TEL AVIV UNIVERSITY אוניברסיטת תל-אביב

THE IBY AND ALADAR FLEISCHMAN FACULTY OF ENGINEERING

הפקולטה להנדסה על שם איבי ואלדר פליישמן

Project 3

Mapping and perception for an autonomous robot/ 0510-7591

By Avi Epstein

Tel Aviv University

December 2021

Abstract

In this project we will be implementing and analyzing the following algorithms:

- Particle filter
- Visual Odometry (mono)

In the first section we will implement the Particle filter algorithm, we will load a pre defined data set containing landmarks and odamatry data. using this we will plot the ground truth trajectory starting at point (x_0, y_0, θ_0) based on the following motion model:

$$\begin{pmatrix} x_t \\ y_t \\ \theta_t \end{pmatrix} = \begin{pmatrix} x_{t-1} \\ y_{t-1} \\ \theta_{t-1} \end{pmatrix} + \begin{pmatrix} \delta_{trans} \cos(\theta_{t-1} + \delta_{rot1}) \\ \delta_{trans} \sin(\theta_{t-1} + \delta_{rot1}) \\ \delta_{rot1} + \delta_{rot2} \end{pmatrix}$$

We then added guassian noise to the motion model and to ground truth sensor measurements, we then initialized N particle's with an initial Gaussian distribution around (x_0, y_0, θ_0) generating our hypothesis pose for each particle. In addition, we give each particle an initial weight. We then run our motion model on all particles (with a small amount of Gaussian noise added before) and apply sensor measurement (sensor measurement is assumed as a spin LiDAR 2D sensor (1 layer, 360 degrees) which in each iteration calculates the range and azimuth (r, φ) only from the closet landmarks).
We then apply sensor correction by recalculating the weight of each particle by using "normal Mahalanobis distance" and hence measuring how far is the hypothesis particle measurement from the ground truth and giving it a new weight based on this distance. We then resample the particles using low variance resampling from (Thrun, Burgard, and Fox's "Probabilistic Robotics") making our particle's with higher weight be resampled with high probability. We repeat this process of noised motion, measurement, weight correction and resampling until convergence to the true trajectory. We then calculate the MSE from the 50th frame using the pose of the particle with the highest weight and see how it performs. In addition, will then find the minimal amount of particle's that still give us good performance based on the MSE criteria.

In the second section we will implement a simple monocular Visual odometry algorithm.
We will extract from each frame its features using sift/orb/klt, each one of these feature extractions work a little different and will be explained in this report we then match the found features in both frame using brute force matching or FLANN, to get corresponding points between both frames. We then compute the essential matrix and remove outliers based on RANSAC, and decompose the essential matrix to recover the translation and rotation between the 2 camera positions the pose recovery verifies possible pose hypotheses by doing cheirality check. (which means that the triangulated 3D points should have positive depth)
Because we are using a mono camera it is hard to estimate the depth between frames and hence the translation is known up to scale. To correct this as best as we can we refine the translation vale by scaling each translation by a factor ratio between a known measurement such as IMU info or lidar (in our case we used the ground truth trajectory).

We then concatenate all these translations to get the full trajectory estimated from our mono camera rotation and translation we will see how this scale improves the result and why we receive drift in our trajectory.

Contents

List of Figures

List of Movies

Attached movies:

- VO animation sift FLANN with key points.mp4 (best)
- VO animation sift FLANN matched frames.mp4
- VO animation sift BF with key points.mp4
- VO animation sift BF matched frames.mp4
- VO animation orb FLANN with key points sized and oriented
- VO animation orb FLANN with key points.mp4
- VO animation orb BF matched frames.mp4

Note: the movies are not to the end of the trajectory as the run time took to long. running this of collab would work better and run for a longer time

List of Tables

Solutions:

Partical Filter :

Particle filters sample a distribution with a collection of particles, generate a prediction of the distribution by forward predicting each particle using motion model and then compare and update that prediction using a measurement and its uncertainty characteristics.

a) In the first section we loaded the attached data file (Odometry and landmarks data) And plot the ground truth trajectory starting from (x_0, y_0, θ_0) based on the following motion model:

$$\begin{pmatrix} x_t \\ y_t \\ \theta_t \end{pmatrix} = \begin{pmatrix} x_{t-1} \\ y_{t-1} \\ \theta_{t-1} \end{pmatrix} + \begin{pmatrix} \delta_{trans}\cos(\theta_{t-1} + \delta_{rot1}) \\ \delta_{trans}\sin(\theta_{t-1} + \delta_{rot1}) \\ \delta_{rot1} + \delta_{rot2} \end{pmatrix}$$

Figure 1: Odometry motion model

The ground truth we calculated and landmarks are shown in the next figure:

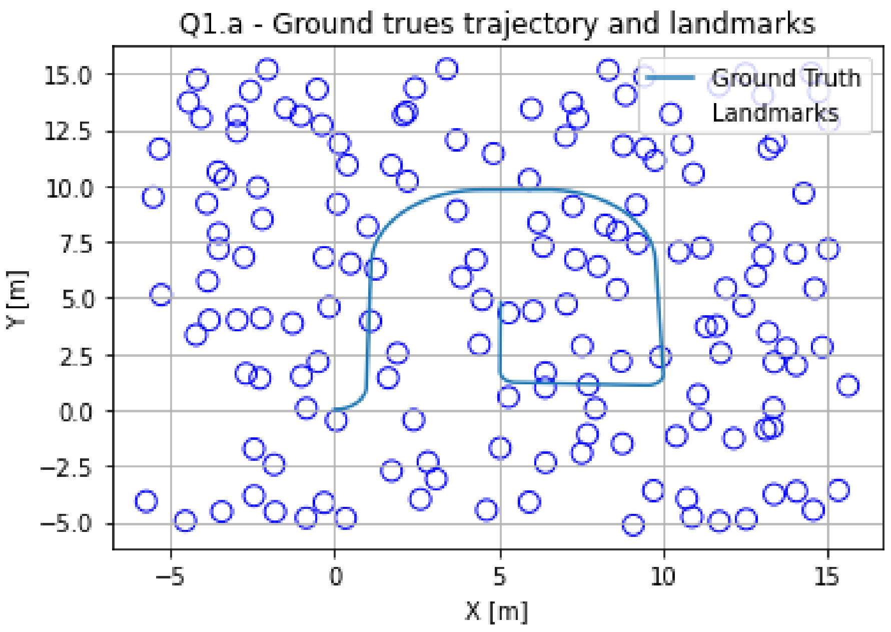


Figure 2: ground truth trajectory and landmarks

We can see that we have an arched path ending at the center of the area the landmarks are densely populated throughout the map.

b) In this section we will add Gaussian noise to the motion model with standered devaition of:

$$\sigma_{rot1} = 0.01\ , \qquad \sigma_{trans} = 0.1\ , \qquad \sigma_{rot2} = 0.01$$

and normalize the angle if needed. This noise allows for each particle to move in a slightly different way then the others hence when motion model is applied making the particle's cover more possible situations that the robot could have been in.

c) We then added Gaussian noise to our sensor. In each ground truth step we calculated the closest land mark giving us range and azimuth (r, φ) we then added Gaussian noise to these sensor measurements with standard deviation of:

$$\sigma_r = 1\ , \sigma_\varphi = 0.1$$

(this is done as data preparation to simulate spin LiDAR 2D sensor (1 layer, 360 degrees) which in each iteration calculates the range and azimuth (r, φ) only from the closet landmarks with noise Q)

d) In this section we initialized a particle array with N = 1000 particles each containing 6 elements:

$$pose\ = (x, y, \theta)$$
$$weight\ =\ w$$
$$sensor\ measurement\ z =\ (r, \varphi)$$

- The pose of each particle was initialized to:
  $$x_0 \sim N(0,2),\ y_0 \sim N(0,2),\ \theta_0 \sim N(0.1,0.1)$$
  Such that all particle's got a slightly different pose around the assumed starting point.
- The weight was initialized to $w = \frac{1}{N}\ , s.t\ N = number\ of\ particles$
- Sensor measurement was calculated after first motion step for each particle.

We then apply a noisy motion model on each particle, making every particle move a little different then the others with the added noise as in section b has standard deviation of:

$$\sigma_{rot1} = 0.01\ , \qquad \sigma_{trans} = 0.1\ , \qquad \sigma_{rot2} = 0.01$$

This allows for the model to cover more location possibilities and be more "flexible" all particle's move slightly differently such that even if we missed the ground truth location our particles can move towards it.
We then apply our observation and update all particle Sensor measurement giving us $z =\ (r, \varphi)$ for each particle.
We then give all the particle's new weights based on their normal Mahalanobis distance from the ground truth measurement calculated in section c.
their normal Mahalanobis distance is defined as:
A measure of the distance between a point P and a distribution D, meaning It is a multi-dimensional generalization of the idea of measuring how many standard deviations away P is from the mean of D.
For a normal distribution in any number of dimensions, the probability density of an observation $\vec{x}$ is uniquely determined by the Mahalanobis distance d:

$$\Pr[\vec{x}]\,d\vec{x} = \frac{1}{\sqrt{\det(2\pi\mathbf{S})}}\exp\left(-\frac{(\vec{x}-\vec{\mu})^{\mathsf{T}}\mathbf{S}^{-1}(\vec{x}-\vec{\mu})}{2}\right)d\vec{x} = \frac{1}{\sqrt{\det(2\pi\mathbf{S})}}\exp(-d^2/2)\,d\vec{x}.$$

Where in our case $S = Q = \begin{pmatrix}\sigma_r & 0\\ 0 & \sigma_\varphi\end{pmatrix} = \begin{pmatrix}1 & 0\\ 0 & 0.1\end{pmatrix}$ is the nonsingular covariance matrix.

we then normalize the weights by dividing by the total weight of all particales making each weight to be between 0 and 1.

We then resample the particle’s using low variance resampling (derived by Thrun, Burgard, and Fox's "Probabilistic Robotics") this technique is a stochastic universal sampling with low variance and linear time complexity the algorithm is presented in the next figure:

```
Low_variance_resampling($\mathcal{X}_t, \mathcal{W}_t$):
1:    $\bar{\mathcal{X}}_t = \emptyset$
2:    $r = \text{rand}(0; J^{-1})$
3:    $c = w_t^{[1]}$
4:    $i = 1$
5:    for $j = 1$ to $J$ do
6:        $U = r + (j-1)J^{-1}$
7:        while $U > c$
8:            $i = i + 1$
9:            $c = c + w_t^{[i]}$
10:       endwhile
11:       add $x_t^{[i]}$ to $\bar{\mathcal{X}}_t$
12:   endfor
13:   return $\bar{\mathcal{X}}_t$
```

Figure 3: Low variance resampling algorithm

In general, the principal of resampling is to replace unlikely samples by more likely ones. Low variance resampling insures this as particle’s with hi state probability are sampled more often then lower ones we will see this in the upcoming sections.
We then repeat this full process.
At the end of the processes we choose the particle that had the highest weight in each iteration as the trajectory estimate.
To sum up our steps are:

- Initialize particle’s distributed in the robots area
- For all time steps:
  1. Perform propagation of particles according to motion model
  2. Weighting of particle’s according to likelihood of their observation
  3. Resampling
  4. Return to step 1
- Choose the best particle of each time step

e) Analysis results:
the best performance was achieved with 500 particles’ hence we will first analyze this implementation of the particle filter:

1. in the attached video we can see that the particle's are initially spread out and after a few iterations lock on to the true trajectory and stay on it until the end.

2. as was explained we can see in the next figure the distribution of the particle's after first resampling:

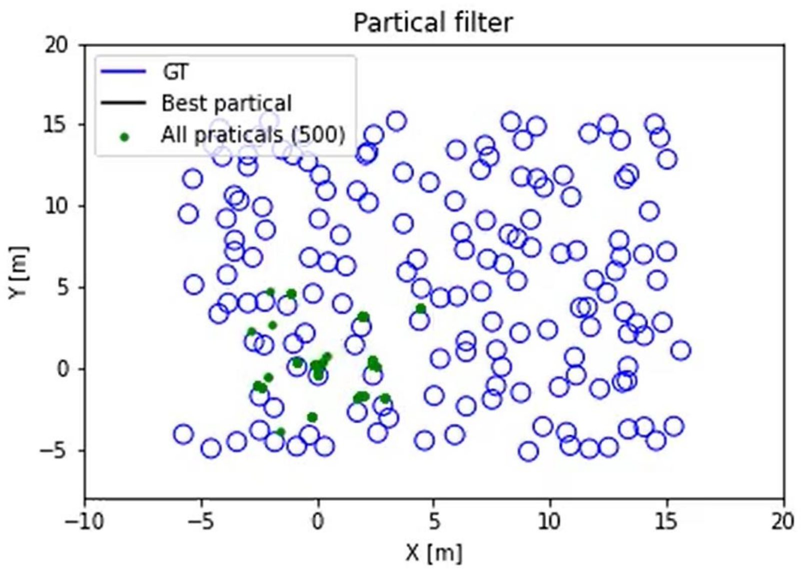


Figure 4: distribution of the particle's after first resampling

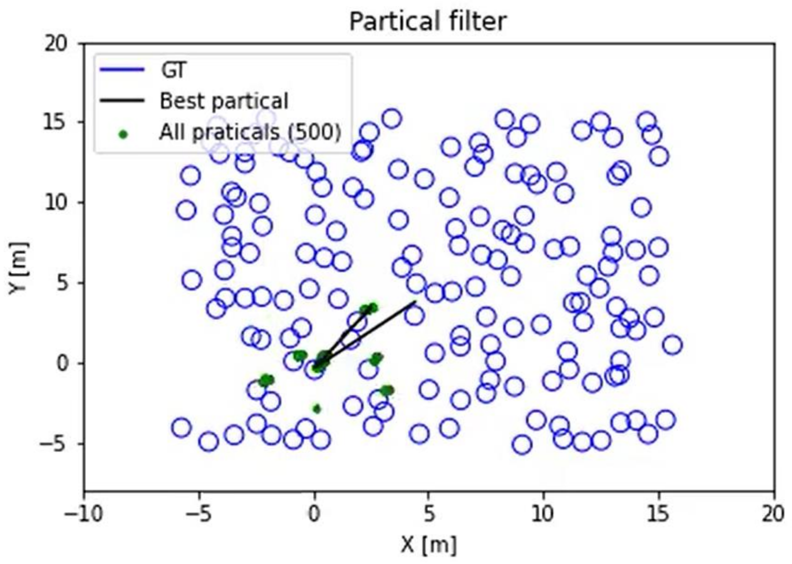


Figure 5: distribution of the particle's after resampling 3rd iteration

As can be seen the particle's are slowly converging to the true location of the robot . particles that have similar measurement to the true measurement survive. However we already see the problem in the way of our observation, because we only take the measurement from the closest land mark there are a lot of states the particle's could be in that would still be similar to the ground truth location but are not the true location we will conclude later that using more landmark observations will make the ground truth

pose more unique and hence the estimation more exact, ie less particales will have hi probability.
3. The full estimated trajectory is presented in the next figure:

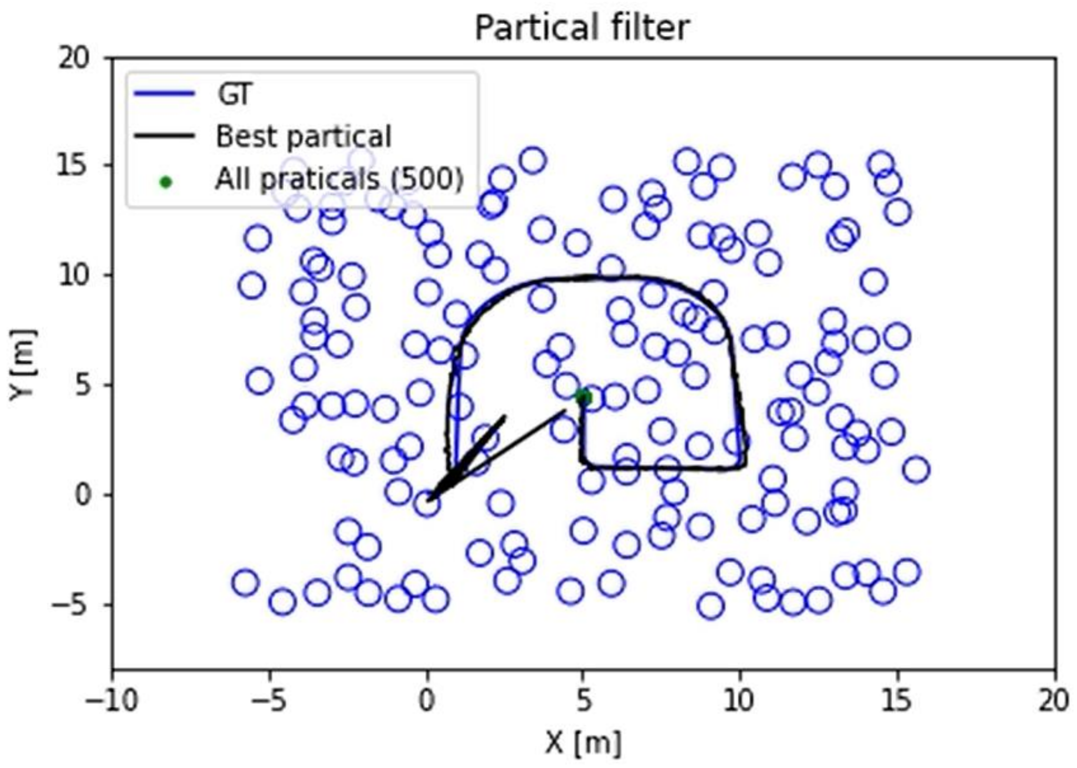


Figure 6: The full estimated trajectory Particle filter

Here we can fully see that after a number of iterations the particle's lock onto the true trajectory and we get a good trajectory estimate. In the video("Partical filter animation (number of particles_ 500)") we can se the noised notion propagation and resampling of the highest probability particle's.

4. Here will calculate MSE (Mean Square error) of frames [50:end] , Our goal will be to minimize the RMSE which is defined as: (i >50)

$$RMSE \triangleq \sqrt{\frac{1}{N}\sum_{i=100}^{N}[e_x^2(i)+e_y^2(i)]}$$

$$e_x(i) \triangleq x_{GT}(i) - x_{Estimate}(i)$$
$$e_y(i) \triangleq y_{GT}(i) - y_{Estimate}(i)$$

$$maxE \triangleq max\{|e_x(i)| + |e_y(i)|\} \quad ,$$

$$100 \leq i \leq N$$

N is last sample.

The results we got for 500 particles' is:

RMSE: 0.2104, maxE: 1.1104

Which is pretty exact.

5. We now find the minimal set of particles (1000, 500,100,50,30,20,10,5,1) which keep still good performance based on RMSE as defined above, attached are videos of the partical filter performance using varying number of particles the results are:

Number of particles	1000	500	100	50	30	20	10	5	1
RMSE	3.673	0.2104	6.696	9.566	3.414	8.877	4.504	3.271	2.843
maxE	8.6899	1.1104	12.308	16.05	8.201	15.885	9.309	7.592	3.8811

Table 1: RMSE and maxE of particle filter with different number of particles

As was pointed out before we noticed that the initial distribution of the particles had a large impact on the quality of the results on some runs we were able to get very good results even using 30 particles(the motion noise is reduced in these cases). In the animations we can see that a lot of the filters start out well and for some random noised motion they all lose the correct state and converge to a wrong trajectory or get stock in place. The reason why we don't get very good results every time is that the landmarks themselves are very densely distributed and our measurement is only considering one land mark making a lot of wrongly placed particles have a high probability and hence the filter doesn't converge to the true trajectory, again lowering the variance of the motion also gave us better results in this case because the trajectory was more truly defined when using enough particle's. making the motion noise larger gave better results when using a smaller number of particles because the randomness of the motion allowed for a higher state coverage. We conclude that the particle filter can be used even with a very small number of particles given it is parametrized correctly.
Another way to improve the model is to add a small number of random particles at each time step, this will help the filter not converge on an incorrect trajectory guaranteeing that no area in the map is totally deprived of particles. (helps also in robot kidnaping)

The code:
Main -> for every time step :
Calc z_t -> pf.apply -> self.motionModel(Ut), self.Observation(), self.resampleParticles(),pf.bestKParticles

Visual Odomatry :

The goal of this section is to implement a simple, monocular, visual odometry (VO) pipeline with the most essential features:

- Initialization of 2D landmarks,
- Extract keypoint tracking between two frames,
- pose estimation using established 2D↔ 2D correspondences,
- extract R and T from the essential matrix
- Refine translation value by r ratio

We will be using kitty visual odometry data set, The input (grayscale) Selecting scenario 1 to be the ground truth trajectory. We will then extract the camera calibration intrinsic parameters.
In our case we are performing mono VO so we use only camera 0. And will Estimate the motion by only two pairs of images (t, t+1)
We will then calculate the scale factor as the ratio magnitude of GT (L2 norm) and estimated trajectory (magnitude GT is simulated odometer measurements).

a) Here we will implement monocular visual odometry! (2D→2D). we will calibrate the algorithm for maximum of ~15 meters (Euclidian distance) from the GT (in the first 500 frames). The algorithm pipeline:

   - ground truth odometry is extracted and displayed in the following figure:
   - for every frame in total number of frames:
     - Step 1: extract 2 frames, the current frame and next frame
     - Step 2: extract key points and discripturs of both frames.(performed with sift/orb/”Shi Tomsis good corners to track)
     - Step 3: match features between both frames (BF/FLANN matchers or optical flow if using good corners to track).
     - Step 4: get essential matrix using the matched points and remove outliers to the over defined equation using RANSAC
     - Step 5: Recover pose by decomposing the essential matrix to rotation matrix and translation vector the pose recovery verifies possible pose hypotheses by doing cheirality check. (which means that the triangulated 3D points should have positive depth)
     - Step 6: transform rotation matrix and translation vector to homogeneous transformation

   - perform scale correction using the ground truth as our data input
   (in general, it could be imu data or lidar) this is performed because in mono Visual Odomatry the translation is only known up to scale as there is no direct depth estimation.

- We then concatenate homogenies transformation from starting point till end to get full trajectory

We will now dive into the detail of each technique of key point identification descriptors of the key points and matchers of these descriptors (readers that are familiar with these techniques can skip this part):

<u>**Key points:**</u>

- **Classic Harris corner detector:** Harris corner detector provides good repeatability under changing illumination and rotation. the harris corner detector finds the difference in intensity for a displacement of (u,v) in all directions. This is expressed as below:

$$E(u,v) = \sum_{x,y} \underbrace{w(x,y)}_{\text{window function}} \, [\underbrace{I(x+u,y+v)}_{\text{shifted intensity}} - \underbrace{I(x,y)}_{\text{intensity}}]^2$$

  The window function is either a rectangular window or a Gaussian window which gives weights to pixels underneath. We then maximize this function for corner detection.
  After Tyler expansion we are able to get the approximation :

$$E(u,v) \approx \begin{bmatrix} u & v \end{bmatrix} M \begin{bmatrix} u \\ v \end{bmatrix}$$

  Where:

$$M = \sum_{x,y} w(x,y) \begin{bmatrix} I_x I_x & I_x I_y \\ I_x I_y & I_y I_y \end{bmatrix}$$

  I_x and I_y are image derivatives in x and y directions respectively.
  The result is then scored to determine if a window contains a corner or not, the score by Harris:

$$R = \det(M) - k(\operatorname{trace}(M))^2$$

  Where:

  det(M)=λ1λ2, trace(M)=λ1+λ2, λ1 and λ2 are the eigenvalues of M

  So the magnitudes of these eigenvalues decide whether a region is a corner, an edge, or flat.

  - When |R| is small, which happens when λ1 and λ2 are small, the region is flat.
  - When R<0, which happens when λ1>>λ2 or vice versa, the region is edge.

- When R is large, which happens when λ1 and λ2 are large and λ1~λ2, the region is a corner

Note: Harris corner it is not scale invariant.

- **Good Features to Track (using Shi Tomasi):** shows better results compared to Harris Corner Detector.
Shi-Tomasi proposed a scoring based on:

$$R = \min(\lambda_1, \lambda_2)$$

If it is a greater than a threshold value, it is considered as a corner.
Good features to track finds N strongest corners in the image by Shi-Tomasi method (can be used with Harris corner detector as well) we specify number of corners we want to find, specify the quality level(between 0-1) and provide the minimum Euclidean distance between corners detected With all this information, the function finds corners in the image. All corners below quality level are rejected. Then it sorts the remaining corners based on quality in the descending order. Then function takes first strongest corner, throws away all the nearby corners in the range of minimum distance and returns N strongest corners.

- **SIFT (Scale-Invariant Feature Transform):**
Sift uses scale-space filtering. In it, Laplacian of Gaussian is found for the image with various σ values. LoG acts as a blob detector which detects blobs in various sizes due to change in σ. In short, σ acts as a scaling parameter. To save costs SIFT algorithm uses Difference of Gaussians which is an approximation of LoG. Difference of Gaussian is obtained as the difference of Gaussian blurring of an image with two different σ, let it be σ and kσ. This process is done for different octaves of the image in Gaussian Pyramid. It is represented in below figure:

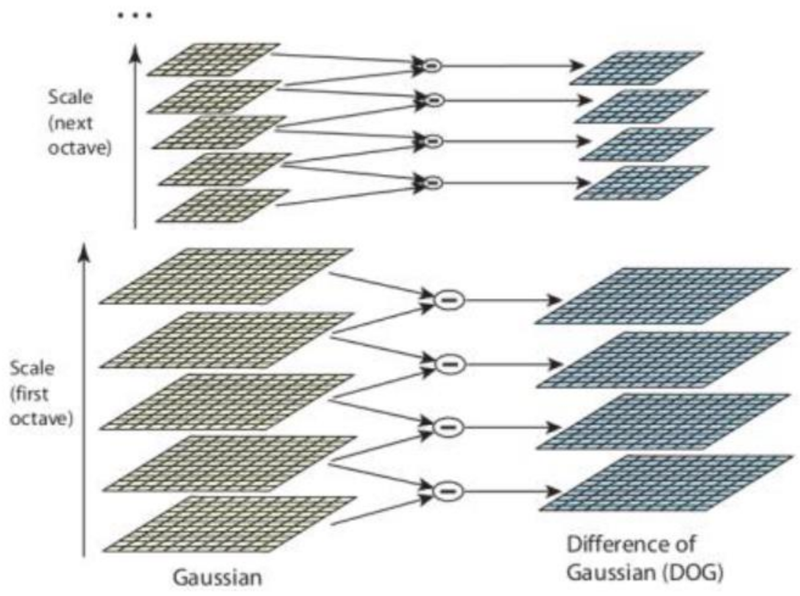


Figure 7: Gaussian Pyramid

Once this DoG are found, images are searched for local extrema over scale and space. If it is a local extrema, it is a potential keypoint. As can be seen in the next figure:

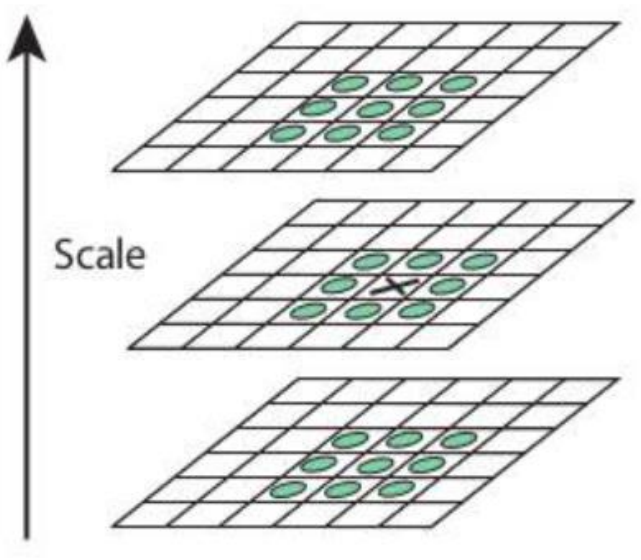


Figure 8: local extrema over scale and space

Once potential keypoints locations are found, they have to be refined to get more accurate results. They used Taylor series expansion of scale space to get more accurate location of extrema, and if the intensity at this extrema is less than a threshold it is rejected. called contrastThreshold. DoG has higher response for edges, so edges also need to be removed. For this, a concept similar to Harris corner detector is used. If this ratio is greater than a threshold, that keypoint is discarded. So it eliminates any low-contrast keypoints and edge keypoints and what remains is strong interest points.
Now an orientation is assigned to each keypoint to achieve invariance to image rotation. A neighbourhood is taken around the keypoint location depending on the scale, and the gradient magnitude and direction is calculated in that region. An orientation histogram with 36 bins covering 360 degrees is created (It is weighted by gradient magnitude and gaussian-weighted circular window with σ equal to 1.5 times the scale of keypoint). The highest peak in the histogram is taken and any peak above 80% of it is also considered to calculate the orientation. It creates keypoints with same location and scale, but different directions. It contributes to stability of matching.

- **FAST**: (Features from Accelerated Segment Test)
  1. Select a pixel p in the image which is to be identified as an interest point or not. Let its intensity be Ip.
  2. Select appropriate threshold value t.
  3. Consider a circle of 16 pixels around the pixel under test.
  4. Now the pixel p is a corner if there exists a set of n contiguous pixels in the circle (of 16 pixels) which are all brighter than Ip+t, or all darker than Ip−t.. n was chosen to be 12.
  5. A high-speed test was proposed to exclude a large number of non-corners. This test examines only the four pixels at 1, 9, 5 and 13 (First 1 and 9 are tested if they are too brighter or darker. If so, then checks 5 and 13). If p is a corner, then at least three of these must all be brighter than Ip+t or darker than Ip−t. If neither of these is the case, then p cannot be a corner. The full segment test criterion can then be applied to the passed candidates by examining all pixels in the circle. This detector in itself exhibits high performance, but there are several weaknesses:

     a. It does not reject as many candidates for n < 12.
     b. The choice of pixels is not optimal because its efficiency depends on ordering of the questions and distribution of corner appearances.
     c. Results of high-speed tests are thrown away.
     d. Multiple features are detected adjacent to one another.

  These weaknesses can be addressed using machine learning corner detector , or nonmaximal suppression

  Fast is several times faster than other existing corner detectors. But it is not robust to high levels of noise and It is dependent on a threshold.

- **ORB:** ORB is basically a fusion of FAST keypoint detector and BRIEF descriptor with many modifications to enhance the performance. First it uses FAST to find keypoints, then applys Harris corner measure to find top N points among them. It also uses pyramids to produce multiscale-features. The problem is that FAST doesn't compute the orientation so it computes the intensity weighted centroid of the patch with located corner at center. The direction of the vector from this corner point to centroid gives the orientation. To improve the rotation invariance, moments are computed with x and y which should be in a circular region of radius r, where r is the size of the patch.

There are more key point detectors such as SURF but they are not addressed in this project.

Descriptors:

- **SIFT:** 16x16 neighbourhood around the keypoint is taken. It is divided into 16 sub-blocks of 4x4 size. For each sub-block, 8 bin orientation histogram is created. So a total of 128 bin values are available. It is represented as a vector to form keypoint descriptor. In addition to this, several measures are taken to achieve robustness against illumination changes, rotation.
- **BRIEF:** provides a shortcut to find the binary strings directly without finding descriptors. It takes smoothened image patch and selects a set of n_d(x,y) location pairs in an unique way (explained in paper). Then some pixel intensity comparisons are done on these location pairs. For eg, let first location pairs be p and q. If I(p)<I(q), then its result is 1, else it is 0. This is applied for all the n_d location pairs to get a n_d-dimensional bitstring. This n_d can be 128, 256 or 512.
- **ORB:** ORB uses BRIEF descriptors but because BRIEF performs poorly with rotation so orb steers brief according to the orientation of the keypoints. ORB discretize the angle to increments of 2π/30 (12 degrees), and construct a lookup table of precomputed BRIEF patterns. As long as the keypoint orientation θ is consistent across views, the correct set of points S_θ will be used to compute its descriptor. BRIEF has an important property that each bit feature has a large variance and a mean near 0.5. But once it is oriented along keypoint direction, it loses this property and become more distributed. High variance makes a feature more discriminative, since it responds differentially to inputs. Another desirable property is to have the tests uncorrelated, since then each test will contribute to the result. To resolve all these, ORB runs a greedy search among all possible binary tests to find the ones that have both high variance and means close to 0.5, as well as being uncorrelated. The result is called rBRIEF. For descriptor matching, multi-probe LSH which improves on the traditional LSH, is used. ORB is a good choice in low-power devices for panorama stitching etc.
- **SURF:** not performed in this project.

Matchers:

- **BF:** brute force matcher takes the descriptor of one feature in first set and is matched with all other features in second set using some distance calculation. And the closest one is returned.
- **FLANN:** Fast Library for Approximate Nearest Neighbors contains a collection of algorithms optimized for fast nearest neighbor search in large datasets and for high dimensional features. It works faster than BF Matcher for large datasets. In our case we use kd tree.

An alternative to matching we can use KLT algorithim to find the optical flow of detected key points and hence find the next location of the good features.

- **Optical flow:** Optical flow is the pattern of apparent motion of image objects between two consecutive frames caused by the movement of object or camera. It is 2D vector field where each vector is a displacement vector showing the movement of points from first frame to second.

  Optical flow works on several assumptions:

  1. The pixel intensities of an object do not change between consecutive frames.
  2. Neighbouring pixels have similar motion.

  Consider a pixel I(x,y,t) in first frame It moves by distance (dx,dy) in next frame taken after dt time. So since those pixels are the same and intensity does not change, we can say,

$$I(x,y,t) = I(x+dx, y+dy, t+dt)$$

  Then take taylor series approximation of right-hand side, remove common terms and divide by dt to get the following equation:

$$f_x u + f_y v + f_t = 0$$

  Where:

$$f_x = \frac{\partial f}{\partial x} \; ; \; f_y = \frac{\partial f}{\partial y}$$

$$u = \frac{dx}{dt} \; ; \; v = \frac{dy}{dt}$$

  Above equation is called Optical Flow equation. In it, we can find fx and fy, they are image gradients. Similarly, ft is the gradient along time. But (u,v) is unknown. We cannot solve this one equation with two unknown variables. So several methods are provided to solve this problem and one of them is Lucas-Kanade.

  Lucas-Kanade methode: will assume that all the neighbouring pixels will have similar motion. Lucas-Kanade method takes a 3x3 patch around the point. So all the 9 points have the same motion. We can find (fx,fy,ft) for these 9 points. So now our problem becomes solving 9 equations with two unknown variables which is over-determined. A better solution is obtained with least square fit method. Below is the final solution which is two equation-two unknown problem and solve to get the solution.

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \sum_i f_{x_i}^2 & \sum_i f_{x_i} f_{y_i} \\ \sum_i f_{x_i} f_{y_i} & \sum_i f_{y_i}^2 \end{bmatrix}^{-1} \begin{bmatrix} - \sum_i f_{x_i} f_{t_i} \\ - \sum_i f_{y_i} f_{t_i} \end{bmatrix}$$

So from the user point of view, the idea is simple, we give some points to track, we receive the optical flow vectors of those points. But again, there are some problems. Until now, we were dealing with small motions, so it fails when there is a large motion. To deal with this we use pyramids. When we go up in the pyramid, small motions are removed and large motions become small motions. So by applying Lucas-Kanade there, we get optical flow along with the scale.

The pinhole coordinate system:

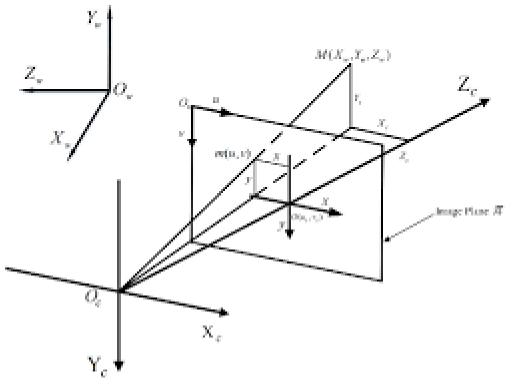


Figure 9: pinhole coordinating system

Forward movement of the vehicle is Z and the side translation is X in the pinhole coordinate system.

Essential matrix:

The Essential Matrix is a 3 x 3 matrix that encodes epipolar geometry Given a point in one image, multiplying by the essential matrix will tell us the epipolar line in the second view.

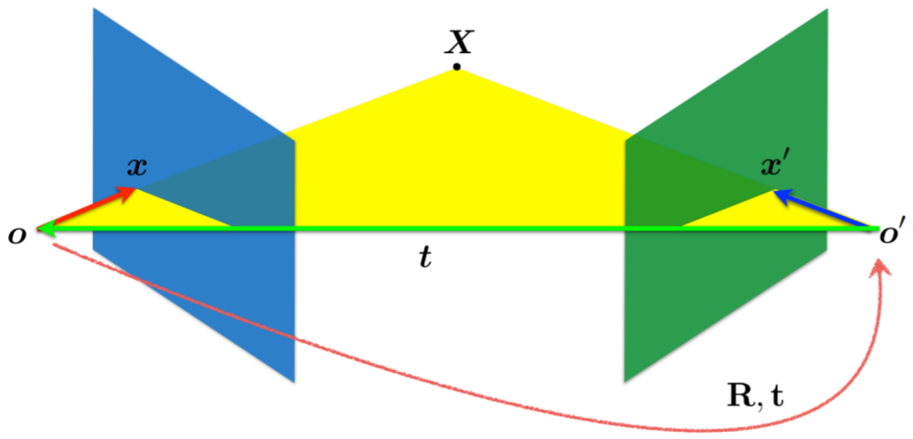


Figure 10: epipolar geometry

From this we can derive the epipolar constraint where x, x'(corresponding points), t are coplanar:

rigid motion $\quad\quad$ coplanarity

$$\boldsymbol{x}' = \mathbf{R}(\boldsymbol{x} - \boldsymbol{t}) \qquad (\boldsymbol{x} - \boldsymbol{t})^\top (\boldsymbol{t} \times \boldsymbol{x}) = 0$$

$$(\boldsymbol{x}'^\top \mathbf{R})(\boldsymbol{t} \times \boldsymbol{x}) = 0$$
$$(\boldsymbol{x}'^\top \mathbf{R})([\mathbf{t}_\times]\boldsymbol{x}) = 0$$
$$\boldsymbol{x}'^\top (\mathbf{R}[\mathbf{t}_\times])\boldsymbol{x} = 0$$

$$\boldsymbol{x}'^\top \mathbf{E}\boldsymbol{x} = 0$$

The essential matrix is then defined as: $E = R[T]_x$

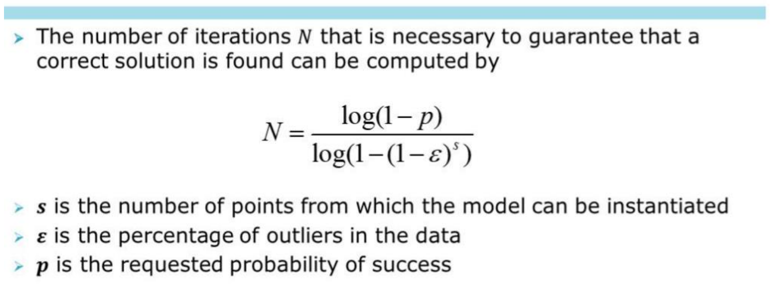
To compute the essential matrix we need at least 5 corresponding points and the more we have the better as we will refine the calculation using ransac algorithm

Solving:

$$[p_2;1]^T K^{-T} E K^{-1} [p_1;1] = 0$$

where E is an essential matrix, p1 and p2 are corresponding points in the first and the second images, respectively. We will then pass the essential matrix to recover the pose. to recover the relative pose between cameras.

Random sample consensus (**RANSAC**) is an iterative method to estimate parameters of a mathematical model from a set of observed data that contains outliers, when outliers are to be accorded no influence on the values of the estimates. Therefore, it also can be interpreted as an outlier detection method. In the following figure we can see how to define ransac algorithm iterations:



- The number of iterations N that is necessary to guarantee that a correct solution is found can be computed by

$$N = \frac{\log(1-p)}{\log(1-(1-\varepsilon)^s)}$$

- s is the number of points from which the model can be instantiated
- ε is the percentage of outliers in the data
- p is the requested probability of success

Figure 11: RANSAC number of iterations

we then estimate the pose recovering the relative camera rotation and the translation from an estimated essential matrix and the corresponding points in two images, using cheirality check.

his works by decomposes an essential matrix using decomposeEssentialMat ,which decomposes the essential matrix E using svd decomposition giving four possible poses exist for the decomposition of E. They are [R1,t], [R1,−t], [R2,t], [R2,−t].

If E gives the epipolar constraint:

$$[p_2; 1]^T K^{-T} E K^{-1} [p_1; 1] = 0$$

between the image points p1 in the first image and p2 in second image, then any of the tuples [R1,t], [R1,−t], [R2,t], [R2,−t] is a change of basis from the first camera's coordinate system to the second camera's coordinate system. However, by decomposing E, one can only get the direction of the translation. For this reason, the translation t is returned with unit length. making the translation is known up to scale.

We then decide which one of the four options is the most fitting using cheek reality meaning making sure that the triangulated 3D points should have positive depth.

We know have the rotation and translation vectores where the translation is known up to scale,

We now transform to homogeneous transformation:

$$T_k = \begin{bmatrix} R_{k,k-1} & t_{k,k-1} \\ 0 & 1 \end{bmatrix} = \arg\min_{X^i, C_k} \sum_{i,k} \underbrace{\| p_k^i - g(X^i, C_k) \|^2}_{\text{Minimize reprojection error}}$$

Scale correction:

In order to recover the unkown scale factor we use the true translation distance as our info measure it could have been made better with lidar info and gps info of true distance between two points as well we used the following scale factor

$$scale_i = \frac{median(diff|GT_{0:i}|))}{median(diff(|VO_{0:i}|))}$$

We find the median of all distances of difference of GT[0:i] devided by the same operation on the estimated visual odamatry.

The code:

main -> Q2 -> dataodamatry retreval -> diplay ground truth -> vo.apply_vo_on_all_frames(detector_type = 'sift'/orb/optical flow,matcher = 'BF'/FLANN) -> for all frames :

find_rotation_and_translation_between_frames_no_scale -> extract 2 frames -> extract_key_points_and_decripters_between_two_frames -> match_features_between_two_frames -> (these 2 steps are slightly different for optical flow as expleind before as matching is KLT based) -> esimate_motion_between_2_frames -> findEssentialMat + recover pose -> transform to homogenous

-> scale_correction -> calc_cur_rot_and_trans_with_scale
-> plot trajectories

b. Analysis results:

A. in the animation we can see the matched key points of the sequential frames and the plotting of the ground truth trajectory the estimated trajectory without scale and estimated trajectory with scale correction, this has been implemented an all types of matchers descriptors and key point techniques however we will display only the runs:
- VO animation sift FLANN with key points.mp4 (best)
- VO animation sift BF with key points.mp4
as they received the best result.
We can from the images of the key points that we get a generally good points for feature extraction however they are not sparse enough this may have impacted the result. More over we see that there is a point in the drive which goes under a tunnel changing the illumination in the whole image this also could have caused the drift from the ground truth in the next figure we see an example of the matched points and the trajectory in VO sift animation you can get an even better perspective.

B. in the following figure we show the ground truth trajectory:

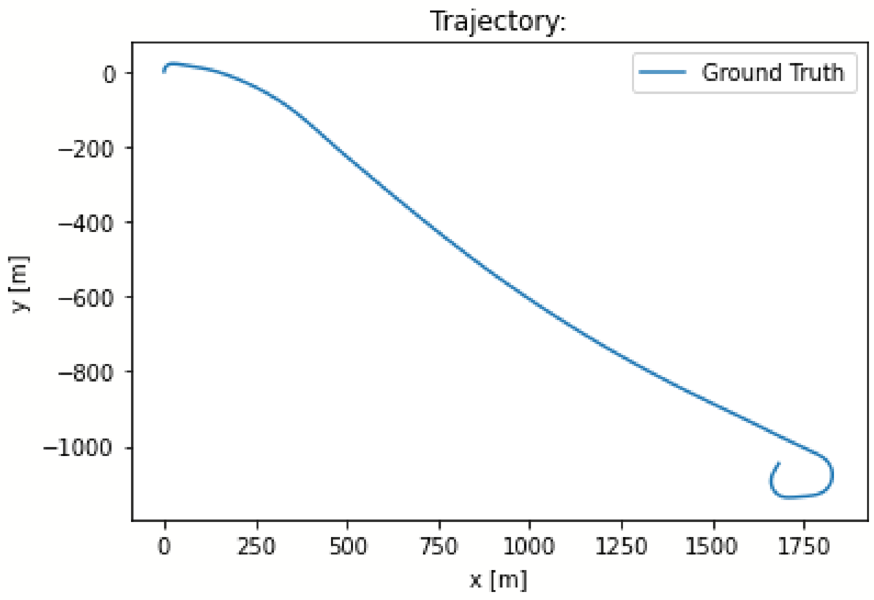


Figure 12: the ground truth trajectory

the trajectory starts at a turn to the right then continues straight with a slight inclination also to the right continuing on that road for about a kilometer and a half then turns right again in a spiral manor. The road is relatively open and looks like a sort of country side highway. On the way cars pass by and we pass under a bridge most of the scenery trees.

We will now analyze the performance we were able to achieve:

- sift bfm matcher ratio test and RANSAC:

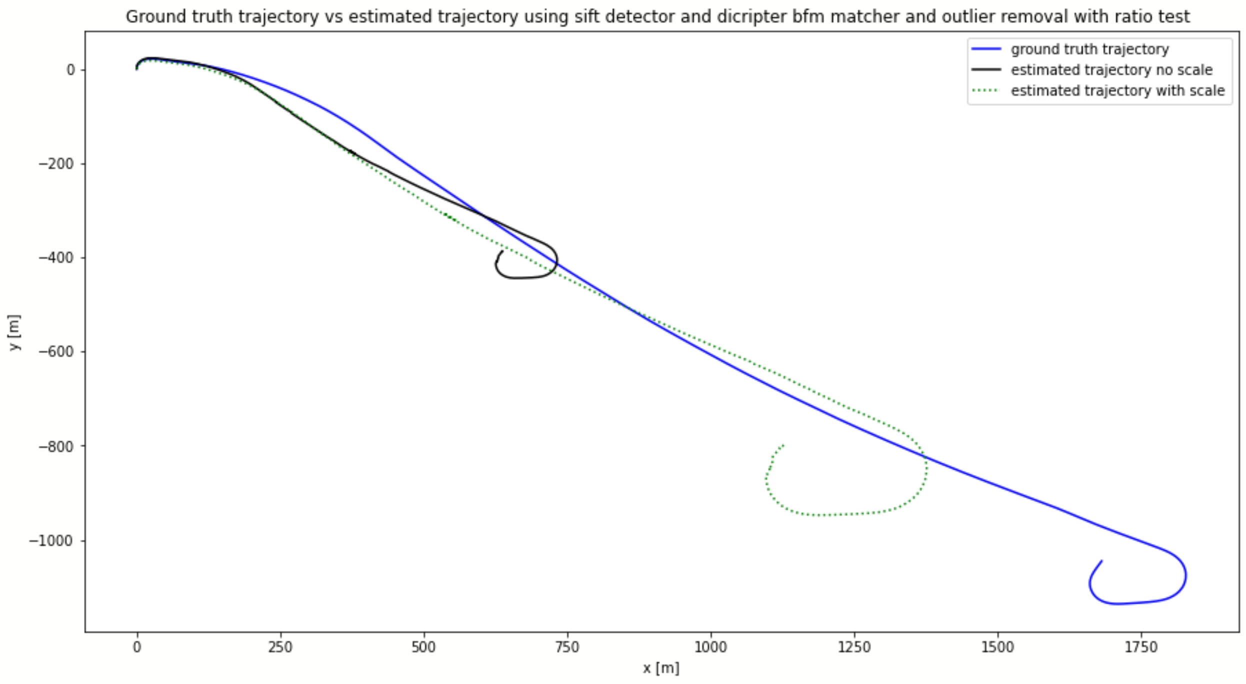


Figure 13: ground truth trajectory vs estimated trajectory and estimated trajectory with scale using sift bfm matcher ratio test and RANSAC

As can be seen in the figure the general shape of the estimation is very similar to the ground truth shape however because we are using a mono camara the translation is only known up to scale as a result we see that the estimation without scale is very far off and it predicts that the full trajectory ends at around (650, -450) and the ground ground truth ends around (1650, -1000). Using scale correction, we are able to make our results better however it is still pretty far off but is much better then the result without scale the areas that we usually drift during turns where it is harder to match key points and change of illumination for example passing throw the tunnel.
In the next figure we see that matched points of 2 frames on the 2 images and the current trajectory:

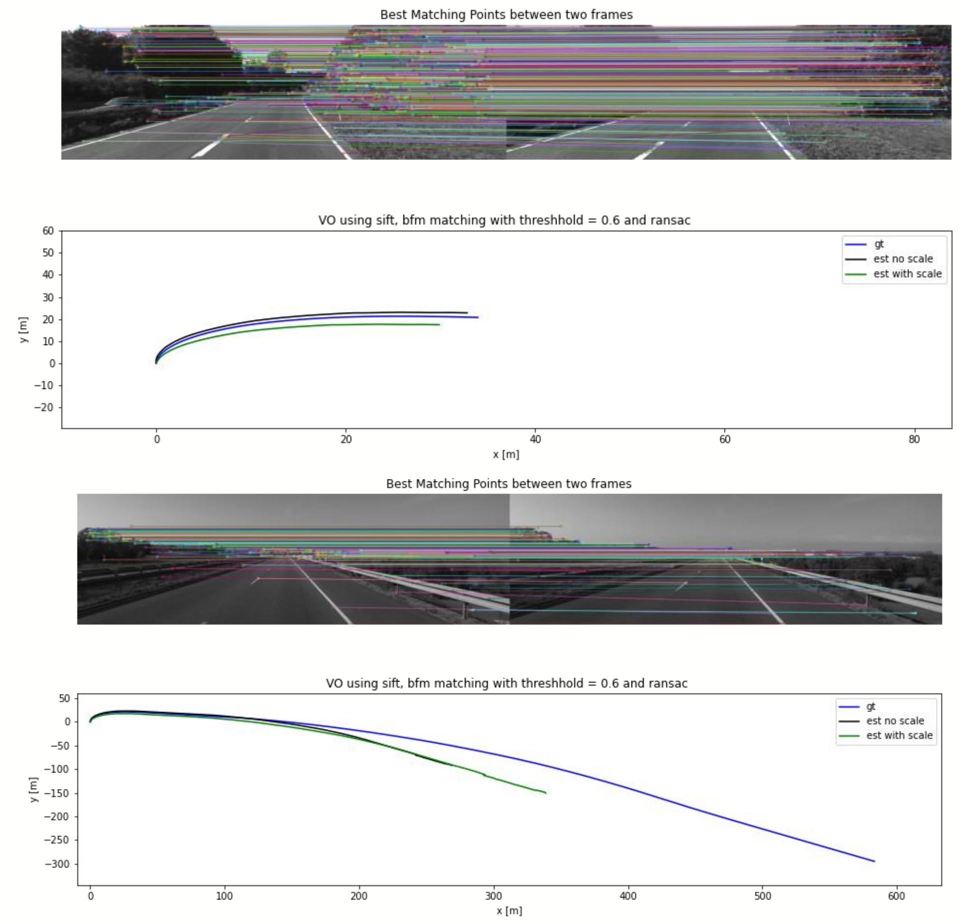


Figure 14: matched points between 2 frames sift BFM

As can be seen we get strong matching points (around 300 points are found however we are displaying the best 100) we would like to have them a little more scattered around the image that might have improved the reslut but in general there are suffeciant number of good key points

C. To fully solve these issues would be to use a stereo camera where we could retreve the depth of the key points and use PNP Algorithm to match the points in 3D, we could also fuse the lidar and GPS info to get an even better results more over we can try to refine the threshold parameters even more and possibly try to use different scale techniques, more techneches could use deep learning to find depth of the mono images. More over we have not tried surf algorithm which could also give an interesting insight.

Analyzing other methods:

- SIFT FLAN matcher ratio test and RANSAC:

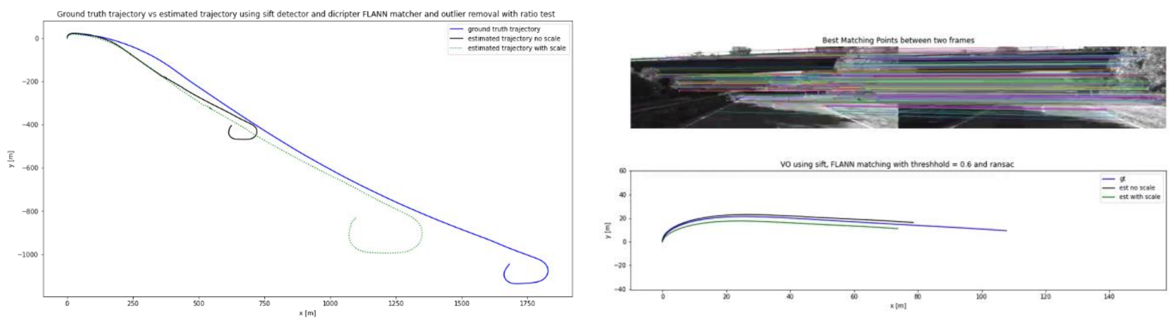


Figure 15: matched points betw een 2 frames sift FLAMM

Here we can see that matching points before going under the bridge. Using FLANN matcher we can see we get similar results to the BF matcher however the estimated trajectory is a little more aligned with the ground truth.

- ORB BF matcher ratio test and RANSAC:

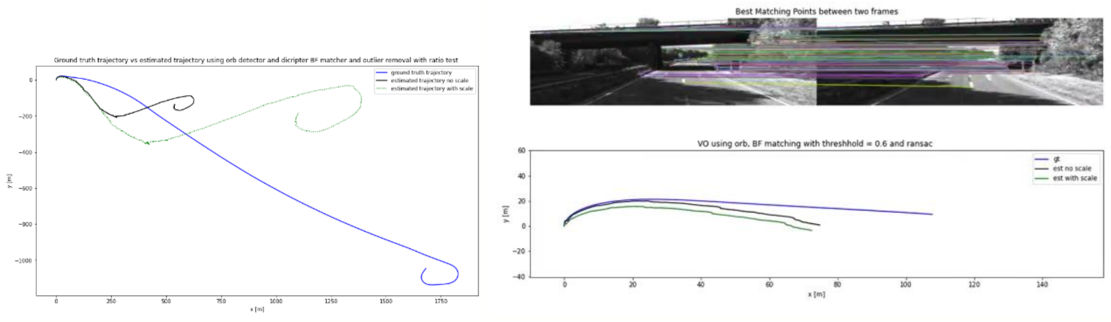


Figure 16: matched points betw een 2 frames sift FLAMM

Here we see a much bigger divergence from the ground truth trajectory. In the right image we see how the shadows of the trees and the entrance to the tunnel change the illumination making the matching process more difficult and hence the divergence from the ground truth starts there. Possibly by refining the threshold parameters we could have got a better result.

- ORB FLANN matcher ratio test and RANSAC:

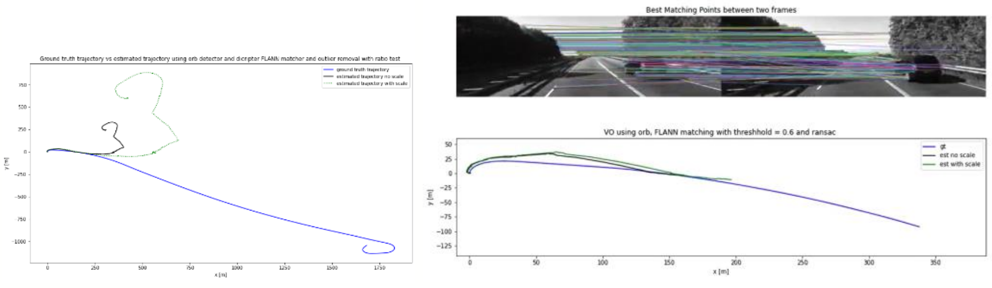


Figure 17: matched points betw een 2 frames ORB FLANN

Here we see that there is a very difference between the ground truth an the estimate trajectory in the right image we can see the key points latch on to a dynamic car in front of them that isn’t the best keypoint to take. Perhaps we could improve this result with different threshold parameters.

- Shi Tomasi Optiacal flow:

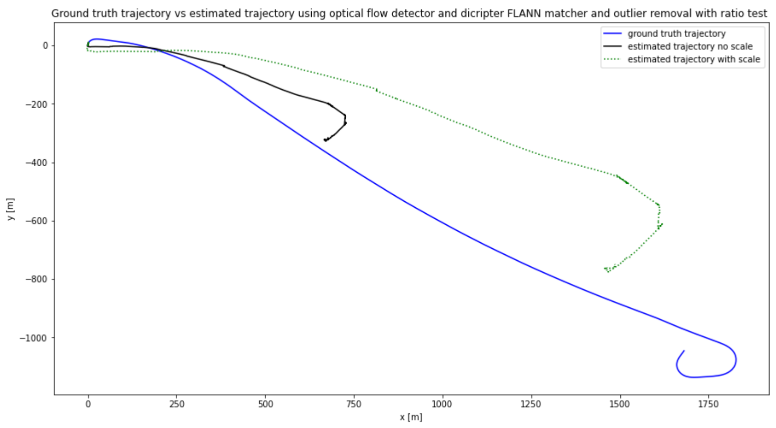


Figure 18: Optical flow

Here we see the performance with optical flow the results are not amazing and probably could be refined with the right hyper parameters.

Summary

In this project we implemented the particle filter we saw how we can estimate the pose of a robot (given the robots true odamatry data and landmark measurements).
Using particle's hypothesis that are normally spread out on the map and by noised motion, landmark measurement, weighting and resampling the particles we can reach the true robot trajectory, with very low RMSE and maxE In the second part we saw how to implement Visual Odamatry algorithm on a mono camera we found key points matched them between frames and estimated the rotation and translation and scale in various techniques to achieve the pose estimation using a mono camera. We saw how we encounter a scale ambiguity using only one camera and how to improve the result using additional info to correct the scale.
In general we were able to get good trajectory estimations in both algorithms.

Code

```
import os
from visual_odometry import VisualOdometry
from data_loader import DataLoader


class ProjectQuestions:
    def __init__(self,vo_data):
        assert type(vo_data) is dict, "vo_data should be a dictionary"
        assert all([val in list(vo_data.keys()) for val in ['sequence', 'd
ir']]), "vo_data must contain keys: ['sequence', 'dir']"
        assert type(vo_data['sequence']) is int and (0 <= vo_data['sequenc
e'] <= 10), "sequence must be an integer value between 0-10"
        assert type(vo_data['dir']) is str and os.path.isdir(vo_data['dir'
]), "dir should be a directory"
        self.vo_data = vo_data


    def Q2(self):
        vo_data = DataLoader(self.vo_data)
        vo = VisualOdometry(vo_data)
        xy_gt = vo.display_gt_trajectory()
        #detector_type = 'optical flow' 'sift', 'orb'    matcher = 'BF','F
LANN'
        detctor = 'sift'
        match = 'FLANN'
        T_list = vo.apply_vo_on_all_frames(detector_type = detctor,matcher
 = match)
        vo.calc_trajectory(T_list)
        xy_est, xy_est_scaled = vo.calc_trajectory(T_list,xy_gt,scale=True
)
        vo_data_for_animation1 = DataLoader(self.vo_data)
        vo_data_for_animation2 = DataLoader(self.vo_data)
        ani = vo.build_animation(xy_gt[:xy_est_scaled.shape[0]],xy_est[:xy
_est_scaled.shape[0]], xy_est_scaled, vo_data_for_animation1.images,vo_dat
a_for_animation2.images,'VO using {}, {} matching with threshhold = 0.6 an
d ransac'.format(detctor,match),'x [m]', 'y [m]' , 'gt' ,'est no scale', '
est with scale')
        vo.save_animation(ani, '/content/drive/MyDrive/mapping_and_percept
ion/project_3/Q_2_visual_odomatry', "VO animation {} {} with key points" .
format(detctor,match))



######################################################################


import numpy as np
import cv2
from data_loader import DataLoader
from camera import Camera
import matplotlib.pyplot as plt
import graphs
import matplotlib.animation as animation
import os
import copy

class VisualOdometry:

# write your code here
  def __init__(self, vo_data):
    self.frame_itr = vo_data.images
    self.intrinsics_k = vo_data.cam.intrinsics
    self.extrinsics = vo_data.cam.extrinsics
    self.gt_poses = vo_data.gt_poses
    self.number_of_frames = vo_data.N
    #plt.imshow(next(self.frame_itr),cmap = 'gray')
    #plt.show()
    #plt.imshow(next(self.frame_itr),cmap = 'gray')

  # generate 2 images (Image sequence):
```

```
  # Feature detection:
  # find key points dicripturs or by orb(Fast+brief) or by sift() or by  c
v.goodFeaturesToTrack()(bassed on harris or shi tomasi corner detector) an
d then runing sdiscritors on that area
  # do this fo every 2 sequential images:
  def extract_2_frames(self,cur_frame):
    frame_i_img = cur_frame
    frame_i_plus_1_img = next(self.frame_itr)
    return frame_i_img, frame_i_plus_1_img


  def match_with_optical_flow(self,frame1, frame2):

    # params for ShiTomasi corner detection
    feature_params = dict( maxCorners = 300,
                           qualityLevel = 0.2,
                           minDistance = 10,
                           blockSize = 1 )
    # Parameters for lucas kanade optical flow
    lk_params = dict( winSize  = (7, 7),
                      maxLevel = 3,
                      criteria = (cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERI
A_COUNT, 10, 0.03))


    frame1_points = cv2.goodFeaturesToTrack(frame1, mask = None, **feature
_params)

    # calculate optical flow
    frame2_points, st, err = cv2.calcOpticalFlowPyrLK(frame1, frame2, fram
e1_points, None, **lk_params)
    if frame2_points is not None:
      good_new_points = frame2_points[st==1]
      good_old_points = frame1_points[st==1]
    print("good_new_points", len(good_new_points))

    return good_old_points, good_new_points #,frame1_points

  def extract_key_points_and_decripters_between_two_frames(self,frame1, fr
ame2, detector_type = 'sift'):

    if detector_type == 'sift':
      #detector = cv2.xfeatures2d.SIFT_create()
      detector = cv2.SIFT_create()
    elif detector_type == 'orb':
      detector = cv2.ORB_create()

    frame1_keypoints, frame1_descriptor = detector.detectAndCompute(frame1
, None)
    frame2_keypoints, frame2_descriptor = detector.detectAndCompute(frame2
, None)
    return frame1_keypoints, frame1_descriptor, frame2_keypoints, frame2_d
escriptor


  def match_features_between_two_frames(self,frame1_descriptor,frame2_desc
riptor,detector_type = 'sift', matcher = 'BF'):
        # Create a Brute Force Matcher object.
    if matcher == 'BF':
      if detector_type == 'orb':
        bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck = True)
        match_list = bf.match(frame1_descriptor, frame2_descriptor)
        # The matches with shorter distance are the ones we want.
        matches = sorted(match_list, key = lambda x : x.distance)




      elif detector_type == 'sift':
        bf = cv2.BFMatcher()
        match_list = bf.knnMatch(frame1_descriptor,frame2_descriptor,k=2)
        # Apply ratio test
        matches = []
        for m,n in match_list:
          #print("m.distance", m.distance, "n.distance",n.distance)
          if m.distance < 0.6*n.distance:
              matches.append(m)

    elif matcher == 'FLANN': #fix this from this site !!!! Feature Matchin
g !!!! in open cv
        #using KD/LSH

        if detector_type == 'orb':
          FLANN_INDEX_LSH = 6
          index_params= dict(algorithm = FLANN_INDEX_LSH,
                             table_number = 6, # 12
                             key_size = 12,      # 20
                             multi_probe_level = 1) #2

        elif detector_type == 'sift':
          FLANN_INDEX_KDTREE = 1
          index_params = dict(algorithm = FLANN_INDEX_KDTREE, trees = 5)


        search_params = dict(checks=50)
        matcher = cv2.FlannBasedMatcher(index_params, search_params)

        # Need to draw only good matches, so create a mask
        #matchesMask = [[0,0] for i in range(len(matches))]
        # ratio test as per Lowe's paper
        if detector_type == 'sift':
          match_list = matcher.knnMatch(frame1_descriptor, frame2_descript
or, k=2)
          matches = []
          #print("match_list", match_list)
          for m,n in  match_list:
              if m.distance < 0.6*n.distance:
                  matches.append(m)

        elif detector_type == 'orb':
          match_list = matcher.match(frame1_descriptor, frame2_descriptor)
          matches = sorted(match_list, key = lambda x : x.distance)

    return matches








  def esimate_motion_between_2_frames(self,matches,frame_i_kp,frame_i_plus
_1_kp,detector_type = 'sift'):


    #essential_mat = np.zeros((3,4))
    ##print("self.intrinsics_k",self.intrinsics_k)

    frame_i_points = np.float32([frame_i_kp[m.queryIdx].pt for m in matche
s])
    frame_i_plus_1_points =  np.float32([frame_i_plus_1_kp[m.trainIdx].pt
for m in matches])
```

```
    essential_mat = cv2.findEssentialMat( frame_i_points,
                          frame_i_plus_1_points,
                          self.intrinsics_k)[0]
    ##print("essential_mat" , essential_mat)

    #This function decomposes an essential matrix using decomposeEssential
Mat and then verifies possible pose hypotheses by doing cheirality check.
    #The cheirality check means that the triangulated 3D points should hav
e positive depth:
    _ , rotation_mat , translation_vector, _ = cv2.recoverPose( essential_
mat,
                                                              frame_i_poin
ts,
                                                              frame_i_plus
_1_points,
                                                              self.intrins
ics_k)

    return rotation_mat, translation_vector , frame_i_points, frame_i_plus
_1_points

  def esimate_motion_between_2_frames_optical_flow(self,frame1,frame2):
    frame_i_points, frame_i_plus_1_points = self.match_with_optical_flow(f
rame1, frame2)
    essential_mat = cv2.findEssentialMat( frame_i_points,
                      frame_i_plus_1_points,
                      self.intrinsics_k)[0]
    _ , rotation_mat , translation_vector, _ = cv2.recoverPose( essential_
mat,
                                                              frame_i_poin
ts,
                                                              frame_i_plus
_1_points,
                                                              self.intrins
ics_k)
    return rotation_mat, translation_vector , frame_i_points, frame_i_plus
_1_points

  def find_rotation_and_translation_between_frames_no_scale(self,cur_frame
,detector_type = 'sift' ,matcher = 'BF'):


      frame_i_img, frame_i_plus_1_img = self.extract_2_frames(cur_frame)
      if detector_type != 'optical flow':
        frame_i_keypoints, frame_i_descriptor, frame_i_plus_1_keypoints, f
rame_i_plus_1_descriptor = self.extract_key_points_and_decripters_between_
two_frames(frame_i_img, frame_i_plus_1_img, detector_type)
        matches = self.match_features_between_two_frames(frame_i_descripto
r, frame_i_plus_1_descriptor, detector_type, matcher)
        rotation_mat, translation_vector , frame_i_points, frame_i_plus_1_
points = self.esimate_motion_between_2_frames(matches, frame_i_keypoints,
frame_i_plus_1_keypoints)
      elif detector_type == 'optical flow':
        rotation_mat, translation_vector , frame_i_keypoints, frame_i_plus
_1_keypoints = self.esimate_motion_between_2_frames_optical_flow(frame_i_i
mg, frame_i_plus_1_img)

      #transform to homogines transformation:
      homogines_trans = np.eye(4)
      homogines_trans[:3,:3] = rotation_mat
      homogines_trans[:3,3] = translation_vector.T
      return frame_i_plus_1_img , homogines_trans, frame_i_keypoints, fram
e_i_plus_1_keypoints ,matches


  def apply_vo_on_all_frames(self,detector_type = 'sift',matcher = 'BF'):

    #get first frame:
    self.detector_type = detector_type
    self.matcher = matcher
    self.frame_i_points_list = []
    self.frame_i_plus_1_points = []
    self.matches_list = []
    frame_i_img = next(self.frame_itr)
    T_list = []
    for i in range(self.number_of_frames-1):
      frame_i_plus_1_img , homogines_trans, frame_i_points, frame_i_plus_1
_points , matches = self.find_rotation_and_translation_between_frames_no_s
cale(frame_i_img,detector_type,matcher)
      frame_i_img = frame_i_plus_1_img
      T_list.append(homogines_trans)
      self.frame_i_points_list.append(frame_i_points)
      self.frame_i_plus_1_points.append(frame_i_plus_1_points)
      self.matches_list.append(matches)
    return T_list
  def display_mateched_key_point(self,img1,img2,img1_keypoints, img1_descr
iptor,img2_keypoints, img2_descriptor,matches):
    '''
    image1 = np.copy(img1)
    image2 = np.copy(img2)

    #cv2.drawKeypoints(img1, img1_keypoints, keypoints_without_size, color
 = (0, 255, 0))

    image1 = cv2.drawKeypoints(img1, img1_keypoints, image1, flags = cv2.D
RAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
    image2 = cv2.drawKeypoints(img2, img2_keypoints, image2, flags = cv2.D
RAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)


    # Display image with and without keypoints size
    fx, plots = plt.subplots(1, 2, figsize=(40,20))

    plots[0].set_title("image1 keypoints With Size")
    plots[0].imshow(image1, cmap='gray')

    plots[1].set_title("Train keypoints With Size")
    plots[1].imshow(image2, cmap='gray')
    plt.show()
    '''
    # Print the number of keypoints detected in the training image
    #print("Number of Keypoints Detected In The Training Image1: ", len(im
g1_keypoints))

    # Print the number of keypoints detected in the query image
    #print("Number of Keypoints Detected In The Query Image2: ", len(img2_
keypoints))

    result = cv2.drawMatches(img1,img1_keypoints,img2,img2_keypoints,match
es[:],None,flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)
    '''
    # Display the best matching points
    plt.rcParams['figure.figsize'] = [16.0, 7.0]
    plt.title('Best Matching Points')
    plt.imshow(result)
    plt.show(result)
    '''
    # Print total number of matching points between the training and query
 images
    #print("\nNumber of Matching Keypoints Between The Training and Query
Images: ", len(matches))
    return result


  def display_gt_trajectory(self):

    x_gt = [0]
    y_gt = [0]
    for gt_pose in self.gt_poses :
      x_gt.append(gt_pose[0][3])
      y_gt.append(gt_pose[2][3])

    xy_gt = np.vstack((x_gt,y_gt)).T
    graphs.plot_single_graph(xy_gt,'Trajectory:', 'x [m]','y [m]','Ground
Truth')
    return xy_gt

# Feature matching (tracking) (outlier removal):
  # match them using bf matcher and remove outliears using ransac , if usi
ng orb use FlannBasedMatcher




# Motion estimation 2D-2D:
  # compute essntial matrix from image pair l_k , l_k-1
  #decompose essential matrix into R_k and t_k to form T_k
  #compute relitive scale and rescale t_k acourdingly
  # concatinate transformation by computing C_k = c_k-1 @ T_k

# Local optimization:

  def diff(self, diff_arry_till_i_minus_1, xy_i):
    diff_arry_till_i_minus_1.append(xy_i - diff_arry_till_i_minus_1[-1] )
    return diff_arry_till_i_minus_1

  #make more efficaint! by adding elemnts to diff array each time
  def scale_correction(self,diff_gt_array_till_i_minus_1,diff_vo_array_til
l_i_minus_1,gt_xy_i,vo_xy_i):
    diff_gt_0_to_i = self.diff(diff_gt_array_till_i_minus_1, gt_xy_i)
    #print("diff_gt_0_to_i", diff_gt_0_to_i)
    diff_vo_0_to_i = self.diff(diff_vo_array_till_i_minus_1, vo_xy_i)
    #print("diff_vo_0_to_i", diff_vo_0_to_i)
    scale_i = np.median(np.linalg.norm(diff_gt_0_to_i,axis = 1))/np.median
(np.linalg.norm(diff_vo_0_to_i, axis = 1))
    #print("scale_i", scale_i)
    return scale_i , diff_gt_0_to_i , diff_vo_0_to_i

  def calc_cur_rot_and_trans_with_scale(self,accumulated_t ,cur_t, accumul
ated_R, cur_R, scale):
    t = accumulated_t + scale*np.matmul(accumulated_R,cur_t)
    R = np.matmul(accumulated_R,cur_R)
    return R, t

  def calc_trajectory(self,T_list,xy_gt = None, scale = False):

    x_est = [0]
    y_est = [0]
    C_acumilated = np.eye(4)
    C_list = []
    # intlizing xy est
    for T in T_list:
      C_acumilated = np.matmul(C_acumilated,T)
      C_list.append(C_acumilated)
      x_translation = C_acumilated[0,3] # x_est[-1] +
      y_translation = C_acumilated[2,3] #y_est[-1] + C_acumilated[1,3]
      x_est.append(x_translation)
      y_est.append(-y_translation)
    xy_est = np.vstack((np.array(x_est),np.array(y_est))).T
    #print ("C_list",C_list)


    if(scale == False):
      graphs.plot_single_graph(xy_est,'Trajectory:', 'x [m]','y [m]','esti
mated trajectory without scale')

    elif (scale == True):
      #get scale_i
      diff_gt_array_till_i_minus_1 = [np.array([0.,0.])]
      diff_vo_array_till_i_minus_1 = [np.array([0.,0.])]
      scale_array = []
      #print(xy_gt)
      for i in range(xy_gt.shape[0]-1):
        scale_i , diff_gt_array_till_i_minus_1 , diff_vo_array_till_i_minu
s_1 = self.scale_correction(diff_gt_array_till_i_minus_1,diff_vo_array_til
l_i_minus_1,xy_gt[i],xy_est[i])
        scale_array.append(scale_i)
      #print("scale_array", scale_array)
      scale_array = scale_array[2:]
      #print("scale_array", scale_array)

      print("self.number_of_frames", self.number_of_frames)
      x_est_scaled_t_minus_1 = [0]
      y_est_scaled_t_minus_1 = [0]
      x_est_scaled = []
      accumulated_R = np.eye(3)
      accumulated_t = np.array([0,0,0])
      #cheek again :
      print("len(scale_array)",len(scale_array))
      print("len(T_list)",len(T_list))
      for i, T in enumerate(T_list):
        if(i<len(scale_array)):
          accumulated_R, accumulated_t = self.calc_cur_rot_and_trans_with_
scale(accumulated_t, T[:3,3], accumulated_R,T[:3,:3] ,scale_array[i])
          x_translation = accumulated_t[0]
          y_translation = accumulated_t[2]
          x_est_scaled_t_minus_1.append(x_translation)
          y_est_scaled_t_minus_1.append(-y_translation)


      xy_est_scaled = np.vstack((np.array(x_est_scaled_t_minus_1),np.array
(y_est_scaled_t_minus_1))).T
      graphs.plot_single_graph(xy_est_scaled,'Trajectory:', 'x [m]','y [m]
','estimated trajectory with scale')


      graphs.plot_three_graphs(xy_gt, xy_est, xy_est_scaled,'Ground truth
trajectory vs estimated trajectory using {} detector and dicripter {} matc
her and outlier removal with ratio test '.format(self.detector_type,self.m
atcher) ,'x [m]', 'y [m]' , 'ground truth trajectory', 'estimated trajecto
ry no scale','estimated trajectory with scale')

      return xy_est, xy_est_scaled



  def build_animation(self,X_Y0, X_Y1, X_Y2,frame_itr1,frame_itr2, title,
xlabel, ylabel, label0, label1, label2):

    frames = []
    fig = plt.figure(figsize=(16, 8))
    ax_img = fig.add_subplot(2,1,1)
    ax = fig.add_subplot(2,1,2)
    self.i = 0

    print("Creating animation")

    x0, y0, x1, y1, x2, y2 = [], [], [], [], [],[]
    val0, = plt.plot([], [], 'b-', animated=True, label=label0)
    val1, = plt.plot([], [], 'k-', animated=True, label=label1)
    val2, = plt.plot([], [], 'g-', animated=True, label=label2)
    frame_i_plus_1_img = next(frame_itr2)
    plt.legend()
    values = np.hstack((X_Y0, X_Y1, X_Y2))


    def init():
        ax.set_title(title)
        ax.set_xlabel(xlabel)
        ax.set_ylabel(ylabel)
        val0.set_data([],[])
        val1.set_data([],[])
        val2.set_data([],[])
        return val0, val1, val2,


    def update(frame):

        frame_i_img = next(frame_itr1)
        result = copy.copy(frame_i_img)
        #print(self.frame_i_points_list[self.i])
        result = cv2.drawKeypoints(frame_i_img,self.frame_i_points_list[se
lf.i],result) #,flags=cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
        self.i = self.i+1
        '''
        #to diplay matched points between images (runs slowly)
        frame_i_plus_1_img = next(frame_itr2)
        frame_i_keypoints, frame_i_descriptor, frame_i_plus_1_keypoints, f
rame_i_plus_1_descriptor = self.extract_key_points_and_decripters_between_
two_frames(frame_i_img, frame_i_plus_1_img, detector_type = self.detector_
type)
        matches = self.match_features_between_two_frames(frame_i_descripto
r, frame_i_plus_1_descriptor, detector_type = self.detector_type, matcher
= self.matcher)
```

```
        result = self.display_mateched_key_point(frame_i_img,frame_i_plus_
1_img,frame_i_keypoints, frame_i_descriptor, frame_i_plus_1_keypoints, fra
me_i_plus_1_descriptor,matches)
        '''

        #print("result", result)
        # Display the best matching points
        #ax_img.rcParams['figure.figsize'] = [16.0, 7.0]
        ax_img.set_title('Best Matching Points between two frames')
        ax_img.imshow(result)
        ax_img.set_axis_off()


        ax.set_xlim(-10, 50 + frame[0])
        ax.set_ylim(-50 + frame[1], 60)
        x0.append(frame[0])
        y0.append(frame[1])
        x1.append(frame[2])
        y1.append(frame[3])
        x2.append(frame[4])
        y2.append(frame[5])
        val0.set_data(x0, y0)
        val1.set_data(x1, y1)
        val2.set_data(x2, y2)



        return val0, val1, val2,# frame_i_img

    anim = animation.FuncAnimation(fig, update, frames=values, init_func=i
nit, interval=1, blit=True)
    return anim


  def save_animation(self,ani, basedir, file_name):
      print("Saving animation")
      Writer = animation.writers['ffmpeg']
      writer = Writer(fps=50, metadata=dict(artist='Me'), bitrate=1800)
      ani.save(os.path.join(basedir, f'{file_name}.mp4'), writer=writer)
      print("Animation saved")
```

Appendix

The code:

Main -> for every time step :

Calc z_t -> pf.apply -> self.motionModel(Ut), self.Observation(), self.resampleParticles(),pf.bestKParticles

The code:

main -> Q2 -> dataodamatry retreval -> diplay ground truth -> vo.apply_vo_on_all_frames(detector_type = 'sift'/orb/optical flow,matcher = 'BF'/FLANN) -> for all frames :

find_rotation_and_translation_between_frames_no_scale -> extract 2 frames -> extract_key_points_and_decripters_between_two_frames -> match_features_between_two_frames ->(these 2 steps are slightly different for optical flow as expleind before as matching is KLT based) -> esimate_motion_between_2_frames ->findEssentialMat + recover pose -> transform to homogenous

-> scale_correction -> calc_cur_rot_and_trans_with_scale
-> plot trajectories