Smart Systems

Calvin Bootsman, Ciaran Maher, Fotios Koutkos

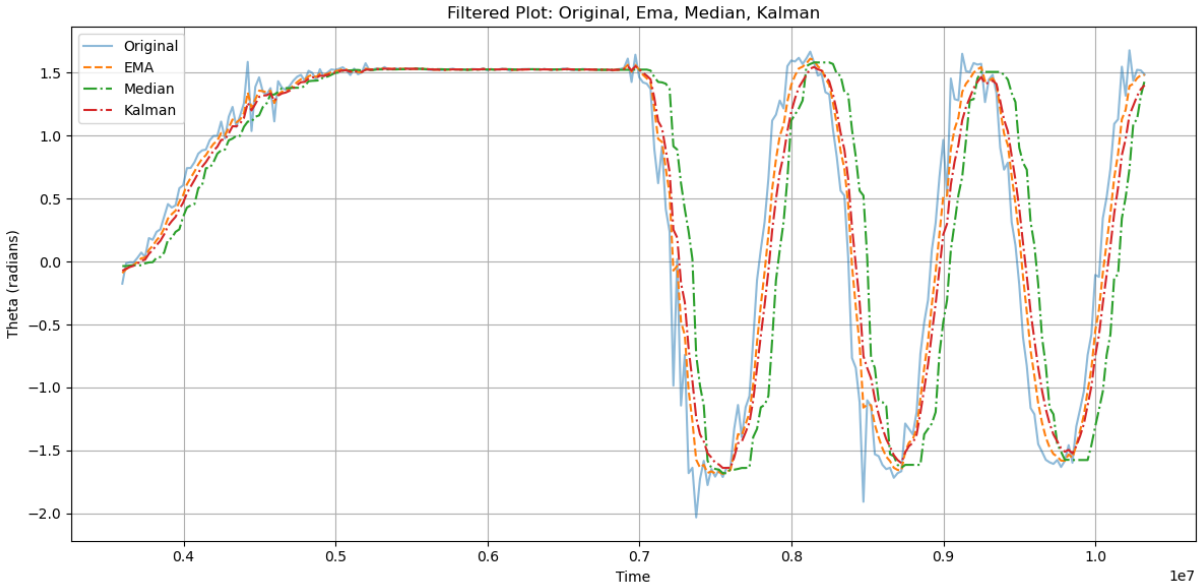
LAB 2

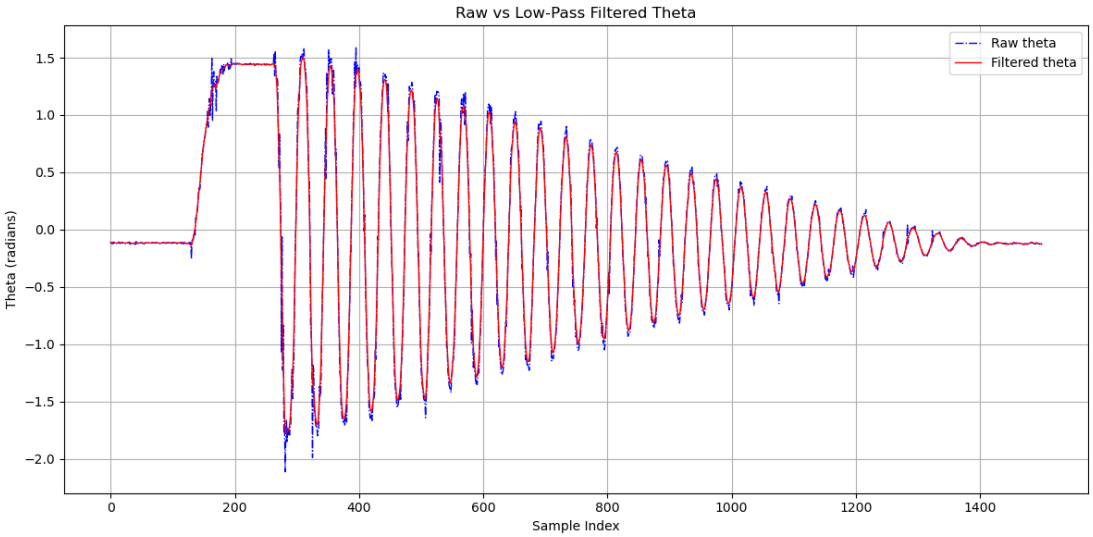
1. What type of noise is present?

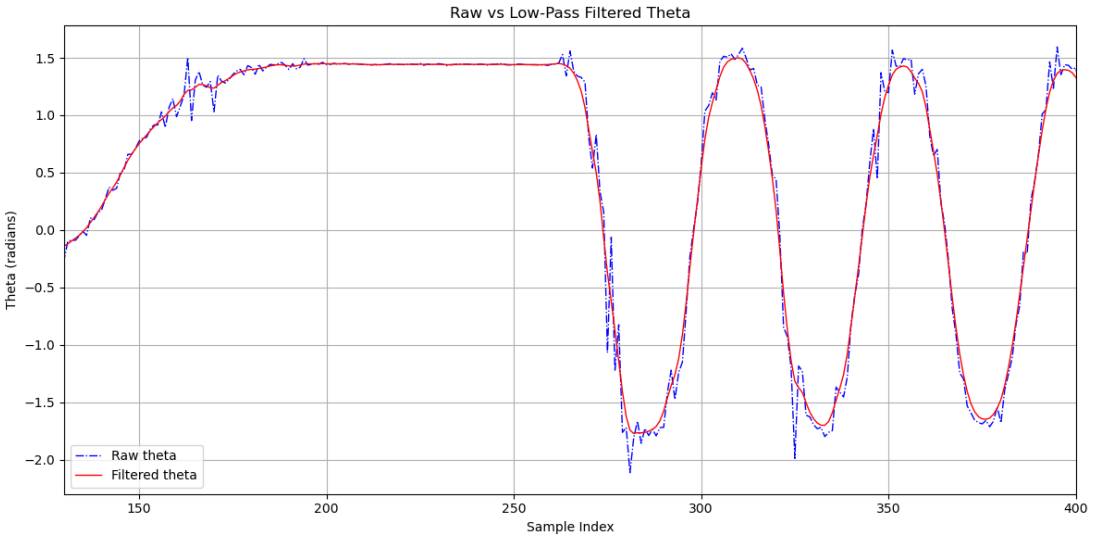
Plotting the ***test_data.csv (provided), sensor_data.csv (collected ourselves)*** and **data_points_free_fall_40Hz (collected ourselves)** we observed quantization noise and additive white noise, which indicates low-resolution analog-to-digital conversion. However, the quantization noise is more visible in the test_data.csv due to low-resolution encoding and a relatively low sampling rate. This sampling rate may have introduced aliasing, making the noise more pronounced.

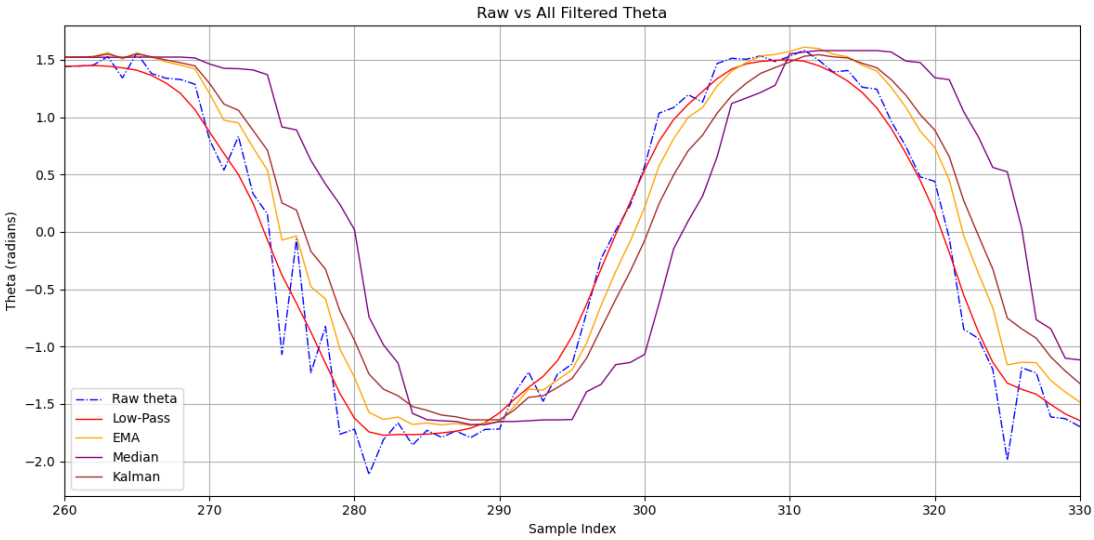
2. Filtering techniques, generated data comparison, preferred filtering method

We applied four filtering methods to the pendulum data: a simple 1D Kalman filter, an Exponential Moving Average (EMA) filter, a Median filter, and a Low-Pass Butterworth filter. Initially, we implemented the three required filters. As shown in the plots, the EMA and Kalman filters outperformed the Median filter. Subsequently, we evaluated the Butterworth filter, which produced better results. It effectively removed outliers and smoothed the data without introducing delay. In contrast, while the EMA and Kalman filters performed well, they introduced a slight temporal shift, leading us to prefer the Low-Pass filter.

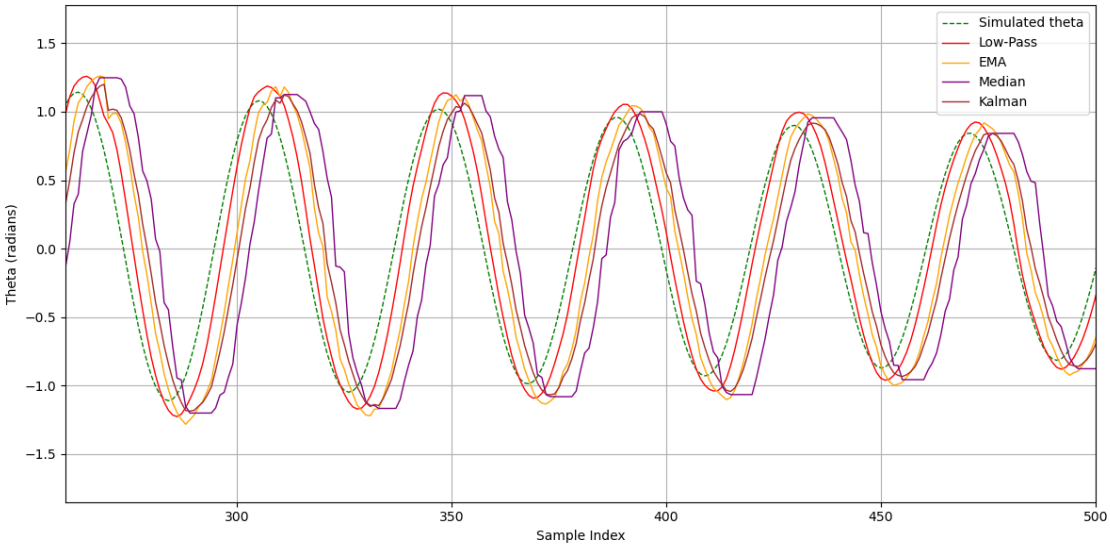



Raw vs Low-Pass Filtered Theta
Raw theta
Filtered theta
Theta (radians)
Sample Index


Raw vs Low-Pass Filtered Theta
Raw theta
Filtered theta
Theta (radians)
Sample Index


Raw vs All Filtered Theta
Raw theta
Low-Pass
EMA
Median
Kalman
Theta (radians)
Sample Index

We generated data from the digital twin to compare the filtered data with the simulated data and calculated the RMSE.



Error Metric: Simulated vs Filtered Data	
Filter	**RMSE** (radians)
Kalman	0.547731
EMA	0.483396
Median	0.772427
Butterworth	**0.275753**

After comparing the filtered output with the simulated data, we observe that the Butterworth filter achieves the lowest RMSE at 0.275753 radians, outperforming the EMA (0.483396 radians), Kalman (0.547731 radians), and Median (0.772427 radians) filters. Although the Kalman and EMA filters show reasonable RMSE values, they introduce temporal delays. Therefore, the Low-Pass Butterworth filter is preferred for its minimal error and absence of delay.

3. Information on the energy dissipation within the system on the filtered data

Using the formulas for,

Kinetic energy (KE): $E = \frac{1}{2}m(L\dot{\theta})$

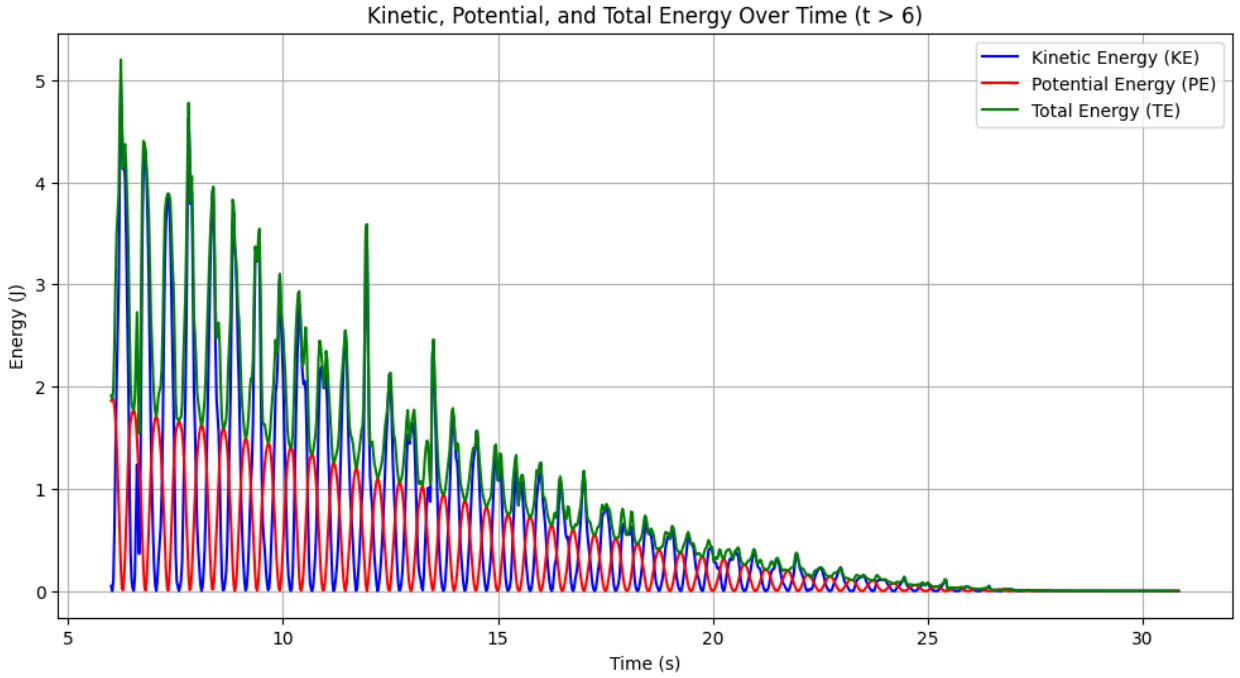
Potential energy (PE): $E = mg(1 - \cos(\theta))$

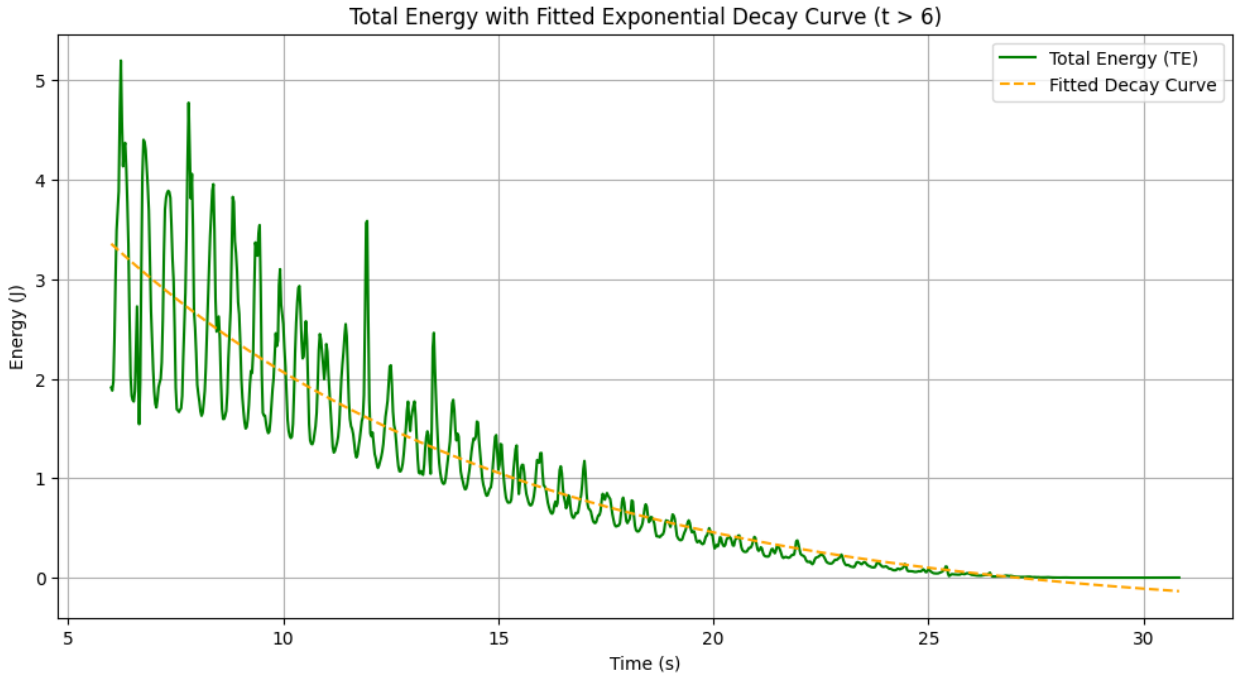
Where,

- m: mass of the pendulum
- L: length of the pendulum

- g: gravity
- θ: angular displacement (radians) of pendulum
- $\dot{\theta}$: angular velocity of pendulum (rads/s)

The collected data provides information into energy dissipation within the pendulum system by analyzing the decay of Total Energy (TE) over time. We can calculate the total energy of the system (PE + KE) and is displayed in green in the below graph. As this is not an ideal pendulum (system with no friction) we can see that the PE, KE and total energy is decreasing to zero. We see that the PE and KE are out of phase, as this is to be expected for a pendulum (KE is highest at the bottom of the pendulum and PE highest at the two highest points in oscillation). From these graphs we can deduce that the KE to PE exchange is not perfect. This is shown in the spikes in TE where this is the energy lost to friction. It is also possible to determine the energy dissipated by friction as the decay rate (b) in the exponential fit plot below.





LAB 3

1. **How would adding the filtering affect the optimization?**

By utilizing the filtered data, you get closer to what has happened on the real model. Especially with the data we collected, by moving the pendulum to 90 degrees and then releasing it, waiting until it stops, we have a scenario which can be approximated using physics-based simulations.

If you don't filter it, the noise in the data will remain and the optimization might try to optimize for the noise, instead of what is happening.

2. **How could the grid search be improved?**

There are multiple ways to improve the grid search.

A. We've created a program which will run the simulation on the GPU. This allowed us to search through 2.500.000 combinations within 5 seconds.
B. By using more vector calculations, the grid search can be sped up, even on the CPU.
C. If it were running on a CPU, it would benefit from parallelization.
D. We also started with a broad search space, which we kept narrowing down, depending on the previous results.

3. **Can you change the code so that the difference in sampling in the recording and simulation is minimized?**

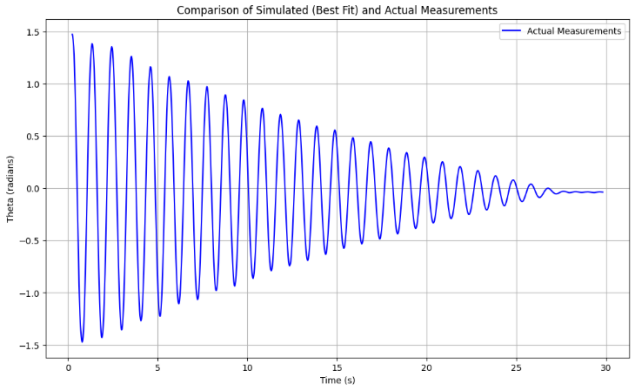
We have two files:

- Lab_3_model_optimization.py
- Lab_3_GPU.py

Both have copied the get_theta_double_dot function from the digital_twin.py file, so that the files can be run standalone, without the need for the digital_twin.py file. For the GPU notebook, the get_theta_double_dot function has been vectorized, to optimize GPU usage.

To minimize the difference in sampling, we recorded the data at a sample rate of 40Hz and changed the delta to 1/40Hz = 0.025s.

4. **What are the optimal parameters? And is the error acceptable? If not, could you include additional parameters in your model? What would they be and why do you think so? Show the values in a table**



For optimizing the parameters, we first needed to understand how the parameters themselves affect the simulation.

Air friction: In the image above, the angular velocity is the highest in the first 5 seconds. Since air friction is dependent on the velocity, the air friction has the most effect on the first part of the signal.

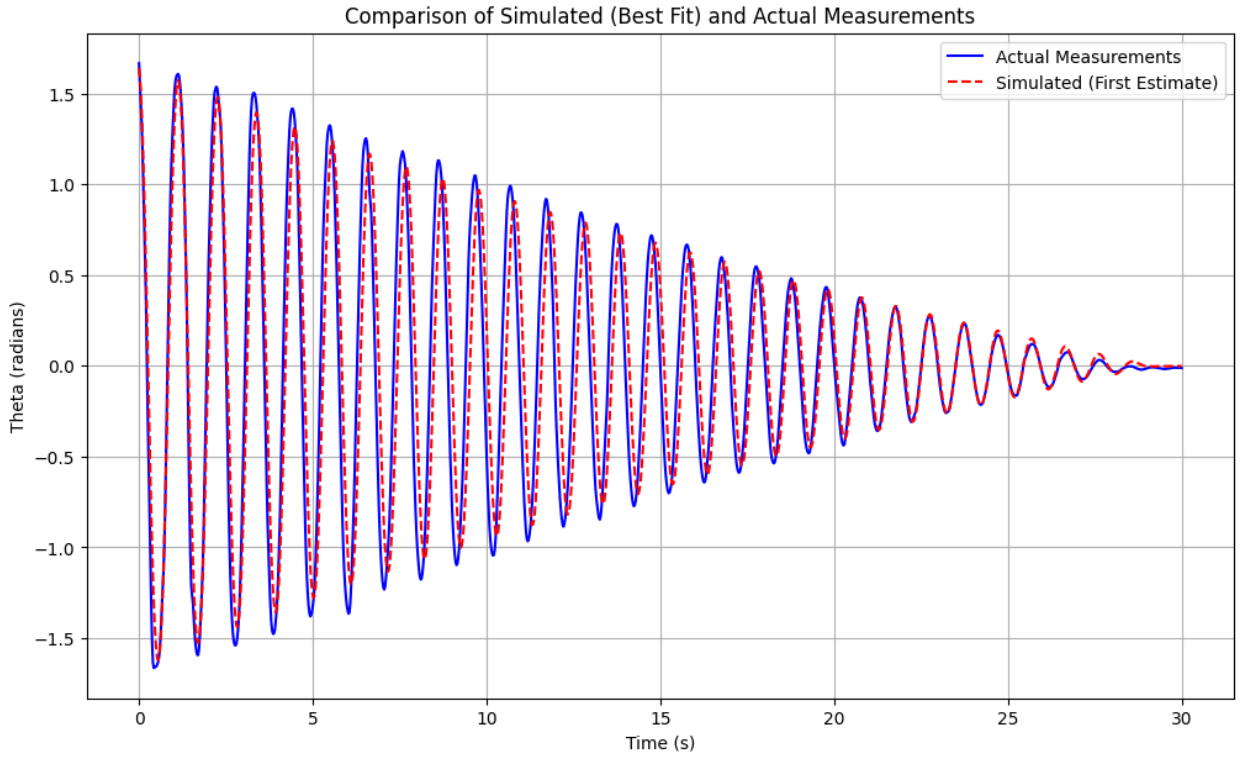
Coulomb friction: Coulomb friction is independent on the velocity. It will have proportionally the biggest effect during the last seconds of the data gathered.

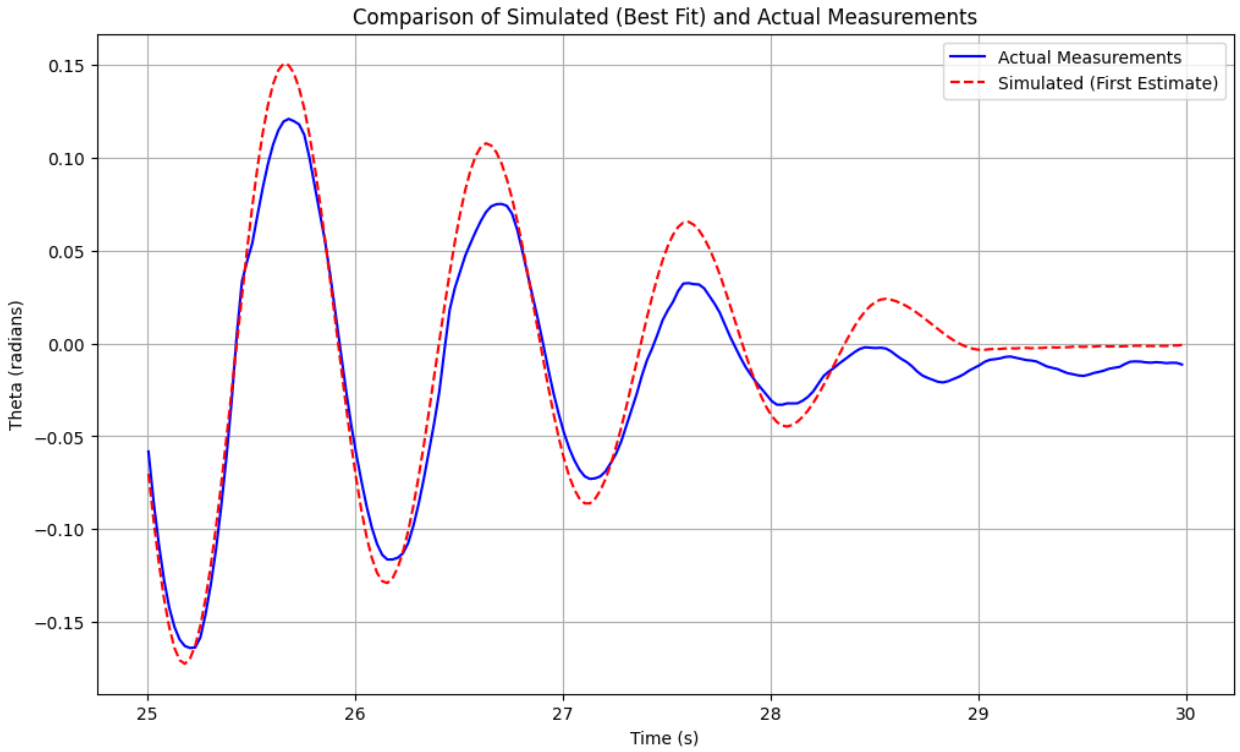
Length: Since we are not accounting for inertia, we try to find an equivalent length to match the frequency found in the gathered data.

We used the GPU to speed up the process and documented it in the notebook of Lab 3. We started off by checking values around the values we found initially in Lab 1 and 2. By knowing how the parameters affect the simulation we started with a broad search space and kept narrowing it down, until we found the right combination of parameters that were almost correct. From then on, we manually tweaked the values until we landed upon these values below.

The optimal parameters are:

Variable	Value
c_air	0.0004564646464646465
c_c	0.0071535353535353
l	0.2358244897959183
RMSE	0.3177211924680939





We think the error rate is acceptable, especially since the recorded data is rather noisy and not very granular. The lowest RMSE we had was found through grid search, 0.16, however, we felt the manual found parameters based on these grid-search found values were following the frequency and amplitude of the signal better.

Additional parameters can be added to simulation. Especially since we do not use the actual length of the pendulum, but an equivalent to get the pendulum swinging with the right frequency. This could be changed by including the inertia of the pendulum. The formula for the frequency of a pendulum, when taking inertia into account is as follows:

$$f = \frac{1}{2\pi\sqrt{\frac{m * g * l}{I}}}$$

Which can be rewritten to:

$$I = \frac{\left(\frac{1}{2\pi f}\right)^2}{m * g * l}$$

With

Symbol	**Value**	**Comment**
f	1 Hz	Approximately
m	0.3 kg	Estimated
g	9.81 m/s^2	
l	0.35 m	Measured
I	*0.025 kg*m^2*	*calculated*

5. **Optimizing a model can be computationally expensive. Discuss how you could further**

minimize the computation you would need for the optimization. Could you reduce the data you use? Could you evaluate in a sparse way, e.g. at random points? Could you automatically estimate the upper and lower range of the parameter values?

You could reduce the number of points needed by calculating the velocity over time first. Then you have everything you need for calculating the angular acceleration. Then you could sample random points in the data and see if the simulation matches the measurements.

With this method you could also split up the training data into chunks. The beginning and the end are important sections. If the parameters perform well in those areas, it can be inferred that it will perform well with the rest of the dataset. This way large parts of the data can be skimped over.

As said before, we started with a big search space which we kept narrowing down. This massively decreased the number of computations needed to find the parameters. This gave us an upper bound for the magnitude in which we needed to search. The lower bound is always > 0.

LAB 4

For lab 4 we used the provided code as the base for our final code. However, we modified the code so it would be able to run on more CPU cores in parallel. This allowed us to speed up the training code by approximately 20 times.

3.1. Discuss the following with your group before you start: What are the specific challenges in using genetic algorithms for controlling a pendulum and achieving an upright position?

3.2. First set the self.delta t in the Digital Twin.py to 0.005 as this is what we use during the optimization process: self.delta t = 0.005.

3.3. Now test the functioning of the digital twin with Lab 1 model simulation.py, if that works you can continue.

3.4. The objective is to generate an output sequence for the model, such as the following example: [3, 0, 7, 0, 3, 0, 7, 0, 3]. In Lab 2, it was demonstrated how these sequences correspond to movements to the right and left for a specified duration. Now, by employing a genetic algorithm, the objective is to identify a sequence that leads to achieving the upright position. How should the parameters of the genetic algorithm (e.g., population size, mutation rate, crossover rate) be determined? What strategies can be implemented to fine-tune these parameters effectively?

1. **Decide the simulation length, in what amount of time do you think the process could be performed?**

   we've set the maximum simulation duration for evaluation at 10 seconds. During training, however, our approach uses progressive lengthening. We begin with shorter simulations of 500 steps and gradually increase this duration over the generations.

2. **What would your population size be?**

The code sets a population size of 50 individuals, as indicated in the run_simulation function (population_size=50). This population size is usually a good balance between diversity and computational expense. We did a small grid search to see if 100 and 200 populations were performing well, and they didn't necessarily perform better, but the computational costs did increase.

3. **What would the parent size be (the individuals going to the next generation)?**

As for the population size, we tried grid search for the right parent size and found a 50% parent size was ideal. In our case that is 25 parents.

4. How does the mutation rate affect the result?

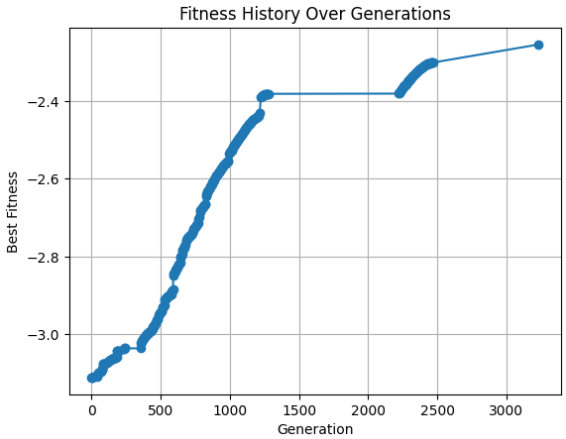
The mutation rate has a large effect on the training. We noticed that 10% mutation rate was ideal. With higher and lower mutation, it would converge to a lower (more negative) value.

5. How many generations does it on average take you to achieve the goal?

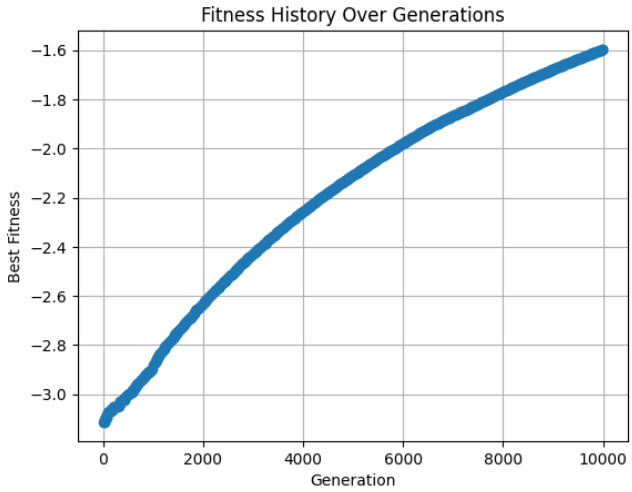
Our training uses a variable length of training. It first simulates the first 5 seconds at generation one, and every generation will increase the simulation length a fraction, until the maximal simulation time is reached in the last generation. This affects how fast the GA will learn.

With a low number of max generations (500), there is a big chance it will get the pendulum up. However, the chances of balancing it are low. With more generations, the chance of balancing properly will increase.

The image below shows an average training for 5000 generations.



But with 10k generations it looks like this:



6. **Can you come-up with another method of evaluation that will improve the learning rate?**

We came up with the fitness function that will sum all the theta values for the whole simulation of an individual and divide it by the simulation steps. 0 means that it is performing perfectly, lower than zero means it is performing less than perfect. Al though, starting from a hanging position means that it can never reach zero.

7. **Bonus question: What is fundamentally the problem with the current crossover function in the GA? Can you come up with a better version?**

The current crossover function chooses the crossover point randomly. This means that if the crossover is at the first value, one parent is completely dominating the solution which isn't diverse enough. Changing the single crossover point to be closer to the middle is better for diversity.

Our crossover function: Segmented crossover function

It firstly divides the parent solution into three segments, start, middle & end which is then combined to create the offspring. A critical length is defined as the point of crossover from the first value. The function then compares the absolute sum of these values from both parents and chooses the one that is higher. This is the encourage larger movements in the early parts of the simulation to swing the pendulum up.

The end segment is similar, it compares the variance of the critical length. This makes the crossover at this length from the last value in the same parent as the doner of the start section. The parent with the smaller variance is chosen. This encourages small adjustments at the end to stabilize the pendulum in the inverted position.

The middle section is directly taken from the second parent. This is then joined together to make the offspring.

8. **Tasks 4.1. Learning these controls can be computationally expensive. Discuss how you could further minimize the computation you would need for this optimization. Could you learn step by step (instead of x seconds at ones)? Could you find a better initialization of the population?**

We did the following things to decrease the computation time:

- Run the program on multiple CPU cores
- Progressively increase the simulation duration, from 5 seconds at the start to 10 seconds at the end.