Lab #5 Report: Localization

Team 14

Michael Lu
Enrique Montas
Jon Stenger
Grey Sarmiento

6.141 Robotics: Science and Systems

April 4, 2022

1 Introduction (Jon S.)

Monte Carlo Localization (MCL) is a very useful and important topic in robotics. Localization allows the robot to determine its pose in an environment in which a map is already given. This process is similar to trying to find one's own location in a city. When given a map as a reference, the person will use other locations on the map to try and determine where they are.

In this lab, five main tasks were completed including: implementing a motion model, sensor model, and particle filter, and finally testing the particle filter for both the simulation as well as on the physical racecar.

At a high level, the process is a repeated cycle as follows: prediction, update, and resampling. In the prediction phase, many particles will try to best estimate the location of the robot in which noise is also added. Next, in the update phase each particle hypothesis is compared with the ground truth distance to allow each particle to be assigned a probability that it is the correct location. Finally, in the resampling phase only the particles that have a high enough probability are used and the rest are excluded. This process is repeated at each time step and eventually the particles will converge to a location that should match up to the ground truth location.

2 Technical Approach (Michael L., Grey S., Enrique M.)

To approach the technical challenge of implementing Monte Carlo localization, three software modules were developed to handle different components of the localization algorithm.

1. particle_filter.py: Main module that implemented Monte Carlo localization on a set of generated particles.
2. motion_model.py: A module that updated the pose of particles with some additional noise (if specified) given odometry data from the robot's sensors.
3. sensor_model.py: A module that calculated the likelihood of recording a particle given its sensor measurements.

2.1 Motion Model (Grey S., Jon S.)

The motion model estimates the car's position over time by integrating odometry data to calculate how much position has changed since initialization. At a high level, the model initializes several particles which each represent an estimate of the robot's current pose. As odometry data is provided at each time step, each particle is updated to reflect the measured pose change with a small amount of noise to account for measurement error.

2.1.1 Particle Pose Transformation

Each particle is updated by transforming the initial pose, in the world frame, by the transformation set by the odometry, which is in the robot frame. From time t-1 to time t, this calculation can be expressed as follows:

$$P_{\text{world}}^{\text{t-1}} T_{\text{t-1}}^{\text{t}} = P_{\text{world}}^{\text{t}} \tag{1}$$

$$\begin{bmatrix} cos\theta_{t-1} & -sin\theta_{t-1} & x_{t-1} \\ sin\theta_{t-1} & cos\theta_{t-1} & y_{t-1} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} cos\theta_{t-1} & -sin\theta_{t-1} & \Delta x \\ sin\theta_{t-1} & cos\theta_{t-1} & \Delta y \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} cos\theta_{t-1} & -sin\theta_{t-1} & x_t \\ sin\theta_{t-1} & cos\theta_{t-1} & y_t \\ 0 & 0 & 1 \end{bmatrix}$$

Here, P represents the positions of the particles before and after update, and T represents the transformation obtained by the odometry data. All poses in this calculation are represented by 3x3 pose matrices. These matrices were obtained and transformed for the entire set of N particles through the use of NumPy arrays.

The final pose of each particle is then:

$$\begin{pmatrix} x_t \\ y_t \\ \theta + \Delta\theta \end{pmatrix}$$

2.1.2 Added Noise

Because odometry measurements in the real world are not perfect, one particle that estimates pose based on broadcasted odometry will not be sufficient to accurately predict location, especially as error accrues over time. To counteract this, a small random noise term is added to each particle so that a group of distributed particles are more likely to accurately approximate location. Over time, these particles will spread out as they become more noisy, and will be resampled by the sensor model in the final localization algorithm.

A different amount of noise was added to each odometry reading per time stamp per particle based on two different random sampling schemes: uniform random sampling within a coefficient, and Gaussian sampling within a set standard deviation. These experiments are covered in more detail in Section 3.

2.2 Sensor Model (Michael L.)

2.2.1 Probabilistic Model for Sensor Data

The sensor model was responsible for determining how likely it is to record a given sensor reading and ground truth distance from a hypothesis position in a known, static map at a particular time. The results of sensor model are used to filter through the particles as the motion model tries to spread them out. Particles will be kept if the sensor reading is consistent with the map from the point of view of that particle.

The likelihood of a scan was represented as $p(z_k^{(i)}|x_k, m, d)$, where z_k was the sensor reading, x_k was the hypothesis position, m was the static map, k was the time, and d was the ground-truth distance. The likelihood of a scan was computed as the product of the four likelihoods of each of range measurements in the scan:

1. The probability of detecting a known obstacle in the map, represented as a Gaussian distribution centered around the ground truth distance d:

$$p_{hit} = p_{hit}(z_k^{(i)}, d|x_k, m) = \begin{cases} \eta \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(z_k^{(i)}-d)^2}{2\sigma^2}\right) & \text{if} \quad 0 \leq z_k \leq z_{max} \\ 0 & \text{otherwise} \end{cases}$$

2. The probability of detecting a short measurement, possible due to internal lidar reflections such as the vehicle itself or unknown obstacles, represented as a downward sloping line:

$$p_{short}\left(z_k^{(i)}, d|x_k, m\right) = \frac{2}{d} \begin{cases} 1 - \frac{z_k^{(i)}}{d} & \text{if} \quad 0 \leq z_k^{(i)} \leq d \text{ and} d \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

3. The probability of a large/missed measurement because a lidar beam reflects off an object and does not return to the sensor due to strange reflective probabilities, represented by an approximation of the delta function $\delta(z_k^{(i)} - z_{max})$:

$$p_{max}(z_k^{(i)}, d|x_k, m) = \begin{cases} \frac{1}{\epsilon} & \text{if} \quad z_{max} - \epsilon \leq z_k^{(i)} \leq z_{max} \\ 0 & \text{otherwise} \end{cases}$$

4. The probability of detecting a completely random event, represented by a small uniform value:

$$p_{rand}(z_k^{(i)}, d|x_k, m) = \begin{cases} \frac{1}{z_{max}} & \text{if} \quad 0 \leq z_k^{(i)} \leq z_{max} \\ 0 & \text{otherwise} \end{cases}$$

These probabilities were multiplied by respective normalized coefficients $a_{hit}, a_{short}, a_{max}, a_{rand}$ (coefficients sum to 1) to weight their values to produce a "weighted average." Thus, the final probability distribution becomes:

$$p(z_k^{(i)}, d|x_k, m) = \alpha_{hit} \cdot p_{hit}(z_k^{(i)}, d|x_k, m) + \alpha_{short} \cdot p_{short}(z_k^{(i)}, d|x_k, m)$$

$$+\alpha_{max} \cdot p_{max}(z_k^{(i)}, d|x_k, m) + \alpha_{rand} \cdot p_{rand}(z_k^{(i)}, d|x_k, m)$$

2.2.2 Precomputing the Probabilistic Model

To implement the probabilistic model for sensor data in software, a few adjustments were necessary to improve its performance.

One improvement involved first discretizing all values of z_k and d to be integer values between 0 and 201 pixels. This finite set of possible values then allowed the sensor model to precompute all probability values for all possible measured distances z_k and ground truth distances d, creating a probability table of size $z_{max} = 201$ by $d_{max} = 201$. Computing this table once on initialization of the sensor model would prevent the need for reeavluating probability values for every new measurement. When discretized, the function p_{max} had $\epsilon = 1$ or 1 pixel, so it simplified to:

$$p_{max}(z_k^{(i)}|x_k, m) = \begin{cases} 1 & \text{if} \quad z_k^{(i)} = z_{max} \\ 0 & \text{otherwise} \end{cases}$$

Another improvement involved normalizing the $p(z_k^{(i)}, d|x_k, m)$ function, first normalizing all the precomputed p_{hit} values across columns of d, then normalizing the combined probability values across columns of d. This normalization ensured all probabilities would sum to 1 to be more accurate.

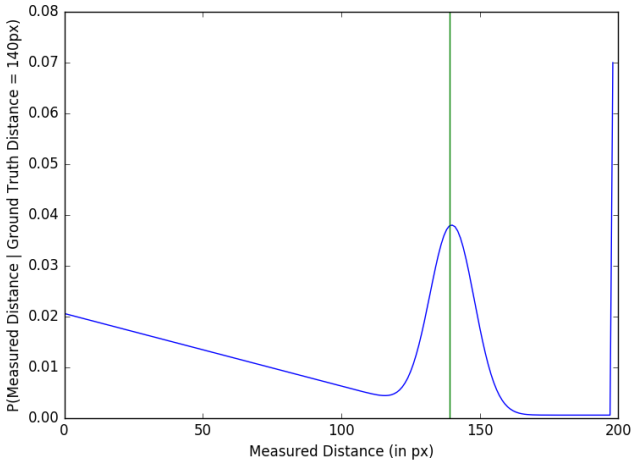

Figure 1: Distribution of scan probabilities for measured distances. The distribution is nonuniform and has changes in shape due to the piecewise nature of $p_{hit}, p_{short}, p_{max}$, and p_{rand} and each function becoming nonzero at different conditions and thus determining the shape of the distribution. From MIT RSS 2022 Lab 5 Localization (https://github.com/mit-rss/localization/blob/master/README.ipynb)

With a precomputed table of sensor value probabilities, the sensor model could quickly access the corresponding probability upon receiving a given measured distance and ground truth distance without having to compute all these probabilities. Leveraging this table would improve the sensor model's performance, allowing it to smoothly integrate into the larger implementation of the Monte Carlo Localization algorithm.

$p(1,1)$	$p(1,2)$	$\ldots$	$p(1,d_{max})$
$p(2,1)$	$p(2,2)$	$\ldots$	$p(2,d_{max})$
$\ldots$	$\ldots$	$\ldots$	$\ldots$
$p(z_{max},1)$	$p(z_{max},2)$	$\ldots$	$p(z_{max},d_{max})$

Figure 2: Structure of precomputed scan probabilites table. For each measured distance $z_k^{(i)}$ and ground truth distance $d^{(j)}$, a scan probability $p(z_k^{(i)}, d^{(j)}|x_k, m)$ can be precomputed as an entry in the table at the i-th column and j-th column.

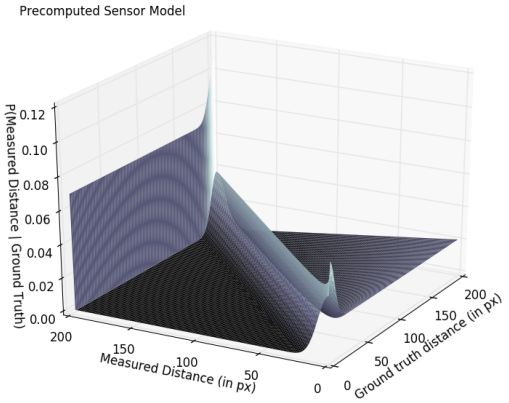

Figure 3: Distribution of scan probabilities for measured distances and ground truth distances with $\alpha_{hit} = 0.74, \alpha_{short} = 0.07, \alpha_{rand} = 0.12$, and $\sigma_{hit} = 8.0$. For each measured distance and ground truth distance, a scan probability can be precomputed, producing a tree-dimensional distribution that can be stored in a precomputed probability table. From MIT RSS 2022 Lab 5 Localization (https://github.com/mit-rss/localization/blob/master/README.ipynb)

2.2.3 Generating Particle Scores

With the precomputed scan probability table, the particle filter could efficiently calculate how likely each particle would be given an observed scan at time k. Given a list of n particle poses, each pose containing a particle's x, y, and θ, the sensor model would first use ray casting to generate a simulated list of lidar output values of $z_k^{(i)}$ for each particle.

$$\begin{bmatrix} x_1 & y_1 & \theta_1 \\ x_2 & y_2 & \theta_2 \\ \dots & \dots & \dots \\ x_n & y_n & \theta_n \end{bmatrix} \rightarrow \begin{bmatrix} z_1^{(1)} & z_1^{(2)} & \dots & z_1^{(m)} \\ z_2^{(1)} & z_2^{(2)} & \dots & z_2^{(m)} \\ \dots & \dots & \dots & \\ z_n^{(1)} & z_n^{(2)} & \dots & z_n^{(m)} \end{bmatrix}$$

Figure 4: Generated particle lidar scans from ray casting. For each of the n particles, the sensor model used ray casting to generate m lidar readings. These readings will be used to calculate particle probabilities.

Using this generated data, the sensor model would convert the lidar data from meters to discretized integer pixel values by dividing by the product of the simulation map resolution and the ratio of the lidar values scale and simulation map scale. Following the conversion, all discretized values would be clipped to

be between 0 and 201, the maximum pixel value.
Then, the sensor model would then go through each particle and generate a particle score $P(z, d)$ for resampling leveraging the precomputed probability table to pick out $p(z_k^{(i)}, d^{(j)}|x_k, m)$ as described in the previous section. For the i-th particle, the particle score was the product of the values in the precomputed probability table for each generated lidar scan point $z_i^{(j)}$ and measured distance from the actual lidar scan $d^{(j)}$. After this product was computed, it was raised to the power of $\frac{1.0}{2.2}$ to squash the probability score.

$$\begin{bmatrix} P_1 \\ P_2 \\ \dots \\ P_n \end{bmatrix}$$

where

$$P_i(z_i^{(j)}, d^{(j)}) = (\prod_{j=1}^{m} p(z_k^{(i)}, d^{(j)}|x_k, m))^{\frac{1.0}{2.2}}$$

and

$$\sum_{i=1}^{n} P_i = 1$$

A particle with a higher score was more likely to be used by the particle filter localization algorithm than those with a lower score when resampling particles.

2.3 Particle Filter (Enrique M.)

2.3.1 Combining Motion and Sensor Models

Monte Carlo Localization uses both odometry and sensor data to determine the pose of a moving robot. As the robot moves, both odometry and sensor data is published. When a message with odometry data is published, use the motion model to update particle positions. When a message with scan data is published use the sensor model to compute the likelyhood of each particle and resample to preserve the most likely positions. After updating the particles a circular average is taken and the average is published as the pose estimate.

2.3.2 Odometry Callback

When the robot publishes an odometry message the `odom_callback` is triggered. If there is no map or no previous time step the callback returns none. Otherwise, the twists of the x, y, and θ directions are taken; this is the velocities in the x, y, and θ direction respectfully. To go from these velocities to positions multiply the velocity by the elapsed time since the previous odometry message. After converting from velocities to positions the evaluate function of the motion model called to update active particles.

2.3.3 Lidar Callback

When the robot publishes a LIDAR message the `lidar_callback` is triggered. IF there is no map the callback returns none. Otherwise, the evaluate method of the sensor model is called and the current particles are scored. After scoring the particles, the scores are normalized and used to resample the existing particles.

$$Given \begin{bmatrix} P_1 = 0.3 \\ P_2 = 0.6 \\ P_3 = 0.07 \\ P_4 = 0.03 \end{bmatrix} Poses = \begin{bmatrix} x_1 & y_1 & \theta_1 \\ x_2 & y_2 & \theta_2 \\ x_3 & y_3 & \theta_3 \\ x_4 & y_4 & \theta_4 \end{bmatrix} \rightarrow \begin{bmatrix} x_1 & y_1 & \theta_1 \\ x_2 & y_2 & \theta_2 \\ x_2 & y_2 & \theta_2 \\ x_1 & y_1 & \theta_1 \end{bmatrix},$$

Figure 5: Particle Filter Resampling Example. Given four normalized particle scores, particles 1 and 2 have higher scores from the sensor model than particles 3 and 4, therefore the particle filter resampling algorithm is more likely to choose them.

2.3.4 Determining Pose from Particles

After the particles are updated and resampled the average is taken to estimate the pose of the robot. Since the particles are listed with an x, y and theta component a simple average doesn't suffice, rather the circular average is computed.

$$\texttt{circular\_average} = \frac{1}{n} \sum_{i=1}^{n} \texttt{particle}$$

To find the average theta, split into an x and y component then reconstructed using arctan.

$$\texttt{theta\_average} = \tan^{-1}\left(\frac{\texttt{average\_theta\_y}}{\texttt{average\_theta\_x}}\right)$$

After finding the average pose, it is transformed from Eüler to Quaternion to transform the pose to the world frame. After converting to a quaternion it is published.

2.3.5 Particle Filter Pseudocode

Algorithm 1: Particle Filter

Data: odometry or scan
Result: updates to particles or pose
motion model $\leftarrow$ *odometry*
sensor model $\leftarrow$ *scan*
particles $\leftarrow$ *resample*(*particles*)
publisher $\leftarrow$ world view

2.3.6 Real World Implementation

After the particle filter was functional for the simulator the team focused on localization for the real world driving robot. Since localizing in real time durring the drive was met with technical difficulties, to remedy this while driving the robot a rosbag recorded the odometry and lidar information. By replaying the rosbag on a simulation starting in the same point (estimate) the localization is visualized. The main adjustments of using the real world robot are updating the "odom" topic from "/odom" to "/vesc/odom" and downsampling the LIDAR data. Changing the `odom` topic was a quick change to `params.yaml`. To downsample the LIDAR data a check was added to the `lidar_callback`. The check ensures the number of beams equals the `num_beams_per_particle` parameter also found in `params.yaml`, if there are too many beams they're sampled down. (Note about how the area where we tested is slightly different for the map and the current features. Could add to any error that we are finding.)

3 Experimental Evaluation (Grey S.)

3.1 Noise Coefficient Experiments

Because there is real world noise in odometry data, noise must be added to motion model location estimations so that the certainty of the estimate is captured over time by particle distribution. This was done using two different random noise sampling schemes: uniform random sampling and Gaussian distribution sampling.

It was found that although random noise performs well at small noise coefficients, Gaussian noise performs better overall and provides a more balanced noise profile as variance increases.

3.1.1 Setup

To assess the performance of different noise sampling schemes, both random sampling and Gaussian distribution sampling were implemented on the particle filter with various coefficients. For random sampling, the range of values (where sampled x was between -c and c) was varied. For Gaussian sampling, the standard deviation (σ) was varied. The following values were tested as summarized in the table. Values were chosen to capture both linear variation and exponential variation by factors of 10.

Random Sampling c	Gaussian σ
0.01	0.01
0.1	0.1
0.5	0.3
1.0	0.5
	1.0

To most comprehensively assess noise performance, ground truth and particle filter results for robot location were needed to compare error. The particle filter was run in simulation on a segment of the Stata Basement map with each set of noise parameters, and the ROS tool `rosbag` was used to capture the `tf` tree of the simulation over time. This included both the ground truth transform from the world frame to the robot and the estimate by the particle filter, represented as the transform from the world frame to the average particle frame.

From here, the bag files were played back along with a node that looked up the ground truth and predicted positions and published the Cartesian difference between them. This distance error was plotted over time with `rqt plot`.

3.1.2 Results

Although random noise sampling achieved the lowest error result, Gaussian noise sampling was found to maintain accuracy as noise increased. All error graphs can be seen at the very end of the report, in Section 6.

Randomly sampled noise seems to result in relatively steady error over time, with the exception of c=1.0 meters, which seems to be too noisy to accurately capture the location estimate. Random noise at a very low coefficient of 0.01 resulted in the lowest error for this set of trials. With too much noise, the sensor model is unable to reduce the particles back down and noise accrues over time, as seen in Figure 9.

In contrast, Gaussian sampled noise performed relatively well even as σ increased, but was slightly more noisy at smaller standard deviations. As seen in figures 12 and 13, although larger noise led to larger error, noise was more likely to "balance" itself out and keep error much lower than random sampling. In Figure 13, it becomes apparent that even with significant noise the particle filter was able to improve its accuracy over time, and error decreases toward the right of the graph.

3.2 Options for Further Evaluation

It should be noted that it is difficult to evaluate random noise and Gaussian noise exactly side by side because of the difference in the ways they are calculated - for example, a variance of 0.01 meters for random noise does not translate to a σ of 0.01 for Gaussian noise. This translation could be more rigorously defined.

There is still room to evaluate these types of noise on different maps or different obstacle sets. This evaluation was performed on a hallway, which, although it was asymmetrical, was still relatively uniform. Evaluation on corners or other more cluttered scenes may be elucidating.

4 Conclusion (Jon S.)

Overall, our team was able to successfully implement a particle filter for localization on our racecar. The system was tested with various experiments using different noise sampling as well as multiple noise coefficients. This lab will be very important and a big foundation for the next path planning lab.

In the future it will be useful to be able to implement real time localization on the racecar. Another point to keep in mind will be being more aware of issues that can come up in localization due to symmetry. Although it was not a problem for this lab it will be good to be more aware in case it does happen. The last feature that might be needed in the future is the use of threading with the motion and sensor model working simultaneously.

5 Lessons Learned

5.1 Jon Stenger

This lab helped us to learn all about the challenges of implementing a particle filter for MCL. We had to create our own experiments that were meaningful and showed that our algorithm was working as expected. In this lab we also had to help each other out to figure out bugs along the way, as there were many things that were very subtle at first glance. We also had to spend much more time working with the simulation, as it was also much easier to visualize what was happening there related to the particles and laser scans. This in turn allowed to spend less time overall working on the physical robot, as our implementation was working better by then.

5.2 Michael Lu

This lab helped us sharpen our skills working with ROS, especially in real life, where many simulation tools are not available. We learned how to test, debug, and validate our software without the convenience of simulation visualizers and simulation test cases. We also refined our abilities to record real life data and play it in simulation through the use of rosbags and rviz as well as to generate error graphs. This ability to analyze real-time data was crucial in debugging issues with our localization algorithm.
Finally, this lab provided an opportunity to collaborate with others in a technical and interpersonal setting, learning how to code collaboratively and determine logistics for working together. This was especially important for this lab, where the modularity of software was conducive to dividing work among group members and learning how to safely combine all components for a functioning final product.

5.3 Enrique Montas

This lab gave me more experience using ROS, particularly working with launch files, bags and simulations. We needed to get creative in working around bugs such as the import simulator. Since we used simulations and rviz to evaluate the localization on the robot it helped me make the connection between the robot in simulation and the robot in the real world. Working with the team on this lab was good, I saw more comfort in our group as a whole when compared to lab 3.

5.4 Grey Sarmiento

This lab made us much more comfortable switching between bag files, simulation, and the real world. I feel like we are becoming more smooth in working as a team to split up work and debug each other's things, and we are also considerably faster at debugging and effectively using git branches to keep our work clean and organized.

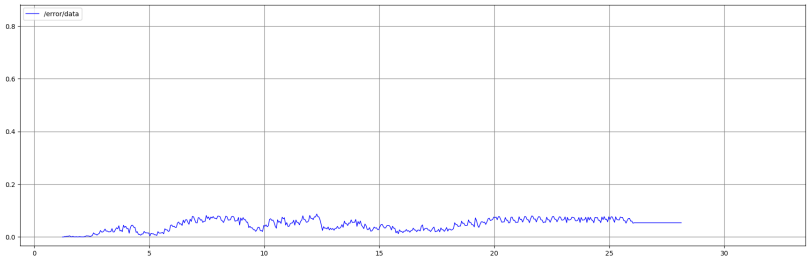

Figure 6: c=0.01

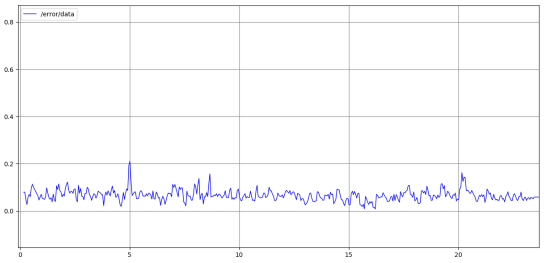

Figure 7: c=0.1

6 Error Graphs

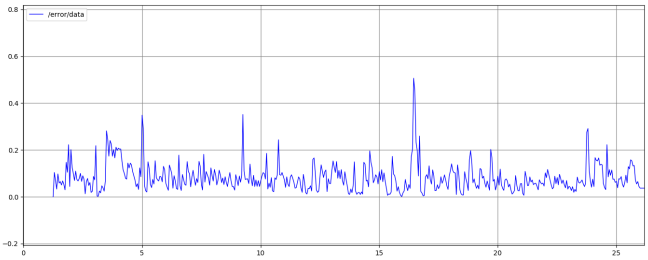

Figure 8: c=0.3

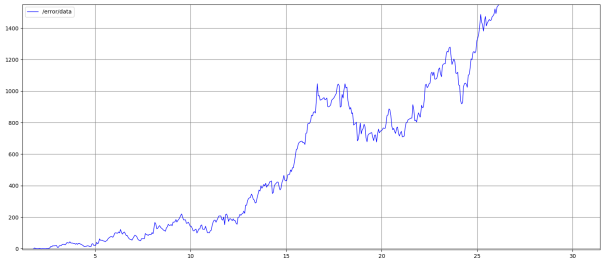

Figure 9: c=1.0

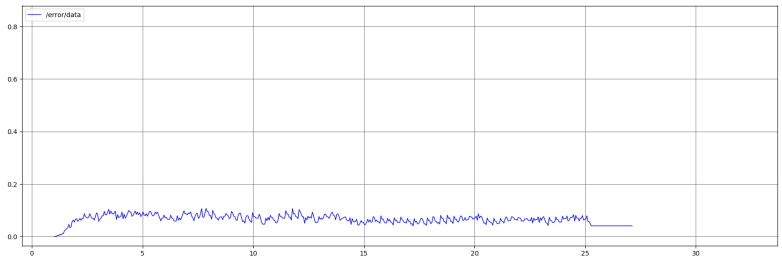

Figure 10: $\sigma = 0.01$

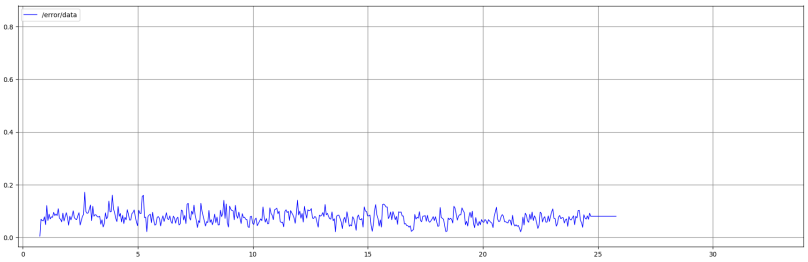

Figure 11: $\sigma = 0.1$

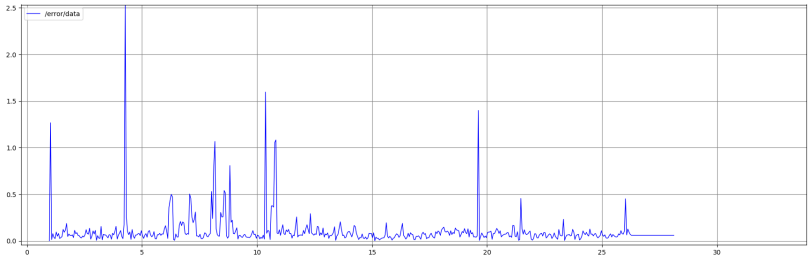

Figure 12: $\sigma = 0.5$

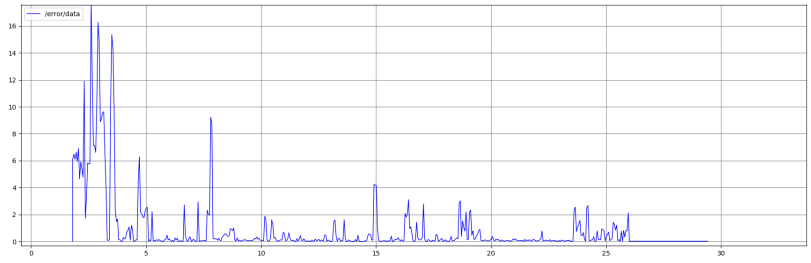

Figure 13: $\sigma = 1.0$