Lucas-Kanade Tracker

Contents

In this exercise, we will replace the crude matching method from exercise 3 with a Lucas-Kanade Tracker.

1 Preliminaries

1.1 Outline of the exercise

As seen in today's lecture, a Lukas-Kanade Tracker (KLT[1]) can be used to track the position of a point accross frames. In this exercise, we will implement KLT from scratch. We will start with a well-defined problem where we warp an input image ourselves and see whether KLT can recover the warp. Then, we will apply our KLT to the KITTI sequence and see how it improves on the matching from exercise 3.

[1] for some reason the names are inverted in the acronym

Figure 1: Tracking 50 keypoints with the KLT implementation of this exercise. See https://youtu.be/iaRPafeG9zw for a video. In this exercise, we apply KLT to downsampled images, since implementing the coarse-fine estimation is outside of the scope of this exercise.

Figure 2: Results for the different image warps in part 1.

1.2 Provided code

As usual, `main.m` contains one section for each part of the exercise which you can run individually with `Ctrl+Enter` . Functions to be implemented are provided as documented stubs and are called in `main.m`.

1.3 Conventions

We depart from the notation used in class only slightly, to better distinguish between the **image** in which the template is defined (reference image) and the **coefficients** of the patch of the template. We call the reference image I_R and the patch coefficients

$$T = \{\boldsymbol{x} | \boldsymbol{x} = \boldsymbol{x}_T + \Delta\boldsymbol{x} \quad \forall \Delta\boldsymbol{x} \in [-r_T, r_T] \times [-r_T, r_T] \subset \mathbb{Z} \times \mathbb{Z}\} \tag{1}$$

where we call $\boldsymbol{x}$ the patch center and r_T the patch radius, consistently with previous exercises.

Throughout the exercise, you will need to pay attention to **indexing**! Confusion can ensue because we formulate most things as (x, y), but matrix, and in particular image elements are accessed with (row, column), which corresponds to (y, x)!

2 Part 1: Warping images

Given an image $I(x, y)$, we say $I(W(x, y, \boldsymbol{p}))$ is the image I warped by the warp W which is itself parametrized with the parameters $\boldsymbol{x}$. We define the warp with a 2×3 matrix $\mathbf{W}$ such that

$$W(x, y, \boldsymbol{p}) = \mathbf{W} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}, \quad \mathbf{W} = \begin{bmatrix} p_1 & p_3 & p_5 \\ p_2 & p_4 & p_6 \end{bmatrix}. \tag{2}$$

As seen in class, this is called an affine warp, and generalizes translation, rigid and similarity warps, which are each a special case of the affine warp. In order to get a better understanding of KLT and facilitate debugging, we will manually warp an image with $\mathbf{W}$. Unfortunately, it is not very intuitive to directly manipulate the coefficients of $\mathbf{W}$. Let us instead provide an interface which allows us to get a similarity warp parametrized by the translation $(\Delta x, \Delta y)$, a rotation angle α and the scale change λ. Write the function `getSimWarp` which returns $\mathbf{W}$ such that

$$W(x, y, \boldsymbol{p}) = \lambda(\begin{bmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix}). \tag{3}$$

Then, write the function `warpImage` which warps the input image given $\mathbf{W}$. Note that $W(x, y, \boldsymbol{p})$ is likely not to return integer coefficients. You can for now return $I(\lfloor W(x, y, \boldsymbol{p}) \rfloor)$, later this will need to be changed to bilinear interpolation to increase robustness of KLT. Also, note that $W(x, y, \boldsymbol{p})$ could land outside the image dimensions. In that case just return an intensity of 0. If all goes well, you should obtain the results as in Figure 2.

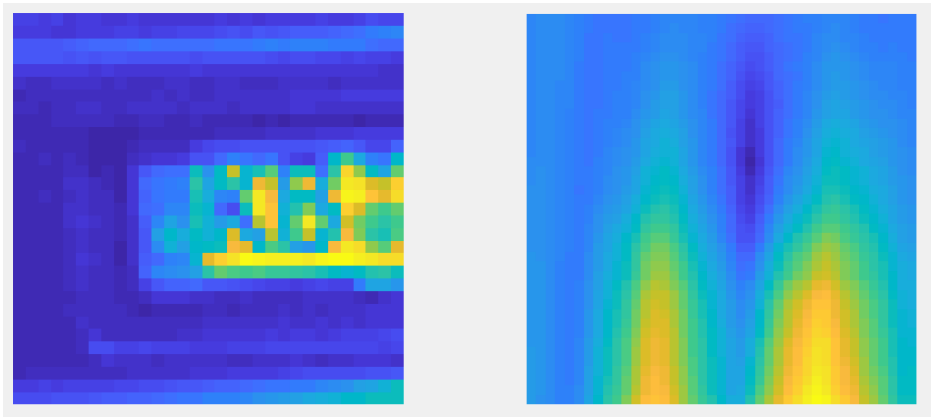

Figure 3: Template and SSDS for Part 2.

3 Part 2: Recovering a simple warp with brute force

An essential part of the KLT implementation will be extracting warped patches:

$$I(W(x, y, \boldsymbol{p})) \quad \forall (x, y) \in T, \tag{4}$$

see (1) for the definition of the patch T. However, there is a slight issue with the current way in which we apply warps. As you can see in Figure 2, rotation and scaling are applied in the top left corner. This makes sense, since that is the origin. But it is also problematic, since for the warping of any image patch not in the top left of the image, rotation and translation are now coupled. This is tedious to think about, but also problematic for the optimization process. In order to avoid this problem, we re-define $I(W(T, \vec{p}))$ such that the origin of the warp is now at the center of the image patch $\vec{x}_T$:

$$I(W(x, y, \boldsymbol{p})) := I(\boldsymbol{x}_T + W(\boldsymbol{x} - \boldsymbol{x_T}, \boldsymbol{p})), \quad \boldsymbol{x} = (x, y) \in T\}, \tag{5}$$

where $T, \boldsymbol{x}_T$ and $\Delta\boldsymbol{x}$ express the image patch, see 1. Implement the function `getWarpedPatch`, which returns $I(W(x, y, \boldsymbol{p}))$. As in the previous part, you can for now return $I(\lfloor W(x, y, \vec{p}) \rfloor)$, but Parts 4 and 5 will not really work well with that. For those parts, you should use bilinear interpolation. You can copy bilinear interpolation from the reference solution of undistortImage in exercise 1.

We can now implement a first, simple, brute force tracker, as seen in the first part of the lecture. The goal of tracking is to find the warp parameters $\mathbf{p}$ which minimizes the sum of square difference (SSD) E between the template and a warped patch from the image in which the template is supposed to be tracked:

$$E(\mathbf{p}) = \sum_{(x,y)\in T} (I(W(x, y, \mathbf{p})) - I_R(x, y))^2 \tag{6}$$

Because the warp is also performed on the patch domain T, we can simply extract how the center point $\boldsymbol{x}_T$ has moved from the resulting warp matrix $\mathbf{W}$: $\Delta\boldsymbol{x}_T = \begin{bmatrix} p_5 & p_6 \end{bmatrix}^T$. Implement `trackBruteForce`, which recovers **translation-only** warp by trying out different values for $\boldsymbol{p} = (p_5, p_6)$ and selects the one with the lowest such SSD:

$$\underset{\boldsymbol{p}}{\operatorname{argmin}} E(\boldsymbol{p}), \quad \mathbf{W}' = \begin{bmatrix} 1 & 0 & p_5 \\ 0 & 1 & p_6 \end{bmatrix}, \tag{7}$$

where the range of (p_5, p_6) to try is itself defined as a patch D with parameters $(x_D, y_D) = (x_T, y_T)$ but a different radius r_D. For the given test cases, the plots should look like in Figure 3. `main.m` gives you feedback on what the recovered (p_5, p_6) should be.

4 Part 3: Recovering the warp with KLT

As discussed in the lecture, brute force tracking works well, but has the disadvantage that it has a large space of possible $\mathbf{p}$ to explore. In the previous part, this space was reduced to two dimensions by only using two parameters for $\mathbf{p}$. But the size of the space grows exponentially with the parameters, and a general affine warp has six parameters!

Thankfully, it turns out that $E(\mathbf{p})$ is mostly smooth and locally convex around the optimal solution $E(\mathbf{p}^\star)$. This is illustrated for (p_5, p_6) aka $(\Delta x, \Delta y)$ on the right side of Figure 3. This local convexity suggests that we can use gradient descent to find $\boldsymbol{p}$, assuming that we start with an initial guess that is suffiently close to $\boldsymbol{p}^\star$. KLT uses the Gauss-Newton method for gradient descent. As seen in class, this is first formulated as finding an increment $\Delta\boldsymbol{p}$ to that initial guess $\boldsymbol{p}$ that minimizes the error E:

$$E(\boldsymbol{p}) = \sum_{(x,y)\in T} (I(W(x, y, \boldsymbol{p} + \Delta\boldsymbol{p})) - I_R(x, y))^2 \tag{8}$$

$$\frac{\partial}{\partial \Delta\boldsymbol{p}} \sum (I(W(x, y, \boldsymbol{p} + \Delta\boldsymbol{p})) - I_R(x, y))^2 = 0 \tag{9}$$

If we wanted to solve this directly, we would need to find a differentiable expression for $I(W(T, \boldsymbol{p} + \Delta\boldsymbol{p}))$. You are welcome to try, but you will find that this requires a pixel-wise case distinction. This would require much more computation than just finding $\boldsymbol{p}$ with the above brute force method. The Gauss-Newton algorithm avoids this problem by linearizing around $\boldsymbol{p}$, that is, making the assumption $I(W(T, \boldsymbol{p}+\Delta\boldsymbol{p})) \approx I(W(x, y, \boldsymbol{p})) + \frac{\partial I}{\partial \boldsymbol{p}}\Delta\boldsymbol{p}$ (first-order Taylor approximation). Using this approximation will result in an error, but solving the approximation iteratively should converge to the local minimum[2]. With this approximation, (9) becomes

$$\frac{\partial}{\partial \Delta\boldsymbol{p}} \sum_{(x,y)\in T} (I(W(x, y, \boldsymbol{p})) + \frac{\partial I(W(x, y, \boldsymbol{p}))}{\partial \boldsymbol{p}}\Delta\boldsymbol{p} - I_R(x, y))^2 = 0 \tag{10}$$

which we can now derive with respect to $\Delta\boldsymbol{p}$. For simplicity of expression, let us use vector calculus. We define the column vectors $\boldsymbol{e}(\boldsymbol{p}, \Delta\boldsymbol{p}), \boldsymbol{i}(\boldsymbol{p}), \boldsymbol{i}_R$ which are the vectorizations of the expression inside the square in (10) over all pixels in the template,$I(W(T, \vec{p}))$ and $I_R(T)$, respectively. Note, that vectors are written as bold lower case, matrices as bold upper case and scalars as lower case letters. We define:

$$\boldsymbol{e}(\boldsymbol{p}, \Delta\boldsymbol{p}) = \boldsymbol{i}(\boldsymbol{p}) + \frac{\partial \boldsymbol{i}(\boldsymbol{p})}{\partial \boldsymbol{p}}\Delta\boldsymbol{p} - \boldsymbol{i}_R : \tag{11}$$

Recall that a central component of vector calculus is the Jacobian matrix $\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}$, which expresses the derivative of a function $\boldsymbol{f}(\boldsymbol{x}) : \mathbb{R}^n \to \mathbb{R}^m$ as follows:

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \tag{12}$$

Note that $\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}$ is $(m \times n)$. It is conveniently defined such that $\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{A}\boldsymbol{x} \Rightarrow \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}} = \boldsymbol{A}$.

So, with (11), we can rewrite (10) as:

$$\frac{\partial}{\partial \Delta\boldsymbol{p}} \boldsymbol{e}(\boldsymbol{p}, \Delta\boldsymbol{p})^T \boldsymbol{e}(\boldsymbol{p}, \Delta\boldsymbol{p}) = 0. \tag{13}$$

Let us derive with respect to $\Delta\boldsymbol{p}$. Remember that $\vec{e}(\boldsymbol{p}, \Delta\boldsymbol{p})$ depends on $\Delta\boldsymbol{p}$, so we need to apply the chain rule:

$$\frac{\partial}{\partial \Delta\boldsymbol{p}} \boldsymbol{e}(\boldsymbol{p}, \Delta\boldsymbol{p})^T \boldsymbol{e}(\boldsymbol{p}, \Delta\boldsymbol{p}) = \frac{\partial}{\partial \boldsymbol{e}} \boldsymbol{e}(\boldsymbol{p}, \Delta\boldsymbol{p})^T \boldsymbol{e}(\boldsymbol{p}, \Delta\boldsymbol{p}) \cdot \frac{\partial \boldsymbol{e}(\boldsymbol{p}, \Delta\boldsymbol{p})}{\partial \Delta\boldsymbol{p}} \tag{14}$$

[2]See Newton's method for finding the root (zero-crossing) of one-dimensional functions for some intuitive understanding.

It is simple to show that $\frac{\partial}{\partial \boldsymbol{e}}\boldsymbol{e}(\boldsymbol{p},\Delta\boldsymbol{p})^T\boldsymbol{e}(\boldsymbol{p},\Delta\boldsymbol{p}) = 2\boldsymbol{e}(\boldsymbol{p},\Delta\boldsymbol{p})^T$. You should be able to show that with pen and paper by writing out the coefficients of $\boldsymbol{e}(\boldsymbol{p},\Delta\boldsymbol{p})^T\boldsymbol{e}(\boldsymbol{p},\Delta\boldsymbol{p})$ and then writing out the Jacobian $\frac{\partial}{\partial \boldsymbol{e}}$ for these coefficients. As for $\frac{\partial \boldsymbol{e}}{\partial \Delta\boldsymbol{p}}$, it is simply $\frac{\partial I}{\partial \boldsymbol{p}}$, as can be seen from (11). So we now have:

$$2\boldsymbol{e}(\boldsymbol{p},\Delta\boldsymbol{p})^T \cdot \frac{\partial \boldsymbol{i}(\boldsymbol{p})}{\partial \boldsymbol{p}} = 0 \tag{15}$$

Dividing by two and applying $(\boldsymbol{AB})^T = \boldsymbol{B}^T\boldsymbol{A}^T$ for later convenience, we get:

$$\frac{\partial \boldsymbol{i}(\boldsymbol{p})}{\partial \boldsymbol{p}}^T \cdot \boldsymbol{e}(\boldsymbol{p},\Delta\boldsymbol{p}) = \frac{\partial \boldsymbol{i}(\boldsymbol{p})}{\partial \boldsymbol{p}}^T (\boldsymbol{i}(\boldsymbol{p}) + \frac{\partial \boldsymbol{i}(\boldsymbol{p})}{\partial \boldsymbol{p}}\Delta\boldsymbol{p} - \boldsymbol{i}_R) = 0 \tag{16}$$

and can now proceed to solve for $\Delta\boldsymbol{p}$, by taking it out from the product, moving the other terms to the right side and eliminating the factor $\Delta\boldsymbol{p}$ is multiplied with:

$$\frac{\partial \boldsymbol{i}(\boldsymbol{p})}{\partial \boldsymbol{p}}^T (\boldsymbol{i}(\boldsymbol{p}) + \frac{\partial \boldsymbol{i}(\boldsymbol{p})}{\partial \boldsymbol{p}}\Delta\boldsymbol{p} - \boldsymbol{i}_R) = \underbrace{\frac{\partial \boldsymbol{i}(\boldsymbol{p})}{\partial \boldsymbol{p}}^T \frac{\partial \boldsymbol{i}(\boldsymbol{p})}{\partial \boldsymbol{p}}}_{\boldsymbol{H}(\boldsymbol{p})}\Delta\boldsymbol{p} + \frac{\partial \boldsymbol{i}(\boldsymbol{p})}{\partial \boldsymbol{p}}^T (\boldsymbol{i}(\boldsymbol{p}) - \boldsymbol{i}_R) = 0 \Leftrightarrow \tag{17}$$

$$\boldsymbol{H}(\boldsymbol{p})\Delta\boldsymbol{p} = \frac{\partial \boldsymbol{i}(\boldsymbol{p})}{\partial \boldsymbol{p}}^T (\boldsymbol{i}_R - \boldsymbol{i}(\boldsymbol{p})) \quad \Leftrightarrow \quad \Delta\boldsymbol{p} = \boldsymbol{H}(\boldsymbol{p})^{-1}\frac{\partial \boldsymbol{i}(\boldsymbol{p})}{\partial \boldsymbol{p}}^T (\boldsymbol{i}_R - \boldsymbol{i}(\boldsymbol{p})), \tag{18}$$

where we have introduced $\boldsymbol{H}(\boldsymbol{p})$, the so-called Hessian. With this derivation, you should be able to write the function trackKLT, which iteratively updates $\boldsymbol{p} \leftarrow \boldsymbol{p} + \Delta\boldsymbol{p}$ with the update rule we just derived. Start with a $\boldsymbol{p}$ which corresponds to an identity warp. Note that $\frac{\partial \boldsymbol{i}(\boldsymbol{p})}{\partial \boldsymbol{p}}$ can be further decomposed as seen in the class. Because $\boldsymbol{i}(\boldsymbol{p}) = \boldsymbol{i}(\boldsymbol{w}(\boldsymbol{p})), \boldsymbol{w}(\tilde{\boldsymbol{x}},\boldsymbol{p}) = \mathbf{W}\tilde{\boldsymbol{x}}$, you can use the chain rule: $\frac{\partial \boldsymbol{i}(\boldsymbol{p})}{\partial \boldsymbol{p}} = \frac{\partial \boldsymbol{i}(\boldsymbol{p})}{\partial \boldsymbol{w}} \cdot \frac{\partial \boldsymbol{w}(\tilde{\boldsymbol{x}},\boldsymbol{p})}{\partial \boldsymbol{p}} = \begin{bmatrix}\boldsymbol{i}_x(\boldsymbol{p}) & \boldsymbol{i}_y(\boldsymbol{p})\end{bmatrix} \frac{\partial}{\partial \boldsymbol{p}} W(x,y,\boldsymbol{p}) = \begin{bmatrix}\boldsymbol{i}_x(\boldsymbol{p}) & \boldsymbol{i}_y(\boldsymbol{p})\end{bmatrix} \begin{bmatrix}\Delta x & 0 & \Delta y & 0 & 1 & 0 \\ 0 & \Delta x & 0 & \Delta y & 0 & 1\end{bmatrix}$ where $\Delta x, \Delta y$ are the pixel coefficients relative to $\boldsymbol{x}_T$, see (5). Here, $\boldsymbol{i}_x(\boldsymbol{p}), \boldsymbol{i}_y(\boldsymbol{p})$ are nothing but the x- and y-gradients of the **warped** image patch, vectorized. This all sounds good until you try to implement it and realize that you can't simply implement it as a matrix multiplication, because $\Delta x, \Delta y$ have different values for different pixels, that is, different rows of $\begin{bmatrix}\boldsymbol{i}_x(\boldsymbol{p}) & \boldsymbol{i}_y(\boldsymbol{p})\end{bmatrix}$. So, to construct $\frac{\partial \boldsymbol{i}(\boldsymbol{p})}{\partial \boldsymbol{p}}$, you will need to multiply each row of $\begin{bmatrix}\boldsymbol{i}_x(\boldsymbol{p}) & \boldsymbol{i}_y(\boldsymbol{p})\end{bmatrix}$ with a numerically different $\frac{\partial \boldsymbol{w}(\tilde{\boldsymbol{x}},\boldsymbol{p})}{\partial \boldsymbol{p}}$. Hints:

- Use `getWarpedPatch` to get I and I_R, then obtain $\boldsymbol{i}(\boldsymbol{p})$ and $\boldsymbol{i}_R$ by vectorizing the matrices.
- Similarly, it is convenient to obtain the patch gradients by convolving I first, and vectorizing the patch later. For convenience, you can get the image gradients by convolving with $\begin{bmatrix}1 & 0 & -1\end{bmatrix}$ rather than with $\begin{bmatrix}1 & -1\end{bmatrix}$. Note that this convolution, if applied validly, will reduce the patch size by one pixel on each side. To make sure that the gradient is defined on the full patch, use as input to the convolutions an accordingly larger patch.
- A good first step when debugging is to check the whether the shapes of your intermediate products make sense. For example, in (18) the shapes of matrices $\boldsymbol{H}(\boldsymbol{p})^{-1}, \frac{\partial \boldsymbol{i}(\boldsymbol{p})}{\partial \boldsymbol{p}}^T, (\boldsymbol{i}_R - \boldsymbol{i}(\boldsymbol{p}))$ should be, respectively, $(6\times 6), (6 \times |T|), (|T| \times 1)$, where $|T|$ is the amount of pixels in the patch.
- You should be able to write down convenient intermediate products (try to avoid redundant calculations!) from the above derivation. If you struggle, consult the lecture notes.
- For debugging and visualization, we recommend to plot $I_R, I, (I_R - I)$, the warped image gradients I_x, I_y and the steepest descent images $\nabla I \frac{\partial W}{\partial \vec{p}}$, which you can obtain by reshaping each column of $\frac{\partial \vec{i}}{\partial \vec{p}}$ into an image patch. See Figure 4 and https://youtu.be/3g-zEHoVwvM for how these look like in our implementation.

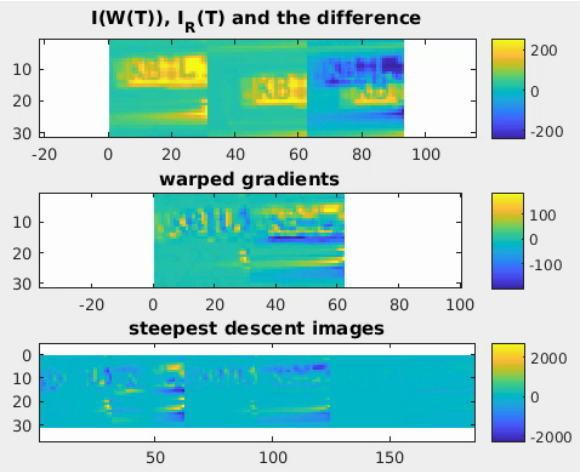

Figure 4: Visualization of KLT gradient descent for the very first iteration. See https://youtu.be/3g-zEHoVwvM for the video.

5 Part 4: Applying KLT to KITTI

You should now be able to run the next section, in which we take the keypoint tracker and apply it to the KITTI dataset. This will work much better if you have implemented bilinear interpolation in `getWarpedPatch`. Note that we have downsampled the images by a factor of 4. This ensures that a sufficient amount of them are tracked; with a higher resolution, the motion of too many points is outside of the convergence basin of KLT. As you should see, some points are still not correctly tracked. Next, we will implement a simple method to discard points which are not correctly tracked.

6 Part 5: Outlier rejection with the bidirectional error

The bidirectional error check is a simple test to verify that a point is consistently tracked. Let us define the result of tracking a point $\boldsymbol{x}_T$ from image I_i to image I_{i+1} with KLT as

$$\Psi_i^{i+1}(\boldsymbol{x}_T) = \boldsymbol{x}_T + \begin{bmatrix} p_5 \\ p_6 \end{bmatrix}. \tag{19}$$

Then, the bidirectional error test verifies that when we try to track the point back to I_i, we are close to $\boldsymbol{x}_T$ again:

$$\Psi_{i+1}^{i}(\Psi_i^{i+1}(\boldsymbol{x}_T)) - \boldsymbol{x}_T < \lambda \tag{20}$$

Implement `trackKLTRobustly` which encapsulates $\Psi_i^{i+1}(\boldsymbol{x}_T)$ and also indicates whether the tracking has passed the bidirectional error test. Then, when you run the corresponding section of main.m, you should get something like in Figure 1 and https://youtu.be/iaRPafeG9zw .