1 Image convolution

Firstly, the kernel is rotated 180° and the image is padded with zeros on all sides before sliding the kernel horizontally across the image and calculating the sum of the element-wise product of the flipped kernel and image. This is expressed as: $(I * f)(x, y) = \sum_k \sum_l I(k, l) f(x - k, y - l)$. Results of this function were shown to match the output of MATLAB's built-in `conv2` function (`convolutionTestScript.m`). A range of images from the training dataset were convolved with a range of kernels using blurs of different kernel sizes and σ values, and edge detectors in different directions. See `convolution.m` for details. A larger σ in a Gaussian kernel produced a smoother blur. Applying Sobel high pass filters displayed edges. Combining filters allowed effects such as sharpening due to the distribution property of $*$.

2 Intensity-based template matching

Initially, a Gaussian Pyramid (GP) was created by applying a Gaussian filter $G(x, y, \sigma)$ to each training image class at a variety of rotations before sub-sampling by a factor of 2 across a series of octaves, to generate templates. Each resulting blurred image can be expressed as $L(x, y, \sigma) = G(x, y, \sigma) * I(x, y)$. Intensity-based matching (IBM) necessitated computing the correlation between templates and test images. The template yielding the maximum correlation across the test image was determined for each training image class before a non-maxima suppression (NMS) strategy was applied to ensure detection occurred at sensible correlation values. NMS involved suppressing classes of templates with a bounding box overlapping that of the template with the maximum correlation score.

A computationally-expensive approach slides the patch across the entire test image and computes the correlation at each point (x, y) using the formula: $cor(x, y) = \sum_{i,j} T(i, j) I(x + i, y + j)$. A more efficient approach involves using the Fast Fourier Transform (FFT) to transform the image signal into the frequency domain for faster processing. Multiplication in the Fourier domain is equivalent to convolution in the real domain ($f * g = \mathcal{F}^{-1}\{\mathcal{F}\{f\} \cdot \mathcal{F}\{g\}\}$), so the correlation result can also be efficiently determined by taking the inverse FT of the product of the FT of the image and template. MATLAB's `normxcorr2` function was used as it utilises this property. This was applied separately to each RGB channel before averaging the result. See `runIntensityMatching.m` for details.

Table 1 shows the optimum results of running IBM detection on a test image with NMS applied and a 5×5 Gaussian kernel with $\sigma = 1, 2, 4, 8$ across 5 octaves (chosen as the smallest image across all test images was a factor of 2^{-4} of the original size). Each octave's image had 12 rotations applied before subsampling which was a pragmatic compromise as the average runtime with this number was already 1039.1s. IBM (Fig 1) finds difficulty where icons in test images were directly in-between multiples of the rotation or size of the templates in the GP as pixels do not line up as optimally. IBM's runtime complexity is $O(ijknm \log nm)$ for i rotations, j octaves, k training images where n is the image height and m is the image width. Memory complexity is therefore $O(ijknm)$ to store all images in the GP. IBM generated templates invariant to rotation and scale, but not to illumination. It is also heavily dependent on the quantity of unique rotations and scales used in the pyramid. As such, Task 3 used SIFT to generate *features* for more robust template matching.

TP Rate	FP Rate	ACC	Avg Runtime
56.7%	8.9%	87.0%	1039.1s

Table 1: TP, FP, ACC Results and Runtime for Task 2.

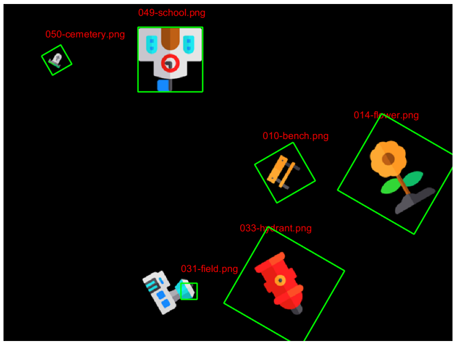

Figure 1: IBM detection on a test image.

3 Feature-based template matching (SIFT)

Lowe's SIFT algorithm [1, 2] was used to identify correspondences between features of the training set and each test image in order to identify the classes of objects present in each test image. Keypoint localisation involved creating a Difference of Gaussian (DoG) pyramid of 4 octaves (levels), with a set of 4 blurs applied, varying σ at each octave to form an approximation for the Laplacian of Gaussians (LoG) (i.e. $\nabla^2 g_\sigma \approx (g_{\sigma_1} - g_{\sigma_2}) * I$). This allowed for extrema in the (x, y, σ) space to be identified as keypoints by comparing a point (x, y) in the current DoG against all its neighbours in a $3 \times 3 \times 3$ patch across the current, previous and next DoGs. All points (x, y, σ) identified as unique extrema were filtered by discarding points in the DoG with a contrast below a threshold (< 0.12). Then, the Hessian matrix $\mathbf{H}$ was approximated using the image gradient at each keypoint to discard edges (points in the DoG with $r > 10$). Corner values were favoured as they hold more valuable information (Fig 2).

$$\mathbf{H} = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{xy} & D_{yy} \end{bmatrix} \qquad \frac{Tr(\mathbf{H})^2}{Det(\mathbf{H})} = \frac{(r+1)^2}{r}$$

A rotation orientation was then calculated for each keypoint $L(x, y, \sigma)$ equal to the dominant orientation $\theta(x, y)$ of the gradient $m(x, y)$, based on a histogram of 36 bins in which the orientations of values in a surrounding 15×15 window were stored. Values were weighted by the magnitude of the gradient at each keypoint and a Gaussian-weighted circular window with a scale of 1.5σ. This characterised keypoints as (x, y, σ, θ) to provide rotation invariance by rotating keypoints by this value. Histogram peaks $> 80\%$ of the highest peak were converted into new keypoints (x, y, σ, θ'). Finally, a SIFT feature (128 dimensional vector) was generated for each keypoint using a 16×16 window split into 4×4 cells with each cell being represented by an 8 bin histogram of gradient magnitude and orientation (Fig 3). Each bin entry was weighted by a value $w = 1 - d$ where d represents the distance of the value from the central value, normalised by the width of the bin. The window also had a Gaussian weighting applied. The resulting 16 histograms formed the SIFT descriptor. After features are generated for training and test images, a suitable correspondence between two feature vectors $\phi^{(1)}, \phi^{(2)}$ was found by an SSD matching function. This was calculated between all the features in each training image class against all the features in the test image. Matches were refined by filtering with an empirically-tested SSD threshold of 0.3 and nearest-neighbour ratio (NNR) to discard weak matches (where $R > 0.8$).

$$SSD = \sum_{i=1}^{128} (\phi_i^{(1)} - \phi_i^{(2)})^2 \qquad R = \frac{SSD(\phi^{(1)}, \phi^{(2)})}{SSD(\phi^{(1)}, \phi'^{(2)})}$$

For SIFT object recognition in a given test image (Fig 4), the summation of the SSDs of all feature matches (after thresholding and NNR) was calculated for each training image class. This value was normalised by dividing by the number of total matches detected between the training image in question and the test image. The training image class yielding the minimum normalised SSD summation was selected as the most suitable class. Matches containing features in the test image and the most suitable training image class were removed from the list of all matches before the process was repeated for each of the remaining objects in the test image. See `runSIFT.m` for details.

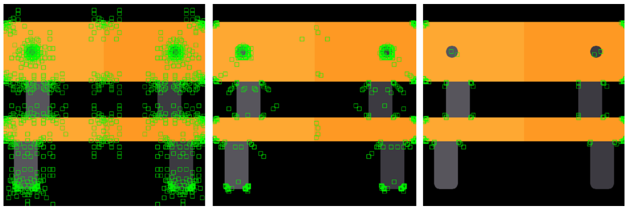

Figure 2: All keypoints (left), post-edge removal (mid) and post-contrast thresholding and edge removal (right).

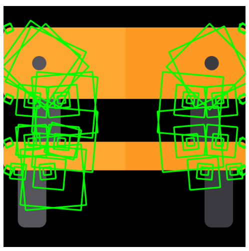

Figure 3: SIFT features extracted from a training image.

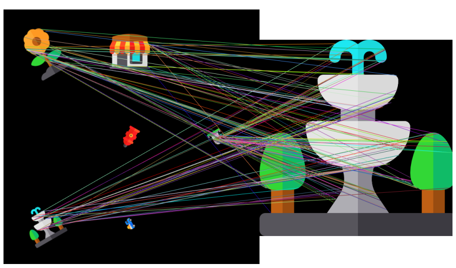

Figure 4: SIFT matching between a test (left) and training (right) image.

Table 2 shows the optimum results of running SIFT detection on a test image. This was produced using a 5×5 Gaussian kernel with $\sigma = 0.4$ initially and increasing by a factor of 2.5 per octave across 5 octaves with the SSD threshold and R values described previously. This parameter choice was justified by empirical results (Fig 5), which was inspired by the empirical graphs of Lowe [2]. The implementation of edge detection was not optimal, which meant the SIFT detector found difficulty distinguishing icons with similar edge features such as the Bank and Courthouse. To improve this implementation, subpixel localisation using the Taylor series would increase accuracy of keypoints, and Hough Transform voting could assist with cluster identification [2]. SIFT ran in real-time and could perform detection 10 times faster (≈ 97.3s) than IBM. SIFT's runtime complexity was $O(n_1 n_2 + m)$ for n_1 training features, n_2 test features and m matches as the complexity of SSD operations is constant due to the fixed feature vector size and the number of matches is what determines the remaining processing. In other words, SIFT detection's runtime complexity is independent of image size (unlike IBM) and dependent on the number of features and matches between training and test images instead. Memory complexity is dominated by the number of feature vectors and is therefore $O(n_1 + n_2)$, making SIFT more scalable than IBM when larger input images are considered.

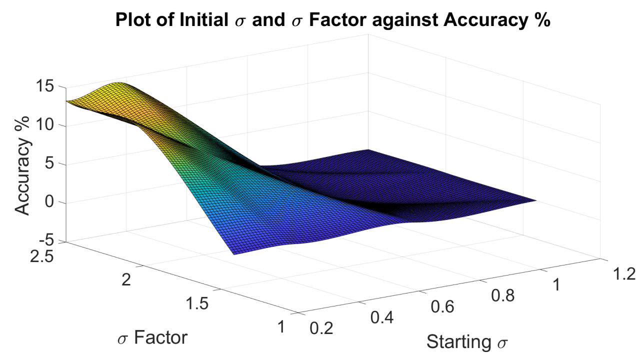

Figure 5: Initial σ and σ factor against accuracy %.

ACC	Avg Runtime
14.2%	97.3s

Table 2: ACC Results and Runtime for Task 3.

References

[1] D. G. Lowe. Object recognition from local scale-invariant features. In *Proceedings of the Seventh IEEE International Conference on Computer Vision*, volume 2, pages 1150–1157, Sep. 1999. doi: 10.1109/ICCV.1999.790410.

[2] David G. Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision*, 60(2):91–110, November 2004. ISSN 0920-5691. doi: 10.1023/B:VISI.0000029664.99615.94. URL `https://doi.org/10.1023/B:VISI.0000029664.99615.94`.

Contributions

Name	Username	Contribution	Contribution %
Christopher Davies	cjd47	Code and report	50%
James Armitstead	ja663	Code and report	50%

A Code

calcGradientAngle.m

```
function [angle] = calcGradientAngle(input)
% This function calcualtes the angle betwewen the poitive x axis and the
% graident of an image patch

fy = input(3, 2) - input(1, 2);
fx = input(2, 3) - input(2, 1);

% Check which quadrant the gradient vector lies in and adjust the value
% produces from atan so that the angles re in the range 0-360 degrees
if fx > 0 && fy > 0
    angle = 90 - radtodeg(atan( fx/fy ));
elseif fx > 0 && fy < 0
    angle = 270 + abs(radtodeg(atan( fx/fy )));
elseif fx < 0 && fy > 0
    angle = abs(radtodeg(atan( fx/fy ))) + 90;
elseif fx < 0 && fy < 0
    angle = 270 - radtodeg(atan( fx/fy ));
end

% Tan is not defined well at 90, 180, 270 and 360 degress so this is
% calculated manually
if fx == 0 && fy > 0
    angle = 90;
elseif fx == 0 && fy < 0
    angle = 270;
elseif fy == 0 && fx > 0
    angle = 0;
elseif fy == 0 && fx < 0
    angle = 180;
end

% If the gradient is constant across the patch then the angle is set to -1
% so that this feature can be discarded
if fx == 0 && fy == 0
    angle = -1;
end

if angle == 360
    angle = 0;
end
```

```
end
```

convolution.m

```
function [resultIMG] = convolution(image, kernel, pad, sameSize)

% This function performs the 2D correlation between a kernal and image.

kernelSize = size(kernel);

% Calculate how much to pad the image
padSize = kernelSize - 1;
resultDepth = size(image, 3);

% Rotate the kernel 180 degrees
kernel = rot90(rot90(kernel));
kernel = repmat(kernel, 1, 1, resultDepth);

origHeight = size(image, 1);
origWidth = size(image, 2);

resultHeight = origHeight + padSize(1);
resultWidth = origWidth + padSize(2);

% Pad the image
image = padarray(image, padSize, pad, 'both');
resultIMG = zeros(resultHeight, resultWidth, resultDepth);

% Loop through every pixel and compute thesum of the elementwise multiplication
% between surrounding pixels and the kernel
for i = 1: resultHeight
    for j = 1: resultWidth
        submatrix = image(i: i + kernelSize(1) - 1, j: j + kernelSize(2) - 1, :);
        calculatedmatrix = submatrix .* kernel;
        resultIMG(i, j, :) = sum(sum(calculatedmatrix, 1), 2);
    end
end

% Crop image if the same size parameter is true
if sameSize == true
    startHeight = 1 + idivide(int32(kernelSize(1)), 2);
    startWidth = 1 + idivide(int32(kernelSize(2)), 2);
    endHeight = startHeight + origHeight - 1;
```

```
    endWidth = startWidth + origWidth - 1;
    resultIMG = resultIMG(startHeight : endHeight, startWidth : endWidth, :);
end

end
```

doesIntersect.m

```
function [result] = doesIntersect(boundingBox1,boundingBox2)
% This function checks if bounding box 1 intersects with bounding box 2


minX = boundingBox1(1);
minY = boundingBox1(2);
maxX = minX + boundingBox1(3);
maxY = minY + boundingBox1(4);

boundingBox2Points = zeros(4,2);
boundingBox2Points(1,:) = boundingBox2(1:2);
boundingBox2Points(2,:) = boundingBox2(1:2) + [boundingBox2(3), 0];
boundingBox2Points(3,:) = boundingBox2(1:2) + [0, boundingBox2(4)];
boundingBox2Points(4,:) = boundingBox2(1:2) + boundingBox2(3:4);

result = false;
for i = 1 : 4
    % Check if each point of bounding box 2 intersects with bounding box 1
    if minX <= boundingBox2Points(i, 1) && boundingBox2Points(i, 1) <= maxX && minY <=
    ↪  boundingBox2Points(i, 2) && boundingBox2Points(i, 2) <= maxY
        result = true;
    end
end

end
```

drawFeatureMatches.m

```
function [] = drawFeatureMatches(leftImage,rightImage,matches)
% This function draws SIFT matches between the test image features and a
% training image's features

% Calulate how much to translate the training image match coordinates due
% to the images being displayed side by side
paddingYTop = (size(leftImage, 1) - size(rightImage, 1) + 1) / 2;
```

```
paddingYBottom = (size(leftImage, 1) - size(rightImage, 1) - 1) / 2;
paddingX = size(leftImage, 2);
for i = 1 : length(matches)
    translatedMatches(i) = matches(i);
    translatedMatches(i).rightImageY = translatedMatches(i).rightImageY + paddingYTop;
    translatedMatches(i).rightImageX = translatedMatches(i).rightImageX + paddingX;
end

% Pad the training image to be the same height as the test image
paddedImage = padarray(rightImage, [paddingYTop 0], 1, 'pre');
paddedImage = padarray(paddedImage, [paddingYBottom 0], 1, 'post');
compositeImage = [leftImage paddedImage];

figure;
imshow(compositeImage);
hold on;
% Draw each match
for i = 1 : length(translatedMatches)
    line([translatedMatches(i).leftImageX; translatedMatches(i).rightImageX],
     ↪  [translatedMatches(i).leftImageY; translatedMatches(i).rightImageY], 'Color', rand(1,3));
end

hold off;

end
```

drawIntensityMatches.m

```
function [] = drawIntensityMatches(bestMatches,testImage)
% Draws boxes around intensity based matches

figure;
imshow(testImage);
hold on;
for i = 1 : size(bestMatches, 1)

    % y x are the coordinates of the bottom left corner of the box
    y = bestMatches{i, 5}(2);
    x = bestMatches{i, 5}(1);
    rotation = bestMatches{i, 1};
    featureSize = bestMatches{i, 5}(3);
    cornerCoordinates = [x;y];

    % Calculate coordinates for top right of box
```

```
    endCornerCoordinates = cornerCoordinates + featureSize;
    translation = featureSize / 2 + cornerCoordinates;

    % Translate all coordinates so that the box is centred on the origin
    cornerCoordinates = cornerCoordinates - translation;
    endCornerCoordinates = endCornerCoordinates - translation;
    allCornerCoordinates = [cornerCoordinates(1), endCornerCoordinates(1), endCornerCoordinates(1),
    ↪   cornerCoordinates(1), cornerCoordinates(1); cornerCoordinates(2), cornerCoordinates(2),
    ↪   endCornerCoordinates(2), endCornerCoordinates(2), cornerCoordinates(2)];
    rotationMatrix = [cos(deg2rad(rotation)), -sin(deg2rad(rotation)) ; sin(deg2rad(rotation)),
    ↪   cos(deg2rad(rotation))];

    % Rotate box using rotation matrix
    rotatedCoordinates = rotationMatrix * allCornerCoordinates;

    % Translate box back to orginal position
    translatedBackCoordinates = rotatedCoordinates + translation;

    % Draw class names
    text(bestMatches{i, 5}(1), bestMatches{i, 5}(2)-30, bestMatches(i, 2), 'Color',
    ↪   'red','FontSize',14);

    % Plot box
    plot(translatedBackCoordinates(1,:), translatedBackCoordinates(2,:), 'g','LineWidth', 2);
end
hold off;
end
```

drawSIFTDescriptors.m

```
function [] = drawSIFTDescriptors(features, image, drawRotation)
% This function draws rotated and scaled boxes around SIFT features

figure;
imshow(image);
hold on;
for i = 1 : length(features)

        centreY = features(i).y;
        centreX = features(i).x;
        octave = features(i).octaveNumber;
        rotation = features(i).rotation;
        featureSize = power(2, 3 + octave);
```

```
        % Apply any translation needed due to displaying the features on a
        % full sized image
        centreY = centreY * power(2, octave - 1);
        centreX = centreX * power(2, octave - 1);
        centreCoordinates = [centreX;centreY];

        % Move from centre of box to corner
        cornerCoordinates = centreCoordinates - (featureSize / 2 - 1);

        if drawRotation == false
            rectangle('Position', [cornerCoordinates(1) cornerCoordinates(2) featureSize
            ↪  featureSize], 'EdgeColor', 'g');
        else
            % Calculate coordinates of the top right corner of the box
            endCornerCoordinates = cornerCoordinates + featureSize;
            translation = featureSize / 2 + cornerCoordinates;
            % Translate box so that the centre of the box is at the origin
            cornerCoordinates = cornerCoordinates - translation;
            endCornerCoordinates = endCornerCoordinates - translation;
            allCornerCoordinates = [cornerCoordinates(1), endCornerCoordinates(1),
            ↪  endCornerCoordinates(1), cornerCoordinates(1), cornerCoordinates(1);
            ↪  cornerCoordinates(2), cornerCoordinates(2), endCornerCoordinates(2),
            ↪  endCornerCoordinates(2), cornerCoordinates(2)];
            rotationMatrix = [cos(deg2rad(rotation)), -sin(deg2rad(rotation)) ;
            ↪  sin(deg2rad(rotation)), cos(deg2rad(rotation))];
            % Apply rotation to the box coordinates
            rotatedCoordinates = rotationMatrix * allCornerCoordinates;
            % Translate box back to original position
            translatedBackCoordinates = rotatedCoordinates + translation;
            % Plot box
            plot(translatedBackCoordinates(1,:), translatedBackCoordinates(2,:), 'g', 'LineWidth',
            ↪  2);
        end
end
hold off;
end
```

generateMatches.m

```
function [matches] = generateMatches(testImageFeatures, trainImageFeatures, scores,
↪  trainImageNumber)
% This function generates the coordinates of the SIFT test and training features
% that matched
```

```
counter = 1;
for i = 1 : length(scores)
    if scores(i).trainImageNumber == trainImageNumber
        yTest = testImageFeatures(scores(i).bestTestFeatureNumber).y;
        xTest = testImageFeatures(scores(i).bestTestFeatureNumber).x;

        % Apply scaling due to differences in octaves
        octaveTest = testImageFeatures(scores(i).bestTestFeatureNumber).octaveNumber;
        factorTest = 2^(octaveTest - 1);
        yTest = yTest * factorTest;
        xTest = xTest * factorTest;

        % Apply scaling due to differences in octaves
        yTrain = trainImageFeatures{scores(i).trainImageNumber}(scores(i).trainFeatureNumber).y;
        xTrain = trainImageFeatures{scores(i).trainImageNumber}(scores(i).trainFeatureNumber).x;
        octaveTrain = trainImageFeatures{scores(i).trainImageNumber}(scores(i).trainFeatureNumber).
        ↪  octaveNumber;
        factorTrain = 2^(octaveTrain - 1);
        yTrain = yTrain * factorTrain;
        xTrain = xTrain * factorTrain;

        matches(counter).leftImageY = yTest;
        matches(counter).leftImageX = xTest;
        matches(counter).rightImageY = yTrain;
        matches(counter).rightImageX = xTrain;
        counter = counter + 1;
    end
end
end
```

generateResults.m

```
function [finalResult] = generateResults(scores, trainImageFeatures, removeMacthedFeatures)
% This function generates the guesses for SIFT object recognition

finalResult = zeros(6, 1);

% Loop 6 times for each icon in the test image
for i = 1 : 6
    % Check that scores is not empty
    if isfield(scores, 'secondBestSSD') ~= 0 && length(scores) > 1
        sumSSDs = zeros(50, 3);
        % Sum the SSDs for all the scores
        for j = 1 : length(scores)
```

```
            sumSSDs(scores(j).trainImageNumber, 1) = sumSSDs(scores(j).trainImageNumber, 1) +
            ↪  scores(j).bestSSD;
            sumSSDs(scores(j).trainImageNumber, 2) =
            ↪  length(trainImageFeatures{scores(j).trainImageNumber});
            sumSSDs(scores(j).trainImageNumber, 3) = sumSSDs(scores(j).trainImageNumber, 3) + 1;
        end

        % Calculate best result
        results = zeros(50, 4);
        results(:, 1) = (1:50);
        results(:, 2) = sumSSDs(:, 1) ./ sumSSDs(:, 2);
        results(:, 3) = sumSSDs(:, 1) ./ sumSSDs(:, 3);
        results(:, 4) = sumSSDs(:, 2) ./ sumSSDs(:, 3);
        sortedResults = sortrows(results, 3);
        finalResult(i) = sortedResults(1,1);

        if removeMacthedFeatures == true
            counter = 1;
            discardFeatures = zeros(1,1);
            % If a test feature was matched by best training image then remember it
            for j = 1 : length(scores)
                if scores(j).trainImageNumber == finalResult(i)
                    discardFeatures(counter) = scores(j).bestTestFeatureNumber;
                    counter = counter + 1;
                end
            end
            counter = 1;
            newScores = scores(1);
            newScores(:)= [];
            % Remove macthes involving test features that matched to the
            % best training image guess in this iteration
            for j = 1 : length(scores)
                if ismember(scores(j).bestTestFeatureNumber, discardFeatures) == false
                    newScores(counter) = scores(j);
                    counter = counter + 1;
                end
            end
            scores = newScores;
        end

    end
end
end
```

generateScores.m

```
function [scores] = generateScores(trainImageFeatures, testImageFeatures, numTrainImages)
% This function take the test image and training image features and returns
% the matches that were better than the threshold and had ratio of more
% than 0.8 to second best match

trainFeatureCounter = 0;

for i = 1 : numTrainImages
    for j = 1 : length(trainImageFeatures{i})
        trainFeatureCounter = trainFeatureCounter + 1;
        scores(trainFeatureCounter).trainImageNumber = i;
        scores(trainFeatureCounter).trainFeatureNumber = j;
        scores(trainFeatureCounter).bestSSD = inf;
        for k = 1 : length(testImageFeatures)
            if isfield(trainImageFeatures{i}, 'descriptor') ~= 0 && isfield(testImageFeatures,
            ↪ 'descriptor') ~= 0
                % Calcualte the SSD between each test feature and each
                % training feature
                difference = trainImageFeatures{i}(j).descriptor - testImageFeatures(k).descriptor;
                SSD = sum(difference(:).^2);

                % If new lowest SSD is fouind then store it
                if SSD < scores(trainFeatureCounter).bestSSD
                    % Bump previous best SSD into second best SSD slot
                    if isfield(scores(trainFeatureCounter), 'bestTestFeatureNumber') == 0
                        scores(trainFeatureCounter).secondTestFeatureNumber = k;
                        scores(trainFeatureCounter).secondBestSSD = SSD;
                    else
                        scores(trainFeatureCounter).secondTestFeatureNumber =
                        ↪ scores(trainFeatureCounter).bestTestFeatureNumber;
                        scores(trainFeatureCounter).secondBestSSD =
                        ↪ scores(trainFeatureCounter).bestSSD;
                    end
                    scores(trainFeatureCounter).bestTestFeatureNumber = k;
                    scores(trainFeatureCounter).bestSSD = SSD;
                elseif SSD < scores(trainFeatureCounter).secondBestSSD
                    scores(trainFeatureCounter).secondTestFeatureNumber = k;
                    scores(trainFeatureCounter).secondBestSSD = SSD;
                end
            end
        end
    end
```

```
end

end
```

generateTemplates.m

```
function [templateIMGs] = generateTemplates(numTrainImages, trainImageDIR, listTrainImages,
↪  numRotations, rotationStep, numSizes, greyScale)
% This function generates all the rotated and scaled template images for
% intensity based matching.

% Generate gaussian kernels for down sampling
gaussianKernels = cell(numSizes - 1, 1);
for i = 1 : numSizes - 1
    gaussianKernels{i} = fspecial('gaussian', 5, 2^(i-1));
end

for i = 1 : numTrainImages
    % Read in training image
    inputIMG = im2double(imread(strcat(trainImageDIR, listTrainImages(i).name)));

    if greyScale == true
        inputIMG = mean(inputIMG, 3);
    end

    % Rotate training image
    for j = 1 : numRotations
        templateIMGs{j, 1, i} = imrotate(inputIMG, (j - 1) * rotationStep);
    end

    % Scale each rotated training image n times
    for j = 1 : numRotations
        for k = 2 : numSizes
            % Apply gaussian blur and downsample
            templateIMGs{j, k, i} = resizeImage(templateIMGs{j, k - 1, i}, gaussianKernels{k - 1},
            ↪  2);
        end
    end
end
end
```

getAllFeatureDescriptors.m

```
function [features] = getAllFeatureDescriptors(image, startingSigma, sigmaFactor,
↪  contrastThreshold, drawKeyPoints)
% This function gets all the SIFT features for a given image

numBlurs = 4;
numOctaves = 4;

% Generate DoG pyramid and every blurred test image
[DoGs Blurs] = getDoGsAndBlurs(image, numOctaves, numBlurs, startingSigma, sigmaFactor);

% Get extrema from the DoGs
extrema = getExtrema(DoGs, image, startingSigma, sigmaFactor, contrastThreshold, drawKeyPoints);

features = struct();

if isfield(extrema, 'octaveNumber') ~= 0
    numOriginalExtrema = length(extrema);
    newExtremaCounter = 1;

    % Get rotation of each key point and create new key points for key
    % points that have more than one dominant rotation
    for i = 1 : numOriginalExtrema
        [rotation, secondRotation] = getRotationOrientation(extrema(i),
        ↪  Blurs{extrema(i).octaveNumber, extrema(i).blurNumber});
        extrema(i).rotation = rotation;
        if secondRotation ~= -1
            extrema(numOriginalExtrema + newExtremaCounter) = extrema(i);
            extrema(numOriginalExtrema + newExtremaCounter).rotation = secondRotation;
            newExtremaCounter = newExtremaCounter + 1;
        end
    end

    featureCounter = 1;
    % Get the feature descriptor for each extrema/key point
    for i = 1 : length(extrema)
        if extrema(i).y - 8 >= 1 && extrema(i).x - 8 >= 1  && extrema(i).y + 9 <=
        ↪  size(Blurs{extrema(i).octaveNumber, extrema(i).blurNumber}, 1) && extrema(i).x + 9 <=
        ↪  size(Blurs{extrema(i).octaveNumber, extrema(i).blurNumber}, 2)
            subSection = Blurs{extrema(i).octaveNumber, extrema(i).blurNumber}(extrema(i).y - 8 :
            ↪  extrema(i).y + 9, extrema(i).x - 8 : extrema(i).x + 9);
            features(featureCounter).descriptor = getFeatureDescriptor(subSection,
            ↪  extrema(i).rotation);
```

```
            features(featureCounter).y = extrema(i).y;
            features(featureCounter).x = extrema(i).x;
            features(featureCounter).octaveNumber = extrema(i).octaveNumber;
            features(featureCounter).rotation = extrema(i).rotation;
            featureCounter = featureCounter + 1;
        end
    end
end
end
```

getAllMaxCorrelations.m

```
function [maxSet] = getAllMaxCorrelations(numRotations, rotationStep, numSizes, numTrainImages,
↪  listTrainingImages, templateIMGs, testImage)
% This function returns the maximum correlation between each training image
% and the test image along with additional information


maxSet = cell(numTrainImages, 5);
for i = 1 : numTrainImages
    maxSet{i, 4} = -inf;
end

% Calcualte correlation for each template and store the maximum for each
% training class
for i = 1 : numRotations
    for j = 2 : numSizes
        for k = 1 : numTrainImages
            [correlation, position] = maxCorrelation(testImage, templateIMGs{i, j, k});
            if correlation > maxSet{k, 4}
                maxSet{k, 1} = (i - 1) * rotationStep;
                maxSet{k, 2} = listTrainingImages(k).name;
                maxSet{k, 3} = j;
                maxSet{k, 4} = correlation;
                maxSet{k, 5} = position;
            end
        end
    end
end
end
```

getBestMatches.m

```
function [bestMatches] = getBestMatches(sortedResults)
% This function takes the guessed bounding box for each training image
% class and discards any guesses that have bounding boxes that intersect
% with the best guesses

bestMatches = cell(1,5);
bestMatchesCounter = 1;
overlappingMatchesExist = true;

% Loop until only non overlapping bounding box guesses remain
while overlappingMatchesExist == true
    bestMatches(bestMatchesCounter, :) = sortedResults(1, :);
    sortedResults(1, :) = [];
    index = 1;
    maxIndex = size(sortedResults, 1);
    for i = 1 : maxIndex
        % Calcualte if two bounding boxes intersect
        if doesIntersect(bestMatches{bestMatchesCounter, 5}, sortedResults{index, 5}) == true ||
        ↪   doesIntersect(sortedResults{index, 5}, bestMatches{bestMatchesCounter, 5}) == true
            sortedResults(index, :) = [];
        else
            index = index + 1;
        end
    end
    bestMatchesCounter = bestMatchesCounter + 1;
    if size(sortedResults, 1) == 0
        overlappingMatchesExist = false;
    end
end

end
```

getDoGsAndBlurs.m

```
function [DoGs, Blurs] = getDoGsAndBlurs(image, numOctaves, numBlurs, startingSigma, sigmaFactor)
% This function calcualtes the DoGs and Blurs for a given image
Blurs = cell(numOctaves, numBlurs + 1);
DoGs = cell(numOctaves, numBlurs);

for i = 1 : numOctaves
    previousBlurredImage = image;
    Blurs{i, 1} = previousBlurredImage;
```

```
    sigma = startingSigma;
    for j = 1 : numBlurs
        gaussianKernel = fspecial('gaussian', 5, sigma);
        % Blur image
        blurredImage = convn(image, gaussianKernel, 'same');
        Blurs{i, j + 1} = blurredImage;
        % Calulate DoG
        DoGs{i, j} = blurredImage - previousBlurredImage;
        previousBlurredImage = blurredImage;
        sigma = sigma * sigmaFactor;
    end
    % Downsample image to next octave
    image = imresize(image, 0.5);
end
end
```

getExtrema.m

```
function [extrema] = getExtrema(DoGs, image, startingSigma, sigmaFactor, contrastThreshold,
↪  drawKeyPoints)
% This function get all extrema/key points for a given set of DoGs and gets
% rid of low contrast and edge key points
numOctaves = size(DoGs, 1);

allKeyPoints = image;

counter = 1;

% For each DoG loop through every value and slect extrema from a
% neighbourhood of pixels
for i = 1 : numOctaves
    imageSize = size(DoGs{i, 1});
    for j = 2 : imageSize(1) - 1
        for k = 2 : imageSize(2) - 1
            currentMax = max(max(DoGs{i, 2}(j - 1 : j + 1, k - 1 : k + 1)));
            currentMin = min(min(DoGs{i, 2}(j - 1 : j + 1, k - 1 : k + 1)));
            [minCount, maxCount] = minMaxCount(DoGs{i, 2}(j - 1 : j + 1, k - 1 : k + 1));

            % Look for maxima in DoG 2
            if DoGs{i, 2}(j, k) == currentMax && maxCount == 1
                previousMax = max(max(DoGs{i, 1}(j - 1 : j + 1, k - 1 : k + 1)));
                nextMax = max(max(DoGs{i, 3}(j - 1 : j + 1, k - 1 : k + 1)));
                if DoGs{i, 2}(j, k) > previousMax && DoGs{i, 2}(j, k) > nextMax
                    extrema(counter).octaveNumber = i;
```

```
            extrema(counter).y = j;
            extrema(counter).x = k;
            extrema(counter).sigma = startingSigma * sigmaFactor;
            extrema(counter).blurNumber = 2;
            extrema(counter).DoGNumber = 2;
            if i == 1
               allKeyPoints(j, k) = 1;
            end
            counter = counter + 1;
        end
    % Look for minima in DoG 2
    elseif DoGs{i, 2}(j, k) == currentMin && minCount == 1
        previousMin = min(min(DoGs{i, 1}(j - 1 : j + 1, k - 1 : k + 1)));
        nextMin = min(min(DoGs{i, 3}(j - 1 : j + 1, k - 1 : k + 1)));
        if DoGs{i, 2}(j, k) < previousMin && DoGs{i, 2}(j, k) < nextMin
            extrema(counter).octaveNumber = i;
            extrema(counter).y = j;
            extrema(counter).x = k;
            extrema(counter).sigma = startingSigma * sigmaFactor;
            extrema(counter).blurNumber = 2;
            extrema(counter).DoGNumber = 2;
            if i == 1
               allKeyPoints(j, k) = 1;
            end
            counter = counter + 1;
        end
    end

    currentMax = max(max(DoGs{i, 3}(j - 1 : j + 1, k - 1 : k + 1)));
    currentMin = min(min(DoGs{i, 3}(j - 1 : j + 1, k - 1 : k + 1)));
    [minCount, maxCount] = minMaxCount(DoGs{i, 3}(j - 1 : j + 1, k - 1 : k + 1));
    % Look for maxima in DoG 3
    if DoGs{i, 3}(j, k) == currentMax && maxCount == 1
        previousMax = max(max(DoGs{i, 2}(j - 1 : j + 1, k - 1 : k + 1)));
        nextMax = max(max(DoGs{i, 4}(j - 1 : j + 1, k - 1 : k + 1)));
        if DoGs{i, 3}(j, k) > previousMax && DoGs{i, 3}(j, k) > nextMax
            extrema(counter).octaveNumber = i;
            extrema(counter).y = j;
            extrema(counter).x = k;
            extrema(counter).sigma = startingSigma * sigmaFactor ^ 3;
            extrema(counter).blurNumber = 4;
            extrema(counter).DoGNumber = 3;
            if i == 1
               allKeyPoints(j, k) = 1;
```

```
                    end
                    counter = counter + 1;
                end
            % Look for minima in DoG 3
            elseif DoGs{i, 3}(j, k) == currentMin && minCount == 1
                previousMin = min(min(DoGs{i, 2}(j - 1 : j + 1, k - 1 : k + 1)));
                nextMin = min(min(DoGs{i, 4}(j - 1 : j + 1, k - 1 : k + 1)));
                if DoGs{i, 3}(j, k) < previousMin && DoGs{i, 3}(j, k) < nextMin
                    extrema(counter).octaveNumber = i;
                    extrema(counter).y = j;
                    extrema(counter).x = k;
                    extrema(counter).sigma = startingSigma * sigmaFactor ^ 3;
                    extrema(counter).blurNumber = 4;
                    extrema(counter).DoGNumber = 3;
                    if i == 1
                       allKeyPoints(j, k) = 1;
                    end
                    counter = counter + 1;
                end
            end
        end
    end
end

% Draw key points before refinements
if drawKeyPoints == true
    figure;
    subplot(1,3,1);
    imshow(allKeyPoints);
end

afterCornerKeyPoints = image;

% Reject key points that are edges
counter = 1;
for i = 1 : length(extrema)
    octaveNumber = extrema(i).octaveNumber;
    y = extrema(i).y;
    x = extrema(i).x;
    DoGNumber = extrema(i).DoGNumber;
    if isCorner(DoGs{octaveNumber, DoGNumber}(y - 1 : y + 1, x - 1 : x + 1)) == true
        cornerExtrema(counter) = extrema(i);
        if extrema(i).octaveNumber == 1
            afterCornerKeyPoints(y, x) = 1;
```

```
        end
        counter = counter + 1;
    end
end

if drawKeyPoints == true
    subplot(1,3,2);
    imshow(afterCornerKeyPoints);
end

% Reject key points that have a low contrast
afterCornerAndThresholdKeyPoints = image;
extrema(:) = [];
counter = 1;
for i = 1 : length(cornerExtrema)
    octaveNumber = cornerExtrema(i).octaveNumber;
    y = cornerExtrema(i).y;
    x = cornerExtrema(i).x;
    DoGNumber = cornerExtrema(i).DoGNumber;
    if abs(DoGs{octaveNumber, DoGNumber}(y, x)) > contrastThreshold

        extrema(counter) = cornerExtrema(i);
        if cornerExtrema(i).octaveNumber == 1
            afterCornerAndThresholdKeyPoints(y, x) = 1;
        end
        counter = counter + 1;
    end
end

if drawKeyPoints == true
    subplot(1,3,3);
    imshow(afterCornerAndThresholdKeyPoints);
end

end
```

getFeatureDescriptor.m

```
function [histogram] = getFeatureDescriptor(input, orientation)
% This function generates 16 historgrams with 8 bins each to represent a
% SIFT feature.

histogram = zeros(16, 8);
windowSize = 16;
```

```
gaussianKernel = fspecial('gaussian', windowSize, windowSize / 2);

% Loop through 16 x 16 window calculating gradient angles and magnitudes
for i = 2 : 17
    for j = 2 : 17
        magnitude = ( (input(i + 1, j) - input(i - 1, j))^2 + (input(i, j + 1) - input(i, j - 1))^2
        ↪ )^0.5;
        angle = calcGradientAngle(input(i - 1 : i + 1, j - 1 : j + 1));
        if angle ~= -1
            % Adjust features orientation based on key point orientation
            angle = mod(angle + (360 - orientation), 360);
            binNumber = idivide(int16(floor(angle)), int16(45)) + 1;
            histogramNumber = floor((i - 2) / 4) * 4 + ceil((j - 1) / 4);

            % Trilinear interpolation so that values closer to the centre
            % of the bin are weighted more heavily
            centreOffsetWeight = 1 - (abs(angle - (double(binNumber) * 45) - 22.5)) / 45;
            % Apply gaussian wieghting
            histogram(histogramNumber, binNumber) = histogram(histogramNumber, binNumber) +
            ↪ (magnitude * gaussianKernel(i - 1, j - 1) * centreOffsetWeight);
        end
    end
end

% Normalise histogram
rootOfSquaredSum = sum(sum(power(histogram, 2))) ^ 0.5;
histogram = histogram / rootOfSquaredSum;

% Set values above 0.2 to 0.2
for i = 1 : 16
    for j = 1 : 8
        if histogram(i, j) > 0.2
           histogram(i, j) = 0.2;
        end
    end
end

% Normalise histogram again
rootOfSquaredSum = sum(sum(power(histogram, 2))) ^ 0.5;
histogram = histogram / rootOfSquaredSum;

end
```

getImagePaths.m

```
function [imageList] = getImagePaths(imageDIR)
% This function returns all images in a given directory
images = dir(sprintf('%s/*.png', imageDIR));
imageList = images;
end
```

getRotationOrientation.m

```
function [rotation, secondRotation] = getRotationOrientation(extrema, blurredImage)
% This function get the dominent and second dominent orientation for a key
% point

rotation = -1;
secondRotation = -1;
windowSize = 15;
halfWindowSize = (windowSize + 1) / 2;

% Check key point is not too close to the edge of the image to look at a
% neighbourhood of gradients
if extrema.y - halfWindowSize >= 1 && extrema.x - halfWindowSize >= 1 && extrema.y + halfWindowSize
↪  <= size(blurredImage, 1) && extrema.x + halfWindowSize <= size(blurredImage, 2)

    % Take neighbourhood subsection
    subSection = blurredImage(extrema.y - halfWindowSize : extrema.y + halfWindowSize, extrema.x -
    ↪  halfWindowSize : extrema.x + halfWindowSize);
    histogram = zeros(36,1);
    gaussianKernel = fspecial('gaussian', windowSize, extrema.sigma * 1.5);

    % Magnitude calculation and Orientatioin calculation
    for i = 2 : windowSize + 1
        for j = 2 : windowSize + 1
            magnitude = ( (subSection(i + 1, j) - subSection(i - 1, j))^2 + (subSection(i, j + 1) -
            ↪  subSection(i, j - 1))^2 )^0.5;
            angle = calcGradientAngle(subSection(i - 1 : i + 1, j - 1 : j + 1));
            if angle ~= -1
                binNumber = idivide(int16(floor(angle)), int16(10)) + 1;
                % Add each rotation to the historgram with gaussain
                % weighting
                histogram(binNumber) = histogram(binNumber) + (magnitude * gaussianKernel(i - 1, j
                ↪  - 1));
            end
        end
```

```
    end
    % Find max histogram bin
    maximum = max(histogram);
    for i = 1 : 36
        if maximum == histogram(i)
            rotation = (i - 1) * 10 + 5;
            % If the second maximum bin is within 80% of the maximum bin
            % then calcualte the secondary rotation
        elseif 0.8 * maximum < histogram(i)
            secondRotation = (i - 1) * 10 + 5;
        end
    end
end

end
```

isCorner.m

```
function [corner] = isCorner(patch)
% This function detect if a pixel is on a corner or an edge
r = 10;

% Calculate Hessian matrix
DXX = patch(2,3)+patch(2,1)-2*patch(2,2);
DYY = patch(3,2)+patch(1,2)-2*patch(2,2);
DXY = (patch(1,1)+patch(3,3)-patch(1,3)-patch(3,1)) / 4;
trace = DXX+DYY;
determinant = DXX*DYY-DXY*DXY;
curvature = trace*trace/determinant;

% Check if the pixel is on a corner
if curvature > (r+1)^2/r || determinant < 0
    corner = false;
else
    corner = true;
end

end
```

maxCorrelation.m

```
function [correlation, position] = maxCorrelation(image, patch)
% This function computes the maximum correlation between an image and patch
```

```
% For RGB do correlation for each colour channel and compute the mean
if size(image, 3) > 1
    correlations(:,:,1) = normxcorr2(patch(:,:,1), image(:,:,1));
    correlations(:,:,2) = normxcorr2(patch(:,:,2), image(:,:,2));
    correlations(:,:,3) = normxcorr2(patch(:,:,3), image(:,:,3));
    correlations = mean(correlations, 3);
else
    correlations = normxcorr2(patch, image);
end

correlation = max(correlations(:));
[ypeak, xpeak] = find(correlations==max(correlations(:)));

% Compute translation from max location in correlation matrix
yOffset = ypeak-size(patch,1);
xOffset = xpeak-size(patch,2);

% If more than one maximum is found then only take the first one
if length(yOffset) > 1
    yOffset = yOffset(1);
end

if length(xOffset) > 1
    xOffset = xOffset(1);
end

position = [xOffset+1, yOffset+1, size(patch,2), size(patch,1)];

end
```

minMaxCount.m

```
function [minCount, maxCount] = minMaxCount(input)
% This function calcualtes the number of times that maxmimum and minimumm
% values occur in a matrix;
minCount = 0;
maxCount = 0;
maximum = max(input(:));
minimum = min(input(:));
inputSize = size(input, 1) * size(input, 2);
for i = 1 : inputSize
    if input(i) == maximum
        maxCount = maxCount + 1;
```

```
    end
    if input(i) == minimum
        minCount = minCount + 1;
    end
end
end
```

plotSIFTAccuracy.m

```
% This script plots the accuracy values for differing starting sigma and
% sigma factor values

clear;
rootTwo=sqrt(2);
startingSigma = [0.2, 0.4, 0.8, 1.2, 0.2, 0.4, 0.8, 1.2, 0.2, 0.4, 0.8, 1.2];
sigmaFactor = [rootTwo, rootTwo, rootTwo, rootTwo, 2, 2, 2, 2, 2.5, 2.5, 2.5, 2.5];
accuracy = [0,0,0,0,0.125,0.083333333,0,0,0.133333333,0.141666667,0.008333333,0];
accuracy = accuracy .* 100;

xq = linspace(min(startingSigma), max (startingSigma));
yq = linspace(min(sigmaFactor), max (sigmaFactor));
[X,Y] = meshgrid(xq,yq);
Z = griddata(startingSigma,sigmaFactor,accuracy, X, Y, 'cubic');
figure;
surf(X,Y,Z);
grid on;
title('Plot of Initial \sigma and \sigma Factor against Accuracy %')
set(gca,'FontSize',36)
xlabel('Starting \sigma')
ylabel('\sigma Factor')
zlabel('Accuracy %')
```

reduceScores.m

```
function [newScores] = reduceScores(scores, SSDThreshold)
% This function gets rid of matches that are below the SSD threshold and matches that
% do not have a ratio of at least 0.8 between the best and second best SSD
newScoreCounter = 1;
newScores = scores(1);
newScores(:) = [];
for i = 1 : length(scores)
    if isfield(scores, 'secondBestSSD') ~= 0 && scores(i).bestSSD ~= Inf
        if scores(i).bestSSD / scores(i).secondBestSSD < 0.8 && scores(i).bestSSD < SSDThreshold
```

```
            newScores(newScoreCounter) = scores(i);
            newScoreCounter = newScoreCounter + 1;
        end
    end
end

end
```

resizeImage.m

```
function [outputIMG] = resizeImage(inputIMG, gaussianKernel, subsampleRate)
% This function resizes images after applying a gaussiaan blur

% For RGB do correlation for each colour channel seperately
if size(inputIMG, 3) > 1
    image(:, :, 1) = convn(inputIMG(:, :, 1), gaussianKernel, "same");
    image(:, :, 2) = convn(inputIMG(:, :, 2), gaussianKernel, "same");
    image(:, :, 3) = convn(inputIMG(:, :, 3), gaussianKernel, "same");
else
    image = convn(inputIMG, gaussianKernel, "same");
end

% Subsample image
outputIMG = image(1 : 2^(subsampleRate-1) : end, 1 : 2^(subsampleRate-1) : end, :);
end
```

runIntensityMatching.m

```
function [] = runIntensityMatching(testImageNumber)
% This function runs the whole intensity based matching task

numSizes = 5;
numRotations = 2;
greyScale = true;
rotationStep = 360 / numRotations;

% Create test image path
testImagePathPart1 = 'dataset\Test\test_';
testImagePathPart2 = '.png';
testImagePath = strcat(testImagePathPart1, num2str(testImageNumber), testImagePathPart2);

% Read in training images
trainImageDIR = 'dataset\Training\png\';
```

```
listTrainingImages = getImagePaths(trainImageDIR);
numTrainImages = length(listTrainingImages);

% Generate all templates
templateIMGs = cell(numRotations, numSizes, numTrainImages);
templateIMGs = generateTemplates(numTrainImages, trainImageDIR, listTrainingImages, numRotations,
↪  rotationStep, numSizes, greyScale);

testImage = im2double(imread(testImagePath));
if greyScale == true
    testImage = rgb2gray(testImage);
end

% Get maximum correlation between each training image class and the test
% image
maxSet = cell(numTrainImages, 5);
maxSet = getAllMaxCorrelations(numRotations, rotationStep, numSizes, numTrainImages,
↪  listTrainingImages, templateIMGs, testImage);

sortedResults = sortrows(maxSet, 4, 'descend');
bestMatches = cell(1,5);
bestMatches = getBestMatches(sortedResults);

% Draw matches
drawIntensityMatches(bestMatches,testImage);

end
```

runSIFT.m

```
function [results, accuracy] = runSIFT(testImageNumber, testImagePath, SSDThreshold, startingSigma,
↪  sigmaFactor, contrastThreshold, removeMacthedFeatures)
% This function runs the whole SIFT algorithm

trainImagesRGB = cell(50, 1);
trainImagesGray = cell(50, 1);
trainImageFeatures = cell(50, 1);
drawKeyPoints = false;
trainImageDIR = 'dataset\Training\png\';
listTrainImages = getImagePaths(trainImageDIR);
numTrainImages = length(listTrainImages);

% Load in all the training images and generate their SIFT features
for i = 1 : numTrainImages
```

```
    trainImagesRGB{i} = im2double(imread(strcat(trainImageDIR, listTrainImages(i).name)));
    trainImagesGray{i} = rgb2gray(trainImagesRGB{i});
    %trainImageFeatures{i} = getAllFeatureDescriptors(trainImagesGray{i}, startingSigma,
    ↪  sigmaFactor, contrastThreshold, drawKeyPoints);
end

% Load the correct answers to the test images
load('answers.mat');

% Load training image features for speed
load('train_features.mat');
%save('train_features', 'trainImageFeatures');

% Read in the test image
testImageRGB = im2double(imread(testImagePath));
testImageGray = rgb2gray(testImageRGB);
testImageFeatures = getAllFeatureDescriptors(testImageGray, startingSigma, sigmaFactor,
↪  contrastThreshold, false);

scores = generateScores(trainImageFeatures, testImageFeatures, numTrainImages);
if length(scores) > 1
    scores = reduceScores(scores, SSDThreshold);
end

results = generateResults(scores, trainImageFeatures, removeMacthedFeatures);

% Commented out function calls for displaying results

% drawSIFTDescriptors(trainImageFeatures{46}, trainImagesRGB{46}, true);
% drawSIFTDescriptors(testImageFeatures, testImageRGB, true);
%matches = generateMatches(testImageFeatures, trainImageFeatures, scores, 24);
%drawFeatureMatches(testImageRGB, trainImagesRGB{24}, matches);

% Calculate accuracy of SIFT matching
accuracy = 0;
for i = 1 : length(results)
    if ismember(results(i), answers(testImageNumber, :)) == 1
        accuracy = accuracy + 1;
    end
end

end
```

B Test Scripts

convolutionTestScript.m

```
% This script shows that our covolution function produces the same results
% as conv2 by calculating the difference between the output from our
% convolution function and conv2 with a variety of images and kernels

image = double(int32(rand(3,3) * 100));
kernel = [-1,0,1;-2,0,2;-1,0,1];

convolution(image, kernel, 0, false) - conv2(image, kernel)

image = double(int32(rand(4,4) * 100));
convolution(image, kernel, 0, false) - conv2(image, kernel)

kernel = [1,2,3,4;5,6,7,8;9,10,11,12;13,14,15,16];
convolution(image, kernel, 0, false) - conv2(image, kernel)

image = double(int32(rand(3,3) * 100));
convolution(image, kernel, 0, false) - conv2(image, kernel)


% This shows convolution with the output being the same size as the input
image = double(int32(rand(3,3) * 100));
kernel = [1,2,3;4,5,6;7,8,9];

convolution(image, kernel, 0, true) - conv2(image, kernel, 'same')

image = double(int32(rand(4,4) * 100));
convolution(image, kernel, 0, true) - conv2(image, kernel, 'same')

kernel = [1,2,3,4;5,6,7,8;9,10,11,12;13,14,15,16];
convolution(image, kernel, 0, true) - conv2(image, kernel, 'same')

image = double(int32(rand(3,3) * 100));
convolution(image, kernel, 0, true) - conv2(image, kernel, 'same')

image = double(int32(rand(7,7) * 100));
kernel = [1,2,3,4,5;6,7,8,9,10;11,12,13,14,15;16,17,18,19,20;21,22,23,24,25];

convolution(image, kernel, 0, true) - conv2(image, kernel, 'same')

image = double(int32(rand(6,6) * 100));
```

```
convolution(image, kernel, 0, true) - conv2(image, kernel, 'same')

kernel = [1,2,3,4,5,6;7,8,9,10,11,12;13,14,15,16,17,18;19,20,21,22,23,24;25,26,27,28,29,30;31,32,33
↪   ,34,35,36];
convolution(image, kernel, 0, true) - conv2(image, kernel, 'same')

image = double(int32(rand(7,7) * 100));
convolution(image, kernel, 0, true) - conv2(image, kernel, 'same')
```