Horn–Schunck method of estimating Optical Flow

ECE 6560 Project Report

Harsh Bhate

Contents

Acknowledgement

I'd like to thank Prof. Anthony Yezzi for his guidance and support through every step of the project.

Problem Description

Images can be considered as the 2-D projection of 3-D points in space. Concurrently, videos maye be considered as 2-D projection of 3-D points in space across time. Let us now consider the motion of a point in 3-D space across time. The set of velocities corresponding to each point in space is known as the *motion field* as shown below.

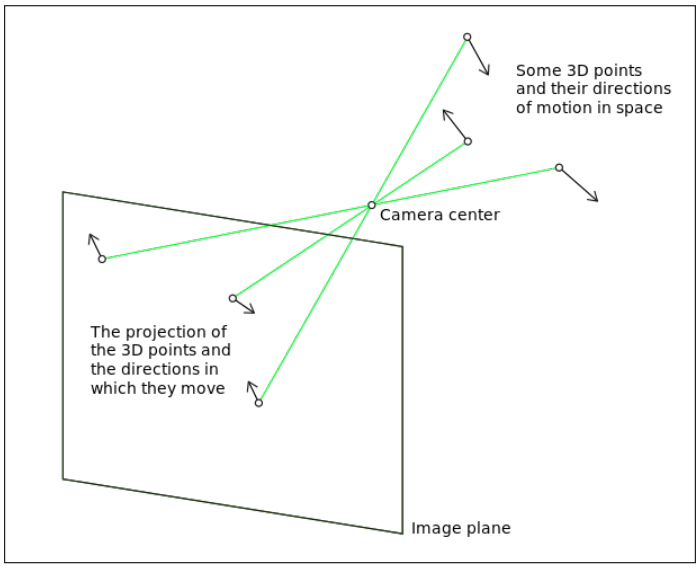

Figure 1: Motion Field

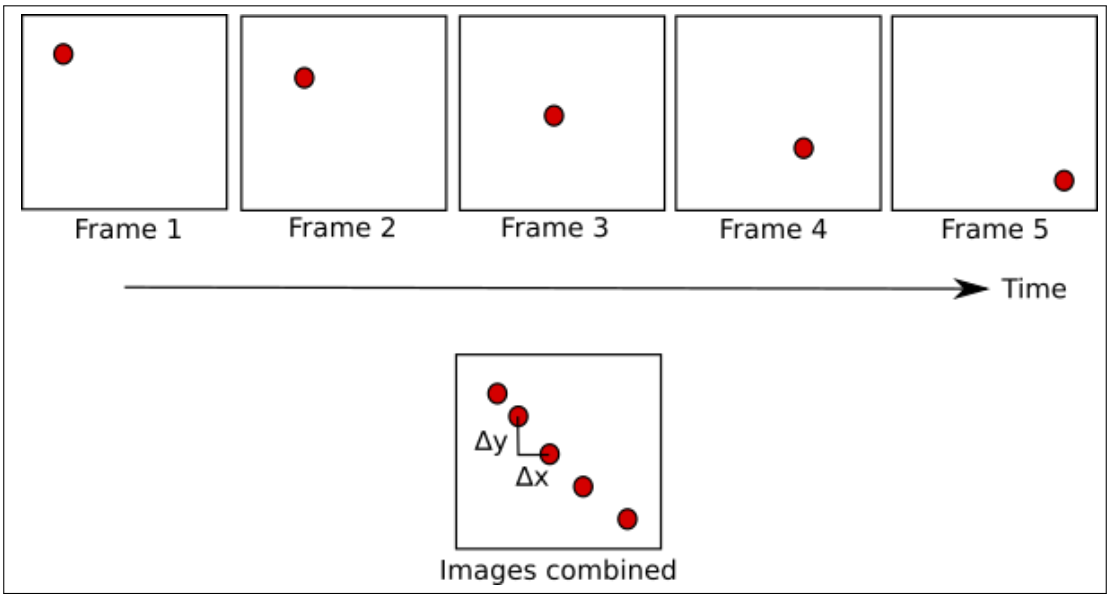

Figure 2: Motion Across Frames

The Figure 2 describes the motion of particle across time. *Optical flow* estimates the velocity of this particle through frames.

Optical flow may be considered as an approximation of *motion field*. In this project, a global method to estimate the optical flow is discussed. Created by B.K.P. Horn and B.G. Schunck in 1981, the method is aptly called as *Horn-Schunk* method.

Mathematical Description

Continuous Description

Let us consider two frames of a video taken $\triangle t$ seconds apart. Let $I(x, y, t)$ be the first frame. Thus, the second frame can be expressed as $I(x + \triangle x, y + \triangle y, t + \triangle t)$ where $(\triangle x, \triangle y)$ is the spatial shifting in the video. Using Taylor Series expansion, the second frame may be expressed as:

$$I(x + \triangle x, y + \triangle y, t + \triangle t) = I(x, y, t) + \triangle x I_x(x, y, t) + \triangle y I_y(x, y, t) + \triangle t I_t(x, y, t) + \mathcal{O}(\triangle x^2, \triangle y^2, \triangle t^2) \quad (1)$$

If the timestep $(\triangle t)$ is small enough, $I(x, y, t)$ and $I(x+\triangle x, y+\triangle y, t+\triangle t)$ may be assumed to be equal. That is,

$$I(x, y, z) \approx I(x + \triangle x, y + \triangle y, t + \triangle t) \quad (2)$$

Putting (2) in (1) and ignoring the higher order terms $(\mathcal{O}(\triangle x^2, \triangle y^2, \triangle t^2))$ in (1), we get,

$$\triangle x I_x(x, y, t) + \triangle y I_y(x, y, t) + \triangle t I_t(x, y, t) = 0$$

Or,

$$\frac{\triangle x}{\triangle t} I_x(x, y, t) + \frac{\triangle y}{\triangle t} I_y(x, y, t) + I_t(x, y, t) = 0$$

The terms $\frac{\triangle x}{\triangle t}$ and $\frac{\triangle y}{\triangle t}$ represent the "velocity" (rate of change of brightness along x and y) for small changes. Let $u = \frac{\triangle x}{\triangle t} \approx \frac{dx}{dt}$ and $v = \frac{\triangle y}{\triangle t} \approx \frac{dy}{dt}$. Thus,

$$\boxed{u I_x(x, y, t) + v \triangle t I_y(x, y, t) + I_t(x, y, t) = 0} \quad (3)$$

The equation (3) is known as ***Brightness Constancy Constraint (BCC)*** equation. Solving the *BCC* equation $\forall (x, Y) \in \Omega$ where Ω is the domain (set of all pixels) of the image would yield the *Optical flow field*, $\begin{pmatrix} u(x, y) \\ v(x, y) \end{pmatrix}$.

However, solving the BCC is not straightforward as the system represents two unknown parameters (u, v) and one equation. Also, the BCC doesn't work at points of singularity or large variation in I_x and I_y. To offset this, we *regularize* the BCC. Regularization may be done by assuming a local constraint or a global constraint in the flow field.

The HS method considers a global constraint. The HS algorithm considers the following energy function (ε):

$$\varepsilon = \int_x \int_y \left[I_x u + I_y v + I_t \right]^2 dx dy \quad (4)$$

Thus, minimizing ε will lead to a solution for flow field. However, as discussed, a constraint needs to be added to make the solution tractable. As a global constraint, we consider penalizing the derivatives of flow fields (u, v) along the domain to ensure that the flow field doesn't vary drastically in its neighborhood. The regularization term (Γ), also known as *smoothness* function, is:

$$\Gamma = \int_x \int_y \left[u_x^2 + u_y^2 + v_x^2 + v_y^2 \right] dx dy \quad (5)$$

Updating (4) with (5), we get,

$$\varepsilon = \int_x \int_y \left[\left[I_x u + I_y v + I_t \right]^2 + \lambda \left(u_x^2 + u_y^2 + v_x^2 + v_y^2 \right) \right] dx dy \quad (6)$$

Where λ is the *smoothness* weight.

Upon observing (6), we note that the equation is a *Euler-Lagrange* equation with the following Lagrangian:

$$\mathcal{L} = \left[I_x u + I_y v + I_t \right]^2 + \lambda \left(u_x^2 + u_y^2 + v_x^2 + v_y^2 \right) \quad (7)$$

From Euler-Lagrange equation, we obtain the following set of equations:

$$\begin{aligned} \mathcal{L}_u - \frac{\partial}{\partial x}\mathcal{L}_{u_x} - \frac{\partial}{\partial y}\mathcal{L}_{u_y} = 0 \\ \mathcal{L}_v - \frac{\partial}{\partial x}\mathcal{L}_{v_x} - \frac{\partial}{\partial y}\mathcal{L}_{v_y} = 0 \end{aligned} \tag{8}$$

Simplifying, we get,

$$\begin{aligned} \bigtriangledown^2 u = \frac{1}{\lambda}(I_x u + I_y v + I_t)I_x \\ \bigtriangledown^2 v = \frac{1}{\lambda}(I_x u + I_y v + I_t)I_y \end{aligned} \tag{9}$$

The equations (9) are referred as Aperture constraint. They resemble a *Poisson* Equation:

$$\bigtriangledown^2 \phi = f$$

In case of (9), $f = (I_x u + I_y v + I_t)I_x$ and $\phi = u$.

Gradient Descent Method

The aim of the exercise is to find the (u, v) corresponding to minimum Energy. The Euler-Lagrange Equation as expressed in (8), describes the gradient of Energy, that is,

$$\begin{aligned} \bigtriangledown\varepsilon = -\lambda \bigtriangledown^2 u + (I_x u + I_y v + I_t)I_x \\ \bigtriangledown\varepsilon = -\lambda \bigtriangledown^2 v + (I_x u + I_y v + I_t)I_y \end{aligned} \tag{10}$$

Thus, the gradient descent update can be expressed as:

$$u_t = - \bigtriangledown \varepsilon \text{ and } v_t = - \bigtriangledown \varepsilon$$

Or,

$$\begin{aligned} u_t = - \bigtriangledown \varepsilon = \lambda \bigtriangledown^2 u - (I_x u + I_y v + I_t)I_x \\ v_t = - \bigtriangledown \varepsilon = \lambda \bigtriangledown^2 v - (I_x u + I_y v + I_t)I_y \end{aligned} \tag{11}$$

This can be achieved in two ways:

- Iterative Methods using (9)
- Gradient Descent based update using (11)

In this project, we discuss both the forms of solutions with emphasis on Gradient Descent Based Update.

Discretization

The Laplacian ($\triangle u = \bigtriangledown^2 u$) can be discretized using First order finite difference method as follows:

$$\triangle u = \bigtriangledown^2 u = \frac{u(x+\Delta x, y) - 2u(x,y) + u(x-\Delta x, y)}{\Delta x^2} + \frac{u(x, y+\Delta y) - 2u(x,y) + u(x, y-\Delta y)}{\Delta y^2} \tag{12}$$

Let us consider an equidistant mesh, that is, $\Delta x = \Delta y$. Thus, the equation (12) simplifies to,

$$\triangle u = \bigtriangledown^2 u = \frac{u(x+\Delta x, y) + u(x-\Delta x, y) + u(x, y+\Delta y) + u(x, y-\Delta y) - 4u(x,y)}{\Delta x^2} \tag{13}$$

The Laplacian can be computed with the following stencil.

0	1	0
1	-4	1
0	1	0

Table 1: Stencil to compute Laplacian

Observe that,

$$\bar{u}(x,y)\Delta x^2 = \frac{u(x+\Delta x, y) + u(x-\Delta x, y) + u(x, y+\Delta y) + u(x, y-\Delta y)}{4}$$

Where $\bar{u}(x,y)$ is the average of u at point (x,y). Thus, the Laplacian may be further simplified as:

$$\Delta u = 4\frac{\bar{u}(x,y) - u(x,y)}{\Delta x^2} \tag{14}$$

Substituting (14) in (9), we get,

$$\begin{aligned} \bar{u}(x,y) - \frac{\Delta x^2}{4\lambda} I_t I_x &= (1 + \frac{\Delta x^2}{4\lambda} I_x^2)u + \frac{\Delta x^2}{4\lambda} I_x I_y v \\ \bar{v}(x,y) - \frac{\Delta y^2}{4\lambda} I_t I_y &= (1 + \frac{\Delta y^2}{4\lambda} I_y^2)v + \frac{\Delta y^2}{4\lambda} I_x I_y u \end{aligned} \tag{15}$$

Or,

$$\begin{bmatrix} \left(1 + \frac{\Delta x^2}{4\lambda} I_x^2\right) & \frac{\Delta x^2}{4\lambda} I_x I_y \\ \frac{\Delta x^2}{4\lambda} I_x I_y & \left(1 + \frac{\Delta y^2}{4\lambda} I_y^2\right) \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \bar{u}(x,y) - \frac{\Delta x^2}{4\lambda} I_t I_x \\ \bar{v}(x,y) - \frac{\Delta y^2}{4\lambda} I_t I_y \end{bmatrix} \tag{16}$$

The Spatio-Temporal partial derivatives I_x, I_y and I_t may be discretized using Foward Finite Difference Approximation as:

$$I_x \approx \frac{1}{4}\Bigg[\frac{I(x+\Delta x,y,t)-I(x,y,t)}{\Delta x}+\frac{I(x+\Delta x,y+\Delta y,t)-I(x,y+\Delta y,t)}{\Delta x} + \frac{I(x+\Delta x,y,t+\Delta t)-I(x,y,t+\Delta t)}{\Delta x}+\frac{I(x+\Delta x,y+\Delta y,t+\Delta t)-I(x,y+\Delta y,t+\Delta t)}{\Delta x}\Bigg]$$

$$I_y \approx \frac{1}{4}\Bigg[\frac{I(x,y+\Delta y,t)-I(x,y,t)}{\Delta y}+\frac{I(x+\Delta x,y+\Delta y,t)-I(x+\Delta x,y,t)}{\Delta y} + \frac{I(x,y+\Delta y,t+\Delta t)-I(x,y,t+\Delta t)}{\Delta y}+\frac{I(x+\Delta x,y+\Delta y,t+\Delta t)-I(x+\Delta x,y,t+\Delta t)}{\Delta y}\Bigg]$$

$$I_t \approx \frac{1}{4}\Bigg[\frac{I(x,y,t+\Delta t)-I(x,y,t)}{\Delta t}+\frac{I(x+\Delta x,y,t+\Delta t)-I(x+\Delta y,y,t)}{\Delta t} + \frac{I(x,y+\Delta y,t+\Delta t)-I(x,y+\Delta y,t)}{\Delta t}+\frac{I(x+\Delta x,y+\Delta y,t+\Delta t)-I(x+\Delta x,y+\Delta y,t)}{\Delta t}\Bigg]$$

Since I_x, I_y and I_t can be computed, the equation (9) represents a Linear system. Such a system can be solved as follows:

$$\boxed{\begin{bmatrix}u\\v\end{bmatrix}=\frac{1}{\begin{vmatrix}\left(1+\frac{\Delta x^2}{4\lambda}I_x^2\right) & \frac{\Delta x^2}{4\lambda}I_xI_y\\ \frac{\Delta x^2}{4\lambda}I_xI_y & \left(1+\frac{\Delta y^2}{4\lambda}I_y^2\right)\end{vmatrix}}\begin{bmatrix}\left(1+\frac{\Delta y^2}{4\lambda}I_y^2\right) & -\frac{\Delta x^2}{4\lambda}I_xI_y\\ -\frac{\Delta x^2}{4\lambda}I_xI_y & \left(1+\frac{\Delta x^2}{4\lambda}I_x^2\right)\end{bmatrix}\begin{bmatrix}\bar{u}(x,y)-\frac{\Delta x^2}{4\lambda}I_tI_x\\ \bar{v}(x,y)-\frac{\Delta y^2}{4\lambda}I_tI_y\end{bmatrix}} \tag{17}$$

The (17) *represents the iterative solution.*

Gradient Descent based Discretization

The temporal differentiation of u can be approximated as:

$$u_t=\frac{u(x,y,t+\Delta t)-u(x,y,t)}{\Delta t} \tag{18}$$

Substituting the results obtained in (13) and (18) in (11) as follows:

$$\boxed{\begin{aligned}\frac{u(x,y,t+\Delta t)-u(x,y,t)}{\Delta t}&=\lambda\left(\frac{u(x+\Delta x,y)+u(x-\Delta x,y)+u(x,y+\Delta y)+u(x,y-\Delta y)-4u(x,y)}{\Delta x^2}\right)\\&\quad-(I_xu+I_yv+I_t)I_x\\ \frac{v(x,y,t+\Delta t)-v(x,y,t)}{\Delta t}&=\lambda\left(\frac{u(x+\Delta x,y)+u(x-\Delta x,y)+u(x,y+\Delta y)+u(x,y-\Delta y)-4u(x,y)}{\Delta y^2}\right)\\&\quad-(I_xu+I_yv+I_t)I_y\end{aligned}} \tag{19}$$

Courant–Friedrichs–Lewy (CFL) condition

The *Courant–Friedrichs–Lewy* (CFL) condition is a necessary condition for convergence while solving (9) numerically. For a two-Dimensional Partial Differential Equation (PDE), the CFL condition is given by:

$$\frac{|u|\Delta t}{\Delta x} + \frac{|v|\Delta t}{\Delta y} \leq C_{max}$$

where
$|u|, |v|$ are the magnitude of speed,
$\Delta x, \Delta y$ is the grid interval,
Δt is the timestep.

In our case, we maintain a similar grid interval, that is, $\Delta x = \Delta y$. A natural choice for CFL condition is that the distance covered by the image in time Δt will be less than the Δx and Δy so that two consecutive images remain in the same grid element. In other words,

$$\boxed{\Delta t \leq \Delta x \cdot min\left(\frac{1}{|u|}, \frac{1}{|v|}\right)} \tag{20}$$

Since, no apriori knowledge of $min\left(\frac{1}{|u|}, \frac{1}{|v|}\right)$ exists, a sufficiently small number for $min\left(\frac{1}{|u|}, \frac{1}{|v|}\right)$ is substituted.

Let us now examine the case of CFL condition for Gradient Descent. The discretized gradient descent formule (19) is:

$$\begin{aligned}\frac{u(x,y,t+\Delta t) - u(x,y,t)}{\Delta t} = \lambda\Bigg(&\frac{u(x+\Delta x,y) + u(x-\Delta x,y) + u(x,y+\Delta y) + u(x,y-\Delta y) - 4u(x,y)}{\Delta x^2}\Bigg)\\ &- (I_x u + I_y v + I_t)I_x\end{aligned}$$

Taking Discrete Fourier transform on both sides, we get,

$$\begin{aligned}\frac{U(k,l,\tau+\Delta\tau) - U(k,l,\tau)}{\Delta\tau} = \lambda\Bigg(&\frac{U(k,l,\tau)e^{j\Delta xk} + U(k,l,\tau)e^{-j\Delta xk} + U(k,l,\tau)e^{j\Delta yl} + U(k,l,\tau)e^{-j\Delta yl} - 4U(k,l,\tau)}{\Delta x^2}\Bigg)\\ &- I_x^2 U(k,l,\tau) - I_x I_y V(k,l,\tau) - I_t I_x\end{aligned}$$

Or,

$$\begin{aligned}\frac{U(k,l,\tau+\Delta\tau) - U(k,l,\tau)}{\Delta\tau} = \lambda\Bigg(&\frac{e^{j\Delta xk} + e^{-j\Delta xk} + e^{j\Delta yl} + e^{-j\Delta yl} - 4}{\Delta x^2}U(k,l,\tau)\Bigg)\\ &- I_x^2 U(k,l,\tau) - I_x I_y V(k,l,\tau) - I_t I_x\end{aligned}$$

Using Euler's formula, we get,

$$\begin{aligned}\frac{U(k,l,\tau+\Delta\tau) - U(k,l,\tau)}{\Delta\tau} = \lambda\Bigg(&\frac{2cos(\Delta xk) + 2cos(\Delta yl) - 4}{\Delta x^2}U(k,l,\tau)\Bigg)\\ &- I_x^2 U(k,l,\tau) - I_x I_y V(k,l,\tau) - I_t I_x\end{aligned}$$

Further Simplifying, we get,

$$\frac{U(k,l,\tau+\Delta\tau) - U(k,l,\tau)}{\Delta\tau} = \lambda\left(\frac{2cos(\Delta xk) + 2cos(\Delta yl) - 4}{\Delta x^2} - I_x^2\right)U(k,l,\tau) - I_x I_y V(k,l,\tau) - I_t I_x$$

Or,

$$U(k,l,\tau+\Delta\tau)-U(k,l,\tau)=\lambda\Delta\tau\left(\frac{2cos(\Delta xk)+2cos(\Delta yl)-4}{\Delta x^2}-I_x^2\right)U(k,l,\tau)-\Delta\tau I_xI_yV(k,l,\tau)-\tau I_tI_x$$

Or,

$$U(k,l,\tau+\Delta\tau)=\left(\lambda\Delta\tau\frac{2cos(\Delta xk)+2cos(\Delta yl)-4}{\Delta x^2}-\lambda\Delta\tau I_x^2+1\right)U(k,l,\tau)-\Delta\tau I_xI_yV(k,l,\tau)-\tau I_tI_x$$

Or,

$$U(k,l,\tau+\Delta\tau)=\left(\lambda\Delta\tau\frac{2cos(\Delta xk)+2cos(\Delta yl)-4-\Delta x^2I_x^2}{\Delta x^2}+1\right)U(k,l,\tau)-\Delta\tau I_xI_yV(k,l,\tau)-\tau I_tI_x$$

Let

$$\alpha(k,l)=\lambda\Delta\tau\frac{2cos(\Delta xk)+2cos(\Delta yl)-4-\Delta x^2I_x^2}{\Delta x^2}+1$$

Thus,

$$\boxed{U(k,l,\tau+\Delta\tau)=\alpha(k,l)U(k,l,\tau)-\Delta\tau I_xI_yV(k,l,\tau)-\tau I_tI_x} \tag{21}$$

Observing equation (21), the amplification factor is $\alpha(k,l)$. The CFL condition in such a scenario is:

$$|\alpha(k,l)|^2\leq 1$$

Thus,

$$\left(\lambda\Delta\tau\frac{2cos(\Delta xk)+2cos(\Delta yl)-4-\Delta x^2I_x^2}{\Delta x^2}+1\right)^2\leq 1$$

Or,

$$\left(\lambda\Delta\tau\frac{2cos(\Delta xk)+2cos(\Delta yl)-4-\Delta x^2I_x^2}{\Delta x^2}\right)^2+2\lambda\Delta\tau\frac{2cos(\Delta xk)+2cos(\Delta yl)-4-\Delta x^2I_x^2}{\Delta x^2}+1^2\leq 1$$

Or,

$$\left(\lambda\Delta\tau\frac{2cos(\Delta xk)+2cos(\Delta yl)-4-\Delta x^2I_x^2}{\Delta x^2}\right)^2+2\lambda\Delta\tau\frac{2cos(\Delta xk)+2cos(\Delta yl)-4-\Delta x^2I_x^2}{\Delta x^2}\leq 0$$

Let

$$\gamma=\lambda\Delta\tau\frac{2cos(\Delta xk)+2cos(\Delta yl)-4-\Delta x^2I_x^2}{\Delta x^2}$$

Thus,

$$\gamma^2+2\gamma\leq 0$$

Or,

$$\gamma(\gamma+2)\leq 0$$

Observing the above condition, we conclude,

$$-2\leq\gamma\leq 0$$

Observing the formulation of γ, we note that $\Delta\tau>0$, $\lambda>0$ (by choice), $\Delta x^2>0$ and $I_x^2\geq 0$. Thus, $\gamma\leq 0$ always holds true and doesn't need to be explored further. Let us now consider

$$\gamma\geq -2$$

That is,

$$\lambda\Delta\tau\frac{2cos(\Delta xk)+2cos(\Delta yl)-4-\Delta x^2I_x^2}{\Delta x^2}\geq -2$$

Or,

$$\lambda\Delta\tau\frac{4+\Delta x^2I_x^2-2cos(\Delta xk)-2cos(\Delta yl)}{\Delta x^2}\leq 2$$

Thus,

$$\boxed{\Delta\tau\leq\frac{\Delta x^2}{\lambda\Big(4+\Delta x^2I_x^2-2cos(\Delta xk)-2cos(\Delta yl)\Big)}} \tag{22}$$

An analogous result can be obtained in terms of Δy by using update rule for v. This is so because $\Delta x=\Delta y$. Thus,

$$\boxed{\begin{aligned}\Delta\tau_1&\leq\frac{\Delta x^2}{\lambda\Big(4+\Delta x^2I_x^2-2cos(\Delta xk)-2cos(\Delta yl)\Big)}\\ \Delta\tau_2&\leq\frac{\Delta y^2}{\lambda\Big(4+\Delta x^2I_y^2-2cos(\Delta xk)-2cos(\Delta yl)\Big)}\end{aligned}} \tag{23}$$

Akin to (20), the timestep $\Delta\tau$ is chosen to be the minimum of τ_1 and τ_2. That is,

$$\boxed{\Delta\tau=\min\left(\Delta\tau_1,\Delta\tau_2\right)}$$

Implementation

In the scenario of implementation, $\Delta x=\Delta y=1$. Thus, the gradient descent update rule becomes:

$$\boxed{\begin{aligned}u(x,y,t+1)=u(x,y,t)+\Delta t\Bigg[\lambda\Bigg(&u(x+\Delta x,y)+u(x-\Delta x,y)+u(x,y+\Delta y)+u(x,y-\Delta y)-4u(x,y)\Bigg)\\&-I_x^2u-I_xI_yv-I_tI_x\Bigg]\\ v(x,y,t+1)=v(x,y,t)+\Delta t\Bigg[\lambda\Bigg(&v(x+\Delta x,y)+v(x-\Delta x,y)+v(x,y+\Delta y)+v(x,y-\Delta y)-4v(x,y)\Bigg)\\&-I_y^2v-I_xI_yu-I_tI_y\Bigg]\end{aligned}} \tag{24}$$

And the spatio-temporal derivatives are calculated as follows:

$$\begin{aligned}I_x&=\frac{1}{4}\begin{bmatrix}-1&1\\-1&1\end{bmatrix}\circledast(I_t+I_{t+\Delta t})\\ I_y&=\frac{1}{4}\begin{bmatrix}-1&-1\\1&1\end{bmatrix}\circledast(I_t+I_{t+\Delta t})\\ I_t&=\frac{1}{4}\begin{bmatrix}1&1\\1&1\end{bmatrix}\circledast I_{t+\Delta t}+\frac{1}{4}\begin{bmatrix}-1&-1\\-1&-1\end{bmatrix}\circledast I_t\\ \Delta u&=\begin{bmatrix}0&1&0\\1&-4&1\\0&1&0\end{bmatrix}\circledast u\\ \Delta v&=\begin{bmatrix}0&1&0\\1&-4&1\\0&1&0\end{bmatrix}\circledast v\end{aligned}$$

The velocities u and v are initialized as 0. We set Δt with a small value to ensure CFL condition is not violated.

Experimental Set up and Results

Experiment

A set of synthetic images along with their ground truth were used.

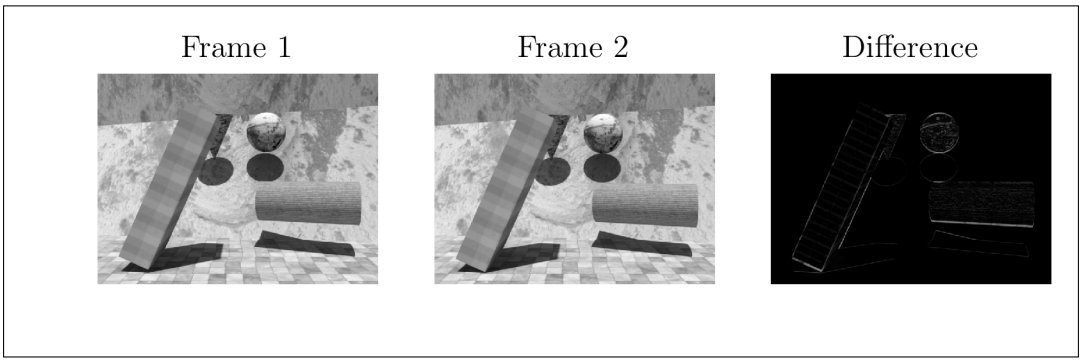

Figure 3: Frames of Synthetic Image used in the experiment

Benchmark

To evaluate the performance of algorithm, two metrics were used:

- **End Point Error (EPE).** EPE is the average norm of the estimated flow vectors and ground truth flow vectors.
- **Angular Error (AE).** AE is the average angle between the estimated flow vectors and ground truth flow vectors.
- **Time.** Time taken for convergence plays an important role in real time processing.

These metrics were used to fix the value of λ. The optimal λ, or λ^*. corresponds to the λ with lowest EPE and AE.

Results

Forstly, we found optimal number of iterations by setting $\alpha = 1, \Delta t = 0.1$ and varying the number of iterations. The following results were obtained:

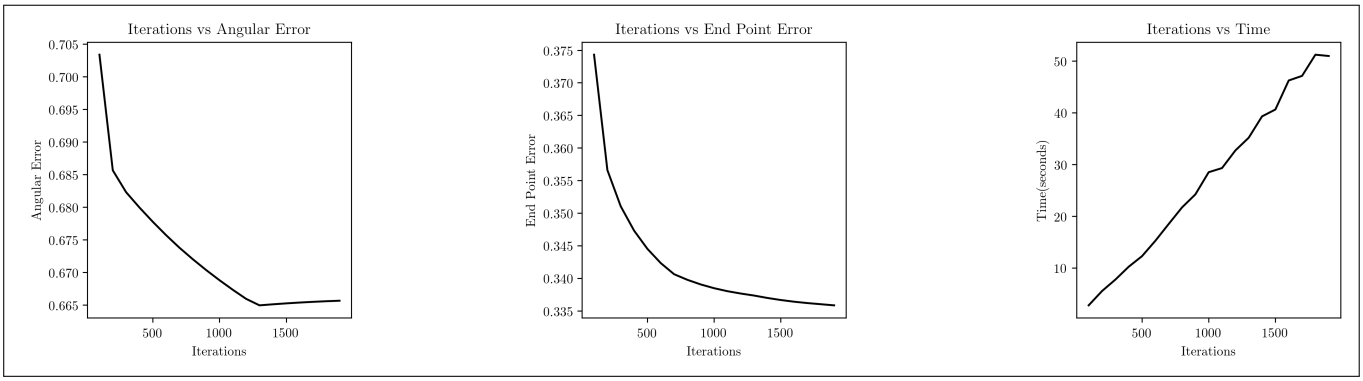

Figure 4: Iteration vs Errors

The iterations was chosen to be fixed at 750 as a reasonable trade-off between error and time. Next, the value of alpha was chosen based on minimum EPE, AE. The following results were obtained:

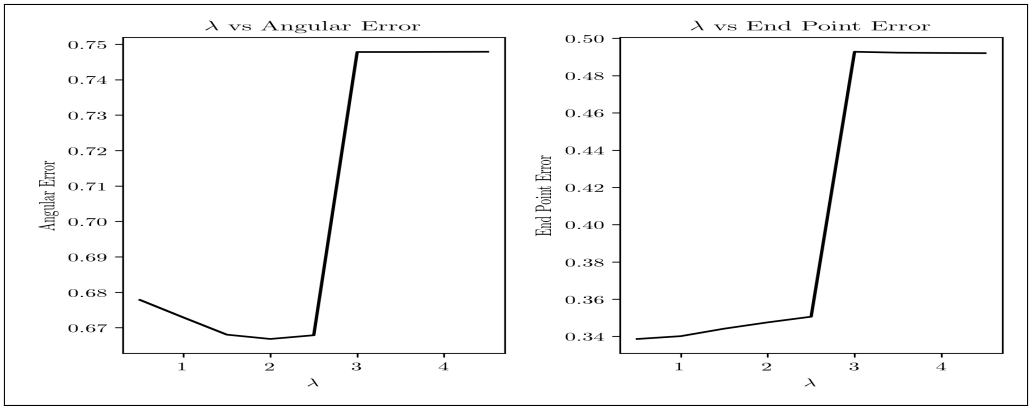

Figure 5: λ vs Errors

Based on the results, $\lambda = 2$ was chosen to be optimal. Thus, the optimum solution comprised of the following parameters:

Parameter	**Value**	**Heuristics**
λ	2s	Minimum EPE,AE
Iterations	750	Time
Δt	0.1	Trial and Error

Table 2: Optimal Parameters

Running the optimal solution on a set of example image, we obtain the following:

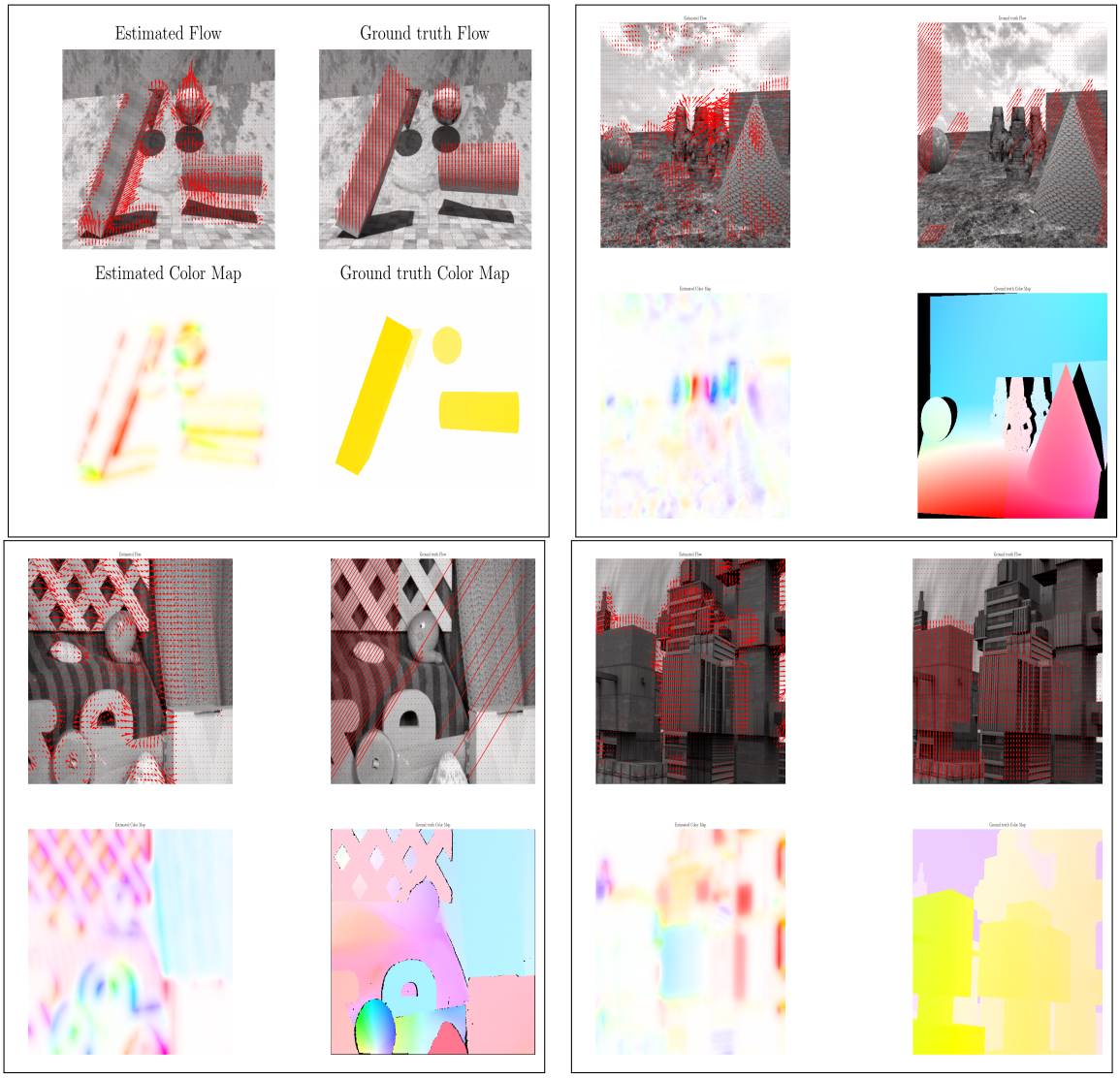

Figure 6: Estimated Flow from Optimal Model for Example Images

Future Works

The Horn-Schunck method suffers from the following drawbacks:

- It is computationally expensive.
- Being a global method, the optimal solution works well only for the family of images for which the optimality was arrived at. This is evident in Figure 6.

Further work can be done on combining elements of global and local solutions.

References

[1] B. K. Horn and B. G. Schunck, "Determining optical flow," Cambridge, MA, USA, Tech. Rep., 1980.

[2] C. Liu, *Motion estimation i*. [Online]. Available: http://6.869.csail.mit.edu/fa13/lectures/MotionEstimation1.pdf.

[3] A. Efros, *Taking derivation by convolution*. [Online]. Available: https://inst.eecs.berkeley.edu/~cs194-26/fa17/Lectures/ConvEdgesTemplate.pdf.

[4] C. Perez-Arancibia, *Von neumann stability analysis*. [Online]. Available: https://ocw.mit.edu/courses/mathematics/18-336-numerical-methods-for-partial-differential-equations-spring-2009/lecture-notes/MIT18_336S09_lec14.pdf.

Appendix

Codes

```
class opticalFlow:
    """Class to implement Horn Schunk Optical Flow"""

    def __init__(self, path, nos_itr = 10, alpha = 1.0):
        """Initialize Path
        """
        frame1_path = os.path.join(path, 'frame1.png')
        frame2_path = os.path.join(path, 'frame2.png')
        self.nos_itr = nos_itr
        self.alpha = alpha
        # Loading the Image
        image = cv2.imread(frame1_path, cv2.IMREAD_GRAYSCALE)
        self.frame1 = cv2.normalize(image,
                                    None,
                                    alpha=0,
                                    beta=1,
                                    norm_type=cv2.NORM_MINMAX,
                                    dtype=cv2.CV_32F)
        image = cv2.imread(frame2_path, cv2.IMREAD_GRAYSCALE)
        self.frame2 = cv2.normalize(image,
                                    None,
                                    alpha=0,
                                    beta=1,
                                    norm_type=cv2.NORM_MINMAX,
                                    dtype=cv2.CV_32F)
        # Reading the ground truth
        ground_truth_flow = read_flow_file(os.path.join(path, 'flow1_2.flo'))
        u_gt_orig = ground_truth_flow[:, :, 0]
        v_gt_orig = ground_truth_flow[:, :, 1]
        self.u_gt = np.where(np.isnan(u_gt_orig), 0, u_gt_orig)
        self.v_gt = np.where(np.isnan(v_gt_orig), 0, v_gt_orig)

    def display_frames(self):
        """Display initial frames"""
        save_path = 'results/frames.png'
        difference = np.abs(self.frame1 - self.frame2)
        # Plotting
        plt.figure()
        plt.title(r"Plot of $p_{X|Y}(x|1)$ vs $x$")
        ax1 = plt.subplot(131)
        plt.axis('off')
        plt.imshow(self.frame1, cmap='gray', vmin=0.0, vmax=1.0)
        ax1.set_title(r'Frame 1')
        ax2 = plt.subplot(132)
        plt.axis('off')
        plt.imshow(self.frame2, cmap='gray', vmin=0.0, vmax=1.0)
        ax2.set_title(r'Frame 2')
        ax3 = plt.subplot(133)
        plt.axis('off')
        plt.imshow(difference, cmap='gray', vmin=0.0, vmax=1.0)
        ax3.set_title(r'Difference')
        fig = plt.gcf()
        plt.show()
        fig.savefig(save_path, dpi=300, bbox_inches='tight')

    def gradients(self, frame1, frame2):
        """Function to compute I_x, I_y, I_t"""
        mask_x = np.array([ [-1/4, 1/4],
                            [-1/4,1/4]])
        mask_y = np.array([ [-1/4, -1/4],
                            [1/4,1/4]])
        mask_t_2 = np.array([   [-1/4, -1/4],
                                [-1/4,-1/4]])
        mask_t_1 = np.array([   [1/4, 1/4],
```

```
                                [1/4,1/4]])
        I_x = signal.convolve2d(frame1,
                                mask_x,
                                mode='same',
                                boundary='symm')\
            + signal.convolve2d(frame2,
                                mask_x,
                                mode='same',
                                boundary='symm')
        I_y = signal.convolve2d(frame1,
                                mask_y,
                                mode='same',
                                boundary='symm')\
            + signal.convolve2d(frame2,
                                mask_y,
                                mode='same',
                                boundary='symm')
        I_t = signal.convolve2d(frame1,
                                mask_t_1,
                                mode='same',
                                boundary='symm')\
            + signal.convolve2d(frame2,
                                mask_t_2,
                                mode='same',
                                boundary='symm')
        return [I_x, I_y, I_t]

    def laplacian(self, image):
        """Function to compute Laplacian"""
        laplacian_kernel = np.array([   [0   , 1,  0],
                                        [1,  -4 , 1],
                                        [0   , 1,  0]])
        delta_I = signal.convolve2d(image,
                                    laplacian_kernel,
                                    mode='same',
                                    boundary='symm')
        return delta_I

    def horn_schunck(self, alpha=1, nos_itr = 100, delta_t = 0.1):
        "Function to perform HS optical flow"
        start_time = time.time()
        self.alpha = alpha
        self.nos_itr = nos_itr
        # Initializing u,v
        u = np.zeros_like(self.frame1)
        v = np.zeros_like(self.frame2)
        # Getting Gradients
        I_x, I_y, I_t = self.gradients(self.frame1, self.frame2)
        # Iterating over frames
        for i in range(self.nos_itr):
            delta_u = self.laplacian(u)
            delta_v = self.laplacian(v)
            u = u + delta_t\
                *(alpha*delta_u - u*(np.power(I_x,2)) - I_x*I_y*v - I_t*I_x)
            v = v + delta_t\
                *(alpha*delta_v - v*(np.power(I_y,2)) - I_x*I_y*u - I_t*I_y)
            # Convert Nans to zero
        u[np.isnan(u)] = 0
        v[np.isnan(v)] = 0
        end_time = time.time()
        self.time = end_time - start_time
        return u, v

    def get_ground_truth(self):
        """Function to return ground truth"""
        return self.u_gt, self.v_gt

    def downsample_flow(self, x, y, stride=10):
        """Function to return downsampled version of signal"""
        return x[::stride, ::stride], y[::stride, ::stride]
```

```
135
136     def plot_flow(self, u, v, stride=10):
137         """Plot Stride"""
138         save_path = 'results/flowplot.png'
139         # Defining Meshgrid for Quiver
140         m, n = self.frame1.shape
141         x, y = np.meshgrid(range(n), range(m))
142         x = x.astype('float64')
143         y = y.astype('float64')
144         # Downsampling for better visibility
145         u_gt_discrete, v_gt_discrete = self.downsample_flow(self.u_gt,
146                                                             self.v_gt,
147                                                             stride=stride)
148         u_discrete, v_discrete = self.downsample_flow(  u,
149                                                         v,
150                                                         stride=stride)
151         x_discrete, y_discrete = self.downsample_flow(  x,
152                                                         y,
153                                                         stride=stride)
154         # Flow Matrix
155         estimated_flow = np.stack((u, v), axis=2)
156         gt_flow = np.stack((self.u_gt, self.v_gt), axis=2)
157         # Plot the optical flow field
158         plt.figure()
159         ax1 = plt.subplot(2, 2, 1)
160         plt.axis('off')
161         plt.imshow(self.frame2, cmap='gray')
162         plt.quiver(x_discrete, y_discrete,
163                    u_discrete, v_discrete,
164                    color='r')
165         ax1.set_title(r'Estimated Flow')
166         ax2 = plt.subplot(2, 2, 2)
167         plt.axis('off')
168         plt.imshow(self.frame2, cmap='gray')
169         plt.quiver(x_discrete, y_discrete,
170                    u_gt_discrete, v_gt_discrete,
171                    color='r')
172         ax2.set_title(r'Ground truth Flow')
173         ax3 = plt.subplot(2, 2, 3)
174         plt.axis('off')
175         plt.imshow(flow_to_color(estimated_flow))
176         ax3.set_title(r'Estimated Color Map')
177         ax4 = plt.subplot(2, 2, 4)
178         plt.axis('off')
179         plt.imshow(flow_to_color(gt_flow))
180         ax4.set_title(r'Ground truth Color Map')
181         fig = plt.gcf()
182         plt.show()
183         fig.savefig(save_path, dpi=300, bbox_inches='tight')
184
185     def normalize(self, x):
186         """Function to perform min-max normalization"""
187         return (x -np.min(x)) / (np.max(x) - np.min(x))
188
189     def norm(self, x, x_g, y, y_g):
190         """Function to compute l-2 norm"""
191         return ((x-x_g)**2 + (y-y_g)**2)**0.5
192
193     def angle(self, x, x_g, y, y_g):
194         """Function to compute angle"""
195         return np.arccos(((x*x_g)+(y*y_g)+1)/(((x+y+1) * (x_g+y_g+1))**0.5))
196
197     def benchmark_flow(self, u, v):
198         """Function to benchmark Optical Flow"""
199         log_file = "results/log.csv"
200         # Normalizing flows
201         norm_u = self.normalize(u)
202         norm_v = self.normalize(v)
203         norm_u_gt = self.normalize(self.u_gt)
204         norm_v_gt = self.normalize(self.v_gt)
```

```
        # Computing End-Point Error and Angular Error
        EPE = self.norm(norm_u, norm_u_gt, norm_v, norm_v_gt)
        AE = self.angle(norm_u, norm_u_gt, norm_v, norm_v_gt)
        # Computing the average error
        EPE_avg = np.mean(EPE[~np.isnan(EPE)])
        AE_avg = np.mean(AE[~np.isnan(AE)])
        # Printing Error
        msg = "Itr = %d, Time(s) = %.2f, Alpha = %.2f, EPE = %.2f, AE = %.2f"\
            %(self.nos_itr, self.time, self.alpha, EPE_avg, AE_avg)
        print (msg)
        # Logging the error and display
        fields = [self.nos_itr, self.time, self.alpha, EPE_avg, AE_avg]
        with open(log_file,'a') as f:
            writer = csv.writer(f)
            writer.writerow(fields)
```