简易能量机关识别

默认你有opencv和eigen。

在build目录下执行

```
cmake ..
make
```

即可。

任务描述

风车绕一中心点以某一服从三角函数分布的角速度旋转，中心点位置可能是随机出现在屏幕上的。风车上有一个扇叶还没有被击中，需要进行匹配与预测，以较低误差预测风车扇叶中心的运动轨迹，并进行跟踪。

特别声明：这只是对风车叶片预测的简单模拟，仅供学习参考使用，尚未完善到解决实际问题的能力。

思路

该问题可以分为两个子问题。第一个就是对待打击叶片进行识别，识别后标记这个叶片的中心点；第二个就是追踪预测这个中心点的位置。

第一个问题的解决方案就是利用opencv自带的一些方法，进行常见的图像操作，以最小矩形框找到待打击叶片。

第二个问题的解决方案就是根据风车运动的已知的角速度形式对风车运动解一个非线性优化问题，拟合待打击叶片中心点的位置。

程序设计逻辑

我编写了一个简易的用于识别风车待击打叶片的类：**WindMillDetection**。

在使用之前先声明一个对象，依次传入参数**start、omega、a、b、fai**。分别代表初始时间、角速度公式中的**w、a、b和φ**。

然后在循环中使用函数**SetFrame**：

```
void SetFrame(const cv::Mat &frame);
```

用于传入当前需要分析的视频帧。

然后调用**Proccess**函数：

```
void Process(double nowtime, int method);
```

该方法传入当前时刻**nowtime**，nowtime减去starttime就是待估计函数中的自变量t；**method**是用于选择模型估计方法的函数，当前只实现了基于**Huber损失函数**的梯度下降法。

该函数中依次实现了三个功能：维护不断采样过程中添加的样本数据、调用私有方法**FindHitFan**对图像进行叶片识别、调用模型估计方法如**ML**进行模型估计。

接着调用**GetImage**函数：

```
cv::Mat GetImage(int which);
```

该函数输入一个选择，返回进行处理时的中间图像或结果图像。

最后调用**GetShowMat**函数：

```
cv::Mat GetShowMat(double now);
```

该函数返回当前时刻的处理后的结果图像。

任务一：识别待打击叶片

寻找风车的旋转圆心

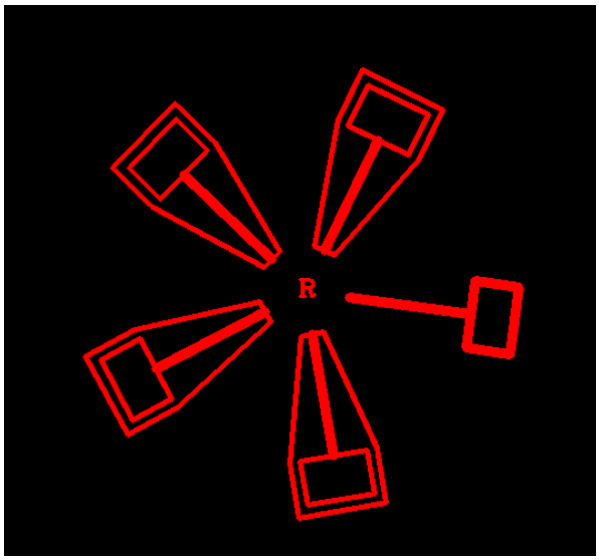
风车不是固定不动的，所以需要先找到风车旋转的圆心，以圆心为参考系再进行后续的估计问题。

风车的圆心是比较好找的，如果只需要找到风车的圆心，只需要进行如下操作即可：

因为图像背景是黑的，风车上有灯条且是红色。所以可以先将图像转化为灰度图像**cvtColor**，再对图像进行一次二值化**threshold**，之后调用函数找出图像轮廓中的最小轮廓矩形框**findContours和minAreaRect**，通过面积筛选出面积最小的矩形求其中心点坐标即可（因为圆心处的*R*被矩形框出来时是面积最小的）。以上的操作都只要调用opencv内置函数即可。

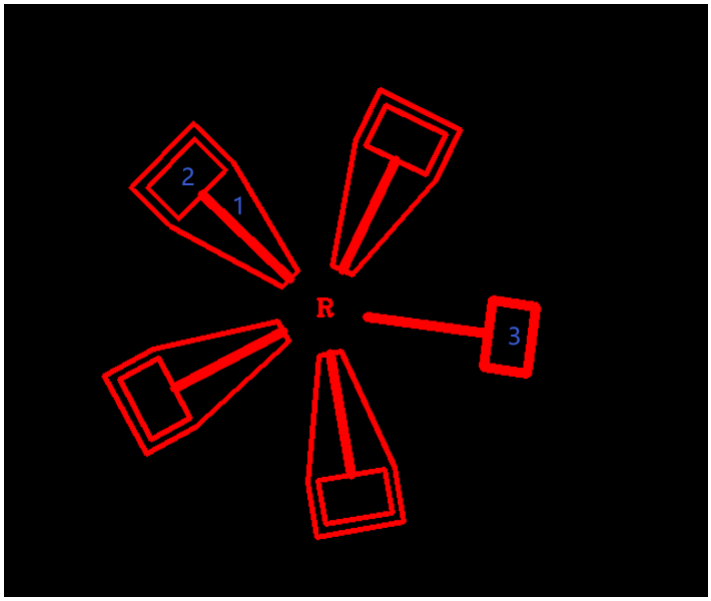
寻找待击打叶片中心点

要寻找待击打叶片，就要先区分出待击打叶片和不需要击打的叶片。



下图中待击打叶片周围没有额外的灯条包围，但是如果直接调用**findContours和minAreaRect**找到的矩形框并不能很好的区分出待击打叶片和不需要匹配的叶片。

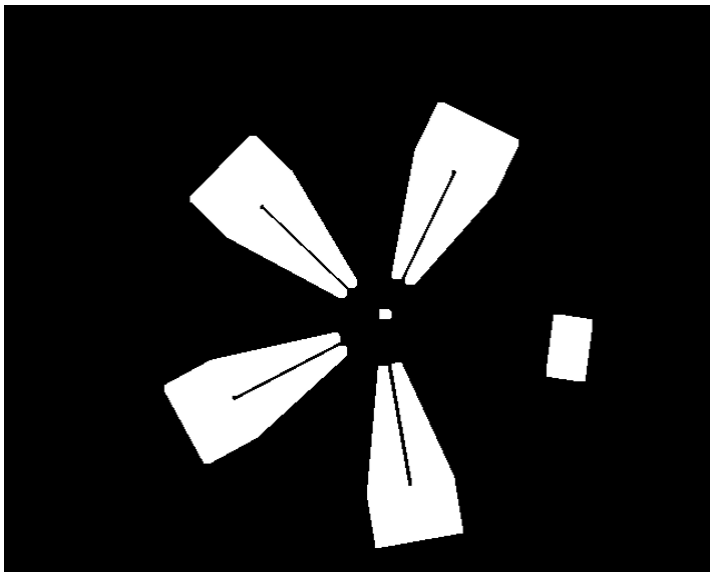
观察到图像上有如图所示的三个区域（蓝字标出）：



这三个区域都各自独立，且不与外界的背景色相连接，所以可以想到的思路就是将背景颜色也变为红色，与风车颜色相融合，所以此时这三个区域就能被孤立出来。所以可以调用opencv的**floodFill**函数，在最左上角那个点处将整个背景图染成红色。

在填充颜色操作后的图中，**3**区域被孤立出来了，这就很方便我们找到它。但是又出现了一个问题，就是**2**区域和**3**区域是不相连接的，且**2**区域和**3**区域的形状完全相同，所以当我们调用函数框出他们时，这两种区域都会被框出，导致我们将无法区分**2**和**3**。

但是又可以发现，虽然**2**区域和**3**区域互相隔离，但是二者之间的却只隔了一条缝隙，所以这个时候我们可以对整幅图像进行一次形态学的**膨胀**操作，设置卷积核，调用**morphologyEx**函数，选择**cv::MORPH_DILATE**膨胀，膨胀可以将二值化的白色部分扩张，从而连接缝隙。如下所示：



这时我们再调用**findContours和minAreaRect**，再通过筛选面积或长宽比等策略，就能找到待打击叶片了，然后通过矩形框的中心就能找到待打击点。

任务二：追踪叶片待打击点

优化问题

已知风车的角速度满足方程$v = 0.785 * sin(1.884t + \varphi) + 1.305$的形式，则可以通过积分求出某一时刻的角度应该满足$\theta_t = \theta_0 + \int_0^t vdt$的形式。所以要追踪并预测待打击点的运动和运动位置，就需要拟合θ_0和φ这两个未知量。这里可以采用简单的梯度下降算法进行求解。

值得注意的是这个地方θ_0也是需要进行不断估计的。可能会有人认为在拟合出比较好的角速度后只需要利用$\int_0^t vdt$计算出来的$\hat{\theta}$和这一时刻的测量值$\theta_{observation}$做一个差就能得到θ_0，就能一劳永逸。但是这是错误的，因为θ是个积累量，在积累过程中可能会有噪声，噪声不断积累可能导致和实际有误差。

第一种尝试

基于观测角的拟合

在第一次尝试时我用积分公式将某一时刻θ_t的表达式写出：$\theta_t = -\frac{a}{w}cos(wt + \varphi) + bt + \theta_0$，打算用旋转角的位移公式对观测到的角度进行拟合，来求出这两个未知量。

python模拟

在进行第一次尝试时，我准备先用python写一遍优化问题，因为python更简单方便，如果实现成功了我只需要将它换成相应的c++代码即可。于是我根据公式
$\theta_{observation} = -\frac{a}{w}cos(wt + \varphi) + bt + \theta_0 + noise$写了一个模拟观察到的角度的函数,其中**noise**是设定的一个高斯白噪声，这里的φ和θ_0都是自己设定的。

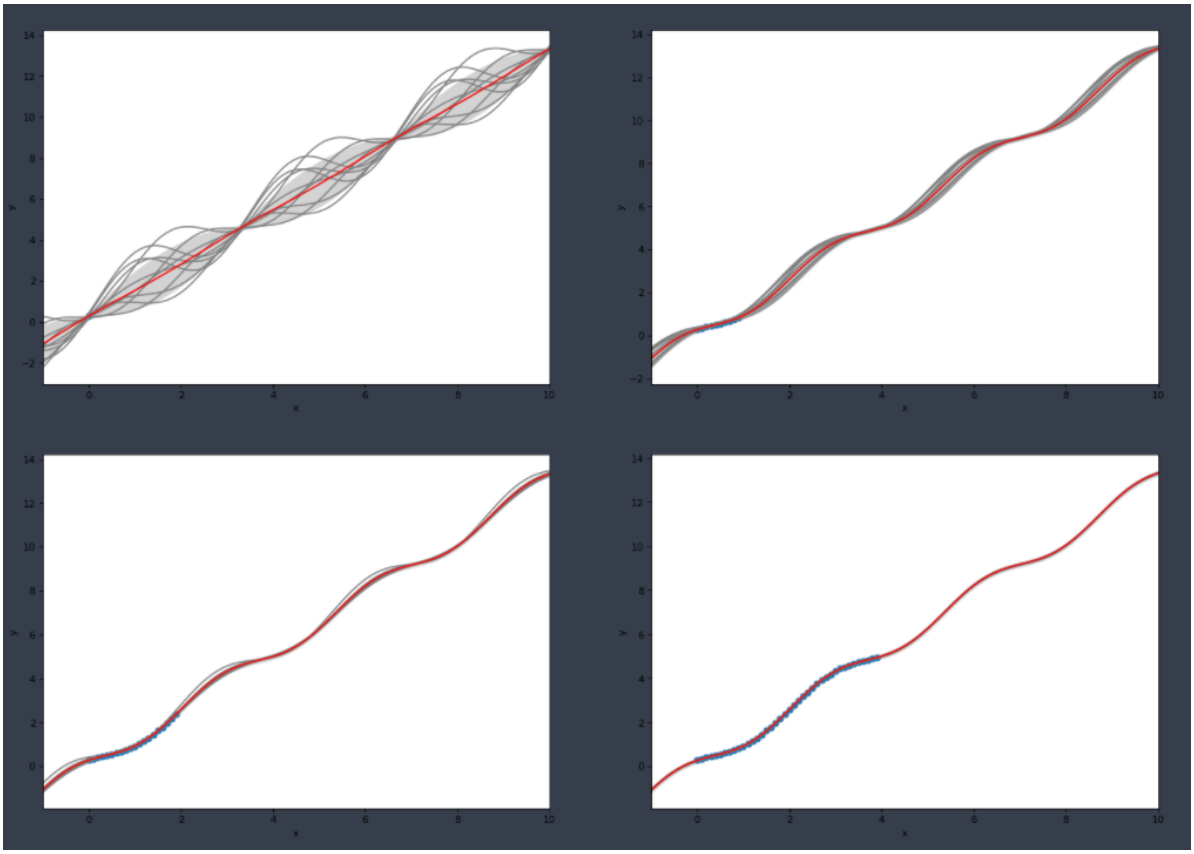
然后利用**平方和损失函数**$L = \frac{1}{n}\sum_t(\theta_t - \theta_{observation})^2$列出对φ和$\theta_0$的偏导数：
$\frac{\partial L}{\partial \varphi} = \frac{2}{n}\sum_t(\theta_t - \theta_{observation}) * \frac{\mathrm{d}\theta_t}{\mathrm{d}\varphi}$和$\frac{\partial L}{\partial \theta_0} = \frac{2}{n}\sum_t(\theta_t - \theta_{observation})$。然后通过梯度下降进行迭代。

因为样本角度是随着时间进行一个个生成的，所以每一次优化的时候所持有的样本数量是不同的，所以这里我还只是先用了一个粗糙的策略：每次优化对所有样本点同时进行拟合，即**batch**。

使用这种简单粗糙的方法，在该模拟中可以很好的对参数进行拟合。

此后我还另外写了一个采用贝叶斯线性估计的方法，也就是将$\theta_t = -\frac{a}{w}cos(wt + \varphi) + bt + \theta_0$拆开为$-\frac{a}{w}cos(\varphi)cos(wt) + \frac{a}{w}sin(\varphi)sin(wt) + bt + \theta_0$，将$-\frac{a}{w}cos(wt)$、$\frac{a}{w}sin(wt)$、$t$、$1$视为四个基函数，对它们各自的系数进行拟合（其实可以预先减去bt这个函数，因为b是已知数）。拟合效果如下，随着样本点逐渐加入，参数值逐渐收敛。

灰色的线是根据这几个基函数的系数各自服从的概率分布随机取的一个值画出来的，红线是根据这几个系数服从的概率分布的均值画出来的。这就是采用了贝叶斯学派的观点。



该方案的问题

我在python上实现了模拟之后，打算将其复现到c++上面去，才发现了这个方法有一些严重的问题。这些问题并不是python代码和c++代码本身的不兼容，而是因为在模拟过程中我没有考虑到这些问题，导致出错。

- 第一个问题：在拟合问题中我们通过一个函数生成一个$\theta_{observation}$，这个值它是一个任意角，所以我们在拟合过程中把它当做了一个连续的递增函数来进行拟合，对于一个连续的函数来说，是很好拟合的，但是在实际问题中，我们测量到的角度观测值$\theta_{observation}$是无法获知它的任意角的！相反，我们只知道它的**[0,2π)**范围内的角度。所以我们不能用连续函数
  $\theta_t = -\frac{a}{w}cos(wt+\varphi)+bt+\theta_0$来对参数进行拟合。
- 第二个问题：在连续函数$\theta_t = -\frac{a}{w}cos(wt+\varphi)+bt+\theta_0$中，存在一个**bt**的直流量，如果这个角度是一个任意角，那么这个值就会随程序运行时间不断增加，直到维护这个值的变量数据溢出，会造成不该有的错误。

因此，基于角度测量值进行拟合的方法在实际应用中是不合理的。

第二种尝试

基于角速度的拟合

第二种拟合方案就是采用基于角速度的拟合方式，这个时候我们就能避开任意角。因为我们已经知道了角速度的函数形式，因此只要我们能够通过测量获取风车转动的角速度，那么就可以对角速度列损失函数方程：$L = \frac{1}{n}\sum_t(v_t - v_{observation})^2$。对于$\theta_0$的估计无须拟合，后面会讲另一种估计方法。

对角速度的采样

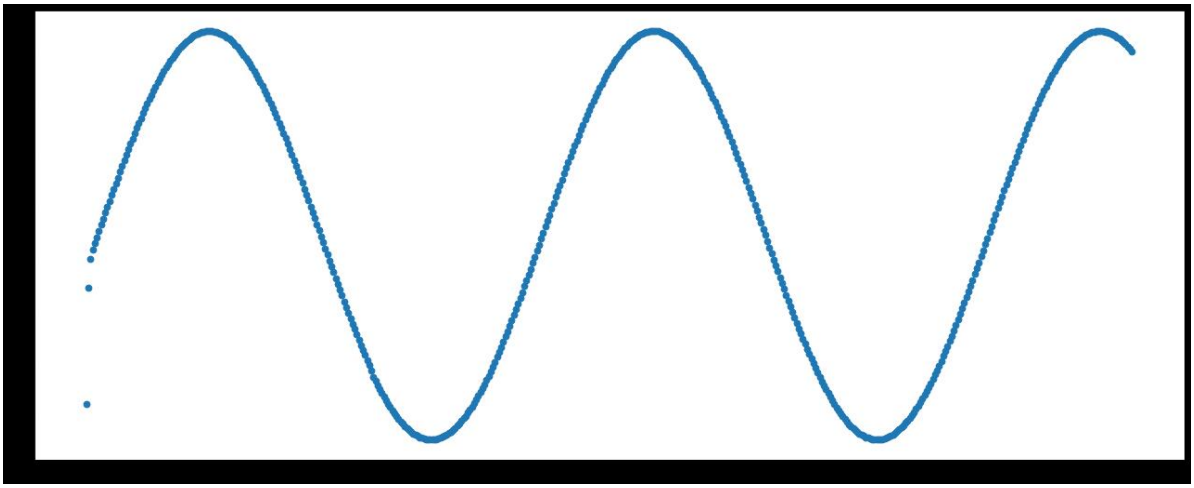
对角速度的采样即计算出每一帧待击打点的角度，通过两帧的测量值做差再除以时间间隔就能近似估计这一时刻的角速度。

但是进行角速度测量的时候可能会由于时间间隔的长短导致产生不同的误差。因为在实际测量中，风车转动肯定是会受到噪声的，假设是一个高斯白噪声加在角度位移上，那么当我们取一个很短的时间间隔来求该时刻角速度时，我们本意是采用小时间间隔以逼近导数，但是因为有一个噪声常数的加入，我们以一个很小的时间步去除一个没那么小的噪声偏移量时，就会得到一个很大的角速度误差，这个角速度甚至会大的离谱，已经不符合$\theta_t = -\frac{a}{w}cos(wt+\varphi)+bt+\theta_0$的导数的分布了，这种样本就叫做**outlier**，即**离群点**；加入我们采用的是一个叫长的时间间隔，虽然此时噪声偏移量的导致的误差会没有那么显著，但是此时的差商以及不再那么逼近导数值了，也是没有那么准确的。

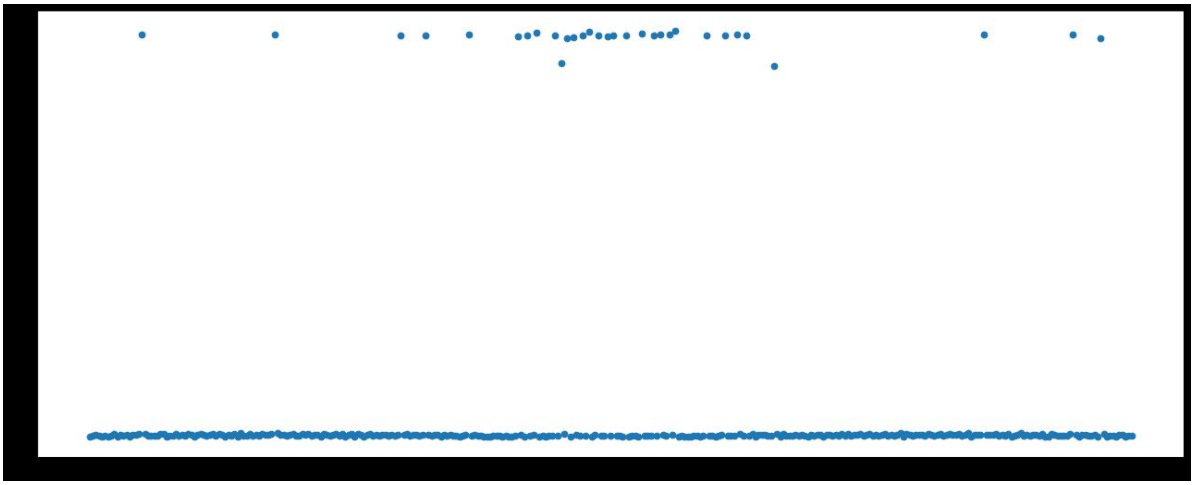
所以对采样时间间隔长短的选取也是有讲究的。

下面三张图是我在用python模拟的加入噪声和不加入噪声时，对时间间隔长短的选取所测量到的角速度的分布。

第一张图是我没有加入高斯噪声时，采用很小的时间间隔，对角速度的测量采样：（忽略前面那两个点，那两个点因为程序设计的一点问题，是不准确的）

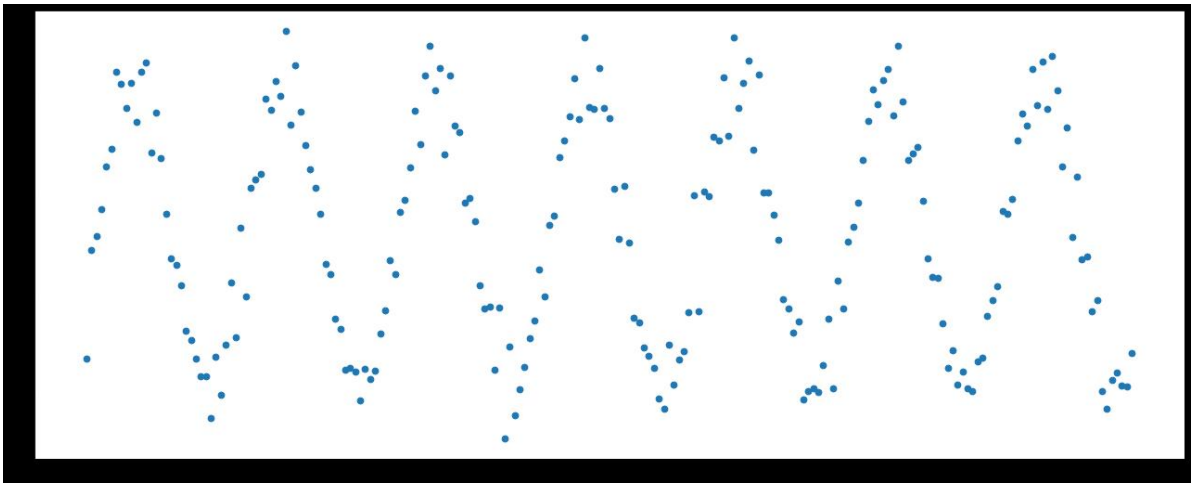


第二张图是我在采用很短的时间间隔，但是加入了高斯噪声时，对角速度的测量采样：



可见第二张图上方有很多离群点，而正常测量的数据因为坐标轴绘制尺度的原因被挤压成了一条直线。

第三张图是我依旧加入高斯噪声，但是采用了比较长的时间间隔对角速度的测量采样：



可见此时至少是基本有三角函数的分布的形态的。

测量角速度时离群点的来源：一方面是比较大的高斯噪声偏移量除以很小的时间间隔，另一方面是错误的角速度计算（后面讲到）。在实际应用中肯定还会有很多其他的噪声。

损失函数的选取

上面提到了当取短时间间隔时会有很多离群点，这些点如果直接放在平方和损失函数中进行拟合时会造成较大误差，甚至无法拟合成功。此时就可以采用**Huber损失函数**：

$$L=\begin{cases}\dfrac{1}{2}(v-v_{observation})^2 & |v-v_{observation}|\le\delta\\ \delta|v-v_{observation}|-\dfrac{1}{2}\delta^2 & |v-v_{observation}|>\delta\end{cases}$$

采用这个函数进行梯度下降时就能比较好的处理离群点。

样本处理策略

随着程序的运行，我们会收集到大量的样本数据点。在第一次尝试中，我使用收集到的所有样本做参数估计，但是随着时间进行，收集到的样本越来越多，一方面样本越来越多时所需的内存空间也就越来越大，直到我们无法再维护这么多的样本，程序就会崩溃；另一方面，样本越多，对所有样本进行模型估计的时间会越来越久。所以每次训练时对自程序运行开始所收集到的所有样本进行训练时不合理的。

于是我采用的策略是设置一个最大样本容量的阈值，当收集到的样本没有达到阈值时，每次收集到一个样本时将其维护起来，然后对现有的所有样本进行训练，模型拟合。当收集到的样本达到阈值时，下一次收集到一个新样本后，就会以队列的形式将当前维护的样本中最早加入的那个剔除，保持样本总量维持在一个固定的量。

同时还需要设置一个下限阈值，这个下限指的是开始训练所需要的最少样本量。因为样本很少时，模型拟合起来效果可能不好，如果需要训练的模型的损失函数是一个非凸函数，在样本很少的时候就开始训练可能会造成模型收敛到局部极小值，所以我们希望当收集到的样本达到一定的数量时才开始训练。

角度的参考系以及角度处理中出现的一些问题

我对角度的划分

在这里，我对角度的划分是：假设**x轴**水平向右，**y轴**竖直向上。如果一个点和原点的连线顺时针旋转θ后能使点落在x轴上，那么这个点的角就是θ。也就是说逆时针旋转为正，顺时针为负。同时角度的变化范围为[0,2π)。

角度处理中出现的一些问题

在计算角速度时，我们通过前后两帧做差商求得角速度。如果我们使用的是任意角，那么这种求法只会引入位移与差商近似导数产生的误差。但是如果我们考虑角度在**[0,2π)**时，就必须得处理好前后角度关系，否则就会出现不必要的出错误。比如前一帧角度在2π附近，小于2π，后一帧角度在0附近，大于0，那么如果二者直接进行差商，就会求出一个大得离谱的角速度错误值。

所以这个时候为了正确计算出某一时刻的角速度，我们可以认为在某一个短时间间隔之间，角度的旋转不会有很大的改变，所以当我们通过对两帧的角度进行差分的时候如果差分的绝对值过大，这个时候就需要进行特殊处理。

我的特殊处理方式是：认为相邻两帧样本的角度变化绝对值不会超过180^o。假如这个绝对值超过了180^o，那我就通过如下公式进行校正：

$$\triangle\theta = -(2\pi - |\theta_t - \theta_{t-1}|) * (\theta_t - \theta_{t-1})/|\theta_t - \theta_{t-1}|$$

$$v_t = \frac{\triangle\theta}{\triangle t}$$

参数估计

角速度初相位φ值估计

虽然我们知道了风车角速度的函数形式，但是每次我们开始观测的时间不同，那我们观测到的这个初相φ也是不同的。也就是说程序开始的时间不同，风车转动的角速度的φ值不同。原因是:

$$v = 0.785 * sin(1.884(t + t_0) + \varphi) + 1.305 = 0.785 * sin(1.884t + \varphi + 1.884t_0) + 1.305$$

我们要拟合的φ其实是$\varphi + wt_0$。

于是在我的**process**方法中的最后调用了私有方法:

```
ML(const Eigen::ArrayXXd & time_, const Eigen::ArrayXXd & measured_theta_ ,
const Eigen::ArrayXXd & v_ )
```

该函数传入维护的时间数组*time_*、测量到的角度数组*measured_theta_*和测量到的速度数组*v_*。

在该函数中，设定了一个启动条件，也就是判断传入的这些数组的长度（理论上三者长度都是一样的）是否大于上面提到过的所需的最小样本量，如果少于这个样本量就直接返回，不予训练。

然后在训练部分，最外层是一个*for*循环，我设置的是迭代**10次**。内层是一个用于遍历数组每一个元素来求Huber损失函数梯度的*for*循环（因为我不知道Eigen是否有类似numpy的布尔索引和一些其他的逐元素操作，所以这里还是写了for循环，虽然自从用了python后我就很讨厌用循环）。

在循环中利用**Huber损失函数**对每个样本求相应的对φ的梯度：

$$\frac{\partial L}{\partial \varphi}_i = \begin{cases} (v - v_{observation}) * \dfrac{dv}{d\varphi} & |v - v_{observation}| \le \delta \\ \delta * \dfrac{dv}{d\varphi} & v - v_{observation} > \delta \\ -\delta * \dfrac{dv}{d\varphi} & v - v_{observation} < -\delta \end{cases}$$

然后对所有$\frac{\partial L}{\partial \varphi}_i$求和再取平均数,用于梯度下降：$\varphi = \varphi - lr * \overline{\frac{\partial L}{\partial \varphi}}$。**lr**就是学习率。

如果能选择合适的超参数**δ**和**lr**，并且角速度的采样数据基本正常的话，那么就能比较快的正确拟合出**φ**的估计值$\hat{\varphi}$。此时我们的预测点的速度变化就能和待打击点基本一致，二者之间的角度差就基本固定变化不大。

进行完φ的拟合后（指的是收敛后，因为拟合过程在不断进行），一般来说两点之间还是会有一个角度差，毕竟我们还没有拟合θ_0。而二者之间的角度差还是可能存在一定的微小变化，这是因为一方面角度变化方面带有一定的高斯噪声，另一方面因为拟合出来的$\hat{\varphi}$并非真实值，所以可能导致预测点的角度和待击打点的真实角度之间一直会存在一个服从三角函数分布的误差，但其实这个误差应该说是几乎可以忽略的，因为只要成功使$\hat{\varphi}$收敛到真实值附近，那么这个误差将会是很小的。

初始角度θ_0的估计

在估计完φ以后，预测点和待击打点之间可能还会有一个基本固定的角度，这个角度就是我们要估计的θ_0。

现在我们得到了一个拟合的函数$\check{\theta}_t = -\frac{a}{w}cos(wt+\hat{\varphi})+bt$。一个简单的想法就是直接用我们计算出来的$\hat{\varphi}$代入函数，然后把时间也代入，再和对应时间的$\theta_{observation}$做差就可以了。但是值得注意的是，上面我提到过$\check{\theta}_t = -\frac{a}{w}cos(wt+\hat{\varphi})+bt$计算出来的只是一个**任意角**。所以得先将其化为**[0,2π)**之间（这里都是弧度制）：

$$\check{\theta}_t = (\frac{\check{\theta}_t}{2\pi} - \lfloor\frac{\check{\theta}_t}{2\pi}\rfloor) * 2\pi$$

这样转换完之后我们还不能直接拿它和$\theta_{observation}$做差，因为但凡涉及到**[0,2π)**之间的角度差值，就得考虑0和2π的边界问题，这里都得特殊处理，正如我们上面求角速度时一样。

既然知道θ_0的幅度变化在**[0,2π)**之间，所以我将其设为在**(-π,π]**之间取值。为正数时表示快一个角度相位，为负值时表示慢一个角度相位。然后再求$\triangle\theta_t = \theta_{observation} - \check{\theta}_t$,再判断$\triangle\theta_t$是否在**(-π,π]**内，如果是大于π，那就可以自减2π；如果小于-π，就可以自加2π。然后就能得到正确的$\triangle\theta_t$，再更新$\hat{\theta}_0^t = \check{\theta}_t + \triangle\theta_t$。得到此时刻对$\theta_0$的估计值。

理论上我们可以通过上述方法计算出每一个时刻对θ_0的估计值，那我们是不是要对它们全部做一个平均？其实没有必要。我采用的策略是使用**指数平滑**的方法，仅对采用最新时刻的$\theta_{observation}$来更新$\hat{\theta}_0$。即采用：

$$\hat{\theta}_0 = a\hat{\theta}_0 + (1-a)\hat{\theta}_0^t$$

这里的**a**是一个权重。这样就能减轻对$\hat{\theta}_0$估计的运算负担。

以下是我对待打击点的预测情况：（蓝色的是预测点，预测点覆盖了待打击点）