Advanced Graphics and Image Processing

Image Procesing Practical

Michaelmas 2025/2026

1 Development environment

We recommend completing this practical exercise in Python 3.5 or later. You can optionally work in Matlab if you are more familiar with it, but note that all skeleton code and examples are given in Python.

You need to install a 64-bit version of Python 3.5 or later. The 64-bit version is necessary to allocate large enough matrices for linear solvers. You also need to install the following python packages:

- numpy
- scipy
- matplotlib
- scikit-image
- scikit-sparse

You may also need Gimp (or Photoshop) to prepare your images.

Since the installation of packages in more involved on Windows, we recommend to use Ubuntu instead. If you are an experienced Python programmer, you can stick to your favourite editor/IDE. But if you are unsure, we suggest that you follow the instruction below.

1.1 Ubuntu

- Install from packages: python3, python3-numpy, python3-pip, python3-scipy, python3-tk, python3-virtualenv, python3-matplotlib, libsuitesparse-dev. The package "libsuitesparse-dev" is needed for scikit-sparse. On Ubuntu:

```
sudo apt-get install python3 python3-numpy python3-pip
↪  python3-scipy python3-tk python3-virtualenv
↪  python3-matplotlib libsuitesparse-dev
```

- Create a virtual environment in the directory with the python code (from bash):

  ```
  cd practical
  virtualenv ve_practical
  ```

- Activate that virtual environment

  ```
  source ve_practical/bin/activate
  ```

- Install the required Python packages using pip

  ```
  pip3 install -r requirements.txt
  ```

  (Never install packages with pip3 in the system directories.)

- Install a good Python editor/IDE. One popular choice is Visual Studio Code (https://code.visualstudio.com) and Python extensions. It is important to be able to run the code in the debugger, where you can stop at a break point and investigate all variables and display images at intermediate processing stages.

1.2 Windows

- Install Anaconda from https://www.anaconda.com/products/individual
- From Anaconda prompt, create and the activate a new environment

  ```
  conda create -n practical python=3.6
  conda activate practical
  ```

- Follow the Windows installation instruction for scikit-sparse: https://github.com/EmJay276/scikit-sparse#windows-installation
- Install the remaining packages:

  ```
  conda install -c conda-forge scikit-image
  ```

- Install Visual Studio Code from https://code.visualstudio.com/
- Make sure to select "Python 3.6.* 64-bit ('practical_1':conda)" as the running environment (at the bottom of the VSC window).

2 Submission

In this practical exercise, we ask you to complete the missing parts of the code and to answer questions. Your Moodle submission should consist of two files:

- A ZIP file with the code and images (use only ZIP, no rar, 7z, etc.). Do NOT include the virtual environemnt directory or python cache files;
- A PDF document with a short report, which should include the results (images) and answers to all questions. Make your answers possibly concise.

You must use *your own images* for all the tasks listed below. You can start experimenting with the images we provide, but all the images in the report should be yours. You should preferably use images from your own camera, but you can also use images downloaded from the Internet as long as you do not violate copyright (e.g., use only images with the Creative Common license).

3 Marking

The points assigned to each task are shown next to the tasks. The points are awarded based on the quality of the report, code and the answers given in the demonstration session.

4 Warm-up

The purpose of this warmup section is to help you get familiar with how to work with images in Python. One of the popular packages to work with images in Python is OpenCV, which provides comprehensive algorithms related to computer vision and image processing, but it has its own quirks and is quite bulky. For this practical, we will use a simpler and more light-weight library: scikit-image. It is worth spending some time checking the examples on the scikit-image webpage. You should also get yourself familiar with numpy.

It is very important that you write properly vectorized code using numpy arrays; you should avoid iterating over all elements of an array (matrix) to perform one of the standard operations (addition, multiplication, ...) as those will be executed much faster if you use numpy functions or operators. You may be penalized if you do not write vectorized code. If you have some experience with Matlab, check NumPy for MATLAB users. If you have no experience with Matlab or numpy, spend some time going over numpy tutorials.

Figure 1: *An image before (left) and after (right) histogram equalisation*

Download `practial1.zip` from the Moodle area *Advanced Graphics and Image Processing* → *Practical 1 - Image Processing*, in which you will find the skeleton Python code for all the tasks.

Next, you should run and check the code in `task0_grayscale.py`. This is a simple example showing how to read images, convert between colour spaces, and display them. This code will also help you in testing your development environment.

Task 1: Image Enhancement [2 pt]

The pictures we take in foggy conditions may not appear as vivid as seen in person. This is because visual system can adapt to lower contrast but a camera typically has a fixed response function regardless of the scene. However, we can remedy this with image enhancement operations, which can balance colours, increase contrast or sharpness.

Histogram Equalisation (grey scale)

Your first task is to implement the histogram equalisation discussed in lecture 1. An example result of this operation is shown in Figure 1. Generally, histogram equalisation is performed in three steps:

1. Compute an image histogram for a grayscale image
2. Compute a normalized cumulative histogram

$$c(I) = \frac{1}{N} \sum_{i=0}^{I} h(i)\,,$$

   where $h(i)$ is the histogram value for bin i.

3. Use the cumulative histogram to map pixels to the new values (use it as a look-up table)

First, select your own image that has low contrast (for example a distant landscape with a haze). Then, complete the code in `task1_histeq.py`. Include the original and result images in the report. You must *not* use any of the built-in functions for histogram equalization (such as `exposure.equalize_hist()`) but you can use the numpy function for computing a histogram (`numpy.bincount()`).

Histogram Equalisation for colour images

Now, let us consider colour images. There are several strategies to apply the histogram equalisation on images with more than one colour channel:

1. Apply histogram equalisation to each (R, G, B) channel separately;
2. Apply histogram equalisation to a greyscale image calculated from the RGB image and then restore the colour from colour ratio:

$$r_{new}(x,y) = v_{new}(x,y)\frac{r_{old}(x,y)}{v_{old}(x,y)}\,, \tag{1}$$

   where v_{old} is the grayscale image before enhancement, v_{new} is the image after enhancement and r_{old} is the red channel in the original image. The same formula should be applied to the green and blue channels. Remember to clip the values to the allowable range (0-1 for float, or 0-255 for integers) after this operation.
3. Convert an image from RGB space to HSV space, apply histogram equalisation on the "value" channel, and convert the image back to RGB space.

Try these 3 different strategies and report if there is any noticeable difference between the results. Test them with several images, including ones with vivid, saturated colours. Include all the results in the report. State in the report what strategy is likely to produce the best results.

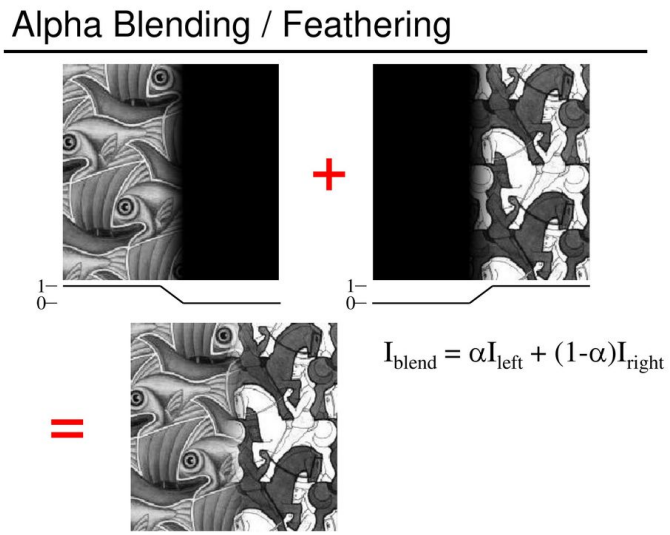
Task 2: Alpha blending [1 pt]

In this task, you will combine two images into one so that the content is "stitched" together in a possibly seamless manner.

The general idea is to blend two images by introducing a gradual transition in pixel values (see Figure 2). The plain alpha blending can be expressed as:

$$I_{blend}(x,y) = \alpha(x,y)I_{left}(x,y) + (1-\alpha(x,y))I_{right}(x,y) \tag{2}$$

Figure 2: *An example of the alpha blending operation*



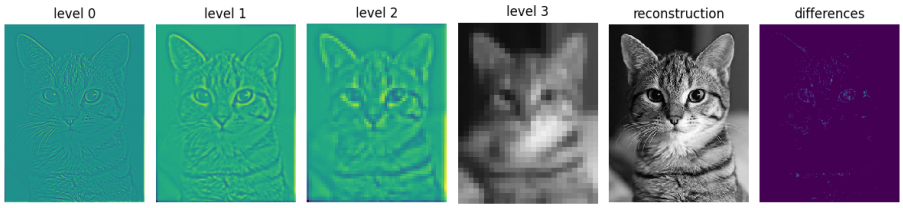


Figure 3: *An image decomposed into 4 levels of a Laplacian pyramid*

where the α is the alpha blending mask with the values between 0 and 1.

Your task is to take two own images of the same resolution, design an alpha-blending mask (as a function of (x, y)) and write the code for alpha blending in `task2_alpha_blending.py`. Put the input images, the alpha mask and the resulting image in the report.

Task 3: Pyramid Blending [4 pt]

Although alpha blending is simple, designing an ideal alpha blending mask for any kind of image is difficult. The transition region in the blending mask should be relatively large when blending smooth regions to avoid visible seams. But

when blending small structures, you want to use a small transition region to avoid ghosting.

For a robust method that can handle both smooth regions and a high-frequency structure, you can perform blending on a Laplacian pyramid. In so-called pyramid blending, both images are decomposed into Laplacian pyramids and the levels from both pyramids are blended one-by-one, using a different alpha mask. As lower frequencies have a larger spatial extend, the transition for those frequencies should be smoother. Higher frequencies have a smaller spatial extend and they need a sharper transition.

We will use an efficient algorithm for decomposing an image into a pyramid proposed by Burt and Adelson [1]. Refer to the lecture slides for the explanation of the algorithm (lecture "Introduction to image processing"). To decompose an image into the Laplacian pyramid, you will need two operations: `gausspyr_reduce` to filter and reduce an image resolution by half, and `gausspyr_expand` to double the size of an image. The latter function is implemented for you in `task3_pyramid_blending.py`, but you need to finish the implementation of `gausspyr_reduce`. You will need to use `sp.signal.convolve2d` to filter rows and the columns with the kernel. Make sure that the `mode='same'` and `boundary='symm'`. Once you implemented and tested `gausspyr_reduce`, you can finish two missing static methods in the class `laplacian_pyramid`: `decompose()` and `reconstruct()`. When those functions are correctly implemented, you should be able to run the code and see the result of the decomposition simular to the one shown in Figure 3.

The next step is to implement the function `pyramid_blending` (hint, use `alpha_blending` and `laplacian_pyramid`).

To test pyramid blending, you need a pair of images that contain both fine structure and smooth regions (gradients). If you cannot find such an image pair, test your pyramid blending on two image pairs, one containing “seam” in the smooth region, the other in which the seams goes across some fine structure. Experiment with different alpha masks for each level to achieve the best results. Use the same masks for all images you test on. Put your final image(s) in the report and briefly explain how you designed he alpha blending mask for each level of the pyramid to obtain the best results.

Task 4: Gradient domain reconstruction [4 pt]

It is strongly recommended that you refer to the lecture notes on gradient domain processing before attempting this task (the written description at the end of your printed slides, not the slides alone). The notes explain how to formulate reconstruction from a gradient field as a sparse linear problem.

The general motivation behind the gradient-based methods is that the human visual system is more sensitive to pixel differences (gradients) than absolute pixel values. Therefore, operations performed on image gradients should focus on visually important information and potentially produce more plausible results.

We will start with the reconstruction of the image from a gradient field. This could be formulated as the solution to the least-square problem:

$$\arg\min_I \left|\left| \begin{bmatrix} \nabla_x \\ \nabla_y \end{bmatrix} I - \begin{bmatrix} G^{(x)} \\ G^{(y)} \end{bmatrix} \right|\right|^2, \tag{3}$$

which can be reformulated as solving a sparse linear system (refer to the notes).

Your first task is to write the code for reconstructing an image from a gradient field in `task4_grad_domain.py`. Because setting up all the sparse matrices is tricky, we provide the code for the two forward derivative operators ∇_x and ∇_y in `reconstruct_grad_field` function as `Ogx` and `Ogy`. You need to combine those with the gradient field and the weights to compute the sparse matrix A and vector b and solve a sparse linear system. We also provide the code for converting an image into the gradient field in `img2grad_field`.

Use `sparse.linalg.spsolve` solver for this task.

Refer to the sparse and sparse.linalg *scipy* web pages to check how to operate on sparse matrices. Note also that in numpy/scipy operator "@" is used for matrix multiplication and "*" for element-wise multiplication. Matrix element indices start from 0 (rather than 1) and matrices are stored in the row-major order (in contrast to Matlab, which indexes from 1 and uses the column-major order).

A better solver for the linear system

The default matrix solver `sparse.linalg.spsolve` makes no assumption about the matrix when solving linear equation of the form $Ax = b$. But in many cases it is possible to find a faster or more effcient solver once we know what type of matrix we are deadling with. In our case, the matrix A is both symmetric and positive definite. For that kind of a sparse matrix, the preferable solver is the one that is based on the Sparse Cholesky decomposition. Use the module from scikit (sksparse.cholmod) to add another solver to your code. Then report the timing for the two solvers (`sparse.linalg.spsolve` and Cholesky solver) for both task 4 and task 5.

Figure 4: *An image before (left) and after (right) enhancement in the gradient domain.*

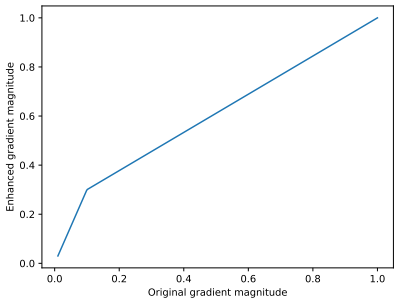


Figure 5: *Gradient magnitude enhancement function that could be used in Task 5.*

Task 5: Gradient domain image enhancement [4 pt]

In this task, you will perform image enhancement in the gradient domain and reconstruct the image from the modified gradient field using the code from the previous exercise. Figure 4 shows an example of the result that you can produce.

You should operate on a greyscale image and recover colour using any of the methods from Task 1. Your enhancement should keep gradient orientations, but replace gradient magnitudes with enhanced magnitudes. You could use, for example, a piecewise-linear function as the one shown in Figure 5. Such a function can be implemented as a linear interpolation using `scipy.interpolate.interp1d` class. Experiment with different enhancement functions to obtain the desired result. In this task you should also use a weighted least-squares formulation to avoid pinching artefacts.

In your report:

- Show input and enhanced images (the best is to use a low-contrast image

Figure 6: *An example of gradient domain copy & paste in which roughly selected source objects are pasted into the target image.*

for this task);

- Plot the gradient editing function you used;
- Test enhancement on images of different resolution: from 320×200 (or similar) to 3000×2000 or larger. Report the timings for each resolution as a plot of the number of (mega-)pixels vs. processing time. Show the measurements as markers on the plot. Remember to add labels to both axes.

Task 6: Gradient domain copy & paste [10 pt]

In the final task, you will implement the gradient domain copy & paste operations (seamless cloning) from the paper by Pérez et al. [2]. An example of such a cloning operation is shown in Figure 6. A similar method is also used for the healing brush tool in Adobe Photoshop.

You can prepare your own images for this task in Gimp (or Photoshop): make the source and target images of the same resolution and use the alpha-mask to mark the region in the source image to be pasted into the target image.

The basic idea behind this method is to copy the portion of the gradient field from the source image to the target image and then solve for the pasted pixels. However, to ensure that the pasted region seamlessly merges into the target, the values of the pixels that belong to the edge of the pasted region need to be

(soft-)constrained so that their values are close to the pixel values in the target image.

The problem formulation for this task is different from the one in the two previous tasks. Instead of solving for all pixels in the image, you will be solving the linear system only for the pasted pixels. This will make the solver much faster. But you cannot reuse the partial derivative operators from the last task and need to write your own code. Hint: use a "for" loop instead of trying to come up with a clever vectorised solution. We recommend to use `scipy.sparse.csr_matrix` to create sparse matrices — put matrix indices and values into pre-allocated vectors and then call this function to create a sparse matrix. This will be much faster than changing sparse matrix values one at a time in a "for" loop.

The problem can be formulated as:

$$\arg\min_I \left|\left| \begin{bmatrix} \nabla_x \\ \nabla_y \\ E \end{bmatrix} I - \begin{bmatrix} G^{(x)} \\ G^{(y)} \\ T_E \end{bmatrix} \right|\right|^2, \tag{4}$$

where E is a $K \times N$ matrix with a single 1 in each row, which corresponds to the pixel that belongs to the edge. N is the total number of pixels in the pasted region and K is the number of pixels at the edge. I is the vector with the N unknown pixels belonging to the pasted region. T_E is the vector with the values of the edge pixels in the target image. The remaining symbols are the same as in Equation 3 and the course notes.

You can achieve good results if you solve this problem separately for the red, green and blue colour channels. There is no need to include weighting as pinching usually is not an issue. You can use erosion and Boolean operations to find the pixels that belong to the edge of the pasted region.

References

[1] P. Burt and E. Adelson. The Laplacian Pyramid as a Compact Image Code. *IEEE Transactions on Communications*, 31(4):532–540, apr 1983.

[2] Patrick Pérez, Michel Gangnet, and Andrew Blake. Poisson Image Editing. *ACM Transactions on Graphics*, 3(22):313–318, 2003.