An Implementation of the Classic++ Algorithm for Optical Flow

Anon. author 1 Anon. author 2

Abstract

Optical flow algorithms calculate the movement of objects between consecutive frames of video, by attempting to reconstruct the displacement of every pixel from frame to frame. All optical flow algorithms are built on minimizing an objective function, but there are many ways to do this minimization, and also many possible forms of the objective function. In this project, we will implement the Classic++ optical flow algorithm and analyze the performance of different forms of smoothing and regularization.

1. Introduction

Optical flow can be applied to a variety of problems, for example, it allows driverless vehicles to detect the movement of pedestrians and other cars on the road. It also allows the car to estimate its own movement. Given the motion of pixels in a series of video frames, optical flow methods can interpolate pixel locations between frames, resulting in a smoother video and a sharper display when bandwidth is limited.

Optical flow is defined as the temporal change in brightness patterns of an image due to motion of an object relative to the camera. At its core, it is the problem of calculating where each pixel moves from one frame of a video to the next. [2] However, optical flow similar to but is not the exactly the same as a motion field. For example, a rotating perfectly uniform sphere will have no optical flow, because the brightness pattern remains the same, but it does have a motion field. In contrast, if we fix the sphere and move the light source, there will be an optical flow field due to change in brightness even though the sphere does not move.

The changes in pixel intensity between frames are represented as a vector field. The motion of each pixel is represented as a flow vector $[x, y]$. The essential constraint governing the image I and the flow fields X and Y is

$$I_x \cdot X + I_y \cdot Y + I_t = 0 \tag{1}$$

In qualitative terms, the change in a pixel intensity over time (I_t) has to equal the difference in intensity between the original pixel and the one upstream from it in the flow field. Eq. 1 can also be expressed in discrete space:

$$I[x, y, t_1] = I[x + U[x, y], y + V[x, y], t_2] \tag{2}$$

However, this equation results in only one constraint per pixel, whereas there are two flow vector unknowns per pixel. Therefore, this problem is underconstrained. [4] Optical flow algorithms will add additional constraints, called a penalty function, usually to represent the requirement that the optical flow of two neighboring pixels is similar. Intuitively, this is due to the fact that in natural images neighboring pixels do not move independently of each other. While almost all optical flow algorithms use Eq. 1, different algorithms apply different penalty functions and numerical techniques to solve or optimize the resulting equations.

2. Prior Work

The optical flow formulation was originally proposed by Horn and Schunck (HS) [3], who recognized that discontinuities only occur as a result of occlusion and neighboring pixels generally do not move independently. This discovery allowed them to constrain the problem by minimizing the square of the gradient of the flow vector. A flow field that changes abruptly from pixel to pixel will have a large-magnitude gradient on average, whereas a smooth flow field, such as a brightness pattern, will have a small gradient. The HS algorithm minimizes the following quantity:

$$e = \int\int_{img} (I_x \cdot X + I_y \cdot Y + I_t)^2 + \lambda(||\nabla X||^2 + ||\nabla Y||^2) dA \tag{3}$$

(Throughout this document, matrix multiplication will be element-wise, not dot-product-wise, unless otherwise noted.)

The first term represents how well the flow vector field explains the pixel motion in the video, and should be 0 for a perfect flow field. The second term is the penalty function discussed above, which penalizes sudden changes in flow. The λ parameter controls the desired smoothness of the outcome. The HS paper derived a closed form expression for the minimizing values of X and Y, but computing power was too expensive at the time for the expression to

be practical. Instead, an approximation scheme was used to iteratively converge on the minimizing values.

Since then, many modifications have been proposed to the HS minimization problem. Robust penalty terms have been developed, in which the derivatives of the flow vector are raised to a smaller power. For example, a popular penalty function, called the Charbonnier penalty function (Eq. 4), is used in the Classic++ algorithm.

$$\sqrt{||\nabla X||^2 + ||\nabla Y||^2 + \epsilon} \tag{4}$$

It is essentially linear in the gradients. Intuitively, penalty terms with smaller powers are more tolerant of some sudden changes in flow direction, which results in sharper edges in the flow field. Most penalty schemes do not result in an analytical expression for the minimum, thus optimization algorithms are required to solve the resulting problem.

The optimization method is another point of innovation. Sun [1] claims that the HS algorithm was limited by the approximation scheme it used. Modern computers can solve for the flow vector exactly, resulting in much better performance. Many recent implementations of optical flow use incremental multi-resolution optimization, where the image is first shrunk to a smaller size, and optical flow is calculated on the smaller image. Then, the smaller optical flow is used as the initial value in approximating the optical flow on a scaled-up image. This process can be repeated multiple times, until optical flow for the original image is found.

Multi-resolution approaches can result in faster convergence, by providing a good initial value to the large, expensive final approximation layer. This approach also improves accuracy when large, uniformly-colored objects, such as a car, are moving. If optical flow is calculated directly on such an object, it will appear that only the edges of the object are moving; the algorithm does not detect motion in the middle of the object. On the other hand, if the image is first scaled down so that the large object has mostly edge pixels, the algorithm will correctly infer that the entire object is moving and propagates this information to the largest layer.

In this project we will implement the Classic++ algorithm developed by Sun et al. This approach improves upon the HS algorithm by using modern optimization methods, which results in improved flow estimation. It uses the Charbonnier penalty function (Eq. 4), which results in a smoother penalty function, and spline-based bi-cubic interpolation.

3. Solving the HS Objective Function

To find optical flow, we need to minimize the objective function found in Eq. 3. To do this, we will use the Euler-Lagrange equation, which gives us conditions for minimizing e. Namely, if e depends on a function L and

$$e = \int\int L(x, y, f, f_x, f_y)dxdy \tag{5}$$

then e is minimized when

$$\frac{\partial L}{\partial f} - \frac{\partial}{\partial x}\frac{\partial L}{\partial f_x} - \frac{\partial}{\partial y}\frac{\partial L}{\partial f_y} = 0 \tag{6}$$

The Euler-Lagrange equation applies directly to our problem, if we use two instances of it to constrain the flow fields X and Y separately. For us, L is simply the objective function e:

$$L = \int\int_{img} (I_x \cdot X + I_y \cdot Y + I_t)^2 + \lambda(X_x^2 + X_y^2 + Y_x^2 + Y_y^2)dA \tag{7}$$

To constrain X, we let $f = X$, so

$$\frac{\partial L}{\partial X} = 2I_x(I_x X + I_y Y + I_t) \tag{8}$$

$$\frac{\partial L}{\partial X_x} = 2\lambda X_x \tag{9}$$

$$\frac{\partial}{\partial x}\frac{\partial L}{\partial X_x} = 2\lambda X_{xx} \tag{10}$$

$$\frac{\partial L}{\partial X_y} = 2\lambda X_y \tag{11}$$

$$\frac{\partial}{\partial y}\frac{\partial L}{\partial X_y} = 2\lambda X_{yy} \tag{12}$$

And our final constraint is

$$I_x(I_x X + I_y Y + I_t) = \lambda(X_{xx} + X_{yy}) \tag{13}$$

This equation gives us one constraint for every pixel in the image. We can apply the same logic with $f = Y$ to get another constraint per pixel:

$$I_y(I_x X + I_y Y + I_t) = \lambda(Y_{xx} + Y_{yy}) \tag{14}$$

Sometimes, these constraints are shown in block matrix form:

$$\begin{bmatrix} I_x^2 + \lambda(D_x^2 + D_y^2) & I_x I_y \\ I_x I_y & I_y^2 + \lambda(D_x^2 + D_y^2) \end{bmatrix} \begin{bmatrix} X \\ Y \end{bmatrix} = \begin{bmatrix} I_x I_t \\ I_y I_t \end{bmatrix} \tag{15}$$

There are some subtleties hiding in this equation. X and Y are vectors of length $w \times h$ - they are the flow vectors smushed into one column. All of the image derivatives (I_x, etc.) in the left side matrix are diagonal matrices of width and height each $w \times h$. They contain the image data along the diagonal, the same way X and Y contain the flow vector data in a column. D_x^2 performs a second image derivative in the x direction to the flow vector on which it is applied. (This is an exception to our convention that matrix multiplication means pairwise product. D_x is applied like a dot product.) Finally, the right hand side is in fact a single column of size $2wh \times 1$. Unlike on the left hand side $I_x I_t$ and $I_y I_t$ are single-column, not diagonal.

4. Implementing the Euler-Lagrange Solution

We implemented a program in MATLAB that finds the flow vectors X and Y given two images. We use the block matrix formulation (Eq. 15) to express the system of equations. The image derivatives are computed and converted into diagonal sparse matrices. We approximate $D_x^2 + D_y^2$ with convolution by the kernel

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} \tag{16}$$

, known as the Laplacian kernel. This means that $L = D_x^2 + D_y^2$ is a $(wh) \times (wh)$ sparse matrix, with approximately $5wh$ non-zero entries. It is defined as follows:

$$L(a,b) = \begin{cases} 4 & \text{if } a = b \\ -1 & \text{if } a \text{ and } b \text{ are neighbors} \\ 0 & \text{otherwise} \end{cases} \tag{17}$$

With the terms in Eq. 15 defined, solving for X and Y is simply a matter of matrix inversion.

In practice, we run multiple iterations of Eq. 15 to find the final flow. This is because the HS objective is only a linearized version of Eq. 2, the true flow error. For large flow values, it is not accurate. After each iteration, we warp image 1 using the flow field found so far. We then run another iteration using the warped image 1, and add the new flow to the total flow so far.

Unfortunately, there is no obvious stopping point for this algorithm. Repeatedly minimizing the HS objective will refine the flow, making the warped image 1 look more and more like image 2. However, this results in "overfitting". The ground truth flow is not in fact a perfect transformation from image 1 to image 2 - there are changes in illumination and occlusion between the two images. Empirically, we found that running too many iterations caused the endpoint error to increase. In our tests, we used 10 iterations, the same number used in [1].

The results of our algorithm are shown in Figure 1. The standard HS algorithm is predictably confused by the spinning circle in the lower left hand corner. The boundaries are also somewhat blurry. Finally, the shell in the bottom middle is very poorly labeled.

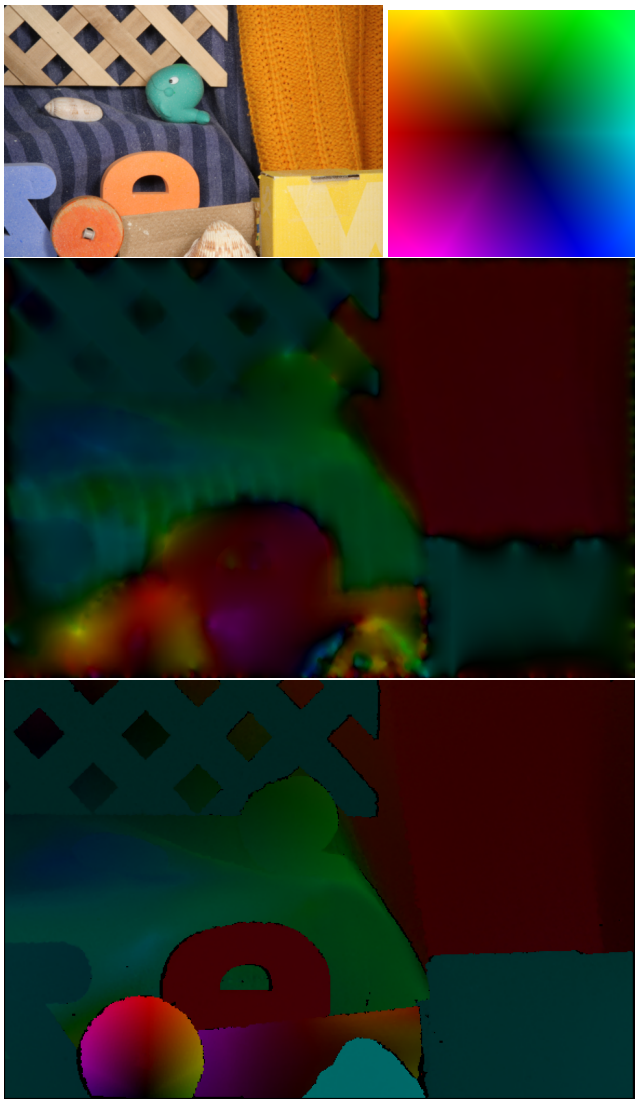

Figure 1. Top: the "RubberWhale" test image for optical flow, and a key for the optical flow visualizations in this paper. Middle: optical flow according to our implementation of the HS algorithm. Bottom: ground truth flow, according to manual human labeling.

References

[1] S. R. D. Sun and M. Black. A quantitative analysis of current practices in optical flow estimation and the principles behind them. *International Journal of Computer Vision*, 10(1007), 2013.

[2] D. Fleet and Y. Weiss. Optical flow estimation. *Mathematical Models of Computer Vision: The Handbook*, pages 239–258, 2005.

[3] B. Horn and B. Schunck. Determining optical flow. *Artificial Intelligence*, 17:185–203, 1981.

[4] Y. Wu. Optical flow and motion analysis, 2006. Northwestern University EECS432 - Advanced Computer Vision Notes Series 6.