Shi-Tomasi, Harris corners and KLT Tracker

Mark Fiala
Ryerson University
Mark.fiala@ryerson.ca
Motivating Interest Points
Finding Correspondences: comparing patches of pixels

• Task: find points **between 2 images** that correspond to the same object – then use these correspondences for computer vision applications (finding pose, SLAM, building 3D models, locating objects, etc...)

• Single pixel typically not distinctive – use patch of pixels in neighbourhood around a point

• Compare a 2D patch of one image to a patch of the same size in another image – apply some **similarity measure** (score)

• One similarity measure is **SSD** (Sum of Squared Differences) – add up the square of differences between pixels in corresponding positions
Finding Correspondences

Compare patches

$$SSD = \Sigma_{i=1}^{n} (X_i - Y_i)^2$$

best score = lowest SSD
Selecting Image Patches likely to match

Given similarity measures, how can we find corresponding image points?

1. **Brute force test every possible patch in first image with every possible location in the other image**
   - Prohibitive computational cost.
   - Also, most patches are on edges or blank regions who aren’t finding reliable matches anyways

2. **Use an interest point detector or corner detector to find a few hundred candidates – just match those**
   - How can we figure out if a patch is likely to have a unique match in the other image? We can examine a patch first, and declare it an interest point.
   - We could test each image patch **within its own image first** before comparing it with the other image.
   - See if a patch matches a neighbourhood of points around it. If there is no good match nearby then it is a distinctive patch – label it an interest point. We reduce computational cost by wh.
   - Is there an even better way to do this, to save on patch comparisons around each point?
Types of Patches

- Blank region matches many spots
- 2-D uncertainty in matching
- Will match with itself well in close neighbourhood in all directions – and thus will not match uniquely in other image

- Ambiguous matches along edge
- 1-D uncertainty in matching
- Will match with itself well in close neighbourhood along the edge – and thus will not match uniquely in other image

- Distinct region (corner), not ambiguous
- Will NOT match with itself well in close neighbourhood in any directions – and thus could match uniquely in other image
Finding Patches that don’t match their Neighbours

• See if a patch matches a neighbourhood of points around it. If there is no good match nearby then it is a distinctive patch – label it an interest point.

• Is there an even better way to do this, to save on patch comparisons around each point?

• Look at spatial derivatives \( \frac{dI}{dx} \) and \( \frac{dI}{dy} \)
• use first order assumption that each pixel will change by \( \frac{dI}{dx} \delta x + \frac{dI}{dy} \delta y \)
• Find SSD patch comparison as a function of small change \( [\delta x, \delta y]^t \)

• SSD \( \approx ||D|| = D^tD \) where \( D= \) Assume difference in a pixel is defined by linear approximation – use first derivative X displacement vector

\[
D = \begin{bmatrix}
\frac{dI_0}{dx} & \frac{dI_0}{dy} \\
\frac{dI_1}{dx} & \frac{dI_1}{dy} \\
\frac{dI_2}{dx} & \frac{dI_2}{dy} \\
\frac{dI_3}{dx} & \frac{dI_3}{dy} \\
\vdots
\end{bmatrix}
\begin{bmatrix}
\delta x \\
\delta y
\end{bmatrix}
\]

\[
SSD = \begin{bmatrix}
\delta x & \delta y
\end{bmatrix}
\begin{bmatrix}
(\Sigma_i \frac{dI_i}{dx})^2 & (\Sigma_i \frac{dI_i}{dx}) (\Sigma_i \frac{dI_i}{dy}) \\
(\Sigma_i \frac{dI_i}{dy}) (\Sigma_i \frac{dI_i}{dx}) & (\Sigma_i \frac{dI_i}{dy})^2
\end{bmatrix}
\begin{bmatrix}
\delta x \\
\delta y
\end{bmatrix}
\]
Finding Patches that don’t match their Neighbours

• correlate patch with patches from same source image
• if a patch matches its neighbours well, it likely won’t be uniquely found in other image
• one way – brute force compare patch with neighbours
• needs \( b^2p^2 \) pixel operations – with \( p=11 \), \( b=11 \) this is \(~10^4\) operations per pixel

Can we reduce these operations?
• Approximate SSD using linear assumption of constant spatial derivatives
• Create corner matrix using \( dI/d_x \) and \( dI/d_y \)
• find SSD using equation

\[
SSD = [x \ y][C][x \ y]
\]

• Corner matrix \( C \)

\[
C = \begin{bmatrix}
\sum I_x^2/n_x & \sum I_x I_y/n_y \\
\sum I_x I_y/n_x & \sum I_y^2/n_y
\end{bmatrix}
\]

• Approximate SSD using \( C \)
Using C Matrix to find Interest Points

- use `klt_corner_gui.exe` (can download from http://www.scs.ryerson.ca/~mfiala)
- 2x2 C matrix decomposed to find ellipse major and minor axes
- use minimum of the two axes (smaller eigenvalue of C)
- large minimum eigenvalue = tight ellipse
  = large change in SSD for small change in position = distinctive point

Edge patch – not so distinct (min eigenvalue=2.5)

```
C=[ 3.6697  5.1656
   -5.1656 25.5682]
= [ 0.98 -0.22
    0.22 0.99][ 2.5 0
    0 26.7][ 0.98 0.22
    -0.22 0.98]
pptch width: 11
SUM(dI/dx)^2: 36697
SUM(dI/dx)(dI/dy): 51656
SUM(dI/dy)^2: 255002
```

Bland region – no real change in SSD, not distinct at all (min eigenvalue=0.1)

```
C=[ 0.9542 -0.0006
   -0.0006 0.9925]
= [ 1.00 -0.02
    0.02 1.00][ 0.1 0
    0 0.1][-0.02 1.00
    -0.02 1.00]
pptch width: 11
SUM(dI/dx)^2: 542
SUM(dI/dx)(dI/dy): 6
SUM(dI/dy)^2: 925
```
More patches

**Edge patch – not so distinct** (min eigenvalue=0.5)

\[
C = \begin{bmatrix}
  60.2957 & -10.3660 \\
  -10.3660 & 2.2885 \\
\end{bmatrix}
\]

\[
= \begin{bmatrix}
  -0.99 & -0.17 \\
  0.17 & -0.99 \\
\end{bmatrix} \begin{bmatrix}
  62.1 & 0 \\
  0 & 0.5 \\
\end{bmatrix} \begin{bmatrix}
  -0.99 & 0.17 \\
  0.17 & -0.99 \\
\end{bmatrix}
\]

**patch width: 11**

SUM(dI/dx)^2: 602957
SUM(dI/dx)(dI/dy): -103660
SUM(dI/dy)^2: 22885

**Edge patch – not so distinct** (min eigenvalue=0.1)

\[
C = \begin{bmatrix}
  0.0594 & -0.0060 \\
  -0.0060 & 13.5441 \\
\end{bmatrix}
\]

\[
= \begin{bmatrix}
  1.00 & -0.00 \\
  0.00 & 1.00 \\
\end{bmatrix} \begin{bmatrix}
  0.1 & 0 \\
  0 & 13.5 \\
\end{bmatrix} \begin{bmatrix}
  1.00 & 0.00 \\
  0.00 & 1.00 \\
\end{bmatrix}
\]

**patch width: 11**

SUM(dI/dx)^2: 594
SUM(dI/dx)(dI/dy): -60
SUM(dI/dy)^2: 135441

**region with edges in both directions, more distinct** (min eigenvalue=8.1)

\[
C = \begin{bmatrix}
  18.9757 & 1.1979 \\
  1.1979 & 8.1854 \\
\end{bmatrix}
\]

\[
= \begin{bmatrix}
  -0.99 & 0.11 \\
  -0.11 & -0.99 \\
\end{bmatrix} \begin{bmatrix}
  19.1 & 0 \\
  0 & 8.1 \\
\end{bmatrix} \begin{bmatrix}
  -0.99 & -0.11 \\
  0.11 & -0.99 \\
\end{bmatrix}
\]

**patch width: 11**

SUM(dI/dx)^2: 189757
SUM(dI/dx)(dI/dy): 11979
SUM(dI/dy)^2: 81854
More patches

**Edge patch – not so distinct** (min eigenvalue=3.0)

\[ C = \begin{bmatrix} -3.4970 & 1.3368 \\ 1.00 & 1.00 \\ 0 & -0.09 \\ 1.00 & 0 \end{bmatrix} \]

\[ \text{patch width: 11} \]

\[ \text{SUM}(d/dx)^2 = 31070 \]

\[ \text{SUM}(d/dx)(d/dy) = -13368 \]

\[ \text{SUM}(d/dy)^2 = 178397 \]

**Corner patch – more distinct** (min eigenvalue=5.0)

\[ C = \begin{bmatrix} 6.9338 & -0.4271 \\ 0.21 & -0.21 \\ 0 & 5.0 \end{bmatrix} \]

\[ \text{patch width: 11} \]

\[ \text{SUM}(d/dx)^2 = 69598 \]

\[ \text{SUM}(d/dx)(d/dy) = -4227 \]

\[ \text{SUM}(d/dy)^2 = 51211 \]

**even more distinct** (min eigenvalue=19.3)

\[ C = \begin{bmatrix} 19.2933 & -3.8639 \\ 3.8639 & 28.4263 \end{bmatrix} \]

\[ \text{patch width: 11} \]

\[ \text{SUM}(d/dx)^2 = 192933 \]

\[ \text{SUM}(d/dx)(d/dy) = -38639 \]

\[ \text{SUM}(d/dy)^2 = 284263 \]
Min Eigen Image

Calculate min eigenvalue for each pixel position
Min Eigen Image

Calculate min eigenvalue for each pixel position
Find local peaks – write these out as interest points
Ways to Speed up Corner Detection

- Finding eigenvalues of corner matrix $C$ requires some calculation
- We can cut some corners if we use the fact that the trace and determinant of the matrix do not change with rotation ($U,V$ matrices from SVD)

\[
C = \begin{bmatrix}
19.2933 & -3.8639 \\
-3.8639 & 28.4263
\end{bmatrix}
\]

\[
= \begin{bmatrix}
0.94 & -0.34 \\
0.34 & 0.94
\end{bmatrix}
\begin{bmatrix}
17.9 & 0 \\
0 & 29.8
\end{bmatrix}
\begin{bmatrix}
0.94 & 0.34 \\
-0.34 & 0.94
\end{bmatrix}
\]

- Label the two eigenvalues $(A,B)$, trace $= A+B$, determinant $= A \times B$
- All we are interested in is if the smaller of $A$ and $B$ is greater than a threshold
- Harris corner detector uses metric

\[\text{det}(A) - \kappa \text{trace}^2(A)\]

- Suggested $k=0.25$. Only if this quantity is above a threshold do we calculate the full eigenvalues – saves lots of calculations
Finding KLT corners – boat example
Finding KLT corners – boat example
Finding KLT corners – car example
Finding KLT corners – car example

KLT/Harris corners doesn’t give good results for all images
OpenCV interest point detector – cvGoodFeaturesToTrack()

Implements C-matrix, min eigenvalue method (Lec 5 KLT/Harris corner detector).

- Needs greyscale IplImage as input, provides CvPoint2D32f list output

```c
int main(int argc, char **argv)
{
    IplImage *cvmg=cvLoadImage("lab_1.jpg");
    CvSize img_sz = cvSize(cvmg->width, cvmg->height);
    IplImage *greyImg = cvCreateImage( img_sz, IPL_DEPTH_8U, 1 );

    //convert to greyscale since cvGoodFeaturesToTrack() needs a grey image
    cvCvtColor( cvimg, grayImg, CV_BGR2GRAY );

    //allocate some working space and output point list for cvGoodFeaturesToTrack()
    IplImage* eig_image = cvCreateImage( img_sz, IPL_DEPTH_32F, 1 );
    IplImage* tmp_image = cvCreateImage( img_sz, IPL_DEPTH_32F, 1 );
    int corner_count = MAX_CORNERS;

    CvPoint2D32f* cornersA = new CvPoint2D32f[ MAX_CORNERS ];

    //find interest points
    cvGoodFeaturesToTrack(grayImg,
                          eig_image, tmp_image,
                          cornersA, &corner_count,
                          0.01,5.0,0.0,3.0,0.04);

    //draw corners over original image
    for( int i=0; i<corner_count; i++ )
    {
        CvPoint pt1,pt2,pt3,pt4;
        pt1.x=(int)cornersA[i].x-3; pt1.y=cornersA[i].y;
        pt2.x=(int)cornersA[i].x+3; pt2.y=cornersA[i].y;
        pt3.x=(int)cornersA[i].x; pt3.y=cornersA[i].y-3;
        pt4.x=(int)cornersA[i].x; pt4.y=cornersA[i].y+3;
        cvLine(cvmg,pt1,pt2,CV_RGB(255,0,0),1);
        cvLine(cvmg,pt3,pt4,CV_RGB(255,0,0),1);
    }

cvNamedWindow("cvGoodFeaturesToTrack()");
cvShowImage("cvGoodFeaturesToTrack()",cvmg);

cvWaitKey(0);

    //clean up memory
    cvReleaseImage(&eig_image);
    cvReleaseImage(&tmp_image);
    cvReleaseImage(&cvmg);
    cvReleaseImage(&grayImg);
}
```

Pg 332-334
KLT tracker

Aperture Problem: a small pixel neighbourhood can only detect motion perpendicular to edge -therefore each pixel position can only constrain optic flow $V$ to a 1D space.

Optic Flow Equation:

$$\nabla I^T \cdot \vec{V} = -I_t$$

Use optic flow equation for each pixel in patch – use least squares fit to find $V$

$$I_{x1} V_x + I_{y1} V_y = -I_{t1}$$
$$I_{x2} V_x + I_{y2} V_y = -I_{t2}$$
$$\vdots$$
$$I_{xn} V_x + I_{yn} V_y = -I_{tn}$$
KLT tracker

Optic Flow Equation:

\[ \nabla I^T \cdot \vec{V} = -I_t \]

Use optic flow equation for each pixel in patch – use least squares fit to find \( V \)

\[
\begin{align*}
I_{x1} V_x + I_{y1} V_y &= -I_{t1} \\
I_{x2} V_x + I_{y2} V_y &= -I_{t2} \\
&\vdots \\
I_{xn} V_x + I_{yn} V_y &= -I_{tn}
\end{align*}
\]

Use optic flow equation for each pixel in patch – use least squares fit to find \( V \)

This is of the form \( Ax = B \). Least squares solution is

\[
\begin{bmatrix}
V_x \\
V_y
\end{bmatrix} = \left( A^T A \right)^{-1} A^T B
\]

Notice left quantity is inverse of \( C \) matrix used in corner detection.
KLT tracker

Some links:


OpenCV KLT tracker – cvCalcOpticalFlowPyrLK()

Implements KLT tracking using $C^{-1}$ matrix (Lecture 6)

- Needs start points – uses output of cvGoodFeaturesToTrack()
- Iterates a few times for each point (each step gives linear movement)
- Processes on multiple levels (image pyramid)
- Image pyramid for each (greyscale) image must be created first (pyramid consists of a set of images of different size)

```cpp
// Call the Lucas Kanade tracking algorithm from frame 1 to 2

char features_found[ MAX_CORNERS ];
float feature_errors[ MAX_CORNERS ];

CvSize pyr_sz = cvSize( imgA->width+8, imgA->height/3 );
IplImage* pyrA = cvCreateImage( pyr_sz, IPL_DEPTH_32F, 1 );
IplImage* pyrB = cvCreateImage( pyr_sz, IPL_DEPTH_32F, 1 );
CvPoint2D32f* cornersB = new CvPoint2D32f[ MAX_CORNERS ];
cvCalcOpticalFlowPyrLK(grayImg1, grayImg2, pyrA, pyrB, cornersA, cornersB, corner_count,
  win_sz, 5, features_found, feature_errors,
  cvTermCriteria( CV_TERMCRIT_ITER | CV_TERMCRIT_EPS, 20, .3 ),0 );
```

```cpp
// Lucas-Kanade tracker - compare current frame to first frame
cvCalcOpticalFlowPyrLK(ref_grayImg, track_grayImg, pyrA, pyrB,
  ref_corners, // initial interest points from first image
  tracked_corners, // corresponding interest points from current frame
  corner_count,
  lk_win_sz, 5, features_found, feature_errors,
  cvTermCriteria( CV_TERMCRIT_ITER | CV TERMCRIT_EPS, 20, .3 ),0 );
```

Mark Fiala 2010
OpenCV KLT tracker – another example