关于在瞳孔检测定位项目中的一些总结

--主要是关于代码优化方面的总结

接手这个项目也已经有几个月了，虽然还没有全部弄好，但刚好现在有空，而且也快到 2017 年春节了，打算随便讲述下这几个月在这个项目上的一些心得体会、还有各种总结什么的，主要是写给自己的，所以会写得随便点，语法、是否通顺、逻辑顺序等等什么的也就比较随便了。

其实这个项目要做的就是在一张有人眼的图像上，识别出眼睛里的瞳孔位置，运用的算法是椭圆差分算法。椭圆差分算法大概意思是这样的：人眼的虹膜从不同角度看会呈现出不同形状的椭圆，而虹膜中心则是瞳孔。所以我们就是通过检测出虹膜的形状和位置，间接得到瞳孔位置。在一张灰度图像中，虹膜这个椭圆的周围差分最大，因为虹膜灰度值较低，与周围灰度值相差较大。也就是说，我们要找差分值最大的椭圆，这个椭圆，就是我们要找的虹膜，并得到瞳孔位置。算法大概就是这样的。

图 1

其实看起来这个算法并不难，我也觉得不难，没几个小时就把这个算法的大致思路写好了，代码当然没有优化，而且也还有 bug。一开始程序是对整个图像做暴力检索，老师也是这么跟我们说的，因为现在的 CPU 的性能已经比较好了，再加上后面还要用 GPU 跑，所以决定用暴力检索的方式。一开始用 CPU 一跑，当时等了俩小时都没跑出来一张图。。。/笑哭，当时也不知道它要跑多久，总不能就这么等吧，后来没办法，就把 640×480 的图像改成了在压缩为 200×150 的图像上跑，最后跑了一个小时，跑出来了，跑出来的效果并不好，只能说大致位置

对了，但是偏差有点大。当然这没问题，毕竟才刚开始，哪有那么容易就能把它弄好。首先要改善的就是时间，为什么呢？是这样的，这个最后是要在视频上不断读取每一帧图像，并检测定位出位置，如果算它每秒 25 帧，也就是识别一张图像只有 40ms 的时间,即使达不到,也得要在 100ms 以内吧。但是就算是 100ms，就算后面 GPU 能加速 10 倍 20 倍，那也还差远了。所以说，减少程序运行时间才是当时最主要的任务。等等，好像当时不是 200×150 的分辨率跑出的 1 个小时时间，好像是 120×90,还是多少来着，具体不记得了，不管它，这不是关键。后面开始做代码的优化，一直到现在，弄了几个月，当然毕竟是在学期中做的，只是在平时空闲时候才会做，所以比较慢。比如说，12 月初到 1 月初这段时间是考试月，整整一个月没弄。。。另外在这时间优化过程中，“顺便”把检测定位的效果也改善了。当中不断改进程序运行时间，到现在最好的程序版本跑的平均时间是：30ms 左右（如果是在比较好的台式机上测，估计能进 15ms）。中间的具体过程记不清了，就大概把这些有用的方法写下来。这些不是按时间顺序写的，想到什么就写什么。

1. 将用到的 cosΘ、sinΘ、cosα 等值在在一开始程序初始化的时候就计算好并保存在数组中，这样后面用到的时候只需要查表就可以使用，否则这样重复计算的话非常耗时间，毕竟正余弦函数的时间开销可不小，如果我没记错的话，当时做了这项改进后，运行速度大约提升几倍。

2. 先做大致定位，再在这基础上进行微调。我的具体做法是，先将图像分辨率降低为 108×81,在这个图上做检测的话，能极大地减小暴力检索的范围。之后，将得到的并不精确的椭圆位置在原图上（分辨率 640×480）做一个微小的调整，最终得到瞳孔位置，这个时候效果是相当好的。后来我又做了改进，将在压缩的图上（分辨率 108×81）的检索范围继续减小。具体做法是做一个二值化，如下图。

图 2

这个时候我们发现，有一个比较接近圆的轮廓，就是我们要找的椭圆，我们要想办法得到这个椭圆的大概位置，具体做法是，利用 opencv 的找轮廓函数，这时候会得到许多个轮廓，那么哪个是我们要找的呢？很简单，我用 opencv 的找最小外接圆函数，得到每个轮廓的外接圆。如果是我们要找的椭圆，那么这个椭圆轮廓与这个外接圆应该比较接近，通过计算轮廓上每个点$[(x - x_0)^2 + (y - y_0)^2 - r^2] / r^2$应该会比较小（注：点(x, y)为轮廓任一点坐标，点(x_0, y_0)为外接圆圆心，r 为外接圆半径）。所以通过找该值的平均值的最小值，就可以找到我们想要找的椭圆。之后，在这个外接圆基础上稍微扩大一点，就得到了我们的下一步要进行检索的范围，这样，就又进一步减少了我们的检索范围了。

3. 减少最内层循环的分支语句的使用，比如 if-else 语句。这个程序一共有 6 重循环，椭圆 5 个参数(长轴 a, 短轴 b, 中心点 x、 y, 偏转角 α)，再加上对每 5 个椭圆参数要进行椭圆圆周上的各点差分计算。我们看到图 1 那里人眼里有一些比较亮的光斑，这些光斑会极大地影响我们后面的检测。光斑点的灰度值在 200 以上，它那里的差分会非常大。为了消除光斑对我们后面的影响，我们一开始采用的办法是用一个 if 语句来判断是否是光斑，是就不算它的差分计算。这个 if 语句在最内层循环，除此以外，还要判断椭圆圆周上的点是否在图像内，否则容易溢出。等等，这样在最内层就有了好几个 if 判断，这对减少程序运行时间是十分不利的。毕竟 if 语句非常耗时间。我的做法是把这些判断全部扔到外面去，不要放在最内层循环。具体做法是，比如光斑的影响，那我就在一开始进行图像处理的时候就把光斑抹掉，怎么抹掉呢？就是通过光斑周围的点的像素值，来“预测”原来的光斑位置的像素值。抹掉光斑之后，后面就再也不需要判

断是否是光斑了。对于要判断点是否在图像的矩阵内，只需要在这之前保证这个椭圆圆周所有点都在图像矩阵内就好了。总之就是，把最内层循环的分支语句尽量抛到外面去。

4. 减少乘法的运算。当时遇到的情况是这样的：椭圆不同的 5 个参数会有不同的形状或位置，就有不同的圆周上的点，做暴力检索遍历的时候是这 5 个循环体随便放置的吗？显然不是，但我们一开始就是这样的，对每 5 个不同的参数，计算这 5 个参数确定的椭圆圆周上的点，再计算差分值。其实不需要，怎么说呢？实际上，椭圆只需要 3 个参数(a, b, α)就可以确定它的形状，再通过中心点(x, y)确定位置，所以，当先将 a, b, α 的循环体放在前面，这样对每个(a, b, α)，我们先计算出中心点为(0, 0)的各个点坐标，存在数组中，后面在对 x, y 进行遍历的时候，只需要将这些点进行 x、y 的偏移，即可得到新的点的坐标。这里，只需要加法。但是由(a, b, α)来计算椭圆圆周上的点坐标时，是需要乘法的，而且还不少。这样，我们就减少了计算量，时间再次减少。

5. 减少对数组的访问和计算。在对图像像素点进行累加时，我原来大概是这样的：

```
valueAccum += img.data[i];
```

后来大概改成：

```
tempImgElement = img.data[i];
valueAccum += tempImgElement;
```

这样子时间又减少不少，具体怎么做的我记不清了，大概意思就是这样吧。

6. 公式拆分。前面第 4 点我说过，对每个(a, b, α)，我们先计算出中心点为(0, 0)的各个点坐标，存在数组中，后面在对 x, y 进行遍历的时候，只需要将这些点进行 x、y 的偏移，即可得到新的点的坐标。设中心点为(0, 0)的各个点坐标为(x1, y1), (x2, y2)… 对(x1, y1)点，它在某个中心点为(x, y)的情况下的新坐标为(x1 + x, y1 + y)，它在图像中的像素点为 img.data[(y1 + y) * imgWidth + (x1 + x)]。但是其实这也有不少冗余计算，怎么说呢？我们把公式拆分来看看，(y1 + y) * imgWidth + (x1 + x) = (y1 * imgWidth + x1) + (y * imgWidth + x)，这里拆分为两部分，我们看到，前面我们保存在数组中的是(x1, y1)，但是实际上，我们只需要保存(y1 * imgWidth + x1)这一个值就够了。接下来当对 x, y 做遍历时，只需要计算一次(y * imgWidth + x)，后面对圆周上的各个点进行遍历时，只需要将这两部分加起来就 OK 了，不仅减少了计算量，还减少了前面数组的空间，减了一半，原来是(x1, y1)两个数，现在只有(y1 * imgWidth + x1)这一个数了。

7. 循环展开。这个方法是我从《深入理解计算机系统》这本书中看到的，举个例子，这里是要对数组进行循环累加：

下面是普通方法：

```
for (i = 0; i < limit; i ++)
{
    valueAccum += a[i];
}
```

下面是循环展开 2 次：

```
int length = (limit / 2) * 2;
for (i = 0; i < length; i += 2)
{
    valueAccum += a[i] + a[i + 1];
}
for (; i < limit; i ++)
{
    valueAccum += a[i];
}
```

在这一次我使用过程中，用了循环展开 5 次，速度大约提升 10%-20%这样吧，具体没太仔细对比，反正是有一些优化的。

8. 尽量避免在最内层循环，也就是调用次数最多的代码处进行函数调用。如果我没记错的话，当时改进了下算法，之后在最内层循环处去掉了取绝对值函数 abs()，速度比之前又提升了十几倍。其实意思就是，在最内层循环，或者说在执行次数最多的代码处尽量简单，避免使用分支语句或者函数调用等。

9. 跳过一些特征相差太大的情况。对一些特征明显不是我们要找的情况的，直接跳过，即使是暴力搜索。这样不仅仅是减少了运行时间，还减少了这种特征明显相差太远的情况的干扰。比如说，在对瞳孔中心位置进行暴力搜索时，由于瞳孔中心灰度值本身就不高，所以对一些中心点灰度值过高的椭圆，可以认为这不可能是我们的目标，可直接跳过。

10. 注意灰度图的二值化问题，尤其是采用固定值阈值的二值化，很容易会崩。什么意思呢？你想，当处于不同的光照环境下的时候，图像整体的灰度值是不一样的。比方说你设定阈值为 100，但是换了个环境，这个环境比较暗，整个图像灰度值几乎都在 100 以下，那么这个阈值 100 也就没什么大用了。我换了次样本图像，果然就出现了这个问题。

时间过去好久了，一下子也想不起来这中间的一些优化方案，或者注意事项等，而且有些方法对性能提升不大，也没太注意，就大概先写到这，等什么时候想起来其它的，再往里面添加进去。对于代码时间/速度上的优化，有一句话很重要，就是：减少冗余计算。有时候一个你并不在意的方案，它的速度可能会提升好几倍。也遇到过我以为肯定会提升速度的方案结果改出来一点效果都没有，甚至还会增加运行时间。中间有一次我要改成查表，用到了 3 维数组，结果发现运行时间增加了，当时没去深究，我想，可能是对 3 维数组的访问速度比较慢，可能会超过计算一次的运行速度，这也不是完全不可能，有时候数组过大，CPU 需要不断跳跃读取内存的数据，导致缓存命中不高，而且 CPU 从内存读取数据速度比较慢，这才导致程序的性能瓶颈在数据读取上，而不是 CPU 的计算上。说起这个突然想起来，在最内层循环，我曾经去掉一条语句”count ++;”，我本以为速度会提升不少，毕竟本身最内层循环的代码就没多少行，结果去了之后发现，速度并没有提升，几乎没有变化，当时令我有些吃惊。后来我想，有可能是这样：时间有可能是消耗在对数组的读取上，这么大的一个数组，操作系统可能只把部

分放在高速缓存中，又由于高速缓存命中不高，才导致时间耗在对数据的读取上。而”count ++;”这个语句很可能是一开始就被系统放在寄存器中，所以消耗的时间很少。

上面有很多都只是我的猜测，并没有去深究它，这也折射出我能力上的欠缺，以及内心的浮躁，有时候太急于看到成果，没能对更多事情去做深究。不仅如此，内心的浮躁还体现在码代码上，有时候太急于看到程序的效果，没有一步一步把代码写好，容易导致后面需要更多时间去做调试。对代码的注释上，也是没能做好，有时候一股脑写起代码来就忘了加注释了，如果不是经常看，时间长一些就容易忘。还有代码安全性，在代码安全性方面的考虑更多的是依靠以前的知识，其实还有很多需要考虑的东西，我现在根本就没考虑过，还需要花些时间去增加这方面的知识。安全性不高，很容易会出现程序崩溃的情况，真正要应用到实际的程序，安全性问题必须要重视。

在项目的管理上我是一个失败者，这个是毫无疑问的。我身为组长，没能多去调动队友的积极性，也没有经常去了解其他人的进度，有时候只顾自己埋头苦干。不过，经验总是靠积累起来的，自己还是要多去积累这方面的经验。这次虽然表面上说是组长，但其实说真的，更多不过是个虚名而已，自己并没有担当起更多的责任。

好了，大概先写到这吧，等以后想到什么，或者等后面自己再遇到什么东西，再往里面添加吧。马上要过年了，祝所有人新年快乐，新的一年平平安安、健健康康，继续努力吧！

罗家意

2017 年 1 月 27 日，大年三十