# LAPORAN SEGMENTASI CITRA

# CITRA DIGITAL


**Ditulis oleh:**

**Renaldi Septian**

**NIM. 226201029**

**D3 TEKNIK KOMPUTER**

**TEKNOLOGI INFORMASI**

**POLITEKNIK NEGERI SAMARINDA**


**2024**

# BAB 1
# LANDASAN TEORI

## Segmentasi Citra

Segmentasi citra digital adalah proses esensial dalam pengolahan citra yang bertujuan memisahkan antara objek dan latar belakang. Salah satu metode yang sering digunakan dalam segmentasi citra grayscale adalah *thresholding* dengan Metode Otsu. Metode ini bertujuan untuk mencari nilai ambang secara otomatis berdasarkan histogram grayscale citra. Otsu thresholding dikenal cukup akurat karena kemampuannya dalam membedakan objek yang tersegmentasi dari latar belakang secara optimal.

Proses segmentasi dengan metode ini diawali dengan penghitungan nilai ambang terbaik dari citra grayscale. Namun, sebelum segmentasi dilakukan, penting untuk melakukan beberapa tahap pra-pemrosesan, seperti *invert image* dan *noise removal*. Hal ini diperlukan untuk menghilangkan area yang bukan objek sehingga hanya objek yang relevan saja yang diolah.

Setelah itu, langkah perbaikan citra dilakukan melalui operasi *morphology* yang bertujuan memperbaiki citra biner dengan cara menggabungkan titik-titik latar di dalam objek agar menjadi bagian dari objek. Salah satu teknik yang sering digunakan dalam proses ini adalah *filling holes*, yang berfungsi untuk mengisi area lubang dalam objek yang dikelilingi oleh piksel latar depan.

Setelah perbaikan citra selesai, langkah selanjutnya adalah menghitung tingkat akurasi dari hasil segmentasi. Untuk mendapatkan akurasi ini, digunakan metode pelabelan objek dengan *region properties* atau *regionprops*. Akhirnya, akurasi segmen dihitung dengan membandingkan hasil segmentasi terhadap *Ground Truth*, yang merupakan hasil pengamatan langsung oleh pengguna.

## Algortima K-Means

Metode clustering partisional yang digunakan untuk mengelompokkan data berdasarkan kemiripan. Proses dimulai dengan menentukan jumlah cluster yang diinginkan (K) dan memilih centroid awal secara acak. Algoritma ini kemudian mengiterasi untuk menyesuaikan posisi centroid dengan menghitung jarak antara setiap objek dalam dataset dan centroid menggunakan rumus Euclidean Distance. Objek yang memiliki jarak terdekat dengan centroid tertentu akan dimasukkan ke dalam cluster tersebut (Bangoria et al., 2018).

Proses ini berlanjut hingga tidak ada perubahan lagi pada keanggotaan cluster, artinya setiap objek telah dikelompokkan dengan stabil. K-Means berfokus pada kepadatan data, sehingga pemilihan centroid awal yang strategis dapat meningkatkan kinerja algoritma. Hasil akhirnya adalah pengelompokan objek ke dalam cluster berdasarkan jarak terdekat ke centroid, yang menghasilkan segmentasi citra yang efektif.

Alasan digunakannya K Means Clustering adalah karena data harus dikelompokkan untuk membantu proses klasifikasi (Agrawal & Gupta, 2018). Algoritma untuk K Means Clustering adalah :

1. Pilih k buah titik centroid secara acak.
2. Kelompokkan data sehingga terbetuk K buah cluster dengan titik centroid dari setiap centroid dari setiap cluster merupakan titik centroid yang telah dipilih sebelumnya.
3. Perbaharui nilai titik centroid.
4. Ulangi langkah 2 dan 3 sampai nilai dari titik centroid tidak lagi berubah.

## Metode Sobel

Metode Sobel menggunakan dua kernel konvolusi berukuran 3x3 untuk menghitung gradien horizontal dan vertikal dari citra. Kernel Sobel dirancang dengan nilai-nilai yang berbeda untuk mendeteksi kedua jenis gradien ini. Untuk mendeteksi gradien horizontal, kernel Sobel memiliki nilai positif di bagian atas, nilai nol di tengah, dan nilai negatif di bagian bawah. Sebaliknya, untuk mendeteksi gradien vertikal, kernel memiliki nilai positif di sebelah kiri, nilai nol di tengah, dan nilai negatif di sebelah kanan. Ketika kedua kernel ini diterapkan pada citra, mereka melakukan operasi konvolusi di seluruh gambar, menghitung setiap nilai piksel baru berdasarkan piksel sekitarnya.

Setelah gradien horizontal dan vertikal dihitung, langkah selanjutnya adalah menggabungkan kedua gradien tersebut untuk mendapatkan magnitude tepi. Ini dilakukan dengan menghitung nilai akar kuadrat dari penjumlahan kuadrat gradien horizontal dan vertikal di setiap piksel. Hasilnya adalah citra yang menunjukkan kekuatan atau magnitude dari tepi di setiap piksel; semakin besar nilai magnitude, semakin tajam tepi yang terdeteksi pada titik tersebut.

Penerapan algoritma Sobel sangat membantu dalam mengidentifikasi perubahan tajam dalam intensitas citra, yang sering kali menandakan batas objek atau fitur dalam citra. Metode ini merupakan langkah penting dalam analisis citra untuk segmentasi objek, ekstraksi fitur, dan banyak aplikasi lainnya dalam pengolahan citra. Dengan menggunakan dua kernel konvolusi yang berbeda untuk mendeteksi gradien horizontal dan vertikal, serta menggabungkan hasilnya untuk mendapatkan magnitude tepi, algoritma Sobel memberikan metode yang efektif dan terpercaya untuk mendeteksi tepi dalam citra digital.

## Metode Prewitt

Operator Prewitt merupakan pengembangan metode Robert dengan menggunakan filter HPF yang diberi satu angka nol penyangga. Metode ini mengambil prinsip dari fungsi laplacian yang dikenal sebagai fungsi untuk membangkitkan HPF, untuk mempercepat komputasi bagian yang bernilai nol tidak perlu diproses. Persamaan gradient pada operator Prewitt sama dengan operator sobel, tetapi menggunakan nilai C=1.

Metode prewitt ini menekankan pembobotan pada piksel-piksel yang lebih dekat. Metode prewitt hampir sama dengan metode sobel dimana metode sobel ini menggunakan HPF yang nilai konstantanya adalah Nol. Dalam penelitian ini ada beberapa objek yang akan di uji dengan metode prewitt yang kemudian hasilnya dibandingkan dengan metode cany, dimana hasil dari kedua metode tersebut nantinya akan dibandingkan untuk mencari nilai terbaik. Penggabungan ke dua metode ini hanya membandingkan nilai pixel setiap objek.

Cara kerja dari Algoritma Prewitt dengan cara menghitung hasil maksimum dari suatu kernel konvolusi untuk menemukan orientasi deteksi sisi disekitarnya pada tiap pixel. Algoritma ini juga disebut sebagai edge template matching, karena gambar mengalami proses matching dengan sebuah template yang berupa sisi.

Semakin besar tingkat kerapatan objek akan menentukan resolusi dari komponen citra yang dimiliki sebuah objek. Pixel akan sangat mempengaruhi proses kuantitasi semakin besar pixel maka semakin besar variasi nilai warnanya yang dihasilkan dan semakin mirip dengan warna objeks aslinya. Algoritma metode Prewitt dalam mendeteksi tepi citra digital adalah:

1) Citra masukan berupa grayscale.
2) Konvolusikan citra grayscale dengan kernel Prewitt arah horisontal dan vertikal.
3) Hitung besar gradient dengan persamaan.
4) Citra keluaran merupakan hasil dari besar gradient (G).

## Metode Robert

Metode Robert merupakan dua filter berukuran 2x2. Ukuran filter yang kecil menjadi kelebihan dari metode ini kerena membuat komputasi terjadi sangat cepat. Kelemahan dari metode ini yakni karena ukurannya yang kecil sehingga sangat terpengaruh oleh derau. Selain itu metode Robert memberikan tanggapan yang lemah terhadap tepi, kecuali jika tepi tersebut sangat tajam. Perhitungan gradient metode Robert dihitung dengan persamaan.

Algoritma metode Robert dalam mendeteksi tepi citra digital adalah:

1) Citra masukan berupa grayscale.
2) Konvolusikan citra grayscale dengan kernel Robert arah horizontal dan vertikal.
3) Hitung besar gradient dengan persamaan.
4) Citra keluaran merupakan hasil dari besar gradient (G).
5) Citra masukan berupa citra grayscale berukuran 5x5 piksel, untuk mempermudah dan menyederhanakan perhitungan.

## Metode Laplacian of Gaossian

Operator Laplacian of Gaussian merupakan kombinasi dari operator gaussian dan operator laplacian. Deteksi tepi orde kedua yang makin kurang sensitif terhadap derau adalah Laplacian of Gaussian (LoG). Hal ini disebabkan penggunaan fungsi Gaussian yang memuluskan citra dan berdampak terhadap pengurangan derau pada citra. Akibatnya, operator mereduksi jumlah tepi yang salah terdeteksi.

Cara kerja operator LoG adalah citra dikonvolusi dengan operator gaussian bertujuan untuk mengaburkan dan memperlemah noise (derau). Namun, pengaburan ini mengakitbatkan pelebaran tepi objek. Kemudian, operator laplacian diterapkan untuk menemukan titik potong dengan sumbu x dalam fungsi turunan kedua yang bersesuaian dengan puncak dalam fungsi turunan pertama. Kemudian, lokasi tepi diperoleh dari resolusi subpiksel menggunakan interpolasi linier. Metode Laplacian of Gauss (LoG) dilakukan dengan menentukan bagian tepi citra menggunakan orde turunan kedua. Operator laplacian menghasilkan kepekaan terhadap noise pada tiap bagian piksel.

# Metode Zerro Crossing

Zero Crossing adalah metode dalam pengolahan citra digital yang digunakan terutama untuk deteksi tepi (edge detection). Metode ini bekerja dengan mencari perubahan tanda dari positif ke negatif atau sebaliknya pada hasil turunan kedua dari intensitas gambar. Ketika turunan kedua dari gambar berubah tanda, nilai pada titik tersebut disebut "zero crossing." Teknik ini sangat berguna untuk mendeteksi tepi halus atau perubahan yang tidak terlalu kontras pada gambar.

Langkah-Langkah Penerapan Algoritma Zero Crossing:

1. Konvolusi dengan Filter Laplacian:

   Langkah pertama dalam penerapan zero crossing adalah mengonvolusikan gambar dengan operator Laplacian, atau operator serupa yang merupakan turunan kedua dari intensitas gambar. Operator ini mendeteksi perubahan intensitas dengan mengidentifikasi daerah di mana terdapat perubahan signifikan dari terang ke gelap.

   Beberapa varian operator yang sering digunakan adalah Laplacian of Gaussian (LoG) atau Difference of Gaussian (DoG), yang membantu meredam noise sebelum menghitung turunan kedua.

2. Mendeteksi Titik Zero Crossing:

   Setelah turunan kedua dari citra diperoleh, metode zero crossing mendeteksi tempat di mana nilai berubah dari positif ke negatif atau sebaliknya. Ini menunjukkan adanya perubahan intensitas pada gambar, yang merupakan indikasi adanya tepi (edge).

3. Pengelolaan Noise:

   Salah satu tantangan terbesar dalam penerapan metode zero crossing adalah sensitivitas terhadap noise. Oleh karena itu, sering kali citra perlu terlebih dahulu dihaluskan menggunakan filter seperti Gaussian untuk meredam noise yang dapat menghasilkan false positive dalam deteksi tepi.

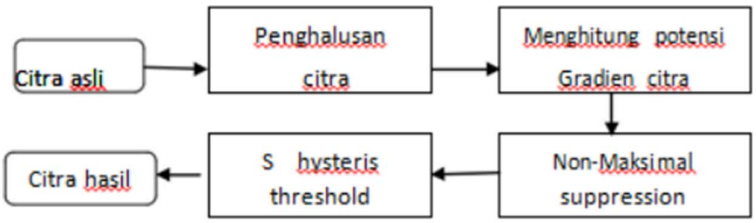
## Metode Canny

Deteksi tepi Canny diciptakan oleh John F. Canny pada tahun 1986. Algoritma ini dibuat untuk memenuhi tiga kriteria utama: deteksi tepi yang baik, penempatan yang tepat, dan respons minimal. Beberapa langkah utama diambil dalam proses ini: penggunaan filter Gaussian untuk menghaluskan gambar, menghitung gradien intensitas, penghapusan non-maksimal untuk menghilangkan piksel yang bukan merupakan bagian dari tepi, dan penggunaan ambang hysteresis untuk mengidentifikasi dan menghubungkan tepi yang kuat dan lemah. Keunggulan utama metode ini adalah kemampuan untuk mendeteksi tepi dengan presisi tinggi bahkan pada gambar yang mengandung noise.(Krishna, 2023)

Algoritma deteksi tepi Canny mengikuti beberapa kriteria sebagai berikut :

a. Good detection. Kriteria ini bertujuan memaksimalkan nilai signal to noise ratio (SNR) sehingga semua tepi dapat terdeteksi dengan baik atau tidak ada yang hilang.
b. Good localization. Tepi yang terdeteksi berada pada posisi yang sebenarnya, atau dengan kata lain behwa jarak antara posisi sebenarnya adalah seminimum mugkin (idealnya adalah 0)
c. Only one response to a single. (hanya satu respon untuk sebuah tepi). Artinya detektor tidak memberikan tepi yang bukan tepi sebenarnya.

Urutan proses pendeteksian tepi menggunakan metode canny ditunjukkan pada gambar sebagai berikut:




*Gambar 1 Blog diagram pendeteksian tepi canny.*

# BAB 2
# INPUT PROGRAM

Implementasi Source Code

Segmentasi Citra

```
import cv2
import numpy as np

threshold = 0.5
kernel5 = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (20, 20))
x_co = 0
y_co = 0
hsv = None
H = 125
S = 220
V = 170
thr_H = 180 * threshold
thr_S = 255 * threshold
thr_V = 255 * threshold

# Mouse callback function to capture color when clicking on the image
def on_mouse(event, x, y, flag, param):
    global x_co, y_co, H, S, V, hsv
    if event == cv2.EVENT_LBUTTONDOWN:
        x_co = x
        y_co = y
        p_sel = hsv[y_co][x_co]
        H = p_sel[0]
        S = p_sel[1]
        V = p_sel[2]

cv2.namedWindow("image", 1)
cv2.namedWindow("mask", 2)
cv2.namedWindow("segmented", 3)

# Load image from file
image_path = 'semangka.jpeg'  # Change this to the path of your image
src = cv2.imread(image_path)

# Check if the image was loaded successfully
if src is None:
    print("Failed to load image.")
    exit()

# Apply blur to smooth the image
src = cv2.blur(src, (3, 3))

# Convert to HSV color space
hsv = cv2.cvtColor(src, cv2.COLOR_BGR2HSV)

# Set the mouse callback for color selection
```

```
cv2.setMouseCallback("image", on_mouse, 0)

while True:
    # Define color threshold range
    min_color = np.array([H - thr_H, S - thr_S, V - thr_V])
    max_color = np.array([H + thr_H, S + thr_S, V + thr_V])
    mask = cv2.inRange(hsv, min_color, max_color)

    # Morphological transformation to close small holes
    mask = cv2.morphologyEx(mask, cv2.MORPH_CLOSE, kernel5)

    # Show the HSV values of the selected point
   cv2.putText(mask, f"H:{H} S:{S} V:{V}", (10, 30), cv2.FONT_HERSHEY_PLAIN,
2.0, (255, 255, 255), thickness=1)

    # Display mask and original image
    cv2.imshow("mask", mask)
    cv2.imshow("image", src)

    # Segment the selected color region
    src_segmented = cv2.add(src, src, mask=mask)
    cv2.imshow("segmented", src_segmented)

    # Break the loop when the 'ESC' key is pressed
    if cv2.waitKey(10) == 27:
        break

# Close all windows
cv2.destroyAllWindows()
```

K-Means

```
import cv2
import numpy as np
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt

# Load the image
image = cv2.imread('686027dc0db04ae6272ac72b6a7ed195.jpg')
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

# Reshape the image to a 2D array of pixels
pixels = image.reshape(-1, 3)

# Number of clusters (K)
k = 5

# Apply K-means clustering
kmeans = KMeans(n_clusters=k)
kmeans.fit(pixels)

# Get the cluster centers (dominant colors)
colors = kmeans.cluster_centers_.astype(int)

# Assign each pixel to the nearest cluster center
```

```
labels = kmeans.labels_
segmented_image = colors[labels].reshape(image.shape)

# Display the original and segmented images
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.title('Original Image')
plt.imshow(image)
plt.axis('off')

plt.subplot(1, 2, 2)
plt.title('Segmented Image with K-means')
plt.imshow(segmented_image)
plt.axis('off')
plt.show()
```

## Edge Detection

```
import cv2
import numpy as np
from scipy import ndimage

# Read the original image
img = cv2.imread('686027dc0db04ae6272ac72b6a7ed195.jpg')
# Display original image
cv2.imshow('Original', img)

# Convert to grayscale
img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
# Blur the image for better edge detection
img_blur = cv2.GaussianBlur(img_gray, (3,3), 0)

# Sobel Edge Detection
sobelx = cv2.Sobel(src=img_blur, ddepth=cv2.CV_64F, dx=1, dy=0, ksize=5) #
Sobel Edge Detection on the X axis
sobely = cv2.Sobel(src=img_blur, ddepth=cv2.CV_64F, dx=0, dy=1, ksize=5) #
Sobel Edge Detection on the Y axis
sobelxy = cv2.Sobel(src=img_blur, ddepth=cv2.CV_64F, dx=1, dy=1, ksize=5) #
Combined X and Y Sobel Edge Detection
# Display Sobel Edge Detection Images
cv2.imshow('Sobel X', sobelx)
cv2.imshow('Sobel Y', sobely)
cv2.imshow('Sobel X Y using Sobel() function', sobelxy)

# Canny Edge Detection
edges = cv2.Canny(image=img_blur, threshold1=100, threshold2=200) # Canny
Edge Detection
cv2.imshow('Canny Edge Detection', edges)

# Prewitt Edge Detection
prewittx = cv2.filter2D(img_blur, -1, np.array([[1, 0, -1], [1, 0, -1], [1,
0, -1]])) # Prewitt X
prewitty = cv2.filter2D(img_blur, -1, np.array([[1, 1, 1], [0, 0, 0], [-1,
-1, -1]])) # Prewitt Y
prewittxy = prewittx + prewitty
```

```
cv2.imshow('Prewitt X', prewittx)
cv2.imshow('Prewitt Y', prewitty)
cv2.imshow('Prewitt XY', prewittxy)

# Roberts Edge Detection
robertsx = cv2.filter2D(img_blur, -1, np.array([[1, 0], [0, -1]])) # Roberts
X
robertsy = cv2.filter2D(img_blur, -1, np.array([[0, 1], [-1, 0]])) # Roberts
Y
robertsxy = robertsx + robertsy
cv2.imshow('Roberts X', robertsx)
cv2.imshow('Roberts Y', robertsy)
cv2.imshow('Roberts XY', robertsxy)

# Laplacian of Gaussian (LoG) Edge Detection
log = cv2.Laplacian(img_blur, cv2.CV_64F, ksize=3)
cv2.imshow('Laplacian of Gaussian (LoG)', log)

# Zero-Crossing Edge Detection (menggunakan Laplacian)
laplacian = cv2.Laplacian(img, cv2.CV_64F)
zero_crossing = np.zeros_like(laplacian)
zero_crossing[laplacian > 0] = 255
zero_crossing[laplacian < 0] = 0
cv2.imshow('Zerro Crossing', zero_crossing)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

Penjelasan Source Code Segmentasi

```
import cv2  # Mengimpor pustaka OpenCV untuk pemrosesan gambar
import numpy as np  # Mengimpor pustaka NumPy untuk manipulasi array

threshold = 0.5  # Menentukan ambang batas untuk warna
kernel5 = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (20, 20))  #
Membuat kernel berbentuk elips untuk operasi morfologi
x_co = 0  # Koordinat X untuk klik mouse
y_co = 0  # Koordinat Y untuk klik mouse
hsv = None  # Variabel untuk menyimpan gambar dalam ruang warna HSV
H = 125  # Nilai awal untuk hue
S = 220  # Nilai awal untuk saturation
V = 170  # Nilai awal untuk value
thr_H = 180 * threshold  # Ambang batas untuk hue
thr_S = 255 * threshold  # Ambang batas untuk saturation
thr_V = 255 * threshold  # Ambang batas untuk value

# Fungsi callback mouse untuk menangkap warna saat mengklik gambar
def on_mouse(event, x, y, flag, param):
    global x_co, y_co, H, S, V, hsv  # Mengakses variabel global
    if event == cv2.EVENT_LBUTTONDOWN:  # Jika tombol kiri mouse
ditekan
        x_co = x  # Simpan koordinat X
        y_co = y  # Simpan koordinat Y
        p_sel = hsv[y_co][x_co]  # Ambil nilai HSV dari piksel yang
dipilih
        H = p_sel[0]  # Set nilai H
        S = p_sel[1]  # Set nilai S
        V = p_sel[2]  # Set nilai V

cv2.namedWindow("image", 1)  # Membuat jendela untuk gambar asli
cv2.namedWindow("mask", 2)  # Membuat jendela untuk mask
cv2.namedWindow("segmented", 3)  # Membuat jendela untuk gambar
tersegmentasi

# Memuat gambar dari file
image_path = 'semangka.jpeg'  # Ganti ini dengan path gambar Anda
src = cv2.imread(image_path)  # Membaca gambar

# Memeriksa apakah gambar berhasil dimuat
if src is None:
    print("Failed to load image.")  # Menampilkan pesan jika gambar
gagal dimuat
    exit()  # Keluar dari program

# Menerapkan blur untuk menghaluskan gambar
```

```
src = cv2.blur(src, (3, 3))  # Menggunakan filter blur 3x3

# Mengonversi gambar ke ruang warna HSV
hsv = cv2.cvtColor(src, cv2.COLOR_BGR2HSV)  # Konversi dari BGR ke HSV

# Menetapkan callback mouse untuk pemilihan warna
cv2.setMouseCallback("image", on_mouse, 0)  # Mengaitkan fungsi callback mouse ke jendela gambar

while True:
    # Mendefinisikan rentang ambang warna
    min_color = np.array([H - thr_H, S - thr_S, V - thr_V])  # Nilai minimum warna
    max_color = np.array([H + thr_H, S + thr_S, V + thr_V])  # Nilai maksimum warna
    mask = cv2.inRange(hsv, min_color, max_color)  # Membuat mask berdasarkan rentang warna

    # Transformasi morfologi untuk menutup lubang kecil
    mask = cv2.morphologyEx(mask, cv2.MORPH_CLOSE, kernel5)  # Menutup area kecil dalam mask

    # Menampilkan nilai HSV dari titik yang dipilih
    cv2.putText(mask, f"H:{H} S:{S} V:{V}", (10, 30), cv2.FONT_HERSHEY_PLAIN, 2.0, (255, 255, 255), thickness=1)

    # Menampilkan mask dan gambar asli
    cv2.imshow("mask", mask)  # Menampilkan jendela mask
    cv2.imshow("image", src)  # Menampilkan jendela gambar asli

    # Mengsegmentasikan area warna yang dipilih
    src_segmented = cv2.add(src, src, mask=mask)  # Menggunakan mask untuk menggabungkan gambar
    cv2.imshow("segmented", src_segmented)  # Menampilkan gambar tersegmentasi

    # Menghentikan loop saat tombol 'ESC' ditekan
    if cv2.waitKey(10) == 27:  # Kode 27 adalah tombol ESC
        break

# Menutup semua jendela
cv2.destroyAllWindows()  # Menghentikan semua jendela OpenCV
```

Penjelasan Source Code K- Means

```
import cv2  # Mengimpor pustaka OpenCV untuk pemrosesan gambar
import numpy as np  # Mengimpor pustaka NumPy untuk manipulasi array
from sklearn.cluster import KMeans  # Mengimpor KMeans dari scikit-learn untuk klustering
import matplotlib.pyplot as plt  # Mengimpor pustaka matplotlib untuk visualisasi

# Load the image
image = cv2.imread('686027dc0db04ae6272ac72b6a7ed195.jpg')  # Membaca gambar dari file
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)  # Mengonversi dari BGR ke RGB

# Reshape the image to a 2D array of pixels
pixels = image.reshape(-1, 3)  # Mengubah bentuk gambar menjadi array 2D (jumlah piksel, 3 warna)

# Number of clusters (K)
k = 5  # Menentukan jumlah kluster untuk K-Means

# Apply K-means clustering
kmeans = KMeans(n_clusters=k)  # Membuat objek KMeans dengan jumlah kluster K
kmeans.fit(pixels)  # Menerapkan K-Means pada data piksel

# Get the cluster centers (dominant colors)
colors = kmeans.cluster_centers_.astype(int)  # Mendapatkan pusat kluster (warna dominan)

# Assign each pixel to the nearest cluster center
labels = kmeans.labels_  # Mendapatkan label kluster untuk setiap piksel
segmented_image = colors[labels].reshape(image.shape)  # Mengganti setiap piksel dengan warna pusat kluster

# Display the original and segmented images
plt.figure(figsize=(10, 5))  # Menetapkan ukuran figure untuk tampilan
plt.subplot(1, 2, 1)  # Membuat subplot untuk gambar asli
plt.title('Original Image')  # Menambahkan judul untuk gambar asli
```

```
plt.imshow(image)  # Menampilkan gambar asli
plt.axis('off')  # Menyembunyikan sumbu

plt.subplot(1, 2, 2)  # Membuat subplot untuk gambar tersegmentasi
plt.title('Segmented Image with K-means')  # Menambahkan judul untuk gambar
tersegmentasi
plt.imshow(segmented_image)  # Menampilkan gambar tersegmentasi
plt.axis('off')  # Menyembunyikan sumbu

plt.show()  # Menampilkan semua gambar
```

Penjelasan Source Code Edge Detection

## SOBEL

```
# Sobel Edge Detection
sobelx = cv2.Sobel(src=img_blur, ddepth=cv2.CV_64F, dx=1, dy=0, ksize=5)
# Sobel untuk mendeteksi tepi horizontal (X)
sobely = cv2.Sobel(src=img_blur, ddepth=cv2.CV_64F, dx=0, dy=1, ksize=5)
 # Sobel untuk mendeteksi tepi vertikal (Y)
sobelxy = cv2.Sobel(src=img_blur, ddepth=cv2.CV_64F, dx=1, dy=1, ksize=5)
# Sobel untuk mendeteksi tepi kombinasi (XY)

# Display Sobel Edge Detection Images
cv2.imshow('Sobel X', sobelx)    # Menampilkan hasil tepi horizontal (X)
cv2.imshow('Sobel Y', sobely)    # Menampilkan hasil tepi vertikal (Y)
cv2.imshow('Sobel X Y using Sobel() function', sobelxy)  # Menampilkan hasil
kombinasi tepi (XY)
```

## PREWITT

```
# Prewitt Edge Detection
prewittx = cv2.filter2D(img_blur, -1, np.array([[1, 0, -1], [1, 0, -1], [1,
0, -1]]))  # Prewitt untuk mendeteksi tepi horizontal (X)
prewitty = cv2.filter2D(img_blur, -1, np.array([[1, 1, 1], [0, 0, 0], [-1,
-1, -1]]))  # Prewitt untuk mendeteksi tepi vertikal (Y)
prewittxy = prewittx + prewitty  # Menggabungkan tepi horizontal dan vertikal
(XY)

# Display Prewitt Edge Detection Images
cv2.imshow('Prewitt X', prewittx)   # Menampilkan hasil tepi horizontal (X)
cv2.imshow('Prewitt Y', prewitty)   # Menampilkan hasil tepi vertikal (Y)
cv2.imshow('Prewitt XY', prewittxy) # Menampilkan hasil kombinasi tepi (XY)
```

## ROBERT

```
# Roberts Edge Detection
robertsx = cv2.filter2D(img_blur, -1, np.array([[1, 0], [0, -1]]))  # Roberts
untuk mendeteksi tepi diagonal (X)
robertsy = cv2.filter2D(img_blur, -1, np.array([[0, 1], [-1, 0]]))  # Roberts
untuk mendeteksi tepi diagonal (Y)
robertsxy = robertsx + robertsy  # Menggabungkan tepi diagonal (XY)

# Display Roberts Edge Detection Images
cv2.imshow('Roberts X', robertsx)   # Menampilkan hasil tepi diagonal (X)
cv2.imshow('Roberts Y', robertsy)   # Menampilkan hasil tepi diagonal (Y)
cv2.imshow('Roberts XY', robertsxy) # Menampilkan hasil kombinasi tepi
diagonal (XY)
```

## LAPLACIAN OF GAOSSIAN

```
# Laplacian of Gaussian (LoG) Edge Detection
log = cv2.Laplacian(img_blur, cv2.CV_64F, ksize=3)  # Menggunakan Laplacian
untuk mendeteksi tepi setelah Gaussian Blur
cv2.imshow('Laplacian of Gaussian (LoG)', log) # Menampilkan hasil deteksi
tepi menggunakan LoG
```

## ZERRO CROSSING

```
# Zero-Crossing Edge Detection (menggunakan Laplacian)
laplacian = cv2.Laplacian(img, cv2.CV_64F) # Menghitung Laplacian dari gambar
asli untuk mendeteksi tepi
```

```
zero_crossing = np.zeros_like(laplacian) # Membuat citra kosong untuk menyimpan hasil zero-crossing
zero_crossing[laplacian > 0] = 255   # Menandai piksel positif pada Laplacian sebagai 255 (putih)
zero_crossing[laplacian < 0] = 0   # Menandai piksel negatif pada Laplacian sebagai 0 (hitam)
cv2.imshow('Zero Crossing', zero_crossing) # Menampilkan hasil deteksi tepi menggunakan zero-crossing
```

**CANNY**

```
# Canny Edge Detection
edges = cv2.Canny(image=img_blur, threshold1=100, threshold2=200)  # Menggunakan metode Canny untuk mendeteksi tepi
cv2.imshow('Canny Edge Detection', edges)  # Menampilkan hasil deteksi tepi menggunakan Canny
```

# BAB 3
# OUTPUT PROGRAM

Implementasi Hasil Source Code

Segmentasi Citra Gambar

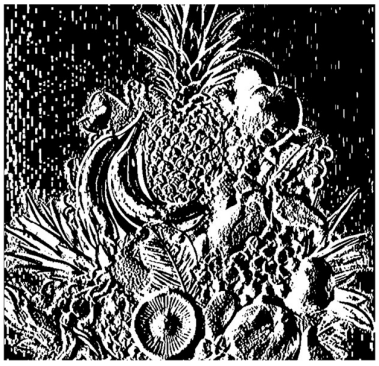

*Image 1 Original* *Image 2 Mask* *Image 3 Segmen*

Algoritma K-Means

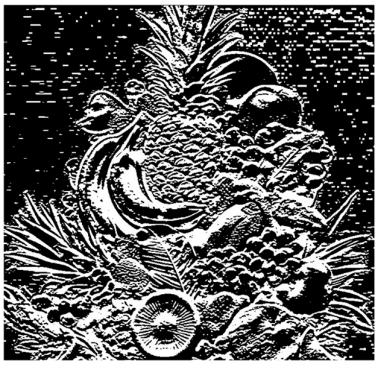

Edge Detection

Original Image

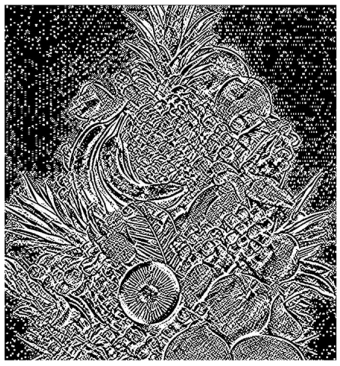
Sobel



*Edge Detection 1 Sobel XY*



*Edge Detection 2 Sobel Y*



*Edge Detection 3 Sobel X*

Prewitt

*Edge Detection 4 Prewitt X*

*Edge Detection 5 Prewitt Y*

*Edge Detection 6 Prewitt XY*

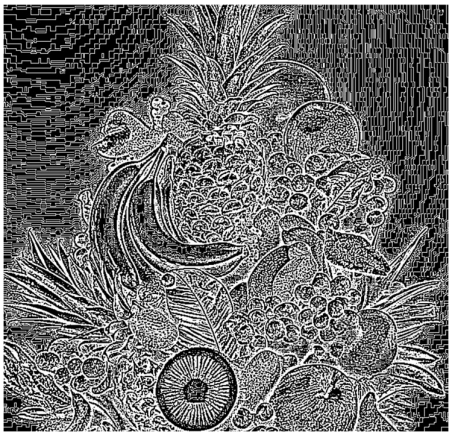
Roberts

*Edge Detection  7 Roberts X*

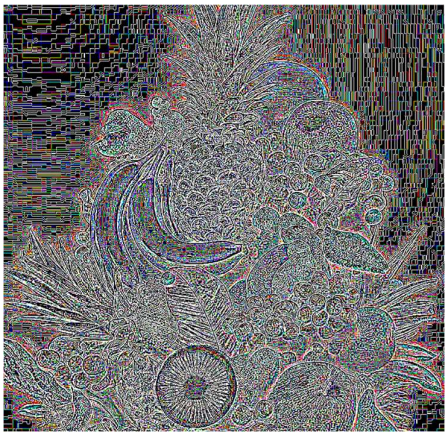
*Edge Detection  8 Roberts Y*

*Edge Detection  9 Roberts XY*

Laplacian of Gaossian



Zerro Crossing



Canny

# BAB 4
# PENUTUP

Kesimpulan

Citra yang dihasilkan dari setiap metode segmentasi memiliki karakteristik yang berbeda. Pada metode thresholding, citra yang dihasilkan adalah citra biner. Meskipun citra input berupa citra berwarna (RGB), proses segmentasi melibatkan konversi citra menjadi grayscale. Proses konversi ini menghasilkan citra hitam-putih, di mana kualitas citra bergantung pada nilai ambang yang ditentukan. Piksel dengan intensitas yang lebih tinggi atau sama dengan nilai ambang akan tampak putih, sementara piksel dengan intensitas lebih rendah dari ambang akan tampak hitam.

Setiap algoritma deteksi tepi memiliki karakteristik yang berbeda. **Prewitt** dan **Sobel** cocok untuk deteksi tepi sederhana, dengan Sobel memberikan hasil yang lebih halus. **Roberts** lebih sensitif terhadap perubahan intensitas yang tajam, tetapi sangat terpengaruh oleh noise. **LoG** dan **Zero Crossing** bekerja baik untuk mendeteksi tepi halus dan presisi tinggi, terutama ketika citra mengandung noise. **Canny** dianggap salah satu metode terbaik karena menggabungkan peredaman noise, penelusuran tepi, dan penggunaan ambang ganda untuk mendeteksi tepi yang lebih akurat.

Dengan menggunakan berbagai metode ini, kita dapat membandingkan hasil deteksi tepi dari setiap algoritma dan memilih yang paling sesuai dengan kebutuhan spesifik citra atau aplikasi yang digunakan.

# DAFTAR PUSTAKA


Efran, F. A., Jumadi, J., & Khairil. (2022). Implementasi Metode K-Means Clustering Pada Segmentasi Citra Digital. *Jurnal Media Infotama*, 291.

Putra, A., Sihombing, V., & Munandar, M. H. (2021). RANCANG BANGUN APLIKASI DETEKSI TEPI CITRA DIGITAL MENGGUNAKAN ALGORTIMA PREWITT. *TEKINKOM, 4*, 214.

Supiyandi, Trisatin, P., Nuzul, R., Sri, R. D., & Salsabila, Y. (2024, Juli 3). Deteksi Tepi Sederhana Pada Citra Menggunakan Operator Sobel. *Repeater : Publikasi Teknik Informatika dan Jaringan, 2*, 43-56.

Syafi'i, S. I., Wahyuningrum, R. T., & Muntasa, A. (2015). SEGMENTASI OBYEK PADA CITRA DIGITAL MENGGUNAKAN METODE OTSU THRESHOLDING. *Jurnal Informatika*, 1-8.