SIFT, SuperPoint, and SPyNet for SFM

Zhuonan Lin

April 6, 2019

A Warm up

A.1 Assignment Setup

Our SFM algorithm is carried on the modified *libviso2*[1], with the provided codes. To set this up, we download and put *dataset*, *libviso2* and *FlowCode* in the same directory. We will use the Matlab interface for all SFM tasks with pre-features/matches from different methods. To do this, we need to first compile the C++ source codes to *mex* modules by running *make.m* in *libviso2/matlab/*. In my case, I run the assignment codes on a Windows machine with '**Microsoft Visual C++ 2017**' for C++ language compilation. After compiling successfully, we can run the codes for the assignment.

A.2 SFM Results Using *libviso2*

By running *libviso2/matlab/demo_viso_mono.m*, we can carry on the baseline SFM algorithm by the modified *libviso2*. The provide codes will both output: (1) rotation error (defined as mean of Frobenius norm of difference) and translation error (defined as mean of the L2 norm between difference) of the camera matrix parts, and (2) predicted trajectory compared to ground truth. These two types of results are organized and showed in Figure 1, the final translation error and rotation error are 1.323 and 2.66×10^{-3}, respectively.

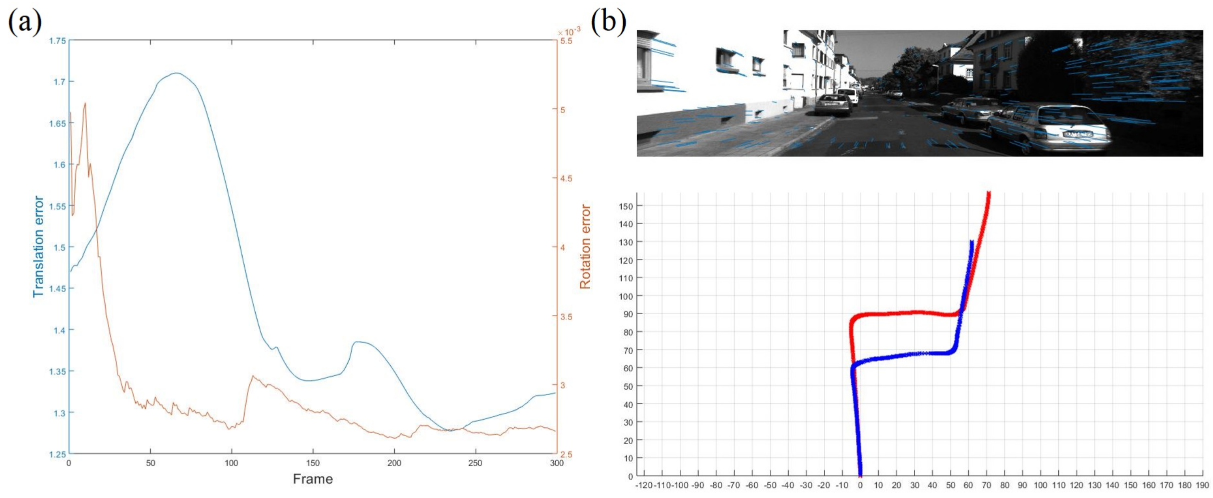

Figure 1: Results from original *libviso2* package. (a) Rotation and translation error in all frames, (b) screen-shot of the final predicted trajectory (blue) compared to the ground truth (red).

A.3 The *libviso2* package

A.3.1 Absolute Pose in SFM

In practical SFM systems, it is more common to use 3-point absolute pose for fundamental matrix estimation rather than 8-point relative pose. The 3-point absolute pose algorithm shows, as we learned in the lecture, after found camera pose from given 3D-2D correspondences, 3 points are sufficient to determine (absolute) pose. All techniques will yield fourth-degree polynomial for further process.

The reason that 3-point absolute pose is more piratical to use in SFM is two-fold, first, 3-point is easier to handle than obtaining 8 points, second, the absolute pose is better to be used in SFM than the relative pose, since it avoid dealing with minimal problems, which is numerically lighter.

A.3.2 Bucketed Feature Points

The features are bucketed during the *Egomotion Estimation* step in *libviso2*, in codes it is carried out in *matcher.cpp: Line*285-326. The basic idea for bucketing, as mentioned in Ref [1], is to **reduce the number of features and spread them uniformly over the image domain**. The specific method for bucketing (in the codes) are: (1) find max values ranges (u_max, v_max, for columnwise and row-wise dimensionality respectively), (2)allocate number of buckets needed, by tuning the *bucket_width*, (e.g. $bucket\_cols = u\_max/bucket\_width) + 1$, (3) assign matches to their buckets, and (4) refill *p_matched* back from buckets, during this process, we add up to *max_features* features from this bucket to *p_matched*. After the bucketing procedure, the new features will be then used for the 'circular' feature matches scheme for egomotion estimation.

A.3.3 Scale of Translation With one Camera

In *viso_mono.cpp : Line* 175, the *libviso2* package handles the scale of translation by:

```
t = t*param.height/d.val[0][best_idx];
```

where t is the translation related matrix, *param.height* is the camera height above ground (in meters), and matrix d is a little bit complicated to explain. It is defined by $d = n{*}x\_plane$, where $n = (cos(-param.pitch), sin(-param.pitch))^T$, and the *param.pitch* is the camera pitch (in rad), and the *x_plane* is the coordinate matrix that contains project features to 2d. The *best_idx* refers the index for find best plane, which is found by a loop earlier. Here, the basic idea behind the translation scale is the method by knowing the height of camera as additional information, as we discussed in Lecture 8, Page 23.

A.3.4 RANSAC in *libviso2*

The basic idea of RANSAC algorithm is outlined below (description mainly borrowed from CSE 252A class lecture note):

RANSAC Algorithm

1. Randomly select a sample of s data points from S and instantiate the model from this subset.
2. Determine the set of data points S_i which are within a distance threshold t of the model. The set S_i is the consensus set of samples and defines the inliers of S.
3. If the size of S_i is greater than some threshold T, reestimate the model using all the points in S_i and terminate
4. If the size of Si is less than T, select a new subset and repeat the above.
5. After N trials the largest consensus set S_i is selected, and the model is re-estimated using all the points in the subset S_i

The RANSAC algorithm in *libviso2* is carried on *viso_mono.cpp : Line* 91-109 as following:

```
 // initial RANSAC estimate of F
Matrix E,F;
inliers.clear();
for (int32_t k=0;k<param.ransac_iters;k++) {

// draw random sample set
vector<int32_t> active = getRandomSample(N,8);

// estimate fundamental matrix and get inliers
fundamentalMatrix(p_matched_normalized,active,F);
vector<int32_t> inliers_curr = getInlier(p_matched_normalized,F);

// update model if we are better
if (inliers_curr.size()>inliers.size())
inliers = inliers_curr;
}

// are there enough inliers?
if (inliers.size()<10)
return vector<double>();
```

It carries out the general RANSAC algorithm on our specific case, with **a little different in logic**. *param.ransac _iters* is the total iterations run for RANSAC. In each iteration, 8 random unique samples are gained by *getRandomSample()* function (step 1), and these 8 samples will be used for calculating the fundamental matrix F for this RANSAC iteration, through $fundamentalmatrix()$ function, F will be then used to get the number of inliers by using function $getInlier()$. Instead of evaluating the inlier result with a threshold T immediately, here we just keep the model (F) that yields largest number of inliers, which corresponds to the best model among all $param.ransac_iters$. An additional step after all RANSAC iteration is to examine if the best model still has inliers less than 10, if so, just discard the model and return a void result (similar to step 4 above).

B Using SIFT for SFM

B.1 Feature Detector and Descriptor Using SIFT

In this part, we will use the feature from SIFT [2]. The SIFT algorithm can be carried out by *VLFEAT* package[3], which has handy Matlab interfaces. After downloading the binary package, we can quickly deploy the codes by running *toolbox/vl_setup.m*.

With *VLFEAT* package, we can use the SIFT detector and descriptor on the trainning sequence images. We need to output results from SIFT detector and descriptor in *000abc_sift.mat* files for SFM later. The following script will be run in *dataset/sequences/00/image_0/* for this propose.

```
% SIFT.m, carrying on SIFT detector and descriptor on images
% output 000abc_sift.mat files for SFM

% setup VLFEAT package
run('../../../../vlfeat-0.9.21/toolbox/vl_setup.m');

% clear up
```

```
delete *.mat;
clear all; clc;

disp('Carrying on SIFT on sample images...');
% reading all image names as a struct
imgs = dir('*.png');
N = size(imgs, 1);
fprintf('Total number of images: %d.\n', N);
disp('SIFT begins...');
for i = 1:N
    % output message for every 20 images
    if rem(i, 20) == 0
        fprintf('%d images have been done...\n', i);
    end
    % readingin image name and output file name
    img_name = imgs(i).name;
    file_name = [img_name(1:end-4), '_sift'];

    % read in image and convert to single data type for VLFEAT package
    I = single(imread(img_name));
    % carry our SIFT, f is feature dectected and d is the descriptor output
    [f, d] = vl_sift(I);
    % get output data, use f(1:2,:) for position only
    % note that to convert to single data type for output
    feature = [single(f(1:2, :)); single(d)];
    % save output for one single image
    save(file_name, 'feature');
end
disp('SIFT has finished on all images successfully!\n');
```

An example for detected feature with orientation assignment is showed in Figure 2.

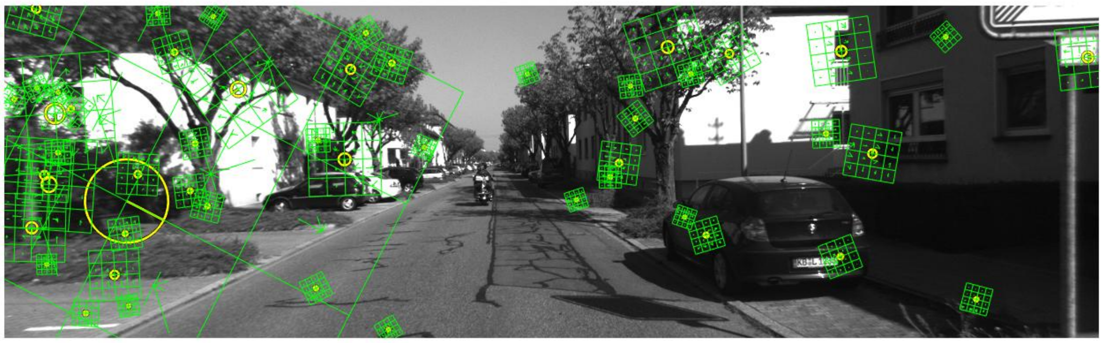

Figure 2: SIFT detected features in the first frame, showing the top 50 ones with the orientation assignment.

B.2 SFM Results Using SIFT Features

With feature outputs from the previous part, we can carry on the SFM algorithm by running *libviso2/matlab/demo_viso_mono_on_preFeature.m*. In this case, we need to set *feature_suffix = 'sift'* and *param_step_size =*

130. The same kinds of results (errors and trajectory) given by this method are showed in Figure 3, the final translation error and rotation error are 1.357 and 2.61×10^{-3}, respectively.

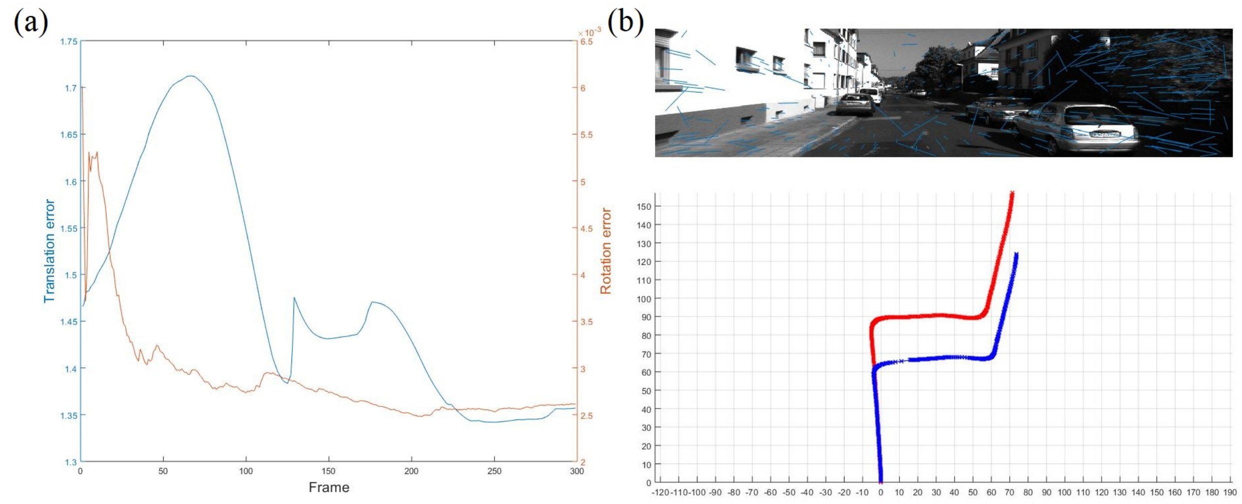

Figure 3: Results from *libviso2* package with SFIT detector and descriptor. (a) Rotation and translation error in all frames, (b) screen-shot of the final predicted trajectory (blue) compared to the ground truth (red).

B.3 Comparison to Original *libviso2*

The translation and rotation error metric from SuperPoint are comparable to the error from original *libviso2*. However, the predicted trajectory is not as goog as the original libviso2. Similar difference between the ground truth and the predicted results are observed for both SIFT and original *libviso2*, but the last part of the predicted trajectory in SIFT is furtherer to the ground truth than the original one, as shown in Figure 3(b). The reason for a worse result even with more accurate pre-found features from SIFT, can possibly result from that we only extract the position information from the SIFT, while neglecting the scale and orientation information from the SIFT detector (i.e. $f(3 : end, :)$ from the detector), so that it is not as suitable as the original detector in this situation.

B.4 SIFT Invariance

B.4.1 Illumination

The illumination invariance is achieved after the descriptor stage. The descriptor itself is a histogram formed from the gradient of the grayscale image. A 4 by 4 spatial grid of gradient angle histograms is used and each of the spatial bins contains an angle histogram divided into 8, which as a result, yields a $4 \times 4 \times 8 = 128$ length feature vector.

the feature vector is then modified to gian the illumination invariance. First, the vector is normalized to unit length. A change in image contrast in which each pixel value is multiplied by a constant will multiply gradients by the same constant, so the vector normalization will cancel this contrast change. A brightness change in which a constant is added to each image pixel will not affect the gradient values, as they are computed from pixel differences. Therefore, the descriptor is invariant to affine changes in illumination. Next, for non-linear change in illumination, we reduce the influence of large gradient magnitudes by thresholding the values in the unit feature vector that is larger than a threshold, and then renormalizing to unit length. This procedure means that matching the magnitudes for large gradients is no longer as important, and that the distribution of orientations has greater emphasis. By these manipulation on the feature vector, the SIFT can achieve illumination invariance.

B.4.2 Rotation

The key step in SIFT to achieve invariance to rotation is the orientation assignment. as the keypoint descriptor can be represented relative to this orientation and therefore achieve invariance to image rotation. After detected the features with eliminating edges and low contrast ones, we carry on an orientation assignment on each feature. To be specific, the idea is to collect gradient directions and magnitudes around each keypoint in a window, whose size is determined by the scale level. Next, we make a histogram, with 10 degree as bin size, is made. The histogram sums up the magnitude of the gradient. Given the histogram, the keypoint will be assigned with a number that represents the largest value in the histogram. Also, any peaks above 80% of the highest peak are converted into a new keypoint. This new keypoint has the same location and scale as the original. But it's orientation is equal to the other peak. Therefore, orientation can split up one keypoint into multiple keypoints. This whole orientation assignment will make SIFT invariant to rotation.

B.4.3 Scale

The scale invariance of SIFT comes from the space scales and the difference-of-Gaussian (DoG) based on that. First, we carry on graduate increasing Gaussian blur on the image. Next, we resize the image as half as original one, and also carry on the similar graduate increasing Gaussian. As a result, we can a set of progressively Gaussian blurring images for a set of progressively small size images (called *octave*).

Instead of the Laplacian for corner/edge detection, SIFT uses the DOG for the approximate Laplacian. DOG is computed from the difference of two nearby scales separated by a constant multiplicative factor k. Combined the analytical derivative property of Gaussian and the numerical expression, the DOG can be related to Laplacian:

$$\sigma \bigtriangledown^2 G = \frac{\partial G}{\partial \sigma} \approx \frac{G(x,y,k\sigma) - G(x,y,\sigma)}{k\sigma - \sigma} \tag{1}$$

then the Laplacian is approximated by:

$$G(x,y,k\sigma) - G(x,y,\sigma) \approx (k-1)\sigma^2 \bigtriangledown^2 G \tag{2}$$

the scale-normalized Laplacian of Gaussian ($\sigma^2 \bigtriangledown^2 G$) with σ^2 is studied in the literature that is required for the true scale invariance. The DOG can be easily calculated based on the different scales for all different octaves.

Once the DOG is calculated, the features will be searched by comparing the pixel in neighbors in current scale, as well as the corresponding locations in the upper and lower scale, also sub-pixel features from Taylor expansion will also be tested, so that the feature in all scales will all be detected, this will also insure the scale invariance.

C Using SuperPoint for SFM

C.1 Feature Detector and Descriptor Using SuperPoint

In this part, we use SuperPoint[4] for detector and descriptor. The demo codes for SuperPoint with pretrained weight are provided on GitHub (*SuperPointPretrainedNetwork/demo_superpoint.py*). The original demo script will only output video or screenshots with detected features. For the feature data we need for our SFM codes, we need to add several lines to the main part in the demo codes, which takes the information from the *VideoStreamer* object, the feature from SuperPoint net (*pts* and *desc*), and output them to *.mat* files. The codes below are added in $Line 673 - 682$ in *SuperPointPretrainedNetwork/demo_superpoint.py.*

```
#SuperPointPretrainedNetwork/demo_superpoint.py
if __name__ == '__main__':

    ...

    # import the module to output Matlab style files (i.e. .mat)
    import scipy.io as sio
    # get the current image name from vs object, add suffix for
       output file
    mat_file_name = vs.listing[vs.i-1][:-4] + "_superpoint.mat"
    print("Saving feature data in .mat format to: " +
       mat_file_name)
    # feature array combined by postions (pts[:2, :]) and
       descriptor (desc)
    feature = np.vstack((pts[:2, :], desc))
    # save to Matlab file, i.e. 000abc__superpoint.mat
    # note to convert the np.array to sigle precision (np.float32
       in Python)
    sio.savemat(mat_file_name, {'feature': np.array(feature,
       dtype=np.float32)})

    ...
```

Then we can run the demo by appropriate flags as following:

```
./demo_superpoint.py ../dataset/sequences/00/image_0/ --cuda --W=1241 --H=376 --no_display --write
```

where --W and --H are the width and height of our data image, respectively. And we use --no_display and --write to suppress the dynamic frame window and output the screenshot of each frame instead. The features .mat files will be written in *../dataset/sequences/00/image_0/* from the output codes above.

An example of SuperPoint detected keypoints is shown in Figure 4

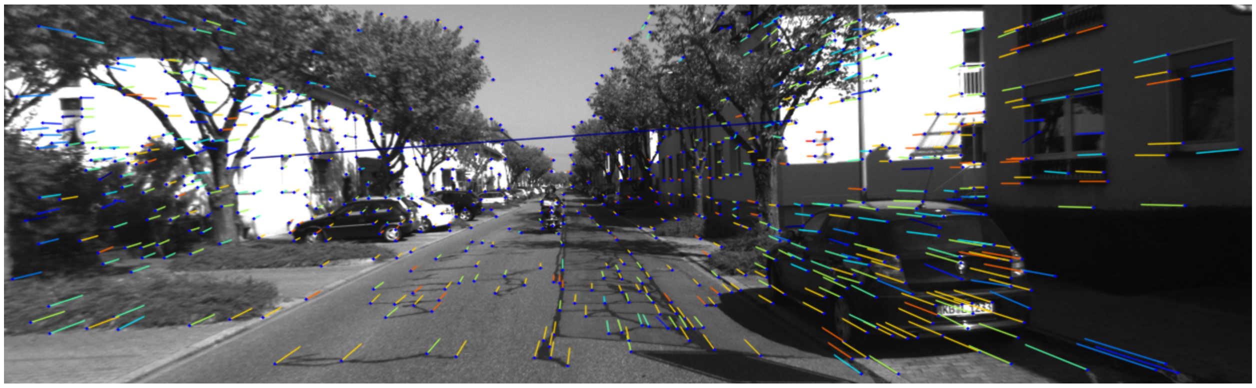

Figure 4: SuperPoint detected keypoints on the first frame.

C.2 SFM Results Using SuperPoint Features

The SFM with SuperPoint net is also a pre-detected feature method. Thus, we will also run the *libviso2/matlab/demo_viso_mono_on_preFeature.m* script with *feature_suffix = 'superpoint'* and *param_step_size* = 258. Again, the errors and trajectories results are showed in Figure 5. the final translation error and rotation error are 1.359 and 3.38×10^{-3}, respectively.

C.3 Comparison to Original *libviso2*

The translation and rotation error metric from SuperPoint are comparable to the error from original *libviso2*. However, the predicted trajectory is closer to the ground truth more than the original one, especially the last some frames. Figure 5(b) shows that the last part of the predicted trajectory is almost same to the ground

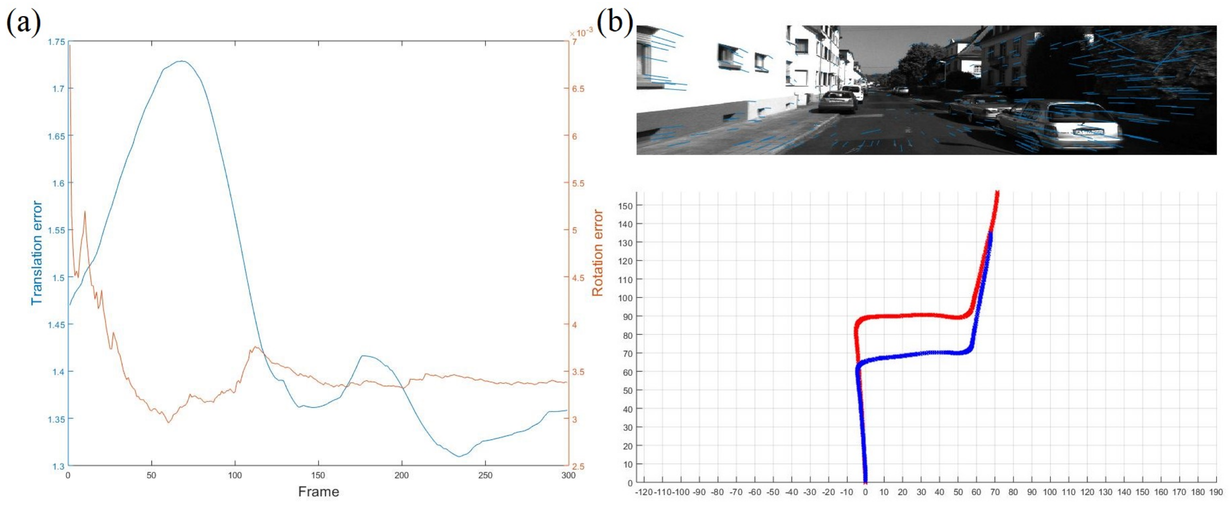

Figure 5: Results from *libviso2* package with SuperPoint detector and descriptor. (a) Rotation and translation error in all frames, (b) screen-shot of the final predicted trajectory (blue) compared to the ground truth (red).

truth. The reason for a better result from SuperPoint can be possibly due to that SuperPoint network is the state-of-art technique for keypoint detection, so that it will provide more appropriate features to the SFM method so that a more accurate result.

C.4 SuperPoint Properties

C.4.1 Learning Ground Truth of Keypoints

The (pseudo) ground truth keypoints can be given by the detector part of the *SuperPoint* network. To be specific, it contains tow parts: *MagicPoint* and *Homographic Adaptation*. The MagicPoint is the base detector in the network, which contains a (shared) encoder and interest point decoder. The encoder is a convolutional network with VGG-style, in order to reduce the dimensionality of the image. The smaller spatial dimension with greater channel depth output given by the encoder will be fit in a sequent interest point detector. The decoder used in MagicPoint is an explicit decoder without parameters, which is known as "sub-pixel convolution" or "pixel shuffle" in PyTorch. The input to the detector contains an extra "no interest point" dustbin, which can be removed after a channel-wise softmax. Also the detector performs a reshape on its input from encoder and get a output with original image dimensionality.

The MagicPoint network is trained on the *Synthetic Shapes* dataset created by the authors. It consists of simplified 2D geometry via synthetic data rendering of simple geometry shapes (eg triangles, lines, ellipses) with homographic warps. The advantage of the artificial dataset is that the samples are simple enough to remove label ambiguity by modeling interest points, which provides good ground truth for the training samples for MagicPoint. Surprisingly, although it is trained on simple synthetic shape data, it performs will on real-world images, especially for those with corner-like structures.

However, the MagicPoint does not perform well under viewpoint changes, and therefore the author invented the *Homograph Adaption*. It applies random homographies ($\mathcal{H}$) to warped copies of the input image and combines the result by unwarping the heatmaps ($\mathcal{H}^{-1}$). In real application, the homograpies are chosen within a pre-determined ranges for translation, scale, in-plane rotation and symmetric perspective distortion and composed together with an initial root center crop to help avoid bordering artifacts. Also, experiments show the number of homographic warps to average the responses on should be around $N_h = 100$ to balance the gain and computational cost. Also this homographic adaptation will be implemented iteratively to continually self-supervise and improve the interest point detector.

The SuperPoint (detector head) consists of the MagicPoint and Homographic Adaptation introduced above, and it can be implemented on unlabeled dataset, such as MS-COCO, to provided ground truth keypoints in a self-supervised fashion.

C.4.2 Ground Truth Matches

After generating the ground truth of keypoints of an image I, the network while perform a random but more restrictive homograph $\mathcal{H}$ on I, and this pair of images will be fed in to train the network (see next subsection for more explanation). Since the corresponding image pair is related by a homograph, with the keypoints detected by SuperPoint detector, therefore, the ground truth of matches can be cheaply obtained between these pairs. For example, the keypoints in the original image will go through the same homograph operation $\mathcal{H}$ as the whole image, thus the corresponding match points coordinates are known to establish the matches cheaply.

C.4.3 Correlated Detection and Description

Since the SuperPoint architecture consists of a deep stack of convolutional layers which extract multi-scale features, it is straightforward to then combine the interest point detector with an additional interest point descriptors. The descriptor takes in the output from the shared encoder mentioned before. It carries on a model similar to UCN to first output a semi-sense grid of descriptor to save memory and time and then performs bi-cubic interpolation of the descriptor for a dense one with fixed length. At last, it L2-normalizes the activations to be unit vector. Note that similar to the decoder head, the descriptor is also non-learnable.

When training the correlated detector-descriptor system, the network uses a final loss function combined the two aspects. The pairs of images that have pseudo-ground truth keypoints and ground truth matches are fed into the network. Denote $\mathcal{L}_p$ and $\mathcal{L}_d$ are the loss functions for detector and descriptor respectively (the detail form of the two loss functions can be found in the paper), then the final loss is given by:

$$\mathcal{L} = \mathcal{L}_p^1 + \mathcal{L}_p^2 + \lambda\mathcal{L}_d \tag{3}$$

where the first two terms are the loss for detector on the two images, and the λ is the coefficient to balance the two kinds of loss. With the structure of network and loss function to train above, SuperPoint can learn a correlated feature representation for the keypoint detection and description.

D Using SPyNet for SFM

D.1 Matches Using SPyNet

The SPyNet [5] written with PyTorch is available on GitHub. Additionally, a demo to run the SpyNet for our SFM algorithm is provided, so that we can simply run the demo script with appropriate flags as following:

```
./demo_spynet.py --cuda --last_frame=299 --weights_path='../SuperPointPretrainedNetwork/superpoint_v1.pth'
```

where --last_frame=299 defines the number of last frame in training data (in our case 299), and --weights_path identifies the path of pretrained weight that the SpyNet model will use. Note that the model used by the SpyNet has been defined in the demo. After running the demo, the *000abc_flow.mat* files will be outputed automatically.

D.2 SFM Results Using SPyNet

In this case, we carry out the pre-matched method with the SPyNet results by running *libviso2/matlab/demo_vi so_mono_on_preMatches.m* directly. Similar errors and trajectories results to A-C will be given by the script and showed in Figure 6.

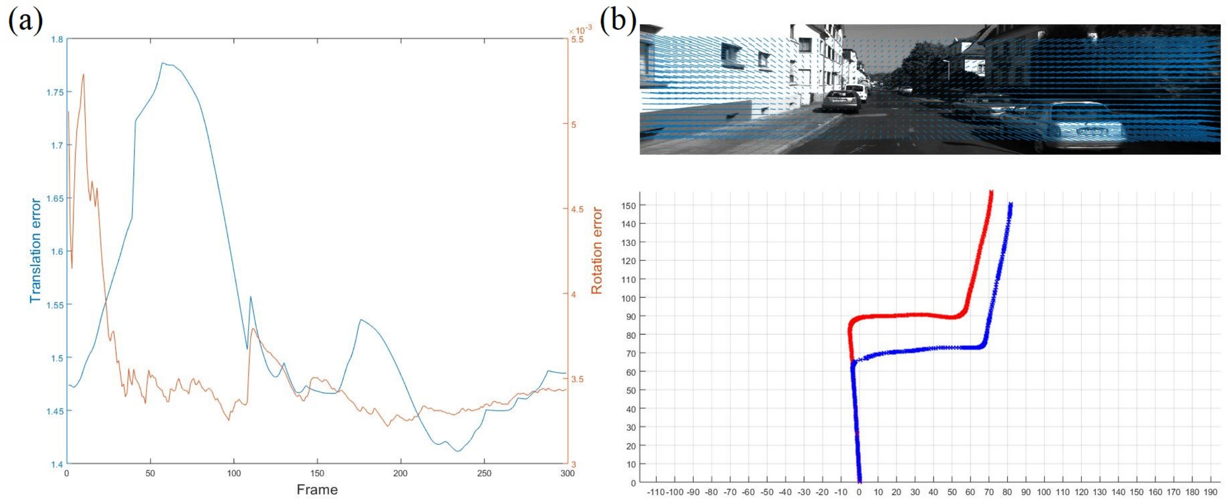

Figure 6: Results from *libviso2* package with SPyNet optical flow matches. (a) Rotation and translation error in all frames, (b) screen-shot of the final predicted trajectory (blue) compared to the ground truth (red).

D.3 Comparison to Original *libviso2*

The translation and rotation error metric from SuperPoint are comparable to the error from original *libviso2*. However, it is kind of counter-intuitive that the predicted trajectory with the optical flow is not as good as the original one. One reason for this is possibly because we use the *superpoint_v1.pth* from pre-trained SuperPoint as the weight in SPyNet, by default in the code.

D.4 SPyNet Properties

The idea behind SPyNet is to combine conventional spatial network with convolutional networks. At each level of the spatial network, the flow is solved by individual convolutional network (G) and then up-sample the flow to next pyramid level. This coarse-to-fine spacial pyramid structure is designed so that the displacement at each level is always less than a few pixels, and therefore the combined convolutional networks can be small with less parameters.

Additionally, this significant reduction in model does not sacrifice the accuracy. The reason is two-fold. (1) *warp function*: with classical formulations, the network warps one image towards the other using the current flow, and repeat this process at each pyramid level. By using the warping function directly, the convolutional net does not need to learn it. (2). *Flow increment*: instead of minimizing a classical objective function, the networks are trained to predict the flow increment at that level, namely the flow correction at each level and add this to the flow output of the network above, in a from coarse to fine fashion. This residual-like learning restricts the range of flow fields in the output space, so that each network only has to model a smaller range of velocities at each level of the spatial pyramid.

The specific structure used by the author is a 5-level spacial pyramid and at each level, the combined network G has 5 layers with 240,050 trainable parameters. The total number of parameters learned by the entire network is 1,200,250 with a usage of memory as small as 9.7M, while FlowNetS and FlowNetC has 32,070,472 and 32,561,032 trainable parameters respectively. SpyNet is about 96% smaller than FlowNet. This enables the SPyNet can be trained and implemented with significantly lower computational cost. Moreover, this simpler framework enables itself to be fit entirely on the GPU for a significant speed up as well as embedded systems for mobile applications.

References

[1] Andreas Geiger, Julius Ziegler, and Christoph Stiller. Stereoscan: Dense 3d reconstruction in real-time. In Intelligent Vehicles Symposium (IV), 2011.

[2] David G Lowe. Distinctive image features from scale-invariant keypoints. IJCV, 60(2):91–110, 2004.

[3] A. Vedaldi and B. Fulkerson. VLFeat: An open and portable library of computer vision algorithms. http://www.vlfeat.org/, 2008.

[4] Daniel DeTone, Tomasz Malisiewicz, and Andrew Rabinovich. Superpoint: Self-supervised interest point detection and description. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops, pages 224–236, 2018.

[5] Anurag Ranjan and Michael J Black. Optical flow estimation using a spatial pyramid network. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 4161–4170, 2017.