Interactive Tutorial Optical Flow, part 1: 2D Local Optical Flow

Stefan Karlsson

November 2013, v1.01

1 Introduction

Start by running the script ‘runMe’. This will display a synthetic image sequence together with the estimated optical flow. Use the arrow keys and *q,a,w,s,e,d,p* to modify the motion and lag-time of this sequence.

- arrow keys: move the pattern around
- q/a : increase/decrease lag time (delay) between frame-updates
- w/s : increase/decrease speed of motion along a predefined trajectory (shaped like an **8**)
- e/d : increase/decrease speed of rotation
- p: pauses rendering and calculations, all visualizations freeze.

Notice that as you interact with the keyboard, the title bar of the figure changes to indicate the state of rendering.

”runMe.m” sets up the call to ”vidProcessing2D” which is our entry point for video processing. You can finish this tutorial without looking inside ”vidProcessing2D” but the code has been made to be pedagogical. Once its called, the figure will open and display the video together with visualizations. The code supports different sources of video. Instead of a synthetic sequence, you can load a video by changing ‘movieType’ as indicated in ‘runMe’. Try the provided ‘LipVid.avi’ file.

I recommended that you use a webcam for this tutorial. If you are running on windows you dont need any extra toolboxes[1], just plug the webcam in, and chose movieType = ’camera’.

Once you have finished viewing the results of the video processing, simply close the window to return back from the function. At that point, the datastructures dx, dy, dt and U and V are returned. The first three are the numerical

[1]image acquisition toolbox is recommended for all platforms, but you will get slower camera input even without it on windows

estimates of the derivatives of the video sequence (I_x,I_y and I_t) and the last two are the estimates of optical flow. The derivatives and optical flow returned will all be w.r.t the last frame that was viewed (just as the figure was closed).

This tutorial consists of completing a few pedagogically selected pieces of code. The code you need to complete will be less than 10 lines in total and will aid you in understanding both the principles of video processing and optical flow. The code you will modify is found in 3 separate m-files, listed in the order you need to fix them:

1. grad2Dm.m, which is used calculate the derivatives of the video sequence (dx, dy, and dt).
2. DoEdgeStrength.m, which is used for edge detection (this is to aid your fundamental grasp of dx, dy and dt).
3. DoFlow.m, which uses dx, dy and dt to calculate optical flow

At this moment, however, you have working versions of all these three files lying in your main folder. First thing to do is to ***remove the working m-files, and replace them with the broken versions that are found in the folder "tutorialFiles"***. The working files are your correct solutions, so you can review them if you get stuck. The broken files wont make the application crash or perform chaotically, it will simply result in the output derivatives and flow field to be zero.

2 Estimating derivatives - grad2Dm.m

When processing video, it is crucial to be timely. There is alot of data and we do not want to shuffle it around needlessly. The key is to have only what you need, and only when you need it. The gradient calculating module is the only place where we require 2 frames of the video in the application, and therefore it makes sense to store a local copy of the previous frame.

For calculating derivatives, we make use of 3-by-3 filter masks, that are generated by sampled Gaussians and Gaussian derivatives. It is not within the scope of this tutorial to give a background into the more basic aspects of convolution, so this is left for the reader to look up for himself if needed.

The code for grad2Dm.m that you are required to fill in consists of the final step of the calculation of I_x and I_y component images(lines of code labeled "L1" and "L2" in remarks). It is a question of using the conv2 function correctly. Finally, on the line labeled "L3" you are required to realize how to use a simple difference of frames as estimate for temporal derivative, I_t.

When you are done with the derivatives implementation, we can show the 3 component images and try to interpret them. This can be done by setting the argument 'method' inside of 'runMe' to method = 'gradient'. Do this, and re-run 'runMe.m'. Use the synthetic video.

Are the gradient components as you expect them to be? Can you already now see a pattern? How does the motion affect the dt component compared to the dx and dy?

If you notice a relation between the dt component and the dx, dy component, odds are that it is the so-called "optical flow constraint".

2.1 Optical Flow Constraint

An important assumption to most optical flow algorithms, is the brightness constancy constraint(BCC). In plain words, this means simply that the brightness of a point remains constant from one frame to the next, even though its position will not. A first order approximation of the BCC is sometimes called the optical flow constraint equation. It can be written as:

$$I_t + v_x I_x + v_y I_y = 0 \tag{1}$$

where $\vec{v} = \{v_x, v_y\}$ is the motion we are trying to estimate with optical flow algorithms. We can also write $\vec{v} = -|\vec{v}|\{\cos(\phi), \sin(\phi)\}$, where ϕ is the angular direction of the motion, and write the optical flow constraint equation as:

$$I_t = -|\vec{v}|\left(\cos(\phi)I_x + \sin(\phi)I_y\right) \tag{2}$$

The quantity $(\cos(\phi)I_x + \sin(\phi)I_y)$ is found in the r.h.s, and is what we call a 'directional derivative'. It is the rate of change in a particular direction[2], namely ϕ. Now, run again the script 'runMe', with the same settings as before (use synthetic image sequence), observe the derivative component images.

Does the optical flow constraint equation seem to hold? Does I_t resemble a directional derivative? Does it look like a linear combination of I_x and I_y?

When you close the figure, try to do so as the pattern is moving at an angle $\phi = 45^o$ (this is when the pattern is moving **towards the lower right corner**[3] as indicated in figure 1. After you have closed the figure, enter the code:

```
figure;subplot(1,2,1);imagesc(dt);
colormap gray; axis image; title('dt');

subplot(1,2,2);imagesc(-(dx+dy));
colormap gray; axis image; title('-(dx+dy)');
```

The images should be very similar, explain why[4].

[2]In fact, I_x and I_y are both directional derivatives with the special directions of along the x- and y-axis

[3]remember that matlab has its origin at the top left corner of the figure, and that the y axis is pointing downwards

[4]Hint: use Eq. 2

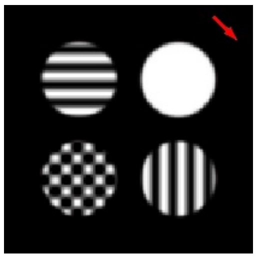

Figure 1: the position of the pattern as it is moving with direction $\phi = 45^o$. This is what you should see when shutting down the figure(time it well). Red arrow shows the motion vector.

Now lets experiment with faster motions, by speeding up the synthetic video. While the program is running, and the main figure is selected, hit the ”w” key to make the video faster, ”s” slows it down. You can also introduce a rotation of the pattern by the keys ”e” and ”d”, and move the pattern around with the arrow keys. Get an idea of how the derivatives change as a result. Hit ”p” at any time to pause for a snapshot.

Make the motion in the sequence 10 times faster, with no rotation. Shutting down the figure at the right time may now prove difficult (depending on how fast your computer is). An extra lag time (delay between each frame update) can be increased by hitting ”q”, while ”a” reduces it. Pausing the sequence using ”p” before you shut down the figure makes it easier as well. While the speed is 10 times faster, make the lag time sufficient for you to shut down the figure just as the motion is $\phi = 45^o$. After that, run once more the code

```
figure;subplot(1,2,1);imagesc(dt);
colormap gray; axis image; title('dt');

subplot(1,2,2);imagesc(-(dx+dy));
colormap gray; axis image; title('-(dx+dy)');
```

Are the two images still similar under higher motion?

Redo the same experiment once more, but this time change the scale at which the gradient is calculated. The code supports 2 different scales (coarse and fine). Change the argument ‘bFineScale’ to bFineScale=0, in order to use coarse scale derivatives. Run the experiment at a motion of 10 as before.

Are the two images ‘dt’ and ‘-(dx+dy)’ similar now?

2.1.1 take-home message

You should understand that higher motions makes the optical flow constraint equation (Eq. 1, equivalently Eq. 2) fail. You should understand that this can be compensated for by using a coarser scale of derivatives[5]. you should also understand that the coarser scale destroys some of the important detail in the image[6].

3 Edge filtering - DoEdgeStrength.m

Your second coding task is to implement edge detection in the image sequence. This is a common task in video processing, both for motion tasks as well as a range of other computer vision challenges. The function 'DoEdgeStrength.m' is there for this task. The way we will do this edge detection is very simple. Let the 2D gradient of the image be $\nabla_2 I(\vec{x}) = \{I_x(\vec{x}), I_y(\vec{x})\}$, then edges are found by high values of $|\nabla_2 I(\vec{x})|$.

Your task is to implement the function 'DoEdgeStrength.m' so that it returns such a 2D image.

DoEdgeStrength is particularly easy, as it will be one single line. In order to view the edge detection in realtime[7] on the video feed, change the argument in the runMe script:

```
Method = 'edge';
```

4 Optical Flow - DoFlow.m

Optical flow is the apparent 2D motion in an image sequence. Every position, in every video-frame will have a 2D vector associated with it. The vector tells the estimate of motion of that position, as it moves from one frame to the next. Any region Ω where optical flow is to be estimated reliably must contain "nice texture"[8]. In this context "bad textures" are regions that have either...

- constant gray value(no information), or..
- regions of linear symmetry(information in only one direction).

We can say that "nice textures" are those with linearly independent 2D gradients $\nabla I(\vec{x_i})$. In practical language, we must be sure that we do not have a region of the kind we find in the barber pole illusion (see fig 2 or google "barber pole illusion" for a proper animated version).

In the synthetic test sequence, 4 examples textures are given within the support of 4 separate disks. Only one of the textures are nice(checkerboard

[5]coarser scale derivatives corresponds to a coarser scale image. Intuitively, downsampling the video in x and y makes larger motions correspond to smaller ones in the coarser scale

[6]There are many ways of estimating gradients. For optical flow, the optical flow constraint equation is the most important constraint to hold. Testing how well it holds in the fashion described above, tests how well suited your derivative estimation is for optical flow purposes. This is better than tuning the filters based on the stability of the observed flow output, because that stability depends on a larger region as well as other computational steps.

[7]this is especially fun if you can get a camera working, and viewing yourself

[8]with "nice" we will mean textures with well-distributed structure. If that does not make sense to you right now, just read on

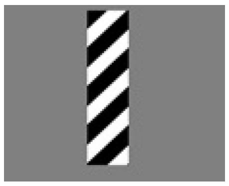

Figure 2: The barberpole illusion. A pattern of linear symmetry is wrapped around a cylinder, and rotated. The true motion is rotation left or right but the perceived motion by the observer is up or down

pattern), two are linear symmetric images (vertical and horizontal bars) and one is a constant value(just a disk).

Most of the time, the question of whether a region is "nice" or "bad", depends on how big we make it. Making a region bigger, makes it more likely to gather observations from the image that provide new directional information. For the synthetic image, if we consider the entire image as the region, it is "nice" as a whole. If we consider a region covering only the horizontal and vertical bars, then that region is nice as well (although individually they are "bad"). Making a region bigger will have the drawback of reducing the resolution of the resulting optical flow, so a comprimise is necessary:

- big region $\rightarrow$ better data,
- small region $\rightarrow$ better resolution.

This phenomenon is often referred to as the aperture problem (the aperture meaning the shape of the region we do the estimate over).

4.1 2D Structure Tensor

Whether the region is "nice" or "bad" is given by the 2D structure tensor, defined as:

$$S_{2D} = \begin{pmatrix} m_{20} & m_{11} \\ m_{11} & m_{02} \end{pmatrix}$$

where the elements are so-called "spectral moments", for example: $m_{11} = \int\int D_x f D_y f$. A region is "nice"(well distributed structure) if both eigenvalues are large. This is equivalent to saying that S_{2D} is "well conditioned". In practical terms, it means that S_{2D} can be inverted with no problems. If S_{2D} can *not* be inverted, its because we have a "bad" texture, and the 2 cases are distinguished naturally as:

- constant gray value(both eigenvalues of S_{2D} are zero)
- linear symmetry(one eigenvalue of S_{2D} is zero).

In practice, we will always consider a local region Ω, and will always have discrete images. Therefore we can write here $m_{11} = \sum w I_x I_y$, where w is a smooth window function (e.g. Gaussian) covering the region Ω, I_x is the x-derivative of the image I, and I_y is its y derivative. In the code we have the datastructures dx, dy and dt that stores our estimates of these derivatives. In general, we will write $m_{ijk} = \sum w I_x^i I_y^j I_t^k$. Assuming that we can move the smooth window function w around, we can consider different regions Ω, differing only where they have their window functions centered (at some $\vec{x}$, say). We will therefore write $m_{11}(\vec{x})$ to indicate positioning of Ω at $\vec{x}$.

These so-called moment images will be central to our implementation of optical flow, and you can find how the $m_{20}(\vec{x})$ and $m_{02}(\vec{x})$ are calculated in the function "DoFlow.m"(lines starting with the label "MOMENT CALCULATIONS"). The steps are:

1. make elementwise product:
   `momentIm = dx.^2;`
2. smooth with large seperable gaussian filter
   `momentIm = conv2(gg,gg,momentIm,'same');`
3. downsample to specified resolution
   `m200 = imresizeNN(momentIm ,[flowRes flowRes]);`

In this approach, the region Ω is a gaussian and represented by the filter `gg` in the code. To view what Ω looks like, execute the following lines:

```
gaussStd = 2;
gg=gaussgen(gaussStd); %gaussgen is in 'helperFunctions'
imagesc(gg'*gg); colormap gray; axis image
```

Here, `gg` is a one dimensional filter used in seperable filtering[9] and `gg'*gg` is its outer product: the equivalent 2D filter we use.

It will be your task to write the expressions for several $m_{ijk}(\vec{x})$ images in "DoFlow.m", and you should do that now.

The moment images makes it possible to define local structure tensors[10], one for each position in the image as:

$$S_{2D}(\vec{x}) = \begin{pmatrix} m_{20}(\vec{x}) & m_{11}(\vec{x}) \\ m_{11}(\vec{x}) & m_{02}(\vec{x}) \end{pmatrix}$$

[9]We could equally well use a 2D filter directly, but making slower code.
[10]a.k.a tensor field

4.2 Optical flow by Lucas and Kanade

Start by considering the optical flow constraint equation (Eq. 1 or equivalently Eq. 2). We can make many observations of I_x, I_y and I_t if we consider many positions in some region (thus we can index different observations as $I_t(\vec{p})$). We will estimate motion over a region Ω, centered at some $\vec{x}$, and so we consider only derivatives within that region(those are the positions $\vec{p}_1$, $\vec{p}_2$ ... $\vec{p}_N$). If all our observations are gathered into d and D:

$$d = \begin{pmatrix} I_t(\vec{p}_1) \\ I_t(\vec{p}_2) \\ \cdots \\ I_t(\vec{p}_N) \end{pmatrix}, \mathrm{D} = \begin{pmatrix} I_x(\vec{p}_1) & I_y(\vec{p}_1) \\ I_x(\vec{p}_2) & I_y(\vec{p}_2) \\ \cdots & \\ I_x(\vec{p}_N) & I_y(\vec{p}_N) \end{pmatrix}$$

.

Then we can express the optical flow constraint for every position as the following system of equations

$$d = -\mathrm{D}\vec{v}(\vec{x}) \tag{3}$$

In most cases, this N-by-2 system cannot be solved (too many rows usually means overdetermined). The LK method solves this using the least square error method which is a standard numerical approach for such systems. We will do it explicitly here:

Instead of an exact solution for $\vec{v}$, we seek one that "fits the data" best. This is a $\vec{v}$ that will try to conform as "best it can"[11] to all the rows of Eq. 3, meaning all the observations at all position in the region. This solution can be achieved by simply formulating a new system of equations:

$$\mathrm{D}^T d = -\mathrm{D}^T \mathrm{D}\vec{v}(\vec{x})$$

This changes the system fundamentally to be 2-by-2, and this new system can be interpreted as[12]:

$$\begin{pmatrix} m_{101}(\vec{x}) \\ m_{011}(\vec{x}) \end{pmatrix} = -\begin{pmatrix} m_{20}(\vec{x}) & m_{11}(\vec{x}) \\ m_{11}(\vec{x}) & m_{02}(\vec{x}) \end{pmatrix} \vec{v}(\vec{x})$$

For clarity, we can introduce a vector $\vec{b}$,

$$\vec{b}(\vec{x}) = \begin{pmatrix} m_{101}(\vec{x}) \\ m_{011}(\vec{x}) \end{pmatrix}$$

[11] In the least square error sense. Note that modern state-of-the-art algortihms in optical flow use tend to use other error minimization

[12] To interpret it this way, we also weight the neighbourhood with the window function. This is sometimes called a weighted least squares formulation.

and remind ourselves what the 2D structure tensor is:

$$S_{2D}(\vec{x}) = \begin{pmatrix} m_{20}(\vec{x}) & m_{11}(\vec{x}) \\ m_{11}(\vec{x}) & m_{02}(\vec{x}) \end{pmatrix}$$

yielding the grand expression for the LK method of optical flow:

$$\vec{v}(\vec{x}) = -S_{2D}^{-1}\vec{b} \tag{4}$$

From the previous section we know of cases when S_{2D} can not be inverted! This happens for the "bad textures" (the case when we have the "barber pole" for example). We must somehow deal with this, and one way is to check how well-conditioned S_{2D} is before inverting(thereby simply skipping those regions altogether). To check if a matrix is well-conditioned one commonly uses the conditional number. In Matlab, we use the function "rcond" for this[13], as seen on the line labelled "L1" in DoFlow.m.

Once you are done, the configuration for running the LK algorithm is with arguments:

```
method = 'LK';
```

Run the LK algorithm on the synthesized sequence. Does the algorithm perform as you would expect? Are there any points in the sequence where performance is better than other places?

4.2.1 take-home message

This implementation sort of works with the synthetic sequence, but if you experiment with recorded video or live feed from a camera, you notice quickly that the method has two drawbacks:

1. Unstable (unable to handle linear symmetry regions and explodes on occasion), and
2. Inefficient(slow computations)

Regarding the first issue, changing the threshold "EPSILONLK" in the DoFlow function, will allow you to tune it somewhat for different data. Next, we deal with the two drawbacks listed above, in much better way by regularizing and vectorizing the method.

4.3 Regularization and Vectorization - "flow1"

Inside of the DoFlow function, there is an example of a vectorized and regularized version of LK (activated by method = 'flow1'). Lets take a moment and derive this implemention as well. It deals with the two drawbacks of instabillity and ineffeciency of the standard LK implementation.

[13]Another approach is to apply some form of "regularization", which we will do in the following section.

4.3.1 Local Regularization

The aperture problem was addressed by investigating the conditional number("rcond" in Matlab) of the structure tensor, before inverting it. With local regularization[14], we aim to change the conditional number instead of only investigating it. The simplest, most commonly used approach for this is so-called Tikhonov regularization. This consists of adding a small value to the diagonal elements of the matrix to be inverted. It makes a huge difference to the stability of the algorithm, and will also allow it to handle the linear symmetry textures, although it gives only the motion that is parallel to gradients for those regions.

This is done on the lines labelled "L2", introducing a different parameter to tune, "TikConst". As with the "EPSILONLK" parameter, play around with TikConst to suit your data. TikConst will affect the resulting flow field in the following way:

- make it too small $\rightarrow$ numerically unstable, ill-conditioned solutions (the flow field will wiggle around and explode occasionally),
- make it too large $\rightarrow$ all flow vectors will tend to shrink in magnitude

4.3.2 Vectorization

Implementing the LK by using vectorized programming[15] will make use of the built in parallelism in Matlab.

Lets revisit the grand expression for the LK method, Eq. 4. This is a 2-by-2 system, and its solution can be derived analytically. To derive the full symbolic expression for the solution of the system, use the matlab symbolic toolbox[16] by typing:

```
syms m200 m020 m110 m101 m011;

b   = [m101; ...
       m011];

S2D = [m200, m110; ...
       m110, m020];

v = -S2D\b
```

This is implemented on lines labelled "L3" in "DoFlow.m" as:

```
U =(-m101.*m020 + m011.*m110)./(m020.*m200 - m110.^2);
V =( m101.*m110 - m011.*m200)./(m020.*m200 - m110.^2);
```

[14]The keyword "Local" is used here to not confuse the topic with global regularization, and so called variational approaches. I will hopefully cover simple such approaches in part 3 or 4 of the planned series

[15]in particular, using no for loops

[16]Mathematica or Maple are also good tools for these sorts of tasks

With a vectorized formulation, we can now run the algorithm with ease at the same resolution as the original video (i.e. one flow vector per pixel). A dedicated streamlined m-file for "flow1" can be used by putting method = 'flow1Full'. When dealing with higher resolution optical flow, it becomes inefficient to illustrate the flow using vectors, and it is customary to display it with a colorcoding instead, illustrated in figure 3.

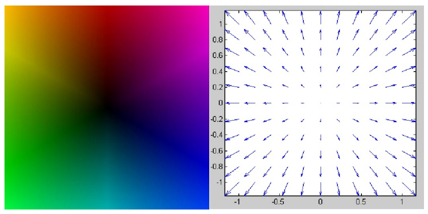

Figure 3: Color coding of the optical flow. Left, the higher resolution, color equivalent of the right side vector version.