Assignment 2: High Dynamic Range (HDR) Images

Guilherme Gomes Haetinger - 00274702

Proposal:
The goal of this assignment is to familiarize the students with high dynamic range (HDR) images. For this, you will be exploring and creating HDR images from a sequence of low dynamic range (LDR) images captured with multiple exposures. In order to display them on LDR displays, you will be also implementing some simple tone mapping operator. The assignment will also familiarize you with the concept of camera response curves used for mapping exposure values to pixel values.

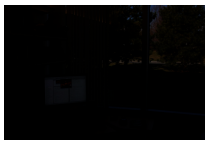
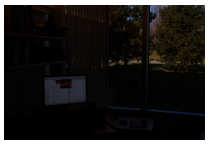
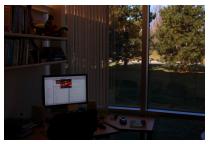
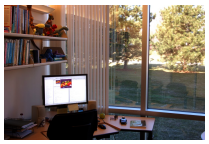
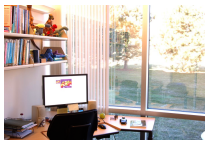
Introduction

HDR images are used by most people today to take images without any over/underexposed areas. Most phones already have this functionality and allow us to create beautiful portrait or landscape photos.

I have decided to use Julia for this assignment. I used Matlab to use the **makehdr** command needed for Task 5.

Task 1

Using the **FastStone** image viewer, I was able to retrieve the exposure times of the input images:

Image	Exposure Time
	$\frac{1}{30}$ s
	0.1s
	0.33s
	0.63s
	1.3s
	4s

I was also able to notice how detailed the EXIF metadata for the `.CR2` images from the last assignment were. They contained `date`, `camera model`, `flash`, `focal length`, `exposure program (Aperture priority)`, `FNUM`, `Metering`, `Exposure time`, `ISO` and even more information.

```
exposure_table =   [0.0333333, 0.1, 0.33, 0.63, 1.3, 4.0]
```

Task 2

Having the previous exposure time table set, I was able to insert them into the **HDR Shop** tool to calculate the **CRC** (Camera Response Curve). We can, thus, read the curve here in Julia.

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

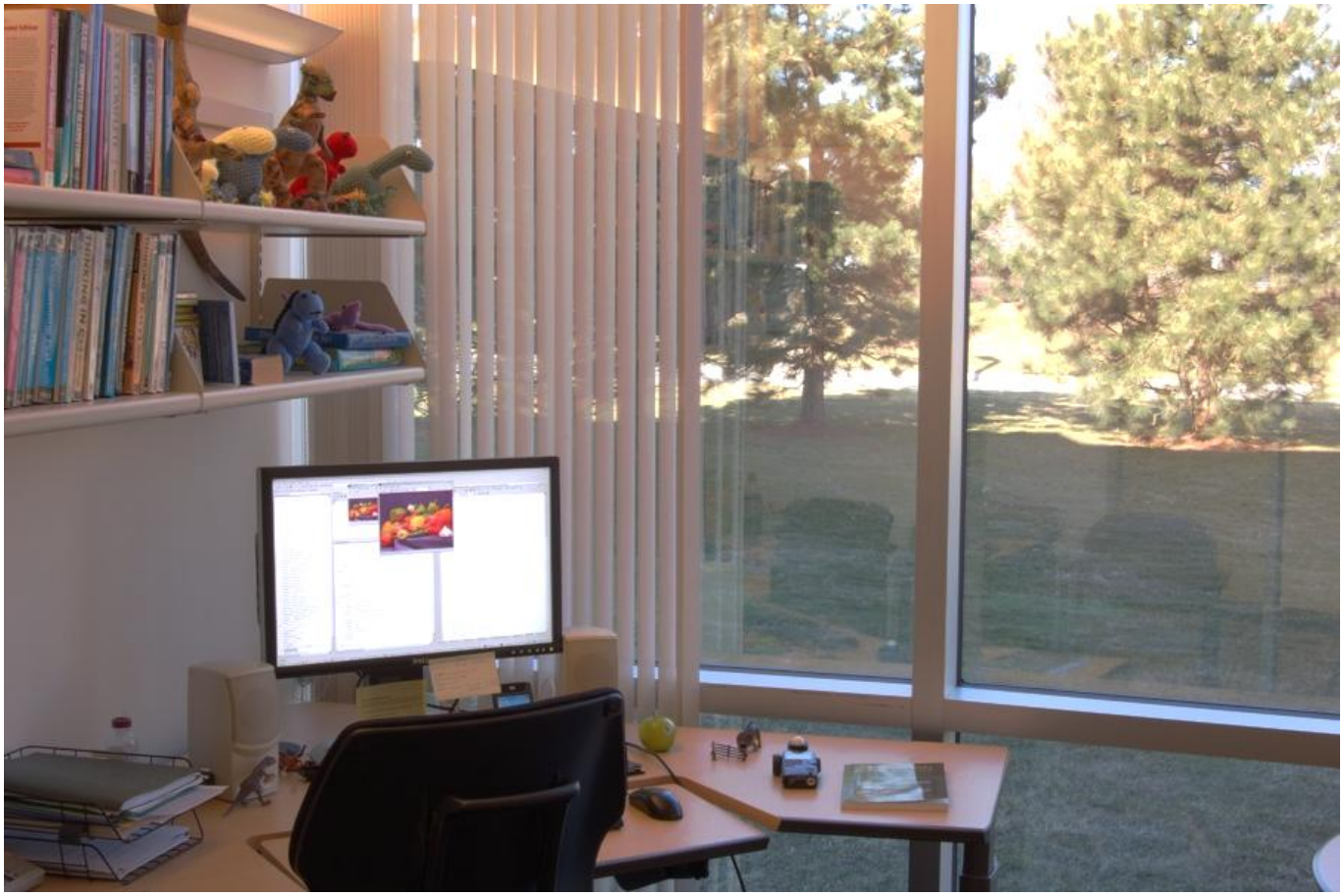
```
function read_log_curve(filename)
    content = readlines("./res/curve.m")[2:end-2]
    arrays = split.(content, " ") .|> arr -> inverted_parse.(arr, Float64)
    return exp.(reshape(reduce(vcat, arrays), (3, :)))
end
```

```
curve =
3×255 Matrix{Float64}:
 0.00994892  0.0222018  0.0271387  0.0335629  …  7.71267  8.68017  9.21806  9.21806
 0.00573321  0.0129325  0.017334   0.0233878     7.62424  8.4422   9.81282  9.81282
 0.00813407  0.016034   0.0219561  0.0311741     7.85469  7.89386  9.80821  9.80821
```

```
curve = read_log_curve("./res/curve.m")
```

Task 3

Still using **HDR Shop**, we can retrieve the following HDR image:



Task 4

Now for the main part: retrieving the HDR image through the curve. First of all, we need to load and gamma-decode our images:

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

```
function gamma_decode(image, γ=1/1.75)
    channels = channelview(image)
    R = channels[1, :, :] .^ (1/γ)
    G = channels[2, :, :] .^ (1/γ)
    B = channels[3, :, :] .^ (1/γ)

    return colorview(RGB, R, G, B)
end
```

```
images_no_decode = readdir(image_dir) .|> name -> joinpath(image_dir, name) .|> load;
```

```
images = images_no_decode .|> gamma_decode;
```

not decode :

γ-decoded :

Clamping method

Now that we need to apply the curve to the image channels and put them together with a mean, my first thought on how to get rid of extremes and fit to the curve was to clamp values from the channels to $[1, 255]$. Now, this means we consider every pixel in every image, not discarding the over/underexposed pixels.

Hence, we can achieve this by clamping every image before applying the curve. The following function does that and later on applies the curve and divides every value by the image exposure to retrieve the irradiance value of a pixel.

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

```
function get_image_irradiance(image, curve, exposure)
    channels = channelview(image)
    floor_int(x) = floor(Int64, x)
    clamp_255(x) = clamp(x, 1, 255)
    R = channels[1, :, :] .* 255 .+ 1 .|> floor_int .|> clamp_255
    G = channels[2, :, :] .* 255 .+ 1 .|> floor_int .|> clamp_255
    B = channels[3, :, :] .* 255 .+ 1 .|> floor_int .|> clamp_255

    R_crc = curve[1, :][R]
    G_crc = curve[2, :][G]
    B_crc = curve[3, :][B]

    R_n = R_crc ./ exposure
    G_n = G_crc ./ exposure
    B_n = B_crc ./ exposure

    colorview(RGB, R_n, G_n, B_n)
end
```

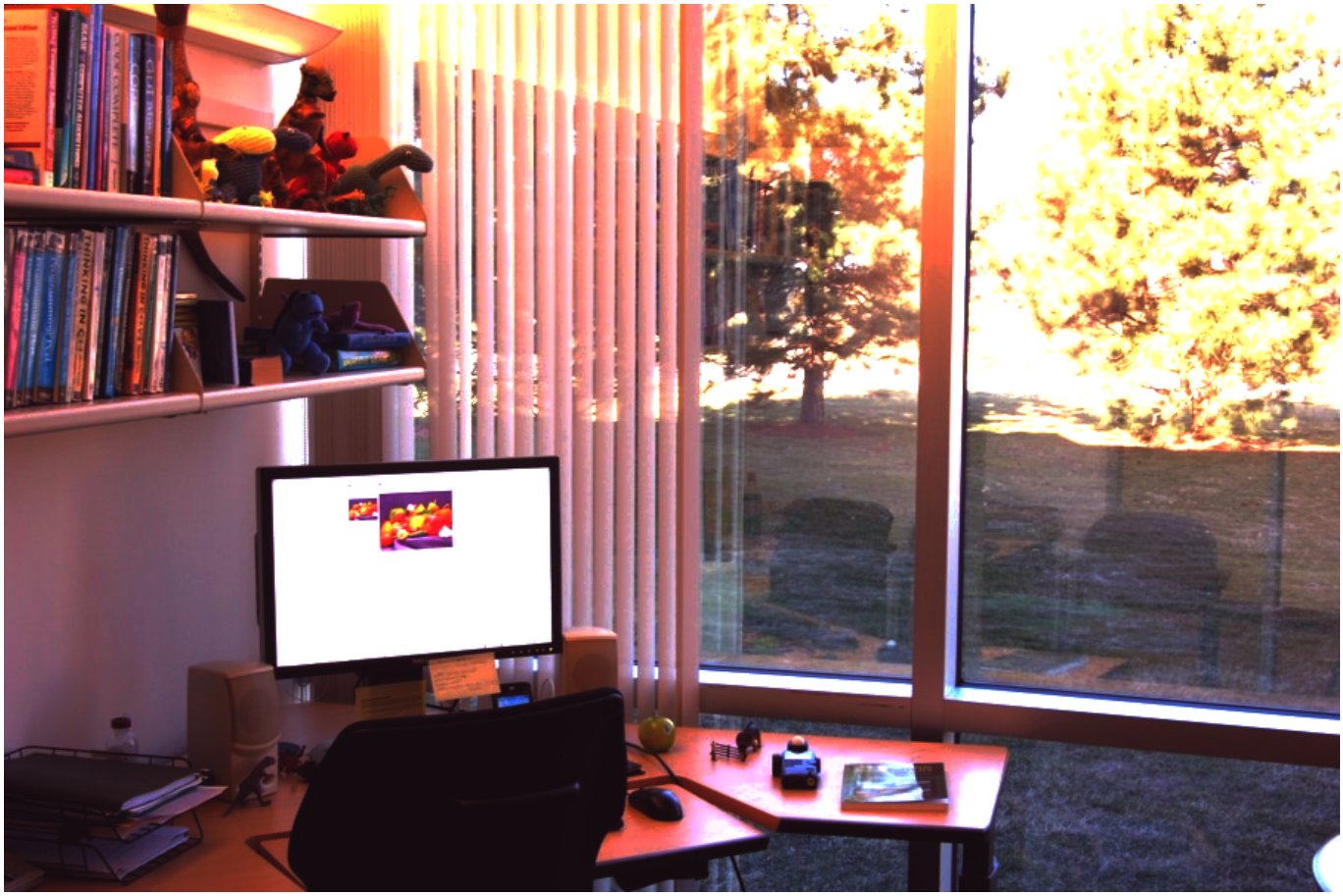
We can, then, compute this for every image and calculate the mean:

```
md"""
We can, then, compute this for every image and calculate the mean:
"""
```

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

```
function clamping_curve_map(curve, images, exposures)
    mean([get_image_irradiance(images[i], curve, exposures[i]) for i ∈
    1:length(images)])
end
```

This results in the following HDR image (~70ms):



- `clamping_curve_map(curve, images, exposure_table)`

Discarding method

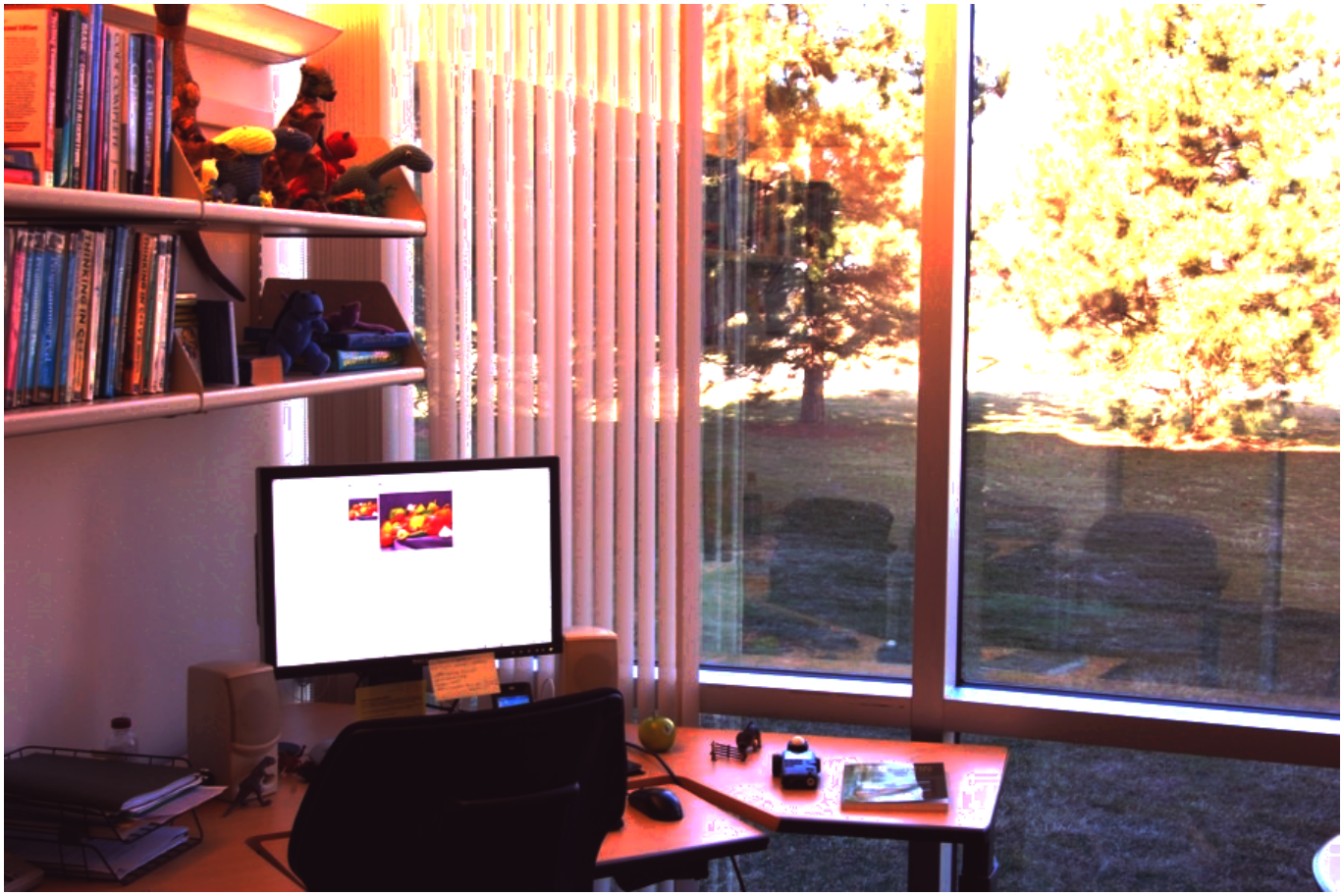
After using clamping, I decided to compare it with the "discard under/overexposed" pixels method using a for loop and recording which pixels should be accounted for:

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

```
function discarding_curve_map(curve, images, exposures)
    sz = size(images[1])
    non_discarded = zeros(sz)
    accumulator = zeros((3, sz...))
    (rows, cols) = sz
    for (idx, image) ∈ enumerate(images)
        for row ∈ (1:rows)
            for col ∈ (1:cols)
                discard = 0
                final_color = [0., 0., 0.]
                for channel ∈ [(red, 1), (green, 2), (blue, 3)]
                    color = first(channel)(image[row, col])
                    color = round(Int64, color * 255 + 1)
                    clamped_color = clamp(color, 1, 255)
                    curved = curve[last(channel), clamped_color]
                    final_color[last(channel)] += curved ./ exposures[idx]
                    if (color < 1 || color > 255)
                        discard += 1
                    end
                end
                if (discard < 3)
                    accumulator[:, row, col] += final_color
                    non_discarded[row, col] += 1
                end
            end
        end
    end

    return colorview(RGB, accumulator[1, :, :], accumulator[2, :, :], accumulator[3,
    :, :]) ./ non_discarded
end
```

This method results in an almost exact result with a much higher computation time (~10s):

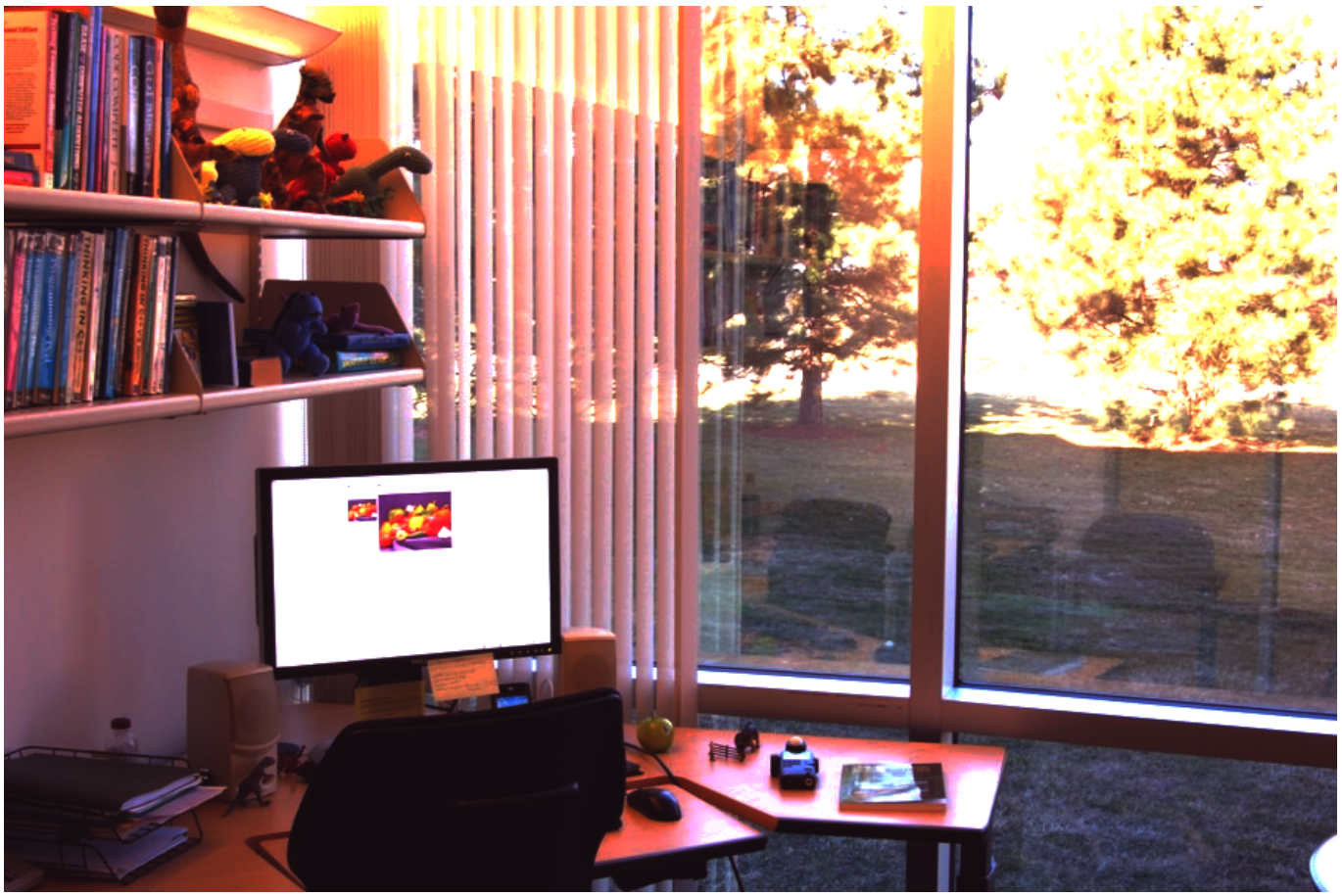


- `discarding_curve_map(curve, images, exposure_table)`

Playing with exposure

Now that we've retrieved the HDR image from the image sequence, we can multiply it by whatever exposure we want. Increasing exposure gives us a sequence like the following:

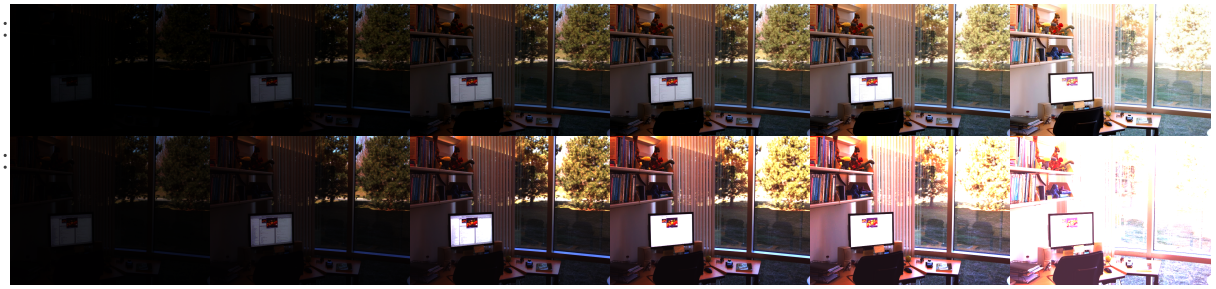
`hdr =`



```
hdr = clamping_curve_map(curve, images, exposure_table)
```

image sequence :

exposure multiplied images :



Although this isn't a perfect reproduction of the original images, we are getting all of this from the same image, which is very impressive.

Tonemap command

Using Matlab's *tonemap* command, we get the following image:

Task 5

Using Matlab's **makehdr** command with the image sequence, we get the following image:

```
save_to_matlab(hdr, "./out/office.mat")
```

a =

This image is much different from our HDR image. It seems to be extremely overexposed while ours seems more saturated and still shows every component of the picture (tree, computer, chair, ...). This might be because of the fact that I saved Matlab's image to PNG, removing some of the data particularities (probably clamping a lot of the data to **1.0**) and resulting on this predominantly white image.

Obviously, this image isn't supposed to be used as the final result. We're still missing the the tonemapping step. Applying **tonemap** to it, we get the following image:

Even the tonemapping result seems different. While mine seems to have more contrast, Matlab's seems more uniform. This could be a product of my custom γ value on γ-decoding.

Task 6

Using the algorithm learned in class, we aim to find the proper global luminance for each pixel and replace the currently present one with it. To do so, we start by retrieving the luminance of the linearized intensity with the following function:

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

```
function luminance(image)
    channels = channelview(image)
    R = channels[1, :, :]
    G = channels[2, :, :]
    B = channels[3, :, :]

    return (0.299 .* R) .+ (0.587 .* G) .+ (0.114 .* B)
end
```

```
L = luminance(hdr);
```

Once we have the $L(x, y)$ image, we can get the **Average Log Luminance** by computing

$$\tilde{L} = e^{\frac{1}{N}\sum \log(L(x,y)+\delta)}$$

```
δ = 1.0e-8
```

```
L̃ = 0.44795768895556287
```

```
L̃ = exp(mean(log.(L .+ δ)))
```

Moreover, we calculate the **Scaled Luminance** using the suggested $\alpha = 0.18$:

$$L_s = \frac{\alpha * L}{\tilde{L}}$$

```
α = 0.18
```

```
Lₛ = (α * L)/L̃;
```

Finally, we move L_s to a period $[0, 1)$ by computing the global operator:

```
LG = Ls ./ (Ls .+ 1);
```

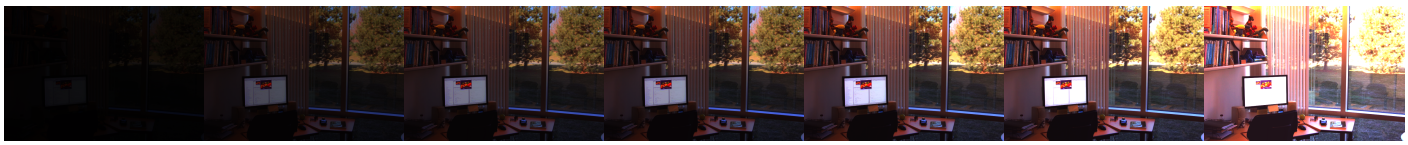
We define the saturation as 1 as we replace the **linearized intensity** by the **Global operator**

```
saturation = 1
```

tonemapped =

```
tonemapped = LG .* hdr ./ L .* saturation
```

Playing with saturation, we get the following sequence:



We notice the image is still a bit dark, but increasing saturation might make things look overexposed. This is because we are missing a step: γ-encoding. reintegrating γ is much like we did in the previous assignment:

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

```
function gamma_encode(image, γ=1/1.75)
    channels = channelview(image)
    R = channels[1, :, :] .^ γ
    G = channels[2, :, :] .^ γ
    B = channels[3, :, :] .^ γ

    return colorview(RGB, R, G, B)
end
```

`gamma_encoded_tonemapped =`

```
gamma_encoded_tonemapped = gamma_encode(tonemapped) .|> RGB
```

Needless to say, my final result is much more colorful and beautiful than Matlab's. I also thing this is because of the fact I picked a good value for γ ($\mathbf{1.75}$).

Extra task: *Reinhard's local photographic operator*

Having all the steps for the global operator set, it's rather easy to setup the local operator. The first thing we need to do is define the key parameters for this:

- ϕ (sharpness parameter)
- σ_list, which is defined by
  - σ_step
  - max_σ
- ϵ (threshold for $\boldsymbol{W(x, y)}$)

The following parameters were the ones that gave me the best result:

```
ϕ = 10
```

```
max_σ = 29
```

```
max_σ = 29
```

```
σ_step = 2
```

```
σ_step = 2
```

```
σ_list =  [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28]
```

```
σ_list = collect(0:σ_step:max_σ)
```

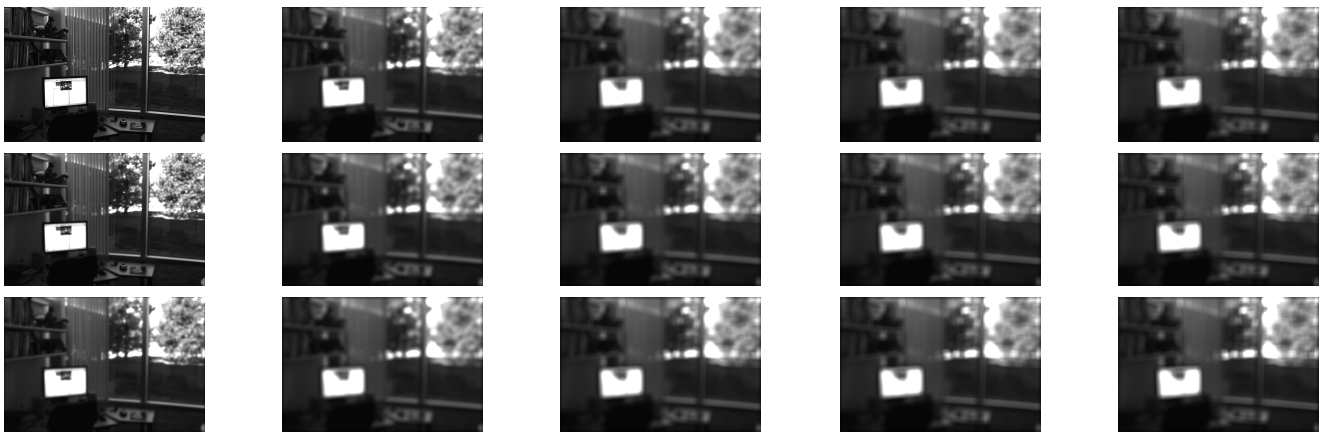
```
ϵ = 0.0001
```

```
ϵ = 0.0001
```

We can apply gaussian matrices and crop them accordingly to get a sequence of blurred **Scaled Luminance** values:

```
md"""
We can apply gaussian matrices and crop them accordingly to get a sequence of blurred
**Scaled Luminance** values:
"""
```

```
σ_stack = vcat([Lₛ], [conv(collect(Kernel.gaussian([s, s], [31, 31])), Lₛ)[16:end-15,
16:end-15] for s ∈ σ_list[2:end]]);
```



We define a function for $W(x, y)$

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

```
W(x, y, σ, σ_idx, σ_stack) =
    (σ_stack[σ_idx][y, x] - σ_stack[σ_idx + 1][y, x])/(((2^ϕ) / (σ^2)) +
    σ_stack[σ_idx][y, x])
```

Now we create a for loop function that goes through each σ value and each pixel, checking whether $|W(x, y)| < \epsilon$. If it is, it defines a maximum σ for it and doesn't visit that pixel again. Once we have every maximum value, we replace $L_s(x, y)$ by the value of the maximum given σ stack at the (x, y) pixel.

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

```
function local_tonemapping(Lₛ)
    (rows, cols) = size(Lₛ)
    σ_max = zeros(Int64, size(Lₛ))
    W_filled = zeros(Bool, size(Lₛ))
    for σ_idx ∈ (2:length(σ_list)-1)
        for row ∈ (1:rows)
            for col ∈ (1:cols)
                if (W_filled[row, col])
                    continue
                end
                W_xy = W(col, row, σ_list[σ_idx], σ_idx, σ_stack)
                if (abs(W_xy) < ϵ)
                    W_filled[row, col] = true
                end
                σ_max[row, col] = σ_idx
            end
        end
    end
    new_Lₛ = zeros(Float64, size(Lₛ))
    for row ∈ (1:rows)
            for col ∈ (1:cols)
                idx = σ_max[row, col]
                new_Lₛ[row, col] = σ_stack[idx][row, col]
            end
    end
    return new_Lₛ
end
```

We then do the same thing that we had done for the global operator:

```
Ll = Lₛ ./ (local_tonemapping(Lₛ) .+ 1);
```

```
local_tonemapped = Ll .* hdr ./ L * saturation;
```

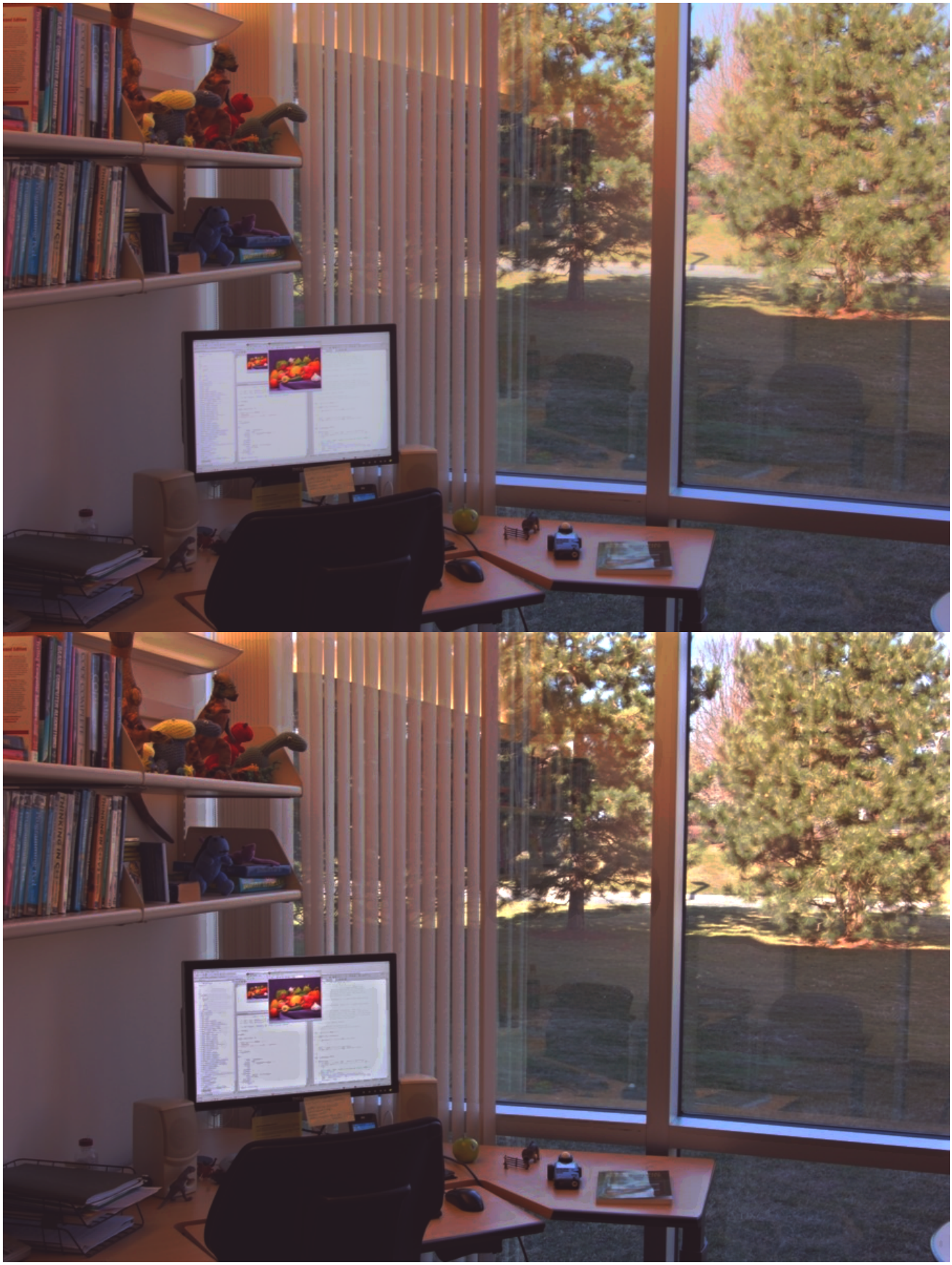
`local_gamma_tonemapped =`

- `local_gamma_tonemapped = gamma_encode(local_tonemapped)`

Result comparison

We can see the results below (top: global, bottom: local). The differences are easily perceivable: we have sharper details on the monitor and on the background trees.

There are some artifacts in the monitor, but mostly because I exagerated on some parameters in order for the differences to be **very perceivable**.

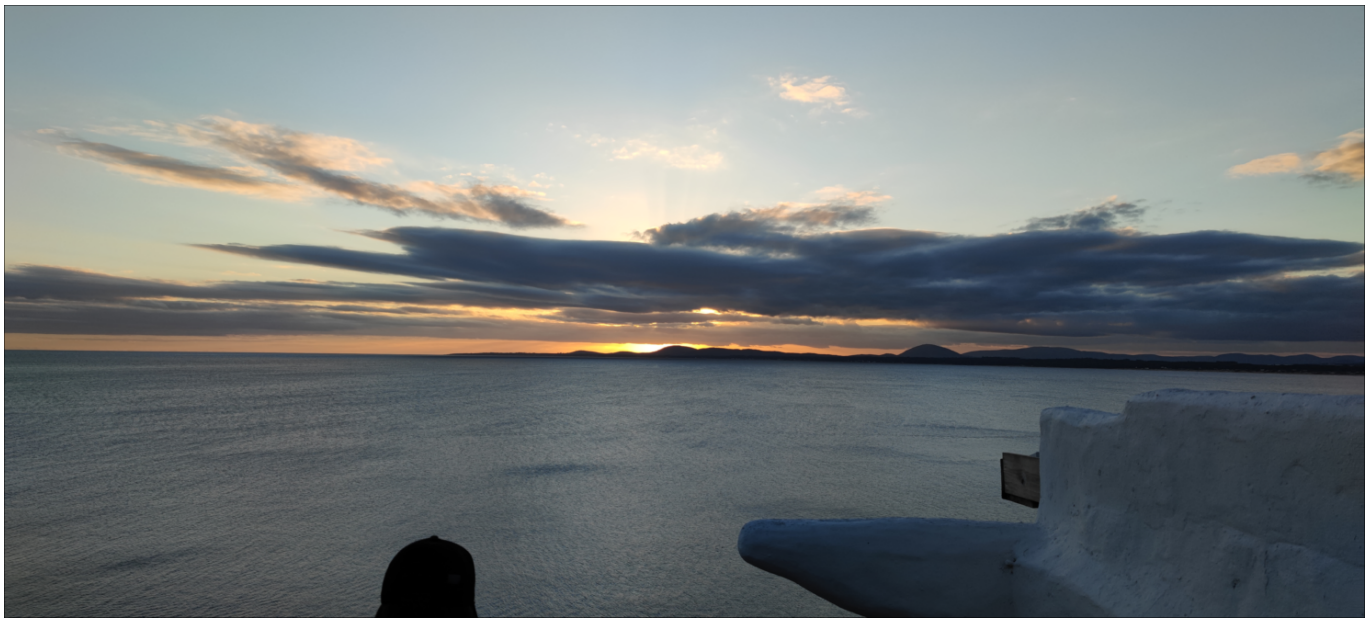


Here we can see a bit more how much sharper the local operator makes the image.

Conclusion

It's truly amazing to see how this algorithm pans out in the end. HDR is a very simple idea that, when you dive deep enough, becomes a bit complicated but, nonetheless, elegant. Implementing this was a real treat and I think my results were great, specially the **Local operator** one. I wish I could've understood a bit more about how the local operator parameters affect the final result (ϕ didn't seem to have much effect).

I love taking landscape pictures with HDR on! Implementing it gave me a better insight to how it works and when to use it. Here's a picture I took with HDR on my trip to Punta del Este.



Setup

```
using DSP, Images, ImageFiltering, HypertextLiteral, MAT, PlutoUI, Statistics, Latexify
```