Final Project: Visual Odometry – Technical Report

CMSC 426: Computer Vision

Ahmed Ashry
5-16-2023

1. Introduction

This technical report provides a detailed exploration of a project focused on the application of Visual Odometry, a critical discipline within the field of computer vision. Visual Odometry involves the process of determining the position and orientation of a robot or vehicle by analyzing the associated camera images. The subject of this project was a series of sequential driving frames captured by a camera mounted on a car.

The objective was to reconstruct the three-dimensional (3D) trajectory of the camera, which, by implication, represents the movement path of the vehicle itself. The project embarked on a systematic series of steps, starting with the identification of shared keypoints across the series of frames. Subsequent steps involved estimating the fundamental and essential matrices between frames, decomposing the essential matrices into translation and rotation parameters, and plotting the camera center to visualize the 3D trajectory.

In order to achieve this, two distinct routes were undertaken. The first route was centered on the use of built-in functions provided by openCV, a popular open-source computer vision and machine learning software library. This approach offers the advantages of well-optimized, pre-existing algorithms and methods, thus enabling a streamlined pipeline for the project. However, the functions are black box, which is not easily adaptable or adjustable.

The second route sought to challenge the reliance on openCV's built-in functions by constructing an alternative implementation. This route involved creating custom functions to estimate the Fundamental and Essential matrices between successive frames, as well as to estimate the rotation and translation of the camera center. The motivation behind this approach was to gain a deeper understanding of the underlying processes and to explore the potential for customization and optimization beyond the capabilities provided by off-the-shelf functions.

This report will walk through both the approaches, discussing their benefits, challenges, and findings.

2. Implementation of Visual Odometry Using OpenCV Built-In Functions

The implementation of visual odometry using OpenCV built-in functions consisted of a series of steps, each of which is critical to understanding the process. This section discusses each step and the functions used in detail.

1. **Camera Calibration**:

The project commenced with camera calibration, a critical step that allows us to correctly interpret images from the camera. It was carried out using the $ReadCameraModel$ function, which yielded intrinsic parameters of the camera such as the focal length in x and y directions (fx, fy), and the

optical center coordinates (cx, cy). The intrinsic matrix K was then constructed with these parameters, serving as a blueprint of the camera's internal characteristics.

A Lookup Table (LUT) for undistortion was also obtained, which is essentially a precomputed map for pixel locations used to correct lens distortion. Undistorting the image is a crucial step as it translates the distorted image captured by the camera into a more accurate representation of the real world.

2. **Loading and Undistorting Images**:

The $load\_and\_undistort\_image$ function was created to facilitate the process of loading and undistorting images. $cv2.imread$ was used to load an image into memory, while $cv2.cvtColor$ was used to convert the raw image from $BayerGR$ to BGR format. In digital cameras, a Bayer filter mosaic is used for color image sensing, resulting in a $BayerGR$ image, which is then converted to a standard BGR color image.

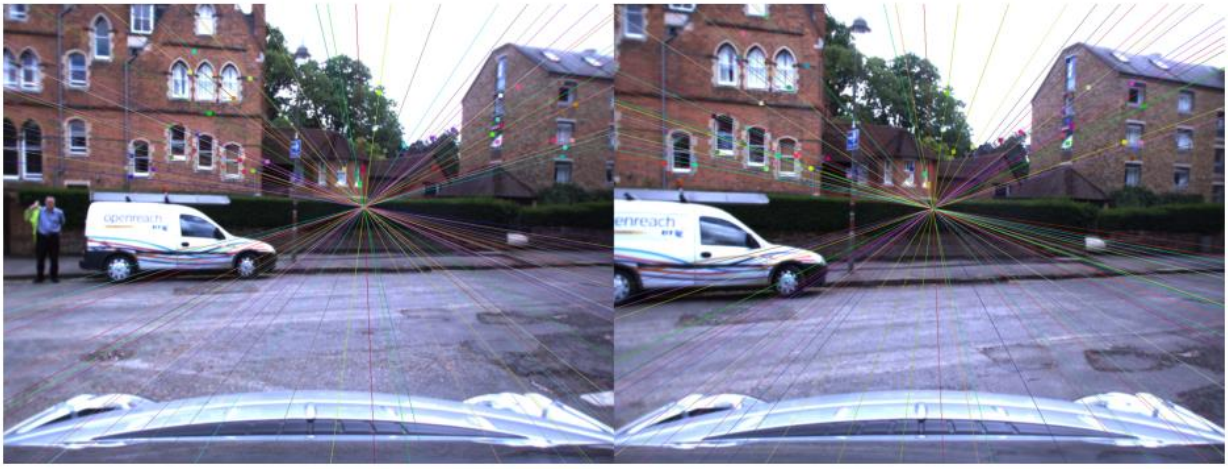
The UndistortImage function, which takes the color image and the LUT as inputs, was then used to undistort the images. This function essentially remaps the pixels of the image using the LUT, correcting for lens distortion.

3. **Identifying Keypoints and Matches**:

The $find\_keypoints\_and\_matches$ function employs the Scale-Invariant Feature Transform (SIFT) to detect and compute keypoints in images. SIFT keypoints are invariant to image scale and rotation, and they provide robust matching across a substantial range of affine distortion, change in 3D viewpoint, addition of noise, and change in illumination. The function then uses the Fast Library for Approximate Nearest Neighbors (FLANN) based matcher to find matches between the descriptors of two images. FLANN is an algorithm for performing fast approximate nearest neighbors searches in high dimensional spaces, and it's particularly well-suited for matching SIFT descriptors. Lowe's ratio test is then applied to filter out the best matches. This test rejects poor matches by computing the ratio between the best and second-best match. If this ratio is below a certain threshold (in this case, 0.5), the match is classified as 'good'. The figure below shows the matching points found after applying the above implementation for two random consequent frames.

4. **Estimating the Fundamental Matrix**:

The $estimate\_fundamental\_matrix$ function employs $cv2.findFundamentalMat$ to estimate the fundamental matrix using the RANSAC algorithm. The fundamental matrix encapsulates the geometric relationship between corresponding points in stereo images. It is a 3x3 matrix that satisfies the epipolar constraint, where a point in one image, when transformed by the fundamental matrix, should lie on the corresponding epipolar line in the other image. The RANSAC (Random Sample Consensus) method is a non-deterministic algorithm that robustly estimates parameters of a mathematical model by handling outliers. It iteratively selects a random subset of the data, fits the model to this subset, and then checks the fit against all data, counting inliers that fall within a specific threshold. The figure shown below is an example of drawing epipolar lines on two images, while enforcing a rank=2 constraint on the fundamental matrix.



5. **Estimating the Essential Matrix**:

The $estimate\_essential\_matrix$ function is used to compute the essential matrix from the corresponding points in two images. The essential matrix is a variant of the fundamental matrix with the additional constraint that it can only exist between two images taken from the same camera (or two identical cameras with parallel optical axes), as it encodes both the rotation and translation between the two images. $cv2.findEssentialMat$ is used to estimate the essential matrix. It also employs RANSAC to handle outliers and returns a mask that filters the inlier points.

6. **Estimating Camera Motion:**

The $estimate\_motion$ function brings together all the previous steps to estimate the camera motion between two consecutive images. $cv2.recoverPose$ is used to recover the rotation matrix (R) and the translation vector (t) from the essential matrix and the inlier matches. This function uses the singular value decomposition (SVD) technique and the properties of the essential matrix

to extract possible solutions for the camera's relative pose, and disambiguates between the four possible solutions using the cheirality check.

7. **Computing the Visual Odometry**:

The $visual\_odometry$ function implements the full pipeline of visual odometry, estimating the trajectory of the camera within the world frame. The function computes the visual odometry for all image pairs in a sequence, cumulatively updating the rotation and translation of the camera in the world frame. The new position of the camera is then appended to the trajectory.

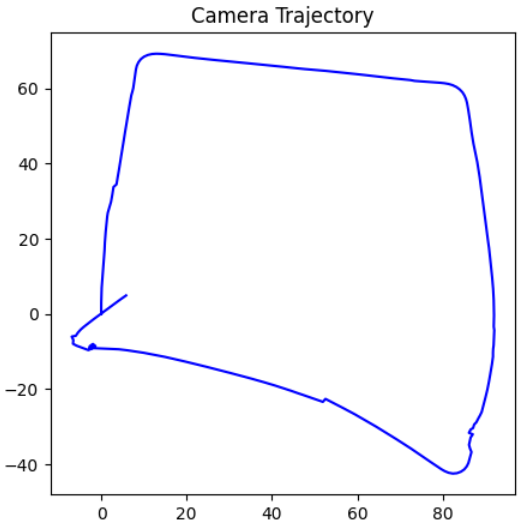
8. **Trajectory Visualization**:

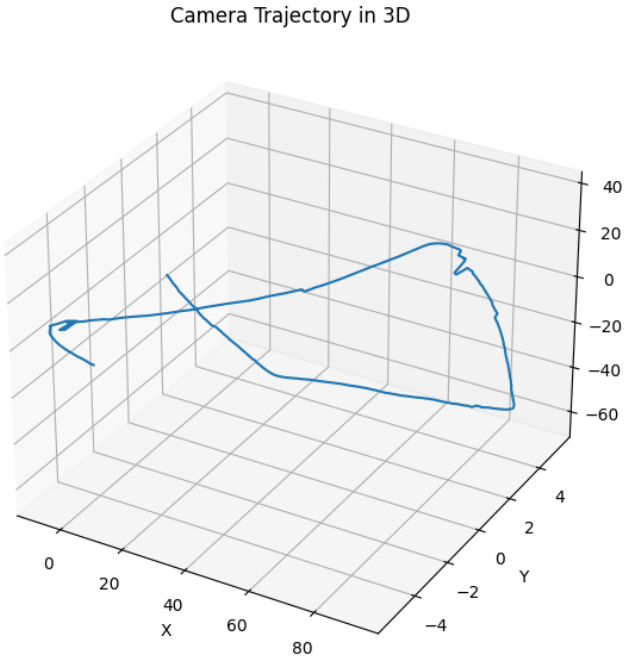
The final step is the visualization of the camera's trajectory. Using matplotlib's plt.plot function, the x and z coordinates of the camera positions are plotted. The y-coordinate is discarded in this case to represent the 3D trajectory in a 2D plane.

In summary, this visual odometry implementation utilizes various OpenCV functions that are designed for robust performance. While these functions provide a convenient and efficient way to perform complex tasks, understanding their underlying principles, as detailed above, is vital for the effective application and further customization of the pipeline.

Results – Using OpenCV:

The plot below shows the camera trajectory in 2D and 3D.





3. Manual Implementation

This implementation of visual odometry is more manually intensive and doesn't rely on OpenCV's built-in functions for fundamental or essential matrix calculation, camera pose recovery, or triangulation. The steps in this process are as follows:

1. **Reading the Camera Model**:

Like before, this code first reads the camera model from the provided file using the ReadCameraModel function. The function returns camera intrinsic parameters fx, fy, cx, cy, and a look-up table LUT for undistortion.

2. **Loading and Undistorting Image**:

Also, like earlier, the load_and_undistort_image function reads a raw image and converts the Bayer color format to BGR using OpenCV's cv2.cvtColor. The image is then undistorted using a custom UndistortImage function, which presumably uses the look-up table LUT from the camera model.

3. **Finding Keypoints and Matches**:

The $find\_keypoints\_and\_matches$ function is used to extract and match features between two images. Here, the SIFT (Scale-Invariant Feature Transform) algorithm is utilized to detect and compute keypoint descriptors. These descriptors provide a unique fingerprint of each keypoint, which allows for matching between different images. The matching is performed using the Brute-Force matcher ($BFMatcher$), a straightforward approach that calculates the distance between each pair of descriptors and selects the pairs with the smallest distances. To ensure the quality of matches, Lowe's ratio test is implemented, which filters out matches where the distance to the closest descriptor is not significantly smaller than the distance to the second-closest. This step helps in reducing the number of false matches. The function then returns the coordinates of the good matches in both images, along with their corresponding keypoints.

4. **Generating Zhang's Grid:**

The $generate\_zhang\_grid$ function is used to create a spatial grid over the image. This method divides the image into a specific number of cells (in this case, an 8x8 grid). Each grid cell will then contain the keypoints (matches) that fall within its boundaries. The objective is to ensure a spatially diverse set of matches for the subsequent 8-point algorithm, making the matches more evenly distributed across the image and thus improving the robustness of the fundamental matrix estimation.

5. **Implementing Eight-Point Algorithm to Estimate Fundamental Matrix with RANSAC:**

The $eight\_point\_algorithm$ function adopts the classic 8-point algorithm, which is used for estimating the **fundamental matrix** F from point correspondences, with the integration of RANSAC for outlier rejection and Zhang's method for spatial diversity. The function starts by creating a grid using the previously described $generate\_zhang\_grid$ function. It then runs through multiple iterations (RANSAC iterations). In each iteration, a point is randomly selected from each cell of the grid, ensuring a diverse distribution of points across the image. These points

are then normalized to have zero mean and unit variance, which is a crucial step to condition the numerical problem correctly. The fundamental matrix F is then calculated by solving a system of linear equations formed by the point correspondences. The computed F is rank-deficient, meaning it has a rank of 2 rather than 3. This property is enforced by singular value decomposition (SVD) and setting the smallest singular value to zero. After that, the fundamental matrix is denormalized back to its original scale. The inliers are then counted by checking if the epipolar constraint (the geometric constraint enforced by F) holds for each point within a predefined threshold. If it does, the point is considered an inlier; if not, it's an outlier. The iteration that yields the most inliers (i.e., the best F) is kept, and F is then further refined by considering only the inliers.

The $normalize$ function is used to prepare the point correspondences for the 8-point algorithm. The normalization is performed by translating and scaling the input points such that their centroid is at the origin (0,0), and their average distance from the origin is equal to $sqrt(2)$. This normalization step is crucial for conditioning the 8-point problem correctly, reducing numerical instability and ensuring a more accurate estimation of the fundamental matrix.

7. **Estimating Essential Matrix:**

The $estimate\_essential\_matrix$ function uses the computed fundamental matrix and camera intrinsic params to calculate the essential matrix E. The essential matrix describes the epipolar geometry between two images and can be obtained from the fundamental matrix when the camera's intrinsic parameters are known. After calculating E, its rank-2 property is enforced by setting its smallest singular value to zero.

8. **Calculating Camera Pose**:

The $calculate\_camera\_pose$ function decomposes the essential matrix into possible rotation R and translation t matrices using SVD. Four possible poses are generated, but only one of them is physically possible, which can be determined by checking the chirality condition – a criterion ensuring that the reconstructed 3D points are in front of both cameras.

9. **Implementing Linear Triangulation**:

The $linear\_triangulation$ function is used to estimate the 3D coordinates of a point given its 2D projections in two images and the relative pose between the two cameras. The function first normalizes the points by pre-multiplying them with the inverse of the camera matrix K. Then, it forms and solves a homogeneous linear system to find the 3D point. The system is solved using SVD, and the solution is the eigenvector associated with the smallest singular value.

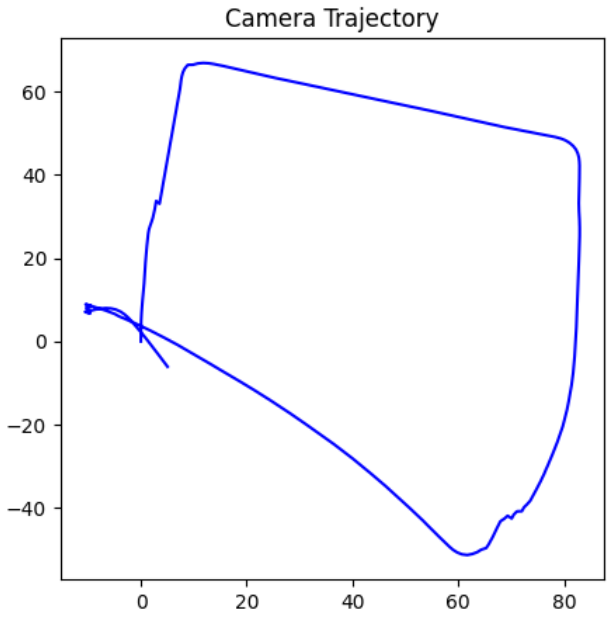
10. **Calculating Visual Odometry to Reconstruct the Trajectory**

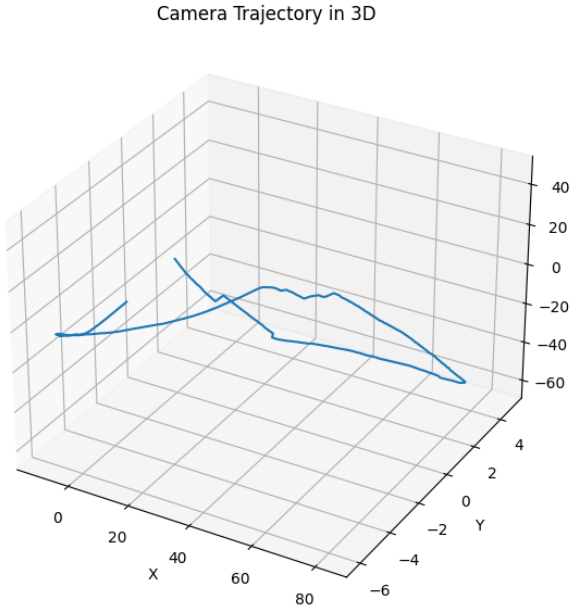
Finally, the $visual\_odometry$ function orchestrates the entire pipeline. It loops through all pairs of consecutive images and for each pair, it loads and undistorts the images, finds keypoint matches, estimates the fundamental matrix using the 8-point algorithm with RANSAC, computes the essential matrix, recovers the camera pose, and then estimates 3D points using linear triangulation. The calculated camera poses are accumulated to form the trajectory of the camera, representing the visual odometry.

This process is repeated for each consecutive pair of images, and the obtained camera trajectory provides an estimation of the path that the camera has traversed in the scene.

Results – Using Custom Implementation

Below are plots for 2D and 3D trajectory of the camera center over the different frames.





4. Challenges and Discussion

In this section, we delve into the performance analysis, challenges, and potential improvements related to our visual odometry pipeline.

1. **Performance Analysis**:

My experimentation and evaluation process involved implementing the pipeline using both OpenCV functions and custom-developed functions. The resulting trajectories from both implementations seemed to accurately reflect the expected motion of the camera, signifying the conceptual correctness of the implemented pipeline.

However, when comparing the two, it was observed that the OpenCV-based implementation performed better in terms of both computational speed and accuracy. The efficiency of OpenCV functions can be attributed to their highly optimized codebase, which leverages lower-level languages, parallelism, and hardware acceleration. In contrast, the custom functions, while correctly implemented, have not been similarly optimized and hence run slower.

2. **Sensitivity to Parameters**:

A significant aspect to note in this visual odometry pipeline is the sensitivity of the results to various parameters. Particularly, RANSAC's performance heavily depends on the threshold used to determine inliers and outliers. A lower threshold may lead to a higher chance of rejecting good matches, while a higher threshold may include more outliers, both of which could affect the accuracy of the fundamental matrix estimation.

Furthermore, other parameters such as the number of RANSAC iterations, the number of grid cells in Zhang's method, and the number of keypoints to extract can also greatly impact the performance. These parameters require careful tuning and may need to be adjusted depending on the specific characteristics of the dataset, including image quality, scene complexity, and camera motion.

3. **Need for Custom Function Optimization:**

The promising results obtained from the OpenCV-based implementation indicate that there's room for improvement in the custom implementation. While the custom functions have been correctly formulated and implemented as per the theory, their performance could potentially be enhanced through algorithmic optimization and more efficient coding practices. For instance, vectorization of certain operations, utilization of more efficient data structures, or even the use of parallel computing techniques could be considered to enhance the computational speed.

4. **Challenges and Potential Improvements:**

One of the significant challenges in implementing visual odometry is dealing with image noise and outliers, which can severely distort the estimation of the fundamental matrix and subsequent steps. Implementing robust outlier detection and noise handling mechanisms can greatly improve the accuracy and reliability of the pipeline.

The sensitivity to parameters could be addressed by incorporating automated parameter tuning techniques, like grid search or genetic algorithms. This could help in finding the optimal set of parameters that yield the best performance on a given dataset.

The computational complexity of feature matching can also be reduced by using more efficient matching algorithms or by incorporating a feature tracking approach, where keypoints are tracked across frames instead of being independently detected in each frame. This could potentially improve both the speed and the consistency of the matches.

Moreover, the robustness of the pipeline could be enhanced by integrating more sophisticated algorithms for fundamental matrix estimation, such as the 5-point algorithm, which can handle cases where the 8-point algorithm fails. Also, the integration of bundle adjustment for globally consistent optimization of the camera trajectory could potentially yield more accurate results.

In conclusion, while the implemented visual odometry pipeline shows promising results, it also opens up several avenues for improvement and optimization. These improvements, when effectively addressed, could lead to a more robust and efficient visual odometry system, capable of operating accurately under a wider range of conditions.

5. Conclusion

Throughout this technical report, we have delved into the intricate world of visual odometry, exploring the intricacies of estimating camera motion from successive image frames. We systematically reviewed the theoretical principles underpinning this technology and detailed our approach to implementing a visual odometry pipeline.

Our implementations, both using OpenCV functions and custom functions, have been tested and evaluated. Results showed that both versions were conceptually accurate, with the OpenCV-based implementation displaying superior performance in terms of computational efficiency and accuracy. However, our custom version provided an invaluable opportunity to deeply understand the mechanisms at play within the visual odometry pipeline.

We also discussed the challenges we faced and the potential improvements that could be made to our pipeline. The sensitivity to various parameters, the need for improved outlier handling, and the opportunity for further optimization in our custom functions all present avenues for future work. Our discussions also highlighted the potential for integrating more sophisticated algorithms and globally consistent optimization methods.

In conclusion, our exploration of visual odometry has been an insightful journey into the complex process of extracting motion information from visual data. While our current implementation demonstrates promising results, it is clear that there is room for enhancement and refinement. As we continue our work, we are excited about the potential for visual odometry technology to contribute to various applications, from autonomous navigation to virtual reality and beyond.

References

1. https://cmsc733.github.io/2022/proj/p3/
2. https://stackoverflow.com/questions/31737688/epipolar-geometry-pose-estimation-epipolar-lines-look-good-but-wrong-pose
3. https://engineering.purdue.edu/kak/computervision/ECE661Folder/Lecture21.pdf
4. Scaramuzza, D., & Fraundorfer, F. (2011). Visual Odometry [Tutorial]. IEEE Robotics & Automation Magazine, 18(4), 80-92.