CV202, HW #7

Oren Freifeld and Meitar Ronen
The Department of Computer Science, Ben-Gurion University

Release Date: 3/5/2020
Submission Deadline: 23/6/2020, 20:00

Abstract

Please make sure you have carefully read the general instructions in the course website at https://www.cs.bgu.ac.il/~cv202/HW_Instructions. These instructions address, among other things, what to do or not do with figures, Jupyter Notebooks, *etc.*, as well as what compression files are legit or not.

This assignment starts by briefly touching upon the topic of image warping and then shifts the focus to optical flow. The data file for this assignment is the same file from HW 2:
https://www.cs.bgu.ac.il/~cv202/wiki.files/hw2_data.tar.xz.

Some of the exercises in this assignment are marked as optional; *i.e.*, you do not have to submit them to get 100 in the assignment, and if you submit them, it will not raise your grade.
In some of the computer exercises in this assignment you will need to solve a linear system of the form $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ where $\boldsymbol{A}$ is square. When implementing it in Python, do not invert the matrix as this is usually a bad idea in this context; rather, solve the system using *scipy.linalg.solve* or *scipy.linalg.lstsq* (or *scipy.sparse.linalg.lsqr*, if the matrix is large and sparse but just make sure first that your $\boldsymbol{A}$ uses a sparse data structure; see more details below).

Version Log

- 1.00, 3/5/2020. Initial release.

Contents

1 Image Warping

Suppose we have a nominal optical flow, $\boldsymbol{u}(\cdot) = (u(\cdot), v(\cdot))$; *i.e.* at every pixel $\boldsymbol{x} = (x, y)$, suppose we know the value of $\boldsymbol{u}(\boldsymbol{x}) \in \mathbb{R}^2$. Given an image I, we would like create (on a computer) the so-called warped image,

$$I_{\text{warpped}}(x', y') = I(x, y) \qquad \text{where } x' = x + u(\boldsymbol{x}) \text{ and } y' = y + v(\boldsymbol{x})\,. \tag{1}$$

The equation above assumes a continuous setting; we, however, care about digital images. Usually, even if x and y are integers, $x+u(x,y)$ and/or $y+v(x,y)$ are not. On a computer, when we create the image that contains the values of I_{warpped}, we need to define the values of that image in integral locations. The solution involves an interpolation, *e.g.*, a bilinear one.

Definition 1 (bilinear interpolation) *Let f be a function from $\mathbb{R}^2$ to $\mathbb{R}$. Suppose we know the values of f at integral locations, and want to find $f(x, y)$ where $x \notin \mathbb{Z}$ and/or $y \notin \mathbb{Z}$. The so-called bilinear interpolation approximates $f(x, y)$ via a calculation that uses the values of f at the (integral) corners of a square containing (x, y). Particularly, if we choose a coordinate system in which the four points where f is known are (0, 0), (0, 1), (1, 0), and (1, 1), then*

$$f(x, y) \approx f(0, 0)(1 - x)(1 - y) + f(1, 0)x(1 - y) + f(0, 1)(1 - x)y + f(1, 1)xy\,. \tag{2}$$

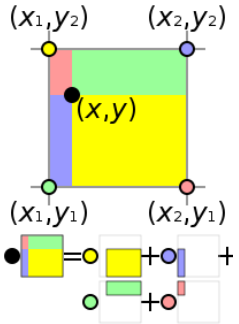

Figure 1: Bilinear interpolation (figure source: Wikipedia).

Note that this approximation is a polynomial, from $\mathbb{R}^2$ to $\mathbb{R}$, of the form

$$a_{00} + a_{10}x + a_{01}y + a_{11}xy = \sum_{i=0}^{1}\sum_{j=0}^{1} a_{ij}x^i y^j\,. \tag{3}$$

♢

In image warping, however, the original setting is somewhat reversed, complicating the usage of interpolation techniques such as bilinear interpolation. Consider images with N_{rows} rows and N_{cols} columns. The locations

$$(x + u(\boldsymbol{x}), y + v(\boldsymbol{x})) \qquad \forall x \in \{1, \ldots, N_{\text{rows}}\} \text{ and } \forall y \in \{1, \ldots, N_{\text{cols}}\}\,, \tag{4}$$

usually fall in non-integral locations, while the values we want to find,

$$I_{\text{warpped}}(\tilde{x}, \tilde{y}) \qquad \forall \tilde{x} \in \{1, \ldots, N_{\text{rows}}\} \text{ and } \forall \tilde{y} \in \{1, \ldots, N_{\text{cols}}\}\,, \tag{5}$$

are placed in integral locations. This means we need to interpolate values at integral locations from values at non-integral locations. Such interpolation methods exist, but are harder, cumbersome and often problematic. Thus, the easiest and standard way to do warping is, for every pixel $(\tilde{x}, \tilde{y})$ in I_{warpped}, to see "where it came from" and then interpolate between the values of the pixes in the original image that are around that location. In other words, let $T : \mathbb{R}^2 \to \mathbb{R}^2$ be defined by $T : (x, y) \mapsto (x + u(x, y), y + v(x, y)$. Thus, $I_{\text{warpped}} = I \circ T$. We define the value of $I_{\text{warpped}}(\tilde{x}, \tilde{y})$ as

$$(I \circ T^{-1})(\tilde{x}, \tilde{y}) \triangleq I(T^{-1}(\tilde{\boldsymbol{x}}))\,,$$

where $\tilde{\boldsymbol{x}} = (\tilde{x}, \tilde{y})$. While $(T^{-1}(\tilde{\boldsymbol{x}}))$ is usually not an integral location, we are now back in the setting of standard interpolation so we can interpolate between the values of I in the integral locations around $T^{-1}(\tilde{x}, \tilde{y})$. One catch is that T might not be invertible; *e.g.*, the flow may send both pixels to the same location. Another problem is that even if it is invertible, we may not know how to invert it. The most popular practical "solution" to these issues is to ignore both of them and, pretending the negative flow is a good approximation of the inverse map, just set $T^{-1}(\tilde{x}, \tilde{y}) \approx (\tilde{x} - u(\tilde{x}, \tilde{y}), \tilde{y} - v(\tilde{x}, \tilde{y}))$. If a mathematician catches you doing it, just change the topic of the conversation.

Remark 1 *Alternatively, instead of warping I_1 toward I_2, we can warp I_2 toward I_1; in other words, use the known values of T as the values of $(T^{-1})^{-1}$ which are required in order to create $I_2 \circ T^{-1}$.* ♢

Before we continue, familiarize yourself first with numpy's `mgrid` (the Matlab equivalent is meshgrid); *e.g.*:

```
In [1]: from numpy import mgrid
In [2]: yy,xx=mgrid[0:3,0:4]
In [3]: xx
Out[3]:
array([[0, 1, 2, 3],
       [0, 1, 2, 3],
       [0, 1, 2, 3]])

In [4]: yy
Out[4]: array([[0, 0, 0, 0],
       [1, 1, 1, 1],
       [2, 2, 2, 2]])
```

The following is from OpenCV Doc for cv2.remap: (with some mild edits and omissions:

```
cv2.remap(src, map1, map2, interpolation,dst,...)
src: Source image (e.g., I1)
dst: Destination image (e.g., I1 ∘ T)
map1: x values after applying T^-1 (e.g., xx − u),
map2: y values after applying T^-1 (e.g., yy − v),
interpolation: Interpolation method
```

Remark 2 *Importantly, opencv expects map_1 and map_2 to have dtype=np.float32. So,* e.g.*, if you pass xx as map_1, and xx.dtype is np.int, as is the case for example if it was defined via yy,xx=mgrid[0:3,0:4], then you will get an error message. So make sure you do proper casting beforehand as needed.* ♢

There are many standard implementations for image warping. For example, the function cv2.remap transforms the source image using the specified map: $\texttt{dst}[y,x] = \texttt{dst}(x,y) = \texttt{src}(\texttt{map}_1(x,y), \texttt{map}_2(x,y))$ where values of pixels with non-integer coordinates are computed using one of the available interpolation methods.

Notation 1 *The convention used above is that $I(x,y)$ (round parentheses where x is the first coordinate) is used for "math" and $I[y,x]$ (square brackets where y is the first coordinate) is used for "code". That said, we won't always be that religious about this distinction, and might mix the notation using the legit excuse that the context should make it clear what we mean. Deal with it. That's what happens in literature as well. Deal with that too.* ♢

Computer Exercise 1 *Write your own function for bilinear interpolation.* ♢

Computer Exercise 2 *Get img1, u and v from* `imgs_for_optical_flow.mat`*. Use your bilinear interpolation function to warp img1 using the (u,v) flow. Show your result along with the result obtained by using a standard implementation of image warping;* e.g.*, OpenCV's remap function.* ♢

Of course, if the flow is invertible and the inverse is known and has a closed form, we had better use it explicitly. For example, this is the case for affine flow, provided it is also invertible (recall that an affine map might fail to have an inverse).

Computer Exercise 3 *Get img1 from* `imgs_for_optical_flow.mat`*. Use your bilinear interpolation function to warp img1 using the (invertible) affine flow:*

$$\boldsymbol{u}(\boldsymbol{x}) = \begin{bmatrix} 0.5 & 0.2 & 0 \\ 0 & 0.5 & 8 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Show your result along with the result obtained by using a standard implementation of affine image warping; e.g.*, OpenCV's cv2.warpAffine function (note that this function takes the inverse map as its input).* ♢

From now on, you are free to use standard implementations of image warping (which are probably faster then yours, and usually also offer additional and better methods for interpolation; *e.g.*, cubic splines). So you do not have to use your own implementation in the following exercises.

2 Optical Flow

Problem 1 (the connection between a quadratic error and a Gaussian likelihood model) *Show that*

$$x, \mu \in \mathbb{R} \quad \sigma > 0 \quad x \sim \mathcal{N}(\mu, \sigma^2) \Rightarrow \arg\min_{\mu} (x-\mu)^2 = \arg\max_{\mu} \mathcal{N}(x; \mu, \sigma^2) \quad (6)$$

where

$$\mathcal{N}(x;\mu,\sigma^2) \triangleq \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2}\left(\frac{(x-\mu)^2}{\sigma^2}\right)\right). \tag{7}$$

♢

Problem 2 (normal flow) *In the notation from the lecture, show that*

$$\boldsymbol{u}_{\text{normal}} = \frac{-I_t}{I_x^2+I_y^2}\begin{bmatrix} I_x \\ I_y \end{bmatrix} = \frac{-I_t}{\|\nabla_{\boldsymbol{x}} I\|^2}\begin{bmatrix} I_x \\ I_y \end{bmatrix} \tag{8}$$

and

$$\|\boldsymbol{u}_{\text{normal}}\| = \frac{|I_t|}{\|\nabla_{\boldsymbol{x}} I\|}. \tag{9}$$

♢

The gradient-constraint equation for gray-scale 2D images is

$$\boxed{I_x(\boldsymbol{x},t)u(\boldsymbol{x}) + I_y(\boldsymbol{x},t)v(\boldsymbol{x}) + I_t(\boldsymbol{x},t) = 0}. \tag{10}$$

Problem 3 (optical flow for RGB images) *Consider the RGB case;* i.e.*,* $I(\boldsymbol{x},t) = \begin{bmatrix} R(\boldsymbol{x},t) & G(\boldsymbol{x},t) & B(\boldsymbol{x},t) \end{bmatrix}^T$*. Suggest an analogous gradient-constraint equation for optical flow between RGB images. Explain under what conditions the new equation will have: 1) infinitely-many solutions (as we had in the 2D case – recall,* e.g.*, our discussion on normal flow); 2) a unique solution; 3) no solution. Also, draw examples of the different cases in the uv plane.* ♢

In the 2D case, for gray-scale images, a simple per-pixel cost function that penalizes deviations from the gradient-constraint equation is

$$\varepsilon^2(u(\boldsymbol{x}),v(\boldsymbol{x})) = (I_x(\boldsymbol{x},t)u(\boldsymbol{x}) + I_y(\boldsymbol{x},t)v(\boldsymbol{x}) + I_t(\boldsymbol{x},t))^2. \tag{11}$$

Problem 4 *Suggest an analogous cost function for the RGB case. Is your suggested function convex? Explain.* ♢

Problem 5 (3D optical flow) *Derive the gradient-constraint equation for optical flow between volumetric images (*e.g.*, CT images);* i.e.*, the image "sequence" is now* $I(x,y,z,t)$*.* ♢

Problem 6 *Let* $X \sim \mathcal{N}(\mu,\sigma^2)$ *and let* $Y = aX+b$ *where* $a > 0$ *and* $b \in \mathbb{R}$*. Show that* $p_Y(y) = \left|\frac{1}{a}\right| p_X\left(\frac{y-b}{a}\right)$ *(a standard application of the change of variables formula*[1]*) is consistent with the fact that* $Y \sim \mathcal{N}(a\mu+b, a^2\sigma^2)$*.* ♢

Problem 7 *Let* Y *be a random variable such that* $y = \mu+\varepsilon$ *where* μ *is constant and* ε *is a realization (*i.e.*, a sample) from a zero-mean Gaussian noise with variance* σ^2*. How is* Y *distributed?* ♢

Problem 8 *Let us write the gradient-constraint equation as*

$$\mu \triangleq I_x(\boldsymbol{x},t)u_x + I_y(\boldsymbol{x},t)u_y = -I_t(\boldsymbol{x},t).$$

[1] Look for change of variables in https://en.wikipedia.org/wiki/Probability_density_function

Consider the LHS, μ, as an unknown deterministic scalar which depends on the two deterministic (but unknown) parameters, $u(\boldsymbol{x},t)$ and $v(\boldsymbol{x},t)$. Since we do not expect the equation to hold perfectly, let us treat the RHS, $-I_t(\boldsymbol{x},t)$, as a noisy observation of μ, under the assumption of an additive Gaussian noise: $-I_t(\boldsymbol{x},t) = I_x(\boldsymbol{x},t)u_x + I_y(\boldsymbol{x},t)u_y + \varepsilon = \mu + \varepsilon$ where

$$p(\varepsilon;\sigma^2) = \mathcal{N}(\varepsilon;0,\sigma^2) \triangleq \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2}\frac{\varepsilon^2}{\sigma^2}\right) \qquad \sigma > 0\,.$$

Under these assumptions, how is $-I_t(\boldsymbol{x},t)$ distributed? ♢

For the following computer exercises, get img1, img2, img3, img4, img5, and img6 from `imgs_for_optical_flow.mat`. We will consider five pairs of frames:

pair #1: I_1=img1 and I_2=img2;

pair #2: I_1=img1 and I_3=img3;

pair #3: I_1=img1 and I_4=img4;

pair #4: I_1=img1 and I_5=img5;

pair #5: I_1=img1 and I_6=img6.

In HW 4, you already computed the spatial derivatives of I_1.

Computer Exercise 4 *For each one of the pairs, compute and display the (approximated) temporal derivatives by simply subtracting I_1 from each one of the other images.* ♢

In what follows we assume images of N_{rows} rows and N_{cols} columns.

2.1 Coarse-to-fine and/or Iterative Optical-flow Estimation

In this section there are no exercises nor computer exercises, just a detailed explanation how to do coarse-to-fine and/or iterative optical-flow Estimation. This explanation may come handy in the computer exercises in the next section in case you will want to use one of these options.

Suppose we want to do a coarse-to-fine HS (for now, without iterations) or a coarse-to-fine LK (without iterations). Suppose we have L=3 levels and I_1 and I_2 are the original images. Suppose the original images are of size 100x100. At level 3, we work with 25x25 images/flows. At level 2, we work with 50x50 images/flows. At level 1, we work with 100x100 images/flows. Then, a coarse-to-fine approach means we do the following:

- Set $I_1^{[1]} = I_1$ and $I_2^{[1]} = I_2$ (where here the bracketed superscripted index denotes the pyramid level).
- Set $I_1^{[2]}$ = downsampled-by-2 version of $I_1^{[1]}$.
- Set $I_2^{[2]}$ = downsampled-by-2 version of $I_2^{[1]}$.
- Set $I_1^{[3]}$ = downsampled-by-2 version of $I_1^{[2]}$.

- Set $I_2^{[3]}$ = downsampled-by-2 version of $I_2^{[2]}$.
- Compute $I_t^{[3]}$ using $I_1^{[3]}$ and $I_2^{[3]}$.
- Compute the spatial derivatives using $I_1^{[3]}$.
- Using these derivatives, compute the flow (either HS or LK, depending what we want) between $I_1^{[3]}$ and $I_2^{[3]}$. Call it $u^{[3]}$ and $v^{[3]}$. Note that (in the example here) $I_1^{[3]}$, $I_2^{[3]}$, $u^{[3]}$ and $v^{[3]}$ are 25x25 arrays.
- Upsample $u^{[3]}$ and $v^{[3]}$ to make them 50x50. Use their upsampled version to warp $I_1^{[2]}$ (which is 50x50 in my example here).
- Compute $I_t^{[2]}$ using (i) the warped $I_1^{[2]}$ and (ii) $I_2^{[2]}$.
- Compute the spatial derivatives using the warped $I_1^{[2]}$.
- Using these derivatives, compute the flow (denoted here as $\delta u^{[2]}$ and $\delta v^{[2]}$; these are 50x50) between (i) the warped $i_1^{[2]}$ and (ii) $I_2^{[2]}$.
- Set $u^{[2]} = \delta u^{[2]}$ + upsampled version of $u^{[3]}$
- Set $v^{[2]} = \delta v^{[2]}$ + upsampled version of $v^{[3]}$
- Upsample $u^{[2]}$ and $v^{[2]}$ to make them 100x100. Use their upsampled version to warp $I_1^{[1]}$ (which is 100x100 in my example here).
- Compute $I_t^{[1]}$ using (i) the warped $I_1^{[1]}$ and (ii) $I_2^{[1]}$.
- Compute the spatial derivatives using the warped $I_1^{[1]}$. Using these derivatives, compute the flow (denoted here as $\delta u^{[1]}$ and $\delta v^{[1]}$; these are 100x100) between (i) the warped $I_1^{[1]}$ and (ii) $I_2^{[1]}$.
- Set $u^{[1]} = \delta u^{[1]}$ + upsampled version of $u^{[2]}$.
- Set $v^{[1]} = \delta v^{[1]}$ + upsampled version of $v^{[2]}$.
- $u^{[1]}$ and $v^{[1]}$ are the final estimate (which you can use to warp the original I_1 toward the original I_2 (of course, at each level you can also add iterations as we mentioned when we discussed the iterative version of LK).

Regarding the iterative version: You can either predefine the number of iterations, or use some threshold on the difference between the estimated flow in two consecutive iterations to declare convergence.

2.2 Horn-and-Schunck Optical Flow

The Horn-and-Schunck cost function (in its discrete from – which is the only form of it we discussed) is:

$$\begin{aligned}
E(u,v,I_1,I_2) &= E_{\text{data}}(u,v,I_1,I_2) + \lambda E_{\text{smoothness}}(u,v) \\
&= \left(\sum_{i,j} (I_x(i,j,t)u_{i,j} + I_y(i,j,t)v_{i,j} + I_t(i,j,t))^2 \right) \\
&+ \lambda \sum_{i,j} \left[(u_{i,j} - u_{i+1,j})^2 + (u_{i,j} - u_{i,j+1})^2 + (v_{i,j} - v_{i+1,j})^2 + (v_{i,j} - v_{i,j+1})^2 \right] \\
&= \sum_{i,j} (I_x(i,j,t)u_{i,j} + I_y(i,j,t)v_{i,j} + I_t(i,j,t))^2 \\
&+ \lambda \left[(u_{i,j} - u_{i+1,j})^2 + (u_{i,j} - u_{i,j+1})^2 + (v_{i,j} - v_{i+1,j})^2 + (v_{i,j} - v_{i,j+1})^2 \right]
\end{aligned} \tag{12}$$

where $\lambda > 0$ controls the tradeoff, and we use (i,j) to denote pixel locations in order to emphasize the discrete nature of the domain. For computing the gradient, the derivatives are (for a pixel not on the border of the image):

$$\frac{\partial E_{\text{data}}}{\partial u_{i,j}} = 2(I_x(i,j,t)u_{i,j} + I_y(i,j,t)v_{i,j} + I_t(i,j,t))I_x(i,j,t) \tag{13}$$

$$\frac{\partial E_{\text{data}}}{\partial v_{i,j}} = 2(I_x(i,j,t)u_{i,j} + I_y(i,j,t)v_{i,j} + I_t(i,j,t))I_y(i,j,t) \tag{14}$$

$$\frac{\partial E_{\text{smoothness}}}{\partial u_{i,j}} = 2\lambda(4u_{i,j} - (u_{i+1,j} + u_{i,j+1} + u_{i-1,j} + u_{i,j-1})) \tag{15}$$

$$\frac{\partial E_{\text{smoothness}}}{\partial v_{i,j}} = 2\lambda(4v_{i,j} - (v_{i+1,j} + v_{i,j+1} + v_{i-1,j} + v_{i,j-1})) \tag{16}$$

Problem 9 *Complete the details: assuming one-based indexing, write the gradient terms for a pixel on the boundary;* i.e.*, where $i = 1$ or $j = 1$ or $i = N_{rows}$ or $j = N_{cols}$; do not forget to include corner cases: ($i = 1$,$j = 1$), ($i = 1, j = N_{cols}$), ($i = N_{rows}, j = 1$), and ($i = N_{rows}, j = N_{cols}$).* ♢

Problem 10 *(i) What $(u(\cdot), v(\cdot))$ will minimize E_{data}? Note that at the minimizer, $\frac{\partial E_{data}}{\partial u_{i,j}}$ and $\frac{\partial E_{data}}{\partial v_{i,j}}$ are zero.*

(ii) What $(u(\cdot), v(\cdot))$ will minimize $E_{smoothness}$? Again, note that at a minimizer, $\frac{\partial E_{smoothness}}{\partial u_{i,j}}$ and $\frac{\partial E_{smoothness}}{\partial v_{i,j}}$ are zero. Show that the minimizer is not unique. ♢

To make the notation more compact, let a single index, s (short for site), denote a generic (i,j) pair, and, for $s = (i,j)$, let $s \sim s'$ denote the fact that s' is a neighbor of s according to 4-connectivity; *i.e.*,

$$s' \in \{(i+1,j), (i-1,j), (i,j+1), (i,j-1)\}. \tag{17}$$

We now rewrite the cost function, while also dropping the dependency on t:

$$E(u, v, I_1, I_2)$$
$$= \sum_s \left[(I_x(s)u(s) + I_y(s)v(s) + I_t(s))^2 + \lambda \sum_{s':s\sim s'} \left[(u(s) - u(s'))^2 + (v(s) - v(s'))^2 \right] \right] \tag{18}$$

where the summation $\sum_{s':s'\sim s}$ is done over the (usually 4) neighbors of s, and the double counting in that summation is accounted for by adjusting λ by a factor of 2.

Problem 11 *Consider an $n \times n$ lattice with 4-connectivity (i.e., North, South, East, West). Let $f : \mathbb{R}^{n\times n} \to \mathbb{R}$ be defined by*

$$f(u) = \sum\nolimits_s \sum\nolimits_{s':s'\sim s} (u(s) - u(s'))^2 \ . \tag{19}$$

Show that, for every s (regardless if the pixel s is on border of the image or not),

$$\frac{\partial}{\partial u(s)} f(u) = 4 \sum_{s':s'\sim s} (u(s) - u(s')) \, . \tag{20}$$

$\Diamond$

At every pixel s, not on the border of the image:

$$\frac{\partial}{\partial u(s)} E = 2(I_x^2(s)u(s) + I_x(s)I_y(s)v(s) + I_x(s)I_t(s)) + 2\lambda \left(2 \sum_{s':s\sim s'} (u(s) - u(s')) \right) \tag{21}$$

$$\frac{\partial}{\partial v(s)} E = 2(I_y(s)I_x(s)u(s) + I_y^2(s)v(s) + I_y(s)I_t(s)) + 2\lambda \left(2 \sum_{s':s\sim s'} (v(s) - v(s')) \right) \tag{22}$$

which is of course similar to what we had before.

To get critical points, we set the gradient to zero (so can ignore the 2) and obtain the **normal equations**:

$$I_x^2(s)u(s) + I_x(s)I_y(s)v(s) + I_x(s)I_t(s) + 2\lambda \sum_{s':s\sim s'} (u(s) - u(s')) = 0 \tag{23}$$

$$I_y(s)I_x(s)u(s) + I_y^2(s)v(s) + I_y(s)I_t(s) + 2\lambda \sum_{s':s\sim s'} (v(s) - v(s')) = 0 \, . \tag{24}$$

Problem 12 *Complete the details: assuming one-based indexing, write the gradient terms for a pixel s on the boundary.* $\Diamond$

This leads to a very large, but also very sparse, linear system of the form $\boldsymbol{A}\boldsymbol{\xi} = \boldsymbol{b}$. If $N = N_{\text{cols}} \times N_{\text{rows}}$ is the number of pixels, then $\boldsymbol{A}$ is $2N \times 2N$, while $\boldsymbol{\xi}$ and $\boldsymbol{b}$ are $2N \times 1$. For example, suppose we use the following ordering:

$$\boldsymbol{\xi} = \left[\, u(1,1) \ v(1,1) \ u(1,2) \ v(1,2) \ \ldots \ u(2,1) \ v(2,1) \ u(2,2) \ v(2,2) \ \ldots \ u(N_{\text{rows}},1) \ v(N_{\text{rows}},1) \ u(N_{\text{rows}},2) \ v(N_{\text{rows}},2) \ \ldots \, \right]^T \ . \tag{25}$$

Then (using 1-based indexing), for a pixel $s = (i, j)$ not on the border of the image, with $i_s = (i-1) \times N_{\text{cols}} + j$ denoting the linear index of s, the row of $\boldsymbol{A}$ that corresponds to $u(s)$, *i.e.*, row $\#(2i_s - 1)$, is

$$\left[\dots 0 \underbrace{-2\lambda}_{2i_s-2N_{\text{cols}}-1} \underbrace{0}_{2i_s-2N_{\text{cols}}} \dots \underbrace{-2\lambda}_{2i_s-3} \underbrace{0}_{2i_s-2} \underbrace{(I_x^2(s)+8\lambda)}_{2i_s-1} \underbrace{I_x(s)I_y(s)}_{2i_s} \underbrace{-2\lambda}_{2i_s+1} \underbrace{0}_{2i_s+2} \dots \underbrace{-2\lambda}_{2i_s+2N_{\text{cols}}-1} \underbrace{0}_{2i_s+2N_{\text{cols}}} \dots \right] \tag{26}$$

while the row of $\boldsymbol{A}$ that corresponds to $v(s)$, *i.e.*, row $\#(2i_s)$, is

$$\left[\dots 0 \underbrace{0}_{2i_s-2N_{\text{cols}}-1} \underbrace{-2\lambda}_{2i_s-2N_{\text{cols}}} \dots \underbrace{0}_{2i_s-3} \underbrace{-2\lambda}_{2i_s-2} \underbrace{I_x(s)I_y(s)}_{2i_s-1} \underbrace{(I_y^2(s)+8\lambda)}_{2i_s} \underbrace{0}_{2i_s+1} \underbrace{-2\lambda}_{2i_s+2} \dots \underbrace{0}_{2i_s+2N_{\text{cols}}-1} \underbrace{-2\lambda}_{2i_s+2N_{\text{cols}}} \dots \right] \tag{27}$$

where all the other entries of these two rows equal to zero.

Problem 13 *Complete the details: assuming one-based indexing, write the rows of $\boldsymbol{A}$ that correspond to pixels on the border of the image;* i.e., *where $i = 1$ or $j = 1$ or $i = N_{rows}$ or $j = N_{cols}$; do not forget to include corner cases: ($i = 1, j = 1$), ($i = 1, j = N_{cols}$), ($i = N_{rows}, j = 1$), and ($i = N_{rows}, j = N_{cols}$).* ♢

Finally:

$$\boldsymbol{b} = \left[\dots \underbrace{-I_x(s)I_t(s)}_{2i_s-1} \underbrace{-I_y(s)I_t(s)}_{2i_s} \dots \right]^T . \tag{28}$$

Computer Exercise 5 *For each one of the five frame pairs, construct $\boldsymbol{A}$ and $\boldsymbol{b}$ (use a sparse matrix for $\boldsymbol{A}$) and, using a sparse linear solver, solve for $\boldsymbol{\xi}$. Display the estimated flows, and the results of warping I_1 using these flows. Repeat this with several different values of λ and visually explore the effect of λ on the resulting flows.* ♢

`Python commands you may find useful for the exercise above:`
`scipy.linalg.lstsq; scipy.sparse.linalg.lsqr.`

Computer Exercise 6 (this exercise is optional) *Repeat the exercise above, but do this in a coarse-to-fine manner, possibly with iterations at each level. After estimating the flow in each level, and before propagating it to the next finer level, you may want to apply median filtering to the estimated flow.* ♢

`Python commands you may find useful for the exercise above:`
`cv2.pyrDown; cv2.pyrUp.`

Computer Exercise 7 (this exercise is optional) *Repeat the previous exercise but this time do the robust version using Geman-McClure function and Iterative Reweighted Least Squares.* ♢

There is an obvious connection between the HS smoothness term, and Gaussian MRF prior; the former being the minus log of the latter. Note, however, that the corresponding Gaussian prior is what is called an improper prior; *i.e.*, integrating it gives ∞, so it cannot be normalized. In other words, the variance

is ∞. That situation is ok, and arises often in Bayesian analysis. The important thing here is that the posterior is (usually) a proper one. Particularly in the context of the HS prior, the prior is improper since $\boldsymbol{u}$ enters the expression of the prior only via differences of its components. To get an idea what is going on here, let us consider a smaller and simpler problem instead.

Problem 14 *Part (i) Find* $\boldsymbol{A} \in \mathbb{R}^{2\times 3}$ *such that*

$$\begin{bmatrix} y-x \\ z-y \end{bmatrix} = \boldsymbol{A} \begin{bmatrix} x \\ y \\ z \end{bmatrix} . \tag{29}$$

Part (ii) Find a 3×3 *symmetric matrix* $\boldsymbol{Q}$ *such that*

$$(y-x)^2 + (z-y)^2 = \left\| \begin{bmatrix} y-x \\ z-y \end{bmatrix} \right\|_{\ell_2}^2 = \begin{bmatrix} x & y & z \end{bmatrix} \boldsymbol{Q} \begin{bmatrix} x \\ y \\ z \end{bmatrix} . \tag{30}$$

Part (iii) By inspection of its eigenvalues, show that $\boldsymbol{Q}$ *is a symmetric positive-semidefinite (SPSD) matrix but not a symmetric positive-definite (SPD) matrix. Thus,* $\boldsymbol{Q}$ *is singular.*

With such a $\boldsymbol{Q}$*, it follows that*

$$\int_x \int_y \int_z \exp\left(-0.5 \begin{bmatrix} x & y & z \end{bmatrix} \boldsymbol{Q} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \right) = \infty \ . \tag{31}$$

$\Diamond$

Interpreting $\boldsymbol{Q}$ from the last problem as a precision matrix ("inverse covariance"), it follows that the covariance matrix has one of its eigenvalues equals to ∞. For example,

$$\exp\left(-0.5 \begin{bmatrix} x & y & z \end{bmatrix} \boldsymbol{Q} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \right)$$

is a positive constant along the 3D line $x = y = z$. Since this line is of infinite length, the variance of each of the three variables is ∞. However, if we view this function as a prior (even if as an improper one), we note that once we have some measurements, *e.g.*, $x' \sim \mathcal{N}(x, 1)$ and $y' \sim \mathcal{N}(y, 1)$, then the posterior becomes proper (*i.e.*, it can be integrated to one):

$$p(x, y, z | x', y') \propto \tag{32}$$

$$\exp\left(-0.5 \begin{bmatrix} x'-x & y'-y \end{bmatrix} \begin{bmatrix} x'-x \\ y'-y \end{bmatrix} \right) \exp\left(-0.5 \begin{bmatrix} x & y & z \end{bmatrix} \boldsymbol{Q} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \right) , \tag{33}$$

which is nothing more than the pdf of a random Gaussian random vector. In other words, we can find an SPD matrix $\boldsymbol{Q'}$, and three real scalars (μ_x, μ_y, μ_z), such that

$$p(x, y, z | x', y') \propto \exp\left(-0.5 \begin{bmatrix} x-\mu_x & y-\mu_y & z-\mu_z \end{bmatrix} \boldsymbol{Q'} \begin{bmatrix} x-\mu_x \\ y-\mu_y \\ z-\mu_z \end{bmatrix} \right) . \tag{34}$$

2.3 Lucas-and-Kanade Optical Flow

2.3.1 The Lucas-Kanade Matrix and the Aperture Problem

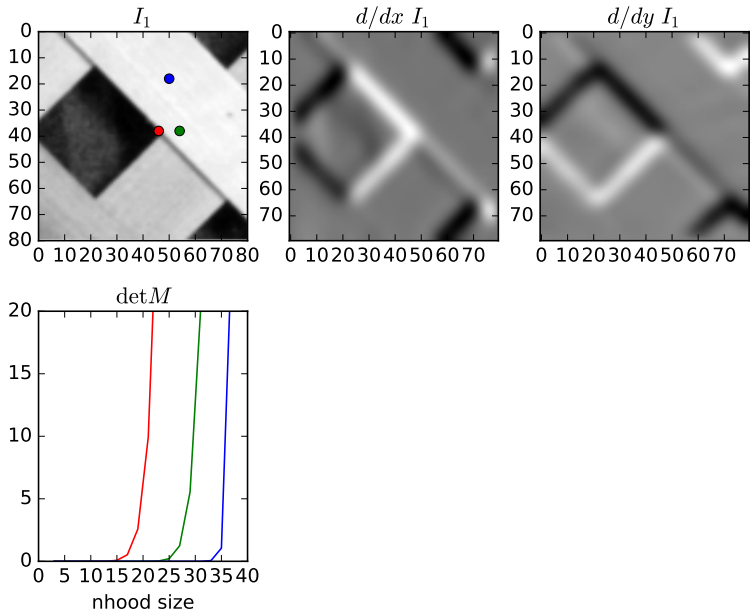

Figure 2: Top row: Original Image; its spatial derivative w.r.t. x, obtained via its convolution with d/dxg (where g is the normalized Gaussian weight function); its convolution with d/dyg. Bottom row: $\det \boldsymbol{M}$ as a function of the neighborhood size. For small neighborhoods (*e.g.*, 5×5) $\boldsymbol{M}$ is (almost) singular in all cases; *i.e.*, for such a small neighborhood, $\boldsymbol{M}(\boldsymbol{x})$ is (almost) singular at almost every $\boldsymbol{x}$. The required neighborhood size for $\det \boldsymbol{M}$ to become large enough is the smallest for the point closest to a corner (red), and largest for the point in the middle of the homogeneous region (blue); *i.e.*, the farther $\boldsymbol{x}$ is from a corner, the larger the required neighborhood size is.

Recall, using the notation from the lecture, the Lucas-Kanade Matrix:

$$\boldsymbol{M}(\boldsymbol{x}) = \begin{bmatrix} \sum gI_x^2 & \sum gI_xI_y \\ \sum gI_xI_y & \sum gI_y^2 \end{bmatrix} \qquad \boldsymbol{b}(\boldsymbol{x}) = \begin{bmatrix} -\sum gI_xI_t \\ -\sum gI_yI_t \end{bmatrix} \tag{35}$$

$$\hat{\boldsymbol{u}}_{\mathrm{LK}}(\boldsymbol{x}) = \boldsymbol{M}^{-1}(\boldsymbol{x})\boldsymbol{b}(\boldsymbol{x}) \tag{36}$$

Let λ_1, λ_2 denote the eigenvalues of $\boldsymbol{M}$ (with $\lambda_1 \geq \lambda_2$). In a homogeneous region, both λ_1 and λ_2 tend to be small. Near or on an edge, λ_1 is large and λ_2 is small. Near or on corner, both are large. A good question is how far we should be from an edge or a corner to call it "near". As Figure 2 shows, even near a corner we may need to use a fairly-large neighborhood; *e.g.*, as we can see from the red plot, even for the point near a corner we needed a neighborhood

larger than 15×15 before $\det \boldsymbol{M}$ becomes significantly larger than zero. The exact rate in which these plots increase also depends in how much blurring was used when computing the spatial derivatives; recall you can either first blur and then differentiate, or just convolve the original image with the derivative of a Gaussian filter.

One take-home message from all this is that when you implement the LK optical flow, if you pick a neighborhood which is too small (and/or your weight function did not blur enough), you will get a singular matrix $\boldsymbol{M}$.

2.3.2 Lucas-and-Kanade Optical Flow

Let $\boldsymbol{x}_i = (x_i, y_i)$ be in some neighborhood of $\boldsymbol{x} = (x, y)$. An affine flow model has the following form:

$$\begin{aligned}\begin{bmatrix} u(\boldsymbol{x}_i) \\ v(\boldsymbol{x}_i) \end{bmatrix} &= \begin{bmatrix} \theta_1 & \theta_2 & \theta_3 \\ \theta_4 & \theta_5 & \theta_6 \end{bmatrix} \begin{bmatrix} x_i - x \\ y_i - y \\ 1 \end{bmatrix} \\ &= \underbrace{\begin{bmatrix} x_i - x & y_i - y & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_i - x & y_i - y & 1 \end{bmatrix}}_{A(\boldsymbol{x}, \boldsymbol{x}_i) \triangleq} \underbrace{\begin{bmatrix} \theta_1 \\ \theta_2 \\ \theta_3 \\ \theta_4 \\ \theta_5 \\ \theta_6 \end{bmatrix}}_{\boldsymbol{\theta} \triangleq}\end{aligned} \tag{37}$$

Together with the gradient-constraint equation, we get:

$$\nabla_{\boldsymbol{x}} I(\boldsymbol{x}_i, t) \boldsymbol{A}(\boldsymbol{x}, \boldsymbol{x}_i) \boldsymbol{\theta} + I_t(\boldsymbol{x}_i, t) = 0 \tag{38}$$

The Lucas-Kanade solution for affine flow is then:

$$\hat{\boldsymbol{\theta}}_{\mathrm{LK}}(\boldsymbol{x}) = \underbrace{\boldsymbol{M}^{-1}(\boldsymbol{x})}_{6 \times 6} \underbrace{\boldsymbol{b}(\boldsymbol{x})}_{6 \times 1} \tag{39}$$

where

$$\boldsymbol{M}(\boldsymbol{x}) = \sum_i g(\boldsymbol{x}, \boldsymbol{x}_i) \boldsymbol{A}(\boldsymbol{x}, \boldsymbol{x}_i)^T \nabla_{\boldsymbol{x}} I(\boldsymbol{x}_i, t)^T \nabla_{\boldsymbol{x}} I(\boldsymbol{x}_i, t) \boldsymbol{A}(\boldsymbol{x}, \boldsymbol{x}_i) \tag{40}$$

$$\boldsymbol{b}(\boldsymbol{x}) = -\sum_i g(\boldsymbol{x}, \boldsymbol{x}_i) \boldsymbol{A}(\boldsymbol{x}, \boldsymbol{x}_i)^T \nabla_{\boldsymbol{x}} I(\boldsymbol{x}_i, t)^T I_t(\boldsymbol{x}_i, t) \tag{41}$$

and g is a weight function (as in the lecture slides).

Problem 15 *Show that the solution in Equation (39) (where $\boldsymbol{M}(\boldsymbol{x})$ and $\boldsymbol{b}(\boldsymbol{x})$ are given by Equation (40) and Equation (41), respectively) indeed minimizes*

$$\sum_{i=1}^{N} g(\boldsymbol{x}, \boldsymbol{x}_i) \left(\nabla_{\boldsymbol{x}} I(\boldsymbol{x}_i, t) \begin{bmatrix} u(\boldsymbol{x}_i) \\ v(\boldsymbol{x}_i) \end{bmatrix} + I_t(\boldsymbol{x}_i, t) \right)^2 \tag{42}$$

under the affine-flow assumption. ◊

We can use this kind of method in two different ways (yielding different results).

1. Solve this system just once, for the entire image, using a weight function whose support is the entire image (where typically the weights are all 1). In which case, we may as well take $\boldsymbol{x}$ to be the center of the image. This results in a single affine flow model for the entire image (since this is really a global model, I prefer referring to the overall LK approach as patch-based – where in this particular case the patch is the entire image – rather than referring to it as a local approach).

2. Solve such a system for each pixel $\boldsymbol{x}$, where typically the support of the weight function is limited to a small neighborhood (*e.g.*, 3×3, 5×5, etc.). In which case, we typically let $g(\boldsymbol{x}, \boldsymbol{x}_i)$ decay with the distance between $\boldsymbol{x}$ and $\boldsymbol{x}_i$; *e.g.*, in a Gaussian way.

We we will refer to the first option as globally-affine flow and to the second as locally-affine flow.

Computer Exercise 8 (this exercise is optional) *Implement the iterative LK algorithm for a locally-affine flow, and apply it to each one of the five frame pairs. Display the estimated flows, and the results of warping I_1 using these flows. Explain the quality of the estimates for different points in the first image.*$\diamondsuit$