Interactive Tutorial on Optical Flow Motion Estimation

Stefan Karlsson, Ph.D., Josef Bigun, Ph.D.
Contact: stefan.karlsson987(at)gmail.com

Feb 2021, v1.07 (First version: June 2013)

1 Let's get started

Start by running the script `runMe`. This will display interactive synthetic video, with green arrows displaying the optical flow. Use the arrow keys and **Q,A,W,S,E,D,P** to modify the rendering as specified below.

- **Arrow Keys**: move the pattern around

- **E/D**: increase/decrease speed of rotation

- **P**(toggle): pauses rendering and calculations, all visualizations freeze.

- **W/S**: increase/decrease speed of motion along a predefined trajectory (shaped like an **8**)

- **Q/A**: increase/decrease lag time (delay) between frame-updates

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 `vidProcessing` which is our interface with the toolbox. The toolbox 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:

```
%  in runMe.m  ...  %
in.movieType = 'lipVid.avi'; % assumes a file 'LipVid.avi' in current folder.
```

It is recommended that you use a camera for this tutorial. If you are running on windows you don't need any toolboxes[1], just set:

```
in.movieType = 'camera';       %assumes a connected camera to your computer
```

If at any time during the tutorial you would like to see the principles applied to real-world data, simply activate a different `movieType` input.

---

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

2 Introduction

This tutorial is about motion in video; dense optical flow. With optical flow we are interested in estimating motion at every position in the image. A good example is provided as you call `runMe.m` with the default settings. Another way to measure motion, that we do not deal with in this tutorial, is by tracking points from one frame to the next. Sometimes, point tracking is called sparse optical flow. With point tracking, we select (automatically or manually) points over the video and display their new position for each subsequent frame. With point tracking, we process positions in the image that change over time, whereas with dense flow, the positions are fixed.

Our intent is to get new minds interested in the topics we love, but also to advance the field by promoting the use of good engineering approaches. At the time of making the first version of this tutorial (early 2013) a widespread myth(in the computer vision community) was that dense optical flow can only be achieved at extreme computational cost. This tutorial arrives with ease at implementations that run in real-time, even in pure Matlab without using the GPU. Estimating optical flow that is ***accurate enough***, and stable and robust for many real-world problems is easy.

This tutorial/toolbox is written/coded/maintained by Stefan Karlsson and has come about through work done with Josef Bigun at CAISR, Halmstad University. It is intended as a free educational resource as well as a toolbox for those experimenting with real-world motion algorithms.

2.1 Outline

This tutorial is centered on completing some code; about 10 lines in total. We will work with 4 separate m-files, listed in the order you need to fix them:

1. `grad3D.m`, which is used to calculate the derivatives of the video sequence (`dx`, `dy`, and `dt`).

2. `DoEdgeStrength.m`, which is used for edge detection.

3. `FlowLK.m`, which uses `dx`, `dy` and `dt` to calculate optical flow.

4. `Flow1.m`, which provides improvements to the flow estimation.

At this moment, however, you have working versions of all these four files 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.

3 Estimating derivatives - grad3D.m

We will denote partial derivatives as $I_x = \frac{\partial I}{\partial x}$, $I_y = \frac{\partial I}{\partial y}$ and $I_t = \frac{\partial I}{\partial t}$. Corresponding numerical estimates in Matlab are denoted: `dx`, `dy` and `dt`.

The function header of `grad3D` shows 2 inputs, and 3 outputs:

```
%  in grad3D.m    %
[dx, dy, dt] = grad3D(imNew,imPrev)
%calculates the 3D gradient from two images.
```

`imNew` and `imPrev` are the new and previous frames of the video respectively.

For derivatives: I_x and I_y, we can make use of 3-by-3 differential filters[2], sometimes referred to as central difference over a compact stencil. The simplest way to estimate the I_t derivative is by taking the difference between frames.

In `grad3D.m` the code labelled "L1" and "L2" in remarks are for you to fill in. It is a question of using the `conv2` function correctly. Finally, on the line labeled "L3" you should use a difference of frames to estimate I_t.

When you are done with the derivatives implementation, we can show the 3 component images and try to interpret them in real-time. This can be done by setting the argument `method` inside of `runMe` to:

```
 in.method = 'gradient';        %makes the program visualize the gradient only
```

Re-run `runMe.m`, use the synthetic video.

Are the gradient component images as you expect them to be? Do you notice a relation between `dt` and `dx`, `dy` images? Odds are that you notice something of the relationship often called **"optical flow constraint"**(read on).

3.1 Optical Flow Constraint

An important assumption to most optical flow algorithms, is the brightness constancy constraint(BCC). This means 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:

$$I_t + vI_x + uI_y = 0 \tag{1}$$

where $\vec{v} = \{v, u\}$ 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 ϕ[3].

---

[2]The well-used Sobel operator is a special case of this, where the filter sum is not normalized and contains only integers (1,2,4) of basis 2

[3]In fact, I_x and I_y are both directional derivatives with $\phi = 0$ and $\phi = 90^o$

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?

Set the motion of the pattern to be along the pattern **8**, using keys **W**/**S**. The title bar of the figure contains text indicating the parameters of the rendering. Set the 'speed' to be equal to 1. Pause the motion (keyboard button **P**) as the pattern is moving at an angle $\phi = \pi/4 = 45°$ (this is when the pattern is moving **towards the lower right corner**[4] as indicated in figure 1. When paused, enter the following code into the matlab command prompt:

```
% We display  the time derivatives (first)

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

% ...Next to it we display  a specific combination of the spatial derivatives
phi = pi/4; subplot(1,2,2);
imagesc(-(cos(phi)*dx+(sin(phi)*dy)));
colormap gray;axis image;
```
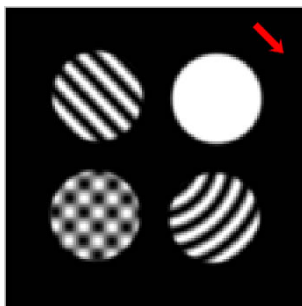
The images should be similar according to Eq. 2.

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

Now lets experiment with faster motions. Set the speed parameter to 2.5, by hitting **W**. Get an idea of how the derivatives change as a result. Pausing the figure at the right time may now be tricky. An extra lag time can be added by hitting **Q**, while **A** reduces it. Pause just as the motion is $\phi = 45°$ as before, and run the same code as before to display `dt` and `cos(phi)*dx+sin(phi)*dy`.

This time the images are no longer very similar. The optical flow constraint is valid as an approximation only when the motion is small.

---

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

Redo the same experiment once more, but this time change the scale (the resolution) at which the gradient is calculated. The toolbox allows you to define any resolution of the input video by the argument `vidRes`. Default height and width of the video is 128, lets make that half by (in `runMe.m`):

```
% [Height Width]:
in.vidRes  = [64 64];  %video resolution, for camera and synthetic input
```

Run the experiment at a higher motion as before. With the coarser scale, the two images will once more be similar. As we reduce the size of the video resolution, large motions become small motions(as measured in pixels/s). However, notice that the video has far less detail in it.

3.1.1 Tuning the gradient filters

There are many ways of estimating gradients, and what is the best method depends on what we wish to use it for, and what demands we have on our final algorithm in terms of stability, accuracy and timeliness.

The optical flow constraint equation should be a guiding principle for testing any single gradient estimation algorithm when the aim is motion. This can be done in the fashion outlined in section 3.1, and does not need for any explicit optical flow to be calculated.

On that topic, we can once more re-visit `grad3d.m`. If you look into the original file (that occupied the main folder before you replaced it with the one from "tutorial files"), it describes 2 more approaches of gradient estimation, constructed for the optical flow constraint to hold better. They generally yield better result in the end. In short, they add the following:

- **Correctly centered spatio-temporal filtering**. First approach does not correspond to correctly centered filtering if one consider a spatio-temporal volume. We can centre the filtering inbetween frames. This corresponds to having the same stencil for all the gradient components (3x3x2).

- **Boundary effects**. First approach gets edge effects near the image boundary. To avoid this single-sided differences are applied near the boundaries. This is similar to the built-in function `gradient`, except we use wider kernels/stencils (3x3x2 in middle of image, 2x3x2 and 3x2x2 at boundaries).

Feel free to implement any kind of gradient estimation you like and visualize it with the toolbox. Just make sure that the size of `dx`, `dy` and `dt` are always the same: identical to the frame-size of the video.

4 Edge filtering - DoEdgeStrength.m

The 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 function header shows us 4 inputs, and a single output:

```
function edgeIm = DoEdgeStrength(in, imNew, imPrev,edgeIm)
```

As input we have the new and the previous video frames (`imNew`, `imPrev`). We also have the structure `in`, which contains all the parameters set in `runMe.m`, as well as the previous `edgeIm` that was generated.

The image `edgeIm` will contain the strength of edges as given by the 2D gradient $\nabla_2 I(\vec{x}) = \{I_x(\vec{x}), I_y(\vec{x})\}$. The edge strength is defined as the value: $P = |\nabla_2 I| = \sqrt{I_x^2 + I_y^2}$.

This first implementation of `DoEdgeStrength` will be just 2 lines. When you are done, in order to view the edge detection in realtime[5] change the argument in the runMe script:

```
in.Method = 'edge';
```

4.1 Gamma Correction

What is meant by an edge depends on the application. A way to change the sensitivity of our detector and let more candidates be highlighted is by $P = |\nabla_2 I|^\gamma = \left(I_x^2 + I_y^2\right)^{\frac{\gamma}{2}}$. Try implementing this, and then experiment with a few different γ values. In the original version of `DoEdgeStrength.m`, there is the possibility to modify the γ parameter interactively using the keyboard ($\mathbf{R}/\mathbf{F}$).

4.2 Temporal integration

An important topic for optical flow is temporal integration. Many algorithms for flow estimation are improved by incorporating more images of the sequence. Storing more images of the video sequence is not desirable but there are other ways. One trick is a recursive filter[6]. Before we approach optical flow, lets try this principle on our edge detection algorithm.

---

[5]this is especially fun if you can get a camera working, and viewing yourself
[6]first order linear recursive filter to be exact

`DoEdgeStrength` receives 4 arguments `dx`, `dy`, `in` and `edgeIm`(the previous output of `DoEdgeStrength`). The idea: lets add the previous value to the current estimate. We denote our integrated edge strength at time t as $\hat{P}(x, y, t)$:

$$\hat{P}(x, y, t) = \alpha \hat{P}(x, y, t - 1) + (1 - \alpha)P(x, y, t)$$

In `DoEdgeStrength`, $\hat{P}(x, y, t - 1)$ is the input argument `edgeIm`, and α is the input `in.tIntegration`. After you implement this, lets view the result with a large temporal integration factor, by setting (in `runMe`):

```
in.Method       = 'edge';
in.tIntegration = 0.9;
```

Having such a high integration is not very useful for edge detection, but if you put it to a lower value, such as 0.2, you will see a reduced amount of noise. This should be especially clear when you have a connected webcam, and with low γ values.

5   Aperture Problem

Any region Ω where a motion vector $\vec{v}$ is to be estimated reliably must contain "nice texture". 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 pattern), two are linear symmetric images (straight and curved bars) and one is a constant value(just a disk).

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).

A region of constant value is bad for motion estimation because there is no information to work with. How about the linearly symmetric textures? Why are they so bad? Run the function:

```
ApertureIllustration;
```

In it, a region Ω is illustrated as a red circle that you can move around by clicking in the figure. Mouse scroll, or keys **Q/A** changes its radius, and **W/S** changes its boundary. A background circular motion is present of a linear symmetric pattern. If you put your aperture in the middle of the image, then it will be impossible to determine any true motion, except for the component that is aligned with the gradients. Notice that if you bring your aperture to cover parts of the edge of the pattern, you can almost instantly perceive the true motion; the edges contain more directionality for your vision system to work with.

We can say that "nice textures" are those with linearly independent 2D gradients $\nabla_2 I(\vec{x_i})$. In practical language, we must be sure that we do not have a region of the kind we find in barber poles (fig 2). To see an animated version of the barberpole illusion, type:

```
ApertureIllustration(2);
```


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

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. 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.

6   Optical flow by Differences or Correlation

Simply formulated, our goal is to find where a pattern has gone to in the next frame. The vector between the original location and the new location is the velocity vector, or the optical flow. We can answer the question by picking "all" candidate patterns in the next frame and decide for the (location of the) one which is most similar to the original pattern. Practically, picking candidate patterns by cutting/copying one patch having the same size as the original patch at a time is heavy computationally.

Instead, we shift the entire frame with a fixed vector, that we call (dxh, dyh) in the software. The net effect is that we displace all candidate patches at *the same relative location* in comparison to the original patch such that they are aligned with the original patch. We save each displaced frame under a unique identity, id in the software. The similarity of two patches can be evaluated in a number of ways from displaced frames, though it is most common to use one of the two comparison methods as follows.

- **Norm** of the difference between the pattern in the displaced frame and the original pattern, that is eq. (12.110) of the book

- **Correlation** between the pattern in the Displaced Frame and the original pattern, that is eq. (12.110) of the book

8

In the lab exercise you will have the opportunity to implement both. You will need to fill in places of "FILL_IN_PLEASE" in two functions *dfd.m* and *correlation.m* (in the directory *tutorialFiles*. We refer to section 2.10 of the book for theoretical exposure of the two, which of course are related to each other fundamentally.

7 2D Structure Tensor

Whether a 2D region is "nice" or "bad" is determined by its structure tensor:

$$\mathbf{S} = \begin{pmatrix} \iint I_x^2 \mathrm{d}\vec{x} & \iint I_x I_y \mathrm{d}\vec{x} \\ \iint I_x I_y \mathrm{d}\vec{x} & \iint I_y^2 \mathrm{d}\vec{x} \end{pmatrix} = \begin{pmatrix} m_{200} & m_{110} \\ m_{110} & m_{020} \end{pmatrix}$$

A region is "nice" if both eigenvalues are large. This is equivalent to saying that $\mathbf{S}$ is "well conditioned". In practical terms, it means that $\mathbf{S}$ can be inverted with no problems. If $\mathbf{S}$ can *not* be inverted, its because we have a "bad" texture, and the 2 cases are distinguished naturally as:

- constant gray value region (both eigenvalues of $\mathbf{S}$ are zero)

- linear symmetry region (one eigenvalue of $\mathbf{S}$ is zero).

In practice, we will always consider a local region Ω, and will always have discrete images. Therefore we can write here $m_{110} = \sum w I_x I_y$, where w is a window function covering the region Ω(the function `ApertureIllustration` gives you a nice view of a window function that you can position anywhere). 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 as window functions centered at some $\vec{x}$. We will therefore write $m_{110}(\vec{x})$ to indicate positioning of Ω at $\vec{x}$.

8 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 $I_t(\vec{p}_i)$). 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 $\vec{p}_1, \vec{p}_2 ... \vec{p}_N$).

The LK method handles this using the least square method which is a common numerical approach to solve over-determined linear systems of equations. Then we seek $\vec{v}$ that "fits the data" best. This is a $\vec{v}$ that will try to conform as "best it can" to all the observations at all positions in the region Ω. We wish to find the solution that "minimizes" all errors:

$$e(p_i) = u I_x(\vec{p}_i) + v I_y(\vec{p}_i) + I_t(\vec{p}_i) \tag{3}$$

where $e(p_i)$ is the error associated with the BCC equation generated by the point $p_i \in \Omega$. The expression $e(p_i)$ is called the error because it should ideally

be zero, in which case the equation becomes the BCC Equation. If there are N points in Ω then there are N such errors e_i:

$$\vec{e} = (e(p_1), e(p_2), \cdots e(p_N))^T \tag{4}$$

Defining the matrix $\mathbf{D}$ and the vector $\vec{d}$, which hold the observations, as

$$\mathbf{D} = \begin{pmatrix} I_x(\vec{p}_1) & I_y(\vec{p}_1) \\ I_x(\vec{p}_2) & I_y(\vec{p}_2) \\ \cdots & \cdots \\ I_x(\vec{p}_N) & I_y(\vec{p}_N) \end{pmatrix} \qquad \mathbf{d} = \begin{pmatrix} I_t(p_1) \\ I_t(p_2) \\ \cdots \\ I_t(p_N) \end{pmatrix} \tag{5}$$

a system of equations can be obtained

$$\vec{e} = \mathbf{D} \begin{pmatrix} u \\ v \end{pmatrix} + \vec{d} = \mathbf{D}\vec{v} + \vec{d} \tag{6}$$

Here we should ideally have $\vec{e} = \vec{0}$. However, we are far from the ideal in reality and $\vec{e}$ will never vanish for any fixed $\vec{v}$, in fact even the best $\vec{e}$ may not achieve to produce zero error at all equations, although it may come close to this goal. We are nontheless interested in finding the best $\vec{v}$. To find it, we will attempt to minimize

$$E_{LK}(\vec{v}) = \|\mathbf{D}\vec{v} + \vec{d}\|^2 = \vec{v}^T \mathbf{D}^T \mathbf{D}\vec{v} + d^T d + 2\vec{v}^T \mathbf{D}^T \vec{d} = \sum_{p_i \in \Omega} (uI_x(\vec{p}_i) + vI_y(\vec{p}_i) + I_t(\vec{p}_i))^2 \tag{7}$$

by choosing $\vec{v} = (u, v)^T$. Technically, $\vec{v}$ can be found by setting the derivatives of E_{LK} w.r.t. u, v and solving these equations[7]. In other words we solve[8] $\vec{0} = \left(\frac{\partial E_{LK}}{\partial u}, \frac{\partial E_{LK}}{\partial v} \right)^T$ yielding

$$\begin{aligned} \vec{0} &= \mathbf{D}^T \mathbf{D}\vec{v} + \mathbf{D}^T \vec{d} \\ &= \begin{pmatrix} \sum_{p_i \in \Omega} I_x^2(p_i) & \sum_{p_i \in \Omega} I_x(p_i)I_y(p_i) \\ \sum_{p_i \in \Omega} I_x(p_i)I_y(p_i) & \sum_{p_i \in \Omega} I_y^2(p_i) \end{pmatrix} \vec{v} + \begin{pmatrix} \sum_{p_i \in \Omega} I_x(p_i)I_t(p_i) \\ \sum_{p_i \in \Omega} I_y(p_i)I_t(p_i) \end{pmatrix} \\ &= \begin{pmatrix} m_{200} & m_{110} \\ m_{110} & m_{020} \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix} + \begin{pmatrix} m_{101} \\ m_{110} \end{pmatrix} = \vec{0} \end{aligned} \tag{8}$$

where we remind the reader of the notation $m_{ijk} = \sum_{p_i \in \Omega} I_x^i I_y^j I_t^k$. We recall at this point that the matrix term is the 2D structure tensor $\mathbf{S}$ for the current neighborhood Ω (around the current pixel x), and introduce the vector $\vec{b}$:

$$\mathbf{S} = \begin{pmatrix} m_{200} & m_{110} \\ m_{110} & m_{020} \end{pmatrix} = \mathbf{D}^T \mathbf{D}, \qquad \vec{b} = \begin{pmatrix} m_{101} \\ m_{011} \end{pmatrix} = \mathbf{D}^T \vec{d}$$

---

[7]This called least square minimization, which sholuld not to be mixed with total least square minimization.

[8]When we use least squares approach, we should make sure our error function is "convex". All the objective functions we deal with in this tutorial will be of this type, meaning that a global extrema is found by looking for where the gradient of the error is zero.

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

$$\vec{v} = -(\mathbf{D}^T\mathbf{D})^{-1}\mathbf{D}^T\vec{d} = -\mathbf{S}^{-1}\vec{b} \qquad (9)$$

From the previous section we know of cases when $\mathbf{S}$ 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 $\mathbf{S}$ is before inverting (thereby skipping those regions).

According to the original LK algorithm, observations of I_x, I_y and I_t within our region Ω, should all be weighted equally. This would amount to a window function w that is strictly Boolean in value. However, we will use a smooth window function (fuzzy definition of Ω), effectively weighting observations less that are positioned further away from the center of Ω. This is sometimes called a "weighted least squares" approach.

9   Optical Flow - FlowLK.m

The *moment* images will be central to our implementation of optical flow, and you can find how the $m_{200}(\vec{x})$ and $m_{020}(\vec{x})$ are calculated in the function FlowLK.m as:

```
% 1) Compute dx by convolution and form the elementwise product
momentIm = dx.^2; %Note that .^ means elementwise power

% 2) smooth with large seperable gaussian filter (spatial integration)
momentIm = conv2(gg,gg,momentIm,'same');

% 3) downsample to specified resolution:
m200 =  imresizeNN(momentIm ,flowRes);
```

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

```
gaussStd = 1.4;
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 Ω we use.

It will be your task to write the expressions for several $m_{ijk}(\vec{x})$ images in FlowLK.m. Continue reading when this is done.

The moment images makes it possible to define local structure tensors[10], one

---

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

for each neighborhood (around each pixel x) in the image as:

$$\mathbf{S}(\vec{x}) = \left(\begin{array}{cc} m_{200}(\vec{x}) & m_{110}(\vec{x}) \\ m_{110}(\vec{x}) & m_{020}(\vec{x}) \end{array} \right)$$

These should all be inverted in order to estimate motion according to Eq. 9, but we need to check if each matrix is well-conditioned (to make sure we have "nice texture"). One commonly uses the conditional number of a Matrix to see if it is worthwhile to invert. In Matlab, we use the function `rcond` for this, as seen on the line labelled "L1" in `FlowLK.m`.

You are now well prepared to finish the missing code in `FlowLK`. Once you are done, the configuration for running the LK algorithm is:

```
in.method =  @FlowLK;
```

Run the LK algorithm on the synthesized sequence. Does the algorithm perform as you would expect? There are points in the sequence where performance is better than other places due to the texture being nicer. Try to relate your observations to what you know about the aperture problem, and what you could observe from running:

```
ApertureIllustration;
```

9.1 Issues with the classical LK flow method

`FlowLK` sort of works with the synthetic sequence, but we note that regions without nice texture are missing flow estimates. If you run `FlowLK` on real-world data (use recorded video or live feed from a camera by changing `movieType` argument in `runMe`), 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 `FlowLK`, will allow you to tune the algorithm somewhat for different data. However much you tune it, never expect an explicit implementation of the LK algorithm to behave too well on real-world data.

10   Regularization, Temporal Integration and Vectorization - Flow1.m

For a better method of optical flow estimation, set:

```
in.method =  @Flow1;        %Locally regularized and vectorized method
```

Lets take a moment[11] and derive `Flow1.m`. It deals with the two drawbacks of instability and inefficiency of the standard LK implementation. This is where our tutorial starts to scratch the surface of useful algorithms.

10.1 Local Regularization

The aperture problem was addressed by investigating the conditional number (built-in function `rcond`) of the structure tensor, before inverting it. With local regularization[12], we aim to force a change of the conditional number instead of only investigating it. We will use so-called Tikhonov regularization. In practice, we will add a positive value to `m20` and `m02` before we invert. 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.

With this regularization, we are minimizing a different error than the traditional LK. The idea is to add a term to E_{LK} of Equation 3, so that the error is always higher for larger flow vectors ($|\vec{v}|$) thus favoring solutions that are smaller. In the following error we add a term $c|\vec{v}|^2$, and call c our tunable Tikhonov constant:

$$E_1(u, v) = \frac{1}{2} c |\vec{v}|^2 + E_{LK} =$$

$$\frac{1}{2} \left(cu^2 + cv^2 + \sum_{\forall i \in \Omega} (uI_x(\vec{p}_i) + vI_y(\vec{p}_i) + I_t(\vec{p}_i))^2 \right)$$

yielding an error gradient expression:

$$\nabla E_1(u, v) = \ldots$$

$$c \begin{pmatrix} u \\ v \end{pmatrix} + \sum_{\forall i \in \Omega} \begin{pmatrix} uI_x^2 + vI_xI_y + I_xI_t \\ vI_x^2 + uI_xI_y + I_yI_t \end{pmatrix} =$$

$$\begin{pmatrix} m_{200} + c & m_{110} \\ m_{110} & m_{020} + c \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix} + \begin{pmatrix} m_{101} \\ m_{110} \end{pmatrix} = \mathbf{0}$$

Thus, Tikhonov regularization amounts to adding a tunable constant to the m_{200} and m_{020} moments.

10.2 Temporal Integration

So far, we have used integration only over the x and y coordinates of the image for generating the moment images $m_{ijk}(\vec{x})$. With `DoEdgeStrength` we saw how easy it was to incorporate information of previous frames for better estimates. We did this without storing any extra images by recursive filtering. If we do

---

[11] pun intended :)

[12] The keyword "Local" is used here to not confuse the topic with global regularization, and so called variational approaches

the same thing with our moment images, it will make regions of interest, Ω, that stretch into the temporal dimension(t in addition to x and y) giving a more stable tensor field for optical flow estimation. Lets denote the temporally integrated moments by $\hat{m}_{ijk}(x, y, t)$:

$$\hat{m}_{ijk}(x, y, t) = \alpha \hat{m}_{ijk}(x, y, t-1) + (1-\alpha)m_{ijk}(x, y, t)^\gamma$$

As with the edge detection example, we should expect a delayed, temporal blurring effect in the flow estimation if we make the $\alpha \in [0, 1)$ value too large.

In the code of `flow1.m` we use the variable `TC` for our Tikhonov constant (the implementation of c above), and we have `tInt` for our α. The implementation of the Tikhonov regularization and temporal integration is found in the following lines:

```
% 1) make elementwise product
momentIm = dx.^2;

% 2) smooth with large seperable gaussian filter (spatial integration)
momentIm = conv2(gg,gg,momentIm,'same');

% 3) downsample to specified resolution
momentIm =  imresizeNN(momentIm ,flowRes);

% 4) ...  add Tikhonov constant if a diagonal element (for m200, m020):
momentIm =  momentIm + TC;

% 5) update the moment output (recursive filtering, temporal integration)
m200 = tInt*m200 + (1-tInt)*momentIm;
```

Your first task in `flow1.m` is to fill in the expressions for the missing moments. Careful so that that only `m200` and `m020` gets the `TC` constant added, and that the one liners have their brackets correctly placed.

As with the `EPSILONLK` parameter, play around with `TC` to suit your data. `TC` 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

10.2.1 Vectorization

Implementing the LK by using vectorized programming[13] will make use of the built in parallelism in Matlab.

Lets revisit the grand expression for the LK method, Eq. 9. 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[14] by typing:

```
%declares symbolic variables:
syms m200 m020 m110 m101 m011;
```

---

[13]in particular, using no for loops

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

```
b   = [m101; ...
       m011];

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

v   = -S2D\b
```

The output of above code should be implemented on lines labelled "L2" in `Flow1.m`. Make sure you *do not forget the Matlab dot* to indicate elementwise operations when needed.

With a vectorized formulation, we can now run the algorithm at the same resolution as the original video (i.e. one flow vector per pixel). A streamlined m-file for `Flow1.m` is `Flow1Full.m`, found in the folder 'helperFunctions'. Open it up to view its contents. Notice that `Flow1Full.m` is understandable, short, fast and robust compared to the previous implementation of LK. The core algorithm is implemented on the following lines:

```
%%% The regularized, temporally integrated and vectorized flow algorithm:
%Tikhonov Constant:
TC  = single(150);

%generate moments
m200= tInt*m200 + (1-tInt)*(conv2(gg,gg, dx.^2 ,'same')+TC);
m020= tInt*m020 + (1-tInt)*(conv2(gg,gg, dy.^2 ,'same')+TC);
m110= tInt*m110 + (1-tInt)* conv2(gg,gg, dx.*dy,'same');
m101= tInt*m101 + (1-tInt)* conv2(gg,gg, dx.*dt,'same');
m011= tInt*m011 + (1-tInt)* conv2(gg,gg, dy.*dt,'same');

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

It is activated by:

```
in.method =  @Flow1Full;    %Flow1 streamlined for full resolution
```

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 color coding instead, illustrated in figure 3.
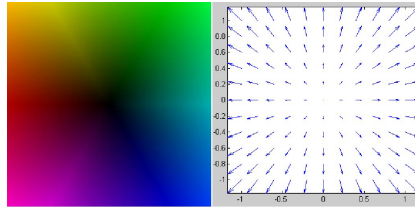

Figure 3: Color coding of the optical flow. Left, the higher resolution, color equivalent of the right side vector version.

We choose to superimpose the optical flow as color on the grayscale video, in order to get a an intuitive feel for the performance. If you wish to see the flow in the traditional color coding (seperate windows, one for flow and one for video),

you can choose to save the output to file and use function `flowPlayback.m` (this is illustrated in `exampleUsage.m`, and will be discussed in more detail below).

11 Some Challenges

We have shown how to implement a dense optical flow algorithm in Matlab (with ease). The final algorithm is very short in code, understandable, (quite) robust, and efficient.

There are several classical issues with optical flow estimation we have yet to discuss, and that our algorithm does not deal with.

11.1 Higher Motion

The first issue is higher motion which we discussed in section 3.1. The standard way to deal with this is through pyramid, multiscale approaches (sometimes called multi-grid solutions). The main problem with that approach is that finer-scale details in the images disappear for the coarser scales of the pyramid.

11.2 Failing the BCC

The starting point of deriving our optical flow method is the BCC. We would expect our algorithm to be quite dependent on it. Important examples of when the BCC fails include varying light field in the scene(such as shadows) and automatic gain control in the camera (including automatic camera parameter setting such as shutter speed).

An important phenomenon, in low-end web cams especially, is the presence of flicker.

11.3 Multiple Motions

The questions of having several motions in the same Ω causes some quite specific problems. The background motion has some distribution of gradients, while the foreground motion has others. A perceived motion over such an aperture is not obvious. Run:

```
ApertureIllustration(4);
```

This renders two bar patterns moving with one partly obscuring the other.

11.4 Noise

As with all sensors, cameras have noise. This is the bane of many optical flow algorithms.

11.5   Generating test sequences

The test sequences we have generated so far has been very kind for optical flow algorithms. In order to render images that

- fail the BCC(through flicker),

- has multiple motions(a strong constant edge in the background) and

- has zero mean, additive Gaussian noise

use the following settings:

```
in.movieType = 'synthetic'; %generate synthetic video
in.method    = 'synthetic'; %generate (Lo Res)groundtruth motion

in.syntSettings.backWeight = 0.3;   %background edge pattern weight
in.syntSettings.edgeTilt   =2*pi/10;%tilt of the edge of the background
in.syntSettings.edgeTiltSpd=2*pi/300;%speed of rotation of background edge

in.syntSettings.flickerWeight= 0.3; %amount of flicker in disks (in [0,1])
in.syntSettings.flickerFreq = 0.6;  %frequency of flicker (in range (0,Inf))

in.syntSettings.noiseWeight = 0.3;  %signal to noise (in range [0,1])
```

12   Extras

In the software we have provided with this tutorial, there are some useful features that will allow you easily to experiment with optical flow, and to test new algorithms as you go along. The most classical algorithmm of so-called variational approaches(global optimization) to optical flow is Horn and Schunk, and it is activated as:

```
in.method = 'HSFull';
```

The toolbox then calls function FlowHS.m, which could be interesting to take a look into.

In the file exampleUsage.m there are a couple of more settings to play around with. It includes a method for saving and reading files from experiments you may want to conduct.

This can be especially useful if you want to save some video for consistent testing.

12.1   Generating Groundtruth motion from synthetic images

We already noticed how to activate the groundtruth motion generation by

```
in.movieType = 'synthetic'; %generate synthetic video
in.method    = 'synthetic'; %generate (Lo Res)groundtruth motion
```

for a high resolution version, with the now familiar color coding:

```
in.method     = 'syntheticFull'; %generate (Hi Res)groundtruth motion
```

In order to save the video and the output flow, we put:

```
in.bRecordFlow = 1;
```

This will automatically generate new subfolders where the full data is stored. The data will be stored in compressed form to save space, and enable streaming reading of it later. A function getSavedFlow has been provided that reads saved experiments. It is used by the function flowPlayBack.m to generate nicely looking playback of saved data:

```
in.bRecordFlow = 1;
[dx, dy, dt,U1,V1,pathToSave] = vidProcessing(in);
flowPlayBack(pathToSave);
```

More examples are given in exampleUsage.