**暨南大学本科实验报告专用纸**

课程名称 数字图像处理实验 成绩评定

实验项目名称 基于YOLO和颜色识别的多目标追踪方法 指导教师 汤知日 实验项目编号 实验项目类型 实验地点 实B304 学生姓名 陈金、邱帅钧、陈峻锋、刘义源 学号 2022101927、2022101928、2022101926、2022101924 学院 智能科学与工程学院 专业 物联网工程 实验时间 2024 年 12 月 30 日

**实验报告写作要求和评分标准**

1. **使用教务处统一要求的“本科实验报告专用纸”（含首页和附页）；**
2. **正文由四部分构成：实验目的、实验仪器、实验内容（含实验原理20分）、实验总结（分析实验数据20分，得出实验结论20分）；**
3. **格式规范10分，全文字体使用黑色，宋体，四个部分的标题用四号字加粗，其他内容字体为小四号，不加粗，1.5倍行距，页码底部居中，实验报告文件名格式“xx实验（序号）姓名学号”，序号用汉字数字，一、二、三、四等；**
4. **电路图可以从实验指导书截图复制，数据记录表格需要自行设计；**
5. **图表整齐（10分），内容表述有条理（10分），数据分析准确，结论可信有说服力。**
6. **波形图片压缩，每份实验报告word文档占存储空间不超过1MB。**

**实验目的**

YOLO识别算法运用深度学习，可以进行多目标的检测和跟踪，但是它往往需要占用大量的计算机资源，在一些内存和浮点数算力受限大的平台上检测速度和稳定性较差；我们考虑通过颜色追踪算法mean-Shift和图像特征点识别算法ORB弥补这2种问题。

**实验原理**

1. **YOLO检测算法**
2. 基本原理

YOLO识别算法是一种目标检测算法，用于解决计算机视觉中的物体检测问题。与传统的目标检测方法不同，YOLO 通过单次前向传播就能完成对图像中所有目标的识别和定位，具有极高的检测速度和较好的检测精度。YOLO的核心思想是将目标检测问题转化为一个回归问题，即将图像分成一个网格，每个网格负责预测目标的类别和边界框，整个过程在一次神经网络的前向传播中完成，因此 YOLO 可以同时预测多个目标的类别和位置。

1. 工作流程
2. 划分网格

YOLO将输入图像划分为一个 S × S 的网格。网格的大小 S 是超参数，通常选择 7 或 13。每个网格负责检测图像中位于该区域内的物体并预测物体的类别和边界框。

1. 网格单元的预测

对于每个网格单元，YOLO进行以下的预测：

① 中心点坐标（x, y）：表示边界框的中心点相对于网格单元的偏移量，通常使用相对于网格单元的坐标进行归一化。

② 宽度和高度（w, h）：表示边界框的宽度和高度，通常使用相对于图像的宽度和高度进行归一化。

③ 置信度：表示该边界框包含物体的概率以及框的预测和真实框的重叠程度。

④ 类别概率：每个网格单元还会预测每个类别的概率。假设有 C 个类别，那么每个网格单元输出一个长度为 C 的向量，表示该网格中物体属于每个类别的概率，而最终的类别预测是通过对每个边界框的置信度与该类别的概率进行乘积得到的。

1. 输出结果

YOLO 的输出是一个维度为 S × S × (B × 5 + C) 的张量。对于每个网格单元，YOLO 输出以下信息：

1. 每个边界框的 6/7个值：x, y, w, h, [Id,] Confidence, Class。

② 每个类别的概率：一个长度为 C 的向量。

1. 目标检测后的处理

YOLO 在预测之后，使用非极大值抑制（NMS）来去除冗余框，并得到最终的检测结果。

1. **ORB特征识别算法**
2. 基本原理

ORB识别算法是一种高效的计算机视觉算法，广泛用于图像中的特征点检测与描述，结合了 FAST角点检测算法和 BRIEF特征描述符，并在此基础上增加了对旋转不变性的支持。ORB算法的核心思想是通过快速且高效的角点检测、特征描述符生成，并结合旋转不变性，使其能够在各种应用中提供良好的性能，尤其适用于实时处理。

1. 工作流程
2. 特征点检测

ORB使用FAST算法进行特征点检测。FAST是一种快速的角点检测算法，适用于实时计算。在FAST算法中，对于图像中的每个像素点，都会考虑其周围的一些邻域像素点，通常是以该像素为中心的圆形区域。如果该中心像素点与邻域像素点的灰度值差异很大，就认为该像素点是一个角点。

1. 主方向计算

为了让ORB能够对旋转有不变性，ORB算法会为每个角点计算一个主方向。对于每个检测到的角点，ORB首先计算该点周围一个小区域内所有像素的梯度方向。根据角点周围梯度方向的直方图，ORB通常会选择直方图中出现频率最高的梯度方向作为角点的主方向。根据这个主方向，ORB对后续的特征描述符进行旋转调整，从而确保ORB的特征描述符对旋转具有不变性。

1. 特征描述符生成

ORB算法使用 BRIEF描述符来表示每个角点的特征。BRIEF是一种二进制描述符，它通过比较像素对的亮度来生成二进制位，表示图像局部区域的特征。BRIEF的生成过程如下。

1. 随机选择像素对：对于每个角点，ORB会随机选择图像中的像素对，通常为角点周围某个固定大小的区域中的像素对。
2. 比较像素亮度：对于选择的像素对，ORB会比较它们的亮度值，如果左边的像素亮度大于右边的像素亮度，则该位为1，否则为0。
3. 构建描述符：通过这种方式，每个角点会得到一个由若干二进制位构成，长度一般为256位或512位的描述符。
4. BRIEF改进

ORB对BRIEF进行改进，使得BRIEF描述符对旋转具有不变性。ORB通过角点的主方向来旋转BRIEF描述符，使得描述符能够适应旋转变换。旋转后的BRIEF描述符被对齐到主方向上，这样即使图像发生旋转，ORB依然能保持一致的描述符。

1. 特征点匹配

在ORB中，特征点匹配是通过比对特征描述符的二进制位来完成的。由于描述符是二进制的，因此比对过程非常高效。常见的比对方法为汉明距离法，即计算两个二进制描述符之间的异或操作结果的1的个数，1的个数越少，两个特征点的相似度越高。比对完成后，通过最近邻匹配策略来选择最匹配的特征点对，同时使用比值测试法来避免错误匹配。

1. **Mean Shift颜色识别算法**
2. 基本原理

Mean Shift识别算法是一种基于非参数化密度估计的迭代优化算法，广泛应用于图像分割、目标跟踪、密度估计等任务。其基本原理是通过寻找数据分布的局部最大值，并将样本点向该局部最大值“偏移”，从而在无监督的情况下进行聚类，实现目标定位或区域分割。该算法在面对图像中的部分遮挡、复杂背景干扰或动态变化时，仍能有效地追踪目标。

1. 工作流程
2. 数据空间中的密度估计

Mean Shift通过估计数据空间中的概率密度函数（PDF）来找到数据的“高密度区域”，最常见的方法为使用核密度估计来来近似估计数据的概率分布。

在二维数据的情况下，设定一个核函数K，如高斯核函数，然后给定一个数据点x，Mean Shift 通过将 x 周围点进行加权平均，更新x的位置，使其逐渐移向数据的密度峰值。

其中，是数据集中的其他点，是核函数，是新的质心（更新后的x的位置）。该过程会持续进行，直到算法收敛，即质心不再发生显著变化。

1. 核心更新过程

Mean Shift的核心更新过程通过计算每个点周围的均值来寻找。

其中，m(x)表示Mean Shift向量，表示当前点x正在朝哪个方向移动。

1. 迭代过程

对于每一个点x，使用上述的 Mean Shift 向量进行迭代更新。每次迭代后，点x会被移向数据密度的高峰区域。迭代过程会一直进行，直到点的移动距离小于设定的阈值，或者达到最大迭代次数。

1. 收敛和聚类

算法收敛后，所有点不再显著更新，将会集中到一组高密度区域，这些区域对应着数据集中的不同簇。每个簇的中心就是一个局部密度的峰值点，称为模式，这些模式可以作为聚类的代表。

1. **系统实现**

YOLO首先检测到画面中的物体，输出物体的位置框、类别；在一定条件下，有时还可以记忆并追踪识别到的物体，输出唯一的临时ID号。

接着使用这些信息标定物体区域，作为模板分别初始化一个ORB特征识别器和一个mean-Shift颜色追踪器，使它们绑定该物体的ID号，实验证明这两种算法的运算速度相对较快，初始化速度快。

对于ORB特征识别算法的更新，由于它使用了角落点检测方法和KNN比较方法，可以有效识别目标遮挡和三维方向信息；但是也导致了其识别速度较慢，而且当目标在高速运动时，由于图像拖影模糊使角落点检测出错，可能导致运算量增大和跟踪失败，于是我们设置一个关心区域，使其略大于上一次识别到目标的区域，当ORB算法只在这个区域检测目标时，运算量和误报率将大大降低。

对于mean-Shift算法的更新，由于它只使用了模板和图像的颜色相关信息，识别速度和鲁棒性极高，在目标高速移动时也能获得很好的效果，当ORB算法跟踪失败时，根据分配的唯一ID号就可以找到mean-Shift输出的位置更新关心区域。但是，mean-Shift使用的颜色信息不方便检测目标图像投影区域大小变化，在图像中出现多个与目标颜色相同的物体时，可能跟踪错误，因此它需要使用ORB算法定时矫正。

当YOLO再次识别物体时，它可能由于可信度变化没有识别出标记过的物体，或把标记过的物体识别为另一个物体，此时我们使用ORB算法和目标的位置区域信息自动矫正这些错误，提升跟踪稳定性。

**实验内容**

伪代码：

1. ORB特征识别算法的实现

首先定义一个特征匹配器类方便对多目标操作：

class FeatureStruct:

    def \_\_init\_\_(self, dst\_image, min\_mum=10, low\_rate=0.7, max\_error=5.0,

                 confidence=0.98, search\_checks=50, area\_rate=0.1, method='SIFT'):

这段代码初始化 FeatureStruct 类的实例。FeatureStruct 类用图像特征提取（如 ORB 或 SIFT）进行图像特征匹配和跟踪。

初始化FlannBasedMatcher匹配器：

            FLANN\_INDEX\_LSH = 6

            index\_params = dict(algorithm=FLANN\_INDEX\_LSH,

                                table\_number=12,  # 16

                                key\_size=20,  # 12

                                multi\_probe\_level=2)  # 1

FlannBasedMatcher是OpenCV提供的一个高效的特征点匹配器，用于在多个图像之间进行描述符匹配。FLANN（Fast Library for Approximate Nearest Neighbors）是一个近似最近邻查找库，index\_params指定了匹配的算法、表格数量等参数。

目标图像模板初始化：

    def set\_dst(self, dst\_image):

        if dst\_image.ndim == 3:

            dst\_image = cv2.cvtColor(dst\_image, cv2.COLOR\_BGR2GRAY)

        kp1, desc1 = self.orb.detectAndCompute(dst\_image, None)

        if desc1 is None:

            raise ValueError('Can not detect dst image.')

        self.dst\_image = dst\_image

        self.kp1, self.desc1 = kp1, desc1

在set\_dst方法中，将目标图像转换为灰度图像，并使用ORB检测特征点（kp1）和描述符（desc1）。ORB特征点和描述符是后续匹配的基础。

获取目标图像当前的ROI区域：

    def get\_roi(self, size):

        corners = self.corners[:, 0, :]

        xyxy = (\*np.maximum(corners.min(axis=0) - corners[4] + self.pos, 0),

                \*np.minimum(corners.max(axis=0) - corners[4] + self.pos, size))

        return tuple(map(int, transform\_roi(xyxy)))

get\_roi方法根据当前的物体位置（pos）和角点位置（corners）计算ROI的边界框。ROI的大小是根据角点和位置来确定的，transform\_roi(xyxy)是对边界框进行坐标转换以适应图像尺寸。

ORB特征匹配和更新：

    def \_\_call\_\_(self, image):

        if image.ndim == 3:

            image = cv2.cvtColor(image, cv2.COLOR\_BGR2GRAY)

        # 找到两幅图像的特征

        self.kp2, self.desc2 = self.orb.detectAndCompute(image, None)

        if self.desc2 is None:

            # print('Can not detect.')

            return None, None, None

        # 特征匹配

        matches = self.matcher.knnMatch(self.desc1, self.desc2, k=2)

在更新函数中，输入图像通过ORB进行特征提取，得到特征点（kp2）和描述符（desc2）。然后对当前图像与目标图像的描述符进行KNN匹配，得到匹配结果。

选择匹配点：

        good = []

        if self.method == 'ORB':

            for ms in matches:

                if not ms: continue     # ORB don't hand in one or more values in some case, special treatment.

                elif len(ms) == 1:

                    good.append(ms[0])

                elif len(ms) == 2 and ms[0].distance < self.low\_rate\*ms[1].distance:

                    good.append(ms[0])

在匹配结果中，进行筛选，只保留质量较好的匹配点。这里采用的是低于距离比的策略，即选择较优的匹配对。匹配点会根据距离的比值进行过滤，low\_rate控制了匹配的严格度。

更新物体的角点坐标：

        self.matrix, mask = cv2.findHomography(src\_pts, dst\_pts, cv2.RANSAC, self.max\_error, maxIters=500, confidence=self.confidence) # 计算变换矩阵（透视变换）

        self.corners = corners = cv2.perspectiveTransform(self.corner, self.matrix) # 透视变换与ROI更新

使用RANSAC算法（随机采样一致性算法）来计算源图像和目标图像之间的透视变换矩阵，mask指定了哪些点是内点（即匹配的准确点）。然后将目标图像的角点位置进行透视变换，得到在当前图像中物体的新位置。

判断物体是否有效：

        if nc[4, 0] <= 0 or nc[4, 1] <= 0:

            pass

        elif nc[4, 0] > w or nc[4, 1] > h: # 物体中心位置超出图像边界

            pass

        elif (nc[:4, 0] < -w/2).any() or (nc[:4, 1] < -h/2).any():

            pass

        elif (nc[:4, 0] > w+w/2).any() or (nc[:4, 1] > h+h/2).any(): # 物体角点变形超长

            pass

        elif triangle\_area(nc[0], nc[1], nc[2])+triangle\_area(nc[0], nc[2], nc[3]) <= h\*w\*self.area\_rate: # 物体的ROI面积小于限制比例

            pass

        elif not np.isclose(triangle\_rote(nc[0], nc[1], nc[2])+triangle\_rote(nc[1], nc[2], nc[3])+triangle\_rote(nc[2], nc[3], nc[0])+triangle\_rote(nc[3], nc[0], nc[1]), 2\*np.pi, 0, 1e-1): # 物体不是凸四边形

            pass

绘制匹配结果（如果可能）：

    def draw(self, image, Mask, good, corners):

        if corners is not None: # 绘制角点和ROI的边界框

            # 应用透视变换

            nc = corners[:, 0, :]

            h, w = image.shape[:2]

            for i, p in enumerate(nc):

                cv2.circle(image, (int(p[0]), int(p[1])), 5+2\*i, (255,255,0), 3)

            if good:

                print(f'w:{nc[4,0]/w:.2f}',

                      f'h:{nc[4,1]/h:.2f}',

                      f'area: {triangle\_area(nc[0], nc[1], nc[2]):.0f}+{triangle\_area(nc[0], nc[2], nc[3]):.0f}')

            # Draw a rect when matched out it

            image = cv2.polylines(image, [np.int32(corners[:4])], True, 255, 3, cv2.LINE\_AA)

        elif good is not None:

            print("Not enough matches are found - {}/{}".format(len(good), self.min\_mum))

... # 绘制透视变换后的角点和匹配线条

        return image

接着定义一个字典方便存储不同目标的ORB检测器，通过分配的唯一ID号进行查询：

orbs = {}

模板初始化和掩码处理：

    for Id, roi in zip(ids, rois):

        center = getROICenter(roi)

        mask.fill(0)

        mask0=cutROI(mask, roi)

        mask0[:]=255

从YOLO中得到目标ID和目标ROI区域。对于每个目标，首先计算其ROI区域的中心位置，然后创建一个与ROI大小相同的掩码。

然后查询已有ORB跟踪器：

        for ID, orb in orbs.items():

            if similarROI(roi, orb.get\_roi(size)) > similar0:

                orb.set\_dst(gray\_roi)

                orb.pos = center

                ids[ids.index(Id)] = ID

                Id = ID

                break

遍历已存在的ORB实例（orbs字典），通过当前ROI与ORB实例的ROI相似度判断目标是否一样，更新ORB实例的目标灰度图特征和位置，并纠正ID。

创建新ORB实例：

        else: # 没能够通过相似区域找到匹配的ORB实例

            if Id in orbs: # 如果目标ID已存在于orbs中，直接更新模板

                orbs[Id].set\_dst(gray\_roi)

                orbs[Id].pos = center

            else:

                if Id == -1: # YOLO没有输出ID号，则随机获取一个

                    while 1:

                        Id = random.randint(0, 100)

                        if Id not in orbs: break

                orb = FeatureStruct(gray\_roi, method='ORB')

                orb.pos = center

                orbs[Id] = orb

更新所有的ORB检测器：

    for Id, orb in orbs.items():

        roi\_pad = padROI(orb.get\_roi(size), pads, size)

        gray\_roi = cutROI(gray, roi\_pad) # 只让ORB在较小的区域内识别，降低失误、提升计算速度

        try:

            Mask, good, corner = orb(gray\_roi)

            if Mask: # 识别成功，更新中心位置并绘制角点

                orb.pos = ((orb.pos[0]-roi\_pad[2]/2, orb.pos[1]-roi\_pad[3]/2) + corner[4, 0]).astype(np.int16)

                orb.draw(cutROI(c, roi\_pad), None, None, corner)

            elif Id in shifts: # 识别失败，从mean-Shift追踪器获取备用信息

                shift = shifts[Id]

                orb.pos = getROICenter(shift.box)

        except cv2.error:

            pass

展示结果：

     cv2.imshow('det', c)

1. Mean Shift颜色追踪算法的实现

首先定义一个颜色追踪器类方便对多目标操作：

class ShiftStruct:  
 def \_\_init\_\_(self, init\_frame=None, box=None, colors=None, pad=100, counts=10, mode='Cam'):

对参数配置进行初始化，包括第一帧、跟踪框的初始位置、跟踪的颜色范围、初始化时的边距、最大迭代次数以及跟踪模式。

初始化跟踪区域并计算均衡直方图：

def init\_frame(self, frame, box=None, colors=None):   
 # set up the ROI for tracking  
 roi = frame[box[1]:box[1] + box[3], box[0]:box[0] + box[2]]  
 hsv\_roi = cv2.cvtColor(roi, cv2.COLOR\_BGR2HSV)  
 mask = cv2.inRange(hsv\_roi, colors[0], colors[1])  
 self.roi\_hist = cv2.calcHist([hsv\_roi], [0], mask, [180], [0, 180])  
 cv2.normalize(self.roi\_hist, self.roi\_hist, 0, 255, cv2.NORM\_MINMAX)

初始化目标跟踪所需的所有信息，并进行直方图的计算。首先，从提供的frame中根据给定的box提取目标区域ROI，然后将ROI从BGR转换为HSV色彩空间，接着生成掩膜以筛选出目标颜色。通过cv2.calcHist计算目标区域的颜色直方图，并进行归一化处理。这个直方图用于后续的MeanShift跟踪。

mean-Shift跟踪更新函数：

def \_\_call\_\_(self, frame, box=None, colors=None):   
 hsv = cv2.cvtColor(frame, cv2.COLOR\_BGR2HSV)  
 if self.src.shape != frame.shape[:2]:  
 self.src = np.zeros\_like(frame, shape=frame.shape[:2])  
 cv2.calcBackProject([hsv], [0], self.roi\_hist, [0, 180], 1, self.src)   
 ret, box = self.model(self.src, self.box, self.term\_crit)

接着定义一个字典方便存储不同目标的mean-Shift追踪器，通过分配的唯一ID号进行查询：

shifts = {}

模板初始化和掩码处理：

for Id, roi in zip(ids, rois):  
 center = getROICenter(roi) # 计算其ROI区域的中心位置  
 mask.fill(0)  
 mask0=cutROI(mask, roi) # 创建掩码  
 mask0[:]=255  
 mask\_roi = cutROI(mask, roi\_pad)  
 colors = ShiftStruct.scan\_inRange(hsv, mask, 0.9) # 优势颜色查找

然后查询已有mean-Shift颜色跟踪器：

for ID, shift in shifts.items():  
 if similarROI(roi, shift.box) > similar0:  
 shift.init\_frame(c, roi, colors)  
 ids[ids.index(Id)] = ID  
 Id = ID  
 break

如果目标框与现有的ShiftStruct对象匹配，则使用init\_frame函数来初始化或更新该目标的 MeanShift 跟踪器。

else:  
 if Id in shifts:  
 shifts[Id].init\_frame(c, roi, colors)  
 else:  
 if Id == -1:  
 while 1:  
 Id = random.randint(0, 100)  
 if Id not in shifts: break  
 shift = ShiftStruct(c, roi, colors, mode='mean')  
 shifts[Id] = shift

反之，目标框没有与现有的颜色追踪器匹配，则创建新的ShiftStruct对象，并用mean-Shift模式进行初始化。

接着进行颜色追踪与更新：

mask.fill(0)

for Id, shift in shifts.items():  
 box = shift.box  
 shift(c) # 进行追踪  
 if np.sqrt((box[0]-shift.box[0])\*\*2+(box[1]-shift.box[1])\*\*2)>pads[0]:  
 if Id in orbs:  
 orb = orbs[Id]  
 shift.box = (max(0,int(orb.pos[0] - box[2]/2)), max(0,int(orb.pos[1]-box[3]/2)), box[2], box[3])  
 mask |= shift.src # 展示反相投影结果，能反映优势颜色的分布

如果目标框位置发生了较大的变化，超过预设值，认为目标丢失，则通过orb对象更新目标框的位置。

xyxy\_hsv = transform\_xyxy(shift.box)

cv2.retangle(c, xyxy\_hsv[:2], xyxy\_hsv[2:], (0, 0, 255), 2)

cv2.putText(c, 'id=%d' % Id, xyxy\_hsv[:2], cv2.FONT\_HERSHEY\_SIMPLEX, 0.75, (Id\*2, Id\*2, 200-2\*Id), 2)

最后将跟踪更新后的目标框、ID号绘制在当前帧上。

1. YOLO物体识别算法的实现

定义了CAPTUREStruct类，用于视频识别捕捉的初始化设置：

class CAPTUREStruct:  
 def \_\_init\_\_(self, path: str or int or tuple or cv2.VideoCapture,  
 size: tuple = None, fps: int = None, fourcc: str = None):

初始化参数包含视频源的路径，指定帧的宽度和高度的元组、视频的帧率及其编解码器。

再定义一个VIDEOStruct的类，用于最终结果的视频文件的创建和写入：

class VIDEOStruct:  
 def \_\_init\_\_(self, path, fourcc: str, fps: int, size: tuple):  
 assert fourcc in map(lambda x: x[0].lower(), self.FOURCC)  
 assert path.isascii()  
 self.video = cv2.VideoWriter()  
 self.video.open(path, (cv2.VideoWriter.fourcc  
 if not hasattr(cv2, 'VideoWriter\_fourcc')

else cv2.VideoWriter\_fourcc)(\*fourcc),fps, size, True)

初始化方法接收四个参数，包含视频文件的保持路径、视频编解码器的fourcc代码、视频的帧率、视频帧的尺寸。创建一个cv2.VideoWriter对象，将视频帧以彩色模式写入文件，最后获取fourcc代码并使用提供的参数打开视频文件。

定义YOLO类，方便使用。

class YOLOv8(YOLO):  
 def \_\_init\_\_(self, model="./yolov8n.pt", \*\*kwargs):   
 self.objs = []  
 self.results = None

声明调用函数接口，用于进行对象识别：

def \_\_call\_\_(self, frame):  
 # YOLO 跟踪  
 self.results = self.track(frame, persist=True, tracker="bytetrack.yaml", max\_det=8, verbose=self.verbose)[0]  
 self.objs.clear()  
 for detection in self.results.boxes.data:  
 if len(detection) < 7:  
 confidence, cls = detection[4:6]  
 obj\_id = -1  
 else:  
 # 获取检测信息，含ID  
 obj\_id, confidence, cls = detection[4:7]

输入列表和检测的帧，返回检测对象的信息。对输入帧持续跟踪，每帧最多检测8个对象，同时清空objs列表，以储存新的检测对象。

之后选取检测图像：

obj\_id = int(obj\_id) # 转换ID为整数  
 cls = int(cls) # integer class  
 self.objs.append((obj\_id, cls, detection[:4]))  
 return self.objs

遍历列表中的每个结果。对于包含ID号的对象，获取其信息，包含ID、类、置信度；对于不包含ID号的对象，提取其置信度和类，将其ID设置为-1，并将它们转为整数，添加到列表中。

定义绘制方法：

def draw(self, image):  
 if self.results:  
 image = self.results.plot(img=image, conf=True, ) #展示对象ROI区域、ID、名称、置信度  
 return image

首先实例化使用的类并初始化

# 设置低通滤波的内核大小  
gauss\_brush\_size = 7  
gauss\_brush\_low = 0.25  
gauss\_brush\_sigma = np.sqrt(- (gauss\_brush\_size-1)\*\*2 / 8 / np.log(gauss\_brush\_low))

out\_put: VIDEOStruct= None # 输出视频  
size = (640, 480) # 输入、输出视频大小  
clahe = cv2.createCLAHE(2.0, (int(size[0]/size[1]\*6), 6)) # 自适应直方图均衡化对象

images = iter(img\_iter(cam\_path, 'blue-')) # 输入视频  
similar0 = 0.6 # 遮罩相似度置信阈值  
pads = (50, 40) # 最大追踪许可范围  
mask = np.zeros(size[::-1], dtype=np.uint8) # 遮罩  
yolo = YOLOv8(verbose=True) # YOLO识别器

接着处理输入图像并迭代：

c = next(images)

cv2.GaussianBlur(c, (gauss\_brush\_size,gauss\_brush\_size), gauss\_brush\_sigma, c)

hsv = cv2.cvtColor(c, cv2.COLOR\_BGR2HSV)

gray = hsv[:,:,2] = clahe.apply(hsv[:,:,2], )

cv2.cvtColor(hsv, cv2.COLOR\_HSV2BGR, c)

定义变量接收输入图像，高斯滤波去除噪声，对图像进行颜色空间转换和直方图均衡化处理，设置参数，将处理后的HSV图像转换回BGR空间。

调用YOLO获取画面中物体的ROI区域和ID号信息：

objs = yolo(c)

ids = [obj[0] for obj in objs]  
 rois = [transform\_roi(obj[2].cpu().numpy().astype(np.uint16)) for obj in objs]  
 print(ids, rois)

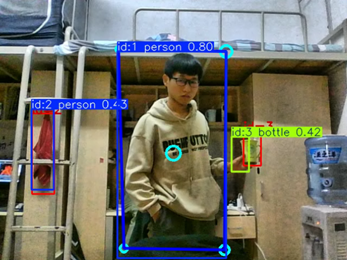
...

模拟YOLO识别器卡顿、丢失的情况：

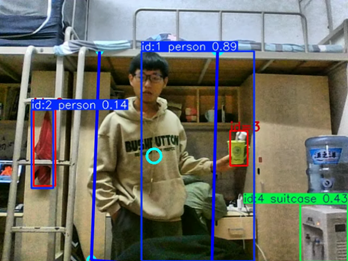
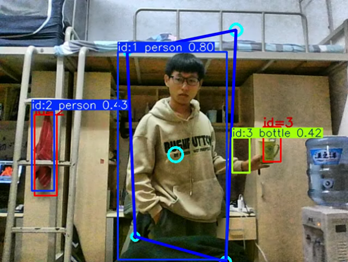
for i in range(60): # simulate YOLO as lower speed  
 c = next(images)  
 ...  
 c = yolo.draw(c) # 绘制YOLO识别的物体信息  
 cv2.imshow('det', c) # 展示跟踪效果  
 out\_put.write(c) # 保存输出结果

**实验结果和讨论**

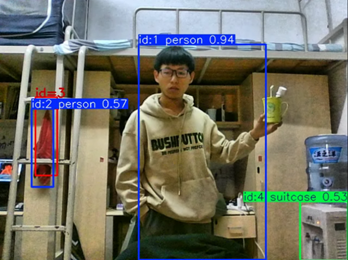
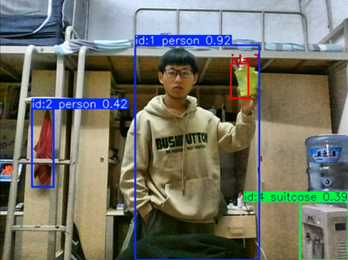
输出结果（带物体名称的是YOLO标签，带角点的是ORB识别结果，带ID号的红色方框是mean-Shift追踪结果）：



图像 1



图像 2



图像 3

**实验总结**

由处理结果可以看出：

1. （图一 左右）当目标大小较大且运动速度较低时，ORB特征检测能获得较好的效果。
2. （图二 左右）当目标变形较小时，ORB检测比较稳定；当目标连续移动、失去YOLO标签时mean-Shift也能很好地跟踪。
3. （图三 左）mean-Shift颜色追踪器对灰度不敏感，不能锁定白色饮水机（id=4）。
4. （图四 右）可能目前的程序对目标被遮挡的处理不好，当运动时，水杯（id=3）和毛巾（id=2）区域重叠时，id=3的颜色追踪器锁定了错误的颜色范围，导致跟踪失误。

目前的不足和发展建议

1. mean-Shift颜色追踪算法仅利用了目标的色相信息，在目标为白色、黑色或灰色时不能识别。
2. 可以进一步利用ORB特征识别算法识别目标遮挡信息和三维方向信息。

**参考文献**

1. Zhou H Y, Yuan Y, Shi C M. Object tracking using SIFT features and mean shift[J]. Computer Vision and Image Understanding, 2009, 345-352.
2. Yuntao C, Bin W, Guangzhi L, et al. Multi-target tracking algorithm based on YOLO+DeepSORT [J]. Journal of Physics: Conference Series, 2022, 2414 (1)
3. Zhao Y J, Cui J. Tracking of Traffic Monitoring Targets in Complicated Traffic Scene Based on MeanShift Algorithm [J]. Applied Mechanics and Materials, 2015, 3843 (744-746): 2012-2018.

**小组成员及分工表**

|  |  |  |  |
| --- | --- | --- | --- |
| 小组成员 | 姓名 | 学号 | 负责部分 |
| 组长 | 陈金 | 2022101927 | 系统联调 |
| 组员 | 刘义源 | 2022101924 | YOLO识别算法的实现 |
| 陈峻锋 | 2022101926 | ORB特征检测算法的实现 |
| 邱帅钧 | 2022101928 | mean Shift颜色追踪算法的实现 |