# Computational Project 1
# (Speed Detection)

Lado Turnamidze

25.10.2024

Task of the Computational Project: Extract number of objects and conventional motion speed from a video with edge detection and clustering. In the first section, I will talk about edge detection with an example and code in Python. In the second part, I will explain clustering algorithm. Finally, I will combine both to get the job done.

## 1 Edge Detection

I will use the following image and transform it step by step to get to the final result.

Figure 1: Original 'Son_Goku.png'

## 1.1 Smoothing Kernels

Smoothing filters are used in preprocessing step mainly for noise removal. I will be using $5 \times 5$ Gaussian Kernel with $\sigma = 1$. You can find the corresponding matrix in 1.3.1. Here is the constructor for `CannyEdgeDetector` class:

```
def __init__(self, sigma=1, kernel_size=5, weak_pixel=75,
    strong_pixel=255, lowthreshold=0.05, highthreshold=0.15):
  self.img_smoothed = None
  self.gradient = None
  self.theta = None
  self.nonMaxImg = None
  self.thresholdImg = None
  self.weak_pixel = weak_pixel
  self.strong_pixel = strong_pixel
  self.sigma = sigma
  self.kernel_size = kernel_size
  self.lowThreshold = lowthreshold
  self.highThreshold = highthreshold
```

Figure 2: After applying $5 \times 5$ Gaussian Kernel with $\sigma = 1$

```
def gaussian_kernel(self, size, sigma=1):
  size = int(size) // 2
  x, y = np.mgrid[-size:size + 1, -size:size + 1]
  normal = 1 / (2.0 * np.pi * sigma ** 2)
  return np.exp(-((x ** 2 + y ** 2) / (2.0 * sigma ** 2))) * normal
```

## 1.2 Edge Detection Kernels

Edges indicate the boundaries of objects, making edge detection a crucial pre-processing step in any object detection or recognition task. Here are some basic edge detection kernels:

Prewitt Operators:

$$I_x(x,y) = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \quad I_y(x,y) = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$$

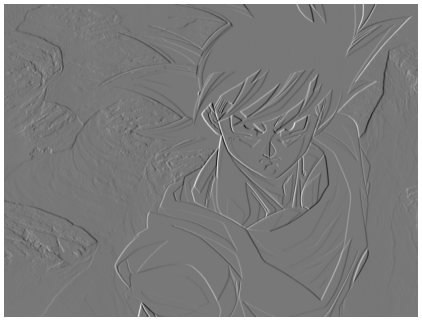
Sobel Operators:

$$I_x(x,y) = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}, \quad I_y(x,y) = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

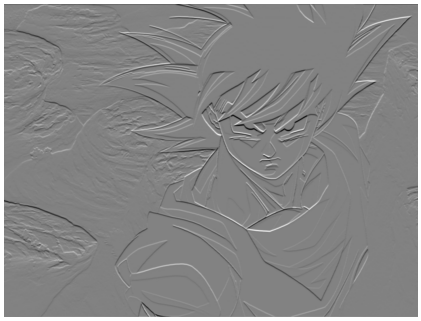
$$\text{Laplacian: } \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Code for convolution with Sobel Operators:

```
def sobel_filters(self, image):
  K_x = np.array([[1, 0, -1], [2, 0, -2], [1, 0, -1]], np.float32)
  K_y = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]], np.float32)

  I_x = ndimage.convolve(image, K_x)
  I_y = ndimage.convolve(image, K_y)

  magnitude = np.hypot(I_x, I_y)
  magnitude = magnitude / magnitude.max() * 255 if magnitude.max()
                                  > 0 else magnitude
  direction = np.arctan2(I_y, I_x)

  return (magnitude, direction)
```



(a) Sobel $I_x(x,y)$



(b) Sobel $I_y(x,y)$

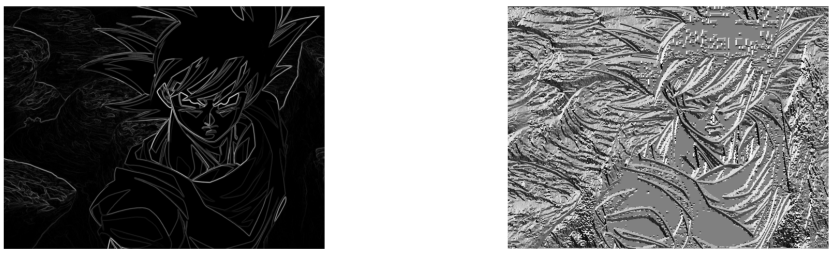
Figure 3: Sobel Gradients

Gradient magnitude represents the "strength" of the edge of the image and relates to both directions $I_x(x, y)$ and $I_y(x, y)$:

$$I_{xy} = \sqrt{I_x(x, y)^2 + I_y(x, y)^2}$$

Gradient angle represents the direction of the edge or direction of intensity variation:

$$I_\theta = \tan^{-1} \frac{I_y(x, y)}{I_x(x, y)}$$



(a) $I_{xy}$ (b) $I_\theta$

Figure 4: Gradient Magnitude and Direction

## 1.3 Canny Edge Detection Algorithm

This 5-stage algorithm, developed by John Canny in 1986, detects wide range of edges. First 2 steps have already been visualized. 5 steps of the algorithm are:

1. Smoothing for Noise Removal
2. Finding Gradients
3. None-Maximum Suppression
4. Double Thresholding
5. Edge Tracking by Hysteresis

### 1.3.1 Smoothing for Noise Removal

The first stage in canny edge detection algorithm is smoothing to remove noise that may cause false edges. Kernel used in this step is $5 \times 5$ Gaussian kernel with $\sigma = 1$:

$$\frac{1}{273}\begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix}$$

### 1.3.2 Finding Gradients

This is done using the Sobel operators in both $x$ and $y$ direction and getting gradient magnitude as described above:

$$I_x(x,y) = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}, \quad I_y(x,y) = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

$$I_{xy} = \sqrt{I_x(x,y)^2 + I_y(x,y)^2}$$

$$I_\theta = \tan^{-1} \frac{I_y(x,y)}{I_x(x,y)}$$

### 1.3.3 None-Maximum Suppression

Our goal is to suppress all weak edges while retaining local maxima. To achieve this, we utilize the gradient direction image.

1. Compare the edge strength of the current pixel with the edge strengths of the pixel in both the positive and negative gradient directions.
2. If the value of the current pixel is lower than the values of the pixels in the same direction, suppress it by setting it to zero.

Direction must be quantized to 8 directions or angles to use 8-Connectivity. The result of this stage would be an edge image with thin edges.

Figure 5: After Non-Maximum Suppression

```
def non_max_suppression(self, image, degrees):
  height, width = image.shape
  output = np.zeros_like(image)
  direction = degrees * 180. / np.pi
  direction[direction < 0] += 180

  for i in range(1, height - 1):
    for j in range(1, width - 1):
      q = 255
      r = 255

      if (0 <= direction[i, j] < 22.5) or
        (157.5 <= direction[i, j] <= 180):  # 0
        q = image[i, j + 1]
        r = image[i, j - 1]
      elif 22.5 <= direction[i, j] < 67.5:  # 45
        q = image[i + 1, j - 1]
        r = image[i - 1, j + 1]
      elif 67.5 <= direction[i, j] < 112.5:  # 90
        q = image[i + 1, j]
        r = image[i - 1, j]
      elif 112.5 <= direction[i, j] < 157.5:  # 135
        q = image[i - 1, j - 1]
        r = image[i + 1, j + 1]

      if (image[i, j] >= q) and (image[i, j] >= r):
        output[i, j] = image[i, j]
      else:
        output[i, j] = 0

  return output
```

### 1.3.4 Double Thresholding

After non-maximum suppression, we need to remove pixels with low gradient values to retain only strong edges. While non-maximum suppression addresses only local weak edges, this step focuses on globally eliminating weak edges. We use two thresholds, $T_l$ and $T_h$, which are chosen by the user.

**Algorithm 1** Thresholding Algorithm

```
for each pixel (x, y) in image do
    if image[x, y] < T_l then
        image[x, y] ← 0
    else if image[x, y] > T_h then
        image[x, y] ← 1
    end if
end for
```

Figure 6: Combining Threshold

```
def double_thresholding(self, img):
  highThreshold = img.max() * self.highThreshold
  lowThreshold = highThreshold * self.lowThreshold

  height, width = img.shape
  res = np.zeros((height, width), dtype=np.float32)

  strong_i, strong_j = np.where(img >= highThreshold)
  weak_i, weak_j = np.where((img >= lowThreshold) & (img <
                                highThreshold))

  res[strong_i, strong_j] = self.strong_pixel
  res[weak_i, weak_j] = self.weak_pixel

  return res
```

### 1.3.5 Edge Tracking By Hysteresis

Finally, for pixels with values between $T_l$ and $T_h$, we examine the 8 neighboring pixels to determine if any of them contain a strong point. If a strong point is found among the neighbors, the pixel is considered part of the edge. Otherwise, it is suppressed.

```
def hysteresis(self, img):
  height, width = img.shape
  weak = self.weak_pixel
  strong = self.strong_pixel

  result = np.copy(img).astype(np.float32)

  dx = [-1, -1, -1, 0, 0, 1, 1, 1]
  dy = [-1, 0, 1, -1, 1, -1, 0, 1]

  for i in range(1, height - 1):
    for j in range(1, width - 1):
      if result[i, j] == weak:
        connected_to_strong = False
        for k in range(8):
          if result[i + dx[k], j + dy[k]] == strong:
            connected_to_strong = True
            break

        result[i, j] = strong if connected_to_strong else 0

  return result
```

Figure 7: Result of Canny Edge Detection Algorithm

Canny Edge Detection Algorithm using previously defined methods:

```
def edge_detect(self, image):
  if len(image.shape) == 3:
    image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
  image = image.astype(np.float32)

  self.img_smoothed = ndimage.convolve(image, self.gaussian_kernel(
                                  self.kernel_size, self.sigma))
  self.gradient, self.theta = self.sobel_filters(self.img_smoothed)
  self.nonMaxImg = self.non_max_suppression(self.gradient, self.
                                  theta)
  self.thresholdImg = self.double_thresholding(self.nonMaxImg)
  img_final = self.hysteresis(self.thresholdImg)

  return img_final
```

# 2 Clustering Algorithm

For purposes of this task, I found density based clustering algorithms to be the most suitable, since we do not know how many objects will be in the video. Density based algorithms do not need to specify the number of clusters, unlike K-means or K-medoids.

## 2.1 DBSCAN + K-d Trees

*DBSCAN* would be one of those density based algorithms and it does work correctly, but there is a major problem. It works in $O(n^2)$, which is slow, even for low resolution videos. For that reason, I started searching for faster solutions and it turns out, *DBSCAN* with spatial index works (most of the time) in $O(n \cdot \log n)$, which is obviously better.

## 2.2 K-d Trees Overview

A K-d Tree (k-dimensional tree) is a data structure that organizes points in a k-dimensional space. It is particularly useful for applications that involve multi-dimensional search keys, such as range searches and nearest neighbor searches. Here's how it works:

### 2.2.1 Structure of K-d Tree

1. **Node Representation**: Each node in the K-d Tree represents a point in k-dimensional space. The `Node` class encapsulates this concept, holding the data (the point) and references to left and right child nodes.

2. **Building the Tree**:

   - The K-d Tree is built recursively. The `_build` method takes a list of points and a depth parameter.
   - The depth determines which dimension (or axis) to split on. For example, at depth 0, the tree splits on the first dimension (x-axis), at depth 1 on the second dimension (y-axis), and so on.
   - The data points are sorted based on the current axis, and the median point is chosen as the root of the subtree. This median point ensures that half of the points are on one side and half on the other, balancing the tree.
   - The left and right subtrees are then constructed recursively from the points to the left and right of the median, respectively.

3. **Searching the Tree**:

   - The `_search` method is used to find a target point in the K-d Tree.
   - Starting from the root, the search compares the target point with the current node's data. If they match, the node is returned.
   - If the target point is less than the current node's data along the current axis, the search continues in the left subtree; otherwise, it continues in the right subtree.
   - This process continues recursively until the target is found or a leaf node is reached (indicating that the target is not in the tree).

K-d Trees allow for efficient searching, especially in multidimensional spaces. The average time complexity for search operations is $O(\log n)$ for balanced trees, making them faster than linear search methods.

K-d Trees are widely used in various applications, including computer graphics, machine learning, and spatial databases

## 2.3 K-d Trees Code

### 2.3.1 Constructor

```
class KDTree:
  def __init__(self, points, dim, norm=None):
    self.dim = dim
    self._root = self._make(points, 0)
    self._distance = norm if norm is not None else \
      lambda a, b: sum((a[i] - b[i]) ** 2 for i in range(dim))
```

The constructor initializes the tree. The `dim` parameter specifies the number of dimensions of the space. The `points` parameter is a list of points to build the tree using `_make` function. If a custom distance norm is provided, it is used for distance calculation. Otherwise, the Euclidean distance is used by default.

### 2.3.2 Building the Tree

```
def _make(self, points, i=0):
  if len(points) > 1:
    points = points[points[:, i].argsort()]
    i = (i + 1) % self.dim
    m = len(points) // 2
    return [self._make(points[:m], i), \
              self._make(points[m + 1:], i), points[m]]
  if len(points) == 1:
    return [None, None, points[0]]
  return None
```

This private method recursively builds the K-d Tree. It sorts the points by one of the dimensions (rotating through each dimension in turn), selects the median point, and recursively divides the space into two halves. Each node in the tree contains the left and right children and the median point.

### 2.3.3 Adding a Point

```
def _add_point(self, node, point, i=0):
  if node is not None:
    dx = node[2][i] - point[i]
    for j, c in ((0, dx >= 0), (1, dx < 0)):
      if c and node[j] is None:
        node[j] = [None, None, point]
      elif c:
        self._add_point(node[j], point, (i + 1) % self.dim)

def add_point(self, point):
  if self._root is None:
    self._root = [None, None, point]
  else:
    self._add_point(self._root, point)
```

The add_point method adds a new point to the K-d Tree. It calls the private method _add_point, which traverses the tree recursively to find the correct location based on the dimensions and inserts a new point at a leaf node.

### 2.3.4 Nearest Neighbor Search

```
def get_nearest(self, point, return_dist_sq=True):
  l = self._get_knn(self._root, point, 1, return_dist_sq, [])
  return l[0] if len(l) else None

def get_knn(self, point, k, return_dist_sq=True):
  return self._get_knn(self._root, point, k, return_dist_sq, [])

def _get_knn(self, node, point, k, res_sq_dist, heap, i=0, t_brk=1)
                                    :
  if node is not None:
    dist_sq = self._distance(point, node[2])
    dx = node[2][i] - point[i]
    if len(heap) < k:
      heapq.heappush(heap, (-dist_sq, t_brk, node[2]))
    elif dist_sq < -heap[0][0]:
      heapq.heappushpop(heap, (-dist_sq, t_brk, node[2]))
    i = (i + 1) % self.dim

    for b in (dx < 0, dx >= 0)[:1 + (dx * dx < -heap[0][0])]:
      self._get_knn(node[int(b)], point, k, res_sq_dist, heap, i, (
                                    t_brk << 1) | int(b))
    if t_brk == 1:
      return [(-h[0], h[2]) if res_sq_dist else h[2] for h in
                                    sorted(heap)][::-1]
```

The `get_nearest` method returns the nearest point to the given query point. The `get_knn` method returns the `k` nearest neighbors. Both methods use the private method `_get_knn`, which performs the actual search using a priority queue (python's *heapq* library) to track the best candidates. The `_get_knn` method performs a recursive search to find the k nearest neighbors. It uses a heap to keep track of the closest points found so far. It calculates the squared distance between the query point and the current node and determines whether to continue searching the left or right subtree based on the current dimension.

### 2.3.5 Tree Traversal and Iteration

```
def _walk(self, node):
  if node is not None:
    for j in 0, 1:
      for x in self._walk(node[j]):
        yield x
    yield node[2]

def __iter__(self):
  return self._walk(self._root)
```

The `_walk` method is a generator that yields all points in the tree by performing an in-order traversal. The `__iter__` method allows for iteration over the points in the tree, making the class compatible with Python's iteration protocol.

### 2.3.6 Test and Plot

```
if __name__ == "__main__":
  import numpy as np
  import matplotlib.pyplot as plt

  points = np.array([
    [2, 3],
    [5, 4],
    [9, 6],
    [4, 7],
    [8, 1],
    [7, 2]
  ])

  kdtree = KDTree(points, dim=2)

  # Plot the KDTree with dimension splits
  plt.figure(figsize=(8, 8))
  plot_kdtree(kdtree._root, kdtree.dim)
  plt.xlim(-1, 10)
  plt.ylim(-1, 10)
  plt.gca().set_aspect('equal', adjustable='box')
  plt.title('K-d Tree Dimension Splits')
  plt.show()
```

```
def plot_kdtree(tree, dim, depth=0, bounds=None):
  if tree is None:
    return
  node = tree[2]
  if node is None:
    return
  axis = depth % dim
  if bounds is None:
    bounds = [[-10, 10], [-10, 10]]  # Example bounds for 2D case
  plt.scatter(*node[:2], c='red')
  if axis == 0:
    plt.plot([node[0], node[0]], bounds[1], 'k--')
    plot_kdtree(tree[0], dim, depth + 1, [[bounds[0][0], node[0]],
                                          bounds[1]])
    plot_kdtree(tree[1], dim, depth + 1, [[node[0], bounds[0][1]],
                                          bounds[1]])
  else:
    plt.plot(bounds[0], [node[1], node[1]], 'k--')
    plot_kdtree(tree[0], dim, depth + 1, [bounds[0], [bounds[1][0],
                                           node[1]]])
    plot_kdtree(tree[1], dim, depth + 1, [bounds[0], [node[1],
                                          bounds[1][1]]])
```

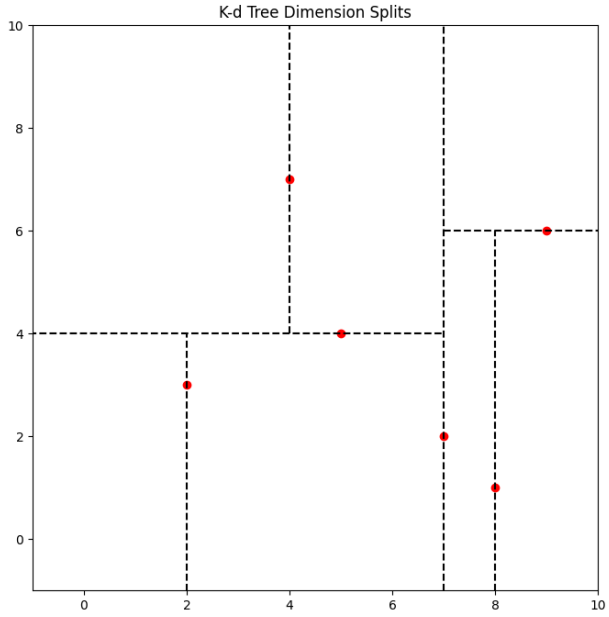



Figure 8: K-d Tree splitting dimensions along points

## 2.4 DBSCAN implementation with K-d Trees

```
def __init__(self, eps, min_pts, norm=norm2):
  self.eps = eps
  self.min_pts = min_pts
  self.norm = norm
  self.labels_ = None
  self.cluster_centers_ = None
```

```
def dbscan(self, X):
  n_samples = X.shape[0]
  self.labels_ = np.full(n_samples, -1, dtype=int)
  tree = KDTree(X, dim=X.shape[1], norm=self.norm)
  cluster_id = 0

  for i in range(n_samples):
    if self.labels_[i] != -1:
      continue

    neighbors = tree.get_knn(X[i], n_samples, return_dist_sq=True)
    neighbors = [(dist, point) for dist, point in neighbors if dist
                                  <= self.eps ** 2]

    if len(neighbors) < self.min_pts:  # Mark as noise
      self.labels_[i] = -2  # -2 represents noise points
      continue

    self.labels_[i] = cluster_id
    # Seed Set gets resized, while neighbors list does not
    seed_set = [(dist, point) for dist, point in neighbors]
    while seed_set:
      _, curr_point = seed_set.pop(0)
      curr_idx = np.where(np.all(X == curr_point, axis=1))[0][0]

      if self.labels_[curr_idx] == -1:  # Unvisited point
        self.labels_[curr_idx] = cluster_id
        curr_neighbors = tree.get_knn(curr_point, n_samples,
                                      return_dist_sq=True)
        curr_neighbors = [(dist, point) for dist, point in \
          curr_neighbors if dist <= self.eps ** 2]
        if len(curr_neighbors) >= self.min_pts:
          seed_set.extend(curr_neighbors)
      elif self.labels_[curr_idx] == -2:  # Noise point
        self.labels_[curr_idx] = cluster_id

    cluster_id += 1

  self.cluster_centers_ = []
  for cluster_id in set(self.labels_):
    if cluster_id >= 0:  # Ignore noise points
      cluster_points = X[self.labels_ == cluster_id]
      center = np.mean(cluster_points, axis=0)
      self.cluster_centers_.append(center)

  return self
```

Perform *DBSCAN* clustering on data $X$ of type `numpy.ndarray` of shape `(n_samples, n_features)`. It starts by initializing a labels array, `self.labels_`, where `-1` represents unvisited points and `-2` indicates noise points. `KDTree` is constructed using the input data for efficient neighborhood queries.

For each unvisited point, *DBSCAN* retrieves its neighbors from the tree within a radius `eps`. If the number of neighbors is less than `min_pts`, the point is labeled as noise. Otherwise, it becomes part of a new cluster, and its neighbors are further explored. The algorithm expands the cluster by iterating through the neighbor set, adding more points if they meet the criteria for clustering.

This process continues until all points are either assigned to a cluster or marked as noise. The method returns `self`, with the labels stored in `self.labels_` and centers of clusters stored in `self.cluster_centers_`.

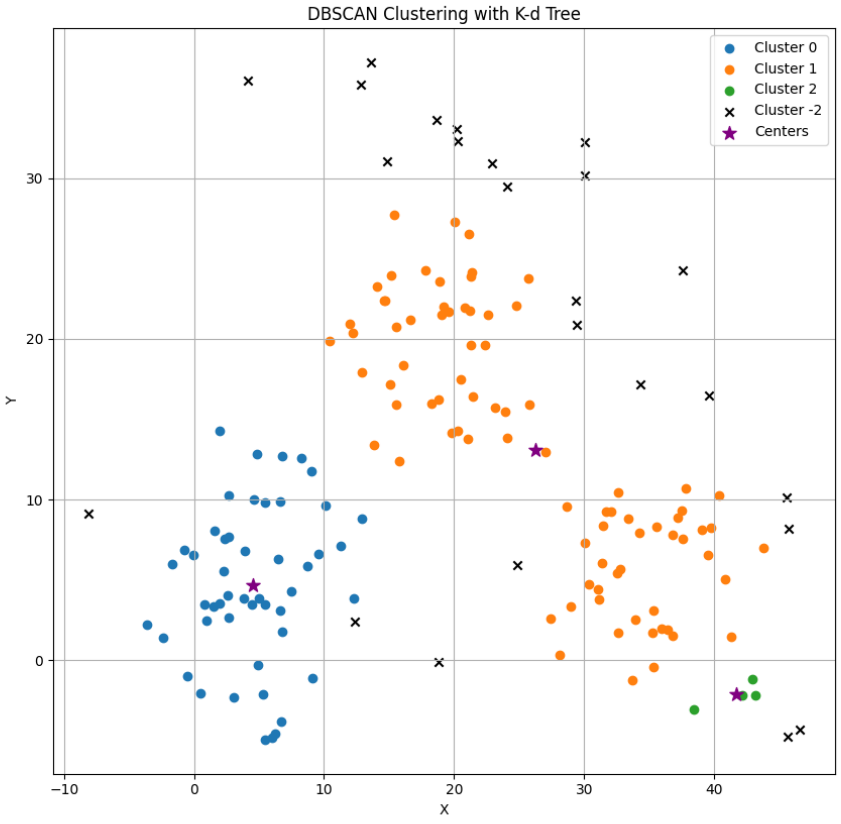



Figure 9: DBSCAN with K-d Tree

Test code:

```
if __name__ == "__main__":
  import matplotlib.pyplot as plt

  np.random.seed(42)
  cluster_1 = np.random.randn(50, 2) * 1.5 + [5, 5]
  cluster_2 = np.random.randn(50, 2) * 1.5 + [20, 20]
  cluster_3 = np.random.randn(50, 2) * 1.5 + [35, 5]
  outliers = np.random.uniform(low=0, high=40, size=(10, 2))
  X = np.vstack((cluster_1, cluster_2, cluster_3, outliers))
  dbscan = DBSCAN(eps=2, min_pts=5, norm=norm2)
  dbscan.dbscan(X)

  plt.figure(figsize=(10, 10))
  for cluster_id in set(dbscan.labels_):
    if cluster_id == -2:  # Noise points (outliers)
      color, marker = 'k', 'x'
    else:
      color = plt.colormaps['tab10'](cluster_id % 10)
      marker = 'o'

    cluster_points = X[dbscan.labels_ == cluster_id]
    plt.scatter(cluster_points[:, 0], cluster_points[:, 1],
          c=[color], marker=marker, label=f"Cluster {cluster_id}")


  if dbscan.cluster_centers_:
    centers = np.array(dbscan.cluster_centers_)
    plt.scatter(centers[:, 0], centers[:, 1], c='purple', s=100,
                                  marker='*', label='Centers')

  plt.title("DBSCAN Clustering with K-d Tree")
  plt.xlabel("X"), plt.ylabel("Y")
  plt.legend(), plt.grid(True), plt.show()
```

# 3 Putting It ∀ Together

Recall the task: Extract the number of objects and conventional motion speed from a video using edge detection and clustering. To count the number of objects in the video, each frame will undergo edge detection, and the difference between two consecutive frames will be used as the third dimension when clustering the activated pixels. As for the speed, I will use the centroids of each cluster and their movement over time to determine speed in units of pixels per frame (p/f).

Below is the constructor for `ObjectTracker` class:

```
def __init__(self, video_path: str, skip_frames: int = 5):
  print(f"\n[{datetime.now().strftime('%H:%M:%S')}] Initializing
                                    Tracker...")
  self.video_path = video_path
  self.skip_frames = skip_frames

  self.edge_detector = CannyEdgeDetector(
    sigma=1, kernel_size=5,
    weak_pixel=75, strong_pixel=255,
    lowthreshold=0.05, highthreshold=0.15
  )
  self.dbscan = DBSCAN(eps=7, min_pts=50)

  # Setup video properties
  cap = cv2.VideoCapture(video_path)
  self.frame_width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
  self.frame_height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
  self.total_frames = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))
  cap.release()

  # Initialize tracking variables
  self.scale_factor = 0.5
  self.new_width = int(self.frame_width * self.scale_factor)
  self.new_height = int(self.frame_height * self.scale_factor)
  self.previous_frame = None
  self.previous_centroids = None
  self.frame_count = 0
  self.object_speeds = {}

  self.colors = [ # A list of BGR colors for visualization
    (255, 0, 0),  # Blue
    (0, 255, 0),  # Green
    (0, 0, 255),  # Red
    (255, 255, 0),  # Cyan
    (255, 0, 255),  # Magenta
    (0, 255, 255),  # Yellow
    (128, 0, 0),  # Dark Blue
    (0, 128, 0),  # Dark Green
    (0, 0, 128),  # Dark Red
    (128, 128, 0)  # Dark Cyan
  ]

  print("Initialization complete!\n")
```

`CannyEdgeDetector` instance is initialized in straightforward manner, while parameters for `DBSCAN` were chosen after trial and error on test video. Following `cv2` features CAP_PROP_FRAME_WIDTH, CAP_PROP_FRAME_HEIGHT and CAP_PROP_FRAME_COUNT are used to get width, height and total number of frames in the video, respectively. Several tracking-related variables are initialized: the video is scaled down by 50% for performance (scale_factor = 0.5), and variables are set up to store the previous frame and object centroids for motion tracking. The object_speeds dictionary will store velocity information for tracked objects. Finally, color map for visualization is created in Blue-Green-Red format(no idea why this is done this way, visualization was done after prompting Claude AI by Anthropic).

```
def preprocess_frame(self, frame):
  frame_small = cv2.resize(frame, (self.new_width,self.new_height))
  frame_gray = cv2.cvtColor(frame_small, cv2.COLOR_BGR2GRAY)
        if len(frame_small.shape) == 3 else frame_small
  edges = self.edge_detector.edge_detect(frame_gray)
  return edges.astype(np.uint8)
```

`process_frame` function is resizing frame for speed, gray-scaling if necessary, applying `CannyEdgeDetector.edge_detect` and casting result as `np.uint8`.

```
def get_frame_difference(self, current_frame):
  if self.previous_frame is None:
    diff = np.zeros_like(current_frame)
  else:
    diff = cv2.absdiff(current_frame, self.previous_frame)
  _, diff = cv2.threshold(diff, 30, 255, cv2.THRESH_BINARY)
  return diff
```

This method helps identify regions in the video where movement has occurred between consecutive frames. The resulting binary difference image can then be used for further processing. Second to last line(since it might be the most confusing one) creates a binary image where white pixels (255) indicate areas with significant changes, black pixels (0) indicate areas with little or no change. The underscore ignores the first return value of cv2.threshold() which is the threshold value used.

```
def helper_2D_2_3D(self, diff_frame):
  y_coords, x_coords = np.nonzero(diff_frame)
  if len(y_coords) > 1000:
    indices = np.random.choice(len(y_coords), 1000, replace=False)
    y_coords = y_coords[indices]
    x_coords = x_coords[indices]
  return np.column_stack((x_coords, y_coords, diff_frame[y_coords,
                                  x_coords]))
```

`helper_2D_2_3D` essentially converts a 2D difference image into a more manageable set of 3D points for use with the `DBSCAN` and to be able to detect objects and speed in the first place. This format makes it easier to identify and group regions of motion in the video frame.

```
# Generated by Claude AI. I had no idea how to do what I asked for.
                                    Uncanny resemblance?
def create_visualization(self, frame, labels, points, speeds):
  vis_frame = cv2.cvtColor(frame, cv2.COLOR_GRAY2BGR)
  unique_clusters = np.unique(labels[labels >= 0])

  for cluster_id in unique_clusters:
    cluster_mask = (labels == cluster_id)
    cluster_points = points[cluster_mask]
    centroid = np.mean(cluster_points[:, :2], axis=0).astype(int)
    color = self.colors[cluster_id % len(self.colors)]

    for x, y, _ in cluster_points:
      cv2.circle(vis_frame, (int(x), int(y)), 1, color, -1)

    if cluster_id in speeds and speeds[cluster_id]:
      speed_text = f"{speeds[cluster_id][-1]:.1f} p/f"
      cv2.putText(vis_frame, speed_text, tuple(centroid),
          cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 255, 255), 2)
      cv2.putText(vis_frame, speed_text, tuple(centroid),
          cv2.FONT_HERSHEY_SIMPLEX, 0.5, color, 1)

  return vis_frame
```

As stated in the comment, this code was generated by Claude AI. It's purpose is final output video, which gives color to each cluster after edge detection of whole video and assign average speed to each cluster.

```
def _calculate_speeds(self, current_centroids):
  if not self.previous_centroids:
    return

  curr_cents = np.array(current_centroids)
  prev_cents = np.array(self.previous_centroids)
  distances = np.sqrt(((curr_cents[:, np.newaxis, :2]
              - prev_cents[np.newaxis, :, :2]) ** 2).sum(axis=2))

  for i, dists in enumerate(distances):
    closest_idx = np.argmin(dists)
    min_dist = dists[closest_idx]
    speed = min_dist / (self.skip_frames * self.scale_factor)
    if i not in self.object_speeds:
      self.object_speeds[i] = []
    self.object_speeds[i].append(speed)
```

Given current centroids, if previous centroids exist, `_calculate_speeds` calculates distances between centroids and determines the speed of each object by identifying the closest previous centroid. The method stores these speeds in a dictionary, allowing for tracking of speed over time for each object based on its centroid's movement. This functionality is crucial in applications such as object tracking in video analysis, where understanding the speed of moving objects can provide insights into their behavior and interactions.

```
def track_objects(self, visualize=True):
  object_counts, last_print = [], 0
  last_print = 0
  print(f"[{datetime.now().strftime('%H:%M:%S')}] Processing video
                                    ...")
  cap = cv2.VideoCapture(self.video_path)
  if visualize:
    fourcc = cv2.VideoWriter_fourcc(*'mp4v')
    out = cv2.VideoWriter('Fancy Video.mp4', fourcc,
        30.0 / self.skip_frames, (self.new_width, self.new_height))

  while True:
    ret, frame = cap.read()
    if not ret:
      break

    if self.frame_count - last_print >= 5:
      progress = (self.frame_count / self.total_frames) * 100
      print(f"\rProgress: {progress:.1f}% | "
        f"Frame: {self.frame_count}/{self.total_frames} | "
        f"Time: {datetime.now().strftime('%H:%M:%S')}", end="")
      last_print = self.frame_count

    if self.frame_count % self.skip_frames != 0:
      self.frame_count += 1
      continue
    edges = self.preprocess_frame(frame)
    frame_diff = self.get_frame_difference(edges)

    if np.sum(frame_diff) > 1000:
      points = self.helper2D_2_3D(frame_diff)
      if len(points) >= self.dbscan.min_pts:
        self.dbscan.dbscan(points)
        labels = self.dbscan.labels_
        centroids = self.dbscan.cluster_centers_
        n_objects = len(set(labels[labels >= 0]))
        if n_objects > 0:
          object_counts.append(n_objects)
          if self.previous_centroids is not None
              and len(centroids) > 0:
            self._calculate_speeds(centroids)
          if visualize:
            vis_frame = self.create_visualization(edges, labels,
                                  points, self.object_speeds)
            out.write(vis_frame)

        self.previous_centroids = centroids

    self.previous_frame = edges
    self.frame_count += 1

  if visualize:
    out.release()
  cap.release()
  print("\nProcessing complete!")
  return object_counts, self.object_speeds
```

This is the big guy. The `track_objects` method is designed to process a video file to track objects over time, with an option to visualize the tracking results. The method begins by initializing an empty list to keep track of the number of objects detected in each frame and sets a variable to manage the timing of progress updates.

The method prints a message indicating that video processing has started and attempts to open the video file specified by `self.video_path`. If visualization is enabled, it sets up a video writer to save the processed frames into "Fancy Video.mp4" with a specified frame rate and resolution.

The method enters a loop that continues until all frames of the video have been processed. Within this loop, it reads each frame from the video. If reading a frame fails, the loop breaks, since there are no more frames. The method checks if enough frames have been processed to update the progress display. If so, it calculates the percentage of frames processed and prints this information to the console.

If the current frame count does not match the specified skip_frames, the method increments the frame count and continues to the next iteration, effectively skipping frames. For processed frames, the method first preprocesses the frame to extract edges and then computes the difference between the current frame and the previous one.

If the sum of the frame difference exceeds a threshold, indicating significant movement, the method converts the frame difference into 3D points. If the number of points meets a minimum requirement, it applies the `DBSCAN` clustering algorithm to identify clusters of points, which correspond to detected objects. The method then counts the number of unique objects detected and appends this count to the `object_counts` list.

If there are previous centroids available and new centroids have been identified, the method calculates the speeds of the objects using the `_calculate_speeds` method. If visualization is enabled, it creates a visual representation of the current frame, including the detected objects and their speeds, and writes this frame to the output video.

After processing the frame, the method updates the previous centroids and increments the frame count. Once all frames have been processed, the method releases the video write and the video capture object, and prints a completion message. Finally, it returns the list of object counts and the recorded object speeds.

```
[23:17:42] Initializing Tracker...
Initialization complete!

[23:17:42] Processing video...
Progress: 97.4% | Frame: 190/195 | Time: 23:20:06
Processing complete!

Average objects detected: 1.6

Object speeds (pixels/frame):
Object 0: 14.2 p/f
Object 1: 32.6 p/f
Object 2: 51.1 p/f
Object 3: 38.0 p/f
Object 4: 86.3 p/f
```

Figure 10: Test Result in Terminal

```
def analyze_video(self, visualize=True):
  object_counts, speeds = self.track_objects(visualize)
  avg_count = np.mean(object_counts) if object_counts else 0
  avg_speeds = {obj_id: np.mean(speeds) for obj_id, speeds in self.
                                    object_speeds.items()}
  return avg_count, avg_speeds
```

This final method called `analyze_video` is pretty self-explanatory, so I won't explain it. You can also see test case code I used:

```
if __name__ == "__main__":
  PATH = "test_60f_854x480.mp4"
  tracker = ObjectTracker(PATH, skip_frames=5)
  avg_count, avg_speeds = tracker.analyze_video(visualize=True)

  print(f"\nAverage objects detected: {avg_count:.1f}")
  print("\nObject speeds (pixels/frame):")
  for obj_id, speed in avg_speeds.items():
    print(f"Object {obj_id}: {speed:.1f} p/f")
```

I tested for videos taken by front camera of Google's Pixel 7. Its original resolution is $854 \times 480$, while its original frame rate is 60 frames per second. My test video was 3.25 seconds long. In the test case, $\text{test}_60\text{f}_854x480\text{.mp4}$ refers to the 60 frames per second and $854 \times 480$ resolution. I also tried 3 other combinations with 30 frames per second and $420 \times 236$ resolution. Accuracy seemed to somewhat decrease for lower resolutions. Here is an image screenshot from final video(keep in mind, this was with `epsilon` $= 7$, `min_pts` $= 50$ and several other parameters set to specific values), where Oscar toy is colored in red, bottle I am moving is colored in green and I am colored in blue:


Figure 11: Single frame from final video

Obviously, more frames and higher resolution requires more computation. A 3.25 second video at 60 frames per second and a resolution of $854 \times 480$ took $\approx 150$ seconds to complete, while a video less than 1 second long video took $\approx 11$ seconds.

(a) Original frame

(b) Frame after object detection

Figure 12: When the code does not work as expected

While the code works when object are easily separable, and angle of the camera is barely changing, it has trouble identifying objects that are close to each other. On the example image, there are actually at least 5 grapes, but code recognized a single grape object and pattern on cloth of the table. Granted, this image would be extremely troublesome even for the bleeding-edge object detection algorithms, but nevertheless, it is a case where the code fails!

I did not include the import of libraries in the snippets, since most editors will automatically detect the dependencies and suggest their import. I also need to give credit to OpenAI's ChatGPT and Anthropic's ClaudeAI, which I used every time I messed up indexing, axes, or some `numpy` operation with which I was not really familiar, as well as `cv2`, which was completely new to me. Overall, this project helped me learn a great deal about K-d trees, speeding up algorithms with specific data structures, and object detection in general!