.0
pathlib>=1.0.1
```

2.2 Docker Deployment

For deployment on Jetson Nano, a Docker container is provided. Build and run the container using:

```
# Build the container
docker build -t visual_slam_jetson .

# Run the container
docker run --runtime nvidia \
           --network host \
           --device /dev/video0:/dev/video0 \
           -v /tmp/.X11-unix:/tmp/.X11-unix \
           -e DISPLAY=$DISPLAY \
           visual_slam_jetson
```

3 System Architecture

3.1 Core Components

The system consists of several key components:

- Feature Detection and Matching
- Pose Estimation
- 3D Point Triangulation

- Scale Estimation
- Map Generation

3.2 Class Structure

The main `MonocularSlam` class contains the following key methods:

- `process_frame`: Main entry point for processing new frames
- `_match_frames`: Feature matching between consecutive frames
- `_triangulate_points`: 3D point reconstruction
- `_estimate_scale`: Scale recovery for monocular setup

4 Implementation Details

4.1 Feature Detection and Matching

The system uses SIFT features for robust detection and matching:

```
self.feature_detector = cv2.SIFT_create(nfeatures=2000)
FLANN_INDEX_KDTREE = 1
index_params = dict(algorithm=FLANN_INDEX_KDTREE, trees=5)
search_params = dict(checks=50)
self.matcher = cv2.FlannBasedMatcher(index_params, search_params)
```

4.2 Pose Estimation

Camera pose is estimated using the Essential matrix decomposition:

- Compute Essential matrix using RANSAC
- Decompose into rotation and translation
- Verify solution using triangulated points

4.3 Point Cloud Generation

3D points are reconstructed using triangulation:

- Filter points based on triangulation angle
- Remove points with large reprojection error
- Apply scale estimation for consistent reconstruction

5 Camera Calibration

5.1 Calibration Process

The system includes a camera calibration tool that:

- Uses a checkerboard pattern
- Captures multiple views
- Computes intrinsic parameters
- Saves calibration results

5.2 Usage

To calibrate your camera:

```
python camera_calibration.py
```

6 Usage Guide

6.1 Dataset Mode

To run the system with a dataset:

```
# In main.py
use_dataset = True
data_dir = "path/to/dataset"
```

6.2 Live Video Mode

To run with live video:

```
# In main.py
use_dataset = False
```

7 Output and Visualization

The system generates:

- 2D trajectory plot (`2D_trajectory.png`)
- 3D point cloud (`3D_map.pcd`)
- Real-time visualization of tracked features

8 Performance Considerations

8.1 Parameters

Key parameters affecting system performance:

- `min_matches`: Minimum feature matches (default: 100)
- `min_triangulation_angle`: Minimum angle for triangulation (default: 2.0°)
- `max_point_distance`: Maximum allowed point distance (default: 30.0m)

8.2 Optimization

Performance optimization strategies:

- Feature detection limit
- Point cloud filtering
- Keyframe selection criteria

9 Troubleshooting

Common issues and solutions:

- Poor tracking: Adjust feature detection parameters
- Incorrect scale: Check scale estimation parameters
- Missing points: Adjust triangulation filters

10 Future Improvements

Potential enhancements:

- Loop closure detection
- Bundle adjustment
- Dense reconstruction
- Real-time optimization