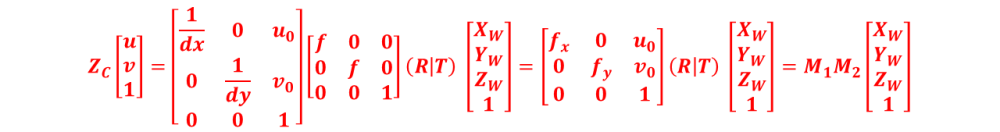
# 视觉感知-3D

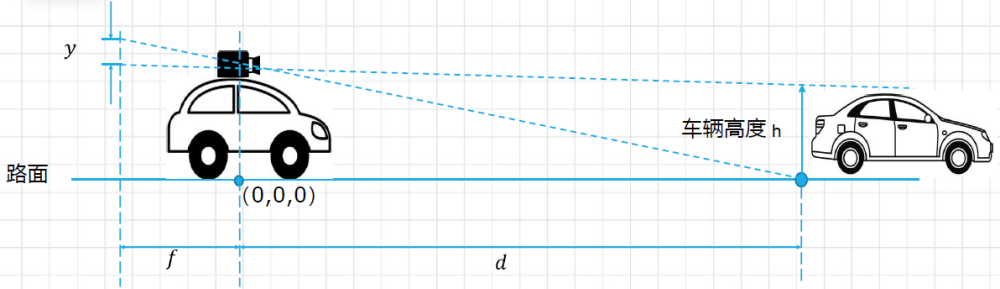
## 相机标定



注意相机内参与外参的表示，也就是从世界坐标系(一般就是雷达坐标系)变换到图像坐标系上时，需要除以Zc进行坐标对齐。

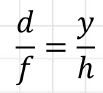
## 深度估计

### 通过车辆高度来计算距离

a

**检测框的下边框中点就是接地点**

距离计算公式:



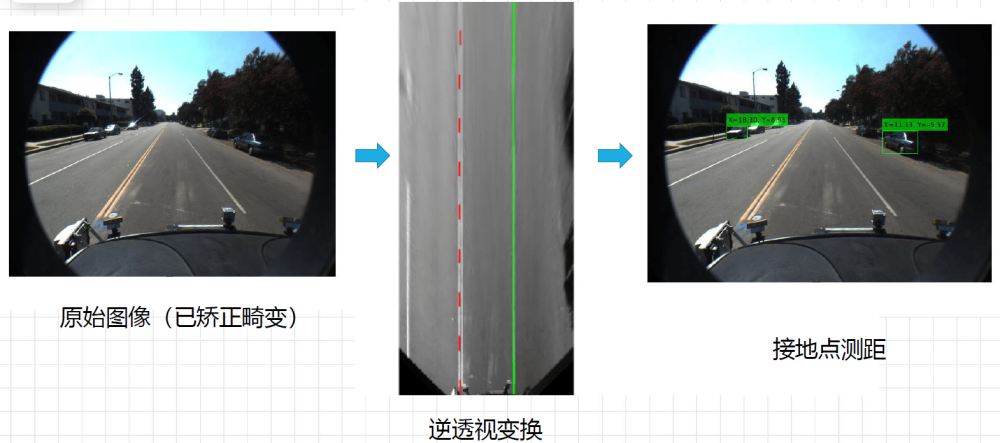
**主要假设:**

• 要求知道车辆的物理高度（可通过神经网络预测）

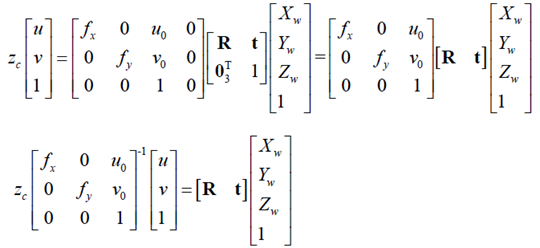
• 误差比例与距离无关，与车辆高度精度、检测框误差有关

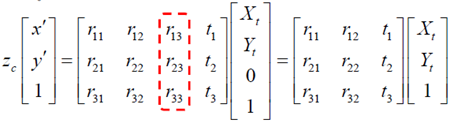
• 可以与IPM方法相结合使用，难点在于车辆高度的预测

### IPM（逆透视变换）



这里主要使用了单应矩阵的理论。假设路面上是一个平面Z面，则就可以根据计算单应矩阵的方法计算出距离。单应矩阵本身就是将一个平面上的点，映射到图像中。





**单应矩阵的特性:**

* 平面假设，所有的像素都在同一平面（地面）

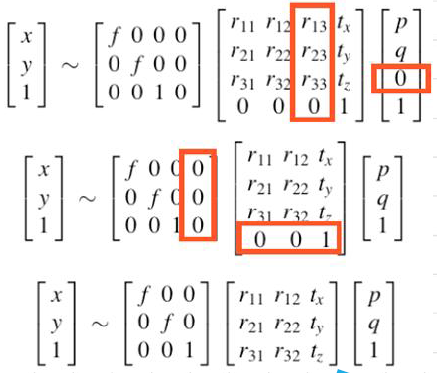
* 只需要4个点，就可以求解出变换矩阵

* 已知相机内外参情况下可直接给出变换矩阵

**• IPM相当于将图像上的像素，转换到地面的平面，因此Z轴为0**

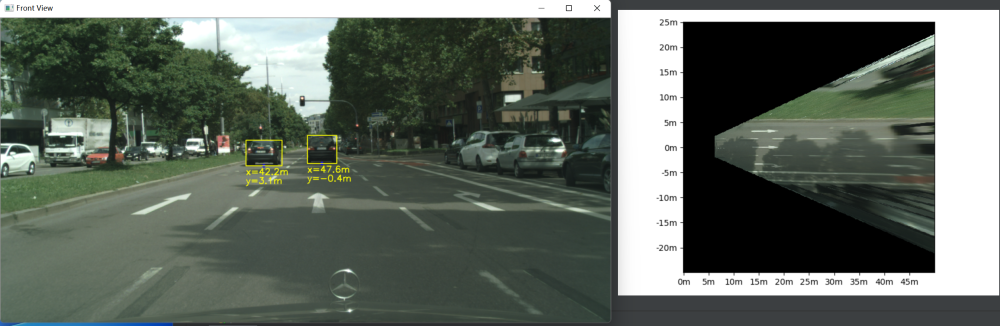
• 实际的车辆运行过程中，外参会发生变化因此需要用矫正之后的外参（在线标定）

代码简化过程:

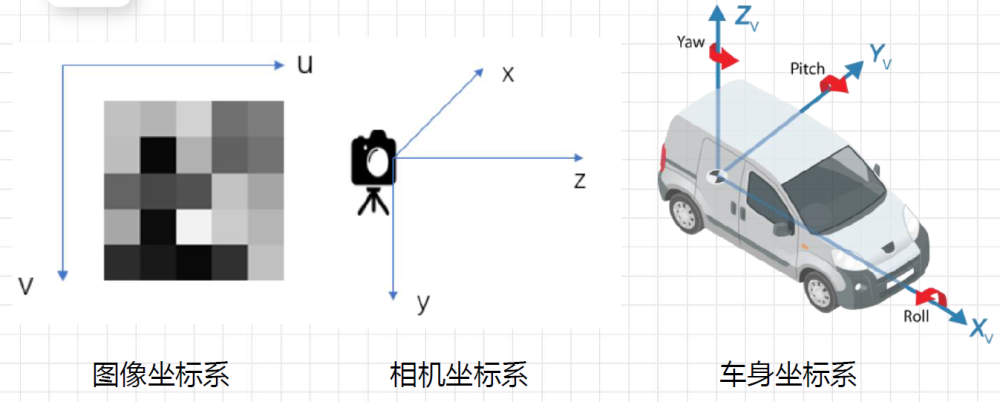


这里的推到过程中，需要除以距离Zc，**但是有一个平面约束，因此可以得到准确的值**。

import json  
import cv2  
import matplotlib.pyplot as plt  
import numpy as np  
  
  
def rotation\_from\_euler(roll=0., pitch=0., yaw=0.):  
 sr, sp, sy = np.sin(roll), np.sin(pitch), np.sin(yaw)  
 cr, cp, cy = np.cos(roll), np.cos(pitch), np.cos(yaw)  
 R1 = np.array([  
 [cy, -sy, 0],  
 [sy, cy, 0],  
 [0, 0, 1]  
 ])  
 R2 = np.array([  
 [cp, 0, -sp],  
 [0, 1, 0],  
 [sp, 0, cp]  
 ])  
 R3 = np.array([  
 [1, 0, 0],  
 [0, cr, -sr],  
 [0, sr, cr]  
 ])  
 R = np.identity(4)  
 R[:3, :3] = R1 @ R2 @ R3  
 return R  
  
  
def translation\_matrix(vector):  
 M = np.identity(4)  
 M[:3, 3] = vector[:3]  
 return M  
  
  
def load\_camera\_params(file):  
 with open(file, 'rt') as handle:  
 p = json.load(handle)  
  
 fx, fy = p['fx'], p['fy']  
 u0, v0 = p['u0'], p['v0']  
  
 pitch, roll, yaw = p['pitch'], p['roll'], p['yaw'] # 旋转角度  
 x, y, z = p['x'], p['y'], p['z']  
  
 # Intrinsic  
 K = np.array([[fx, 0, u0],  
 [0, fy, v0],  
 [0, 0, 1]]) # 相机内参矩阵  
  
 # Extrinsic  
 # 外参是相机相对于车身坐标的偏移，需要转成车身相对于相机的偏移  
 R\_cam2veh = rotation\_from\_euler(roll, pitch, yaw) # 正交矩阵  
 R\_veh2cam = np.transpose(R\_cam2veh) # 正交矩阵的逆等于转置  
 T\_veh2cam = translation\_matrix((-x, -y, -z))  
  
 # Rotate to camera coordinates  
 R = np.array([[0., -1., 0., 0.],  
 [0., 0., -1., 0.],  
 [1., 0., 0., 0.],  
 [0., 0., 0., 1.]]) # 这里主要是由于车辆自身坐标轴与相机坐标系之间的关系决定的  
 RT = R @ R\_veh2cam @ T\_veh2cam  
 R = RT[:3, :3]  
 T = RT[:3, 3:]  
 return K, R, T  
  
  
def get\_homography(K, R, T):  
 RT1 = np.concatenate((R[:, :2], T), axis=1)  
 Hi = K @ RT1  
 H = np.linalg.inv(Hi)  
 return H  
  
  
def ipm(coord, K, R, T):  
 H = get\_homography(K, R, T)  
 xyz = H @ coord  
 xyz = xyz / xyz[2, 0] # 归一化坐标  
 return xyz[:, 0]  
  
  
def draw\_ipm(image, K, R, T, target\_w, target\_h):  
 H = get\_homography(K, R, T)  
  
 K\_top = np.array([ # H矩阵是将图像中的坐标映射到了地面上，还需要将地面进行BEV视角下的变换  
 [10, 0, 0], # 每10个像素代表1米  
 [0, -10, 250], # y需要翻转一下，图像的y轴从上到下，而车坐标y轴从下到上  
 [0, 0, 1]])  
  
 H = K\_top @ H  
 warped = cv2.warpPerspective(image, H, (target\_w, target\_h),  
 flags=cv2.INTER\_LINEAR,  
 borderMode=cv2.BORDER\_CONSTANT,  
 borderValue=0)  
 return warped  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 # Retrieve camera parameters  
 image = cv2.imread('stuttgart\_01\_000000\_003715\_leftImg8bit.png')  
 TARGET\_H, TARGET\_W = 500, 500  
 K, R, T = load\_camera\_params('camera.json')  
  
 # Warp the image  
 warped = draw\_ipm(image, K, R, T, TARGET\_W, TARGET\_H)  
  
 # roi = cv2.selectROI("please select a car.", image, False, False)  
 for roi in [(823, 410, 121, 86), (1029, 394, 99, 94)]:  
 x, y, w, h = roi  
 xc = int(x + w \* 0.5)  
 yc = y + h  
 bottom\_center = np.array([[xc], [yc], [1.0]])  
 depth, dy, \_ = ipm(bottom\_center, K, R, T)  
 print(depth, dy)  
  
 cv2.circle(image, (xc, yc), 3, (255, 0, 0), 3)  
 cv2.rectangle(image, (x, y), (x+w, y+h), (0, 255, 255), 2)  
 cv2.putText(image, f"x={depth:.1f}m", (x, y + h + 30),  
 cv2.FONT\_HERSHEY\_SIMPLEX, 1.0, (0, 255, 255), 2)  
 cv2.putText(image, f"y={dy:.1f}m", (x, y + h + 60),  
 cv2.FONT\_HERSHEY\_SIMPLEX, 1.0, (0, 255, 255), 2)  
 cv2.imshow("Front View", cv2.resize(image, (image.shape[1]//2, image.shape[0]//2)))  
 cv2.waitKey(0)  
 plt.imshow(cv2.cvtColor(warped, cv2.COLOR\_BGR2RGB))  
 plt.xticks(np.arange(0, 500, 50), [str(x) + "m" for x in np.arange(0, 50, 5)])  
 plt.yticks(np.arange(0, 500, 50), [str(x) + "m" for x in -np.arange(-25, 25, 5)])  
 plt.tight\_layout()  
 plt.show()

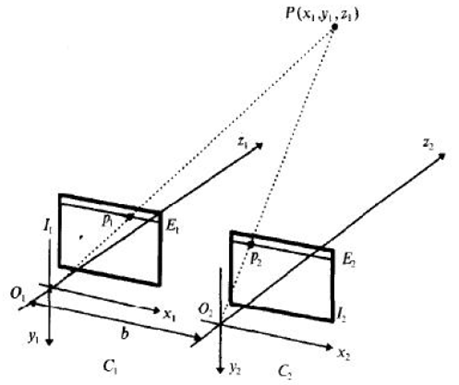


可以直接完成视角变换，具体见代码。涉及到的坐标系的方向如下图所示:

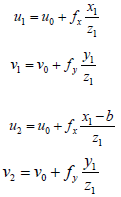


• 注意点：**车身坐标向前是X轴，向左是Y轴**

### 双目相机测距



从上面可以得到四个公式:如下所示:



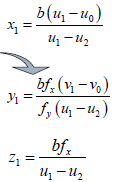
这里的u1和v1是点P在左相机的像平面的像素坐标。u2跟v2也是一样的。

这里因为点P在左相机的坐标为P(x,y,z)而在右相机的坐标为P(x-b,y,z)

**这里我们有三个未知数，有四个方程，因此可以求解出来P点的坐标的。**

**空间点三维座标位置求解**

空间点坐标



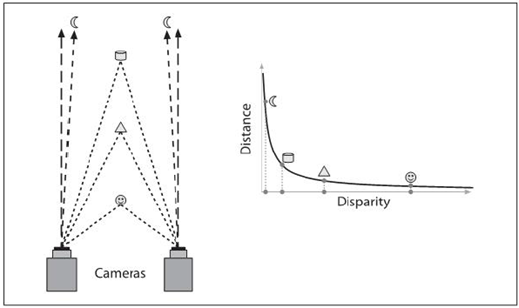
这里的b就是两个相机之间的连线的长度，也就是基线长度。

u1-u2就是视差，就是同一个点在两个相机像平面的x轴的差别。

视差和深度成反比关系

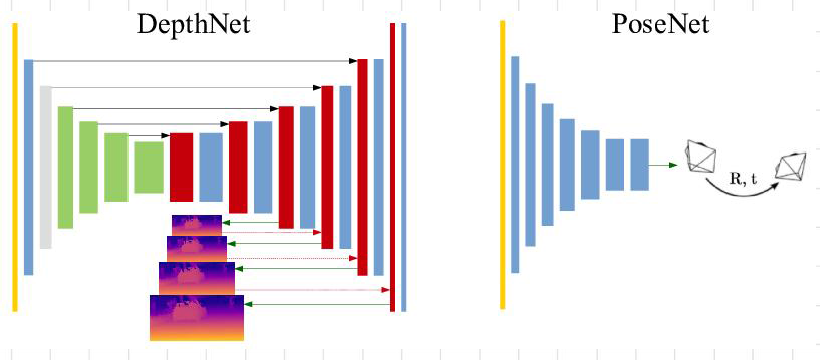
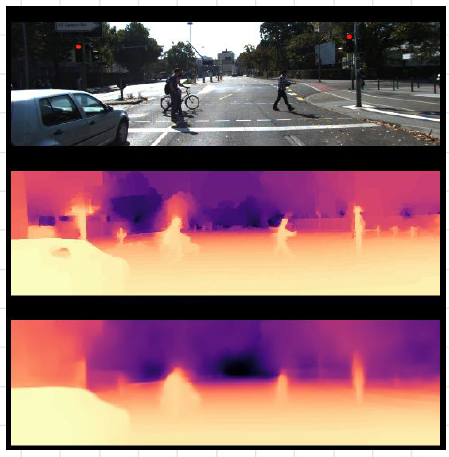


其实这句话的意思就是如果物体离我越远，就是z1越大，也就是z1-z2越小，也就是越不能准确分辨出来。同样的，如果我们想分辨率高一些，那么可以增加b，也就是基线的长度。U1和u2是以像素为单位的，因此小于一个像素的视差是识别不了的。



**通过极线搜索的方式，可以加速双目立体匹配。**

#### 单目深度估计网络——监督学习

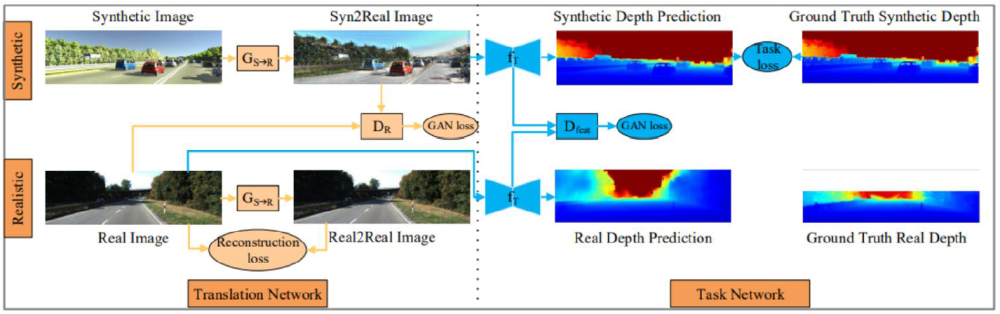
 

• 直接通过CNN来学习图像每个像素的深度（距离）

• 需要有带有深度信息的图像作为训练数据，一般通过**激光雷达或者RGBD相机**获取真值

• 将**深度估计看作是一个回归问题或者分类**问题

#### 单目深度估计——风格迁移

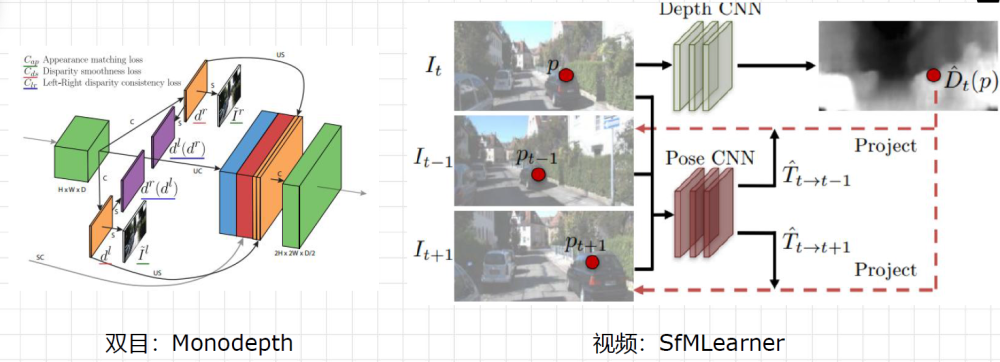


**T2Net**

• 使用GAN网络，利用合成图像（深度已知）与真实图像相结合来实现无监督训练

• 优点是低成本获取大量带有depth的图像，并同时利用真实图像

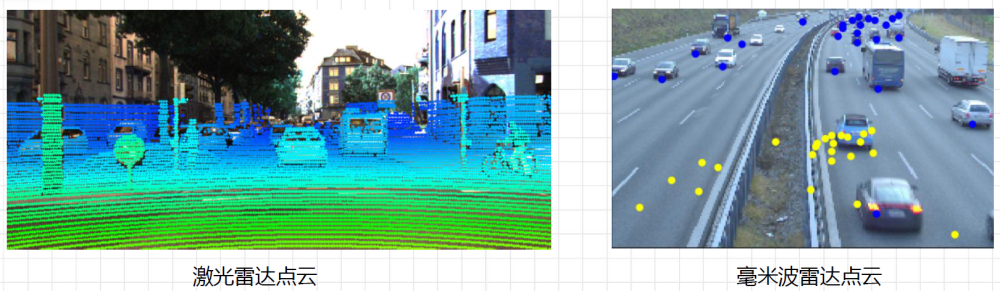
#### 单目深度估计——无监督学习



• 使用双目或者视频序列来做无监督学习

**• 利用网络估计出的相机pose和深度，将一张图片投影到另一张图片，并计算投影图像与真实图像的光度误差**

#### 融合雷达测距

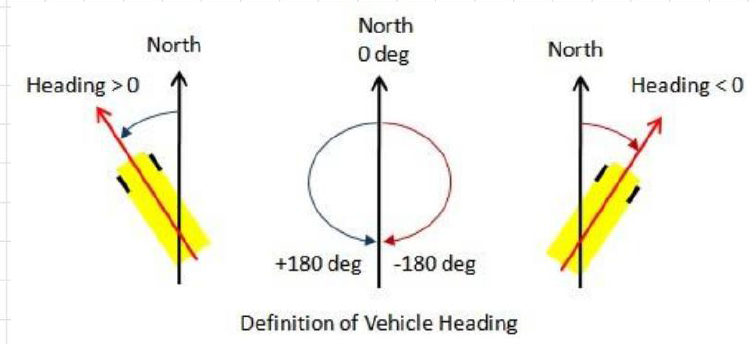


• 将雷达的点云投影到图像，并将检测框内的点取出做平均（或用直方图法）

• 要求雷达和图像时间戳对齐

• 对于行人、自行车等目标，容易将背景点包含进来（**可使用图像分割来解决**）

#### 航向角估计



• 2D转3D方法中，航向角估计是一个必要的输出

• 航向角对于下游预测、规划控制任务至关重要

• 航向角也用来判断车辆是否正在切入、切出