Computer Vision  
  
Project 7 – Image Mosaicing

15 April 2014

Christopher Kives  
Aaron Maus

Jonathan Redmann

Table of Contents

|  |  |
| --- | --- |
| Main Program------------------------------------------------------ | 3 |
| 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 |

**Main Program**

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

%

% STITCHIMAGES – Master Image Stitching Function

%

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

%ARGS:

% path – the path the images to stitch together

% fileExt – the extension of the images

%OUTPUT:

% outImages – A cell array of all the output images for each octave

function outImages = stitchImages( path, fileExt)

% Global Properties

ipNumber = 50; % the number of interestPoints desired after NMAS

ipPerTransform = 6; % num of interest points needed per affine transform

corrThreshold = .1; % value of the cross correlation threshold

n = 4; % the number of octaves to use

% Read images into a cell array

fname = dir( strcat( path, fileExt ) );

numberImages = length(fname);

imagesArray = cell( numberImages, 1 );

%dah = 'at the top'

%numberImages

for index = 1:numberImages

im = imread( strcat( path, fname(index).name ) );

imagesArray{ index } = im;

end

% Scale each image to n different octaves scaled images

% are stored into a cell array where each row is the

% set of all images at the same scale and each column

% is the same image at n different scales

% 2D cell array to store all images at each scale

scaledImagesArray = cell(4, numberImages);

% Loop over images generating sa scaling pyramind, see [xx]

for index = 1:numberImages

image = imagesArray{ index };

pyramid = multiscale( image, n );

scaledImagesArray(:, index) = pyramid;

end

% END generating scaled images

% Generate the descriptors for each image, storing the set

% of descriptors

% Current ocatave or scale being used, must loop through 0 to 3

% octaves.

outImages = cell(n, 1);

for octave = 0:n-1

% this is a matrix containing the concat of all

% descriptors for a given octave

allDescriptors = [];

allInterestPoints = [];

%dah = 'Before loop'

%numberImages

% Loop over each image at a given octave getting that images

% descriptors. Then store the descriptors in a "matrix accumulator"

% called allDescriptors.

for index = 1:numberImages

%dah = 'in loop'

image = scaledImagesArray{ octave + 1, index };

%dah = 'image size'

%size( image )

%imshow( image );

[strengthMat, hessians] = harrisDetector( image, 6, 0.04, 1);

interestPoints = NMAS( ipNumber, strengthMat, octave );

allInterestPoints = [ allInterestPoints; interestPoints ];

descriptors = MOPS( image, interestPoints, octave, hessians );

allDescriptors = [ allDescriptors; descriptors ];

end

mapping = interestPointMatching( allDescriptors,

ipNumber,

ipPerTransform,

corrThreshold );

imList = scaledImagesArray( octave + 1, : );

outIm = stitching(imList, mapping, allInterestPoints);

imshow(outIm);

outImages{octave + 1, 1} = outIm;

end

end

**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)

[~, 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,1);

gn = 6;

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

pyramid{1} = image;

for i = 1:(n-1)

scale = 1 / 2^i;

newIm = imresize(image, scale);

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, octave )

patchSize = uint8( 40 / 2^octave );

% Remove interest points that are within patch size of the image edge

halfPatch = ceil( (1/2)\*patchSize );

topStartRow = 1;

topEndRow = halfPatch + 1;

bottomStartRow = size( R, 1 ) - halfPatch - 1;

bottomEndRow = size( R, 1 );

frontStartCol = 1;

frontEndCol = halfPatch + 1;

backStartCol =size( R, 2 ) - halfPatch - 1;

backEndCol = size( R, 2 );

R(topStartRow:topEndRow, : ) = -10000;

R(bottomStartRow:bottomEndRow, : ) = -10000;

R(:, frontStartCol:frontEndCol) = -10000;

R(:, backStartCol:backEndCol) = -10000;

sortedValues = sort( R(:) );

maxValues = sortedValues( end-150:end );

maxIndex = ismember( R, maxValues );

index = find( maxIndex );

[x y] = ind2sub( size(R), index );

idx = sub2ind( [size(R, 1) size(R, 2) ], y, x );

v = R( idx );

newRow = y;

newCol = x;

newVal = v;

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

for i = 2: size( newRow, 1 )

rowVecDist = distMatrix( i, : );

strength = sortedVal( i );

upperIndex = find( sortedVal > strength, 1, 'last' );

potentialVec = rowVecDist( 1:upperIndex );

radius = min( potentialVec );

pointRadii( i, 1 ) = 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 )];

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 = [];

patchSize = uint8( 40 / 2^octave );

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

halfPatch = ceil( (1/2)\*patchSize );

startRow = y - halfPatch;

endRow = y + halfPatch;

startCol = x - halfPatch;

endCol = x + halfPatch;

% 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 = im2double(imresize(patch, [8 8]));

% 3) normalize

avg = mean2(patch);

stdDev = std2(patch);

patch = patch - avg;

if stdDev ~= 0

patch = patch ./ stdDev;

else

patch(1,1) = 1;

end

% save in descriptors

descriptors = [ descriptors; 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)

%get the size of the descriptors and the size of each patch

[sizex, sizePatch] = size(descriptors);

%calculates the number of descriptors exist

numDesc = sizex/sizePatch;

%calculates the number of images we are looking at

numIms = numDesc/numPerIm;

%zeros out a square matrix that we will store the correlations in

corrMat = zeros(numDesc);

%for each descriptor

for i=1:numDesc

index = (sizePatch\*(i-1))+1; %calculate the starting index

curDescriptor = descriptors(index:(index+sizePatch-1),1:sizePatch);

%get the correlation

xcorrMatResult = normxcorr2(curDescriptor, descriptors);

%get the image index that we are working on

curImDesc = floor(i / numPerIm) + 1;

%store results

for j=1:numDesc

%if the descriptors are for the same image ignore them

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

%set the vert correlation found

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

corrMat(j,i) = corrMat(i,j); %set the horizontal

else

%ignore the descriptors from the same image

corrMat(i,j) = -1;

end

end

end

%Find all image mappings

%First and second index of image.

%Third and forth index of interest point

mappedIndexes = [];

%for each image set

for fromIm = 1:numIms

%calc the start of the im range in columns

colIdx = numPerIm \* (fromIm-1) + 1;

%grab all the columns of that image and zero out anything

%less than the threshold

curCol = corrMat(:,colIdx:(colIdx+numPerIm-1)) .\*

(corrMat(:,colIdx:(colIdx+numPerIm-1)) >= corrThresh);

%from the fromImage onward

for toIm = (fromIm+1):numIms

%calc the start of the im range in the row

rowIdx = numPerIm \* (toIm-1) + 1;

curRow = curCol(rowIdx:(rowIdx+numPerIm-1),:);

%get columns indexs of fromIm with maxes in columns

[colMaxes, colIndexes] = max(curRow, [],2);

mat = zeros(numPerIm, numPerIm); %create storage

indexes = sub2ind([numPerIm, numPerIm],

(1:numPerIm)', colIndexes);

%set the rows to their maxes with zeros elsewhere

mat(indexes) = colMaxes;

[rowMax, rowIndexes] = max(mat, [], 1); %get the max of the rows

%sort the maxes and get their indices

[selectedMaxes,rowOrigIndexes] = sort(rowMax, 'descend');

numFound = find(selectedMaxes); %count the maxes

%if the maxes are more than the amount we are looking for

if (size(numFound,2) >= ipToFindPerIm)

insertVal(1:ipToFindPerIm,1) = fromIm; %store from im index

insertVal(1:ipToFindPerIm,2) = toIm;%store to im index

insertVal(1:ipToFindPerIm,3) =

rowOrigIndexes(1:ipToFindPerIm);

insertVal(1:ipToFindPerIm,4) =

rowIndexes(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)

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

[n,~] = size(corrVec);

mat = [];

vec = corrVec(:,3:4);

newVec = size(2\*n,1);

for i=1:n

x1 = corrVec(i,1);

y1 = corrVec(i,2);

index = (2\*i)-1;

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

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

index = 2\*(i-1);

newVec(index+1) = vec(i,1);

newVec(index+2) = vec(i,2);

end

newVec = newVec';

aVec = mat\newVec;

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

transform = reshape(transformVec, size(transformVec,1)/2,

size(transformVec,1)/2,1);

translationVec = [-1\*aVec(size(aVec,1)-1);aVec(size(aVec,1),:)];

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

**Image Mapping**

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

%

% IMAGEMAPPING – apply a transformation to an image

%

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

%ARGS:

% mappedIndex – the image to transform

%OUTPUT:

% mapping – a cell array of the images that map together

% maxIdx -

function [mapping, maxIdx]=imageMapping(mappedIndex)

%get the column of the images that are the from images

imCol = mappedIndex(:,1);

maxImNum = max(imCol, [],1);%get the max number for the mapping

mapping = cell(maxImNum);%create the cell array

maxLen = 0;

maxIdx = -1;

%for each image

for i=1:length(mapping)

row = imCol == i; %get the rows for that image

%get the unique values in the row

mapping{i} = unique(mappedIndex(row,2)');

if (size(mappedIndex,2) > maxLen) %get maxes

maxIdx = i;

maxLen = size(mappedIndex,2);

end

end

end

**Stitching**

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

%

% STITCHING – Given a mapping, and interest points, stitch images together

%

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

%ARGS:

% imList – the list of images to stitch

% mapping – the mapping of images that have corresponding interest points

% interestPoints – the interest points

%OUTPUT:

% outIm – the output image

function outIm = stitching(imList, mapping, interestPoints)

%get a list of images the current image is mapped to

[imMapping, baseImIdx] = imageMapping(mapping);

warpedIms = cell(length(imMapping));

visitedList = [];%zeros(1,length(imMapping));

corners = cell(length(imMapping));

%calculates data for stitching

[corners, visitedList, warpedIms] = prepStitch(imList, imMapping,

baseImIdx, mapping,

interestPoints,

visitedList,corners,

eye(3),warpedIms);

%get data for where our corners will be placed (this is used for

% translation

cornersMat = zeros(4,4\*size(visitedList,2));

for i=1:size(visitedList,2)

index = (4\*(i-1))+1;

corners{i}

cornersMat(1:3,index:index+3) = corners{i};

end

%get the max and min of the corners so we can define our bounding box

maxXY = max(cornersMat, [], 2);

minXY = min(cornersMat, [], 2);

%get first image to place in the list

im = imList{visitedList(1)};

[sizex,sizey] = size(im);

%get upper and lower bounds based on corners

lowerBoundY = abs(minXY(1));

upperBoundY = lowerBoundY + maxXY(1);

lowerBoundX = abs(minXY(2));

upperBoundX = lowerBoundX + maxXY(2);

%set base images offset

offsetToBaseImX = 1+lowerBoundX;

offsetToBaseImY = 1+lowerBoundY;

%build out image

outIm = zeros(ceil(upperBoundX), ceil(upperBoundY));

outIm(offsetToBaseImX:offsetToBaseImX+sizex-1,offsetToBaseImY:offsetToBaseImY+sizey-1) = im;

%loop through all the

for imNum = 2:size(visitedList,2)

%get the index of the image we are displaying next

imIdx = visitedList(imNum);

curIm = warpedIms{imIdx}; %get the image

[sizex,sizey] = size(curIm); %get the size

%calculate where the minimum point is and tranlate it to that point

curMinXY = corners{imIdx};

imMinX = offsetToBaseImX+curMinXY(2);

imMinY = offsetToBaseImY+curMinXY(1);

%create a new image

newIm = zeros(ceil(upperBoundX), ceil(upperBoundY));

newIm(imMinX:(imMinX+sizex-1),imMinY:(imMinY+sizey-1)) = curIm;

%blend with other images

outIm = imfuse(outIm, newIm, 'blend');

end

end

**Results**

Given the time constraints we were unable to get this fully working. We still have many small bugs in our program, particularly dealing with handling scaled images as the scale gets small and the original image sizes vary. Of particular difficulty is getting meaningful correspondences from the interest points. We believe this is a result of not analyzing enough interest points in Non-Maximal Area Suppression, but without more time we can not fully explore the issue.

Below are some examples of output we were able to generate for the test\_t dataset.

**Work Distribution**