homework 2, version 3

Homework 2 - *convolutions*

`18.S191`, Fall 2023

This notebook contains *built-in, live answer checks*! In some exercises you will see a coloured box, which runs a test case on your code, and provides feedback based on the result. Simply edit the code, run it, and the check runs again.

Feel free to ask questions!

Initializing packages

When running this notebook for the first time, this could take up to 15 minutes. Hang in there!

```
begin
    using Images, ImageIO, FileIO
    using PlutoUI
    using HypertextLiteral: @htl, @htl_str
    using OffsetArrays
end
```

Exercise 1 - *Convolutions in 1D*

As we have seen in the lectures, we can produce cool effects using the mathematical technique of **convolutions**. We input one image M and get a new image M' back.

Conceptually we think of M as a matrix. In practice, in Julia it will be a `Matrix` of color objects, and we may need to take that into account. Ideally, however, we should write a **generic** function that will work for any type of data contained in the matrix.

A convolution works on a small **window** of an image, i.e. a region centered around a given point (i, j). We will suppose that the window is a square region with odd side length $2\ell + 1$, running from $-\ell, \ldots, 0, \ldots, \ell$.

The result of the convolution over a given window, centred at the point (i, j) is a *single number*; this number is the value that we will use for $M'_{i,j}$. (Note that neighbouring windows overlap.)

To get started, in Exercise 1 we'll restrict ourselves to convolutions in 1D. So a window is just a 1D region from $-\ell$ to ℓ.

Exercise 1.1

Let's create a vector `v` of random numbers of length `n=100`.

```
n = 100
```

```julia
n = 100
```

```julia
v = rand(n)[1:50]
```

Feel free to experiment with different values!

Let's use the function `colored_line` to view this 1D number array as a 1D image.

```julia
colored_line(v)
```

```
colored_line (generic function with 2 methods)
```

👉 Try changing `n` and `v` around. Notice that you can run the cell `v = rand(n)` again to regenerate new random values.

Exercise 1.2

We need to decide how to handle the **boundary conditions**, i.e. what happens if we try to access a position in the vector `v` beyond `1:n`. The simplest solution is to assume that v_i is 0 outside the original vector; however, this may lead to strange boundary effects.

A better solution is to use the *closest* value that is inside the vector. Effectively we are extending the vector and copying the extreme values into the extended positions. (Indeed, this is one way we could implement this; these extra positions are called **ghost cells**.)

👉 Write a function `extend(v, i)` that checks whether the position i is inside `1:n`. If so, return the ith component of `v`; otherwise, return the nearest end value.

```
extend (generic function with 2 methods)
```

```julia
function extend(v::AbstractVector, i)
    if (0<i) && (i<size(v)[1])
        return v[i]
    elseif i<1
        return v[1]
    else
        return v[size(v)[1]]
    end
end
```

Some test cases:

5

```
extend([5,6,7], 1)
```

5

```
extend([5,6,7], -8)
```

7

```
extend([5,6,7], 10)
```

Got it!

You got the right answer!

```
example_vector =  [0.8, 0.2, 0.1, 0.7, 0.6, 0.4]
```

```
example_vector = [0.8, 0.2, 0.1, 0.7, 0.6, 0.4]
```

```
colored_line(example_vector)
```

- Extended with 0:

```
colored_line([0, 0, example_vector..., 0, 0])
```

- Extended with your `extend` function:

Exercise 1.3

👉 Write (or copy) the `mean` function from Homework 1, which takes a vector and returns the mean.

```
mean (generic function with 1 method)
```

```
function mean(v)

    return sum(n)/size(v)[1]
end
```

👉 Write a function `box_blur(v, l)` that blurs a vector `v` with a window of length `l` by averaging the elements within a window from $-\ell$ to ℓ. This is called a **box blur**. Use your function `extend` to handle the boundaries correctly.

Return a vector of the same size as `v`.

```
box_blur (generic function with 1 method)
```

```
function box_blur(v::AbstractArray, l)

    new_v=zeros(size(v)[1])
    window_sum=0
    for i in 1:size(v)[1]
        for j in i-l:i+l
            window_sum += extend(v,l)
        end
        new_v[i]=window_sum/2*l+1
    end
    return new_v
end
```

```
colored_line(box_blur(example_vector, 1))
```

Exercise 1.4

👉 Apply the box blur to your vector `v`. Show the original and the new vector by creating two cells that call `colored_line`. Make the parameter ℓ interactive, and call it `l_box` instead of `l` to avoid a naming conflict.

```
v =
 [0.876859, 0.918873, 0.387318, 0.296203, 0.0386661, 0.397943, 0.0769918, 0.864796, 0.429968,
```

```
v= rand(10)
```

```
colored_line(v)
```

```
@bind l Slider(1:10)
```

```
colored_line(box_blur(v,l))
```

Hint

Exercise 1.5

The box blur is a simple example of a **convolution**, i.e. a linear function of a window around each point, given by

$$v'_i = \sum_m v_{i-m} \, k_m,$$

where k is a vector called a **kernel**.

Again, we need to take care about what happens if v_{i-m} falls off the end of the vector.

👉 Write a function `convolve(v, k)` that performs this convolution. You need to think of the vector k as being *centred* on the position i. So m in the above formula runs between $-\ell$ and ℓ, where $2\ell + 1$ is the length of the vector k.

You will either need to do the necessary manipulation of indices by hand, or use the `OffsetArrays.jl` package.

```
convolve (generic function with 2 methods)
```

```
function convolve(v::AbstractVector, k)
    n_v, n_k = length(v), length(k)
    new_v = zeros(Float64, n_v)
    offset = n_k ÷ 2

    for i in 1:n_v
        window_sum = 0.0
        for j in 1:n_k
            v_index = i - offset + j - 1
            window_sum += extend(v, v_index) * k[j]
        end
        new_v[i] = window_sum
    end
    return new_v
end
```

Hint

test_convolution = [2.0, 11.0, 110.0, 1100.0, 11000.0]

```
test_convolution = let
    v = [1, 10, 100, 1000, 10000]
    k = [1, 1, 0]
    convolve(v, k)
end
```

Edit the cell above, or create a new cell with your own test cases!

Got it!

Awesome!

Exercise 1.6

👉 Define a function `box_blur_kernel(l)` which returns a *kernel* (i.e. a vector) which, when used as the kernel in `convolve`, will reproduce a box blur of length `l`.

box_blur_kernel (generic function with 1 method)

```
function box_blur_kernel(l)

    return ones(l)
end
```

box_blur_kernel_test = [1.0, 1.0, 1.0]

```
box_blur_kernel_test = box_blur_kernel(box_kernel_l)
```

Let's apply your kernel to our test vector `v` (first cell), and compare the result to our previous box blur function (second cell). The two should be identical.

```
let
    result = convolve(v, box_blur_kernel_test)
    colored_line(result)
end
```

```
let
    result = box_blur(v, box_kernel_l)
    colored_line(result)
end
```

Hint

The returned vector has the wrong dimensions.

Exercise 1.7

👉 Write a function `gaussian_kernel`.

The definition of a Gaussian in 1D is

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(\frac{-x^2}{2\sigma^2}\right),$$

or as a Julia function:

Write a function `gauss` that takes `σ` as a keyword argument and implements this function.

```
gauss (generic function with 1 method)
```

```julia
gauss(x::Real; σ=1) = 1 / sqrt(2π*σ^2) * exp(-x^2 / (2 * σ^2))
```

We need to **sample** (i.e. evaluate) this at each pixel in an interval of length $2n+1$, and then **normalize** so that the sum of the resulting kernel is 1.

```
gaussian_kernel_1D (generic function with 1 method)
```

```julia
function gaussian_kernel_1D(n; σ = 1)
    sum_gauss=0
    for i in -n:n
        sum_gauss += gauss(i,σ)
    end

    new_vec= ones(2n+1)
    for i in -n:n
        new_vec[i+n+1]=gauss(i,σ)/sum_gauss
    end


    return new_vec
end
```

Got it!

Yay ❤

```julia
colored_line(gaussian_kernel_1D(4; σ=1))
```

You can edit the cell above to test your kernel function!

Let's try applying it in a convolution.

```julia
@bind gaussian_kernel_size_1D Slider(0:6)
```

```
colored_line(test_gauss_1D_a)
```

```
test_gauss_1D_a =
 [0.876859, 0.918873, 0.387318, 0.296203, 0.0386661, 0.397943, 0.0769918, 0.864796, 0.429968,
```

```
test_gauss_1D_a = let
    k = gaussian_kernel_1D(gaussian_kernel_size_1D)

    if k !== missing
        convolve(v, k)
    end
end
```

```
colored_line(test_gauss_1D_b)
```

```
test_gauss_1D_b =
 [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
```

```
test_gauss_1D_b = let
    v = create_bar()
    k = gaussian_kernel_1D(gaussian_kernel_size_1D)

    if k !== missing
        convolve(v, k)
    end
end
```

```
create_bar (generic function with 1 method)
```

Exercise 2 - *Convolutions in 2D*

Now let's move to 2D images. The convolution is then given by a **kernel matrix** K:

$$M'_{i,j} = \sum_{k,l} M_{i-k,j-l} \, K_{k,l},$$

where the sum is over the possible values of k and l in the window. Again we think of the window as being *centered* at (i, j).

A common notation for this operation is $\star$:

$$M' = M \star K$$

Exercise 2.1

👉 Write a new method for `extend` that takes a matrix `M` and indices `i` and `j`, and returns the closest element of the matrix.

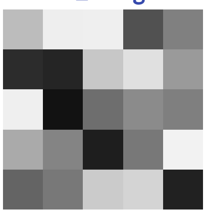
```
extend (generic function with 2 methods)
```

```julia
function extend(M::AbstractMatrix, i, j)

    clamped_i=clamp(i,1,size(M)[1])
    clamped_j=clamp(j,1,size(M)[2])
    return M[clamped_i,clamped_j]

end
```

Hint

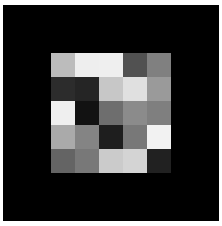
Let's test it!

`small_image =`

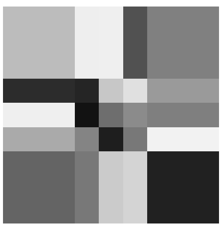


```julia
small_image = Gray.(rand(5,5))
```

- Extended with `0`:



- Extended with your `extend` function:



Got it!

Good job!

Extending Philip

```
url =
"https://user-images.githubusercontent.com/6933510/107239146-dcc3fd00-6a28-11eb-8c7b-41aaf6
```

```
url = "https://user-images.githubusercontent.com/6933510/107239146-dcc3fd00-6a28-11eb-8c7b-41aaf6618935.png"
```

```
philip_filename = "C:\\Users\\Dell\\AppData\\Local\\Temp\\jl_EtM7exqDmZ"
```

```
philip_filename = download(url) # download to a local file. The filename is returned
```

```
philip = load(philip_filename);
```

```
philip_head = philip[470:800, 140:410];
```

```
[
    extend(philip_head, i, j) for
        i in -50:size(philip_head,1)+51,
        j in -50:size(philip_head,2)+51
]
```

Exercise 2.2

👉 Implement a new method `convolve(M, K)` that applies a convolution to a 2D array `M`, using a 2D kernel `K`. Use your new method `extend` from the last exercise.

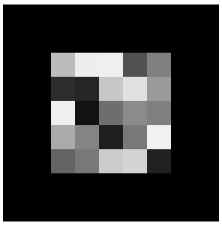
```
convolve (generic function with 2 methods)
```

```julia
function convolve(M::AbstractMatrix, K::AbstractMatrix)
    xM_d=size(M)[1]
    yM_d=size(M)[2]
    xK_d=size(K)[1]
    yK_d=size(K)[2]
    x_offset=xK_d÷2
    y_offset=yK_d÷2
    new_M=similar(M,RGB{Float64})
    for i in 1:xM_d
        for j in 1:yM_d
            window_sum=RGB(0.0,0.0,0.0)
            for k in 1:xK_d
                for l in 1:yK_d
                    window_sum += extend(M,(i-x_offset+k-1),(j-y_offset+l-1))*K[k,l]
                end
            end
            new_M[i,j]=window_sum
        end
    end
    return new_M
end
```

Hint

Let's test it out! 🎃

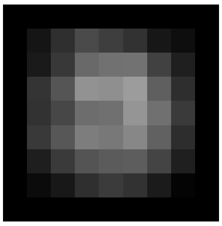
test_image_with_border =



```julia
test_image_with_border = [get(small_image, (i, j), Gray(0)) for (i,j) in Iterators.product(-1:7,-1:7)]
```

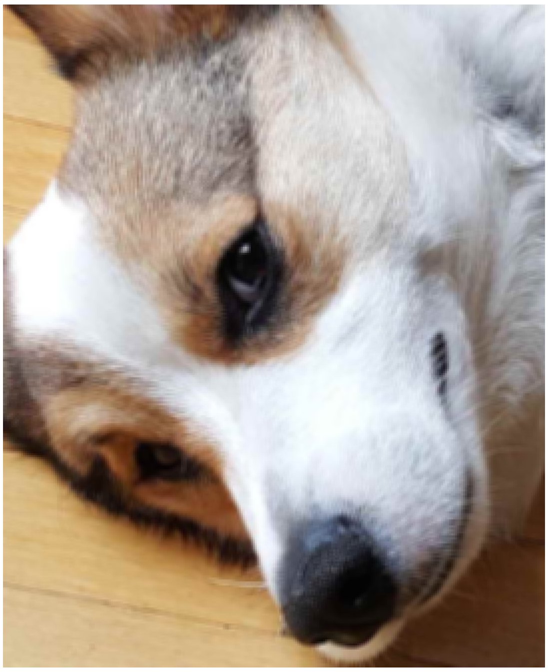
```
3×3 Matrix{Rational{Int64}}:
 1//9  1//9  1//9
 1//9  1//9  1//9
 1//9  1//9  1//9
```

```
begin
    K_test = [
        1   1  1
        1   1  1
        1   1  1
    ]
    K_test= K_test .* 1//9
end
```



```
convolve(test_image_with_border, K_test)
```

Edit `K_test` *to create your own test case!*



```
convolve(philip_head, K_test)
```

You can create all sorts of effects by choosing the kernel in a smart way. Today, we will implement two special kernels, to produce a **Gaussian blur** and a **Sobel edge detection** filter.

Make sure that you have watched the lecture about convolutions!

Exercise 2.3

The 2D Gaussian kernel will be defined using

$$G(x,y)=\frac{1}{2\pi\sigma^2}\exp\left(\frac{-(x^2+y^2)}{2\sigma^2}\right)$$

How can you express this mathematically using the 1D Gaussian function that we defined before?

```
gauss (generic function with 2 methods)
```

```julia
gauss(x, y; σ=1) = 2π*σ^2 * gauss(x; σ=σ) * gauss(y; σ=σ)
```

👉 Write a function that applies a **Gaussian blur** to an image. Use your previous functions, and add cells to write helper functions as needed!

```
gaussian_2d_kernel (generic function with 1 method)
```

```julia
function gaussian_2d_kernel(σ,l)
    sum_2d_gauss=0
    for i in 1:l, j in 1:l
        sum_2d_gauss += gauss(i,j,σ=σ)
    end
    M=ones(l,l)
    for i in 1:l,j in 1:l
        M[i,j]=gauss(i,j)/sum_2d_gauss
    end
    return M
end
```

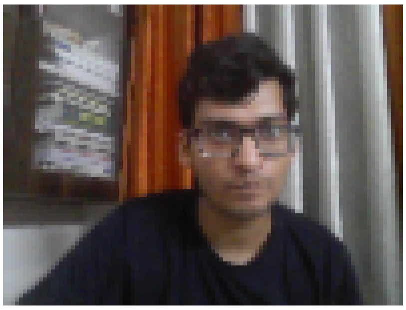
```
with_gaussian_blur (generic function with 1 method)
```

```julia
function with_gaussian_blur(image; σ, l)
    convolve(image,gaussian_2d_kernel(σ,l))
end
```

Hint

Let's make it interactive.

Enable webcam



```
with_gaussian_blur(gauss_camera_image; σ=face_σ, l=face_l)
```

1.0

```
@bind face_σ Slider(0.1:0.1:10; show_value=true)
```

3

```
@bind face_l Slider(0:20; show_value=true)
```

When you set `face_σ` to a low number (e.g. `2.0`), what effect does `face_l` have? And vice versa?

Exercise 2.4

👉 Create a **Sobel edge detection filter**.

Here, we will need to create two filters that separately detect edges in the horizontal and vertical directions, given by the following kernels:

$$G_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}; \qquad G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

We can think of these filters as derivatives in the x and y directions, as we discussed in lectures.

Then we combine them by finding the magnitude of the **gradient** (in the sense of multivariate calculus) by defining

$$G_\text{total} = \sqrt{G_x^2 + G_y^2},$$

where each operation applies *element-wise* on the matrices.

Use your previous functions, and add cells to write helper functions as needed!

```
3×3 Matrix{Int64}:
  1   2   1
  0   0   0
 -1  -2  -1
```

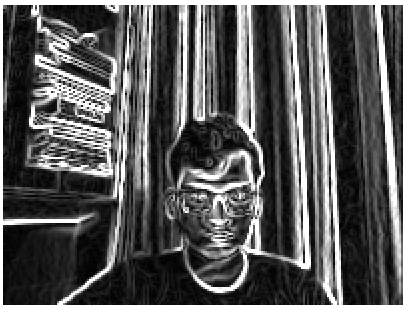
```julia
begin
    sobel_x=[1 0 -1
             2 0 -2
             1 0  -1]
    sobel_y=[1 2 1
             0 0 0
            -1 -2 -1]
end
```

```
with_sobel_edge_detect (generic function with 1 method)
```

```julia
function with_sobel_edge_detect(image)
    image_x=convolve(image,sobel_x)
    image_y=convolve(image,sobel_y)
    image_x_gray=Gray.(image_x)
    image_y_gray=Gray.(image_y)
    new_image= .√((image_x_gray .^ 2) .+ (image_y_gray .^ 2))

    return new_image
end
```

Enable webcam

```julia
@bind sobel_raw_camera_data camera_input(;max_size=200)
```



```julia
Gray.(with_sobel_edge_detect(sobel_camera_image))
```

Function library

Just some helper functions used in the notebook.

```
hint (generic function with 1 method)
```

```
almost (generic function with 1 method)
```

```
still_missing (generic function with 2 methods)
```

```
keep_working (generic function with 2 methods)
```

```
yays =
  [Great!, Yay ❤, Great! 🎉, Well done!, Keep it up!, Good job!, Awesome!, You got the right answ
```

```
correct (generic function with 2 methods)
```

```
not_defined (generic function with 1 method)
```

```
camera_input (generic function with 1 method)
```

```
process_raw_camera_data (generic function with 1 method)
```