Trilateration | ED18B027 Report

Created	@May 9, 2021 9:34 PM
Status	
Tag	

Task 1: Dataset Generation

A **100×100 grid** was simulated as a **Set** with each element of the set being a **Tuple** of the **x** and **y** coordinate.

Randomly generated **100 unique anchor location sets** and for each randomly generated anchor location set, **50 random unique node locations** were generated.

A Set was used instead of directly selecting a random x and y because we need unique points. Since it's easier to sample **unique** points from an iterable, which also meant we needed to remove used points aka create **subsets**, using Sets was very convinient instead of if-else statements to check for repeatition.

For each of the 100 anchor cases, 3 grid cells were sampled from (**grid - used_anchors**) this makes sure each triplet is completely unique, i.e., it eleminates **(A,B,C)** from repeating but also **(A,B,D)** from happening. For each anchor triplet, 50 grid cells were sampled from (**grid - anchor_triplet**). These are stored in **true_locations.csv**.

For each of the 5000 nodes, **range_triplets** were found using the corresponding anchor_triplet. These were stored in **pure_ranges.csv**. **Gaussian** noise of ***mean*** **0.5**, **1** and **2** with ***variance*** **0.1** were added and stored in **noisy_ranges_05.csv**, **noisy_ranges_2.csv**, **noisy_ranges_2.csv** respectively.

All this was implemented in **generate_dataset.py**

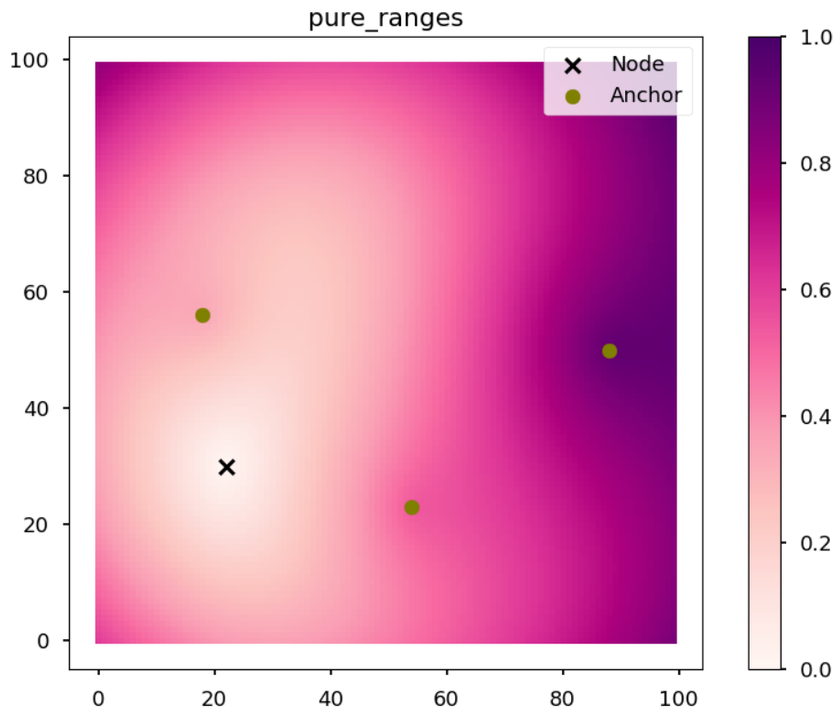
NOTE: Delimiter of **" | "** was used instead of commas as it was easier to split them later, *as lists and tuples also had commas*.

Task 2: Brute-Force Localization

For a random anchor triplet configuration and a random node location in the dataset, RMS cost (Root Mean Square Error) of every cell (x,y) of the 100×100 grid was calculated and plotted as a heatmap. The error in the cost function was calculated as **calc_range - measured_range**. Where calc_range comes from **euclidean_dist(anchor, (x,y))** and measured_range from the dataset (**ast_eval** is used to convert string tuples into python tuples).

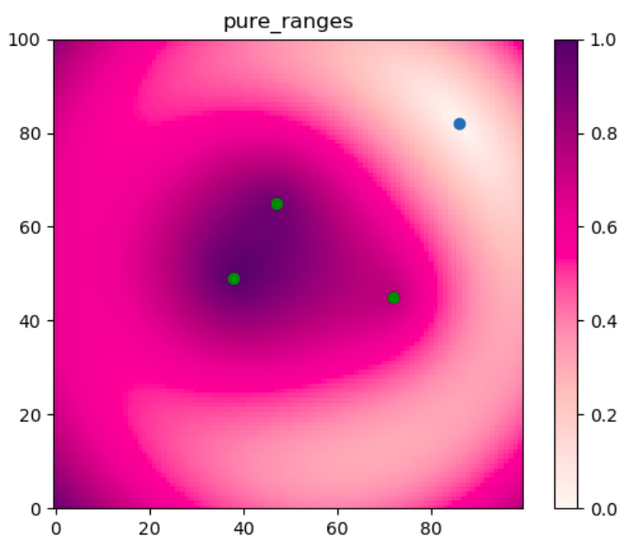
All this is implemented in **brute_force_localization.py**

In the example heatmap below, the darker the color the more the error. Anchors are marked by green dots and actual node by **x**.

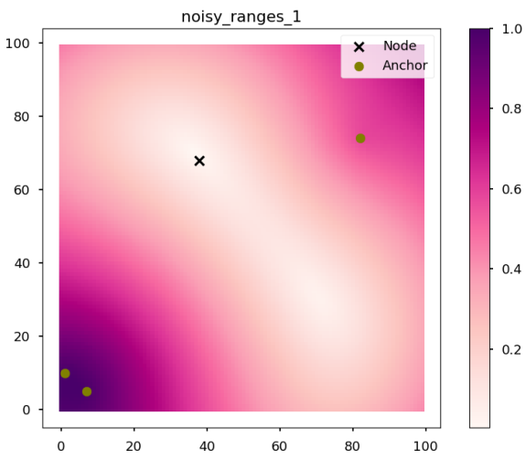


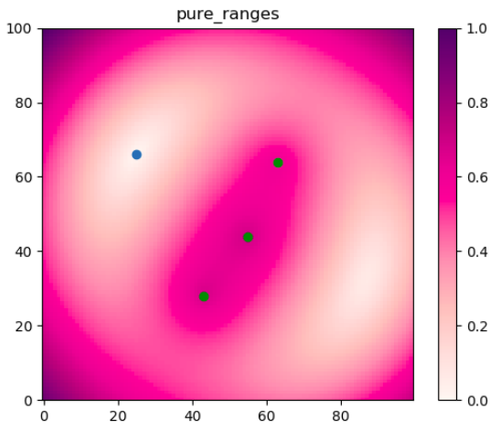
Observations

- This is a brute force approach of checking every cell to find the solution by finding global minima.
- The global minimas lie almost exactly on top of the actual node location always with very little variation with noise.
- It is observed that as range errors increases the heatmap becomes more "**blurry**" as errors increase and darker regions encroach more area.
- Local minimas depend more on the anchor locations than on the noise. Eg: For the case below, there may be a local minima at the bottom because of the geometry of the anchors.



- It is also observed that anchor points have higher errors which makes sense by looking at the range equations.
- There also may be cases where there are more than 1 global minima because of alignment of anchors. This also creates a symmetry as seen below. This causes more chance of picking the wrong point.





- Random picking of anchor points leads to many situations which look bad. There may be many cases where the anchors are too close or almost colinear. Hence it becomes important to place stationary anchors properly or make sure moving anchors are coordinated so that they maintain proper distance between themselves. (like GPS/GNSS)

Task 3: Trilateration

LMFIT is used as an optimizer to trilaterate a node which is then rounded off to the nearest integer. The main logic for the above can be found in the **trilaterate()** function.

A normal Least Square implementation for triangulation with LMFIT would include defining **Parameters** for **x** and **y**, then **minimizing residuals** given **anchor triplet** and **range triplet**. More precisely, the residual function takes in the potential node, anchor triplet and range triplet and returns (measured_range_triplet - calculated_range_triplet) as residual error which is used by LMFIT to minimize using the **Levenberg-Marquardt** method taking the starting point as **(50,50)**.

There is also a **brute-step** user input that the function takes to figure out the best starting point using brute force methods. This is implemented in the first half of trilaterate. The reason for using this is explained later below.

This is applied to all the 100×50 nodes for each of the 4 error cases. The output for which is stored in 4 files **{}_pure_loc.csv, {}_noisy_locs_05.csv, {}_noisy_locs_1.csv, {}_noisy_locs_2.csv**. Where **{}** is the brute-step used or nothing if not used. The starting point is taken as **50,50** if no brute-step is given.

All of the above is implemented in **trilateration.py**

Trilateration per Range Measurement Error

The trilaterated nodes are now compared to the groundtruth location of the nodes and trilateration errors are calculated for each of the 4 range measurement error cases.

A user input **--file_id** is asked which is the prefix of the particular files to use which happens to be the **brute_step**. No input defaults to using the files with prefix **"_"**.

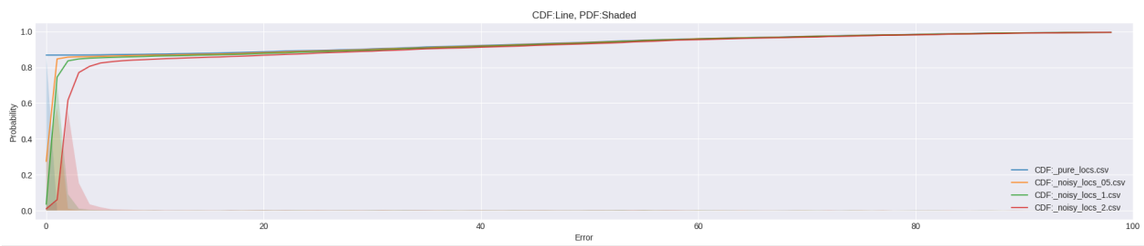
For each range error the following are printed to the terminal:

- Trilateration Count
- Max Error
- Median Error
- 75^{th} Percentile Error
- 95^{th} Percentile Error

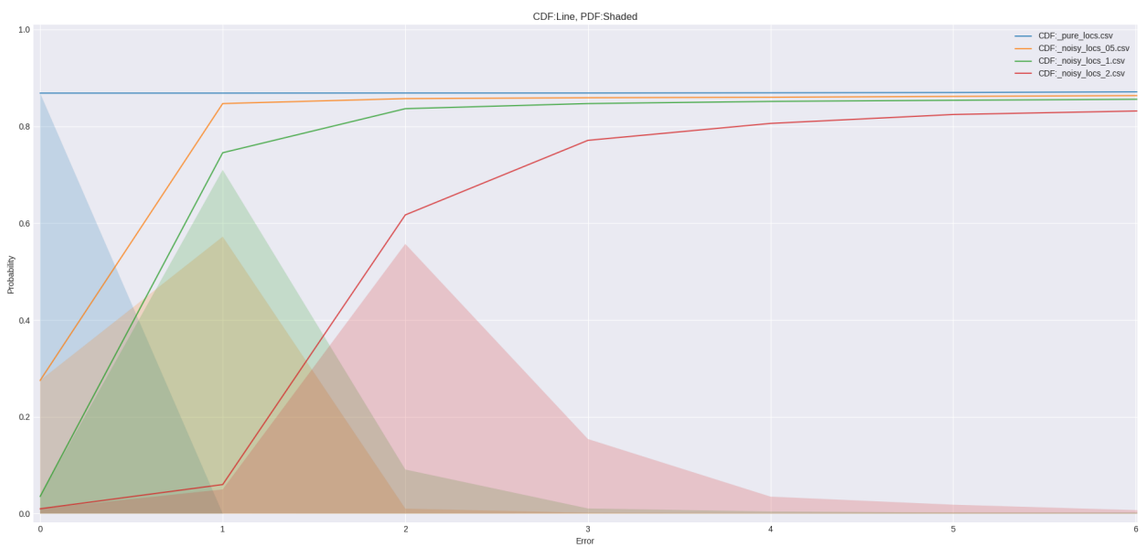
```
Parsing file _noisy_locs_1.csv and true_locations.csv pairwise...
Trilat Count: 5000
Max Error: 126.37246535539299
Median Error: 1.4142135623730951
75th Percentile Error: 2.0
95th Percentile Error: 56.445974477886665

Plotting CDF of corresponding trilateration errors...
```

The **CDF** of the trilateration errors is plotted as a line underwhich the **PDF** is shaded. All 4 of these plots are drawn on the same graph for comparison.

Full Plot:

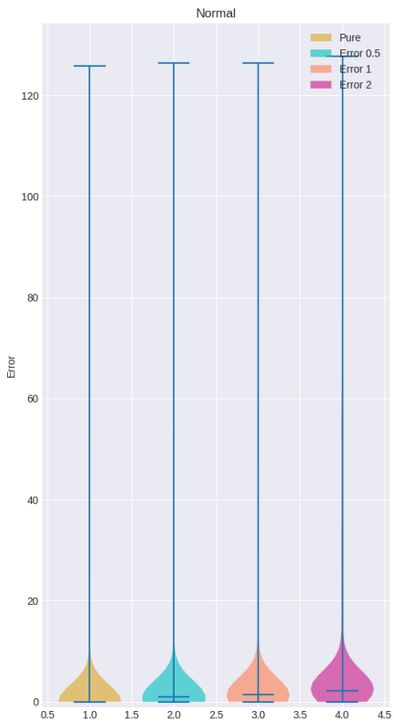


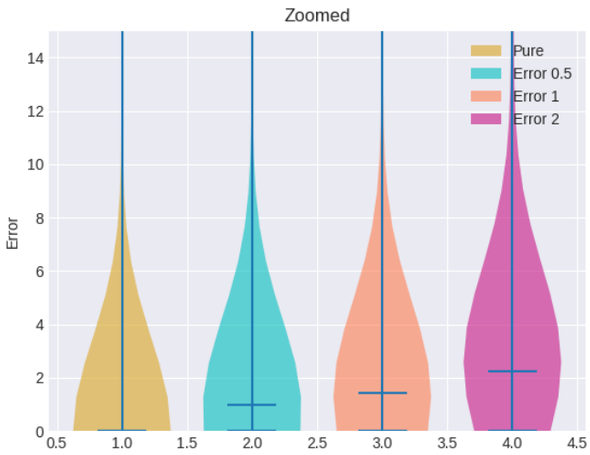
Zoomed Plot:



Apart from this, a **Violin Plot** is drawn for the 4 range errors to give a different representation of the distribution. It shows the **Max**, **Median** and **PDF** distrbution of the errors. A **zoomed** version is also plotted right after.

All this is implemented in the file **trilateration_errors_per_range_error.py**

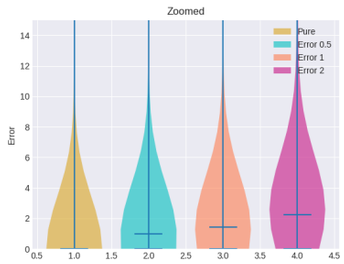




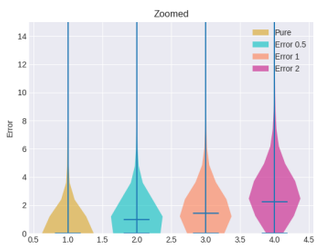
Observation

- The CDF tells us that the more gradually the slope changes(slower convergence to 1), more is the variance. Therefore in this case, the more the range error, the more the variance even though we started out with the same variance of **0.1**. The mean is proportional to variance in this case. This may be because there is no negative error which would allow for a gaussian distribution. So it is getting **"reflected"**. Maybe if the means were large enough all of them would have the same variance.
- Effect of rounding off can be seen in the CDF and PDF (discretization) and in the medians. **1.414** corresponds to $\sqrt{2}$ and **2.236** corresponds to $\sqrt{5}$.
- Most of the error lies in a small region around the median but the Max error is around **125** and the 95^{th} percentile error is around **55** - **60** which is bad. This may be happening because the starting point chosen is **(50,50)** which might not work for all node and anchor point configurations. Hence, the nodes may land up in local minimas. This can be solved by brute forcing to find a good starting point. Hence **--brute-step**. Brute step of 10 would give a best guess in the multiple of 10, similarly for 5.
- This decreases 95^{th} percentile error to **5** - **8** which is a major improvement. This means 95% of the time errors are less than 5 - 8 grid cells. The variances also reduce. The Max error also reduces to around **120** but not by much. Using a brute step of **5** gives even better results. Max error reduces to around **95**, variance is even smaller and 95^{th} percentile error is even smaller **1** - **5**.
- This only solves the local minima problem. There might be cases as shown above in which there are more than 1 global minima or almost global minima,

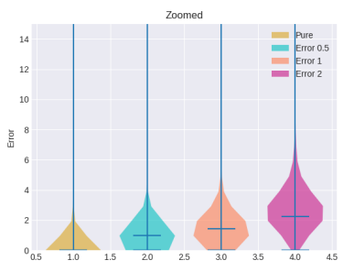
which is hard to decide hence causing very large errors 5% of the time. Maybe the symmetry argument can be used to check for multiple solution.



Starting point (50,50)



Brute-Step 10



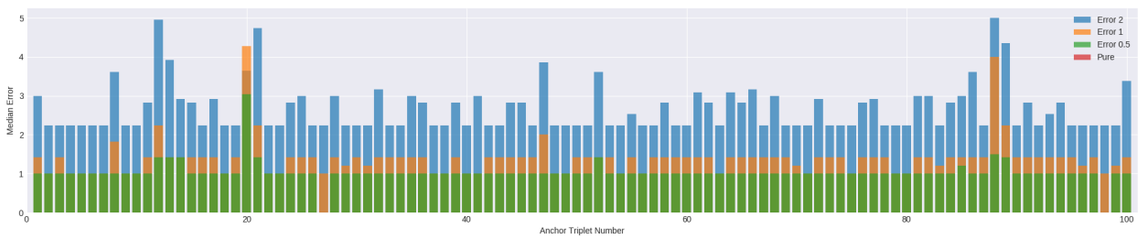
Brute-Step 5

- But this comes with a tradeoff, smaller the brute step more the computation. It takes about 2 seconds for **50 trilaterizations** with no brute step, **4** - **5** seconds for brute-step **10** and **8** - **9** seconds for brute step **5**. This seems to be reasonable for a real-time system but trying out other methods from LMFIT take about ***1min for 50 nodes! which would mean around 7 hours to run the script!!***

Task 4: Impact of anchor point

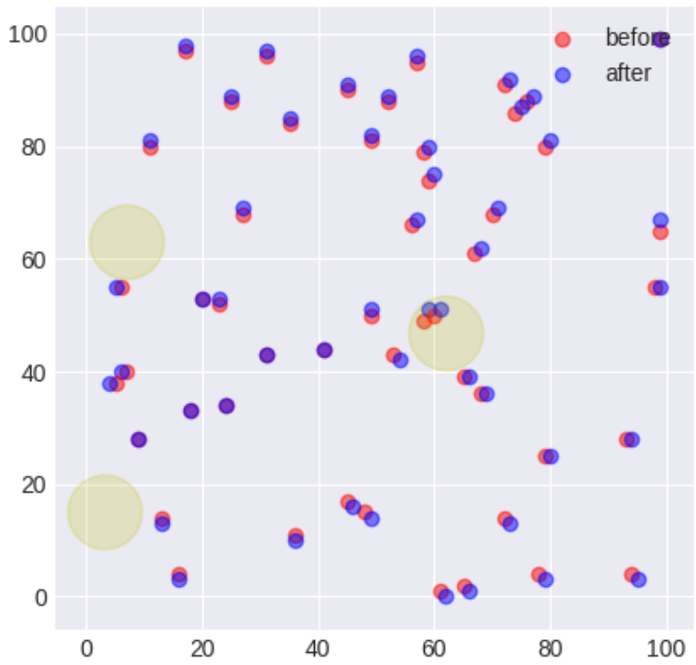
For each of the **100** anchor triplets, the median error of the **50** nodes are calcuated and shown on a bar graph. This is done for all the **4** range error cases and shown on the same plot for comparison.

Similar to the above a user input **--file_id** is asked which is the prefix of the particular files to use.



Apart from this a before-after comparison for each anchor triplet is visualised as a scatter plot. This can be activated using the **--show_before_after** flag.

The **Yellow** transparent larger circles mark the 3 anchor points. **Red** shows actual ground truth positions and **Blue** shows trilaterated locations. **Purple** shows the extent of overlap.

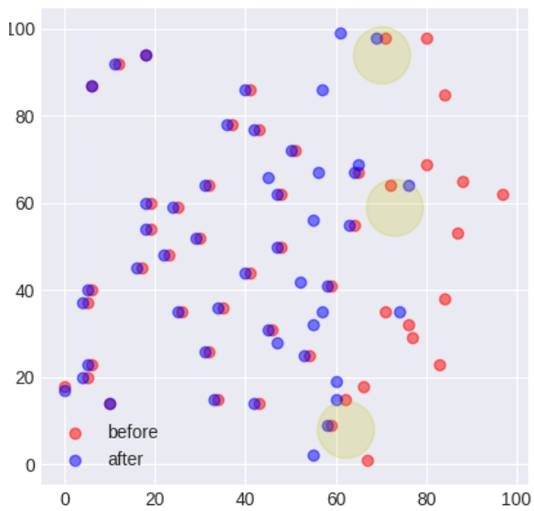


Starting Point (50,50), Range Error 1

All of the above is implemented in **trilateration_errors_per_anchor_triplet.py**

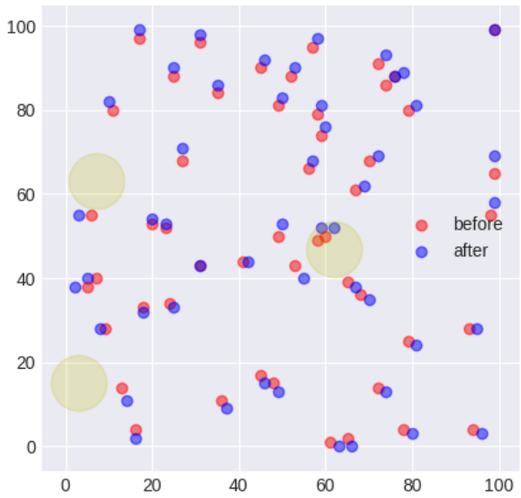
Observations

- Median errors for all anchor triplet configuration is **0** when range error is **0**.
- Median errors in general decreases as **brute-step** decreases.
- As seen from the Bar-graph, some anchor configurations have more median errors than others. This is because of **GDoP** observed because of the configuration of the anchor points. One such example is shown below which is the first bar in the bar graph.



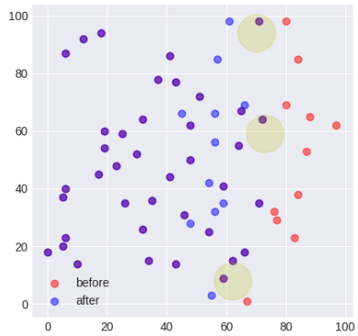
Starting Point (50,50), Range Error 0.5

- From the below graph it is evident that some locations have a better chance of trianguating correctly than others because of the geometry. It is also observed that the nodes are **pushed away radial** from the anchor points. This is because we **added** an error of 2 to the ranges. If we had **added/subtracted** randomly (**+ or - 2**) we would have gotten a different result.

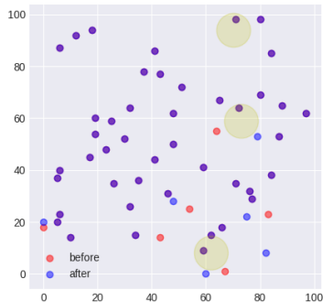


Starting Point (50,50), Range Error 2

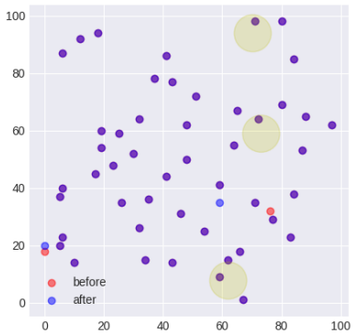
- As seen below, decreasing the **brute-step** leads to better trilaterization.



Starting (50,50), Pure Range



Brute Step 10, Pure Range



Brute Step 5, Pure Range

Metric to Correlate Anchor Locations with Magnitude of Error

A naive approach to choose $F((x_1, y_1), (x_3, y_3), (x_3, y_3))$ would be area of triangle formed, but this would mean it would capture colinear cases but also cases likes a closely located equilateral triangle which would produce less errors.

A better approach would be to use a **normalized dot product** so that all triangles are normalized. Then an appropriate cut-off can be choosen.

To correlate localization error only the anchor locations wouldn't be enough, we would also need information about the **region of interest** or the **bounding box**. Example in the above example if the colinear triplet is located at the edge of the 100×100 grid then the localization error would be almost **0** as there are no multiple (global) minimas or symmetry. Basically, we don't care what happens after 100. So, this also has to be factored into the metric somehow.

Yes, this could also be learned using **ML techniques** by searching for a pattern by looking at multiple training examples.

Resources

- LMFIT: https://lmfit.github.io/lmfit-py/intro.html
- Numpy Percentile: https://numpy.org/doc/stable/reference/generated/numpy.percentile.html

- ArgParse: https://docs.python.org/3/howto/argparse.html
- AST: https://stackoverflow.com/questions/15197673/using-pythons-eval-vs-ast-literal-eval
- Matplotlib