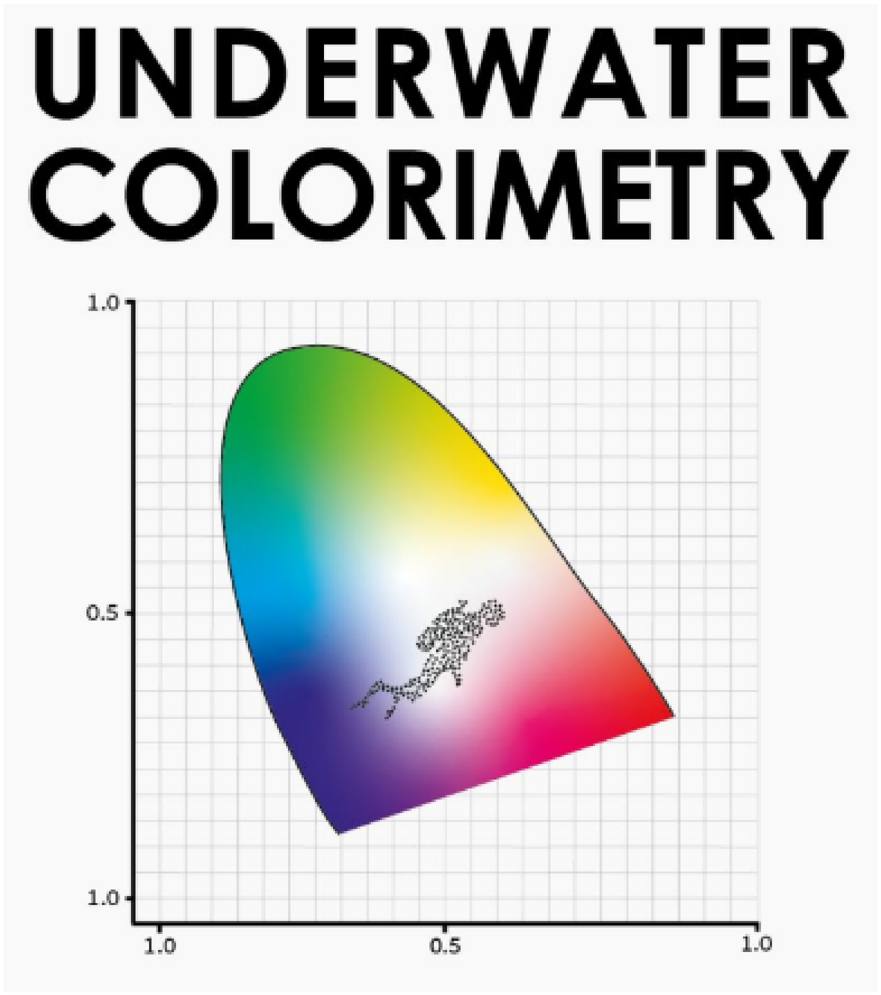
Underwater Colorimetry - fall 2024
Lab 1

IUI

January 2025



Basic image formation & RAW image manipulation

Session objectives:

1. Hands-on experience with the fundamentals of image formation model.
2. Linearity check of camera sensors.
3. Working with different image formats based on the required application.

Required equipment:

1. Laptop
2. MATLAB or Python
3. Camera able to capture RAW images

Provided data:

Underwater Colorimetry GitHub Repository

Download the repository as a .zip file. It is very important to place the repository in a folder whose path **does not** contain any spaces or special characters!

Provided file	**Comment**
`MacbethColorCheckerReflectances.csv`	Reflectances of all patches of a Macbeth ColorChecker: The 1-24 corresponds to the patches in the numbering order given here.
`illuminant-D65.csv,illuminant-A.csv`	Spectral power distributions of two CIE standard illuminants: daylight: D65, incandescent: A.
`NikonD90.csv, Canon_1Ds_Mk_II.csv`	Spectral responses of two cameras, Nikon D90 and Canon 1Ds II.
`CIEStandardObserver.csv`	CIE 1931 2-degree standard observer curves.
`NikonImage.NEF`	An image containing a Macbeth ColorChecker acquired with a Nikon D90 in RAW format.
`CanonImage.CR2`	An image containing a Macbeth ColorChecker acquired with a Canon 600D in RAW format.

Table 1: Provided files and their descriptions.

Exercise 1

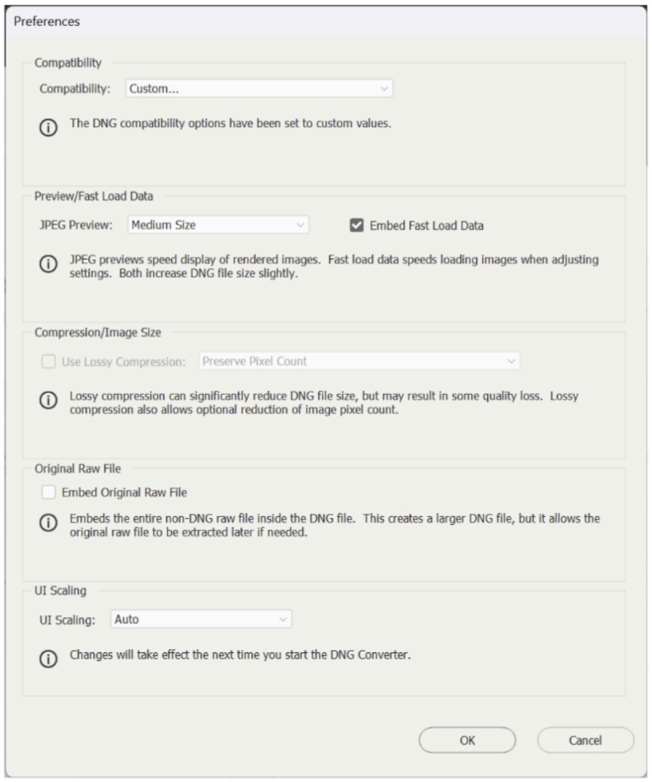
Converting a RAW image to a linear image

Use a camera that can capture an image in RAW format to take a picture of a Mcbeth ColorChecker in daylight. Some smartphones might be able to do this. If your smartphone cannot produce RAW images, use one of the provided GoPros. While taking the images, make sure not to cast shadows of your hand or camera onto the color chart.

Fun fact:
An issue that can arise when working with color images is the loss of linearity due to the codec. Unfortunately, the most widely spread image codec .jpg is not linear. Together, we will utilize a custom image pipeline to obtain linear .tiff and .png files from proprietary RAW images. You will need to make use of this workflow when you are getting your files ready for exercise 2 of this lab and later sessions.

Steps

1. Download your RAW image to your computer and place it in a designated folder.
2. Download Adobe DNG Converter for your operating system from here. (https://helpx.adobe.com/camera-raw/using/adobe-dng-converter.html).
3. Double click and open the program. In section 1, click the "Select Folder" button and point it to "raw" folder you just made.
4. In Section 2, click "Select Folder", and choose the destination folder where you prefer your .dng files to go. It is suggested that you have a folder named "dng" at the same level as the "raw" folder.
5. No need to do anything in Section 3.
6. In section 5, click the "Change Preferences" button. In the next screen that opens up, click the "Compatibility" drop down list. Select "Custom" and check both "linear" and "uncompressed".
7. Set JPEG preview to Medium Size.



Preferences

Compatibility

Compatibility: Custom...

The DNG compatibility options have been set to custom values.

Preview/Fast Load Data

JPEG Preview: Medium Size [✓ Embed Fast Load Data]

JPEG previews speed display of rendered images. Fast load data speeds loading images when adjusting settings. Both increase DNG file size slightly.

Compression/Image Size

Use Lossy Compression: Preserve Pixel Count

Lossy compression can significantly reduce DNG file size, but may result in some quality loss. Lossy compression also allows optional reduction of image pixel count.

Original Raw File

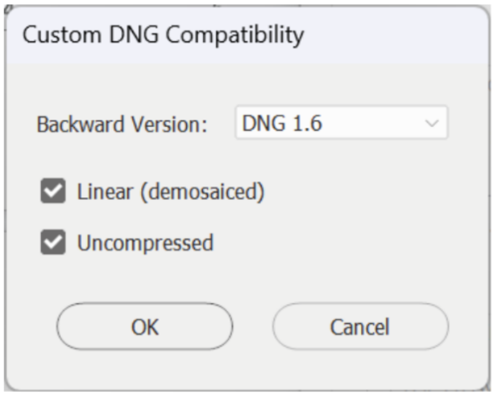
Embed Original Raw File

Embeds the entire non-DNG raw file inside the DNG file. This creates a larger DNG file, but it allows the original raw file to be extracted later if needed.

UI Scaling

UI Scaling: Auto

Changes will take effect the next time you start the DNG Converter.



Custom DNG Compatibility

Backward Version: DNG 1.6

[✓ Linear (demosaiced)]

[✓ Uncompressed]

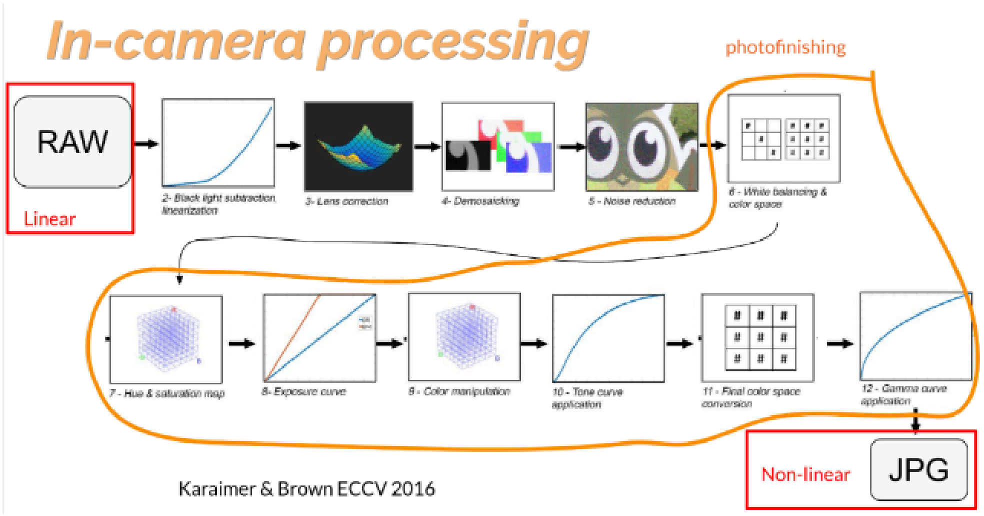
8. Leave "Lossy Compression" box UNchecked.
9. Leave Embed Original Raw File UNchecked.
10. Click OK to close the second window.
11. Click "Convert" to start raw to dng conversion! The dng file(s) should appear in the output folder you specified.
12. Now we will convert DNG images to .tiff files. Create a folder called "tiff" and one called "dng" at the same level as where you have "raw" and "dng" files.
13. Open Matlab. We will run the:

    ```
    Exercise1.m
    ```

    script to create the tiff and png versions of the dng images you made with Adobe DNG converter in the previous step. This function relies on other functions in the folder named: "camera-pipeline-nonUI-master" in the repository. Make sure the repository, and all subfolders are on your path (use addpath, genpath functions if needed).

14. The script will also run the tiff2png function to create smaller and compressed files (png uses lossless compression, unlike jpg), because our lab computers cannot handle hundreds of full-size tiff files. In addition to converting a .tiff image to .png, this script will resize the images to half their original size.
    Here is how to run the function:

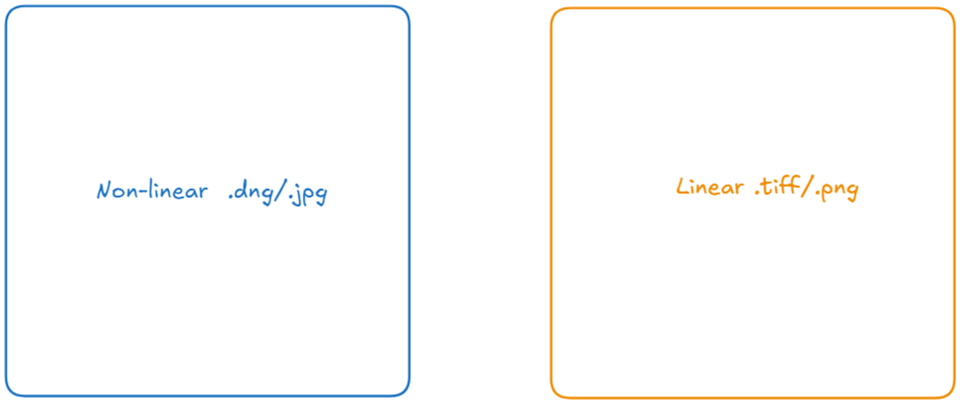
```
The arguments will be the full path of your folders,
something like this:

pngPath = L:\Underwater_Colorimetry_2023\Mydata\png
tiffPath = L:\Underwater_Colorimetry_2023\Mydata\tiff

stage = 4
(this is the step just before photofinishing starts
in the digital camera pipeline).
```


In-camera processing
photofinishing
RAW
Linear
2- Black light subtraction, linearization
3- Lens correction
4- Demosaicking
5 - Noise reduction
6 - White balancing & color space
7 - Hue & saturation map
8- Exposure curve
9 - Color manipulation
10 - Tone curve application
11 - Final color space conversion
12 - Gamma curve application
Non-linear
JPG
Karaimer & Brown ECCV 2016

Well done! Now you have a .tiff image, that is linear with radiance, that you can open in Matlab or python, and manipulate. But it's huge in size, isn't it? Let's reduce it a bit, without losing its linearity.
RAW images are naturally very dark (they do not contain any non-linearities that enhance their brightness), so to visualize what is in the scene, you may multiply an image with 2 or 3 to brighten it when you display it (don't save it with brightened values)!

Include in your report - Exercise 1

1. A “brightened”, side by side comparison of the non linear .dng/.jpg and linear .tiff/.png images.
2. Discuss whether there are any visible differences.



Exercise 2

Check the linearity of a sensor from a photo containing a color chart

As we learned in the lecture today, it is important to know certain characteristics of your camera before using it to collect scientific data. One of the most important properties is a sensor that is linear.

But how do you check for linearity of your camera's sensor?

In this exercise, you will use your own image from exercise 1 as well as two RAW images provided to you to check for sensor linearity.
From the two images provided (CanonImage, NikonImage) as well as your own:

1. Extract average RGB values of the achromatic patches (grays).
2. Plot these values against the Y value of each achromatic patch (you can calculate that yourself or use published values).
3. Check if the provided images are linear.

Include in your report - Exercise 2

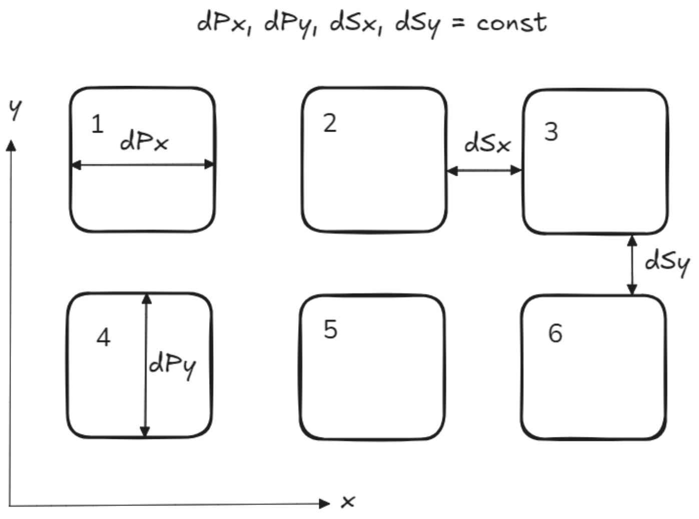
1. Plots of Y-RGB, checking for linearity.
2. Discuss what the data show.
3. Are the responses of all cameras linear? Why? Why not?
4. How are the images different? What do you see?
5. Explain how you got average RGB values for each patch.

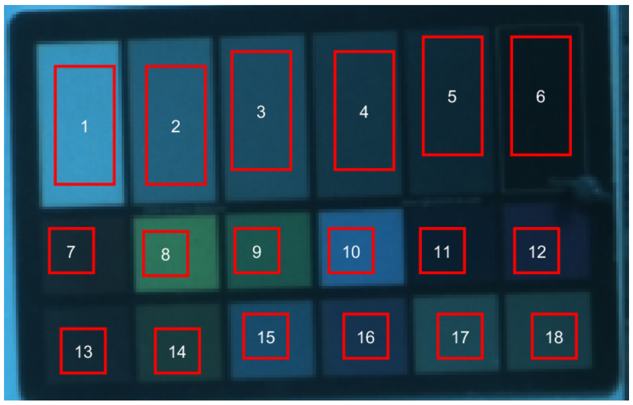
Guidance for writing the code

1. Steps to extract average RGB values of the achromatic patches (grays).
   To get the average RGB values of each achromatic patch, or any patch in general, we will need a mask.

   Step 1: Create a squared mask which will allow you to obtain average RGB values of each patch.
   You can write the mask algorithm as you see fit but remember that there are known parameters in the scene which will simplify the problem, e.g.

   - Size of the color-chart.
   - Size of each patch.
   - Distance between patches.





   Step 2: Inspect the masks on top of the image.

   Better fit = Better results

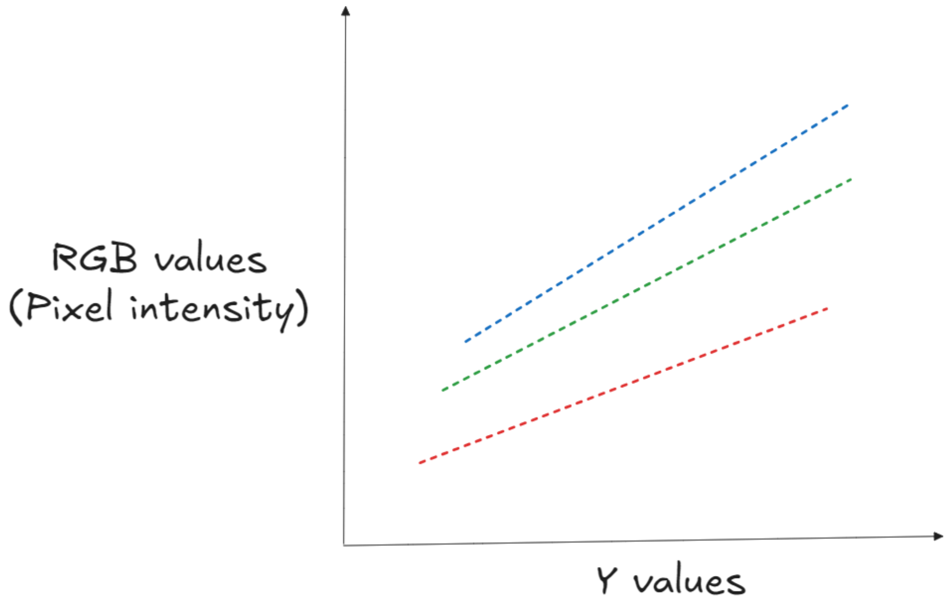
   Make adjustments if necessary so that in each mask there will be nothing else, i.e. noise, other then the dedicated patch.

Step 3: Get average RGB values within each mask.
It is recommended to write a dedicated function such as:

```
avgRGB = getAverageRGB(inImg, masks)


% inImg: Input image (NxMx3 matrix)

% masks: A cell array containing mask
coordinates as [x, y, width, height] for each region

% avgRGB: Struct containing average RGB
values for each mask
```

2. Once we obtained the RGB values for each patch, plotting RGB vs Y values should be simple task. Consider what you expect to see and why.

3. Linearity check

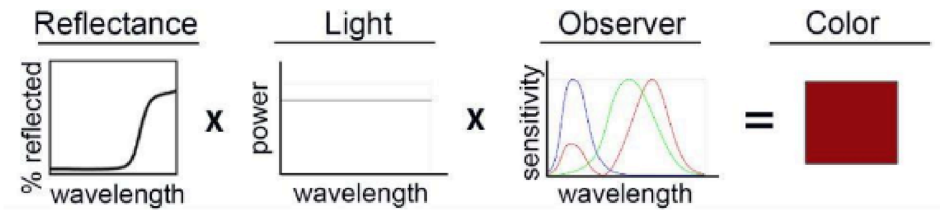


Consider:

- When and why should we apply white-balance?
- How white-balancing would have change your results?

Exercise 3

Simulate taking a "photo" of a Macbeth ColorChecker in air under a given illuminant with two different cameras



The goal is to see how the components of the image formation model influence the appearance of the image and how white balancing works and what it does.

You will learn how to visualize the difference in color data in chromaticity diagrams to answer the following questions:

- Do two images of the same scene taken by different cameras look the same?
- How can we compare color images taken with two different cameras?

	Nikon	Cannon	GoPro
Illuminant A	R L S'	R L S'	R L S'
Illuminant D65	R L S'	R L S'	R L S'

Include in your report - Exercise 3

Steps 1 - 3

1. 4 white-balanced figures:
   - 2 Nikon
   - 2 Cannon

   Each camera under 2 different lighting conditions (A & D65).
2. 4 **not** white-balanced figures:
   - 2 Nikon
   - 2 Cannon

   Each camera under 2 different lighting conditions (A & D65).

Show the color charts side-by-side, as “taken” by the 2 cameras for each illuminant.

Are there differences between the resulting charts?
Why?
Discuss. 1 paragraph maximum!

Steps 4 - 6

1. Two chromaticity diagrams showing the xy data of both cameras for the 24 patches of the color chart. One for each illuminant.

Writing the code

Step 1: Simulate a Macbeth ColorChecker

- Get the necessary data from the GitHub repository:
  - Reflectances of Macbeth ColorChecker.
  - Cameras spectral sensitivities (Nikon, Cannon and GoPro).
  - Light data (illuminant D65 and illuminant A).
- Make sure that all the imported data is on the same wavelength range, if needed use the MATLAB function interp1 and Interpolate to a common range (400:10:700) for calculations.

```
%   Vq = INTERP1(X,V,Xq)
interpolates to find Vq,
the values of the underlying
function V=F(X) at the query
points Xq.
```

- **Calculate radiance** for the ColorChecker for a given illuminant and a given camera. It is recommended to write your own dedicated function to calculate radiance.
  Further details:

```
radiance = getradiance(reflectance,light,observer)

% This function calculates radiance
from given reflectance, light, and
observer spectra.
```

As showed in the lecture, calculating radiance can be expressed as a simple matrices multiplication.

$$Reflectance = \rho \in R^{(m\times n)}$$

$$Light = L \in R^{(1\times n)}$$

$$Observer = S_c \in R^{(3\times n)}$$

Where:

- m is the number of different reflectances.
- $m = 24$ for a MacbethColorChecker.
- N is the number of wavelength steps.

To take advantage of the power in matrices multiplication we will need to check our dimensions.

$$Radiance = \rho \cdot diag(L) \cdot S_c'$$

And dimension wise:

$$(24 \times 3) = (24 \times 31) \cdot (31 \times 31) \cdot (31 \times 3)$$

- **Visualize Colors**
  Write the function:

```
function mcc = visualizeColorChecker(RGB)

% RGB is expected to be a matrix of 24 x 3.
% The input colors are expected to be in the
right (typical) order for a Macbeth ColorChecker.
```

The visualizeColorChecker function takes a 24×3 matrix of RGB values representing the Macbeth ColorChecker.

- **visualizeColorChecker(RGB) Logic:**
  1. A for Loop (for i = 1:24) iterates over each row of the RGB matrix, where RGB(i,:) represents the RGB values of the i-th color patch.
  2. For each color patch, it calls the function visualizeColor with the RGB values and a size of 100 pixels (a default size for visualization).
  3. The result is stored in cell array called *imgs*.
  4. After generating the image for each color patch, use:
     ```
     imtile(imgs, ’GridSize’, [4 6])
     ```
     The *imtile()* function will arrange the images in a grid of size 4×6 (which corresponds to the typical layout of a Macbeth ColorChecker with 24 color patches).
  5. **Output**:
     mcc: The combined image representing the entire Macbeth ColorChecker.

For each of the 24 colors, you will need to cal, and write, a second function called **visualizeColor** to generate a color patch.

```
function testPatch = visualizeColor(RGB,M)

% The size of the square image patch to create.
% Set M = 100.
```

These patches are arranged into a 4×6 grid to visually mimic the Macbeth ColorChecker.

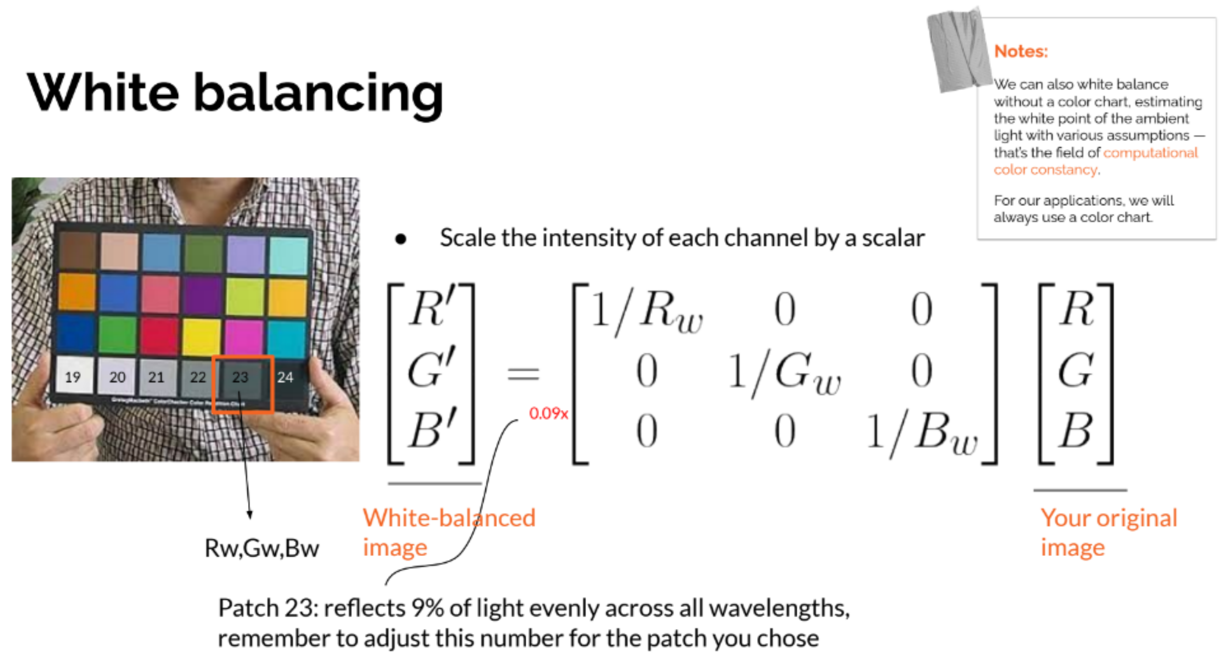
- **visualizeColor(RGB,M) Logic**
  1. **Input:**
     (a) **RGB**: A 1×3 vector representing the RGB values for a single color.
     (b) **M = 100**
  2. **Initialization:**
     A M×M×3 matrix (representing an RGB image) is initialized to zero, where each color channel (red, green, blue) is represented by a separate slice along the third dimension.
     ```
     testPatch = zeros(M,M,3)
     ```
  3. **Looping over color channels:**
     ```
     for j = 1:3
     ```
     (a) The loop fills the color channels (red, green, blue) of the matrix *testPatch* using the values from the RGB input vector.
     (b) The RGB values from the input are assigned to the respective color channels across the entire M×M patch.

4. **Output:**
   testPatch: A square patch of size M×M, filled with the specified RGB color.

```
function testPatch = visualizeColor(RGB,M)

if nargin<2
    M = 500;
end

testPatch = zeros(M,M,3);
rows = 1;
cols = 1;

for j = 1:3
    testPatch(1 + (rows-1)*M:M*rows,1+(cols-1)*M: ...
    M*cols,j) = RGB(j);
end
```

Step 2: White-balance

- Choose a gray patch (23rd) for white balancing (9% reflectance).
- Divide radiance values by the selected patch value.


19
20
21
22
23
24

White balancing

Notes:

We can also white balance without a color chart, estimating the white point of the ambient light with various assumptions — that's the field of computational color constancy.

For our applications, we will always use a color chart.

- Scale the intensity of each channel by a scalar

$$\underbrace{\begin{bmatrix} R' \\ G' \\ B' \end{bmatrix}}_{\text{White-balanced image}} = \begin{bmatrix} 1/R_w & 0 & 0 \\ 0 & 1/G_w & 0 \\ 0 & 0 & 1/B_w \end{bmatrix} \underbrace{\begin{bmatrix} R \\ G \\ B \end{bmatrix}}_{\text{Your original image}}$$

0.09x

Rw,Gw,Bw

Patch 23: reflects 9% of light evenly across all wavelengths, remember to adjust this number for the patch you chose

Step 3: Visualize White-Balanced Colors

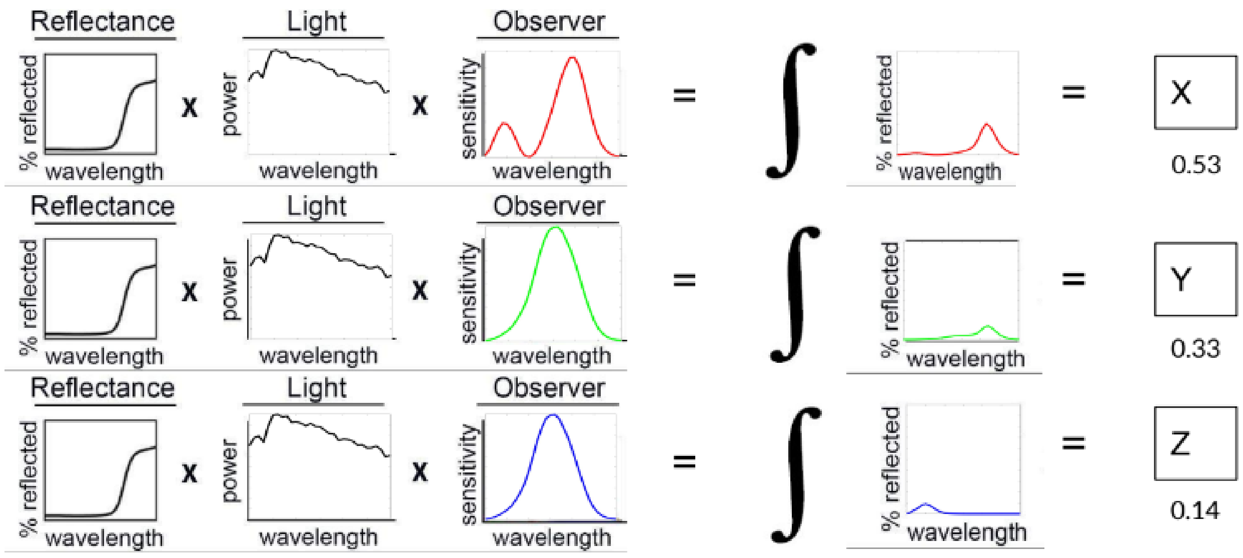
- Use the same visualization function.
- Save the image as:

```
Macbeth_wb.png
```

Step 4: Calculate the XYZ and xy values of the Macbeth ColorChecker and plot on the chromaticity diagram

- Load standard observer curves from the file CIEStandardObserver.csv.
- Adjust the wavelength range to match the previous data.
- Calculate XYZ Values:
  Utilize the getradiance function (from exercise 2) to calculate XYZ values.

Who will be the observer?
Does it remains simply S_c?



XYZ tristimulus values

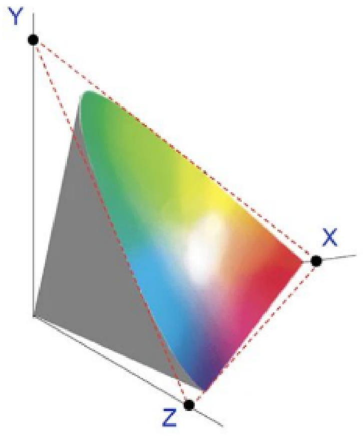
- Calculate xy Values:
  Derive xy from XYZ by dividing each row by the sum of the row.

$$x = \frac{X}{X+Y+Z}$$

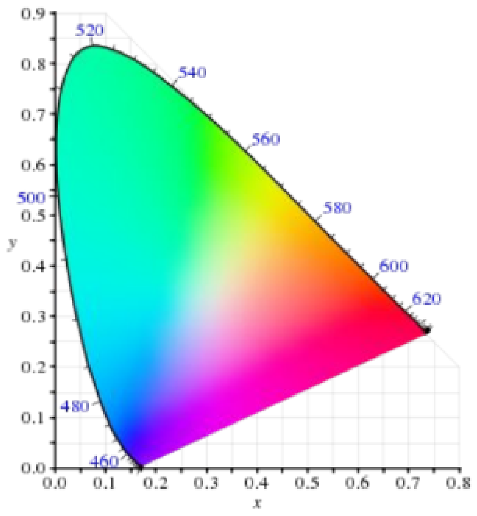
$$y = \frac{Y}{X+Y+Z}$$

$$z = \frac{Z}{X+Y+Z}$$

- Chromaticity Diagram:
  - Use *plotChromaticity* to visualize the data.
  - Represent each color patch with black squares.
  - Optionally, consider coloring squares based on the patch's own color.
- Save the Plot:
  - Save the diagram using the *exportgraphics* function.

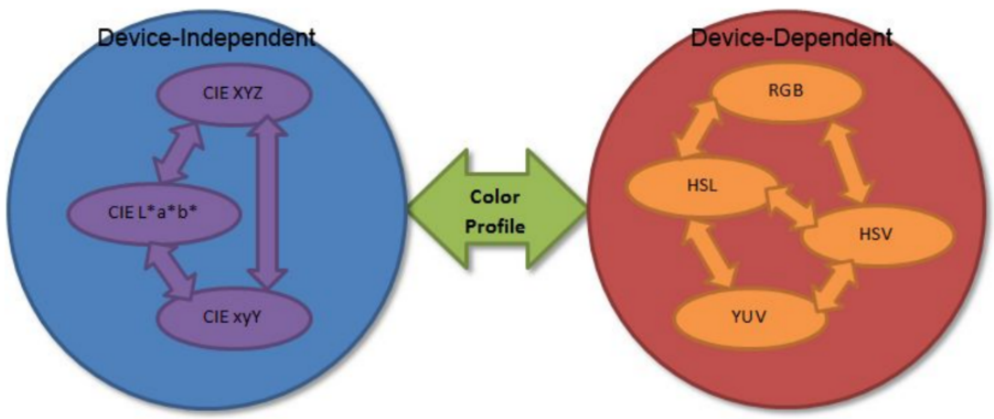


3D XYZ space



2D CIE Chromaticity Diagram

Step 5: Camera RGB to XYZ Transformation



Remember: *Once in XYZ color space, you can transform colors to many other color spaces!*

- Ensure white balance for XYZ.
- Derive a 3x3 transform from white-balanced camera RGB to XYZ.

$$\underbrace{[XYZ]}_{24x3} = M\underbrace{[RGB]'}_{3x24}$$

M is a 3x3 linear transformation matrix that maps colors from the camera space to the standard observer space.

Notes

· M is illumination specific.

· The illumination you used must closely approximate the illumination you will have in the real world.

· M depends on the number and kind of patches selected (i.e. a chart with fewer patches or all gray patches will not result in a good transform)

- Plot on the chromaticity diagram.
- Remember to overlay the other camera too.

Step 6: XYZ to Standard RGB Transformation

- Utilize *standard matrices* for this transformation.
- If XYZ values are already white balanced, skip the CAT.
- Display the ColorChecker visualization.