

## 理论部分

在 RoboMaster 比赛中，机器人需要精准的命中敌方机器人身上搭载的四块装甲板才能造成有效伤害，这就需要我们精准的获取目标装甲板的位置用于计算发射时的云台姿态。在比赛中，有两种不同的装甲模块，大型装甲板（230mm\*127mm）和小型装甲板（135mm\*125mm）。它们都贴有数字标签（从 1 到 5）以及一些特殊图案，并且在装甲模块的边缘有两个灯条，灯条的灯光可以是红色或蓝色。当己方为红队时，机器人上的灯光将是红色的，目标机器人上的灯光将是蓝色的。同时装甲模块上的数字标签用于区分不同的机器人。不同的机器人在装甲模块上有不同的数字标签。



图 2.1 不同类型的装甲板模块

同时由于比赛中发射的是塑胶弹丸，弹速一般为 10~15m/s，会在出膛后有一定的飞行时间，因此估计出目标的速度进行目标的预测是非常有必要的。

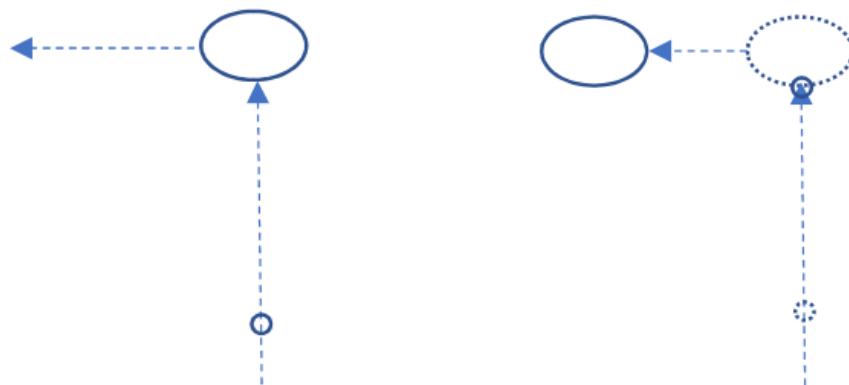


图 2.2 没有进行预测时的射击效果

如图 2.2 所示，在左图状态下发射一枚弹丸，当弹丸到达目标在发射时的位置时，目标已经离开了瞄准的位置。

总的来说，想要在比赛中实现良好的自动瞄准效果，我们需要：

- (1) 在图像中实时的识别目标并精准的获取目标相对己方机器人的三维位置；
- (2) 较为精确的估计目标的运动状态以获得实时的预测瞄准位置。

## 三维重建算法

由于我们需要获取目标相对己方机器人的三维位置才可计算出弹丸发射时的云台姿态，因此我们需要使用三维重建算法。

三维重建方法有很多种，包括但不限于：

- (1) 基于 RGB-D 深度相机<sup>[8]</sup>或融合相机及激光雷达<sup>[9]</sup>直接获取深度信息计算对应三维位置；
- (2) 基于深度学习的深度估计<sup>[10]</sup>；
- (3) 通过单目 PnP 从单张图像中恢复物体的三维位置姿态<sup>[6]</sup>。

方法（1）可以直接通过硬件实现获取深度信息，因此在合适的距离下的距

离精度应当是最高的，但受制于目前市面上的深度相机及激光雷达 FOV 较大，在远距离情况下得到的深度图较为稀疏，取装甲板大小的深度信息抖动较大，并且由于相机 FOV 较大，识别器的有效识别距离不足，以 Intel Realsense D455 深度相机为例，最远识别距离仅为 3m，无法满足比赛的需求。方法（2）分为绝对深度估计及相对深度估计，在该任务场景下我们需要绝对深度估计，而由于绝对深度估计泛化能力极差，并且由于网络推理延时较高，也不满足自动瞄准任务的需求。方法（3）的劣势主要为需要知晓目标三维角点的先验信息，但在 RoboMaster 比赛中目标都是固定且可知的，并且方法（3）只需要单目 RGB 图像即可获取目标的位置姿态，而一般工业相机都可以提供高帧率的 RGB 图像，还可根据不同机器人识别距离的需求更换不同焦距的镜头，故笔者选用方法（3）作为自动瞄准系统的三维重建算法。

## 目标识别算法

由 2.2.1 中可知，我们需要获得图像中的目标对应角点的信息，因此需要选用合适的目标识别算法。由 1.2 可知，目前在 RoboMaster 比赛中应用的识别算法主要分为传统方法识别灯条结合机器学习进行图像分类以及使用神经网络直接回归装甲板角点。传统方法一般为对图像二值化中找寻符合装甲板灯条特征的轮廓，再对候选灯条进行匹配。图像分类任务与常见的数字分类任务比较接近，数字分类方法一般有：

- （1）逻辑回归<sup>[11]</sup> (Logistic Regression)，用于预测二元结果，如是/否，通过/失败等。
- （2）朴素贝叶斯<sup>[12]</sup> (Naive Bayes)，用于计算一个数据点属于某个类别的概率，基于特征之间的独立性假设。

- (3) K-最近邻<sup>[13]</sup> (K-Nearest Neighbors), 用于根据训练数据集中最接近的 k 个数据点来对新数据点进行分类。
- (4) 决策树<sup>[14]</sup> (Decision Tree), 用于通过不断将数据集分割为更小的子集来构建分类或回归模型。
- (5) 支持向量机<sup>[15]</sup> (Support Vector Machines), 用于通过寻找能够最大化两个类别之间边界距离的超平面来进行分类。
- (6) 多层感知机<sup>[16]</sup> (Multilayer Perceptron), 用于分类或回归问题, 通过反向传播算法来学习网络的权重和偏置, 以最小化损失函数。
- (7) 卷积神经网络<sup>[17]</sup> (Convolutional Neural Network), 它的人工神经元可以响应一部分覆盖范围内的周围单元, 对于大型图像处理有出色表现。

而通过神经网络回归装甲板角点主要分为 1-stage 方法及 2-stage 方法, 代表方法为:

- (1) 1-stage 的单发多框检测器<sup>[18]</sup> (Single Shot MultiBox Detector) 及 YOLO<sup>[19]</sup> (You Only Look Once), 可以在一次前向传播中完成目标的定位和分类, 而不需要借助候选区域生成的步骤。
- (2) 2-stage 的 Fast RCNN<sup>[20]</sup>, 先使用选择性搜索 (selective search) 算法在图像中提取候选区域 (region proposals), 再将原始图像输入卷积神经网络 (CNN) 中, 得到整个图像的卷积特征图 (feature map), 再通过神经网络进行分类及回归。

考虑到自动瞄准系统对于算法实时性以及角点定位精确度的高要求, 笔者选择采用传统方法识别灯条结合机器学习进行图像分类的方法。在分类器的选择上,

考虑到目标图案为印刷体数字，并且不会有位移及形变，而且需要进行多分类，故笔者选择使用多层感知机作为装甲板中图案的分类器。

## 跟踪算法

由于图像中会同时出现多个目标，在自动瞄准时云台应该保持持续锁定同一目标，所以跟踪算法是非常有必要的。跟踪算法主要有基于图像信息的 KCF 跟踪算法<sup>[21]</sup>，是一种基于核相关滤波（Kernel Correlation Filter）的目标跟踪算法，它是在 CSK 算法的基础上进行了改进，提高了跟踪的准确度和鲁棒性，以及使用卡尔曼滤波器（Kalman Filter）对每个目标的状态进行预测和更新同时仅使用 IoU 作为数据关联的指标的多目标跟踪算法 SORT<sup>[6]</sup>（Simple online and realtime tracking）等。考虑到对于实时性的要求以及在该自动瞄准任务中，目标的图像特征都十分相似，KCF 不适合用于该任务的跟踪，故笔者选用 SORT 算法中对于目标滤波及关联的处理思路作为单目标跟踪器的算法基底。

## 状态估计算法

由 2.1 可知我们需要对于运动目标有一定的预测，这需要我们对于目标的运动状态（例如目标的运动速度）有一定的估计，而除位置姿态外其他的状态是无法直接被我们的系统所观测的，这就需要状态估计器来从观测数据来推断系统状态。

状态估计算法有很多种，其中比较常见的有：

- （1）卡尔曼滤波（Kalman Filter）：一种基于线性系统模型和高斯噪声假设的最优状态估计算法，它利用系统的状态转移方程和观测方程，通过预测和更新两个步骤，来递归地求解最小均方误差的状态估计。
- （2）扩展卡尔曼滤波（Extended Kalman Filter）：一种基于非线性系统模

型的状态估计算法,它通过对系统的状态和观测方程进行线性化近似,然后应用卡尔曼滤波的框架,来处理非线性系统的状态估计问题。

(3) 无迹卡尔曼滤波 (Unscented Kalman Filter): 一种基于非线性系统模型的状态估计算法,它通过对系统的状态分布进行采样,然后根据采样点的变换和加权,来近似系统的非线性变换和预测分布,从而避免了线性化近似带来的误差。

(4) 粒子滤波 (Particle Filter): 一种基于蒙特卡洛方法的状态估计算法,它通过对系统的状态空间进行随机采样,然后根据观测数据对采样点进行重要性加权和重采样,来近似系统的后验分布和最优状态估计。

(5) 多变量状态估计技术 (Multivariate State Estimation Technique, MSET): 一种基于非线性多元预测诊断技术的状态估计算法,它通过分析对比实际监测参数与设备正常运行时的健康数据为基础,对正常运行时的各个参数进行运算并做出估计,以这种正常的状态估计作为标准。

考虑到在 RoboMaster 比赛中,目标机器人为了规避弹丸的击打,运动状态一般为旋转运动叠加平移运动,而我们观测的对象为目标机器人上所搭载的装甲板,为了获取最佳的击打效果,我们应该估计出目标机器人的整体运动状态而非单一装甲板的运动状态,这意味着状态空间到观测空间的转换是一非线性过程,所以卡尔曼滤波是不能直接使用的。同时考虑到这一过程的非线性程度不大,再加上系统对于实时性的要求,故笔者采用扩展卡尔曼滤波作为自动瞄准系统的状态估计算法。

综上所述,由于我们需要在图像中实时的识别目标并精准的获取目标相对己方机器人的三维位置以及较为精确的估计目标的运动状态以获得实时的预测瞄

准位置，我们需要使用三维重建、识别、跟踪及状态估计算法来实现这一过程。

考虑到比赛中击打距离的需求、待识别目标的特征、整体运动状态以及系统对于实时性的要求，笔者选用单目 PnP 算法进行三维重建，传统方法识别灯条结合机器学习进行图像分类进行目标识别，采用 SORT 思想的单目标跟踪器进行目标跟踪，以及扩展卡尔曼滤波进行状态估计。

## 算法系统框架

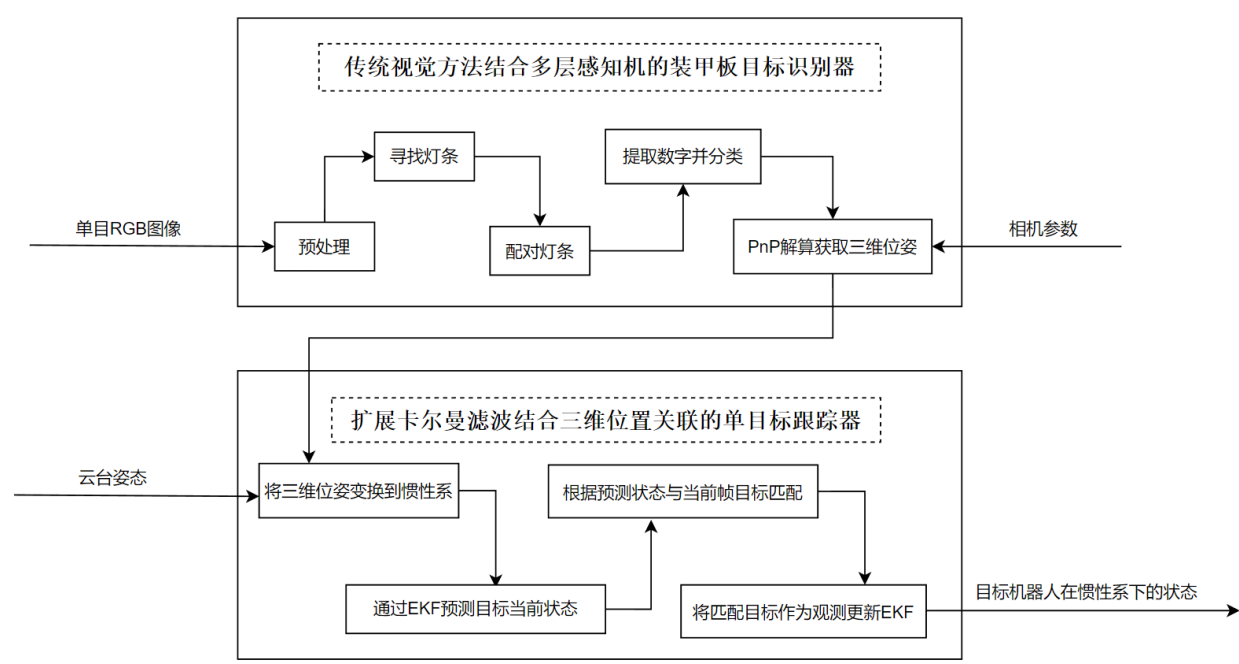


图 3.1 自动瞄准系统整体框架示意图

该算法系统的输入为单目 RGB 图像以及相机的内参矩阵及镜头畸变系数，以及云台姿态信息，最终输出在惯性系下跟踪着的机器人状态，其中包含机器人中心位置及机器人的运动线速度和角速度，以及两对装甲板的相对位置关系。

## 具体实践部分

# 1 边缘及轮廓检测

## 1.1 一、为什么要做边缘检测

大多数图像处理软件的最终目的都是识别与分割。识别即“是什么”，分割即“在哪里”。而为了将目标物体从图片中分割出来，如果这个物体有着鲜明的特征，使得目标物体和背景有着极大的区分度（如黑暗中的亮点，大面积的色块），我们就可以比较容易的将这个物体提取出来。

因为现在的目标物体和背景有着极大的区分度，也就意味着**目标和背景有着明显的“分界线”，也就是边缘**；而多个连续的边缘点，就构成了这个物体的**轮廓**。所以我们可以将检测物体这个任务，转换为检测物体和背景的分界线，也就是边缘检测。

## 1.2 二、如何进行边缘检测

在进行边缘检测之前，我们首先需要明确，我们想对图像中的哪种信息进行边缘检测。一般来讲，我们会**对图像的亮度信息进行边缘检测**，也就是在单色灰度图上检测边缘，此时检测到的边缘点是亮度变化较大的点。但有的时候，目标和背景的亮度差异不大，没法通过亮度边缘确定目标和背景的分界线；但目标和背景的颜色差异可能很大，这时就会**对图像的颜色信息进行边缘检测**，此时检测到的边缘点就是颜色变化最大的点。

在确定了我们想检测怎样的边缘后，我们就需要一个方法把边缘给找出来。下面介绍几个常用的方法（假设我们现在是要检测亮度边缘）

### 1.2.1 二值化

由于目标和背景的亮度差异很大，那么最简单的想法就是设定一个阈值，亮度高于该阈值的像素设为目标，亮度低于该阈值的像素设为背景。而这两片区域的交界处便是边缘。

再特殊一点：目标的亮度不一定就是很高，或者很低，而是在一个范围内（如100~150），此时我们的二值化就和上面有一定的区别，将这两个阈值范围内的像素设为目标，不在该范围内的设为边缘。



更进一步：二值化指的是一个函数  $f(x)$ ，其自变量是某个像素的亮度值，其因变量（或者说函数的输出）是 255 或 0，分别代表目标和背景。

### 1.2.2 自适应二值化

由于图片的亮度很容易受到环境的影响，比如环境亮度不同，相机曝光不同等因素都可能影响到最终成像出来的图片的亮度。这样，原本在较亮环境下设定的 180 的亮度阈值可以较好和分割出目标，到了较暗环境下效果就变差，甚至完全不起作用了。

但是环境对成像图片亮度的影响是整体的，也就是说整张图片一起变亮或者一起变暗，原本比背景亮的目标物体，在较暗环境下同样应该比背景亮。

基于这一点，我们可以提出一个简易的自适应二值化方法：对图像所有像素的亮度值进行从大到小排序，取前 20%（该数值为人为设定的阈值参数）的像素作为目标，其余为背景。

### 1.2.3 基于梯度的边缘

在上述两种方法中，通过一个阈值将整张图片分为两个部分，而两部分的交界处就作为边缘。这样的做法还有另一个缺点，如果图像中有一片区域亮度从低逐渐过渡到高，二值化同样会把这片区域分为两块。即，二值化得出的边缘，并不一定是图像中亮度变化最大（或较大）的地方。由于目标和背景亮度差异较大，所以交界处一定是图像中亮度变化最大（或较大）的地方。

为了解决该问题，可以使用基于梯度的边缘。其基本思想是：首先计算图片中每个像素点的亮度梯度大小（一般使用 Sobel 算子），然后设定一个阈值，梯度高于该阈值的作为边缘点。同样，类似与自适应二值化，这个阈值也可以设定成一个比值。

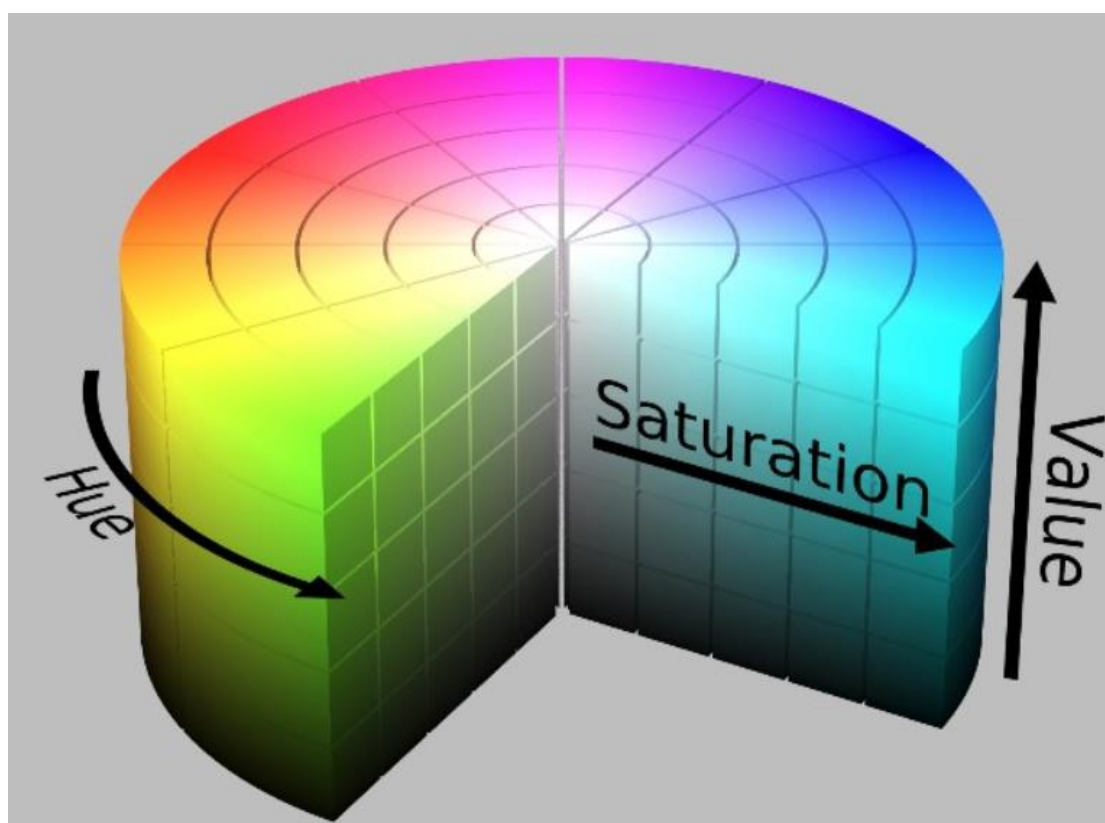
在实际使用中，我们通常会使用 Canny 算法进行基于梯度的边缘检测，这个算法中做了很多额外措施，使得边缘检测的效果较好。详细算法内容这里不作介绍，有兴趣可以自行百度。

## 1.2.4 补充：检测颜色边缘

在上面几种方法中，我们都是进行亮度边缘检测，亮度边缘检测有一个明显的特征，即每个像素的亮度都可以用一个数值进行表达。但当我们想进行颜色边缘检测时，我们似乎并不能用一个数值来表达该像素的颜色差异，必须使用

RGB 三通道数值才能表达一个像素的颜色。

首先，在 RGB 颜色表示方法中，每个颜色分量都包含了该像素点的颜色信息和亮度信息。我们希望对 RGB 颜色表示进行一个变换，使得像素点的颜色信息和亮度信息可以独立开来。为此，我们可以使用 HSV 颜色空间。



在 HSV 颜色空间中，H 分量代表色度，即该像素是哪种颜色；S 分量代表饱和度；V 分量代表亮度（如上图）。这种颜色表示方法很好地将每个像素的颜

色、饱和度和亮度独立开。至于 RGB 颜色空间如何转换为 HSV 颜色空间，这里不作介绍，有兴趣可以自行百度。

有了 HSV 颜色空间，由于其 H 通道就代表了像素的颜色，我们就可以在 H 通道上使用上述几种边缘检测方式，从而得出颜色边缘。

### 1.3 三、边缘检测的后处理

不论是使用二值化、还是自适应二值化、还是基于梯度的边缘检测方法，其检测结果都不可能正好分毫不差的将目标完整保留下来，并将背景完全剔除。即使图像质量极佳，或者目标特征极为明显，使得正好将目标和背景区分开，检测结果也还停留于像素层面，即每个像素是目标还是背景，而我们想要的则是目标在哪片区域。

所以后处理的目的主要有三个：**剔除错误的背景边缘、补充缺失的目标边缘、将目标表达成一个区域。**

对于前两点，我们通常会首先使用开闭运算处理边缘图。其中开运算连接断开区域，闭运算删除游离的噪声区域。详细算法的计算方式，这里不作介绍，有兴趣可以自行百度。

对于第三点，我们会使用轮廓检测。轮廓可以理解为一系列连通的边缘点，并且这些边缘点可以构成一个闭合曲线。

仅仅使用开闭运算，对前两点的改善十分有限，为了进一步从大量边缘中找到目标边缘，我们在进行完轮廓提取后，还会进行形状筛选。即根据目标的形状信息，剔除形状不正确的的轮廓（这里的形状同样包括大小等各种目标独特的特征）。形状筛选的方式通常有，计算轮廓面积，计算最小外接矩形，椭圆拟合，多边形拟合等。

### 1.4 四、OpenCV 中的边缘检测

这一节，我们介绍一下使用哪些函数可以在 OpenCV 中进行边缘检测。

---

```
double cv::threshold(InputArray src,
                     OutputArray dst,
                     double thresh,
                     double maxval,
                     int type);
```

该函数是 OpenCV 中进行二值化的函数，其第一个参数是需要进行二值化的图像，第二个参数接收处理结果的位置，后几个参数详细使用方法请百度，全部介绍一遍比较冗长。

使用示例：

```
cv::threshold(src, bin, 150, 255, cv::THRESH_BINARY);
```

上述函数调用将图像 src 中大于 150 的像素设为 255，小于 150 的像素设为 0，并保存在图像 bin 中。

---

```
void cv::Canny(InputArray image,
               OutputArray edges,
               double threshold1,
               double threshold2,
               int apertureSize = 3,
               bool L2gradient = false);
```

该函数是 OpenCV 进行 canny 边缘检测的函数，其第一个参数是待处理的图像，第二个参数是接收处理结果的位置，后几个参数详细使用方法请百度，全部介绍一遍比较冗长。

使用示例：

```
cv::Canny(src, edge, 50, 150);
```

上述函数调用对图像 src 进行 canny 边缘检测，并将检测结果保存在图像 edge 中，而 50 和 150 是 canny 算法需要的两个参数。

---

```
// 轮廓提取函数，详细用法请百度
cv::findContours(InputOutputArray image,
                 OutputArrayOfArrays contours,
                 OutputArray hierarchy, int mode,
                 int method,
                 Point offset=Point());

// 最小外接矩形，详细用法请百度
RotatedRect cv::minAreaRect(InputArray points);
```

```
// 椭圆拟合，详细用法请百度
RotatedRect cv::fitEllipse(InputArray points);

// 多边形拟合，详细用法请百度
void cv::approxPolyDP(InputArray curve, OutputArray
approxCurve, double epsilon, bool closed);

// 开闭运算，详细用法请百度
void morphologyEx(InputArray src, OutputArray dst,
                  int op, InputArray kernel,
                  Point anchor=Point(-1,-1), int
iterations=1,
                  int borderType=BORDER_CONSTANT,
                  const Scalar&
borderValue=morphologyDefaultBorderValue());
```

上述函数常用于后处理阶段。



扣除以上图片中的苹果

# 2.Ubuntu 的基本使用

## 1. Ubuntu 及 Linux 简介

Ubuntu 是一个以桌面应用为主的 Linux 系统，是开放源代码的自由软件。

我们平常使用的系统一般为 Windows 和 MacOS，虽然能够满足日常的使用需求，但是其性能与稳定性却不是代码运行的最好平台。Linux 系统不仅系统性能稳定，而且是开源软件。其核心防火墙组件性能高效、配置简单，保证了系统的安全。因此经常被用于服务器。所以我们的代码也以 Linux 系统为开发平台，并运行于其上，这也能在一定程度上防止代码出现一些奇怪的未知 bug。

## 2.Ubuntu/Linux 常用命令

- **dpkg: package manager for Debian**
- `dpkg-i package` # 安装
- `dpkg-r package` # 卸载
- `dpkg-P|--purge package` # 卸载并删除配置文件

如果安装一个包时说依赖某些库，可以先 `apt-get install somelib`。

- **apt:**
- `apt-get install packs` # 安装
- `apt-get update`  
# 更新源
- `apt-get upgrade`  
# 升级系统
- `apt-get autoremove` # 自动删除无用的软件
- `apt-get remove packages` # 删除软件(--purge 并清除配置文件)
- **系统命令:**
- `/etc/issue` 或 `lsb_release -a` # 获取 Ubuntu 版本号
- 
- `ls <目录>` # 列出目录（默认当前目录）下的文件列表，默认按字母顺序排序（-a 包含隐藏文件）
- `cd directory_name` # 改变当前工作目录位置
- `pwd` # 打印当前工作目录的完整路径
- `touch <文件名>` # 快速创建文档
- `cat <文件名>` # 查看文档

- `mkdir <目录名>` # 创建目录 (`-p` 如果路径中某些目录不存在将自动创建)
- `rm <文件|目录>` # 删除文件和目录 (`-r` 将参数中所有目录及其子目录递归删除)
- `mv <需要移动的文件> <指定目录>` # 移动文件或目录到指定目录下, 并具有重命名功能
- `which <系统命令>` # 看到某个系统命令是否存在, 以及执行的到底是哪一个位置的命令
- .....

## 2. OpenCV

### OpenCV 简介

OpenCV 是一个基于 BSD 许可 (开源) 发行的跨平台计算机视觉和机器学习软件库, 可以运行在 Linux、Windows、Android 和 Mac OS 操作系统上。

### 2.1 OpenCV 常用函数

边缘检测: 颜色+形状 图像几何变换: 仿射变换 相机标定: 内参, 畸变

特征点匹配: SIFT, ORB, FAST 三维定位: 单目+双目 追踪: KCF 等

### 2.2 OpenCV 库

#### 1. C++ + Cmake + OpenCV + Clion

参考教程: <https://sjtu-robomaster-team.github.io/vision-learning-5-opencv/>

如果不需要使用 `contrib` 库的功能, 那么只需要一行命令就可以了。

```
sudo apt-get install libopencv-dev python-opencv
```

#### 2. Python + OpenCV + Pycharm

在命令行中直接输入 `pip install opencv-python`, 安装完成后输入 `python, import cv2` 无报错就可以使用了。

有些时候可能也需要源码编译安装, 过程与 C++ 比较类似。



# 3. 相机的标定

## 一、相机标定的原理

在图像测量过程以及机器视觉应用中，为确定空间物体表面某点的三维几何位置与其在图像中对应点之间的相互关系，必须建立相机成像的几何模型，这些几何模型参数就是相机参数。在大多数条件下这些参数必须通过实验与计算才能得到，这个求解参数（内参、外参、畸变参数）的过程就称之为相机标定（或摄像机标定）。无论是在图像测量或者机器视觉应用中，相机参数的标定都是非常关键的环节，其标定结果的精度及算法的稳定性直接影响相机工作产生结果的准确性。因此，做好相机标定是做好后续工作的前提，提高标定精度是科研工作的重点所在。在 RM 的比赛中，如果相机标定参数有比较大的问题，那么极有可能出现自瞄锁定后云台发疯的情况。

## 1. 坐标系关系

相机标定主要涉及三个坐标系：图像坐标系、摄像机坐标系和世界坐标系。

### 图像坐标系

摄像机采集的图像变换为数字图像后，每幅数字图像在计算机内为  $H \times W$  数组， $H$  行  $M$  列的图像中每一个元素（pixel）数值就是图像点的亮度（灰度）。如图，在图像上定义直角坐标系（ $U, V$ ），该坐标系以像素为单位，由于（ $u, v$ ）只能表示像素位于数组中的列数与行数，并没有使用物理单位表示该像素在图像中位置，所以需要再建立以物理单位（mm）表示的图像坐标系，该图像坐标系以图像内某一点  $O_1$  为原点， $X$  轴和  $Y$  轴分别平行于  $U$  轴和  $V$  轴。在  $X, Y$  坐标系中，原点  $O_1$  定义在摄像机光轴与图像平面的交点，该点一般位于图像中心。但是由于制造原因，很多情况下会有偏移。这也是相机需要标定的原因，需要找到这两个坐标系之间的位移关系，这在相机内参矩阵中对应两个数值。

16213741\_es3j

上图中，（ $u, v$ ）表示以像素为单位的图像坐标系的坐标，（ $x, y$ ）表示以 mm 为单位的图像坐标系的坐标。

两者存在一下关系式：

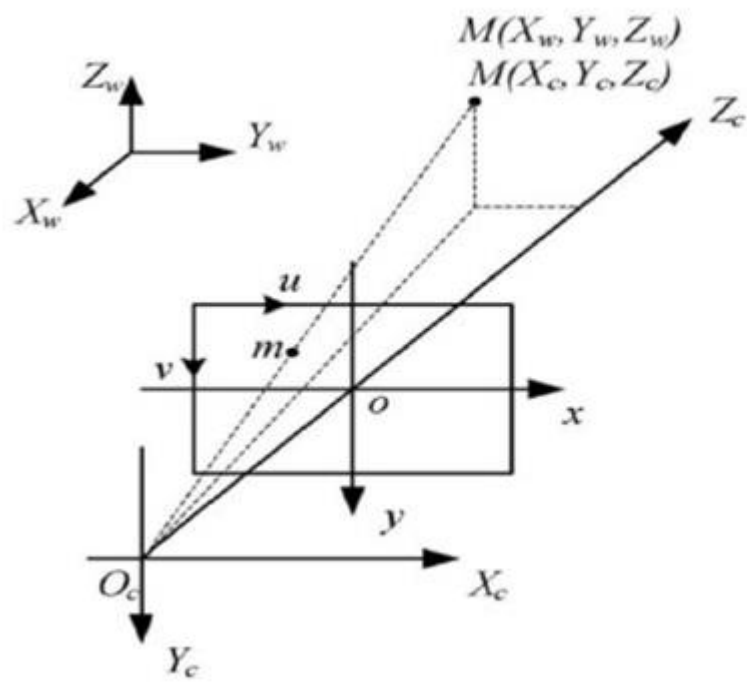


$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{d_x} & 0 & u_x \\ 0 & \frac{1}{d_y} & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

即

$$\left\{ \begin{array}{l} u = \frac{x}{d_x} + u_0 \\ v = \frac{y}{d_y} + v_0 \end{array} \right.$$

### 摄像机坐标系

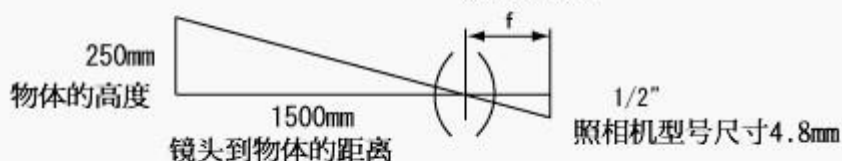


如图， $O_c$  点为摄像机光心， $X_c$  轴和  $Y_c$  轴与图像的  $X$  轴与  $Y$  轴平行， $Z_c$  轴为摄像机光轴，它与图像平面垂直。光轴与图像平面的交点，即为图像坐标系的原点。

由点  $O_c$  与  $X_c$ 、 $Y_c$ 、 $Z_c$  轴组成的直角坐标系称为摄像机坐标系， $f$  为摄像机焦距。

## 焦距的计算方法

$$\text{镜头的焦距} = \frac{(\text{镜头到物体的距离}) \times (\text{照相机型号尺寸})}{\text{物体的高度}}$$



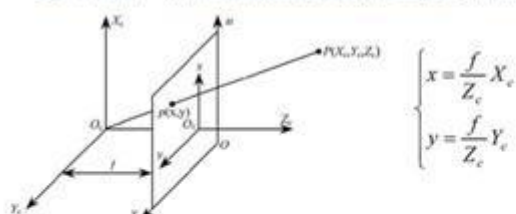
$$\text{焦距 } f = \frac{1500 \times 4.8}{250} = 28.8\text{mm}$$

/sloan6

要理解图像坐标系与摄像机坐标系的关系，就需要理解**针孔模型**。针孔模型又称为线性摄像机模型，任何空间点  $M$  在图像中的投影位置  $m$ ，为光心  $O_c$  与  $M$  的连线  $O_cM$  与图像平面的交点。在这个模型中包含着大量的相似三角形，但是要注意像平面和现在讨论的平面其实是关于  $O_c$  对称的，所以计算的时候不要忘记了负号，不然出来的图像就是倒着的。这种关系也称为重心摄影或者透视投影。此时有比例关系如下：

$$Z_c \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix}$$

空间任何一点  $P$  在图像上的成像位置可以用针孔模型近似表示：



$$Z_c \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} 1/dx & 0 & u_0 \\ 0 & 1/dy & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ 0^T & 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} f_x & 0 & u_0 & 0 \\ 0 & f_y & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ 0^T & 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} = \mathbf{M}_1 \mathbf{M}_2 \mathbf{X}_w = \mathbf{M} \mathbf{X}_w$$

$\mathbf{M}$  为  $3 \times 4$  矩阵，称为投影矩阵；

$\mathbf{M}_1$  完全由  $f_x$ 、 $f_y$ 、 $u_0$ 、 $v_0$  决定，只与摄像机内部结构有关，这些参数称为摄像机内部参数；

$\mathbf{M}_2$  完全由摄像机相对于世界坐标系的方位决定，称为摄像机的外部参数；

$\mathbf{X}_w$  为空间点在世界坐标系下的齐次坐标

## 世界坐标系

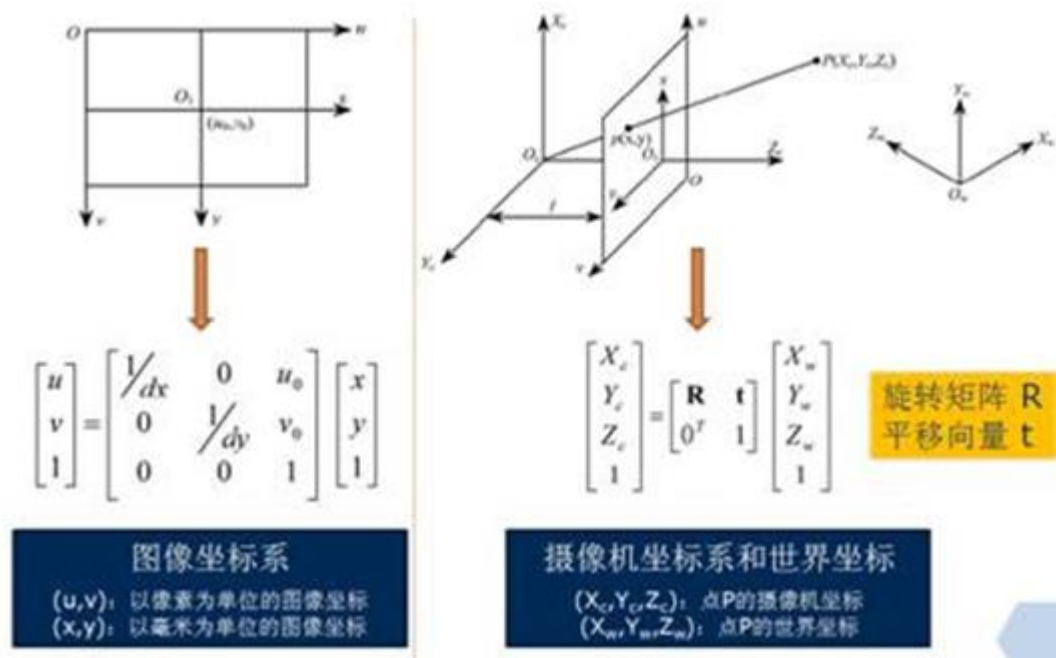
由于摄像机可以安放在现实环境中任意位置，所以在环境中任选一个基准坐标系来描述摄像机位置，并用它描述环境中任何物体的位置，该坐标系为世界坐

标系。它有  $X_w$ 、 $Y_w$  和  $Z_w$  轴组成，摄像机坐标系与世界坐标系之间的关系可以用旋转矩阵  $R$  与平移向量  $t$  来描述。这就是两个刚体坐标系上的坐标关系。

世界坐标系和摄像机坐标系之间的转化关系符合如下公式：

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} = \begin{bmatrix} R & t \\ 0^T & 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$$

总结来说，三个坐标系之间有如下图的转化关系：



综合以上公式，可以得到

$$Z_c \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} \alpha & \gamma & u_0 \\ 0 & \beta & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} R & t \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$$

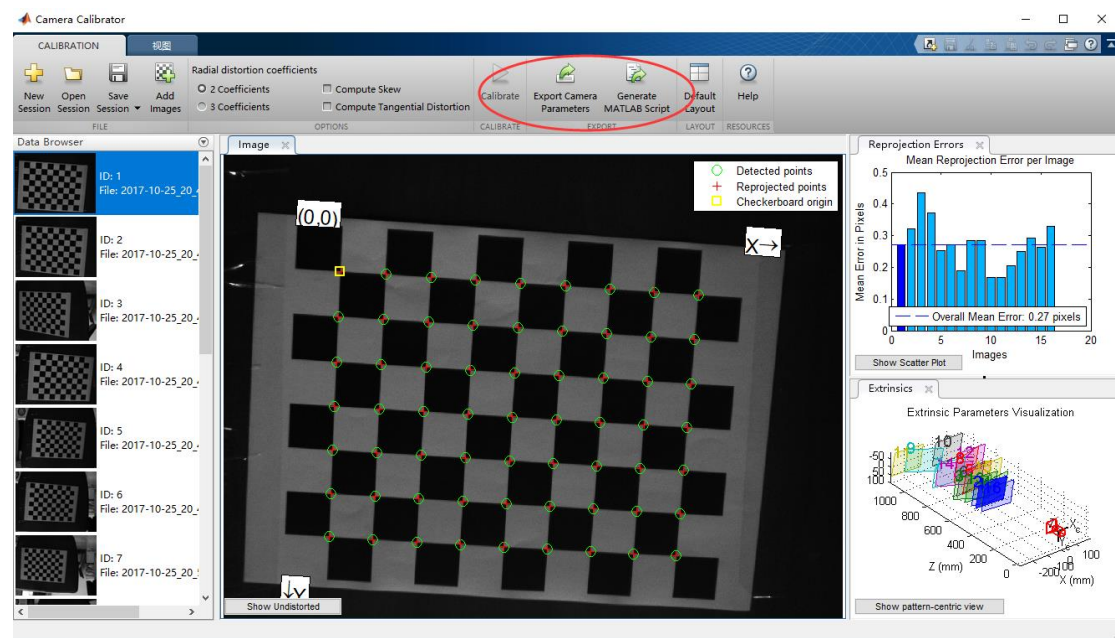
$\alpha = f/dx$ ,  $\beta = f/dy$ , 分别代表了以  $X$  轴与  $Y$  轴方向上的像素为单位表示的等效焦距。  $\gamma$  在较高精度的相机模型中引入，表示图像平面中以像素为单位的坐标轴倾斜程度的量度，  $\gamma = \alpha \cdot \tan(\theta)$ ，  $\theta$  是相机 CCD 阵列  $v$  轴的偏斜角度。

## 1.4.1 2. 转化过程

由以上的公式我们可以知道，如果已知摄像机的内外参数，就知道投影矩阵  $M$ ，这时候对任何空间点就可以求出其对应图像坐标，但是如果已知空间某点的像点  $m$  位置  $(u, v)$  即使已经知道摄像机内外参数， $X_w$  也不能唯一确定，因为在投影过程中消去了  $Z_c$  的信息。所以已知世界中固定坐标的几个点，用 pnp 算法就能得到相机的旋转和平移，而已知图像中的几个点的图像坐标和相机的旋转平移矩阵，却不足以推算这几个点的世界坐标，还需要知道  $Z_c$  的值。这在之后的测距教程中会提及。

## 二、实战指导

1. 检测角点（需提前输入角点数）`findChessboardCorners`
2. 对角点进行亚像素精确化 `find4QuadCornerSubpix`
3. 可选：将角点显示 `drawChessboardCorners`
4. 根据角点数和尺寸创建一个理想的棋盘格（用 `point` 向量存储所有理论上的角点坐标）`calibrateCamera`
5. 由理想坐标和实际图像坐标进行标定 `projectPoints`
6. 计算反向投影误差
7. 可选：可以使用 `matlab` 查看自己的标定精度



## 4.透视变换和仿射变换

我们今天学习两种平面几何变换：

1. 使用  $2 \times 3$  矩阵的变换，称为“仿射变换”。
2. 基于  $3 \times 3$  矩阵进行变换，称为“透视变换”。“透视变换”在变换图像的视角中应用非常广泛，比如得到路面的俯瞰图。



图 1 输入图像



图 2 输出图像

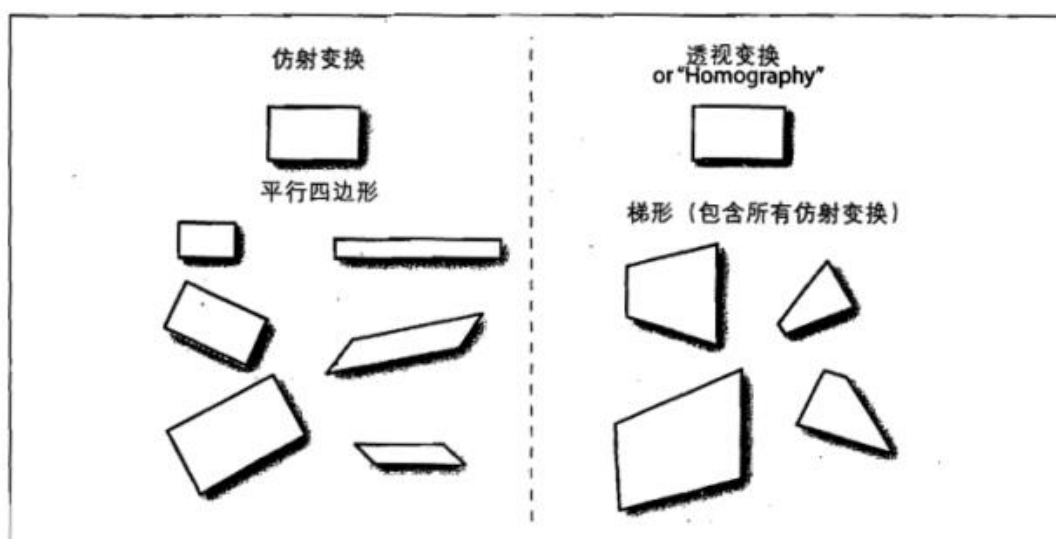


图 6-13: 仿射变换和透视变换

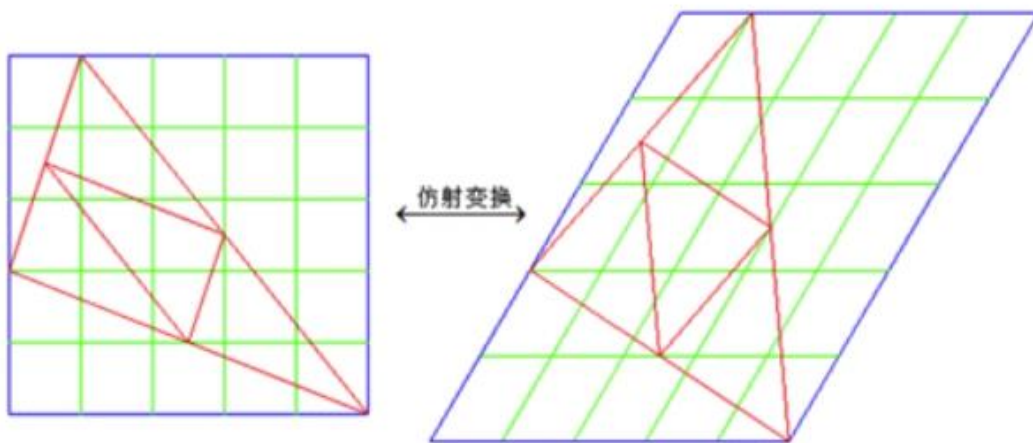
## 一、\*仿射变换\*

在本章中，“变换”定义为 2D 图像之间的一个映射，即  $H(\text{输入图像}) = \text{输出图像}$ 。仿射变换可以理解为：平面中的任何平行四边形  $ABCD$  可以通过一些仿射变换映射到任何其他平行四边形  $A'B'C'D'$ 。如果这些平行四边形的面积不是零，所表示的仿射变换就由两个平行四边形的(三个顶点)唯一定义。一个仿

射变换感性理解就是，将输入图像想象为一个大的矩形橡胶片，然后通过角上的推或拉变形来制作同样样子的平行四边形。

## 仿射变换

简单说来，仿射变换，就是允许图形任意倾斜，而且允许图形在两个方向上任意伸缩的变换（如下图）。



仿射变换，可以保持原来的线共点、点共线的关系不变，保持原来相互平行的线仍然平行，保持原来的中点仍然是中点，保持原来在一直线上几段线段之间的比例关系不变。但是，仿射变换不能保持原来的线段长度不变，也不能保持原来的夹角角度不变。

### 4 图 1 摄像机成像模型

#### \2. 仿射变换的数学表示

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a_1 & a_2 & t_x \\ a_3 & a_4 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad (2.26)$$

其中  $(t_x, t_y)$  表示平移量，而参数  $a_i$  则反映了图像旋转、缩放等变化。将参数  $t_x, t_y, a_i (i=1 \sim 4)$  计算出，即可得到两幅图像的坐标变换关系。

具体请参考：

<https://courses.cs.washington.edu/courses/csep576/11sp/pdf/Transformations.pdf>

### 3. OpenCV 中的仿射变换



```

1  (1) getAffineTransform      //三点法
2  Mat M1=getAffineTransform(const Point2f* src, const Point2f* dst)
3      //参数const Point2f* src: 原图的三个固定顶点
4      //参数const Point2f* dst: 目标图像的三个固定顶点
5      //返回值: Mat型变换矩阵, 可直接用于warpAffine()函数
6      //注意: 顶点数组长度超过3个, 则会自动以前3个为变换顶点; 数组可用Point2f[] 或Point2f*表示

```

Point2f, 就是含 float x 和 float y 的二元组

```

typedef Point_<int> Point2i;
typedef Point_<int64> Point2l;
typedef Point_<float> Point2f;
typedef Point_<double> Point2d;
typedef Point2i Point;

```

, 其他同理。

```

1  (2) getRotationMatrix2D      //直接指定比例和角度
2  Mat M2=getRotationMatrix2D (CvPoint2D32f center, double angle, double scale)
3      //参数CvPoint2D32f center, 表示源图像旋转中心
4      //参数double angle, 旋转角度, 正值表示逆时针旋转
5      //参数double scale, 缩放系数

```

其中 CvPoint2D32f 是过时的 Opencv 二元组结构体, 目前版本已改为 Point2f

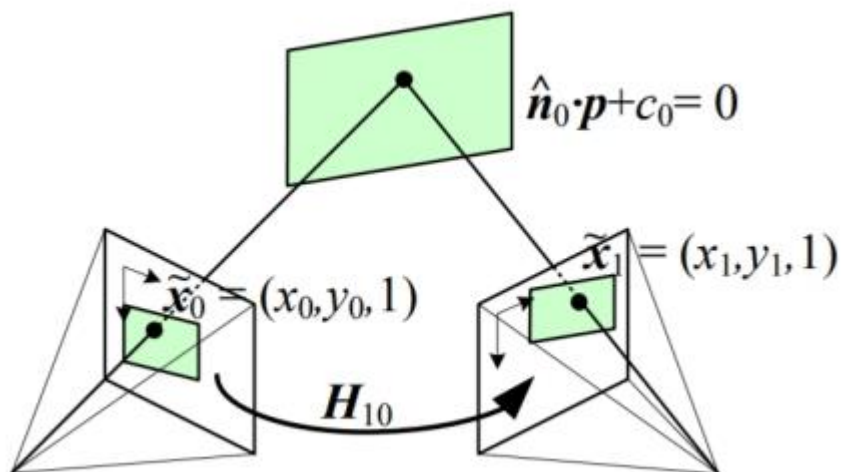
```

void warpAffine(InputArray src, OutputArray dst, InputArray M, Size dsize, int flags = INTER_LINEAR,
    int borderMode = BORDER_CONSTANT, const Scalar& borderValue = Scalar())
//参数InputArray src: 输入变换前图像
//参数OutputArray dst: 输出变换后图像, 需要初始化一个空矩阵用来保存结果, 不用设定矩阵尺寸
//参数InputArray M: 变换矩阵, 用另一个函数getAffineTransform()计算
//参数Size dsize: 设置输出图像大小
//参数int flags = INTER_LINEAR: 设置插值方式, 默认方式为线性插值(另一种WARP_FILL_OUTLIERS)
//参数int borderMode=BORDER_CONSTANT: 边界像素模式, 默认值BORDER_CONSTANT
//参数const Scalar& borderValue=Scalar(), 在恒定边界情况下取的值, 默认值为Scalar (), 即0

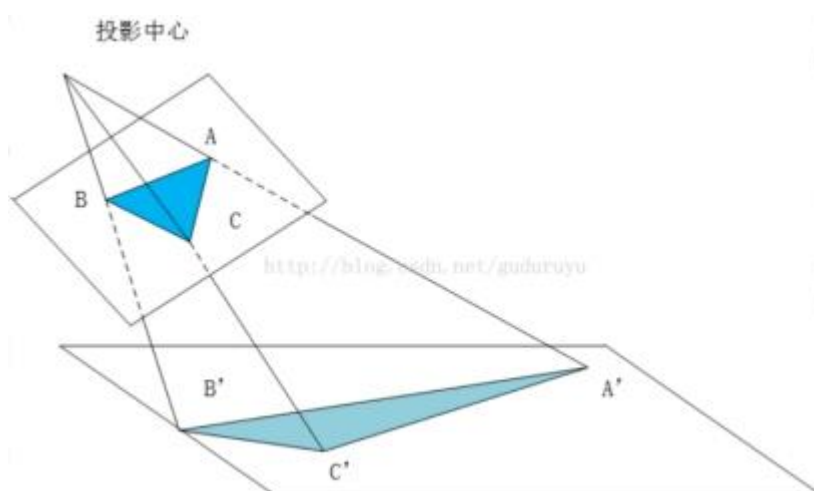
```

## 二、\*透视变换\*

透视变换是将一张图像从一个视角变换到另一个视角的方法。



用两个不同位置处的照相机拍摄同一个绿色平面，两张图片的坐标可以通过  $H_{10}$ （单应性矩阵）来转换。透视变换也称作射影变换，见下图。



## \2. 透视变换的数学表示

详见 2.2 3D Transforms

<https://www.microsoft.com/en-us/research/wp-content/uploads/2004/10/tr-2004-92.pdf>

## 3. OpenCV 中的透视变换函数

```
cv::Mat cv::getPerspectiveTransform( // 返回3x3透视变换矩阵
    const cv::Point2f* src, // 源图像四个顶点坐标（点数组）
    const cv::Point2f* dst // 目标图像上四个顶点的坐标（点数组）
);
```



```
void cv::warpPerspective(  
    cv::InputArray src, // 输入图像  
    cv::OutputArray dst, // 输出图像  
    cv::InputArray M, // 3x3 变换矩阵  
    cv::Size dsize, // 目标图像大小  
    int flags = cv::INTER_LINEAR, // 插值方法  
    int borderMode = cv::BORDER_CONSTANT, // 外推方法  
    const cv::Scalar& borderValue = cv::Scalar() // 常量边界时使用  
);
```

## 2      \*作业：\*

任务：用透视变换将车牌转化为正视图。

基本目标：用代码实现任务。

进阶目标：利用参考文献 1，调用鼠标的 callback 回调函数，实现在原图上标出四个特征点后自动转换为正视车牌图。

拓展目标：利用你学习到的轮廓提取、HSV 颜色空间转换等知识，实现识别并筛选出车牌轮廓后转化为正视车牌图后，通过二值化方法提取出正视的仅有字符的二值图。



## 5. 了解以及 PNP 实践

实践部分：

opencv 函数为：

```
solvePnP(InputArray objectPoints, InputArray imagePoints, InputArray cameraMatrix,  
InputArray distCoeffs, OutputArray rvec, OutputArray tvec, bool useExtrinsicGuess =  
false, int flags = SOLVEPNP_ITERATIVE);
```

参数解释：

- **objectPoints**: 对象在世界坐标系中的三维点集，可以是一个 **vector** 或 **Mat** 类型的输入。
- **imagePoints**: 对应于 **objectPoints** 中三维点的二维图像坐标点集，也可以是一个 **vector** 或 **Mat** 类型的输入。
- **cameraMatrix**: 相机的内参数矩阵，即相机的内部参数。它是一个 3x3 的浮点型矩阵。
- **distCoeffs**: 相机的畸变系数，通常是一个 5x1 的浮点型向量。

- `rvec`: 输出的旋转向量，表示相机的旋转向量。
- `tvec`: 输出的平移向量，表示相机的平移向量。
- `useExtrinsicGuess`: 布尔值，表示是否使用外部猜测来初始化解算器。
- `flags`: 解算器的标志，用于选择解算方法，默认为 `SOLVEPNP_ITERATIVE`。

函数的作用是通过给定的三维电极和对应的二维图像点击，以及相机的内参数和畸变系数，计算相机的姿态（旋转和平移）。输出的旋转向量和平移向量代表了相机从世界坐标系到相机坐标系的变换。

**注意:** `solvePnp` 函数假设相机的投影是透视投影你，并且假设对象在世界坐标系中是静态的，如果对象在世界坐标系中存在动态变换，则需要更加高级的算法，例如视觉里程计（Visual Odometry）或扩展的传感器融合方法。以下为方法举例：

1. 视觉里程计（Visual Odometry）：视觉里程计是一种利用相邻图像之间的运动信息来估计相机或机器人的轨迹的技术。它通过跟踪特征点或提取特征描述子来计算相邻图像之间的运动，并根据这些运动信息估计姿态变化。视觉里程计可以用于实时姿态估计和机器人导航等应用。

2. 扩展的传感器融合方法：除了相机视觉，您可以考虑融合其他传感器的信息来更准确地估计姿态。例如，结合惯性测量单元（Inertial Measurement Unit，简称IMU）的加速度计和陀螺仪数据，可以提供机器人的姿态估计。通过将视觉数据与惯性数据融合，可以实现更鲁棒和准确的姿态估计。

4. 运动估计和状态估计算法：针对机器人的移动和姿态估计，还有其他更高级的算法和方法可供选择。例如，扩展卡尔曼滤波（Extended Kalman Filter，简称EKF）和非线性优化方法（如 Bundle Adjustment）等，可以在动态环境中更好地处理姿态估计问题。

## 6. 三维刚体运动如何表示旋转

由于笔者的水平实在有限，不敢妄自菲薄的传授相关方面知识从而引起学者的误解，但我会从实践的项目开发来介绍我们用的了该部分的哪些知识以及如何使用相关部分的知识。

1. 欧拉角表示旋转，一般会 and 旋转矩阵相结合，这就是为什么我们选择

Eigen 库，因为它提供了我们所能用到的所有线性代数操作的相关函数。具体内容请参考《14 讲》第三章，我已经上传到了 github 的文档部分，链接如下：[pym96/MA\\_training: This is the repository for MA vision group training \(github.com\)](https://github.com/pym96/MA_training)

2. 四元数：无论何种三维刚体运动的旋转表示方式，其实都会出现万向锁的问题（不了解的依然可以参考《14 讲》），但是四元数却没有这个问题，由于其良好的特性故在 ros 的 tf 中普遍使用 tf2::Quaternion 来表示旋转。但是，四元数的概念确实有些难懂，在我的学习过程中也看了很多资料，其中以下的两个资料讲的最直观易懂，相信加以实践读者应该会对四元数有更深入的理解。

<https://eater.net/quaternions/video/intro>

<https://www.youtube.com/watch?v=zjMulxRvygQ>

## 7.坐标系的相互转换

在二维空间中，假如我有一个坐标(x,y)，如果想把这个坐标顺时针旋转 90 度，那旋转矩阵应该是什么样的？

这个问题看起来无比简单，但是如果不了解一定的相关知识就会陷入迷惑，就像你不知道 加法的含义那 1+1 也是不会算的，下边直接给出结论，推导过程你可以自己去搜索相关资料：

在 2d 空间中，旋转矩阵可以表示成

$\begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$

$\theta$  即为旋转的角度。

那么问题来了，对于 Robomaster 比赛来说，很少会涉及到 2d 空间中的坐标系转换，如果在 3d 空间中那我们是否有一定的参考旋转规则呢？答案是有的，结论如下：

对于 2D 空间的旋转，我们可以使用简单的二维旋转矩阵，其中只涉及三角函数和角度。对于 3D 空间的旋转，我们可以通过绕各个坐标轴的旋转来实现。下面是绕 X 轴、Y 轴和 Z 轴的三个旋转矩阵，其中只涉及三角函数和角度：

绕 X 轴旋转：

```

'''
| 1      0      0      |
| 0  cos(θ) -sin(θ) |
| 0  sin(θ)  cos(θ) |
'''

```

绕 Y 轴旋转：

```

'''
| cos(θ)  0  sin(θ) |
|  0      1      0   |
| -sin(θ)  0  cos(θ) |
'''

```

绕 Z 轴旋转：

```

'''
| cos(θ) -sin(θ)  0 |
| sin(θ)  cos(θ)  0 |
|  0      0      1 |
'''

```

这些矩阵可以用来在 3D 空间中绕各个坐标轴进行旋转。选择合适的角度  $\theta$  可以实现所需的旋转效果。例如，如果要绕 Y 轴旋转 90 度，则将  $\theta$  设为  $\pi/2$  (弧度制)。

由于三维刚体运动的旋转矩阵可以叠加，这里举一个极端的例子：你需要将一个坐标绕 x 轴旋转 45 度，绕 y 轴旋转 30 度，绕 z 轴旋转 60 度，那么就可以将坐标先乘以按 x 轴旋转的旋转矩阵，再 y 再 z，写成数学语言就是以下形式：

设总的旋转矩阵为  $R$ ，绕 X 轴旋转的旋转矩阵为  $R_x$ ，绕 Y 轴旋转的旋转矩阵的规则是  $R_y$ ，绕 Z 轴旋转的旋转矩阵的规则是  $R_z$ ，那么  $R = R_x * R_y * R_z$

## 8. 二值化所用到的 OpenCV

cv

cv cvtColor

cv

```

        cv
        cv threshold                                255
        cv
        cv findContours                                cv
cv

```

基本就是上边这三个函数，先转灰度，然后二值化最后找轮廓，以下是详细的教程文档：

当处理固定光源物体的图像时，常见的步骤包括将图像转换为灰度图像，进行二值化处理，然后找到轮廓。以下是一个基于你提供的代码片段的教程，演示如何使用 OpenCV 在 C++ 中进行这些操作。

### ### 图像处理教程：灰度化、二值化和轮廓检测

1. 首先，确保你已经在你的 C++ 项目中包含了 OpenCV 头文件：

```

```cpp
#include <opencv2/opencv.hpp>
```

```

2. 读取彩色图像，并转换为灰度图像：

```

```cpp
cv::Mat rgb_img = cv::imread("path_to_your_rgb_image.jpg"); // Read the RGB
image
cv::Mat gray_img;
cv::cvtColor(rgb_img, gray_img, cv::COLOR_RGB2GRAY); // Convert to
grayscale
```

```

3. 对灰度图像进行二值化处理：

```

```cpp
int binary_thres = 128; // Binary threshold value
cv::Mat binary_img;
cv::threshold(gray_img, binary_img, binary_thres, 255, cv::THRESH_BINARY);
// Apply binary thresholding
```

```

```
```
```

#### 4. 找到图像中的轮廓:

```
```cpp
std::vector<std::vector<cv::Point>>> contours;
std::vector<cv::Vec4i> hierarchy;
cv::findContours(binary_img, contours, hierarchy, cv::RETR_EXTERNAL,
cv::CHAIN_APPROX_SIMPLE); // Find contours
```
```

完整的代码示例如下:

```
```cpp
#include <opencv2/opencv.hpp>

int main() {
    // Read the RGB image
    cv::Mat rgb_img = cv::imread("path_to_your_rgb_image.jpg");

    // Convert to grayscale
    cv::Mat gray_img;
    cv::cvtColor(rgb_img, gray_img, cv::COLOR_RGB2GRAY);

    // Apply binary thresholding
    int binary_thres = 128;
    cv::Mat binary_img;
    cv::threshold(gray_img, binary_img, binary_thres, 255, cv::THRESH_BINARY);

    // Find contours
    std::vector<std::vector<cv::Point>>> contours;
    std::vector<cv::Vec4i> hierarchy;
    cv::findContours(binary_img, contours, hierarchy, cv::RETR_EXTERNAL,
cv::CHAIN_APPROX_SIMPLE);
}
```

```

// Draw the contours on the original RGB image
cv::drawContours(rgb_img, contours, -1, cv::Scalar(0, 255, 0), 2);

// Display the result
cv::imshow("Contours", rgb_img);
cv::waitKey(0);

return 0;
}
'''

```

确保将 `path\_to\_your\_rgb\_image.jpg` 替换为实际的图像文件路径。这个示例代码将读取彩色图像，转换为灰度图像，应用二值化处理，找到轮廓，并在原始图像上绘制轮廓。最后，它会显示带有轮廓的图像窗口。

上边的代码只是找到了轮廓，作为开发者要时刻将提高效能牢记在心，首先思考：二值化的目的是什么？本质就是为了未来的 `pnp` 解算测距，以一张  $1920 * 1080$  的图片距离，无论目标机器人离得多近，相信大部分的地方都不是 ROI(region of interest: 检测区域)，所以我们可以从源头解决这个问题，通过二值化后的轮廓后（可以选择 `dilate`），再由拷贝构造函数传递给原图片，这样一来后续操作的只是包含装甲板灯条的图片，减少了很多无关区域的检测。

```

auto      = light boundingRect
auto      = rbg_img

```

上边代码的含义是，找到灯条（对象类型为 `RotatedRect`），在利用 `boundingRect` 返回包含 ROI 的最小矩形，然后再将其传到原图片 `rbg_img` 中，从而得到只提取 ROI 的操作。

当然，由于笔者水平问题，在写该文档时一定会遇到其他的问题，就个人习惯，我喜欢问 `chatgpt`，以下是我写该部分时问 `chatgpt` 的问题，孔子曰：“取乎其上，得乎其中；取乎其中，得乎其下；取乎其下，则无所得矣”。参考比你水平高的人的问题也是一种较快的学习方式（没有自恋的意思，如果实在不爽看不惯的话欢迎以后打脸）：

1.`cv::Rect` 的确可以当作参数传到一个 `cv::Mat` 对象中并调用 拷贝构造了？



1. 是的，你理解得没错。在 OpenCV 中，`cv::Rect` 类型的对象可以作为参数传递给 `cv::Mat` 构造函数，用于创建一个表示感兴趣区域（ROI）的子图像。
2. 这是因为 `cv::Mat` 类的构造函数具有多个重载，其中之一允许你指定矩形区域（`cv::Rect` 对象）以及图像的数据类型、数据指针等参数，从而创建一个新的 `cv::Mat` 对象来表示感兴趣区域。
3. 下面是一个示例，演示了如何使用 `cv::Rect` 对象来创建感兴趣区域的子图像（ROI）：
4. `cpp` Copy code
5. 

```
#include <opencv2/opencv.hpp>
int main() { cv::Mat image =
cv::imread(["path_to_your_image.jpg"]); // Define a rectangle to specify the
region of interest (ROI) cv::Rect roi_rect(100, 100, 200, 150); // (x, y, width,
height) // Create a new Mat object representing the ROI cv::Mat roi_image =
image(roi_rect); // Display the original and ROI images
cv::imshow("Original Image", image); cv::imshow("ROI Image",
roi_image); cv::waitKey(0); return 0; }
```
6. 在这个示例中，我们通过创建一个 `cv::Rect` 对象 `roi_rect` 来定义感兴趣区域的矩形位置和尺寸。然后，我们使用 `image(roi_rect)` 来创建一个新的 `cv::Mat` 对象 `roi_image`，该对象表示原始图像中的感兴趣区域。
7. 所以，当你在 `cv::Mat` 的构造函数中传递一个 `cv::Rect` 对象作为参数时，你实际上在创建一个新的 `cv::Mat` 对象，该对象表示原始图像中指定的感兴趣区域。这种方法可以方便地操作图像的局部区域。

## 2. 有没有可能通过 `std::forward` 进行转让而不用拷贝呢？

是的，你可以使用 `std::forward` 来进行转发（forwarding），从而将 `cv::Rect` 对象传递给 `cv::Mat` 构造函数，而不进行额外的拷贝。

`std::forward` 是 C++ 中用于完美转发（perfect forwarding）的工具，通常在模板函数中使用。它可以将参数转发给其他函数，同时保留参数的值类别（lvalue 或 rvalue）和引用修饰等。

下面是一个示例，演示如何在函数中使用 `std::forward` 来传递 `cv::Rect` 对象，以便创建一个感兴趣区域（ROI）的子图像：

cppCopy code

```
#include <opencv2/opencv.hpp> #include <utility> cv::Mat createROI(const
cv::Mat& image, cv::Rect&& rect) { return image(rect); } int main() { cv::Mat
image = cv::imread(["path_to_your_image.jpg"]); cv::Rect roi_rect(100, 100, 200,
150); // (x, y, width, height) cv::Mat roi_image = createROI(image,
std::move(roi_rect)); cv::imshow("ROI Image", roi_image); cv::waitKey(0);
return 0; }
```

在上面的代码中，`createROI` 函数接受一个 `cv::Mat` 对象和一个右值引用的 `cv::Rect` 对象。通过使用 `std::move(roi_rect)`，我们将 `roi_rect` 转换为右值，然后将其传递给 `image(rect)`，从而在 `createROI` 函数内部创建感兴趣区域的子图像。

这样做可以避免额外的拷贝，因为 `createROI` 函数将右值引用传递给了 `image(rect)`，并直接在子图像中使用了这个右值引用。

### 3. 右值引用不会创建额外空间相当于在原对象上直接修改对吗？

是的，你理解得很正确。右值引用允许你在不创建额外内存的情况下，直接访问和修改对象的数据。

当你使用右值引用传递一个对象时，它允许你有效地“窃取”对象的资源或进行移动语义操作，而不是进行拷贝。这在很多情况下可以提高性能和效率。

在上面的示例中，使用 `std::move(roi_rect)` 将 `cv::Rect` 对象转换为右值引用，然后将其传递给 `image(rect)`。这不会创建额外的空间或进行拷贝，而是直接在原始图像数据上创建一个子图像区域。

所以，在右值引用的情况下，不会创建额外的空间，而是在原始对象上进行操作，从而可以有效地实现性能优化。

## 问题和细节

对检测区域内的图进行二值化，也就是预处理，思路就是先将图片用灰度阈值进行二值化，这样子可以将图片中发光的物体给提取出来（装甲板灯条以及日光灯等等），然后再用某种方法将图片中红色或蓝色的区域提取出来，之后再膨胀一下，跟灰度二值化图片逻辑与，最后得到的就是一张只有红/蓝灯条的二值图，在这上面进行接下去的操作。

一种方法是用 RGB 的红蓝通道相减，根据设定的阈值得到一张二值图，这种方法虽好，但是在识别蓝色的时候，有时候无法排除掉日光灯干扰，除此之外，操作简洁，耗时低，国赛我们选用的就是这种方法。

另一种是先将图片转化成 HSV 颜色空间再用通道范围将红色/蓝色提取出来（网上可找到对应的表），这种方法可以排除掉很多干扰，但是近距离的时候装甲板灯条发白，如果膨胀不到位的话会出现灯条断裂的情况，膨胀的卷积核过大又会造成预处理耗时过久，因此要权衡一下。