Interactive Tutorial on Real-Time Optical Flow Motion Estimation

Stefan Karlsson, (Ph.D.),
Josef Bigun, (Ph.D. Professor)

Contact:stefan.karlsson987(AT)gmail.com

Date: May 2015, v1.05 (First version: June 2013)

Illustration: Color coded optical flow. Left is higher resolution equivalent of the right.

Let's get started

Run the script `runMe.m` to display interactive synthetic video, with green arrows displaying optical flow. The figure's title bar displays the state of the rendering.

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

Stop the session by shutting down the figure. The **"Fancy Flow Player"** will start playback of the recording. It displays motion as color (see cover figure), and has a handy keyboard and mouse interface (see `FancyFlowPlayer.m` for details). You can interface with the seekbar by clicking and dragging. You can zoom and pan in the view windows using mouse scroll and mouse dragging.

`runMe` has one purpose: to set up the call to `vidProcessing`, the 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[1], set:

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

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

[1] Image acquisition toolbox recommended. However, on windows you can get camera input without it

Chapter 1

The Toolbox and Local Optical Flow

1.1 Introduction

We provide 2 resources: a toolbox (in Matlab) and a tutorial (this PDF).

- The toolbox is about ***dense optical flow*** motion algorithms.
- The tutorial is about ***gradient based dense optical flow***.

With ***dense*** estimation we are interested in motion at fixed positions in the image. Another way to measure motion, *not* dealt with here, is by tracking points from one frame to the next. With tracking, points are selected (automatically or manually) and their positions detected in new frames. With point tracking, positions are processed that change over time. With dense optical flow, the positions are fixed.

The tutorial deals with the ***gradient based*** approach to optical flow, but there are many other methods[1]. The toolbox is well suited for use with any approach, as long as it is dense motion.

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 an educational resource as well as a toolbox for experimenting with motion algorithms. If you find this resource useful (or want to suggest changes) tell me about it.

1.1.1 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] good examples of non-gradient based methods include *block-matching* and *phase based* methods

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.

On our path to getting this done, we will learn the basics of optical flow and its 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.

1.2 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` 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
in.bRecordFlow = 0;         %turn off the recording
```

Execute `runMe`, with the synthetic video input selected.

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 a relationship called **"optical flow constraint"** (read on).

[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

1.2.1 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:

$$I_t + vI_x + uI_y = 0 \tag{1.1}$$

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:

$$I_t = -|\vec{v}| \left(\cos(\phi)I_x + \sin(\phi)I_y\right) \tag{1.2}$$

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]. Now, run again the script `runMe`, with the same settings as before (use synthetic image sequence).

Does the optical flow constraint 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 (**P**) as the pattern is moving at an angle $\phi = \pi/4 = 45^o$ (this is when the pattern is moving **towards the lower right corner**[4] as indicated in figure 1.1. When paused, enter the following code into the matlab command prompt:

```
% get the derivatives from the toolbox (assumed still running)
[dx, dy, dt] = getCurrentGrad3D();

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

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. 1.2.

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 (**Q**/**A**). Pause just as the motion is $\phi = \pi/4$ as before, and run the same code as before to display `dt` and `cos(phi)*dx+sin(phi)*dy`.

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

[4] Matlab has the origin at the top left corner, with the y axis is pointing downwards

Figure 1.1: the position of the pattern as it is moving with direction $\phi = \pi/4$. This is what you should see when pausing (time it well). Red arrow shows the motion vector.

This time the images are not very similar. The optical flow constraint is valid as an approximation only when the motion is small.

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`):

```
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.

Choosing and tuning gradient filters

There are many ways of estimating gradients, and what is the best method depends on what we wish to use it for. In the end we care about the final algorithm in terms of **stability**, **accuracy** and **timeliness**[5].

The optical flow constraint equation should be a guiding principle for testing gradient estimation when the aim is motion. This can be done in the fashion outlined in section 1.2.1, and does not need for any explicit optical flow to be calculated.

On that topic, we can once more re-visit `grad3d`. 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 usually 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-

[5]often the terms 'speed', 'efficiency' and/or 'performance' are used instead of timeliness, but these can be mis-interpreted in our context

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.

1.3 Edge filtering (`DoEdgeStrength.m`)

The second coding task is edge detection. This is a common task in video processing for motion and other challenges. The function `DoEdgeStrength` has 5 inputs, and a single output:

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

As input we have the structure `in` set in `runMe` and the new and the previous video frames (`imNew`, `imPrev`). `gam` is a normalization constant we can set interactively by keyboard (**R**/**F**) and `edgeIm` (as input) is the former output of the function.

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 can be the value: $P = |\nabla_2 I| = \sqrt{I_x^2 + I_y^2}$.

The first implementation of `DoEdgeStrength` will be just 2 lines. Estimate the gradient, and use it as indicated above. When you are done, view the edge detection in realtime[6]:

```
% in runMe.m
in.Method = 'edge';
```

1.3.1 Edge normalization

What is meant by an edge depends on the application. A way to change the sensitivity of our detector is by $P = |\nabla_2 I|^\gamma$. Another approach, which is more robust to noise is:

$$P = \frac{m|\nabla_2 I|^2}{|\nabla_2 I|^2 + m\gamma^2}$$

[6] this is especially fun if you have a camera and can view yourself

where m is the maximum possible value for the gradient magnitude[7]. If you use the original `grad3D`, then $m = 512$. Implement this (using the input `gam` for γ), and then set different γ values interactively using the keyboard (**R**/**F**).

1.3.2 Temporal integration

Algorithms for video processing are often improved by using more images of the sequence. However, using more memory is not desirable. A trick: 1st order recursive filtering allows us to include more temporal information with no extra memory required. Before we approach optical flow, lets try this on our edge detection algorithm.

`DoEdgeStrength` receives a fifth input argument `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 should see a reduced amount of noise. Try with a connected webcam, and setting a low γ value.

1.4 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. One of the textures is nice (checkerboard pattern), two are linear symmetric (bars) and one is constant (a white 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 linearly symmetric textures? Why are they so bad? Run the function:

```
ApertureIllustration;
```

[7] e.g consider an 8 bit image and a 2 point derivative filter. This gives $\max(I_x^2) = \max(I_y^2) = 256^2$ and $m = \sqrt{\max(I_x^2) + \max(I_y^2)} \approx 362$

In it, a region Ω (the "Aperture") 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 1.2). To see an animated version of the barberpole illusion, type:

```
ApertureIllustration('barber');
```

Figure 1.2: 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 known as the *aperture problem*.

1.4.1 2D Structure Tensor

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

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

A region is "nice" if both eigenvalues are large. This is equivalent to S_{2D} being "well conditioned", i.e. it 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 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 $m_{110} = \sum w I_x I_y$, where w is a window function($w \in [0,1]$) 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 $\vec{x}$.

1.5 Optical flow by Lucas and Kanade

Start by considering the optical flow constraint equation (Eq. 1.1 or equivalently Eq. 1.2). 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:

$$E_{LK}(u,v) = \frac{1}{2} \sum_{\forall i \in \Omega} \left(u I_x(\vec{p}_i) + v I_y(\vec{p}_i) + I_t(\vec{p}_i) \right)^2 \tag{1.3}$$

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[8]:

$$\nabla E_{LK}(u,v) = \sum_{\forall i \in \Omega} \begin{pmatrix} u I_x^2 + v I_x I_y + I_x I_t \\ v I_x^2 + u I_x I_y + I_y I_t \end{pmatrix} = \begin{pmatrix} m_{200} & m_{110} \\ m_{110} & m_{020} \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix} + \begin{pmatrix} m_{101} \\ m_{110} \end{pmatrix} = \mathbf{0} \tag{1.4}$$

[8]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 of the error is zero

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 $\vec{b}$:

$$S_{2D} = \begin{pmatrix} m_{200} & m_{110} \\ m_{110} & m_{020} \end{pmatrix}, \qquad \vec{b} = \begin{pmatrix} m_{101} \\ m_{011} \end{pmatrix}$$

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

$$\vec{v} = -S_{2D}^{-1}\vec{b} \tag{1.5}$$

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 skipping those regions).

1.6 Optical Flow (`FlowLK.m`)

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.

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 `FlowLK` 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 = 1.4;
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`. Continue reading when this is done.

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

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

$$S_{2D}(\vec{x}) = \begin{pmatrix} m_{200}(\vec{x}) & m_{110}(\vec{x}) \\ m_{110}(\vec{x}) & m_{020}(\vec{x}) \end{pmatrix}$$

These should all be inverted in order to estimate motion according to Eq. 1.5, but we need to check if each matrix is well-conditioned ("nice texture"). In Matlab, we use the function `rcond` for this (Matrix conditional number), as seen on the line labelled "L1" in `FlowLK`.

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;
```

1.7 Performance evaluation

`FlowLK` works poorly. It gives us better than random guesses on the synthetic sequence, but we note that regions without nice texture are missing.

The toolbox generates ground-truth optical flow to aid in evaluation. To enable viewing of ground-truth together with your estimated flow, set:

```
in.bDisplayGT = 1;           %display groundtruth flow, if available
```

This will plot red vectors, behind the green ones (in case of perfect estimation, no red will show). With `FlowLK`, the difference between estimation and ground-truth is high (commonly called the end-to-end point error). Also note that the estimations wiggle around more than the ground-truth, and occasionally explodes. A small change in data, does not guarantee a small change in output - `FlowLK` is not very stable. 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.

Its a good idea to try the algorithm with live feed from a camera, or on recorded video by changing `movieType` argument in `runMe`. This should further convince you that the implementation suffers from:

1. Low **accuracy** (deviates from groundtruth, ignores linear symmetry regions)

[10] a.k.a tensor field

2. Low **stability** (wiggles around and explodes on occasion)

3. Low **timeliness** (Inefficient, slow computations)

1.7.1 Problems with Evaluation

Quantifying performance of optical flow can be a tricky business. The metric with least complications is the so-called **interpolation error**. This error deals with how well we can use the flow to transform the first image (`imPrev`) into the second (`imNew`). This error is useful to applications such as video-compression and interpolation.

Interpolation error is not well suited for applications such as tracking and ego-motion, where the flow is used as a measure of real-world motion. The most popular error metric in the computer vision field, is the **end-to-end point** (vector) accuracy (distance between estimation and ground-truth). For this kind of evaluation, synthetic sequences are usually required. A good example of long and rich synthetic sequence with ground-truth is the Sintel dataset, the classical ground-truth data to use is the Middelbury dataset.

Both interpolation and end-to-end point error fall short for applications such as human-machine-interaction and robotic self stabilization, where stability and timeliness are more important than high accuracy. As an example close at hand, consider an optical mouse. With some of the earlier mice to appear, the mouse cursor could wiggle and jump around tiny distances seemingly random as you moved it smoothly. Erratic behavior was more likely during motion over strong linear patterns when the mouse was also rotating (which does not add to the translational motion a mouse is designed to look for). These erratic occurrences would be relatively rare, but would interrupt the natural interaction with the device. The optical mouse had high accuracy and timeliness, but inadequate stability. Later generations of mice became better, because smoothness of hand would better give smoothness of cursor, even though accurate motion was not estimated. State of the art algorithms, as measured on Middelbury dataset, typically perform poorly in applications such as the faceMouse and the squiggle for the same reason.

1.7.2 Comparing methods, saving data

For a better method of optical flow estimation, lets re-activate `Flow1`:

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

Running this algorithm together with ground-truth flow shows large improvements in accuracy, stability and timeliness, compared to `FlowLK`. Lets expose `Flow1` to a bit more challenging data, by introducing a background edge and some noise in our synthesis:

```
in.syntSettings.backWeight = 0.7;     %background edge
in.syntSettings.noiseWeight = 0.2;    %signal to noise weight (in range [0,1])
```

Its valuable to have live interaction in the synthesis, but we can never be sure to generate exactly the same data twice. Lets save some data with ground-truth, and use it repeatedly.

Step 1: Save ground-truth data

Put these settings in `runMe`, and when running use the regular keyboard interface to affect the sequence during recording.

```
% notice that both "movieType" AND "method" need to be set to 'synthetic'
in.movieType  = 'synthetic';       % Generate synthetic video
in.method     = 'synthetic';       % Ground-truth optical flow

%set recording:
in.pathToSave = 'GroundTruthData'; % Define directory for saving
in.bRecordFlow = 1;                %record the video and flow
```

Step 2: View and access data

After you have saved data to the folder "GroundTruthData", you can access it easily using a combination of `FancyFlowPlayer` and `getSavedFlow`:

```
% first, find a frame you are interested in, and then close
% down the FancyFlowPlayer...
frameNr = FancyFlowPlayer('GroundTruthData');

% ... then access the data of the frame:
[im, u, v] = getSavedFlow(frameNr, 'GroundTruthData');

% im is the video frame
subplot(1,2,1);imagesc(im); colormap gray;
axis image;title('video frame');

% u and v are the components of the flow
subplot(1,2,2);imagesc(sqrt(u.^2+v.^2));colormap gray;
axis image; title('Motion magnitude(groundtruth)')
```

Step 3: Use saved data with the toolbox

Lets use our saved ground-truth data and experiment on the LK implementation.

```
in.movieType ='GroundTruthData';  % Folder with previously saved video
in.endingTime='eof';              % Indicates when to end the session.
                                  % 'EOF'- end when reached end of data
in.method =  @FlowLK;             % lets experiment on LK

%set recording:
in.pathToSave ='TestLK';          % Define directory for saving
in.bRecordFlow= 1;                %record the video and flow
```

With `in.endingTime='eof'` the session will end when the end of the data file is reached. This means that two folders "GroundTruthData" and "TestLK" will contain optical flow of exactly the same video (one estimation by `FlowLK`, the other ground-truth). If you leave the figure open until "EOF" is reached,

then both folders contain data of the same duration. Example of exploring the results:

```
frameNr = FancyFlowPlayer('TestLK');

% Access data:
 imPrev           = getSavedFlow(frameNr-1, 'TestLK'); %Previous frame
[imNew, u  , v  ] = getSavedFlow(frameNr,   'TestLK'); %New frame
[~    , uGt, vGt] = getSavedFlow(frameNr, 'GroundTruthData');

% Display data:
subplot(2,2,1);imagesc(imPrev); colormap gray;
axis image;title('video frame, (t-1)');
subplot(2,2,2);imagesc(imNew); colormap gray;
axis image;title('video frame, (t)');
subplot(2,2,3);imagesc(sqrt(u.^2+v.^2));colormap gray;
axis image; title('Motion magnitude(Lucas Kanade)');
subplot(2,2,4);imagesc(sqrt(uGt.^2+vGt.^2));colormap gray;
axis image; title('Motion magnitude(Ground-truth)');
```

1.8 Flow1.m

Lets take a moment and derive Flow1. It improves on FlowLK by *local regularization*, *Temporal Integration* and *Vectorized coding*.

1.8.1 Local Regularization

The aperture problem was addressed by investigating the conditional number (rcond), before inverting S_{2D}. With local regularization[11], 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 1.3, 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:

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

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

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

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

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

1.8.2 Temporal Integration (Optical flow)

So far, we have only used spatial integration (Ω stretching over x and y) for generating the moment images $m_{ijk}(\vec{x})$. For the edge detection (section 1.3.2) we integrated previous frames by recursive filtering. We can do the same thing in our optical flow estimation on at least 3 different levels: ***gradients***, ***moments*** and ***motion vectors***. The most straightforward(but not the best) approach would be to apply it to the the motion vectors, the end result of our algorithm. We chose to put the filtering on the *gradient* and *moment* level instead.

Parts of our algortihm amplifies the effect of noise alot (they are non-linear). We want to get best performance with as little temporal blurring as possible (recall the effect on edge detection when $\alpha \in [0, 1)$ is high). We have 2 non-linear parts of our algorithm: 1) making moment images, and 2) inverting matrices, so we do filtering just before these non-linear steps are applied.

For the moment images, the filtering will make regions of interest, Ω, that stretch into the t direction in addition to x and y, giving a more stable tensor field. Lets denote the temporally integrated moments by $\hat{m}_{ijk}(x, y, t)$:

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

On the gradient we can choose a different α. We have found empirically that $\sqrt{\alpha}$ works well for the filtering of the gradient, which relates the two recursive filterings with each other through a single parameter:

$$\hat{I}_x(x, y, t) = \sqrt{\alpha}\hat{I}_x(x, y, t-1) + (1-\sqrt{\alpha}) I_x(x, y, t)$$

Implementation

Gradient estimation by this principle is provided in function `grad3Drec`, which is used by `Flow1`. The recursive moment estimation is part of the coding task in `flow1` 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 filtering)
m200 = tInt*m200 + (1-tInt)*momentIm;
```

Your first task in `flow1` 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

1.8.3 Vectorization

Implementing the LK by using vectorized programming[12] will make use of the built in parallelism in Matlab.

Lets revisit the grand expression for the LK method, Eq. 1.5. 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[13] by typing:

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

b   = [m101; ...
       m011];

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

v   = -S2D\b
```

For readers who do not have the symbolic toolbox, the answer will be:

```
v =
  (m011*m110 - m020*m101)/(- m110^2 + m020*m200)
 -(m011*m200 - m101*m110)/(- m110^2 + m020*m200)
```

The output of above code should be implemented on lines labelled "L2" in `Flow1`. You will need to use the Matlab dot-operator to indicate elementwise operations for this to work.

With a vectorized formulation, we can run the algorithm at the same resolution as the original video (i.e. one flow vector per pixel). Resolution of the flow field is controlled by the argument `flowRes`. Set in `runMe`:

[12] in particular, using no for loops

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

```
in.vidRes  = [200 200];   %video resolution
in.flowRes = in.vidRes;   %flow resolution
```

When dealing with full resolution optical flow the toolbox uses the color coding of figure **??**.

Color coded flow is superimposed on the grayscale video. This trick allows the observer to focus on only one view (as opposed to the traditional 2 views of Funky Flow Player). This makes interaction and overview easier at the cost of some detail. If you wish to see the flow in the traditional color display (seperate views), save the output and view it with Funky Flow Player.

1.9 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) stable, and with a high timeliness (it's the only real-time optical flow algorithm we know of, entirely implemented in Matlab[14]).

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

1.9.1 Noise

As with all sensors, cameras have noise. This is the downfall of many optical flow algorithms. To add noise to the synthetic video:

```
in.syntSettings.noiseWeight = 0.2;    %signal to noise weight (in range [0,1])
```

1.9.2 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. We can add flicker to the synthetsis by:

```
in.syntSettings.flickerWeight= 0.8;  %amount of flicker    (in range [0,1])
in.syntSettings.flickerFreq  = 0.8;  %frequency of flicker (in range (0,Inf])
```

1.9.3 Higher Motion

Higher motion was discussed in section 1.2.1. The standard way to deal with this is through pyramid, multiscale approaches (sometimes called multi-grid

[14] Many implementations have convenient wrappers for Matlab, such as the computer vision toolbox, but they are not implemented in native Matlab code

solutions). Coarsest scale of the pyramid is estimated for motion first, where robustness for high motion comes at the price of image detail. The pyramid can be used in combination with iterative approaches to optimize for accuracy.

However, an even greater challenge with higher motion (in low-end cameras especially) is motion blur. The multi-scale approach exacerbates the problem rather than address it[15].

1.9.4 Optimizing for accuracy

At the onset, we use the optical flow constraint (Eq. 1.1), which is an approximation that almost never hold exactly. We can achieve better accuracy by iteratively applying the same method (`Flow1`, `FlowLK` or any method based on Eq. 1.1). If we interpolate ("warp") `imPrev` using the estimated motion, we yield a prediction of `imNew`. We can let this prediction replace `imPrev`, and repeat the same algorithm. A new flow field results, which is (hopefully) smaller in magnitude than the first. The new flow field is added to the old, and used to warp imPrev into a new prediction, and the iterations proceed.

With advantage, this approach is implemented in multi-scale, so that the first iterations are done in coarse scale, while gradually descending to finer scale. The main cost to timeliness lies in that these steps invoke the entire method (e.g. `FlowLK`), and that each step depends critically on the previous, leaving no room for parallelization. Each step also introduces image interpolation (warping) as an added costly operation.

1.9.5 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. You may have noticed the effect when we activated the rendering of a background edge. To make it even more observable, you can synthesize a moving background edge by:

```
in.syntSettings.backWeight = 0.5;       %background edge pattern
in.syntSettings.edgeTiltSpd=2*pi/300;   %speed of rotation of background edge
```

A perceived motion over such an aperture is not obvious. Run:

```
ApertureIllustration('multiple');
```

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

[15] Motion blur is one of those effects that makes you marvel at the human visual system. Its presence improves our perception and reaction times. For most people outside the fields of cognition/perception and computer graphics, it seems, motion blur is treated as an evil source of noise. Some have made attempts to use the motion blur as a source of information for their motion algorithms, but usually with little success, and always at the level of higher vision. Humans apparently use motion blur already in lower level vision, and very successfully so. Perhaps a reader will be encourage to find out how it all really works in the ganglion cells and beyond. If you do, get back to me one day and share.

1.9.6 Filling in the gaps of local flow

The interior sections of objects may lack nice texture, giving gaps where no optical flow is estimated. The only way to fill these gaps with local algorithms is to make the Ω regions larger, which in turns destroys the fine detail required near the motion boundaries. There is no way to solve this(to my knowledge) unless we dive into the topic of Global algorithms. This will be the topic of the next chapter.