Stereo Visual Odometry

-Shail Dalal

Table Of Contents

1. Personal Views

The field of computer vision, which originated from machine vision developed for commercial applications, has significantly transformed over the years. Initial algorithms were designed to analyze static images and extract data, laying the foundation for a critical technology in image and video processing. Today, computer vision enables machines to interpret and make decisions based on visual data, much like humans do, fundamentally altering the interaction between machines and the physical world.

The evolution of computer vision has been nothing short of transformative, expanding the capabilities of machines to perceive and understand their surroundings. Recent advancements in deep learning and neural networks have markedly improved the accuracy and functionality of computer vision systems. These systems now excel at complex tasks such as object detection, scene segmentation, and activity recognition with impressive precision. Additionally, integrating computer vision with technologies like the Internet of Things (IoT) and edge computing has expanded its range of applications. For instance, in smart cities, computer vision is utilized for traffic management and surveillance, enhancing urban safety and efficiency. In healthcare, it aids in diagnostic imaging, facilitating early disease detection with higher accuracy. The rapid advancements in this field continue to drive innovation, making computer vision an indispensable component of modern technology ecosystems.

However, these advancements come with their own set of challenges. The development of sophisticated computer vision systems often requires large, meticulously labeled datasets and substantial processing power, which are not always necessary for traditional vision algorithms. For example, Convolutional Neural Networks (CNNs) demand a diverse range of data points to make accurate predictions. This requirement, while demanding, also enables the extraction of intricate details and improves matching accuracy, capabilities that might be missed by traditional methods. Both traditional and modern methods have their flaws and necessitate extensive testing. The choice of method should be based on the specific requirements of the task at hand, with a preference for traditional methods when feasible.

My experience in the computer vision course at Northwestern was enlightening, providing a deep dive into the field. I had previously used many fundamental algorithms taught in this class through open-source packages, but the course provided a comprehensive understanding of the underlying logic and mathematics.

Algorithms like CCL helped me understand image segmentation and labeling meaning areas in an image. I also gained experience in image morphology which helped me understand image processing and how to obtain meaningful images from noisy images using erosion, dilation, closing, and opening. This also introduced me to how images can be treated as matrices and the basic idea of convolutions. I was also introduced to the concepts behind histogram equalization to smoothen the image which I used to detect skin color with some training and segment skin from a picture. My favorite thing to implement was canny edge detection. This was probably the most challenging assignment as it had a lot of moving parts like Gaussian filtering, calculating gradient, suppressing non-maxima, and then finding a threshold value to find the meaningful edges. Motion analysis was another one of that topics where we used a template to track the motion of the person moving around. This was particularly interesting as the implementation was left up to us and the template update was interesting to implement.

Implementing the mathematical concepts in assignments was particularly rewarding, as it solidified my understanding and ability to explain the functionality and rationale behind the code and results. The course structure was well-designed, with the first half focusing on basic algorithms and implementations, and the second half delving into more advanced topics. This progression offered a holistic view, linking foundational knowledge with advanced applications, and provided valuable insights into the real-world implementation of these algorithms.

I plan on developing my skills in computer vision and its implementation in robotics. I enjoyed the second half of the lectures where I learned about 3D reconstruction, stereo vision as well as texture modeling, and ways to perform object detection and recognition. I would like to learn how to implement neural networks for CV applications. An introduction to some state-of-the-art algorithms like YOLO would have helped me connect a lot of things I learned with robotics applications. I feel this class was an amazing introduction to a lot of topics and Dr. Wu’s insights into these topics helped me understand topics in the detail I wanted to. I plan to build on the knowledge I have gained in this class and develop robust algorithms for robotics applications like object manipulation, exploration, and manufacturing implications.

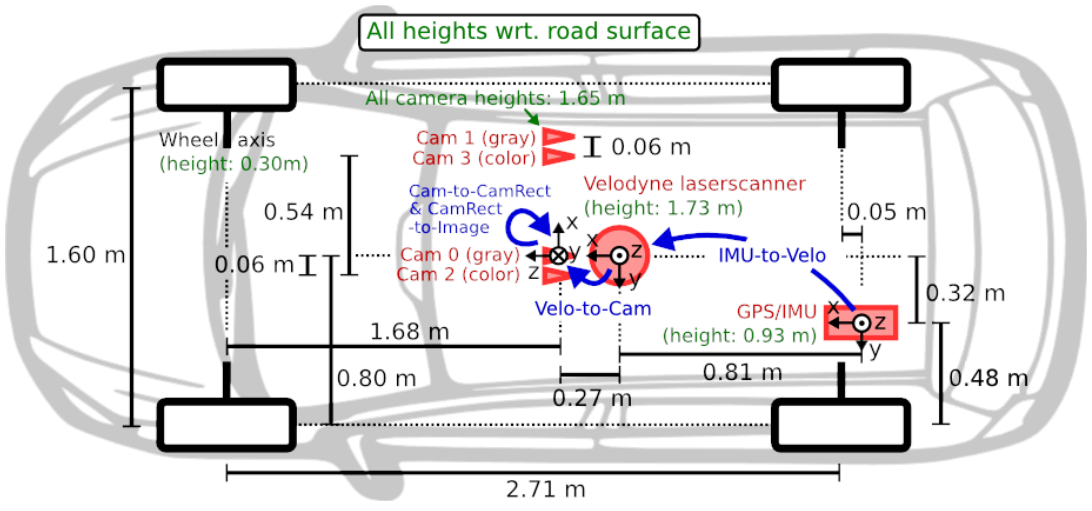
2. Project Description

I chose to work on visual odometry using stereo vision on the KITTI dataset. The goal of this project was to estimate the 3D motion of the car by analyzing changes in consecutive frames. My partner and I followed the same guidelines and created the code together.

Visual odometry is a subset of computer vision focused on determining an object's position and orientation using inputs from a stereo camera. The stereo camera setup allows us to recover depth which can be implemented to a variety of tasks. Some important applications are in robotics exploration and visual SLAM which can also be implemented in autonomous driving. Visual odometry can be used to reconstruct the 3D map and also help with localization within that map.

The methods used for visual odometry estimate the position using consecutive frames. This is done using matching features. This method is prone to errors as matching features is not 100% precise which leads to an accumulation of error throughout the path. There are ways to minimize the error but this is not in the scope of this project. This will be discussed in future works.

The KITTI dataset is a comprehensive dataset used extensively for benchmarking and research in computer vision tasks such as visual odometry, 3D object detection, and scene understanding. It is collected from a car equipped with various sensors, providing high-quality data for these applications. The dataset contains a ton of data including RGB and black and white images from the left and right camera, 3D point cloud data from a lidar, the sensor calibration data as well as the ground truth poses of the car. The sensors on the car were set up as in the picture below.



The code was developed in Python using primarily basic packages and OpenCV. While Python is typically not used for real-time applications due to its slower processing capabilities, it excels in visualization and offers a vast array of powerful libraries. This made Python an ideal choice for learning and initial implementation of visual odometry.

3. Project Pipeline

The pipeline is divided into eight major steps:

1. Data Processing
2. Compute Disparity

3. Compute Depth Map
4. Feature Extraction
5. Feature Matching
6. Filtering Matches
7. Estimate motion
8. Visual odometry – Putting it all together

1. Data Processing

The Kitti dataset consists of a lot of data that needs to be handled the right way. Processing and loading all the data at the same time requires a lot of RAM and processing power. For the first step of the pipeline, I created a DatasetHandler class which deals with all the data processing. Due to there being a lot of images, it is not efficient to store all the data in RAM until and unless you have the processing power. For this, I used a generator where the required image is loaded in as required.

This class also stores the calibration and the ground truth data which will be required for calculations and analysis. The lidar data is also considered with left and right images to set up all the data types. For this project, the images loaded are only grayscale.

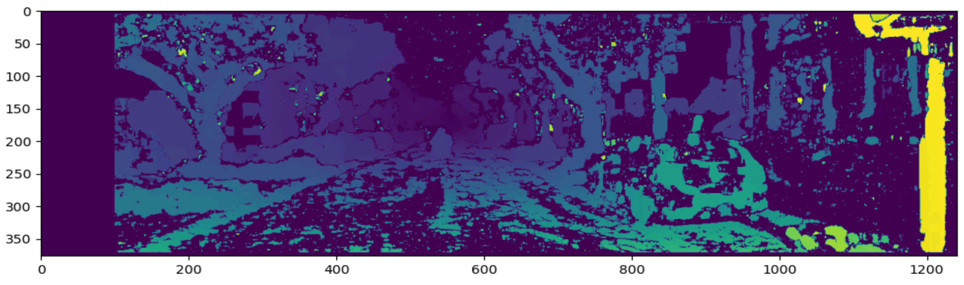
2. Computing Disparity

The disparity function computes a disparity map between images taken by the left and right camera of the stereo pair. The disparity is a calculation of how much a point in one image has shifted horizontally to align with the same point in the other image. The disparity map is generated with a lot of noise and discontinuous boundaries are not preserved well. There are two methods which were tested.

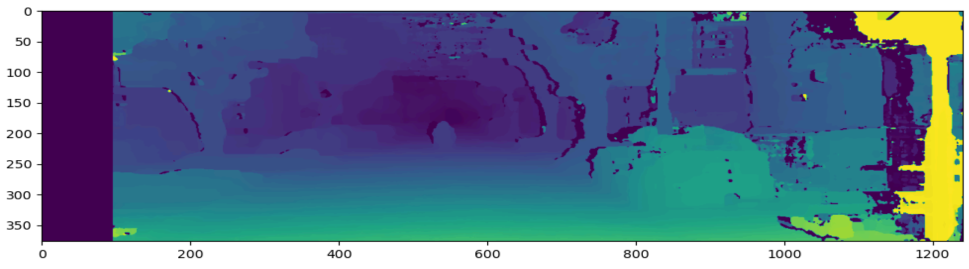
The first one was StereoBM (Block Matching) which performs a local block matching by comparing the left and right image pixels directly. It searches for the best match by comparing the intensity within a defined window.

The StereoSGBM (Semi Global Block Matching) was another method I employed. Here the matching includes global information of the picture. Due to this addition, this method considers larger regions that handle occlusions and discontinuities better.

While implementing the two methods, it was observed that StereoBm was a lot faster to compute the map than StereoSGBM. While StereoBm was faster, it is very discontinuous in comparison to StereoSGBM. Their effect on the final result will be discussed in the result analysis. I used OpenCV's implementation of StereoBM and StereoSGMB using cv2.StereoBM_create() and cv2.StereoSGHBM_create(). The parameters for these functions were selected based on OpenCV documentation as well as some trial and error.



Disparity Map using BM

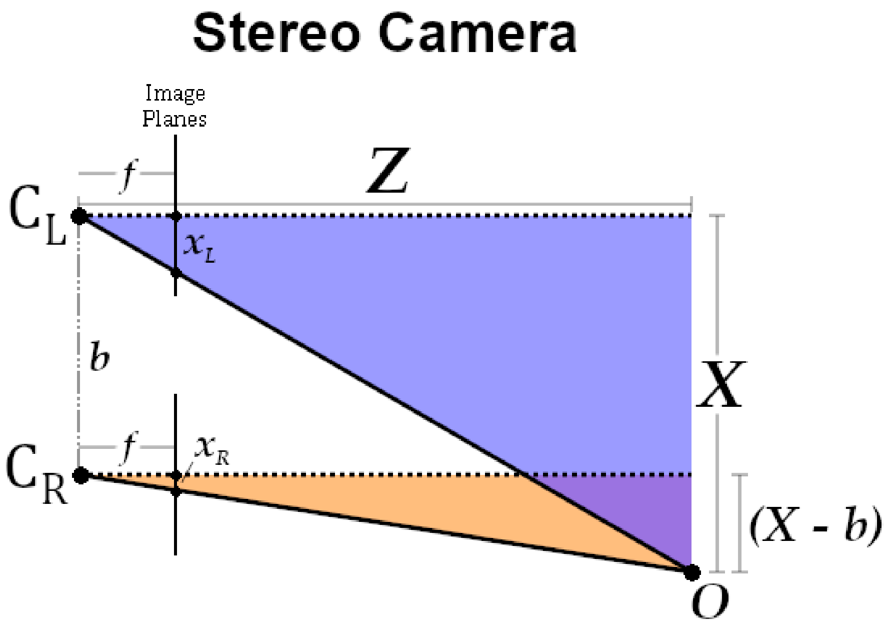


Disparity map using SGBM

3. Computing Depth Map

This function calculates the depth map from the disparity map created in the last function. It utilizes the geometry of the stereo camera placement and retrieves depth for each pixel using the camera's intrinsic information. An important feature of this function is the ability to work with both rectified and non-rectified matrices. Here the translations (p_x) of the rectified matrices are of the form [a, b, c, 1] and not [a, b, c, d]. This is an important step as all of the matrix math depends on this.

The depth is extracted using the following image as a reference. Here the Cl is the left camera and Cr is the right camera.



Using similar Triangles:

$$\frac{Z}{f} = \frac{X}{x_L} \text{ and } \frac{Z}{f} = \frac{X-b}{x_R}$$

$$Z\,x_L = fX \text{ and } Z\,x_R = f(X - b)$$

$$Z\,x_R = Z\,x_L - fb$$

$$fb = Z(x_L - x_R)$$

Now disparity is,

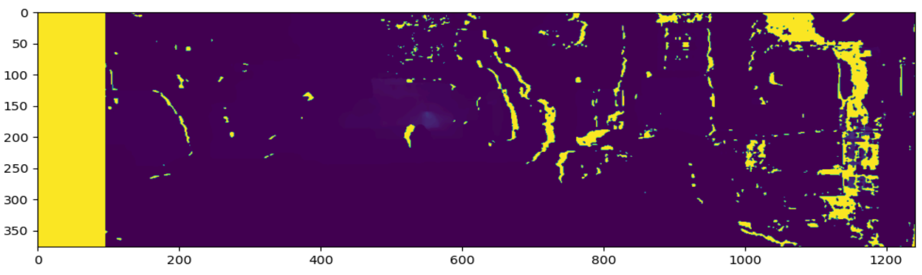
$$d = x_L - x_R$$

$$Z = \frac{fb}{d}$$

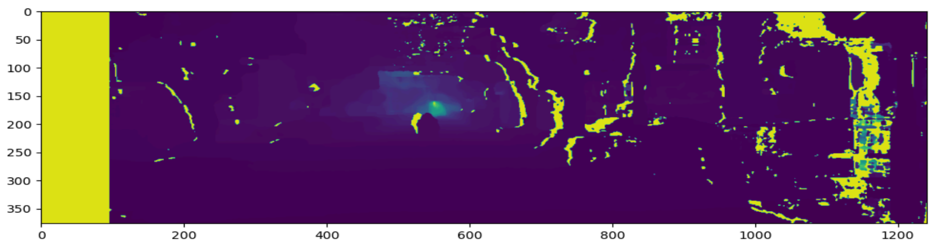
Here Z is the recovered depth, f is the focal length of the camera, b is the baseline and d is the disparity of each pixel. Using this a depth map is generated which is a calculation of how far each pixel is from the camera and gives us a recovered depth.

The intrinsic data of the camera is extracted from the calibration projection matrix provided in the dataset. Using the cv2.decomposeProjectionMatrix(), the returned k, r, t are the intrinsic properties of the camera which are used for different tasks,

Due to this being a left camera depth, there is a portion of the image that will have no depth data. To not waste computational power on that a mask is generated which will be used when all the information is put together.



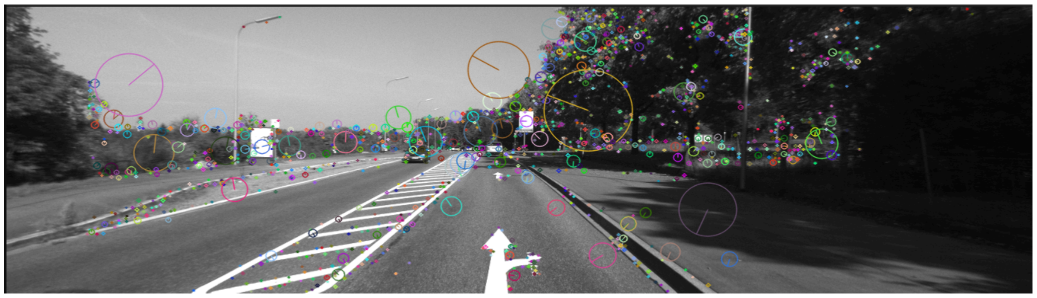
Depth map using BM (The green area is depth recovered)



Depth map using SGBM (The green area is depth recovered)

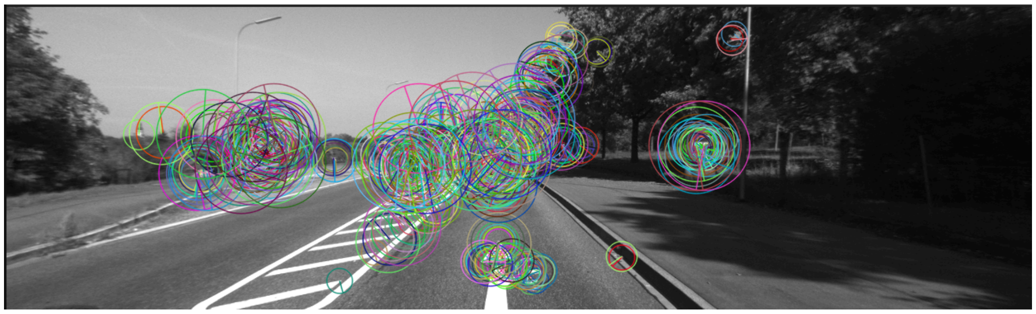
4. Feature Extraction

- Feature extraction is probably one of the most important if not the most important factor in visual odometry. The feature extraction algorithm is responsible for selecting the important features from the images which will then be matched to the next image. The methods used for feature extraction are SIFT (Scale Invariant Feature Transform) and ORB (Oriented FAST and Rotated BRIEF). Both of them have their upsides and downsides. OpenCV has implementations of both the methods which I have used here.
- SIFT - This method is known for its robustness to change in scale, rotation, and illumination which makes it very effective for this use case. SIFT works by first finding the key points in an image at different scales and orientations, and then generating descriptors based on local image gradients around each key point. This leads to the descriptor having information about the keypoint surroundings which allows accurate matching. SIFT is highly effective but takes time to compute. This method can be used well in an offline setting where accuracy is preferred over computing power.



Feature Detection using SIFT

- ORB - This method finds key points by detecting corner-like features in images and then generates a descriptor based on a binary test on intensity comparison between pixel pairs. Image rotations are also handled by incorporating orientation estimation. While ORB may not match SIFT in terms of robustness, it excels in real-time applications and scenarios where computational resources are limited. Its ability to provide reasonably accurate matching with significantly faster computation times makes ORB particularly well-suited for online settings where speed is critical.



Feature Detection using ORB

5. **Feature Matching**

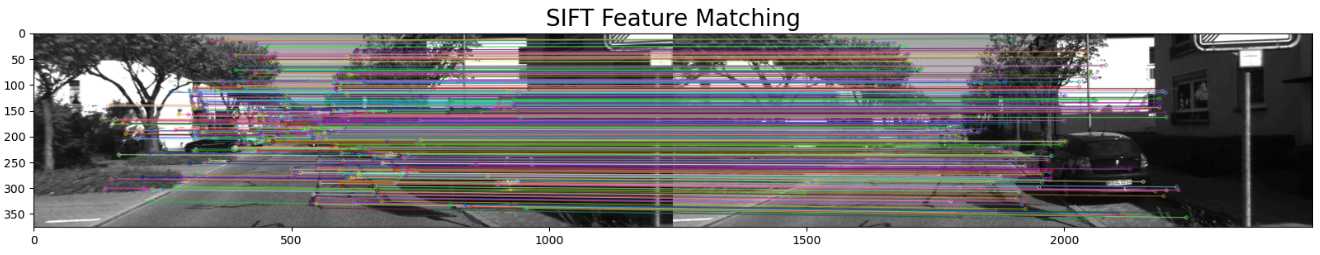
After extracting the features, the next step is to match them with the features of the next image in the sequence. This is done using the descriptors extracted from both images. The matching method used here is a brute force method. For this method, it takes the first set of descriptors and compares it against all descriptors in the second set. This method works differently for both feature detection methods (orb and sift. I also used k = 2 which specifies the number of nearest neighbors to find for each descriptor.

- SIFT - SIFT descriptors are 128-dimensional vectors that describe the local appearance of key points in an image. Each vector component is a real number, which makes SIFT descriptors suitable for floating-point arithmetic.
    - Euclidean Distance (L2 norm): This measures the straight-line distance between two points in the 128-dimensional space. The Euclidean distance is ideal for SIFT because it effectively captures the differences in the high-dimensional space where SIFT descriptors reside. It accounts for

variations in both the magnitude and direction of the descriptor vectors, which is crucial for accurate matching.

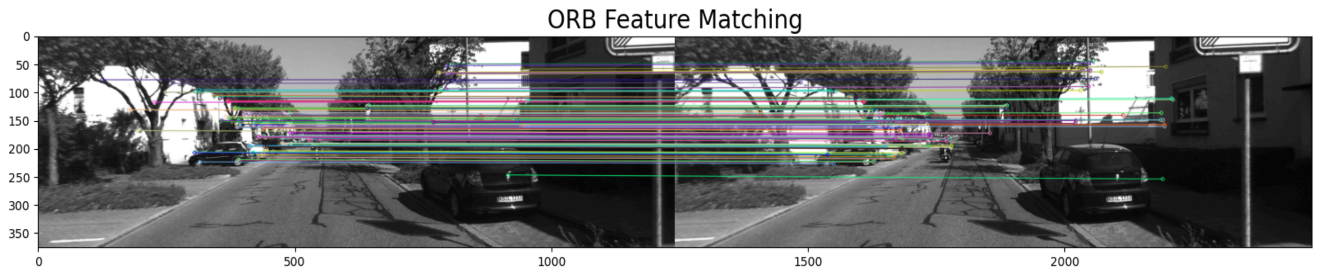
- For two vectors d1 and d2, Euclidean distance is:

$$distance = \sqrt{\sum_{i=1}^{128} (d1_i - d2_i)^2}$$



- ORB - ORB descriptors are binary strings, typically 256 bits long. Each bit in the descriptor represents the result of a simple intensity comparison test between pairs of image patches around the key point. This binary representation is highly efficient for both storage and computation.
  - Hamming Distance: This measures the number of bit positions at which the corresponding bits are different. The Hamming distance is ideal for ORB descriptors because it quickly quantifies the difference between binary strings. This is computationally efficient and leverages the binary nature of ORB descriptors.
  - For two binary strings b1 and b2, the Hamming distance is:

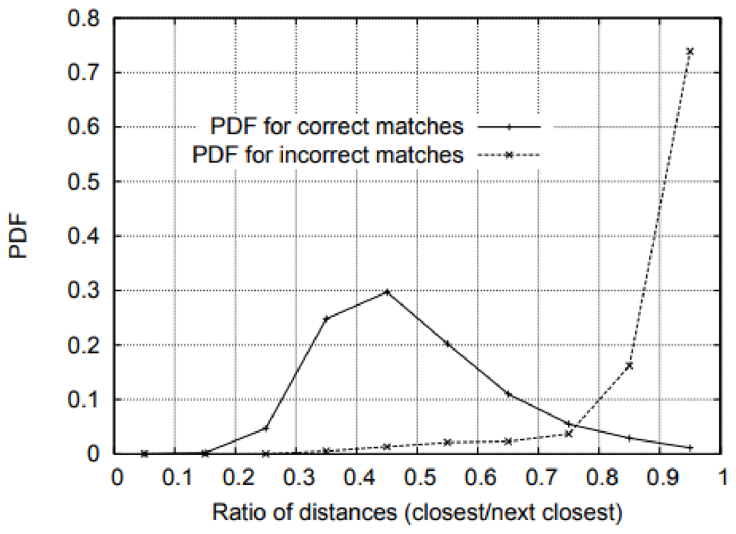
$$distance = \sum_{i=1}^{256} (b1_i \oplus b2_i)$$, where $\oplus$ denotes XOR operation



6. Filtering Matches

To get the best results, we need to filter out the matches used for tracking. We do this by finding the two closest matches in the second image for each feature in the first image. Then, we compare the distances to see if the closest match is much better than the second-closest match. If the closest match is not a lot better, it means the feature is not very unique and is too similar to other features. These non-unique features should be left out to make sure the tracking is accurate and reliable.

The graph below is a Probability distribution for the correct and incorrect matches when a database of 40,000 key points was used. From the regular line, we can see that a good spot for the correct match ratio is between 0.3 and 0.5. This might be the ideal scenario and this worked great for SIFT but for ORB matches, I had to use a higher value of about 0.7 to get enough matches to get an estimated pose.



7. Estimate Motion

The final step of this process is to use the filtered matches and key points from two images, the camera intrinsic matrix, and also the depth map to estimate the motion of the camera. Camera intrinsic parameters are extracted from the k matrix which is cx, cy, fx, and fy. Using these the 3D information is converted to 2D motion estimation

The main functionality works in a way where we loop over all the key points from the first image. In the loop, the depth for each location is extracted and if the depth is greater than a max depth parameter, we delete it from the keypoint list. This helps us keep everything in check as sometimes points with no information can be considered as key points. After this for each valid key point, we calculate the 3D point value (x, y, z) using the equations below:

$$x = \frac{(u - cx) * depth}{f_x}, y = \frac{(v - cy) * depth}{f_y}, z = depth$$

For every key point, these values are accumulated as a matrix. Using OpenCV's solvePnPRansac function, a rotation vector and a translation vector are obtained. This rotation vector is converted to a rotation matrix using Rodrigues's formula. This translation and rotation is an estimate between two frames.

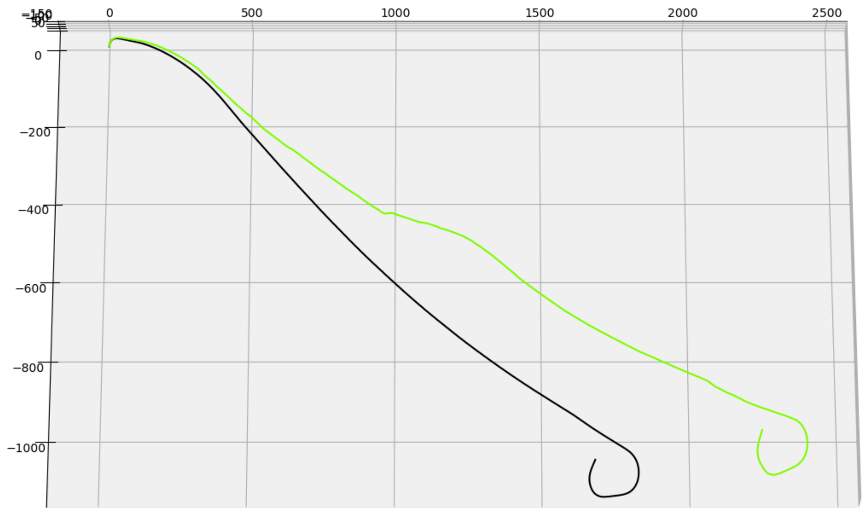
8. Visual odometry – Putting it all together

After getting the translation and rotation between two frames, everything needs to be put together to achieve an entire path. This is done by constructing a T_total Matrix. This matrix is an accumulation of all the transformation matrices along the path. To do this, at the beginning, the transformation is an identity matrix. For each iteration, the rotation and translation vectors are converted into a transformation matrix. This matrix is T_nextframe_lastframe, whereas the T_total can also be considered as

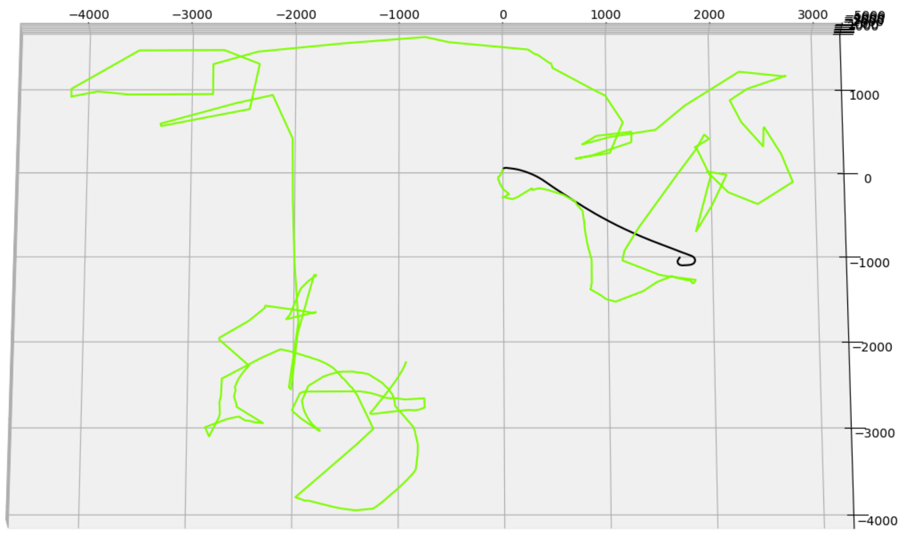
T_world_lastframe. To get T_world_nextframe, using properties of transformation matrices, an inverse of T_nextframe_lastframe is multiplied by T_world_lastframe. From this accumulation, the x, y, and z of the camera are extracted and accumulated as a trajectory to be plotted and used for evaluation.

3. Result Analysis

The most critical aspect was identifying a set of hyperparameters that worked well together. These included choosing between StereoBM and StereoSGBM for disparity maps and deciding between ORB or SIFT, along with the ratio for filtering. Each of these choices had different effects on the output, with notable differences in error and runtime. The reasons for these differences were discussed in detail in the pipeline. The best results, in terms of error, were observed with SIFT. In contrast, the ORB method resulted in a high error margin. The following discussion focuses on the SIFT analysis with a 0.3 distance ratio with SGBM disparity analysis, and ORB analysis with a 0.8 distance ratio with SGBM disparity analysis.



Result Using SiFT (SGBM and 0.3 distance ratio)



Result Using ORB (SGBM and 0.8 distance ratio)

Three metrics used for position error detection are Mean Squared Error (MSE), Root Mean Square Error (RMSE), and Mean Absolute Error (MAE). These metrics capture the overall difference between the provided ground truth data and the estimated path from SIFT and ORB.

Another metric for analysis is the time taken to complete the entire segment, as well as the time per frame. This is crucial because the time required for each frame is integral when the system is set up in real time. The results below, from the KittiDataset sequence 01, indicate that the overall error for SIFT is only about 1000 meters, which is very small. In terms of time, ORB for sequence 01 is about twice as fast as SIFT, making it an optimal choice for an online option on a robot. However, SIFT tends to lose some frames, which leads to a large error metric.

	SIFT (Dataset 01)	ORB (Dataset 01)
MAE	911.82	2891.26

RMSE	1088.91	1185723.96
MSE	3125.60	9769361.80
Time	3m 0.32s	1m 37.23s
Time/Frame	0.166898s	0.08831s

4. Improvements and Future work

This project has successfully demonstrated visual odometry using stereo vision and the KITTI dataset. However, several improvements can enhance the results. Key areas include refining feature matching, minimizing errors, optimizing for real-time performance, and integrating advanced techniques.

Improving feature matching with advanced detectors like AKAZE or SuperPoint and better outlier rejection methods can boost robustness. Integrating bundle adjustment and loop closure detection will minimize trajectory errors. Optimizing the code for real-time performance using languages like C++ and leveraging hardware acceleration can enhance speed. Incorporating deep learning models and end-to-end learning approaches for feature extraction and matching will improve accuracy and efficiency.

Fusing LiDAR data with stereo odometry using a Kalman filter and optimizing ORB parameters for better performance are critical steps. Employing SLAM techniques will further improve trajectory estimates and reduce drift. Tailoring the system for applications like autonomous driving and robotics by integrating additional sensor modalities will broaden its practical use.

I believe focusing on sensor fusion will lead to great results. Something I would like to learn more about is the implementation of deep learning models like SuperPoint and how they compare to a steady algorithm like SIFT.

References:

[1] D. Scaramuzza and F. Fraundorfer, "Visual Odometry [Tutorial]," in IEEE Robotics & Automation Magazine, vol. 18, no. 4, pp. 80-92, Dec. 2011, doi: 10.1109/MRA.2011.943233.

[2] Lowe, D. G. (2004). Distinctive Image Features from Scale-Invariant Keypoints. International Journal of Computer Vision, 60(2), 91-110.

[3] Geiger, A., Lenz, P., & Urtasun, R. (2012). Are We Ready for Autonomous Driving? The KITTI Vision Benchmark Suite. Conference on Computer Vision and Pattern Recognition (CVPR), 3354-3361.

[4] Cibik, Nate. “KITTI Odometry with OpenCV Python” Youtube, uploaded by Nate Cibik, 21 Feb. 2021, https://www.youtube.com/watch?v=SXW0CplaTTQ.