An Implementation of the Classic++ Algorithm for Optical Flow

Felix Sun Veronica Lane

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 algorithms attempt to detect the motion of an object relative to the camera by measuring the temporal change in brightness patterns of an image. Estimations of optical flow rely on the assumption that the brightness of a pixel remains the same when it moves. 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 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. [5] 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 since neighboring pixels are generally part of the same object, they do not move independently. This assumption of spatial smoothness 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 objective function

$$\begin{aligned} E(\mathbf{u}, \mathbf{v}) = \sum_{i,j} &(\rho(I_1(i, j) - I_2(i + u_{i,j}, j + v_{i,j}))^2 \\ &+ \lambda(\rho(u_{i,j} - u_{i+1,j}) + \rho(u_{i,j} - u_{i,j+1}) \\ &+ \rho(v_{i,j} - v_{i+1,j}) + \rho(v_{i,j} - v_{i,j+1}))dA \end{aligned} \tag{3}$$

where the penalty function $\rho(x) = x^2$. 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 penalizes sudden changes in flow. The regularization parameter, λ, controls the desired smoothness of the outcome. The HS paper derived a closed form expression for the minimizing values of the square of the magnitudes of the optical flow gradients, 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.

The optical flow estimates produced by the HS objective function are significantly affected by outliers due to occlusion, reflection, and motion boundaries. 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, the Charbonnier penalty function, (Eq. 4), which is used in the Classic++ algorithm, is a differentiable, convex function which is essentially linear in the gradients.

$$\rho(x) = \sqrt{x^2 + \epsilon^2} \tag{4}$$

Intuitively, penalty terms with smaller powers are more tolerant of sudden changes in flow direction, which results in sharper edges in the flow field. The convexity of the Charbonnier function because it is easier to optimize than non-convex alternatives, such as the Lorentzian penalty function (Eq. 5).

$$\rho(x) = \log 1 + \frac{x^2}{2 * \sigma^2} \tag{5}$$

We implemented the Classic++ algorithm developed by Sun et al [1]. This approach improves upon the HS algorithm by using multi-resolution optimization, median filtering, and the Charbonnier penalty function (Eq. 4).

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

$$E = \int \int L(x, y, f, f_x, f_y) dx dy \tag{6}$$

then e is minimized when

$$\frac{\partial L}{\partial f} - \frac{\partial}{\partial x}\frac{\partial L}{\partial f_x} - \frac{\partial}{\partial y}\frac{\partial L}{\partial f_y} = 0 \tag{7}$$

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:

$$L = \int \int_{img} (I_x \cdot X + I_y \cdot Y + I_t)^2 + \lambda(X_x^2 + X_y^2 + Y_x^2 + Y_y^2) dA \tag{8}$$

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

$$\frac{\partial L}{\partial X} = 2I_x(I_x X + I_y Y + I_t) \tag{9}$$

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

$$\frac{\partial}{\partial x}\frac{\partial L}{\partial X_x} = 2\lambda X_{xx} \tag{11}$$

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

$$\frac{\partial}{\partial y}\frac{\partial L}{\partial X_y} = 2\lambda X_{yy} \tag{13}$$

And our final constraint is

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

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:

$$I_y(I_x X + I_y Y + I_t) = \lambda(Y_{xx} + Y_{yy}) \tag{15}$$

Sometimes, these constraints are shown in block matrix form:

$$\begin{bmatrix} I_x^2 - \lambda(D_x^2 + D_y^2) & I_x I_y \\ I_x I_y & I_y^2 - \lambda(D_x^2 + D_y^2) \end{bmatrix} \begin{bmatrix} X \\ Y \end{bmatrix} = -\begin{bmatrix} I_x I_t \\ I_y I_t \end{bmatrix} \tag{16}$$

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. 16) 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

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} \tag{17}$$

, 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:

$$L(a,b) = \begin{cases} 4 & \text{if } a = b \\ -1 & \text{if } a \text{ and } b \text{ are neighbors} \\ 0 & \text{otherwise} \end{cases} \quad (18)$$

We calculate the image derivatives I_x and I_y using a 5-point filter $\frac{1}{12}[-1, 8, 0, -8, 1]$, and we used $\lambda = 0.1$. With the terms in Eq. 16 defined, solving for X and Y is simply a matter of matrix inversion.

In practice, we run multiple iterations of Eq. 16 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. (Spinning objects violate the assumption that the flow has a small gradient.) The boundaries are also somewhat blurry. To measure the accuracy of our flow estimate, we use average endpoint error (EPE), which is the average displacement between each flow vector and its ground truth counterpart. The EPE on this image is 0.322.

We test our implementation on the Middlebury optical flow benchmark [4], using the 8 publicly-available training images. The results (along with those from our improved algorithms) are shown in Figure 3.

From Figure 3, it is obvious that our implementation of HS does not perform equally well on all test cases. Figure 4 shows the test image with the highest error ("Urban2"). Urban2 has flow vectors with much greater magnitude, compared to RubberWhale. The building in the foreground moves particularly quickly between frames. The displacement is 5 pixels on average, which is further than the spatial derivatives can propagate information. Therefore, HS cannot track the displacement of the building.

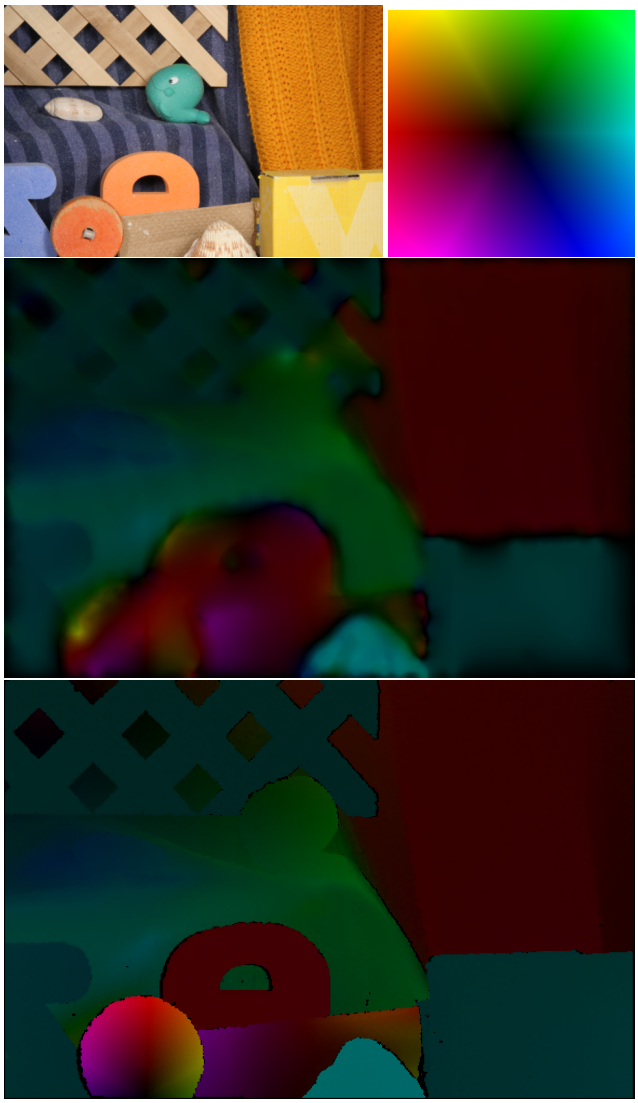
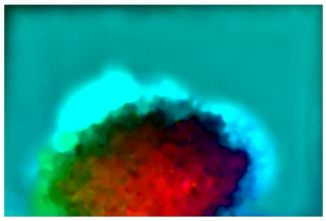


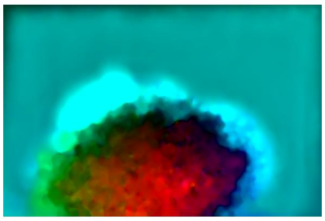
Figure 1: Top: the "RubberWhale" test image for optical flow, and a key for the optical flow visualizations in this paper. Middle: optical flow according to our implementation of the HS algorithm. Bottom: ground truth flow, according to manual human labeling.

5. Multi-Resolution Optimization

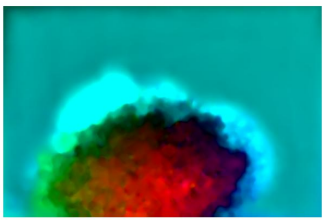
To improve estimates of large displacment, our algorithm use incremental multi-resolution optimization. It constructs an image pyramid, using gaussian filtering at each level. The optical flow is first computed for the image at the smallest level and it is is used as the initial estimate for the next-largest pyramid level, this is repeated until the largest pyramid level is reached. Multi-resoltion not only decreases the convergence time, but it increases accuracy when large moving objects of uniform brightness. Accuracy improves because at the largest pyramid level, the op-



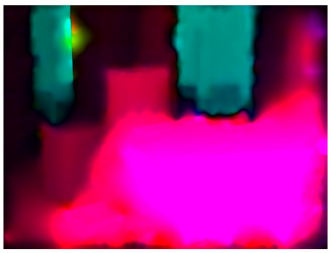
(a) Hydrangea: 3 pyramid levels
Score: .3796



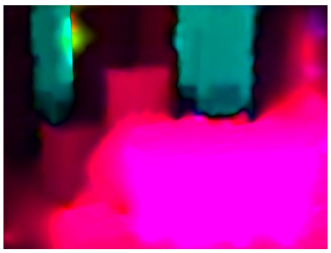
(b) Hydrangea: 5 pyramid levels
Score: .3127



(c) Hydrangea: 6 pyramid levels
Score: .3135



(d) Urban: 3 pyramid levels
Score: 1.9045



(e) Urban: 6 pyramid levels
Score: 1.0641

(f) Urban: 9 pyramid levels
Score: 0.9838

Figure 2: Results for different numbers of pyramid levels for the Hydrangea (top) and Urban (bottom) test images from the Middlebury dataset. As the pyramid levels increase, the optical flow estimate improves. For the hydrangea, the flow estimate for the non-hydrangea part of the image decreases since there is no movement in that part of the image. For the urban image, the flow estimate of the edge of the large object becomes more uniform as the number of pyramid levels increases.

tical flow algorithm only detects movement of the edges of the object. However, the algorithm better detects the movement of the entire object when multi-resolution optimization is used, because the flow of the objects inner pixels is propogated through the pyramid levels. Test cases in which large objects moved required more pyramid levels than those in which small objects moved, the larger the object, the more pyramid levels required to achieve the best score. Too many pyramid levels worsened the score, this is likely to due the fact that at the lowest level the object becomes too small and no noticeable movement occurs.

6. Median Filtering

Applying a median filter between each iteration at each pyramid level significantly improves the optical flow estimation by eliminating outliers. We applied various sizes of median filters and discovered that each image had a different optimal size of median filter. However, for most images, while adding a filter had a significant score impact, changes in the size of the median filter had a small impact on improving the optical flow estimation. Based on the data 5, the best median filter size to apply to all images is 9x9, since increasing the size to 13x13 resulted in very small (less than 2%) improvements. Certain images had much more significant improvements when a median filter applied, images with the greatest improvement had high optical flow estimation scores and the objects moving in the image were larger than in the images with less improvement.

7. Solving Optical Flow with a Charbonnier Penalty

We would now like to use the Charbonnier penalty in our flow objective function. With the Charbonnier penalty on flow smoothness, our objective function becomes

$$e = \int\int_{img} (I_x \cdot X + I_y \cdot Y + I_t)^2 + \lambda\sqrt{X_x^2 + X_y^2 + Y_x^2 + Y_y^2 + \epsilon}\, dA \quad (19)$$

To derive a minimum for this function, it's best to express it in discrete form:

$$e = \sum_{i,j} (I_x(i,j)X(i,j) + I_y(i,j)Y(i,j) + I_t(i,j))^2 + \lambda \sqrt{\begin{array}{c}(X(i,j) - X(i-1,j))^2 \\ +(X(i,j) - X(i,j-1))^2 \\ +(Y(i,j) - Y(i-1,j))^2 \\ +(Y(i,j) - Y(i,j-1))^2 + \epsilon\end{array}} \quad (20)$$

	No Filter	5x5	9x9	13x13	15x15
Hydrangea	0.7417	0.6730	**0.6605**	0.6643	0.6783
RubberWhale	0.3351	0.3210	0.3053	**0.3022**	0.3035
Urban2	1.6571	0.9838	0.8949	0.8692	**0.8620**
Venus	1.0443	.6232	0.5716	**0.5696**	0.5725

Figure 5: Score for optical flow estimation with different median filter sizes. We computed the flow at the optimal number of pyramid levels determined from testing with the 5x5 median filter.

Image name	HS	Multi-res HS	Charbonnier
Dimetrodon	0.5068	0.3832	0.4365
Grove2	1.4710	0.4281	1.5171
Grove3	2.4258	1.2280	2.4279
Hydrangea	1.2445	0.8741	0.8515
RubberWhale	0.3221	0.3630	0.2266
Urban2	6.9539	1.7945	6.5737
Urban3	5.4880	1.8573	10.0828
Venus	1.9489	0.7809	1.9970
Average	2.5451	0.9636	3.0141

Image name	Multi-res Charbonnier
Dimetrodon	0.2954
Grove2	0.2618
Grove3	1.0128
Hydrangea	0.6237
RubberWhale	0.3083
Urban2	1.3724
Urban3	1.3881
Venus	0.6191
Average	0.7352

Figure 3: Endpoint error by test image, for the various algorithms presented in this paper.

To save space, let us name the error and regularization terms:

$$Err(i,j) = I_x(i,j)X(i,j) + I_y(i,j)Y(i,j) + I_t(i,j) \quad (21)$$

$$Reg(i,j) = (X(i,j) - X(i-1,j))^2 + ... \quad (22)$$

Now, our objective function becomes:

$$e = \sum_{i,j} Err(i,j)^2 + \lambda\sqrt{Reg(i,j) + \epsilon} \quad (23)$$

To minimize the objective function in Eq. 23, we take the straightforward approach: calculate a derivative with respect to $X(i,j)$. By setting each derivative equal to 0, we hope to get an approximation for the X and Y that minimize e.

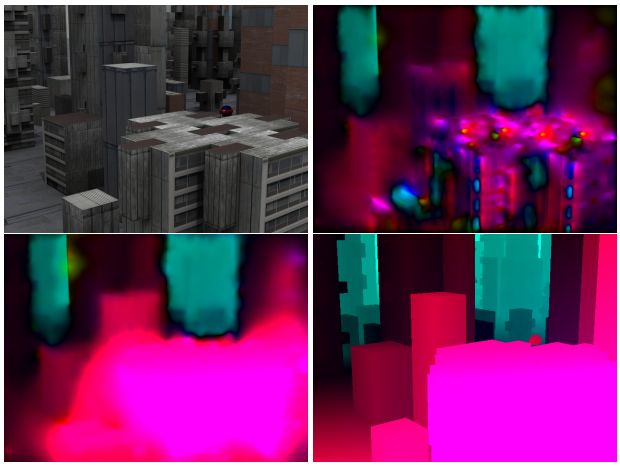


Figure 4: Results for the "Urban2" test image. Top left: one of the two input frames. Top right: optical flow using the HS algorithm, with average error 6.95. Bottom left: optical flow using Classic++, with average error 1.79. Bottom right: ground truth. The flow magnitude is too large for HS to handle correctly. Using classic++, we can locate large flows with much better accuracy.

$$\begin{aligned}\frac{\partial e}{\partial X(i,j)} &= \frac{\partial Err(i,j)}{\partial X(i,j)} \cdot 2Err(i,j) \\ &+ \lambda\frac{\partial Reg(i,j)}{\partial X(i,j)} \cdot \frac{1}{2}(Reg(i,j) + \epsilon)^{-0.5} \\ &+ \lambda\frac{\partial Reg(i+1,j)}{\partial X(i,j)} \cdot \frac{1}{2}(Reg(i+1,j) + \epsilon)^{-0.5} \\ &+ \lambda\frac{\partial Reg(i,j+1)}{\partial X(i,j)} \cdot \frac{1}{2}(Reg(i,j+1) + \epsilon)^{-0.5}\end{aligned} \quad (24)$$

Notice that only one error term depends on $X(i,j)$, but three separate regularization terms depend on $X(i,j)$.

Now, we want to evaluate the individual partial derivatives:

$$\frac{\partial Err(i,j)}{\partial X(i,j)} = I_x(i,j) \quad (25)$$

$$\frac{\partial Reg(i,j)}{\partial X(i,j)}$$
$$= 2(X(i,j) - X(i-1,j)) + 2(X(i,j) - X(i,j-1))$$
$$= 2X(i,j) - X(i-1,j) - X(i,j-1) \tag{26}$$

$$\frac{\partial Reg(i+1,j)}{\partial X(i,j)} = -2(X(i+1,j) - X(i,j)) \tag{27}$$

$$\frac{\partial Reg(i,j+1)}{\partial X(i,j)} = -2(X(i,j+1) - X(i,j)) \tag{28}$$

Finally, we assume that nearby regularization terms are approximately equal: $Reg(i,j) \approx Reg(i+1,j) \approx Reg(i,j+1)$. We can now simplify the derivative in Eq. 24, resulting in a very familiar form:

$$\begin{aligned} 0 = \frac{\partial e}{\partial X(i,j)} &= 2I_x(i,j)Err(i,j) \\ &+\lambda(\\ &\quad 4X(i,j) - X(i+1,j) \\ &\quad - X(i-1,j) - X(i,j+1) - X(i,j-1) \\ &\quad) \cdot (Reg(i,j) + \epsilon)^{-0.5} \end{aligned} \tag{29}$$

Note that the terms in parentheses are exactly $(D_{xx} + D_{yy})X$, the sum of the second derivatives of X evaluated at (i,j). Also, $(Reg(i,j) + \epsilon)^{-0.5}$ is the only term not linear in X. Therefore, we can use iteratively-reweighted least squares to solve this equation. In this formulation,

$$W(i,j) = (Reg(i,j) + \epsilon)^{-0.5} \tag{30}$$

will be the weights. We will repeatedly solve for X and Y, recompute the weights, then re-solve for X and Y. Our derivative constraint in Eq. 29 now becomes

$$\begin{aligned} 2I_x(I_xX + I_yY + I_t) \\ -\lambda(D_{xx} + D_{yy})X \cdot W = 0 \end{aligned} \tag{31}$$

There exists an analogous equation obtained by taking a derivative with respect to Y:

$$\begin{aligned} 2I_y(I_xX + I_yY + I_t) \\ -\lambda(D_{xx} + D_{yy})Y \cdot W = 0 \end{aligned} \tag{32}$$

We can combine all $2wh$ equations into block matrix form, to obtain an equation very similar to Eq. 16:

$$\begin{bmatrix} I_x^2 - \lambda W(D_x^2 + D_y^2) & I_xI_y \\ I_xI_y & I_y^2 - \lambda W(D_x^2 + D_y^2) \end{bmatrix} \begin{bmatrix} X \\ Y \end{bmatrix} = -\begin{bmatrix} I_xI_t \\ I_yI_t \end{bmatrix} \tag{33}$$

Once again, an aside on notation is necessary here. The term $\lambda W(D_x^2 + D_y^2)X$ refers to the following steps: (1) evaluate the sum of the second derivatives at X using the Laplacian filter. Put these values in a column vector of length wh, because X is a column vector and D_x is an operator. (2) Multiply the column vector by the diagonal matrix λW.

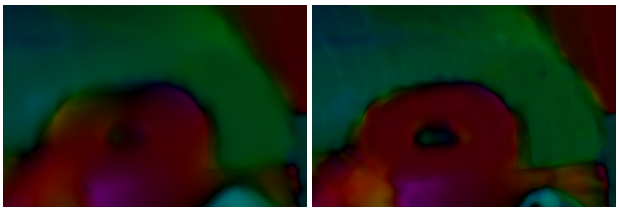


Figure 6: Comparing HS (left) and HS with Charbonnier penalty (right) on a detail in the RubberWhale test image. The Charbonnier penalty results in crisper edges and better definition in smaller regions.

To solve for the flow using this equation, we use the same technique developed for solving the original HS equation, with one addition: we recalculate W at every turn. W is recalculated using the values of X and Y found in the last iteration, according to Eq. 30.

Note that this technique can be applied to any penalty function. For a penalty function $f(Reg)$, the weight update rule is

$$W(i,j) = \frac{\partial f}{\partial Reg}(Reg(i,j)) \tag{34}$$

For example, HS uses a quadratic penalty, in which $f(Reg) = Reg$, so $W(i,j) = 1$. As expected, this gives us the iterative scheme we used to solve the HS objective.

The results of this algorithm on the Middlebury training set are shown in Figure 3, under the "Charbonnier" column. We used $\epsilon = 0.01$, with all other parameters the same as our HS implementation. Adding the Charbonnier penalty to HS results in a significant performance boost on some test cases. For example, in Figure 6, we see that using a Charbonnier penalty instead of a quadatic penalty gives better resolution at the edges of objects. For the RubberWhale image shown in this figure, the regular HS algorithm achieves an EPE of 0.32, while HS with Charbonnier penalty achieves an EPE of 0.23.

However, adding the Charbonnier penalty seems to decrease performance on test cases with large magnitude flows, like the Urban2 example discussed in Figure 4. To fix this problem, we add multi-resolution optimization to our Charbonnier algorithm. This results in a marked improvement in performance (see the "Multi-res Charbonnier" column of Figure 3), beating the multi-resolution HS algorithm on every test case.

To summarize all of the algorithms presented in this paper, Figure 7 compares their performance on the "Venus" image. We can see that running that solving for the flow

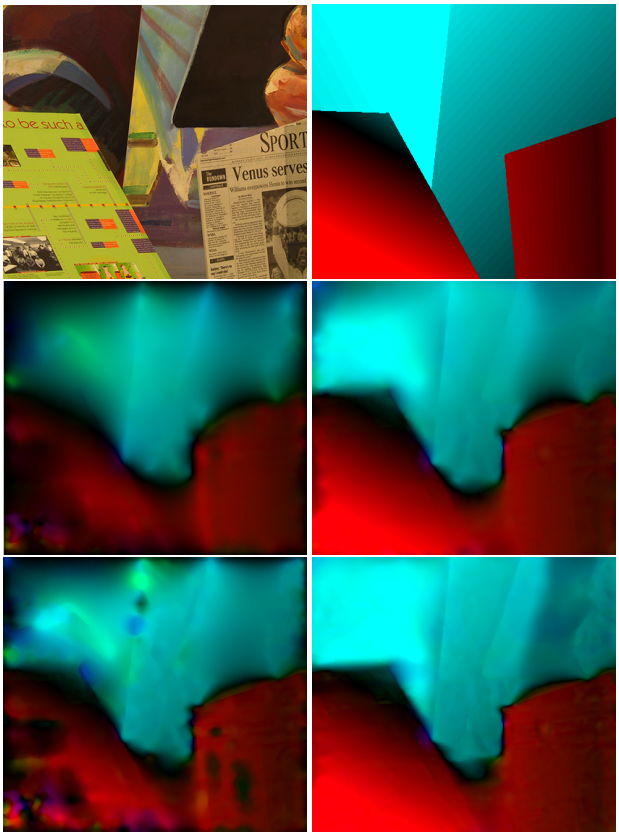


Figure 7: Comparing all four algorithm variations on the Venus test image from the Middlebury dataset. Top: one frame of the original test images; ground truth flow. Middle: HS; Classic++. Bottom: Charbonnier; Charbonnier with multi-resolution optimization. Using a multi-resolution pyramid helps the algorithm discover large areas of high-magnitude flow, while the Charbonnier penalty makes edges somewhat better-defined.

at multiple resolutions helps discover regions of large flow, like in the lower left corner of the image. On the right-center of the image, adding a Charbonnier penalty makes the top boundary of the newspaper cleaner.

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] J. L. S. R. M. J. B. S. Baker, D. Scharstein and R. Szeliski. A database and evaluation methodology for optical flow. *IEEE International Conference on Computer Vision*, pages 1–8, 2007.

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