

**北京航空航天大学**

物理研究性实验报告

基于图像识别和追踪算法的密立根油滴实验改进

第一作者：14231014 金晖明 所在院系：高等工程学院

第二作者：14231010 武洋阳 所在院系：高等工程学院

第三作者：14231043 马长征 所在院系：高等工程学院

摘 要

本文使用计算机视觉的Hough变换识别算法与CMT追踪算法，对密立根油滴的油滴图像获取与处理过程进行改进，使用计算机对油滴进行识别与追踪，计算出每个油滴的速度并筛选出合适速度的油滴。这一方法极大地简化了人工寻找合适油滴的过程，增加了实验精度，减少了时间与实验油的消耗。本报告先介绍了密立根油滴的实验的原理与传统实验过程，在第五章详细介绍了实验改进的原理，在第六章介绍了实验改进的实现。通过模拟实验，得到结论：本文介绍的算法具有很好的识别与追踪能力，实用性很高。

**关键词**：密立根油滴实验，Hough变换，CMT追踪

目 录

一.报告概述┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄ 3

二.实验原理┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄ 3

三.实验仪器┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄ 5

四.实验步骤┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄ 5

五.实验改进┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄ 5

六.方案实现┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄ 6

七.应用前景┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄ 8

八.分析总结┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄ 9

九.参考文献┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄10

十.代码附录┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄┄10

# 报告概述

该实验报告的灵感基于笔者在进行密立根油滴实验时寻找合适油滴以及计时过程中的体验。在本实验中，实验者大量的时间都耗费在了找到屏幕中合适的油滴上。由于人眼只能追踪一个油滴，导致在每次的筛选过程中由于人的预判可能有误，在预先选定的点移动速度不合适的情况下，也会致使很多其它可能满足移动速度要求的点同时被移出视野，直接导致了大量实验油的浪费。本次研究性改进的目的在于用数字化手段快速找到合适的油滴快速开始实验甚至通过数字手段得到测量数据。在本实验报告中我们会详细叙述让摄像头以及计算机自主寻找合适油滴的算法以及原理并且附上代码便于今后的应用与优化。

# 实验原理

本实验被誉为物理实验中的典范，设计此实验的科学家密立根也因此实验以及其它关于电子电荷的研究获得了1923年诺贝尔物理学奖。由于电子电荷十分微小，该实验是通过测定大量带电油滴的带电量，找到其中的公约数从而确定单个电子的带电量的。

在本实验中，我们采用的是静态平衡法测量带电油滴的电量：

用喷雾器将油滴喷入两块相距为d的平行极板之间。油在喷射撕裂成油滴时，一般都是带电的。设油滴的质量为m，所带的电量为q，两极板间的电压为V ，如图 1 所示。

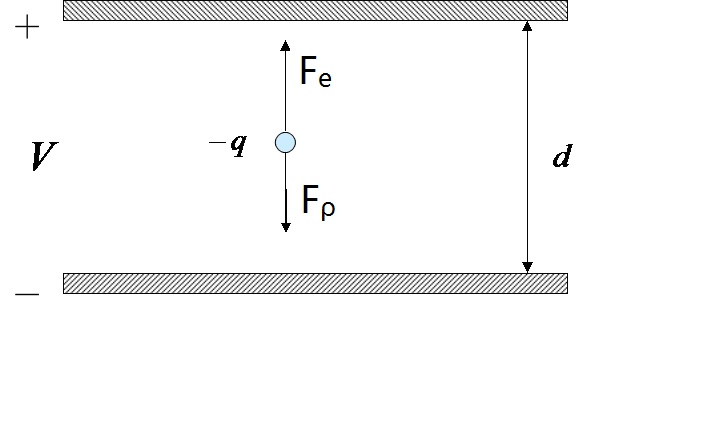


图1

如果调节两极板间的电压V ，可使两力达到平衡，这时：

 (1)

为了测出油滴所带的电量q，除了需测定平衡电压V 和极板间距离d 外，还需要测量油滴的质量m。因m很小，需用如下特殊方法测定：平行极板不加电压时，油滴受重力作用而加速下降，由于空气阻力的作用，下降一段距离达到某一速度后，阻力与重力mg平衡，如图 2 所示（空气浮力忽略不计），油滴将匀速下降。此时有：

 (2)

其中是空气的粘滞系数，是油滴的半径。经过变换及修正，可得斯托克斯定律：

 (3)

其中b是修正常数， b=6.17×10-6m·cmHg,p为大气压强，单位为厘米汞高。

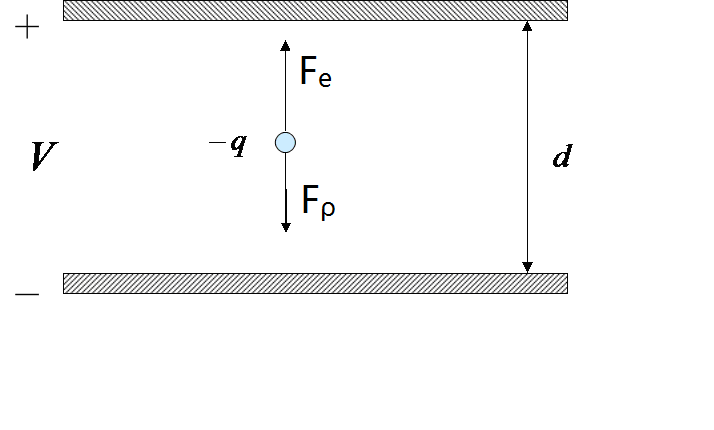


图2

至于油滴匀速下降的速度，可用下法测出：当两极板间的电压V为零时,设油滴匀速下降的距离为，时间为t ，则

 (4)

最后得到理论公式：

 (5)

# 实验仪器

根据实验原理，实验仪器——密立根油滴仪，应包括水平放置的平行极板（油滴盒），调平装置，照明装置，显微镜，电源，计时器（集成于密立根油滴仪之中），实验油，喷雾器等。

# 实验步骤

原始的密立根油滴实验有大致三个部分：调整仪器、练习测量和正式测量。

调整仪器之后，在原始实验的练习测量部分，我们首先要练习控制油滴，即调节好平衡电压之后，观察屏幕中大量的油滴哪些处于相对平衡的状态。这时通过调节平衡电压可以使油滴完全平衡。加大电压、平衡电压、0V电压分别对应了带电油滴的三种状态：上升、静止和下降。这时可以进行下一步练习。

练习测量油滴下降的时间：通过控制油滴，首先让油滴静止在起始线上。这时按下0V电压的按钮，油滴开始下降，由于按下了密立根油滴实验装置上的”联动“按钮，设备能够自己计时，从开始下降开始，向下匀速移动6个格子，此时停止计时，得到的时间便是我们实验中的待测量。

下一步的练习是选择合适的油滴。在已经达到平衡的白点中选择不是很亮也不是很大的油滴（很亮很大的油滴肯定会下降速度过快导致误差很大），这时使油滴向下运动，选择运动轨迹稳定（布朗运动可以忽略）且经过六格的时间在8-25s之间的油滴。

在正式测量中更改十种在120-300V之间的不同的电压值每组测量五组数据即可。

# 实验改进

5.1 实验改进的整体思路与步骤

我们所作的改进是将CCD摄像头获取到的图像利用计算机进行图像识别，利用设计的算法使得挑选油滴的过程更加精确并省时省力。改进的整体思路与步骤如下：

1. 将电压调至0V。
2. 油滴盒内的图像通过CCD传入计算机。
3. 使用Hough算法实时识别出图像中的每个油滴。
4. 使用CMT算法对每个油滴进行小段时间的追踪。
5. 计算机找到符合要求的油滴后，在屏幕上将其标记出来，待进一步测量。

本方法的好处是：摄像头、油滴盒均保持不动，画面中油滴的移动均是因为其自身的运动，因此通过油滴图像的运动情况，可以计算出油滴的运动情况，这和改进前的油滴实验所使用的测量技术是相同的。

5.2 利用Hough变换算法识别油滴

首先，我们需要在图像中选取待追踪的对象，即油滴的影像。此环节采用的是Hough变换算法。Hough变换算法的作用是从复杂的图像中找到目标形状，一般是从复杂图像中定位圆形或者直线等等[1]，该算法最早由Hough在1962年提出。

在密立根油滴实验中，油滴的形状恰好都是圆形，这就使得Hough变换成为了识别算法的首选，在通过代码实现之后也确实证明了通过Hough变换来识别油滴的效果很好。

Hough变换算法的主要思想是利用点与线的对偶性，将图像空间的线条变为参数空间的一个点，从而检测给定图像是否存在给定性质的曲线。例如二维平面上的圆曲线：

(6)

如果我们已知圆的圆心和半径r，就可以知道圆上每一个点的坐标，但如果现在知道的是一些散点的坐标，想获得圆的圆心和半径该怎么办呢？Hough变换是这样做的：首先我们对图像进行二值化处理，并对其进行边缘检测，找到一些特征点。然后将圆方程的参数与变量对调，得到一个新的方程：

(7)

这个方程和之前的方程的形式是类似的，但区别在于、、r变成了变量，而和是参数，代表每个像素点的横纵坐标。假如我们有n个特征点，就有n个这样的方程，每个方程代表三维空间中的一个圆锥，当n很大时，n个圆锥必然交于同一点，这个点的坐标就是，这个点的三个坐标就是我们所要求的方程(6)中的三个参数。

在实际使用中，三个维度的处理计算量较大，因此我们做一些技术上的处理：在边缘检测时，同时可以检测出边缘的法线方程，圆心必然落于法线上，因此圆心的坐标可由n条法线的交点得到。

当图像中有许多圆时，会有好几个交点。同时，图像会有一些噪声，这时法线的交点就会有许许多多个，但他们具有一定的特点：即属于同一个圆的特征点的法线的交点会比较集中，即分布在理论上圆心的周围，因此我们使用聚类算法得到几个交点的聚集中心，每个聚集中心就是一个圆心。

5.3 利用CMT算法追踪油滴并计算速度

识别油滴之后，计算机便可以记录油滴的位置。接下来计算机需要追踪其位置，这样就可以计算出油滴的速度。正如实验者所做的那样，计算机需要筛选出哪些油滴的下降速度适中。人眼判断下降速度是不准确的，在该实验中唯一判断下降速度的办法就是让油滴下降六格的距离，通过所用的时间（8至25秒）来判断下降的速度是否合适。这样的过程不但使选择合适的油滴耗费大量时间，而且在每次将电压设为0V时，所有平衡的油滴都会向下移动，如果预判不准确，之前认定的油滴会被舍弃，而其它可能符合要求的油滴也会被移出视野。这样就导致要多次喷油。经过实际观察，很多同学都会耗费大量实验油来找到合适的油滴。然而计算机视觉能够弥补人只能关注于一点的缺憾。通过追踪算法，计算机可以追踪视频中所有预设好的物体，通过移动的距离和时间就可计算出物体（油滴）的速度，进而筛选出速度合适的油滴，并进行标记，供进一步实验。所谓追踪，所实现的就是在视频一帧一帧变化的过程中，在每一帧中都框住上一帧中所框的物体，这里的物体在我们的实验中指的就是油滴。

下面介绍本文的第二个关键算法：CMT追踪算法。这个算法的全称是Clustering of Static-Adaptive Correspondences for Deformable Object Tracking，发表于CVPR2015会议。这个算法中的主要思想是：记录物体的每个特征点的信息，在每一帧图像中都通过特征点提取算法（例如SIFT、FAST等，这些也都是成熟的计算机视觉算法，在此不深入展开）获取一次特征点，然后与上一帧的特征点进行匹配，通过匹配上的特征点就能得到物体新的位置。

CMT算法的整体流程如下[2]：

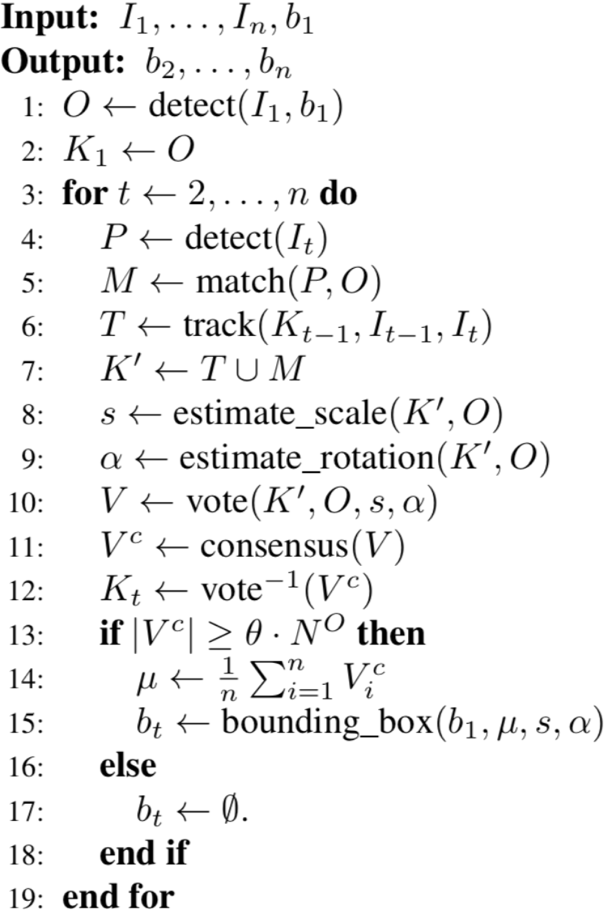


图3 CMT算法的伪代码

1. 获取第一帧的特征点集O。
2. 将O存入K中。
3. 从第2帧到第n帧做循环。
4. 获取第t帧的特征点集P。
5. 对O和P中的特征点进行匹配（利用特征点周围的信息），将匹配到的点存入M中。
6. 用光流法，利用两帧图像和上一帧的特征点，计算出这一帧特征点的估计位置，存入T中。
7. 将T和M合并，存入。
8. 估计相对O的伸缩比例s。
9. 估计相对O的旋转比例。
10. 利用每个特征点的旋转和伸缩比例进行对旋转、伸缩中心进行投票，这样的话大多数特征点的旋转、伸缩中心会得到较高的票数。
11. 由于可能有误差，因此使用聚类方法得出旋转、伸缩中心的聚点，作为旋转，伸缩中心。
12. 利用投票信息反算出有效特征点，也就是说将票投给旋转、伸缩中心的特征点是有效特征点。
13. 从这一步开始，就是利用有效特征点计算出包围物体的框。

这个算法有两个关键点，一是对特征点进行匹配，二是利用物体的形状不变性筛选出有效特征点。

首先利用特征点周围的信息进行匹配，SIFT、FAST等特征点提取算法会得到每个特征点周围几个像素的信息，因此周围像素相似的特征点可以相互匹配上。Hamming距离是常用的特征匹配度量，它的定义如下：

(8)

其中XOR代表异或运算，代表两个特征点及其背景的差距的大小，这个值越小则两个特征点越相似。

光匹配上特征点还不行，因为可能有很多相似的特征点，这就需要使用光流法得出最优可能的相匹配的特征点。光流这个概念最早由Gibson在1950提出，它是空间运动物体在观察成像平面上的像素运动的瞬时速度。光流法是一种基于图像亮度的运动识别方法，它利用图像序列中各个像素在时间上的变化以及相邻帧之间的相关性来找到上一帧跟当前帧之间存在的对应关系，从而计算出相邻帧之间物体的运动信息的一种方法。具体实现方法很多，在此就不再展开。

根据一个基本假设：相邻帧之间特征点的运动很小，可以得出一个结论：根据光流法算出的运动方向上的相邻最近的匹配上的特征点是我们要找的新的一帧的特征点。利用光流法加上特征点匹配，我们就可以得出一些我们要追踪的物体在新的一帧中的特征点。

为了提高匹配成功率，筛选出最好的特征点，还需利用被追踪物体的几何不变性对特征点进一步进行筛选。物体可能进行三个维度上的运动，但物体各个特征点之间的距离变化比例以及旋转比例不会变化，因此我们可以计算出任意两个特征点之间的伸缩比例和相对于某一点的旋转角度。即使计算会出现误差，被追踪物体上的特征点的伸缩、旋转特性一定是相似的，因此我们可以使用聚类的方法得到伸缩、旋转特性最相似的点的集合，就可以反求出有效的特征点。

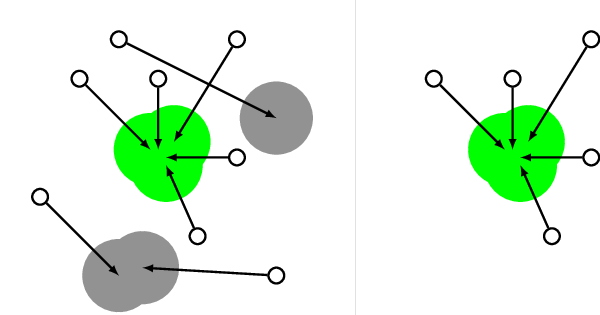


图4 CMT算法聚类示意，其中绿色区域是聚类后得到的旋转中心

# 方案实现

由于自行改进CCD摄像头及其电路是条件所不允许的，我们不能破坏实验室现有的实验仪器，而本实验的改进主要是在该实验中加入了计算机视觉的成分，其中算法是主要部分。因此我们使用开源的OpenCV机器视觉框架编写了手机软件，使用手机的摄像头来模拟油滴实验仪中的CCD摄像头。

起初我们尝试了使用手机来识别实验室中小屏幕里显示的图像，但实际发现这样做的效果并不好，因为实验室的灯光较亮，并且实验室中的小屏幕上同时会显示出网格线、电压和时间，这些会对我们的识别算法产生干扰。虽然我们的程序仍然可以捕获到一些油滴，但识别率不高。



图5 在实验室现场试验



图6 实验室现场试验的结果图

于是我们进行了思考，既然我们的重点是算法，我们没有必要让图像经过两次摄像头，这样既违反初衷，也增大了误差。因此我们继续使用手机摄像头和手机软件，但是使用黑暗环境和没有网格线、文字显示的模拟程序来模拟油滴试验仪，如图7所示。

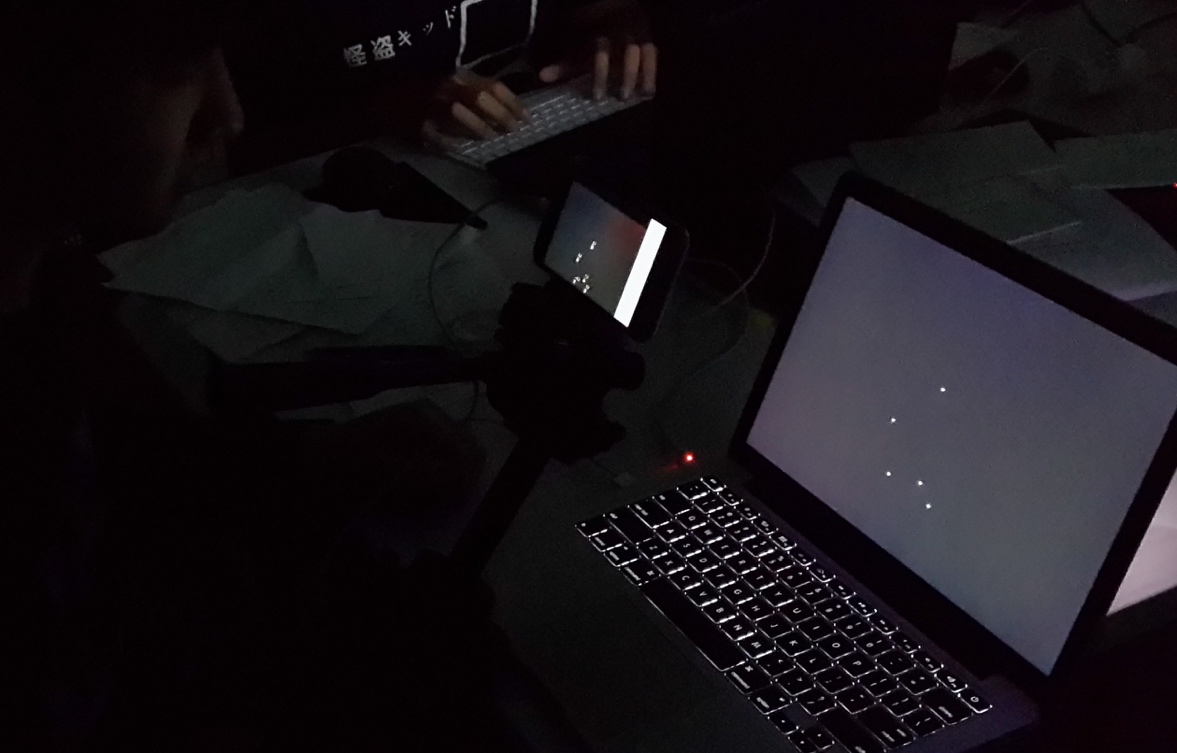


图7 在黑暗环境下模拟油滴实验仪

这一次，我们实验的结果很好，我们的程序可以识别出每一个油滴点，并对其进行追踪。为了便于观察，我们用框框住了每一个油滴点，并显示了其速度（微米/秒），这里的速度是使用逐差法计算出的一小段时间内的速度，近似可以认为是瞬时速度，并且是经过一定比例缩放的，为的是模拟油滴实验仪中的环境，因此单位是微米/秒。其中符合我们要求的油滴点用红色框框住，不符合要求的油滴点用绿色框框住。

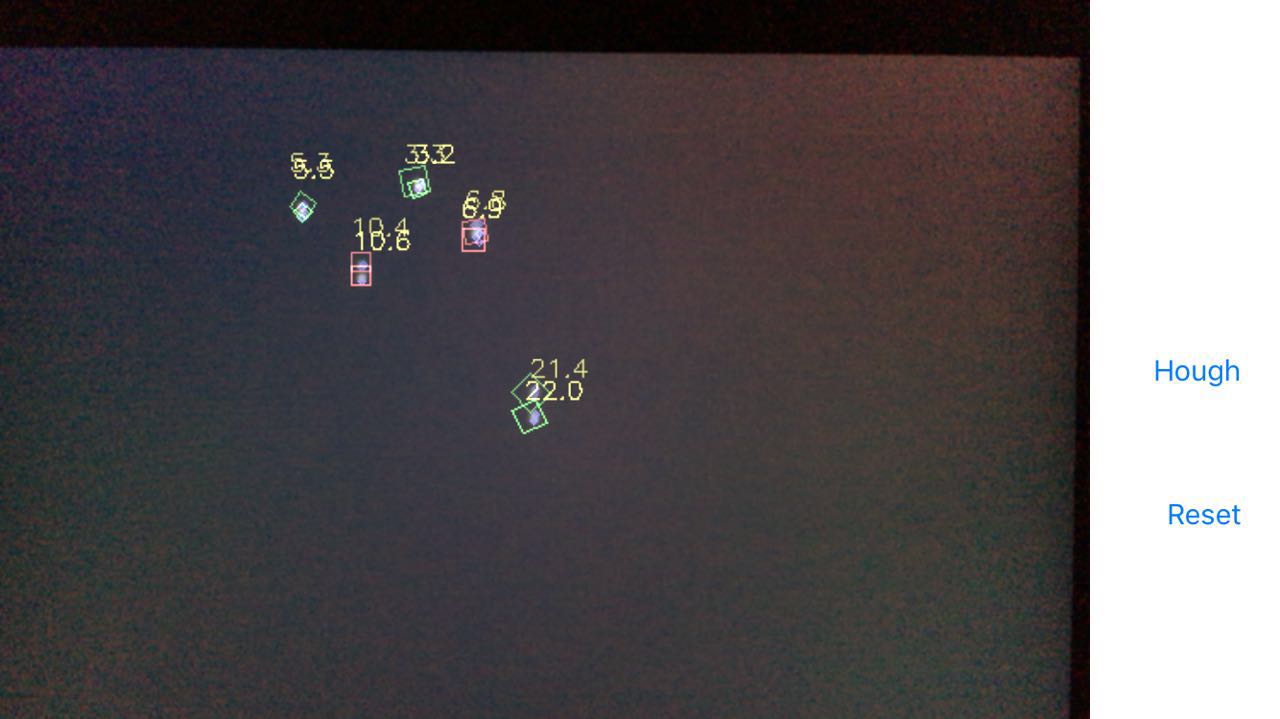


图8 实验时的手机屏幕截图，由于有图像的运动，因此图像有些重影

从图8中可以看出，最下面的点运动过快，因此需要被舍弃。最上面的两个点运动过慢，也不能用于测量。中间两个点的速度较为合适，被我们的程序识别出来并用红框进行了标记。实验结果令人满意，通过图像计算出的速度与我们预先在油滴模拟程序中设定的速度符合得很好。

# 应用前景

本文的思路与算法能够应用在很多机械化的可识别图像模式之中。只要在实验过程中CCD采集的图像之中有物体是保持不变的，就可以通过算法采集特征点实现追踪，返回物体的位置随时间的变化。实现密立根油滴实验的一套算法理论上能够推广到任何二维图像的数据采集工作中。

本文研究的结果对提高实验的自动化程度与精度有很大帮助，可以免去人的操作这一不稳定因素，并且可以节省实验人员的时间。

# 分析总结

8.1 实验结果分析

在我们的模拟实验中，我们的算法的实现令人满意。在多次的实验中，我们的程序均能找到每个油滴，进行追踪并找出速度合适的油滴。通过我们的程序，可以很大地提高密立根油滴实验的实验速度，甚至可以进一步研究机器自动化实验。

但问题也是存在的，本文使用的算法计算量较大，当油滴过多时图像会有明显的 计算延迟，这会降低实验的精度。如果进行进一步改进，则需要努力降低算法的时间复杂度，以降低图像延迟。

8.2 收获与感想

在本次实验改进中，我们用自己的思考和制作与实践大幅简化了密立根油滴实验的复杂程度，也大幅缩短了实验时间。

通过对理论想法进行实践上的实现，我们发现实践过程充满了艰辛。编写程序、解决BUG（程序错误），模拟实验。这些步骤并不像想象中的那么简单。通过这次的研究学习，我们学习了相关的计算机视觉知识，提高了自学能力与查阅文献能力，丰富了课外知识，坚实了动手写程序的能力。既让我们对物理实验有了更深的了解与思考，也让我们有了对科研的的初步认识，提高了发现问题解决问题的能力。为以后更复杂的专业学习与设计制作提供了经验。

# 参考文献

[1] Hough, P.V.C. Method and means for recognizing complex patterns[P]. U.S. Patent 3,069,654, Dec. 18, 1962.

[2] Nebehay G, Pflugfelder R. Clustering of Static-Adaptive Correspondences for Deformable Object Tracking[C]. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2015: 2784-2791.

# 代码附录

由于我们使用了OpenCV开源框架，相关的代码很多，因此我们只列出最主要的部分的代码。代码使用的语言是Objective-C++。

#define RATIO 640.0/568.0

#define HOUGH\_DETECT\_TIME 10

#define HOUGH\_TRACK\_TIME 0

#define HOUGH\_TRACK\_DIEDAI 200

#define JUDGE\_CENTER\_THRESHOLD 0.5

#define JUDGE\_RADIUS\_THRESHOLD 1

#define RADIUS\_RATE 2

#define TEXT\_NUM\_OFFSET 20

#define TEXT\_SIZE 0.5

#define COLOR\_NON\_SELECTED\_REC Scalar(0,255,0)

#define COLOR\_SELECTED\_REC Scalar(0,0,255)

#define TRACK\_Y\_SIZE 12

#define V\_MIN 0.0

#define V\_MAX 0.01875

#define SCREEN\_SCALE 0.2

#define SCREEN\_PIXELS 568.0

- (void)houghTracking:(cv::Mat &)image

{

static int track\_itor = 0;

static int tracklen = 0;

hough\_cnt++;

Mat img\_gray, img\_gray\_hough;

/\*转为灰度图\*/

cvtColor(image,img\_gray,CV\_RGB2GRAY);

/\*侦测阶段\*/

if ( hough\_cnt < HOUGH\_DETECT\_TIME ){

/\*转为另一份灰度图\*/

cvtColor(image,img\_gray\_hough,CV\_RGB2GRAY);

/\*模糊化，去除噪点\*/

GaussianBlur( img\_gray\_hough, img\_gray\_hough, cv::Size(9, 9), 2, 2 );

/\*找圆\*/

vector<Vec3f> t\_circles;

HoughCircles( img\_gray\_hough, t\_circles, CV\_HOUGH\_GRADIENT, 1, img\_gray\_hough.rows/18, 200, 5, 0, 20 );

/\*与之前找到的合并\*/

bool bj = true;

for (int j = 0; j< (int)t\_circles.size(); j++){

bj = true;

/\*对每个点，查看是否有与上一帧匹配的点，若匹配上则认为是同一个点\*/

for (int i = 0; i < (int)circles.size() && bj; i ++) {

if(sqrt(((circles[i][0]-t\_circles[j][0])\*(circles[i][0]-t\_circles[j][0]))+((circles[i][1]-t\_circles[j][1])\*(circles[i][1]-t\_circles[j][1]))) < circles[i][2]/JUDGE\_CENTER\_THRESHOLD && fabs((circles[i][2] - t\_circles[j][2])/circles[i][2]) < JUDGE\_RADIUS\_THRESHOLD ) {

/\*如果找到了相同的点，则更新老点的信息\*/

bj = false;

circles[i][0] = t\_circles[j][0];

circles[i][1] = t\_circles[j][1];

circles[i][2] = t\_circles[j][2];

circ\_box[i].x = circles[i][0] - RADIUS\_RATE\*circles[i][2];

circ\_box[i].y = circles[i][1] - RADIUS\_RATE\*circles[i][2];

circ\_box[i].width = circ\_box[i].height = 2\*RADIUS\_RATE\*circles[i][2];

circles\_curr\_y[i] = t\_circles[j][1];

circles\_last\_time[i] = [[NSDate date] timeIntervalSince1970];

}

}

/\*匹配失败，找到了新的圆\*/

if (bj){

circles.push\_back(\*new Vec3f(t\_circles[j][0],t\_circles[j][1],t\_circles[j][2]));

circ\_box.push\_back(cv::Rect(t\_circles[j][0] - RADIUS\_RATE\*t\_circles[j][2], t\_circles[j][1] - RADIUS\_RATE\*t\_circles[j][2], 2\*RADIUS\_RATE\*t\_circles[j][2], 2\*RADIUS\_RATE\*t\_circles[j][2]));

circles\_init\_y.push\_back(t\_circles[j][1]);

circles\_curr\_y.push\_back(t\_circles[j][1]);

circles\_init\_time.push\_back([[NSDate date] timeIntervalSince1970]);

circles\_last\_time.push\_back([[NSDate date] timeIntervalSince1970]);

}

}

/\*识别阶段结束，开始追踪\*/

beginInit = true;

}

/\*追踪阶段\*/

else {

if (beginInit) {

/\*清空追踪器\*/

if (circ\_trackers.size() != 0) {

for(track\_itor = 0; track\_itor < circ\_trackers.size(); track\_itor++) {

if(NULL != circ\_trackers[track\_itor]) {

delete circ\_trackers[track\_itor];

circ\_trackers[track\_itor] = NULL;

}

}

circ\_trackers.clear();

}

/\*初始化迭代器\*/

track\_itor = 0;

/\*计算圆的个数\*/

tracklen = (int)circ\_box.size();

/\*结束初始化开始阶段\*/

beginInit = false;

/\*开始追踪\*/

startTracking = true;

}

if (show\_box.size() < circ\_box.size()) {

/\*依次初始化每个追踪器\*/

/\*由于每次初始化时有时间间隔，因此依据前几次识别的y坐标预估出此时的y坐标\*/

circ\_box[track\_itor].y += ([[NSDate date] timeIntervalSince1970] - circles\_last\_time[track\_itor]) \* ((circles\_curr\_y[track\_itor] - circles\_init\_y[track\_itor]) / (circles\_last\_time[track\_itor] - circles\_init\_time[track\_itor]));

/\*存储初始化完成的点\*/

show\_box.push\_back(circ\_box[track\_itor]);

/\*创建追踪对象\*/

circ\_trackers.push\_back(new cmt::CMT());

/\*这个结构体用于计算速度\*/

struct trackY tempty;

for(int ti = 0; ti < TRACK\_Y\_SIZE; ti++) {

tempty.y[ti] = 0;

tempty.t[ti] = 0;

}

circles\_track\_y.push\_back(tempty);

/\*初始化追踪对象\*/

circ\_trackers[track\_itor] -> initialize(img\_gray, circ\_box[track\_itor]);

/\*初始化下一个油滴\*/

track\_itor++;

}

/\*初始化成功，开始正式追踪\*/

if (startTracking) {

/\*计算追踪对象个数\*/

int tracklen = (int)show\_box.size();

/\*依次更新每个对象的位置\*/

for(track\_itor = 0; track\_itor < tracklen; track\_itor++) {

/\*处理图像\*/

circ\_trackers[track\_itor]->processFrame(img\_gray);

/\*计算此时的y坐标\*/

double y = 0;

for(size\_t i = 0; i < circ\_trackers[track\_itor]->points\_active.size(); i++){

circle(image, circ\_trackers[track\_itor]->points\_active[i], 2, Scalar(255,0,0));

y += circ\_trackers[track\_itor]->points\_active[i].y;

}

y /= circ\_trackers[track\_itor]->points\_active.size();

/\*更新短时间内的速度\*/

for(int ti = TRACK\_Y\_SIZE - 2; ti >=0 ; ti--) {

circles\_track\_y[track\_itor].y[ti+1] = circles\_track\_y[track\_itor].y[ti];

circles\_track\_y[track\_itor].t[ti+1] = circles\_track\_y[track\_itor].t[ti];

}

circles\_track\_y[track\_itor].y[0] = y;

circles\_track\_y[track\_itor].t[0] = [[NSDate date] timeIntervalSince1970];

/\*用逐差法计算短时间内的速度\*/

double v = 0;

for(int ti = 0; ti < TRACK\_Y\_SIZE/2; ti++ ){

v += (circles\_track\_y[track\_itor].y[ti] - circles\_track\_y[track\_itor].y[ti+(TRACK\_Y\_SIZE/2)])/(circles\_track\_y[track\_itor].t[ti] - circles\_track\_y[track\_itor].t[ti+(TRACK\_Y\_SIZE/2)]);

}

v /= (TRACK\_Y\_SIZE/2);

v = v / SCREEN\_PIXELS \* SCREEN\_SCALE ;

/\*计算出包围的框\*/

RotatedRect rect = circ\_trackers[track\_itor]->bb\_rot;

Point2f vertices[4];

rect.points(vertices);

/\*判断速度，并用相应的颜色画出框\*/

for (int i = 0; i < 4; i++) {

if ( v >= V\_MIN && v <= V\_MAX ) {

line(image, vertices[i], vertices[(i+1)%4], COLOR\_SELECTED\_REC);

}

else {

line(image, vertices[i], vertices[(i+1)%4], COLOR\_NON\_SELECTED\_REC);

}

}

/\*显示速度\*/

putText( image, /\*输出图像\*/

cv::String([[NSString stringWithFormat:@"%.1f", v\*1000] UTF8String]), /\*输出字符串\*/

cv::Point(vertices[0].x, vertices[0].y - TEXT\_NUM\_OFFSET), /\*输出位置，字符串的左下角位于这个点\*/

FONT\_HERSHEY\_SIMPLEX, /\*字体\*/

TEXT\_SIZE, /\*字体大小\*/

cvScalar(0,255,255) /\*颜色，顺序为BGR\*/

);

}

}

}

}