Optical Flow Explanation and Team Tasks

Miles Lindheimer

November 2015

1 Overview

"**Optical flow** is the pattern of apparent motion of objects, surfaces, and edges in a visual scene caused by the relative motion between an observer (an eye or a camera) and the scene." - Andrew Burton and John Radford

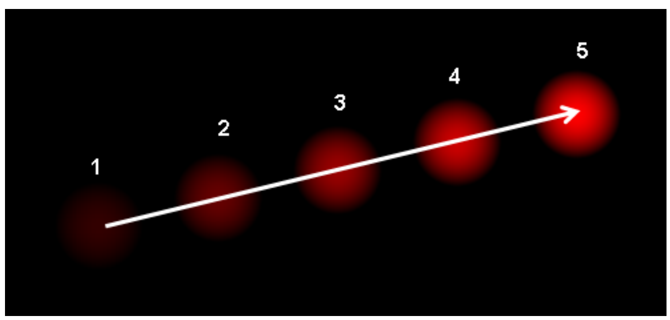

Figure 1: Optical Flow of a ball

Brightness of pixel (x, y) at time t:

$$I(x, y, t)$$

Assuming constant brightness:

$$I(x, y, t) = I(x + dx, y + dy, t + dt) = I(x, y, t) + \frac{\delta I}{\delta x}dx + \frac{\delta I}{\delta y}dy + \frac{\delta I}{\delta t}dt$$

We're trying to solve for u and v:

$$I_x V_x + I_y V_y = -I_t$$

where $u = \frac{dx}{dt}$ and $v = \frac{dy}{dt}$

But before we do that, we need to first find the image derivatives I_x and I_y.

2 Image Derivatives

"The **Sobel Operator** computes an approximation of the gradient of the image intensity function. At each point in the image, the result of the Sobel-Feldman operator is either the corresponding gradient vector or the norm of this vector."
- Wikipedia

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * A, \quad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * A$$

In order to obtain our image derivatives I_x and I_y, we convolve with each of the kernels above. I_t is constant and always looks like

$$I_t = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}$$

since we know that time doesn't change between frame to frame.

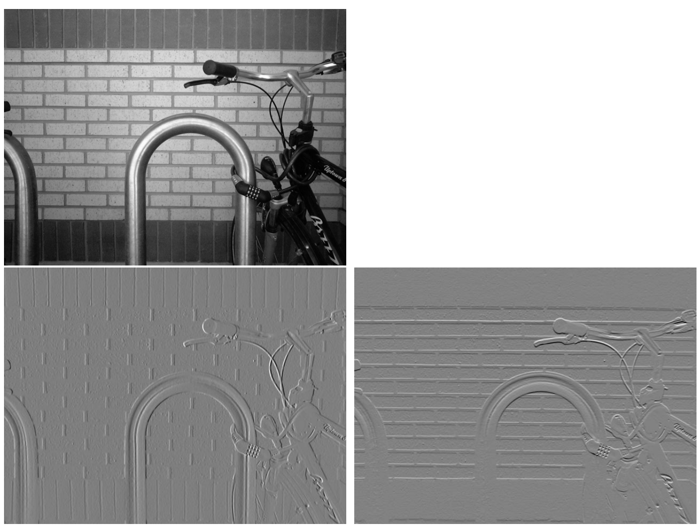

Figure 2: Original grayscale image on top, and after convolution with G_x and G_y on left and right, respectively

3 Lucas-Kanade

Now that know how to find our image derivatives I_x and I_y, and we keep I_t constant, we can now solve for our flow vector V. Since images tend to be big and expensive, and we may need to adjust how much we look at depending on the nature of the movement, we'll use the Lucas-Kanade method, which utilizes a sliding window to solve for each V in our image.

Using a window size of k:

$$\begin{bmatrix} I_x(q_1) & I_y(q_1) \\ \vdots & \vdots \\ I_x(q_k) & I_y(q_k) \end{bmatrix} \begin{bmatrix} V_x \\ V_y \end{bmatrix} = \begin{bmatrix} -I_t(q_1) \\ \vdots \\ -I_t(q_k) \end{bmatrix}$$

Now we can solve for V using linear least squares:

$$V = (A^T A)^{-1} A^T b$$

$$\begin{bmatrix} V_x \\ V_y \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{k} I_x(q_i)^2 & \sum_{i=1}^{k} I_x(q_i) I_y(q_i) \\ \sum_{i=1}^{k} I_x(q_i) I_y(q_i) & \sum_{i=1}^{k} I_y(q_i)^2 \end{bmatrix}^{-1} \begin{bmatrix} -\sum_{i=1}^{k} I_x(q_i) I_t(q_i) \\ -\sum_{i=1}^{k} I_y(q_i) I_t(q_i) \end{bmatrix}$$

We do this until we run out of pixels.

Great! We can find flow vectors in our image, so we're done - except that this may gives us weird and screwy results... so we need something a little more robust...

4 Pyramidal Approach

We just need to run Lucas-Kanade on the same pair of images at different resolutions. For that we set up a Gaussian pyramid for the images and perform Lucas-Kanade for each resolution (Figure 3) and interpolate the results (Figure 4).

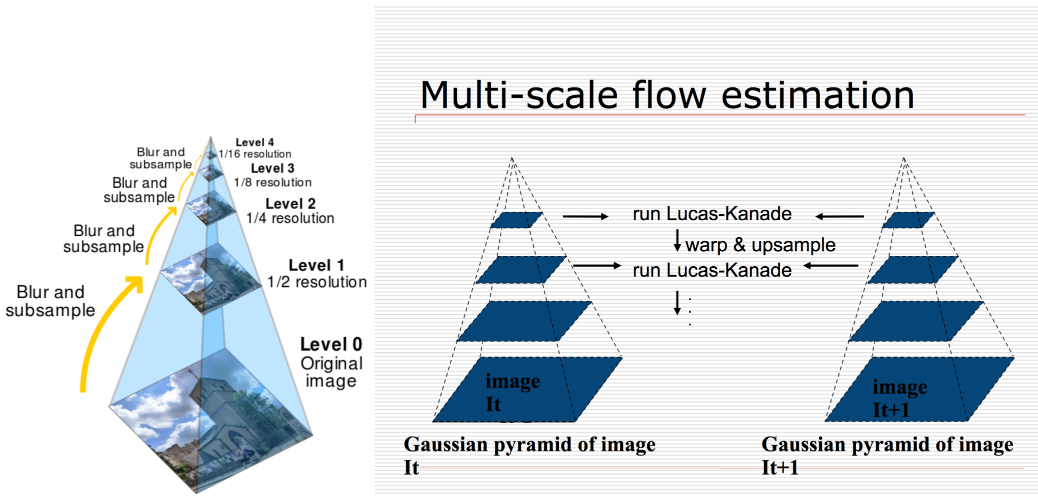

Figure 3: Gaussian Pyramid (left) and Pyramidal Lucas-Kanade (right)

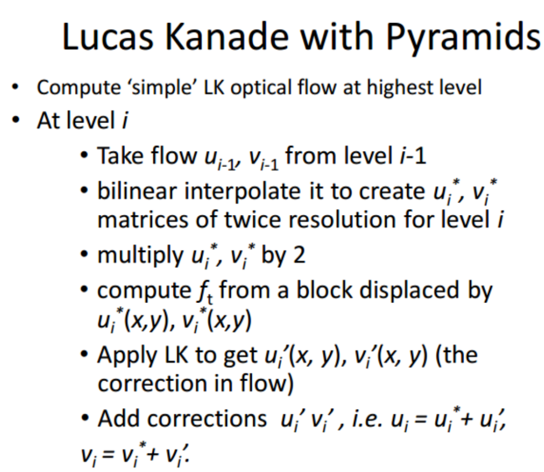

Figure 4: Pseudocode for PLK

```matlab
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [fx, fy, ft] = ComputeDerivatives(im1, im2);
%ComputeDerivatives Compute horizontal, vertical and time derivative
%                            between two gray-level images.

if (size(im1,1) ~= size(im2,1)) | (size(im1,2) ~= size(im2,2))
   error('input images are not the same size');
end;

if (size(im1,3)~=1) | (size(im2,3)~=1)
   error('method only works for gray-level images');
end;


fx = conv2(im1,0.25* [-1 1; -1 1]) + conv2(im2, 0.25*[-1 1; -1 1]);
fy = conv2(im1, 0.25*[-1 -1; 1 1]) + conv2(im2, 0.25*[-1 -1; 1 1]);
ft = conv2(im1, 0.25*ones(2)) + conv2(im2, -0.25*ones(2));

% make same size as input
fx=fx(1:size(fx,1)-1, 1:size(fx,2)-1);
fy=fy(1:size(fy,1)-1, 1:size(fy,2)-1);
ft=ft(1:size(ft,1)-1, 1:size(ft,2)-1);
```

```matlab
function [u, v] = LucasKanade(im1, im2, windowSize);
%LucasKanade  lucas kanade algorithm, without pyramids (only 1 level);
%REVISION: NaN vals are replaced by zeros
[fx, fy, ft] = ComputeDerivatives(im1, im2);
u = zeros(size(im1));
v = zeros(size(im2));
halfWindow = floor(windowSize/2);
for i = halfWindow+1:size(fx,1)-halfWindow
   for j = halfWindow+1:size(fx,2)-halfWindow
      curFx = fx(i-halfWindow:i+halfWindow, j-halfWindow:j+halfWindow);
      curFy = fy(i-halfWindow:i+halfWindow, j-halfWindow:j+halfWindow);
      curFt = ft(i-halfWindow:i+halfWindow, j-halfWindow:j+halfWindow);

      curFx = curFx';
      curFy = curFy';
      curFt = curFt';


      curFx = curFx(:);
      curFy = curFy(:);
      curFt = -curFt(:);

      A = [curFx curFy];

      U = pinv(A'*A)*A'*curFt;

      u(i,j)=U(1);
      v(i,j)=U(2);
   end;
end;
u(isnan(u))=0;
v(isnan(v))=0;
```

Figure 5: Implementation of Lucas-Kanade in Matlab