Interactive Tutorial on Optical Flow (BETA)

Stefan Karlsson(Ph.D.Cipherstone, Senior Vision Engineer),
Josef Bigun(Ph.D.Halmstad University, Professor Image Analysis)

November 2014, v1.04b

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

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 entry point for video processing. You can finish this tutorial without looking inside `vidProcessing` but the code is quite easy to understand. 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.

It is 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 in `runMe.m` set:

```
in.movieType = 'camera';
```

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

Once you have finished viewing the results of the video processing, close the window to return back from the function. At that point datastructures `dx, dy, dt` and `U` and `V` are returned. The first three are the numerical estimates of the derivatives of the video sequence ($\frac{\partial I}{\partial x}$,$\frac{\partial I}{\partial y}$ and $\frac{\partial I}{\partial t}$) and the last two are the estimates of optical flow. The derivatives and optical flow returned will all be with respect to the last frame that was viewed (just as the figure was closed).

2 Introduction

NOTE: This is a beta version, and so I expect some issues. I would like to know about them, so that I can improve this major update. We are working on a publication in connection to this, so please check back for a proper reference if you plan on using this for your work.

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. If we did point tracking, we would want to select a few points distributed 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 this tutorial (early 2013) a widespread myth is that dense optical flow can only be achieved at extreme computational cost. We will only scratch the surface in this tutorial but will arrive with ease at implementations that run in real-time, even as pure Matlab implementations without using the GPU. Estimating optical flow that is ***accurate enough***, and stable and robust for many real-world problems is easy.

This tutorial is maintained by Cipherstone Technologies and has come about through work done at Halmstad University. It is intended as a free educational resource. If you are interested in scratching more than just the surface, we provide services for education and production of proprietary software in environments such as C/C++ and Cuda for various platforms.

2.1 Outline

This tutorial consists of completing a few pedagogically selected pieces of code. The code you need to complete will be about 10 lines in total and will aid you in understanding the basic principles of video processing and optical flow. The

code you will modify is found in 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 (this is to aid your grasp of what is to come).
3. DoFlowLK.m, which uses dx, dy and dt to calculate optical flow.
4. DoFlow1.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

The tutorial is an example of a "streaming" application, the preferred paradigm in many video-processing applications when focus is on performance. There is a lot of data and we don't want to shuffle it around needlessly or store it in more places than we need. The key is to have only what you need, and only when you need it: process and discard data in rapid fashion.

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, replacing the old data as new data is provided. This is implemented through persistent variables, the most important one being imPrev for the storage 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.

The code for grad3D.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 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
```

Do this, and re-run runMe.m. Use the synthetic video.

Are the gradient components as you expect them to be? Can you see a relationship between them during motion? How does the motion affect the `dt` image compared to the `dx` and `dy`?

If you notice a relation between `dt` and `dx`, `dy` images try to put it into words, if only as a mental note. Odds are that the relationship you see is the so-called "optical flow constraint".

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[2]. 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:

```
 %dt, dx and dy, has just been set as you closed the rendering figure
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 $\phi = 0$ and $\phi = 90^o$

[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

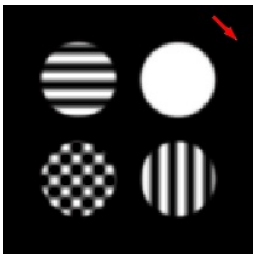

Figure 1: the position of the pattern as it is moving with direction $\phi = 45^o$. This is what you should see when shutting down the figure(time it well). Red arrow shows the motion vector.

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. Get an idea of how the derivatives change as a result.

Make the motion in the sequence 10 times faster. Viewing and shutting down the figure at the right time may now prove difficult. An extra lag time (delay between each frame that does not affect the optical flow) 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:

```
in.bFineScale=0;          %in order to use coarse scale derivatives.
```

Run the experiment at a higher motion as before.

Are the two images `dt` and `-(dx+dy)` more similar now?

3.1.1 Take-Home Message

Higher motions makes the optical flow constraint equation (Eq. 1, equivalently Eq. 2) fail. This can be compensated for by using a coarser scale of derivatives[5].

[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

However, coarser scale destroys some of the important detail in the image[6].

4 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. Let the 2D gradient of the image be $\nabla_2 I(\vec{x}) = \{I_x(\vec{x}), I_y(\vec{x})\}$, then the edge strength P is defined as the value: $P = |\nabla_2 I| = \sqrt{I_x^2 + I_y^2}$. `DoEdgeStrength` should return an image: the strength of edges at each position.

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

This first implementation of `DoEdgeStrength` 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:

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

4.1 Gamma Correction

It is in general not well-defined what is meant by an edge in an image: it depends on the application. A way to change the sensitivity of our detector and let more candidate edges 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 correctly implemented version of `DoEdgeStrength.m`, there is the possibility to modify the γ parameter interactively using the keyboard (**R**/**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. With our streaming application, storing more images of the video sequence is not desirable but there are other ways. One such trick, that does not take up more memory, is a recursive filter[8]. Before we approach optical flow, lets try this principle on our edge detection algorithm.

[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.

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

[8] first order linear recursive filter to be exact

DoEdgeStrength receives 4 arguments dx, dy, tInt and edgeIm(the previous output of DoEdgeStrength). The idea is simple: 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 tInt. 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;
```

Hopefully, it is intuitively clear what is happening. Having such a high integration is not very useful for edge detection[9], but if you put it to a lower value, such as 0.2, you will see a reduced amount of noice. This should be especially clear when you have a connected webcam, and with low γ values.

5 Aperture Problem

Any region Ω where an optical flow 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 (vertical and horizontal 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).

It is obvious that a region of constant value is bad for motion estimation, there simply isn't any observation 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

[9] for other computer vision challenges, such as background subtraction, this could make sense however

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 $\nabla I(\vec{x_i})$. 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.

This phenomenon is often referred to as the *aperture problem*.

5.1 2D Structure Tensor

Whether a region is "nice" or "bad" is given by the so-called 2D structure tensor:

$$S_{2D} = \begin{pmatrix} \iint \left(\frac{\partial I}{\partial x}\right)^2 \mathrm{d}\vec{x} & \iint \frac{\partial I}{\partial x}\frac{\partial I}{\partial y}\mathrm{d}\vec{x} \\ \iint \frac{\partial I}{\partial x}\frac{\partial I}{\partial y}\mathrm{d}\vec{x} & \iint \left(\frac{\partial I}{\partial y}\right)^2 \mathrm{d}\vec{x} \end{pmatrix} = \begin{pmatrix} m_{20} & m_{11} \\ m_{11} & m_{02} \end{pmatrix}$$

A region is "nice" 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 region (both eigenvalues of S_{2D} are zero)
- linear symmetry region (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 $m_{11} = \sum wI_xI_y$, 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 $m_{ijk} = \sum wI_x^iI_y^jI_t^k$. 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 $m_{11}(\vec{x})$ to indicate positioning of Ω at $\vec{x}$.

6 Optical Flow - DoFlowLK.m

The 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 `DoFlowLK.m` as:

```
% moment m200, calculated in 3 steps explicitly
% 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:
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 = 1.8;
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[10] 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 `DoFlowLK.m`. Continue reading when this is done.

The moment images makes it possible to define local structure tensors[11], one for each position in the image as:

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

This matrix is important for optical flow. It will pop up naturally in the derivation of the Lucas-Kanade (LK) optical flow method below.

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

6.1 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 standard numerical approach for when we have more observations then parameters to solve for. 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" to all the observations at all positions in the region Ω. We wish to find the solution that minimizes the error:

$$E_{LK}(u,v) = \frac{1}{2}\sum_{\forall i \in \Omega} \left(uI_x(\vec{p}_i) + vI_y(\vec{p}_i) + I_t(\vec{p}_i)\right)^2 \tag{3}$$

To find the minimizer, we equate the gradient $\nabla E_{LK} = \left(\frac{\partial E_{LK}}{\partial u}, \frac{\partial E_{LK}}{\partial v}\right)^{\mathrm{T}}$ to zero[12]:

$$\nabla E_{LK}(u,v) = \sum_{\forall i \in \Omega} \begin{pmatrix} uI_x^2 + vI_xI_y + I_xI_t \\ vI_x^2 + uI_xI_y + I_yI_t \end{pmatrix} = \\ \begin{pmatrix} m_{20} & m_{11} \\ m_{11} & m_{02} \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix} + \begin{pmatrix} m_{101} \\ m_{110} \end{pmatrix} = \mathbf{0} \tag{4}$$

where we remind the reader of the notation $m_{ijk} = \sum w I_x^i I_y^j I_t^k$ for a window function w covering Ω. We recall the structure tensor S_{2D}, and introduce $\vec{b}$:

$$S_{2D} = \begin{pmatrix} m_{20} & m_{11} \\ m_{11} & m_{02} \end{pmatrix}, \qquad \vec{b} = \begin{pmatrix} m_{101} \\ m_{011} \end{pmatrix}$$

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

$$\vec{v} = -S_{2D}^{-1}\vec{b} \tag{5}$$

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, as seen on the line labelled "L1" in `DoFlowLK.m`.

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

[12] Whenever we try such an approach, we should first 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 is zero

```
in.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? Relate your observations to what you know about the aperture problem.

6.1.1 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 `DoFlowLK`, will allow you to tune the algorithm somewhat for different data. However much you tune it, never expect this explicit implementation of the LK algorithm to behave too well on real-world data.

7 Regularization, Temporal Integration and Vectorization - DoFlow1.m

Lets take a moment[13] and derive `DoFlow1.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.

7.1 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[14], we aim to force a change of the conditional number instead of only investigating it. The simplest, most commonly used approach for this is so-called Tikhonov regularization. In practice, we will just 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

[13] dreadful pun :)

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

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:

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

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

$$\begin{pmatrix} m_{20}+c & m_{11} \\ m_{11} & m_{02}+c \end{pmatrix}\begin{pmatrix} u \\ v \end{pmatrix} + \begin{pmatrix} m_{101} \\ m_{110} \end{pmatrix} = \mathbf{0}$$

Thus, Tikhonov regularization amounts to adding a tunable constant to the m_{20} and m_{02} moments.

7.2 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 a simple recursive filtering. If we do 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 → numerically unstable, ill-conditioned solutions (the flow field will wiggle around and explode occasionally),
- make it too large → all flow vectors will tend to shrink in magnitude

7.2.1 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. 5. 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:

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

b   = [m101; ...
       m011];

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

v   = -S2D\b
```

The output of above code should be implemented on lines labelled "L2" in `DoFlow1.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 `Doflow1.m` is `DoFlow1Full.m`, found in the folder 'helperFunctions'. Open it up to view its contents. Notice that `DoFlow1Full.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:
```

[15] in particular, using no for loops

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

```
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:

```
%this will enable full resolution flow, with color coding:
 in.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 color coding instead, illustrated in figure 3.

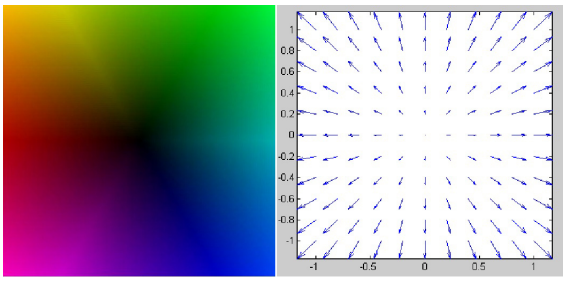

Figure 3: Color coding of the optical flow. Left, the higher resolution, color equivalent of the right side vector version.

We choose to superimpose the optical flow as color on the original 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).

8 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.

8.1 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.

8.2 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.

8.3 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 entirely obvious. Run:

```
ApertureIllustration(3);
```

This renders two bar patterns moving with one partly obscuring the other. To see what this will amount to when color is present, run:

```
ApertureIllustration(4);
```

8.4 Noise

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

8.5 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])
```

9 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 `DoFlowHS.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.

9.1 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`.