UNIVERSITAS
GADJAH MADA

3D Object Volume Detection

CSB – Numerical Methods – Final Project

Group:
Izaaz Rahman Akbar (21/472855/PA/20348)
Matthew Tan (21/478240/PA/20736)
Rabbani Nur Kumoro (21/472599/PA/20310)
Ramzy Izza Wardhana (21/472698/PA/20322)
William Hilmy Susatyo (21/472585/PA/20308)

Main Objective

By avoiding external libraries (such as numpy, scipy.interpolate, and any edge detection algorithm) to gain deeper understanding of the numerical methods principles. We aim to showcase the effectiveness of these approach and its potential applications.

- Implement **edge detection** techniques using **central difference** in numerical methods.
- Implement **edge interpolation** techniques to bridge the gaps between detected edge points, creating a **continuous and smooth edge** representation
- Implement **volume calculation** using **numerical methods** such as **Simpson ⅓ Integration** and **Voxel Grid**

Main Objective

By avoiding external libraries (such as numpy, scipy.interpolate, and any edge detection algorithm) to gain deeper understanding of the numerical methods principles. We aim to showcase the effectiveness of these approach and its potential applications.

- Implement **edge detection** techniques using **central difference** in numerical methods.
- Implement **edge interpolation** techniques to bridge the gaps between detected edge points, creating a **continuous and smooth edge** representation
- Implement **volume calculation** using **numerical methods** such as **Simpson ⅓ Integration** and **Voxel Grid**

Flowchart Process

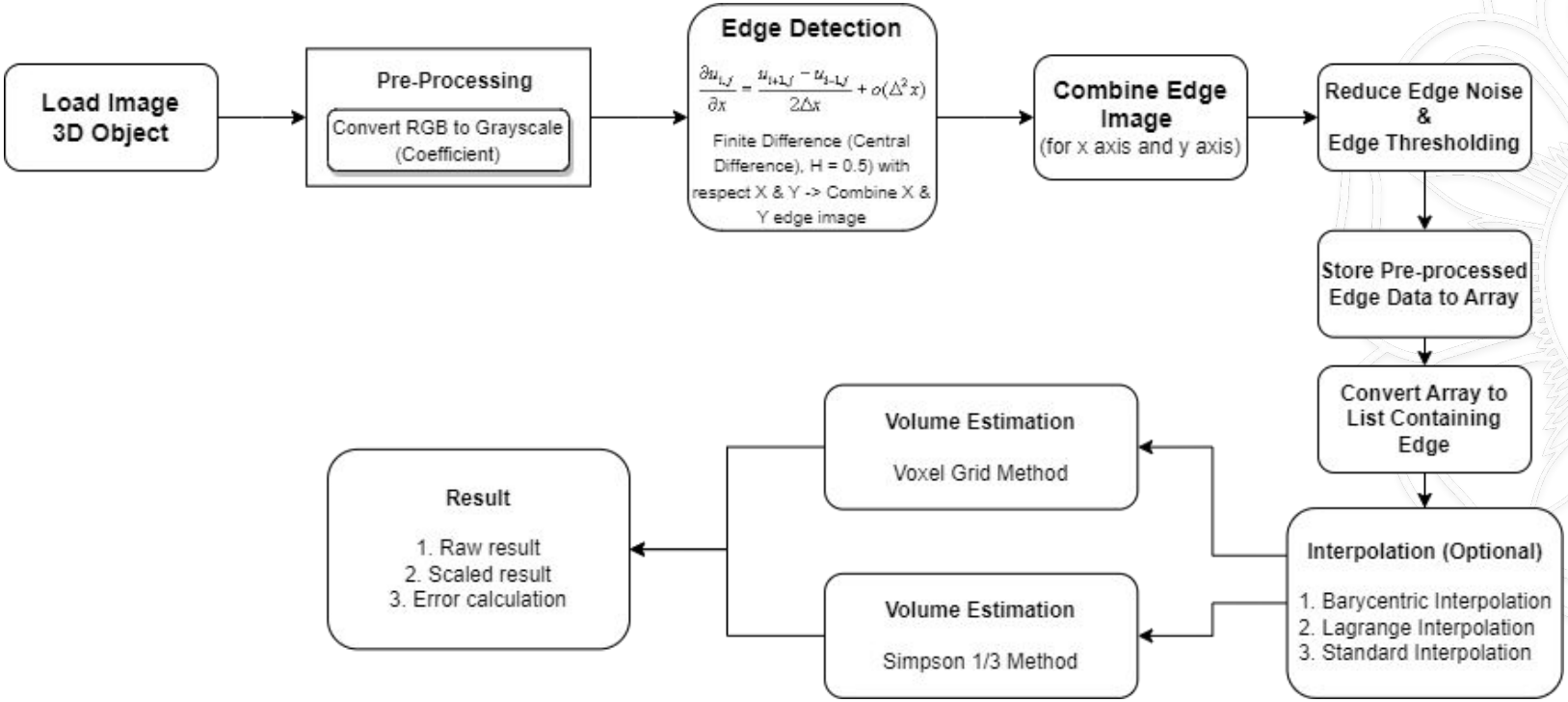


Image Acquisition

Exact Size: 23cm x 22.5cm x 10cm

Exact Volume: 5175cm^3

Shot on 12MP Mobile Phone Camera (3000px X 4000px)

Distance to subject: ± 50cm | Focal Length : ±26mm

Original Image

Cropped Subject

Using Adobe Photoshop CC

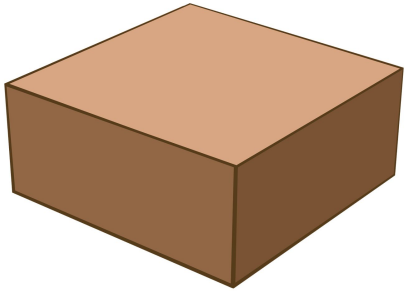
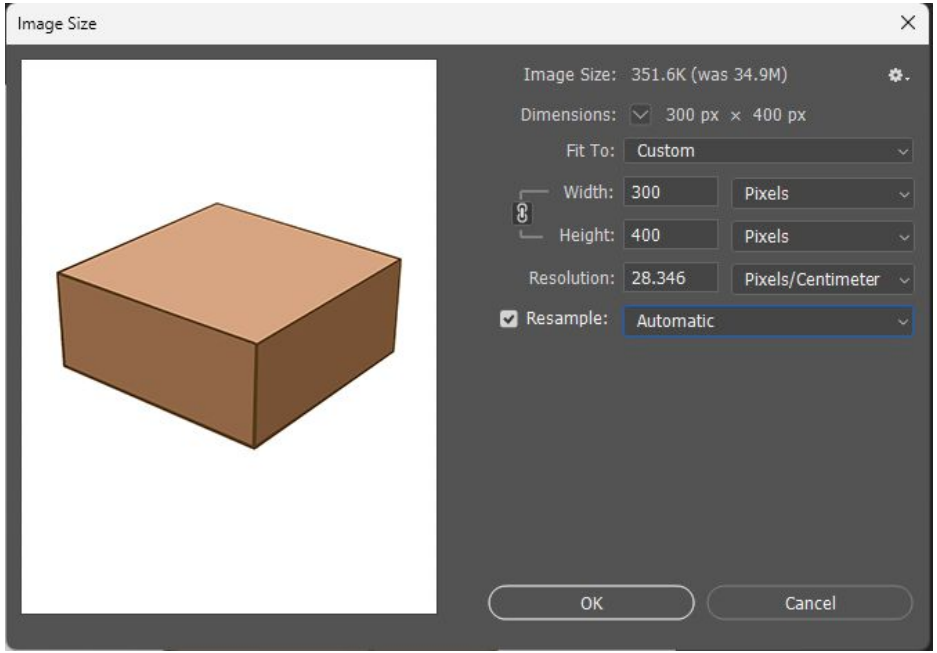


Image Refined

Adjustment on detail, noise, spot, shadow

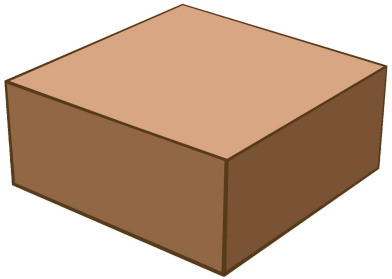
Image Size Adjustment



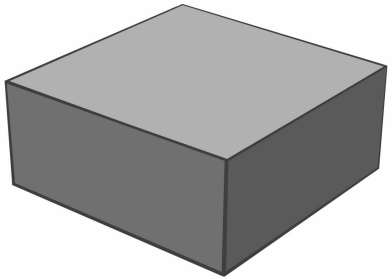
Resolution Size : 300px X 400px
Set the Pixel / Centimeter
= 28.346 (Pixel per centimeter)
≈ 300ppi (Pixel per Inch)

This approach was done to make sure that we can carefully estimate the volume from pixel cube to cm cube and compare it with our exact solution

Preprocessing



Original Image



Grayscale Image

RGB → Grayscale Image

1. Image Initialization: getDimensions function. assigns the width and height of the image to the respective variables.

2. Creating a Grayscale Image through Pixel Conversion: The code utilizes the "L" mode to create a grayscale image and accesses pixel values through iteration.

3. RGB to Grayscale Conversion: Retrieves the RGB values (r, g, b) using the pixel access notation pixels[x, y]. The alpha channel is ignored

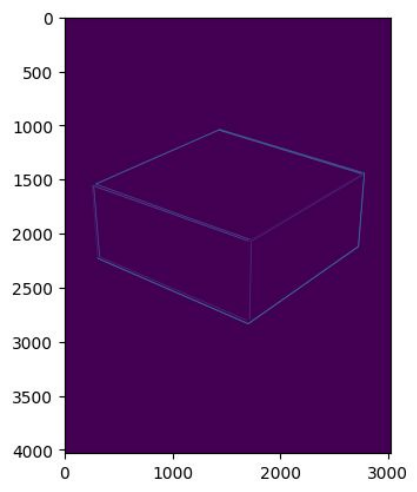
4. Grayscale Calculation: The grayscale value for each pixel is computed using the formula:

0.2989 * r + 0.5870 * g + 0.1140 * b.

These coefficients represent the human eye's sensitivity to different colors in the RGB color space.

Edge Detection using Finite Difference

We decide to use the first derivative instead due to the very high sensitivity that was performed by the second derivative.




```python
from PIL import Image

def getDimensions(image):
    width, height = image.size
    return width, height

def rgbToGreyscale(image):
    width, height = getDimensions(image)
    grey_img = Image.new("L", (width, height))

    pixels = image.load()
    for x in range(width):
        for y in range(height):
            r, g, b, _ = pixels[x, y]
            grey = int(0.2989 * r + 0.5870 * g + 0.1140 * b)
            grey_img.putpixel((x, y), grey)

    return grey_img

def firstDerivative(img):
    width, height = getDimensions(img)
    fx = Image.new("L", (width, height))
    fy = Image.new("L", (width, height))

    img_pixels = img.load()
    fx_pixels = fx.load()
    fy_pixels = fy.load()

    for x in range(1, width - 1):
        for y in range(1, height - 1):
            hx = 0.5
            hy = 0.5
            fx_val = int((img_pixels[x + 1, y] - img_pixels[x - 1, y]) / (2 * hx))
            fy_val = int((img_pixels[x, y + 1] - img_pixels[x, y - 1]) / (2 * hy))
            fx_pixels[x, y] = fx_val
            fy_pixels[x, y] = fy_val

    return fx, fy
```

Combine Edges

```
def combineEdges(fx, fy):
    width, height = getDimensions(fx)
    combined_img = Image.new("L", (width, height))

    fx_pixels = fx.load()
    fy_pixels = fy.load()
    combined_pixels = combined_img.load()

    for x in range(width):
        for y in range(height):
            edge_x = fx_pixels[x, y]
            edge_y = fy_pixels[x, y]
            combined_edge = int((edge_x**2 + edge_y**2)**0.5)
            combined_pixels[x, y] = combined_edge

    return combined_img
```

The process are here as follows:

1. The combineEdges function combines the edge information from the finite difference function in the x and y directions.

2. It creates a new grayscale image with the same dimensions as the derivatives

3. Each pixel in the output image is processed by calculating the magnitude of the combined edge using the Euclidean distance formula.

5. The combined edge value is assigned to the corresponding pixel in the output image and will return as combined_img (edge data of x and y)

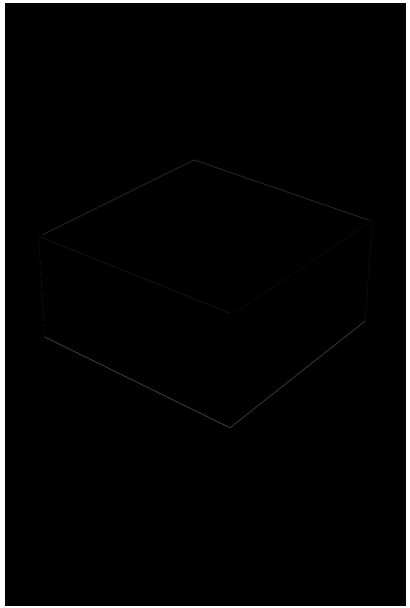
Thresholding & Reducing Noise

Before we calculate the volume, it is important to remove edges noises that we affect the overall accuracy. This can be practically done by observing each neighbor pixel with respect to the x and y axis. If noise detected, then set it to 0 (Black). For consistency for every pixel color, we also perform thresholding to assure that all pixel are colored in white (255).

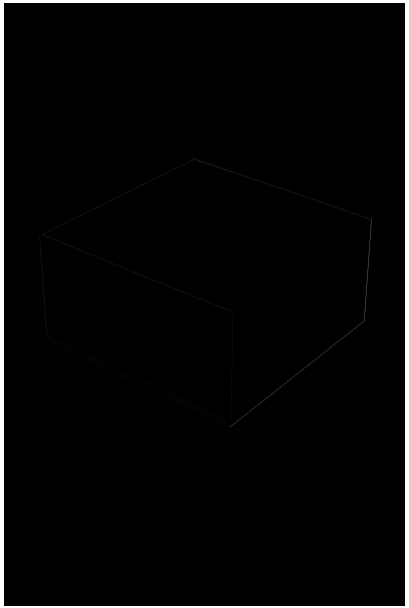
```
def thresholding(array, threshold):
    for i in range (len(array)):
        for j in range (len(array[0])):
            if array[i][j]>threshold:
                array[i][j]=255
    return array

def reduceNoise(array):
    for i in range(1, len(array)):
        for j in range(1, len(array[0])):
            if array[i][j]==255 and ((array[i-1][j]==255 and array[i][j-1]==255 and array[i+1][j]==255)
                                     or (array[i-1][j]==255 and array[i][j-1]==255 and array[i][j+1]==255)
                                     or (array[i-1][j]==255 and array[i+1][j]==255 and array[i][j+1]==255)
                                     or (array[i+1][j]==255 and array[i][j+1]==255 and array[i][j-1]==255)):
                                     array[i][j] = 0
    return array
```

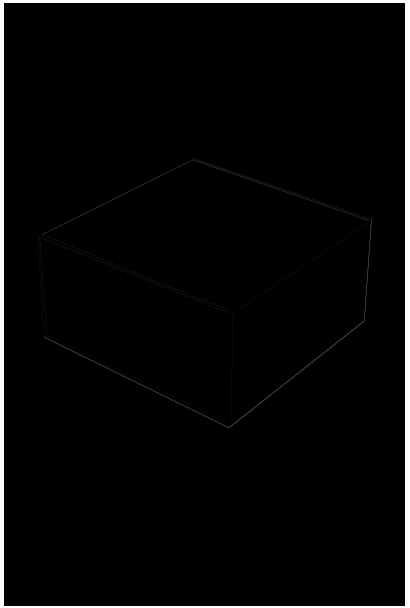
Result of Combine Edges + Thresholding



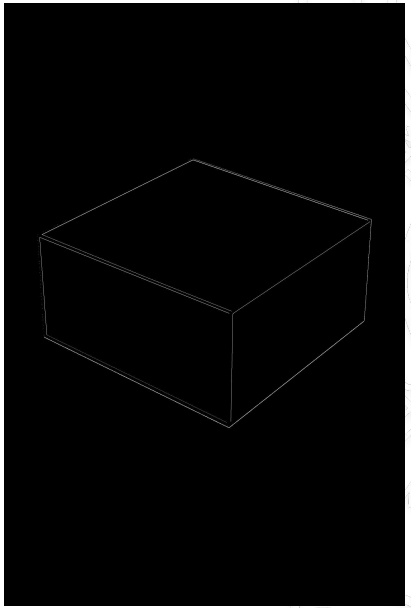
Edge for X



Edge for Y



Combined



Thresholded

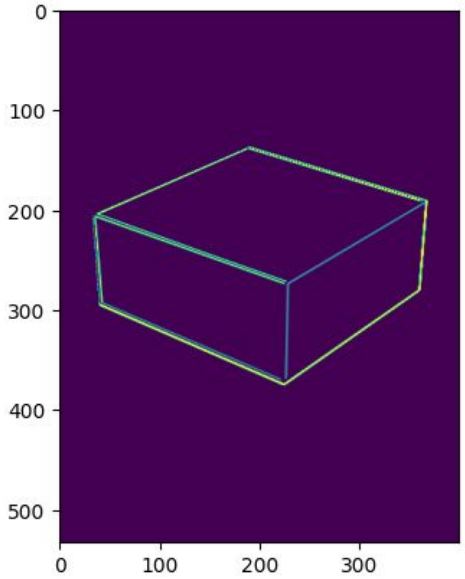
Store Edge Image to Array

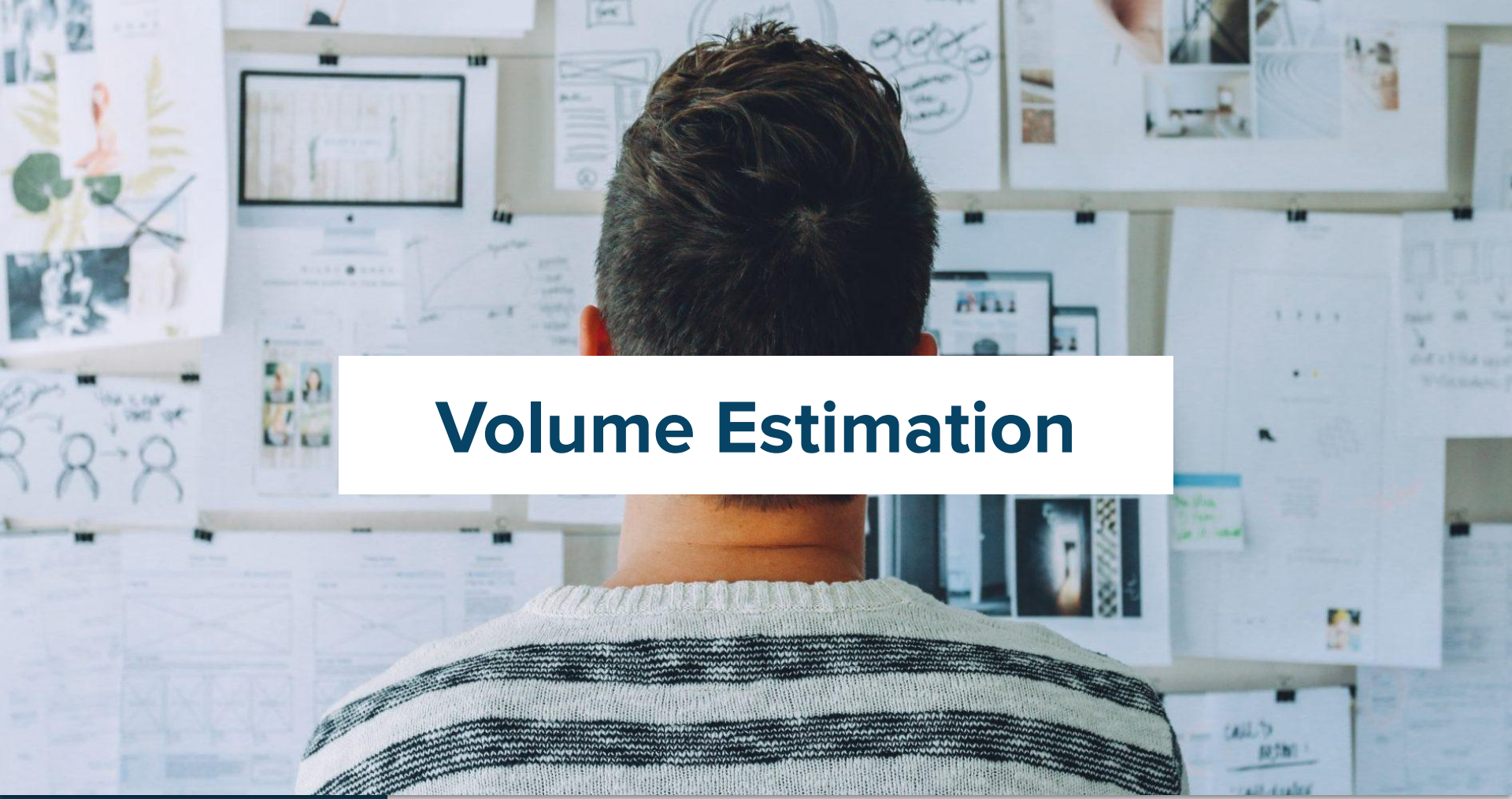
In order to perform more complex mathematical calculations in the upcoming stage, we first need to store all the edge information in an array.

```
def grayscale_to_array(image_path):
    width, height = image_path.size
    pixel_data = list(image_path.getdata())

    array = [pixel_data[i:i+width]
             for i in range(0, len(pixel_data), width)]

    return array
```





Volume Estimation

getPoints

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def append(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node

def getPoints(array):
    linked_list = LinkedList()
    for j in range(len(array[0])):
        for i in range(len(array)):
            if array[i][j] == 255:
                linked_list.append((i, j))
    return linked_list
```

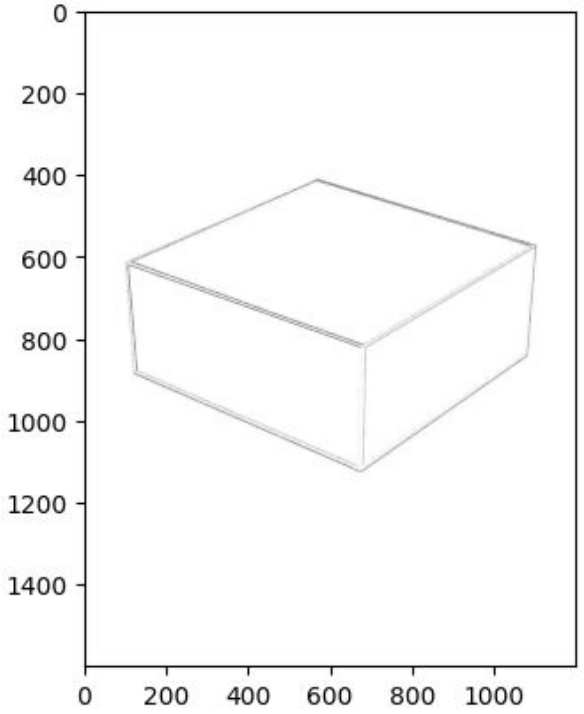
The getPoints function takes an array as input, which is the result of edge detection, thresholding, and noise reduction processes. This Function will return a Linkedlist containing all the edges data.

- It iterates through the elements of the array column-wise and row-wise to examine each pixel.
- If a pixel's value is 255 (indicating an edge pixel), its coordinates [i, j] are appended to a list called lis.
- The function returns the lis list, which contains the coordinates of the detected edge points.

Important Note for Interpolation Steps

In this project, after performing edge detection and doing some pre-processing on both image input and edge data, we observed that the result of **our edge does not necessarily require an interpolation** process. This is due to the fact that our **edge detection process** has **successfully acquired the edge of the object** and there is no unconnected points that may cause discontinuous line.

However, we are still curious and encouraged to experiment with the interpolation process, **although it might not affect significantly** to the result of our edge. Therefore, in the upcoming slides, we will discuss some interpolation process that "maybe" will improve our edge detection.



Lagrange Interpolation

$$L_{n,k}(x) = \prod_{\substack{i=0 \\ i \neq k}}^{n} \frac{x - x_i}{x_k - x_i}$$

$$P_n(x) = \sum_{k=0}^{n} y_k \, L_{n,k}(x)$$

Lagrange Interpolation

```python
def lagrange_interpolation(x, y, x_interpolated):
    n = len(x)
    y_interpolated = []
    for xi in x_interpolated:
        interpolated_value = 0
        for j in range(n):
            basis = 1
            for k in range(n):
                if k != j and x[j] != x[k]:
                    basis *= (xi - x[k]) / (x[j] - x[k])
            interpolated_value += y[j] * basis
        y_interpolated.append(interpolated_value)
    return y_interpolated
```

The lagrange_interpolation function performs Lagrange interpolation to estimate y-values for given x-values.

- It takes three inputs: x (known x-values), y (known y-values), and x_interpolated (x-values for interpolation).
- The function iterates through each x_interpolated value and calculates the corresponding interpolated y-value.
- It uses nested loops to construct basis polynomials and combines them with known y-values to calculate the interpolated y-value for each x_interpolated value. The interpolated y-values are returned as a list.

Barycentric Interpolation

$$P(x) = \frac{\sum_{i=1}^{n} y_i \frac{A_i}{(x-x_i)}}{\sum_{i=1}^{n} \frac{A_i}{(x-x_j)}}$$

$$\text{With } A_i = \frac{1}{\prod_{j=1}^{n}{}_{j\neq i} (x_i - x_j)}$$

Barycentric Interpolation

```python
def barycentric_interpolation(x, y, x_interpolated):
    n = len(x)
    weights = []
    for i in range(n):
        weight = 1
        for j in range(n):
            if i != j and x[i]!=x[j]:
                weight /= (x[i] - x[j])
        weights.append(weight)
    y_interpolated = []
    for xi in x_interpolated:
        numerator_sum = 0
        denominator_sum = 0
        for i in range(n):
            if xi == x[i]:
                interpolated_value = y[i]
                break
            weight = weights[i] / (xi - x[i])
            numerator_sum += weight * y[i]
            denominator_sum += weight
        else:
            interpolated_value = numerator_sum / denominator_sum
        y_interpolated.append(interpolated_value)
    return y interpolated
```

The barycentric_interpolation function performs barycentric interpolation to estimate y-values for given x-values.

- It takes three inputs: x (a list of known x-values), y (a list of known y-values), and x_interpolated (a list of x-values for interpolation).
- The function calculates weights based on the known x-values, which are used to determine the contribution of each known point in the interpolation process.
- Using nested loops, the function iterates through each x-value to be interpolated, calculates the interpolated y-value by summing weighted contributions from the known points, and stores the results in a list of interpolated y-values.

Standard Interpolation

```python
def interpolate_points(edges, num_points):
    interpolated_points = []
    for i in range (1, len(edges)-1):
        start_point = edges[i]
        end_point = edges[i+1]

        # Interpolate the points along the edge
        step_x = (end_point[0] - start_point[0]) / (num_points + 1)
        step_y = (end_point[1] - start_point[1]) / (num_points + 1)

        for i in range(num_points):
            x = start_point[0] + (step_x * (i + 1))
            y = start_point[1] + (step_y * (i + 1))
            interpolated_points.append([x, y])

    return interpolated_points
```

The interpolate_points function is used to add interpolated points along the detected edges when the initial edge detection results are inadequate.

- It takes two inputs: edges (a list of coordinates representing the detected edges) and num_points (the desired number of points to be interpolated between each pair of adjacent edges).
- The function iterates through each pair of adjacent edges and calculates the step sizes in the x and y directions for interpolation based on the number of points.
- Using nested loops, it generates interpolated points by incrementing the x and y coordinates based on the step sizes and appends them to a list. The resulting list contains the additional points interpolated along the edges.

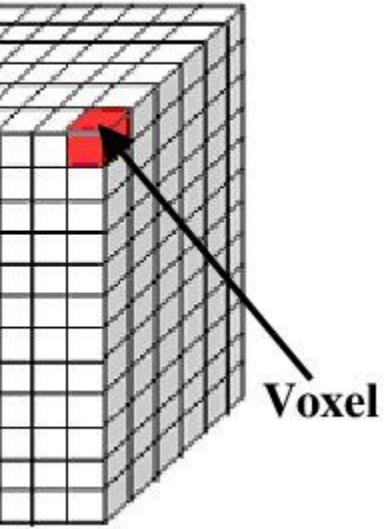
Applying Interpolation

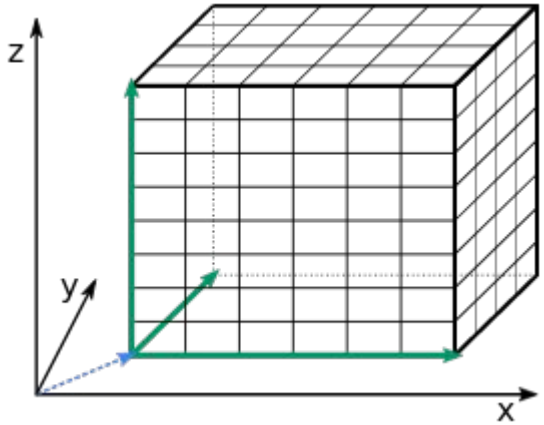
```python
def apply_interpolation(edges, method):
    x_coords, y_coords = zip(*edges)
    x_interpolated = [xi for xi in range(int(min(x_coords)), int(max(x_coords))+1)]
    if method == 'Lagrange':
        y_interpolated = lagrange_interpolation(x_coords, y_coords, x_interpolated)
    elif method == 'Barycentric':
        y_interpolated = barycentric_interpolation(x_coords, y_coords, x_interpolated)
    elif method == 'Standard':
        interpolated_points = interpolate_points(edges, 15)
        return interpolated_points
```

The apply_interpolation function is used to apply interpolation methods to the detected edges when the initial edge detection results are not accurate enough for volume calculation.

- It takes two inputs: edges (a list of coordinates representing the detected edges) and method (a string indicating the interpolation method to be used).
- The function separates the x-coordinates and y-coordinates of the edges into separate lists.
- Depending on the selected method, it either calls the lagrange_interpolation or barycentric_interpolation function or standard to perform the interpolation, using the x-coordinates, y-coordinates, and interpolated x-coordinates.
- The resulting interpolated y-coordinates or additional interpolated points are returned as the output, improving the accuracy of volume calculation when the initial edge detection is insufficient.

Voxel Grid Method





Calculate Volume Using Voxel Grid Method

```python
def compute_volume_voxelpt2(edges):
    # Determine the minimum and maximum coordinates of the image
    min_y = min(edges, key=lambda edge: edge[0])[0]
    max_y = max(edges, key=lambda edge: edge[0])[0]
    min_x = min(edges, key=lambda edge: edge[1])[1]
    max_x = max(edges, key=lambda edge: edge[1])[1]

    # print(min_x, max_x, min_y, max_y)

    tuning = 3750
    # Initialize the volume and voxel size
    volume = 0
    voxel_size = 2  # Assuming each voxel represents a unit square
```

The compute_volume_voxelpt2 function calculates the volume of an object based on its detected edges using the Voxel Grid Method.

- It determines the minimum and maximum coordinates of the image by finding the minimum and maximum values of the x and y coordinates from the detected edges.
- The function iterates over each voxel within the bounding box defined by the minimum and maximum coordinates.

Calculate Volume Using Voxel Grid Method

```
# Iterate over each voxel and check if it intersects with the image edges
for x in range(min_x, max_x + 1):
    for y in range(min_y, max_y + 1):
        # Check if the voxel intersects with any edge
        intersect = False
        for i in range(len(edges)):
            j = (i + 1) % len(edges)
            if (edges[i][1] > y) != (edges[j][1] > y) and x < (
                    edges[j][0] - edges[i][0]) * (y - edges[i][1]) / (
                    edges[j][1] - edges[i][1]) + edges[i][0]:
                intersect = not intersect

        # If the voxel intersects with an odd number of edges, it is part of the image
        if intersect:
            volume += voxel_size * tuning

return volume
```

- It determines the minimum and maximum coordinates of the image by finding the minimum and maximum values of the x and y coordinates from the detected edges.
- For each voxel, it checks if it intersects with any of the edges by comparing the y-coordinate of the voxel with the y-coordinates of the adjacent edges and calculating the x-coordinate of the intersection point.
- If a voxel intersects with an odd number of edges, it is considered part of the object, and the volume is incremented by the product of the voxel size and a tuning factor.
- The function returns the calculated volume, providing an estimate of the object's volume based on the Voxel Grid Method.

Simpson ⅓ Method

$$\int_a^b f(x)dx \approx \frac{\Delta x}{3}\left(f(x_0) + 4f(x_1) + 2f(x_2) + \ldots + 2f(x_{n-2}) + 4f(x_{n-1}) + f(x_n)\right)$$

Where, $\Delta x = \frac{b-a}{n}$

$x_0 = a$ and $x_n = b$

$x_0, x_0, \ldots, x_n$ are the ends of the n sub intervals

Simpson Integration

```python
def f(x):
    return 1

def simpson_integration(a, b, n):
    h = (b - a) / n
    x = a
    sum1 = 0
    sum2 = 0
    for i in range(1, n, 2):
        x += h
        sum1 += f(x)
        x = a + h
        for i in range(2, n, 2):
            x += h
            sum2 += f(x)
        integral = h / 3 * (f(a) + f(b) + 4 * sum1 + 2 * sum2)
        return integral
```

simpson_integration(a, b, n) function performs Simpson's 1/3 integration method to approximate the definite integral of a function over an interval [a, b].

- It takes three inputs: a (lower bound of integration), b (upper bound of integration), and n (number of intervals).
- The function calculates the interval width h by dividing the range (b - a) by the number of intervals n.
- It iteratively calculates the sums of function values at odd and even indices using loops, and then applies the Simpson's 1/3 rule formula to obtain the integral.

Calculate Volume Using Simpson Method

```
def compute_volume_simpson(edges, n):
    volume = 0
    for i in range(len(edges) - 1):
        x1, y1 = edges[i]
        x2, y2 = edges[i + 1]
        area = (y1 + y2) / 2 * (x2 - x1)
        integral = simpson_integration(x1, x2, n)
        volume += area * integral
    return volume
```

The compute_volume_simpson(edges, n) function takes as input the detected edges (represented as a list of coordinates) and the number of intervals for Simpson's 1/3 integration.

- It initializes the volume variable to 0, which will store the calculated volume. The function iterates through each pair of adjacent edges, assuming the edges are sorted in order.
- For each pair of edges, it calculates the area of the trapezoid formed by the edges and extracts the x and y coordinates. It then calls the simpson_integration function to calculate the integral of the function f(x) over the interval defined by the x-coordinates of the edges, using n intervals.
- The integral value is multiplied by the area of the trapezoid and added to the volume. After iterating through all the edge pairs, the function returns the calculated volume.

Implementation

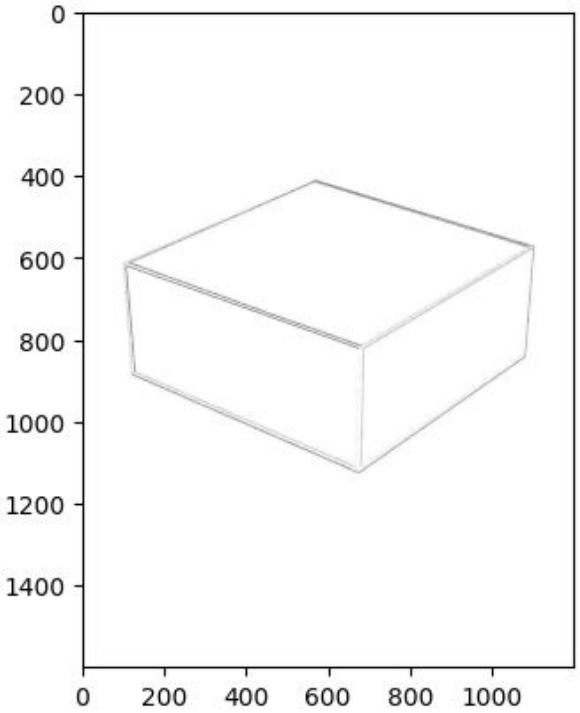
```
image = Image.open('box.jpg')
grey_img = rgbToGreyscale(image)
fx, fy = firstDerivative(grey_img)
combined_edges = combineEdges(fx, fy)
array = grayscale_to_array(combined_edges)
array = thresholding(array, 255)
array = reduceNoise(array)
points = getPoints(array)

# print(points)
# Object Volume Detection with Simpson Method
volume = compute_volume_simpson(points, 30)
print("Raw Volume of an Object using Simpson Method:", volume)
volume_estimation = volume / (28.346 ** 3)
print("Volume of an Object after Scaling:", volume_estimation)
error = (volume_estimation - 5175) / 5175
print("Error:", error * 100)

print("----------------------------------------------------------------")

# Object Volume Detection with Voxel Grid Method
volume3 = compute_volume_voxelpt2(points)
print("Raw Volume of an Object using Voxel Grid:", volume3)
volume3_estimation = volume3 / (28.346 ** 3)
print("Volume of an Object after Scaling:", volume3_estimation)
error = (volume3_estimation - 5175) / 5175
print("Error:", error * 100)
```

Result (Edge)



Result (Volume)

```
Raw Volume of an Object using Simpson Method: 343392417.0555565
Volume of an Object after Scaling: 15077.014492209133
Error: 191.34327521177067
-----------------------------------------------------------------
Raw Volume of an Object using Voxel Grid: 127852500
Volume of an Object after Scaling: 5613.501928475322
Error: 8.473467216914427
```

Side Notes:

Exact Solution = 23cm x 22.5cm x 10xm

Exact Volume: 5175cm^3

Converted using Pixel per CM (Obtained from the previous slide)

Estimated Solution : Exact Volume: 5613cm^3

Demonstration

bit.ly/NumericalMethodFinalProject

Results & Evaluation

1. Volume Estimation could be performed by using **Simpson Method** and **Voxel Grid Method**, each has its own **drawbacks**

2. **Simpson Method** is recommended to use for the object that has **arbitrary** height while **Voxel Grid Method** is preferred for the object that has **constant** height
3. In order to achieve greater accuracy, we need to **trial and error**, especially on handling the tuning (until it converge to particular value) when computing the volume, as well as removing the noise.

We Appreciate Your Attention 🙏