中国研究生创新实践系列大赛

“华为杯”第十六届中国研究生

数学建模竞赛

学　　校	上海交通大学
参赛队号	19102480006
队员姓名	1.程宇豪
	2.戴俊宇
	3.邵华青

中国研究生创新实践系列大赛

“华为杯”第十六届中国研究生

数学建模竞赛

题 目 视觉情报信息分析

摘 要：

本文对视觉情报信息分析问题进行探索和研究。本文分别建立了针对单目摄像头所拍摄的各类单张图片、手机拍摄视频、无人机拍摄视频的空间分析模型，解决了不同的空间距离计算问题。针对该类问题，我们采取消隐点测距法、单双目成像转换法、单目视觉 SLAM 算法、三维重建中的 SFM 算法等算法，不断对模型进行改进和完善。本文算法应用面广泛、算法效果优良，能够较为完美地解决问题中所提到的各类空间距离计算问题。

针对问题一，我们将其视为基于单目图像的空间距离计算问题。本任务根据不同图片的性质和测量目标，分别采用透视变换法和消隐点检测法进行测量，并使用基于深度学习的深度估计算法进行结果印证。透视变换法的思想是根据参照物，测量与之平行的物体的长度比例，然后根据参照物的尺度反推出该物体的真实长度。消隐点测距法则根据地平线消隐点的性质计算三维空间内的距离。针对图一，利用消隐点检测法可得 A 车车头与 B 车车头之间的距离为 28.38 米、拍照者对于马路左侧的距离为 12.07 米。针对图二，利用消隐点检测法

可得 A 车车头与 C 车车尾之间的距离为 28.87 米、拍照者距离 B 车车头的距离为 20.63 米。针对图三，利用消隐点检测法可得拍照者距岗亭 A 的距离为 28.57 米、利用透视变换法可得拍照者距离地面的高度 4.43 米。针对图四，利用消隐点检测法可得 AB=5.04 米、利用透视变换法 CD=3.88 米、CD 与 AB 之间距离为 7.49 米。

针对问题二，我们将其视为三维空间坐标转换问题。本任务首先通过先验的尺寸、镜面矫正原理和平面成像原理求解出了相机的内参，再通过内参可以算出图片中点在原三维坐标下的位置，从而可以计算得出红车与本车之间的距离。之后根据空间位置关系的观察和计算可得速度差。估算得该车和后方红色车辆之间的距离为 49.52 米，估算该车超越第一辆白色车辆时两车的速度差异为 6.76 米/秒。

针对问题三，我们将其视为单目图像转双目图像求解问题。本任务首先通过铁路接触网的长度与时间的先验值来求解出高铁的速度，进而求解出河面的宽度。之后我们进行了合理的假设，在连续图像中我们可将单目图像转换为同等参数的双目图像，进而估计相机参数、将问题简化为双目视角下的测距。估算得测算高铁行驶方向左侧第一座桥桥面距水面的高度为 9.12 米、距高铁轨道的距离为 487. 98 米以及水面宽度为 224.53 米。

针对问题四，我们利用三维重建算法与单目视觉 SLAM 算法相结合的松耦合算法，利用该算法对点云进行降噪，能够更加明显地可视化出其结构特性。估算得（1）估算环绕老宅道路的长度 346.55 米、宽度 3.39 米、望楼高 9.63 米、后花园中树木的最大高度 13.3 米、横

向房屋高 6.32 米、纵向房屋高 6.81 米；（2）估算该老宅的占地面积为 8312 平方米；（3）测算无人机最高高度为 56.44 米、无人机最低高度为 43.18 米、速度为 5.54 米/秒。

关键词：消隐点测距法 单双目成像转换法 单目视觉 SLAM 算法 SFM 算法

目录

一、问题重述

1.1 背景描述

研究表明，一般人所获取的信息大约有 80%来自视觉。视觉信息的主要载体是图像和视频，视觉情报指的是通过图像或者视频获取的情报。

从图像或视频中提取物体的大小、距离、速度等信息是视觉情报分析工作的重要内容之一，如在新中国最著名的“照片泄密案”中，日本情报专家就是通过《中国画报》的一幅封面照片解开了大庆油田的秘密。在当前很热门的移动机器人、无人驾驶、计算机视觉、无人机侦察等领域，更是存在着大量的应用需求。尽管在对未来智能交通系统的设计等工作中，科研人员正在研究使用双目或多目视觉系统或者特殊配置的单目视觉系统获取相关信息，但在某些特定条件下，分析人员所能利用的，只能是普通的图像或视频，其中的信息需要综合考虑各种因素，通过合适的数学模型来提取。

1.2 要解决的问题

本题从实际需求出发，选择单幅图像距离信息分析、平面视频距离信息分析和立体视频距离信息分析几个典型场景，提出如下四项任务：

任务 1：测算图 1-1 中红色车辆 A 车头和白色车辆 B 车头之间的距离、拍照者距马路左侧边界的距离；图 1-2 中黑色车辆 A 车头和灰色车辆 C 车尾之间的距离以及拍照者距白色车辆 B 车头的距离；图 1-3 中拍照者距岗亭 A 的距离以及拍照者距离地面的高度；图 1-4 中

塔体正面(图中四边形 ABCD)的尺寸，即 AB 和 CD 的长度以及 AB 和 CD 之间的距离（已知地砖尺寸为 80cm×80cm)。

图 1-1

图 1-2

图 1-3

图 1-4

任务 2： 附件“车辆.mp4”(右键点击后选择“保存到文件”可导出视频文件）是别克英朗 2016 款车上乘客通过后视镜拍摄的视频。（1）估算该车和后方红色车辆之间的距离；（2）估算该车超越第一辆白色车辆时两车的速度差异。

任务 3：附件“水面.mp4”是高铁乘客拍摄的一块水面，测算高铁行驶方向左侧第一座桥桥面距水面的高度、距高铁轨道的距离以及水面宽度，估算拍摄时高铁的行驶速度。

任务 4： 附件“无人机拍庄园.mp4”记录了某老宅的全景。（1）

估算其中环绕老宅道路的长度、宽度、各建筑物的高度、后花园中树木的最大高度；（2）估算该老宅的占地面积；（3）测算无人机的飞行高度和速度。

二、符号说明

符号	意义
f	相机焦距
P_w	三维空间中的坐标
P_{uv}	图像中坐标
$\boldsymbol{K}$	相机内参矩阵
$\boldsymbol{T}$	相机外参矩阵
$I(x,y)$	图像灰度表达式
m_{pq}	ORB 特征点矩阵
R	相机交换矩阵

三、任务一

3.1 任务分析

任务 1：测算图 1-1 中红色车辆 A 车头和白色车辆 B 车头之间的距离、拍照者距马路左侧边界的距离；图 1-2 中黑色车辆 A 车头和灰色车辆 C 车尾之间的距离以及拍照者距白色车辆 B 车头的距离；图 1-3 中拍照者距岗亭 A 的距离以及拍照者距离地面的高度；图 1-4 中塔体正面(图中四边形 ABCD)的尺寸，即 AB 和 CD 的长度以及 AB 和 CD 之间的距离（已知地砖尺寸为 80cm×80cm)。

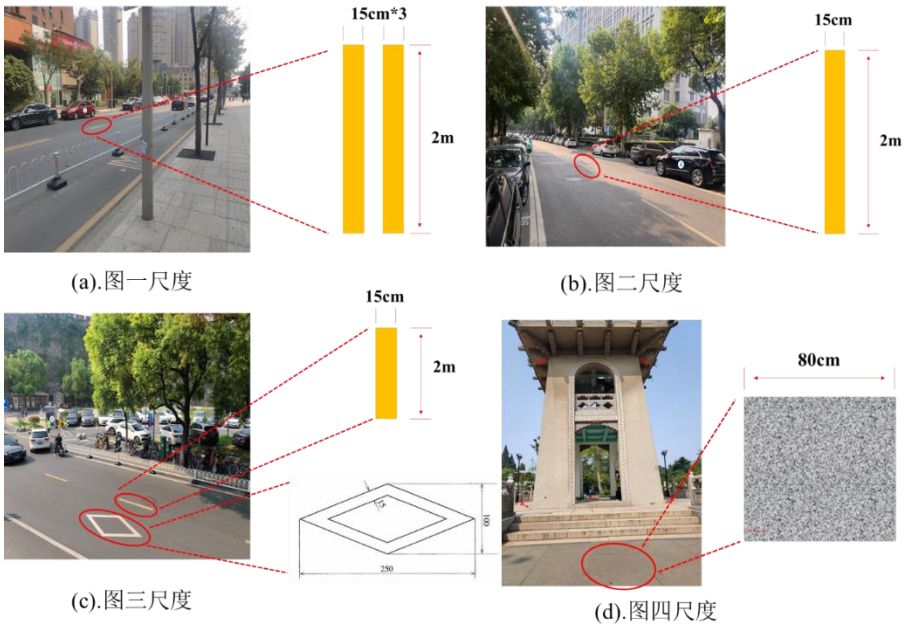
该任务的目的在于没有过多先验信息的情况下，如何根据单张图片推测并获得我们感兴趣的距离信息，是一个单目图片距离测量问题。基于单目距离的测量方法有深度估计法、透视变换法、消隐点测距法等，本任务根据不同图片的性质和测量目标，分别采用透视变换法和消隐点检测法进行测量。

任务可以具体划分为两个目标：求解图片中某两点的真实距离；求解拍摄者与图片中某点的真实距离(拍摄者的高度可以看作到垂直点的距离)。首先需要从图片中选择具有固定长度的物体作为标定物，以其真实长度和像素长度的比例关系作为尺度，然后建立模型并建立感兴趣的两点间像素距离与标定物像素距离的映射关系，根据尺度求解最终结果。

按照上述求解流程，首先需要从各图中选择合适的标定物，如图 3-1 所示，(a)可以选择双黄线的距离和宽度作为尺度，(b)可以选择单黄线的距离和宽度作为尺度，(c)可以选择单黄线及人行横道预告标线的规格作为尺度，(d)可以使用图中已知的台阶下方地砖长宽作为尺度。同时针对图中无法给定的尺度值，如图 3-1(b)中拍摄者距离

旁边车后视镜的位置，可以根据实际场景重现，得到一个合理估计值作为尺度。

根据目标的特性，图一、图二的四个测量目标以及图三的拍照者与岗亭 A 的距离使用消隐点检测法测算，而图三拍照者的高度和图四的两个观测目标可以使用透视变换法测算。



(a).图一尺度　(b).图二尺度

(c).图三尺度　(d).图四尺度

图 3-1

3.2 模型建立

具体两种模型建立方法原理如下：

（1）透视变换法的思想是根据参照物，测量与之平行的物体的长度比例，然后根据参照物的尺度反推出该物体的真实长度。透视变换法的流程为从原图中选择一个真实场景中的矩形，然后根据其角上的四个点，通过透视变换矩阵将图片转化成平面图，如下图 3-2(a)所示，将原本处于立体空间中的矩形转化成平面中的矩形。可以直观的理解为视角上的转换，如图 3-2(b)所示，从转换后的视角 P1，由于

转化后的矩形长宽可以预设，所以其纵向 AB 和 A’B’并不能反映其真实世界中的比例关系，但是横向的 CD 和 C’D’的比例关系是不会更改的，所以可以从其比例关系以及 CD 在真实世界中的长度求得 C’D’的真实世界长度。透视变换之后，视角垂直方向上的平行关系和尺度比例不变。其关键点在于求解透视变换的转化矩阵。

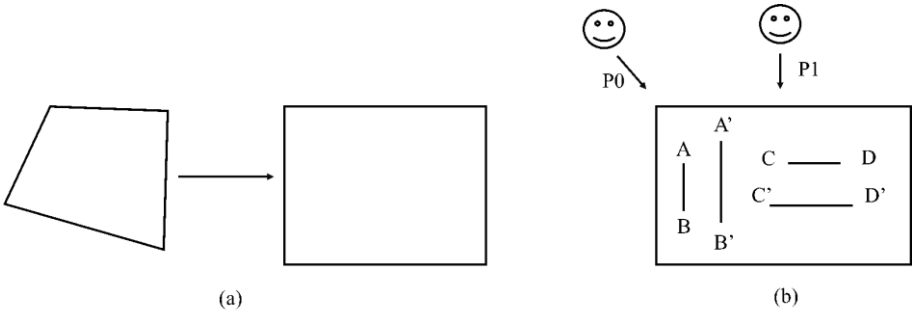


图 3-1

（2）消隐点测距法主要由以下两个原理构成：（a）消隐点是指原本在三维空间平行的直线，在投影到二维空间之后，会在极远处相交于一点，相交点为消隐点。消隐点是无穷远的，并且所有的消隐点在二维图像上构成一条无穷远的直线，即地平线。从这条直线上任意一点引出两条直线，这两条直线是平行的。（b）摄像头将三维空间投射到二维图像上，四个点的直线交比不变，即如下图 3-3 所示。

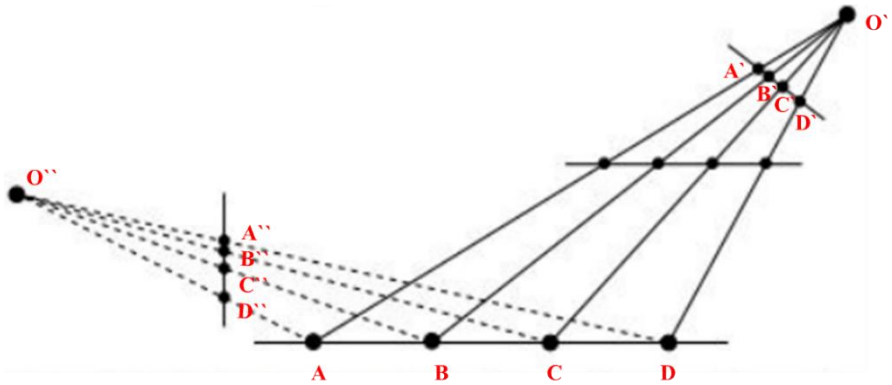


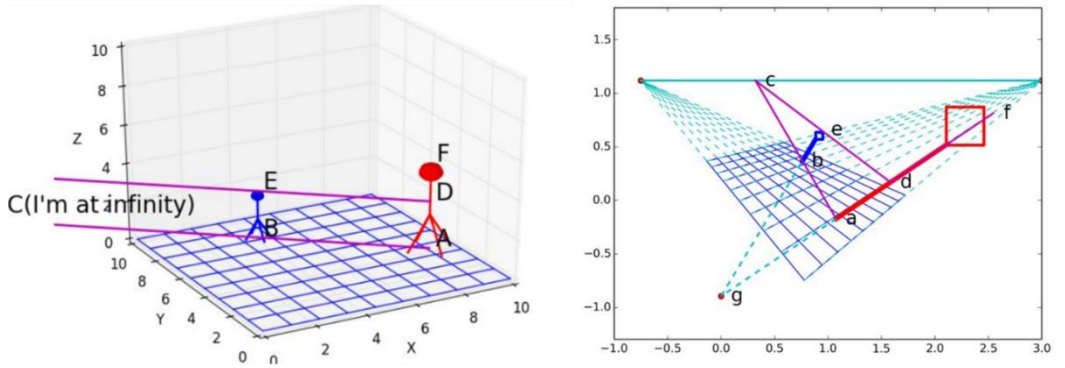
图 3-2

图中 A、B、C、D 四个点是原三维空间中的在同一条直线上的四

个点，点 O`和 O``是照相机的镜头，A`B`C`D`和 A``B``C``D``分别是 ABCD 在镜头 O`和 O``中成像的点。这些点满足如下关系:

$$(AB\times CD)/(AC\times AD)=(A'B'\times C'D')/(A'C'\times A'D')=(A"B"\times C"D")/(A"C"\times A"D")$$

根据上述原理，可以从任意的二维图像中选取位于同一平面上的两对平行线计算出待测平行线段长度的比值关系，并且根据其中一条线段的先验尺度和二维图像中的像素比例关系，可以反推出另外一条线段的实际长度。如图 3-4(a)所示，用大写来表示原来的图中的点，小写来表示新的点。我们假设已知 BE 长度，BE 与 AF 平行，要求的是 AF 对应的长度。



(a) (b)

图 3-3

其算法流程如下：

（1）首先在二维图像中(如图 3-4(b))使用网格点（或者两对平行的线）找到两个消失点，并且连接两个消失点，找到地平线。

（2）将 ab 相连，与地平线交于 c，连接 ce 并延长交 af 于 d，可知 ca 平行于 cd，又 be 平行于 ad，所以 ad 与 be 等长。

（3）将 be 与 af 延长相交于 g 点，这个 g 点也是消失点。所以利用公式(1)有$(AD/AF)/(GD/GF)=(ad/af)/(gd/gf)$，并且由于 G 无

穷远，可以得到 AG/DG 等于 1，所以 $AD/AF = (ad/af)/(gd/gf)$。其中，二维图像中两点的距离可以使用点的像素关系求解 $D_{ab} = \sqrt{(xa-xb)^2+(ya-yb)^2}$。通过以上数值解法，即能够求到 AF 的值。

3.3 模型计算

对于任务一中的四个图，本节使用以下规则来表示线段的意义：

通过先验的同一个平面中两对平行线延长相交于消隐点，并且连接两个消隐点的线段使用蓝色线段表示；通过不同的预估的物体尺寸来表示不同平行线间的尺寸大小使用其他不同的颜色来表示，本节中使用了黄色线段和红色线段来表示不同角度的平行线测量。

对于图 3-5，车辆 A 与车辆 B 之间的距离由 AB 表示，由上一小节中解题方法可得：$AB = AD*(ab*dc)/(ad*bc)$，其中 AD = EF = 2 米，ab、dc、ad、bc 的值可由二维图像中的像素点位置求出，可以求出 AB 为 28.38 米，即 A 车车头与 B 车车头之间的距离为 28.38 米。

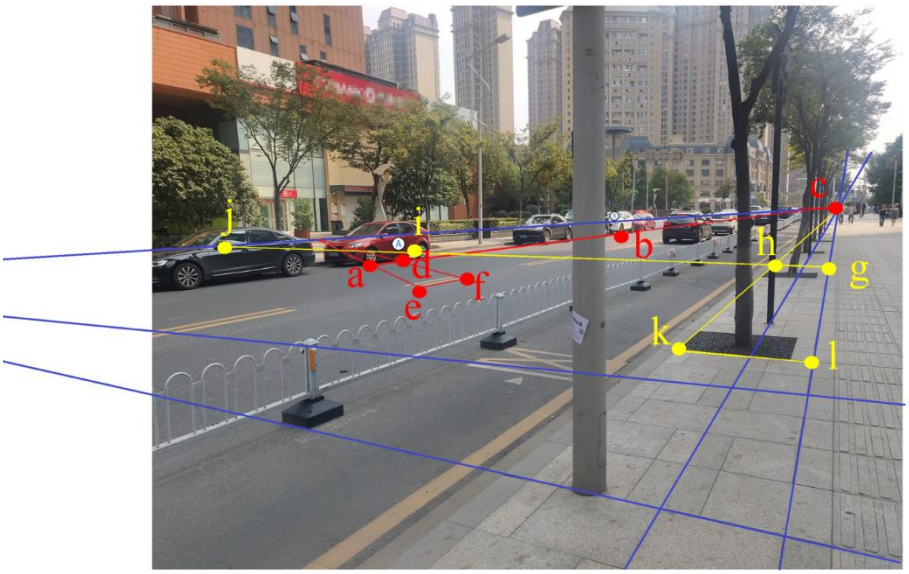


图 3-5

观察者距离马路左侧边界用 GI 来表示，通过消隐点测距法可以

得到：$GI = GH*(gi*hj)/(gh*ij)$，其中我们通过尝试估计 hg 长度约为 0.65 米，gi，hj，gh，ij 长度可由像素点位置代换算出，所以得出 GI 的长度为 12.07 米，即拍照者对于马路左侧的距离为 12.07 米。

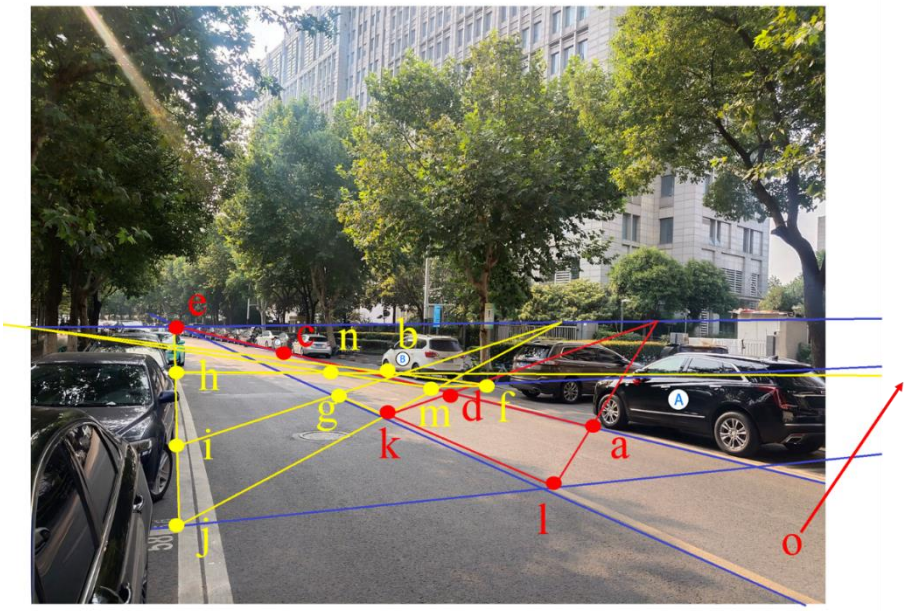


图 3-6

对于图 3-6，A 车车头与 C 车车尾之间的距离用 AC 表示，通过消隐点测距法可以得出 $AC = AD\times(ac\times de)/(ad\times ce)$，其中 AD=KL=4 米，所以求得 AC=28.87 米；拍照者距离 b 车车头的距离，由于没有平行于 B 车车头与拍照者的线，并且拍照者没有出现在图片中，所以通过测量 B 到左侧街道 H 的距离与 H 到 J 的距离和估计拍照者到 J 点的距离来求得。通过消隐点测距法可以得到 $BH = BN*(bh*no)/(bn*ho)$，其中 $BN = MF$，通过对 SUV 宽度与车位宽度的估计，估计 MF=2 米，因此 BH=7.56 米。又可以求得 $JH = IJ*(jh*ie)/(ij*he)$，其中 IJ=GK=2 米，所以求得 HJ=14.68 米。根据场景重现，估计拍照者站在后视镜向车头约 0.5 米的地方，所以估计拍照者距离 j 的距离 4.5 米。利用上述条

件，拍照者距离 B 车车头的距离为$\sqrt{(4.5+JH)^2+BH^2}$，解出为 20.63 米。

对于图 3-7，由于拍照者站在高处，且拍照者对于地面的投影点无法直接得知，所以本文使用 AK 的距离加上估计 K 到拍照者的距离来估计拍照者距离岗亭 A 的距离。AK 的距离使用勾股定理，由 K 对应对面的点 D 来间接求得 $AD=\sqrt{AD^2+KD^2}$。其中 $AD = CD\times(ad\times cb)/(cd\times ab)$，CD=EF=2 米，所以求得 AD=18.93 米；同理 $KD=KH\times(kd\times gh)/(kh\times dg)$，KH=IJ=1 米，所以求得 KD=11.34 米，所以得出 AD=22.07 米。又我们通过场景重现估计出 K 点距离拍照者的距离为 6.5 米，所以求得拍照者距离岗亭 A 距离为 28.57 米。

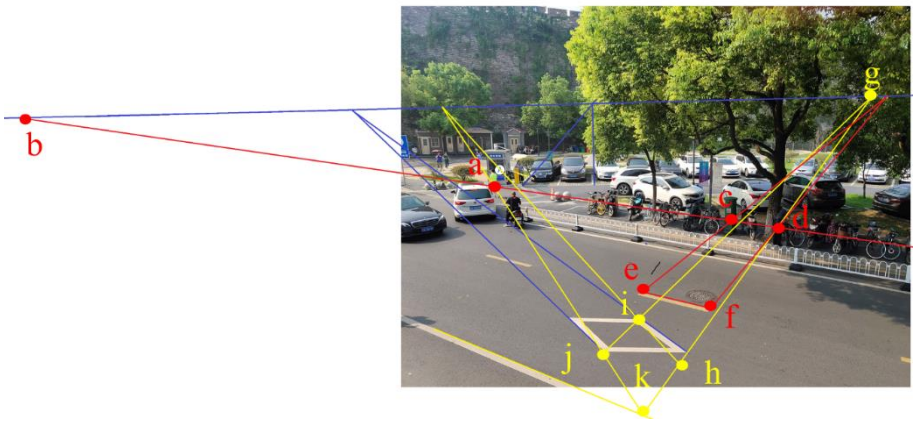
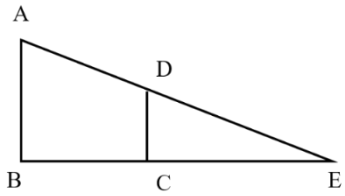


图 3-7

由于无法找到拍照者在图中的位置，所以无法使用消隐点测距法测得拍照者的高度，转而使用透视变换法。首先对面的栏杆与拍照者是平行的，所以对图片做关于栏杆的平面的透视变换，得到图 3-8(b)。对应的角度估计有如 3-8(a)的关系，$AB=CD\times BE/CE$。对面栏杆映射到地面上，通过估计 CE 约为 1.8 米，由原图估计左边车行道宽度为 1.8 米，上述求得道路宽为 7.8 米，因此 BE=11.4 米。通过查询资料

栏杆 DC=0.7 米，所以求得拍照者高度（镜头离地面高度）为 4.43 米。



(a)

(b)

图 3-8

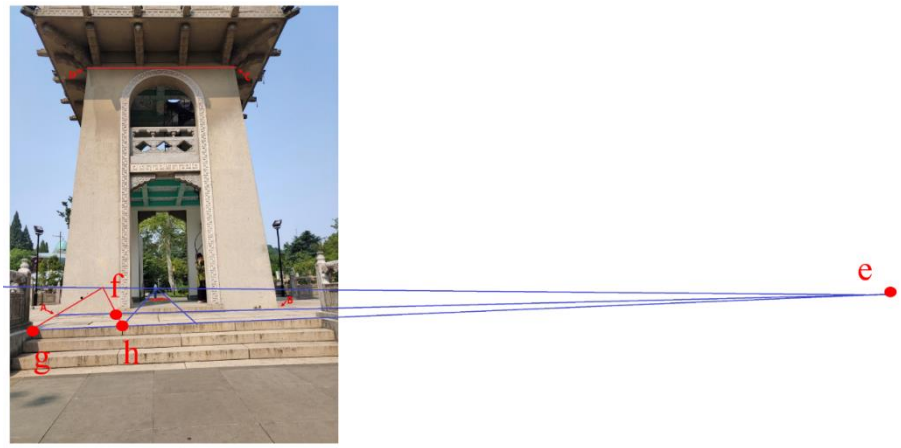


图 3-9

对于图 3-9，前面的地砖的尺寸为 80cm*80cm，所以估计一块台阶石头长度为 1.6 米。AF=GH=1.6 米，$AB = AF \times (ab \times fe)/(af \times be)$，由消隐点测距法求得 AB=5.04 米。由于 CD 与地面平行，且无足够的平行线对可用，所以这里使用透视变换求得。图像对于平面 ABCD 的透视变换得到图 3-10。在图 3-10 中，由于透视变换，有以下关系存在：CD/AB=cd/ab，所以求得 CD=3.88 米。对于 AB 与 CD 之间的距离，本节使用人身高来进行估计预测。根据中国女性平均身高，估计人高度为 1.6 米。CD 与 AB 之间距离和人身高在图 3-10 中成正比，所以求

得 CD 与 AB 之间距离为 7.49 米。

图 3-4

3.4 模型评价

使用 3.1 小节所示的尺度信息及 3.2 小节的模型求解后，任务一的结果如表 3-1 所示。本小节根据图像特性，在图像中选择真实场景中平行存在的线条求解图像中，某平面的消隐点，然后根据消隐点及线段的几何关系推导出待求值，从结果分析取得了较为不错的效果，而针对使用不便使用消隐点测距法的情况，我们也是用透视变换法进行了求解。

表 3-1 任务一结果

图一	A 车头和 B 车头之间的距离	28.38 米
	拍照者距马路左侧边界的距离	12.07 米
图二	A 车头和 C 车尾之间的距离	28.87 米
	拍照者距 B 车头的距离	20.63 米
图三	拍照者距岗亭 A 的距离	28.57 米
	拍照者距离地面的高度	4.43 米
图四	AB 的长度	5.04 米
	CD 的长度	3.88 米
	AB 和 CD 之间的距离	7.49 米

四、任务二

4.1 任务分析

任务 2：附件“车辆.mp4”（右键点击后选择“保存到文件”可导出视频文件）是别克英朗 2016 款车上乘客通过后视镜拍摄的视频。（1）估算该车和后方红色车辆之间的距离；（2）估算该车超越第一辆白色车辆时两车的速度差异。

该任务同前任务一，为无先验信息情况下得单目视觉信息提取。所不同的是本任务的视觉信息是视频，具有多帧图像，而相对于任务一具有更多的信息。但是由于相机处于运动状态，所以无法利用问题一当中的建模方法求解。但是由于先验知道拍照者所坐的车型，所以可以根据拍照者与画面中的固定深度物体的像素坐标，根据多帧图片计算出相机的内参，然后根据相机内参和多帧图片建立在该相机坐标轴下的地面平面坐标。最后根据待测物体的像素坐标，求解出在地面平面上的坐标。

4.2 模型建立

相机将三维世界中的坐标点（单位为米）映射到二维图像平面（单位为像素）的过程能够用一个几何模型进行描述。这个模型有很多种，其中最简单的成为针孔模型，针孔模型是很常用且有效的模型，它描述了一个一束光线通过针孔之后，在针孔背面投射成像的关系。该模型能把外部的三维点投影到相机内部成像平面，构成相机的内参数。

小孔模型能够把三维世界中的物体投影到一个二维成像平面，可以用简单的模型来解释相机的成像过程，如图 4-1 所示。对针孔模型进行几何建模，设$O-x-y-z$为相机坐标系，习惯上我们让z轴指向相机前方，x向右，y向下。O为摄像机的光心，也是针孔模型中的

针孔。现实世界的空间点P，经过小孔O投影之后，落在物理成像平面$O'-x'-y'$上，成像点为P'。设P的坐标为$[X,Y,Z]^T$，P'为$[X',Y',Z']^T$，并且设物理成像平面到小孔的距离为f（焦距）。那么，根据三角形相似关系，有：

$$\frac{Z}{f}=-\frac{X}{X'}=-\frac{Y}{Y'}$$

小孔成像模型

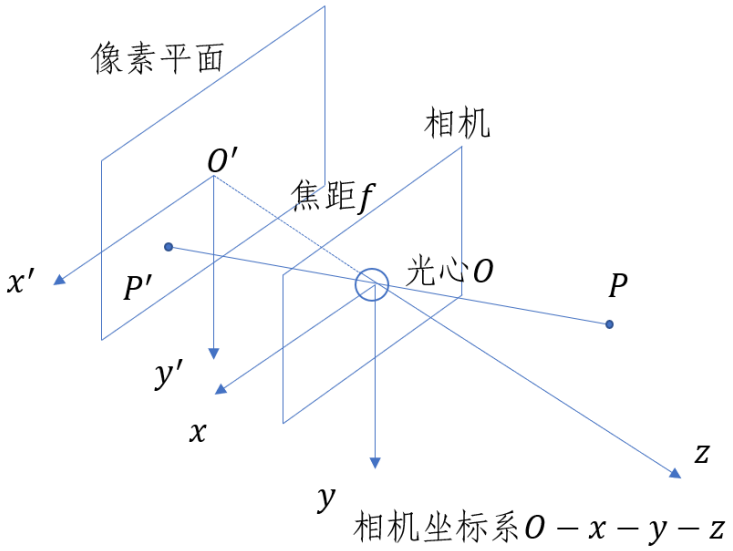


图 4-1

其中负号表示成的像是倒立的。为了简化模型，我们可以把成像平面对称到相机前方，和三维空间点一起放在摄像机坐标系的同一侧，如图所示。这样做可以把公式中的负号去掉，使式子更加简洁：

$$\frac{Z}{f}=\frac{X}{X'}=\frac{Y}{Y'}$$

整理得：

$$X'=f\frac{X}{Z}$$

$$Y' = f\frac{Y}{Z}$$

上述公式描述了点P和它的像之间的空间关系。不过，在相机中我们最终得到的是像素点，这需要都在成像平面上对像进行采样和量化。为了描述传感器将感受到的光线转换成图像像素的过程。我们设在物理成像平面上固定这一个像素平面$o-u-v$。我们在像素平面得到了P'的像素坐标：$[u,v]^T$。

像素坐标系的定义方式为：原点o'位于图像的左上角，u轴向右与x轴平行，v轴向下与y轴平行。像素坐标系与成像平面之间，相差了一个缩放和一个原点的平移。我们设像素坐标在u轴上缩放了α倍，在v上缩放了β倍。同时，原点平移了$[c_x,c_y]^T$。P'的坐标与像素坐标$[u,v]^T$的关系为

$$\begin{cases} u = \alpha X' + c_x \\ v = \beta Y' + c_y \end{cases}$$

将αf合并成f_x，βf合并成f_y，得：

$$\begin{cases} u = f_x\frac{X}{Z} + c_x \\ v = f_y\frac{Y}{Z} + c_y \end{cases}$$

其中f的单位为米，α，β的单位为像素/米，所以f_x，f_y的单位为像素。把该式写成矩阵形式会更加简洁，不过左侧需要用到齐次坐标：

$$\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \frac{1}{Z}\begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} \triangleq \frac{1}{Z}\boldsymbol{KP}$$

将Z挪到左侧：

$$Z\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} \triangleq \boldsymbol{KP}$$

在上式中将中间的量所组成的矩阵称为相机的内参数矩阵$\boldsymbol{K}$。通常认为，相机的内参在出厂之后是固定的，不会再使用过程中发生变化。由于相机在运动，所以相机P的相机坐标应该是世界坐标（记为P_w），根据相机的当前位姿变换到相机坐标系下的结果。相机的位姿有它的旋转矩阵$\boldsymbol{R}$和平移向量$\boldsymbol{t}$来描述。则：

$$\boldsymbol{ZP_{uv}} = Z\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \boldsymbol{K(RP_w + t) = KTP_w}$$

相机的位姿$\boldsymbol{R}$，$\boldsymbol{t}$又称为相机的外参数。相比较不变的内参，外参会随着相机运动发生改变。因为其次坐标乘上非零常数后表达同样的含义，所以可以简单地把Z去掉：

$$\boldsymbol{P_{uv} = KTP_w}$$

利用相机的成像原理，在本题中进行两次空间变化。第一次根据车侧面的坐标值构建空间关系，求解相机内参$\boldsymbol{K}$。第二次根据相机内参求解地面的平面方程，带入已知数值求解该问题。

根据平面成像原理以及图像中所显示出的车内人与车辆的相对位置关系，我们可以合理预估车侧面的车把手和车门相对于相机的位置，即我们可以得到较为准确的坐标$[X,Y,Z]^T$。根据小孔成像原理，如图 4-2 所示，我们可得相机内参。

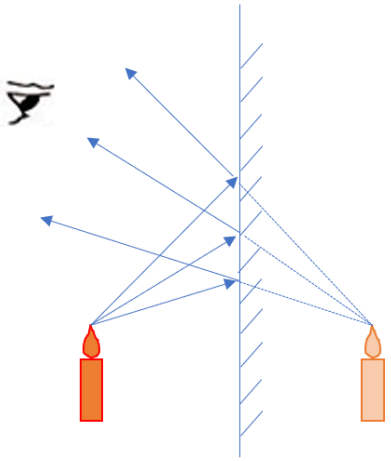


图 4-2 小孔成像原理

第二步我们假设地面的坐标为$AX+BY+CZ+1=0$，根据：

$$X=\frac{u_x}{f_x}Z$$

$$Y=\frac{u_y}{f_y}Z$$

我们可得到新的公式：

$$A\frac{u_x}{f_x}Z+B\frac{u_y}{f_y}Z+CZ+1=0$$

因此：

$$Z=-\frac{1}{A\frac{u_x}{f_x}+B\frac{u_y}{f_y}+C}$$

已知平面任意两点$[X,Y,Z]^T$及$[X',Y',Z']^T$，可得到两点间的距离D为：

$$(X-X')^2+(Y-Y')^2+(Z-Z')^2=D^2$$

用带Z的公式替代X可得：

$$(\frac{u_x}{f_x}Z-\frac{u_x{}'}{f_x}Z')^2+(\frac{u_y}{f_y}Z-\frac{u_y{}'}{f_y}Z')^2+(Z-Z')^2=D^2$$

$$(-\frac{u_x}{Au_x+B\frac{u_yf_x}{f_y}+Cf_x}+\frac{u_x{}'}{Au_x'+B\frac{u_y'f_x}{f_y}+Cf_x})^2$$

$$+(-\frac{u_y}{A\frac{u_xf_y}{f_x}+Bu_y+Cf_y}+\frac{u_y{}'}{A\frac{u_y'f_y}{f_x}+Bu_y'+Cf_y})^2$$

$$+(-\frac{1}{A\frac{u_x}{f_x}+B\frac{u_y}{f_y}+C}+\frac{1}{A\frac{u_x{}'}{f_x}+B\frac{u_y{}'}{f_y}+C})^2=D^2$$

在已知D的基础上我们可得到确切的平面方程，而在此基础上我们可以求出平面上确切点的深度距离。由于是动态视频，我们可以得到某段距离间的时间信息，根据公式：

$$\Delta v=\frac{\Delta D}{t}$$

可以解答题目中相对速度的问题。

4.3 模型计算

本题指出拍摄者所乘车辆为别克英朗 2016 款汽车，所以可以在网上搜索到别克英朗的具体尺寸。可以使用的尺寸如图 4-3 所示：

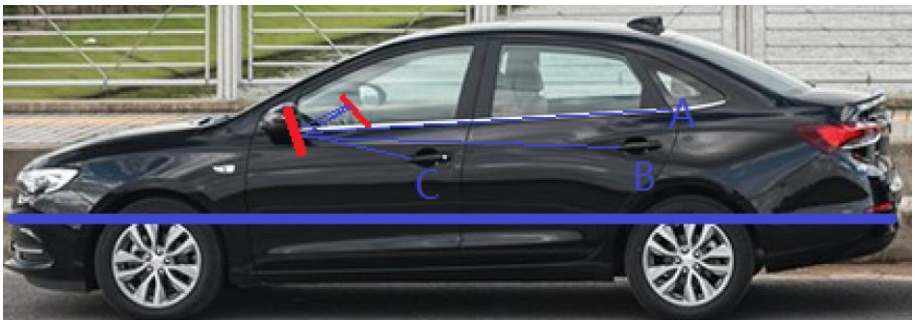


图 4-3

简化车后视镜为平面镜，可以查询到到前车门把手、后车门把手和后窗角度对于车后视镜的尺寸；同时地面上的白虚线通过查询数据，可以知道前后之间的相对差值为 2 米。同时这里假设相机的外姿态是

固定不变的。对于以上信息，关于相机内外参的方程列出如下：

$$Z\begin{pmatrix}u_a\\v_a\\1\end{pmatrix}=\begin{pmatrix}f_x&0&c_x\\0&f_y&c_y\\0&0&1\end{pmatrix}\begin{pmatrix}X_a\\Y_a\\Z_a\end{pmatrix}\ Z\begin{pmatrix}u_b\\v_b\\1\end{pmatrix}=\begin{pmatrix}f_x&0&c_x\\0&f_y&c_y\\0&0&1\end{pmatrix}\begin{pmatrix}X_b\\Y_b\\Z_b\end{pmatrix}$$

$$Z\begin{pmatrix}u_c\\v_c\\1\end{pmatrix}=\begin{pmatrix}f_x&0&c_x\\0&f_y&c_y\\0&0&1\end{pmatrix}\begin{pmatrix}X_c\\Y_c\\Z_c\end{pmatrix}\ Z\begin{pmatrix}\Delta u\\\Delta v\\1\end{pmatrix}=\begin{pmatrix}f_x&0&c_x\\0&f_y&c_y\\0&0&1\end{pmatrix}\begin{pmatrix}\Delta X\\\Delta Y\\\Delta Z\end{pmatrix}$$

其中ΔX，$\Delta Y=0,\ \Delta Z=2$。通过四组数据，解方程可以求得相机的内外参数$\boldsymbol{K}$。根据 4.2 小节方程最终带入可以求得，$D=49.52$米。

然后通过对原视频的查看，估计从拍摄者搭乘车辆超越第一辆白车到白车与后面红车相平行，总共过了$\mathrm{t}=7.33$秒。在这一段超车过程中，假设三辆车做的匀速运动。所以测得最终$\Delta v=\frac{\Delta D}{t}=6.76$米/秒。

综上所述，本题的结果如下表 4-1 所示：

表 4-1 任务二结果

项目	与红车距离	超车时速度差
数值	49.52 米	6.76 米/秒

4.4 模型评价

首先本题中，我们通过已知的尺寸和平面成像原理求解出了相机的内参，再通过内参可以算出图片中点在原三维世界下的位置，从而可以计算得出红车与本车之间的距离，计算方法较为可靠。其次，计算相对速度时，我们通过观察，发现红车和本车之间距离未变，从而假设本车和白车之间的相对速度恒定，通过一段时间内的路程差来求出速度差，具有一定的可行性。

五、任务三

5.1 任务分析

任务 4：附件“无人机拍庄园.mp4”记录了某老宅的全景。(1) 估算其中环绕老宅道路的长度、宽度、各建筑物的高度、后花园中树木的最大高度；(2) 估算该老宅的占地面积；(3) 测算无人机的飞行高度和速度。

本任务可以认为是任务二的进阶版本，任务二可以从固定深度的物体中估计出相机的内参，而本任务所提供的视频信息中，视野中并没有可以用来估计的物体，所以不能照搬任务二的方法。但是本任务的拍摄角度是固定角度，其轨迹运动是线性运动，所以可以使用相邻的图像帧来模拟双目信息，以双目的方式估计相机的内参。估计到相机内参后，可以使用如任务二一致的方式估计出地平面的平面方程，进而求解图像中感兴趣的观测值。

5.2 模型建立

在任务三中我们可以借鉴任务二中的算法，但与其不同的是该场景无法根据所拍摄图像准确地估计相机内部参数，缺少先验的约束条件，因此我们利用所拍摄图像的比例关系合理地假设所使用的摄像机的广角参数为θ，可以通过θ求得相机的焦距f。

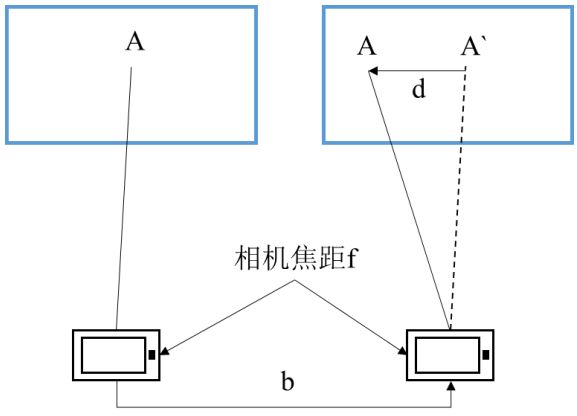


图 5-1

在这里我们可以将一段位移前后的两个视角近似的看做双目摄像机，这两个相机参数一样。在已知图像边界点的坐标为(u,v)的基础上，我们可得近似的双目摄像机示意图 5-1 为：

在双目摄像头中，通过相似三角形代换可以得出目标 A 距摄像头的距离：

$$\mathrm{Z} = \mathrm{b} \times f \div d$$

这样通过近似的双目摄像机就可以对远处的物体进行测距。

在视频中我们可以发现，铁路接触网按照一定频率出现。经过资料查询，我们得知其之间的距离$\Delta d = 47$米，而在视频中两个相邻铁路接触网出现的时间间隔Δt秒。根据速度计算公式我们可得，所乘高铁的速度为：

$$v = \frac{\Delta d}{\Delta t}$$

我们选取两帧图像，第一帧为水面左端在照片中间时的图像，第二帧为水面右端在照片中间时的图像，两帧图像之间的时间间隔记为t_1。因为我们可以得到水面宽度l：

$$l = vt_1$$

对于第一座桥桥面距高铁轨道的距离l'，我们选取桥梁上任意一个固定的点，记录在任意一段时间t_2中，像素点在图像中的横向移动为$\Delta \mathrm{u}$，根据距离与速度的关系我们可得到现实生活中该点的移动距离l_2：

$$l_2 = vt_2$$

根据上一章中的小孔成像原理，我们可以得到第一座桥桥面距高

铁轨道的距离l'：

$$l' = l_2 \times f \div \Delta \mathrm{u}$$

第一座桥桥面距水面的高度h可根据像素之间的关系转换求得。在任意一帧图中，桥梁左右的像素点坐标差为$\Delta u'$，桥梁面距离水面的像素点坐标差为$\Delta v'$。根据像素点与实际距离之间的关系我们可得：

$$\frac{\Delta u'}{\Delta v'} = \frac{l}{\mathrm{h}}$$

因此：

$$\mathrm{h} = \frac{l \Delta v'}{\Delta u'}$$

通过广角相机模型的构建我们可以通过连续视频的图像关系得到问题三的答案。

5.3 模型计算

首先根据铁路接触网的出现时间戳和出现的次序计算出高铁运行的速度，时间戳与铁路接触网出现次序如下表 5-1 所示：

表 5-1 时间戳与铁路接触网出现次序

1	2	3	4	5	6
3.832	4.400	4.998	5.600	6.199	6.799

在这一段时间中，总共经过了 5 次铁路接触网，铁路接触网的长度约为 47 米，所以$\Delta \mathrm{d} = 5 \times 47 = 235$米，而$\Delta t = \mathrm{t}_6 - \mathrm{t}_1 = 2.967$秒，所以解得速度$\mathrm{v} = 79.20$米/秒。

在视频中，我们从画面的左边界刚好与河左岸重合到画面左边界刚好与河右岸重合的时间差值$t_1 = 8.033$ 秒 $- 5.198$ 秒 $= 2.835$秒，所以估计河的宽度$l = \mathrm{v} \times t_1 = 224.53$米。

对于测算第一座桥与高铁桥的距离，首先抽取视频中的两帧，计算出Δu =232 像素，这两帧之间的时间$t_2 = 1.064$秒，所以运动的距离$l_2 = v \times t_2 = 84.27$米。通过查询市面上手机摄像头的参数，可知现有旗舰手机大部分是 117° 广角镜头，焦距与视角之间有经验公式$f = 21.7/\tan\frac{\theta}{2}$，所以可以估计拍摄手机的焦距$f$ =13.30 毫米。此外可以查找现有手机单位像素占物理尺寸大概是$n_0 = 0.0099$米。根据上述条件可以得出$l' = l_2 \times f \div (\Delta u \times n_0) = 487.98$米。

对于测算对面桥距离河的高度，因为桥墩和桥梁位于同一平面且距离相近，所以直接使用像素的比值来进行桥梁高度的测算。

这里假设河在这一段处处相等，那么这里的河长为$l = 224.53$米。那么位于河岸两边的桥墩中还有 10 个桥墩，共 11 段桥。假设这 11 段桥等长，可以得出一段桥的长度$n_0 = l/11 = 20.41$米，对应的像素长度为 85 个像素。桥墩高度的像素长度对应为 38 个像素，所以求得桥距离河的高度为$h = 38 \div 85 \times n_0 = 9.12$米。

综上所述，本题的结果如下表 5-2 所示：

表 5-2 任务三结果

项目	河的宽度	桥离高铁的距离	桥面距河的高度
长度(米)	224.53	487.98	9.12

5.4 模型评价

首先本题通过使用高速铁路上的固有铁路接触网的长度与高铁的通过时间来求解出高铁的速度，进而求解出河面的宽度，是一种较为准确的先验尺寸获得方法。除此之外，可以观察到拍摄过程中相机的姿态几乎没有改变，并且相机参数不变，符合双目估计的条件，并

且可以估计相机的焦距等参数，能够较为准确估计出双目的距离测量。此外，由于桥梁过远，且几乎没有参照物可以参照桥距离河的高度，考虑到桥墩和桥梁位于同一平面，且距离相近，在照片未失真情况下通过像素比对进行桥离水面距离的判断也是有可行性的，总体来说本题的模型和算法不失为一种较好的进行距离估计的算法。

六、任务四

6.1 任务分析

附件“无人机拍庄园.mp4”记录了某老宅的全景。(1）估算其中环绕老宅道路的长度、宽度、各建筑物的高度、后花园中树木的最大高度；(2）估算该老宅的占地面积；(3）测算无人机的飞行高度和速度。

该任务使用无人机作为数据采集设备，围绕房屋采集的多视几何数据，需要在没有其他先验信息的情况下预估相机内外参数以及目标的空间和几何关系。该任务所使用的相机为单目相机，所以任务目标为基于单目 RGB 的相机运动轨迹估计、空间建模及测距。通用的方法有视觉 SLAM 以及三维重建中的运动恢复结构(SFM)，二者都可以根据相机移动所获取的视觉信息(一般是图像或视频)，来估计得到视觉信息中物体的三维点云图。

任务可以具体的划分为两个部分：根据视觉信息来获取其中老宅的三维重建模型；其中无人机的飞行轨迹及姿态。其中无人机在空间中飞行的姿态轨迹与三维重建的模型是强相关的，二者处于同一个世界坐标系下。同时，我们所得到的模型信息，需要预先标定一个尺度作为参考依据，然后依照尺度测量出所需求得的结果。在任务四中，我们可以选取围墙的真实值作为尺度，估计其在建模后的点云距离，然后反推出其他结果。

基于单目 RGB 视觉 SLAM 方式估计得到的点云图，由于缺少先验的深度信息，无法得到可视化程度很高的稠密点云图，具有较多噪点信息。然而，由于其 SLAM 技术本身同时定位与地图构建的特性，可以在客观条件较差，如视频清晰度不够高的情况下依旧获得很好的相

机姿态估计和运动轨迹估计。而 SFM 的方式是三维重建方式中的一种，其类似于单目视觉 SLAM 之于视觉 SLAM，是一种在普通单目摄像头情况下进行三维重建的技术。SFM 能够使用多帧图片获得图片中物体的三维点云信息，且该点云信息是结构化的稠密点云，能够具有较好的可视性，但是该技术对于相机姿态和轨迹的估计不如 SLAM 方法。根据本任务的目标，将视频帧抽取成图片，然后将多帧图片使用 SFM 算法获得稠密且结构化的点云信息，然后结合单目视觉 SLAM，从原视频中获得较为稀疏的点云图与相机轨迹。将 SFM 所获得的稠密点云图与 SLAM 所获得的稀疏点云图做相关性计算，滤除相关性较差的噪点信息，然后再经过泊松分布采样并重建表面，得到可视性较好的重建图后，再进行结果估计。

6.2 模型建立

本任务所使用的单目 RGB 的视觉 SLAM 算法，由 3+1 个平行线程组成，包括 ORB 特征提取、跟踪、局部建图和全局 BA 优化。

ORB 特征提取包括特征点提取和特征点描述。特征点提取需要先经过粗提取，然后使用机器学习的决策树算法进行初筛选，再使用非极大值抑制算法去除临近位置多个特征点的问题。为每一个特征点计算出其响应大小。计算方式是特征点 P 和其周围 16 个特征点偏差的绝对值之和。在比较临近的特征点中，保留响应值较大的特征点，删除其余的特征点。再建立金字塔实现特征点的多尺度不变性，使用矩法来确定 ORB 特征点的方向，其中矩的定义如下：

$$m_{pq} = \sum_{x,y \in r} x^p y^q I(x,y)$$

其中， $I(x,y)$ 为图像灰度表达式。

通过寻找对局部地图特征进行匹配，利用纯运动 BA 最小化重投影误差进行定位每帧图片的相机;并通过执行局部 BA 管理局部地图并优化；在位姿图优化之后再计算整个系统的最优结构和运动结果。

单目 RBG 的 SLAM 系统架构如下图 6-1:

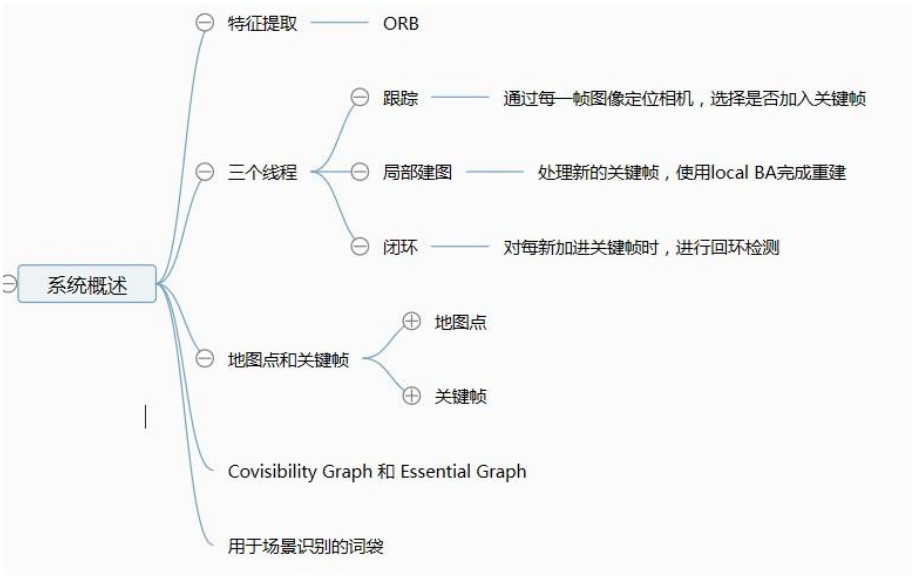


图 6-1

而本任务三维重建所使用的 SFM 算法,其算法流程如下:

（1）特征点的提取与匹配,匹配结果往往有很多误匹配，为了排除这些错误，使用 KNN 算法寻找与该特征最匹配的 2 个特征，若第一个特征的匹配距离与第二个特征的匹配距离之比小于某一阈值，就接受该匹配，否则视为误匹配。

（2）采用 RANSAC 的方法进行对基础矩阵 F 进行估计，每一步迭代的过程中，利用 8 点法进行求解 E 中的九个元素。

（3）估计本质矩阵 E 并使用 SVD 分解为 R 和 T,估计出本质矩阵的目的是为了对之前求得的匹配进行约束，得到的匹配成为几何一致

匹配，不同图像上的几何一致匹配形成了一个 TRACK。

（4）使用三角形法计算三维点云。已经知道了两个相机之间的变换矩阵(R 和 T)，还有每一对匹配点的坐标，通过这些已知信息还原匹配点在空间当中的坐标，根据公式

$$x_2 s_2 = K(R_2 X + T_2)$$

求得 X，其几何意义相当于分别从两个相机的光心作过 x1 和 x2 的延长线，延长线的焦点即为方程的解。

（5）重投影。将三维点三角化并重映射到摄像机得到二维点，计算与最初二维点之间的距离，说明三角化误差。

（6）计算第三个摄像机到到世界坐标系的变换矩阵(R 和 T)。摄像机标定或摄像机之态估计，对于输入的第三幅图片，计算第三幅图片与第二幅图片的匹配点，这些匹配点中，肯定有一部分也是图像二与图像一之间的匹配点，也就是说，这些匹配点中有一部分的空间坐标是已知的，同时又知道这些点在第三幅图像中的像素坐标，即可计算变换矩阵。

（7）得到第三个摄像机的变换矩阵后，就可以计算匹配点的在空间中的坐标，得到三维点云，将新得到的三维点云与之前计算得三维点云进行融合，然后循环迭代。

（8)进行重构的细化与优化，优化图像累加所带来的积累误差。

6.3 模型计算

针对任务四，首先使用 ffmpeg 工具将原视频按一秒三十帧抽取成图片，并将该图片集进行 SFM 算法建模。其初步的建模效果如下图

6-2 所示。

图 6-2

可以大致的观察出房屋及花园的轮廓，但是从三维点云图中可以观察到诸多明显不符合实际的三维点以及偏离测量目标太远的测量点。使用 SLAM 的方法可以得到稀疏的点云图以及相机的轨迹图，将 SLAM 稀疏点云图与 SFM 的初步建模结果计算互相关性，然后将相关性过低的点滤除。其中 SLAM 方法得到的点云图及其相机位姿图 6-3 如下。

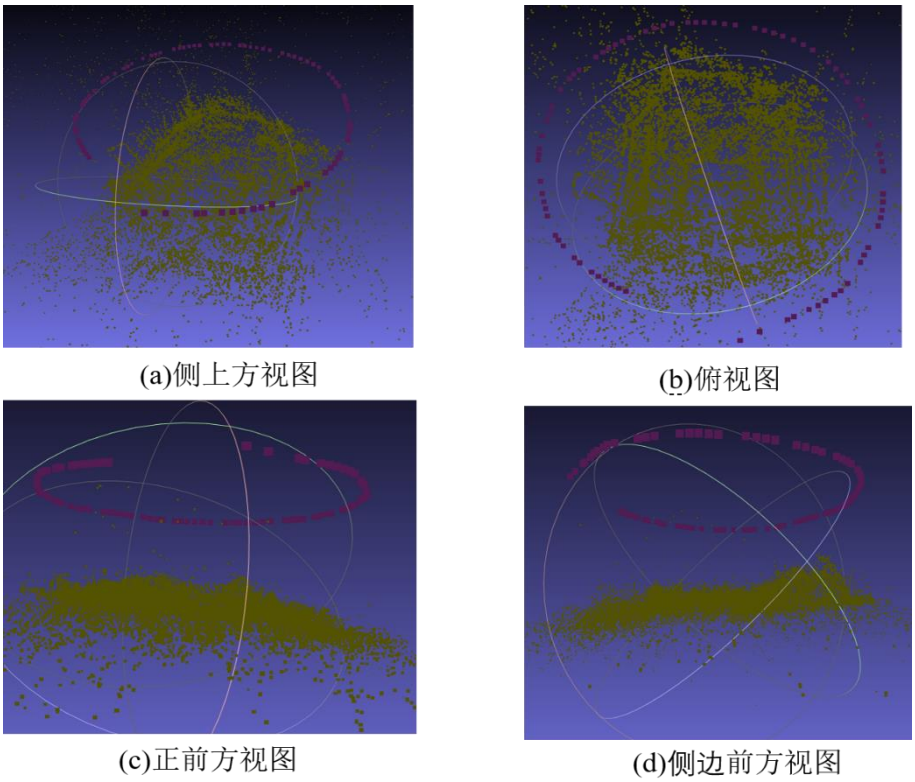
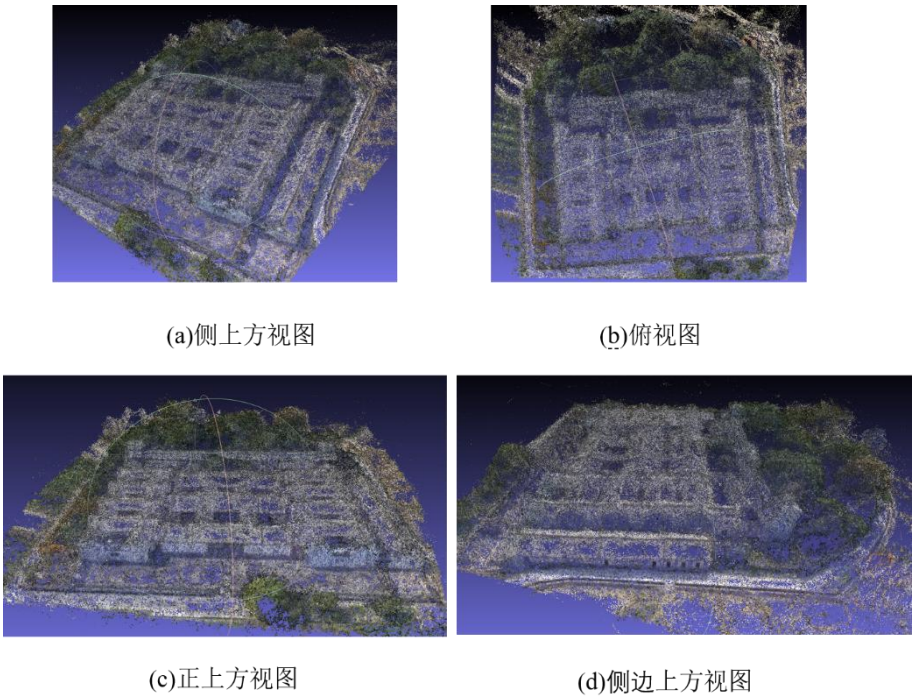


图 6-3

经过滤波并利用泊松分布降采样后的三维重建点云图如下图 6-4 所示。

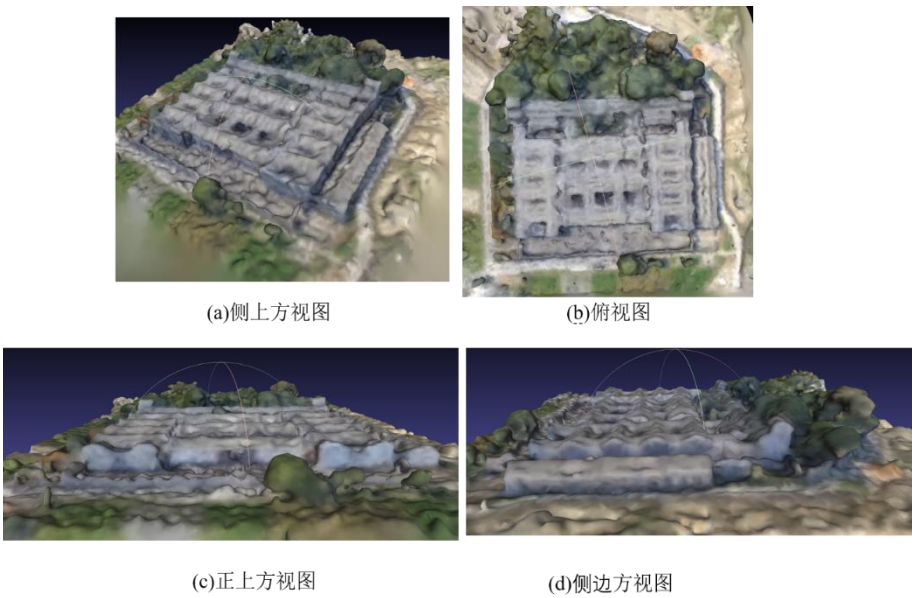


(a)侧上方视图　　(b)俯视图

(c)正上方视图　　(d)侧边上方视图

图 6-4

该点云图已经具有良好的可视性和结构特性了，为了方便测量其中的数据，在点云之间重建表面，得到如下图 6-5 所示。



(a)侧上方视图　　(b)俯视图

(c)正上方视图　　(d)侧边方视图

图 6-5

该三维图即为老宅最终建模出来的三维重建图，可在其中估计我

们的待测目标。首先需要标定一个先验尺度，本文根据视频中最初几帧中，出入老宅大门的人与墙面的高度差，假定老宅外墙的高度约为1.8 米，如图 6-6(a)：

(a)老宅墙高尺度　　(b)三次取值求平均

图 6-6

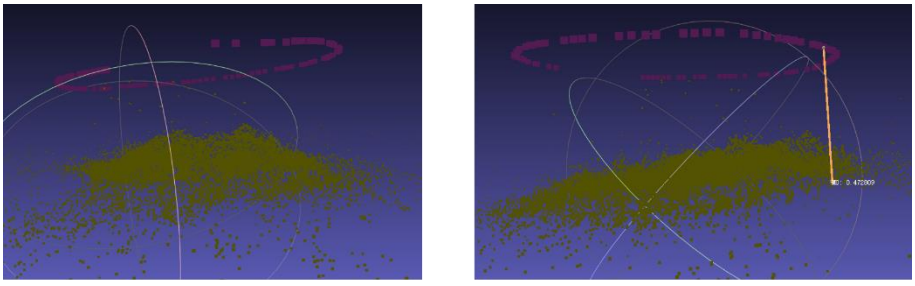
从建模图中，可以获取到建模图中的单位约为 0.022 单位/米。针对每个测量目标，我们从建模图中选取三对值取平均，如图 6-6(b)所示。最终求得环绕老宅的道路宽度约为(0.075+0.077+0.072)/(0.022*3) = 3.39 米。如图 6-7 所示。

图 6-7

可以计算得到 AB 的长度约为 90.12 米，以 B 点往外延伸的非直线部分不考虑在内为前提，BC 的长度等于 AD 的长度约等于 66.93 米，而 CD 弧线部分，使用五段线段近似其真实长度，五段线段的总长约

为 122.57 米，所以围绕老宅的道路长度约为 346.55 米。而老宅中，横宅的高度约为 6.32 米，纵宅最顶高度约为 6.81 米，望楼最顶的高度约为 9.63 米，后花园众最高的树木高度约为 13.3 米。以老宅的四方外墙为边界，不考虑道路在内，根据老宅道路的长度，可以估算出老宅的占地面积约为 8312 平方米。

对于无人机的高度，使用 SLAM 的稀疏点云图来计算无人机与地面点云的坐标差，以纯横向视角观察可以观察到无人机并非在等高的距离飞行，如下图 6-8(a)所示，所以可以根据图 6-8(b)的方式估计出无人机飞行的最高值与最低值，其最高值约为 56.44 米，其最低值约为 43.18 米。



(a)无人机飞行高度差异　　(b)无人机高度求解

图 6-8

根据无人机的飞行轨迹，可以估计出其飞行直径约为 129.14 米，在整个视频中其飞行角度约为 45 度，所以可以求得其飞行距离约为 354.99 米，其整体飞行时长为 64 秒，所以其水平方向飞行速度约为 5.54 米/秒。

6.4 模型评价

利用 6.2 小节所提出的算法对原视频进行建模并测量待测值后，任务四的结果如表 6-1 所示。单纯使用三维重建算法或者单目视觉

SLAM 的方法得到的结果都不够理想，而使用二者相结合的方法对点云进行降噪后，其可视化和结构化的特性就更加明显。

表 6-1 任务四结果

无人机最高高度		56.44 米		无人机最低高度		43.18 米	
横宅高	6.32 米	纵宅高	6.81 米	望楼高	9.63 米	树高	13.3 米
面积	8312 米^2	路宽	3.39 米	路长	346.55 米	速度	5.54 米/秒

七、总结

本文使用消隐点测距法进行单张图片距离的建模估计，该法的优点是通过消隐点与三维空间中的点通过代数代换获得平行线之间比例，从而以先验物品长度来计算出待测物品的长度，具有计算简单、计算准确的优点。任务一中的图像具有角度比较大，待测物体离镜头较远，容易通过先验知识估计物体尺寸的特点，角度比较大使得原图中平行线间角度较大，容易找到消隐点；并且待测物体离镜头较远，使用其他的代数代换方法会造成很大误差，所以消隐点测距法是一种特比适合本文任务的算法。而对于无人机测距任务，由于无人机没有完整的飞完一圈，所以没法使用 SLAM 技术中的回环检测，使得后端优化的效力大打折扣。加之视频清晰度较低，其他的先验知识也比较匮乏，所以无法单纯的使用 SLAM 的方式建模。但是 SLAM 的方式所估计出来的稀疏点云噪点偏移较少，几乎都是待重建结构中的关键点。而利用三维重建的方式，采用多帧图片估计出大量稠密的点云，这些点云可以很好的反映出重建物体的结构信息，却也带来了大量的噪点信息，所以使用 SLAM 的关键点与三维重建的稠密点进行互相关计算滤除大量相关性低的噪点。该模型能够使得本文建模的效果更加突出，方便结果的测量。

八、参考文献

[1] 刘学军，王美珍，甄艳等：单幅图像几何量测研究进展[J]，《武汉大学学报》（信息科学版）,36(8) : pp941 - 947.

[2] 刘军，后士浩，张凯,晏晓娟: 基于单目视觉车辆姿态角估计和逆透视变换的车距测量[J]，《农业工程学报》，Jul. 2018(pp70-76)

[3] 郭磊，徐友春，李克强，et al. 基于单目视觉的实时测距方法研究[J]. 中国图象图形学报，2018，11(1):74-81.

[4] 汪亚兵，冯肖维，肖健梅，et al. 基于单目视觉的实时目标距离测量[J]. 工业控制计算机，2018，31(1): 110-112.

[5] Mur-Artal, Raul and Juan D. Tardós. “ORB-SLAM2: An Open-Source SLAM System for Monocular, Stereo, and RGB-D Cameras.” IEEE Transactions on Robotics 33 (2017): 1255-1262.

[6] Strasdat, H., Davison, A.J., Montiel, J.M., & Konolige, K. (2011). Double window optimisation for constant time visual SLAM. 2011 International Conference on Computer Vision, 2352-2359.

[7] Huang, L., Pan, S., Wang, S., Zeng, P., & Ye, F. (2018). A fast initialization method of Visual-Inertial Odometry based on monocular camera. 2018 Ubiquitous Positioning, Indoor Navigation and Location-Based Services (UPINLBS), 1-5.

[8] Ullman S. and Brenner Sydney The interpretation of structure from motion203Proc. R. Soc. Lond. B

[9] Strasdat, H., Davison, A.J., Montiel, J.M., & Konolige, K. (2011). Double window optimisation for constant time visual SLAM. 2011 International Conference on Computer Vision, 2352-2359.

代码附录

透视变换法代码

```
import cv2
import numpy as np
import matplotlib.pylab    as plt
img = cv2.imread('e://aaa.png')
rows,cols,ch = img.shape
plt.imshow(img)
pos = plt.ginput(4)
pts1 =
np.float32([[pos[0][0],pos[0][1]],[pos[1][0],pos[1][1]],[pos[2][0],pos[2][1]],[pos[3][0],pos[3][1]]]
)
pts2 = np.float32([[0,0],[1200,0],[0,900],[1200,900]])
M = cv2.getPerspectiveTransform(pts1,pts2)
dst = cv2.warpPerspective(img,M,(1200,900))
print(1111)
plt.subplot(121),plt.imshow(img),plt.title('Input')
plt.subplot(122),plt.imshow(dst),plt.title('Output')
#plt.show()
cv2.imwrite("e://test.jpg", dst)
```

基于单目的视觉 SLAM 算法系统架构代码

```
#include
"System.h"
                #include "Converter.h"
                #include <thread>
                #include <pangolin/pangolin.h>
                #include <iomanip>

                namespace ORB_SLAM2
                {

                System::System(const string &strVocFile, const string &strSettingsFile, const
                eSensor sensor,
                                const bool bUseViewer):mSensor(sensor),
                mpViewer(static_cast<Viewer*>(NULL)),
                mbReset(false),mbActivateLocalizationMode(false),
                        mbDeactivateLocalizationMode(false)
                {
                    // Output welcome message
                    cout << endl <<
                    "ORB-SLAM2 Copyright (C) 2014-2016 Raul Mur-Artal, University of
                Zaragoza." << endl <<
                    "This program comes with ABSOLUTELY NO WARRANTY;" << endl
                <<
                    "This is free software, and you are welcome to redistribute it" << endl <<
                    "under certain conditions. See LICENSE.txt." << endl << endl;

                    cout << "Input sensor was set to: ";

                    if(mSensor==MONOCULAR)
                        cout << "Monocular" << endl;
                    else if(mSensor==STEREO)
                        cout << "Stereo" << endl;
                    else if(mSensor==RGBD)
                        cout << "RGB-D" << endl;

                    //Check settings file
                    cv::FileStorage fsSettings(strSettingsFile.c_str(), cv::FileStorage::READ);
                    if(!fsSettings.isOpened())
                    {
                       cerr << "Failed to open settings file at: " << strSettingsFile << endl;
                       exit(-1);
                    }
```

```
    //Load ORB Vocabulary
    cout << endl << "Loading ORB Vocabulary. This could take a while..." <<
endl;

    mpVocabulary = new ORBVocabulary();
    bool bVocLoad = mpVocabulary->loadFromTextFile(strVocFile);
    if(!bVocLoad)
    {
        cerr << "Wrong path to vocabulary. " << endl;
        cerr << "Falied to open at: " << strVocFile << endl;
        exit(-1);
    }
    cout << "Vocabulary loaded!" << endl << endl;

    //Create KeyFrame Database
    mpKeyFrameDatabase = new KeyFrameDatabase(*mpVocabulary);

    //Create the Map
    mpMap = new Map();

    //Create Drawers. These are used by the Viewer
    mpFrameDrawer = new FrameDrawer(mpMap);
    mpMapDrawer = new MapDrawer(mpMap, strSettingsFile);

    //Initialize the Tracking thread
    //(it will live in the main thread of execution, the one that called this
constructor)
    mpTracker = new Tracking(this, mpVocabulary, mpFrameDrawer,
mpMapDrawer,
                             mpMap, mpKeyFrameDatabase,
strSettingsFile, mSensor);

    //Initialize the Local Mapping thread and launch
    mpLocalMapper = new LocalMapping(mpMap,
mSensor==MONOCULAR);
    mptLocalMapping = new
thread(&ORB_SLAM2::LocalMapping::Run,mpLocalMapper);

    //Initialize the Loop Closing thread and launch
    mpLoopCloser = new LoopClosing(mpMap, mpKeyFrameDatabase,
mpVocabulary, mSensor!=MONOCULAR);
    mptLoopClosing = new thread(&ORB_SLAM2::LoopClosing::Run,
mpLoopCloser);
```

```
    //Initialize the Viewer thread and launch
    if(bUseViewer)
    {
        mpViewer = new Viewer(this,
mpFrameDrawer,mpMapDrawer,mpTracker,strSettingsFile);
        mptViewer = new thread(&Viewer::Run, mpViewer);
        mpTracker->SetViewer(mpViewer);
    }

    //Set pointers between threads
    mpTracker->SetLocalMapper(mpLocalMapper);
    mpTracker->SetLoopClosing(mpLoopCloser);

    mpLocalMapper->SetTracker(mpTracker);
    mpLocalMapper->SetLoopCloser(mpLoopCloser);

    mpLoopCloser->SetTracker(mpTracker);
    mpLoopCloser->SetLocalMapper(mpLocalMapper);
}

cv::Mat System::TrackStereo(const cv::Mat &imLeft, const cv::Mat &imRight,
const double &timestamp)
{
    if(mSensor!=STEREO)
    {
        cerr << "ERROR: you called TrackStereo but input sensor was not set
to STEREO." << endl;
        exit(-1);
    }

    // Check mode change
    {
        unique_lock<mutex> lock(mMutexMode);
        if(mbActivateLocalizationMode)
        {
            mpLocalMapper->RequestStop();

            // Wait until Local Mapping has effectively stopped
            while(!mpLocalMapper->isStopped())
            {
                usleep(1000);
            }
```

```
            mpTracker->InformOnlyTracking(true);
            mbActivateLocalizationMode = false;
        }
        if(mbDeactivateLocalizationMode)
        {
            mpTracker->InformOnlyTracking(false);
            mpLocalMapper->Release();
            mbDeactivateLocalizationMode = false;
        }
    }

    // Check reset
    {
    unique_lock<mutex> lock(mMutexReset);
    if(mbReset)
    {
        mpTracker->Reset();
        mbReset = false;
    }
    }

    cv::Mat Tcw = mpTracker->GrabImageStereo(imLeft,imRight,timestamp);

    unique_lock<mutex> lock2(mMutexState);
    mTrackingState = mpTracker->mState;
    mTrackedMapPoints = mpTracker->mCurrentFrame.mvpMapPoints;
    mTrackedKeyPointsUn = mpTracker->mCurrentFrame.mvKeysUn;
    return Tcw;
}

cv::Mat System::TrackRGBD(const cv::Mat &im, const cv::Mat &depthmap,
const double &timestamp)
{
    if(mSensor!=RGBD)
    {
        cerr << "ERROR: you called TrackRGBD but input sensor was not
set to RGBD." << endl;
        exit(-1);
    }

    // Check mode change
    {
        unique_lock<mutex> lock(mMutexMode);
        if(mbActivateLocalizationMode)
```

```
            {
                mpLocalMapper->RequestStop();

                // Wait until Local Mapping has effectively stopped
                while(!mpLocalMapper->isStopped())
                {
                    usleep(1000);
                }

                mpTracker->InformOnlyTracking(true);
                mbActivateLocalizationMode = false;
            }
            if(mbDeactivateLocalizationMode)
            {
                mpTracker->InformOnlyTracking(false);
                mpLocalMapper->Release();
                mbDeactivateLocalizationMode = false;
            }
        }

        // Check reset
        {
        unique_lock<mutex> lock(mMutexReset);
        if(mbReset)
        {
            mpTracker->Reset();
            mbReset = false;
        }
        }

        cv::Mat Tcw = mpTracker->GrabImageRGBD(im,depthmap,timestamp);

        unique_lock<mutex> lock2(mMutexState);
        mTrackingState = mpTracker->mState;
        mTrackedMapPoints = mpTracker->mCurrentFrame.mvpMapPoints;
        mTrackedKeyPointsUn = mpTracker->mCurrentFrame.mvKeysUn;
        return Tcw;
}

cv::Mat System::TrackMonocular(const cv::Mat &im, const double
&timestamp)
{
        if(mSensor!=MONOCULAR)
        {
```

```
            cerr << "ERROR: you called TrackMonocular but input sensor was
not set to Monocular." << endl;
            exit(-1);
        }

        // Check mode change
        {
            unique_lock<mutex> lock(mMutexMode);
            if(mbActivateLocalizationMode)
            {
                mpLocalMapper->RequestStop();

                // Wait until Local Mapping has effectively stopped
                while(!mpLocalMapper->isStopped())
                {
                    usleep(1000);
                }

                mpTracker->InformOnlyTracking(true);
                mbActivateLocalizationMode = false;
            }
            if(mbDeactivateLocalizationMode)
            {
                mpTracker->InformOnlyTracking(false);
                mpLocalMapper->Release();
                mbDeactivateLocalizationMode = false;
            }
        }

        // Check reset
        {
        unique_lock<mutex> lock(mMutexReset);
        if(mbReset)
        {
            mpTracker->Reset();
            mbReset = false;
        }
        }

        cv::Mat Tcw = mpTracker->GrabImageMonocular(im,timestamp);

        unique_lock<mutex> lock2(mMutexState);
        mTrackingState = mpTracker->mState;
        mTrackedMapPoints = mpTracker->mCurrentFrame.mvpMapPoints;
```

```
        mTrackedKeyPointsUn = mpTracker->mCurrentFrame.mvKeysUn;

        return Tcw;
}

void System::ActivateLocalizationMode()
{
        unique_lock<mutex> lock(mMutexMode);
        mbActivateLocalizationMode = true;
}

void System::DeactivateLocalizationMode()
{
        unique_lock<mutex> lock(mMutexMode);
        mbDeactivateLocalizationMode = true;
}

bool System::MapChanged()
{
        static int n=0;
        int curn = mpMap->GetLastBigChangeIdx();
        if(n<curn)
        {
                n=curn;
                return true;
        }
        Else
                return false;
}

void System::Reset()
{
        unique_lock<mutex> lock(mMutexReset);
        mbReset = true;
}

void System::Shutdown()
{
        mpLocalMapper->RequestFinish();
        mpLoopCloser->RequestFinish();
        if(mpViewer)
        {
                mpViewer->RequestFinish();
                while(!mpViewer->isFinished())
```

```
            usleep(5000);
        }

        // Wait until all thread have effectively stopped
        while(!mpLocalMapper->isFinished() || !mpLoopCloser->isFinished() ||
mpLoopCloser->isRunningGBA())
        {
            usleep(5000);
        }

        if(mpViewer)
            pangolin::BindToContext("ORB-SLAM2: Map Viewer");
}

void System::SaveTrajectoryTUM(const string &filename)
{
        cout << endl << "Saving camera trajectory to " << filename << " ..." <<
endl;
        if(mSensor==MONOCULAR)
        {
            cerr << "ERROR: SaveTrajectoryTUM cannot be used for
monocular." << endl;
            return;
        }

        vector<KeyFrame*> vpKFs = mpMap->GetAllKeyFrames();
        sort(vpKFs.begin(),vpKFs.end(),KeyFrame::lId);

        // Transform all keyframes so that the first keyframe is at the origin.
        // After a loop closure the first keyframe might not be at the origin.
        cv::Mat Two = vpKFs[0]->GetPoseInverse();

        ofstream f;
        f.open(filename.c_str());
        f << fixed;

        // Frame pose is stored relative to its reference keyframe (which is
optimized by BA and pose graph).
        // We need to get first the keyframe pose and then concatenate the relative
transformation.
        // Frames not localized (tracking failure) are not saved.

        // For each frame we have a reference keyframe (lRit), the timestamp (lT)
and a flag
```

```
        // which is true when tracking failed (lbL).
        list<ORB_SLAM2::KeyFrame*>::iterator lRit =
mpTracker->mlpReferences.begin();
        list<double>::iterator lT = mpTracker->mlFrameTimes.begin();
        list<bool>::iterator lbL = mpTracker->mlbLost.begin();
        for(list<cv::Mat>::iterator lit=mpTracker->mlRelativeFramePoses.begin(),
            lend=mpTracker->mlRelativeFramePoses.end();lit!=lend;lit++,
lRit++, lT++, lbL++)
        {
            if(*lbL)
                continue;

            KeyFrame* pKF = *lRit;

            cv::Mat Trw = cv::Mat::eye(4,4,CV_32F);

            // If the reference keyframe was culled, traverse the spanning tree to
get a suitable keyframe.
            while(pKF->isBad())
            {
                Trw = Trw*pKF->mTcp;
                pKF = pKF->GetParent();
            }

            Trw = Trw*pKF->GetPose()*Two;

            cv::Mat Tcw = (*lit)*Trw;
            cv::Mat Rwc = Tcw.rowRange(0,3).colRange(0,3).t();
            cv::Mat twc = -Rwc*Tcw.rowRange(0,3).col(3);

            vector<float> q = Converter::toQuaternion(Rwc);

            f << setprecision(6) << *lT << " " <<    setprecision(9) <<
twc.at<float>(0) << " " << twc.at<float>(1) << " " << twc.at<float>(2) << " "
<< q[0] << " " << q[1] << " " << q[2] << " " << q[3] << endl;
        }
        f.close();
        cout << endl << "trajectory saved!" << endl;
}


void System::SaveKeyFrameTrajectoryTUM(const string &filename)
{
```

```
    cout << endl << "Saving keyframe trajectory to " << filename << " ..." <<
endl;

    vector<KeyFrame*> vpKFs = mpMap->GetAllKeyFrames();
    sort(vpKFs.begin(),vpKFs.end(),KeyFrame::lId);

    // Transform all keyframes so that the first keyframe is at the origin.
    // After a loop closure the first keyframe might not be at the origin.
    //cv::Mat Two = vpKFs[0]->GetPoseInverse();

    ofstream f;
    f.open(filename.c_str());
    f << fixed;

    for(size_t i=0; i<vpKFs.size(); i++)
    {
        KeyFrame* pKF = vpKFs[i];

       // pKF->SetPose(pKF->GetPose()*Two);

        if(pKF->isBad())
            continue;

        cv::Mat R = pKF->GetRotation().t();
        vector<float> q = Converter::toQuaternion(R);
        cv::Mat t = pKF->GetCameraCenter();
        f << setprecision(6) << pKF->mTimeStamp << setprecision(7) << " "
<< t.at<float>(0) << " " << t.at<float>(1) << " " << t.at<float>(2)
            << " " << q[0] << " " << q[1] << " " << q[2] << " " << q[3] <<
endl;

    }

    f.close();
    cout << endl << "trajectory saved!" << endl;
}

void System::SaveTrajectoryKITTI(const string &filename)
{
    cout << endl << "Saving camera trajectory to " << filename << " ..." <<
endl;
    if(mSensor==MONOCULAR)
    {
```

```
        cerr << "ERROR: SaveTrajectoryKITTI cannot be used for monocular." << endl;
        return;
    }

    vector<KeyFrame*> vpKFs = mpMap->GetAllKeyFrames();
    sort(vpKFs.begin(),vpKFs.end(),KeyFrame::lId);

    // Transform all keyframes so that the first keyframe is at the origin.
    // After a loop closure the first keyframe might not be at the origin.
    cv::Mat Two = vpKFs[0]->GetPoseInverse();

    ofstream f;
    f.open(filename.c_str());
    f << fixed;

    // Frame pose is stored relative to its reference keyframe (which is optimized by BA and pose graph).
    // We need to get first the keyframe pose and then concatenate the relative transformation.
    // Frames not localized (tracking failure) are not saved.

    // For each frame we have a reference keyframe (lRit), the timestamp (lT) and a flag
    // which is true when tracking failed (lbL).
    list<ORB_SLAM2::KeyFrame*>::iterator lRit = mpTracker->mlpReferences.begin();
    list<double>::iterator lT = mpTracker->mlFrameTimes.begin();
    for(list<cv::Mat>::iterator lit=mpTracker->mlRelativeFramePoses.begin(), lend=mpTracker->mlRelativeFramePoses.end();lit!=lend;lit++, lRit++, lT++)
    {
        ORB_SLAM2::KeyFrame* pKF = *lRit;

        cv::Mat Trw = cv::Mat::eye(4,4,CV_32F);

        while(pKF->isBad())
        {
          //    cout << "bad parent" << endl;
            Trw = Trw*pKF->mTcp;
            pKF = pKF->GetParent();
        }

        Trw = Trw*pKF->GetPose()*Two;
```

```
            cv::Mat Tcw = (*lit)*Trw;
            cv::Mat Rwc = Tcw.rowRange(0,3).colRange(0,3).t();
            cv::Mat twc = -Rwc*Tcw.rowRange(0,3).col(3);

            f << setprecision(9) << Rwc.at<float>(0,0) << " " <<
Rwc.at<float>(0,1)    << " " << Rwc.at<float>(0,2) << " "    << twc.at<float>(0)
<< " " <<
                    Rwc.at<float>(1,0) << " " << Rwc.at<float>(1,1)    << " " <<
Rwc.at<float>(1,2) << " "    << twc.at<float>(1) << " " <<
                    Rwc.at<float>(2,0) << " " << Rwc.at<float>(2,1)    << " " <<
Rwc.at<float>(2,2) << " "    << twc.at<float>(2) << endl;
        }
        f.close();
        cout << endl << "trajectory saved!" << endl;
}

int System::GetTrackingState()
{
        unique_lock<mutex> lock(mMutexState);
        return mTrackingState;
}

vector<MapPoint*> System::GetTrackedMapPoints()
{
        unique_lock<mutex> lock(mMutexState);
        return mTrackedMapPoints;
}

vector<cv::KeyPoint> System::GetTrackedKeyPointsUn()
{
        unique_lock<mutex> lock(mMutexState);
        return mTrackedKeyPointsUn;
}
```

Structure from Motion 深度图估计代码

```
#include
"depthmap.h"
                #include <opencv2/opencv.hpp>
                #include <random>

                namespace csfm {

                bool IsInsideImage(const cv::Mat &image, int i, int j) {
                  return i >= 0 && i < image.rows && j >= 0 && j < image.cols;
                }

                template<typename T>
                float LinearInterpolation(const cv::Mat &image, float y, float x) {
                  int ix = int(x);
                  int iy = int(y);
                  if (ix < 0 || ix + 1 >= image.cols || iy < 0 || iy + 1 >= image.rows) {
                    return 0.0f;
                  }
                  float dx = x - ix;
                  float dy = y - iy;
                  float im00 = image.at<T>(iy, ix);
                  float im01 = image.at<T>(iy, ix + 1);
                  float im10 = image.at<T>(iy + 1, ix);
                  float im11 = image.at<T>(iy + 1, ix + 1);
                  float im0 = (1 - dx) * im00 + dx * im01;
                  float im1 = (1 - dx) * im10 + dx * im11;
                  return (1 - dy) * im0 + dy * im1;
                }

                float Variance(float *x, int n) {
                  float sum = 0;
                  for (int i = 0; i < n; ++i) {
                    sum += x[i];
                  }
                  float mean = sum / n;

                  float sum2 = 0;
                  for (int i = 0; i < n; ++i) {
                    sum2 += (x[i] - mean) * (x[i] - mean);
                  }
                  return sum2 / n;
```

```
}

NCCEstimator::NCCEstimator()
      : sumx_(0), sumy_(0), sumxx_(0), sumyy_(0), sumxy_(0), sumw_(0) {}

void NCCEstimator::Push(float x, float y, float w) {
   sumx_ += w * x;
   sumy_ += w * y;
   sumxx_ += w * x * x;
   sumyy_ += w * y * y;
   sumxy_ += w * x * y;
   sumw_ += w;
}

float NCCEstimator::Get() {
   float meanx = sumx_ / sumw_;
   float meany = sumy_ / sumw_;
   float meanxx = sumxx_ / sumw_;
   float meanyy = sumyy_ / sumw_;
   float meanxy = sumxy_ / sumw_;
   float varx = meanxx - meanx * meanx;
   float vary = meanyy - meany * meany;
   if (varx < 0.1 || vary < 0.1) {
      return -1;
   } else {
      return (meanxy - meanx * meany) / sqrt(varx * vary);
   }
}

void ApplyHomography(const cv::Matx33f &H,
                     float x1, float y1,
                     float *x2, float *y2) {
   float w = H(2, 0) * x1 + H(2, 1) * y1 + H(2, 2);
   *x2 = (H(0, 0) * x1 + H(0, 1) * y1 + H(0, 2)) / w;
   *y2 = (H(1, 0) * x1 + H(1, 1) * y1 + H(1, 2)) / w;
}

cv::Matx33d PlaneInducedHomography(const cv::Matx33d &K1,
                                   const cv::Matx33d &R1,
                                   const cv::Vec3d &t1,
                                   const cv::Matx33d &K2,
                                   const cv::Matx33d &R2,
                                   const cv::Vec3d &t2,
                                   const cv::Vec3d &v) {
```

```
  cv::Matx33d R2R1 = R2 * R1.t();
  return K2 * (R2R1 + (R2R1 * t1 - t2) * v.t()) * K1.inv();
}

cv::Matx33f PlaneInducedHomographyBaked(const cv::Matx33d &K1inv,
                                        const cv::Matx33d &Q2,
                                        const cv::Vec3d &a2,
                                        const cv::Matx33d &K2,
                                        const cv::Vec3d &v) {
  return K2 * (Q2 + a2 * v.t()) * K1inv;
}

cv::Vec3d Project(const cv::Vec3d &x,
                  const cv::Matx33d &K,
                  const cv::Matx33d &R,
                  const cv::Vec3d &t) {
  return K * (R * x + t);
}

cv::Vec3d Backproject(double x, double y, double depth,
                      const cv::Matx33d &K,
                      const cv::Matx33d &R,
                      const cv::Vec3d &t) {
  return R.t() * (depth * K.inv() * cv::Vec3d(x, y, 1) - t);
}

float DepthOfPlaneBackprojection(double x, double y,
                                 const cv::Matx33d &K,
                                 const cv::Vec3d &plane) {
  float denom    = -(plane.t() * K.inv() * cv::Vec3d(x, y, 1))(0);
  return 1.0f / std::max(1e-6f, denom);
}

cv::Vec3f PlaneFromDepthAndNormal(float x, float y,
                                  const cv::Matx33d &K,
                                  float depth,
                                  const cv::Vec3f &normal) {
  cv::Vec3f point = depth * K.inv() * cv::Vec3d(x, y, 1);
  float denom = -normal.dot(point);
  return normal / std::max(1e-6f, denom);
}

float UniformRand(float a, float b) {
  return a + (b - a) * float(rand()) / RAND_MAX;
```

```
}

DepthmapEstimator::DepthmapEstimator()
    : patch_size_(7),
      min_depth_(0),
      max_depth_(0),
      num_depth_planes_(50),
      patchmatch_iterations_(3),
      min_patch_variance_(5 * 5),
      rng_{std::random_device{}()},
      uni_(0, 0),
      unit_normal_(0, 1) {}

void DepthmapEstimator::AddView(const double *pK, const double *pR,
                                const double *pt, const unsigned char
*pimage,
                                const unsigned char *pmask, int
width,
                                int height) {
  Ks_.emplace_back(pK);
  Rs_.emplace_back(pR);
  ts_.emplace_back(pt);
  Kinvs_.emplace_back(Ks_.back().inv());
  Qs_.emplace_back(Rs_.back() * Rs_.front().t());
  as_.emplace_back(Qs_.back() * ts_.front() - ts_.back());
  images_.emplace_back(cv::Mat(height, width, CV_8U, (void
*)pimage).clone());
  masks_.emplace_back(cv::Mat(height, width, CV_8U, (void
*)pmask).clone());
  std::size_t size = images_.size();
  int a = (size > 1) ? 1 : 0;
  int b = (size > 1) ? size - 1 : 0;
  uni_.param(std::uniform_int_distribution<int>::param_type(a, b));
}

void DepthmapEstimator::SetDepthRange(double min_depth, double
max_depth,
                                      int num_depth_planes) {
  min_depth_ = min_depth;
  max_depth_ = max_depth;
  num_depth_planes_ = num_depth_planes;
}

void DepthmapEstimator::SetPatchMatchIterations(int n) {
```

```
  patchmatch_iterations_ = n;
}

void DepthmapEstimator::SetPatchSize(int size) {
  patch_size_ = size;
}

void DepthmapEstimator::SetMinPatchSD(float sd) {
  min_patch_variance_ = sd * sd;
}

void DepthmapEstimator::ComputeBruteForce(DepthmapEstimatorResult
*result) {
  AssignMatrices(result);

  int hpz = (patch_size_ - 1) / 2;
  for (int i = hpz; i < result->depth.rows - hpz; ++i) {
    for (int j = hpz; j < result->depth.cols - hpz; ++j) {
      for (int d = 0; d < num_depth_planes_; ++d) {
        float depth =
            1 / (1 / min_depth_ + d * (1 / max_depth_ - 1 / min_depth_) /
                                  (num_depth_planes_ - 1));
        cv::Vec3f normal(0, 0, -1);
        cv::Vec3f plane = PlaneFromDepthAndNormal(j, i, Ks_[0], depth,
normal);
        CheckPlaneCandidate(result, i, j, plane);
      }
    }
  }
}

void DepthmapEstimator::ComputePatchMatch(DepthmapEstimatorResult
*result) {
  AssignMatrices(result);
  RandomInitialization(result, false);
  ComputeIgnoreMask(result);

  for (int i = 0; i < patchmatch_iterations_; ++i) {
    PatchMatchForwardPass(result, false);
    PatchMatchBackwardPass(result, false);
  }

  PostProcess(result);
}
```

```
void DepthmapEstimator::ComputePatchMatchSample(
      DepthmapEstimatorResult *result) {
   AssignMatrices(result);
   RandomInitialization(result, true);
   ComputeIgnoreMask(result);

   for (int i = 0; i < patchmatch_iterations_; ++i) {
      PatchMatchForwardPass(result, true);
      PatchMatchBackwardPass(result, true);
   }

   PostProcess(result);
}

void DepthmapEstimator::AssignMatrices(DepthmapEstimatorResult *result)
{
   result->depth = cv::Mat(images_[0].rows, images_[0].cols, CV_32F, 0.0f);
   result->plane = cv::Mat(images_[0].rows, images_[0].cols, CV_32FC3,
0.0f);
   result->score = cv::Mat(images_[0].rows, images_[0].cols, CV_32F, 0.0f);
   result->nghbr = cv::Mat(images_[0].rows, images_[0].cols, CV_32S,
cv::Scalar(0));
}

void DepthmapEstimator::RandomInitialization(DepthmapEstimatorResult
*result,
                                             bool sample) {
   int hpz = (patch_size_ - 1) / 2;
   for (int i = hpz; i < result->depth.rows - hpz; ++i) {
      for (int j = hpz; j < result->depth.cols - hpz; ++j) {
         float depth = exp(UniformRand(log(min_depth_), log(max_depth_)));
         cv::Vec3f normal(UniformRand(-1, 1), UniformRand(-1, 1), -1);
         cv::Vec3f plane = PlaneFromDepthAndNormal(j, i, Ks_[0], depth,
normal);
         int nghbr;
         float score;
         if (sample) {
            nghbr = uni_(rng_);
            score = ComputePlaneImageScore(i, j, plane, nghbr);
         } else {
            ComputePlaneScore(i, j, plane, &score, &nghbr);
         }
         AssignPixel(result, i, j, depth, plane, score, nghbr);
```

```
        }
    }
}

void DepthmapEstimator::ComputeIgnoreMask(DepthmapEstimatorResult
*result) {
    int hpz = (patch_size_ - 1) / 2;
    for (int i = hpz; i < result->depth.rows - hpz; ++i) {
        for (int j = hpz; j < result->depth.cols - hpz; ++j) {
            bool masked = masks_[0].at<unsigned char>(i, j) == 0;
            bool low_variance = PatchVariance(i, j) < min_patch_variance_;
            if (masked || low_variance) {
                AssignPixel(result, i, j, 0.0f, cv::Vec3f(0, 0, 0), 0.0f, 0);
            }
        }
    }
}

float DepthmapEstimator::PatchVariance(int i, int j) {
    float patch[patch_size_ * patch_size_];
    int hpz = (patch_size_ - 1) / 2;
    int counter = 0;
    for (int u = -hpz; u <= hpz; ++u) {
        for (int v = -hpz; v <= hpz; ++v) {
            patch[counter++] = images_[0].at<unsigned char>(i + u, j + v);
        }
    }
    return Variance(patch, patch_size_ * patch_size_);
}

void
DepthmapEstimator::PatchMatchForwardPass(DepthmapEstimatorResult
*result,
                                                    bool sample) {
    int adjacent[2][2] = {{-1, 0}, {0, -1}};
    int hpz = (patch_size_ - 1) / 2;
    for (int i = hpz; i < result->depth.rows - hpz; ++i) {
        for (int j = hpz; j < result->depth.cols - hpz; ++j) {
            PatchMatchUpdatePixel(result, i, j, adjacent, sample);
        }
    }
}
```

```
void
DepthmapEstimator::PatchMatchBackwardPass(DepthmapEstimatorResult
*result,
                                          bool sample) {
  int adjacent[2][2] = {{0, 1}, {1, 0}};
  int hpz = (patch_size_ - 1) / 2;
  for (int i = result->depth.rows - hpz - 1; i >= hpz; --i) {
    for (int j = result->depth.cols - hpz - 1; j >= hpz; --j) {
      PatchMatchUpdatePixel(result, i, j, adjacent, sample);
    }
  }
}

void
DepthmapEstimator::PatchMatchUpdatePixel(DepthmapEstimatorResult
*result,
                                         int i, int j, int
adjacent[2][2],
                                         bool sample) {
  // Ignore pixels with depth == 0.
  if (result->depth.at<float>(i, j) == 0.0f) {
    return;
  }

  // Check neighbors and their planes for adjacent pixels.
  for (int k = 0; k < 2; ++k) {
    int i_adjacent = i + adjacent[k][0];
    int j_adjacent = j + adjacent[k][1];

    // Do not propagate ignored adjacent pixels.
    if (result->depth.at<float>(i_adjacent, j_adjacent) == 0.0f) {
      continue;
    }

    cv::Vec3f plane = result->plane.at<cv::Vec3f>(i_adjacent, j_adjacent);

    if (sample) {
      int nghbr = result->nghbr.at<int>(i_adjacent, j_adjacent);
      CheckPlaneImageCandidate(result, i, j, plane, nghbr);
    } else {
      CheckPlaneCandidate(result, i, j, plane);
    }
  }
```

```
    // Check random planes for current neighbor.
    float depth_range = 0.02;
    float normal_range = 0.5;
    int current_nghbr = result->nghbr.at<int>(i, j);
    for (int k = 0; k < 6; ++k) {
      float current_depth = result->depth.at<float>(i, j);
      float depth = current_depth * exp(depth_range * unit_normal_(rng_));

      cv::Vec3f current_plane = result->plane.at<cv::Vec3f>(i, j);
      cv::Vec3f normal(-current_plane(0) / current_plane(2)
                           + normal_range * unit_normal_(rng_),
                       -current_plane(1) / current_plane(2)
                           + normal_range * unit_normal_(rng_),
                       -1.0f);

      cv::Vec3f plane = PlaneFromDepthAndNormal(j, i, Ks_[0], depth,
normal);
      if (sample) {
        CheckPlaneImageCandidate(result, i, j, plane, current_nghbr);
      } else {
        CheckPlaneCandidate(result, i, j, plane);
      }

      depth_range *= 0.3;
      normal_range *= 0.8;
    }

    if (!sample || images_.size() <= 2) {
      return;
    }

    // Check random other neighbor for current plane.
    int other_nghbr = uni_(rng_);
    while (other_nghbr == current_nghbr) {
      other_nghbr = uni_(rng_);
    }

    cv::Vec3f plane = result->plane.at<cv::Vec3f>(i, j);
    CheckPlaneImageCandidate(result, i, j, plane, other_nghbr);
}

void DepthmapEstimator::CheckPlaneCandidate(DepthmapEstimatorResult
*result,
                                            int i, int j,
```

```
                                                    const cv::Vec3f
&plane) {
  float score;
  int nghbr;
  ComputePlaneScore(i, j, plane, &score, &nghbr);
  if (score > result->score.at<float>(i, j)) {
    float depth = DepthOfPlaneBackprojection(j, i, Ks_[0], plane);
    AssignPixel(result, i, j, depth, plane, score, nghbr);
  }
}

void DepthmapEstimator::CheckPlaneImageCandidate(
    DepthmapEstimatorResult *result, int i, int j, const cv::Vec3f &plane,
    int nghbr) {
  float score = ComputePlaneImageScore(i, j, plane, nghbr);
  if (score > result->score.at<float>(i, j)) {
    float depth = DepthOfPlaneBackprojection(j, i, Ks_[0], plane);
    AssignPixel(result, i, j, depth, plane, score, nghbr);
  }
}

void DepthmapEstimator::AssignPixel(DepthmapEstimatorResult *result, int
i,
                                    int j, const float depth,
                                    const cv::Vec3f &plane, const
float score,
                                    const int nghbr) {
  result->depth.at<float>(i, j) = depth;
  result->plane.at<cv::Vec3f>(i, j) = plane;
  result->score.at<float>(i, j) = score;
  result->nghbr.at<int>(i, j) = nghbr;
}

void DepthmapEstimator::ComputePlaneScore(int i, int j, const cv::Vec3f
&plane,
                                          float *score, int *nghbr)
{
  *score = -1.0f;
  *nghbr = 0;
  for (int other = 1; other < images_.size(); ++other) {
    float image_score = ComputePlaneImageScore(i, j, plane, other);
    if (image_score > *score) {
      *score = image_score;
      *nghbr = other;
```

```
      }
    }
}

float DepthmapEstimator::ComputePlaneImageScoreUnoptimized(
      int i, int j, const cv::Vec3f &plane, int other) {
    cv::Matx33f H = PlaneInducedHomographyBaked(
        Kinvs_[0], Qs_[other], as_[other], Ks_[other], plane);
    int hpz = (patch_size_ - 1) / 2;
    float im1_center = images_[0].at<unsigned char>(i, j);
    NCCEstimator ncc;
    for (int dy = -hpz; dy <= hpz; ++dy) {
      for (int dx = -hpz; dx <= hpz; ++dx) {
        float im1 = images_[0].at<unsigned char>(i + dy, j + dx);
        float x2, y2;
        ApplyHomography(H, j + dx, i + dy, &x2, &y2);
        float im2 = LinearInterpolation<unsigned char>(images_[other], y2,
x2);
        float weight = BilateralWeight(im1 - im1_center, dx, dy);
        ncc.Push(im1, im2, weight);
      }
    }
    return ncc.Get();
}

float DepthmapEstimator::ComputePlaneImageScore(int i, int j,
                                                const cv::Vec3f
&plane,
                                                int other) {
    cv::Matx33f H = PlaneInducedHomographyBaked(
        Kinvs_[0], Qs_[other], as_[other], Ks_[other], plane);
    int hpz = (patch_size_ - 1) / 2;

    float u = H(0, 0) * j + H(0, 1) * i + H(0, 2);
    float v = H(1, 0) * j + H(1, 1) * i + H(1, 2);
    float w = H(2, 0) * j + H(2, 1) * i + H(2, 2);

    float dfdx_x = (H(0, 0) * w - H(2, 0) * u) / (w * w);
    float dfdx_y = (H(1, 0) * w - H(2, 0) * v) / (w * w);
    float dfdy_x = (H(0, 1) * w - H(2, 1) * u) / (w * w);
    float dfdy_y = (H(1, 1) * w - H(2, 1) * v) / (w * w);

    float Hx0 = u / w;
    float Hy0 = v / w;
```

```
    float im1_center = images_[0].at<unsigned char>(i, j);

    NCCEstimator ncc;
    for (int dy = -hpz; dy <= hpz; ++dy) {
      for (int dx = -hpz; dx <= hpz; ++dx) {
        float im1 = images_[0].at<unsigned char>(i + dy, j + dx);
        float x2 = Hx0 + dfdx_x * dx + dfdy_x * dy;
        float y2 = Hy0 + dfdx_y * dx + dfdy_y * dy;
        float im2 = LinearInterpolation<unsigned char>(images_[other], y2,
x2);
        float weight = BilateralWeight(im1 - im1_center, dx, dy);
        ncc.Push(im1, im2, weight);
      }
    }
    return ncc.Get();
}

float DepthmapEstimator::BilateralWeight(float dcolor, float dx, float dy) {
    const float dcolor_sigma = 50.0f;
    const float dx_sigma = 5.0f;
    const float dcolor_factor = 1.0f / (2 * dcolor_sigma * dcolor_sigma);
    const float dx_factor = 1.0f / (2 * dx_sigma * dx_sigma);
    return exp(
      - dcolor * dcolor * dcolor_factor
      - (dx * dx + dy * dy) * dx_factor
    );
}

void DepthmapEstimator::PostProcess(DepthmapEstimatorResult *result) {
    cv::Mat depth_filtered;
    cv::medianBlur(result->depth, depth_filtered, 5);

    for (int i = 0; i < result->depth.rows; ++i) {
      for (int j = 0; j < result->depth.cols; ++j) {
        float d = result->depth.at<float>(i, j);
        float m = depth_filtered.at<float>(i, j);
        if (fabs(d - m) / d > 0.05) {
          result->depth.at<float>(i, j) = 0;
        }
      }
    }
}
```

```
DepthmapCleaner::DepthmapCleaner()
      : same_depth_threshold_(0.01), min_consistent_views_(2) {}

void DepthmapCleaner::SetSameDepthThreshold(float t) {
   same_depth_threshold_ = t;
}

void DepthmapCleaner::SetMinConsistentViews(int n) {
   min_consistent_views_ = n;
}

void DepthmapCleaner::AddView(const double *pK, const double *pR,
                                          const double *pt, const float *pdepth,
int width,
                                          int height) {
   Ks_.emplace_back(pK);
   Rs_.emplace_back(pR);
   ts_.emplace_back(pt);
   depths_.emplace_back(cv::Mat(height, width, CV_32F, (void
*)pdepth).clone());
}

void DepthmapCleaner::Clean(cv::Mat *clean_depth) {
   *clean_depth = cv::Mat(depths_[0].rows, depths_[0].cols, CV_32F, 0.0f);

   for (int i = 0; i < depths_[0].rows; ++i) {
      for (int j = 0; j < depths_[0].cols; ++j) {
         float depth = depths_[0].at<float>(i, j);
         cv::Vec3f point = Backproject(j, i, depth, Ks_[0], Rs_[0], ts_[0]);
         int consistent_views = 1;
         for (int other = 1; other < depths_.size(); ++other) {
            cv::Vec3f reprojection = Project(point, Ks_[other], Rs_[other],
ts_[other]);
            float u = reprojection(0) / reprojection(2);
            float v = reprojection(1) / reprojection(2);
            float depth_of_point = reprojection(2);
            float depth_at_reprojection =
LinearInterpolation<float>(depths_[other], v, u);
            if (fabs(depth_at_reprojection - depth_of_point) < depth_of_point *
same_depth_threshold_) {
               consistent_views++;
            }
         }
         if (consistent_views >= min_consistent_views_) {
```

```
            clean_depth->at<float>(i, j) = depths_[0].at<float>(i, j);
          } else {
            clean_depth->at<float>(i, j) = 0;
          }
        }
      }
    }

DepthmapPruner::DepthmapPruner() : same_depth_threshold_(0.01) {}

void DepthmapPruner::SetSameDepthThreshold(float t) {
  same_depth_threshold_ = t;
}

void DepthmapPruner::AddView(const double *pK, const double *pR,
                             const double *pt, const float *pdepth,
                             const float *pplane, const unsigned char
*pcolor,
                             const unsigned char *plabel, const
unsigned char *pdetection,
                             int width, int height) {
  Ks_.emplace_back(pK);
  Rs_.emplace_back(pR);
  ts_.emplace_back(pt);
  depths_.emplace_back(cv::Mat(height, width, CV_32F, (void
*)pdepth).clone());
  planes_.emplace_back(
      cv::Mat(height, width, CV_32FC3, (void *)pplane).clone());
  colors_.emplace_back(cv::Mat(height, width, CV_8UC3, (void
*)pcolor).clone());
  labels_.emplace_back(cv::Mat(height, width, CV_8U, (void
*)plabel).clone());
  detections_.emplace_back(cv::Mat(height, width, CV_8U, (void
*)pdetection).clone());
}

void DepthmapPruner::Prune(std::vector<float> *merged_points,
                           std::vector<float> *merged_normals,
                           std::vector<unsigned char>
*merged_colors,
                           std::vector<unsigned char>
*merged_labels,
                           std::vector<unsigned char>
*merged_detections) {
```

```
    cv::Matx33f Rinv = Rs_[0].t();
    for (int i = 0; i < depths_[0].rows; ++i) {
      for (int j = 0; j < depths_[0].cols; ++j) {
        float depth = depths_[0].at<float>(i, j);
        if (depth <= 0) {
          continue;
        }
        cv::Vec3f normal = cv::normalize(planes_[0].at<cv::Vec3f>(i, j));
        float area = -normal(2) / depth * Ks_[0](0, 0);
        cv::Vec3f point = Backproject(j, i, depth, Ks_[0], Rs_[0], ts_[0]);
        bool keep = true;
        for (int other = 1; other < depths_.size(); ++other) {
          cv::Vec3d reprojection = Project(point, Ks_[other], Rs_[other],
ts_[other]);
          int iu = int(reprojection(0) / reprojection(2) + 0.5f);
          int iv = int(reprojection(1) / reprojection(2) + 0.5f);
          float depth_of_point = reprojection(2);
          if (!IsInsideImage(depths_[other], iv, iu)) {
            continue;
          }
          float depth_at_reprojection = depths_[other].at<float>(iv, iu);
          if (depth_at_reprojection > (1 - same_depth_threshold_) *
depth_of_point) {
            cv::Vec3f normal_at_reprojection =
cv::normalize(planes_[other].at<cv::Vec3f>(iv, iu));
            float area_at_reprojection = -normal_at_reprojection(2) /
depth_at_reprojection * Ks_[other](0, 0);
            if (area_at_reprojection > area) {
              keep = false;
              break;
            }
          }
        }
        if (keep) {
          cv::Vec3f R1_normal = Rinv * normal;
          cv::Vec3b color = colors_[0].at<cv::Vec3b>(i, j);
          unsigned char label = labels_[0].at<unsigned char>(i, j);
          unsigned char detection = detections_[0].at<unsigned char>(i, j);
          merged_points->push_back(point[0]);
          merged_points->push_back(point[1]);
          merged_points->push_back(point[2]);
          merged_normals->push_back(R1_normal[0]);
          merged_normals->push_back(R1_normal[1]);
          merged_normals->push_back(R1_normal[2]);
```

```
                merged_colors->push_back(color[0]);
                merged_colors->push_back(color[1]);
                merged_colors->push_back(color[2]);
                merged_labels->push_back(label);
                merged_detections->push_back(detection);
            }
        }
    }
}

}
```