Lab 4

Beeldverwerken

May 13, 2018

Patrick Spaans (11268476)
Bram Otten (10992456)

But I, Bram, have not done my 50% this time.

Introduction

SIFT is a useful method that can recognize the same object in different pictures, using the keypoints of the objects in the pictures. This allows us to, for example, stabilize videos, since an object that has to be in the center of the picture can be recognized in every frame of the video. By knowing how far the object is off the center, we can transform all the frames with the object off-center into a frame where the object is centered, which gives us a stable video. RANSAC is used to assist the SIFT process, since when comparing the keypoints found in multiple images, mistakes can be made if two keypoints look alike but aren't the same. If the incorrectly matched keypoints are used in the transformation process, the transformation will be (slightly) off. RANSAC is a process that tries to ignore the outlier matches and only uses the matched keypoints that are inliers for the transformation, which leads to better results.

In the following sections, it is explained how SIFT and RANSAC work in detail and how it can be implemented using MATLAB. Answers to specific exercises can be found in the appendix.

Theory

Scale Invariant Feature Transform

By performing the scale invariant feature transform (SIFT) algorithm, features of an image can be detected and represented in a general form. This general form refers to a representation that is not influenced by the scale, rotation, translation, et cetera. of the feature in the image from which it originates. So when the same object is photographed from two angles, performing the SIFT algorithm on both images allows for, in a sense, an 'unrotation' of the objects. In this unrotated form, the objects will match each other (whereas they would not without SIFTing because the different angles would mean completely different pixel values and edge locations relative to other edge locations et cetera).

Features can have varying levels of detail, and for that reasons they are sought at multiple levels of blur of the image. (This is done with Gaussian blurring at varying scales, and large scales means more blur and less detail.) Stacking these versions of the image with varying scales of blur on top of each other, in a third dimension, gives the scale space. The second derivative of certain scales can then be approximated by taking the difference between two Gaussians of a slightly higher and lower scale (DoG or D), which is computationally much cheaper.

Finding initial keypoints (candidates for becoming features) is finding local extrema in the DoG, comparing each sample point to its eight neighbours on its own scale, and nine neighbours for each of the scales (of the DoG) above and below the scale of the sample point. (So there's 26 neighbours for each sample point not at the very smallest or largest scale.)

1% image noise is added to the original image because it experimentally happened to provide the keypoints more accurately. Or that sampling more than 3 scales per octave, which is a subset of the scale space in which a σ doubles, does not improve repeatability of the SIFTing under images that are slightly different. Number of keypoints does increase, but these are less stable and get discarded later in the process.

We may skip over specifics like this from here on, but there are a few more in Lowe's infamous section 3.3, which is that the image size is doubled and some pre-smoothing is applied. (Pre- in relation to making the scale space that's described above.)

So now there are keypoint candidates. These are first made more accurate by fitting to nearby data for better location, scale, and principal curvature ratio. This fitting is done with $D(\hat{\mathbf{x}}) = D + \frac{\partial D^T}{2\partial \mathbf{x}}\hat{\mathbf{x}}$. Keypoint canditates ($\hat{\mathbf{x}}$) are discarded if $|D(\hat{\mathbf{x}})| < 0.03$, which means something like the keypoint not being extreme enough compared to its surroundings in the DoG (D). Lowe calls this 'having low contrast.'

$$\mathbf{H} = \begin{pmatrix} D_{xx} & D_{xy} \\ D_{xy} & D_{yy} \end{pmatrix}$$

This leaves in keypoints on edges, which are not wanted (but the corners are!). On edges, only the derivative in one direction will be high, and thus only one of the components of $\mathbf{H}$ (though it does contain a duplicate). It turns out it's cheap to check for this property because $\mathbf{H}$ is symmetric (and its eigendecomposition is easy).

$$\left(\frac{\text{Tr}(\mathbf{H})}{\text{Det}(\mathbf{H})} = \frac{(\alpha+\beta)^2}{\alpha\beta} = \frac{(r\beta+\beta)^2}{r\beta\beta} = \frac{(r+1)^2}{r}\right) < \left(\frac{(z+1)^2}{z} = 12.1\right)$$

(r denotes the ratio between the largest and smallest eigenvalues, when thresholding/checking like at the end there Lowe calls our z an r too. But it's given a value found experimentally, 10 in this paper, so that's a little confusing.)

Gradient magnitude and orientation for a pixel (x, y) (not entirely clear on which ones, presumably a neighbourhood surrounding a keypoint) are computed compared to each of its two closest neighbours in the x and y directions. Magnitude is Euclidean distance, orientation is angle measured counterclockwise from the positive x-axis. Orientations are grouped into 36 bins, every gradient votes for one of these bins weighted by its magnitude and distance to the keypoint. If there are peaks on this orientation-bin histogram that are within 80% of the highest, they become new keypoints. The highest peak and its two surrounding bin values are interpolated for the orientation and magnitude of this keypoint.

The gradient magnitude and orientation for (in this paper) 16×16 points surrounding the keypoint are now sampled (from the scale closest to the keypoint's scale) and rotated relative to the keypoint. Every gradient is weighted according to a Gaussian with $\sigma = 8$, and put into another histogram per 4×4 sub-array. Values of this histogram are trilinearly interpolated, so that we're left with smoother magnitudes per orientation. So there's a histogram for each of the 4×4 sub-arrays of the 16×16 array surrounding a keypoint.

That means we have 4×4 orientation/magnitude histograms. These have 8 orientation bins, for a 4×4×8=128 sized feature vector for each keypoint. This vector is then normalized to make it contrast invariant. After that, values >0.2 are set to 0.2 and the vector is normalized again. This is another one of those experimentally found values, apparently to overcome camera saturation artifacts or or uneven lighting of 3D objects.

The rest of section 6 is about testing the descriptor. More orientation bins means features can be made more invariant to orientation changes, but more calculations. Et cetera. It's basically saying the parameters he chose are on a good position on that trade-off of accuracy and computation.

The paper is mostly about deriving those 'distinctive invariant keypoints,' but one obvious application is object recognition. We have the descriptors of keypoints in 128-sized feature vectors (descriptors), and can put all of them in a database. Matching a new keypoint to one in the database, then, can be done by simply choosing the database keypoint at the minimum Euclidean distance from the new one.

But this means every new descriptor matches one in the database, which should not be true. A threshold doesn't work because the discriminative value of some descriptors is greater than that of others. So a better alternative is to compare the distance between a descriptor and its closest and second-closest neighbour in the database. Lowe wants a distance ratio of <0.8 before calling two descriptors a match.

But coming up with distances of a new descriptor to every one in a database is computationally expensive. So a best-bin-first algorithm is used [and not explained.]

RANSAC

The purpose of the random sample consensus algorithm (RANSAC) is to find out how well a model fits data that contains outliers. Without such a method, outliers can have a really big impact on whatever metric is used to determine fit, and a model that works well enough is judged to be bad. An example of outliers in our context is occlusion of part of the object. This will provide keypoints that don't match at all and contribute a massive total squared error, while these non-matching keypoints are fine and even expected in real situations.

While there's a MATLAB implementation in the following section, here's a little more explanation. A model (can be a polynomial or something but we'll go with a line here) is fit to m points taken from the data set at random. Then, for the entire dataset, determine the amount of points that 'fits' this line (within a ϵ tolerance). If enough points fit this line, a new but pretty similar line is fit to all the inliers (points within ϵ from line), and all the outliers are discarded. If there's not enough points fitting this line, we start over by picking m random points again.

So there's a lot of parameters going into RANSAC: the model, m, ϵ, that threshold, and the maximum amount of times we try. These depend mostly on the wanted probability of having a sample free of outliers (p) and the probability of one point being an outlier (v). And then we can rewrite our number of wanted iterations as $N = \frac{\log(1-p)}{\log(1-(1-v)^m)}$, but that still leaves m and v (which in turn depends on ϵ) to be divined from the data some other way.

To get the value of N using this formula, we needed to get values of m and v. We set m equal to 8 (which is the 4 points the projection matrix is based on + another 4 points to make sure that there are inliers that aren't part of the original 4) and v being equal to 0.175. v was calculated backwards, by finding one of the best possible projection matrices and dividing the number of inliers by the total number of matches. In this case, there were 33 inliers out of the 40 matches, which equals to an outlier ratio of 0.175. Filling these values in returned the number 16.1946, which means that 17 iterations are required to find an optimal solution 98% of the time.

Algorithm

SIFT and RANSAC can be used in MATLAB using the following three sections of code:

Demo_mosaic.m

edited Demo_mosaic.m, that finds its own points and can create a projection matrix even if there are more than 4 points found.

```
% script to demonstrate image mosaic
% by handpicking 4 matching points
% in the order topleft - topright - bottomright - bottomleft
f1 = imread('nachtwacht1.jpg');
f2 = imread('nachtwacht2.jpg');

% Uses vl_feat to calculate the features and their descriptor sets, which
% are then used to find the matching keypoints.
[F1, D1] = vl_sift(image1);
[F2, D2] = vl_sift(image2);
matches = vl_ubcmatch(D1, D2);

% Uses ransac and the SVD trick to find the least squares solution,
% which allows more points.
iterations = 17;
threshDist = 1;
inlierRatio = 8/size(matches, 2);
projm = ransacProjectionMatrix(F1, F2, matches, iterations, threshDist, inlierRatio)

T = maketform('projective', projm');
[x y] = tformfwd(T,[1 size(f1,2)], [1 size(f1,1)]);

xdata = [min(1,x(1)) max(size(f2,2),x(2))];
ydata = [min(1,y(1)) max(size(f2,1),y(2))];
f12 = imtransform(f1,T,'Xdata',xdata,'YData',ydata);
f22 = imtransform(f2, maketform('affine', [1 0 0; 0 1 0; 0 0 1]), 'Xdata',xdata,'YData',ydata);
subplot(1,1,1);
imshow(max(f12,f22));
```

ransacProjectionMatrix.m

```
function bestprojm = ransacProjectionMatrix(F1, F2, matches, iterations, threshDist, inlierRatio)

% Keeps track of the biggest set of inliers and it's count.
bestinliers = [];
maxinliers = 0;

% The amount of matches which we later have to loop over
max = size(matches, 2);

for i = 1:iterations
    % Picks 4 random matches from the set of matches
    ranmatch = randperm(max, 4);

    % Finds the matching points for both images and stores them in xy and
    % uv. Then, create a projection matrix using those.
    count = 0;
    xy = [];
    uv = [];
    for j = ranmatch
        count = count+1;
        inliers(:, count) = matches(:, j);
        xy(:, count) = F1(1:2, matches(1, j));
        uv(:, count) = F2(1:2, matches(2, j));
    end
    projm = estimateProjectionMatrix(xy', uv');

    % For all other matches, if the distance between the transformed point
    % of the left image and the actual point of the right image is smaller
    % than the threshold, add it to the current set of inliers.
    for j = 1:max
        if not (ismember(j, ranmatch))
            Coords1 = projm * [F1(1:2, matches(1, j))', 1]';
            Coords1 = Coords1(1:2) / Coords1(3);
            Coords2 = F2(1:2, matches(2, j))';
            diff = sqrt((Coords1(1)-Coords2(1))^2 + (Coords1(2)-Coords2(2))^2);
            if (diff < threshDist)
                count = count + 1;
                inliers(:, count) = matches(:, j);
            end
        end
    end

    % If the found set of inliers is bigger than the current best and has
    % more inliers than the minimum requirement, make it the new best.
    if ((count > max * inlierRatio) && count > maxinliers)
        bestinliers = inliers;
        maxinliers = count;
    end
end

if (maxinliers > 0)
    % If there is a best set of inliers, use those to calculate the
    % projection matrix.
    count = 0;
    xy = [];
    uv = [];
    for i = 1:maxinliers
        count = count+1;
        xy(:, count) = F1(1:2, bestinliers(1, i));
        uv(:, count) = F2(1:2, bestinliers(2, i));
    end
    bestprojm = estimateProjectionMatrix(xy', uv');
else
    % If there is no best set of inliers, return 0.
    bestprojm = 0;
end
```

estimateProjectionMatrix.m

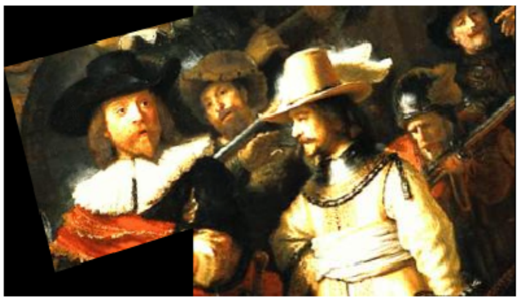
```
function projEst = estimateProjectionMatrix(xy, uv)

% Calculation of projection matrix.
x = xy(:, 1);
y = xy(:, 2);
u = uv(:, 1);
v = uv(:, 2);
o = ones(size(x));
z = zeros(size(x));
Aoddrows = [x, y, o, z, z, z, -u.*x, -u.*y, -u];
Aevenrows = [z, z, z, x, y, o, -v.*x, -v.*y, -v];
projMatrix = [Aoddrows; Aevenrows];
```

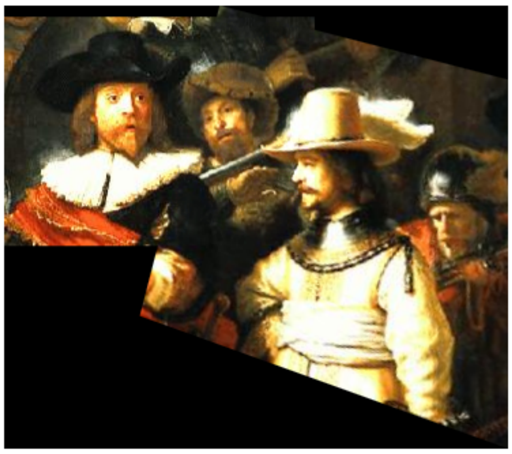
```
% Uses the SVD trick to get the estimated projection matrix.
[u, s, v] = svd(projMatrix);
Mvec = v(:, end);
projEst = reshape(Mvec, 3, 3)';
```

Experiments

By running the edited version of demo_mosaic.m, which inserts Nachtwacht1.jpg into Nachtwacht2.jpg, we obtain the following result:



We can also reverse the images, which inserts Nachtwacht2.jpg into Nachtwacht1.jpg, which gives us the following result:



These experiments are done using the following parameters:

iterations = 17

e = 1

m = 8/size(matches)

Conclusions

SIFT is a powerful tool for image processing and in combination with RANSAC, it works even better. I is able to recognize two halves of a photo, and is able to merge them together even if the image is taken from another angle. In case of the experiments mentioned above, the images are combined perfectly, and no visible border in the merged image can be seen. Misidentifying two keypoints as the same keypoint is the biggest problem that SIFT has and RANSAC solves this issue, which allows SIFT to be really useful in cases where a transformation is needed between two related images.

Appendix

3.3

The scale-space representation of an image is made up of Gaussian convolutions applied to the image. It is of a higher dimensionality: every scale-space representation has it's own z-index (which is the dimension of the image + 1, so for a 2D image it is the 3D value).

If we compare a point to its neighbours (in both its own and neighbouring layers), and it turns out to be a minimum or maximum point, it is a scale space extremum.

3.4

$\frac{\partial G}{\partial \sigma} = \sigma \nabla^2 G \approx \frac{G(x,y,k\sigma)-G(x,y,\sigma)}{k\sigma-\sigma}$
$D(x, y, \sigma) = (G(x, y, k\sigma) - G(x, y, \sigma)) * I(x, y)$
$G(x, y, k\sigma) - G(x, y, \sigma) = \frac{G(x,y,k\sigma)-G(x,y,\sigma)}{k\sigma-\sigma}(k\sigma - \sigma) = (k\sigma - \sigma)\sigma\nabla^2 G$
$(k\sigma - \sigma)$ can be rewritten into $(k - 1)\sigma$, which leads to
$(k - 1)\sigma\sigma\nabla^2 G = (k - 1)\sigma^2\nabla^2 G$
This means that $(G(x, y, k\sigma) - G(x, y, \sigma))$ is equal to $(k - 1)\sigma^2\nabla^2 G$, out of which we can conclude that $(k - 1)\sigma^2\nabla^2 G$ is the convolution filter for D.

3.5

$$\mathbf{H} = \begin{pmatrix} D_{xx} & D_{xy} \\ D_{xy} & D_{yy} \end{pmatrix}$$

Want eigenvalues λ_i.

$$\mathbf{H} - \lambda \mathbf{I}_{2\times 2} = \begin{pmatrix} D_{xx} - \lambda & D_{xy} \\ D_{xy} & D_{yy} - \lambda \end{pmatrix} = 0$$

$$\det(\begin{pmatrix} D_{xx} - \lambda & D_{xy} \\ D_{xy} & D_{yy} - \lambda \end{pmatrix}) = (D_{xx} - \lambda)(D_{yy} - \lambda) - (D_{xy})(D_{xy}) = 0$$

$$\lambda^2 - (D_{xx}\ D_{yy})\lambda + D_{xx}\ D_{yy} - (D_{xy})^2 = 0$$

$$\lambda = \frac{(D_{xx}\ D_{yy}) \pm \sqrt{(D_{xx}\ D_{yy})^2 - 4\ D_{xx}\ D_{yy} - 4\ (D_{xy})^2)}}{2}$$

$$\lambda = D_{xx} \vee \lambda = D_{yy}$$

$$\mathrm{Tr}(\begin{pmatrix} D_{xx} & D_{xy} \\ D_{xy} & D_{yy} \end{pmatrix}) = D_{xx} + D_{yy}$$

This can't always be done, sometimes the values in the Hessian will be such that there can be no eigendecomposition. And/or the abc-formula won't return real values when the discriminant is negative.

3.6

The following magic numbers are found in the implementation:

- On page 8, he mentions that 1% noise is added to the picture, which he doesn't explain. Later on, different amounts of noise are added to other pictures.
- $|D(\hat{x})| < 0.03$ is discarded (Page 11). 0.03 is never explained.
- The experiments use a value of $r = 10$ (Page 12). Why he chose 10 is never explained.
- 36 bins in the orientation histogram (Page 13). Why he chose 36 is never explained.
- $\sigma = 1.5 \times$ the weight of the keypoint. Why he chose 1.5 is never explained.

- Local peaks within 80% are used to create other keypoints (Page 13). Why he chose for 80% is never explained.
- On page 23, he says 'We accept a model if the final probability for a correct interpretation is greater than 0.98'. Why this value was chosen is never stated.

The value of r, the curvature ratio between the keypoints (the distance threshold), is in lowe's article stated as 10, while in our implementation we used a value of 1, because that gave us enough data and allowed us for a really precise transformation.

3.8

[It's a symmetric matrix, so we can rewrite something as something else.]

3.9

[Maybe one dimension is 16×16 and the other is 4×4.]

3.10

Affine transformations are transformations on the complete image, not just parts of it. An affine illumination transformation causes the intensity of all the pixelvalues in the image to change, but since this transformation is done on all the pixels, all pixel values change proportionally. While the image changes slightly as a result of this, the contrast differences between the pixels themself remain the same, so an affine illumination transformation changes the picture but has no impact on the SIFT process.

Matlab: 2 - Projectivity (Demo_mosaic.m)

This is the demo_mosaic.m after the first question, where you still have to pick points, but the projection matrix calculation allows for more than 4 points.

```
% script to demonstrate image mosaic
% by handpicking 4 matching points
% in the order topleft - topright - bottomright - bottomleft
f1 = imread('nachtwacht1.jpg');
f2 = imread('nachtwacht2.jpg');

[xy, xaya] = pickmatchingpoints(f1, f2, 4, 1);

% Uses the SVD trick to find the least squares solution, which allows more
% points.
projm = estimateProjectionMatrix(xy', xaya');
T = maketform('projective', projm');
[x y] = tformfwd(T,[1 size(f1,2)], [1 size(f1,1)]);


xdata = [min(1,x(1)) max(size(f2,2),x(2))];
ydata = [min(1,y(1)) max(size(f2,1),y(2))];
f12 = imtransform(f1,T,'Xdata',xdata,'YData',ydata);
f22 = imtransform(f2, maketform('affine', [1 0 0; 0 1 0; 0 0 1]), 'Xdata',xdata,'YData',ydata);
subplot(1,1,1);
imshow(max(f12,f22));
```

Matlab: 3 - Sift (vl_feat and keypoint transformation)

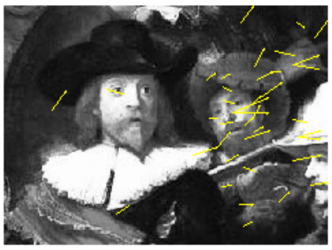
Matches for the Nachtwacht images using vl_feat

```
image1 = im2single(rgb2gray(imread('nachtwacht1.jpg')));
image2 = im2single(rgb2gray(imread('nachtwacht2.jpg')));
[F1, D1] = vl_sift(image1);
[F2, D2] = vl_sift(image2);
matches = vl_ubcmatch(D1, D2);
imshow(image1);
```

```
hold on ;
for match = matches(1, :)
    plot(F1(1, match), F1(2, match), 'r*')
end
hold off;
imshow(image2);

hold on ;
for match = matches(2, :)
    plot(F2(1, match), F2(2, match), 'r*')
end
hold off;
```

This code finds the matches and plots all matched keypoints in their respective image, which results in the following to images:



The yellow lines are the vectors of the keypoints, but they are hard to see because they are pretty small.

Estimating the projection matrix based on 4 matches and using those to calculate the new position of the other matches:

```
for i = 1:4
    xy(:, i) = F1(1:2, matches(1, i));
    uv(:, i) = F2(1:2, matches(2, i));
end

projm = estimateProjectionMatrix(xy', uv')
for i = 5:size(matches, 2)
    Coords1 = projm * [F1(1:2, matches(1, i))', 1]';
    Coords1 = Coords1(1:2) / Coords1(3);
    Coords2 = F2(1:2, matches(2, i))';
    diff(i-4) = sqrt((Coords1(1)-Coords2(1))^2 + (Coords1(2)-Coords2(2))^2);
end
```

This code creates the projection matrix based on the first four matches, then transforms the other 36 matches and calculates the eucledian distance between the calculated location and the actual location, which is then stored in a 1x36 vector that you can see below.

[203.4991, 41.8121, 124.2735, 48.6038, 34.4998, 100.4002, 62.0497, 5.2679, 8.4812, 21.7931, 42.7667, 81.3599, 24.6984, 123.5694, 12.7305, 72.8346, 236.2583, 167.8929, 56.6777, 67.4227, 198.8317, 32.2516, 26.0293, 40.7196, 95.5176, 162.0425, 31.7229, 66.4258, 66.4258, 162.0728, 53.6754, 48.5857, 48.5857, 75.7153, 32.5434, 32.5434]

The projection matrix isn't correct, and is not even close to being correct. Almost every transformed coordinate is pretty far off its actual location, and there is no (nearly) correct transformated point.

If the match was a bad match, the coordinates will not match if the projection matrix is correct. projection_matrix * point_1 with a correct projection_matrix can only return return point_2 if they are actual matches, so if they are bad matches, the result of this projection will not be the same as the expected result.

If the projection matrix is based on a bad match, the projection matrix itself will be wrong, and all projections made with this matrix will be incorrect. This is because the computer doesn't know that the match is wrong, and will use that match as a basis to create the projection matrix. By using the projection matrix on the original 4 points out of which the projection matrix was created, the results will be correct, but for nearly all other points, this won't be the case.