Lab 1

The SIFT descriptor

In this lab you will implement a SIFT-like descriptor and try it on some classification tasks. Keep your code well commented, both for your own sake and because it is required.

Ex 1.1 Make a function

```
patch = get_patch(image, x, y, patch_radius)
```

that takes out a quadratic patch from the image centred at (x, y) with a range `x`/`y` $\pm$ `patch_radius`. In other words, both the patch height and width should be 2 `patch_radius`$+1$ Make sure that your function works both for grayscale and color images.

Ex 1.2 Use

```
test_image = reshape((11:100), 10, 9)
```

to create a test image with numbered pixels. Try extracting a few patches from this image to verify that your function works. Make sure that the x-variable corresponds to the column index and the y-variable to the row index.

Ex 1.3 Modify `get_patch` so it returns an error with an informative error message such as

```
'Patch outside image borders'
```

if the patch doesn't fit inside the image. Use the `error` function for this.

Gradient histograms

Ex 1.4 Create a Matlab function `gaussian_filter` that takes two arguments, one grayscale image and one number specifying a standard deviation. The output should be the image filtered with a Gaussian filter of the specified standard deviation. Example usage:

```
result = gaussian_filter(img, 3.0);
```

The filter size should be at least four standard deviations not to loose precision. You can use `fspecial` to construct a Gaussian filter. It is a good idea to use the `'symmetric'` option with `imfilter`.

Ex 1.5 Make a function

```
[grad_x, grad_y] = gaussian_gradients(img, std)
```

that takes a grayscale image and estimates both gaussian derivatives for each pixel. The output should be two matrices of same size as the input image.

Ex 1.6 Plot your gradients in the image using

```
imagesc(img)
axis image
hold on
quiver(grad_x, grad_y)
```

and verify visually that the gradients are correct.

Ex 1.7 Make a function

```
histogram = gradient_histogram(grad_x, grad_y)
```

that places each gradient into one of eight orientation bins. A useful function is `atan2` in Matlab. Use the bins and bin order from the lecture notes. The provided `plotBouquet` lets you plot the histograms as a bouquet of vectors and might be helpful for debugging. It assumes that you have given all functions exactly the names suggested here and used the bin ordering from the lecture notes.

A SIFT-like descriptor

Next, we will create a SIFT-like descriptor by computing gradient histograms for 3×3 regions and stacking them into a vector. The exact positions and sizes of the regions are not crucial. For example, you might choose whether to use overlapping regions or not. Figure 1.1 shows an example of how to place the nine regions. You can use the provided `paper_with_digits.png` as an example image. For example, there is a digit at $(1290, 950)$.

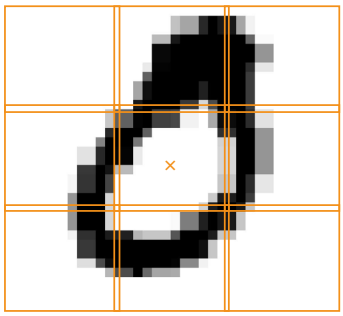

Figure 1.1: Example patch placement. The x marks the coordinates in the argument position .

Ex 1.8 Make a function `[region_centres, region_radius] = place_regions(centre, scale)` that creates 3×3 regions for the descriptor; see Figure 1.1. The output should be an 2×9 array with the centre points of each region and a number with the radius. Use the provided function

```
plot_squares(img, region_centres, region_radius)
```

to plot your regions in an example image. Increasing the input `scale` with a factor K should scale the whole region pattern with a factor K. Save a screenshot of the plotted regions for the report.

Ex 1.9 Make a function

```
desc = gradient_descriptor(image, position, scale)
```

that computes a SIFT-like descriptor at a certain position. The input `scale` controls the size of the regions just as in Ex 1.8.

- Compute gaussian gradients. Let the standard deviation be proportional to `scale`.
- Divide your gradients into 3×3 regions defined by `place_regions`.
- Compute a gradient histogram for the gradients from each region.
- Stack the histograms into a 72-vector.
- Normalize that vector to unit length.

Digit classification

In the file `digits.mat`, there are two lists of images, `digits_training` with 100 training images and `digits_validation` with 50 similar images. Our next goal is to classify each of the validation images by finding the most similar image in the training set and assuming that the query image has the same label. Load the digit data by running

```
load digits.mat
```

The examples are stored in a struct array. To get image number 12 you write `digits_training(12).image` and to get its label you write `digits_training(12).label`.

Ex 1.10 Make a script `prepare_digits.m` that computes a descriptor for each digit in `digits_training`. You need to choose the `position` and `scale` parameters so that all the descriptor regions fit into the images. Store the descriptors in an appropriate place. A suggestion is to store the 12th descriptor in

```
digits_training(12).descriptor
```

Ex 1.11 Make a function

```
label = classify_digit(digit_image, digits_training)
```

that computes a descriptor for the given digit image, goes through all the digits in `digits_training` to find the one with the most similar descriptor and outputs the label of that digit.

You can use `disp` to display text in matlab. For example

```
disp(['I am ' num2str(age) ' years old'])
```

will display your age, assuming that it is stored in the variable `age`.

Ex 1.12 Make a script `classify_all_digits` that runs `classify_digit` for each of the digits in `digits_validation` and displays the percentage of correct answers.

Ex⋆ 1.13 Try to classify a few of the large digits in `paper_with_digits.png`. For example there are digits at

$$(1290,950) \quad , \quad (820,875) \quad , \quad (220,570) \quad , \quad (170,330) \tag{1.1}$$

Note that you need to change the `scale` parameter. Does it work? Any ideas on how to set the scale parameter automatically?

Using the SIFT code from vlfeat

To speed things up a bit, we will use the SIFT descriptor from the vlfeat toolbox in the next few exercises. It is written in C, so it is much more efficient than your Matlab implementation. Use

```
[coords, descriptors] = extractSIFT(img);
```

to compute positions and descriptors for the SIFT features in an image.

Ex 1.14 Try to work out how to use the built-in functions `matchFeatures`.[1] To match descriptors using the Lowe criterion with threshold 0.7, add the following options:

```
corrs = matchFeatures(descs_1, descs_2, 'MatchThreshold', 100, 'MaxRatio', 0.7);
```

Ex 1.15 In `church_data.mat` there is a collection of stored feature points, `feature_collection`. This is a struct with a $128 \times N$-array `feature_collection.descriptors` containing descriptors and a $1 \times N$ array of labels

```
feature_collection.labels
```

indicating what church the feature was collected from. The link between labels and church names is given by

```
feature_collection.names
```

Make a function

```
[label, name] = classify_church(image, feature_collection)
```

that tries to classify a new image by computing feature points for the new image, matching them to the features in the list and letting each match *vote* for the correct church.

Ex 1.16 Try classifying all ten provided church images in `church_test`. How many do you get right? The correct labels are stored in `manual_labels.mat`.

Report

Make a zip file with your code for this lab. Also include the plot from Ex 1.8. Send it in per email and write in the message what percentages you got on the digit and church experiments.

[1] Note that Matlab and vlfeat does not agree on whether to store the descriptors as rows or columns.

Lab 2

Learning and convolutional neural networks

2.1 Learning a linear classifier

In this part, we will try to learn a linear classifier for traffic sign detection. Note that the classifier could also be viewed as a minimal neural network consisting of three parts: a scalar product node (or fully-connected node), a constant (or bias) term and a logistic sigmoid function. To find good parameters we will try to minimize the negative log-likelihood over a small training set.

The output from our classifier is a probability p for the input patch being centered at a cell centre. The sigmoid function will make sure that $0 \leq p \leq 1$. To be more precise the output is

$$p = \frac{e^y}{1+e^y} \quad \text{where} \quad y = I \cdot w + w_0. \tag{2.1}$$

2.2 Preparing the data

Instead of testing a bunch of manually chosen w's and w_0's, we will try to learn good values for all the parameters. This requires training examples, that you find in `training_data.mat`.

Ex 2.1 Load the data using

```
load traffic_data.mat
```

It loads a cell array, `examples`, with all the example patches and 1D array, `labels`, with the labels of each of the examples: `false` for negative examples and `true` for positive examples.

Ex 2.2 View some of the positive examples and some of the negative examples to verify that it looks right.

2.3 Training the classifier

We will try to find parameters that minimize the negative log-likelihood on the training data. More precisely,

$$L(\theta) = \sum_{i \in S_+} -\ln p_i + \sum_{i \in S_-} -\ln(1-p_i) = \sum_i L_i(\theta), \tag{2.2}$$

where p_i refers to the classifier output for the ith training example. As in the lectures we will refer to the terms here as the partial loss L_i.

Before doing the next exercise, you need to work out how to compute the gradient of the partial loss L_i.

Ex 2.3 Make a function

```
[wgrad, w0grad] = partial_gradient(w, w0, example, label)
```

that computes the derivatives of the partial loss L_i with respect to each of the classifier parameters. Let the output `wgrad` be an array of the same size as the weight image, `w` (and let `w0grad` be a number).

At each iteration of stochastic gradient descent, a training example, i, is chosen at random. For this example the gradient of the partial loss, L_i, is computed and the parameters are updated according to this gradient. The most common way to introduce the randomness is to make a random reordering of the data and then going through it in the new order. One pass through the data is called an epoch.

Ex 2.4 Make a function

```
[w, w0] = process_epoch(w, w0, examples, labels)
```

that begins by making a random reordering of the training examples and then goes through them in order performing a stochastic gradient descent step for each example.

Ex 2.5 Initialize `w = 0.01*randn(35,35,3);` and `w0 = 0;` and run 5 epochs on your training examples. Plot `w` after epoch (or after each iteration if you are curious), to get a sense of what is happening.

Ex 2.6 Make a function

```
prob = sliding_window(img, w, w0)
```

that applies your linear classifier at every possible point in a larger image `img`. For simplicity make sure the output `map` is as large as the input image. *Matlab hint:* Use `imfilter` for each colour channel.

Ex 2.7 Use your function `sliding_window` to run your new classifier on a few of the images found in

```
traffic_test/
```

Visualize the obtained probability map on the image using

```
view_with_overlay(image, probmap>0.5)
```

If you have managed well you should have yellow overlays at traffic signs. Include one of the resulting images for the report.

Ex 2.8 Use the function `strict_local_maxima` that you made for the exercises to extract local maxima from the probability map. To avoid detections at the borders, make sure you use the `'symmetric'` option for `ordfilt2`. Due to noise you might get many detections at the same traffic sign. To avoid this you might want to filter the probability map with a Gaussian before finding local maxima. Plot the local maxima in the image using

```
imagesc(img); colormap gray; axis image;\\ hold on; \\  plot(maxima(1,:), maxima(2,:),'r*')
```

Save a screenshot of the result for your report.

2.4 Deep learning

In the last part, your task is to try a system for training convolutional neural networks called MatConvNet. First you need to install it. Our recommendation is that you do this part on a Chalmers computer, but installation on your own laptop should also be possible.

Ex 2.9 First we need to fix a Matlab bug. This is hopefully *only* necessary on the Chalmers computers. In Matlab, run

```
mex -setup C++
fileattrib('$\sim$/.matlab/R2015b/mex_C++_glnxa64.xml', '+w')
edit $\sim$/.matlab/R2015b/mex_C++_glnxa64.xml
```

Find where it says `-std=c++11` and change that to `-std=c++0x`

Ex 2.10 Install MatConvNet. You can do everything from inside Matlab:

```
untar('http://www.vlfeat.org/matconvnet/download/matconvnet-1.0-beta23.tar.gz');
cd matconvnet-1.0-beta23/matlab
vl_compilenn('EnableImreadJpeg', false);
```

The last step requires that you have a Matlab-compatible compiler. The Chalmers computers should be fine but if you like to install MatConvNet on your own laptop you might need to install a compiler as well. (If you have a CUDA-supported graphics card, you might want to enable using the GPU.)

Ex 2.11 Run `vl_setupnn.m` to set the paths for MatConvNet. If you restart Matlab, you need to repeat this step. If you are not in the right folder you can run the file using

```
run $\sim$/matconvnet-1.0-beta23/matlab/vl_setupnn.m
```

Ex 2.12 In `characters_train.mat`, you find a `struct` called `data_train` with 12103 32×32-images of characters. For each character there is also a label between 1 and 26 indicating its place in the alphabet. There are also files `characters_val` and `characters_test`. Compute the mean pixel intensity over all the training images. Keep this value.

The next step is to define a network for classification and initialize all the weights. Since this is not that straightforward in MatConvNet an example is provided in `build_network.m`. Try to understand how it works. Note that the bigger a filter is (in terms of the number of parameters), the smaller its initial values. Also note that there are no fully-connected layers in MatConvNet. A large convolution without padding is used to the same effect.

Ex 2.13 Open the file `build_network.m`. Make a sketch of the network. Try to follow the image patch through the network. If we start with with a 32×32 patch, what is the size of the output after each layer (filtering, nonlinearities, pooling) etc. Note that by default MatConvNet does not use padding for the filters, so the output after a convolutional layer is spatially smaller than the input. Run

```
net = build_network
```

to setup a network.

Ex 2.14 In the initial phases of training, CNN's are very sensitive to scaling of the data. Make a function

```
data = preprocess_data(data)
```

that normalizes all the images in `data.imgs`. First compute a mean pixel intensity over all the images, then subtract this from all images and scales them with 256. Make sure to run this on all data before sending it to the CNN.

The code for actually training a convolutional network is provided in `cnn_train_mod.m` To run we need a training dataset, a validation dataset and a way to randomly select a subset of the data (to perform the stochastic gradient descent). Note that like most systems for deep learning, MatConvNet uses single precision floating point numbers, so we need to convert the input before sending it to MatConvNet. Use

```
 img = single(img)
```

to convert an image to single in Matlab.

Ex 2.15 Make a function

```
[img_array, labels] = random_subset(data, K)
```

that selects `K` random examples from `data`, and places them in an array of type `single`, `img_array` of size $32 \times 32 \times 1 \times K$. It should also return the corresponding labels in a $1 \times K$-array `labels`

To tell `cnn_train_mod` what function to use for sampling random data, we need a way to point to a Matlab function. Just writing its name will call the function and use its return value. To point to the actual function instead we use an `@` before the function name.

Ex 2.16 Load `dataset_train` and `dataset_val`, preprocess and run 5 epochs of training using

```
net = cnn_train_mod(net, data_train, data_val, @random_subset, 'chars/', 5)
```

The input is specifying the network to train, training and validation data, what sampling function to use, where to store the network and how many epochs to run. The number of epochs is the total number of epochs, so if you rerun the file with the same number it won't do anything. Also note that if you make a mistake and want to start from scratch, you need to remove the `chars` folder where the network is saved. Otherwise it will just keep loading the same network.

Ex 2.17 Having run at least 5 epochs, try reducing the learning rate and run a few more. You find the learning rate along with other parameters at the top of the `cnn_train_mod` file. Did it reduce the errors further?

Ex 2.18 Run the network on all the images in `characters_test.mat` using the

```
predicted_labels = cnn_predict(net, data)
```

Compute precision and recall for each of the 26 classes and include these in your mini-report.

Ex 2.19 Save three of the failure cases with names indicating what character they were mistaken for. Include these in your submission. You can use `imwrite(img, 'mistaken_as_5.png')` to save an image if it is correctly scaled. Have a look at the file before submitting it, so it looks right.

For the next exercise you need a way to go from labels as numbers to the actual character that they encode. There is a strange way to do this in Matlab. Set

```
alphabet = 'abcdefghijklmnopqrstuvwxyz'
```

Now you can get the 12th character by taking

```
alphabet(12)
```

Ex 2.20 Make a copy of `build_network.m` and name it `build_my_network.m`. Try modifying the network by adding more layers. (You need to be careful not to change the form of the output.) Use a new folder for storing the new network when you train, otherwise `MatConvNet` will try to load your old network and be confused. Do you manage to improve the result? Include precision and recall for the alternative network in your submission. Also include `build_my_network.m`.

Report

Hand in your code together with the images and results indicated in the exercises.

Lab 3

Image registration

An affine transformation is written

$$\begin{pmatrix} \tilde{x} \\ \tilde{y} \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix} = A \begin{pmatrix} x \\ y \end{pmatrix} + t. \tag{3.1}$$

Apart from rotation, translation and scaling it also allows strecthing the image in an arbitrary dimension.

Ex 3.1 What is the minimal number of point correspondences, K, required in order to estimate an affine transformation between two images?

Once you have found a proper coordinate transformation between two images, you can use the provided function `affine_warp` to warp the source image and create a warped image. Let's try it

Ex 3.2 Load the image `mona.png` to the variable `img`. Try running

```
img = read_image('examples/mona.png');
A = [0.88 -0.48; 0.48 0.88];
t = [100;-100];
target_size = size(img);
warped = affine_warp(target_size, img, A, t);
imagesc(warped);
axis image;
```

Change the values in `A` and `t` to see what happens. Swap `A` for a `eye(2,2)` to try a pure translation. Does it behave as you expect?

For any estimation task it is a good idea to have at least one test case where you know what the answer should be. In the next exercise you should make such a test case for Ransac. Start by generating random points, `pts`, and a random transformation. Then transform these points to create a `pts_tilde`. If you want to make it more realistic, add random noise to the points. You now have two sets of points related by a known affine transformation as in (3.1). In the following exercises you will try to estimate this transformation. As you know the correct answer it is easy to detect if you make a mistake.

Ex 3.3 Make a function

```
[pts, pts_tilde, A_true, t_true] = affine_test_case
```

that generates a test case for estimating an affine transformation. The transformation should map `pts` onto the `pts_tilde`. Don't add any outliers right now. Outputs `pts` and `pts_tilde` should be $2 \times N$-arrays. Also output the *true* transformation, so you know what to expect from your code.

Ex 3.4 Make a minimal solver for the case of affine transformation estimation. In other words, make a function

```
[A, t] = estimate_affine(pts, pts_tilde)
```

that estimates an affine transformation mapping `pts` to `pts_tilde`, where `pts` and `pts_tilde` are $2 \times K$-arrays and K is the number you found in Ex. 3.1. Try your function on points from the test case in Ex 3.3.

Ex 3.5 Make a function

```
residual_lgths = residual_lgths(A, t, pts, pts_tilde)
```

that computes the lengths of the 2D residual vectors. The function should return an array with the N values.

Matlab hint: Given a $2 \times N$ matrix, `M`, the column-wise sum of the squared elements can be computed as `sum(M.^2, 1)`.

Ex 3.6 Modify your function `affine_test_case` so it takes a parameter `outlier_rate` and produces a percentage of outliers among the output points. For example, the outliers could be spread randomly over the image.

Ex 3.7 Make a function

```
[A,t] = ransac_fit_affine(pts, pts_tilde, threshold)
```

that uses Ransac to find an affine transformation between two sets of points. (Like before the transformation should map `pts` onto `pts_tilde`.) Test your function on test cases generated with your function `affine_test_case`. Try different outlier rates. Make sure that you get the right transformation and a reasonable number of outliers.

For the next exercise you should use the function

```
[pts, descs] = extractSIFT(img);
```

from Lab 1 to extract SIFT features. Note that it only works for grayscale images, so if you have a colour image you need to convert it using, e.g.,

```
[pts, descs] = extractSIFT(mean(img,3));
```

To match features you can use the built-in function `matchFeatures`. To use the Lowe criterion (with threshold 0.8) you should use the following options:

```
corrs = matchFeatures(descs1', descs2', 'MaxRatio', 0.8, 'MatchThreshold', 100);
```

Ex 3.8 Write a function

```
warped = align_images(source, target)
```

that uses SIFT and Ransac to align the source image to the target image. To perform the actual warping, use

```
warped = affine_warp(target_size, source, A, t);
```

Be very careful about the order in which you send the points to Ransac!

Ex 3.9 Align `vermeer_source.png` to `vermeer_target.png`. A primitive function

```
switch_plot(warped,target)
```

is provided for viewing. Every time you press a key on the keyboard it will switch between the warped image and the target. (After ten times it stops.)

Medical images often have less local structure, making SIFT matching more difficult. It often works better if we drop the rotation invariance. The provided `extractSIFT` function has an option for this

```
[points, descriptors] = extractSIFT(img, true);
```

assumes that the image has a default orientation.

Ex 3.10 Modify you `align_images` so it takes the outlier threshold, `threshold` as an input and a boolean, `upright` stating whether the images are have the same orientation, i.e.,

```
warped = align_images(source, target, threshold, upright)
```

Try aligning the images `CT_1.jpg` and `CT_2.jpg`. Try with and without rotation invariance and try different outlier thresholds.

Ex 3.11 Try aligning `tissue_fluorescent.tif` and `tissue_brightfield.tif`. In the fluorescent image, the intensities are basically inverted, so you need to invert one of the images before computing descriptors. (Otherwise you won't get any good matches.) If you used `read_as_grayscale` to load the images, they should have values between 0 and 1 so you can invert it by taking

```
inverted_img = 1 - img;
```

Warping

So far you have used Matlabs function for warping. The reason is that it is difficult to write a Matlab function for warping that is not painfully slow. Now you will get to write one anyway, but we will only use it for very small images.

Ex 3.12 Make a function

```
value = sample_image_at(img, position)
```

that gives you the pixel value at `position`. If the elements of `position` are not integers, select the value at the closest pixel. If it is outside the image, return 1 (=white). Try your function on a simple image to make sure it works.

Next, you will do a warping function that warps a 16×16 image according to the coordinate transfomation provided in `transform_coordinates.m`.

Ex 3.13 Make a function

```
warped = warp_16x16(source)
```

that warps `source` according to `transform_coordinates` and forms an output 16×16 image `warped`. Use your function `sample_image_at` to extract pixel values. Try the function on `source_16x16.tif` and plot the answer using `imagesc`. You will know if you get it right.

Least squares

Ex 3.14 Write a function

```
[A, t] = least_squares_affine(pts, pts_tilde).
```

(Depending on how you wrote you `estimate_affine.m`, this might be very easy.) Use it on the inliers from Ransac to refine the estimate. Add this to `align_images` and test it on the Vermeer images for different outlier thresholds. Do you notice an improvement?

Report

For the report, we want you to submit some images as well as all your code. To save an image `img` in Matlab you write something like

```
imwrite(warped, 'CT_warped.png')
```

Submit images `vermeer_warped.png`, `CT_warped.jpg` and `tissue_warped.tif`. It doesn't matter which direction you chose to warp in, but rename the images you used as target to `CT_target.jpg` and `tissue_target.tif` and submit them as well.

Lab 4

Triangulation

4.1 Using Ransac

This whole lab is basically concerned with the camera equation

$$\lambda u = PU. \tag{4.1}$$

For the uncalibrated case u, is a 3-vector with the coordinates of a point in the image (and an added 1)

$$u = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \tag{4.2}$$

and U is a 4-vector containing the coordinates of the corresponding $3D$ point along with a 1.

A function

```
[Ps, us, U_true] = triangulation_test_case(sigma)
```

is provided for creating a simple test case where you know the correct answer (`U_true`). Gaussian noise of standard deviation `sigma` is added to the image points. Use this example to evaluate the your minimal solver.

Ex 4.1 Make a minimal solver for the triangulation problem, that is, a function

```
U = minimal_triangulation(Ps, us)
```

that takes two camera matrices, `Ps`, and two image points, `us`, and triangulates a 3D point. The image points are a 2×2 array whereas the camera matrices is a cell list with one camera matrix in each cell.

Recall that λ in (4.1) is the *depth*. Points with negative depth would lie behind the camera, so negative depths indicate that something is wrong.

Ex 4.2 Make a function

```
positive = check_depths(Ps, U)
```

that takes N camera matrices, `Ps`, and a 3D point, `U` and checks the depth of `U` in each of the cameras. The output should be a an array of boolean values of length N that indicates which depths were positive. (Matlab will print boolean values as zeros and ones, so don't be confused by this.)

Ex 4.3 Make a function

```
errors = reprojection_errors(Ps, us, U)
```

that takes N camera matrices, `Ps`, N image points, `us`, and a 3D point, `U` and computes a vector with the reprojection errors, that is, the lengths of the reprojection residuals. If a point has negative depth, set the reprojection error to `Inf`.

Ex 4.4 Make a function

```
[U, nbr_inliers] = ransac_triangulation(Ps, us, threshold)
```

that implements triangulation using Ransac. Use the number of outliers as loss function. A measurement is deemed as an outlier if the depth is negative or if the reprojection error is larger than `threshold`.

In `sequence.mat` you find a struct array `triangulation_examples` with triangulation examples. Each example has a cell list of camera matrices `Ps` and a $2 \times N$-array `us` with image points. It will take some time to triangulate all 32183 examples so start with the first 1000 or so. I used a Ransac threshold of 5 pixels.

Ex 4.5 Make a script `triangulate_sequence` that runs `ransac_triangulation` for all (or at least 1000) of the examples from `sequence.mat`. Store all triangulated points with at least two inliers and plot them using `scatter3`. There will always be a few outliers among the estimated 3D points that make it harder to view the plot. You can use the provided `clean_for_plot.m` to clean it up a bit. Like this:

```
Uc = clean_for_plot(Us)
scatter3(Uc(1,:),Uc(2,:),Uc(3,:),'.')
axis equal
```

Can you recognize the building? Save an image for the report.

4.2 Least squares triangulation

Just as in the case of registration, the following pipeline is recommended:

- Use Ransac to obtain a rough estimate of the parameters (U).
- Remove all measurements which are outliers with respect to these parameters.
- Estimate the least squares parameters using the remaining measurements.

Note that in this case a *measurement* is a pair consisting of an image point u_i and a camera matrix P_i. Don't forget that points with negative depths should be outliers.

Ex 4.6 Consider a camera matrix

$$P_i = \begin{pmatrix} \leftarrow & a_i^T & \rightarrow \\ \leftarrow & b_i^T & \rightarrow \\ \leftarrow & c_i^T & \rightarrow \end{pmatrix}, \tag{4.3}$$

a 3D point U and an image point u_i. Write a formula for the reprojection error, $r_i(U)$.

As you can see, the residuals are no longer linear, so computing a least squares solution will be significantly harder than in the previous lab. In fact, we cannot be sure to find the least squares solution. What we can do is to use local optimization to reduce the sum of squared residuals. We start at the solution produced by Ransac and use a few Gauss-Newton iterations.

Ex 4.7 Make a function

```
all_residuals = compute_residuals(Ps, us, U)
```

that takes a cell list with N cameras, a $2 \times N$ array of image points and a 3×1 array `U` and computes a $2N \times 1$ array with all the reprojection residuals stacked into a single vector/array. This vector is the $\bar{r}$ from the lecture notes.

Ex 4.8 Find formulas for the partial derivatives in the Jacobian of $\bar{r}$. The Jacobian should be a $2N \times 3$-matrix. (You don't have to hand in anything.)

Ex 4.9 Make a function `jacobian = compute_jacobian(Ps, U)` that computes the Jacobian given a 3×1-vector `U` and a cell array of camera matrices.

Ex 4.10 Use these functions to make a function

```
U = refine_triangulation(Ps, us, Uhat)
```

that uses an approximate 3D point `Uhat` as a starting point for Gauss-Newton's method. Use five Gauss-Newton iterations. Print the sum of squared residuals after each Gauss-Newton step to verify that it decreases.

Ex 4.11 Try your `refine_triangulation` on the data in `gauss_newton.mat`. First plot the points given in `Uhat`, then refine each point using your function and plot the results using `scatter3`. You should see an improvement. Save a screenshot for the report.

Ex⋆ 4.12 Compute the camera positions for the data in `gauss_newton.mat` and plot them together with the estimated 3D points. Try to understand why the noise in the estimated points looks as it does.

For the next exercise try the following way to match features,

```
matchFeatures(d1,d2,'Method','Approximate','MaxRatio',0.9,'MatchThreshold',100);
```

The approximate option makes it go significantly faster. A Lowe ratio of 0.9 is very high, but it makes the model more *dense*.

Ex⋆ 4.13 There are images `duomo.jpg` and `duomo_tilde.jpg` in the lab folder and also a data file `duomo.mat` with two camera matrices. Extract SIFT features from the two images (using `extractSIFT`) and match them using `matchFeatures`. Also store the color of each SIFT point. It doesn't matter from which of the images you select the color information.

Use your triangulation code (don't forget `refine_triangulation`) to triangulate the points and plot them with the right color using `scatter3`. Again, you can use `clean_for_plot.m` to remove outliers among the estimated 3D points. Beware that triangulating all the points will take a little time so work with a subset until you are sure that the code works.

4.3 Report

Submit your code together with images of the reconstructed 3D models.