NORTHEASTERN UNIVERSITY

# Computer Vision Project #3: Optical Flow

Jordan Helderman
Justin Hess

3 December 2017

In computer vision, there is sometimes a need for understanding the apparent displacement of an object of interest between several images of a scene, whether it is due to motion of the objects in the scene or the motion of the camera. This information is known as the optical flow, and it is useful in many video based tracking applications. In this report, we explore one such algorithm for estimating the optical flow, the Lucas-Kanade algorithm. We give a description of the algorithm and show that it can be used to estimate the flow vectors in a benchmark set of images

## 1. INTRODUCTION

Optical flow measures the displacement of objects between objects in two images of the same scene, and it is an quantity of interest in video based object tracking applications and similar tasks. The Lucas-Kanade algorithm is well known algorithm for estimating the optical flow between two images of interest. In the following sections, we will describe our implementation of the Lucas-Kanade algorithm and show its performance on a benchmark set of images. In section 2, we will give a detailed description of our implementation of the algorithm; section 3 will give a presentation of our experimental results and a discussion of their quality; and in section 4, we will summarize our results and discuss additional work we could do in this area.

## 2. Algorithm Descripion

The implementation of the Lucas Kanade algorithm assumes that the image brightness constancy equation yields an accurate estimation of the optical flow. In order for this principle to be satisfied, the local optimal opical flow vector $(u, v)$ must satisfy the following equation where $(I_x, I_y)$ is the spatial gradient and $I_t$ is the temporal gradient.

$$I_x(x, y)\,u + I_y(x, y)\,v + I_t(x, y) = 0$$

Since the optical flow has two unknowns, we cannot completely determine it from this equation. To get more equations, we assume that the optical flow is constant in a region around the point of interest. Using the points in this neighborhood, we can write as many more equations to create an overdetermined system and solve for the optical flow using a least-squares method. This method yields the following system of linear equations where $N$ is the set of pixels in the neighborhood's region of support.

$$\begin{bmatrix} \sum_N I_x^2 & \sum_N I_x I_y \\ \sum_N I_x I_y & \sum_N I_y^2 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = - \begin{bmatrix} \sum_N I_x I_t \\ \sum_N I_y I_t \end{bmatrix}$$

In the ideal case, to find the optical flow, we would simply solve this linear system of equations at each pixel in the image. However, there are a couple of factors that complicate the solution of these equations. First, the computation of derivatives is known to amplify noise. Consequently, if we solve these equations whose coefficients depend on derivative terms, our optical flow vectors will necessarily contain noise. We mitigate this effect by applying a lowpass filter before we take derivatives. Specifically, we use a Gaussian filter with a standard deviation of 1 pixel in both the spatial and temporal dimensions.

Additionally, we made the assumption that the optical flow is constant in the neighborhood. Unfortunately, this assumption does not always hold, and noise in our estimate of the optical flow is a natural result. To reduce this effect, we apply weights to the equations in the neighborhood that assign higher importance to points near to the point of interest. With the weights added, our system of linear equations then become the following.

$$\begin{bmatrix} \sum_N w^2 I_x^2 & \sum_N w^2 I_x I_y \\ \sum_N w^2 I_x I_y & \sum_N w^2 I_y^2 \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = - \begin{bmatrix} \sum_N w^2 I_x I_t \\ \sum_N w^2 I_y I_t \end{bmatrix}$$

We use a set of weights which approximate a Gaussian filter with a standard deviation of 2 pixels in each dimension.

Another problem that the Lucas-Kanade algorithm has is that the algorithm does not detect large motions in the scene. This issue manifests as flow vector estimates that are seemingly random. We address this issue by estimating the optical flow independently at two different scales. The set of multiple resolution images is called an image pyramid. However, there is a better method which operates on a larger image pyramid and iteratively estimates the flow from the most coarse sampling to the most fine sampling.

Finally, the matrix on the left may not be invertible. From our work with the Harris corner matrix, we know that this matrix will be invertible in regions of the image where there is a corner. Therefore, before we try to solve the system of equations, we check to make sure that both eigenvalues of the system are above a certian noise threshold. In some cases, only one of the eigenvalues will be above the threshold. We know, also from the Harris corner matrix, that this case corresponds to an edge. In this case, we will approximate the optical flow via a differential technique from [1] which is described in the following equation. In all other cases, the optical flow is said to be the zero vector.

$$\begin{bmatrix} u \\ v \end{bmatrix} = \frac{-I_t \nabla I}{|\nabla I|^2}$$

Putting this all together, we first smooth the input images with a simple gaussian kernel to smooth out the noise.

```
% smooth in the spatial directions
h = gaussian2d(11, 11, 1 * eye(2));
I = imfilter(I, h, 'same');
% smooth in the time direction
I = imfilter(I, permute(gaussian1d(3, 1), [1, 3, 2]), 'same');
```

Then, with the smoothed images, we can apply our kernel for computing the gradients at each pixel in our input window and calculate the gradients and their products and apply the neighborhood to these products:

```
%% compute spatial gradient
hx = -[-1, 8, 0, -8, 1] / 12;
[Ix, Iy] = imgrad(I, hx);

%% compute temporal gradient
It = temporal_grad(I);

%% compute gradient products
Ix2 = Ix.^2;
Iy2 = Iy.^2;
Ixy = Ix .* Iy;
Ixt = Ix .* It;
Iyt = Iy .* It;

%% apply neighborhood
Ix2 = imfilter(Ix2, w, 'same');
Iy2 = imfilter(Iy2, w, 'same');
Ixy = imfilter(Ixy, w, 'same');
Ixt = imfilter(Ixt, w, 'same');
Iyt = imfilter(Iyt, w, 'same');
```

Next, we form the system of equations we derived for the optical flow with the following MATLAB code, which initializes the velocity vector (u, v), gets the size of the image to loop through all pixels in the input images, and calculate matrices system of equations in the given window:

```
u = zeros(size(I));
v = zeros(size(I));
e1 = zeros(size(I));
e2 = zeros(size(I));
for m = 1:size(I, 1)
    for n = 1:size(I,2)
        for p = 1:size(I,3)
            A = [Ix2(m,n,p), Ixy(m,n,p); Ixy(m,n,p), Iy2(m,n,p)];
            b = [-Ixt(m,n); -Iyt(m,n)];
```

Finally, we compute the eigenvalues and estimate the flow based on how many eigenvalues are above the noise threshold.

```
            % compute the eigenvalues
            e = eig(A);
            e1(m,n,p) = e(1);
            e2(m,n,p) = e(2);
            % check to see whether we're on a corner or edge
            if all(e > tau)
                % matrix is non-singular. just solve the system of
                % equations
                x = A^-1 * b;
                u(m,n,p) = x(2);
                v(m,n,p) = x(1);
            elseif any(e > tau)
                % matrix is singular but we are still on an edge. estimate
                % the optical flow as the normal velocity vector
                grad = [Ix(m,n,p), Iy(m,n,p)];
                grad_norm = norm(grad);
                s = -It(m, n, p) / grad_norm;
                if grad_norm < 1e-3
                    norm_velocity = [0, 0];
                else
                    norm_velocity = s * grad / grad_norm;
                end
                u(m,n,p) = norm_velocity(1);
                v(m,n,p) = norm_velocity(2);
            end
        end
    end
end
```

## 3. Experimental Results and Discussion

We, first, tested our algorithm on an sequence of images that show a noiseless square moving to the right by 10 pixels. This sequence of images can be seen in figures 3.1-2. We chose this test sequence because it is one of the most simple images that could be used to test an optical flow estimate. In this case, we would expect the flow on the square to be proportional to the shift in the same direction and the flow on all other regions to be zero. Applying the Lucas-Kanade algorithm to both levels of the image pyramid, we get the flow vectors depicted in figures 3.3-4. In both levels of the pyramid, we can see that there are flow vectors on the left and right edges of the square that point in the direction of motion and no flow vectors in the region outside the square, as we expected. We can also see that there are some flow vectors on the top and bottom edges of the square that point orthogonal to the edge. This is due to the differential estimate of the optical flow in this region. Since these pixels are on an edge, they only have one large eigenvalue and, consequently, have to be estimated as the time derivative multiplied by the normalized, spatial gradient. Additionally, we can see that there are no flow vectors on the interior of the square. This is due to the fact that there is no texture on the square to indicate that it has moved.

We also experimented with changing the effective size of our neighborhood. Since we used a Gaussian weighted neighborhood, we also change the standard deviation in tandem with the size of the neighborhood. We found that increasing the standard deviation and size too much resulted in optical flow information being lost. This can be seen in figure 3.5, which is the same example as the high resolution flow from before but with a larger neighborhood. We can see that the large neighborhood results on the corners of the box are mostly gone, leaving those on the edge. We found that we start to lose a significant amount of information with a Gaussian kernel with standard deviation 4 and an 11-by-11 kernel size. This led us to chose a Gaussian neighborhood with a standard deviation of 2 pixels and a 5-by-5 kernel size.

Additionally, we tested the algorithm on some benchmark images. We'll take a look at the performance of a representative set here. The images we consider can be seen in figures 3.6-7, and the result of applying our implementation of the Lucas-Kanade algorithm can be seen in figures 3.8-9. Based on the original images we would expect similar results as the simple example we tried before, because the head moves by a small number of pixels to the right between the two images. However, we do not get such clean results. In the image at the original resolution, we can see that many of the flow vectors are pointing in the direction we expect. However, there are also many vectors that are pointing in directions that are unrelated to the direction of motion. This is likely due, in part, to the problem that the Lucas-Kanade has with motion that is relatively large motions. This is supported by the more consistent results we get in lower resolution image.

## 4. Conclusion

To summarize, we implemented and tested a version of the Lucas-Kanade algorithm, and we showed that it can be used to compute the optical flow on a benchmark set of images assuming that the image is at a resolution such that the motion is not too large. In future work on this

Figure 3.1: Simple Test Image #1

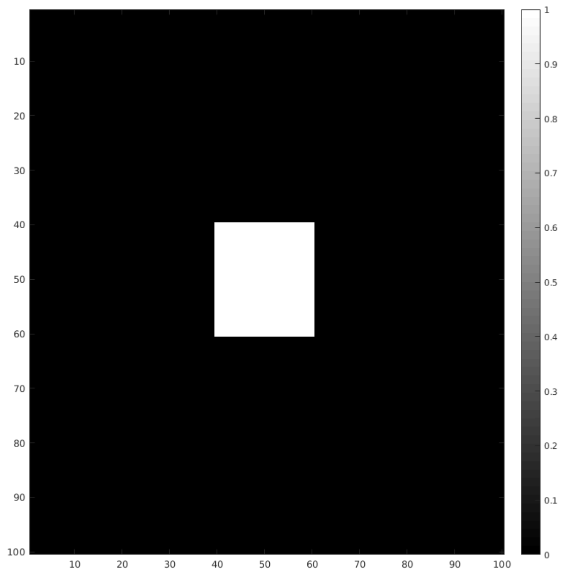


Figure 3.2: Simple Test Image #2

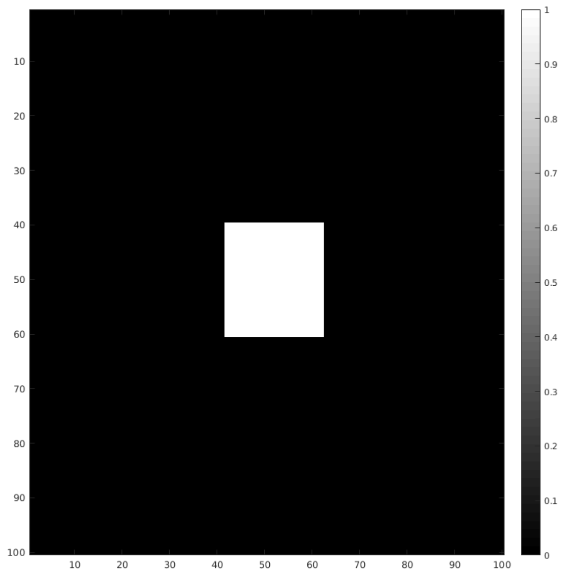


Figure 3.3: Optical Flow (High Resolution)

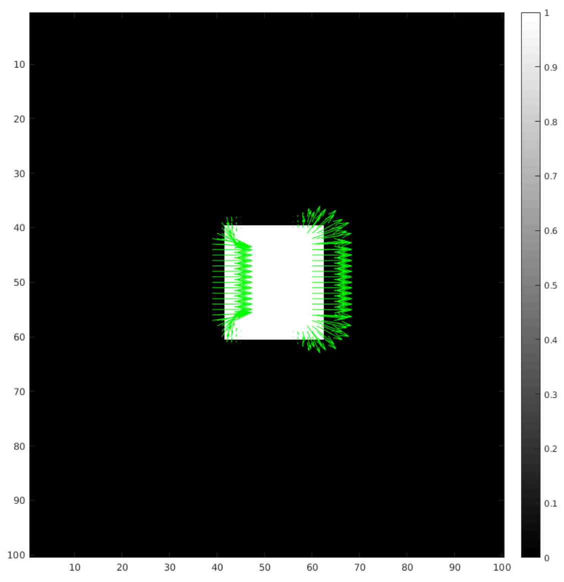


Figure 3.4: Optical Flow (Low Resolution)

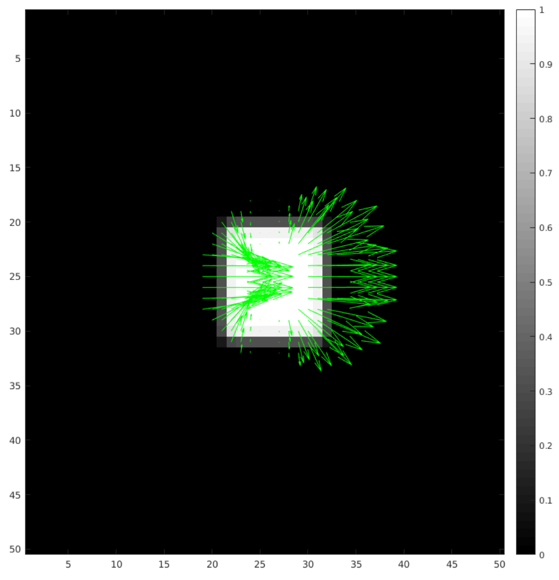


Figure 3.5: Optical Flow (High Resolution, Large Neighborhood)

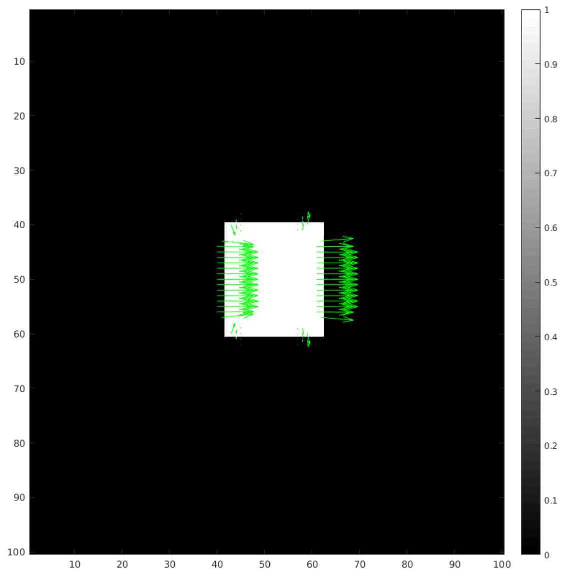


Figure 3.6: Head Image #1

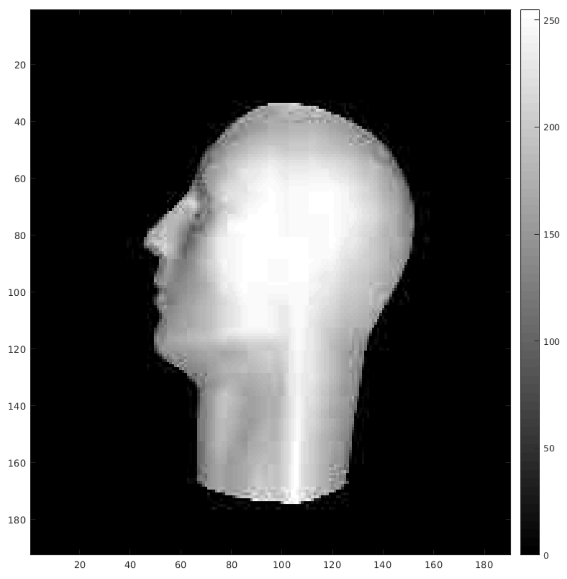


Figure 3.7: Head Image #2

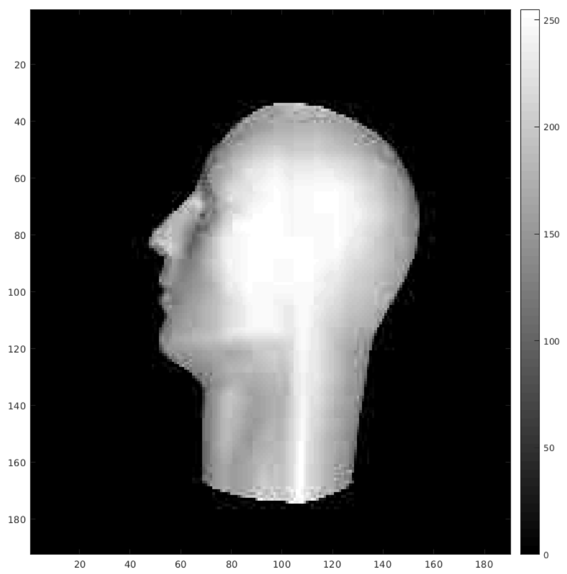


Figure 3.8: Head Optical Flow (High Resolution)

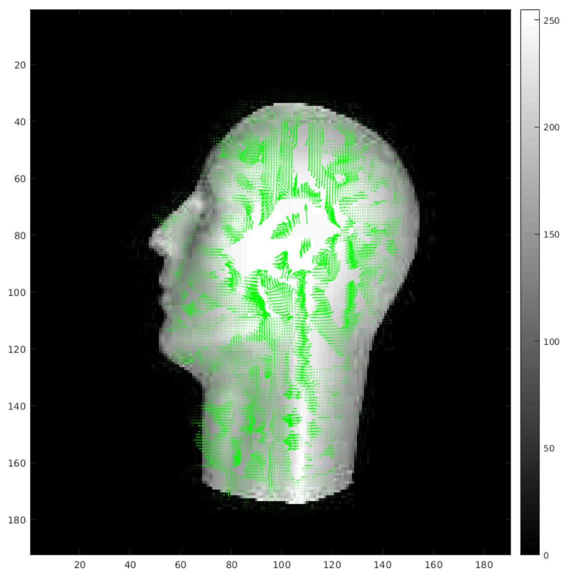
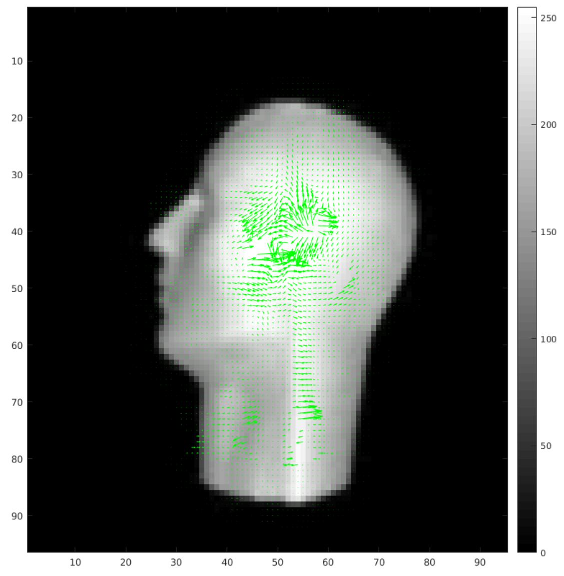


Figure 3.9: Head Optical Flow (Low Resolution)



problem, we could impelment the iterative version of the Lucas-Kanade algorithm to integrate measurement from multiple scales into a single estimate.

## APPENDIX A OPTICAL FLOW CODE

**Main Testing Script**

```
clc; clear; close all;
%% parameters
w = gaussian2d(11, 11, 2*eye(2));
tau = 1e-2;
base_path = '/home/jhelderman/Documents/school/computer-vision/projects/project-3/data/'
p1 = [base_path, 'LKTestpgm/LKTest3im1.pgm'];
p2 = [base_path, 'LKTestpgm/LKTest3im2.pgm'];
medfilt_sz = [3, 3];

%% load images
img1 = imread(p1);
img2 = imread(p2);
I = zeros([size(img1), 2]);
I(:,:,1) = img1;
I(:,:,2) = img2;
```

```
I = double(I);
Iorig = I;
I2orig = impyramid(I, 'reduce');

%% smooth images
h = gaussian2d(11, 11, 1 * eye(2));
I = imfilter(I, h, 'same');
% smooth in the time direction
I = imfilter(I, permute(gaussian1d(3, 1), [1, 3, 2]), 'same');

%% compute the optical flow
I1 = I;
I2 = impyramid(I1, 'reduce');
kern_sz = [3, 3];
[u2, v2] = lucas_kanade(I2, w, tau);
[u1, v1] = lucas_kanade(I1, w, tau);

%% plot the results
figure; imdisp(Iorig(:,:,1));
figure; imdisp(Iorig(:,:,2));
figure;
subplot(2, 2, 1);
imdisp(I1(:,:,1));
subplot(2, 2, 2);
imdisp(I1(:,:,2));
subplot(2, 2, 3);
imdisp(I2(:,:,1));
subplot(2, 2, 4);
imdisp(I2(:,:,2));
figure;
imdisp(Iorig(:,:,1));
hold on;
quiver(u1(:,:,1), v1(:,:,1), 5, 'g');
figure;
imdisp(Iorig(:,:,2));
hold on;
quiver(u1(:,:,2), v1(:,:,2), 5, 'g');
figure;
imdisp(I2orig(:,:,1));
hold on;
quiver(u2(:,:,1), v2(:,:,1), 5, 'g');
figure;
imdisp(I2orig(:,:,2));
hold on;
```

```
quiver(u2(:,:,2), v2(:,:,2), 5, 'g');
```

**Lucas-Kanade Algorithm**

```
function [u, v] = lucas_kanade(I, w, tau)
if nargin < 3
    tau = 1e-4;
end
N = 5;
I = imfilter(I, box2d(N, N));
%% compute spatial gradient
hx = -[-1, 8, 0, -8, 1] / 12;
[Ix, Iy] = imgrad(I, hx);

%% compute temporal gradient
It = temporal_grad(I);

%% compute gradient products
Ix2 = Ix.^2;
Iy2 = Iy.^2;
Ixy = Ix .* Iy;
Ixt = Ix .* It;
Iyt = Iy .* It;

%% apply neighborhood
Ix2 = imfilter(Ix2, w, 'same');
Iy2 = imfilter(Iy2, w, 'same');
Ixy = imfilter(Ixy, w, 'same');
Ixt = imfilter(Ixt, w, 'same');
Iyt = imfilter(Iyt, w, 'same');

%% solve system of equations
u = zeros(size(I));
v = zeros(size(I));
e1 = zeros(size(I));
e2 = zeros(size(I));
for m = 1:size(I, 1)
    for n = 1:size(I,2)
        for p = 1:size(I,3)
            A = [Ix2(m,n,p), Ixy(m,n,p); Ixy(m,n,p), Iy2(m,n,p)];
            b = [-Ixt(m,n); -Iyt(m,n)];
            % compute the eigenvalues
            e = eig(A);
            e1(m,n,p) = e(1);
            e2(m,n,p) = e(2);
```

```
                % check to see whether we're on a corner or edge
                if all(e > tau)
                    % matrix is non-singular. just solve the system of
                    % equations
                    x = A^-1 * b;
                    u(m,n,p) = x(2);
                    v(m,n,p) = x(1);
                elseif any(e > tau)
                    % matrix is singular but we are still on an edge. estimate
                    % the optical flow as the normal velocity vector
                    grad = [Ix(m,n,p), Iy(m,n,p)];
                    grad_norm = norm(grad);
                    s = -It(m, n, p) / grad_norm;
                    if grad_norm < 1e-3
                        norm_velocity = [0, 0];
                    else
                        norm_velocity = s * grad / grad_norm;
                    end
                    u(m,n,p) = norm_velocity(1);
                    v(m,n,p) = norm_velocity(2);
                end
            end
        end
    end
end
```

**Image Gradient Function**

```
function [Ix, Iy] = imgrad(I, hx, hy)
%% filters
if nargin == 1
    hx = [-1, 0, 1;...
          -2, 0, 2;...
          -1, 0, 1];
end
if nargin < 3
    hy = hx.';
end

%% filter the image
Ix = imfilter(I, hx);
Iy = imfilter(I, hy);
end
```

**Temporal Gradient Function**

```
function [It] = temporal_grad(I)
%% filter impulse response
h = [-1, 1];
h = permute(h, [1, 3, 2]);

%% filter the video
It = imfilter(I, h, 'full');
It = It(:,:,1:size(I,3)); % dependent on filter length
end
```

**2D Gaussian Filter Function**

```
function h = gaussian2d(M, N, cov)
%% get the spatial coordinates
x1 = (0.5:M-0.5) - M/2;
x2 = (0.5:N-0.5) - N/2;
[X1, X2] = ndgrid(x1, x2);
X = [reshape(X1, [], 1), reshape(X2, [], 1)];

%% evaluate the Gaussian
h = exp(-0.5 * dot(X * cov^-1, X, 2)) / sqrt(det(2 * pi * cov));
h = reshape(h, M, N);

%% make sum to one
h = h / sum(h(:));
end
```

**1D Gaussian Filter Function**

```
function h = gaussian1d(N, stdev)
%gaussian1d(N,stdev) generate a zero-mean, gaussian filter with the given
%parameters
%
% N: the number of points on which to evaluate the gaussian
% stdev: the standard deviation of the gaussian
%% get the time vector
t = (0.5:N-0.5) - N/2;

%% evaluate the gaussian
h = exp(-0.5 * t.^2 / stdev^2) / sqrt(2 * pi) / stdev;

%% make sum to one
h = h / sum(h);
end
```

## References

[1] J. L. Barron, D. J. Fleet, and S. S. Beauchemin. "Performance of Optical Flow Techniques". In: *Int. J. Comput. Vision* 12.1 (Feb. 1994), pp. 43–77. ISSN: 0920-5691. DOI: 10.1007/BF01420984. URL: http://dx.doi.org/10.1007/BF01420984.