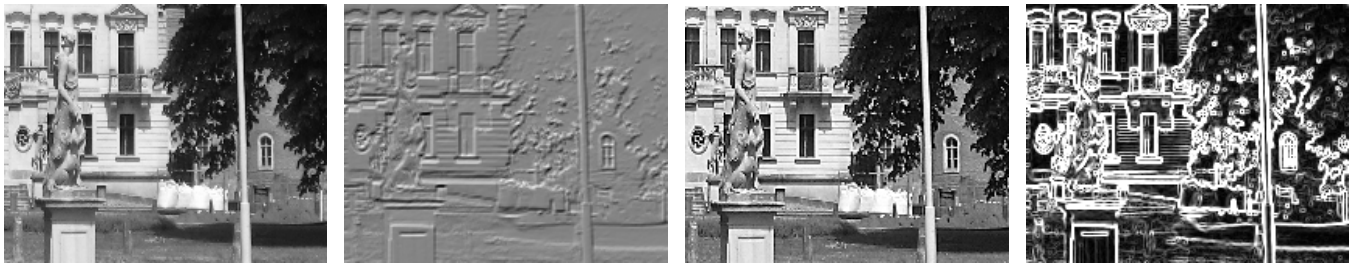
Problem Set 5 – Image processing

Our goal is to write Julia code implementing some basic image processing algorithms, especially *gamma correction, blurring, sharpening* and *edge detection.*



(a) Original picture (b) Embossed image (c) Sharpened image (w=1) (d) Edge detection

Figure 1: Image processing

Images and algorithms

After loading an image in Julia using the `load()` function from the `Images.jl` package, the image is represented as a matrix of pixels, with each pixel being an instance of the Julia type `RGB`. Objects of this type contain three components `r`, `g`, `b` encoding the *red*, *green*, and *blue* color intensities. These are typically represented as a special type of short floating-point number in the range [0, 1].

Gamma correction

An easy method to brighten or darken an image or to change the color composition is the Gamma correction with some chosen γ parameter. We apply to each RGB pixel the map

$$\begin{aligned} f &: [0,1]^3 \to [0,1]^3 \\ (r', g', b') &= f(r, g, b) = (r^{\gamma_r}, g^{\gamma_g}, b^{\gamma_b}) \end{aligned} \tag{1}$$

Choosing equal exponents $\gamma = \gamma_r = \gamma_g = \gamma_b$ will brighten or darken the image. Different exponents change the color composition.

A simple plot of $x \mapsto x^\gamma$ on $[0, 1]$ shows that the values around the fixed points 0 and 1 of this map change the least. The strongest change happens to values in the middle of the interval. For $0 < \gamma < 1$ these mid-range values are amplified while for $\gamma > 1$, they are attenuated.

Therefore, values of $\gamma < 1$ intensify a color channel whereas exponents $\gamma > 1$ suppress the color.



Figure 2: Image transformed with $\gamma_{r,g,b} = (2.4, 0.4, 0.4)$

Converting to grayscale

In order to simplify things we will will apply the more complicated image transformations only to grayscale (black&white) images. A grayscale image is represented as a matrix with entries in the range [0,1] with 0 for black and 1 for white. The following formula shows how to convert from RGB to grayscale.

$$G = 0.2126 \cdot r + 0.7152 \cdot g + 0.0722 \cdot b \tag{2}$$

Two-dimensional (2D) convolution

A beautiful intro to convolution is this 3Blue1Brown video. 2D convolution starts at 8:27min.

Discrete 2D convolution is a special kind of matrix multiplication. Let $\mathbf{A}$ be a matrix representing a color channel or the gray values of an image. Convolution acts on this matrix with a – typically much smaller – matrix $\mathbf{K}$, the *convolution kernel*.

The result of this special operation is the convolution product

$$\mathbf{A}' = \mathbf{K} * \mathbf{A}. \tag{3}$$

$$\begin{bmatrix} k_{-1,-1} & k_{-1,0} & k_{-1,1} \\ k_{0,-1} & k_{0,0} & k_{0,1} \\ k_{1,-1} & k_{1,0} & k_{1,1} \end{bmatrix}$$

We consider here only the case where $\mathbf{K}$ is a quadratic matrix with an odd number of rows, i.e. a $(2n+1) \times (2n+1)$ matrix. Let the row and column indices of this matrix run from -n to n. Then the convolution product $\mathbf{A}' = \mathbf{K} * \mathbf{A}$ is defined as

$$\mathbf{A}'_{xy} = \sum_{i=-n}^{n} \sum_{j=-n}^{n} \mathbf{K}_{ij}\, \mathbf{A}_{x-i,y-j} \tag{4}$$

The construction is based on the following idea: We want to transform the image pixel-by-pixel by calculating a new value for each pixel depending on its neighbours. For example, we want to smooth or blur the picture by replacing each pixel value by the average of the values of the 8 neighbours and the pixel itself. This is precisely convolution with the kernel

$$\mathbf{K} = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}.$$

If we want to use only the horizontal and vertical neighbours and want to give the pixel itself a higher weight in the averaging, we can use the kernel

$$\mathbf{K} = \frac{1}{8} \begin{bmatrix} 0 & 1 & 0 \\ 1 & 4 & 1 \\ 0 & 1 & 0 \end{bmatrix}.$$

With other words, equation (4) does the following: We cut the kernel matrix out of a piece of paper and move the kernel matrix over the image matrix $\mathbf{A}$, from pixel to pixel. When the center $\mathbf{K}[0,0]$ is over a pixel $\mathbf{A}[x,y]$, we calculate the new value for this pixel by multiplying each entry of the kernel with the corresponding entry of $\mathbf{A}$ at the same position and adding the products.

If you read carefully, you may have noticed two fine points:

(a) The matrix wich slides over the image is according to equation (4) not $\mathbf{K}$ itself but a *reflected* version $\mathbf{K}^R$ where $\mathbf{K}^R[i,j] = \mathbf{K}[-i,-j]$.

(b) We need a special rule for the edge pixels when the kernel matrix extends beyond the image matrix. Let us go the easiest way: imagine the picture has an extended white frame. So, every pixel outside the image should be considered as zero when applying equation (4).

Image embossing

The effect of Fig.1b where only sharp changes in pixel density are preserved is achieved by applying the following kernel:

$$\mathbf{K}_e = \begin{bmatrix} +1 & +1 & +1 \\ +1 & 0 & -1 \\ -1 & -1 & -1 \end{bmatrix} \tag{5}$$

Edge detection with Sobel filter

A simple edge detection algorithm is the Sobel filter. We calculate two convolutions, one discovering horizontal edges and one discovering vertical edges using the kernels

$$\mathbf{K}_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}, \qquad \mathbf{K}_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} \tag{6}$$

The two convolutions $\mathbf{B}_x = \mathbf{K}_x * \mathbf{A}$ and $\mathbf{B}_y = \mathbf{K}_y * \mathbf{A}$ are combined at each pixel using

$$\mathbf{B}[i,j] = \sqrt{\mathbf{B}_x^2[i,j] + \mathbf{B}_y^2[i,j]}. \tag{7}$$

Gaussian blur

Gaussian blur softens an image by using a discretized 2D Gaussian distribution as kernel. For example, a 9×9 Gauss filter $\mathbf{G}$ can be constructed easily:

```
G = [ exp(-(i^2 + j^2)/2) for i = -4:4, j = -4:4]
G = G ./ sum(G)
```

Sharpening via unsharp masking

A simple algorithm for image sharpening is the following:

(i) Blur the image using, e.g., a Gaussian kernel.

(ii) Subtract the blurred result from the original image to get high-frequency details.

(iii) Add the high-frequency details with some weight $w \sim 1$ to the original image in order to get a sharper image.

Symbolically, this can be expressed as:

$$\text{image}' = \text{image} + w \cdot (\text{image} - \text{blurred})$$

Coding Tasks

Utilize the Julia packages `Images.jl` and `ImageIO.jl` for their data types (`RGB`, etc.) and functions such as `load()` and `save()`, but implement all algorithms independently.

(i) Develop a function to apply gamma correction to an `RGB` object, allowing either distinct γ values per color channel or a uniform γ for all channels. Additionally, create a function that applies this correction to an image.

(ii) Construct a function to convert a color image into a grayscale image using equation (2). The `Images.jl` package supplies the functions `red()`,`green()` and `blue()` in order to extract color components from a RGB object.

(iii) Implement 2D discrete convolution for grayscale images. Be careful at the edges, as explained above.

(iv) Implement and evaluate various convolution kernels including embossing, Gaussian blur, Sobel edge detection, and sharpening via unsharp masking.

(v) It may be advisable to test with small images, no larger than 400×400 pixel.

(vi) The matrix of grayscale values resulting from such image transformations can be converted to an *grayscale image object* using the `Gray` function. In order to get a valid image object, all matrix entries must be in the range [0,1]. The easiest way to achive this is to `clamp`, i.e., to set negative values to 0 and values larger than one to 1. So, if M is your matrix result, you can do

```
M_img = Gray.(clamp.(M, 0, 1))
save("M.png", M_img)
```