Short story of how I ended up using the α, β & γ filter. I tried using the α, β & γ filter early on but made an error, found some decent values to get me to around 55% hit rate. Decided I should try the real Kalman filter and figured out soon enough that documenting it would be a lot of work. Heard a rumor that the α, β & γ could go above 77% Anders had already achieved 83% average. Got his values and fiddled with them until I reached somewhere around 89%

As for my choices, I've looked at https://www.kalmanfilter.net/alphabeta.html for the formulas. From the formulas I implemented and tested the examples on the website before taking on the task of coding the filter for this task.

How the filter works.

I can split it up in 2 stages,
calculate new estimates and calculate new predictions to help the next estimate.

```
17  def estimate_current_position_and_velocity(self, zn):
18
19      # Estimates
20      self.e_p = self.pred_p + self.alpha * (zn - self.pred_p)
21      self.e_v = self.pred_v + self.beta * ((zn - self.pred_p)/self.t_d)
22      self.e_a = self.pred_a + self.gamma * ((zn - self.pred_p)/((self.t_d**2)*0.5))
23
24      # Predictions
25      self.pred_p = self.e_p + self.e_v * self.t_d + self.e_a * ((self.t_d**2)*0.5)
26      self.pred_v = self.e_v + self.e_a * self.t_d
27      self.pred_a = self.e_a
28
29      return self.e_p, self.e_v
30
```

On line 20 we have the standard formula we are going to follow to get new estimates.

In pseudo code it would look like this:
New estimate = old estimate + a small change based on a new measurement.

Alpha, beta and gamma are how much change we allow the new measurement to make.

As for lines 21 and 22, it's basically the same but since we do not get any measurements of velocity and acceleration then we must calculate them.
Velocity: distance / time.
Acceleration: distance / [(delta time^2) / 2]

Once the estimates are calculated, then we can start calculating a prediction of where the target will move next. These predictions will contribute to the next estimates.

The predictions are made in pseudo code like this:

Prediction of position = estimated position + estimated velocity * delta time + estimated acceleration * (delta time^2 * 0.5)

Prediction of velocity = estimated velocity + estimated acceleration * delta time

Prediction of acceleration = estimated acceleration

We need these predictions for the next estimated values. The reason why we only say “new prediction = old estimate” for acceleration is because there is no change in jerk. That would be how we could see other mathematical equations on these examples. In the app we only work with position, velocity and acceleration changes so it would not make sense to introduce equations with jerk as a value with change.

As for the alpha, beta and gamma values I have landed on, it’s mostly down to trial and error. I landed on using the following:

Alpha = 2.0e-2

Beta = 8.0e-5

Gamma = 3.0e-7

I did try these values:

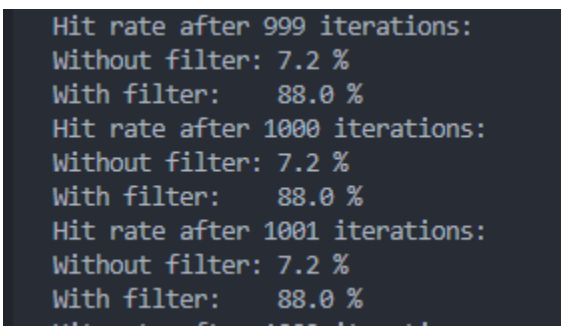
Alpha = 2.0e-2

Beta = 8.13e-5

Gamma = 2.95e-7

But found it affecting the result by less than 0.5% so I went for rounded values so it would be easier to understand.

The code I’ve tested the filter with it 2 methods, let it run on the file we are not allowed to change and let it run through around 1000 iterations and get the follow result:



And running it through the same file but without most of the pygame code, this runs a lot faster. As shown under after 100 000 iterations the result was around 89.3%

```
Hit rate after 100000 iterations:
Without filter: 6.6 %
With filter:    89.3 %
```

The reason we see a difference of 1.3% is because 1000 iterations is too small of a sample size to not be affected by it being “a good or bad run” doing the test again would show a different percentage.

The file we use to test different values can run through about 40 000 iterations in 3 minutes, it was edited by Markus Hartl and small changes have been added to include 2 decimal places of accuracy.

And finally initial values. I found better success from having zero values as initial values versus finding average or middle values from HitTheTarget.py. The following numbers are the middle values I’ve found.

Position =	108.0
Velocity =	0.7
Acceleration =	0.0015

With the values above as the initial predictions then the filter gets to around 72-73% hit rate. However, if I add an initial value for acceleration only, then I achieve a higher hit rate. Fiddling around with this value for a few hours I got an improvement of 9.1%...

In the end I ended up using these values and got the following result:

Position =	0.0
Velocity =	0.0
Acceleration =	0.01

```
Hit rate after 50000 iterations:
Without filter: 5.71 %
With filter:    94.47 %
```

And running it through the unchanged file I got this result:

```
Hit rate after 10445 iterations:
Without filter: 5.6 %
With filter:    94.0 %
```