Development of a control algorithm for a simple robot arm using an evolutionary approach to machine learning

Introduction

The goal of this exercise was to familiarise myself with the practical use of machine learning to solve real-world problems by selecting and then attempting to solve some specific problem. Given only limited access to educational materials on the subject and no Internet access, I have been constrained by:

1. My ability to understand the basics of certain types of machine learning algorithms; and
2. No access to large sets of data that could be used as training data for a machine learning algorithm.

The purpose of this paper is to document the approach followed and lessons learnt along the way.

Choice of problem

Given no access to learning data, two types of problem seemed suitable:

1. Games, such as chess, where the program can play against itself and learn that way; or
2. Physical simulations of simple robots, where the program can quickly run simulations of the actions and outcomes of a robot with a specific goal, and from that data learn a control algorithm for the robot.

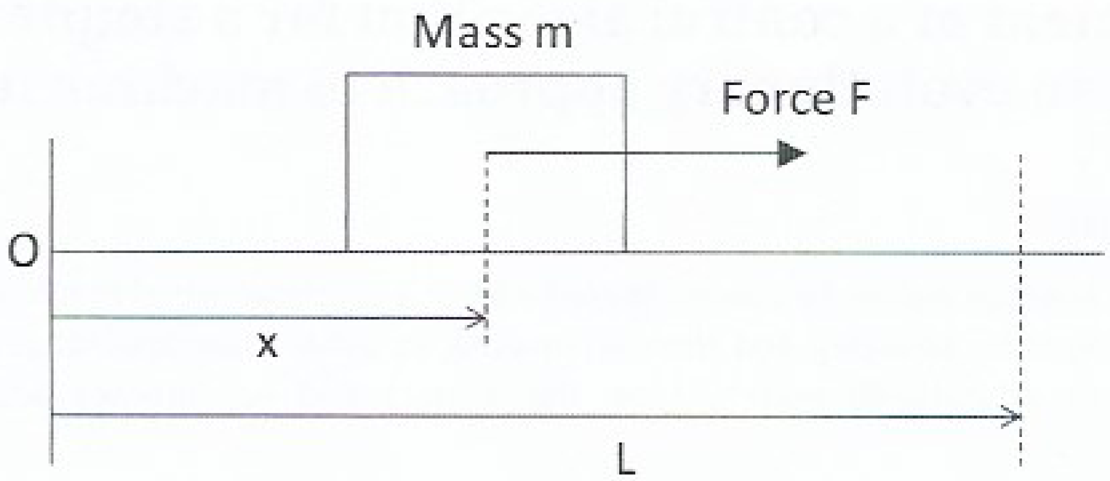
My final choice of problem was that of a very simple robot arm where the control algorithm was to be developed using an evolutionary approach to the learning. This choice was influenced by:

1. My personal experience and interest in both mechanical engineering and evolution in nature; and
2. The types of problem where I felt confident I could grasp both the high-level and detailed concepts required.

Problem definition

My goal was to define an extremely simple physical problem so that my effort was spent on understanding the machine learning element of the problem rather the design and coding of the simulation dynamics.

I settled on an extremely simplified model of a robot "arm" which is required to move an object a given distance in one dimension to a target location. The arm should cope with objects of different mass within a given range and different target distances also within a given range.



The model assumptions and specifications are as follows:

1. A mass of **m** kg must be moved from rest at point **O** to a location **L** metres from **O**.
2. The mass may vary between 0.1 and 10kg, and the target distance may vary between 0.1 and 5m.
3. The mass is moved by an actuator that can deliver a force **F$_{max}$** in both directions. The actuator delivers the force in response to a control input, and the force is given by the following function:

   $$F = F_{max}\tanh(Ap + z_F)$$

   where **A** is a constant, **p** is the input signal, **z$_F$** is a random noise function.

   The form of this function is used primarily to limit the actuator output to a maximum force given any possible control input. Throughout the exercise I set the value of **A** to **2**.
4. The mass sits on a frictionless surface.

Discussion

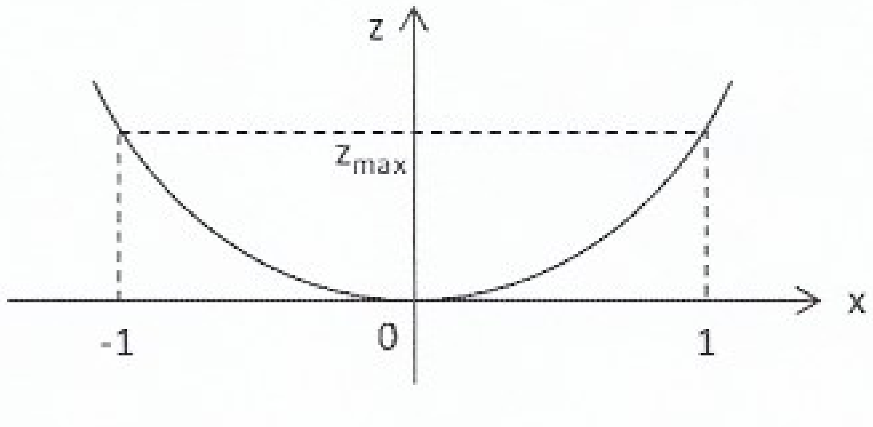
A frictionless surface is assumed only to keep the programming of the simulation model simple. However including friction would not be overly complex. A force D would be included according to the function below, so that the mass is stationary unless the actuator force exceeds the sliding frictional force:

|F| <= µmg : D = -F

|F| > µmg : D = ± µmg where sign is such that D opposes F

5. Sensors on the actuator measure the position **x** and the velocity **v** of the mass. The sensor outputs contain random noise components **z$_x$** and **z$_v$**.
6. Noise is simulated by generating an error **z** at time **i** using the function

   $z = z_{max}x^3$ for positive **x** and then mirrored about the y axis. **x** is a random number generated between -1 and 1. In each case (i.e., for **z$_x$**, **z$_x$** and **z$_v$**), **z$_{max}$** was set at **0.02**.



7. Success is achieved when the mass is:

    a) Within 1 cm of the target location; and

    b) Moving at less than 1 cm / s.

    (Note that the second criterion is necessary because we are using a frictionless surface, and without the speed criteria, success will likely never be achieved.)

Simulation dynamics

The motion of the mass is given by the equation F = ma where **F** is the force applied in Newtons, **m** is the mass in kg, and **a** is acceleration in m/s^2.

For modelling purposes, I assume that the controller takes a sensor input at time intervals of **dt** seconds, so that the force applied to the actuator is constant during any given time of **dt** seconds. The value of **dt** was set to 0.02 seconds.

Thus the equations for position and velocity at time **t**$_i$ where each time step **i** lasts **dt** seconds are:

$$x_{i+1} = x_i + v_i dt + \frac{F}{2m}(dt)^2 \qquad (1)$$

$$v_{i+1} = v_i + \frac{F}{m}dt \qquad (2)$$

Form of control algorithm

A simple and viable control algorithm (based on engineering experience) is one in the form

$$p = A(L - x) + Bv$$

Where **p** is the input signal to the actuator, **L** is the distance from point of origin **O** to the target, and **A** and **B** are constants with positive and negative signs respectively.

Using this function means that:

- The further the mass is from the target the greater the force pushing it towards the target; and
- The force pushing the mass towards the target is reduced the faster the mass is moving toward target.

With an appropriate choice of values for **A** and **B**, this algorithm gives an adequate response for a particular mass and target distance. Suitable values for **A** and **B** can be found using control theory.

Discussion

I am not sure whether this is actually a "good" form to use. I know this is the basic form used in motion tracking, but I wonder whether that may be because it is suited to an analogue controller. For example, a relatively simple function would be one that applied maximum force towards the target then applied maximum force in the opposite direction at half-distance. The point of switching can be continually re-assessed during the motion.

One possible reason why this solution is not preferred is that (with noise) the system could go into serious oscillation switching rapidly between max force in one direction and the other. Although not a problem for a simulation, in the real world this would generate much more physical load and wear on the system.

The algorithm actually used is less susceptible to suffering serious oscillation.

However, initially I wanted to research whether a better algorithm could be found using a polynomial form:

$$p = A_1X + A_2X^2 + \ ... \ + A_nX^n + B_1V + B_2V^2 + \ ... \ + B_mV^m$$

where:

$X = (L - x)/L$; and

$V = v/v_{max}, v_{max} = \sqrt{F_{max}L/m}$ (the velocity achieved at **L/2** if max force applied from **O**)

Discussion

With hindsight, my choice of **v$_{max}$** may not have been sensible. My intention was to "normalise" velocity so that its' magnitude was never greater than 1. Higher order terms have a greater impact on the overall output than lower terms when **|V|** exceeds 1 and lower impact when **|V|** is less than 1. Instinctively I thought this would be undesirable, making a successful algorithm more "difficult" to find.

It was only after my investigations were complete that I realised I had failed to achieve what I had wanted. An algorithm might apply maximum force from **O** and beyond **L/2**, yet still succeed in returning the mass to a "stop" at the target distance. A better **v$_{max}$** might have been $\sqrt{2\,F_{max}L/m}$, the velocity achieved at target distance if max force were applied from **O**. No successful algorithm could apply max force at all points between **O** and **L** and later succeed in bringing the mass to rest at target distance.

I chose this form of algorithm in the hope that it would improve a first order form of solution by:

- Producing a quicker response (i.e., the mass reaches the target in less time); and
- Working over a wider range of masses and target distances.

Both speed and velocity are normalised with the intention of improving the working range of the algorithm – by reducing the effect of changing mass and length parameters.

Discussion

Whether or not control theory can be used to identify optimum values for **A$_1$** to **A$_n$** and **B$_1$** to **B$_m$** is beyond my knowledge, but it is certainly not simple, and I suspect complexity increases with the polynomial order.

At the outset I also didn't know whether the polynomial form of algorithm could produce better results. My assumption was that it probably could, given enough terms, as in theory any continuous function should be representable by a polynomial of sufficient terms.

This then was the first form of control algorithm investigated using evolutionary machine learning. To keep the computational speed up, I limited the learner to polynomials of order 3, so that the algorithm took the following form:

$$p = A_1X + A_2X^2 + A_3X^3 + B_1V + B_2V^2 + B_3V^3$$

where **A$_1$** to **A$_3$** and **B$_1$** to **B$_3$** are then attributes of an individual (its genotype). I then rewrote the expression so as to have an overall "gain" **K** – a single parameter that dominated the "overall responsiveness" of the function:

$$p = \mathrm{K}(X + A_2X^2 + A_3X^3 + B_1V + B_2V^2 + B_3V^3) \qquad (3)$$

Based on initial results and the consequent additional understanding of the problem, I then modified the form of the control algorithm so as to reduce sensitivity to varying mass and target distance. As will be shown later, oscillating responses were a problem at the low mass and distance end of the ranges being tested, so I introduced a variant of the algorithm that included two new variables, $\mathbf{m_{max}}$ and $\mathbf{L_{max}}$, being the maximum mass and target distance the actuator was expected to cope with. These I combined with the actual mass and target distance to create a constant, to which I gave the unwieldy name of "normalising mass distance constant" or $\mathbf{C_n}$ for short:

$$C_n = \left(\frac{mL}{m_{max}L_{max}}\right)^{1/2}$$

The alternative control algorithm became as follows:

$$p = C_n\mathrm{K}(X + A_2X^2 + A_3X^3 + B_1V + B_2V^2 + B_3V^3)$$

The purpose of the normalising mass distance constant was to reduce the control signal amplitude for very small mass and target distance values. Although this meant the theoretical best possible success time was increased, it reduced the likelihood of wild oscillations.

Discussion

Reducing the time interval **dt** (the time interval between each new sensor reading and calculation of force to be applied) should and did have an effect on the working range of the actuator. Clearly when the mass is light and the target distance small, then the mass will arrive at the target distance very quickly. If the time interval is too long then the actuator simply cannot respond fast enough. In my very first attempts **dt** was set to 0.2 seconds, but this proved far too long and I reduced the value to the value stated above of 0.02 seconds. This did not remove the problem of oscillations entirely, but did at least give the controller time to adjust force levels several times before target distance was reached.

Primary investigation

My first goal was to investigate whether using a third order control algorithm did in fact produce better results. In addition I wanted to investigate whether the inclusion of noise in the model had an impact on the algorithms' performance. In summary therefore, I was examining three factors:

1. Order (first vs third).
2. Inclusion / exclusion of normalising mass distance constant.
3. Inclusion / exclusion of noise.

Secondary investigation

I then attempted to develop a more generic approach that allowed the learner to determine the best form of the control algorithm itself, with only the variables specified initially.

Model testing

Although the simulation and the learner were built using Java, I also built a simulation model (not the learner) in Excel so that:

- I could check that my simulation in Java was error-free by comparing simulation results with the Excel version with all parameters set the same.
- I could more easily play with some of the simulation parameters and see the consequences.

Primary Investigation – polynomial algorithm

Genotype form

Each individual has a genotype in the form {K, A_2, A_3, B_1, B_2, B_3} corresponding to equation (3) above.

Overall method

I used the "microbial" approach described in (Harvey, 1996) to replicate a generational approach. The steps were as follows.

- **Generate initial population.** Each individual's attributes were generated randomly within constraints to ensure that the individuals have a reasonable chance of success. The parameters for the initial population were as follows:

Population size	20
K	0.1 to 20
B_1	-5 to 0.1 I know that B_1 should most likely be negative as positive values will probably lead to positive feedback and increasing oscillation. However I included a small positive range to allow for a small proportion of the population to have positive values, in case this produced solutions I had not considered. In hindsight I think this was probably pointless.
A_2, A_3, B_2, B_3	-0.9 to 0.9 Higher than first order terms I restricted to less than 1 as they were intended essentially as harmonics on the underlying first order function.

- **Select individuals and run tournaments.** I ran 20 "generations", so with a population size of 20 that meant 400 tournaments using the microbial model. For each tournament:
  - One individual was selected at random.
  - A second individual was selected at random within the "deme" of the first individual. Given the population size of 20 I used a deme of 5.
  - 8 simulations were run for each individual with the following mass and target distance combinations. 4 were run as "opposing" (high mass, low distance to low mass, high distance) and 4 as "aligned" (low mass, low distance to high mass, high distance).

	Opposing				**Aligned**			
Mass	0.1	3.4	6.7	10	0.1	3.4	6.7	10
Distance	5	3.666	1.733	0.1	0.1	1.733	3.666	5

  - The success times for each individual were added up and the individual with the lower total success time was deemed the winner.
  - The loser's genes were replaced with the winner's genes on a random basis: each of the loser's genes was replaced with 50% probability.
  - All of the loser's genes were mutated by ±1%.

Discussion

The microbial approach replicates a form of sexual selection whereby genes are "recombined". This occurs in the step where a tournament loser's genes are replaced with the winner's genes on a 50% probability basis. In other words, on average, half of the loser's genes are replaced with the corresponding genes from the winner.

I am unsure whether this technique does anything more than the initial random selection of individuals for the population. I can see this might be a useful technique if and only if there is total or near independence between genes. In other words, changing one gene has little or no effect on the "performance" of another. This can be seen in the animal kingdom where the performance of a gene impacting digestion efficiency, say, is unlikely to be affected by the performance of a gene controlling eyesight. Swopping the digestion gene for a "better" digestion gene is likely to improve the overall "fitness" of the animal because the eyesight remains unaffected.

In this case, however, the performance of all genes is strongly related. Let us assume that two algorithms are tested, one outperforms the other and is therefore the tournament winner:

Winner: $p = 12.5(X + 0.6X^2 + 0.8X^3 + (-2.1)V + 0.2V^2 + (-0.5)V^3)$

Loser: $p = 10.7(X + 0.9X^2 + 0.2X^3 + (-7.3)V + 0.8V^2 + (-0.2)V^3)$

In this case, there is no reason to think that replacing the loser's third order X term constant (0.2) with the winner's equivalent (0.8) will produce a better result for the losing algorithm. This is because all the constants work closely together to produce an overall result. They do not represent independent components of a whole, each of which can be optimised in isolation – rather they are interconnected variables that affect each other strongly.

As a result, although I have included the technique in the overall method, I do not see this as introducing anything more than a random shuffle of the genes. It does not represent a technique that "responds" usefully to selection pressure. Using a larger population with fewer iterations and eliminating the recombination would, in my view, have the same result – as the randomness of recombination is replaced by the increased randomness of the initial population selection.

Fitness function

The basic fitness function is simply the time taken for the mass to "stop" at the target position (i.e., be within 1cm of the target position and travelling at a speed of less than 1 cm/s).

However the fitness function is complicated by a number of factors:

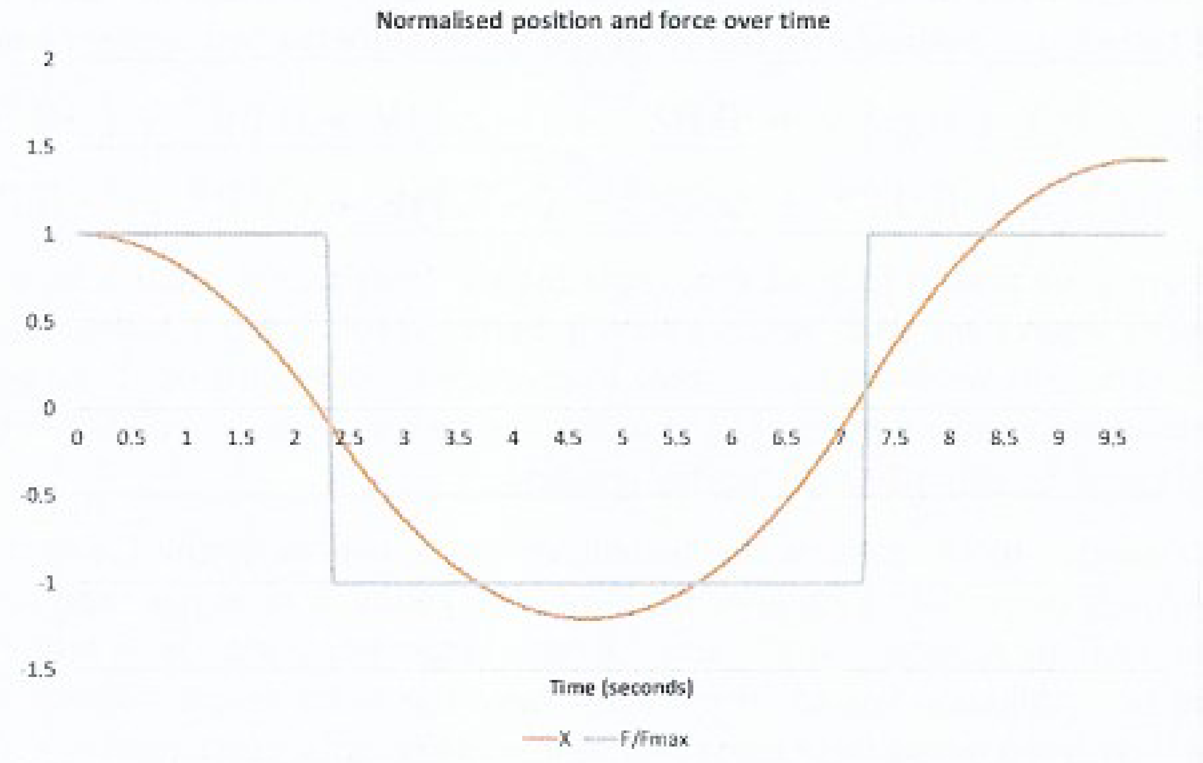
1. **Varying mass and target distance.** As stated above, I ran several simulations for a single "trial", each testing a variety of mass and distance combinations for the same individual. The fitness function then became the sum of the time taken for each simulation to succeed. This form was used so as to "prefer" algorithms that worked well over the required range of mass and distance.

2. **Excessively long times to achieve success.** With certain choices of control algorithm parameters and mass / distance combinations the arm can take an extremely long time to reach success, and this slows up the learning process as simulations are run for a very long time. I decided instead to introduce a "cut-off" time of 10 seconds, at which point the simulation would end. I labelled this as an "out of time" failure and this is measured in the fitness function simply by recording the simulation success time as **10 seconds**. This means there is no differentiation between an individual that would have succeeded in 11 seconds and one that would have succeeded in 100 seconds. Although this is clearly a disadvantage in terms of providing a useful selection pressure, there is a necessary trade-off with the computing time required.

Discussion

Increasing the cut-off time has the effect of making a higher proportion of the initial population "viable" in the sense that they will achieve success before the cut-off and selection pressure can then be applied to more individuals. Using my Excel model, I could get a feel for what would be likely achievable success times, and I was able to choose 10 seconds as a cut-off that would be much longer than many individuals would achieve. However I have not rigorously tested whether a longer cut-off time would actually result in a more efficient learning process.

3. **Failures due to positive feedback.** With certain choices of control algorithm parameters and mass / distance combinations the arm can never succeed because positive feedback results with the mass moving in ever larger oscillations about the target point, as shown in the illustrative chart below.



$X = {(L-x)}/{L}$ so **X=1** represents the start position **O** and **X=0** represents the target position;

F = force applied, and $\mathbf{F_{max}}$ = the maximum force that can be applied by the actuator.

As a result I set a condition that if the mass relative position X goes outside the range -1 ≤ X ≤ +1 at any time after the mass has passed the halfway point X=0.5 for the first time, then the arm is deemed to have "failed". (Only testing after the halfway point is passed prevents fails occurring at the beginning of the simulation when the error signal introduced by the position sensor could give an initial reading of X>1.)

I wanted the fitness function to count this "out of bounds" failure as "worse" than an out of time failure because an "out of bounds" failure could never succeed, whereas an "out of time" failure might possibly succeed, given more time. As a result I decided that an "out of bounds" failure should be given a success time of 10n seconds, where n is the number of simulations in a single trial for an individual. This meant that an individual with a ***single*** "out of bounds" failure (and otherwise all successes) would always score worse than an individual that had "out of time" failures for ***every*** simulation in its trial. This is shown by the illustrative success times for trials for two individuals below:

Simulation	Individual A	Individual B
1	3.5	10 (out of time)
2	4.7	10 (out of time)
3	8.5	10 (out of time)
4	50 (out of bounds)	10 (out of time)
5	5.2	10 (out of time)
Total	**71.9**	**50**

Even though Individual A fails only once, and Individual B fails every time, B beats A because the single A failure is an "out of bounds" failure, and B only has "out of time" failures.

Results

Summary of approach

I ran 8 tests so as to examine the three factors under consideration:

Order	Normalised	Noise
1	No	No
1	No	Yes
1	Yes	No
1	Yes	Yes
3	No	No
3	No	Yes
3	Yes	No
3	Yes	Yes

For each test, I ran the learner with the parameters described earlier:

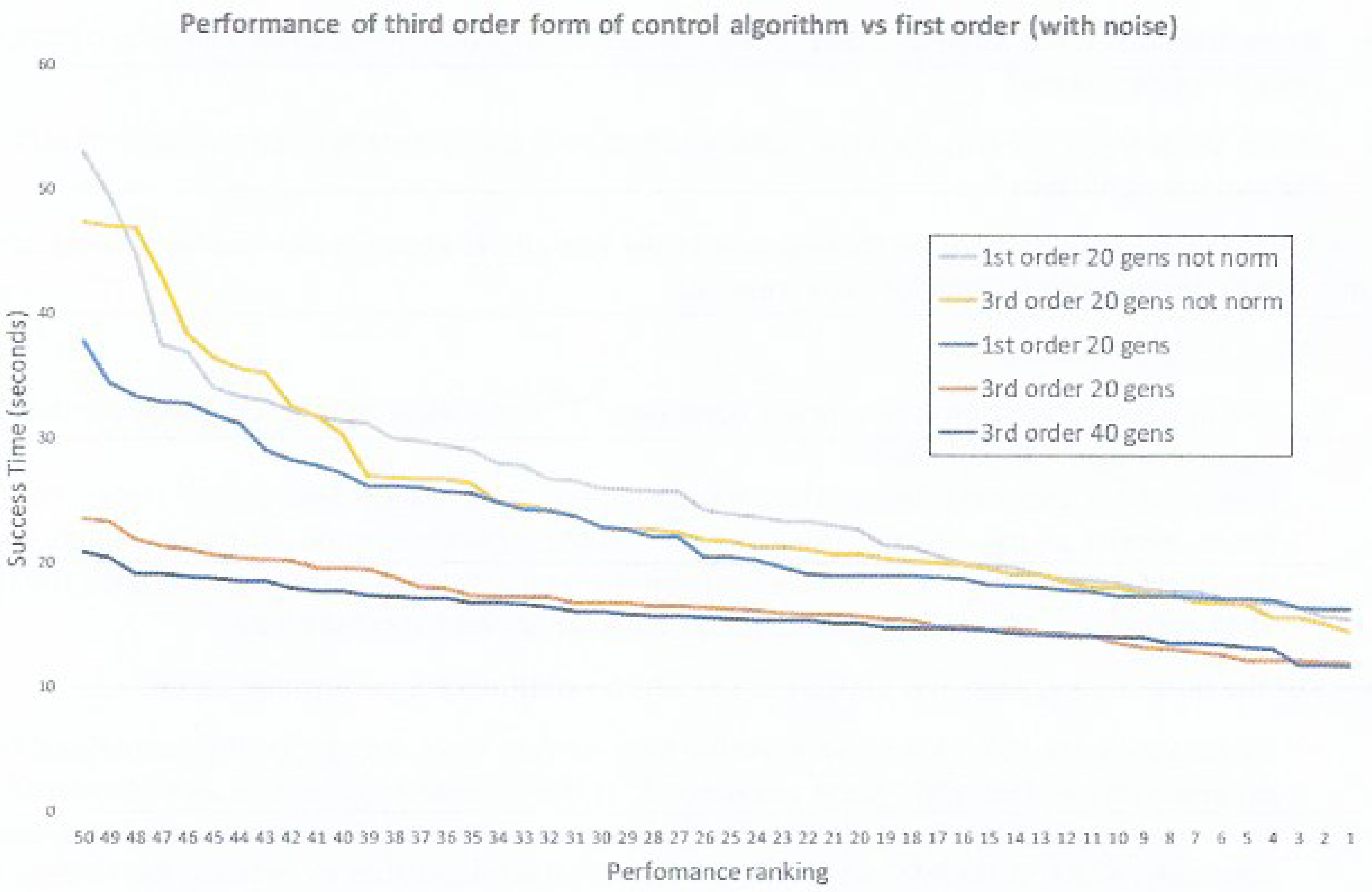
Population	20
Generations	20
Deme	5

However, for each test I also ran the learner 50 times. This meant that an initial population (created randomly) was created 50 times and the overall "best performer" algorithm was the best of the 50 "best performers" produced from the 50 "learns".

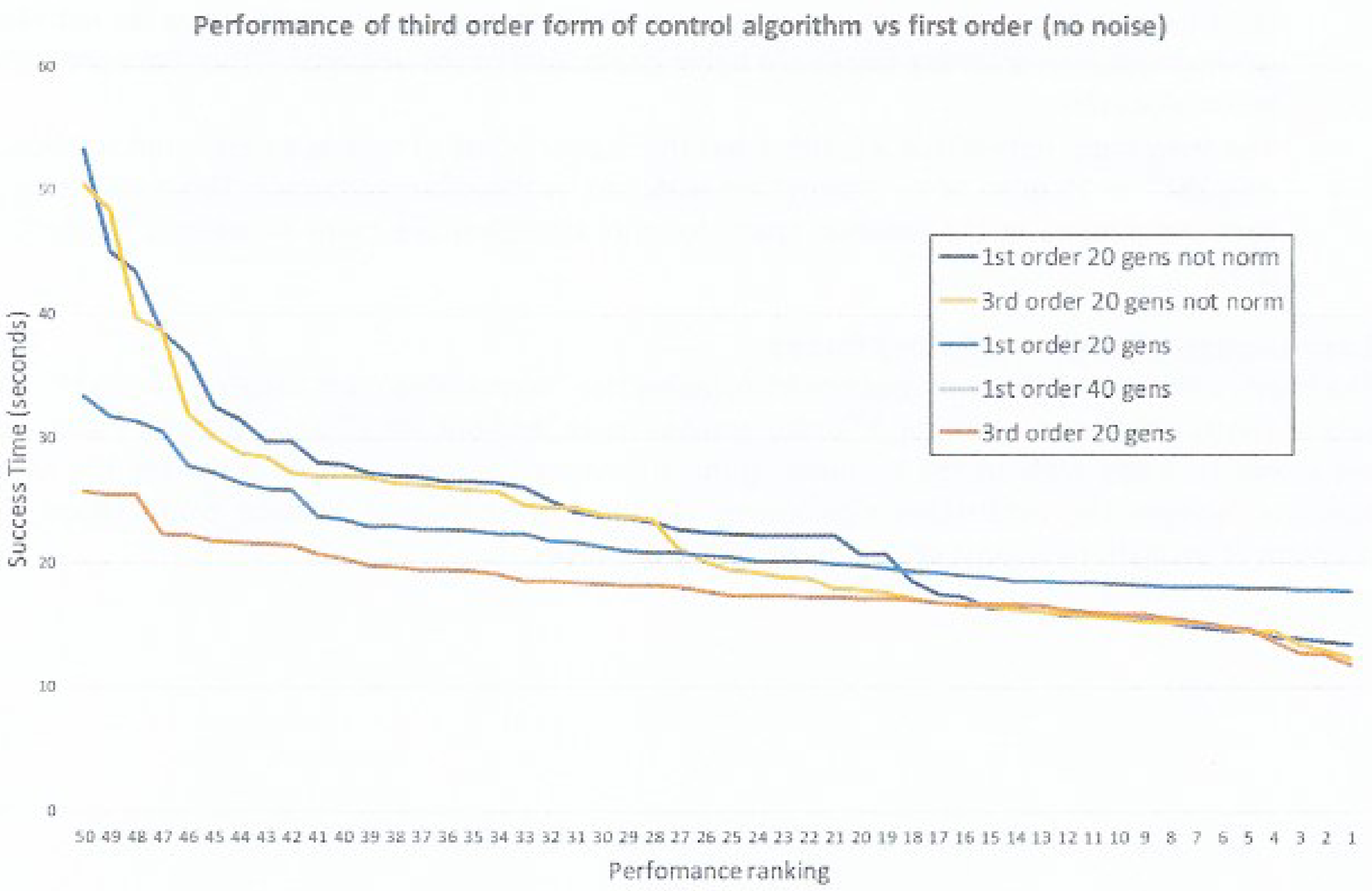
I could have instead run the learner once with a population and deme 50 times larger, and this would have created the same conditions for producing an overall best performer algorithm. However, running the learner 50 times with a smaller population gave an indication of how "easily" each test arrived at an algorithm that was near the overall best performer.

In addition, I ran some limited additional tests where I increased the number of generations (doubling them to 40) to investigate whether more generations would allow a significant improvement in success time, or whether only very small incremental gains could be found.

Overall results



Performance of third order form of control algorithm vs first order (with noise)



Performance of third order form of control algorithm vs first order (no noise)

As a reminder, three factors were being investigated:

- **Order**. Which order algorithm performs better, first order or third order?
- **Normalisation**. What impact, if any, does the use of a "normalising mass distance constant" have on performance?
- **Noise**. What impact, if any, does the inclusion of noise in the system have on the performance of the various algorithms?

The following are my observations looking only at the two charts above, which examine the success time of 50 different populations for each scenario:

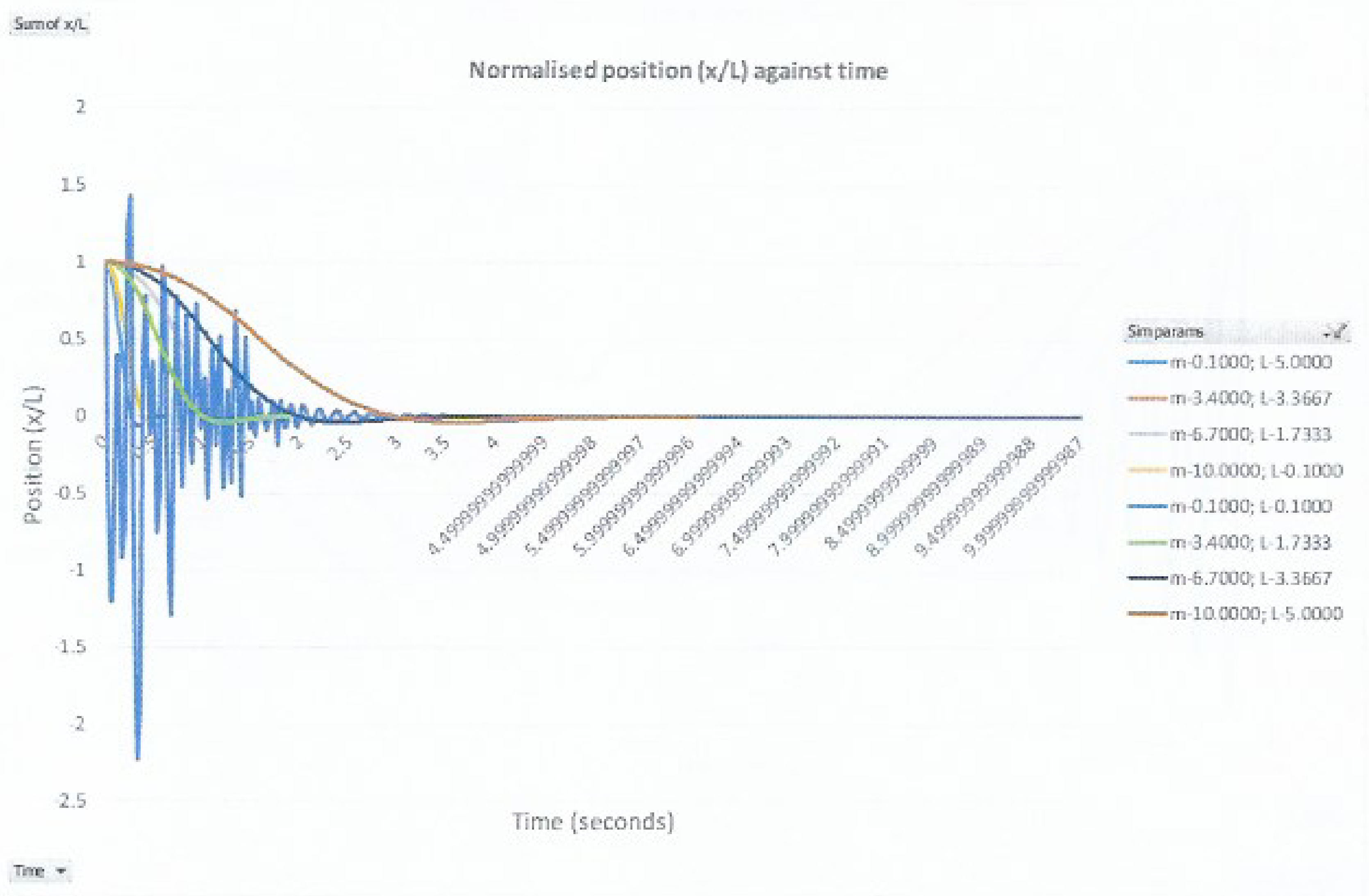
- When noise is included:
  - The optimum algorithm is clearly the normalised 3rd order algorithm. It would appear that 3rd order is better than 1st order.
  - Normalised is generally better than un-normalised. This seems clear for 3rd order. For 1st order, almost all the "learns" using the normalised algorithm produced better results than the "learns" using the un-normalised version. However, the top performers (roughly the top 7) of normalised and un-normalised did produce similar best success times.
- For the noise-free system, it is difficult to say which variations are performing better:
  - All scenarios are able to achieve a similar best success time, except for the normalised first order algorithm. This might seem unexpected, as the normalised algorithm was introduced as an "improvement" on the un-normalised algorithm. However, the charts examine success time only, and it is therefore possible for a normalised algorithm to perform worse than an un-normalised algorithm: the normalising constant is intended to reduce force for low mass / distance combinations so the time to success will increase. The charts above do not show whether severe oscillating forces are being demanded of the actuator. Other data presented below does this.
  - The third order normalised algorithm has the "easier" time of finding an optimum solution as only the top 15 or so of its "learns" are matched by the other variations. This would suggest that the minima in the solution space for this algorithm are more numerous, "wider", or both.

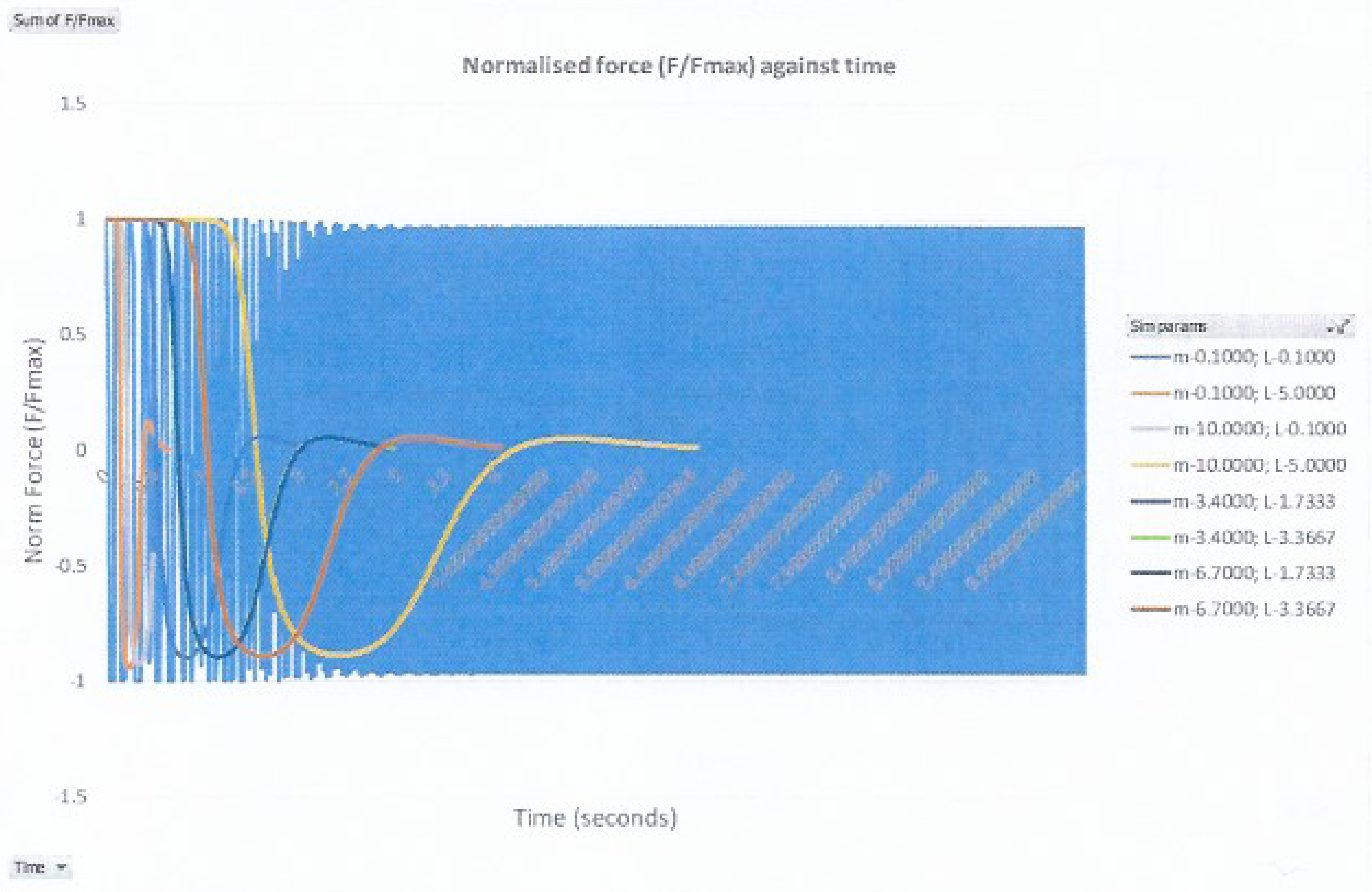
Examination of large oscillatory forces

The charts below show the consequence of including the "normalising mass distance constant". Two sets of charts are shown – one for 3rd order without noise, and one for 1st order without noise. Each set shows with and without the normalising mass distance constant. In both cases the use of the constant reduces the oscillations significantly for the lowest mass / distance combination. The problem of oscillations is most severe in the first order case.

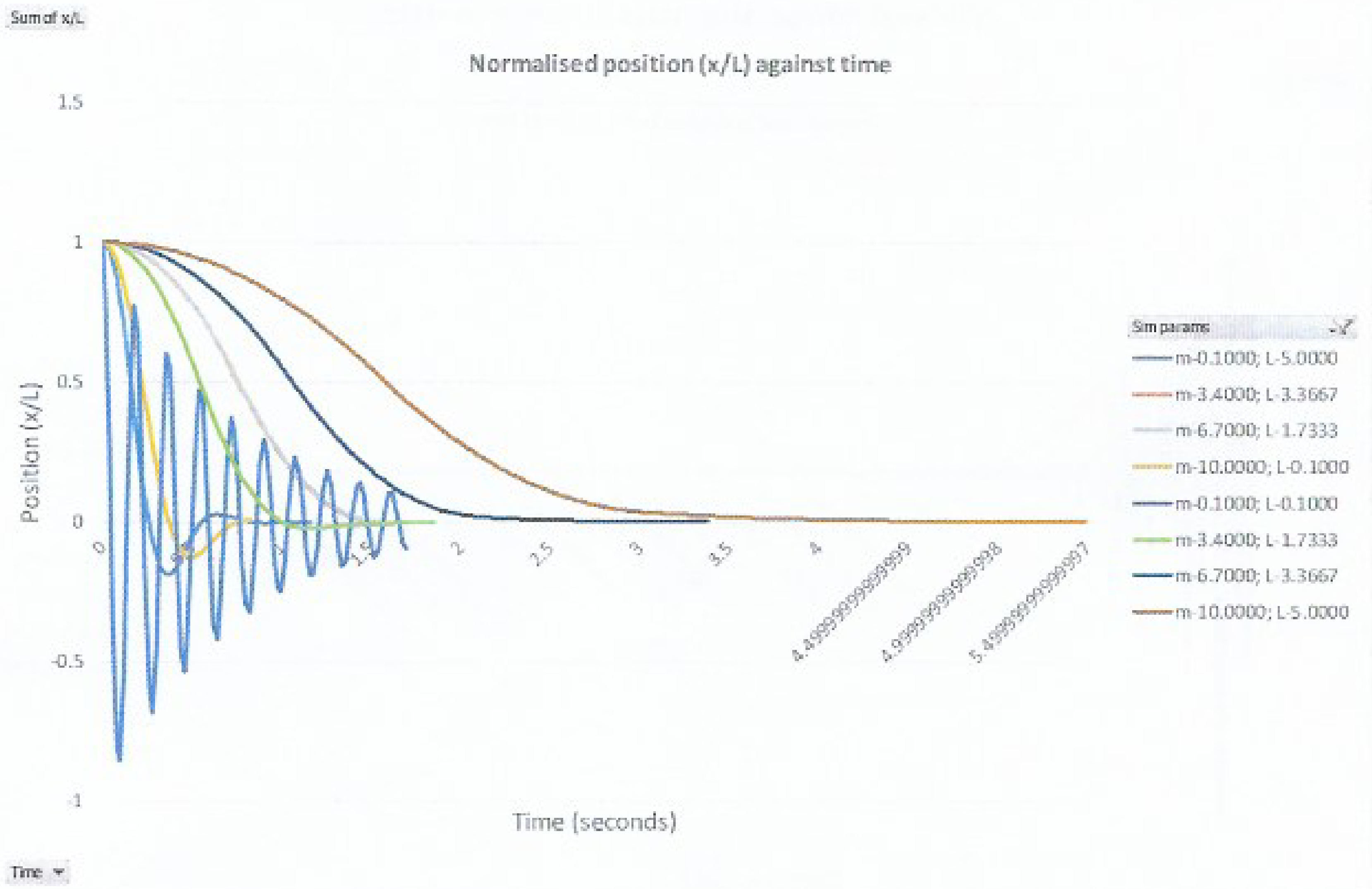
1st order, 20 generations, no noise

Without normalising mass distance constant

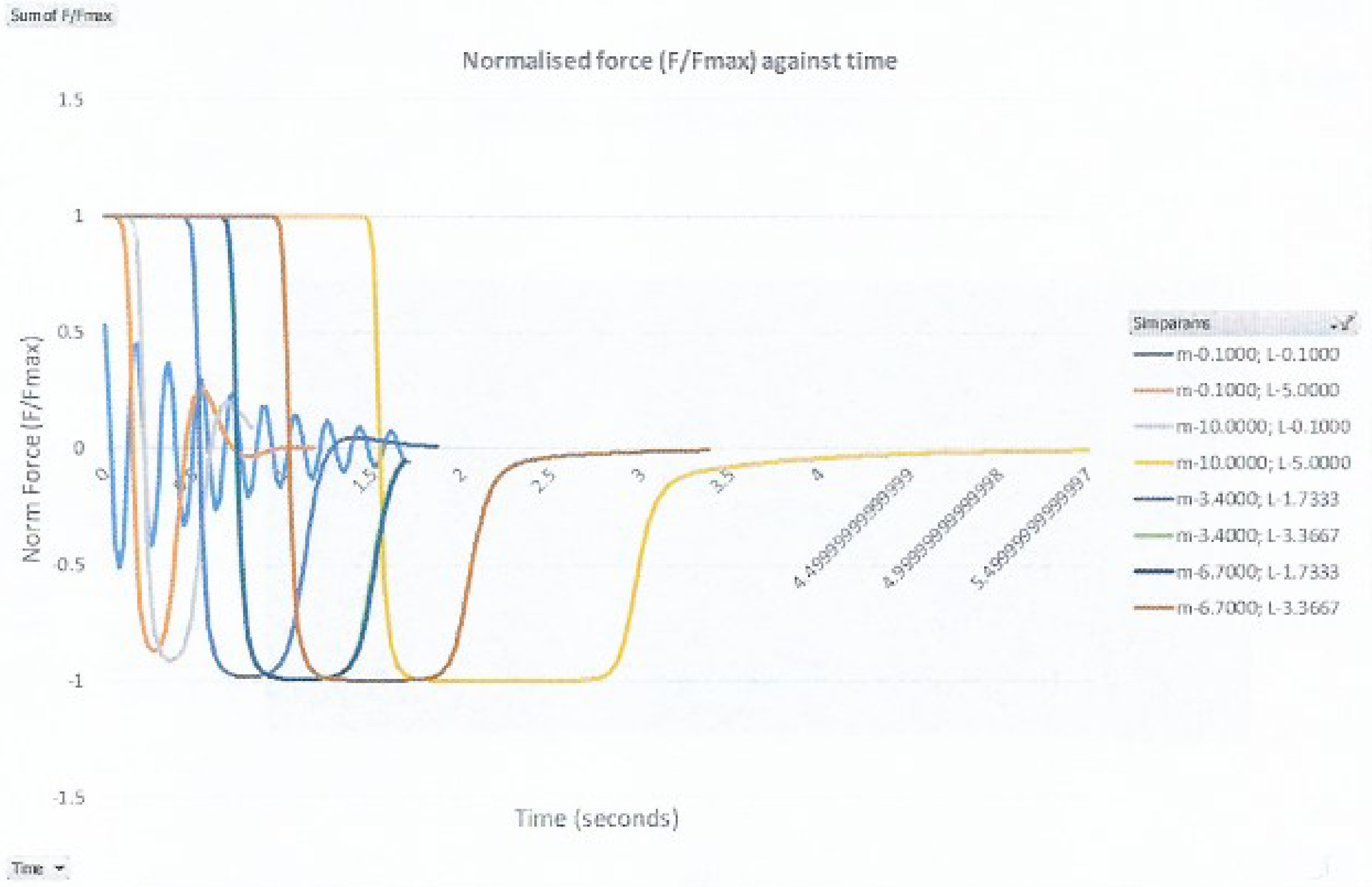




With normalising mass distance constant



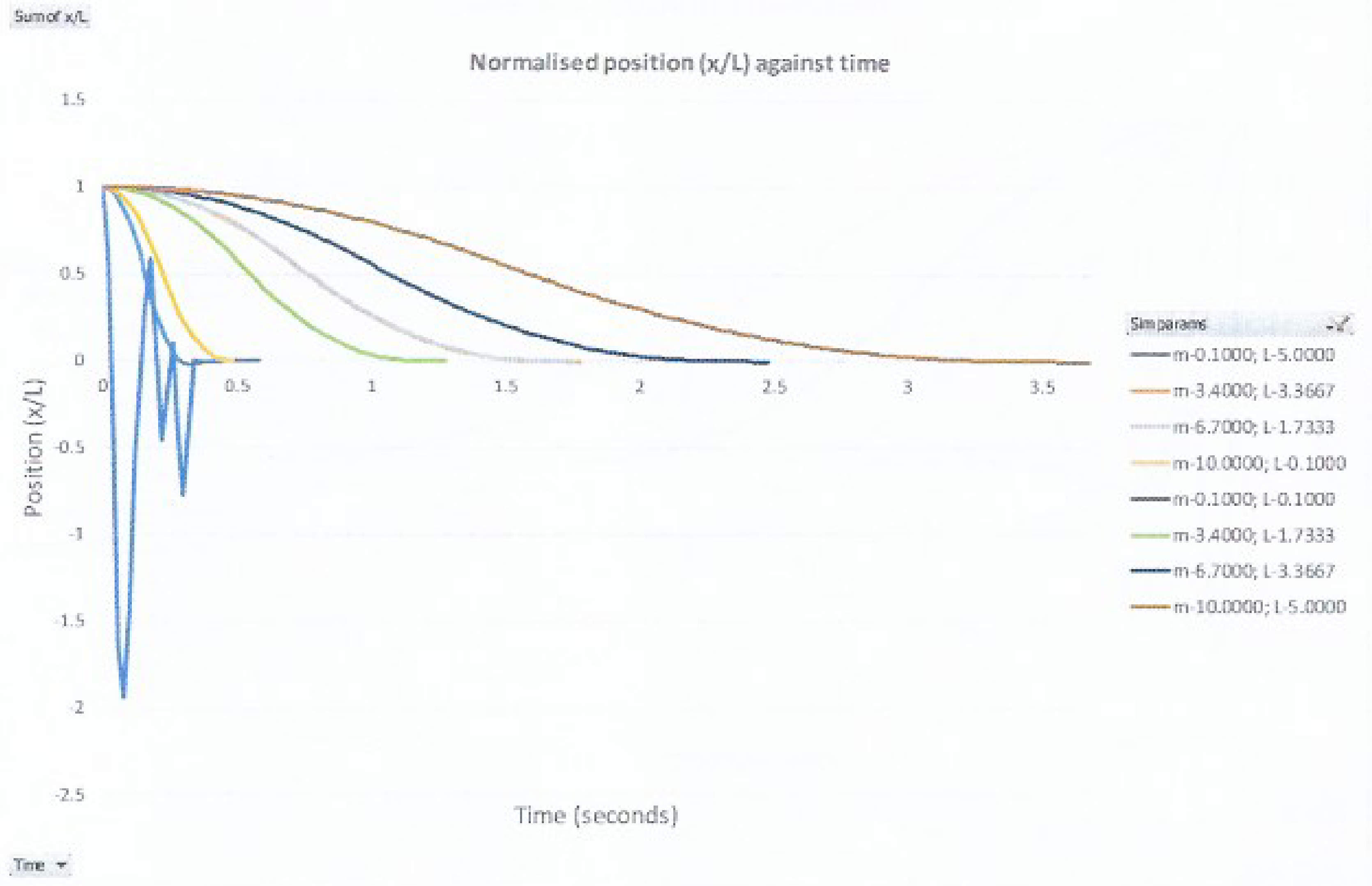
Normalised position (x/L) against time

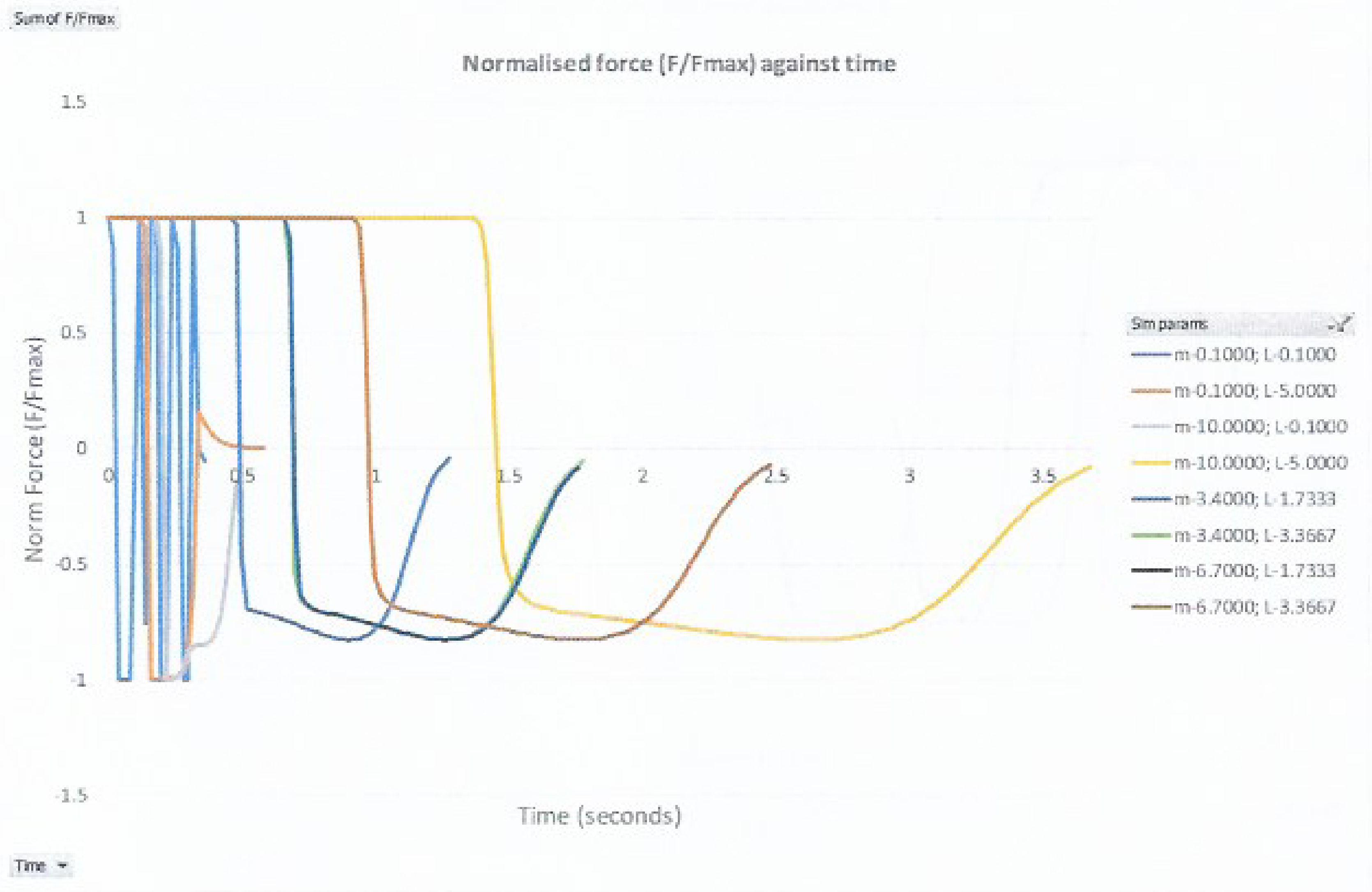


Normalised force (F/Fmax) against time

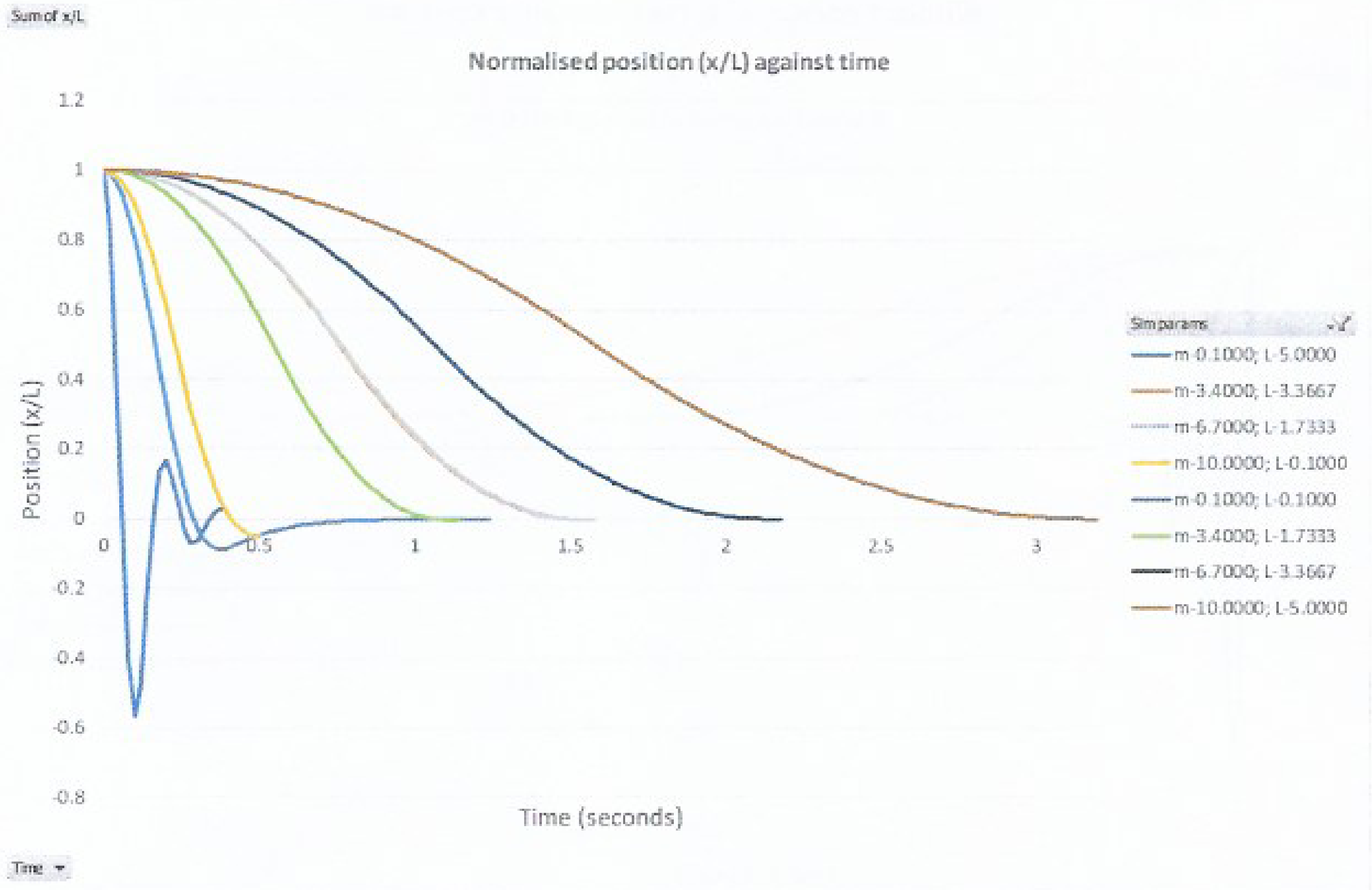
3rd order, 20 generations, no noise

Without normalising mass distance constant

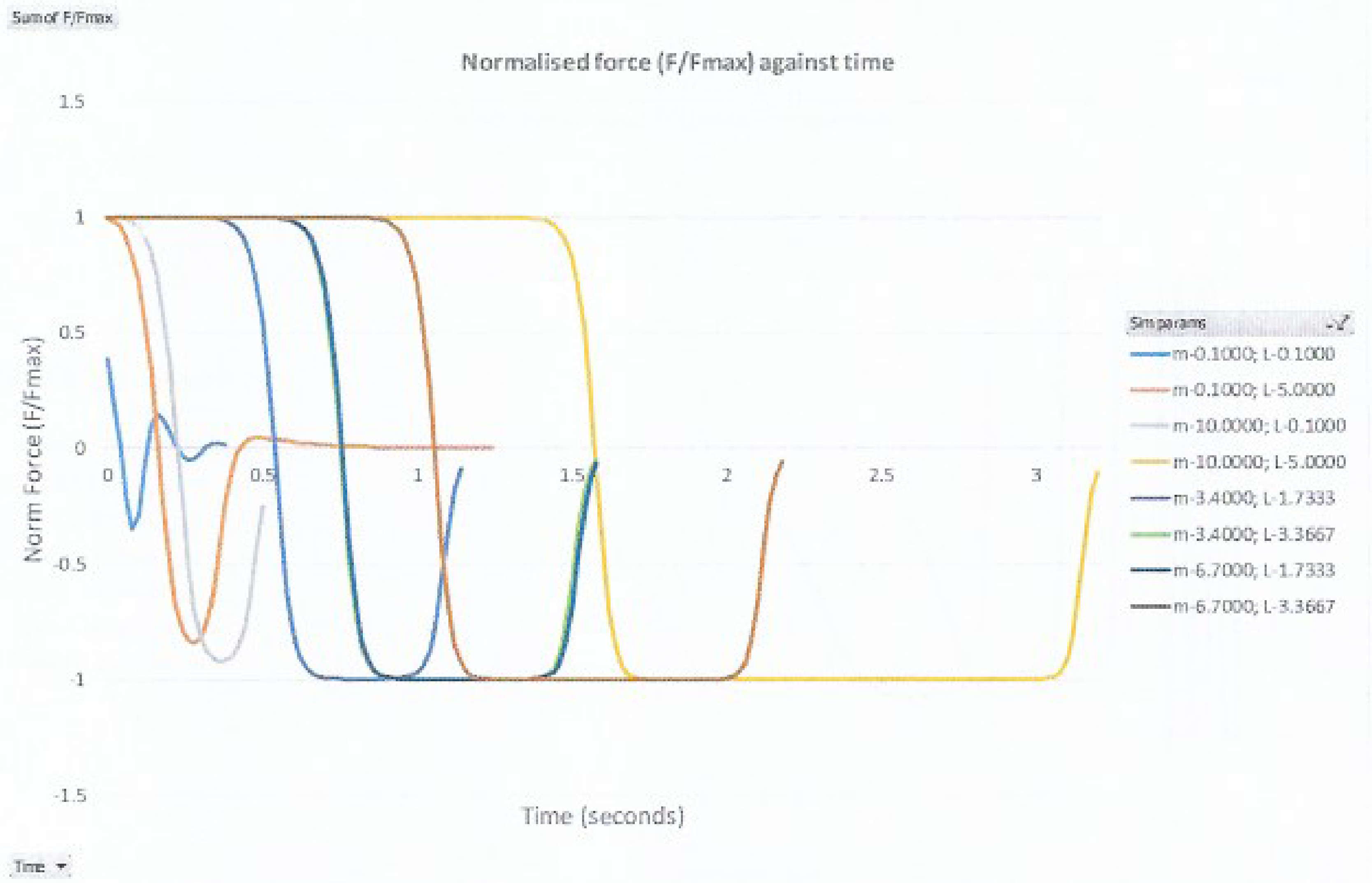




With normalising mass distance constant



Normalised position (x/L) against time



Normalised force (F/Fmax) against time

Secondary Investigation – generic algorithm

The polynomial form of algorithm is already "designed" to suit the particular problem. I wanted to see if I could use a more generic algorithm and, by using an evolutionary learning approach, find a better solution that the polynomial form.

Form of algorithm

I wanted to create a form of algorithm that was flexible enough to reflect any form of algorithm and had the possibility to simplify down to an "elegant" solution that was not a polynomial.

$$p = K_1 X^{a_1} V^{b_1} + K_2 X^{a_2} V^{b_2} + K_3 X^{a_3} V^{b_3} + \ldots + K_n X^{a_n} V^{b_n}$$

where:

- **K**$_i$, **a**$_i$, and **b**$_i$ are all real numbers (not necessarily whole numbers), and can be positive or negative.
- I refer to each $K_i X^{a_i} V^{b_i}$ as a "term".

This form of algorithm therefore allows for division of one variable by another (through negative values of powers) and roots of variables (through powers that are not whole numbers). It does not allow for logarithmic or exponential terms directly, although these could be reasonably approximated by a solution with enough terms.

This algorithm does not work when **a**$_i$ or **b**$_i$ is not a whole number and **X** or **V** respectively is negative, as this would require the calculation of the root of a negative number. So the algorithm was modified so that each term was calculated as $K_i * sign(X) * |X|^{a_i} * sign(V) * |V|^{b_i}$, where the function **sign(C)** returns **-1** where **C<0** and **+1** otherwise.

Evolutionary learning method

The learning method and parameters were the same as for the polynomial algorithm, except that the recombination was for terms only. For the polynomial algorithm, after each tournament the loser's genes were replaced with the winner's genes on a random basis: each of the loser's genes was replaced with 50% probability. For the generic algorithm, each term (i.e., the set of three genes **K**$_i$, **a**$_i$, and **b**$_i$) in the loser was replaced with the corresponding term in the winner with a probability of 50%.

Results

The initial trials of the generic algorithm showed that little no convergence on a solution was occurring within reasonable time limits given laptop computational power.

As a result I decided to modify the initial generation of individuals in the population to give a better starting point. This I did by setting the constraints on their generation so that:

- the first term depends on **X** only (**b**$_1$ is set to 0) and **K**$_1$ is initially positive.
- the second term depends on **V** only (**a**$_1$ is set to 0) and **K**$_2$ is initially negative.

With this modification, solutions were generated, but in all attempts the solutions were worse than those generated by the polynomial algorithm method.

Conclusions

My attempt to find a better form of algorithm failed. This is either because:

- The polynomial form is the best possible form.
- There are one or more forms of algorithm that outperform the polynomial form, but the minima in the solution space corresponding to these algorithms are located in regions that are so small the individuals generated and then evolved through recombination by the learner are highly unlikely to be located closer to a generic algorithm minimum than a polynomial algorithm minimum.

Bibliography

Harvey, I. (1996). *The Microbial Genetic Algorithm.* Unpublished.