Feature Extraction

We are not able to capture an image, represent it in an array format and apply a filter on it. However, robotic applications are interested in finding the answer to the questions such as: what is the pose of the object? What type of object is it? how fast is it moving? how fast am I moving? and so on. The answers to such questions are measurements obtained from the image, and which we call image features. Features are the gist of the scene and the raw material that we need for robot control.

We plan to look at three types of image features. 1. regions, 2. lines, and 3. interest points.

Region Features

Region features are the attributes extracted out from the original image where the extracted image pixels would have similar properties. Instances include area, perimeter, centroid, shape, or boundry description.

Segmentation of an image into regions comprises three subproblems:

- Pixel classification: decision process applied to each pixel that assigns the pixel to one of C number of classes.
- Object instance Representation: adjacent pixels of the same class are connected to form m spatial sets
- Object instance description: the sets S_i are described in terms of compact scalar or vector-valued features such as size, position, and shape. In some applications there is an additional step – semantic classification – where we assign an application meaningful label to the pixel classes or sets, for example, "cat","dog", "coffee cup" etc.

Pixel Classification

The simplest and least computationally heavy option would be to classify pixels into classes based on a threshold for the grayscale images. This threshold can be approximated by analyzing the histogram of pixel-value.

When it comes to color images, a simple threshold will be insufficient to classify pixels in the image into two or more classes. It requires a series of steps to identify the interested pixel regions in a given color image.

For example of separately identifying the tomatoes in the tomatoes.png, we first convert the RGB color space to Lab space. Lab* color space is a device-independent model that separates lightness (L*) from color components (a* for green-red axis, b* for blue-yellow axis), designed to mimic human vision and enable perceptual uniformity. Then we extract only the ab space to remove the illumination from the image.

Once illumination is removed, we can use the k-means algorithm in the image to categorize the image into k number of clusters. It should be easier to identify the red pixels separately after the clustering step. Next, we generate a mask out of the cluster results and smooth it using a morphological operator. This results in better classification of red colored pixels in the image.

The following is the complete code:

```
tmt = cv2.imread('/workspaces/base_ros2/src/image_processing/images/tomatoes
    .jpg')

lab_tmt = cv2.cvtColor(tmt, cv2.COLOR_BGR2LAB)

# Extract a* and b* planes (2-channel image)
ab_tmt = lab_tmt[:, :, 1:3]  # Shape: (height, width, 2)_tmt

# Create a dummy L* channel with neutral value 50, matching the shape of the
    first channel of ab_tmt
```

```
l_channel = np.full_like(ab_tmt[:, :, 0:1], 50)

# Concatenate the dummy L* channel with ab_tmt along the third axis to form
    a 3-channel L*a*b* image
lab_dummy = np.concatenate((l_channel, ab_tmt), axis=2)

# Convert the L*a*b* image back to BGR and save it as 'ab_tmt.jpg'
cv2.imwrite('ab_tmt.jpg', cv2.cvtColor(lab_dummy, cv2.COLOR_Lab2BGR))

# Commented out: Calculate histogram of b_tmt (grayscale image)
# hist = cv2.calcHist([b_tmt], [0], None, [256], [0, 256])

# Commented out: Plot histogram
# plt.plot(hist)
# plt.title('Grayscale Histogram')
# plt.xlabel('Pixel Value')
# plt.ylabel('Frequency')
# plt.savefig('b_tmt_hist.png')
# plt.close()

# Get height and width from ab_tmt shape (excluding channel dimension)
height, width = ab_tmt.shape[:2]

# Reshape ab_tmt into a 2D array of a*b* points for k-means clustering
ab_flat = ab_tmt.reshape((height * width, 2)).astype(np.float32)

# Define k-means termination criteria: epsilon or max 10 iterations with
    precision 1.0
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 10, 1.0)

# Perform k-means clustering with 3 clusters, random centers, 10 attempts;
    returns labels and centers
_, labels, centers = cv2.kmeans(ab_flat, 3, None, criteria, 10, cv2.
    KMEANS_RANDOM_CENTERS)

# Reshape the cluster labels back to the original image dimensions
clustered = labels.reshape((height, width))

# Save the clustered image, scaled to 0-255 range by multiplying by 85
    (255//3)
cv2.imwrite('clustered_ab.png', clustered * (255 // 3))

# Create a mask initialized with 255 (white) for all pixels
mask = np.ones_like(clustered, dtype=np.uint8) * 255

# Set pixels where cluster label is 1 to 0 (black) in the mask
mask[(clustered == 1)] = 0

# Save the mask image as 'clsutered_mask.png' (note: typo 'clsutered' should
     be 'clustered')
cv2.imwrite('clustered_mask.png', mask)

# Define a structuring element for erosion with an ellipse of radius 7
radius = 7
se = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (2*radius+1, 2*radius+1))

# Erode the binary mask to smooth edges and remove small noise
mask_eroded = cv2.erode(mask, se)
```

```
# Save the eroded mask as 'clsutered_mask_smoothed.jpg' (note: typo '
    clsutered' should be 'clustered')
cv2.imwrite('clustered_mask_smoothed.jpg', mask_eroded)

# Apply the inverted mask to an assumed 'tmt' image (assuming tmt is defined
    RGB image)
# Note: 'tmt' should be defined earlier as cv2.imread('tmt.jpg') for this to
    work
masked_tmt = cv2.bitwise_and(tmt, tmt, mask=cv2.bitwise_not(mask_eroded))

# Save the masked tmt image
cv2.imwrite('tmt_masked.jpg', masked_tmt)
```

Object Instance Representation

An object is a spatially contiguous region of pixels of the same class. Objects in a binary scene are commonly known as blobs, regions, or connected components. In this section we consider the problem of allocating pixels to spatial sets $S_0, \ldots, S_{m-1}$ which correspond to the human notion of objects, and where m is the number of objects in the scene.

A blob is a region in an image where some properties, such as intensity or color, are uniform or vary within a defined range. In computer vision, detecting blobs(regions) that differ from their surroundings is a common and powerful technique. A blob can be as simple as a spot of light in an image or as complex as a moving object in a video. Blob detection is crucial in various domains such as microscopy, surveillance, object tracking, astronomy, and medical imaging. Below is a simple example for detecting blobs in a binary image using Computer Vision library's blob detector.

```
tmt_binary = cv2.imread('/workspaces/base_ros2/src/image_processing/images/
    tomatoes.jpg', cv2.IMREAD_GRAYSCALE)

# Detect blobs using SimpleBlobDetector
params = cv2.SimpleBlobDetector_Params()
params.filterByArea = True
params.minArea = 1000
params.maxArea = 100000
params.filterByCircularity = False
params.filterByConvexity = False  # Disabled for broader detection
params.filterByInertia = False  # Disabled for flexibility

detector = cv2.SimpleBlobDetector_create(params)

# Detect blobs
keypoints = detector.detect(tmt_binary)
# Define colors for different blobs (cycle through red, green, blue)
colors = [(0, 0, 255), (0, 255, 0), (255, 0, 0)]  # BGR format

print(keypoints)

# Draw each blob with a different color
for i, kp in enumerate(keypoints):
    color = colors[i % len(colors)]
    center = (int(kp.pt[0]), int(kp.pt[1]))
    radius = int(kp.size / 2)
    cv2.circle(tmt, center, radius, color, 10)  # Draw circle outline
# Save or display result
cv2.imwrite('blobs_detected.jpg', tmt)
```