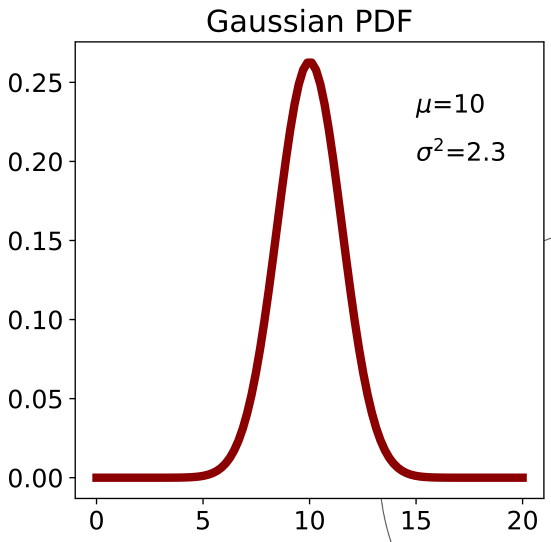
UNIVERSITY OF COPENHAGEN
NIELS BOHR INSTITUTE

Problem Set 2

Advanced Methods in Applied Statistics



Kimi Cardoso Kreilgaard (TWN176)

Submission Date
02.03.2022

1 Problem 1: Neutrinos

1.1 (1A)

The unnormalised PDF for the neutrino energies can be computed directly with the formula given in the assignment. To normalise them we need to find a constant to multiply the function with so the integrated area under the PDF is equal to one. Since the integrand is not converging, we have to chose integral limits a and b. Since the energy cannot be negative, we define the lower limit $a = 0$. We plot the unnormalised distributions to estimate their extent, and find that the distributions flattens out between circa 50 and 250 depending on the black hole mass. We conservatively choose the upper limit as $b = 500$. This integral can be analytically determined, but I had some problem with Sympy so I ended up nummerically integrating the function with `Scipy.integrate.quad` which is their general purpose integrator and should work fine here. The constant that normalises can then be found as $k = 1/\text{area}$, where the are is the definite integral within the bounds. This is all implemented in the function `PDF_norm`, and using this formula for the three different black hole masses, we obtain the normalised PDF's displayed in Fig. 1 below.

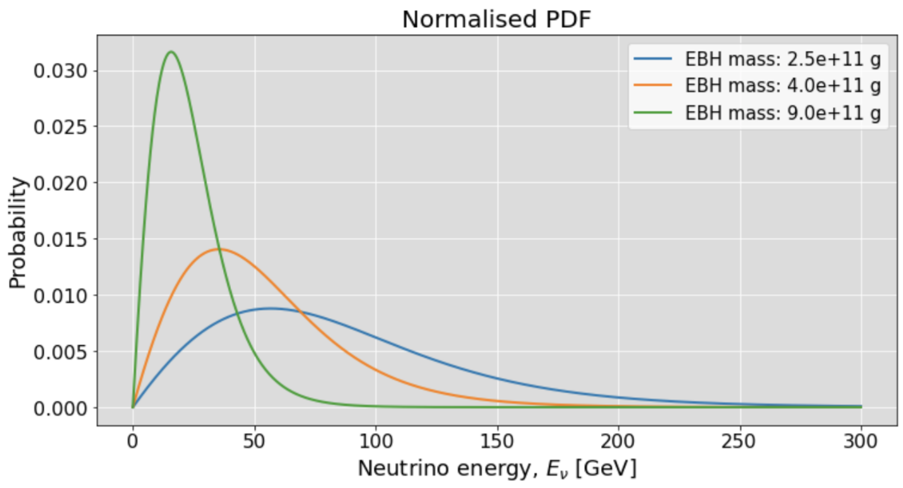


Figure 1

1.2 (1B)

We first save the linked file with simulated neutrino energies as a csv-file. Using `numpy.genfrom.txt` we can import the values into Python. To plot the simulated data together with the calculated normalised PDF's from the previous problem we need to normalise the histogram of simulated data. This function is already included in the `matplotlib.pyplot.hist` function, if the density parameter is `density=True`. This ensures that the integral under the histogram is equal to one. The result can be seen in Fig. 2 below. Just by looking at the plot we can guess that the distribution would be something in between the orange and the green PDF. We thus conclude that the black hole mass should be in between $4 \cdot 10^1 1$g and $9 \cdot 10^1 1$g. My eye ball guess would probably be $6 \cdot 10^1 1$g but it is hard to determine exactly how the function will move with the mass since it is not directly dependent on it, but dependent on the temperature which is a function of the mass.

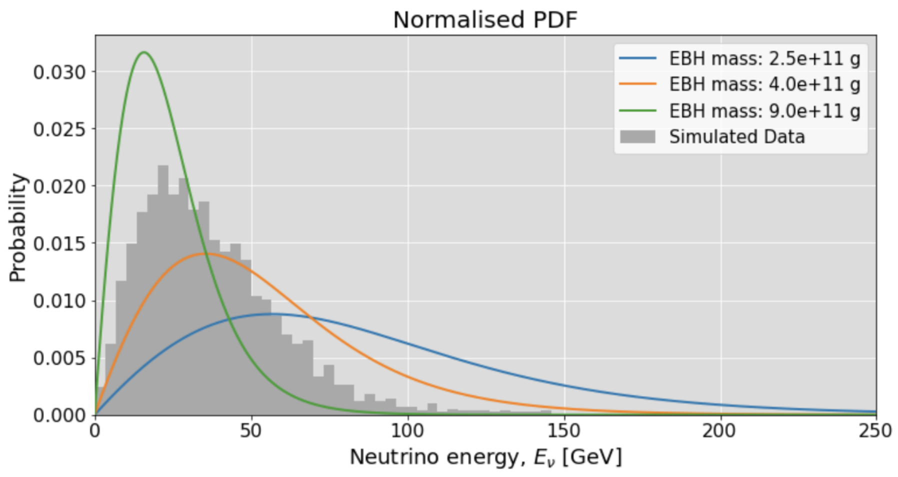


Figure 2

1.3 (1C)

To find a quick estimate of the mass of the black hole that best fit the simulated neutrino data, we perform a 1D raster scan of the unbinned log likelihood of the neutrino energy data. Since we determined in the previous problem that the mass is probably in the range $4 \cdot 10^1 1$g and $9 \cdot 10^1 1$g this is the interval of masses we will be scanning in. The code runs pretty fast so we perform quickly a scan of 1000 values. The results are plotted in Fig. 3. **We find the maximum likelihood estimate of the black hole mass to be $5.7467467467 \cdot 10^1 1$g with a likelihood of -11252.798562014359**

The values of the log likelihood for the three proposed black hole masses from the previous problem are:

- For mass $2.5 \cdot 10^1 1$g, the likelihood was: -12979
- For mass $4 \cdot 10^1 1$g, the likelihood was: -11638
- For mass $9 \cdot 10^1 1$g, the likelihood was: -12025

meaning that of the three the mass of $4 \cdot 10^1 1$g provides a better fit to the simulated neutrino energies.

1.4 (1D)

We now want to find the best mass with minimizing techniques. We therefor wish to find $\min(-\mathcal{L})$. We define a new function that returns the negative log likelihood instead, `neg_llh`. We can now use

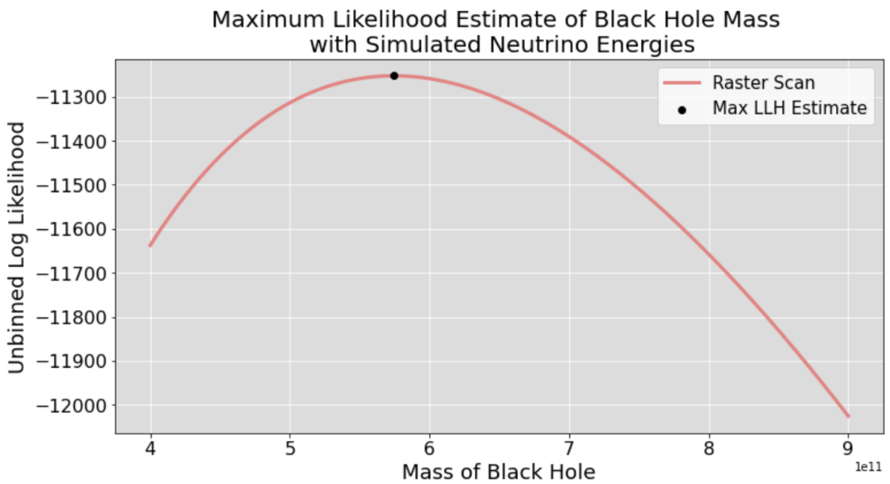


Figure 3

`scipy.optimize.minimize` to minimize the function. The best fit according to the scipy minimizer is $5.7467467 \cdot 10^11$g with a likelihood of 11252.798562014359. This means that we get the exact same solution, at least when we use the mass from the previous problem as our initial guess. But this is a general problem with the minimizer for such large values, it will just return the initial guess. To alleviate this problem within the function of the negative likelihood that we want to minimize I multiply the number with 1e11. So the solution we get from the minimizer should also be multiplied by this factor. I also infer a bound saying the mass has to be larger than 1e10 (0.1 in the minimizer since it is in another scale now). This makes the minimizer work and with an initial guess to the minimizer of 20 (corresponding to $20 \cdot 10^11$g) we find an appropriate solution. **The maximum log likelihood estimate of the mass is** $5.74586382 \cdot 10^11$**g** with a (negative likelihood of 11252.798483865, which is slightly better than the raster scan in terms of the likelihood.)

2 Problem 2: Pace Maker

2.1 (2A)

To find the probability that a defective pace maker came from the A_2 facility, we need to use the discrete version of Bayes theorem:

$$P(A_n|D) = \frac{P(D|A_n)P(A_n)}{\sum_i P(D|A_i)P(A_i)}$$

$P(A_n|D)$ is the probability of A_n, i.e. that the pacemaker cam from facility A_n, given that D has already been observed, where D denotes a defect, and is the posterior probability we are interested in. $P(D|A_n)$ is the likelihood that a pace maker from facility A_n is defect (the defective rate), and $P(A_n)$ is the prior, in this case the probability that any pace maker comes from facility A_n (the production rate). The marginal likelihood is given by $\sum_i P(D|A_i)P(A_i)$ and is what normalises our probability.

We insert A_2 in A_n's place, and find that: **The probability that an observed defective pace maker is from facility A2 is given by $\mathbf{P(A_2) = 18.23\%}$**

To find which facility an observed defective pace maker is most likely to be produced by, we compute the above for all facilities and identify the facility with the largest probability. The result for all facilities are displayed in Tab. 1 below. We find that: **If a defective pace-maker is found, it is most likely to be produced by facility A5, with a probability of $\mathbf{P(A_2) = 23.66\%}$**,

Facility	A1	A2	A3	A4	A5
$\mathbf{P(A_n\|D)}$	21.37%	18.32%	15.27%	21.37%	23.66 %

Table 1

2.2 (2B)

To ensure that all facilities have the same probability of being identified with a failed pace-maker, but still with the least total defects and with no changes to the per facility production, we need to increase the defective product rates. To have the least total defects however, we will not change the defection rate of A5. This means that $P(A_5|D)$ will remain the same. We now just need to change the defective rates $P(D|A_n)$ of the remaining facilities so they uphold the criteria that:

$$P(A_n|D) = P(A_5|D)$$

From this we derive that:

$$P(D|A_n) = \frac{P(D|A_5)P(A_5)}{P(A_n)}$$

and since the production rates $P(A_n)$ remains constant, this is easily computable. We compute this for all of the facilities and find the new defective rates given in Tab. 2 below, listed along with the old defective rate and the probability that an observed defective pace maker is from the given facility (which should all be the same).

Facility	A1	A2	A3	A4	A5
Old $\mathbf{P(D\|A_n)}$	2 %	4 %	10 %	3.5 %	3.1 %
New $\mathbf{P(D\|A_n)}$	2.21 %	5.17 %	15.5 %	3.88 %	3.1 %
$\mathbf{P(A_n\|D)}$	21.37%	18.32%	15.27%	21.37%	23.66 %

Table 2

2.3 (2C)

We repeat what we did in problem (2B), but now with a different data set containing data from 14 facilities. We first transcribe the data into python. We then need to identify the facility that an observed defective pace maker is most likely to be produced by. This will be used like we used facility A5 in the previous problem, i.e. this is the probability $P(A_n|D)$ that all facilities should have in the end. We find that facility A8 is most likely to have produced an observed defective pace maker with a probability of $P(A_n|D) = 13.10\%$. The equation from which we will calculate the new defective rates will thus be:

$$P(D|A_n) = \frac{P(D|A_14)P(A_14)}{P(A_n)}$$

and since the production rates $P(A_n)$ remains constant, this is easily computable. We compute this for all of the facilities and find the new defective rates given in Tab. 3 below, listed along with the constant production rate, the old defective rate and the probability that an observed defective pace maker is from the given facility (which should all be the same).

Facility	**Production Rate** $\mathbf{P(A_n)}$	**Old Defective Rate** $\mathbf{P(D\|A_n)}$	**New Defective Rate** $\mathbf{P(D\|A_n)}$	$\mathbf{P(A_n\|D)}$
A1	0.27	0.02	0.0220	0.0714
A2	0.1	0.04	0.0595	0.0714
A3	0.05	0.1	0.1190	0.0714
A4	0.08	0.035	0.0744	0.0714
A5	0.25	0.022	0.0238	0.0714
A6	0.033	0.092	0.1803	0.0714
A7	0.019	0.12	0.3132	0.0714
A8	0.085	0.07	0.0700	0.0714
A9	0.033	0.11	0.1803	0.0714
A10	0.02	0.02	0.2975	0.0714
A11	0.015	0.07	0.3967	0.0714
A12	0.022	0.06	0.2705	0.0714
A13	0.015	0.099	0.3967	0.0714
A14	0.008	0.082	0.7438	0.0714

Table 3

3 Problem 3: Crab Battle

3.1 (3A)

To calculate the movement of the crab over multiple days, we construct two functions:

- `daily_move` which takes a crab's polar coordinates and returns new polar coordinates. If `return_dist` is set to True it will also return the distance in km that it travelled during its daily walk (We will use this feature in 3B).

- `N_move` which takes a crab's polar coordinates, moves it around for `N_days` with the `daily_move` function. It stores the coordinates after each day, and converts those into cartesian coordinates (if `system='cart'`), making it easy to plot the movement of one crab during 200 days.

To perform the random movement and always move 200m along the randomly selected direction, but without stepping outside the island edge we have to look into some math of how polar vectors add. This is explored in depth in the following link: https://math.stackexchange.com/questions/1365622/adding-two-polar-vectors [accessed: 01.03.2022]. The two equations we will use are:

$$r = \sqrt{r_1^2 + r_2^2 + 2r_1 r_2 \cos(\phi_2 - \phi_1)} \tag{1}$$

$$\phi = \phi_1 + \text{arctan2}(r_2 \sin(\phi_2 - \phi_1), r_1 + r_2 \cos(\phi_2 - \phi_1)), \tag{2}$$

where we add the vectors $\vec{r} = \vec{r_1} + \vec{r_2}$ and r and ϕ denotes the magnitude and the azimuth angle, respectively. The pseudo code of the `daily_move` can now be developed and is as follows:

1. Select a random direction, an angle between 0 and 2π.

2. Calculate the magnitude of the resulting vector with Eq. (1) from adding the initial vector (starting position given to the function) and a new vector symbolising that daily walk with magnitude $r_2 = 200$m (since this is the maximum distance the crab can walk in a day) and the random angle selected in step 1.

3. Check if the magnitude of the resulting vector is larger than the radius of the island. If not, the crab remains on the island and we can directly calculate the new resulting angle with Eq. (2). If yes, the crab is not able to move that far, and we solve a second degree equation to find the magnitude of r_2 that gives the resulting vector a magnitude equal to the island radius, meaning that it would stop directly at the edge. The resulting angle is then calculated with Eq. (2).

4. Return the new polar coordinates.

Using the functions describing above, we can simulate the movement of a crab for 200 days, which is plotted in Fig. 4. Here we can also see that the crab always remain within the island, despite reaching the edge a few times.

3.2 (3B)

To measure the total distance travelled of a single crab until it reaches the edge the first time, we will use the `daily_move` function iteratively, but this time we will return the distance travelled during that day in each iteration. If the distance returned is less than the 200 m a crab will move

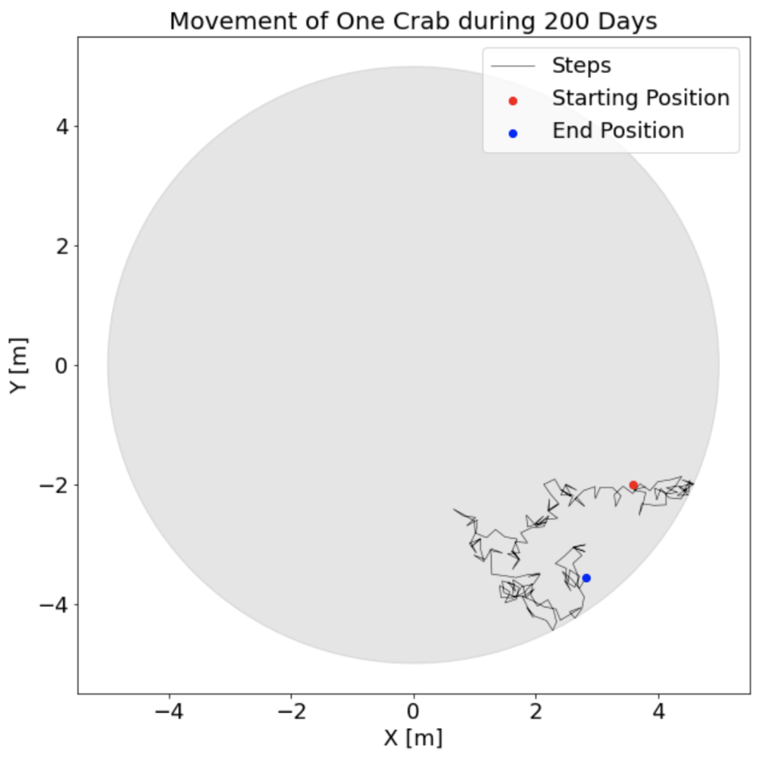


Figure 4. The resulting trajectory of a single crab starting at position (3.6km,-2.0km) and moving for 200 days. The starting position is marked with a red dot, and the final position after 200 days is marked with a blue dot.

each day it means it has reached an edge, and we can now return the sum of the distances returned each day as the total distance the crab travelled before reaching the edge. This is implemented in the `edge_travel` function, which also returns a boolean informing whether the edge was found by the crab, or if the function stopped after simulating `max_steps` days. The `max_steps` parameter is included to avoid the unlikely case that it takes a very long time to reach the edge making the code run ”forever”.
We run this experiment 501 times with a crab starting at cartesian coordinates (3.6km,-2.0km), noting the total distance travelled before reaching the edge for each experiment. Running it with `max_steps=2000` all experiment end with a crab at the edge of the island. The result is plotted in a histogram in Fig. 5, where the bins have a 2 km width and range from 0 to 200 km.

3.3 (3C)

First we load the data provided by the assignment, defining the starting position (Cartesian coordinates) for 20 crabs in km. The crabs now have to battle each other, the winning crab absorbing the mass of the loser crab, which is removed from the simulation. There are multiple parts to this problem, and we will solve them by combining a few functions, that are described below.

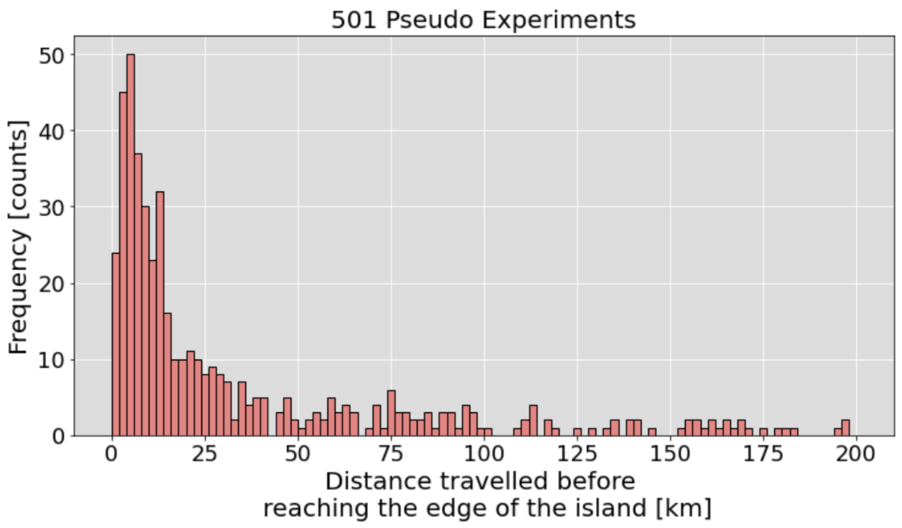


Figure 5. Distribution of the total distance travelled, before a crab at starting position (3.6km,-2.0km) reaches the edge of the island, for 1000 Monte Carlo experiments.

- `battle_tournament` accepts the positions of the crabs. This is just the data file provided by the assignment, but it is generalised so we can start with any number of crabs. It calculates the distance matrix with `scipy.spatial.distance_matrix` between all the crabs in the simulation. To not repeat any distances, the lower triangle of the matrix is filled with `Nan`s which the rest of the code will ignore. The indices of all crabs that are to battle is then found and put into an array, where each element is a tuple containing the indices of the two crabs to battle in that game. The associated distance between the two crabs in each game (battle) is found, and we can now sort this array and the indices array so the first element defines the first battle, the second element defines the second battle and so on, sorted from lowest distance to highest distance. This sorted array of crab indices is called the `tournament`, since it is the order the battles should occur in, and is returned from the function.

- `battle` accepts this `tournament` array along with a mass array containing the mass of each crab. This function defines a status array where all crabs are alive in the beginning. Then for each game (element) in the tournament we check whether both participating crabs are alive. If not the game is forfeited, if yes we can define the probability of each crab winning. It is dependent of the mass as it is defined it assignment. In each game we let the first crab win with the calculated probability, and update the:

  – status array: the status of a dead crab is changed to zero

  – mass array: the mass of the dead crab is transferred to the mass of the winning crab, so the winning crab will have a larger mass and the dead crab will have a mass of zero.

  Both the status array and the mass array are returned from this function.

- `battle_simulation` accepts the starting Cartesian positions of the crabs and the number of days to simulate. This functions loops over the days to simulate and computes the tournament and let the crabs battle. After each tournament of battles (each night) positions, mass and status of dead crabs are removed so the new length of the arrays are the number of alive crabs. In the end it returns the mass array of the remaining crabs.

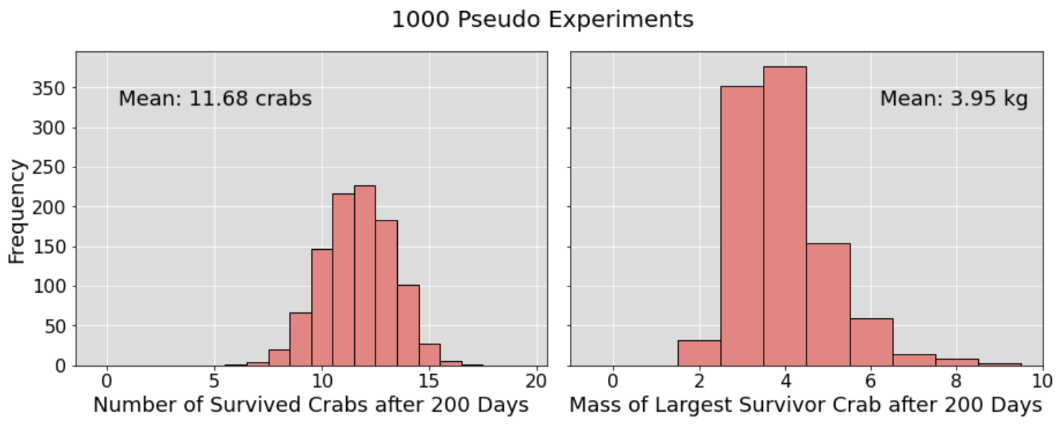


Figure 6

We put these functions together to find the most likely number of survived crabs and most likely mass of the largest surviving crab after 200 days. The number of surviving crabs for each simulation is found as the length of the returned mass array and the mass of the largest surviving crab is found as the largest number in the array. We perform the simulation 1000 times, and the resulting distributions are displayed in Fig. 6. The most likely number of surviving crabs (after 200 days) is found as the mean of the distribution:

$$\langle N_{survivors} \rangle = 11.68\text{crabs} \approx 12\text{crabs}$$

The most likely mass of the largest surviving crab is similarly found as the mean of the distribution and is:

$$\langle \max(m_{survivors}) \rangle = 3.95\text{kg} \approx 4\text{kg}$$

3.4 (3D)

To solve the problem we need to modify the function `battle_simulation` from problem 3C a little bit, and we call this new function `battle_until_N_survivors`. It takes the positions just like before, but now also the argument `N_surv` defining how many remaining crabs we are interested in. It runs the simulation similar to before, but now keeps count of the days simulated as well. In each iteration of the loop it also checks how many surviving crabs that are left, if this number equals `N_surv` the loop breaks and the number of days to obtain this result is returned.

We run this experiment 1000 times, with `N_surv=10` and note the number of days each experiment took. The distribution of the number of days to reduce the crab population from 20 to 10 is plotted in the left panel of Fig. 7. To find the 1σ confidence interval on the number of days at which only 10 crabs remain, we first convert the distribution into a CDF as is displayed in the right panel in Fig. 7. We can now interpolate a function $X(Y)$ from this, which maps the fraction of data to the number of days. Since we want the 1σ interval, the lower confidence level we are interested in is given by $(1 - 0.6827)/2$ and the upper confidence level is accordingly given by $(1 + 0.6827)/2$. We then use the interpolated function to find the number of days associated with those confidence level. **The 1σ confidence region is between 183days and 364days.**

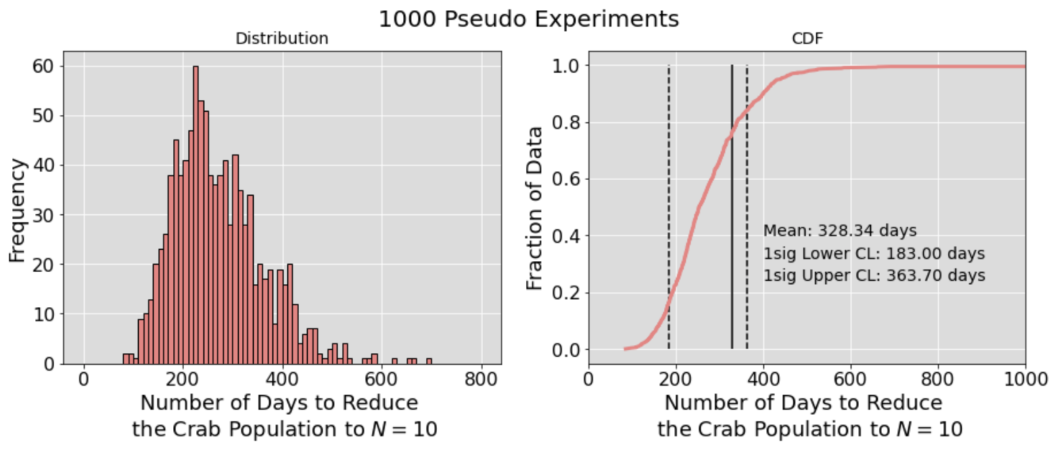


Figure 7