3D Vision
BLG 634E

Professor: Gozde UNAL

Motion and Feature Tracking

Some slides are due Prof. Greg Slabaugh @ Queen's Mary Univ

Recap until now

3D Vision

- Camera calibration
- Homography Estimation
- Stereopsis
  - Matching to establish correspondences
  - Estimate Fundamental matrix
  - Stereo rectification
  - Compute disparity
  - RANSAC
- Feature Extraction and Matching

Overview of today's lecture

Video

Motion estimation

- Motion detection
- Optical flow: Horn-Schunck and Lucas Kanade

Feature tracking using the KLT tracker

Which features to track: min eigenvalue detector /Corner Detection idea

Video

Video is simply a series of images over time.

Cristiano Ronaldo Free Kick tutorial:

https://www.youtube.com/watch?v=dwDds_zdtXI

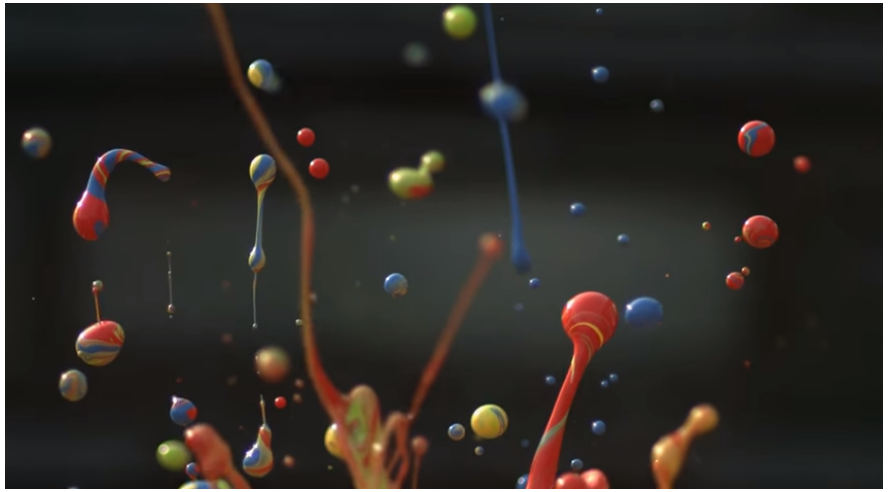
Video as a multi-dimensional signal

We can describe an video as a signal I(x, y, t) with two spatial coordinates x, y and a temporal coordinate, t

Frame rate

One of the defining characteristics of video is the framerate, usually measured in frames per second (FPS), or Hertz (Hz). Common framerates:

- 24 Hz: Traditional film
- 25 Hz: PAL format
- 50/60 Hz: Used in high end TVs. Most modern cameras can record video at this rate.
- Higher framerates possible (depending on hardware):

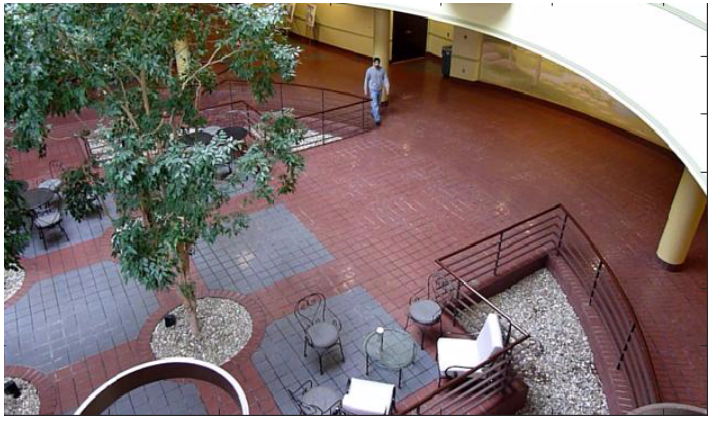


The Slow Mo Guys: paint on a speaker: (2:43) https://www.youtube.com/watch?v=5WKU7gG_ApU

vision.videoFileReader

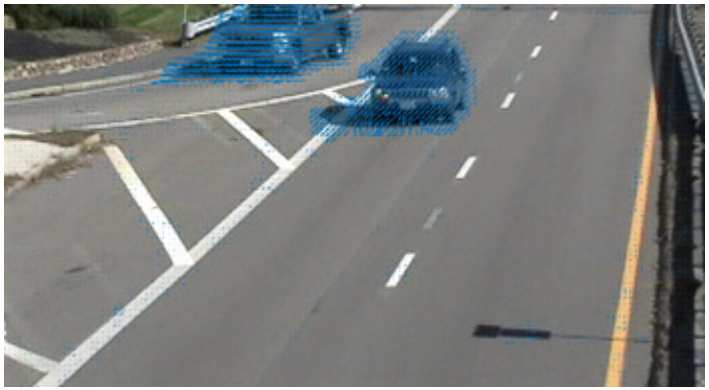
Matlab has a vision.videoFileReader that is quite similar to VideoFileReader, but is part of the Computer Vision System toolbox. There is also a vision.videoFileWriter.

```
videoReader = vision.VideoFileReader('atrium.avi');
I = step(videoReader);
% Show first frame
imshow(I);

% Loop over other frames
while ~isDone(videoReader)
     I = step(videoReader);
     image(I);  % image is faster than imshow
     pause(1/videoReader.info.VideoFrameRate);
end
```

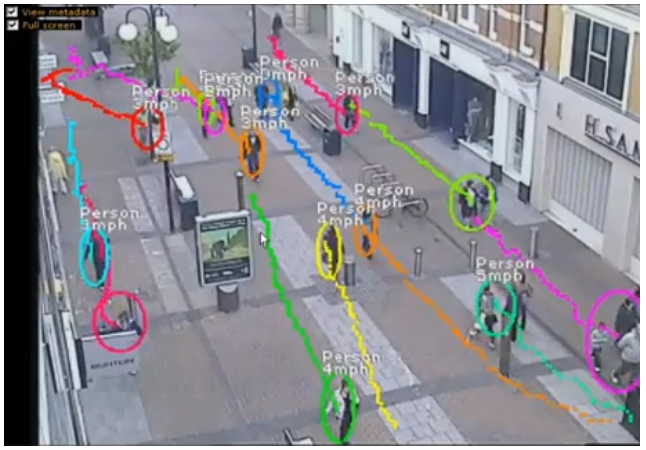


Video (Image Sequence) analysis

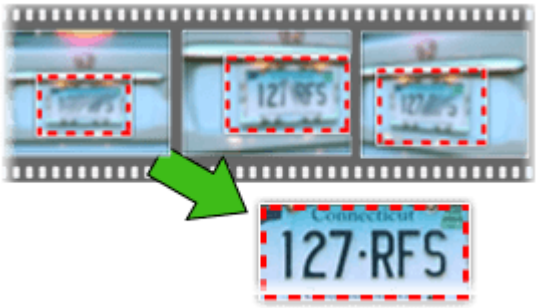
One can apply computer vision techniques to video to solve a variety of problems.



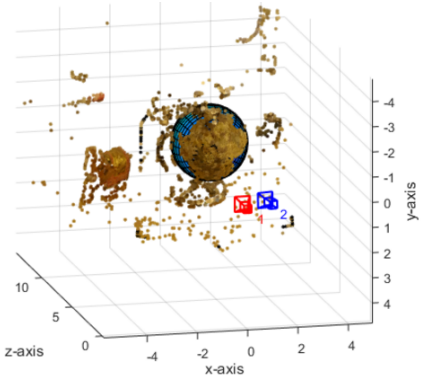
Motion estimation



Tracking



Super-resolution



Structure from motion

Augmenting a video

Place synthetic objects into a video stream by processing *each video frame*.

Background subtraction

- A classic computer vision problem is to detect change in a video. A simple way to achieve this is to use a background image. One can compute the difference between the image and background, and threshold the result.
- This identifies pixels whose colour has changed.

Image (I)

Background (B)

Thresholded difference (M)

$$F(x,y,t) = I(x,y,t) - B(x,y)$$

$$M(x,y,t) = \begin{cases} 1, & |F(x,y,t)| > T \\ 0, & \text{otherwise} \end{cases}$$

http://wordpress-jodoin.dmi.usherb.ca/dataset2014/

In code

```
I = double(imread('frame.jpg'));
B = double(imread('background.jpg'));
figure; imshow(I/255);
figure; imshow(B/255);

% Compute the absolute difference image
F = abs(I-B);

% Find maximum change in each colour
channel
J = max(max(F(:,:,1),
F(:,:,2)),F(:,:,3));

% Threshold
M = J > 40;
figure; imshow(M);
```

⇨ What are some limitations to this approach?

- Dynamic background, camera motion, shadows, changes in scene (weather, illumination), clutter

In code

```
I = double(imread('frame.jpg'));
B = double(imread('background.jpg'));
figure; imshow(I/255);
figure; imshow(B/255);

% Compute the absolute difference image
F = abs(I-B);

% Find maximum change in each colour
channel
J = max(max(F(:,:,1),
F(:,:,2)),F(:,:,3));

% Threshold
M = J > 40;
figure; imshow(M);

% Post processing: keep the largest blob
and fill holes
L = bwlabel(M);
stat =
regionprops(L,'Area','PixelIdxList');
[maxValue,index] = max([stat.Area]);
G = zeros(size(M));
G(stat(index).PixelIdxList) = 1;
H = imfill(G, 'holes');
```

More complex models

- Improved background modeling
  - One can form the background using an average (or median) of the last *N* frames. This will allow the background to adapt, e.g., based on time of day.
  - Although individual frames may have foreground objects, if they move quickly enough, their effect on the background image will be small.

$$B(x,y,t) = \frac{1}{N}\sum_{i-1}^{N} I(x,y,t-i)$$

- Per-pixel Gaussian fitting
  - For *each pixel*, one can determine the mean $\mu(x,y,t)$ and standard deviation $\sigma(x,y,t)$ of the intensity (or colour) based on the last *N* frames.
  - A pixel can then be classified as foreground if its value lies outside some confidence interval of the mean.
  - Note: there are fast ways to incrementally update the mean and standard deviation with each new frame.

$$M(x,y,t) = \begin{cases} 1, & \frac{|I(x,y,t)-\mu(x,y,t)|}{\sigma(x,y,t)} > k \\ 0, & \text{otherwise} \end{cases}$$

vision.ForegroundDetector

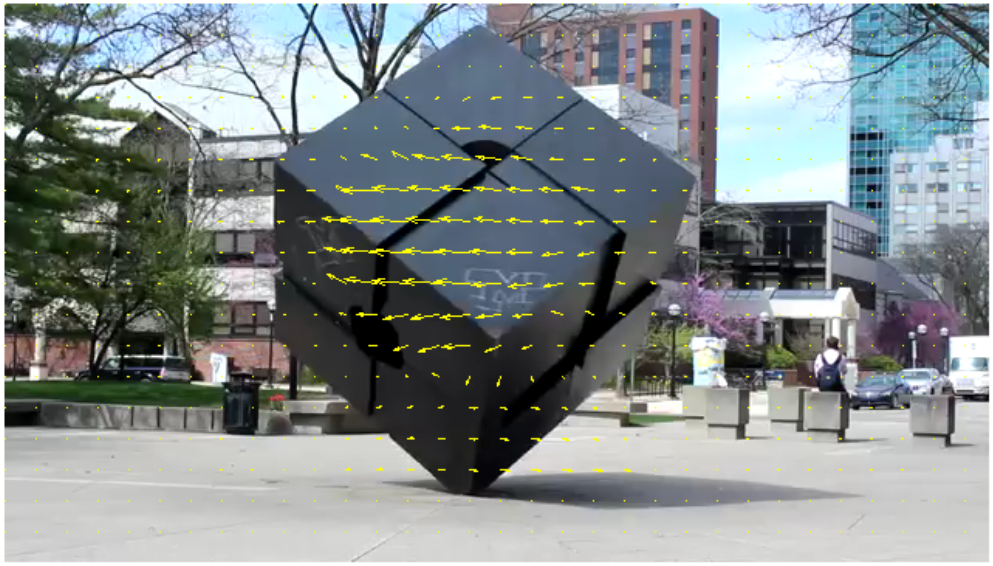
- Matlab comes with vision.ForegroundDetector, which is a system object that detects foreground pixels from a stationary camera, using Gaussian Mixture Models (GMMs).
- This combines K (default 5) Gaussians to model the intensity at a pixel through time. A pixel is compared to the Gaussians, and the best matching Gaussian adapts based on a learning rate.

```
videoSource = vision.VideoFileReader('viptraffic.avi');
detector = vision.ForegroundDetector('NumTrainingFrames', 5);
blob = vision.BlobAnalysis('MinimumBlobArea', 250);
shapeInserter = vision.ShapeInserter('BorderColor', 'White');

figure;
while ~isDone(videoSource)
   frame  = step(videoSource);
   fgMask = step(detector, frame);
   [area, centroid, bbox]   = step(blob, fgMask);
   out    = step(shapeInserter, frame, bbox);
   imshow(out);
end
```

http://www.ai.mit.edu/projects/vsam/Publications/stauffer_cvpr98_track.pdf

Optical flow

- Change detection only says *which* pixels in image have differences due to motion. It doesn't provide any insight on the *direction* objects are moving.
- Optical flow is the motion observed in a video resulting from the relative motion between the camera and the scene.
- It is typically represented as a *vector field*, which is a collection of vectors showing which way pixels are moving in the image.



www.mathworks.com/examples/matlab/community/20141-lucas-kanade-method-example-1

2D velocity field – Motion Field

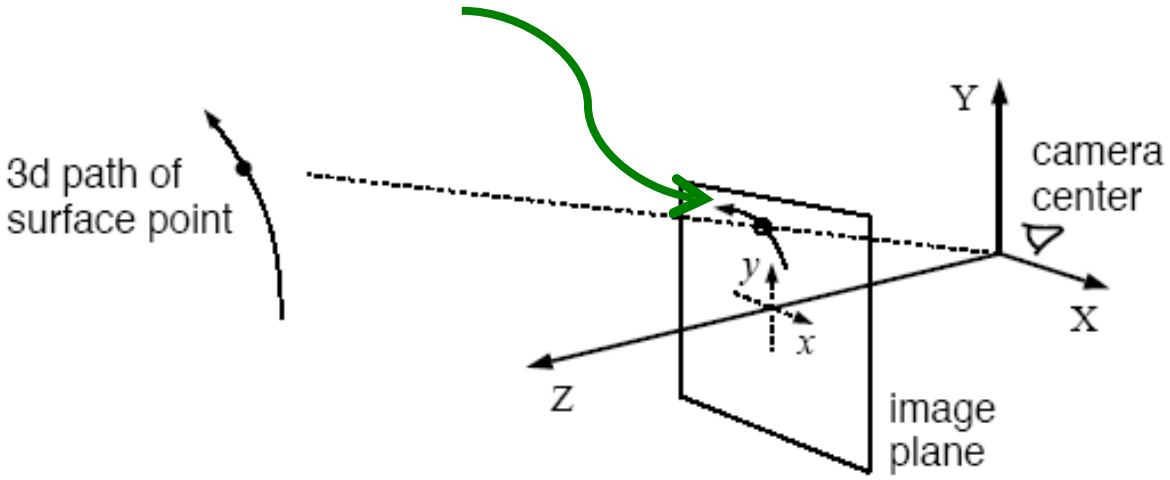


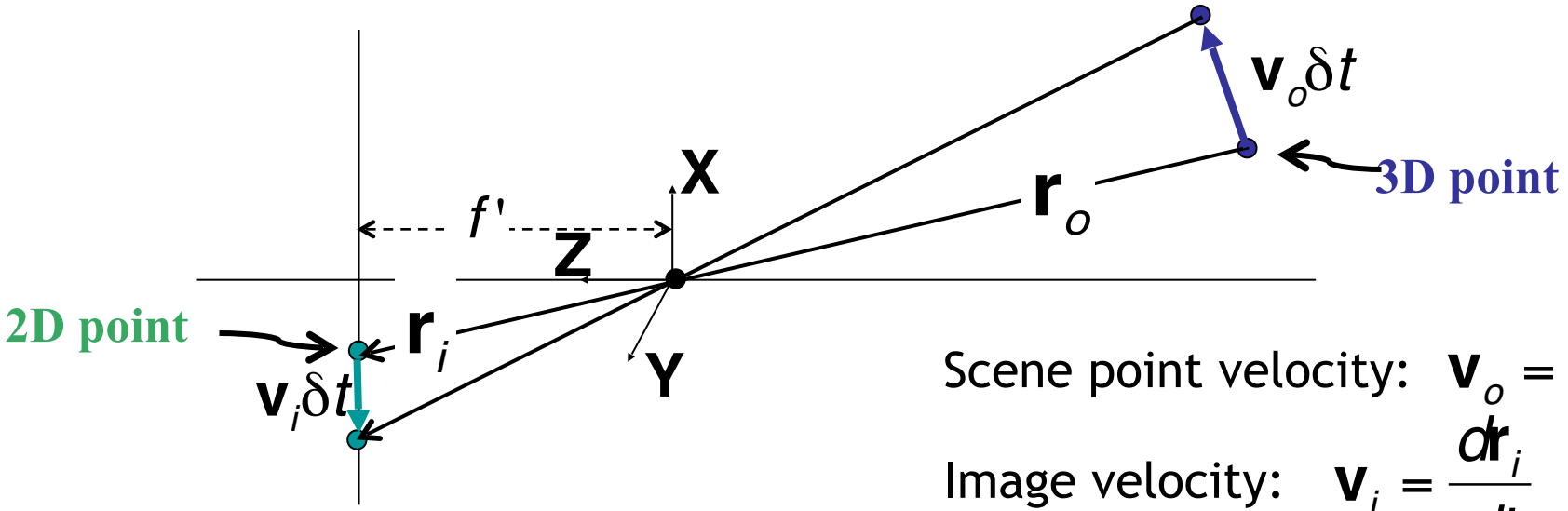
Figure 1: Camera centered coordinate frame and perspective projection. Owing to motion between the camera and the scene, a 3D surface point traverses a path in 3D. Under perspective projection, this path projects onto a 2D path in the image plane, the temporal derivative of which is called 2D velocity. The 2d velocities associated with all visible points defines a dense 2d vector field called the 2d motion field.

(Heeger, 1998)

Def: 2D velocity field or the optical flow field approximates the true motion field: the [2D] projection into the image [plane] of [the sequence's] 3D motion vectors"
"It is a purely geometrical concept" – Horn and Schunk 1993

Motion Field

- Image velocity of a point moving in the scene



Scene point velocity: $\mathbf{v}_o = \frac{d\mathbf{r}_o}{dt}$

Image velocity: $\mathbf{v}_i = \frac{d\mathbf{r}_i}{dt}$

Perspective projection: $\frac{1}{f'}\mathbf{r}_i = \frac{\mathbf{r}_o}{\mathbf{r}_o \cdot \mathbf{Z}}$

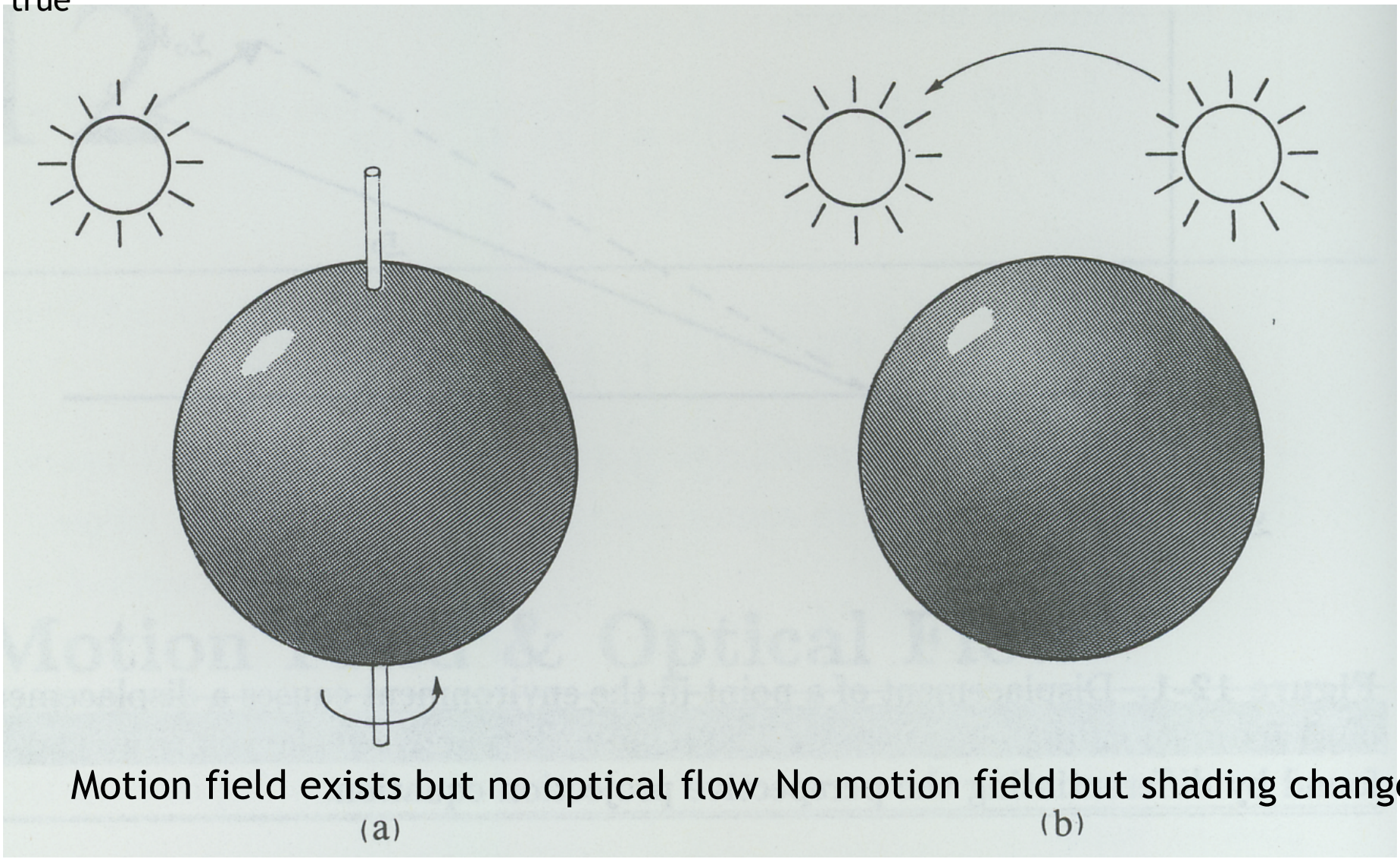
Motion field

$$\mathbf{v}_i = \frac{d\mathbf{r}_i}{dt} = f' \frac{(\mathbf{r}_o \cdot \mathbf{Z})\mathbf{v}_o - (\mathbf{v}_o \cdot \mathbf{Z})\mathbf{r}_o}{(\mathbf{r}_o \cdot \mathbf{Z})^2} = f' \frac{(\mathbf{r}_o \times \mathbf{v}_o) \times \mathbf{Z}}{(\mathbf{r}_o \cdot \mathbf{Z})^2}$$

(axb)xc= -(b . c) a + (a . c) b

Optical Flow ≠ Motion Field

Ideally Motion field = Optical Flow field (apparent motion) but this is not true

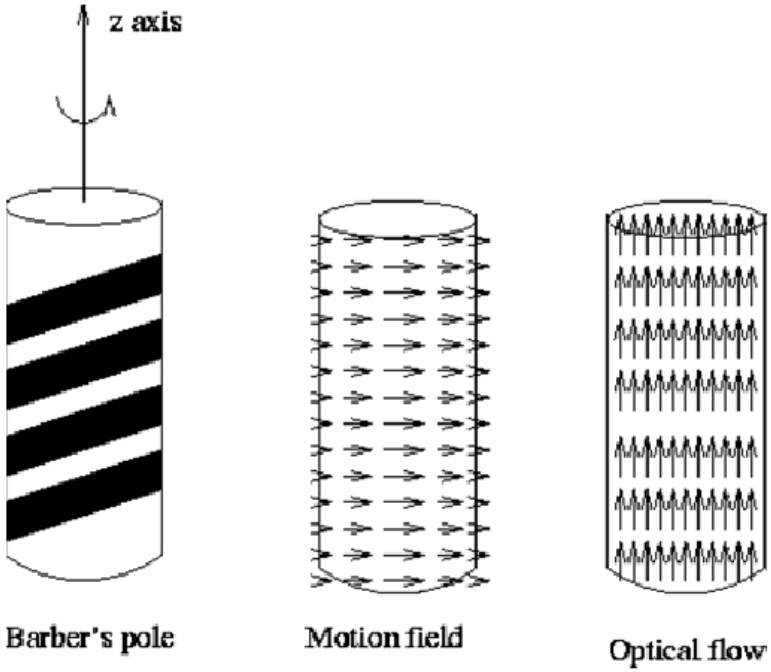


Motion field exists but no optical flow

No motion field but shading changes

Motion Field and Optical Flow

Barber pole illusion



This is due to the aperture problem (we'll see soon), the motion appears to be going *up* rather than rotating *around* the pole.

https://en.wikipedia.org/wiki/Barberpole_illusion

2D Apparent Motion = Optical Flow

- Our goal: to determine the motion (u, v) of a pixel at (x, y)
- Often an assumption that brightness is constant is made. Under this assumption, a pixel at (x, y) moves to a location (x+u, y+v) at the next frame, where (u, v) is a spatial displacement. The colour (brightness) doesn't change as a result of the motion.
- Also, there is a small motion assumption, i.e. the points do not move very far

frame t

frame t+1

- Under the Brightness Constancy assumption, we can write

$$I(x+u\delta t, y+v\delta t, t+\delta t) = I(x, y, t)$$

Taylor series expansion

- If we assume the motion is small, we can use a Taylor series expansion on the left hand side to get

$$I(x, y, t) + \delta x \frac{\partial I}{\partial x} + \delta y \frac{\partial I}{\partial y} + \delta t \frac{\partial I}{\partial t} = I(x, y, t)$$

Image derivative in x Image derivative in y Image derivative in time

dividing by delta t, the above equation can be rewritten as

$$u = \frac{\delta x}{\delta t} \qquad v = \frac{\delta y}{\delta t} \Longrightarrow \qquad u \frac{\partial I}{\partial x} + v \frac{\partial I}{\partial y} + \frac{\partial I}{\partial t} = 0$$

u: Optical flow vector x component

v: Optical flow vector y component

Image Gradient Vector: $\nabla I = \begin{bmatrix} I_x \\ I_y \end{bmatrix} = \begin{bmatrix} \frac{\partial I}{\partial x} \\ \frac{\partial I}{\partial y} \end{bmatrix}$

Optical flow constraint equation

- Hence, using the brightness constancy assumption, we obtain:

$$0 = \frac{\partial I}{\partial x}u + \frac{\partial I}{\partial y}v + \frac{\partial I}{\partial t}$$

which can be rewritten as

$$\nabla I \cdot [u, v]^T = -\frac{\partial I}{\partial t}$$

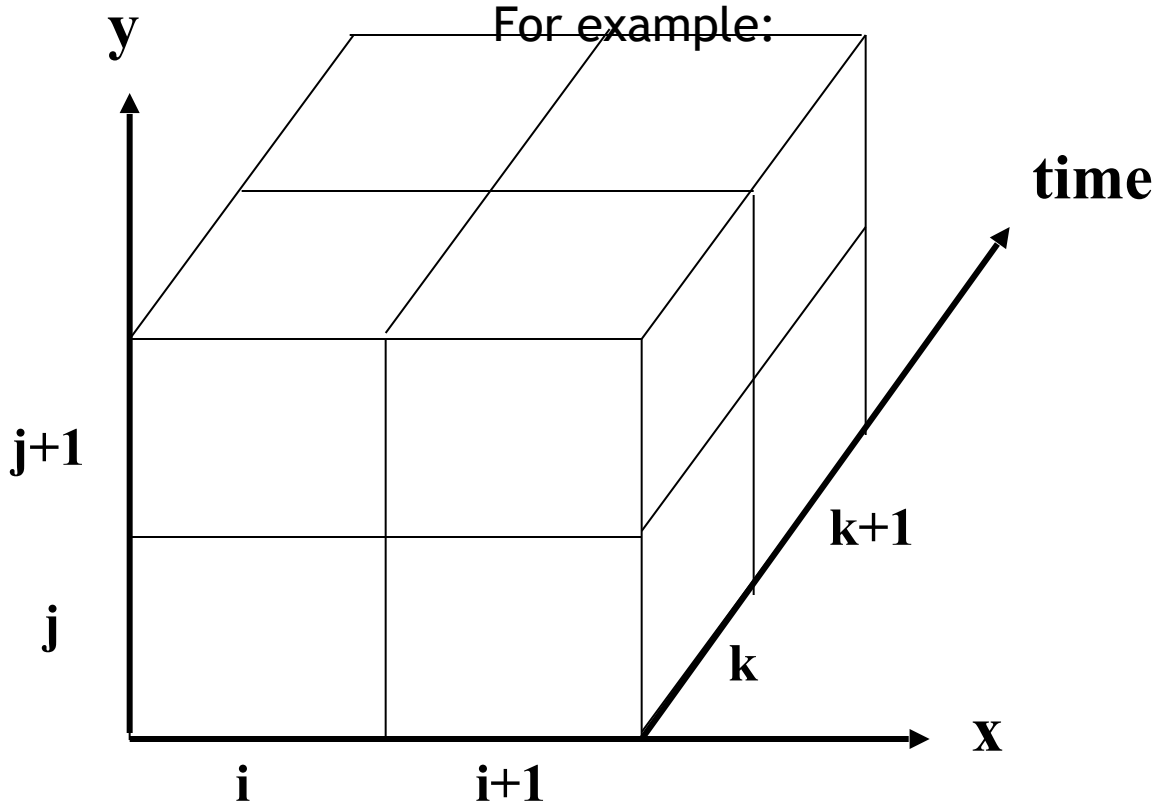
- This equation (or the one above) is known as the **optical flow constraint equation**.

⇨ This equation gives a relationship between the change in brightness (or colour) in the image and the motion (u, v) of a pixel. But is this equation sufficient to recover the image motion?

- No, it provides *one* equation for *two* unknowns (u, v).

Finding Gradients in X-Y-t

We can compute spatial and temporal derivatives
For example:



$$I_x = \frac{1}{4\delta x}[(I_{i+1,j,k} + I_{i+1,j,k+1} + I_{i+1,j+1,k} + I_{i+1,j+1,k+1})$$

$$-(I_{i,j,k} + I_{i,j,k+1} + I_{i,j+1,k} + I_{i,j+1,k+1})]$$

Optical Flow Constraint

- Intuitively, what does this constraint mean?

Q: How can you project a vector onto a given direction?

$$\left(V \bullet \frac{\nabla I}{|\nabla I|}\right)\frac{\nabla I}{|\nabla I|} = \left(\frac{\begin{bmatrix} u \\ v \end{bmatrix} \bullet \begin{bmatrix} I_x \\ I_y \end{bmatrix}}{|\nabla I|}\right)\frac{\nabla I}{|\nabla I|} = \frac{(u\, I_x + v\, I_y)}{|\nabla I|}\frac{\nabla I}{|\nabla I|}$$

$$\left(V \bullet \frac{\nabla I}{|\nabla I|}\right)\frac{\nabla I}{|\nabla I|} = \left(\frac{-I_t}{|\nabla I|}\right)\frac{\nabla I}{|\nabla I|}$$

- The component of the flow in the gradient direction is determined
- The component of the flow parallel to an edge is unknown

The aperture problem

→ The component of the motion parallel to the gradient (edge) cannot be measured this way.

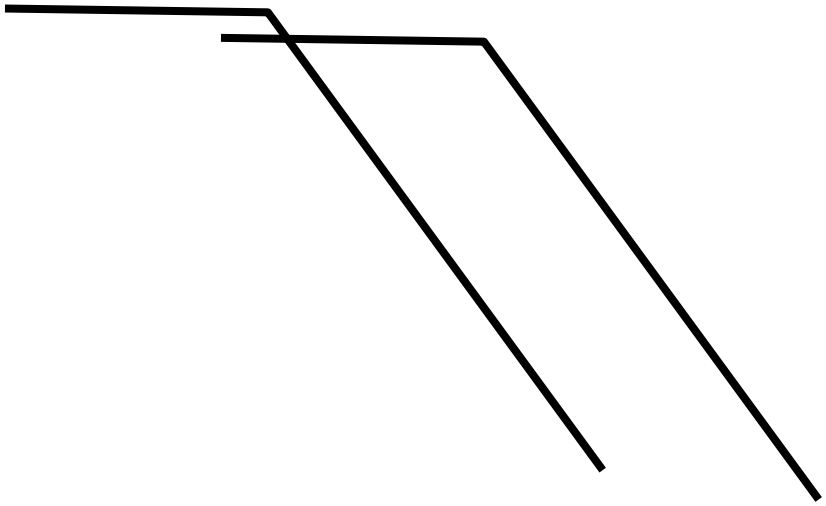
This is called the Aperture Problem:

Only the component of the flow in the image gradient (normal) direction is determined:

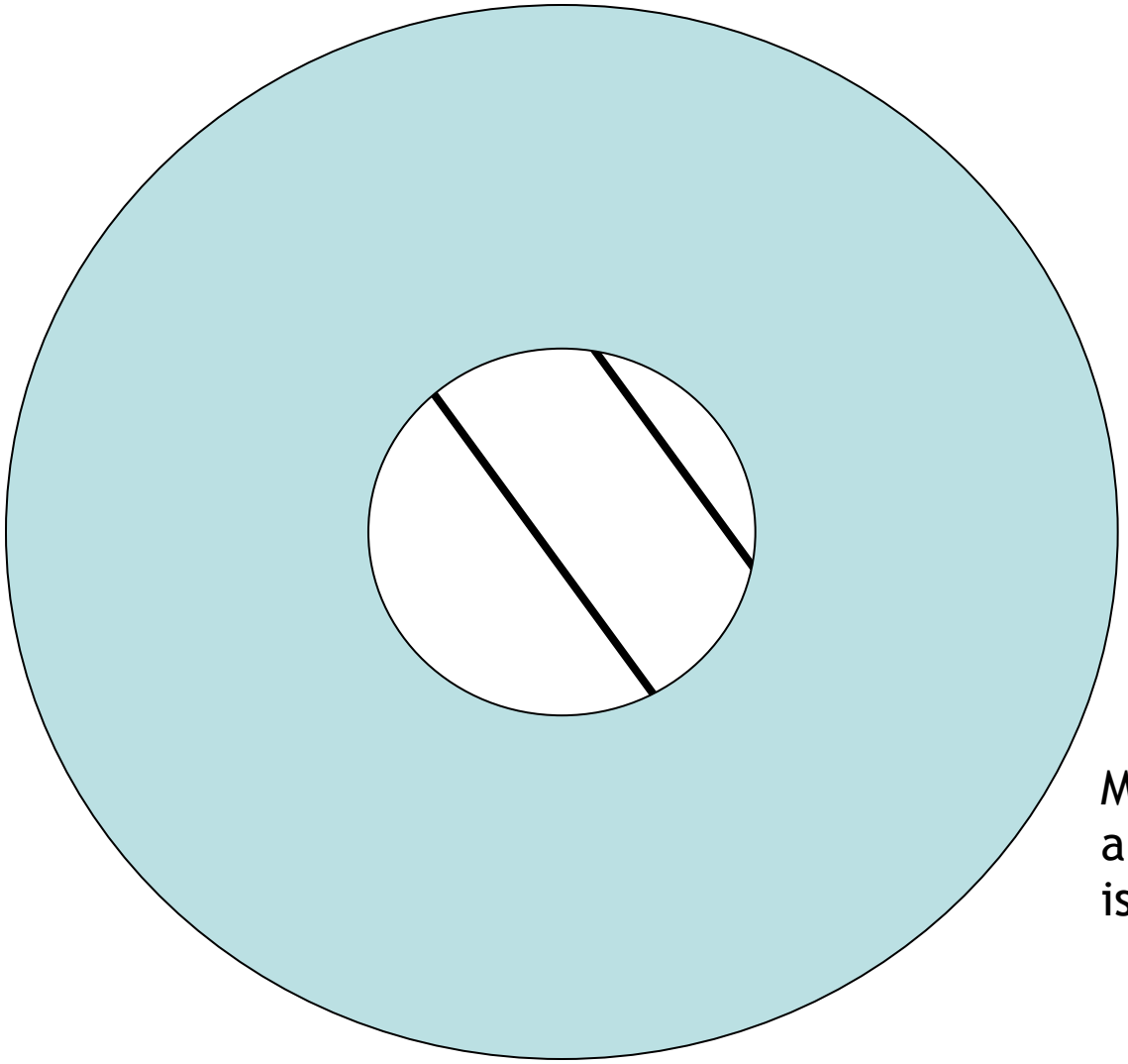
Normal optical flow:

$$\mathbf{u}_n \doteq \frac{\nabla I^T \mathbf{u}}{\|\nabla I\|} \cdot \frac{\nabla I}{\|\nabla I\|} = -\frac{I_t}{\|\nabla I\|} \cdot \frac{\nabla I}{\|\nabla I\|}$$

The aperture problem

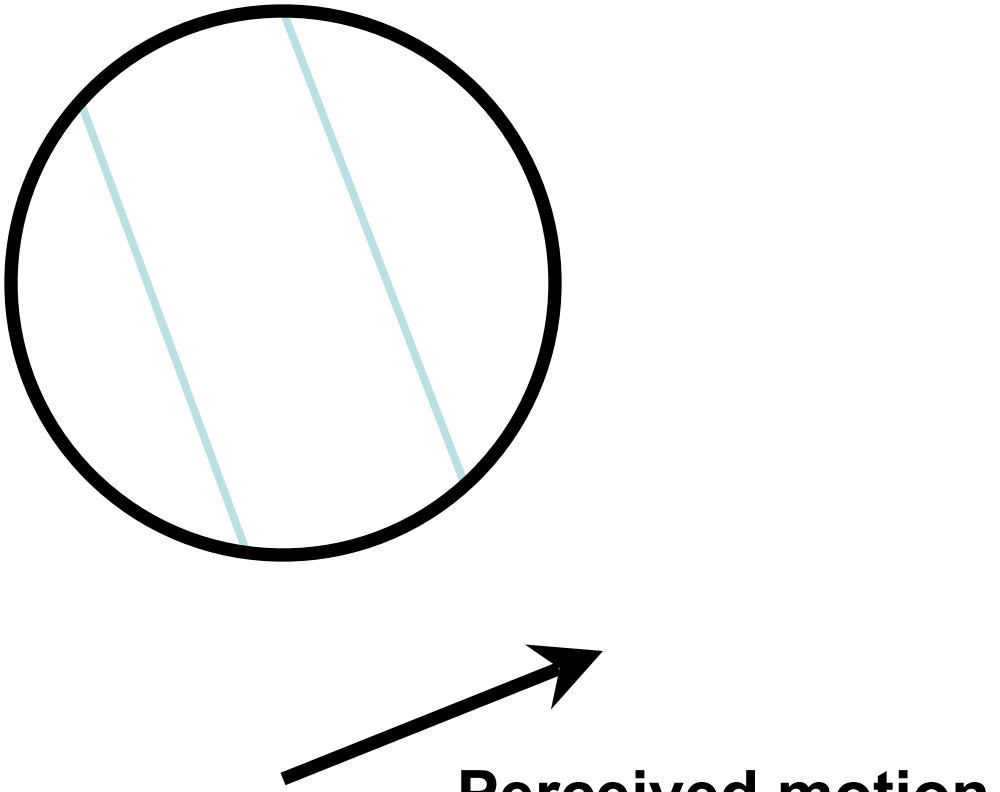


The aperture problem



Motion viewed along just an edge is ambigous

The aperture problem



Horn-Schunck method

- To resolve this ambiguity and solve for the optical flow, we require an additional constraint.
- The Horn-Schunck model additionally assumes that the optical flow field is *smooth*, so that a motion vector (u, v) at a pixel (x, y) is similar to that of its neighbouring pixels.
- Specifically, it seeks to minimise the following energy (cost functional):

$$E = \iint \left(\frac{\partial I}{\partial x}u + \frac{\partial I}{\partial y}v + \frac{\partial I}{\partial t}\right)^2 dxdy + \alpha \iint \left\{\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2\right\} dxdy$$

Should be 0 from brightness constancy

Penalises changes in motion vectors

- One can derive the above energy functional giving the (u, v) that minimises the above equation using Euler-Lagrange equations. Note the solution is iterative.
- You are responsible in understanding the two terms above, how the cost functional is set up, and how Euler-Lagrange derivation is taken.

B.K.P. Horn and B.G. Schunck, "Determining optical flow." *Artificial Intelligence*, vol 17, pp 185–203, 1981.
https://en.wikipedia.org/wiki/Horn%E2%80%93Schunck_method

Computing Optical Flow

- Formulate Error in Optical Flow Constraint: (E: image function)

$$C_b^{\,2} = \iint_{image} (E_x u + E_y v + E_t)^2 \, dx \, dy$$

- We need additional constraints!

- Smoothness Constraint (Horn and Schunck 1981):

  Usually motion field varies smoothly in the image.
  So, penalize departure from smoothness:

$$C_c^{\,2} = \iint_{image} (u_x^2 + u_y^2) + (v_x^2 + v_y^2) \; dx \, dy$$

- Hence use gradient magnitudes of motion field components as a regularizer

Computing Optical Flow by Horn Schunck method

Find (u,v) at each image point that MINIMIZES the total error:

$$C = C_b^{\,2} + \alpha^2\, C_c^{\,2}$$

α^2 : weighting factor

$$C = \iint_{image} (E_x u + E_y v + E_t)^2 + \alpha^2 (|\nabla u|^2 + |\nabla v|^2)\; dx\, dy$$

Computing Optical Flow

* Minimize the total error C using calculus of variations:
Use Euler-Lagrange (E-L) equations to find the necessary condition for a minimizer of C

$$C = \iint_{image} (E_x u + E_y v + E_t)^2 + \alpha^2 (|\nabla u|^2 + |\nabla v|^2)\, dx\, dy$$

Generally, for an integrand with unknown u: $C(u) = \int_{\Omega} f(u, \nabla u, x)\, dx$

Necessary condition that minimizes C is given by: $\rightarrow$ $f_u - div(f_{\nabla u}) = 0$

f_u or f_nabla u indicates derivative of the integrand f wrt to those functions

With E-L conditions, it is possible to take a functional derivative to estimate an unknown function in a cost functional C

Horn-Schunck OF: Minimization

* Minimize the total error C using its Euler-Lagrange eqns:

$$E_x^2 u + E_x E_y v = \alpha^2 \nabla^2 u - E_x E_t,$$
$$E_x E_y u + E_y^2 v = \alpha^2 \nabla^2 v - E_y E_t.$$

Note: In the exam, you are responsible in understanding the derivations until this point, not below. However, it is educational for you to go over those to know about details of one of the pioneering motion estimation algorithms in computer vision.

Using the approximation of the Laplacian – Eq (*) introduced in Horn-Schunck paper (see next slide), the above equations are rewritten as:

$$(\alpha^2 + E_x^2) u + E_x E_y v = (\alpha^2 \bar{u} - E_x E_t),$$
$$E_x E_y u + (\alpha^2 + E_y^2) v = (\alpha^2 \bar{v} - E_y E_t).$$

Horn & Schunck's Optical Flow

Laplacians of u and v are defined as

$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \quad \text{and} \quad \nabla^2 v = \frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}.$$

Approximate the Laplacians of u and v as follows:

$$(*) \qquad \nabla^2 u \approx (\bar{u}_{i,j,k} - u_{i,j,k}) \quad \text{and} \quad \nabla^2 v \approx (\bar{v}_{i,j,k} - v_{i,j,k}),$$

where the local averages $\bar{u}$ and $\bar{v}$ are defined as follows

$$\bar{u}_{i,j,k} = \tfrac{1}{6}\{u_{i-1,j,k} + u_{i,j+1,k} + u_{i+1,j,k} + u_{i,j-1,k}\} + \tfrac{1}{12}\{u_{i-1,j-1,k} + u_{i-1,j+1,k} + u_{i+1,j+1,k} + u_{i+1,j-1,k}\},$$

$$\bar{v}_{i,j,k} = \tfrac{1}{6}\{v_{i-1,j,k} + v_{i,j+1,k} + v_{i+1,j,k} + v_{i,j-1,k}\} + \tfrac{1}{12}\{v_{i-1,j-1,k} + v_{i-1,j+1,k} + v_{i+1,j+1,k} + v_{i+1,j-1,k}\}$$

1/12	1/6	1/12
1/6	-1	1/6
1/12	1/6	1/12

FIG. 3. The Laplacian is estimated by subtracting the value at a point from a weighted average of the values at neighboring points.

Horn & Schunk Optical Flow Algorithm

The determinant of the coefficient matrix equals $\alpha^2(\alpha^2 + E_x^2 + E_y^2)$. Solving for u and v we find that

$$(\alpha^2 + E_x^2 + E_y^2)u = +(\alpha^2 + E_y^2)\bar{u} - E_x E_y \bar{v} - E_x E_t,$$
$$(\alpha^2 + E_x^2 + E_y^2)v = -E_x E_y \bar{u} + (\alpha^2 + E_x^2)\bar{v} - E_y E_t.$$

These equations can be written in the alternate form

$$(\alpha^2 + E_x^2 + E_y^2)(u - \bar{u}) = -E_x[E_x\bar{u} + E_y\bar{v} + E_t],$$
$$(\alpha^2 + E_x^2 + E_y^2)(v - \bar{v}) = -E_y[E_x\bar{u} + E_y\bar{v} + E_t].$$

Horn & Schunk OF: Iterative Solution

- We now have a pair of equations for each point in the image.

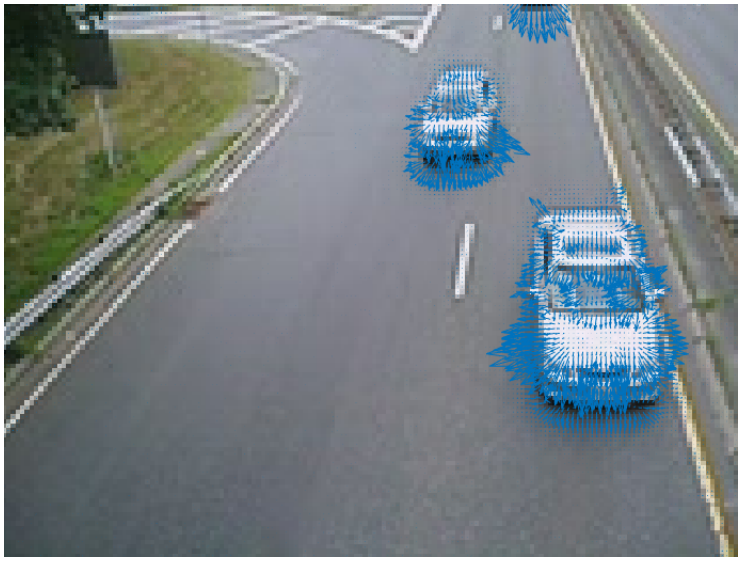
We can compute a new set of velocity estimates (u^{n+1}, v^{n+1}) from the estimated derivatives and the average of the previous velocity estimates (u^n, v^n) by:

$$u^{n+1} = \bar{u}^n - E_x[E_x\bar{u}^n + E_y\bar{v}^n + E_t]/(\alpha^2 + E_x^2 + E_y^2),$$
$$v^{n+1} = \bar{v}^n - E_y[E_x\bar{u}^n + E_y\bar{v}^n + E_t]/(\alpha^2 + E_x^2 + E_y^2).$$

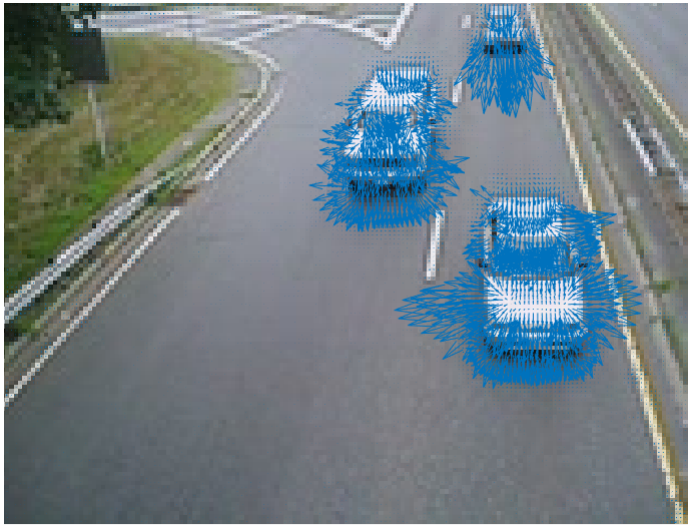
Also, Iterative methods, such as the Gauss-Seidel method can be used. Check the paper or elsewhere for details.

Horn-Schunck in Matlab

- In Matlab, you can implement optical flow using Horn-Schunck using the `estimateFlow` function and an `opticalFlowHS` object.
- Syntax(es)
  - `flow = estimateFlow(I, obj)` where `I` is a grayscale image and `obj` is an opticalFlow object.
  - `obj = opticalFlowHS(Name, Value)` where `Name/Value` are optional arguments, e.g., `'Smoothness'`, 0.5 or `'MaxIteration'`, 15

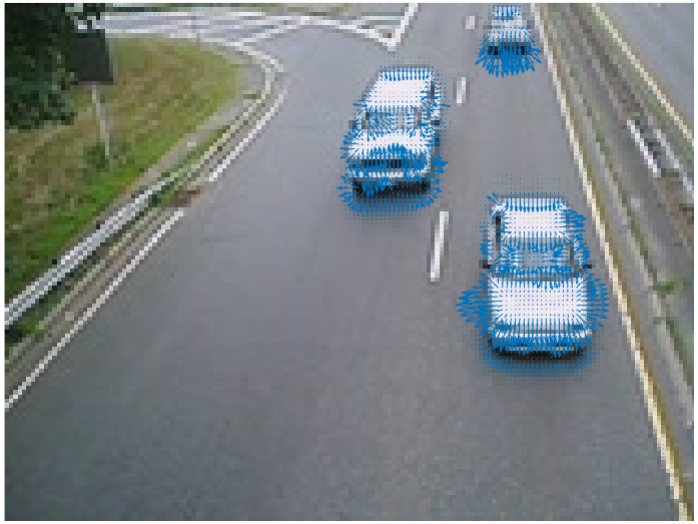


Varying parameters

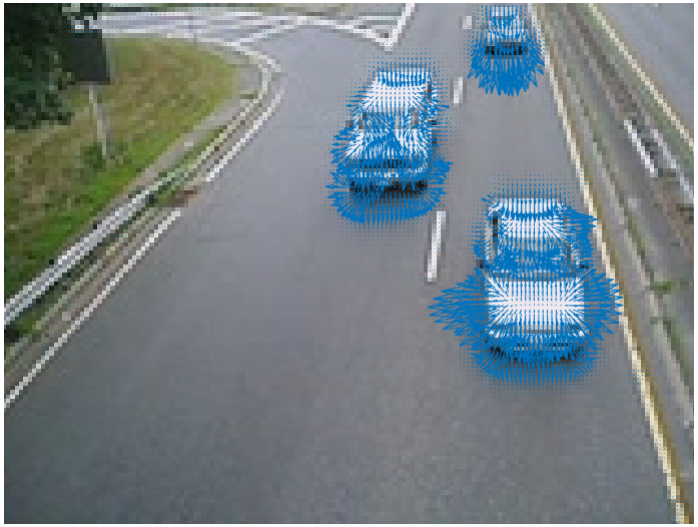


'Smoothness', 0.5

'Smoothness', 1.5



'MaxIteration', 5



'MaxIteration', 15

Lucas-Kanade

- The Lucas-Kanade method also assumes spatial coherence, but unlike Horn-Schunck, it assumes a pixel's neighbours have the *same* (u, v) for a particular t.
- If we use, say a 5x5 window, this gives us *N*=25 equations, namely the optical flow constraint equation for each pixel in the window.

$$\begin{aligned} I_x(x_1, y_1)u + I_y(x_1, y_1)v &= -I_t(x_1, y_1) \\ I_x(x_2, y_2)u + I_y(x_2, y_2)v &= -I_t(x_2, y_2) \\ &\vdots \quad\quad \vdots \\ I_x(x_N, y_N)u + I_y(x_N, y_N)v &= -I_t(x_N, y_N) \end{aligned}$$

- This results in an over-determined set of linear equations -- more equations (*N*) than unknowns (2).

Lucas-Kanade

- We can write this in matrix form as

$$\begin{bmatrix} I_x(x_1,y_1) & I_y(x_1,y_1) \\ I_x(x_2,y_2) & I_y(x_2,y_2) \\ \vdots & \vdots \\ I_x(x_N,y_N) & I_y(x_N,y_N) \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = - \begin{bmatrix} I_t(x_1,y_1) \\ I_t(x_2,y_2) \\ \vdots \\ I_t(x_N,y_N) \end{bmatrix}$$

$$\underset{\text{Nx2}}{A} \quad \underset{\text{2x1}}{d} \quad = \quad \underset{\text{Nx1}}{b}$$

- And solve using least squares as

$$(A^T A)d = A^T b$$

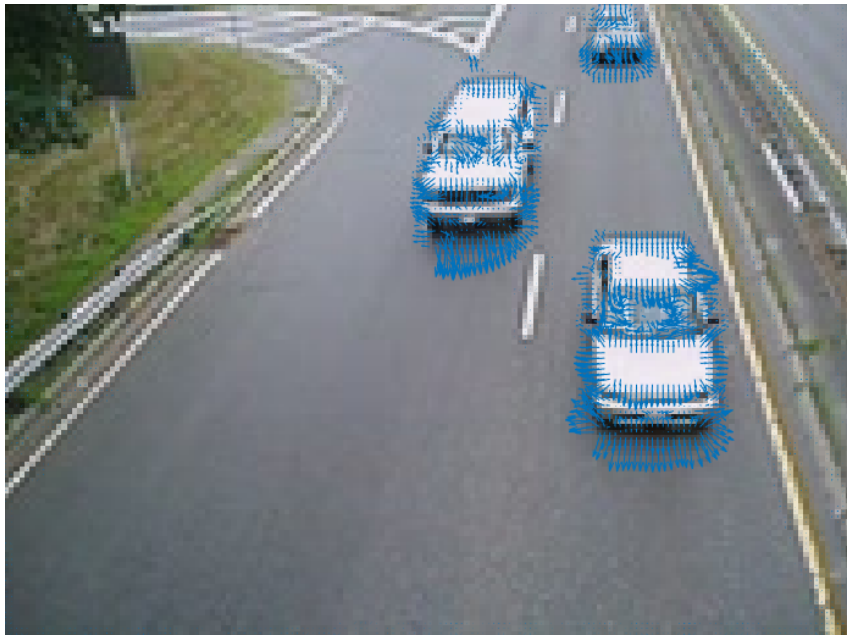
$$\begin{bmatrix} \sum I_x I_x & \sum I_x I_y \\ \sum I_x I_y & \sum I_y I_y \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = - \begin{bmatrix} \sum I_x I_t \\ \sum I_y I_t \end{bmatrix}$$

$$A^T A \qquad d \qquad A^T b$$

- The summations are over all the *N* pixels in the window.

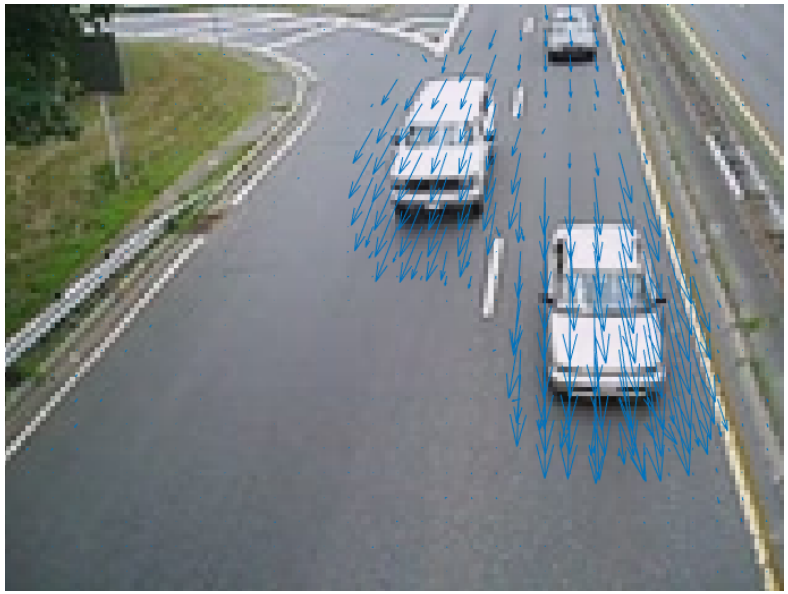
Lucas-Kanade in Matlab

- In Matlab, you can implement optical flow using Lucas-Kanade using the `estimateFlow` function and an `opticalFlowLK` object.
- Syntax
  - `obj = opticalFlowLK(Name, Value)` where `Name/Value` are optional argument for a noise threshold: `'NoiseThreshold'`, 0.005



Farneback's method

- Farneback's method locally models the image brightness as a 2D polynomial function, and computes the translation between frames. It uses a pyramidal (coarse to fine) approach and is iterative.
- Syntax
  - `obj = opticalFlowFarneback(Name, Value)` where `Name/Value` are optional arguments, for example: `'NumPyramidLevels'`, 2 or `'NumIterations'`, 5



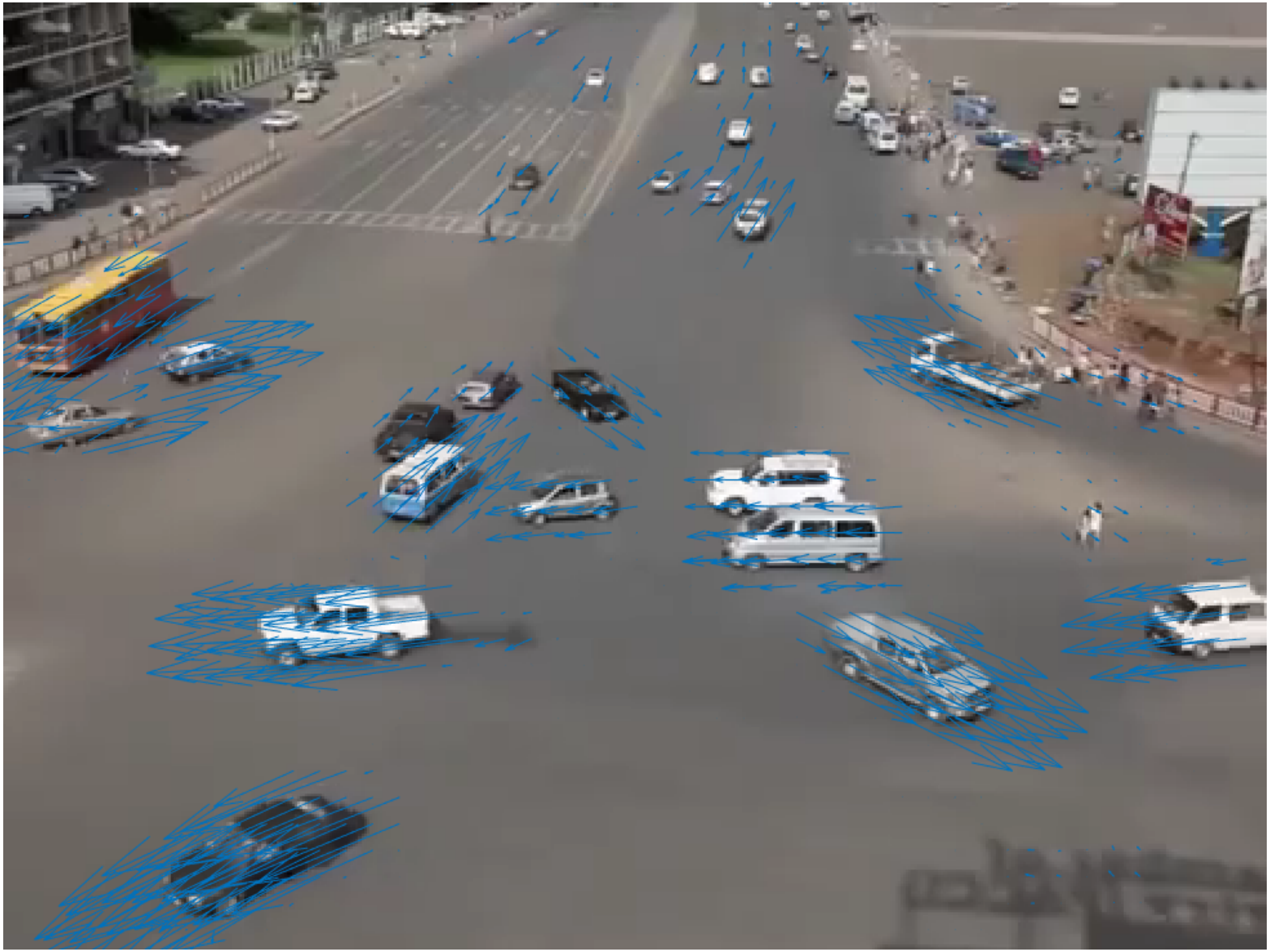
When it works (and when it doesn't)

- When optical flow excels
  - Corners (these do not have an aperture problem)
- When optical flow may fail
  - Brightness constancy is not satisfied
  - Large displacement
  - Textureless regions
  - Edges
  - A point does not move like its neighbours
  - Occlusions

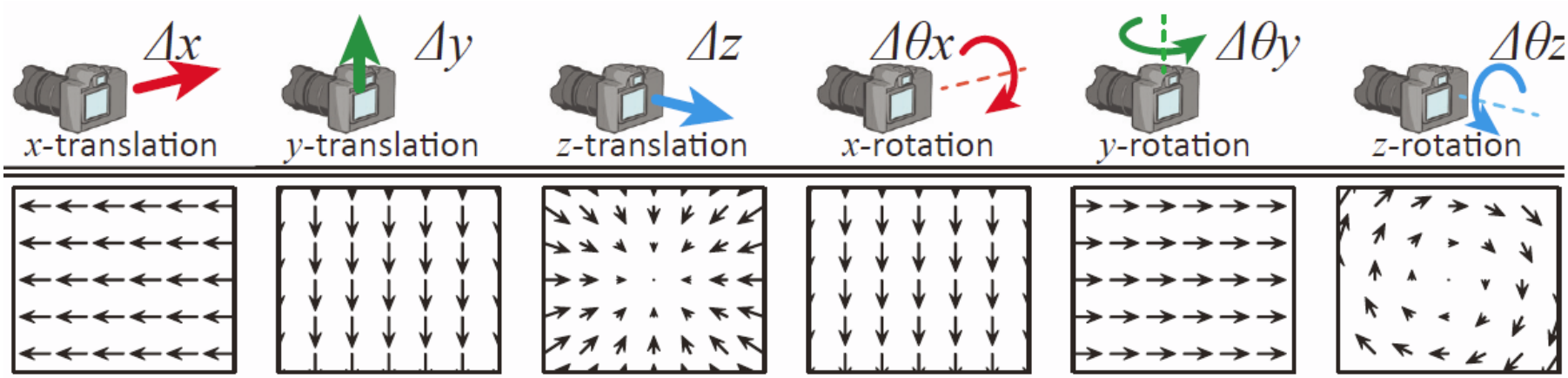
Goal: Determine which direction objects are moving

http://www.mathworks.com/matlabcentral/fileexchange/48745-lucas-kanade-tutorial-example-2

Result



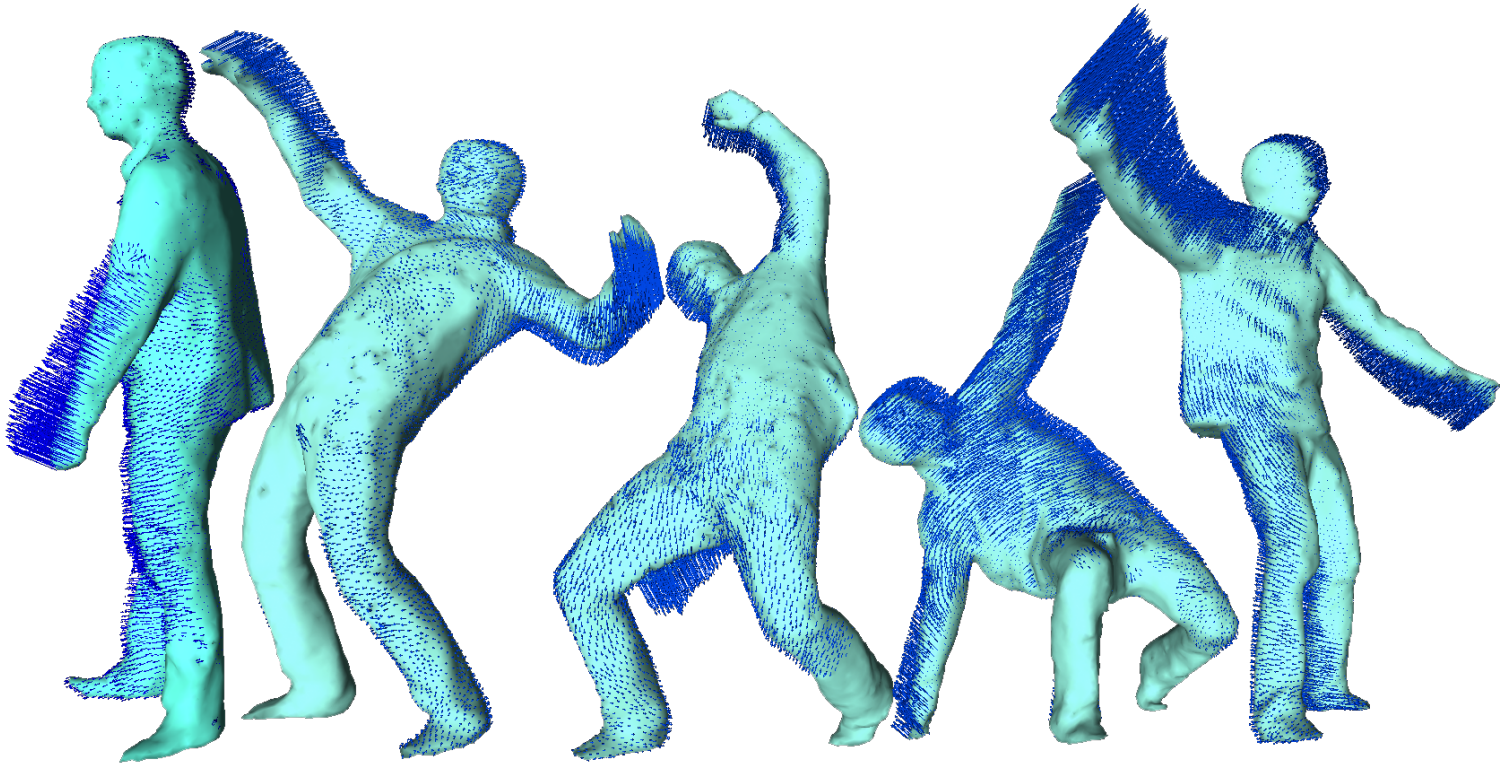
Optical flow fields – moving camera



http://www1.cs.columbia.edu/CAVE/projects/spedo/images/SpeckleFlowForCameraMotions.png

Scene flow

- For dynamic scenes (and a static camera), optical flow is the 2D motion observed in the image of moving objects. Note though, the objects themselves are moving in 3D; the optical flow is merely a projection of the 3D motion.
- *Scene flow* is a 3D vector field that exists *on a surface*, indicating which direction points are moving.



https://raweb.inria.fr/rapportsactivite/RA2011/morpheo/uid39.html

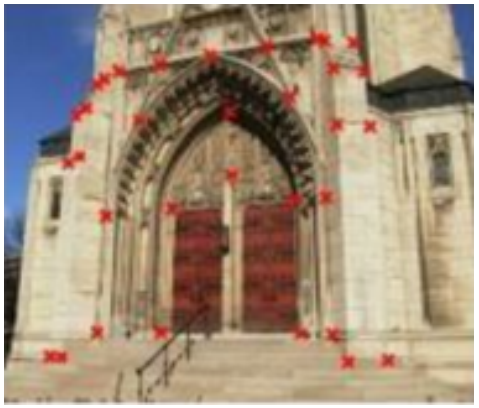
Feature point tracking

- Many computer vision problems involving video require detecting and tracking important points (i.e., features, like corners) *through time*.
- This differs from optical flow. Instead of computing motion vectors at *each* pixel, we identify a smaller set of distinctive *feature points* and track them.
- Typically the problem is easier if the motion is small.
- However, there are challenges:
  - Finding good features to track
  - Efficiency
  - Changing appearance (specular highlights, shadows)
  - Drift (accumulating error)
  - Points getting lost (occlusion, or out of field of view)
  - Occlusion and disocclusion

Feature point tracking

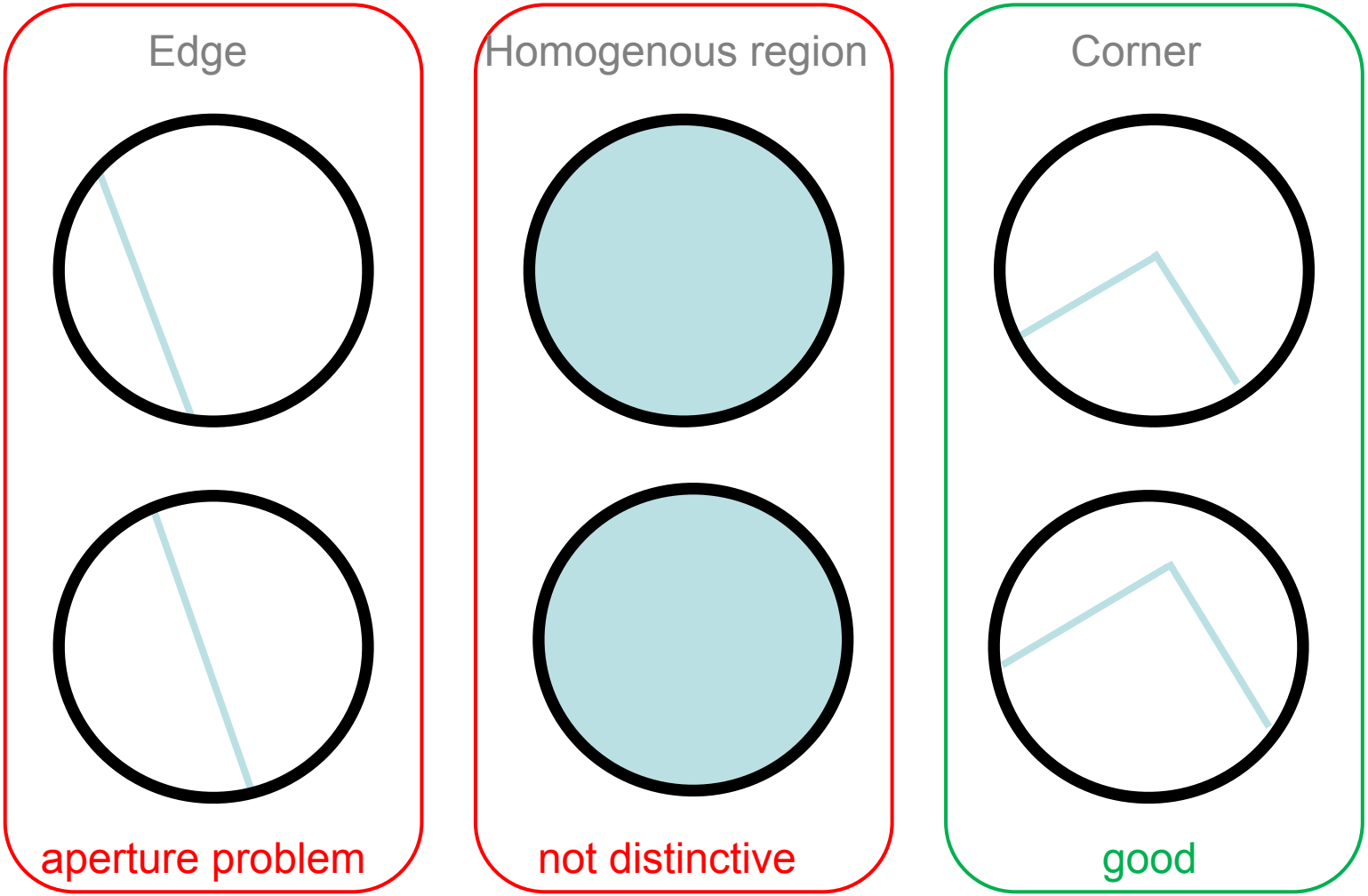
- Minimum eigenfeature + KLT tracking is a good option

- Want to handle the case that new points may come into the field of view, and old points may leave the field of view



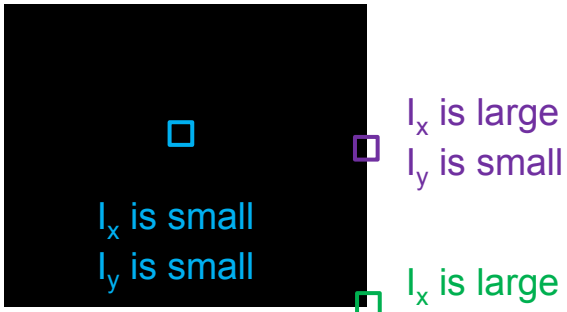
What makes for a good feature?

- A good feature should be *distinctive*, not suffer from the aperture problem, and be *consistent* through the video.

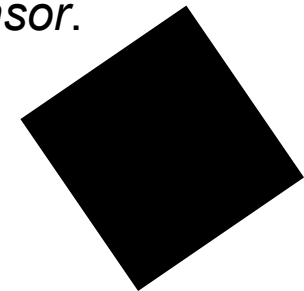


Corner detectors

- Recall corner detection from your earlier CV or Image Proc course (?)
- Corners are a type of feature point, as they are *distinctive* and do not suffer from the aperture problem.
- Corner detectors look for points where there is a *significant* change in intensity in *two* perpendicular directions in a neighbourhood around a pixel.



- However, we'd like to detect corners that are not axis-aligned as in the above example.
- This can be achieved by looking at the *eigenvalues* of the *structure tensor*.



Corner (Harris) Detector (Mathematics)

Let's analyze the local intensity changes :

Change of intensity for the shift [u,v]:

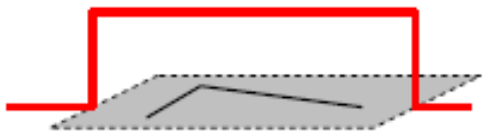
$$E(u,v) = \sum_{x,y} w(x,y)\left[I(x+u, y+v) - I(x,y) \right]^2$$

Window function

Shifted intensity

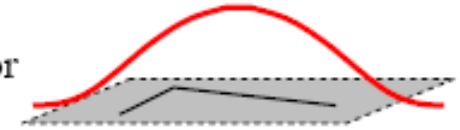
Intensity

Window function $w(x,y)$ =



1 in window, 0 outside

or



Gaussian

Corner Detector (Mathematics)

For small shifts (u,v), we have the following approximation (after Taylor series expansion on I(x+u,y+v) and expanding the quadratic term):

$$E(u,v) \cong \begin{bmatrix} u & v \end{bmatrix} S \begin{bmatrix} u \\ v \end{bmatrix}$$

Where S is a 2x2 matrix computed from image derivatives in the given window:

$$S = \sum_{x,y} w(x,y) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$$

S is called the Image Structure Tensor

Image Structure Tensor

$$S(x,y) = \begin{bmatrix} \sum_W (I_x(x_i,y_i))^2 & \sum_W (I_x(x_i,y_i) I_y(x_i,y_i)) \\ \sum_W (I_x(x_i,y_i) I_y(x_i,y_i)) & \sum_W (I_y(x_i,y_i))^2 \end{bmatrix}$$

- W is the window of a fixed size in your image, (xi,yi) pixel coordinates in that window.

- I_x and I_y are the local approximations to the first order partial derivatives of the image I, which is the filtered image with a Gaussian filter

$$I_x = \frac{\partial I}{\partial x} = \frac{I(x+1,y) - I(x-1,y)}{2} \qquad I_y = \frac{\partial I}{\partial y} = \frac{I(x,y+1) - I(x,y-1)}{2}$$

Structure tensor

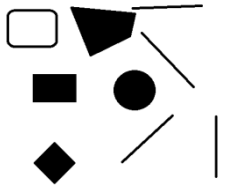
- The structure tensor is a 2D matrix constructed by *blurring* the products of first order derivatives, for example using a Gaussian kernel *G(x,y)*. It is defined as

$$S(x,y) = G(x,y) * \begin{bmatrix} I_x I_x & I_x I_y \\ I_x I_y & I_y I_y \end{bmatrix} = \begin{bmatrix} s_{11} & s_{12} \\ s_{21} = s_{12} & s_{22} \end{bmatrix}$$

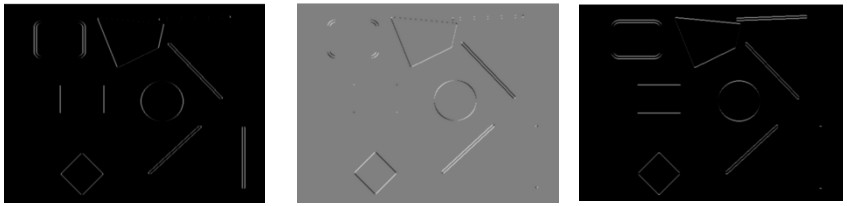
- The structure tensor summarises the distribution of the image gradient in a *neighbourhood* (due to the convolution with *G*) around a point (x, y).

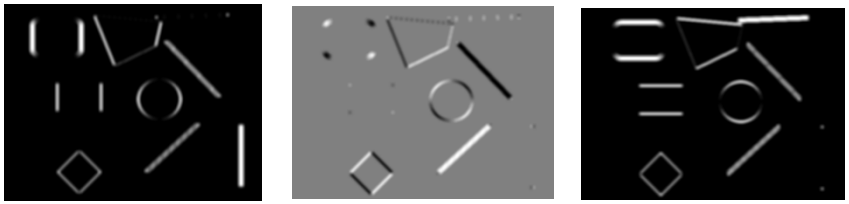
Structure tensor

- Example code



```
J = double(rgb2gray(imread('shapes.png')));
G = fspecial('gaussian', 11, 1);
I = imfilter(J, G);

% Compute derivatives Ix and Iy
Kx = 0.5*[-1 0 1]; Ky = K';
Ix = imfilter(I, Kx);
Iy = imfilter(I, Ky);

% Get elements in the structure tensor
IxIx = Ix.*Ix;
IxIy = Ix.*Iy;
IyIy = Iy.*Iy;

% Blur
G = fspecial('gaussian', 11, 3);
s11 = imfilter(IxIx, G);
s12 = imfilter(IxIy, G);
s22 = imfilter(IyIy, G);
```





Corner Detector (Mathematics)

$$S(x,y)=\begin{bmatrix} \sum_W (I_x(x_i,y_i))^2 & \sum_W (I_x(x_i,y_i)I_y(x_i,y_i)) \\ \sum_W (I_x(x_i,y_i)I_y(x_i,y_i)) & \sum_W (I_y(x_i,y_i))^2 \end{bmatrix}$$

Intensity change in shifting window: perform eigenvalue analysis on S

$$S = V \begin{bmatrix} \lambda_2 & 0 \\ 0 & \lambda_1 \end{bmatrix} V^{-1}$$

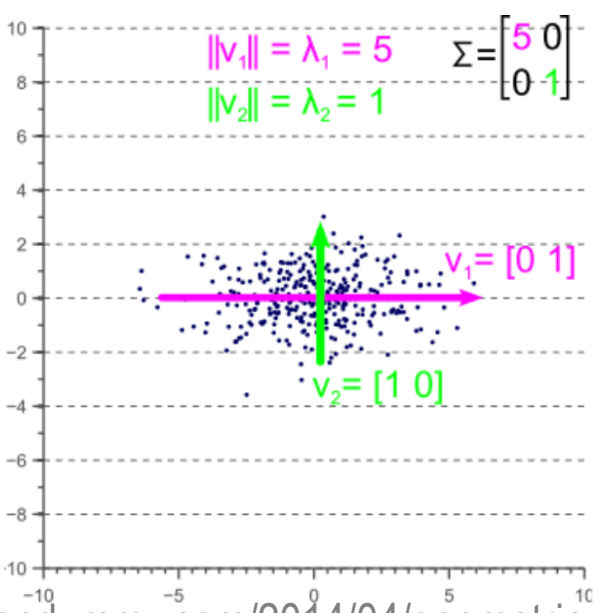
$\lambda_2 \geq \lambda_1$: eigenvalues of S

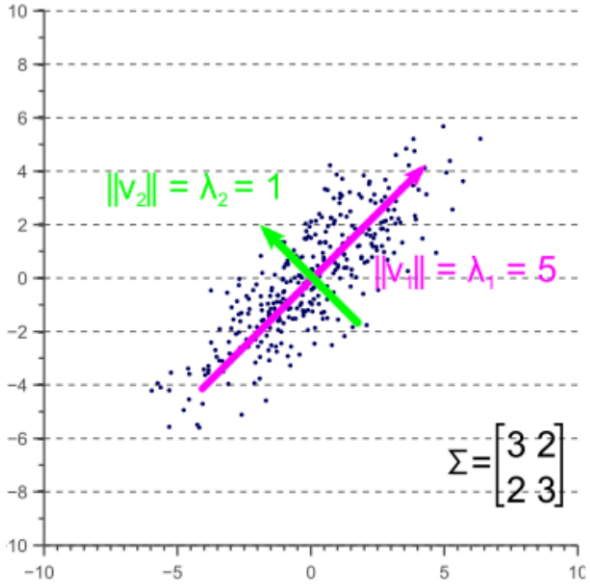
V : Eigenvector matrix of S

Recall:

Σ

covariance matrix of the data





http://www.visiondummy.com/2014/04/geometric-interpretation-covariance-matrix/

Eigenvalues and eigenvectors

- One can perform an eigenanalysis of the structure tensor. This represents the structure tensor as

$$S(x,y) = R^{-1} \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} R$$

where R is a rotation matrix, and λ_1 and λ_2 are eigenvalues that represent the strength of the gradient along the eigendirections, which form the columns of R

- The eigenvalues can be ordered so $\lambda_1 > \lambda_2$. Note that since S is a symmetric matrix, the eigenvalues are positive.

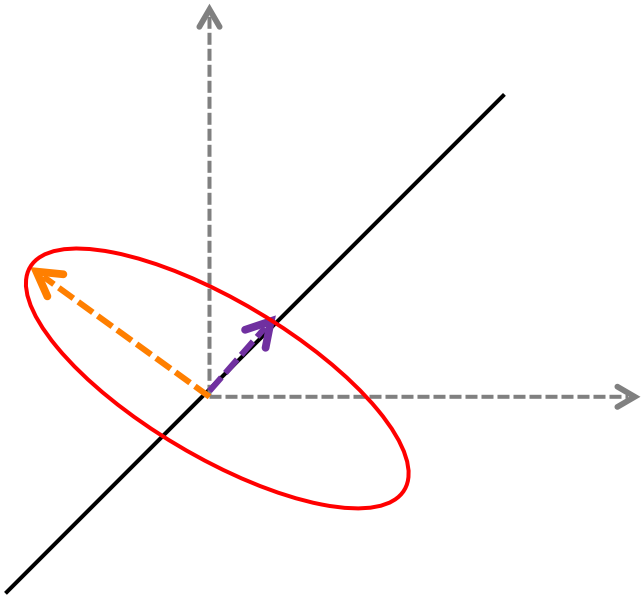
In Matlab

- In Matlab, this can be achieved used `[U, D, V] = svd(S)`, where
  - V is the rotation matrix, composed of eigenvectors along the columns, and
  - D is a diagonal matrix, composed of eigenvalues along the diagonal.

- Toy example:

```
S = [2 -1; -1 2];
[U, D, V] = svd(S)
```

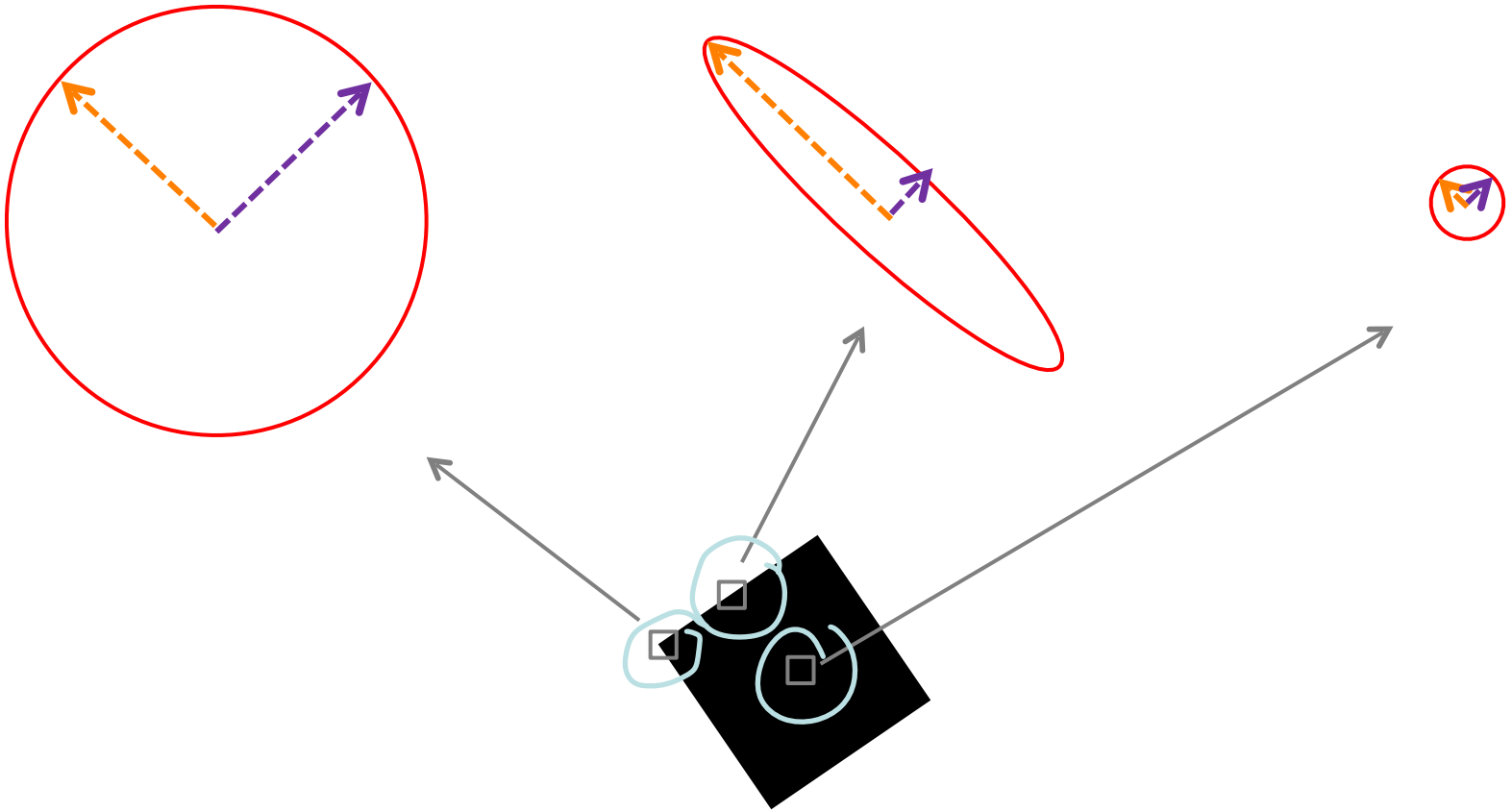
```
U =

   -0.7071    0.7071
    0.7071    0.7071

D =

    3.0000         0
         0    1.0000

V =

   -0.7071    0.7071
    0.7071    0.7071
```



You can think of this as an ellipse, aligned based on the predominant edge

Eigenvalue analysis

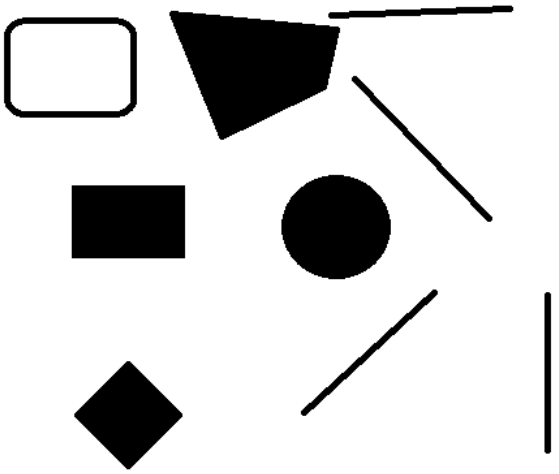
- There are three common scenarios:
  - Both eigenvalues are large: **corner**
  - One eigenvalue is large, the other small: edge
  - Both eigenvalues small: homogeneous region



Minimum eigenvalue corner detection

- Since the eigenvalues are positive, one can look at the *smaller* of the two eigenvalues. If this is *large enough*, then a corner has been identified.
- Example: looking at the value of λ_2 [Shi and Tomasi, '94]

```
% Compute min eigenvalue: see Equation 7 of
% http://www.soest.hawaii.edu/martel/Courses/GG303/Eigenvectors.pdf
lambda2 = 0.5*( (s11+s22)-((s11-s22).^2+4*s12.^2).^0.5 );
figure; imshow(lambda2, []);
```



Image

λ_2

Min eigenvalue detector (Matlab)

- In Matlab, you can use the function detectMinEigenFeatures to find corners based on the minimum eigenvalue technique. The syntax is

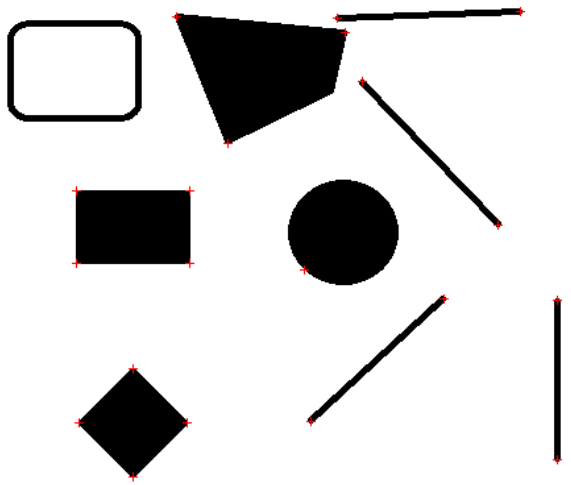
```
points = detectMinEigenFeatures(I, Name, Value)
```

where `I` is a (grayscale) image, and `Name/Value` are optional arguments

  - `'ROI'`, `[1 1 size(I,2) size(I,1)]` (default): the portion of the image in which to look for corners
  - `'FilterSize'`, 5 (default): used to set the Gaussian filter size/std.dev
  - `'MinQuality'`, 0.01 (default): increasing can remove erroneous corners

- Matlab returns points as a cornerPoints object that includes the points and the minEigenvalue

```
I = double(rgb2gray(imread('shapes.png')));
corners = detectMinEigenFeatures(I);
p = corners.selectStrongest(20);
I = insertMarker(I, p, '+', 'Color', 'red');
imshow(I);
```



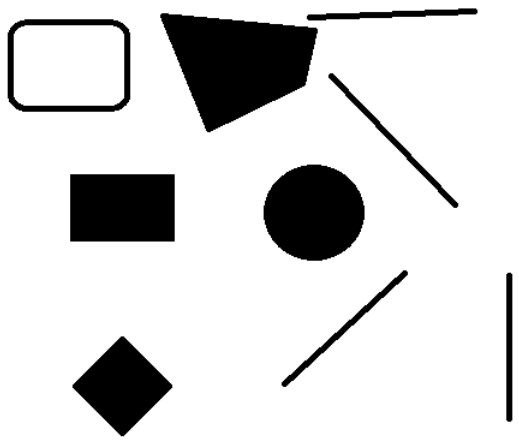
Harris corner detector (in more depth)

- Harris proposed looking for corners based on a cornerness function

$$C = \lambda_1\lambda_2 - \alpha(\lambda_1 + \lambda_2)^2$$

- Where α is a constant in the range [0.04, 0.06]

```
% Compute cornerness in the Harris corner detector
alpha = 0.05;
lambda1 = 0.5*( (a+d)+((a-d).^2+4*b.^2).^0.5 );
lambda2 = 0.5*( (a+d)-((a-d).^2+4*b.^2).^0.5 );
C = lambda1.*lambda2-alpha*(lambda1+lambda2).^2;
figure; imshow(C,[]);
```



Image

C

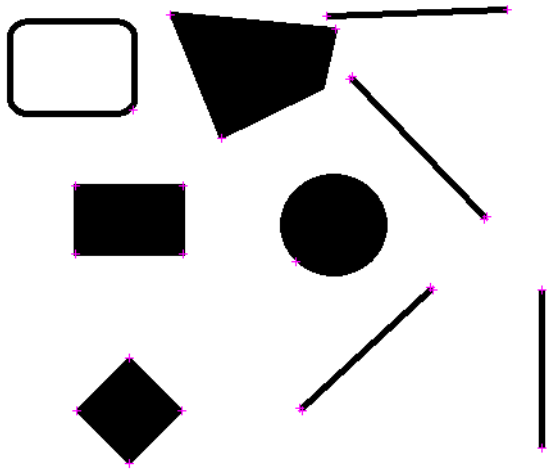
Super easy in fact

- In Matlab, you can use the function detectHarrisFeatures to find corners based on the Harris technique. The syntax is

```
points = detectHarrisFeatures(I, Name, Value)
```

where `I` is a (grayscale) image, and `Name/Value` are optional arguments
  - `'ROI'`, `[1 1 size(I,2) size(I,1)]` (default): the portion of the image in which to look for corners
  - `'FilterSize'`, 5 (default): used to set the Gaussian filter size/std.dev
  - `'MinQuality'`, 0.01 (default): increasing can remove erroneous corners
- Matlab returns points as a cornerPoints object that includes the points and the minEigenvalue

```
I = double(rgb2gray(imread('shapes.png')));
corners = detectHarrisFeatures(I);
p = corners.selectStrongest(20);
I = insertMarker(I, p, '+', 'Color', 'magenta');
imshow(I);
```



Kanade-Lucas-Tomasi (KLT) tracker

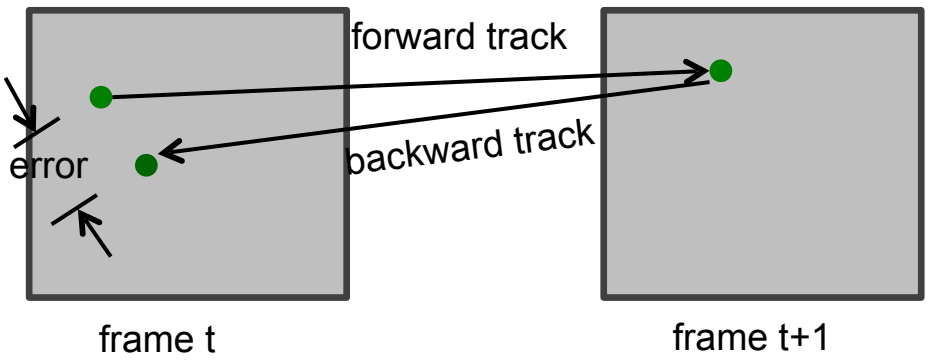
- The KLT tracker is a popular algorithm to track feature points through a video sequence.
- It works best for corners on objects that do not change shape and remain in the field of view.

- It applies the Kanade-Lucas optical flow method to a (sparse) set of feature points.

Kanade-Lucas-Tomasi (KLT) tracker

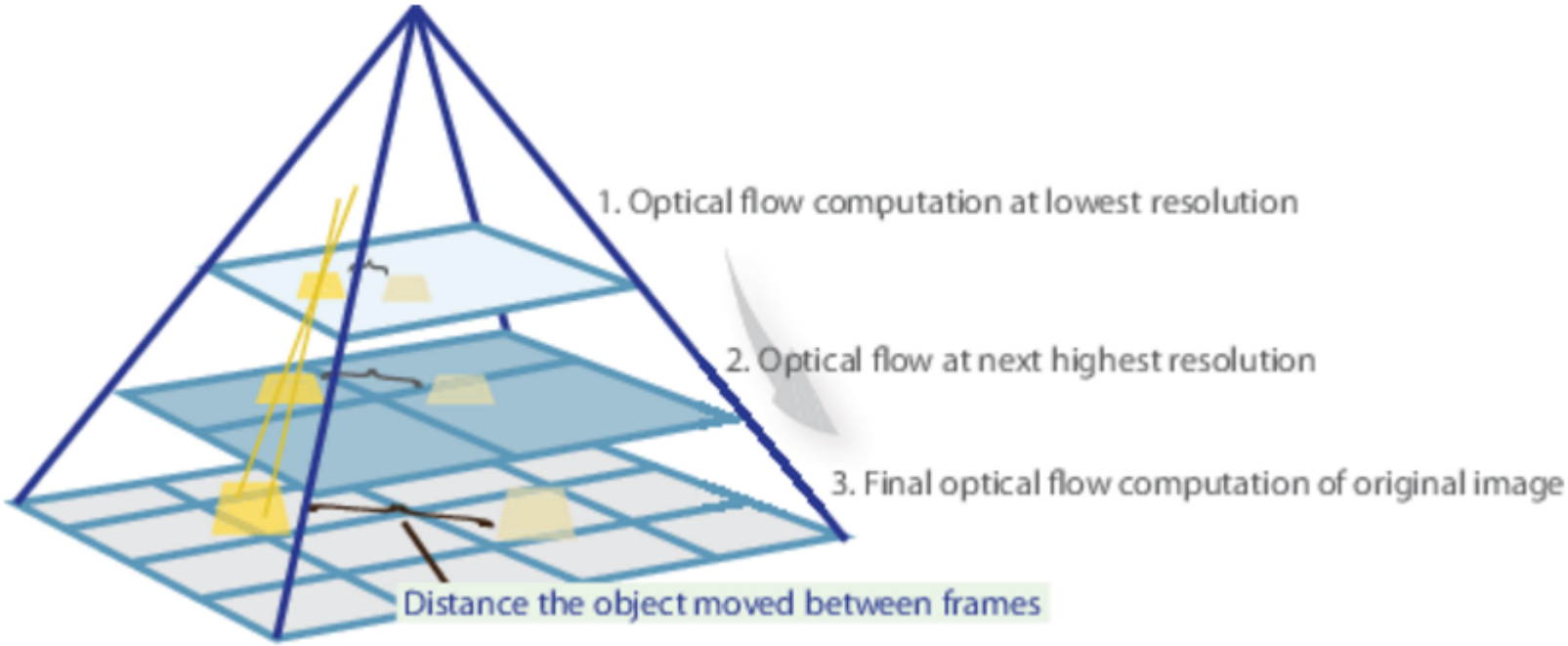
KLT technique in Matlab includes some additional features.

1. **Bidirectional error calculation**. If a feature is correctly tracked from frame t to frame t+1, then it should be possible to track the feature from t+1 back to frame t (e.g., running time backwards). The error, computed as a distance in pixels (from the original location to the final location after backward tracking) should be small. If the error is too high, the point is considered to be "lost".

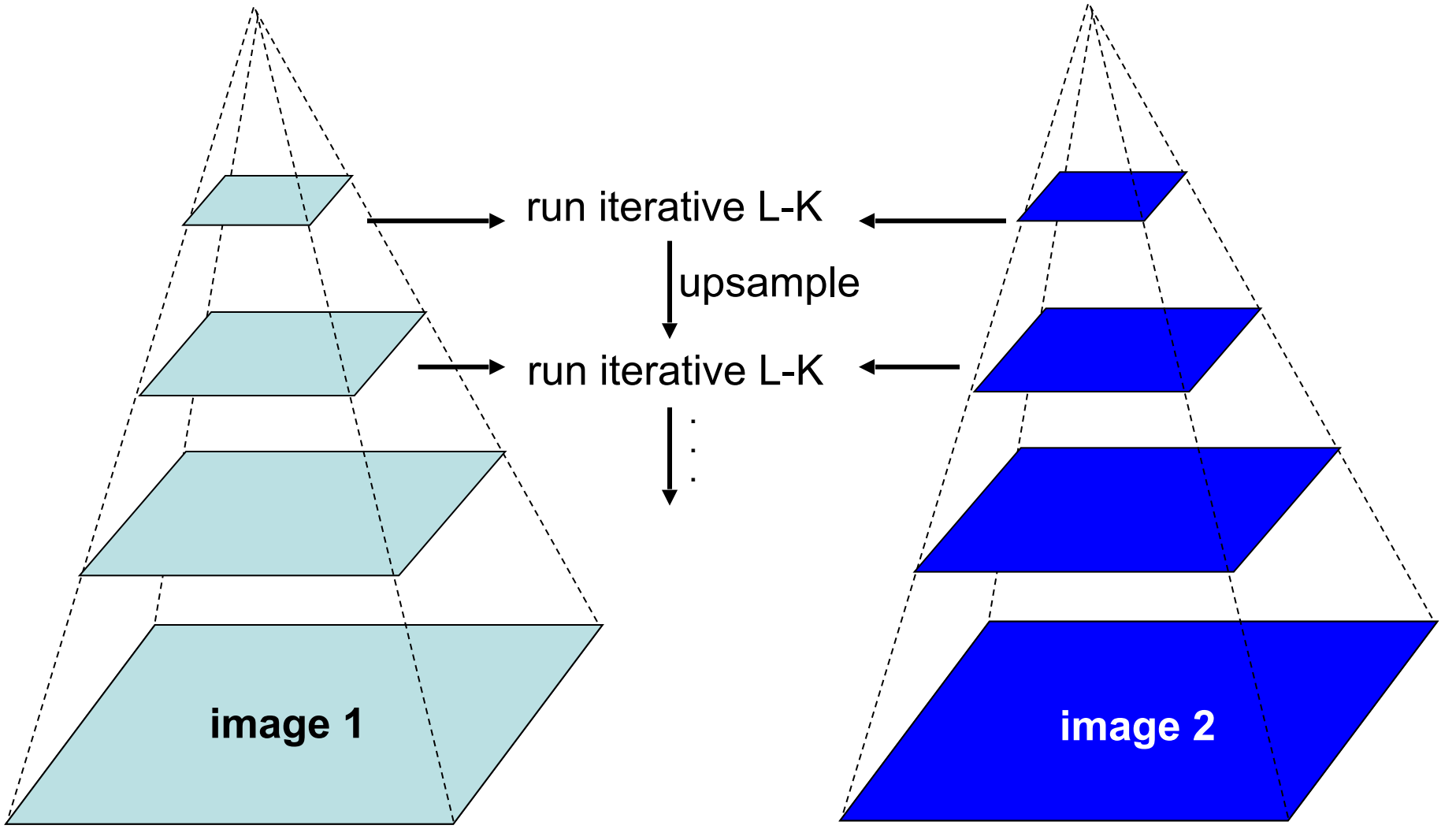


Kanade-Lucas-Tomasi (KLT) tracker

2. **Use of image pyramids**. An image pyramid is a scale-space technique that creates successive downsampled versions of the image and lowering resolutions. The KLT algorithm works in a coarse-to-fine fashion, first tracking on the lowest resolution image and then increasing the resolution. This helps with larger displacements.



Naturally this is done on both frames



Kanade-Lucas-Tomasi (KLT) tracker

3. **Control of neighbourhood size**. The tracker allows the programmer to configure the size of the neighbourhood used in the KL optical flow method. The width and height must be odd integers.

4. **Control of number of iterations**. The tracker is iterative, searching for the new location of each point being tracked until convergence. The algorithm normally converges in 10 iterations, but one can set a limit.

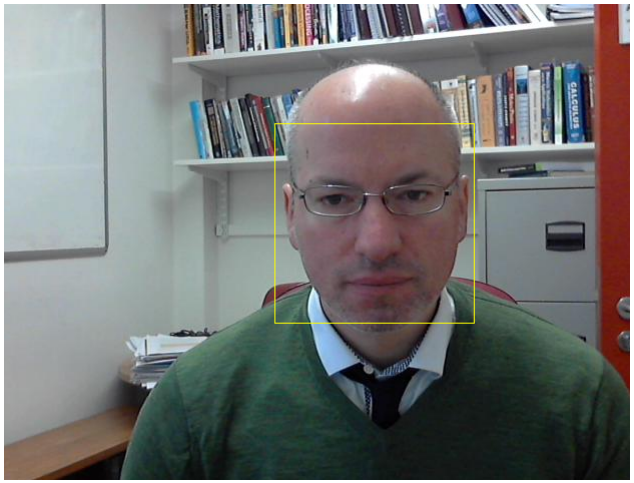
PointTracker

- In Matlab, KLT tracking is implemented using the vision.PointTracker system object. It allows you to track a set of points from one frame to the next.
- To use it,
  1. Create a PointTracker system object.
  2. Initialise the tracking based on some points identified in the first frame.
  3. Track points in the next frame using the `step` method, producing updated point locations.
  4. Go to 3; possibly adding new points to be tracked due to loss

Face tracking example

- First, we detect a face using the cascade object detector

```
videoReader = VideoReader('videoCapture.avi');
I = readFrame(videoReader);


% Create a cascade detector object.
faceDetector = vision.CascadeObjectDetector();
bbox = step(faceDetector, I);
I = insertShape(I, 'Rectangle', bbox);
figure; imshow(I);
```



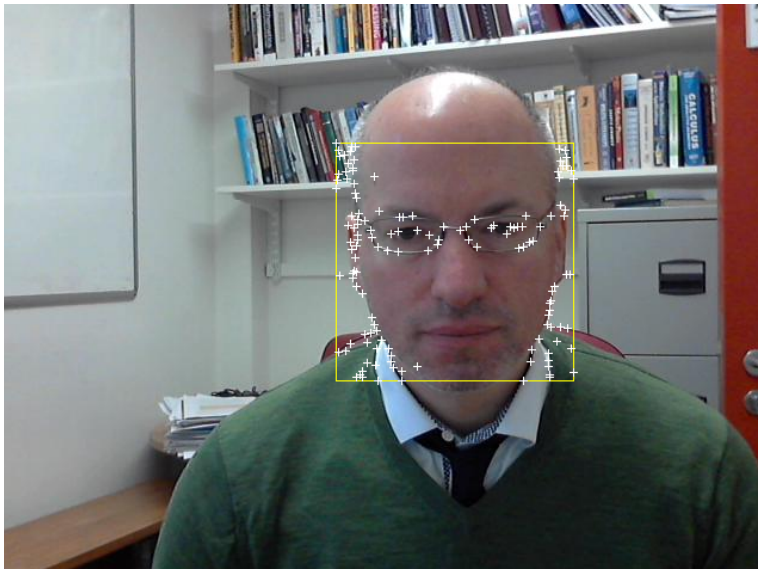
Face tracking example

- Now, let's find some good features to track in the face area

```
points = detectMinEigenFeatures(rgb2gray(I), 'ROI', bbox);
points = points.Location;
I = insertMarker(I, points, '+', 'Color', 'white');
imshow(I);
```

Limit to a bounding box

⇨ How might this be improved?



Face tracking example

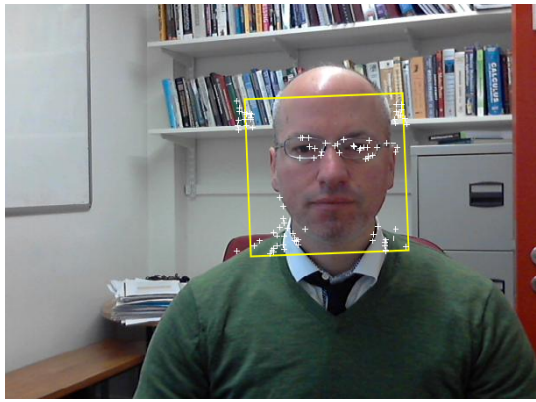
- Let's track those points through the video

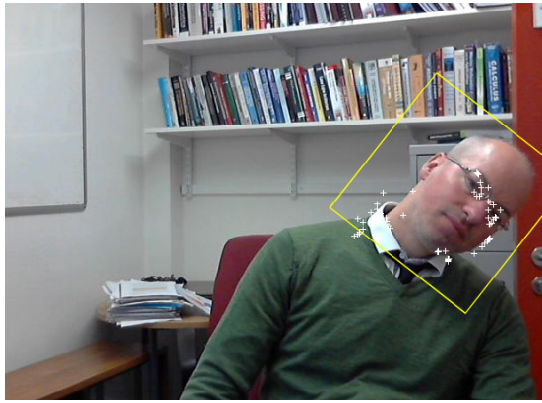
```
pointTracker = vision.PointTracker();
initialize(pointTracker, points, I);

while hasFrame(videoReader)
    I = readFrame(videoReader);

    % Track the points. Note that some points may be lost.
    [points, isFound] = step(pointTracker, I);
    visiblePoints = points(isFound, :);

    I = insertMarker(I, visiblePoints, '+', 'Color', 'white');
    image(I);

    % Update points
    setPoints(pointTracker, visiblePoints);

end

release(pointTracker);
```

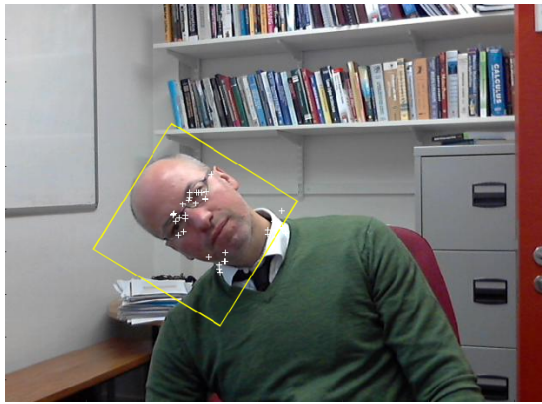
Face tracking example

- Tracking the face with a box

```
pointTracker = vision.PointTracker();
initialize(pointTracker, points, I);

% Get the bounding box, and save the points detected so far
bboxPoints = bbox2points(bbox(1, :));
oldPoints = points;

while hasFrame(videoReader)
    I = readFrame(videoReader);

    % Track the points. Note that some points may be lost.
    [points, isFound] = step(pointTracker, I);
    visiblePoints = points(isFound, :);
    oldInliers = oldPoints(isFound, :);

    [xform, oldInliers, visiblePoints] = estimateGeometricTransform(...
        oldInliers, visiblePoints, 'similarity', 'MaxDistance', 4);

    % Apply the transformation to the bounding box points and insert into image
    bboxPoints = transformPointsForward(xform, bboxPoints);
    bboxPolygon = reshape(bboxPoints', 1, []);
    I = insertShape(I, 'Polygon', bboxPolygon, 'LineWidth', 2);
    I = insertMarker(I, visiblePoints, '+', 'Color', 'white');
    image(I);

    % Reset the points
    oldPoints = visiblePoints;
    setPoints(pointTracker, visiblePoints);

end
release(pointTracker);
```







References

- B.K.P. Horn and B.G. Schunck, "Determining optical flow." *Artificial Intelligence*, vol 17, pp 185–203, 1981.
- B. Lucas and T. Kanade. “An iterative image registration technique with an application to stereo vision.” In *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 674–679, 1981.
- G. Farneback, "Two-Frame Motion Estimation Based on Polynomial Expansion." *Proceedings of the 13th Scandinavian Conference on Image Analysis*. Gothenburg, Sweden, 2003.
- Shi, J. and Tomasi, C. (1994). Good features to track. In IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR’94), pp. 593–600, Seattle.

Video stabilisation example

- In this example, we will identify feature points and track them through a video.
- We'll use the tracked points to estimate an affine transformation from one frame to the next, and composite this transformation to stabilise the video.

Unstabilised video

Stabilised video

Initialising the KLT tracker

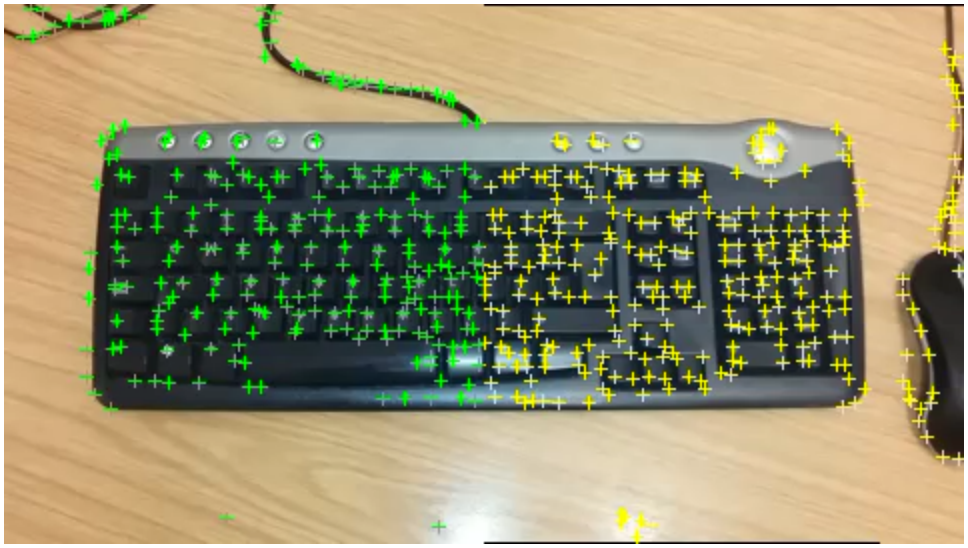
```
% Load a video
videoReader = VideoReader('jerkyKeyboard.mp4');
I = imresize(readFrame(videoReader), 0.25);
[rows, cols, planes] = size(I);

% Initialise the tracker on the feature points
points = detectMinEigenFeatures(rgb2gray(I));
pointTracker = vision.PointTracker();
initialize(pointTracker, points.Location, I);
oldPoints = points.Location;
```

Stabilisation

```
Tcumulative = affine2d;
while hasFrame(videoReader)
    % Process each frame
    I = imresize(readFrame(videoReader), 0.25);
    [points, isFound] = step(pointTracker, I);
    visiblePoints = points(isFound, :);

    % Find a mapping between frames and accumulate from first frame
    tform = estimateGeometricTransform(points, oldPoints, 'affine');
    Tcumulative.T = tform.T *  Tcumulative.T;

    % Warp image and points
    IWarp = imwarp(I, Tcumulative, 'OutputView', imref2d(size(I)));
    warpedPoints = transformPointsForward(Tcumulative, points);

    % Show tracked points on unwarped and warped images
    I = insertMarker(I, points, '+', 'Color', 'green');
    IWarp = insertMarker(IWarp, warpedPoints, '+', 'Color', 'yellow');

    % Composite video to show as a splitscreen
    K = IWarp;
    K(1:rows, 1:cols/2, :) = I(1:rows, 1:cols/2, :);
    imshow(K);

    % Reset the points for tracking on the next frame
    oldPoints = visiblePoints;
    setPoints(pointTracker, oldPoints);

end
```

Result

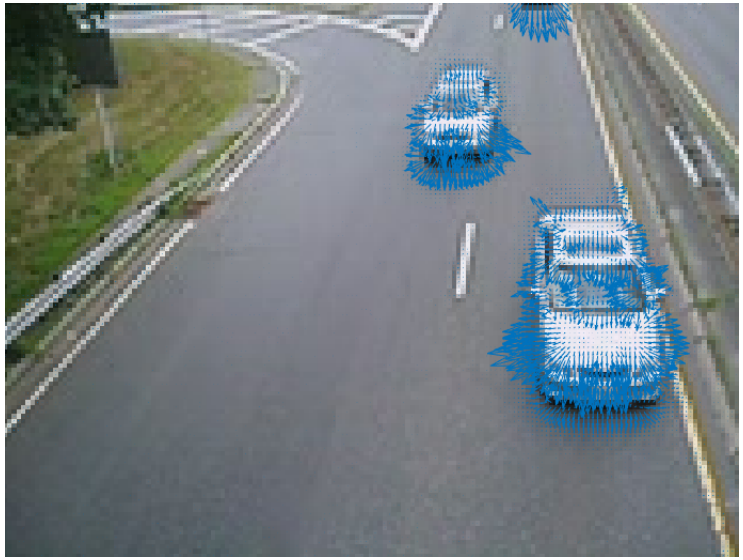
- Some more examples



Horn-Schunck in Matlab

- In Matlab, you can implement optical flow using Horn-Schunck using the `estimateFlow` function and an `opticalFlowHS` object.
- Syntax(es)
  - `flow = estimateFlow(I, obj)` where `I` is a grayscale image and `obj` is an opticalFlow object.
  - `obj = opticalFlowHS(Name, Value)` where `Name/Value` are optional arguments, e.g., `'Smoothness'`, 0.5 or `'MaxIteration'`, 15
- Example:

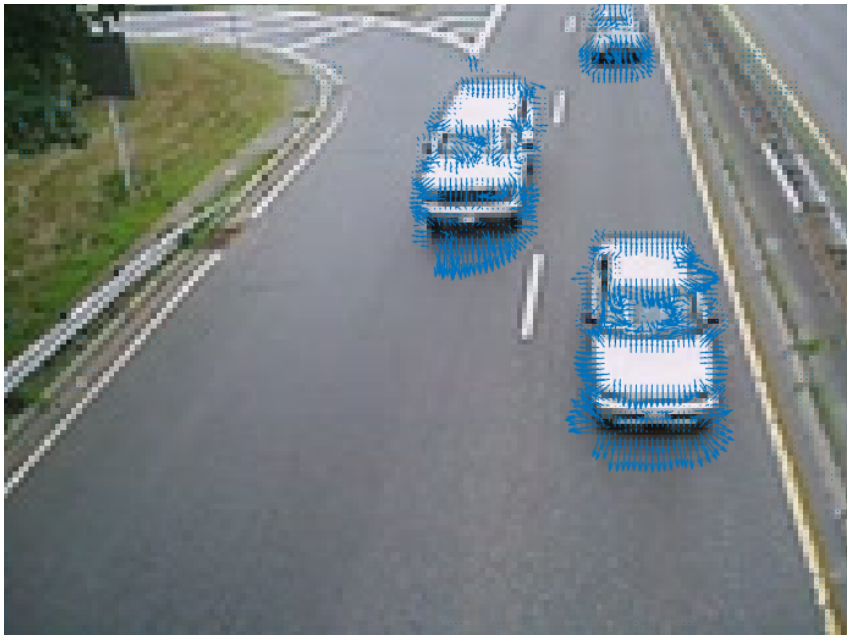
```
vidReader = VideoReader('viptraffic.avi');
ofHS = opticalFlowHS;
while hasFrame(vidReader)
    I = readFrame(vidReader);
    G = rgb2gray(I);

    flow = estimateFlow(ofHS, G);

    imshow(I);
    hold on;
    plot(flow,'ScaleFactor',25);
    hold off;
end
```



Lucas-Kanade in Matlab

- In Matlab, you can implement optical flow using Lucas-Kanade using the `estimateFlow` function and an `opticalFlowLK` object.
- Syntax
  - `obj = opticalFlowLK(Name, Value)` where `Name/Value` are optional argument for a noise threshold: `'NoiseThreshold'`, 0.005
- Example:

```
vidReader =
VideoReader('viptraffic.avi');
ofLK = opticalFlowLK;

while hasFrame(vidReader)
    I = readFrame(vidReader);
    G = rgb2gray(I);

    flow = estimateFlow(ofLK, G);

    imshow(I);
    hold on;
    plot(flow,'ScaleFactor',2);
    hold off;
end
```



Farneback's method

- Farneback's method locally models the image brightness as a 2D polynomial function, and computes the translation between frames. It uses a pyramidal (coarse to fine) approach and is iterative.
- Syntax
  - `obj = opticalFlowFarneback(Name, Value)` where `Name/Value` are optional arguments, for example: `'NumPyramidLevels'`, 2 or `'NumIterations'`, 5
- Example:

```
vidReader =
VideoReader('viptraffic.avi');
ofF = opticalFlowFarneback;
while hasFrame(vidReader)
    I = readFrame(vidReader);
    G = rgb2gray(I);

    flow = estimateFlow(ofF, G);

    imshow(I);
    hold on;
    plot(flow,'DecimationFactor', ...
         [5 5],'ScaleFactor',2);
    hold off;
end
```