MECH_ENG 469: ML & AI In Robotics

Report 1: Search and Navigation

A* Implementation

Michael Jenz

10-27-2025

Table of Contents

1 Part A

1.1 Learning Aim

This course has focused on the implementation of robotics algorithms on the data for a differentially wheeled robot. In homework 0, the Kalman filter algorithm used a state estimation method known as dead reckoning to guess the next state of the robot given the input velocity v and angular velocity ω commands, as shown in 1. In homework 0 we saw how poorly this motion model performed on its own. Dead reckoning is unable to accurately model the robots behavior because it does not account for the real world interaction of the robot. This means that aspects of robot movement such as wheel slip, motor acceleration, or even inertia are not accurately accounted for. Thus, I was motivated to use machine learning algorithms to build a better motion model that better represents the robots motion through space.

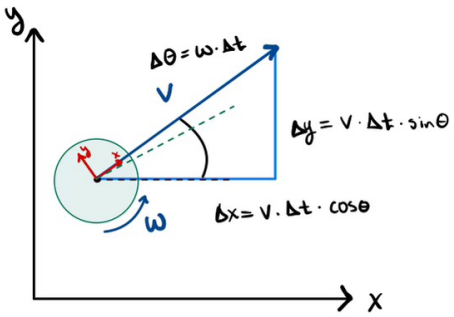

Figure 1: Configuration of mobile differential wheel robot in 2-D space, naming input velocities and changes in robot state.

As a baseline, for a complete run of the velocity commands from dataset 1, we can see in Figure 2 that the positional error of the dead reckoning motion model is quite high, averaging a positional error of $[3.45, 4.23, 1.71]$ for $[x_{error}, y_{error}, \theta_{error}]$ respectively. This figure shows just how poorly the dead reckoning algorithm performs on its own and suggests that there is great room for improvement in modeling the motion of the robot. Another qualitative measure of the motion models performance is how the shape matches that of the ground truth path. It is clear that the dead reckoning algorithm is unable to maintain even the general shape of the robots movement. This presents another area for improvement.

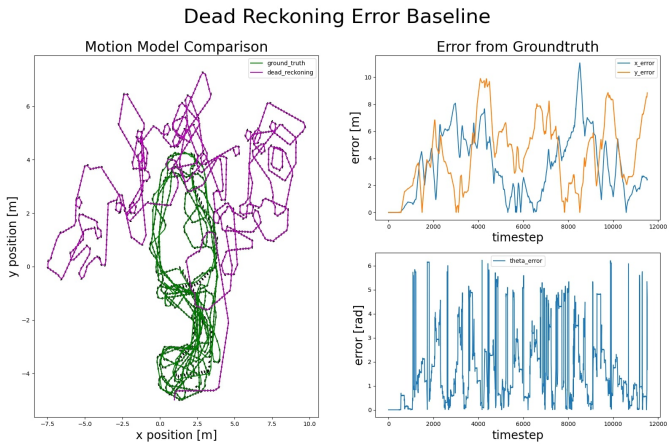

Figure 2: Error (left) and path (right) results from dead reckoning motion model used on data set 1.

1.2 Algorithm Choice

I chose the Locally Weighted Linear Regression (LWL) because of its simplicity and elegance. Furthermore, it aligns closely with my learning aim of improving upon the dead reckoning model. For this algorithm to work, it requires that there is a locally linear relationship between the inputs and the outputs. Originally in my proposal, I suggested using only the raw inputs of robot velocity commands $[v, \omega]$. This I discovered is not sufficient data for the LWL to learn. As I will explain in detail later, I decided to give the model more information on the robot state, which means information on its heading θ and the time step duration for which the velocity commands are given. Giving these inputs separately to a linear algorithm is not acceptable, since there is a multiplicative relationship between the elements. Therefore, I decided to linearize my inputs by pre-multiplying them together. Thus, giving the algorithm the output of the dead reckoning motion model as its input. This formulation is valid for the LWL because there is a linear relationship between these inputs $[\Delta t * v * cos(\theta), \Delta t * v * sin(\theta), \Delta t * w]$ and the outputs $[\Delta x, \Delta y, \Delta \theta]$. To visually understand this linear relationship, see Figure 1. As discussed in homework 0, the inputs of the dead reckoning model themselves are non-linear because of the trigonometry terms in the translational components. However, using the dead reckoning motion model before applying our machine learning algorithm, we essentially linearize the relationship between input commands and the change in robot position. Therefore, the role of the LWL is to learn the real world interaction of the robot and its surroundings, to help account for noise, slippage, etc.

1.3 Learning Problem

As previewed above, the goal of my implementation of the LWL is to learn the real world dynamics of the robot and correct for these inconsistencies in the dead reckoning algorithm. As inputs, I will use the outputs of the dead reckoning algorithm, and as outputs I will get the true change in state of the robot. These are described below in

Equations 1- 5.

$$in_1 = \Delta t * v * cos(\theta) \tag{1}$$

$$in_2 = \Delta t * v * sin(\theta) \tag{2}$$

$$in_3 = \Delta t * \omega \tag{3}$$

$$in = [in_1, in_2, in_3] \tag{4}$$

$$out = [\Delta x, \Delta y, \Delta \theta] \tag{5}$$

In order for my algorithm to learn this relationship, I needed to build a data set with the transformed outputs and a corresponding set of ground truth locations. This data set building is performed by the function create_dataset(). It transforms the raw input commands $[v, \omega]$ into the dead reckoning function space and saves the new dataset, along with the corresponding time-stamps, to the "learning_dataset_controls_ds1.npy" file. Next, it generates the ground truth training set. Since the controls and ground truth data are not collected at the same times, I could not use the raw ground truth file to train my machine learning model. Therefore, I linearly interpolated the ground truth data and estimated the robot position at the timestamps of the controller input data set. I then saved this new ground truth dataset to a file named "learning_dataset_gnd_truth_ds1.npy". These files are in .npy form because it is optimized for quick use by the Numpy Python package.

Furthermore, I separated my training and validation data in the traditional 80-20% split. So, 80% of dataset 1 is used as training points for the algorithm, and the algorithm is tested on the other 20%. Since it is important that the validation data are sequential, I tested the algorithm on three different data partitions. Partition 1 is where the back 20% of the data is used for validation, partition 2 is where the front 20% is used, and partition 3 is where the middle 20% is used.

The only parameter that needs to be tuned in my algorithm is the τ parameter which is defined below as the bandwidth parameter. I tuned this parameter by hand by examining the algorithm performance given various τ values within the reasonable range of [0.01, 0.5]. Generally, I found that using a small τ works best for this formulation of the problem, so I used $\tau = 0.01$.

1.4 Algorithm Code

The LWL algorithm functions by minimizing the cost weighted cost function as defined in Equation 6 below. This equation captures the essence of the training for the LWL algorithm. The weights w are the local weights, calculated in Equation 7 that are directly related to the distance of a given training point $x^i$from the query point x_q. I used the gaussian kernel function to define these weights, which means that there is an additional parameter τ called the bandwidth parameter. This parameter chooses the sort of slope with which the weight decreases as the points get

farther from the query point. A higher τ will allow for a wider range of points to have non-zero weights, but a low τ will only consider close points. The θ term in the cost function is different from the heading variable of the robot. Instead, these are the global weights for the model that are learned. In fact, we can analytically solve for the value of these weights using Equation 8. Then, once we have these learned weights, we can make a prediction $\hat{y}$ for the output of a given query point. This algorithm is a lazy learning algorithm, so nothing is retained from each learning session. The algorithm learns a new set of weights w and θ for each query point x_q [1].

$$J(\theta) = \sum_{i=1}^{m} w^i(\theta^T x^i - y^i)^2 \tag{6}$$

$$w^i = exp(\frac{-(x^i - x_q)^2}{2\tau^2}) \tag{7}$$

$$\theta = (X^T W X)^{-1} X^T W y \tag{8}$$

$$\hat{y} = \theta^T x_q \tag{9}$$

$$\tag{10}$$

In code, my algorithm functions inside a simulation of the robot motion. It is called to update the robot state, much in the same way that the dead reckoning algorithm is called. First, I must initialize my class with the appropriate training data. Then, I simulate the robots movement by passing in the sequential list of control inputs from the validation data set. The output is recorded and then the simulated route of the robot is displayed, as predicted by the learned robot model.

1.5 Algorithm Simple Test

Before deploying my algorithm on the stated learning problem, I first wanted to prove a basic version of this algorithm. So, I tested the LWL that I wrote on a data set of noisy sin data. I generated this data set by creating a list of randomly chosen points centered around a sin function. Then, I gave this data set to the LWL as training data and asked it to predict the value of the function at given query points. See in Figure 3 below how the LWL is able to accurately estimate the true value of the sin function based on the data points surrounding the query point. Another important note is how the LWL is only able to estimate the true sin function where there is training data present. The lazy learning algorithm cannot extrapolate outside the bounds of the dataset it is given.

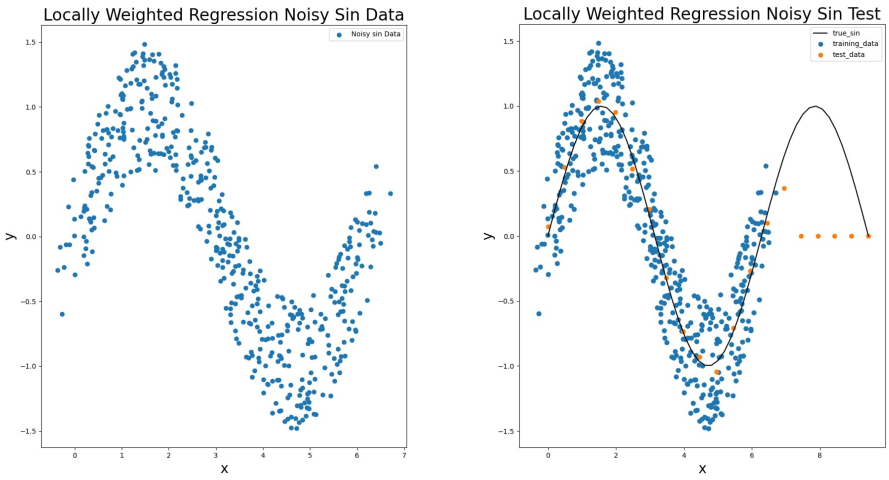

Figure 3: Test of Locally Weighted Linear Regression on a noisy sin data set.

2 Part B

2.1 Introduction

This section provides my results for testing and evaluating the performance of the LWL algorithm on the real data set. To evaluate, I examine the positional error of the LWL motion model over time. The average of this positional error is also taken to give a quantitative measure of this error. Furthermore, I visually compare the preservation of shape in the LWL motion model to that of the ground truth. To give all of these measurements a sort of 'control' to compare to (beyond that of the ground truth), I performed all these tests on the dead reckoning only motion model. Finally, despite it not being a completely linear relationship, I completed these tests on the raw input problem formulation, as defined below in Equation 11- 17 to see if the LWL would learn a better controller than the dead reckoning framework I provided to my original LWL formulation. Together, these measures provide a clear picture of the performance of my LWL formulation.

$$in_1 = \Delta t \tag{11}$$

$$in_2 = v \tag{12}$$

$$in_3 = \omega \tag{13}$$

$$in_4 = cos(\theta) \tag{14}$$

$$in_5 = sin(\theta) \tag{15}$$

$$in = [in_1, in_2, in_3, in_4, in_5] \tag{16}$$

$$out = [\Delta x, \Delta y, \Delta\theta] \tag{17}$$

2.2 Results

The results for the LWL with dead reckoning input formulation can be seen in Figure 4. In these results, we can already see that the learned motion model does not fully account for all external factors. It does not match the ground truth trajectory. However, it does preserve the shape of the ground truth trajectory remarkably well. Without any comparison yet, it is important to note that the heading or θ value predicted by the algorithm is critical. If the heading gets slightly off, then the LWL motion model will be unable to return to the ground truth path. This is because the motion model is trained on the ground truth headings, thus, it assumes that it is pointed in the correct direction. This is one key reason why the shape of the trajectory generally matches, but it is almost translated in space.

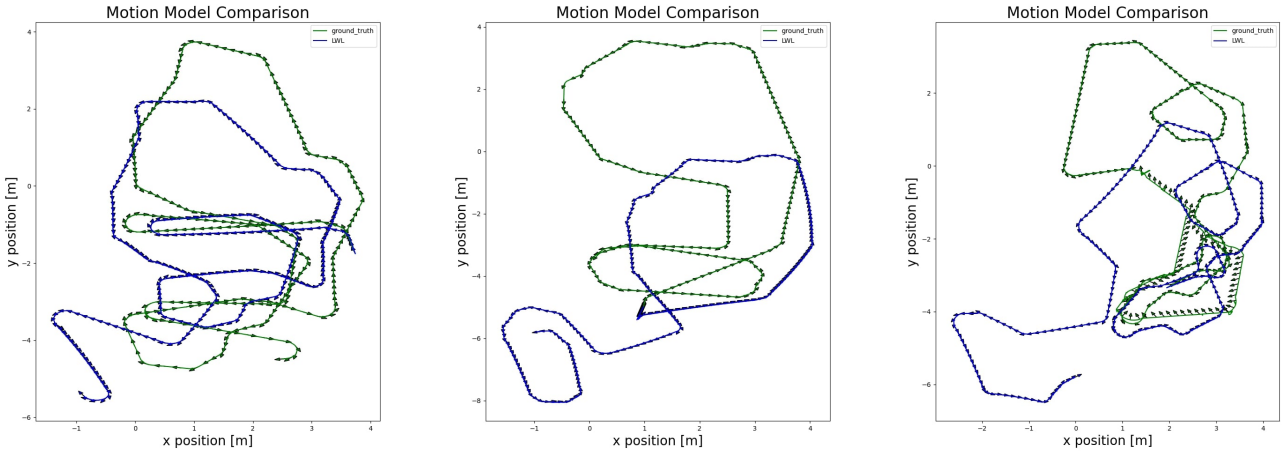

Figure 4: Test of Locally Weighted Linear Regression on real data, from right to left data partition 1, 2, and then 3.

2.3 Performance Evaluation

2.3.1 Positional Error Over Time

The results of the positional error over time can be seen in Figure 5. These results can be slightly misleading because the scales between the dead reckoning and the LWL errors are not unified. If you look closely, the scale of the positional error is consistently much higher on the dead reckoning results compared to the LWL positional error. Furthermore, since the heading error scale is unified between the two motion models, it is clear that the learned motion model does a much better job of tracking the robot heading over time. In both the dead reckoning and learned motion models, the error goes up over time since the robot loses its tracking of the ground truth path more and more over time. Finally, we can compare the shape of the outputs between the dead reckoning results and the learned motion model. It is clear that the learned motion model is capable of preserving the shape of the ground truth path, while the dead reckoning model is almost unrecognizable compared to the ground truth path.

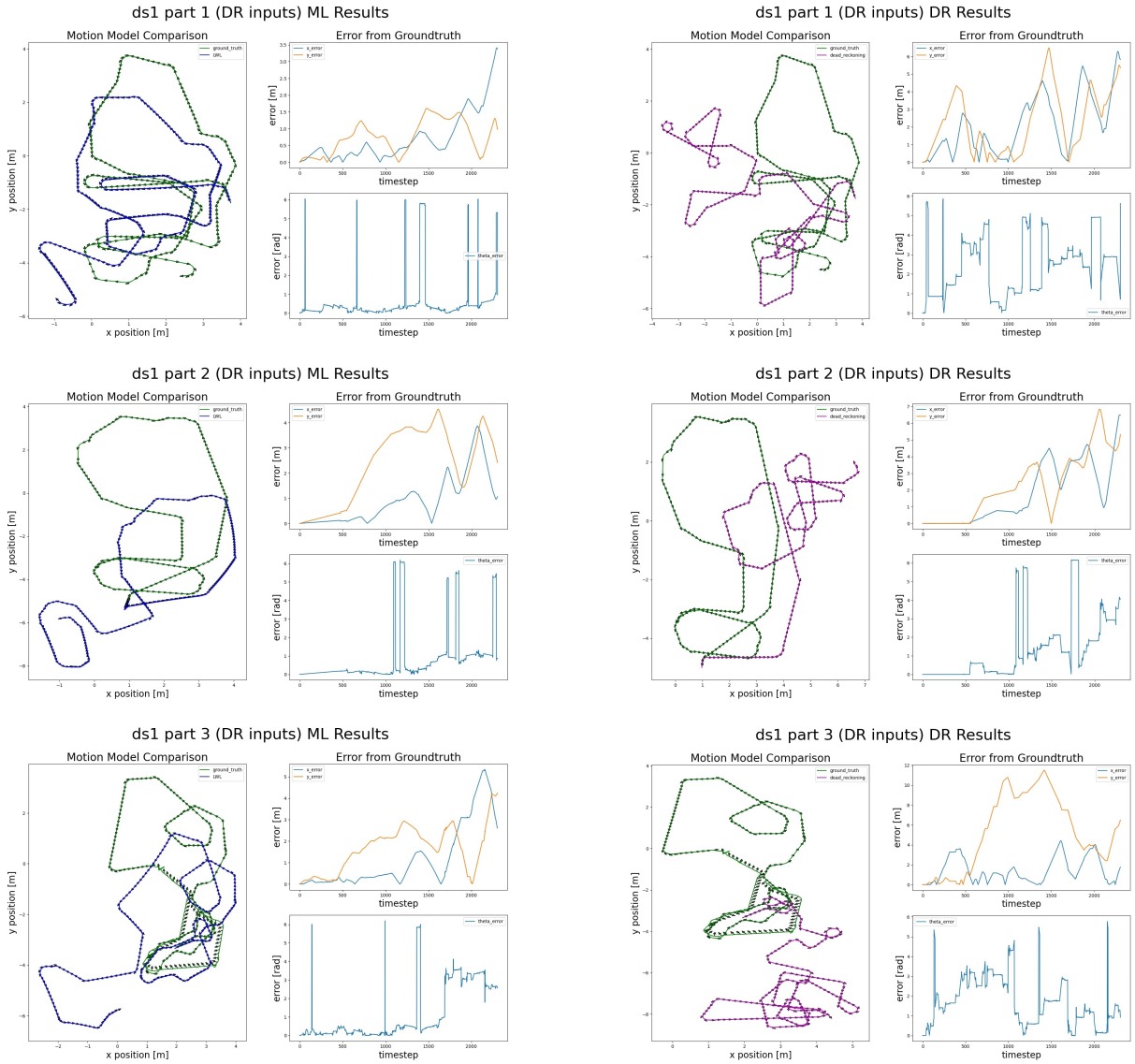

Figure 5: Positional error comparison between LWL motion model and dead reckoning baseline. Left column is LWL results, right is DR results. Rows represent data partitions, from top to bottom 1, 2, and then 3.

2.3.2 Average Positional Error

To give a more quantitative look at the performance of the learned motion model, I took the average positional error for each of the cases above. These results, seen in Table 1, the results for the LWL learned motion model have a consistently lower positional error compared to the dead reckoning results. Furthermore, the standard deviation for the positional error is generally smaller for the learned model, except for the heading prediction. This shows qualitatively that the learned model using dead reckoning input improves upon the dead reckoning algorithm and learns some of the robots wheel slippage and world interaction physics.

Motion Model	Input Type	Part	X Err [m]	Y Err [m]	θ Err [rad]	X Std [m]	Y Std [m]	θ Std [rad]
LWL	DR	1	0.698	0.749	0.543	0.7279	0.4881	1.2076
LWL	DR	2	0.916	2.350	0.779	0.9715	1.4559	1.4184
LWL	DR	3	1.179	1.653	1.152	1.5001	1.0446	1.4728
DR	Raw	1	2.068	2.292	2.397	1.6369	1.7481	1.4416
DR	Raw	2	1.744	2.221	1.320	1.7358	1.8743	1.6479
DR	Raw	3	1.453	5.294	1.697	1.2396	3.7428	1.1998

Table 1: Average positional error and standard deviation for LWL and dead reckoning models.

2.3.3 Alternate Problem Formulation Results

As described in the introduction to this section, I also tested an alternative problem formulation that is similar to my original problem framing. In this framing, I use (mostly) non-transformed inputs to the LWL learning algorithm to see if it can learn a better motion model than dead reckoning. In the visual results shown in Figure 6 we can see that similar to the results using the dead reckoning inputs, this LWL framing produces results with a similar shape to the ground truth path, but translated and distorted slightly. We also see that the positional error increases over time, as in the other motion models.

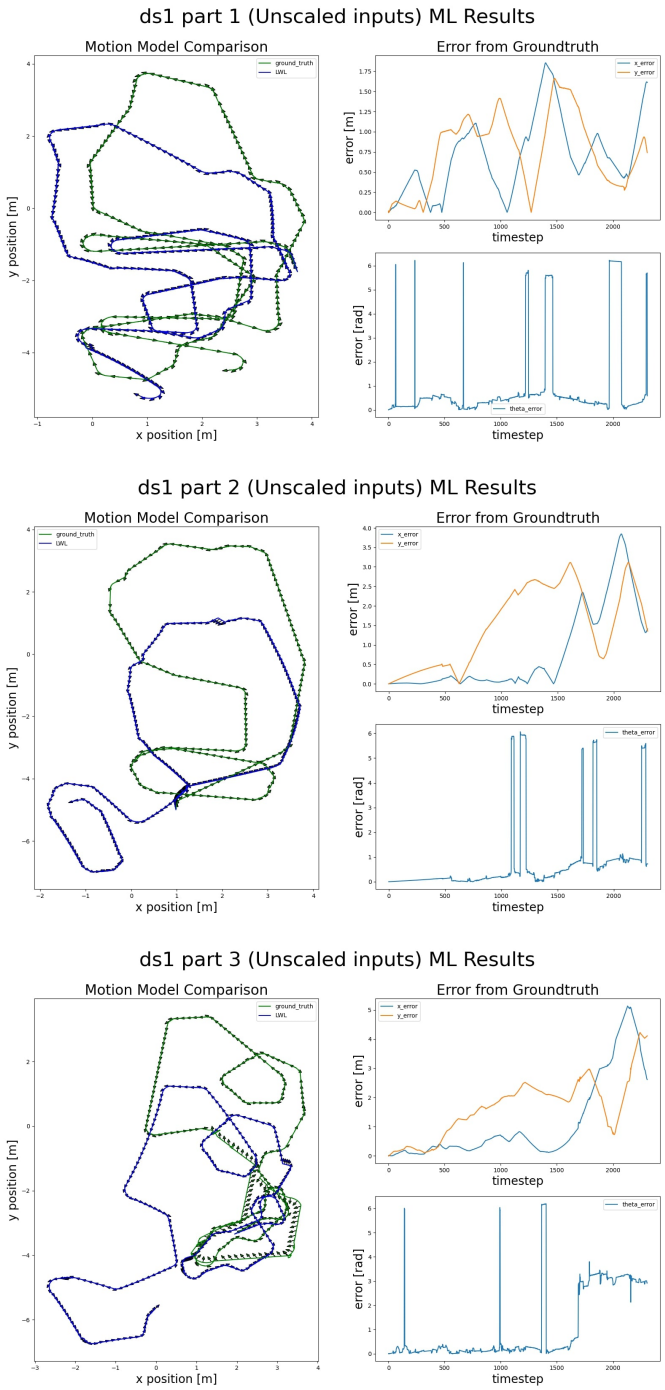

Figure 6: Positional error over time results for raw input inputs to LWL algorithm. Rows represent data partitions, from top to bottom 1, 2, and then 3.

Furthermore, the average positional error and standard deviation are described in Table 2. We can see here that the LWL algorithm using raw inputs performs on average similarly to the algorithm that uses dead reckoning inputs. It even performs better in the part 2 results. This is surprising.

This inability to track the ground truth path is likely related to the problem framing here. Although we avoid having non-linear inputs to this LWL model by transforming the theta representation using trigonometric equations, there

Motion Model	Input Type	Part	X Err [m]	Y Err [m]	θ Err [rad]	X Std [m]	Y Std [m]	θ Std [rad]
LWL	Raw	1	0.695	0.764	0.880	0.4562	0.4645	1.6388
LWL	Raw	2	0.790	1.501	0.713	1.0799	1.0043	1.4086
LWL	Raw	3	1.134	1.658	1.077	1.4484	1.0024	1.4997

Table 2: Average and standard deviation of positional error for LWL motion model (Raw input).

is still a multiplicative relationship between the inputs and the outputs (using the dead reckoning framework as an example). The LWL algorithm is not built to learn this sort of relationship, so it makes sense that the algorithm does not perform better than the dead reckoning input framework.

Together, these results suggest some sort of contradiction in the way I have presented the two problem formulations. It is clear that given their similar (but not perfect) performance, either I have framed the problem incorrectly in either case such that the relationship between the inputs and outputs is not locally linear. However, I have shown the locally linear relationship for the dead reckoning input case, so this does not seem to be the case. Or perhaps the data set I am using does not contain sufficient data to perform perfectly. This might make some sense, given that the LWL algorithm is seeing the validation data for the first time during testing. As we saw with my noisy sin data test, the LWL algorithm is not able to extrapolate from its given dataset well. However, with my given problem framing, I should have the input and output space well covered, suggesting that the data set should not be the issue.

These conflicting signals suggest that future work is required. I would first start by using a higher order regression to learn a controller other than dead reckoning. This means using the raw inputs and a polynomial or similar regression to capture the behavior of the system. This may be able to improve on the results that I have seen here and be able to capture a complete picture of the robots real world dynamics.

2.4 Conclusion

In this report I have presented my implementation of a Locally Weighted Linear (LWL) regression to improve upon the dead reckoning motion model. Although I did not achieve a perfect model of the differential drive robot, I showed results and evaluation metrics which proved a significant improvement over the existing model as evidenced by the lower positional error over time and on average, as well as the preservation of ground truth path shape. Finally, I suggested and tested an alternative problem framing using the LWL algorithm that showed surprisingly similar results to the dead reckoning based motion model.

References

[1] GeeksforGeeks, *Locally weighted linear regression*, GeeksforGeeks, Jan. 2019. Accessed: Nov. 18, 2025. [Online]. Available: https://www.geeksforgeeks.org/machine-learning/ml-locally-weighted-linear-regression/.