A1: HDR Tone Mapping

CS4365 Applied Image Processing

Assignment 1

2023/2024

Petr Kellnhofer

Agenda Lab Sep 13

- Instructions for the Assignment 1
- Working on the assignments and Q&A (TAs: Lukas and Peter)

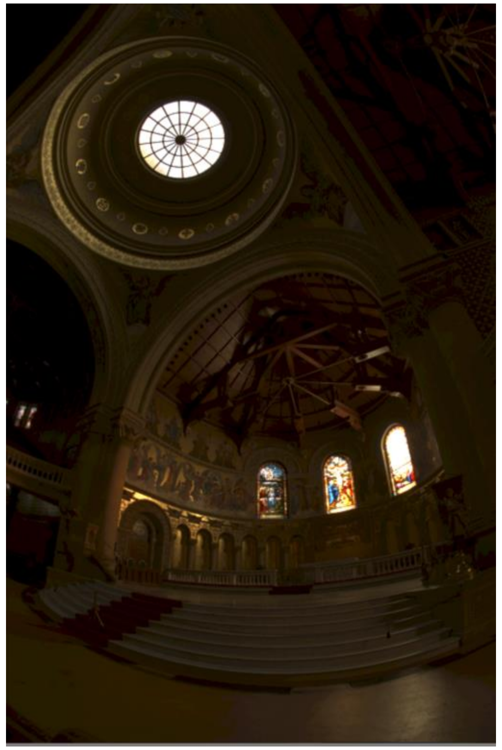
Notes for Assignment 0

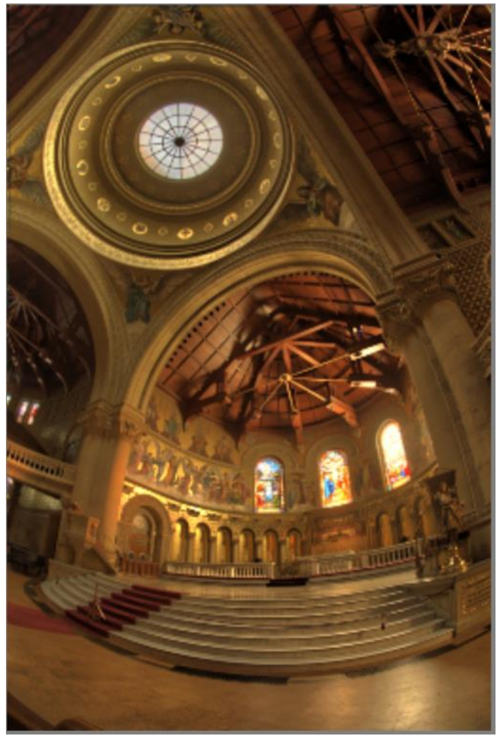
- Confusion about the correct **download link**
  - Will be clear for Assignment 1 (only in Content->Assignments)
- Upload **only your_code_here.h**
  - Brightspace should now check
- Careful about **test code** in your_code_here.h,
  - Put it to main.cpp instead
- Next time submission **deadline enforced**
  - A special new category for late submissions explained later

Goal

- Implement a simplified **gradient-based** tone mapper.
  - Converts high dynamic range (HDR) image to standard range (SDR).
  - See **Lecture 2** for more info.

Input

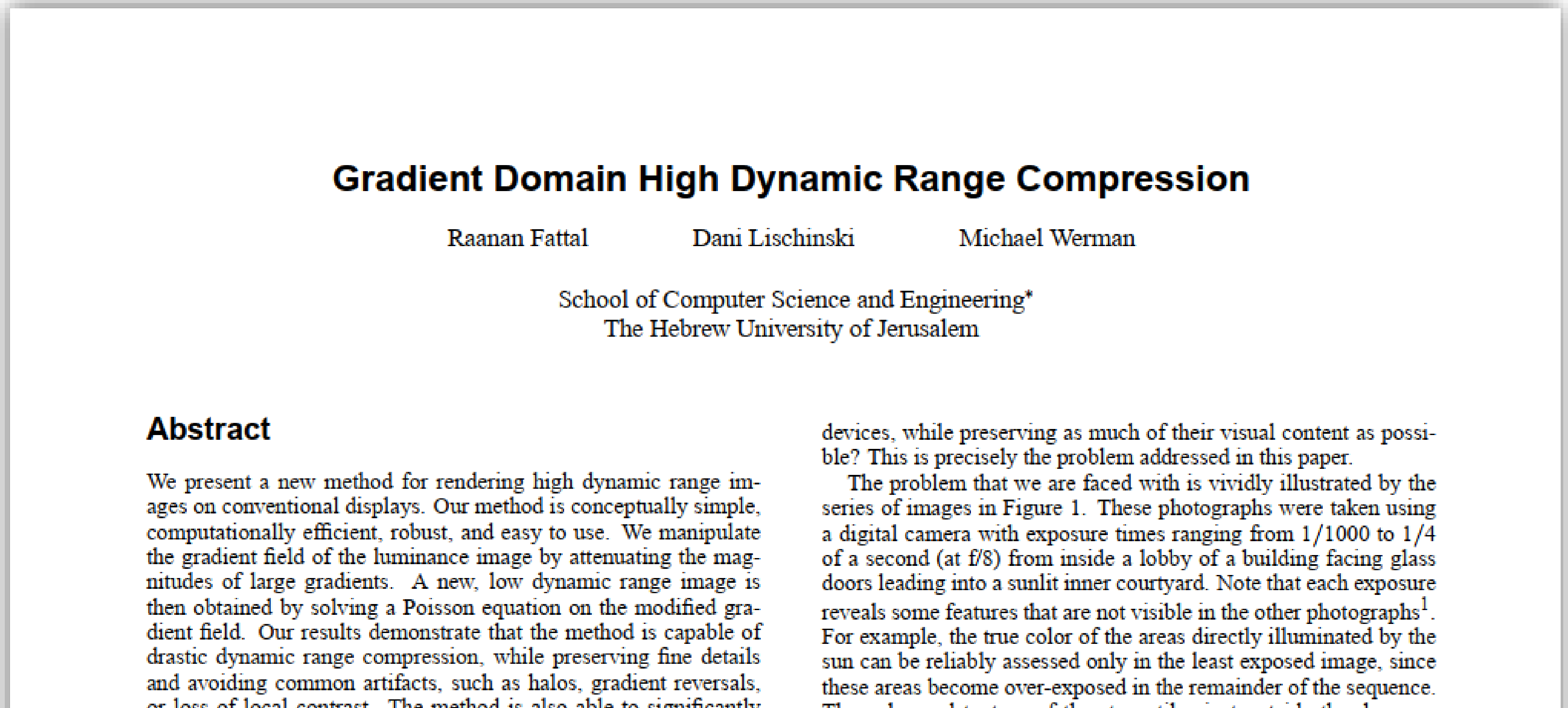




Output

Inspiration

- Raanan Fattal, Dani Lischinski, and Michael Werman: "**Gradient domain high dynamic range compression.**" CGIT 2002.
  - Link: https://courses.cs.washington.edu/courses/cse590b/02au/hdrc.pdf
  - Often referred to as **[Fattal 02]** in HDR processing software



Gradient Domain High Dynamic Range Compression

Raanan Fattal Dani Lischinski Michael Werman

School of Computer Science and Engineering*
The Hebrew University of Jerusalem

Abstract

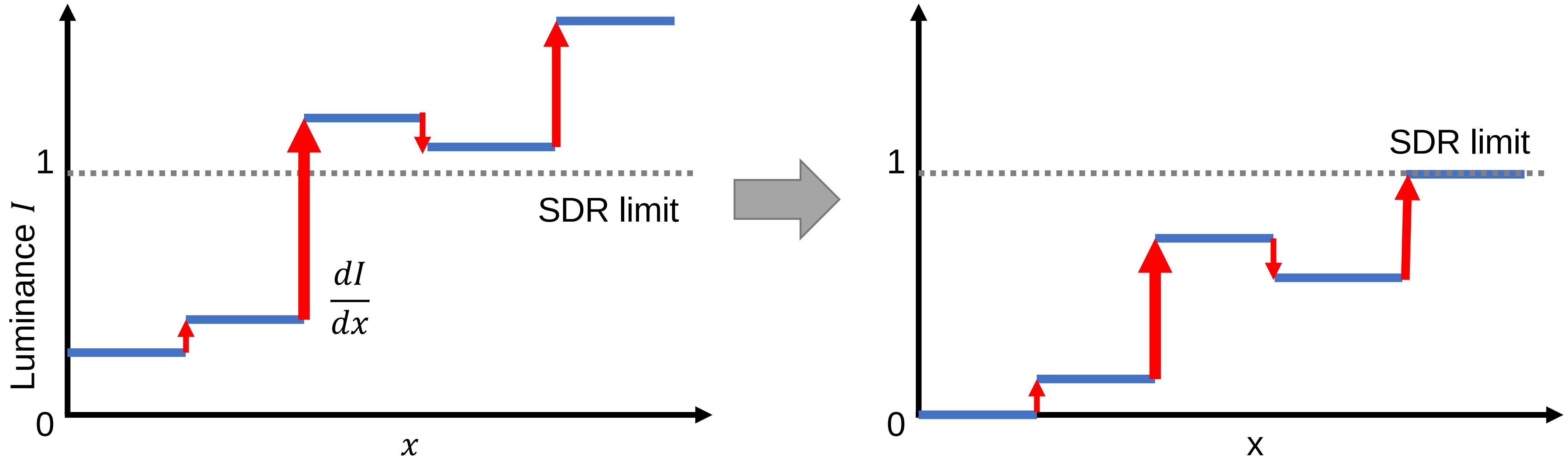
We present a new method for rendering high dynamic range images on conventional displays. Our method is conceptually simple, computationally efficient, robust, and easy to use. We manipulate the gradient field of the luminance image by attenuating the magnitudes of large gradients. A new, low dynamic range image is then obtained by solving a Poisson equation on the modified gradient field. Our results demonstrate that the method is capable of drastic dynamic range compression, while preserving fine details and avoiding common artifacts, such as halos, gradient reversals,

devices, while preserving as much of their visual content as possible? This is precisely the problem addressed in this paper.

The problem that we are faced with is vividly illustrated by the series of images in Figure 1. These photographs were taken using a digital camera with exposure times ranging from 1/1000 to 1/4 of a second (at f/8) from inside a lobby of a building facing glass doors leading into a sunlit inner courtyard. Note that each exposure reveals some features that are not visible in the other photographs[1]. For example, the true color of the areas directly illuminated by the sun can be reliably assessed only in the least exposed image, since these areas become over-exposed in the remainder of the sequence.

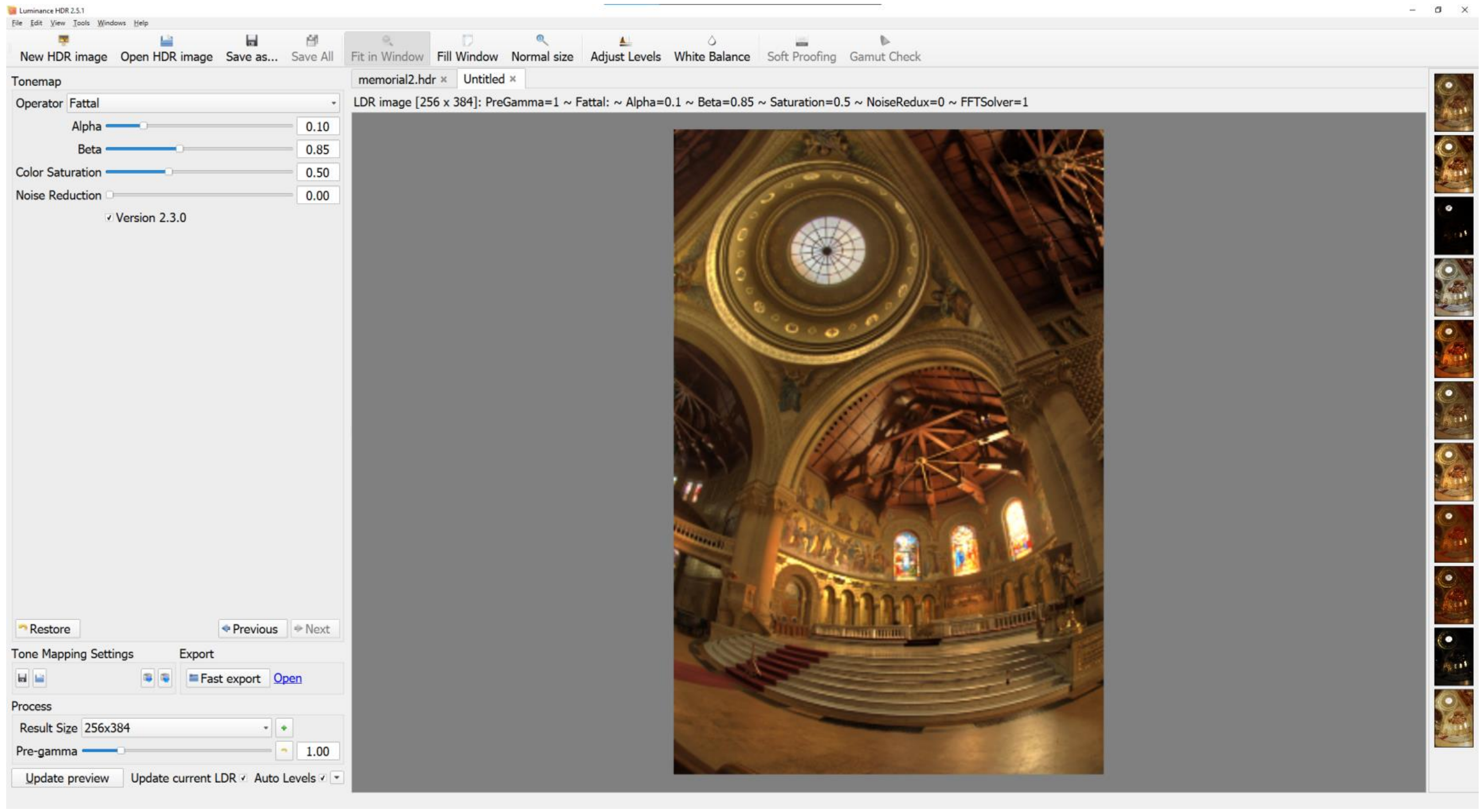
Gradient domain high dynamic range compression

- HDR **gradients** are too large to fit into SDR image.
- We can **compress** large gradients more than small gradients.
  - They will still be visible enough but will take less space.



Demo

- LuminanceHDR software has many HDR mappers implemented.
- https://github.com/LuminanceHDR/



Tonemap

Operator Fattal

Alpha	0.10
Beta	0.85
Color Saturation	0.50
Noise Reduction	0.00

✓ Version 2.3.0

Tone Mapping Settings

Export

Process

Result Size 256x384

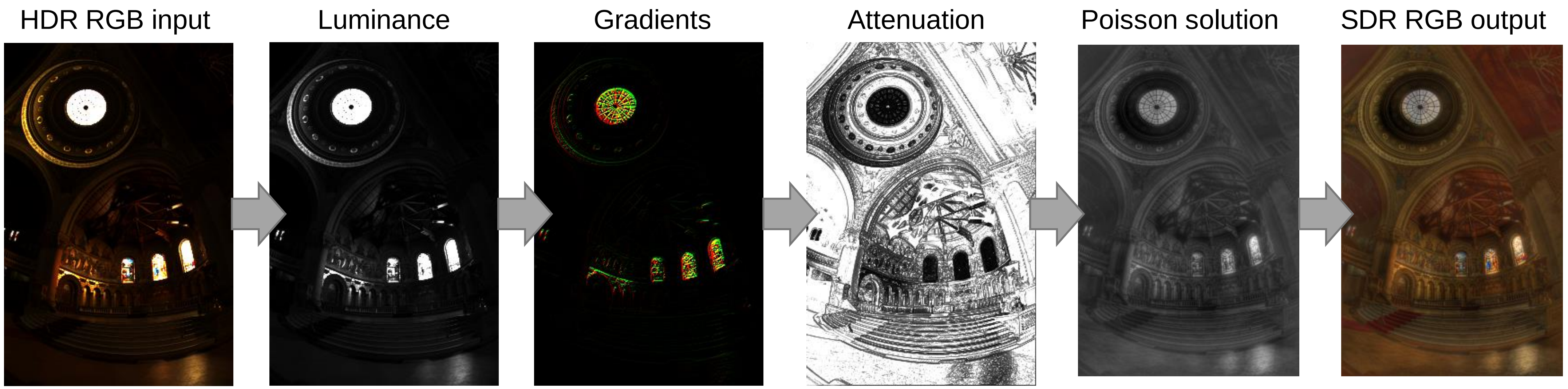
Pre-gamma 1.00

Update current LDR ✓ Auto Levels ✓

LDR image [256 x 384]: PreGamma=1 ~ Fattal: ~ Alpha=0.1 ~ Beta=0.85 ~ Saturation=0.5 ~ NoiseRedux=0 ~ FFTSolver=1

What we DO implement

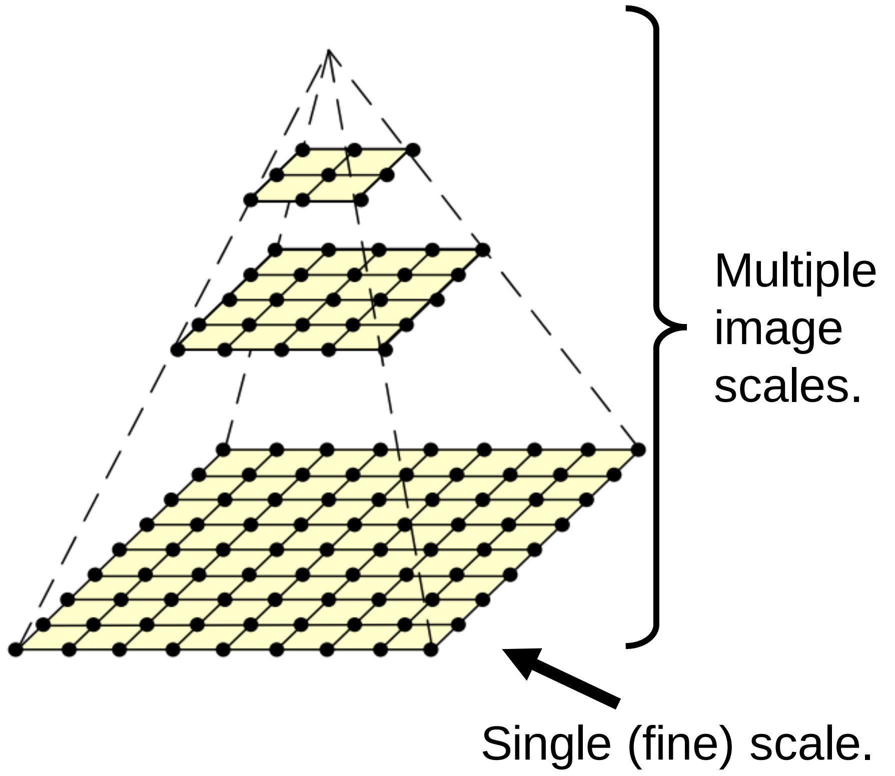
- Extract luminance from RGB image
- Compute image features – gradients, divergence
- Non-linearly rescale image contrast in the gradient space
- Solve Poisson equation
- Reconstruct SDR RGB image

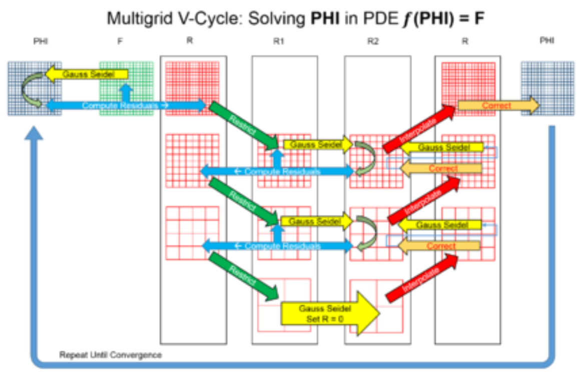


What we DO NOT implem

- We replace **multi-scale** steps with **single-scale** alternatives.
  - We only consider contrast on the finest level.
    - This means we **do not account** for smoother image gradients.
  - We only solve the Poisson equation on the finest grid.
    - This is lot **slower** than multi-grid solvers.

- We get **similar effect** but not as good quality and slow compute.

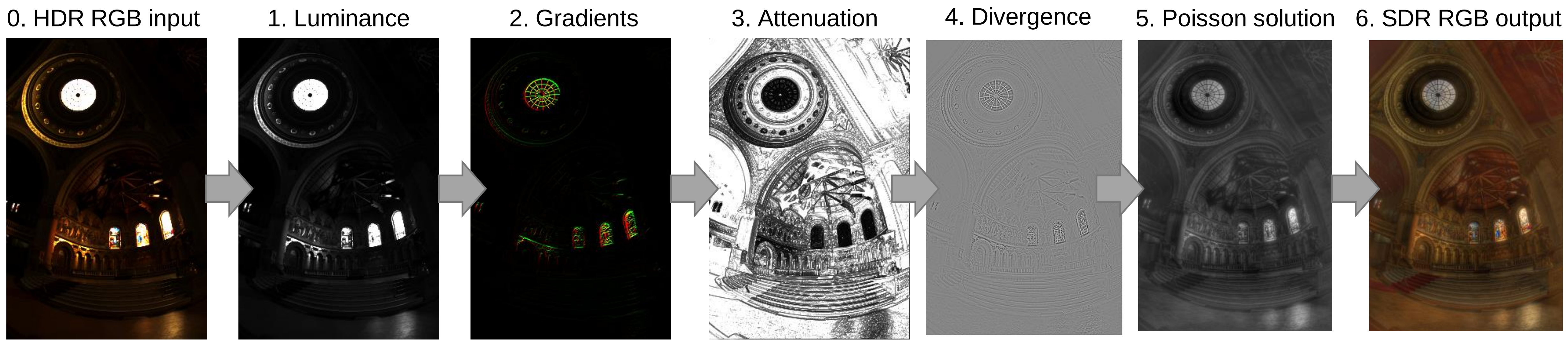




Multi-grid method.

TU Delft

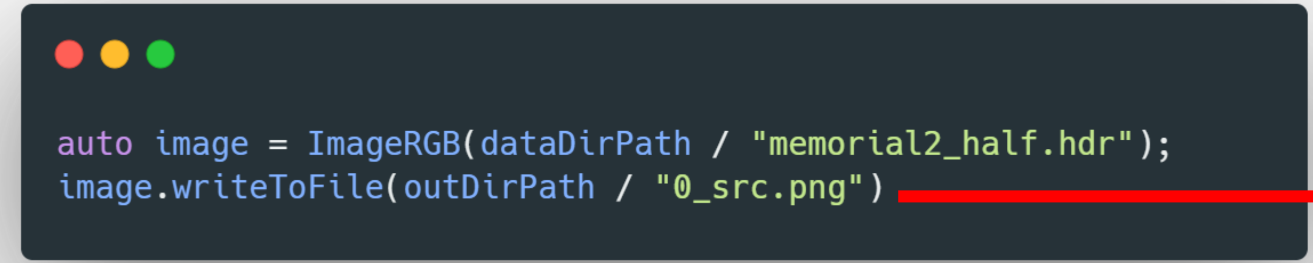
Algorithm


0. HDR RGB input
1. Luminance
2. Gradients
3. Attenuation
4. Divergence
5. Poisson solution
6. SDR RGB output

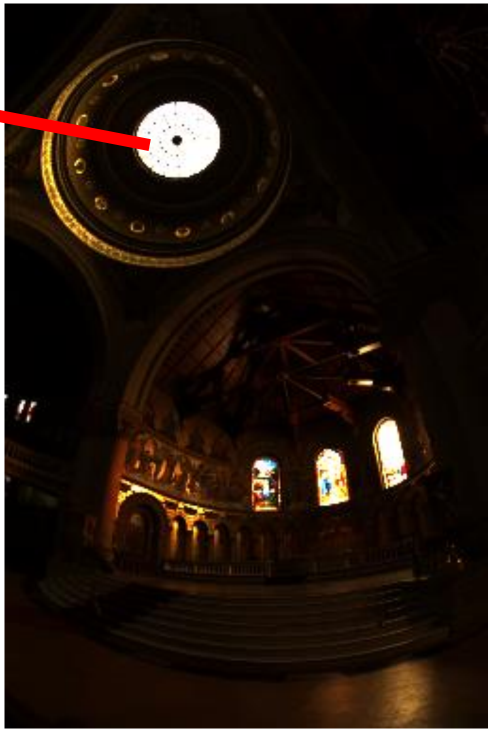
0. The HDR input image.

- Already provided. Loads the image into a 3 channel RGB image.
- Unlike SDR image, HDR values may be >1. Often in physical units.
- Saving it to an SDR format (e.g., PNG below), it truncates the values.

Truncated/saturated values



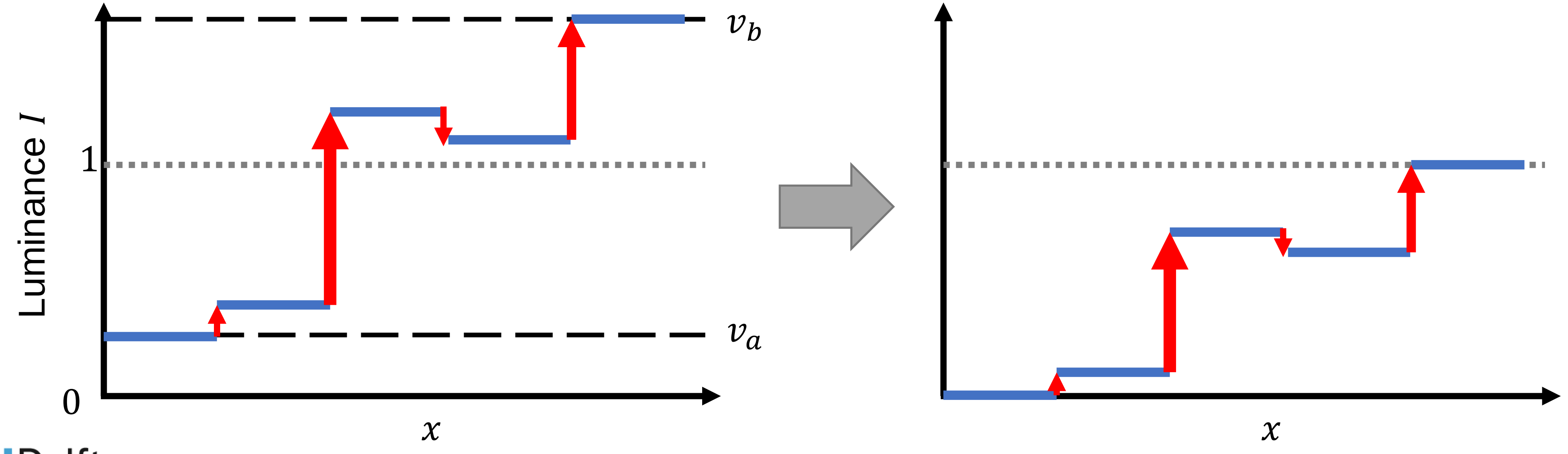
```
auto image = ImageRGB(dataDirPath / "memorial2_half.hdr");
image.writeToFile(outDirPath / "0_src.png")
```



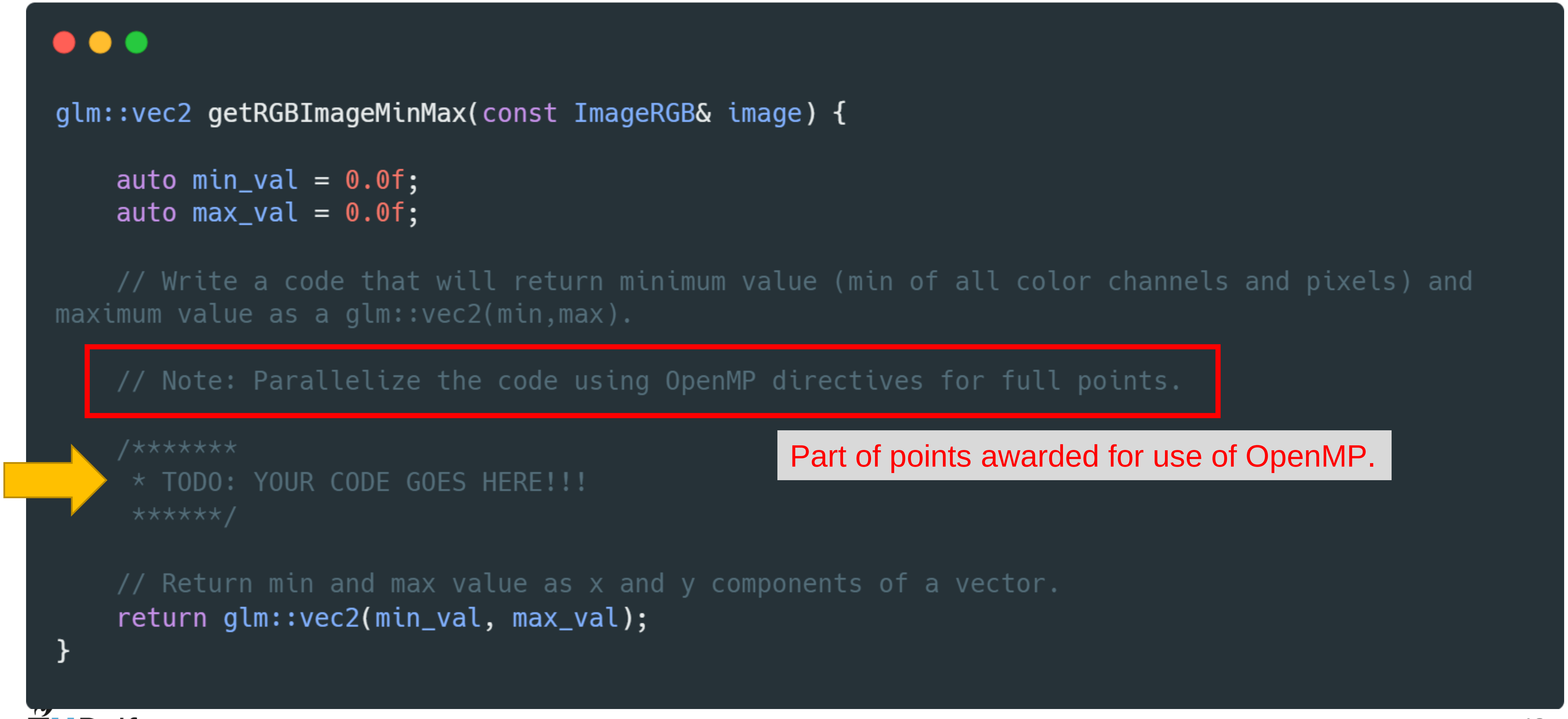
0.a Experiment – Linear normalization

› Find minimum and maximum value of any channel (red, green or blue) and **linearly rescale** the image to the $[0,1]$ range.

$$I_{norm} = \frac{I_{HDR} - v_a}{v_b - v_a} \qquad v_a = \min_{i,j,c}\{I_{HDR}[i,j,c]\} \qquad v_b = \max_{i,j,c}\{I_{HDR}[i,j,c]\}$$



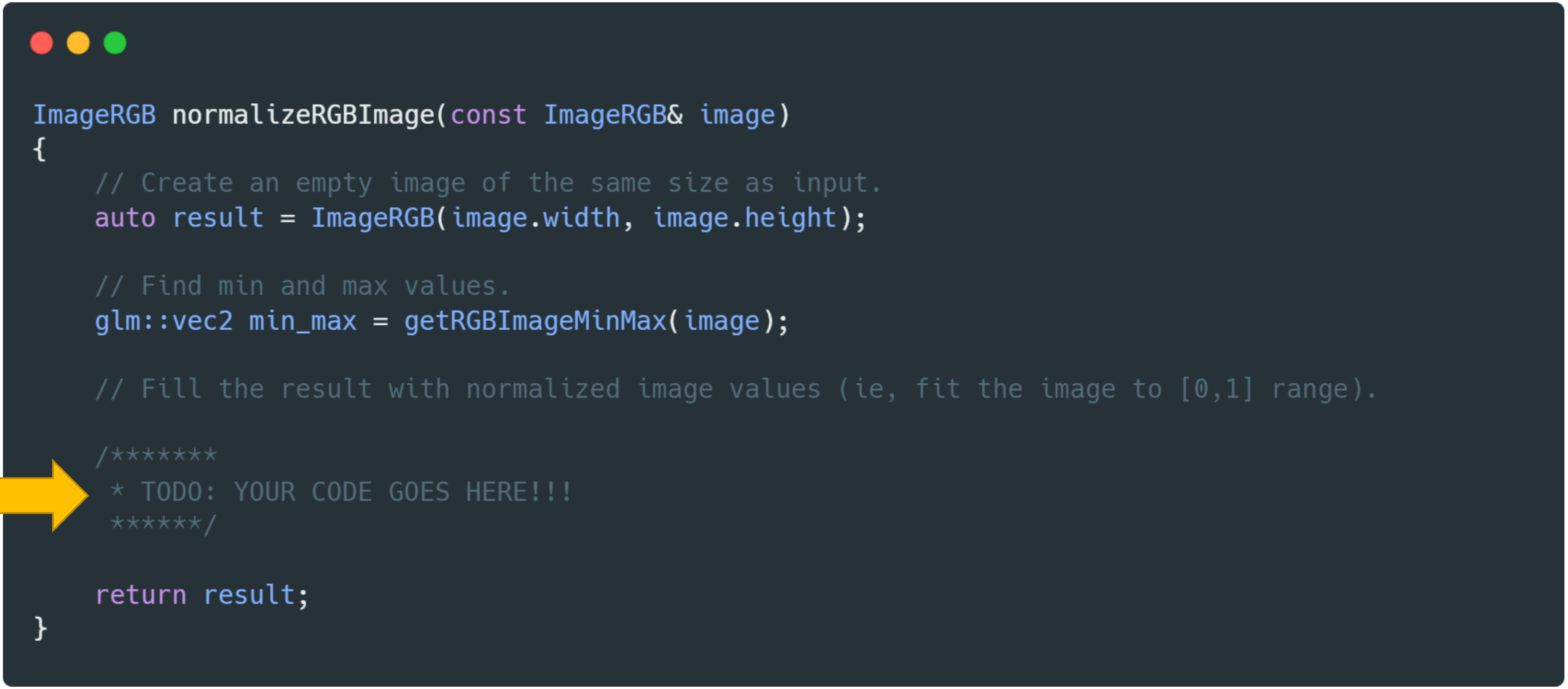
getRGBImageMinMax(...)



```
glm::vec2 getRGBImageMinMax(const ImageRGB& image) {

    auto min_val = 0.0f;
    auto max_val = 0.0f;

    // Write a code that will return minimum value (min of all color channels and pixels) and
maximum value as a glm::vec2(min,max).

    // Note: Parallelize the code using OpenMP directives for full points.

    /*******
     * TODO: YOUR CODE GOES HERE!!!
     ******/

    // Return min and max value as x and y components of a vector.
    return glm::vec2(min_val, max_val);
}
```

Part of points awarded for use of OpenMP.

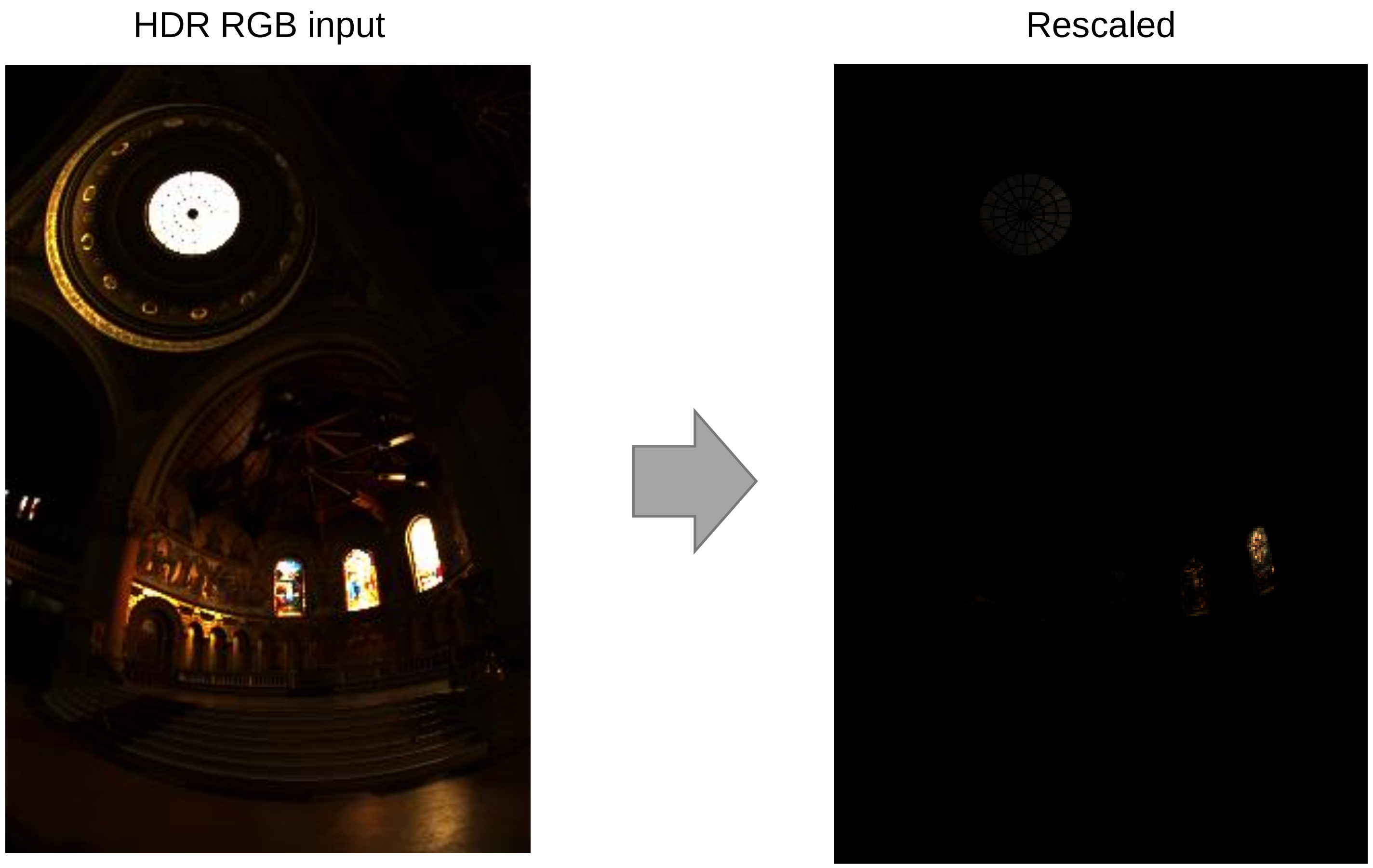
normalizeRGBImage(...)



```
ImageRGB normalizeRGBImage(const ImageRGB& image)
{
    // Create an empty image of the same size as input.
    auto result = ImageRGB(image.width, image.height);

    // Find min and max values.
    glm::vec2 min_max = getRGBImageMinMax(image);

    // Fill the result with normalized image values (ie, fit the image to [0,1] range).

    /*******
     * TODO: YOUR CODE GOES HERE!!!
     ******/

    return result;
}
```

0.a Experiment – Linear normalization

› Results in a very dark image due to **heavy compression** of the value range.

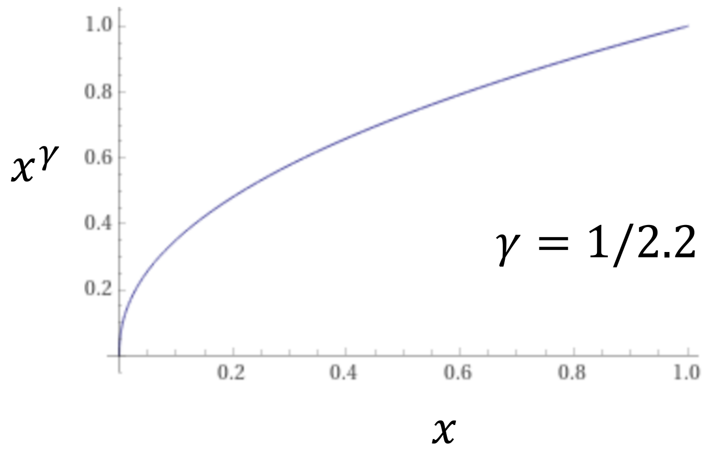


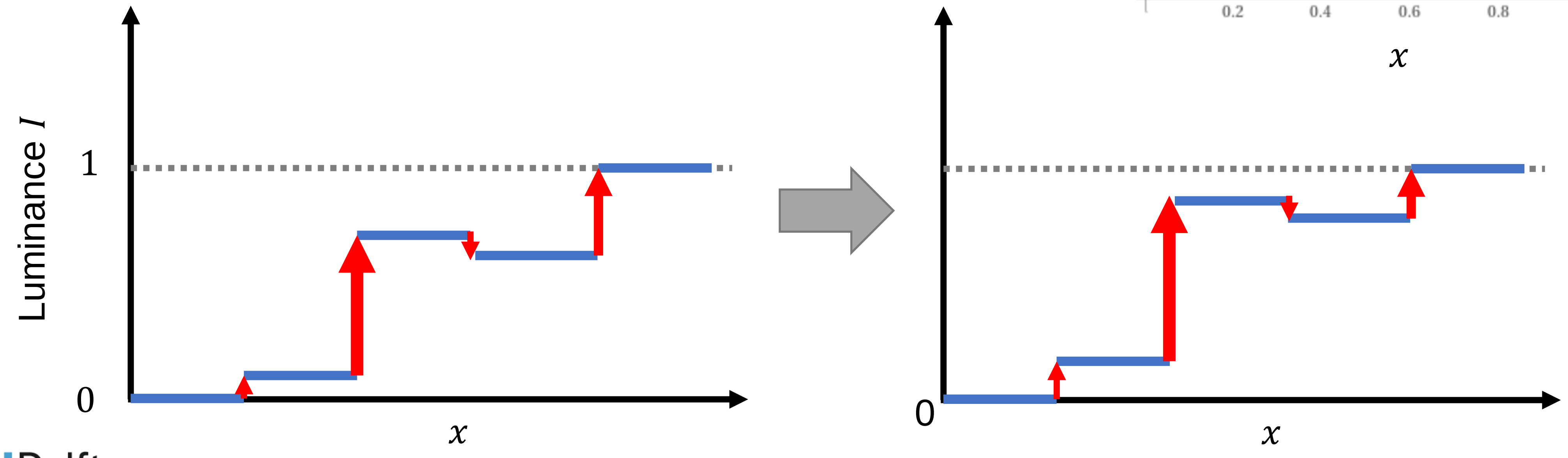
TU Delft

0.b Experiment – Gamma mapping

- Compress the entire image using a non-linear gamma curve.

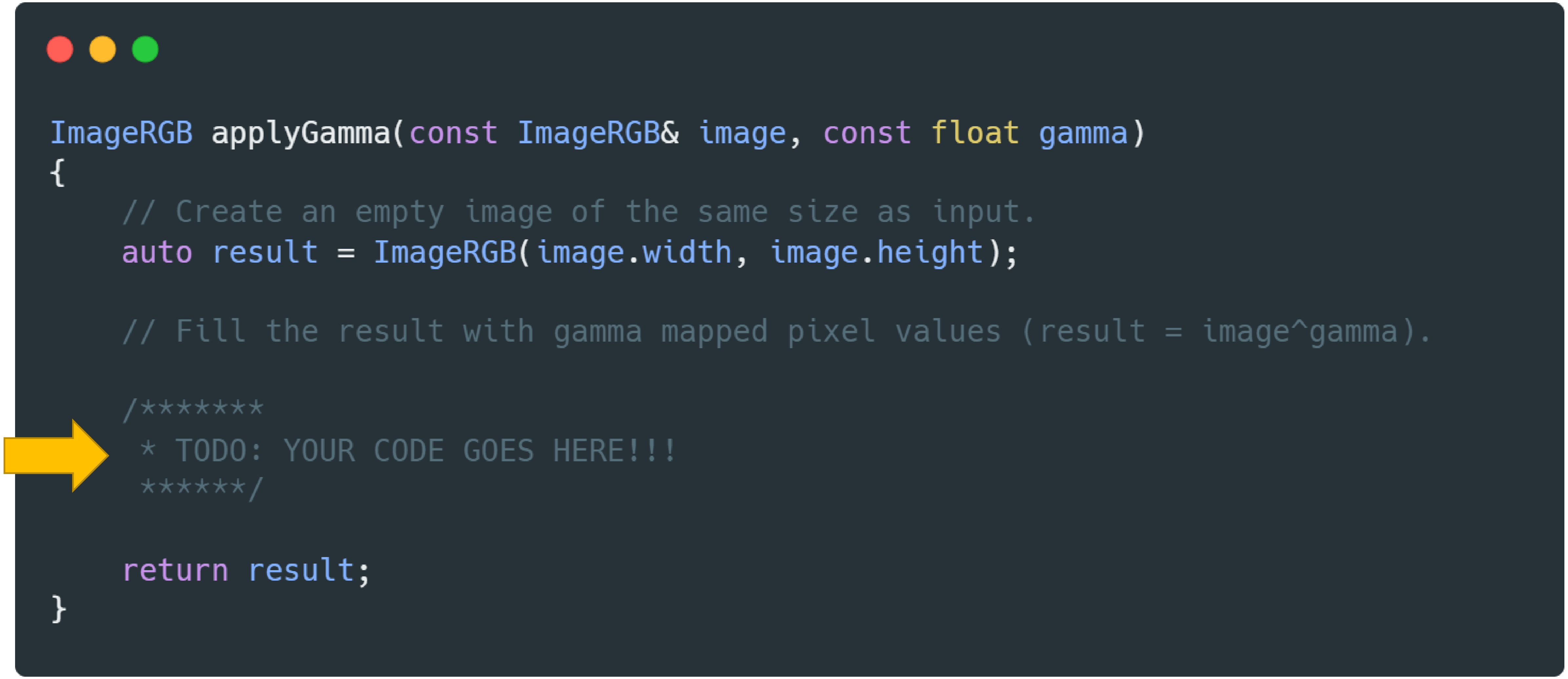
$$I_{gamma} = I_{norm}^{\gamma}$$





TU Delft

applyGamma(...)



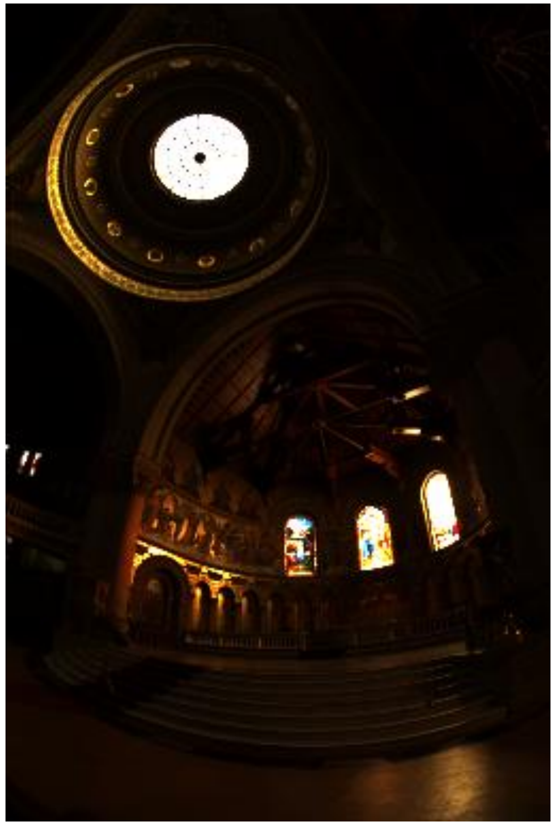
```cpp
ImageRGB applyGamma(const ImageRGB& image, const float gamma)
{
    // Create an empty image of the same size as input.
    auto result = ImageRGB(image.width, image.height);

    // Fill the result with gamma mapped pixel values (result = image^gamma).

    /*******
     * TODO: YOUR CODE GOES HERE!!!
     ******/

    return result;
}
```

TUDelft

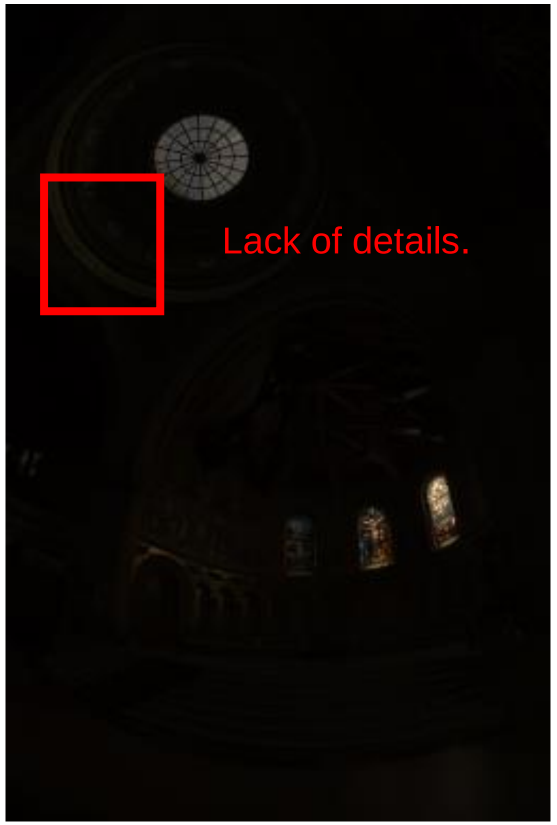
0.b Experiment – Gamma mapping

› Better but still loosing contrast in brighter regions.

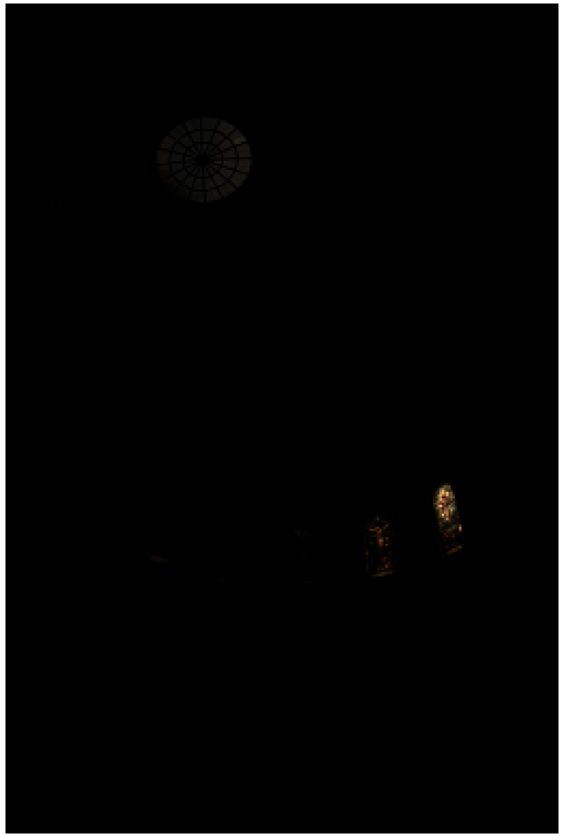
HDR RGB input



Gamma



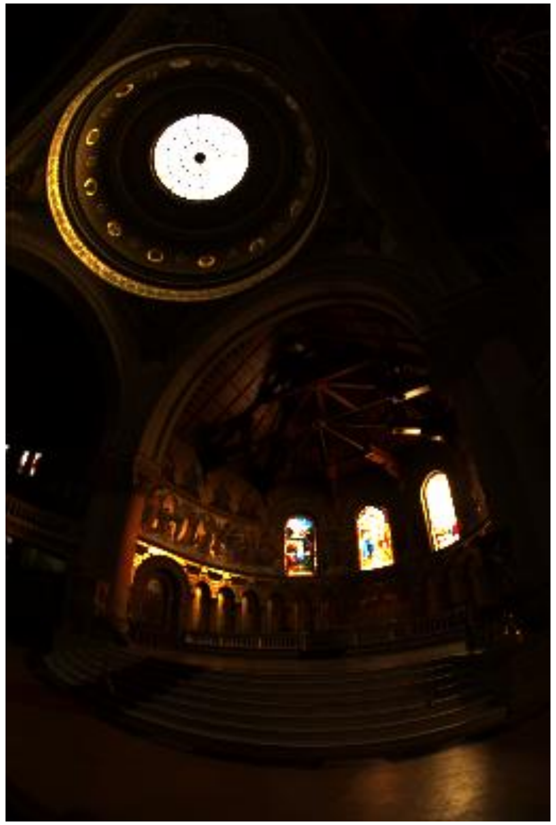
Linear



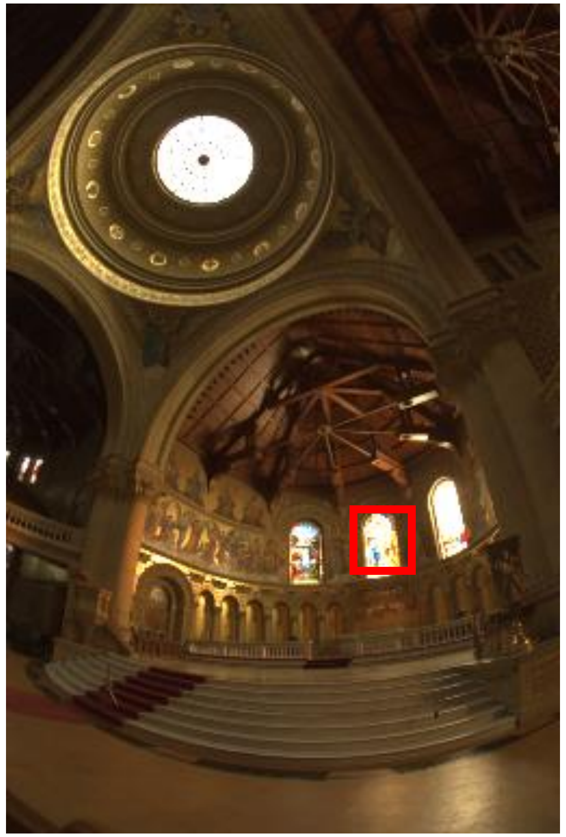
0.b Experiment – Gamma mapping

- We can get better results by scaling the input to gamma.

HDR RGB input



Gamma for non-normalized input



1. Extract luminance from RGB image.

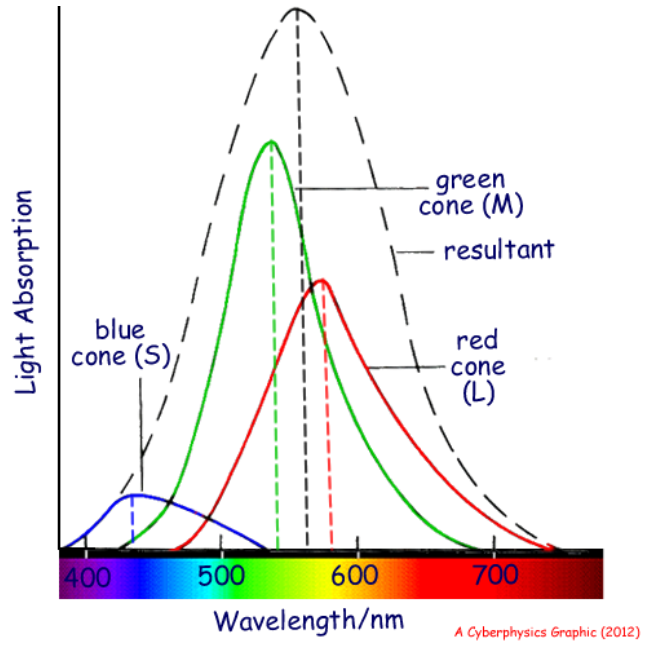
- We aim to modify luminance and not colors. Therefore, we need to isolate **luminance**.
- We will use ITU R-REC-BT.601 standard to extract luminance from the RGB as:

$$luminance = \ 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

High sensitivity to green

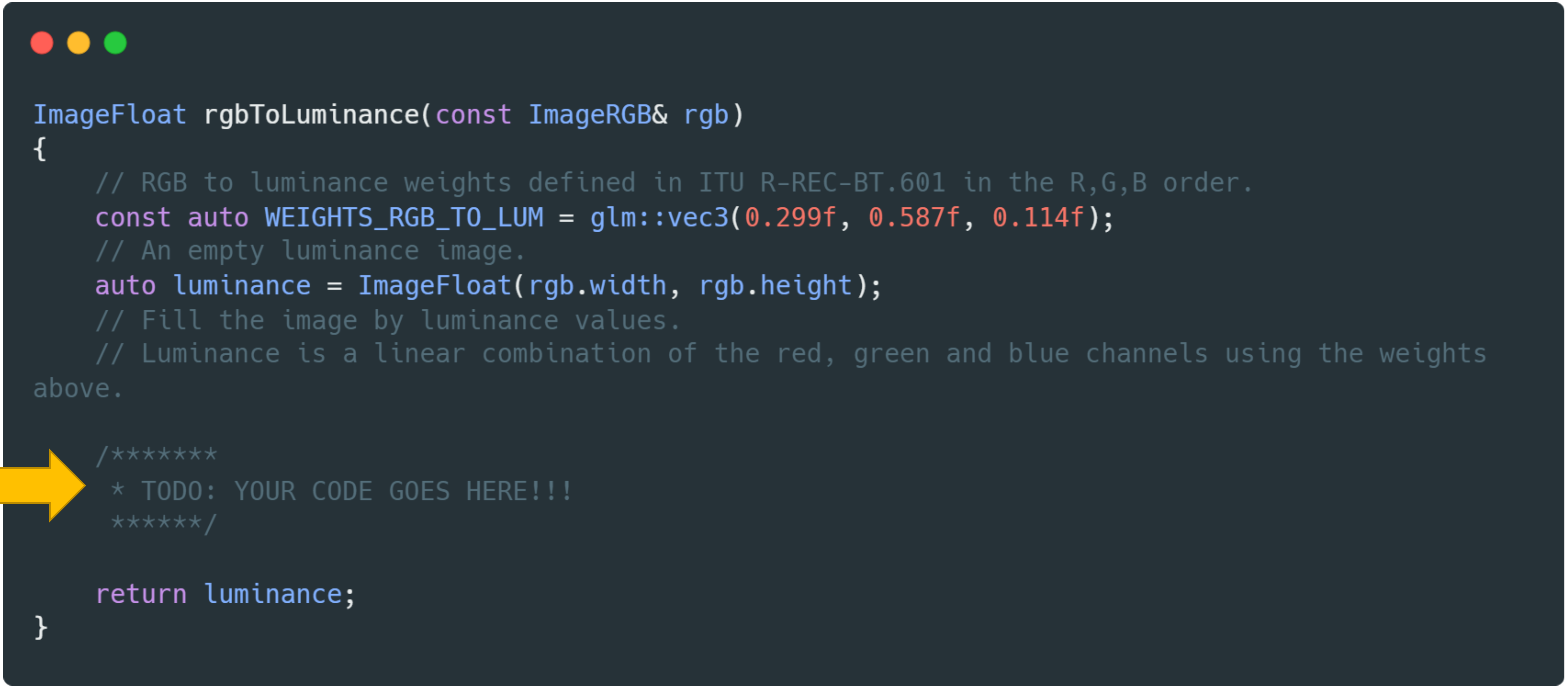
Low sensitivity to blue

The algorithm the proceeds with log-luminance H. This is already implemented in the main.cpp and you do not need to implement it yourself. **Again: Compute linear luminance!**



https://www.cyberphysics.co.uk/topics/medical/Eye/photosensitiveCells.html

rgbToLogLuminance(...)

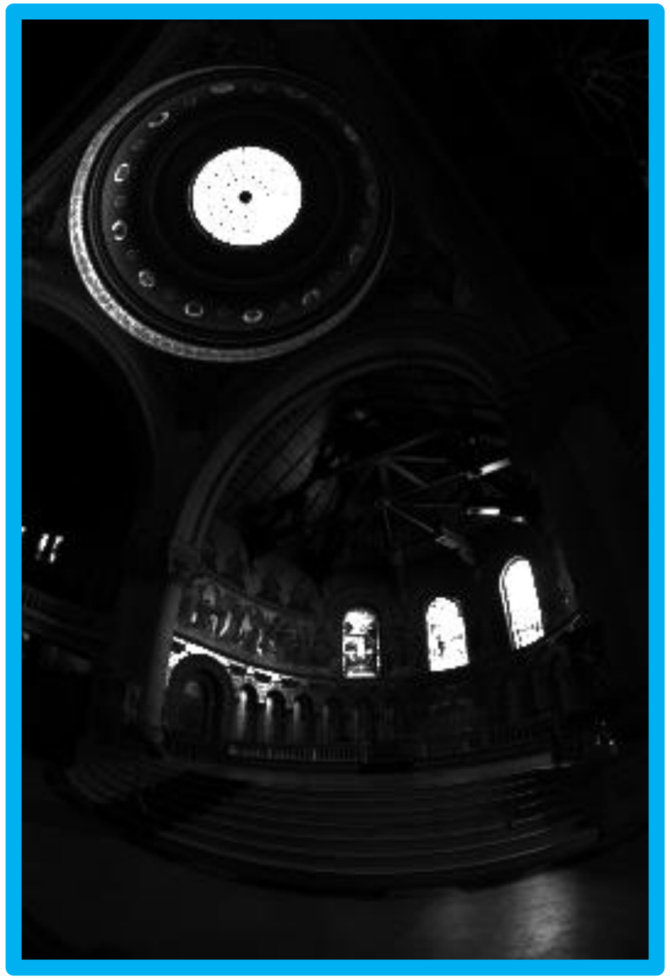


```
ImageFloat rgbToLuminance(const ImageRGB& rgb)
{
    // RGB to luminance weights defined in ITU R-REC-BT.601 in the R,G,B order.
    const auto WEIGHTS_RGB_TO_LUM = glm::vec3(0.299f, 0.587f, 0.114f);
    // An empty luminance image.
    auto luminance = ImageFloat(rgb.width, rgb.height);
    // Fill the image by luminance values.
    // Luminance is a linear combination of the red, green and blue channels using the weights
above.

    /*******
     * TODO: YOUR CODE GOES HERE!!!
     ******/

    return luminance;
}
```

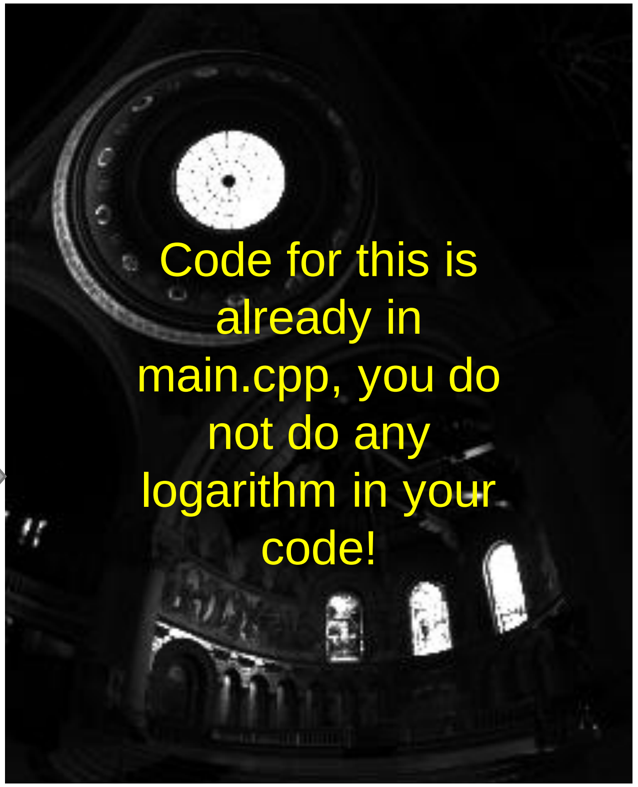
1. Extract luminance from RGB image.

HDR RGB input

(Linear) luminance



Logarithmic luminance H



Code for this is already in main.cpp, you do not do any logarithm in your code!

```
// 3. Get luminance.
auto luminance = rgbToLuminance(image);
luminance.writeToFile(outDirPath / "3a_luminance.png");
auto H = lnImage(luminance);
H.writeToFile(outDirPath / "3b_log_luminance_H.png");
```

2. Compute gradients of H

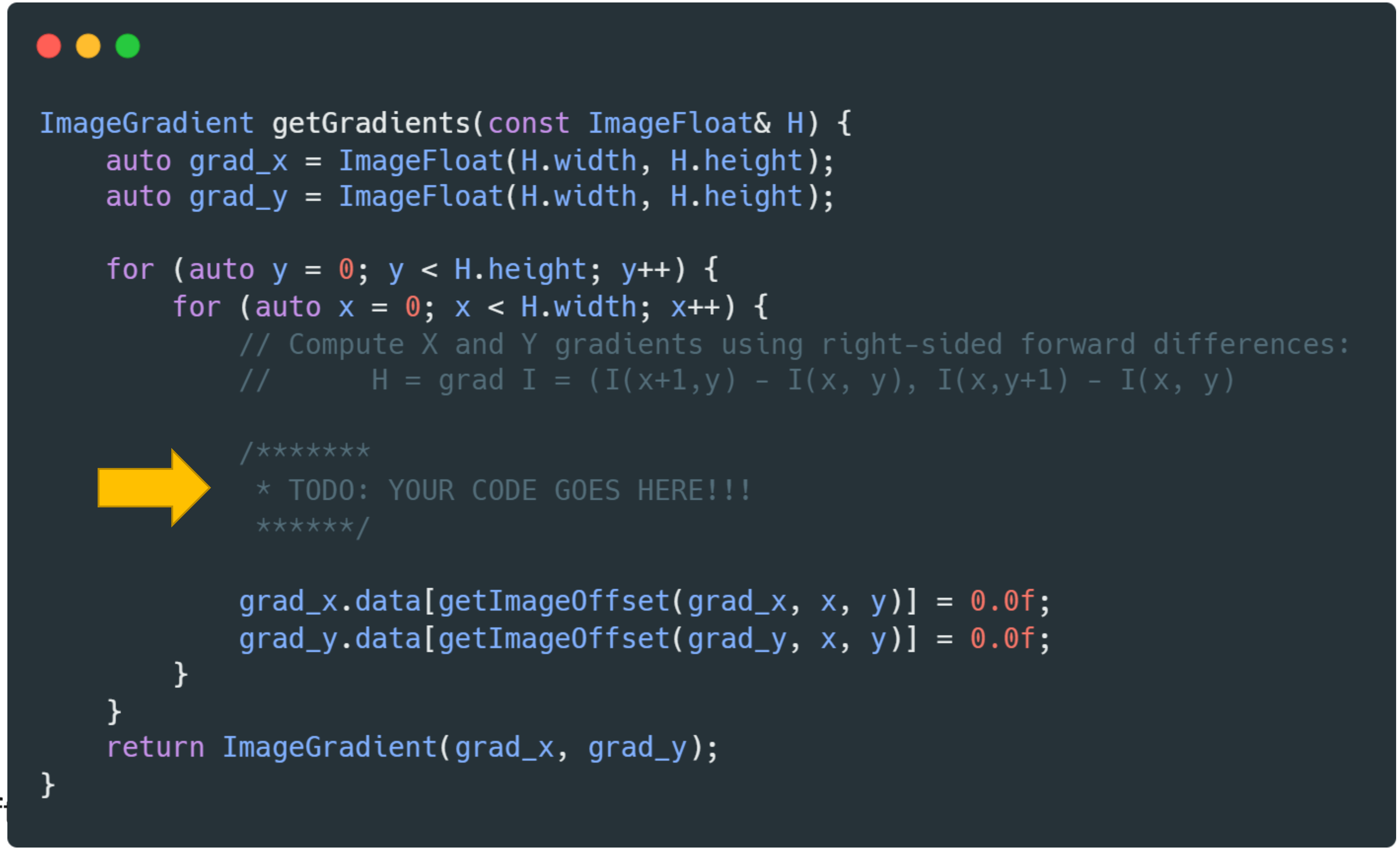
- Compute horizontal (dx) and vertical (dy) **gradients** of the log-luminance H.
- **Section 5** of the paper:

$$\nabla H(x,y) \approx (H(x+1,y) - H(x,y), H(x,y+1) - H(x,y))$$

- Assume zero padding => Pixels outside the image are 0 (black).
  - **Example**: Assuming a 5x5 px image:
    - $\nabla H(4,2) = [H(\mathbf{5},2) - H(4,2), H(4,3) - H(4,2)] \Rightarrow [\mathbf{0} - H(4,2),\ H(4,3) - H(4,2)]$
    - $\nabla H(4,4) = [H(\mathbf{5},4) - H(4,4), H(4,\mathbf{5}) - H(4,4)] \Rightarrow [\mathbf{0} - H(4,4), \mathbf{0} - H(4,4)]$

TU Delft

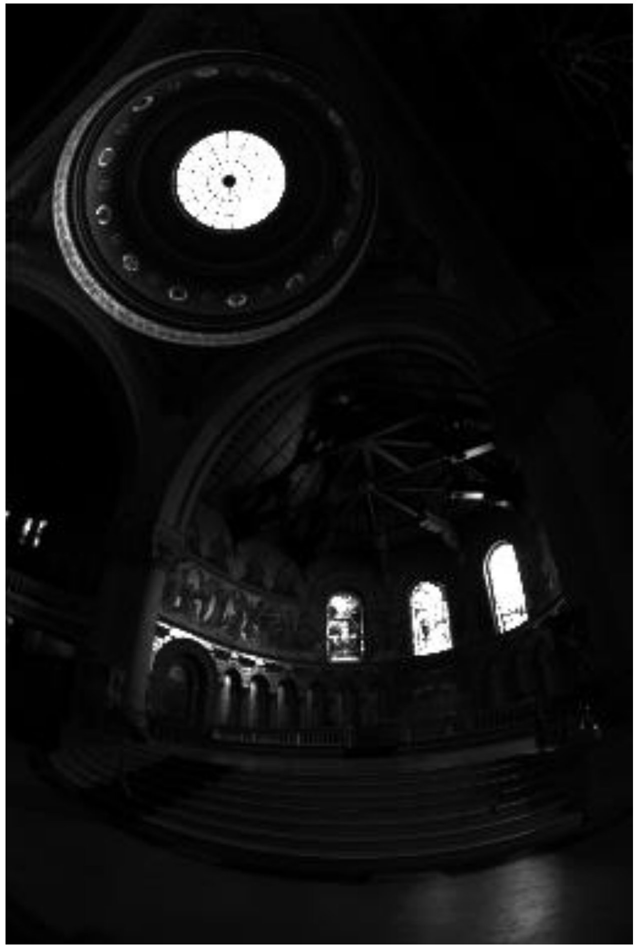
getGradients(...)



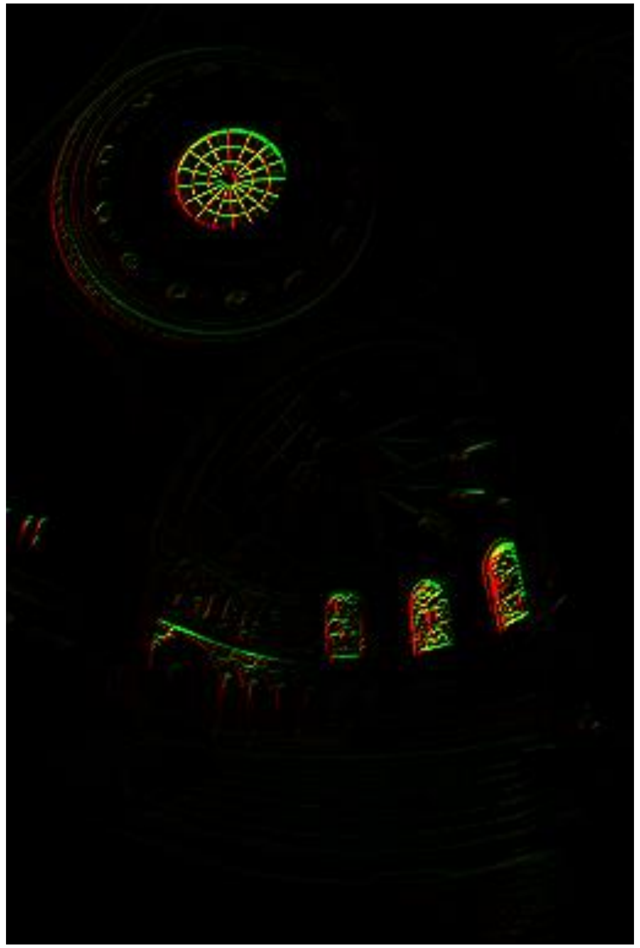
```
ImageGradient getGradients(const ImageFloat& H) {
    auto grad_x = ImageFloat(H.width, H.height);
    auto grad_y = ImageFloat(H.width, H.height);

    for (auto y = 0; y < H.height; y++) {
        for (auto x = 0; x < H.width; x++) {
            // Compute X and Y gradients using right-sided forward differences:
            //      H = grad I = (I(x+1,y) - I(x, y), I(x,y+1) - I(x, y)

            /*******
             * TODO: YOUR CODE GOES HERE!!!
             ******/

            grad_x.data[getImageOffset(grad_x, x, y)] = 0.0f;
            grad_y.data[getImageOffset(grad_y, x, y)] = 0.0f;
        }
    }
    return ImageGradient(grad_x, grad_y);
}
```

2. Compute gradients of H

Logarithmic luminance H

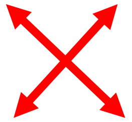


∇H

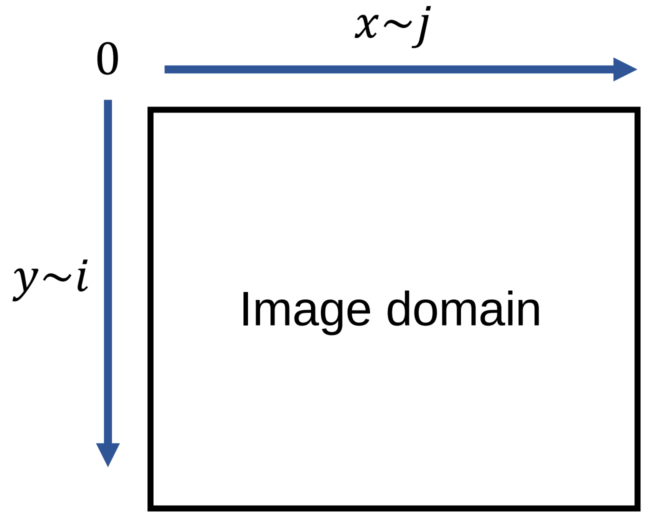


2. Note on 2D (x,y) indexing

- The notation of a 2D **function** $H(x, y)$ vs. index in a **2D array** $H(i, j)$:
  - x = horizontal coordinate = column index = j
  - y = vertical coordinate = row index = i



- **We use $H(x, y)$ notation consistently.**

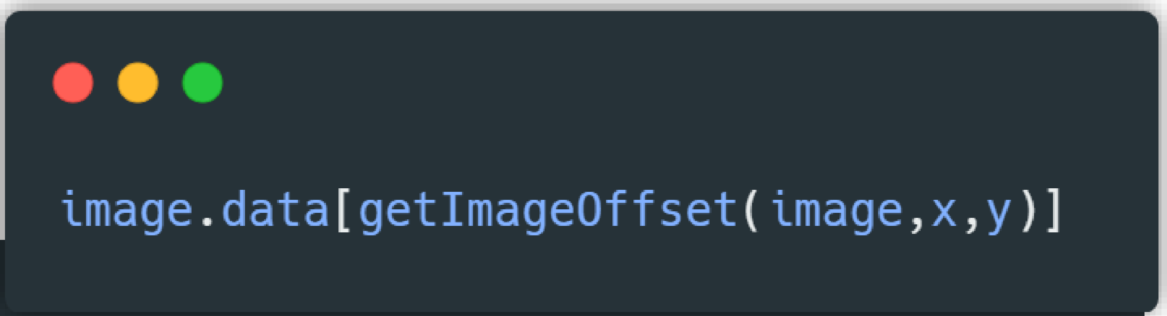


Reminder from Lecture 1

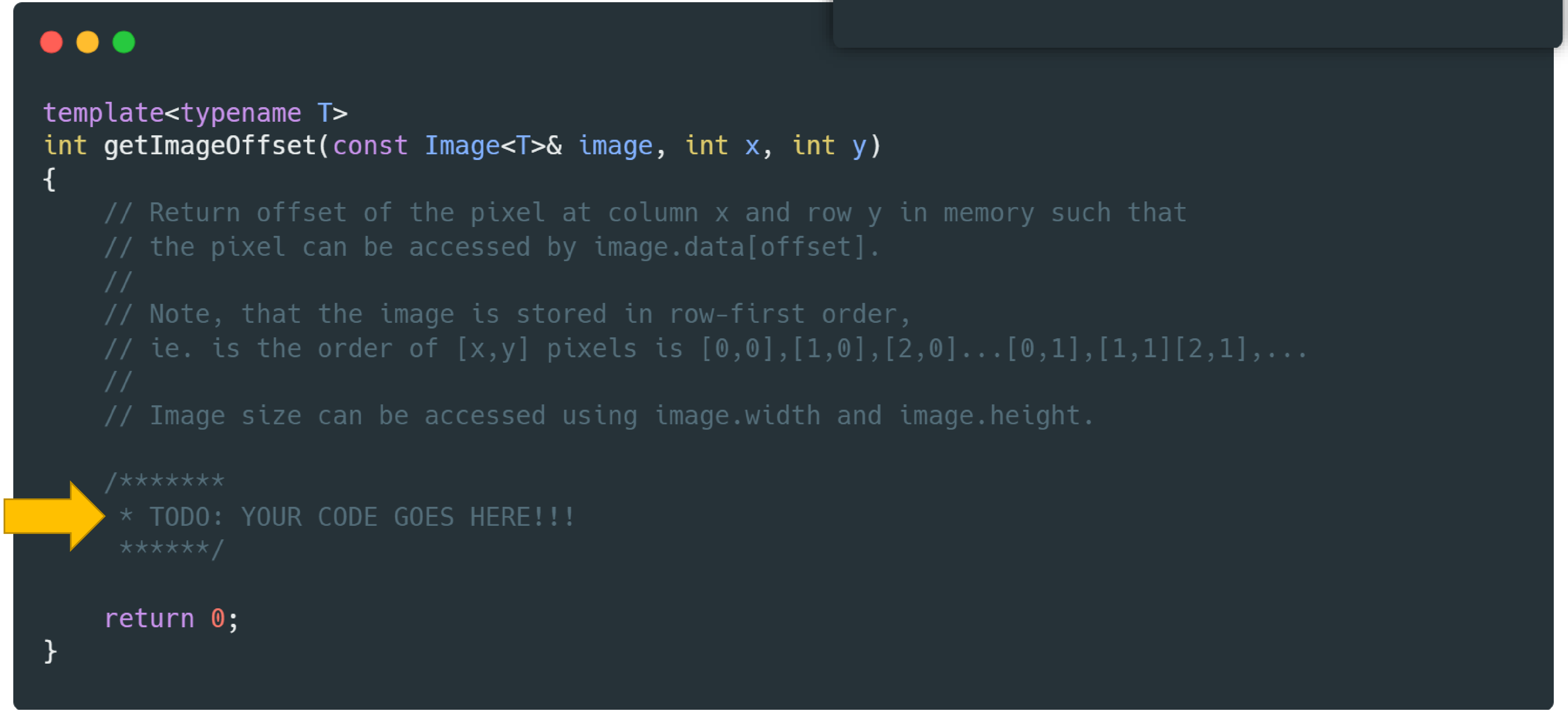
- Our image data are stored linearly in **1D array** row by row (see **Lecture 1**).
- => To make life easier, implement a function that returns the **memory offset** of $H(x, y)$:

```
image.data[getImageOffset(image,x,y)]
```

getImageOffset(...)



```
image.data[getImageOffset(image,x,y)]
```



```
template<typename T>
int getImageOffset(const Image<T>& image, int x, int y)
{
    // Return offset of the pixel at column x and row y in memory such that
    // the pixel can be accessed by image.data[offset].
    //
    // Note, that the image is stored in row-first order,
    // ie. is the order of [x,y] pixels is [0,0],[1,0],[2,0]...[0,1],[1,1][2,1],...
    //
    // Image size can be accessed using image.width and image.height.

    /*******
     * TODO: YOUR CODE GOES HERE!!!
     ******/

    return 0;
}
```

3. Compute gradients attenuation φ

- φ is a power function that compresses large gradients in ∇H.
- **Section 4** of the paper:

$$\varphi(x,y) = \frac{\alpha}{\|\nabla H(x,y)\| + \varepsilon}\left(\frac{\|\nabla H(x,y)\| + \varepsilon}{\alpha}\right)^{\beta}$$

$$\|\nabla H(x,y)\| = \sqrt{\nabla H_x^2 + \nabla H_y^2}$$

(Gradient norm)

  - β is a constant provided as a parameter (e.g., 0.35)
  - k is related to the multi-scale processing and we can ignore it
  - $\varepsilon = 10^{-3}$
  - α can be computed as:

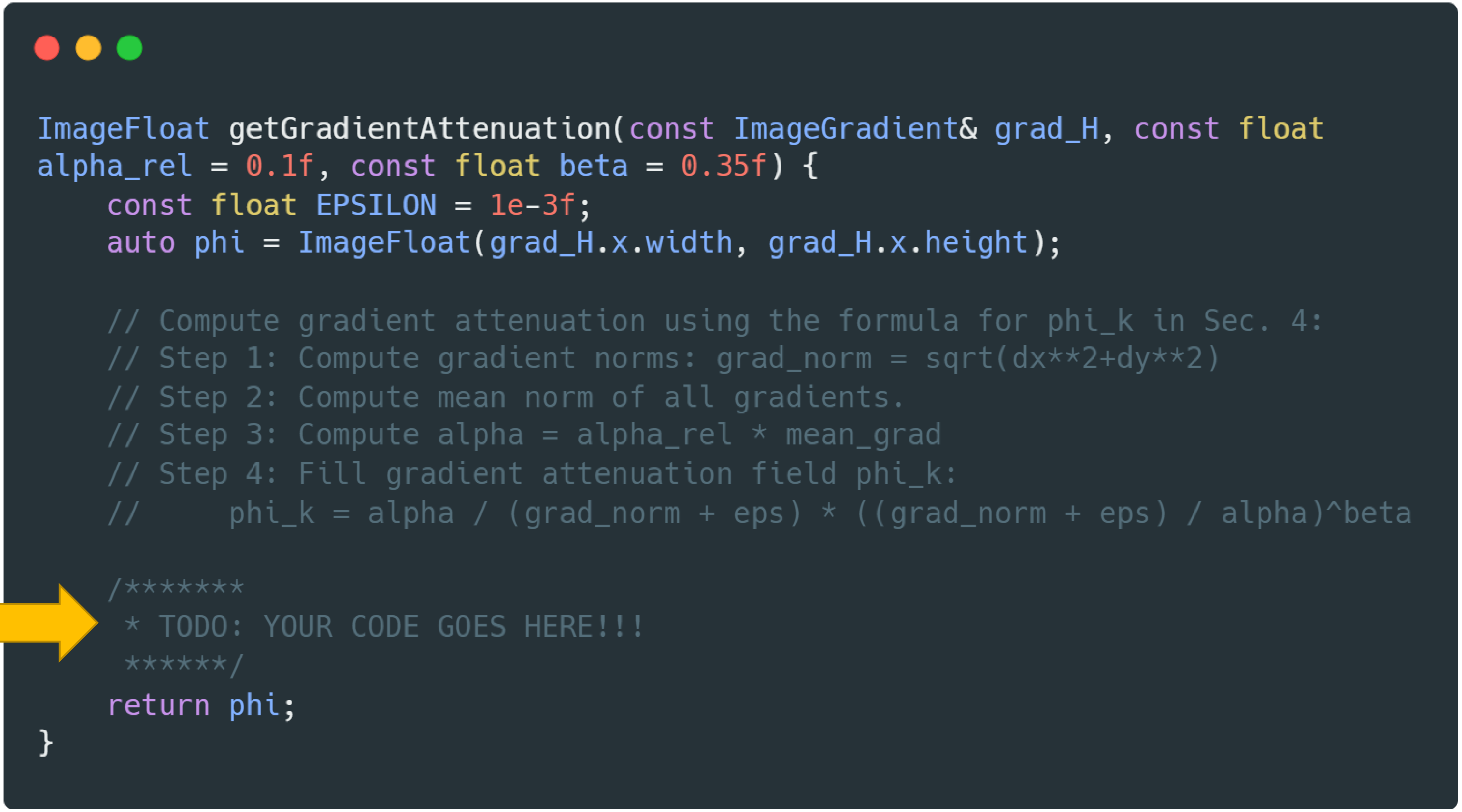
(mean gradient norm)

$$\alpha = \alpha_{rel} \cdot \frac{1}{N}\sum_{i=0}^{N} \|\nabla H\|$$

    - where α_{rel} is a constant provided as a parameter (e.g., 0.1)

TU Delft

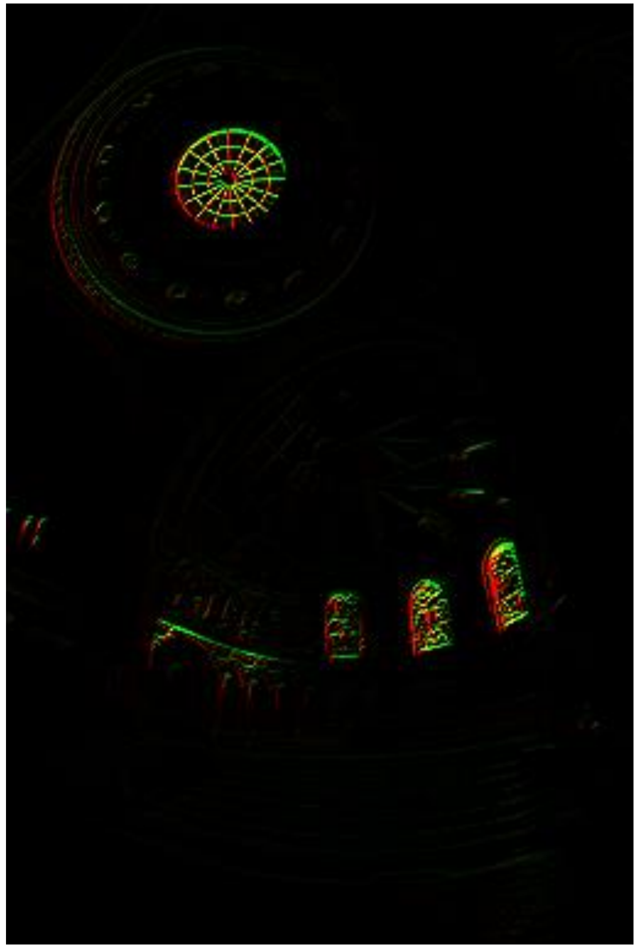
getGradientAttenuation(...)



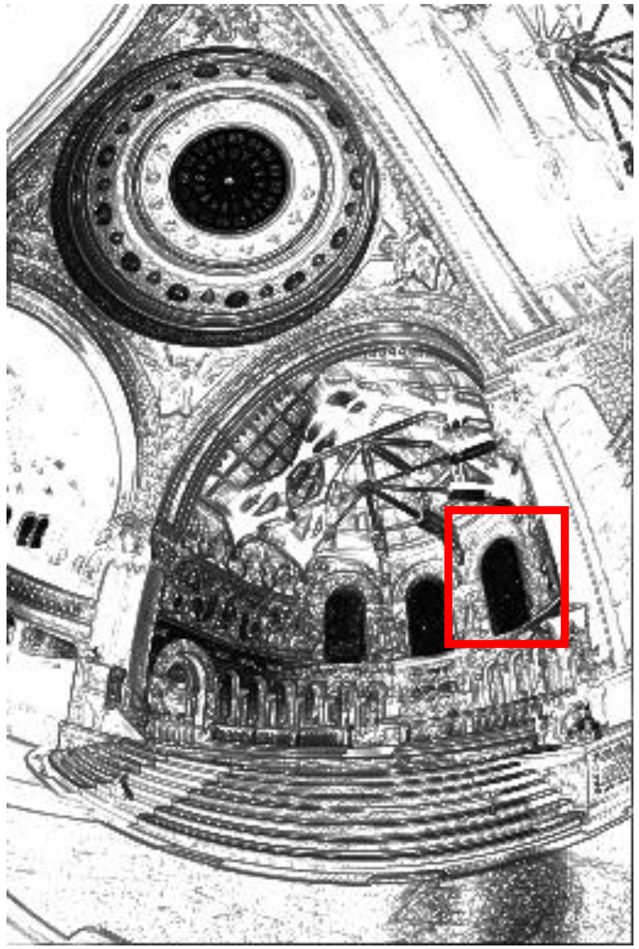
```cpp
ImageFloat getGradientAttenuation(const ImageGradient& grad_H, const float
alpha_rel = 0.1f, const float beta = 0.35f) {
    const float EPSILON = 1e-3f;
    auto phi = ImageFloat(grad_H.x.width, grad_H.x.height);

    // Compute gradient attenuation using the formula for phi_k in Sec. 4:
    // Step 1: Compute gradient norms: grad_norm = sqrt(dx**2+dy**2)
    // Step 2: Compute mean norm of all gradients.
    // Step 3: Compute alpha = alpha_rel * mean_grad
    // Step 4: Fill gradient attenuation field phi_k:
    //     phi_k = alpha / (grad_norm + eps) * ((grad_norm + eps) / alpha)^beta

    /*******
     * TODO: YOUR CODE GOES HERE!!!
     ******/
    return phi;
}
```

3. Compute gradients attenuation φ

∇H



φ



Strong attenuation of bright regions.

TU Delft

4. Compute attenuated divergence $\operatorname{div} G$

- I) Attenuate the gradients ∇H by φ (Section 3):

$$G(x,y) = \nabla H(x,y)\,\varphi(x,y)$$

- II) Compute divergence of G (Section 5):

$$\operatorname{div} G = \frac{\delta G_x}{\delta x} + \frac{\delta G_y}{\delta y}$$

$$\frac{\delta G_x}{\delta x} \approx G_x(x,y) - G_x(x - \mathbf{1}, y)$$

$$\frac{\delta G_y}{\delta y} \approx G_y(x,y) - G_y(x, y - \mathbf{1})$$

Reminder

$$\nabla H \colon \mathbb{R}^2 \to \mathbb{R}^2$$
$$\varphi \colon \mathbb{R}^2 \to \mathbb{R}$$
$$G \colon \mathbb{R}^2 \to \mathbb{R}^2$$
$$\operatorname{div} G \colon \mathbb{R}^2 \to \mathbb{R}$$

Left-sided differences!

(Compare to right-sided ∇H)

TU Delft

4. Compute attenuated divergence $\operatorname{div} G$

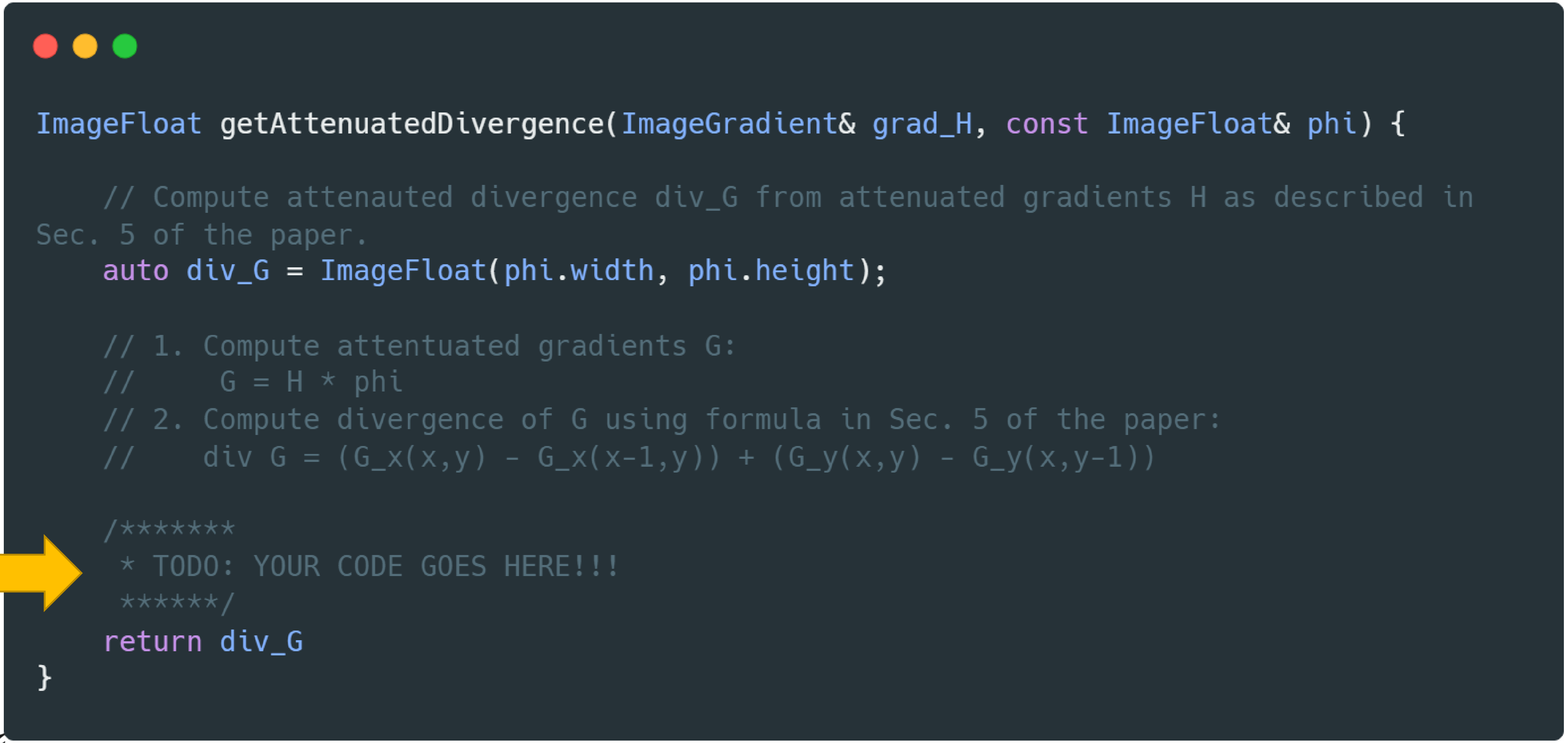
$$\operatorname{div} G = \frac{\delta G_x}{\delta x} + \frac{\delta G_y}{\delta y}$$

$$\frac{\delta G_x}{\delta x} \approx G_x(x, y) - G_x(x \mathbf{- 1}, y)$$

$$\frac{\delta G_y}{\delta y} \approx G_y(x, y) - G_y(x, y \mathbf{- 1})$$

- Assume zero padding => Pixels outside the image are 0 (black).
  - **Example**: Assuming a 5x5 px image:
    - $\frac{\delta G_y}{\delta y}(4,0) = G(4,0) - G(4, \mathbf{-1}) = G(4,0) - \mathbf{0} = G(4,0)$

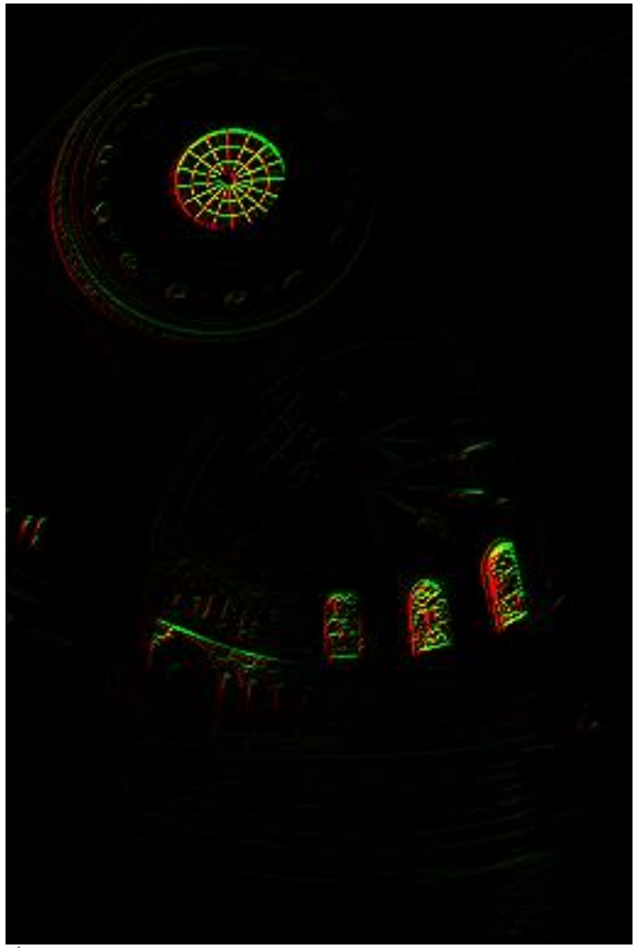
getAttenuatedDivergence(...)



```cpp
ImageFloat getAttenuatedDivergence(ImageGradient& grad_H, const ImageFloat& phi) {

    // Compute attenauted divergence div_G from attenuated gradients H as described in
Sec. 5 of the paper.
    auto div_G = ImageFloat(phi.width, phi.height);

    // 1. Compute attentuated gradients G:
    //      G = H * phi
    // 2. Compute divergence of G using formula in Sec. 5 of the paper:
    //    div G = (G_x(x,y) - G_x(x-1,y)) + (G_y(x,y) - G_y(x,y-1))

    /*******
     * TODO: YOUR CODE GOES HERE!!!
     ******/
    return div_G
}
```

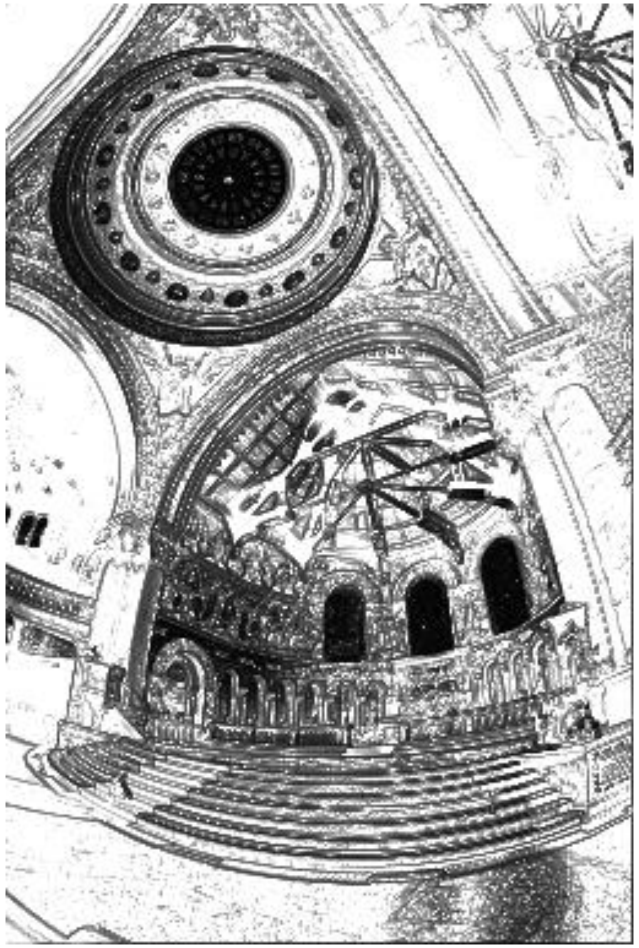
4. Compute attenuated divergence $\operatorname{div} G$

∇H



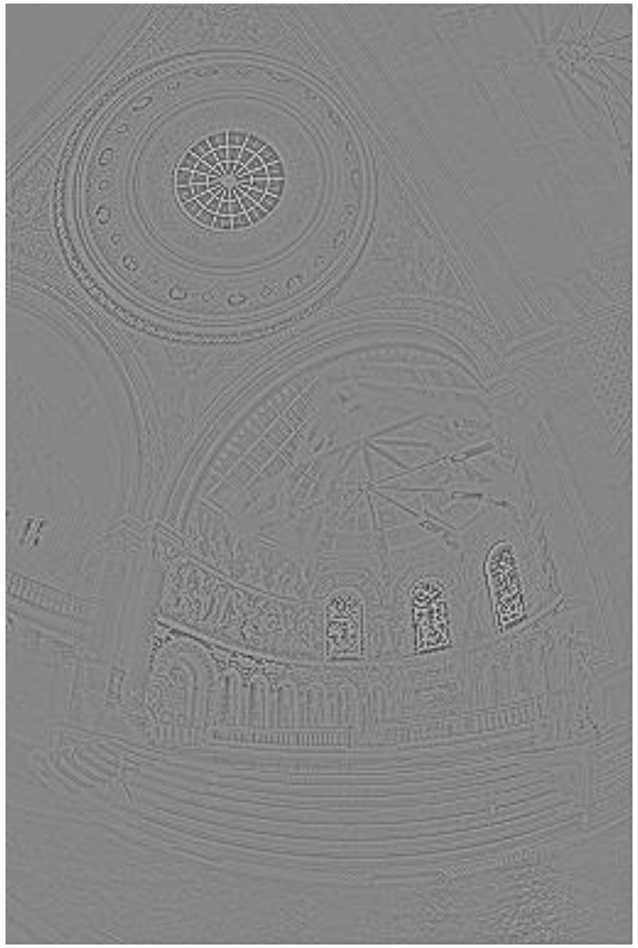
$\bullet$

φ



$\rightarrow$

$\operatorname{div} G$



5. Solve Poisson equation $\nabla^2 I = \text{div}\, G$

- We are looking for a scalar field $I: \mathbb{R}^2 \to \mathbb{R}$ such that (Sec. 3)

$$\nabla^2 I = \text{div}\, G$$

  - where (Sec. 5)

$$\nabla^2 I(x,y) \approx I(x+1,y) + I(x-1,y) + I(x,y+1) + I(x,y-1) - 4I(x,y)$$

- We will use a simple direct solver:
  - For a fixed number of iterations repeat:
    - For each pixel (x, y):
      - I) Find a value $I'(x,y)$ that minimizes $|\nabla^2 I(x,y) - div\ G(x,y)|$
      - II) Update $I = I'$

TU Delft

5. Update rule

$$\nabla^2 I^n(x,y) = div\ G(x,y)$$

$$I^n(x+1,y) + I^n(x-1,y) + I^n(x,y+1) + I^n(x,y-1) - 4 \cdot I^n\ (x,y) = div\ G(x,y)$$

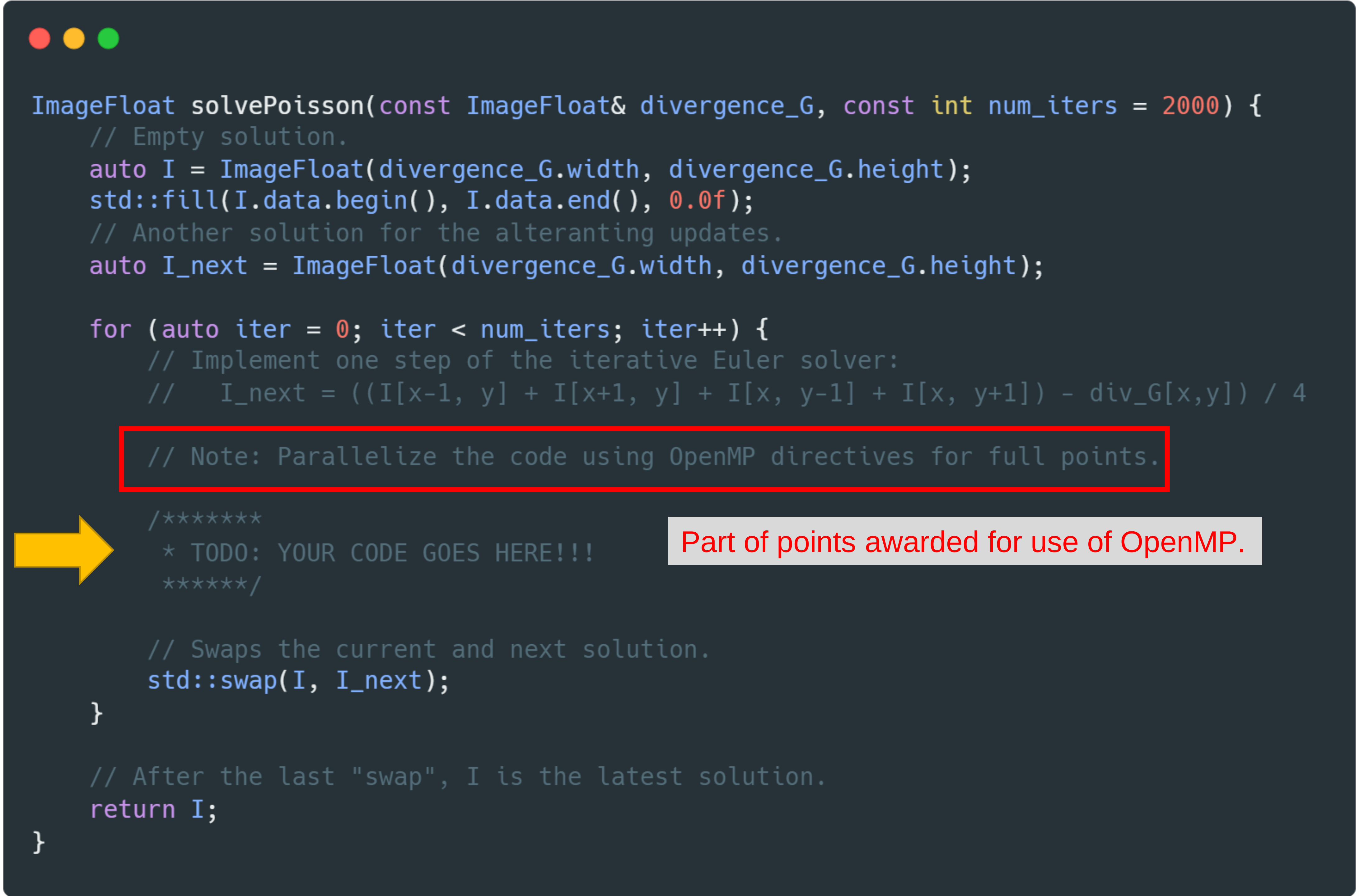
$\Downarrow$ Substitute: $I^n(x,y) \rightarrow I^{n+1}(x,y)$

$$I^n(x+1,y) + I^n(x-1,y) + I^n(x,y+1) + I^n(x,y-1) - 4 \cdot I^{n+1}\ (x,y) = div\ G(x,y)$$

$\Downarrow$ Solve for $I^{n+1}(x,y)$

$$I^{n+1}(x,y) = \frac{1}{4}\Big(\underbrace{\big(I^n(x+1,y) + I^n(x-1,y) + I^n(x,y+1) + I^n(x,y-1)\big)}_{} - div\ G(x,y)\Big)$$

Boundary conditions: Assume zero padding => Pixels outside the image are 0 (black).

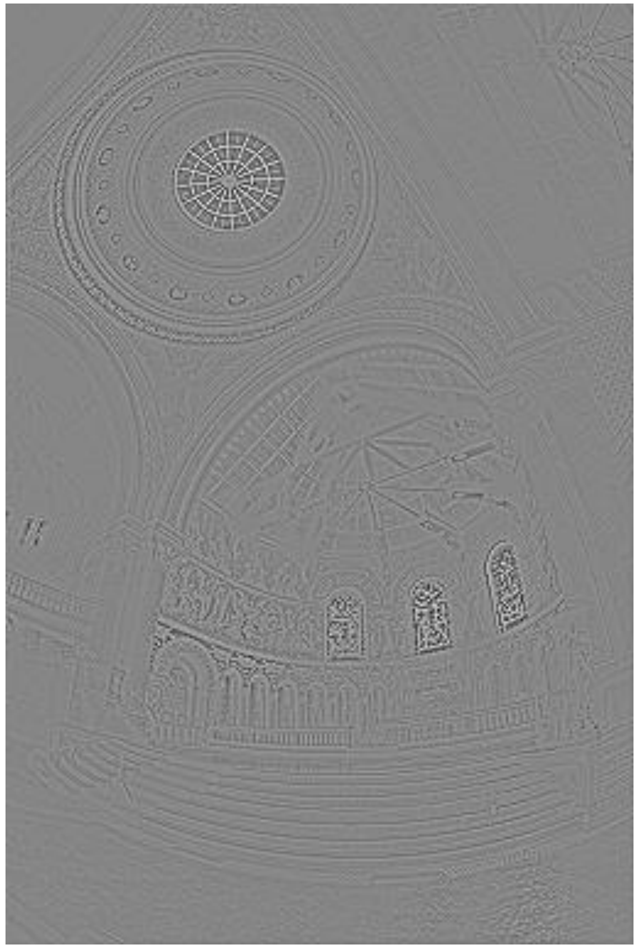


```
ImageFloat solvePoisson(const ImageFloat& divergence_G, const int num_iters = 2000) {
    // Empty solution.
    auto I = ImageFloat(divergence_G.width, divergence_G.height);
    std::fill(I.data.begin(), I.data.end(), 0.0f);
    // Another solution for the alteranting updates.
    auto I_next = ImageFloat(divergence_G.width, divergence_G.height);

    for (auto iter = 0; iter < num_iters; iter++) {
        // Implement one step of the iterative Euler solver:
        //   I_next = ((I[x-1, y] + I[x+1, y] + I[x, y-1] + I[x, y+1]) - div_G[x,y]) / 4

        // Note: Parallelize the code using OpenMP directives for full points.

        /*******
         * TODO: YOUR CODE GOES HERE!!!
         ******/

        // Swaps the current and next solution.
        std::swap(I, I_next);
    }

    // After the last "swap", I is the latest solution.
    return I;
}
```

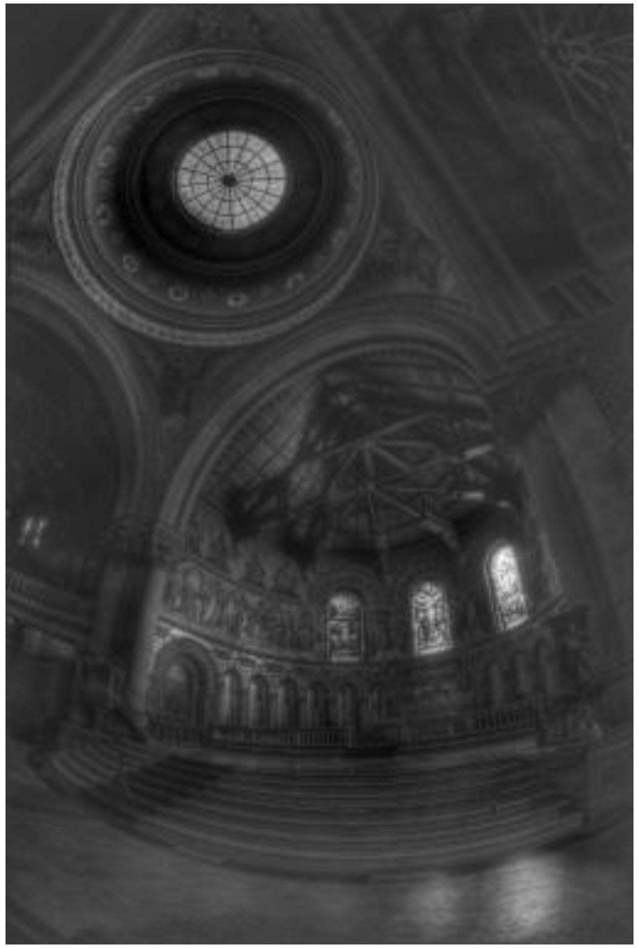
Part of points awarded for use of OpenMP.

5. Solve Poisson equation $\nabla^2 I = \operatorname{div} G$

$\operatorname{div} G$



I



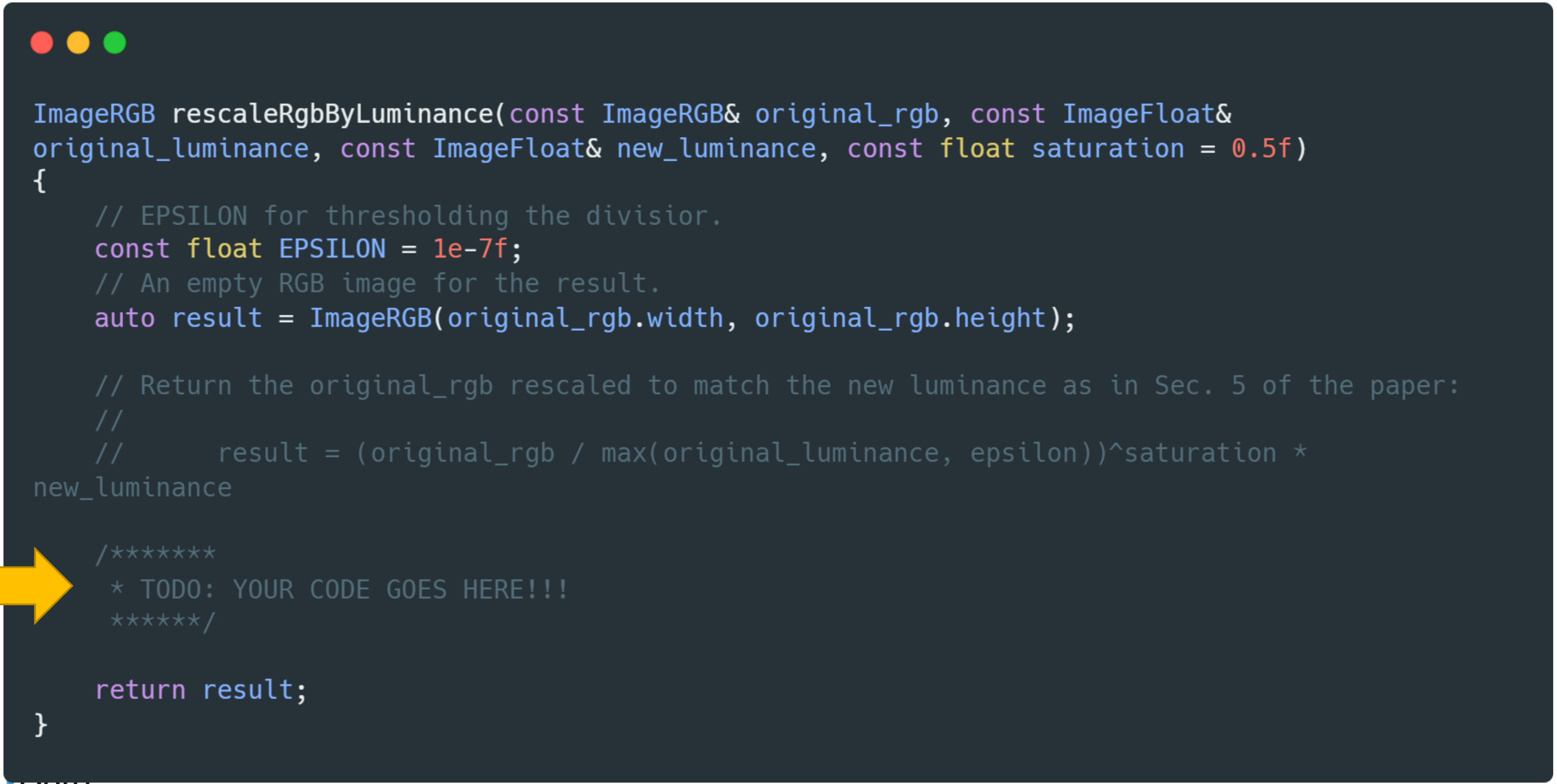
6. Rescale original RGB based on solution

- Scale input chrominance based on the solved luminance **(Sec. 5)** :

$$C_{out} = \underbrace{\left(\frac{C_{in}}{\max(L_{in}, \varepsilon_2)}\right)^{s}}_{\text{Saturation corrected chrominance}} L_{out}$$

  - C_{in} = original HDR image (RGB)
  - L_{in} = original luminance (from step 3)
  - L_{out} = solution of the Poisson equation (step 5)
  - s = constant controlling saturation (provided as a parameter)
  - $\varepsilon_2 = 10^{-7}$
  - C_{out} = tone mapped SDR image (RGB)

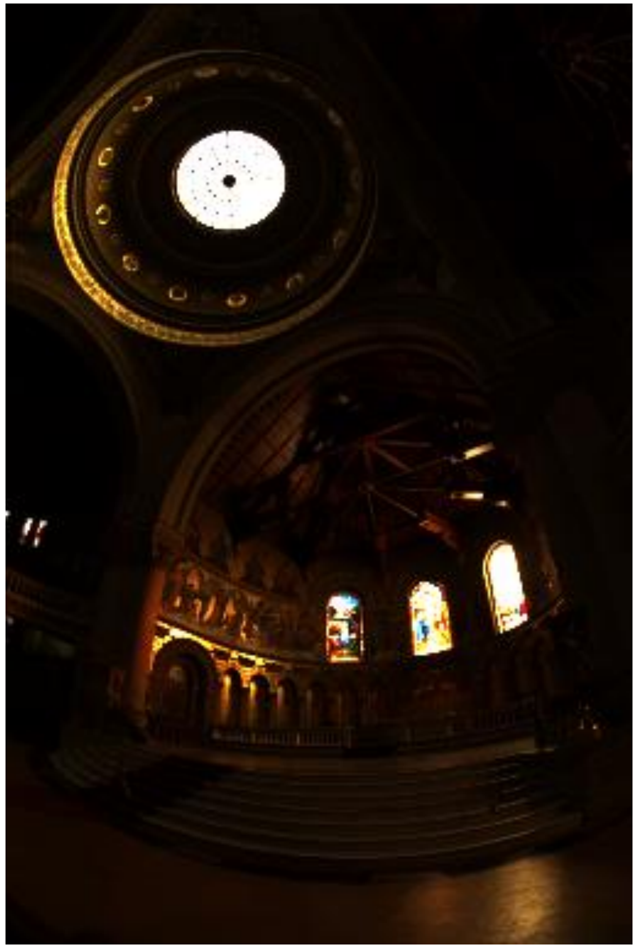
rescaleRgbByLuminance(...)



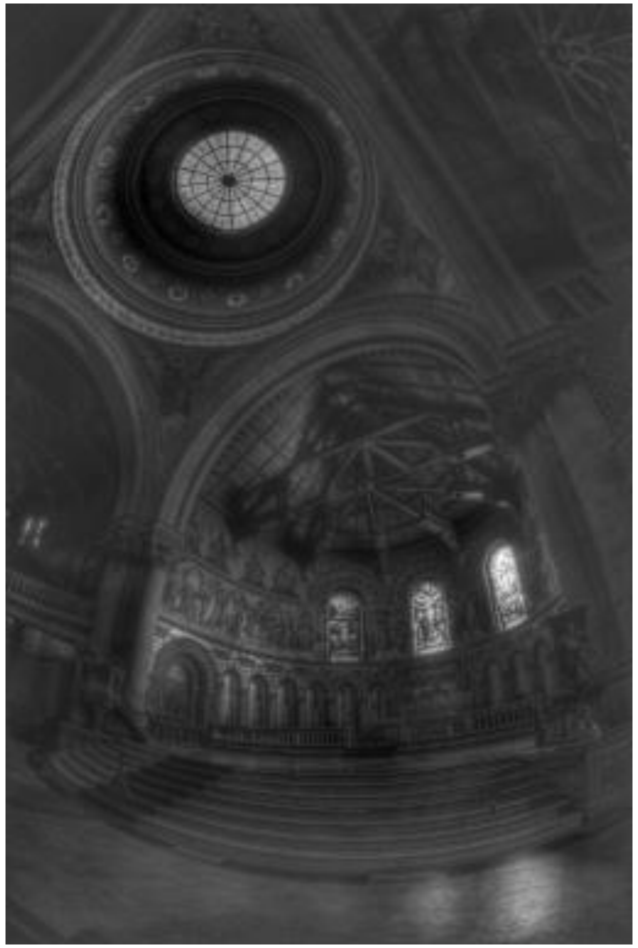
```cpp
ImageRGB rescaleRgbByLuminance(const ImageRGB& original_rgb, const ImageFloat&
original_luminance, const ImageFloat& new_luminance, const float saturation = 0.5f)
{
    // EPSILON for thresholding the divisior.
    const float EPSILON = 1e-7f;
    // An empty RGB image for the result.
    auto result = ImageRGB(original_rgb.width, original_rgb.height);

    // Return the original_rgb rescaled to match the new luminance as in Sec. 5 of the paper:
    //
    //      result = (original_rgb / max(original_luminance, epsilon))^saturation *
new_luminance

    /*******
     * TODO: YOUR CODE GOES HERE!!!
     ******/

    return result;
}
```

6. Rescale original RGB based on solution

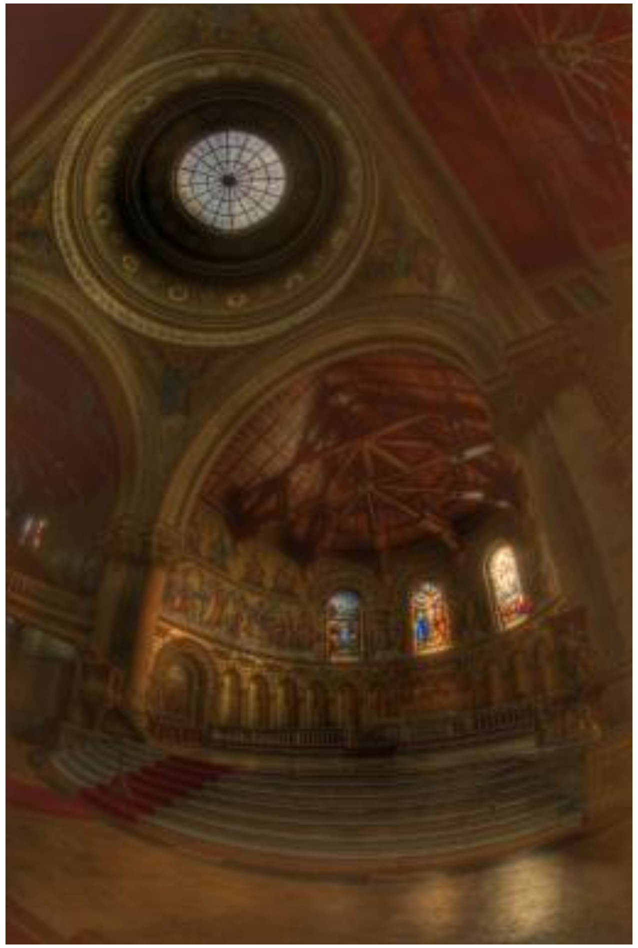
HDR RGB input



I



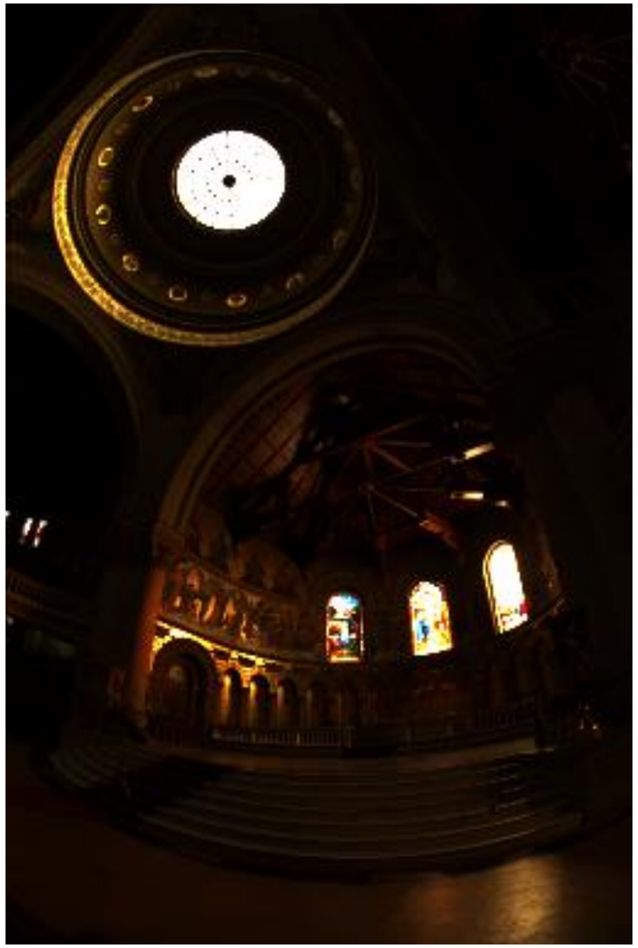
SDR result



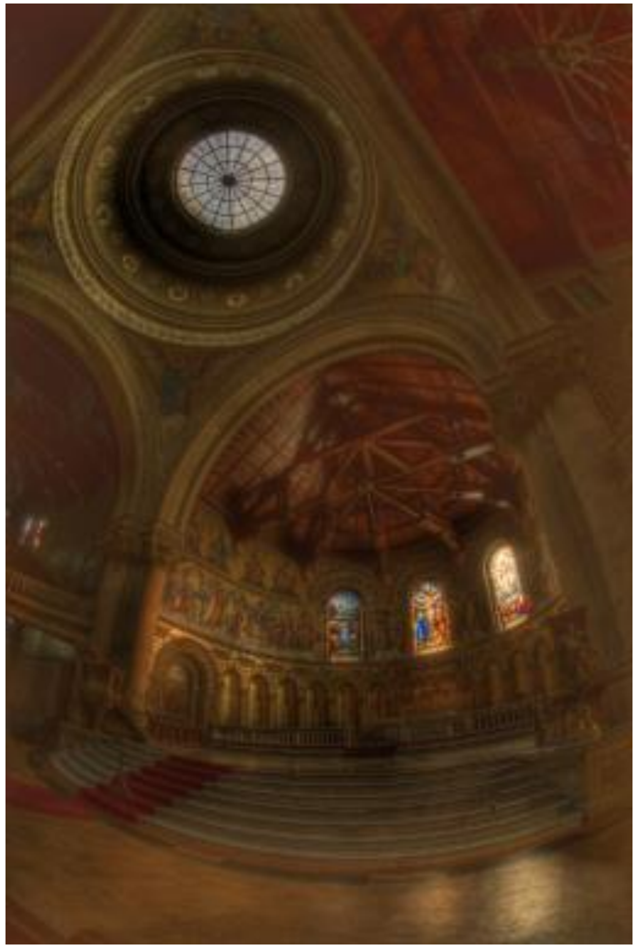
Limitations

- A different parameter choice.
- **Simplifications of the algorithm.**

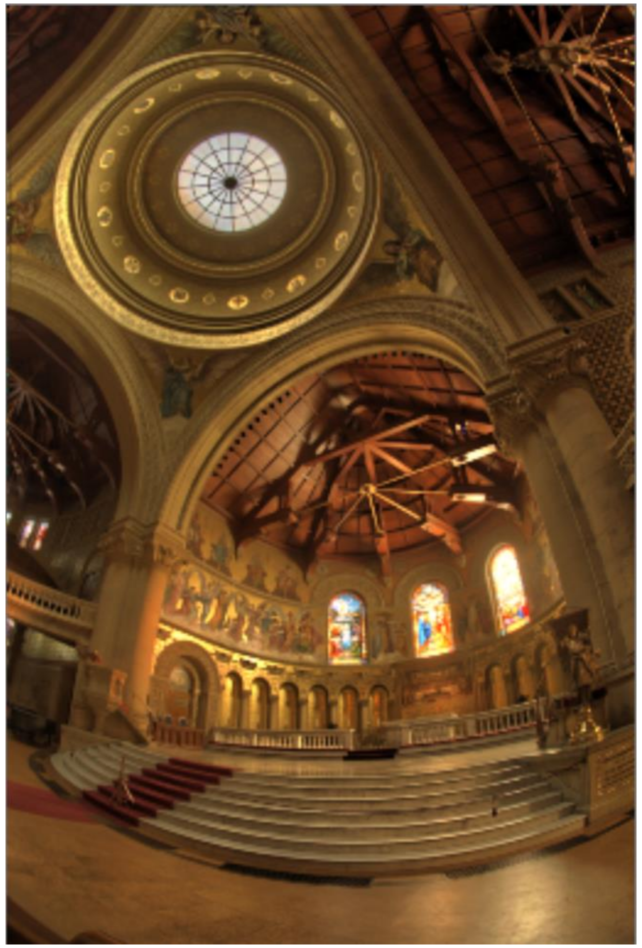
HDR



Our output



Author’s output



Grading

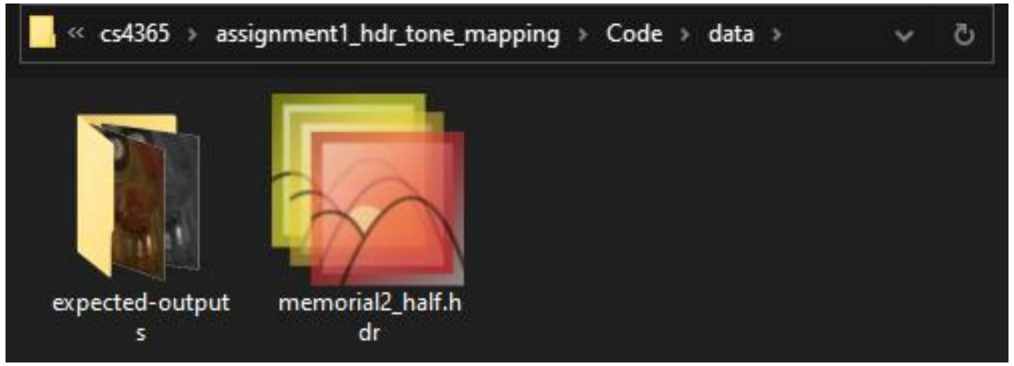
Step	Method	Maximum points
0a	getRGBImageMinMax(…)	1 + 1 point for OpenMP
	normalizeRGBImage(…)	1
0b	applyGamma(…)	1
1	rgbToLuminance(…)	1
2	getGradients(…)	3
3	getGradientAttenuation(…)	2
4	getAttenuatedDivergence(…)	3
5	solvePoisson(…)	3 + 1 point for OpenMP
6	rescaleRgbByLuminance(…)	1
	Total	18

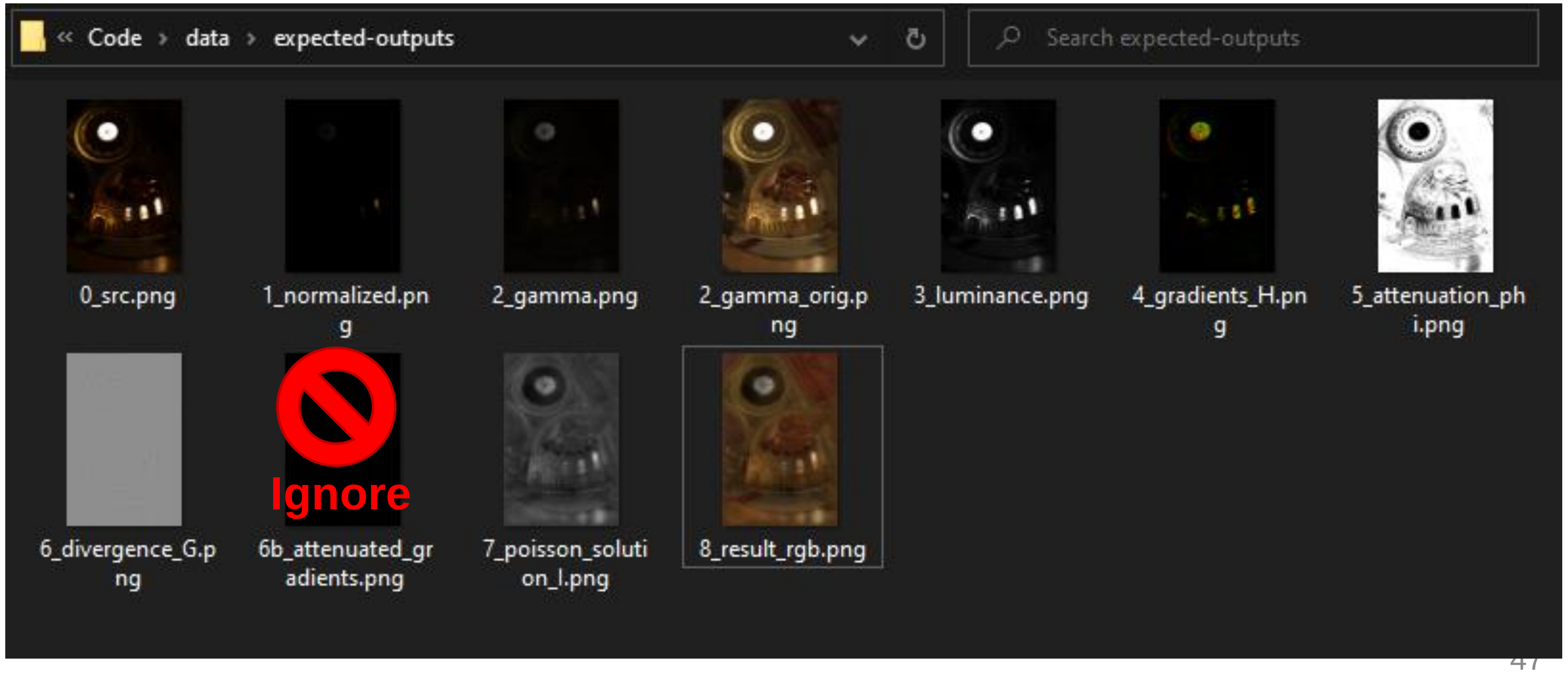
Partial points are awarded based on passed unit tests: $\text{grade} = 1 + \frac{\text{points}}{\text{max points}} * 9$

TU Delft

Testing

- Sample **inputs** provided in `./data`
- Reference **outputs** provided in `./data/reference-outputs`



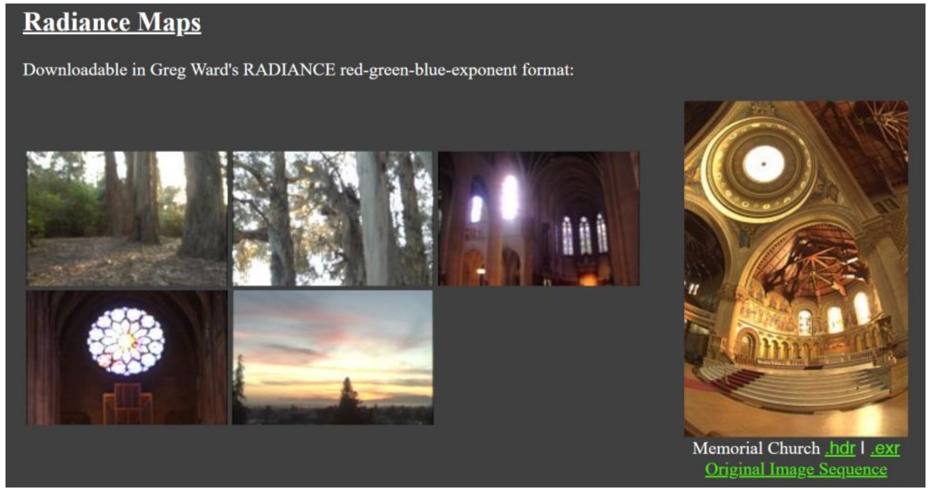


Testing

- **The evaluation is not done on the sample inputs only!**
  - Modify `main.cpp` as much as you want to implement new tests!
- **main.cpp** is **not** uploaded to Brightspace so any changes there have **no effect** on the final score
  - **=> Modify `main.cpp` to implement your tests.**
- **`your_code_here.h`** is the only **evaluated** file submitted to Brightspace.
  - **=> Pull all your solutions only to `your_code_here.h`**
- **Do not modify anything else (e.g., `helper.h`)!**
  - Such modifications would not be available during evaluation on Brightspace and the code would likely not even compile.

Additional HDR images

- HDR image from Paul Debevec: https://www.pauldebevec.com/Research/HDR/



Radiance Maps

Downloadable in Greg Ward's RADIANCE red-green-blue-exponent format:

Memorial Church .hdr | .exr
Original Image Sequence

- Look for other online HDR images in the ***.hdr** format
  - In case of loading issues, try to load & save the image in the **LuminanceHDR**
- Or convert any ***.exr** image to ***.hdr** using the **LuminanceHDR**

Expected behavior

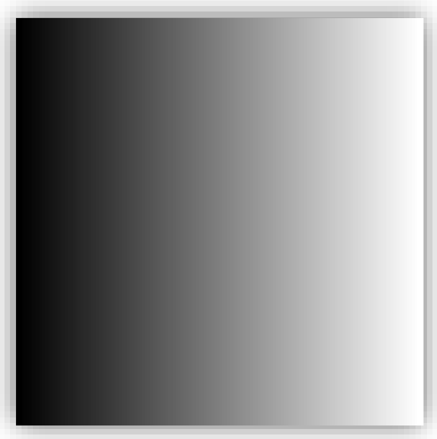
- The code should implement the properties described in here and in the comments.
  - Equations, algorithms,…
- Unspecified cases (e.g., what happens when dividing by zero), should be solved **sensibly** (no application crash).
- Simpler algorithms (e.g., finding min value of an array) are expected to produce **exactly accurate** result.
- More complex algorithms allow for variation in results due to design choices. I.e., the output should look **similar**, but pixels values may be slightly different.

~~Algorithm improvements~~

- Implement the algorithm exactly **as described in this document** (including the simplifications).
- Improvements could lead to different results and **problems during evaluation**.
- You will have an opportunity for invention in the final project.

Unit test suggestions

- Small inputs (e.g., 1x2 px image)
- Input image with constant values
  - Normalization => min=max => division by zero => does the code crash?
- Test trivial and edge cases.
  - The input is already SDR. What happens?
  - The input is an empty image? What happens?
  - The input is a constant color gradient. What happens?
- Test extreme pixels.
  - One pixel on each boundary and all 4 corners.



Unit test implementation

- **Simple approach**: Implement test methods in `main.cpp` and call them from **`main()`**.
  - Either **write** images to drive or compare values using **if** tests.

- **Systematic approach**: Use a unit testing framework.
  - GoogleTest: https://google.github.io/googletest/primer.html
  - Catch2: https://medium.com/dsckiit/a-guide-to-using-catch2-for-unit-testing-in-c-f0f5450d05fb
  - ...

  - Recommended for larger projects, **not really needed for the small assignments**.

Submission

- **Deadline**: Sunday Oct 1, 2023 (23:59 local time)

- Submit `your_code_here.h` to:
  Brightspace->Assignments->Assignment 1: HDR Tone Mapping

- **Late submissions** can be submitted to "Assignment 1 – Late Submissions" with a penalty:

$$\text{adjusted grade} = \text{grade} - 1 - \left\lceil \frac{\text{minutes late}}{10} \right\rceil$$

TUDelft