Optical flow computation

Abstract

This package implements the ECCV 2004 best paper award winning work "High accuracy optical flow using a theory for warping" [3], and its variation that appeared in the work "Particle Video" [7]. This documentation provides information on how to use the code, and some theoretical assumptions that I made while coding, that do not appear in either of these papers. The first part of the document (Using the Code) is for people whose only purpose is to use the code and its results, and do not want to modify the algorithm / code. The second part related to the formulation is for people who would want to understand the code and modify it for their own use.

1. Using the Code

The main file that starts the computation of optical flow is the file `optic_flow_brox.m` or equivalently `optic_flow_sand.m` The first few lines of `optic_flow_brox.m` read like this

```
function [u, v] = optic_flow_brox(img1, img2)

alpha = 30.0 ; % Global smoothness variable.
gamma = 80.0 ; % Global weight for derivatives.

[ht, wt, dt] = size( img1 ) ;

num_levels = 50 ; % for face; Number of pyramid levels. Can also be calculated
  % automatically

im1_hr = gaussianRescaling( img1, power( 0.95, num_levels ) ) ; % Rescaling images
im2_hr = gaussianRescaling( img2, power( 0.95, num_levels ) ) ; % to the top of the
% laplacian pyramid.
```

These lines do the following

```
function [u, v] = optic_flow_brox(img1, img2)
```

The images used for computing optical flow. They must be RGB images of class `uint8`

```
alpha = 30.0 ; % Global smoothness variable.
gamma = 80.0 ; % Global weight for derivatives.
```

These variables set the importance of the smoothness and gradient terms in the energy functional. The smoothness term ensures that the optical flow computation is extended to textureless regions by hole-filling from the neighbourhood, and the gradient term ensures that the difference in image gradients is also minimized along with the intensity difference. The gradient term is useful when the two images are captured under slightly varying lighting conditions. I havent yet tested the code during severe changes. By default, the weight of the intensity term is set to 1.0. See the next section on details of these terms.

```
num_levels = 50 ; % for face; Number of pyramid levels. Can also be calculated
  % automatically
```

The algorithm [3] computes optical flow when large motion occurs between images, by constructing an image pyramid. The coarsest level typically consists of images of the order of ˜20x20 pixels. Optical flow computed at coarser levels are used to initialize the lower, finer level flows. The variable `num_levels` determines the number of levels in the image pyramid. Currently each image in the image pyramid is 0.95 times the image below it, as indicated in the code below.

```
im1_hr = gaussianRescaling( img1, power( 0.95, num_levels ) ) ; % Rescaling images
im2_hr = gaussianRescaling( img2, power( 0.95, num_levels ) ) ; % to the top of the
% laplacian pyramid.
```

The rest of the code calls different functions to compute the flow. Most parameters that are used in the remaining code are fixed and there is hardly any need to modify them. Only two other parameters need modifications.

```
[du, dv] = resolutionProcess_brox( Ikz, Ikx, Iky, alpha, gamma, 1.8, u, v, 3, 500 ) ;
```

Here, the argument 1.8 represents the *relaxation* parameter, and it can have values strictly in the range ˜1.0-2.0. See [1] for more details on the optimization technique and related code (`sor.m`). The other two arguments 3 and 500 represent the number of outer and inner fixed point iterations respectively. See the paper [3] or the thesis [2] for more information on this aspect. Typical values for these parameters are in the range ˜5-10 for outer iterations and ˜100-500 for the inner iterations. The current parameters are adopted from [7].

The code in `optic_flow_sand.m` has two main differences from the above code.

```
alpha_global = 10.0 ;
alpha_local = 15.0 ;
```

These parameters specify the global smoothness value and the local smoothness. The global smoothness value is the same as the alpha parameter in `optic_flow_brox.m`. The local smoothness parameter specifies the smoothness criterion in an gradient dependent manner, parts of the image having edges and texture have lower local smoothness than textureless regions. This ensures that optical flow discontinuities are preserved near edges, which is where they usually occur due to motion / occlusion etc. The second difference comes in the way `num_levels` is computed.

```
num_levels = round( log10( 30 / min( ht, wt ) ) / log10( 0.9 ) )
```

Here, the parameter 30 refers to the minimum image width or height in the laplacian pyramid. 0.9 specifies the scale factor relating adjacent images of the laplacian pyramid. You may change these values to suit your needs. `num_levels` is not automatically determined in `optic_flow_brox.m` because I often need to tweak with it.

After this point, the code is pretty much very readable with lots of comments explaining what every step does. The final output of `optic_flow_brox.m` are (u, v, im1_hr, im2_hr, im2_orig) where (u, v) represent the computed optical flow, (im1_hr, im2_orig) represent the two input images and im2_hr represents the second image warped towards (to look like) the first image im1_hr. Similarly, outputs of the file `optic_flow_sand.m` are (u, v, im1_hr, im2_hrw, im2_hr) in the same order/manner as `optic_flow_brox.m`.

2. Code Related Aspects

The main energy functional minimized for the computation of optical flow is a combination of appearance and smoothness terms of the images involved. Optical flow typically tries to compute a flow field that minimizes the difference between the two images involved. Extension to more than two images is then straightforward, and can be adopted from [2, 4]. The main energy functional can be expressed as

$$E(u,v) = E_{Data}(u,v) + \alpha E_{Smooth}(u,v) \tag{1}$$

$$E_{Data}(u,v) = \int_{\Omega} \Psi \left(|I(\mathbf{x}+\mathbf{w}) - I(\mathbf{x})|^2 + \gamma |\Delta I(\mathbf{x}+\mathbf{w}) - \Delta I(\mathbf{x})|^2 \right) \mathbf{dx} \tag{2}$$

$$E_{Smooth}(u,v) = \int_{\Omega} \Psi \left(|\Delta_3 u|^2 + |\Delta_3 v|^2 \right) \mathbf{dx} \tag{3}$$

Here, α represents the regularization term. Generally it is set to a high value when smooth textureless surfaces are involved. Typical values are in the range of ˜15-30. The variable γ determines the importance of the gradient values in the image over the texture and smoothness terms. Typical values are in the range˜20-100. The main reason to be careful while selecting these values is that higher values not only give undue importance to the smoothness/gradient terms, but also *undermine* the importance of the main data term, the difference of intensities.

A more general functional is described in [7], which modifies the data and smoothness terms to

$$
\begin{aligned}
E_{Data}(u,v) &= \int_\Omega \sum_c \Psi\left([I^{[c]}(\mathbf{x}+\mathbf{w}) - I^{[c]}(\mathbf{x})]^2\right) && (4)\\
E_{Smooth}(u,v) &= \int_\Omega (\alpha_g + \alpha_l \cdot b(\mathbf{x})) \cdot \Psi\left(|\Delta_3 u|^2 + |\Delta_3 v|^2\right) \mathbf{dx} && (5)\\
\mathbf{x} &= (x,y,t)^\top && (6)\\
b(\mathbf{x}) &= N(\sqrt{I_x(\mathbf{x})^2 + I_y(\mathbf{x})^2}; \sigma_b) && (7)
\end{aligned}
$$

where I have dropped the visibility term currently, since I do not explicitly handle occlusions. The value of σ_b is set to 2.0 in the file `getalphaImg.m`.

As can be seen, the above functional (eqn 4) is similar to (eqn 2) except that here the gradient is one of the channels (each subscript c in eqn 4 represents 1 channel). Also, all channels have equal weights (γ is set to 1.0). The remaining implementation details of the code pertain to this formulation. The details of the original formulation (eqn 2) are along the same lines.

The idea in [3] was to minimize the above functional *globally* (for all the pixels simultaneously). Since, this is a non-linear functional, very similar to the very popular Horn-Schunck model [5], it had to be linearized at first. The image difference I_z is linearized around the current image using the image derivatives computed on it.

$$
I_z^{k+1} \approx I_z^k + I_x^k du^k + I_y^k dv^k \qquad (8)
$$

A second linearization is applied to the terms (du^k, dv^k) to sift out the remaining non-linearity in the equations. Although, the original Horn-Schunck formulation [5] also uses this lineariation the *main difference* is the use of linearization after computing the Euler-Lagrange equations for minimization [2]. Following the same approach, the double linearization of the functional (eqn 4) yields

$$
\begin{aligned}
&(\Psi')_D^{k,l}(I_x^{[c],k}(I_z^{[c],k} + I_x^{[c],k} du^{k,l+1} + I_y^{[c],k} dv^{k,l+1})) - && (9)\\
&\qquad \operatorname{div}((\alpha_g + \alpha_l \cdot b(\mathbf{x})) \cdot (\Psi')_S^{k,l} \Delta(u^k + du^{k,l+1})) = 0 \\
&(\Psi')_D^{k,l}(I_y^{[c],k}(I_z^{[c],k} + I_x^{[c],k} du^{k,l+1} + I_y^{[c],k} dv^{k,l+1})) - && (10)\\
&\qquad \operatorname{div}((\alpha_g + \alpha_l \cdot b(\mathbf{x})) \cdot (\Psi')_S^{k,l} \Delta(v^k + dv^{k,l+1})) = 0
\end{aligned}
$$

where the following expansions are used

$$
\begin{aligned}
\Psi'^{k,l}_D &= \sum_c \Psi'((I_z^{[c],k} + I_x^{[c],k} du^{k,l} + I_y^{[c],k} dv^{k,l})^2) && (11)\\
\Psi'^{k,l}_S &= \Psi'(|\Delta(u^k + du^{k,l})|^2 + |\Delta(v^k + dv^{k,l})|^2) && (12)\\
& && (13)
\end{aligned}
$$

The implementation of the smoothness term in the above equations is a slightly modified version of the one described in [2] in the Numerics section, Chapter 2. Thus for every i^{th} pixel, we have

$$\text{div}((\alpha_g + \alpha_l \cdot b(\mathbf{x})) \cdot (\Psi')_S^{k,l} \Delta(u^k + du^{k,l+1})) = \sum_{j \in \mathcal{N}^-(i) \cap \mathcal{N}^+(i)} (\alpha_g + \alpha_l \cdot b(\mathbf{x}))_{i \sim j} (\Psi'_S)_{i \sim j}^{k,l} (u_j^k - u_j^i + du_j^{k,l,m} - du_i^{k,l,m}) \tag{14}$$

$$\text{div}((\alpha_g + \alpha_l \cdot b(\mathbf{x})) \cdot (\Psi')_S^{k,l} \Delta(v^k + dv^{k,l+1})) = \sum_{j \in \mathcal{N}^-(i) \cap \mathcal{N}^+(i)} (\alpha_g + \alpha_l \cdot b(\mathbf{x}))_{i \sim j} (\Psi'_S)_{i \sim j}^{k,l} (v_j^k - v_j^i + dv_j^{k,l,m} - dv_i^{k,l,m}) \tag{15}$$

This implementation can be found in the file `computepsidashFS_sand.m` and `computepsidashFS_brox.m`. Finally, all the terms in the above linear equations have to be put in matrix form ($A\mathbf{x} = b$), where $\mathbf{x}$ represents the increment in optical flow (du, dv), stacked alternatively. Thus the various terms are aggregated separately for du and dv as follows.

$$\text{A} : du : du^{k,l+1} : \sum_c {\Psi'}_D^{k,l} (I_x^{[c],k})^2 \tag{16}$$

$$\text{A} : du : dv^{k,l+1} : \sum_c {\Psi'}_D^{k,l} (I_x^{[c],k} I_y^{[c],k}) \tag{17}$$

$$\text{A} : du : du_j^{k,l+1} : -(\alpha_g + \alpha_l \cdot b(\mathbf{x}))_{i \sim j} (\Psi'_S)_{i \sim j}^{k,l} \tag{18}$$

$$b : du : -\sum_c (\Psi')_D^{k,l} (I_x^{[c],k} I_z^{[c],k}) - (\alpha_g + \alpha_l \cdot b(\mathbf{x}))_{i \sim j} (\Psi'_S)_{i \sim j}^{k,l} (u_i^k - u_j^k) \tag{19}$$

$$\text{A} : dv : du^{k,l+1} : \sum_c {\Psi'}_D^{k,l} (I_x^{[c],k} I_y^{[c],k}) \tag{20}$$

$$\text{A} : dv : dv^{k,l+1} : \sum_c {\Psi'}_D^{k,l} (I_y^{[c],k})^2 \tag{21}$$

$$\text{A} : dv : dv_j^{k,l+1} : -(\alpha_g + \alpha_l \cdot b(\mathbf{x}))_{i \sim j} (\Psi'_S)_{i \sim j}^{k,l} \tag{22}$$

$$b : dv : -\sum_c (\Psi')_D^{k,l} (I_x^{[c],k} I_z^{[c],k}) - (\alpha_g + \alpha_l \cdot b(\mathbf{x}))_{i \sim j} (\Psi'_S)_{i \sim j}^{k,l} (v_i^k - v_j^k) \tag{23}$$

Finally, the files `constructMatrix_brox.m` and `constructMatrix_sand.m` take these computed terms and place them in the appropriate sparse matrices (A, b) which are then passed to the SOR algorithm in `sor.m`.

All the codes have comments in them to explain the various lines. Please see updated version of this document to check how the code / documentation has been modified. Updated versions of the document will be available at

`http://research.iiit.ac.in/~visesh/myweb/Software/oflow.pdf`

This document is only a short explanation of my implementation. Understanding the whole code thoroughly for your use, however, requires that you read and understand all the papers [3, 7], and refer the theses [2, 4]. Although I have not used implementation details given in [6], I recommend a read to anyone interested in understanding the algorithm thoroughly.

References

[1] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, 1994. 2

[2] T. Brox. From pixels to regions: Partial differential equations in image analysis. 2, 3, 4

[3] T. Brox, A. Bruhn, N. Papenberg, and J. Weickert. High accuracy optical flow estimation based on a theory for warping. *European Conference on Computer Vision (ECCV)*, 2004. 1, 2, 3, 4

[4] A. Bruhn. Variational optic flow computation, accurate modelling and efficient numerics. *PhD thesis, Mathematical Image Analysis Group, Department of Mathematics and Computer Science, Saarland University, Saarbrücken, Germany*, 2006. 2, 4

[5] B. K. Horn and B. G. Schunck. Determining optical flow. 1980. 3

[6] P. Sand. Long-range video motion estimation using point trajectories. *PhD thesis, Robotics, Vision, and Sensor Networks Group, Department of Electrical Engineering and Computer Science, Masachusetts Institute of Technology*, 2006. 4

[7] P. Sand and S. Teller. Particle video. *IEEE Computer Vision and Pattern Recognition (CVPR)*, 2006. 1, 2, 3, 4