Assignment 1 – Computer Vision 1 – WS19/20 28/10/2019

Prof. Stefan Roth
Nikita Araslanov
Xiang Chen

This assignment is due on November 12th, 2019 at 23:59.

Group work and grading policy

You are required to work on each assignment in groups of ***two***. It is up to you to form groups, but please note that the group assignments cannot change after the first homework assignment. Moreover, we reserve the right to merge singleton groups randomly for subsequent assignments.

Programming exercises

For the programming exercises you will be asked to hand in Python code. Please use *Python 3.7* as we will use this version to test your solution. Use comments in the source code to clarify the functionality of your implementation in sufficient detail. Even if the computed results are incorrect due to a minor bug, you may still earn partial credit if your reasoning is valid.

You *must* adhere to the naming scheme for functions and files included with each problem. Do *not* alter function files and do *not* change the given function signatures. If you feel that there is a mistake in the assignments, or you find the task ambiguous, contact us on Moodle.

Multiple choice questions

There may be multiple choice questions in the assignments, which are implemented via a function or a class method. We provide a detailed explanation *in the source code* about the data type and format of the return value. Please, read the instructions carefully, because the format may vary depending on the question.

Pen & paper exercises

For theoretical exercises, we encourage typesetting your solutions with LaTeX and submitting the PDF. If you are not familiar with mathematical typesetting, such as LaTeX, you can also hand in a high-resolution scan of a handwritten solution. Please write neatly and legibly to avoid losing points due to ambiguities or misinterpretation.

Files you need

The data and source code skeleton and the PDF with the assignment tasks will be made available on Moodle `https://moodle.tu-darmstadt.de/course/view.php?id=17277`.

Handing in

Please, upload your solutions in the corresponding section on Moodle. Each problem task will specify the files to be included in your submission. You only need to submit one solution per group. Should you have troubles accessing Moodle, get in touch with us as soon as possible. Upload all your solution files as a single `.zip` or `.tar.gz` file. *We do not accept other file formats!*

Late submissions

We will accept late submissions, but you will lose *20% of the total reachable points* for every day of delay. Note that even 15 minute delays will be counted as being one day late! After the assignment solution has been discussed in class, you may no longer hand in.

Other remarks

Your grade will depend on two factors. Firstly, it will be determined by the correctness of your answer. Secondly, it will depend on the clarity of presenting your results and a good writing style. It is your task to find a way to *explain clearly how* you solved the problems. You can still get partial credit even if you did not complete the task.

We encourage interaction about class-related topics both in-class and on Moodle. However, you should not share solutions with other groups, and **everything you hand in must be your own work**. You are also not allowed to copy material from the web without acknowledgment. You must **acknowledge any source of information that you used to solve the homework** (i.e. books other than the course books, papers, etc.). Using acknowledgments will *not* affect your grade, but failing to do so is a clear violation of academic ethics. Note that the university as well as the department is very serious about plagiarism. For more details please see `http://www.informatik.tu-darmstadt.de/index.php?id=202` and `http://plagiarism.org`.

Problem 1 - Getting to know Python (5 points)

In this task you will set up `Python-3.7` on your system using `Miniconda` and learn how to install additional packages. Packages that you will often use are: (i) `Pillow` for loading and saving images; (ii) `NumPy` and `SciPy` for scientific computing; and (iii) `Matplotlib` for plotting and visualisation.

Note: In contrast to the `Anaconda` distribution, `Miniconda` supplies only the package management system. This allows to set up minimalistic environments when the disk space is limited (*e.g.* pool PCs offered by ISP). If you have `Anaconda` already installed and would like to use it instead, the instructions below should also work.

Tasks:

- Installing `Miniconda`:

  1. Download Miniconda

     ```
     # Linux
     $ wget https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
     # MacOSX
     $ curl -O https://repo.anaconda.com/miniconda/Miniconda3-latest-MacOSX-x86_64.sh
     ```

     For Windows, download the installer from `https://repo.anaconda.com/miniconda/Miniconda3-latest-Windows-x86_64.exe`.

  2. Install `Miniconda` into your home directory

     ```
     # Linux
     $ sh Miniconda3-latest-Linux-x86_64.sh -b -p $HOME/miniconda
     # MacOSX
     $ sh Miniconda3-latest-Linux-x86_64.sh
     ```

     For Windows, execute the installer and follow the steps of the installation wizard.

  3. To start using `Miniconda`, you need to make sure that it is in your `$PATH`. For example, on Linux (or MacOSX), you can simply execute

     ```
     $ export PATH=$HOME/miniconda/bin:$PATH
     ```

     You can also add it to your home `.bashrc` (or `.bash_profile` on MacOSX), so that you can skip this step the next time you start the terminal session.

  4. To test if the installation was successful, command `conda list` should return the list of installed packages.

- We will now create a new `conda` environment with the name `cv1`. We assume that the file `requirements.txt` with the dependencies we provide is in the current directory, so simply execute

  ```
  conda env create -f=requirements.txt -n cv1
  source activate cv1
  ```

  Here, the second command activates environment `cv1`.

We are now ready to execute some Python code! You can use the entry-point `main.py` to import the functions from `problem1.py` to test your code. As a warm-up, complete the following tasks in `problem1.py`.

- Load image `data/a1p1.png` using `load_image` function we provide in `main.py`
- Implement `display_image` to show the image using `matplotlib`.

  [1 point]

- Implement `save_as_npy` to save the image (a numpy array) to a binary `.npy` file (use `numpy.save()`).

  [1 point]

- Implement `load_npy` to load the previously saved file. Check that it is the same image with `display_image`

[1 point]

- Implement `mirror_horizontal` to horizontally mirror (flip) the image, *i.e.* The resulting image should revert the pixel order in the horizontal direction.

[1 point]

- Implement `display_images` to display the original and the mirrored (flipped) image from the previous step in *one* plot.

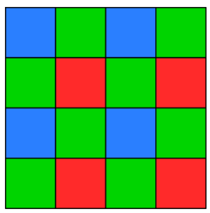
[1 point]

Submission: Please only include your writeup of `problem1.py` in your submission.

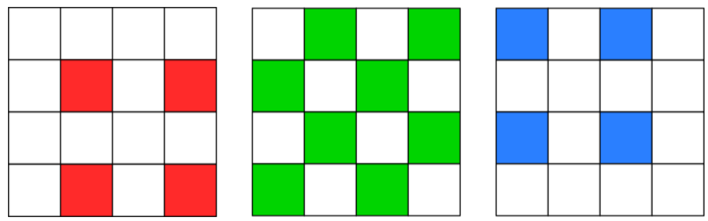
Problem 2 - Simulating Digital Zoom (15 points)

The zoom in your smartphone camera gains no optical resolution, since the properties of the camera lens remain the same. Instead, the center of the image is cropped and upscaled to the original image size. While details of this process vary on the manufacturer and can be rather involved, in this problem assignment, we will implement its simple version.

As we have seen in the lecture, many of today's digital cameras have one single chip based on a *color filter array* (CFA), where alternating sensor cells correspond to different color filters. We will assume that our camera's CFA has the Bayer pattern shown in Fig. 1.



(a) CFA layout.



(b) Red, green and blue channels. The values in the white cells are unknown.

Figure 1: Bayer RGB pattern.

Tasks:

1. **Simulating CFA response.** Your first task is to *simulate* CFA response of a given scene. Load the provided image `data/problem_2.png` of dimension $\mathrm{H} \times \mathrm{W} \times 3$, and implement method `rgb2bayer` to return *three* $\mathrm{H} \times \mathrm{W}$ arrays contraining responses of the red, green and blue filters, respectively. Fig. 1b provides an example for a 4×4 image: the white cells contain zeros, while the color cells retain the original value.

   [5 points]

2. **Scaling up and cropping.** The next task is to upscale and then crop CFA response of the bayer patterns obtained in the previous task. The method `scale_and_crop_x2` takes $\mathrm{H} \times \mathrm{W}$ filter response and returns the central crop of the *same* size. Your implementation should first upscale the input by a factor of 2 using nearest-neighbor interpolation. This is equivalent to partitioning each array cell into four cells as shown in Fig. 2. Next, take the $H \times W$ central crop of the upscaled array. For example, if the input array is 10×10, the upscaling will result in 20×20 array, and the subsequent cropping returns the central 10×10 window.

   [5 points]

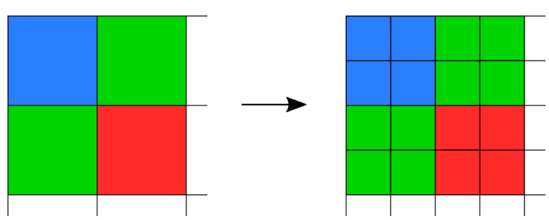


Figure 2: Nearest-neighbor upsampling.

3. **Demosaicing.** Given the upscaled and cropped version of the bayer pattern (see Fig. 2, right), the final task is *demosaicing* – assembling the RGB image from this array by interpolating the missing red, green or blue values. Your implementation will use bilinear interpolation for the red and blue, and the nearest-neighbor interpolation for the green. Method `bayer2rgb` receives the three arrays and returns the final image corresponding to $\times 2$ zoom. You should use `signal.convolve2d` from `scipy` package and find appropriate 3×3 kernels.

   [5 points]

Notes and Tips: The skeleton is given in `problem2.py`. Please, do not change the function signatures, that is the number of variables the function has as the arguments and the return values.

If you are not familiar with (bilinear) interpolation, you can find a tutorial here: `http://www.cambridgeincolour.com/tutorials/image-interpolation.htm`.

Submission: Please only include your writeup of `problem2.py` in your submission.

Problem 3 - Intrinsic Calibration (10 points)

In this exercise, you will find the projection matrix of a camera and its intrinsic and extrinsic components – the process called *camera calibration*. We provide 2D/3D point correspondences extracted from a calibration pattern similar to the one shown in the Fig. 3. You task is to use *homogeneous least squares* approach discussed in the lecture to recover the projection parameters.

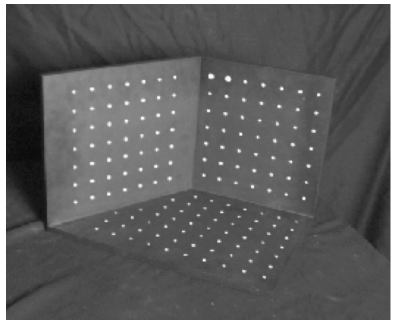


Figure 3: Example of a calibration target

Tasks:

- Implement the function `load_points` to load the 2D and 3D point correspondences from a numpy file `points.npz`. N correspondences are stored as $N \times 3$ and $N \times 4$ arrays and can be accessed via `image` and `world` keys.

  [1 point]

- Construct matrix A, as defined in the lecture slides, in the function `create_A`. Recall that this matrix emerges from the cross-product between *homogeneous* representation of the 2D points and their projected 3D correspondences.

  [2 points]

- We will now use homogeneous least squares to find the elements in our projections matrix P in function `homogeneous_Ax`. Perform Singular Value Decomposition (SVD) of P and compute the right singular vector corresponding to the smallest singular value in function. Function `homogeneous_Ax` should return the reshaped vector into 3×4 matrix, *i.e.* our projection matrix P.

  [3 points]

- Now that we have obtained matrix P, we would like to find the intrinsics K, as well as the rotation R and translation t of the camera frame w.r.t. to the world frame. Implement function `solve_KR` that uses RQ-decomposition of matrix P and returns matrices K and R. Note that RQ-decomposition is not unique, and you might need to flip the signs in K to ensure the elements (*e.g.* focal length) are positive. In this case, remember to flip the sign in matrix R accordingly.

  [2 points]

- To find the translation component t of the extrinsics, we will first find the shift c between the coordinate frames of the camera and the world, as we defined it in the lecture. Implement `solve_c` that uses SVD of P to compute its nullspace and returns c in a *non-homogeneous* form (*i.e.* 3D vector).

  [2 points]

Submission: Please only include your writeup of `problem3.py` in your submission.

Problem 4 - Sobel Operator and Steerable Filters (10 points)

In this problem you will learn more about image filtering and edge detection.

The Sobel filter is defined as

$$s_x = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}, \quad s_y = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$$

Observe that this kernel is *separable*. That is, we can re-write s_x as a convolution of two 1-D vectors as $s_x = g_x * d_x$ (equivalently for s_y). Here, g_x has the smoothing effect and is derived from the Gaussian kernel; d_x is the gradient component.

Tasks:

- Implement `gaussian` that creates 3×1 Gaussian filter using a function argument σ as the standard deviation.
  [1 point]

- Implement `diff` that creates 3×1 filter corresponding to the derivative approximation with central differences. For example, the effect of this operator in x-direction should be $f(x+1; y) - f(x-1; y)$.
  [1 point]

- Now we can construct the Sobel filter using these two components in the `create_sobel` function. This function should return 4 values. The 1^{st} and 2^{nd} are s_x and s_y as defined above, but their computation *must* use the Gaussian and the derivative components you have implemented. Note that depending on how you normalized the Gaussian in your implementation, you might need to scale the resulting kernel by a constant – let us call it z – and adjust σ, the standard deviation. Experiment with different values of z (*e.g.* 1, 2, 4) and σ (*e.g.* 0.5, 0.8, 1.2) and fill in the corresponding values as the 3^{rd} and 4^{th} return values of `create_sobel` function.
  [2 points]

- Load image `data/coins.png` and implement function `apply_sobel` that computes $\sqrt{G_x^2 + G_y^2}$, where we defined $G_x = I * s_x$ and $G_y = I * s_y$. Note that $*$ denotes convolution.
  [2 points]

- We have seen that the Sobel operator can be computed in the x- and y- direction. However, we can compute the approximation of the derivative in any direction specified by an arbitrary angle α. Suppose now that we would like to compute $G(\alpha) = G_x \cos\alpha + G_y \sin\alpha$ using a 3×3 kernel $K(\alpha)$, *i.e.* $I * K(\alpha) = G(\alpha)$. Such kernels corresponding to a linear combination of operators are called *steerable filters*. Implement `sobel_alpha` that takes some angle α and produces $K(\alpha)$ as we have just defined.
  [2 points]

- *Multiple choice question.* Experiment with applying the steerable filters to the image `data/coins.png` by varying α. Which algorithm would you use to improve the result and what will be the expected outcome? Please, read the definitions in `edge_detection` and submit your answer by returning the appropriate values.
  [2 points]

Submission: Please only include your `problem4.py` in your submission.