
Politecnico di Torino

Energy management for IoT
01UDGOV

Master's Degree in Computer Engineering

# Lab 02 - Energy Efficient Displays

## Group 08

Candidates:
Diego Porto (s313169)
Mohamed elsayed Mahmoud elsayed elsisy (s330451)

Referee:
Prof. Massimo Poncino
Matteo Risso

# Contents

# CHAPTER 1

# Assignment 2 - Part one

## 1.1 Evaluation of power consumption

The power consumed to display an image on OLED displays is the sum of the power used to display a single pixel. The power of a pixel depends on the pixel color in terms of RGB components, each channel has a different power consumption that can be determined experimentally. In general, for a single pixel the power consumption is:

$$P_{pixel} = f(R) + h(G) + k(B) \tag{1.1}$$

From this, the power consumption of the image is derived as:

$$P_{image} = C + \sum_{i=1}^{n} (f(R_i) + h(G_i) + k(B_i)) \tag{1.2}$$

As shows in the formula, there is a contribution that is independent of pixel values.

For this lab, the functions that estimate the power consumed to display an image are the following one:

$$P_{pixel} = w_R * R^\gamma + w_G * G^\gamma + w_B * B^\gamma \tag{1.3}$$

$$P_{image} = w_0 + \sum_{i=1}^{n} P_i(R, G, B) \tag{1.4}$$

Where the coefficients are:

| $\gamma$ | $w_0$ | $w_R$ | $W_G$ | $W_B$ |
|---|---|---|---|---|
| 0.7755 | $1.48169521 * 10^{-6}$ | $2.13636845 * 10^{-7}$ | $1.77746705 * 10^{-7}$ | $2.14348309 * 10^{-7}$ |

The Python implementation of the two formulas 1.3 1.4 described before is the following:

```
# Coefficient defines in the slides:
gamma = 0.7755
w_0 = 1.48169521 * 10**(-6)
w_r = 2.13636845 * 10**(-7)
w_g = 1.77746705 * 10**(-7)
w_b = 2.14348309 * 10**(-7)

# First compute the power consumption for each pixel
def compute_power_pixel(r, g, b):
    return w_r * r**gamma + w_g * g**gamma + w_b * b**gamma # From the slides

def compute_power(image_array_rgb):
    """
    Receive an image as input and return the power consumption based on the formula on the slides.
    Works on the R,G,B space of the image
    The image is saved in the variable image_array
    """

    # Split the image into the three channels
    red_channel = image_array_rgb[:, :, 0]
    green_channel = image_array_rgb[:, :, 1]
    blue_channel = image_array_rgb[:, :, 2]

    # Apply the function to all pixels
    sum_pixel_power = 0
    height, width = red_channel.shape   # Get the dimensions of the image,
                                        #assuming all channels have the same dimensions
    for i in range(height):
        for j in range(width):
            power_per_pixel = compute_power_pixel(
            red_channel[i, j],
            green_channel[i, j],
            blue_channel[i, j])
            sum_pixel_power += power_per_pixel

    power_image = w_0 + sum_pixel_power
    return power_image
```

## 1.2 Evaluation of image distorsion

The trasformations applied to the images imply a distortion that can be quantized as the Euclidean distance between the original image and the optimized one.
The function used to compute the distortion (in the LAB color space) is `compute_distortion_lab`. Which takes in input the original and the modified images.

```
# Compute pixel-wise Euclidean distance in Lab space
distortions = np.sqrt(
    np.sum(
    (original_image_lab - modified_image_lab) ** 2, axis=2)
)
```

The provided code snippet computes the distortions between two images, original_image_lab and modified_image_lab, which are represented in the LAB color space. The LAB color space is divided into:

- L: lightness
- a: Red/Green color components
- b: Blue/Yellow color components

The expression `(original_image_lab - modified_image_lab)` computes the element-wise difference between the two images. This operation results in a new array where each element represents the difference in the corresponding color channels (L, a, b) of the two images.
After that, the square of the differences is calculated `** 2`, then the function `np.sum(..., axis=2)` sums these squared differences along the third axis (which corresponds to the color channels). This results in a 2D array where each element represents the sum of squared differences for the corresponding pixel in the original images.
Finally, the square root is computed `np.sqrt(...)`. This step converts the sum of squared differences into the Euclidean distance between the color vectors of the corresponding pixels in the two images.

The resulting distortions array contains the Euclidean distances between the color vectors of corresponding pixels in original_image_lab and modified_image_lab. These distances can be interpreted as the perceptual differences between the pixels in the two images. In order to obtain the total distortion of an image, the sum of the array must be computed as:

```
total_distortion = np.sum(distortions)
```

In our case it's more useful to work with the percentage of the distortion, as written in the code below:

```
w = original_image_lab.shape[0]
h = original_image_lab.shape[1]
max_distortion = math.sqrt(math.pow(100, 2) + math.pow(255, 2) + math.pow(255, 2))
return (total_distortion / (w * h * max_distortion)) * 100
```

**Note:** All the code can be found in the file `lab2.ipynb`.

## 1.3 Image manipulation strategies

### 1.3.1 Image transformation: lower brightness

This is the simplest approach, the brightness of the image is lowered in order to reduced the power consumption. In the code were used different coefficient to reduce the brightness of the final image: from the 90% of the original brightness to 50%. Below the 50% the distortion was too high.

For all the type of transformation the results are analyzed in the next section.

### 1.3.2 OLED displays only

In OLED displays the power consumption depends on color components of a pixel. For this reason, it is possible to save power by modifying the color composition of the image.

#### Image transformation: hungry-blue

It was derived experimentally that the blue channel requires more power to be displayed compared to the red and green channels. Based on this principle, it is possible to tune the blue channel of an image in order to reduce the overall power consumption.
This kind of transformation introduces inevitably a distortion depending on the weight of the blue component on the original image.

#### Image transformation: histogram equalization

The idea behind this technique is that it's possible to reduce the intensity of some pixels in order to "flat" the image. While this can reduce power consumption by lowering overall brightness, it often leads to unnatural looking image.

#### Image transformation: CLAHE

CLAHE (Contrast Limited Adaptive Histogram Equalization) is an advanced variation of histogram equalization. Unlike traditional histogram equalization, which operates on the entire image, CLAHE divides the image into small, localized regions and applies histogram equalization independently to each. To prevent over-amplification of noise, a contrast limiting threshold is enforced, ensuring that the slope of the histogram does not exceed a predefined value.

This localized approach preserves details in homogeneous regions (e.g., skies or shadows) while enhancing contrast in textured areas. For OLED power savings, CLAHE can optimize brightness distribution by reducing the need for uniformly high-intensity pixels, particularly in darker regions.

## 1.4 Analyze power/distortion trade-off

The previous transformations were applied to all the images, the results are plot in 1.1.

The constraints to satisfy are:

- positive power saved (in average)
- lower average distortion

In the analysis, a maximum average distortion of the 4% was choose. Based on those constraints both histogram **equalization** and **brightness_0.8_boost_1.4** are excluded because had negative power saved and despite the lower distortion compared to other transformations they can not be used for our goal.
To save the most amount of power in average the techniques that can be choose are:

- Set the brightness to 60% and boost the non blue pixels with a factor of 1.4 1.2b
- CLAHE 1.2c

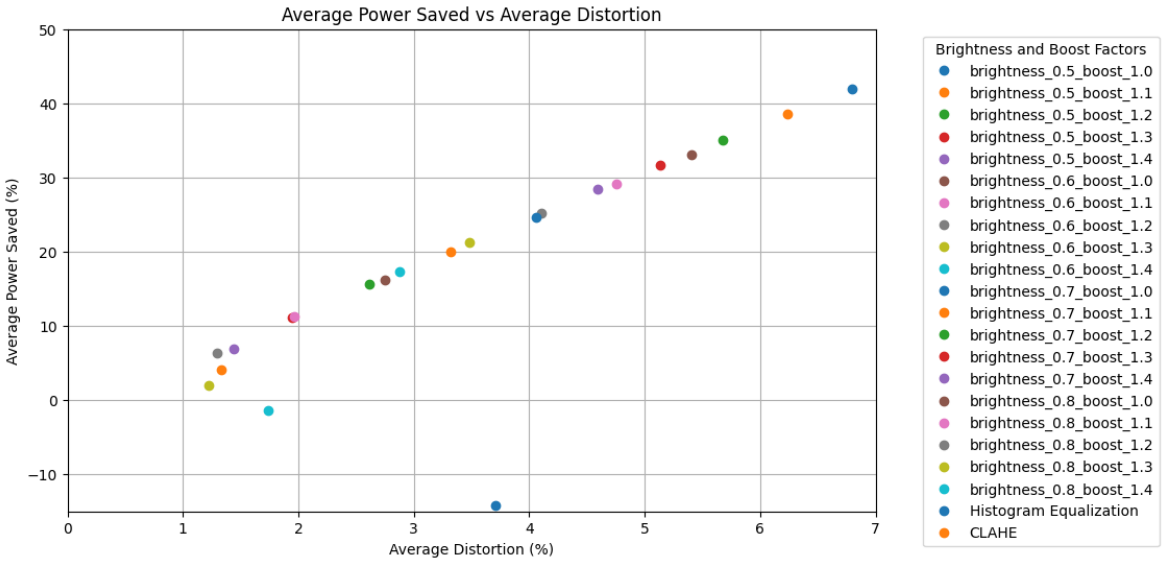


Figure 1.1: Average power saved vs. average distortion

(a) Original (b) BR=0.6 BO=1.4 (c) CLAHE

Figure 1.2: Comparison 27059.jpg

# CHAPTER 2

# Assignment 2 - Part two

In this section, voltage (and current in consequence) is reduces to save power. Some transformations are needed to try to restore some of the quality of the final image. In fact, reducing voltage imply less current through the pixels which cause a color distortion in displayed images. To compensate this distortion it is possible, for example, to work on image luminance.

A possible workflow is:

1. Compute power consumption of the original image
2. Apply DVS and compute both distorsion and power consumption.
3. Modify luminance of the original image and apply again DVS. Compute both distorsion and power consumption.
4. Evaluate the results for the two displayed images.

## 2.1 Evaluation of power consumption

### 2.1.1 New current model

The formula is the following:

$$I_{cell} = \frac{p_1 V_{dd} D_{RGB}}{255} + \frac{p_2 D_{RGB}}{255} + p_3 \tag{2.1}$$

Where:

- $p_1 = +4.251e-5$
- $p_2 = -3.029e-4$
- $p_3 = +3.024e-5$
- Default $Vdd = 15V$

The python implementation of 2.1 used for this part is the following one:

```
def get_pixel_rgb(image: Image, x: int, y: int) -> Tuple[int, int, int]:
    """
    Retrieve the RGB value of a pixel from an image.
    :param image: The image from which to retrieve the pixel value.
```

```
    :param x: The x-coordinate of the pixel.
    :param y: The y-coordinate of the pixel.
    :return: the RGB value of the pixel.
    """
    return image.getpixel((x, y))

def compute_pixel_current(
        image: Image,
        x: int,
        y: int,
        vdd: float = 15,
        p1: float = 4.251e-5,
        p2: float = -3.029e-4,
        p3: float = 3.024e-5
    ) -> Tuple [float, float, float]:

    d_rgb = get_pixel_rgb(image, x, y)
    i_cell = tuple(
        (p1 * vdd * d / 255) + (p2 * d / 255) + p3
        for d in d_rgb
    )

    return i_cell   # tuple of 3 float values in mA
```

### 2.1.2 New power model

The formula to compute the power consumption to display an image is the following:

$$P_{panel} = V_{dd} \sum_{i=1}^{W} \sum_{j=1}^{H} \sum_{R,G,B} I_{cell}(i,j) \tag{2.2}$$

The value obtained with 2.2 is in mW. Its python implementation is:

```
def compute_panel_power(
    image: Image,
    vdd: float = 15,
    ):
    i_panel: float = 0
    for i in range(image.width):
        for j in range(image.height):
            i_cell_ij = compute_pixel_current(image, i, j, vdd)
            for k in range(3):
                i_panel += i_cell_ij[k]

    return vdd * i_panel    # in mW
```

To simulate the effect of DVS on OLED displays the function used is displayed_image(), provided in the slides. It takes as parameter the current drawn by each pixel on the display (in np.array format) and the value of the voltage.
To compute the first parameter, the following function was implemented:

```python
def compute_panel_i(image: Image) -> np.ndarray:
    """
    Compute the current drawn by each pixel of the display.
    :param image: The image displayed on the OLED.
    :return: An array of the currents drawn by each pixel of the display.
    """
    # Initialize the array of currents
    panel_i = np.zeros((image.width, image.height, 3))

    # Iterate over each pixel of the image
    for x in range(image.width):
        for y in range(image.height):
            # Compute the current drawn by the pixel
            panel_i[x, y] = compute_pixel_current(image, x, y)

    return panel_i
```

## 2.2 Application of voltage scaling

The effect to display an image on a OLED display is simulated with the function display_image( I_cell ,V_dd). Before applying the DVS some transformations should be applied to the image to preserve the final quality.

### 2.2.1 Image compensation strategies

Potential strategies to compensate the images before apply the DVS are:

- Brightness scaling
- Contrast enhancement
- Combined BS + CE

In case of a transformation that implies only the contrast enhancement or only the brightness scaling the final results did not meet the quality constraints.
For this reason, the combined BS+CE was used. In order to determine the optimal combination of weights for each parameter to use, the code in **??** was written. Basically, it loops over all the images and apply the transformations defined in the appendix 3.1 and 3.2, saves the result to a new folder called dvs_processed_images_brxx__cntrxxx_voltagexxx based on the values of the parameters. In the meantime the power consumptions and the distortion are computed.

#### Comparison image compensation strategies

The results of the previous implementation are plotted in the chart 2.1. Assuming 4% distortion as an acceptable distortion, the chart shows that the brightness value of 70% achieves the highest power savings (up to 22.65%) at $V_{dd} = 12V$. Using a brightness factor equal to 0.8, a contrast factor equal to 1.9, and Voltage 12.0V (16.26% power saved, 3.20% distortion) is a conservative alternative if image quality is critical. However, it sacrifices significant power savings for minimal distortion reduction 2.2c.
Brightness 0.7, Contrast 1.5, Voltage 12.0V strikes the optimal balance for most images, achieving near-maximal power savings while staying within acceptable quality and distortion limits 2.2b.

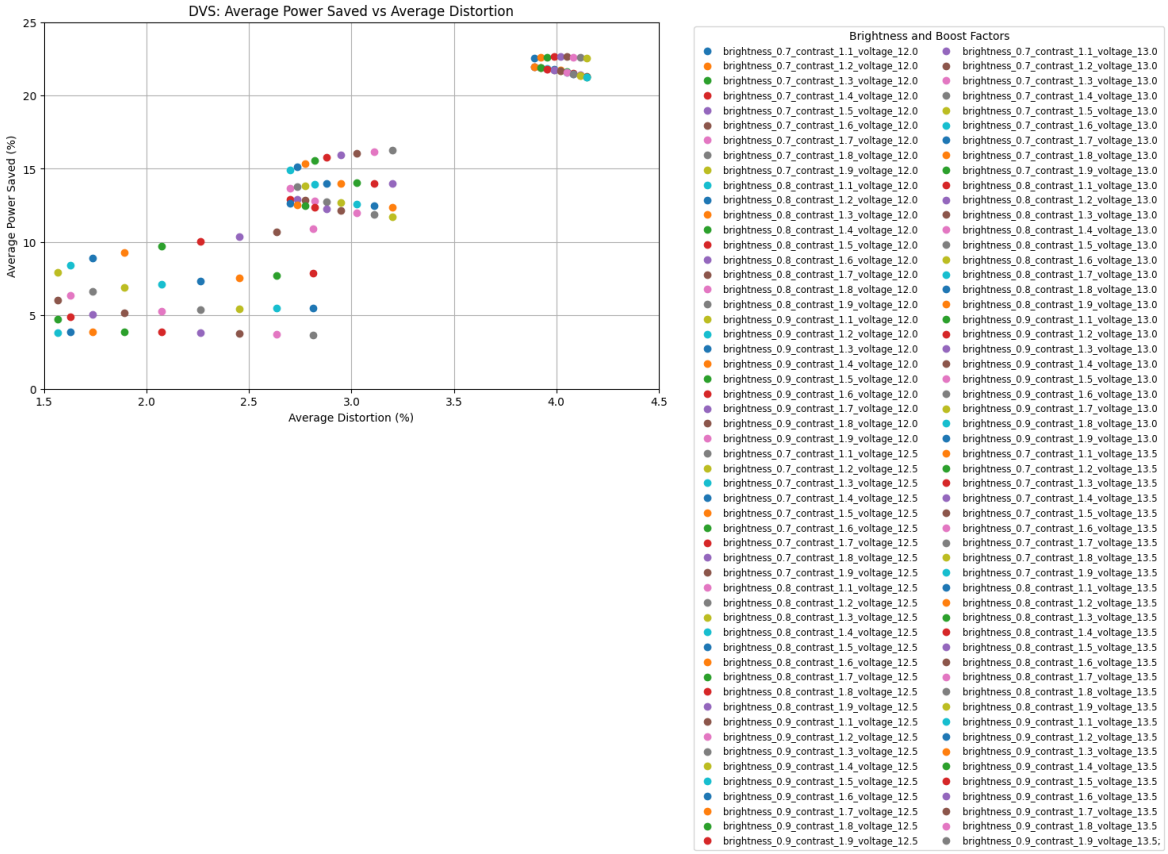


Figure 2.1: Average power saved vs. average distortion with DVS

(a) Original (b) 12V BR=0.7 Contrast=1.5 (c) 12V BR=0.8 Contrast=1.9

Figure 2.2: Comparison 159045.jpg

# CHAPTER 3

# Appendix

## 3.1 Brightness scaling

```
"""
Approach: Increase the brightness of the image
The function tune_brightness receives an image and the brightness_factor.
The brightness_factor is used to boost (or lower) the brightness of the image.
The function returns the modified image
"""

def tune_brightness(img, brightness_factor=1.0):
    """
    Works in LAB space
    """
    img_array = np.array(img)
    if img_array.shape[-1] == 4:  # Check if the image has an alpha channel and remove it.
        img_array = img_array[..., :3]  # This is needed in the case of PNG images (screenshots).

    image_array_lab = rgb2lab(img_array)
    image_array_lab[:, :, 0] = np.clip(
    image_array_lab[:, :, 0] * brightness_factor,
    0, 100)  # Limit the L channel from 0 to 100

    return Image.fromarray((lab2rgb(image_array_lab) * 255).astype(np.uint8))

def test():
    all_images = get_image_paths()
    original_img = Image.open(all_images[0])
    original_img.show()
    o_img = tune_brightness(original_img, 1.3)
    print(compute_power(original_img))
    print(compute_power(o_img))
    o_img.show()

test()
```

## 3.2 Contrast enhancement

```
def increase_contrast(img, contrast_factor=1.0):
    """
    Increase the contrast of an image in the LAB color space.

    Args:
    img: input image.
    contrast_factor (float): The factor by which to increase the contrast.
                             1.0 means no change, less than 1.0 decreases contrast,
                             and greater than 1.0 increases contrast.

    Returns:
    PIL.Image.Image: The image with increased contrast.
    """
    img_array = np.array(img)
    if img_array.shape[-1] == 4:  # Check if the image has an alpha channel and remove it
        img_array = img_array[..., :3]  # This is needed in the case of PNG images (screenshots).

    # Convert the image from RGB to LAB
    image_array_lab = rgb2lab(img_array)

    # Increase the contrast in the L channel
    l_channel = image_array_lab[:, :, 0]
    l_mean = np.mean(l_channel)
    image_array_lab[:, :, 0] = np.clip((l_channel - l_mean) * contrast_factor + l_mean, 0, 100)

    # Convert the modified LAB image back to RGB
    image_array_rgb = lab2rgb(image_array_lab)
    return Image.fromarray((image_array_rgb * 255).astype(np.uint8))


# Test the function
def test():
    all_images = get_image_paths()
    original_img = Image.open(all_images[0])
    original_img.show()
    contrast_img = increase_contrast(original_img, 1.5)
    contrast_img.show()

test()
```