UWB: Positioning Improvement: Fitting a line to three-dimensional data

Eigentümer: Hannes Reindl
Letzte Aktualisierung: gestern um 8:12 AM • Angesehen von 2 Personen • Signatures • Up-to-date

This is a continuation to 3D of this task: Focus Team UWB: KW25, Distance measurements evaluation | Linear regression

The goal is to fit a line to three-dimensional data since temporal information will also be used to estimate the position from the UWB system. Unfortunately, the Linear Regression code previously used to fit a line to two-dimensional data did not work out of the box and some adjustments were required. I found several resources using PCA instead of Linear Regression. While it is possible to extend Linear Regression to 3D, the PCA variant looks way easier to implement since the Python module `numpy` already has a built-in command taking care of everything. Furthermore, I found some code that executes PCA and creates a line using the first component, thus I decided to abandon Linear Regression and use PCA instead.

Fitting the line

The entire Python code to fit a line to 3D data can be found here: Fitting a line in 3D

Further Readings for PCA and SVD:

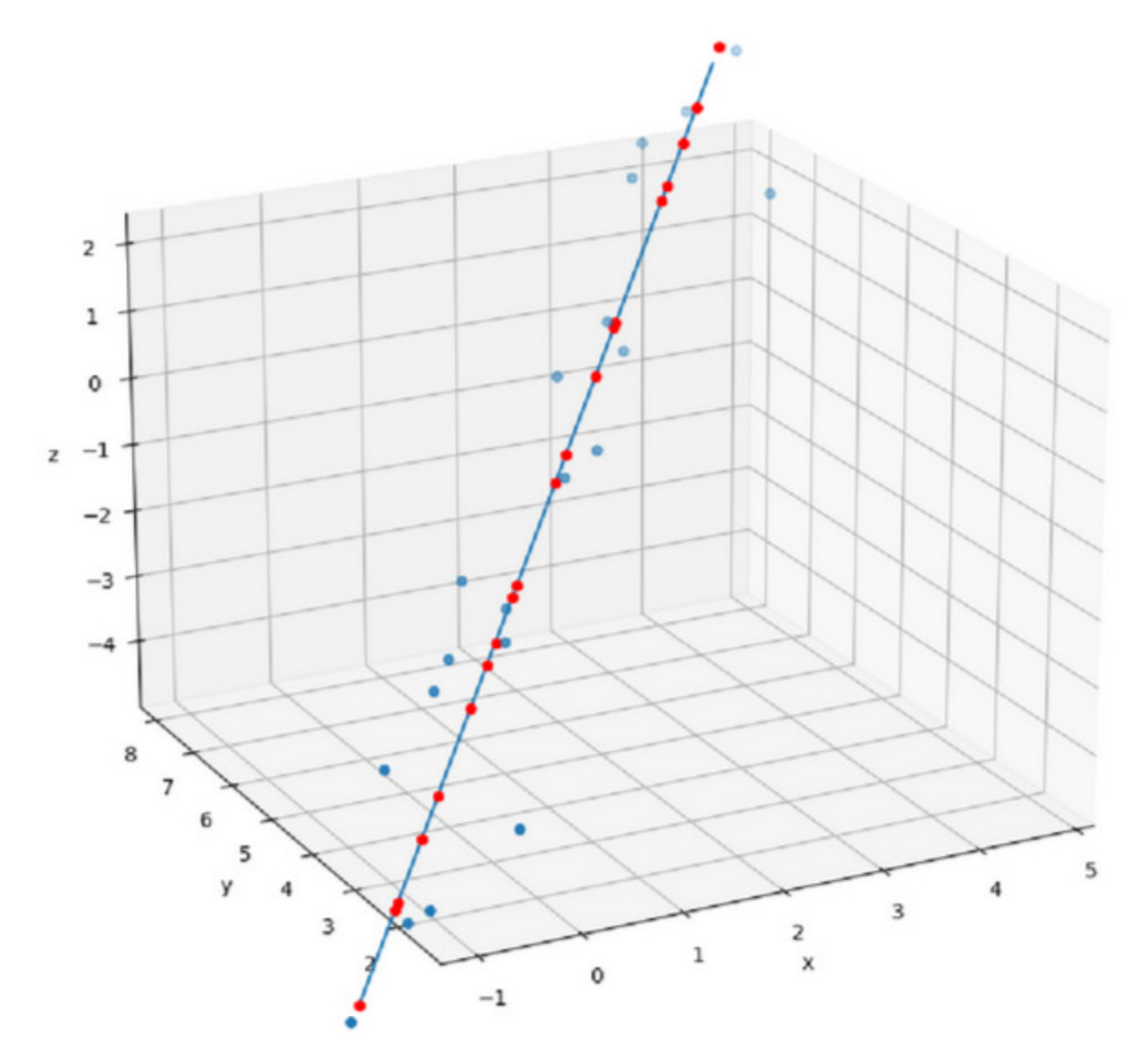
- Singular value decomposition
- What is the Singular Value Decomposition?

> This implementation can be used for any dimensional input without any change to the code.

Orthogonal projection check

With the code for fitting a line already working as implemented, the last step is to project the last measurement onto the fitted line. This was done by following the first comment presented here: Orthogonal projection of a point onto a line.

The figure below depicts 20 samples generated along a line with some additive Gaussian noise (blue dots) as well as the fitted line for all 20 points. The 20 red dots stem from the 20 initial data points projected onto the line. Unfortunately in this figure, it is rather hard to see, that the projection does work as intended. The source for code creation was found here



Testing the execution time of the algorithm

Since the SVD does some heavy matrix manipulations, I tested the execution time. Using reasonable values for the number of samples for the fitting process results in computation times of 50 µs. This should not be a problem.

> Hardware: 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz

n_samples	Average elapsed time
3	132.1388 µs
5	152.2877 µs
10	156.0010 µs
100	172.3165 µs
1000	385.9812 µs
10000	2418.5276 µs

Code

Below is a functioning version of the code, which does all of the things described above. This is only a backup version, the intended code will be on Bitbucket inside the UWB repository.

fit_and_project_line.py
11 Sept. 2023, 05:07 PM

Trusting the time measurements

This section explains a small alteration of the algorithm above, which hopefully improves the estimation.

> The code does work with more than 3 dimensions but for the sake of keeping things more concrete in this documentation I only talk about the 3D case.

In our specific case, the algorithm can be enhanced by trusting the time measurements 100%. The fitting process returns a 3D line function f in the form of

$$f : \begin{bmatrix} x \\ y \\ t \end{bmatrix} = u \cdot \begin{bmatrix} a \\ b \\ c \end{bmatrix} + \begin{bmatrix} x_0 \\ y_0 \\ t_0 \end{bmatrix}$$

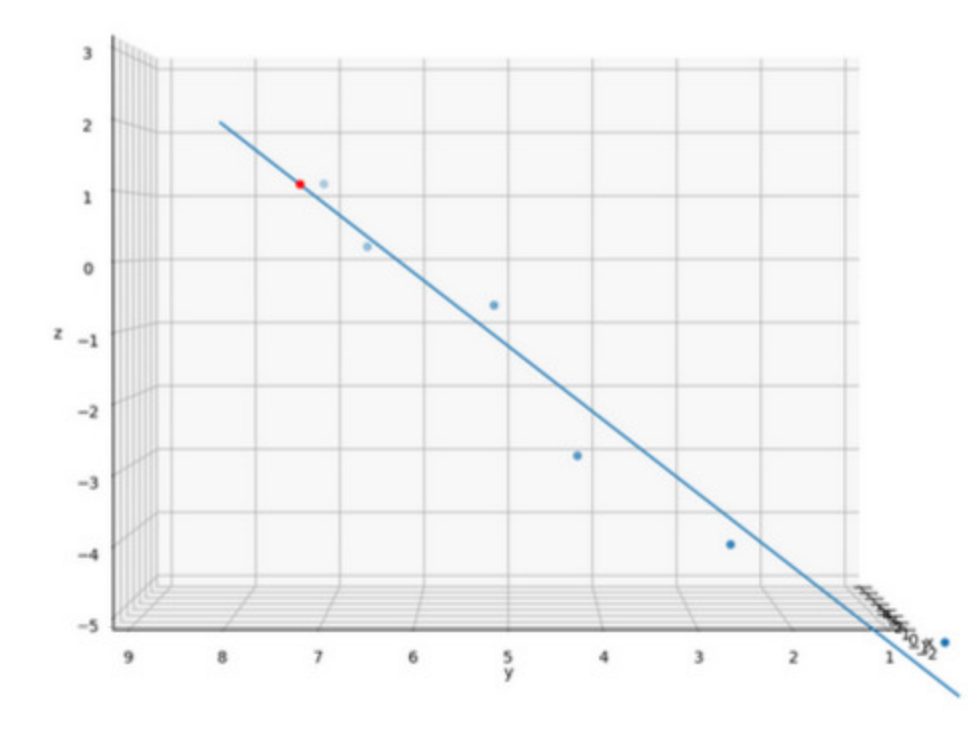
where a,b,c,x0,y0 and t0 are known variables determined during the fitting process (SVD, PCA). The only unknown parameter is the variable u. By trusting the time measurements to be accurate, it is possible to calculate the parameter u, which fully determines the 3D line, since it only has one D.O.F (Degree Of Freedom). Given that we know the value of t, u can be calculated as follows

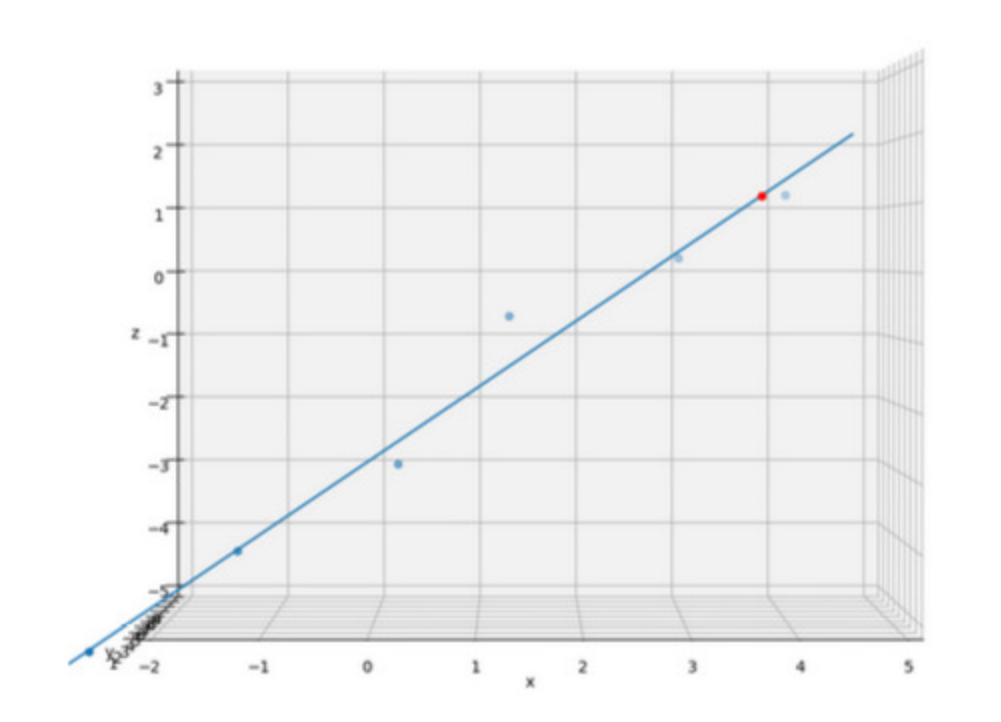
$$t = u \cdot c + t_0$$
$$u = \frac{t - t0}{c}$$

Since the running variable u has been determined, the estimated position can be calculated by evaluating the fitted function f at u:

$$\boldsymbol{x}_{est} = \begin{bmatrix} x \\ y \\ t \end{bmatrix} = f(u)$$

To validate whether this process works as intended, the resulting value for t inside $\mathbf{x}_{est}$ has to be the same as the initial value t. The figure below depicts a visual proof, that this approach works as intended. The blue dots are the measurements, the blue line is the fitted function and the red dot is the estimated position $\mathbf{x}_{est}$. Both plots show the exact same data but once from the y-z plane and once from the x-z plane. The coordinate z is the time. The latest measurement is the point with the highest transparency.





From both figures, it is visible that the z-coordinate did not change between the last measurement and the estimated position. Only the x and y coordinates were changed - validating the code.

While this approach does work rather well with artificially generated data, it plummets in performance when used on actual data, visible in the table below.

One last thought I had, was to fit the line such that it goes through the trusted times at all times and only varies in x- and y-direction, but this is most likely the same as the 2D case.

Another idea would be, to run three 2D line fitters in series. The first fitter uses x and t as input resulting in a, the second fitter uses y and t as input, resulting in b and the last fitter uses and a and b as input. I hope this implicitly weights the time measurements higher since it is used two times during the final fitting process, but this is just an assumption.

Comparing it with the previous implementation

The previous implementation was a simple 2D linear regression compared to the current 3D "linear regression" which is not a linear regression but PCA. Note, that while the 3D data did use time as an additional feature, the final plots for "3D PCA" only display the x- and y- coordinates for comparison reasons.

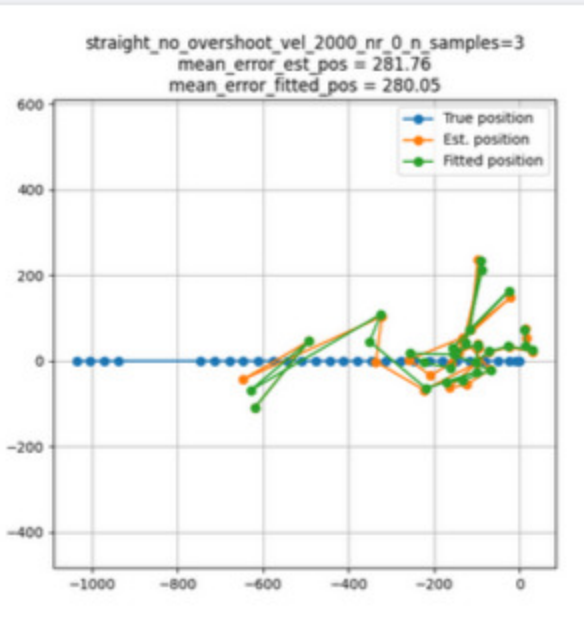
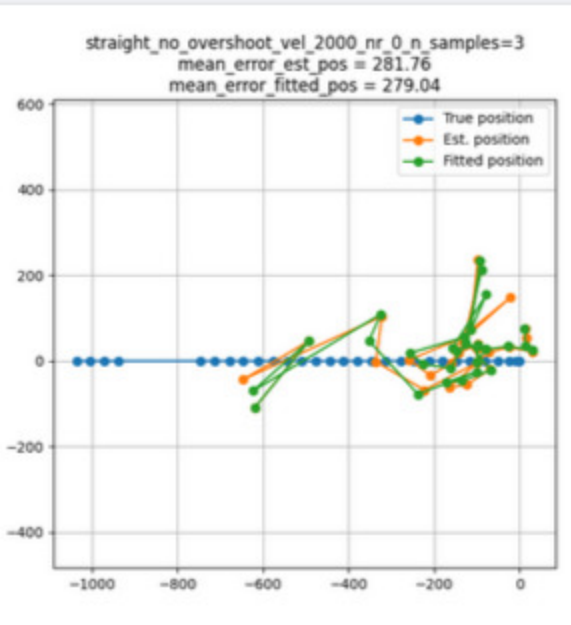
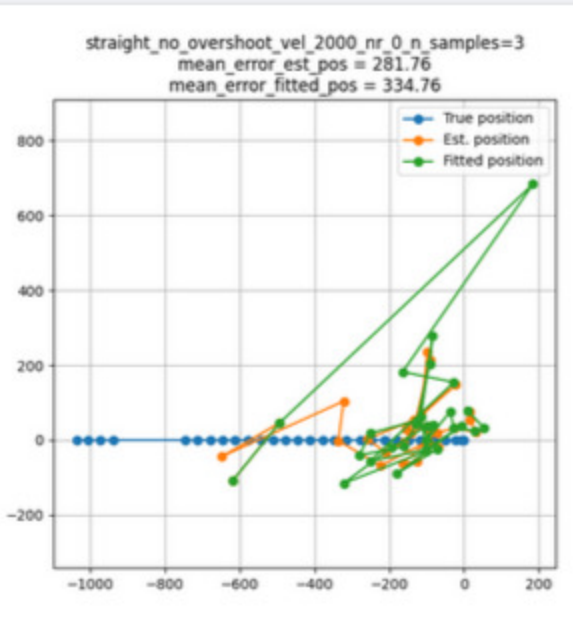
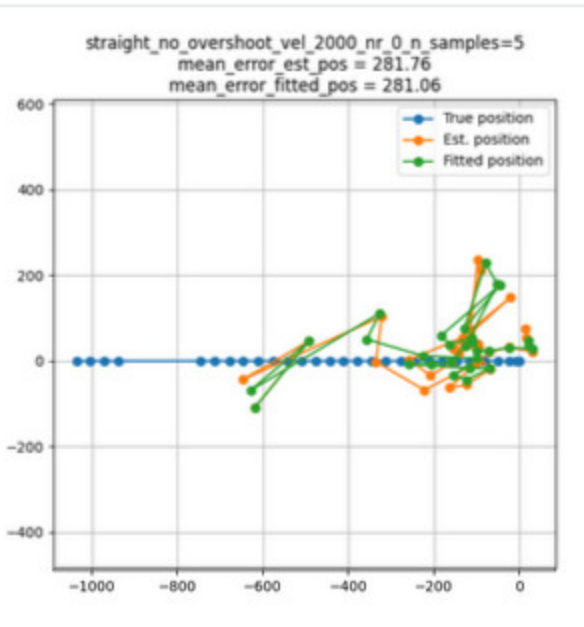
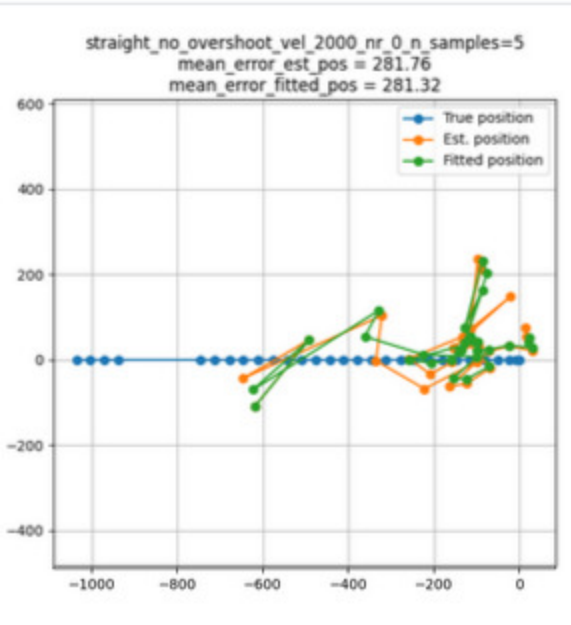
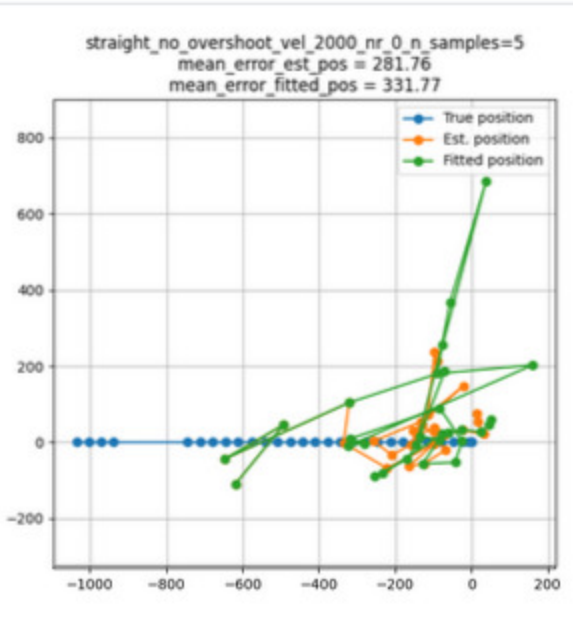
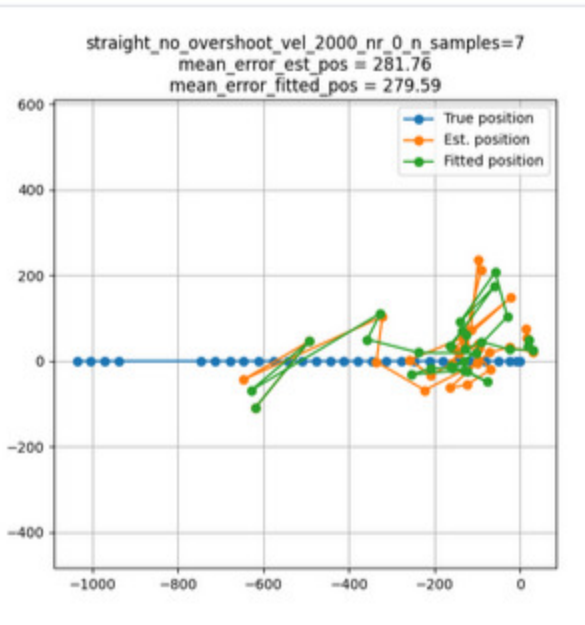
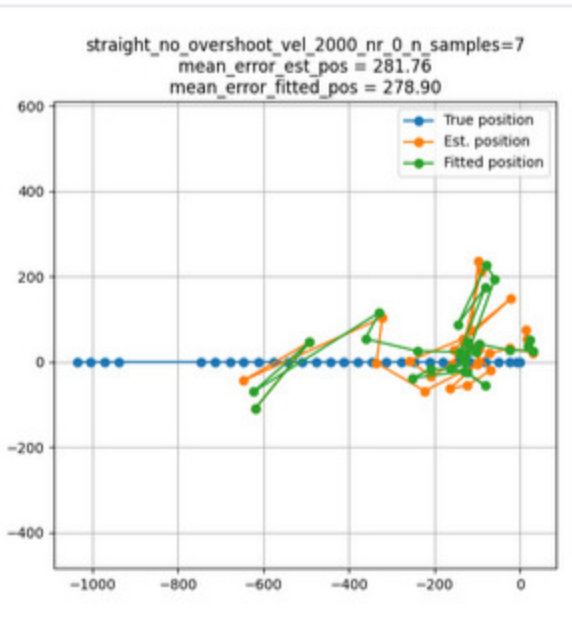
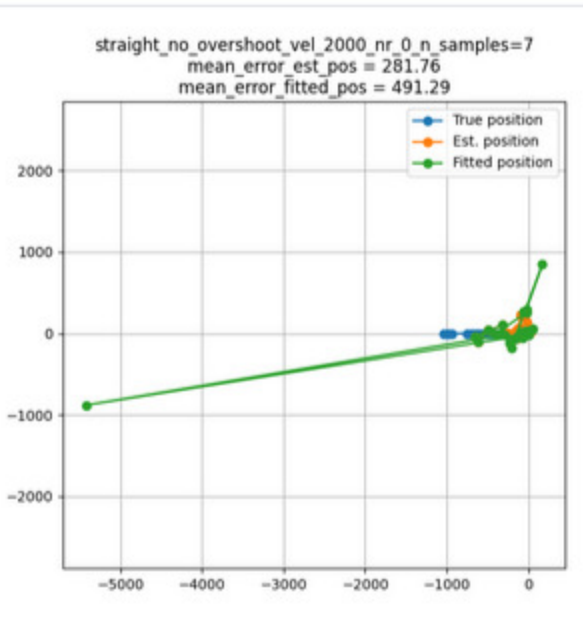
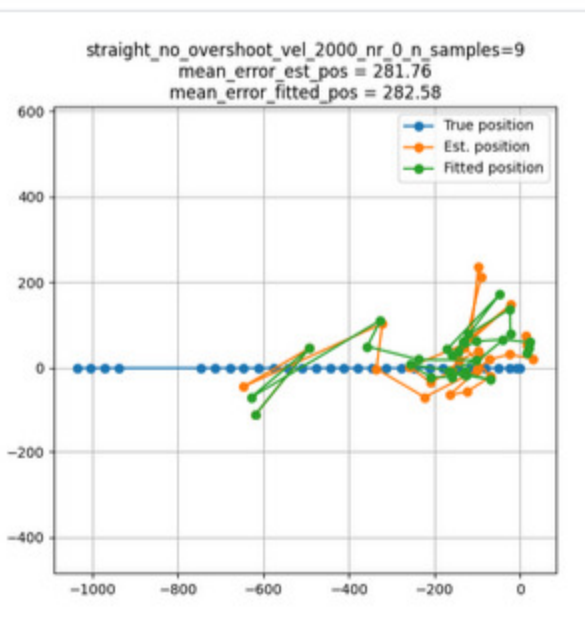
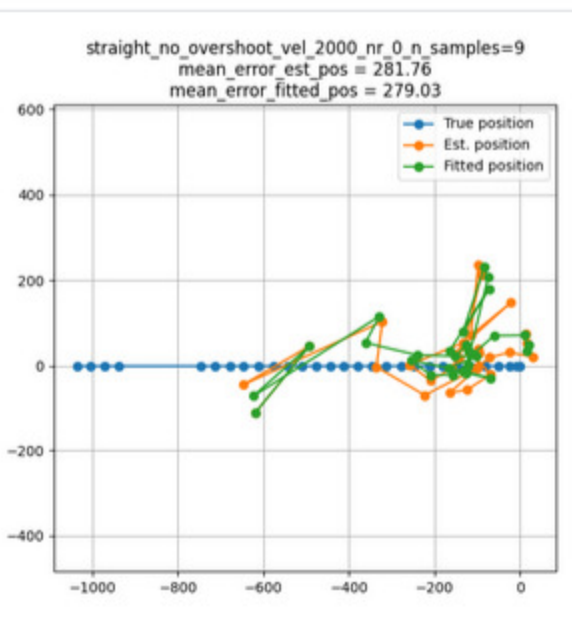
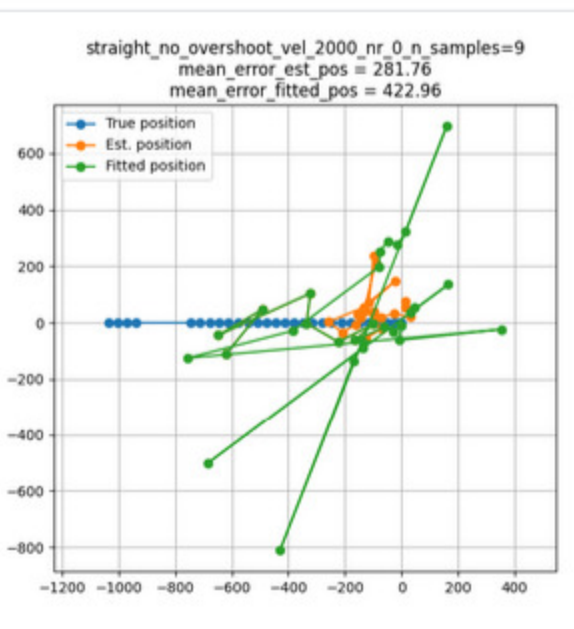
Data:

```
1 easelink/Projects/02_Development_Projects/E07_eTaxi_2.0/5_Software/04_Testing/UWB
```

While the 2D and 3D algorithms deliver similar results, the 3D PCA with trusting-time fucks up majorly. This is because the estimated position now depends more heavily on the fitted line and if the first few samples create a badly fitted line, the orthogonal projection does not come to rescue this time.

> There is also another small deviation of 3D PCA with trusting-time. Since it heavily depends on the fitted line I set min_samples parameter to the same value as n_samples.
>
> The min_samples variable controls after how many measurements the estimation is computed. For example, if n_samples = 12 and min_samples = 3, the estimator would estimate the position after only 3 measurements using these 3 points to fit a line but if the algorithm runs longer it eventually uses 12 measurements to fit the line.

There does not seem to be a huge difference between 2D Linear Regression and 3D PCA. They

	2D Linear Regression	3D PCA	3D PCA with trusting-time
n_samples = 3			
n_samples = 5			
n_samples = 7			
n_samples = 9			

References

- https://stackoverflow.com/questions/2298390/fitting-a-line-in-3d
- https://stackoverflow.com/questions/24747643/3d-linear-regression
- https://en.wikipedia.org/wiki/Singular_value_decomposition
- https://www.youtube.com/watch?v=CpD9XlTu3ys&pp=ygUoc2luZ3VsYXIgdmFsdWUgZGVjb21wb3NpdGlvbiAzYmx1ZTFicm93bg%3D%3D
- https://math.stackexchange.com/questions/404440/what-is-the-equation-for-a-3d-line
- https://math.stackexchange.com/questions/62633/orthogonal-projection-of-a-point-onto-a-line

Zugehörige Seiten

Focus Team UWB: KW25, Distance measurements evaluation
E07: eTaxi 2.0
Gemeinsam organisiert

Report UWB Positioning Algorithm z-coordinate
E07: eTaxi 2.0
Gemeinsam organisiert

6.3 Design Validation
E07: eTaxi 2.0
Gemeinsam organisiert