SciML 2022 Exam

This is the exam solution for Christian Michelsen in the Ph.D. course SciML 2022 at DTU.

The exam is based on the following problems:

1. Solve a damped harmonic oscillator with known parameters and generate data from the ODE solution
2. Fit the data with a non-damped harmonic oscillator using a neural network to model the missing dynamics
3. Symbolically recover the missing terms of the ODE (i.e. retrieve the original damped ODE)
4. Fit the data from 1) using Turing (but with unknown parameters), and quantify the uncertainties

1: Damped Harmonic Oscillator

The damped harmonic oscillator (DHO) is, as the name suggests, a damped version of the classical harmonic oscillator (HO)

$$m\ddot{x} + b\dot{x} + kx = 0. \tag{1}$$

The dampening is determined by the damping parameter b, the mass is m, and the spring constant is k. Here $\dot{x}$ refers to the first derivative of x with respect to time t, and $\ddot{x}$ refers to the second derivative of x. As such, the DHO is an differential equation governed by Newton's second law (the first term), Hook's law (the last term) and the friction term (the second term).

Without loss of generality, I set $m = 1$ and use unitless variables. Equation (1) can then be transformed to the following system:

$$\begin{aligned} \dot{x} &= v \\ \dot{v} &= -kx - bv. \end{aligned} \tag{2}$$

I solve (2) using `DifferentialEquations.jl` in Julia with the following parameters:

$$\begin{aligned} b &= 0.1 \\ k &= 0.1 \\ x_0 &= 1.0 \\ v_0 &= 1.0 \\ t &\in [0, 40] \end{aligned} \tag{3}$$

where x_0 and v_0 are the initial conditions for x and v. The solution to (1) can be seen in Figure 1 as the solid, blue line.

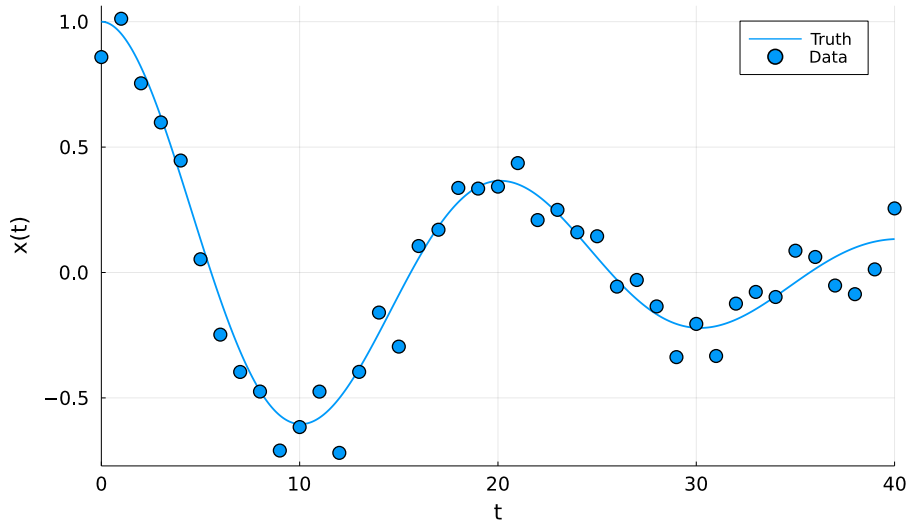

Figure 1: The ODE solution is shown as the solid, blue line, and the simulated measurement data is shown as blue points.

I generate data from the ODE solution by taking equidistant time steps between 0 and 40 such that $\Delta t = 1.0$. On top of this, I add some noise to the data to simulate measurement noise. The blue points in Figure 1 are the data (including noise).

2a: Naive Neural Network

Now, having generated data, I try to fit the data directly with a neural network (NN). That means, that I simply try to fit a neural network to the data without any prior knowledge about the model. The neural network architecture is a classical neural network consisting of three dense layers: $\mathbb{R} \to \mathbb{R}^{64} \to \mathbb{R}$, or, as written in Julia: `Chain(x -> [x], Dense(1, 64, tanh), Dense(64, 1), first)`.

With random initialization, the initial MSE loss is 26.2623. After training (with `Optimization.jl`), it reduces to 0.0679. This seems to be working great, and apparently the neural network has learned how to fit the data quite well. However, if one plots the (naive) neural network solution, it is clear that it has overfit the data, see Figure 2. This is even more apparent in the animations, see the attached file, `animation_NN_no_ode.gif`.

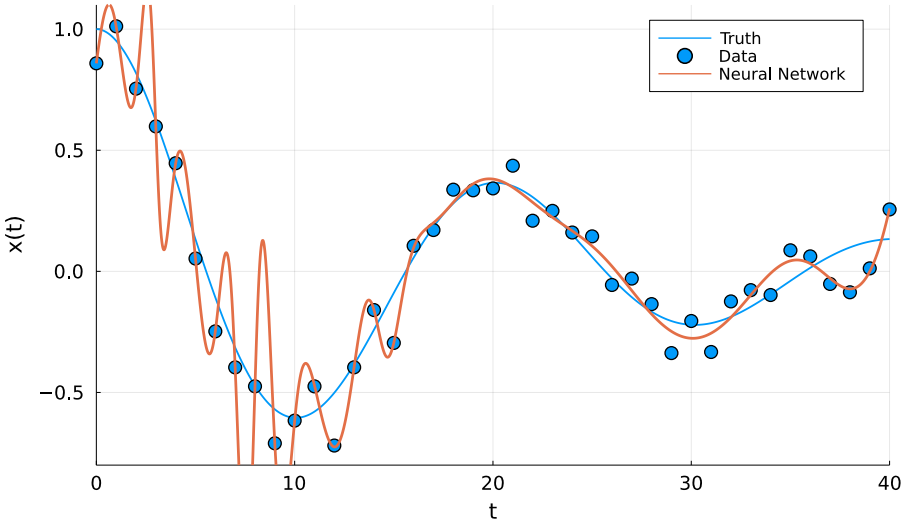

Figure 2: The ODE solution is shown as the solid, blue line, the simulated measurement data is shown as blue points, and the naive neural network solution is shown as the solid, red line.

Another issue with this naive approach, is that the neural network solutions extrapolates very poorly. Compare the extrapolation of the neural network to the true solution, see Figure 3. I know that the solution oscillates around zero and asymptotically approaches it. Yet, the neural network solution increases rapidly away from the true solution – it has lost all predictive power.

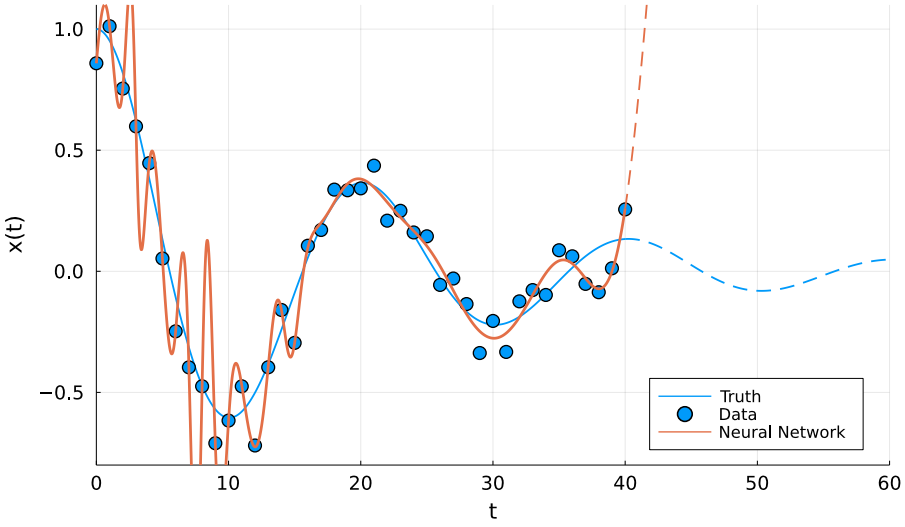

Figure 3: The ODE solution is shown as the solid, blue line, the simulated measurement data is shown as blue points, and the neural network solution is shown as the solid, red line. Extrapolations of the true solution and the neural network solutions are showed as dashed lines.

2b: Neural ODE

Instead if fitting the neural network directly to the data, I can instead combine it with some prior knowledge about the model. In this case, I assume that I already know Newton's second law and Hook's law. As such, it is only the friction term that is missing and what I want the neural network to learn. Using the same form as in (2), I can write this as:

$$\begin{aligned} \dot{x} &= v \\ \dot{v} &= -kx - \mathrm{NN}_\theta(v), \end{aligned} \tag{4}$$

where $\mathrm{NN}_\theta(v)$ is the neural network with parameters θ. Here I have further incorporated the knowledge that the friction does not depend on position, but only velocity. Note that I use a similar same neural network as the one in 3a.

I now optimize the parameters of the neural network (and k), which reduces from a MSE loss of 583.8743 to 0.3275. Even though the loss is higher than for the naive neural network, the solution is a lot better, see Figure 4. This is even more apparent in the animations, see the attached file, `animation_NN_with_ode.gif`.

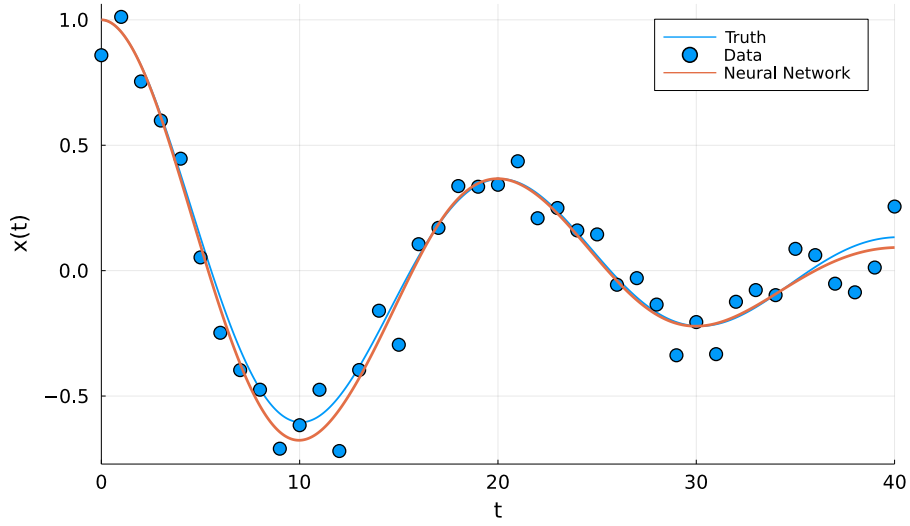

Figure 4: The ODE solution is shown as the solid, blue line, the simulated measurement data is shown as blue points, and the neural ODE solution is shown as the solid, red line.

Not only does this model not overfit, it also extrapolates a lot better, see Figure 5. Compare this to Figure 3 which XXX.

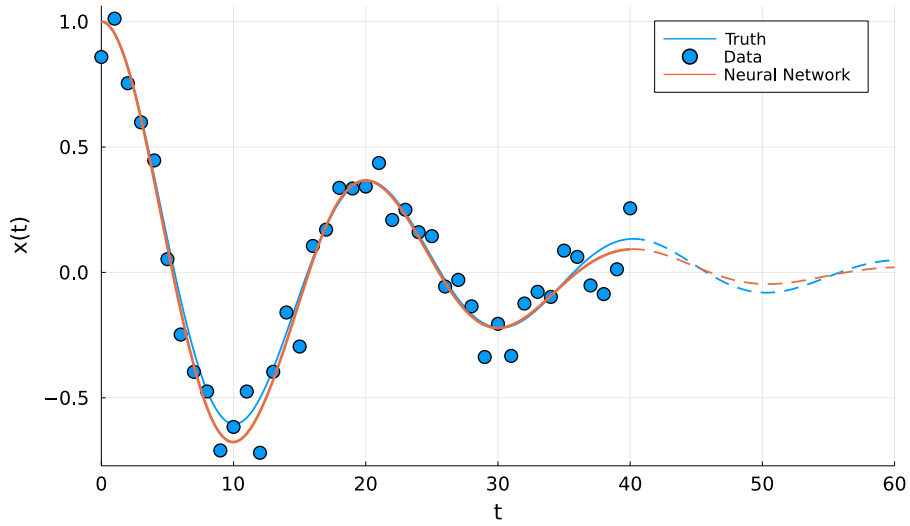

Figure 5: The ODE solution is shown as the solid, blue line, the simulated measurement data is shown as blue points, and the neural ODE solution is shown as the solid, red line. Extrapolations of the true solution and the neural network solutions are showed as dashed lines.

To further understand the neural ODE, I can also compare the true solution to the neural ODE solution in the phase space. This is, instead of showing the solution as a function of time, I show $x(t)$ on one axis and $v(t)$ on the other, see Figure 6. Here I note some discrepencies between the true solution and the

neural ODE solution, however, the neural ODE solution seems to capture the overall shape of the true solution.

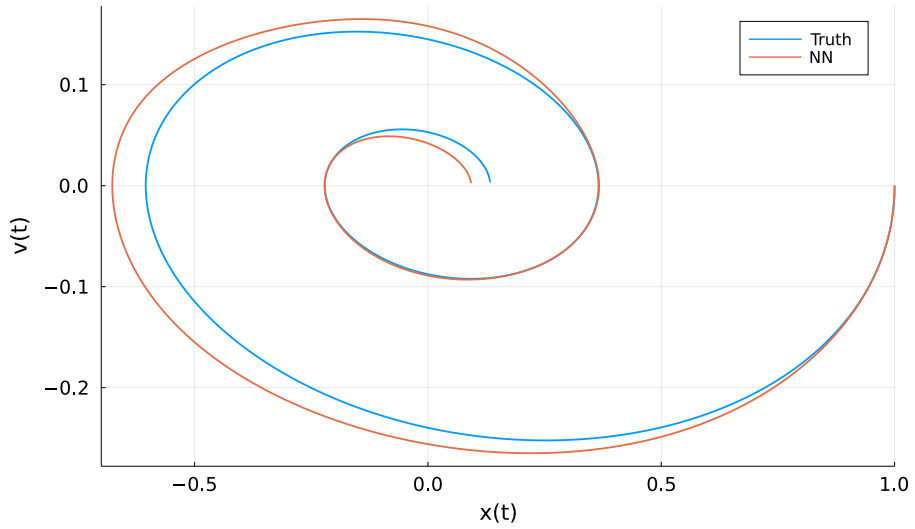

Figure 6: The ODE solution is shown as the solid, blue line, and the neural ODE solution is shown as the solid, red line.

3: Symbolic Regression

Before I do any symbolic regression on the neural network, I first try to visualize it, see Figure 7. At first, it does not seem like any "textbook" function, however, if I only look at the interval from $-0.3 < v < 0.15$, which is the range of values of v that the neural network was actually trained on, see the y-axis of Figure 6, I notice that it is (surprisingly) linear.

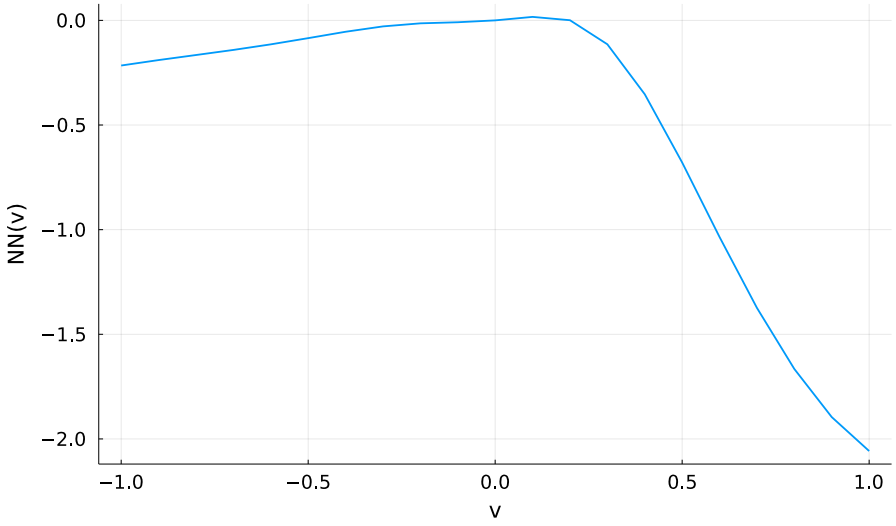

Figure 7: The output of the neural network as a function of velocity, v.

I perform the symbolic regression on the neural network using `DataDrivenDiffEq.jl` where I use the `DirectDataDrivenProblem` function in particular. I use $u[1]$ and $u[2]$ as variables and set up a basis consisting of $\sin(u[1])$, $\sin(u[2])$, $\cos(u[1])$, $\cos(u[2])$ along with polynomial basis functions up to order 5: in total 25 elements. I use the `ADMM` optimizer to introduce sparseness in the solutions (via Lasso). I find that the problem found a solution with a loss of L_2 Norm error of 0.002496. The solution was:

```
Model ##Basis#638 with 1 equations
States : u[1] u[2]
Parameters : p₁
Independent variable: t
Equations
φ₁ = p₁*u[2]
```

The symbolic regression was able to retrieve the correct function form of the missing part of the ODE and correctly found that the solution only depends on $u[2]$ i.e. the velocity and not on $u[1]$, the position:

$$\mathrm{NN}_\theta \approx b * v \tag{5}$$

4: Bayesian Inference

Now that I know the functional form of the ODE, I can use the Bayesian inference to estimate the value of the unknown parameters, b and k, using `Turing.jl`.

I assume that b is a random variable drawn from a normal distribution (truncated to be positive) and similar for k. I first compute the MAP estimate to initialize the MCMCM chains at this point. I then run 4 chains, each for 1000 iterations, using the NUTS sampler, see Figure 8. I also estimate the size of the

noise, σ, which is also shown in this figure. The noise is slightly overestimated, however both b and k matches the true values.

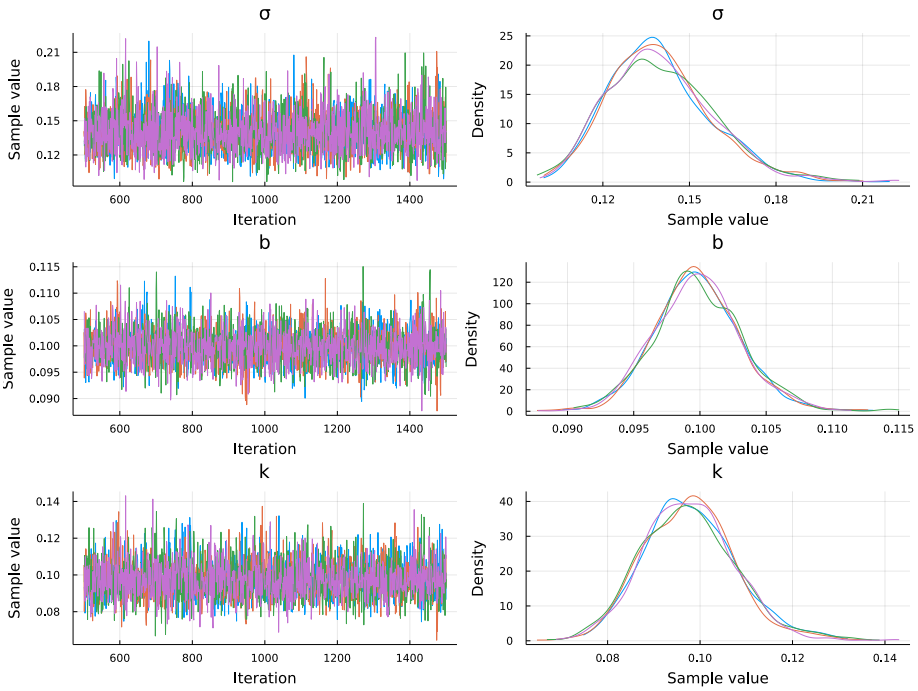

Figure 8: Left) The trace plots for the MCMC chains. Right) The density plots for the MCMC chains.

Having the posterior samples available in the MCMC chains, I also visualize how the uncertainties on the parameters affect the ODE solution in Figure 9. Note that the since I assumed the initial positions to be known without any uncertainties, the $x(t)$ all start at the same value and then slowly increase in uncertainty as a function of time.

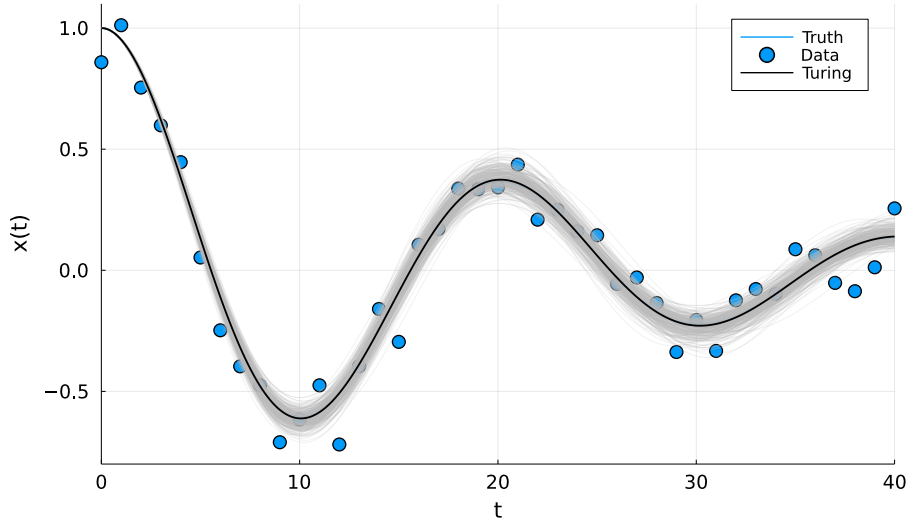

Figure 9: The ODE solution is shown as the solid, blue line, the simulated measurement data is shown as blue points, and the ODE solutions of 300 traces from the posterior distribution are shown grey lines.

Conclusion