Universität de Girona

Intelligent Field and Robotic Systems Master

Lab 4 - Stereo Visual Odometry

Prepared by:

Davina Sanghera u1100402

Rihab Laroussi u1100330

Date: December 28, 2024

Contents

1. Objective

The objective of this lab is to implement two different methods for estimating stereo motion: the 3D-3D method and the 2D-3D method. In addition, the uncertainty associated with each method is visualised and analysed using the Monte Carlo method.

The dataset used in this lab is drawn from the "UTIAS Long-Term Localization and Mapping dataset". Specifically, one run is used which consists of a set of 981 stereo image pairs captured from a stereo system mounted on a Grizzly robot.

The first stages of the livescript have already been implemented. These steps include loading the stereo image pairs, initialising the camera parameters which only have a translational difference and no rotational difference, rectifying the images, extracting features using bucketing, performing circular matching to match the extracted features, estimating motion using both the 3D-3D and 2D-3D methods and finally plotting the trajectories generated by each method.

The remaining functions that we needed to implement were: select_based_on_z_distance, mvg_montecarlo_3d_to_3d_reg and mvg_montecarlo_2d_to_3d_reg. These functions allow Monte Carlo sensitivity analysis to be performed for each motion estimation method. Finally we implemented code to align the visual odometry and GPS trajectories and we visualised the result.

2. Feature Extraction and Circular Matching

Once the stereo pairs were loaded and rectified and the camera parameters were initialised, feature extraction was performed. Since standard feature extractor methods work on the entire image, they typically output features that are concentrated in feature-rich areas of the image and so other areas aren't represented within the feature extraction. Since visual odometry needs there to be consistent features across consecutive frames in order to estimate motion, this non-uniform distribution of features is a problem. Therefore, to address this, a method called bucketing was employed. This is where the image is discretized into a grid and features are extracted independently from each grid cell, making the distribution of extracted features more uniform across the image. A comparison between the original vs bucketed extracted features is shown below.

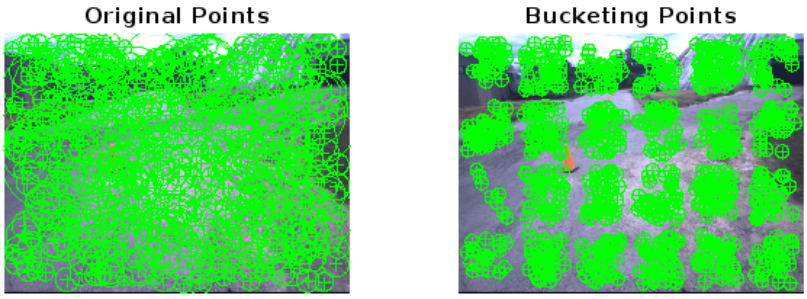

Figure 1: Original vs Bucketed Features

To match the features that were extracted, a method called circular matching was used. Circular matching compares a given feature in the left camera to all of the features in the right camera and takes the best match. The best match is then matched with the right image at the next timestep, t. Then this best match is compared with all of the features in the left image at the timestep, t. If the best match returns to the initial feature that we started with, it means the loop has been closed, indicating that this is a reliable match.

Q1.1 Does the circular matching use geometric constraints?

Circle matching doesn't use any geometric constraints, instead it verifies reliable feature matches by testing whether matching through to the next time step closes the loop and returns to the initial feature. No geometric constraints, such as epipolar constraints are enforced. The only requirement for matches is that they close the loop; if they fail to do so, then the match is discarded. One geometric constraint that you could choose to enforce is an epipolar constraint, which would ensure that matches lie on the corresponding epipolar line. You could impose this by ensuring that a given match (x and x') satisfies the epipolar constraint equation: $x'^{T}Fx = 0$.

Q1.2 Does the circular matching guarantee correct matches?

No circle matching doesn't guarantee correct matches, it only gives us a higher confidence in the matches that close the loop. This is because, due to noise, or too much similarity between features, there may be features that appear the same and are hard to distinguish between. Therefore, matches that are actually incorrect may still be able to close the loop.

3. Comparing Two Methods to Estimate Stereo Motion

With the obtained circular matches, the two different methods for estimating stereo motion were performed, namely the 3D-3D method and the 2D-3D method.

The 3D-3D method uses four images: the stereo pair from time t-1 and the stereo pair from the time t. The variable points3D_t1 contains the 3D scene points that were reconstructed via triangulation using the stereo pair at time t-1 and the variable points3D_t2 contains the same but for the stereo pair at time t. This results in two point clouds, one for each stereo pair.

Following this, 3D points outside of a certain depth (z) range are removed using the select_based_on_z_distance function that we implemented. Our implementation is given below.

```
function [points3D_t1_valid,points3D_t2_valid] = select_based_on_z_distance(points3D_t1,points3D_t2,zmin,zmax)
% Check points that are in a section of the 3D space

% Extract z coords
z_t1 = points3D_t1(:, 3);
z_t2 = points3D_t2(:, 3);

% Get indices where z coords are within range (zmin, zmax)
idxs_t1 = (z_t1 > zmin) & (z_t1 < zmax);
idxs_t2 = (z_t2 > zmin) & (z_t2 < zmax);
common_idxs = idxs_t1 & idxs_t2;

% Filter the points using the valid indices
points3D_t1_valid = points3D_t1(common_idxs, :);
points3D_t2_valid = points3D_t2(common_idxs, :);

end
```

Figure 2: select_based_on_z_distance function

To limit the depth of the 3D points to the range [zmin, zmax], first the z-coordinates of the 3D points are extracted. Then the indices of the z-coordinates that are both above zmin and below zmax are saved. Finally, these indices are used to filter the original set of 3D points and only return the points with valid depth values.

The point clouds both before and after depth range limitation are displayed below.

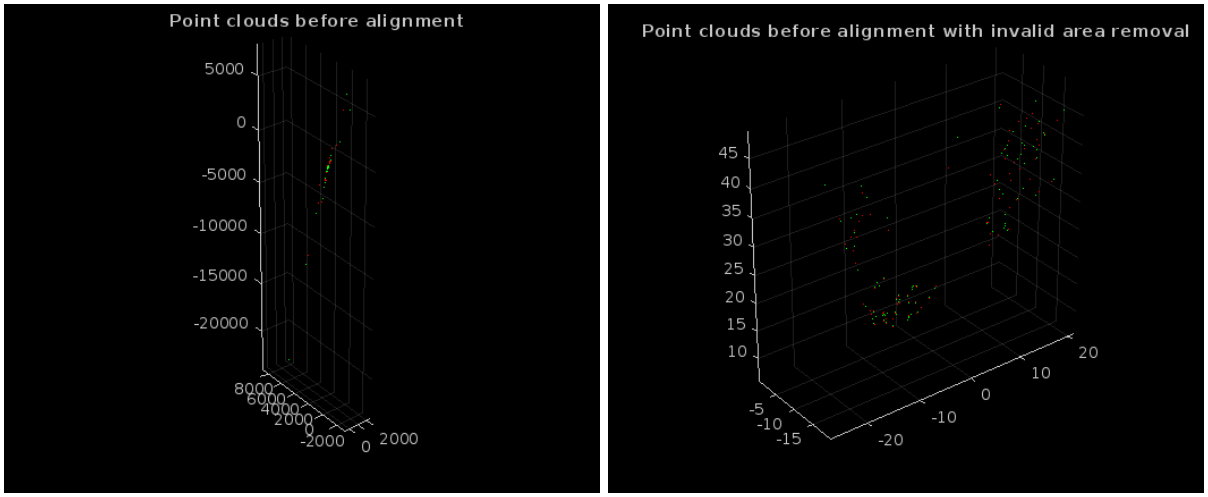

Figure 3: Point Clouds Before vs After Depth Range Limitation

The point clouds where the depth range is limited are much cleaner since the outliers with extreme depth values have now been removed.

Whilst the 3D-3D method estimated the translation between the cameras using a rigid 3D transformation model, the 2D-3D method estimates it by solving the Perspective-n-Point problem.

4. Monte Carlo Analysis

Next the Monte Carlo sensitivity analysis evaluates the robustness of the 3D-3D and 2D-3D methods to noise. This is done by adding Gaussian noise with a standard deviation of 0.1 to the feature points and then analyzing how the estimated translations vary over 1000 iterations. The translation stacks for the 3D-3D and 2D-3D methods were generated using the functions mvg_montecarlo_3d_to_3d_reg and mvg_montecarlo_2d_to_3d_reg respectively.

4.1 Method 3D to 3D

The mvg_montecarlo_3d_to_3d_reg function that we implemented is given below.

```
function [translationStack] = mvg_montecarlo_3d_to_3d_reg(pts_matches_l1,pts_matches_r1,pts_matches_l2,pts_matches_r2,stereoParams,noiseSTD,numRuns)
% function [resultStack] = mvg_montecarlo_3d_to_3d_reg(pts_matches_l1,pts_matches_r1,pts_matches_l2,pts_matches_r2,,numRuns,zmin, zmax)
%
% This function performs a Monte-Carlo runs using the method of 3D to 3D
% registration. Returns a stack with the translations for each run


% Initialize the stack
translationStack = zeros(numRuns,3);

for iterIdx = 1:numRuns

    % Add noise to the image points

    pts_matches_l1_noisy = pts_matches_l1 + randn(size(pts_matches_l1)) * noiseSTD;
    pts_matches_r1_noisy = pts_matches_r1 + randn(size(pts_matches_r1)) * noiseSTD;
    pts_matches_l2_noisy = pts_matches_l2 + randn(size(pts_matches_l2)) * noiseSTD;
    pts_matches_r2_noisy = pts_matches_r2 + randn(size(pts_matches_r2)) * noiseSTD;


    % Triangulation of 3D points from pair 1

    points3D_t1_noisy = triangulate(pts_matches_l1_noisy, pts_matches_r1_noisy, stereoParams);


    % Triangulation of 3D points from pair 2

    points3D_t2_noisy = triangulate(pts_matches_l2_noisy, pts_matches_r2_noisy, stereoParams);
```

```
    % Select only points that are in a section of the 3D space

% uncomment these three lines after completing the rest %%
    zmin = 0.5; % all points must be farther than this, in meters
    zmax = 50;  % all points must be closer than this, in meters
    [points3D_t1_noisy_valid,points3D_t2_noisy_valid] = select_based_on_z_distance(points3D_t1_noisy,points3D_t2_noisy,zmin,zmax);

    % Estimate the geometric transform

%% uncomment these two lines after completing the rest %%
    [estimatedTform_noisy] = estimateGeometricTransform3D(points3D_t2_noisy_valid, points3D_t1_noisy_valid,'rigid','MaxDistance',10);
    translationStack(iterIdx,:) = estimatedTform_noisy.Translation;
end


end
```

Figure 4: mvg_montecarlo_3d_to_3d_reg method

This method first adds the noise to the feature points. It then reconstructs the 3D scene points for each stereo pair using triangulation. Then it uses the select_based_on_z_distance function to limit the depth of the reconstructed points to the range [zmin, zmax]. Finally it estimates the rigid 3D transformation between the two point clouds using the function estimateGeometricTransform3D. The estimated translation is added to the stack, which is returned after the given number of iterations.

4.2 Method 2D to 3D

The mvg_montecarlo_3d_to_3d_reg function that we implemented is given below.

```
function [translationStack] = mvg_montecarlo_2d_to_3d_reg(pts_matches_l1,pts_matches_r1,pts_matches_l2,pts_matches_r2,stereoParams,noiseSTD,numRuns)
% function [resultStack] = mvg_montecarlo_2d_to_3d_reg(pts_matches_l1,pts_matches_r1,pts_matches_l2,pts_matches_r2,noiseSTD,noiseSTD)
%
% This function performs a Monte-Carlo runs using the method of 2D to 3D
% registration. Returns a stack with the translations for each run



% Initialize the stack
translationStack = zeros(numRuns,3);

for iterIdx = 1:numRuns

    % Add noise to the image points

    pts_matches_l1_noisy = pts_matches_l1 + randn(size(pts_matches_l1)) * noiseSTD;
    pts_matches_r1_noisy = pts_matches_r1 + randn(size(pts_matches_r1)) * noiseSTD;
    pts_matches_l2_noisy = pts_matches_l2 + randn(size(pts_matches_l2)) * noiseSTD;
    pts_matches_r2_noisy = pts_matches_r2 + randn(size(pts_matches_r2)) * noiseSTD;


    % Triangulation of 3D points from pair 1

    points3D_t1_noisy = triangulate(pts_matches_l1_noisy, pts_matches_r1_noisy, stereoParams);
```

```
    % Motion estimation

%% uncomment these two lines after completing the rest %%
    intrinsics = stereoParams.CameraParameters1.Intrinsics;
    [~,t] = estimateWorldCameraPose(pts_matches_l2_noisy,points3D_t1_noisy,intrinsics,'MaxReprojectionError', 10);
    translationStack(iterIdx,:) = t;
end


end
```

Figure 5: mvg_montecarlo_2d_to_3d_reg method

The same steps are followed as with the previous function. First noise is added, then the 3D scene points are reconstructed for the first stereo pair using triangulation. Finally the translation for the second stereo pair is estimated using the function estimateWorldCameraPose and it is added to the stack.

In order to see which method is performing better, the translation stacks are visualised. This is shown below.

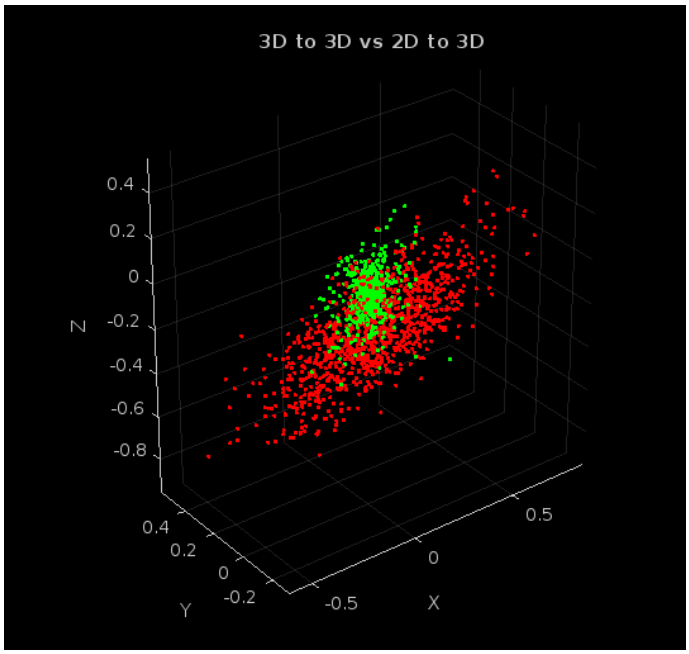

Figure 6: Monte Carlo Sensitivity Analysis

Here the red points are results from the 3D-3D method and the green points are results from the 2D-3D method. It is clear from the visualisation that the 2D-3D method performs much better than the 3D-3D method since the green points are clustered much closer together than the red points. This indicates that the 2D-3D method is more robust to noise since it generates more consistent translation estimates.

Q2 Why does the 3D to 3D method perform worse than the 2D to 3D method despite using 4 images as opposed to 3?

The 3D-3D method performs worse because it depends on two sets of triangulated 3D points, one for each stereo pair. Therefore, any errors such as incorrect feature matches or noise, are compounded when we align the two point clouds. On the other hand, the 2D-3D method only depends on one set of triangulated 3D points, and so errors are limited to this one triangulation and are less likely to accumulate. This is why the 2D-3D method is more robust to noise than the 3D-3D method.

5. Processing the sequence

Continuing with the live script, the next stage is to estimate the motion of the camera over time by processing a sequence of stereo images from the loaded dataset.

We begin by initializing the camera pose, and then we take stereo images at the given interval (step_frames). For each stereo image pair we perform rectification, extract features from them, perform circular matching and then estimate the motion using the 3D-3D method. We then obtain the updated camera pose by multiplying the previous camera pose by this estimated transformation. Each updated camera pose is appended to the variable t_hist, which stores the trajectory of the camera over time. The full estimated trajectory and the ground truth trajectory are visualised below.

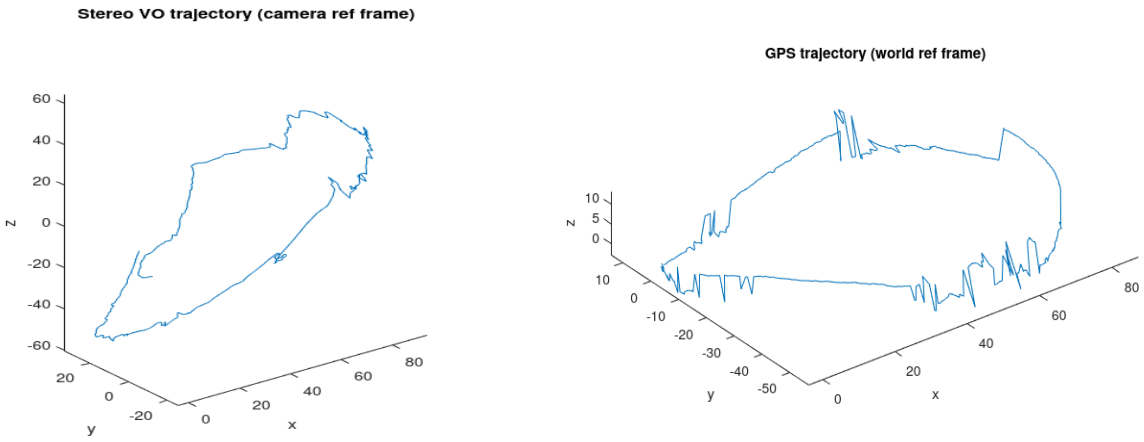

Figure 7: VO vs GPS Trajectory

- As you can see, the simple methods seen in this example have modest performance. How could it be improved? Define and describe additions to these methods, or alternative strategies that you consider that would improve the performance in terms of accuracy.

These simple methods could be improved by using some additional strategies to make them more robust. The first suggestion is to apply RANSAC during the feature extraction phase. The second suggestion is to use bundle adjustment for joint optimisation. The third suggestion is to dynamically limit the depth of the reconstructed points.

Although RANSAC is implicitly used within the estimateWorldCameraPose function for the 2D-3D method, it isn't used during the feature extraction stage. Currently only circular matching is used to verify matches. Therefore, RANSAC could be used as an additional step in order to eliminate any outliers that persist even after the circular matching process. This would improve the reliability of the matches that are used when estimating the camera poses and reconstructing the 3D points.

The next suggestion is to perform bundle adjustment, which jointly optimizes both the estimated camera poses and the reconstructed 3D points. This is done by minimising the reprojection error, which is the difference between the observed 2D feature points and the projected 3D reconstructed points. This process ensures that the 3D reconstruction of the scene is globally consistent across all the cameras and time steps. It also helps to reduce drift in the estimated trajectory. But it is important that the camera poses are initialised well to begin with, so that the optimisation process is less likely to get stuck in a local minima.

The final suggestion is to dynamically limit the depth of the reconstructed 3D points, instead of using the fixed depth limits [zmin, zmax]. By implementing an algorithm that dynamically changes these limits, based on known object sizes, scene geometry or camera motion, we can ensure that only relevant points are used to estimate the motion and points with extreme depth values are ignored.

Finally, we align the first 25% of the visual odometry trajectory with the ground truth GPS trajectory using the code below.

```
% Aligning the trajectories

VO_trajectory = t_hist;
GPS_trajectory = gpsLocation;

% Extract first 25% of the points from each trajectory
limit_25 = round(0.25 * size(VO_trajectory, 1)); % Limit for first 25%
VO_trajectory_25 = VO_trajectory(1:limit_25, :); % First 25% of VO trajectory
GPS_trajectory_25 = GPS_trajectory(1:limit_25, :); % First 25% of GPS trajectory


% Estimate transformation from VO to GPS
[tform, inlierIdx] = estimateGeometricTransform3D(VO_trajectory_25, GPS_trajectory_25, 'rigid', 'MaxDistance', 10);

% Apply the transformation to the complete VO trajectory
[VO_trajectory_aligned] = transformPointsForward(tform, VO_trajectory);

% Step 6: Plot the two trajectories for visualization
figure;
title('Aligned VO and GPS Trajectories');
plot3(GPS_trajectory(:, 1), GPS_trajectory(:, 2), GPS_trajectory(:, 3), 'g-', 'LineWidth', 2); % Plot GPS trajectory
hold on;
plot3(VO_trajectory_aligned(:, 1), VO_trajectory_aligned(:, 2), VO_trajectory_aligned(:, 3), 'r-', 'LineWidth', 2); % Plot aligned VO trajectory
xlabel('x'); ylabel('y'), zlabel('z');
legend('GPS Trajectory', 'Aligned VO Trajectory');
grid on;
axis equal;
```

Figure 8: Aligning the Trajectories

Here, the first 25% of each trajectory is extracted and then the transformation between these extracted trajectories is estimated using the 3D-3D method. This transformation is then applied to the full visual odometry trajectory, and the visualised result can be seen below.

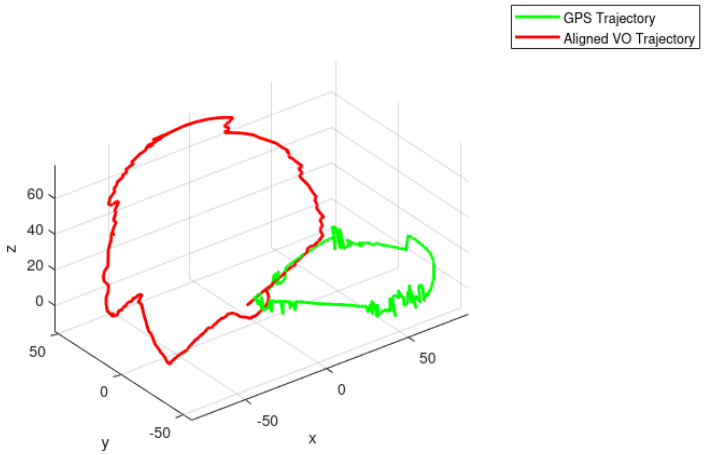

Figure 9: Aligned Trajectories

6. Conclusion

To conclude, this lab gave us hands-on experience with MATLAB's built-in functions to implement the different motion estimation methods that we covered in class. Furthermore, additional techniques such as depth limitation and circular matching were implemented, allowing us to gain a deeper understanding into how these methods operate. We were also introduced to Monte Carlo sensitivity analysis, which is a useful tool for evaluating the robustness of a motion estimation model to noise. Finally, we reflected on additional techniques that could be used to further improve the accuracy of the motion estimation models, ranging from improvements in the feature matching process to optimisation methods that refine the obtained estimations.