Computer Vision  
  
Project 7 – Image Mosaicing

15 April 2014

Christopher Kives  
Aaron Maus

Jonathan Redmann

Table of Contents

|  |  |
| --- | --- |
| Harris Corner Detection------------------------------------------ | 3 |
| Multiscale Corner Detection------------------------------------- | 4 |
| Adaptive Non-maximal Suppresion---------------------------- | 4 |
| MOPS Descriptors-------------------------------------------------- | 7 |
| Matching Correspondences-------------------------------------- | 8 |
| Affine Transformations------------------------------------------- | 10 |
| Image Stitching----------------------------------------------------- | 13 |
| Results--------------------------------------------------------------- | 16 |
| Work Distribution------------------------------------------------- | 16 |
| Citations-------------------------------------------------------------- | 17 |

**Harris Corner Detection**

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%

% HARRISDETECTOR – Uses the Harris method to detect corners in an image

%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%ARGS:

% image - the image

% n – the window size

% k – the classic empirically determined constant from the paper (0.04-0.06)

% type – the method to calculate R

% type 1: harris

% type 2: harmonic

% type 3: shi-tomasi

%OUTPUT:

% R – matrix the same size as the image. Contains the R value for each pixel

% M – matrix same size as image. Contains the hessian for every pixel

function [R, M]=harrisDetector(image, n, k, type)

[r, c, v] = size(image);

if (v > 1)

image = rgb2gray(image);

end

image = im2double(image);

padNum = uint16(floor(n/2));

padRow = zeros(padNum,c);

padCol = zeros(r+(2\*padNum),padNum);

padImg = [padRow;image;padRow];

padImg = [padCol,padImg,padCol];

gauss = fspecial('gaussian', n, n/6);

R = zeros(r,c);

[ix, iy] = imgradientxy(padImg);

M = cell([r,c]);

for x = 1:r

startX = x;

endX = startX+(n-1);

for y = 1:c

startY = y;

endY = startY+(n-1);

w = conv2(single(padImg(startX:endX, startY:endY)), gauss);

tempMat = zeros(2,2);

for i = 1:n-1

for j = 1:n-1

curX = startX+i;

curY = startY+j;

curIx = ix(curX, curY);

curIy = iy(curX, curY);

iMat = [curIx \* curIx, curIx \* curIy;

curIx \* curIy, curIy \* curIy];

tempMat = tempMat + (w(i,j) \* iMat);

end

end

M{x,y} = tempMat;

if (type == 1)

R(x,y) = det(tempMat) - k\*(trace(tempMat))^2;

elseif (type == 2)

R(x,y) = det(tempMat)/(trace(tempMat));

elseif (type == 3)

[eigv, val] = eig(tempMat);

val = diag(val);

R(x,y) = min(val);

else

%error

end

end

end

end

**Multiscale Corner Detection**

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%

% MULTISCALE – Scales down the image n times, returning the pyramid

%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%ARGS:

% im - the image

% n – the number of times to scale down the image

%OUTPUT:

% pyramid – a cell array containing all the scaled images

function pyramid = multiscale(im, n)

image = rgb2gray(im);

pyramid = cell(n);

[r,c] = size(image);

gn = 6;

gauss = fspecial('gaussian', gn, gn/6);

[xi, yi] = meshgrid(1:r, 1:c);

newIm = image;

pyramid{1} = newIm;

for i = 1:(n-1)

newIm = interp2(single(newIm), 2\*xi-(r/2),2\*yi-(c/2));

newIm = imfilter(newIm, gauss);

pyramid{i+1} = uint8(newIm);

end

end

**Adaptive Non-maximal Suppression**

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%

% NMAS – Non-Maximal Area Suppression of the interest points

%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%ARGS:

% N –

% R – an [n x 3] matrix of the potential interest points. Cols 1&2 the coords

% col3 is the R value

% threshold – we will keep interest points with R values above this threshold

% octave – the scaling factor for the patches. The size of a patch is defined

% as 40 / 2^octave

%OUTPUT:

% interestPoints – an [n x 2] matrix. Each row is an interest point

% the first column are the x coordinates

% the second column are the y coordinates

function interestPoints = NMAS( N, R, threshold, octave )

%N = 6;

%threshold = -2.0;

patchSize = 40 / 2^octave;

[ row, col, val ] = find( R.\*(R > threshold) );

% Remove interest points that are within patch size

% of the image edge

startRowVec = col - (1/2)\*patchSize;

endRowVec = col + ((1/2)\*patchSize);

startColVec = row - (1/2)\*patchSize;

endColVec = row + ((1/2)\*patchSize);

if( mod( patchSize, 2 ) == 0 )

endRowVec = endRowVec - 1;

endColVec = endCoVec - 1;

end

goodPoints = ( startRowVec >= 1 & endRowVec <= size( image, 1 )

& startColVec >= 1 & endColVec <= size( image, 2 ) );

newRow = row( goodPoints == 1 );

newCol = col( goodPoints == 1 );

newVal = val( goodPoints == 1 );

[ sortedVal, sortedIndecies ] = sort( newVal, 'descend' );

sortedRow = newRow( sortedIndecies );

sortedCol = newCol( sortedIndecies );

pointsMatrix = [ sortedCol, sortedRow ];

distVector = pdist( pointsMatrix );

distMatrix = squareform( distVector );

pointRadii = zeros( size( newRow, 1 ), 1 );

pointRadii( 1 ) = size( R, 1 );

for i = 2: size( newRow, 1 )

rowVecDist = distMatrix( i, : );

strength = sortedVal( i );

upperIndex = find( abs( sortedVal ) > abs( strength ),

1,

'last' );

potentialVec = rowVecDist( 1:upperIndex );

radius = min( potentialVec );

pointRadii( i ) = radius;

end

[ sortedRadii, sortedRadiiIndicies ] = sort( pointRadii,

'descend' );

finalRow = sortedRow( sortedRadiiIndicies );

finalCol = sortedCol( sortedRadiiIndicies );

finalVal = sortedVal( sortedRadiiIndicies );

interestPoints = [ finalRow( 1:N, 1 ),

finalCol( 1:N, 1 ),

sortedRadii( 1:N, 1 ),

finalVal(1:N)];

end

**MOPS Descriptors**

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%

% MOPS - creates descriptors for all interest points in an image

%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%ARGS:

% image - the image the interest points come from

% interestPoints - an n x 2 matrix, each row contains the x and y of

% an interest point

% octave - an integer representing the octave to analyze

% hessians - a [r c 2 2] matrix holding a hessian for every pixel in

% the image

%OUTPUT:

% descriptors- an [n 8 8] matrix where n is the number of interest points

% basically a matrix holding a descriptor for each interest

% point

function [ descriptors ] = MOPS( image, interestPoints, octave, hessians )

d = size(interestPoints);

numInterestPoints = d(1);

descriptors = cell( 0, 1 );

patchSize = 40 / 2^octave;

image = rgb2gray(image);

image = im2double(image);

for idx = 1:numInterestPoints

currentPoint = interestPoints(idx,:);

x = currentPoint(1);

y = currentPoint(2);

% Realign the image so that the dominant eigenvector of the hessian

% is along the horizontal

[evec, evals] = eig(hessians{x,y});

% The dominant eigenvector is the first eigenvector returned by eig

dominantEigVec = evec(:,1);

% rotate the image so that the dominant eigenvector is horizontal.

angle = double(-radtodeg(atan(dominantEigVec(2)/dominantEigVec(1))));

rotImage = imrotate(image, angle);

% for each interest point

% 1) extract a patch around it (the size of the patch is patchSize x

% patchSize)

% 2) resize the patch to 8x8

% 3) normalize the patch

% create a patch of patchSize x patchSize of pixels around

% interest point

startRow = y - (1/2)\*patchSize;

endRow = y + ((1/2)\*patchSize);

startCol = x - (1/2)\*patchSize;

endCol = x + ((1/2)\*patchSize);

% if the patchsize is even, the interest point can not be

% at the center of the patch. It will be the top left point

% of the lower right quadrant

if(mod(patchSize,2) == 0)

endRow = endRow - 1;

endCol = endCol - 1;

end

% Skip patch if its dimensions exceed matrix dimensions

if( startRow >= 1 && endRow <= size( image, 1 )

&& startCol >= 1 && endCol <= size( image, 2 ) )

% 1) extract out the patch from the rotated image

patch = rotImage(startRow:endRow, startCol:endCol);

% 2) resize the patch to be 8 x 8

% if the image is larger than 8 x 8, it will be sized down

% if the image is smaller than 8 x 8, it will be interpolated

% using bicubic interpolation

patch = imresize(patch, [8 8]);

% 3) normalize

avg = mean2(patch);

stdDev = std2(patch);

patch = patch - avg;

patch = patch ./ stdDev;

% save in descriptors

descriptors{end+1} = patch;

else

error('Holy Hart Picks Batman, we have not eliminated

all the bad corners!');

end

end

end

**Matching Correspondences**

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%

% INTERESTPOINTMATCHING – Match Interest Points from descriptors

%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%ARGS:

% descriptors – all of the descriptors for all images [n x 8 x 8]

% n is the number of descriptors. Each descriptor is

% an 8 x 8 patch.

% numPerIm – the number of descriptors there are per image

% ipToFindPerIm - the number of interest points to find per image

% corrThresh – the minimum value that a cross correlation would be

% considered valid

%OUTPUT:

% mappedIndexes – is a [m x 4] matrix. M is the number of pairs

% col 1 is the index of the first image

% col 2 is the index of the second image

% col 3 is the index descriptor for the first image

% col 4 is the index descriptor for the second image

function mappedIndexes=interestPointMatching(descriptors, numPerIm,

ipToFindPerIm, corrThresh)

[sizex, sizePatch] = size(decriptors);

sizeCorrMat = sizex/sizePatch;

numIms = sizeCorrMat/numPerIm;

corrMat = zeros(sizeCorrMat);

%for each image

for i=1:sizeCorrMat

xcorrMatResult = normxcorr2(descriptors(i:(i+sizePatch-1),

i:(i+sizePatch-1)),

descriptors);

curImDesc = floor(i / numPerIm) + 1;

%store results

for j=1:sizeCorrMat

if (curImDesc ~= floor(j/numPerIm)+1)

corrMat(i,j) = xcorrMatResult(((j-1)\*sizePatch)+1,1);

corrMat(j,i) = corrMat(i,j);

else

corrMat(i,j) = -1;

end

end

end

%mappedIndexs = zeros(ipToFindPerIm,4);

%First and second with image. Third and forth with point index

mappedIndexes = [];

%for each image set

for i = 1:numIms

curCol = corrMat(:,i) .\* (corrMat(:,i) >= corrThresh);

for j = i:numIms

curRow = curCol(j,:);

[colMaxes, colIndexes] = max(curRow,2);%columns of A with maxes

mat = zeros(numPerIm, numPerIm);

mat(colIndexes,:) = colMaxes;

[rowMax, rowIndexes] = max(mat, 1);

[selectedMaxes,rowOrigIndexes] = sort(rowMax);

numFound = find(selectedMaxes);

if (size(numFound,1) >= ipToFindPerIm)

insertVal(1:ipToFindPerIm,1) = i;

insertVal(1:ipToFindPerIm,2) = j;

insertVal(1:ipToFindPerIm,3) = rowIndexes(rowOrigIndexes(

1:ipToFindPerIm));

insertVal(1:ipToFindPerIm,4) = rowOrigIndexes(

1:ipToFindPerIm);

mappedIndexes = [mappedIndexes;insertVal];

end

end

end

end

**Affine Tranformations**

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%

% BUILDTRANSFORMMAT – build a transformation from one image to another

%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%ARGS:

% corrVec – nx4 mat of the form [(x1,y1),(x2,y2)] points. x1,y1 refer to a

% point in im1 and x2,y2 refer to a point in im2

%OUTPUT:

% transform – the transform to transform im2 into im1’s space

function transform=buildTransformMat(corrVec)

[n,~] = size(corrVec);

mat = [];

vec = corrVec(:,3:4);

for i=1:n

x1 = corrVec(1,1);

y1 = corrVec(1,2);

index = (2\*i)-1;

mat(index,:) = [x1,y1,0,0,1,0];

mat(index+1,:) = [0,0,x1,y1,0,1];

end

aVec = mat\vec;

trasformVec = aVec(:,size(aVec,2)-2);

transform = reshape(trasformVec, size(trasformVec,2)/2,size(trasformVec,2)/2,1);

translationVec = aVec(:,size(aVec,2)-1:size(aVec,2));

transform = [transform, translationVec(:)];

transform = [transform; zeros(1,size(transform,2)-1),1];

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%

% APPLYTRANFORMATION – apply a transformation to an image

%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%ARGS:

% im – the image to transform

% affineTransform – the transformation to apply to the image

%OUTPUT:

% out – the transformed image

function out = applyTransformation(im, affineTransform)

out = imwarp(im, affine2d(affineTransform));

end

**Citations**