homework 1, version 4

```
md"_homework 1, version 4_"
```

Submission by: ***Juli Osorio*** (ju@mit.edu)

Homework 1 - *convolutions*

`18.S191`, fall 2020

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.

For MIT students: there will also be some additional (secret) test cases that will be run as part of the grading process, and we will look at your notebook and write comments.

Feel free to ask questions!

```
student = (name = "Juli Osorio", kerberos_id = "ju")
```

```
# edit the code below to set your name and kerberos ID (i.e. email without @mit.edu)

student = (name = "Juli Osorio", kerberos_id = "ju")

# press the ▶ button in the bottom right of this cell to run your edits
# or use Shift+Enter

# you might need to wait until all other cells in this notebook have completed running.
# scroll down the page to see what's up
```

Let's create a package environment:

```
begin
    import Pkg
    Pkg.activate(mktempdir())
end
```

We set up Images.jl again:

```julia
begin
    Pkg.add(["Images", "ImageMagick"])
    using Images
end
```

```julia
bigbreak
```

Exercise 1 - *Manipulating vectors (1D images)*

A `Vector` is a 1D array. We can think of that as a 1D image.

example_vector = Float64[0.5, 0.4, 0.3, 0.2, 0.1, 0.0, 0.7, 0.0, 0.7, 0.9]

```julia
example_vector = [0.5, 0.4, 0.3, 0.2, 0.1, 0.0, 0.7, 0.0, 0.7, 0.9]
```

```julia
colored_line(example_vector)
```

Exerise 1.1

👉 Make a random vector `random_vect` of length 10 using the `rand` function.

random_vect = Float64[0.534029, 0.40331, 0.313604, 0.309896, 0.868636, 0.346217,

```julia
random_vect = rand(10) # replace this with your code!
```

```julia
colored_line(random_vect)
```

Got it!

Great!

Hint

[illegible]

👉 Make a function `mean` using a `for` loop, which computes the mean/average of a vector of numbers.

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

```julia
function mean(x)
    m = 0
    for i =1:length(x)
        m += x[i]
    end
    return m/length(x)
end
```

```
2.0
```

```julia
mean([1, 2, 3])
```

Got it!

Let's move on to the next section.

👉 Define `m` to be the mean of `random_vect`.

```
m = 0.4540710587407634
```

```julia
m = mean(random_vect)
```

Got it!

Keep it up!

👉 Write a function `demean`, which takes a vector `x` and subtracts the mean from each value in `x`.

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

```
function demean(x)
    dem = x .- mean(x)
    return dem
end
```

Let's check that the mean of the `demean(random_vect)` is 0:

Due to floating-point round-off error it may not be exactly 0.

```
1.6653345369377347e-17
```

```
mean(demean(copy_of_random_vect))
```

```
copy_of_random_vect = copy(random_vect); # in case demean modifies `x`
```

Exercise 1.2

👉 Generate a vector of 100 zeros. Change the center 20 elements to 1.

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

```
function create_bar()
    y = zeros(100)
    y[5:5:95] .= 1
    return y
end
```



```
colored_line(create_bar())
```

Got it!

Good job!

Exercise 1.3

👉 Write a function that turns a `Vector` of `Vector`s into a `Matrix`.

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

```
function vecvec_to_matrix(vecvec)
    A = zeros((length(vecvec),length(vecvec[1])))
    for i=1:length(vecvec)
        A[i,:] = vecvec[i]
    end
    return A
end
```

```
2×2 Array{Float64,2}:
 1.0  2.0
 3.0  4.0
```

```
vecvec_to_matrix([[1,2], [3,4]])
```

Got it!

Let's move on to the next section.

👉 Write a function that turns a `Matrix` into a `Vector` of `Vectors` .

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

```
function matrix_to_vecvec(matrix)
    v = fill(Float64[], size(matrix)[1])
    for i =1: size(matrix)[1]
        v[i] = matrix[i,:]
    end
    return v
end
```

```
Array[Float64[6.0, 7.0], Float64[8.0, 9.0]]
```

```
matrix_to_vecvec([6 7; 8 9])
```

Got it!

Good job!

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

```
begin
    colored_line(x::Vector{<:Real}) = Gray.(Float64.((hcat(x)')))
    colored_line(x::Any) = nothing
end
```

Exercise 2 - *Manipulating images*

In this exercise we will get familiar with matrices (2D arrays) in Julia, by manipulating images. Recall that in Julia images are matrices of `RGB` color objects.

Let's load a picture of Philip again.

```
philip_file = "/var/folders/1_/9n_ymmss70q94ytkkmyg_qzh0000gn/T/jl_W1ElOK"
```

```
philip_file = download("https://i.imgur.com/VGPeJ6s.jpg")
```

```
philip =
```

```
philip = let
    original = Images.load(philip_file)
    decimate(original, 8)
end
```

Hi there Philip

```
md"_Hi there Philip_"
```

Exercise 2.1

👉 Write a function **`mean_colors`** that accepts an object called `image`. It should calculate the mean (average) amounts of red, green and blue in the image and return a tuple `(r, g, b)` of those means.

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

```
function mean_colors(image)
    red = 0
    green = 0
    blue = 0
    A = channelview(image)
    n = size(A)[2]
    m = size(A)[3]
    for i = 1:n
        red += mean(A[1,i,:])
        green += mean(A[2,i,:])
        blue += mean(A[3,i,:])
    end
    rgb_avg = (red/n, green/n, blue/n)
    return rgb_avg
end
```

(0.600728, 0.547328, 0.479731)

```
mean_colors(philip)
```

Got it!

Awesome!

Exercise 2.2

👉 Look up the documentation on the `floor` function. Use it to write a function `quantize(x::Number)` that takes in a value x (which you can assume is between 0 and 1) and "quantizes" it into bins of width 0.1. For example, check that 0.267 gets mapped to 0.2.

quantize (generic function with 3 methods)

```
begin
    function quantize(x::Number)

        return floor(x*10)/10
    end

    function quantize(color::AbstractRGB)

        return RGB(quantize(color.r), quantize(color.g), quantize(color.b))
    end

    function quantize(image::AbstractMatrix)
        A = copy(image)
        for i = 1 : size(image)[1], j = 1 : size(image)[2]
            A[i,j] = quantize(image[i,j])
        end
        return A
    end
end
```

```
(0.2, 0.9)
```

```
quantize(0.267), quantize(0.91)
```

```
color =
```

```
color = RGB(0.65,0.34,0.11)
```

```
0.6
```

```
quantize(color).r
```

Got it!

Keep it up!

```
if !@isdefined(quantize)
    not_defined(:quantize)
else
    let
        result = quantize(.3)

        if ismissing(result)
            still_missing()
        elseif isnothing(result)
            keep_working(md"Did you forget to write `return`?")
        elseif result != .3
            if quantize(0.35) == .3
                almost(md"What should quantize(`0.2`) be?")
            else
                keep_working()
            end
        else
            correct()
        end
    end
end
```

Exercise 2.3

👉 Write the second **method** of the function `quantize`, i.e. a new *version* of the function with the *same* name. This method will accept a color object called `color`, of the type `AbstractRGB`.

Write the function in the same cell as `quantize(x::Number)` from the last exercise. 👆

Here, `::AbstractRGB` is a **type annotation**. This ensures that this version of the function will be chosen when passing in an object whose type is a **subtype** of the `AbstractRGB` abstract type. For example, both the `RGB` and `RGBX` types satisfy this.

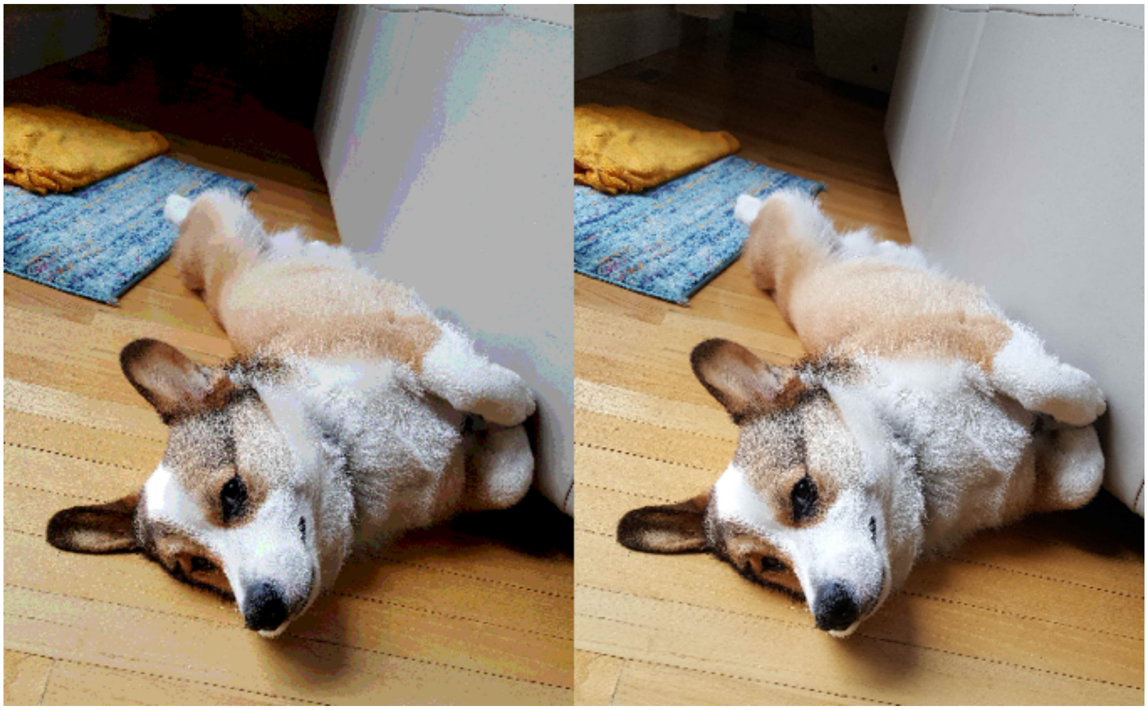
The method you write should return a new `RGB` object, in which each component (r, g and b) are quantized.

Exercise 2.4

👉 Write a method `quantize(image::AbstractMatrix)` that quantizes an image by quantizing each pixel in the image. (You may assume that the matrix is a matrix of color objects.)

Write the function in the same cell as `quantize(x::Number)` from the last exercise. 👆

Let's apply your method!



```
[quantize(philip) philip]
```

Exercise 2.5

👉 Write a function `invert` that inverts a color, i.e. sends (r, g, b) to $(1 - r, 1 - g, 1 - b)$.

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

```
begin
    function invert(color::AbstractRGB)

        return RGB(1-color.r, 1- color.g, 1-color.b)
    end

    function invert(image::AbstractMatrix)
        inverted = copy(image)
        for i=1:size(image)[1], j = 1:size(image)[2]
            inverted[i,j] = invert(image[i,j])
        end
        return inverted
    end
end
```

Let's invert some colors:

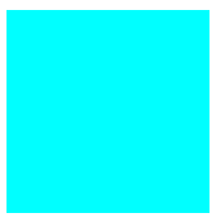
```
black =
```

```
black = RGB(0.0, 0.0, 0.0)
```

```
invert(black)
```

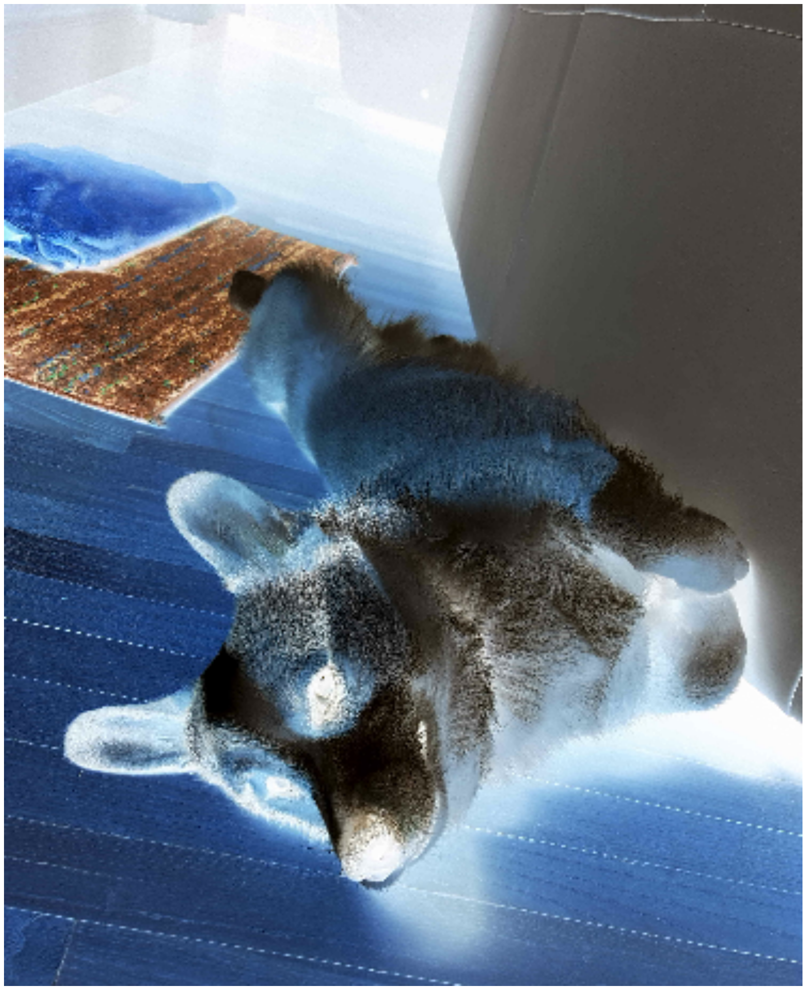
red =

```
red = RGB(1.0, 0.0, 0.0)
```



```
invert(red)
```

Can you invert the picture of Philip?



```
invert(philip)
```

Exercise 2.6

👉 Write a function `noisify(x::Number, s)` to add randomness of intensity s to a value x, i.e. to add a random value between $-s$ and $+s$ to x. If the result falls outside the range $(0, 1)$ you should "clamp" it to that range. (Note that Julia has a `clamp` function, but you should write your own function `myclamp(x)`.)

```
noisify (generic function with 3 methods)
```

```
begin
    function myclamp(x::Number,a,b)
        if (x >a) && (x < b )
            y = x
        elseif x >= b
            y = b
        else
            y = a
        end
        return y
    end

    function noisify(x::Number, s)
        y = myclamp(x + 2*s*rand() - s,0,1)

        return y
    end

    function noisify(color::AbstractRGB, s)

        return RGB(noisify(color.r,s), noisify(color.g, s), noisify(color.b, s))
    end

    function noisify(image::AbstractMatrix, s)
        noise_image = copy(image)
        for i=1:size(image)[1], j = 1:size(image)[2]
            noise_image[i,j] = noisify(image[i,j],s)
        end
        return noise_image
    end
end
```

Hint

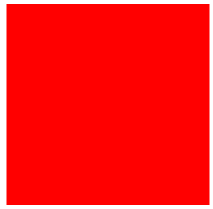
[illegible]

👉 Write the second method `noisify(c::AbstractRGB, s)` to add random noise of intensity s to each of the (r, g, b) values in a colour.

Write the function in the same cell as `noisify(x::Number)` from the last exercise. 👆

0.0

```julia
@bind color_noise Slider(0:0.01:1, show_value=true)
```



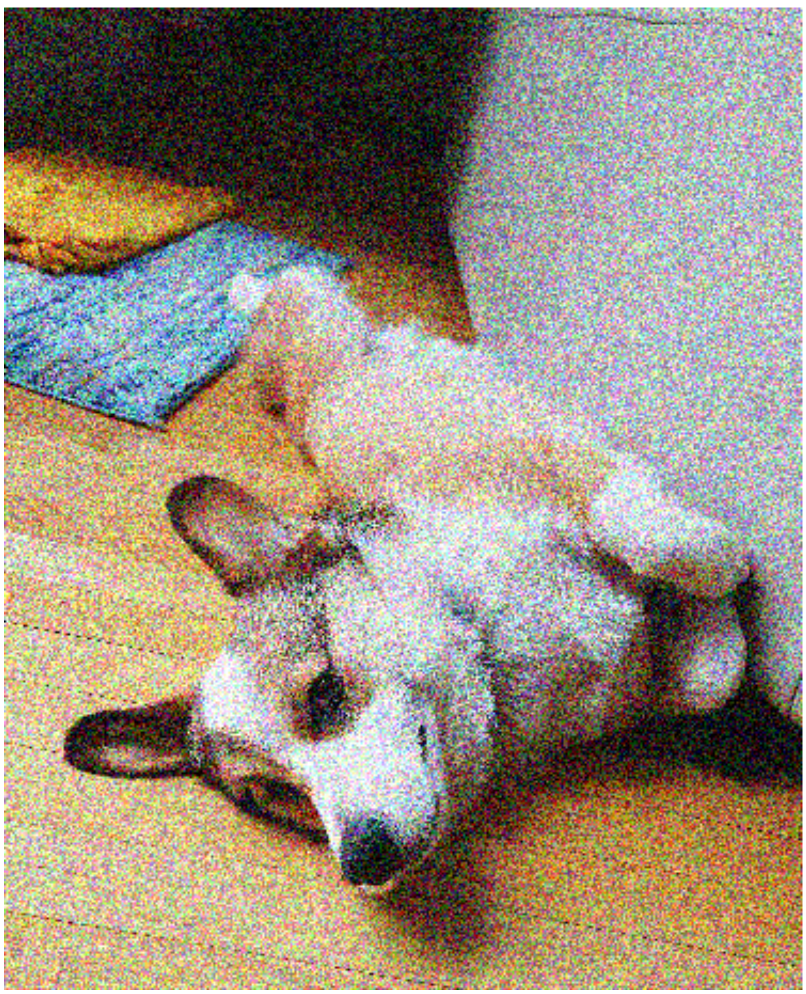
```julia
noisify(red, color_noise)
```

👉 Write the third method `noisify(image::AbstractMatrix, s)` to noisify each pixel of an image.

Write the function in the same cell as `noisify(x::Number)` from the last exercise. 👆

0.32

```julia
@bind philip_noise Slider(0:0.01:8, show_value=true)
```



```
noisify(philip, philip_noise)
```

👉 For which noise intensity does it become unrecognisable?

You may need noise intensities larger than 1. Why?

`answer_about_noise_intensity =`
The image is unrecognisable with intensity bigger than 1.5 approximately

```
begin
    Pkg.add("PlutoUI")
    using PlutoUI
end
```

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

```
decimate(image, ratio=5) = image[1:ratio:end, 1:ratio:end]
```

Exercise 3 - *Convolutions*

As we have seen in the videos, 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 let's restrict ourselves to convolutions in 1D. So a window is just a 1D region from $-\ell$ to ℓ.

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

```
n = 100
```

```
n = 100
```

```
v = Float64[0.990099, 0.756639, 0.070899, 0.0832139, 0.772248, 0.924601, 0.07760
```

```
v = rand(n)
```

Feel free to experiment with different values!

Exercise 3.1

You've seen some colored lines in this notebook to visualize arrays. Can you make another one?

👉 Try plotting our vector `v` using `colored_line(v)`.


```
colored_line(v)
```

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

Exercise 3.2

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 1 method)
```

```
function extend(v, i)
    return v[myclamp(i,1, length(v))]
end
```

Some test cases:

```
0.9900988383831115
```

```
0.9900988383831115
```

```julia
extend(v, -8)
```

```
0.8595618272822978
```

```julia
extend(v, n + 10)
```

Extended with 0:

Extended with your `extend`:

Got it!

Let's move on to the next section.

Exercise 3.3

👉 Write a function `blur_1D(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**.

```
blur_1D (generic function with 1 method)
```

```julia
function blur_1D(v, l)
    v_blured = zeros(length(v))
    for i = 1 : length(v)
        v_blured[i] = mean([extend(v,i+k) for k =-l:l])
    end
    return v_blured
end
```

Exercise 3.4

👉 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 just `l` to avoid a variable naming conflict.

0

```
@bind l_box Slider(0:1:10, show_value=true)
```



```
colored_line(blur_1D(v,l_box))
```

```
colored_line(v)
```

Hint

Exercise 3.5

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

$$v'_i = \sum_n v_{i-n} \, k_n,$$

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

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

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

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

```julia
function convolve_vector(v, k)
    n = length(k)
    conv_vector = zeros(length(v))
    for i =1 : length(v)
        conv_vector[i] = sum([extend(v,i+(2*j -n-1)÷2)*k[j] for j =1:n])
    end
    return conv_vector
end
```

Hint

```julia
colored_line(test_convolution)
```

test_convolution = Float64[1.0, 10.0, 100.0, 1000.0, 10000.0]

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

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

Got it!

Great! 🎉

Exercise 3.6

👉 Write a function `gaussian_kernel`.

The definition of a Gaussian in 1D is

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

We need to **sample** (i.e. evaluate) this at each pixel in a region of size n^2, and then **normalize** so that the sum of the resulting kernel is `1`.

For simplicity you can take $\sigma = 1$.

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

```julia
function gauss(x::Number,s=1)
    g = exp((-x^2)/(2*s^2))/(sqrt(2*pi*s^2))
    return g/sum(g)
end
```

```
Float64[1.0, 1.0, 1.0, 1.0, 1.0]
```

```julia
[gauss(k) for k=-2:2]
```

```
ker = Float64[0.0544887, 0.244201, 0.40262, 0.244201, 0.0544887]
```

```julia
ker = Kernel.gaussian((1,))
```

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

```julia
function gaussian_kernel(n)
    k = [gauss(j) for j= -n:n]
    return k
end
```

Let's test your kernel function!

```julia
@bind gaussian_kernel_size_1D Slider(1:10) # change this value, or turn me into a slider!
```



```
test_gauss_1D_a = Float64[1.47137, 1.25094, 1.02681, 1.49214, 1.52475, 2.11766,
```

```
test_gauss_1D_a = let
    v = random_vect
    k = gaussian_kernel(gaussian_kernel_size_1D)

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



```
colored_line(test_gauss_1D_b)
```

```
test_gauss_1D_b = Float64[0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 0.0, 0.0, 1.0, 1.0, 1.
```

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

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

Exercise 4 - *Convolutions of images*

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 $*$:

$$M' = M * K.$$

Exercise 4.1

👉 Write a function `extend_mat` that takes a matrix `M` and indices `i` and `j`, and returns the closest element of the matrix.

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

```
function extend_mat(M::AbstractMatrix, i, j)
    (m,n) = size(M)
    return M[myclamp(i,1,m), myclamp(j,1,n)]

end
```

syntax: invalid identifier name "?"

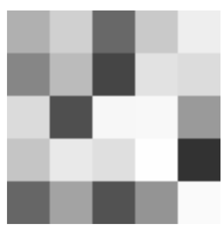
1. **top-level scope** @ *none:1*

```
?
```

Hint

Let's test it!

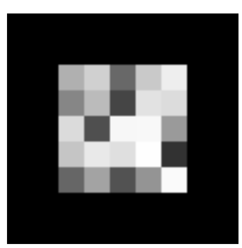
`small_image =`



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

Extended with `0`:

```
md"Extended with `0`:"
```



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

Extended with your `extend`:

```
md"Extended with your `extend`:"
```

1

```
myclamp(1,1,3)
```

```
5×5 Array{Int64,2}:
 1  1  1  2  2
 1  1  1  2  2
 1  1  1  2  2
 3  3  3  4  4
 3  3  3  4  4
```

```
begin
    a = [1 2 ; 3 4]
    [extend_mat(a, i,j) for (i,j) in Iterators.product(-1:3,-1:3)]
end
```

```
[extend_mat(small_image, i, j) for (i,j) in Iterators.product(-1:7,-1:7)]
```

Got it!

Well done!

```
if !@isdefined(extend_mat)
    not_defined(:extend_mat)
else
    let
        input = [42 37; 1 0]
        result = extend_mat(input, -2, -2)

        if ismissing(result)
            still_missing()
        elseif isnothing(result)
            keep_working(md"Did you forget to write `return`?")
        elseif result != 42 || extend_mat(input, -1, 3) != 37
            keep_working()
        else
            correct()
        end
    end
end
```

```
let
    philip_head = philip[250:430,110:230]
    [extend_mat(philip_head, i, j) for (i,j) in Iterators.product(-50:size(philip_head,1)+51,
(-50:size(philip_head,2)+51))]
end
```

Exercise 4.2

👉 Implement a function `convolve_image(M, K)`.

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

```julia
function convolve_image(M::AbstractMatrix, K::AbstractMatrix)
    M_conv = similar(M) # had to copy M so the convol is of the same type
    row, col = size(K)
    for i = 1:size(M)[1], j = 1:size(M)[2]
        M_conv[i,j] = sum([extend_mat(M, i + (2*k -row - 1)÷2, j + (2*l - col -1)÷2)*extend_mat(K,k,l) for (k,l) in Iterators.product(1:row, 1:col)])
    end
    return M_conv
end
```

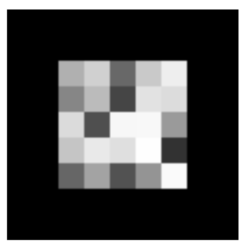
```
0-dimensional Array{DataType,0}:
Array{RGBX{Normed{UInt8,8}},2}
```

```julia
fill(Array{RGBX{Normed{UInt8,8}},2})
```

Hint

Let's test it out! 🎃

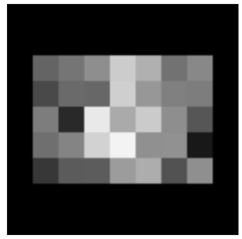
test_image_with_border =



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

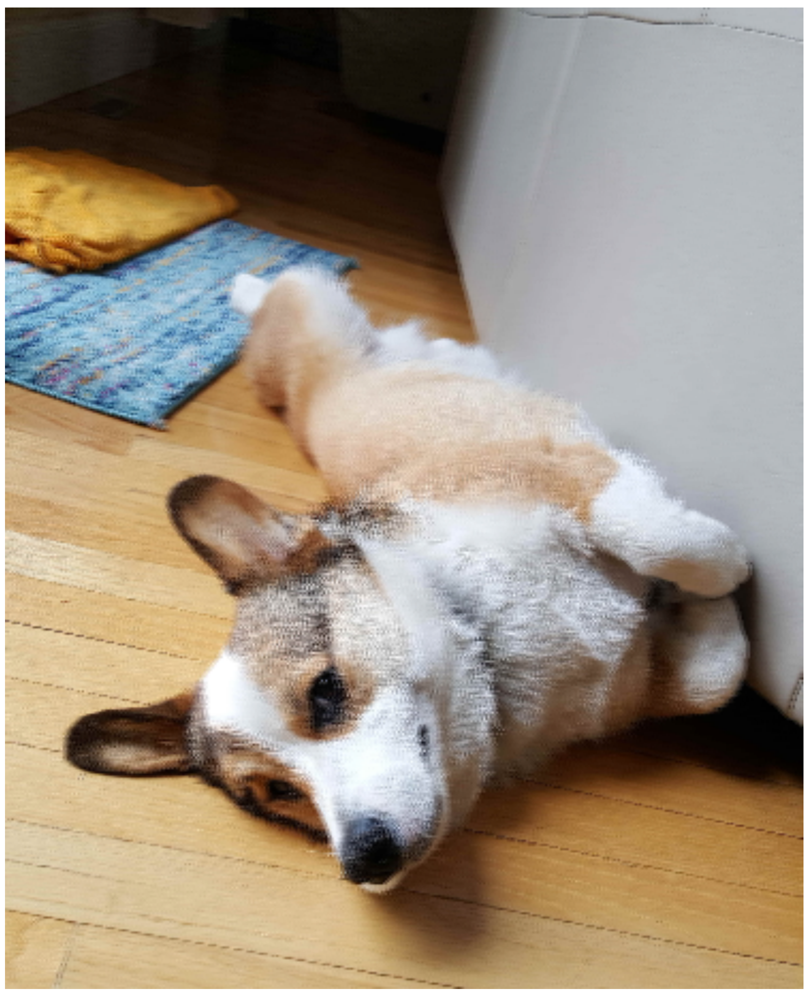
```
K_test = 3×3 Array{Float64,2}:
          0.0  0.0  0.0
          0.5  0.0  0.5
          0.0  0.0  0.0
```

```
K_test = [
    0   0  0
    1/2 0  1/2
    0   0  0
]
```

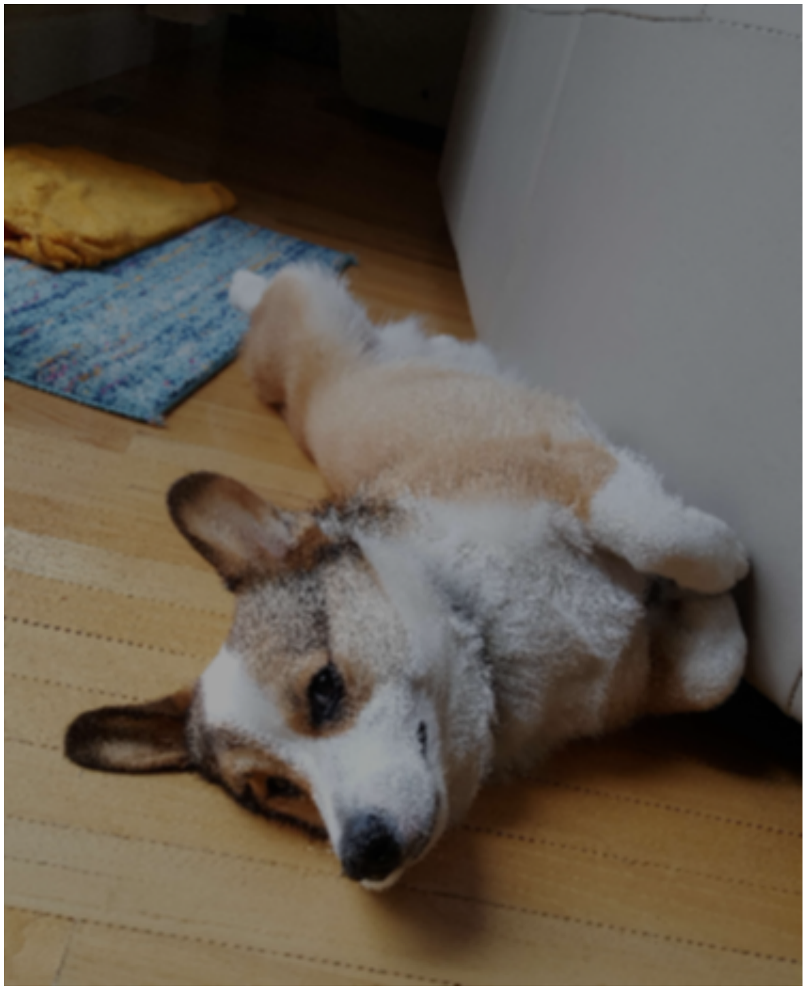


```
convolve_image(test_image_with_border, K_test)
```

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



```
convolve_image(philip, K_test)
```



```
begin
    K_test1 = [1/25 1/25 1/25 1/25;1/25 1/25 1/25 1/25;1/25 1/25 1/25 1/25;1/25 1/25 1/25 1/25]
    convolve_image(philip,K_test1)
end
```

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 detect** filter.

Make sure that you have watched **the lecture** about convolutions!

Exercise 4.3

👉 Apply a **Gaussian blur** to an image.

Here, the 2D Gaussian kernel will be defined as

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

```
gaussian_2D = 5×5 Array{Float64,2}:
 0.05854983152431917    0.013064233284684921   …  3.5974259813700723e-7
 0.013064233284684921   0.0029150244650281935     8.026942353452266e-8
 0.0010723775711956546  0.0002392797792004706     6.5889155203724425e-9
 3.238299669088984e-5   7.225623237724322e-6      1.9896800830595185e-10
 3.5974259813700723e-7  8.026942353452266e-8      2.2103349154917858e-12
```

```julia
gaussian_2D = [exp(-(i^2+j^2)/2)/(2*pi) for (i,j) in Iterators.product(1:5,1:5)]
```

```
5×5 OffsetArray(::Array{Float64,2}, -2:2, -2:2) with eltype Float64 with indices -2:2×-2:2:
 0.002969016743950497  0.013306209891013651  …  0.002969016743950497
 0.013306209891013651  0.059634295436180124     0.013306209891013651
 0.02193823127971464   0.09832033134884574      0.02193823127971464
 0.013306209891013651  0.059634295436180124     0.013306209891013651
 0.002969016743950497  0.013306209891013651     0.002969016743950497
```

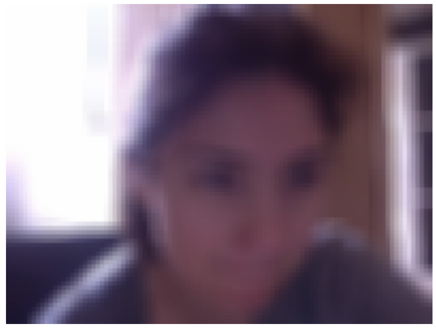
```julia
Kernel.gaussian((1,1))
```

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

```julia
function with_gaussian_blur(image)
    gaussian_2D = [gauss(i,0.75).*gauss(j,0.75) for (i,j) in Iterators.product(1:5,1:5)]
    kernel = gaussian_2D/sum(gaussian_2D)
    conv_image = convolve_image(image, kernel)
    return conv_image
end
```

Let's make it interactive. 💫

Enable webcam



```
with_gaussian_blur(gauss_camera_image)
```

```
with_gaussian_blur(philip)
```

Exercise 4.4

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

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

$$G_x = \left(\begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \otimes [1\ 0\ -1] \right) * A = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} * A$$

$$G_y = \left(\begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} \otimes [1\ 2\ 1] \right) * A = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * A$$

Here A is the array corresponding to your image. We can think of these as derivatives in the x and y directions.

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

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

For simplicity you can choose one of the "channels" (colours) in the image to apply this to.

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

```julia
function with_sobel_edge_detect(image)
    Gx = convolve_image(image,[1 0 -1; 2 0 -2; 1 0 -1])
    Gy = convolve_image(image,[1 2 1; 0 0 0; -1 -2 -1])
    conv_image = sqrt.(Gx.^2 + Gy.^2)
    return conv_image
end
```

Enable webcam

```
RGB.(with_sobel_edge_detect(sobel_camera_image))
```

Exercise 5 - *Lecture transcript*

(MIT students only)

Please see the Canvas post for transcript document for week 1 **here**.

We need each of you to correct about 100 lines (see instructions in the beginning of the document.)

👉 Please mention the name of the video and the line ranges you edited:

```
md"""
## **Exercise 5** - _Lecture transcript_
_(MIT students only)_

Please see the Canvas post for transcript document for week 1 [here](https://canvas.mit.edu/courses/5637/discussion_topics/27880).

We need each of you to correct about 100 lines (see instructions in the beginning of the document.)

👉 Please mention the name of the video and the line ranges you edited:
"""
```

lines_i_edited =
Convolution, lines 100-0 (*for example*)

```
lines_i_edited = md"""
Convolution, lines 100-0 (_for example_)
"""
```

```
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 =  MD[Great!,  Yay ❤,  Great! 🎉,  Well done!,  Keep it up!,  Good job!,  Awesome!
```

```
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)
```