Fitting ODE solutions to noisy data by collapsing onto the manifold of solutions

Harry Braviner

July 8, 2025

1 Statement of the problem

We have a collection of datapoints, $\{\tilde{x}(t_i)\}$, and an ODE, $\mathcal{D}[x;t] = 0$. We believe that the datapoints *may be* noisy samples from *some* solution to this ODE, but we do *not* have a guess for the initial or boundary conditions. How can we evaluate the goodness-of-fit of this ODE (which we probably cannot solve analytically, and which we do not have initial conditions for) to these datapoints?

Let's simplify the problem space by making two big restrictions:

1. We only consider only one-dimensional second-order ODEs. i.e. the differential operator $\mathcal{D}[x;t]$ has the form $\ddot{x}(t) - f(x(t), \dot{x}(t), t)$.
2. The datapoints, $\tilde{x}(t)$, are samples from an evenly-spaced discrete grid of times, $t_0, t_1, \ldots, t_{N-1}$, where $t_i - t_{i-1} = h$. The samples themselves need not be evenly spaced: e.g. our dataset could consist of $\{\tilde{x}(t_0), \tilde{x}(t_1), \tilde{x}(t_2), \tilde{x}(t_{50})\}$. We also assume that we have at most one sample from any t_i. i.e. we never have multiple samples from the same time with different noise components.

I am not concerned at this point about where f comes from. The motivation for everything in this document is that I would like to be able to modify PySR to search for such an f. Doing so requires[1] evaluating the goodness-of-fit of each f to the data, and this work is intended to provide a method of doing so.

The essential idea I shall outline is this: We will approximate the (infinite dimensional) space of functions on $[t_{\text{start}}, t_{\text{end}}]$ by an N-dimensional discretization

$$(\hat{x}(t_0), \hat{x}(t_1), \ldots, \hat{x}(t_{N-1}))$$

Within this N-dimensional space there is some subspace (the *solution space*) that satisfies the ODE.[2] **We shall collapse the N-dimensional discretization onto the solution space *along the path minimizing the MSE of the data.*** I shall be more precise about what this means in section 2, after briefly reviewing other work and ideas.

1.1 Previous work

I haven't done a very comprehensive literature review, so I can't rule out that someone has done something similar to myself. However, a little googling hasn't turned up anything identical. I'll review here what I *did* find, since a lot of this is quite interesting, might provide inspiration along different directions, and I want to at least have it written down for my own purposes.

[1]It also requires modifying PySR to output functions of t, $x(t)$, and $\dot{x}(t)$, but I believe that is comparatively straightforward: have PySR treat t, $x(t)$ and $\dot{x}(t)$ as three independent variables.

[2]And for a one-dimensional second-order ODE this subspace should generally be 2-dimensional.

In [8] the authors consider symbolic regression of first-order autonomous dynamical systems:

$$\begin{aligned}
\dot{x}_1(t) &= f_1\left(x_1, \ldots, x_n\right) \\
\dot{x}_2(t) &= f_2\left(x_1, \ldots, x_n\right) \\
&\vdots \\
\dot{x}_n(t) &= f_n\left(x_1, \ldots, x_n\right)
\end{aligned} \tag{1}$$

The candidate system, $(f_1, f_2, \ldots, f_n)$, is the output of a genetic algorithm, although it does not appear to be PySR. The authors use two methods of evaluating the goodness-of-fit of the ODE to the datapoints:

1. Using the approximate derivative values as a target, i.e.

$$\text{Loss}(f) = \sum_{i,j} \bigg(f_i\left(x_1(t_j), \ldots, x_n(t_j)\right) - \big(x_i(t_{j+1}) - x_i(t_j)\big) \bigg)^2$$

2. Integrating from the initial values $(x_1(t_0), ..., x_n(t_0))$ using a numerical ODE solver.

The method of [8] does not appear to allow for noise in the measurements. The results section shows the method being less successful on motion-captured data vs simulated data, and this supports my belief that this method does not tolerate measurement noise well. They also assume that their datapoints are evenly spaced in time, although I'm not sure this is necessary. However, for method (1) to work, it is necessary for the spacing between the datapoints to be such that the numerical gradient is a good approximation to the true gradient.

[5] also appears to integrate the candidate ODE from initial conditions that are assumed to be know. The authors describe a system which not only evaluates candidate dynamical systems, but also suggests new experiments (i.e. initial conditions) that should best disambiguate between currently-strong candidates. While this is a useful capability, and possibly the reason that the authors assume the initial conditions are known (because you set up the experiment), it neglects the fact that in reality we never quite set up the system as we intend to. The paper also assumes that the system is first order.

In [7] it is not clear how the authors obtain initial conditions, but I think that they simply take the same initial conditions used to generate the data and pass these, along with the candidate system, to the ODE integrator. Again, this paper considers first order dynamical systems.

In [6] the authors implement an end-to-end deep learning approach. Input observations (i.e. $(x_1(t_i), \ldots x_n(t_i), t_i)$) are embedded and then passed to a transformer, which outputs $(f_1, \ldots, f_n)$ in the form of a string of symbols representing the tree form of the fs in prefix notation. The model is trained on a randomly generated synthetic dataset. This has the advantage that the target string is known (because it was generated at random), and the input data (the samples) are generated by numerical integration of the f. The authors do consider noisy data and non-uniform spacing ('sub-sampling'). This paper completely sidesteps the issue of how to evaluate the goodness-of-fit without the underlying initial conditions: we do not have to repeatedly evaluate the goodness-of-fit of fs, because we only have a single candidate, the output of the transformer. Unlike PySR, this method does not produce a Pareto front.

The issue [1] on the PySR github repo suggests that, at least as of March 2024, the accepted method for performing SR for an ODE was having PySR generate both the ODE and the initial conditions. This has two issues. First, the additional time taken to search for initial conditions (and all of the problems I will discuss in the next section regarding sweep-and-shoot approaches). Secondly, outputting an ODE and an initial condition allows us to test only a single dataset. We might have multiple timeseries that we believe obey the same ODE with different initial conditions.

In issue [2] Miles refers someone looking to do SR on ODEs to [3], which suggests performing integration of a function by generating data points for the integrand, and then performing SR on candidate expressions for the integral. Could this method be applied to an ODE (by replacing the dataset generated from the

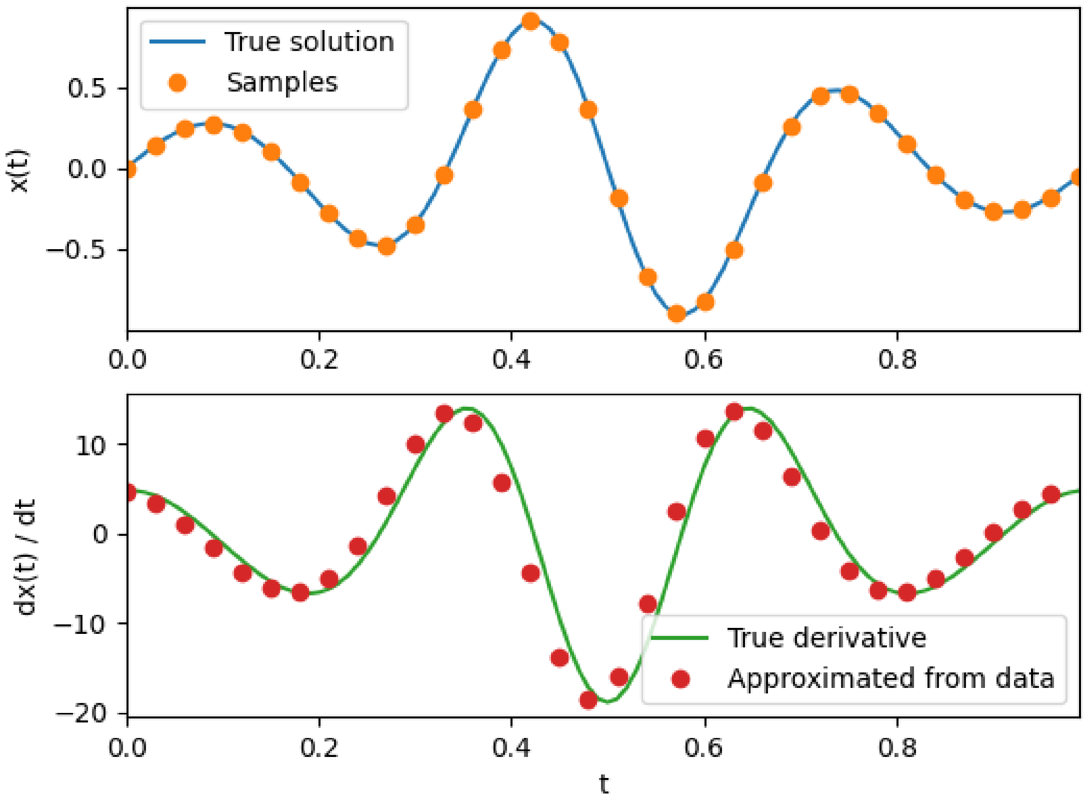

Figure 1: Forward numerical derivative, $(x(t_{i+1}) - x(t_i)) \,/\, (t_{i+1} - t_i)$ from sampled data. There is some error, with the derivative being shifted left, even though the points are densely spaced.

integrand in [3] by the timeseries)? One limitation of doing so is that we are now searching over the expression for the *solution to* the ODE. Many physical systems have a simple expression in terms of ODEs, but no closed form solution to that ODE.

1.2 Alternative methods

In this section I'm going to outline some alternative methods that seem attractive.

1.2.1 Extracting $\dot{x}(t)$ from the data

Several of the papers described above evaluate f directly as the degree of disagreement between $f(x(t))$ and $\dot{x}(t)$ as approximated by forward differences from the data samples. Even if the samples have zero noise, the discretization of the time-sampling can cause considerable error in the approximation (see figure 1), and the addition of noise makes the error much worse (see figure 2). The method I outline in the note should be able to handle much sparser and noisier samples.

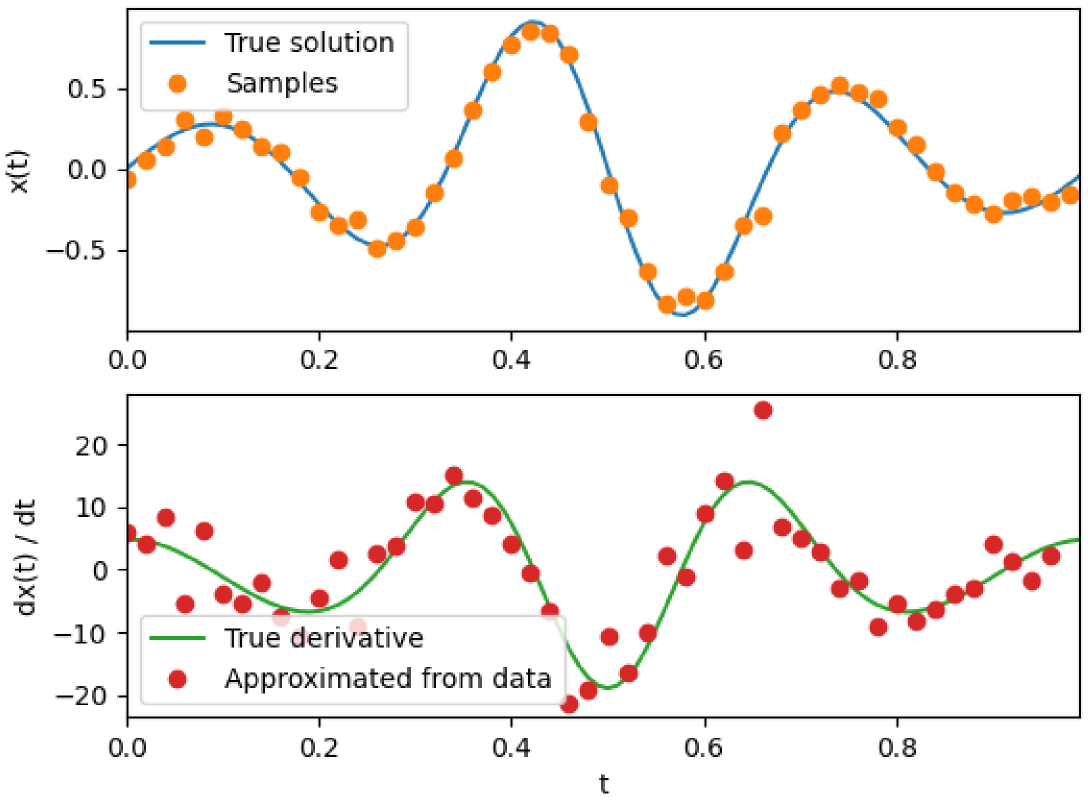

Figure 2: Forward numerical derivative, $(x(t_{i+1}) - x(t_i)) \, / \, (t_{i+1} - t_i)$ from sampled data with Gaussian noise with $\sigma = 0.05$. The error is considerably worse than in figure 1.

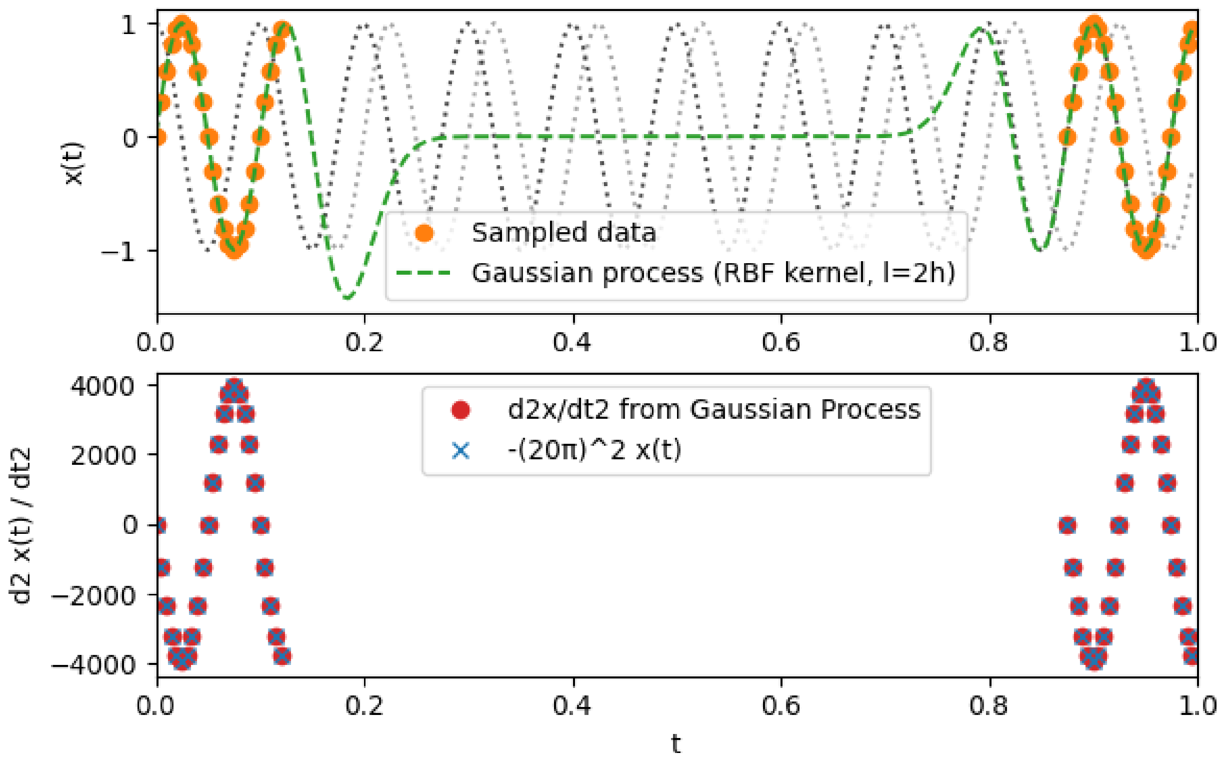

Figure 3: A Gaussian process is fitted through sampled datapoints (top plot), and the second derivative of this process is compared to the right-hand side of the candidate equation $\ddot{x}(t) = -(2\pi 10)^2 x(t)$ (bottom plot) where excellent agreement is seen (the relative error is a few parts-per-thousand). However, the two clusters of datapoints come from solutions of this ODE that have different initial conditions, being 90° out of phase (dotted lines on the top plot).

1.3 Smoothing the data with a kernel

Smoothing the datapoints with some kernel function (e.g. a rolling average window) would go some way toward addressing the issue of noise in the dataset. PySR supports something similar to this feature via the `denoise` option, which first fits a Gaussian process to the data. This could be modified to also output the gradient of the fitted process. One downside of this method is that it requires the choice of hyperparameters for the kernel of the Gaussian process (I don't think PySR exposes these choices to the end user, and I'm not sure if it has some clever way to pick them).

A second, more subtle problem exists with both this method and that of section 1.2.1. If the data sampling is sufficiently irregular, checking that the candidate ODE is obeyed locally is *not* the same as verifying that it is obeyed globally. This is illustrated in figure 3. A Gaussian process is fitted to two clusters of sampled datapoints. Comparing the second derivative of the Gaussian process to the right hand side of $\ddot{x}(t) = -(2\pi 10)^2 x(t)$ gives excellent agreement, and we might conclude that this data comes from a 10Hz harmonic oscillator. But it does not. In the upper plot of figure 3 we also show two solutions for a 10Hz harmonic oscillator, with different initial conditions. It is clear that one cluster of sampled datapoints fits one set of initial conditions, and the other cluster fits the other set. But the whole cluster as a single timeseries does not fit this candidate ODE for any set of initial conditions.

We should note that this is not an artefact of the Gaussian process. This issue would arise even if we had access to measured values of $\ddot{x}(t)$ at the sample points.

One could object that I have cheated by only evaluating the Gaussian process and its second derivative at the sampled points. Should we have compared $\hat{x}$ from the Gaussian process to its second derivative over the whole range? I believe not. Doing so here would show serious divergence from the ODE $\ddot{x} = -(2\pi 10)^2 x$ in the middle of the domain. *But it would also do so if the datapoints were drawn from the same solution.* We would mark *any* candidate ODE as a poor fit to data with large gaps in its time-sampling. The Gaussian process only helps us *interpolate*; it is not reasonable to expect it to also *extrapolate*.

1.4 Sweep-and-shoot methods

If we knew the initial conditions for our dataset, we could integrate from the earliest point using a numerical ODE solver and compare the numerically integrated solution, $\hat{x}(t)$, to the sampled solution $\tilde{x}(t)$, at every later sampling time. This works well if the initial conditions are known with zero noise. In practice, noise may be present, and this method privileges the first point (by treating it as noiseless) in a way that we would probably prefer to avoid. In the case of second order ODEs, we need to either have direct measurement of $\dot{\tilde{x}}$ or use the first two points to approximate it.

In the absence of noiseless initial conditions, we can 'sweep' over plausible values and integrate the ODE forward in time ('shoot') for each set of initial conditions. The noisy values for the first x and $\dot{x}$ could be used as a point around which to centre the sweep.

This approach is viable for a low dimensional problem, but will become exponentially expensive as the number of dimensions increases. If we choose to sweep over 10 values for each initial condition for a 3 dimensional second order ODE (x, $\dot{x}$, y, $\dot{y}$, z, $\dot{z}$) we would have 10^6 combinations to try.

Even in low-dimensional problem, sweep-and-shoot can struggle to evaluate whether the ODE is a good fit to the data. Consider the system

$$\ddot{x}(t) = -x(t)^3 + a^2 x(t) - \nu \dot{x}(t)$$

This system is a double-well potential with stable fixed points at $x = \pm a$ (with small oscillation frequencies of $\sqrt{2a^2 - \nu^2/4}$) and an unstable fixed point at $x = 0$ (with $x(t) \approx A_0 \exp(a(t - t_0)$ around this point).

Figure 4 shows three solutions to this system, for very similar initial conditions. Suppose that our experiment had evolved along the dashed, orange curve ($x_0 = 10^{-11}$, $v_0 = 0$), and we had samples from it. If we were performing a sweep-and-shoot search, we might well have integrated from $x_0 = 10^{-3}$, produced the blue curve, and measured this to have high MSE relative to our data. We would then step our initial condition to our next point, $x_0 = -10^{-3}$, and produce the green curve, having an even higher MSE. We would reject the candidate ODE, despite it being completely correct!

One might object to this, saying that clearly the sweep should be performed in log-space since we are so close to $x = 0$. This is true, but we know this because we are applying our intelligence and knowledge of the underlying ODE. The location of an unstable fixed point will not always be at zero. To first locate the fixed points of the candidate ODE and treat them as special in a sweep-and-shoot procedure would be considerable additional algorithmic complexity. (In fact, taking the initiative to do so might be a good desideratum of an AI physicist!)

1.5 Relaxation methods

The method I describe below will look a bit like a relaxation method (see [4]), and one might wonder if some variation of this could be used to achieve faster convergence. This might be the case, but if so I haven't yet figured out how.

The relaxation method seems to require fixed, known end-points. i.e. it requires boundary conditions. We could take the earliest and last sampled datapoints, and use these as boundary conditions. However, that would (just like shooting from the first point) elevate these points in significance by assuming that they have no measurement noise. We would also have a version of the problem indicated in figure 4: the blue and orange curves have extremely similar boundary conditions if t_{start} is sufficiently early, t_{end} significantly late.

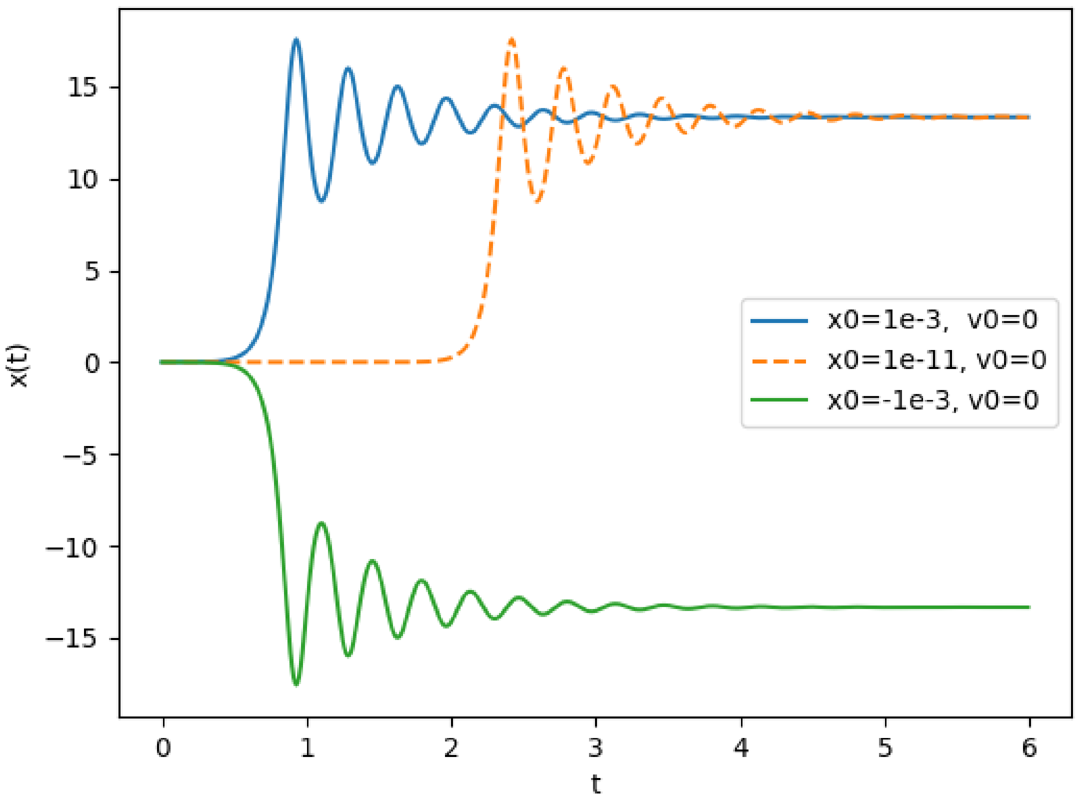

Figure 4: Three (numerically integrated) solutions to $\ddot{x}(t) = -x(t)^3 + a^2 x(t) - \nu \dot{x}(t)$ for $a = 6\pi/\sqrt{2}$ and $\nu = 3$. Note that the initial conditions have identical velocities and differ very small values of the x coordinate.

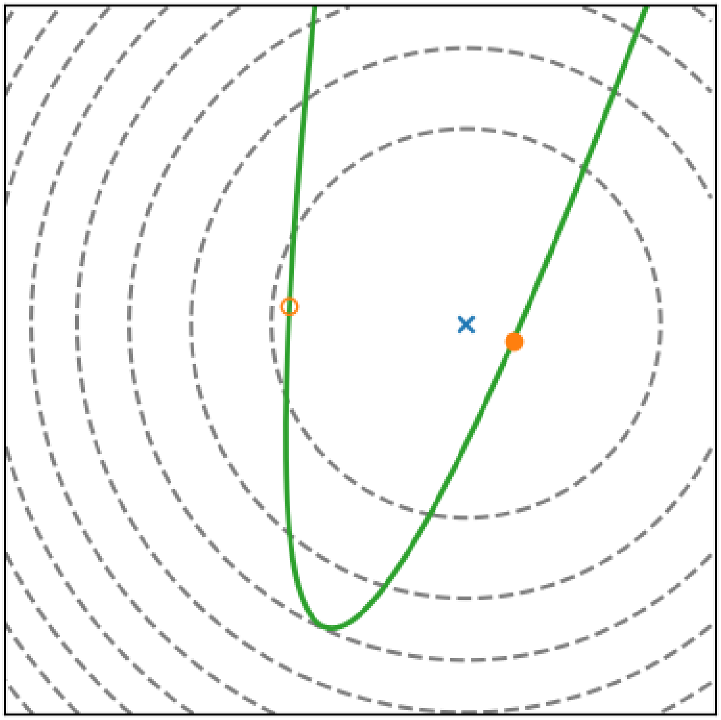

Figure 5: Schematic representation of discretized solution space. Gray dashed curves are contours lines of the data loss, L_{data}. The solid blue dot shows the (global and only) minimum of L_{data}. The heavy green curve is the solution space, and the orange circles denote the minima of L_{data} restricted to the solution space (the hollow circle is a local minimum, the solid circle a global minimum).

2 The Collapse Method

There are two different ways to think of the problem of how well an ODE fits a sampled dataset. Every paper I have been able to find on this subject implicitly does one or the other, but none of them seem to discuss the fact that the other viewpoint exists, or even explicitly acknowledge their choice.

1. Fit the data exactly and measure the extent to which the ODE is violated. e.g. $L_{\text{ODE}} = \sum_i \left(\dot{\tilde{x}}(t_i) - f(\tilde{x}(t_i), \dot{\tilde{x}}(t_i), t_i)\right)^2$. This is the approach of most papers that approximate the derivative by finite differences between datapoints (though, as we discussed in section 1.3, you could fit a smoothing function instead).

2. Insist on the ODE being obeyed, and measure the error of the data relative to the solution, e.g. $L_{\text{data}} = \sum_i \left(\hat{x}(t_i) - \tilde{x}(t_i)\right)^2$. This is the approach of papers where the candidate ODE is numerically integrated. It requires some way of obtaining initial conditions.

The second approach (satisfying the ODE and evaluating the error in fitting the data) feels better-defined. It is unclear that the extent-of-violation of an ODE is well-captured by the squared-error of the difference between the two sides of the equation.

However, there are pitfalls with this. It may be difficult to find initial conditions, as discussed in section 1.4. Even if we were able to backpropagate through the ODE solver to compute the gradient of L_{data} with respect to our initial conditions, the loss function restricted to the solution-space may not be convex. This is illustrated in figure 5: if we start near the local minimum of the solution space, we will never find the global minimum by any method of small steps.

What I have therefore implemented is approach (1) followed by approach (2).

We begin by discretizing the space of functions on $[t_0, t_{N-1}]$ as a vector

$$(\hat{x}(t_0), \hat{x}(t_1), \ldots, \hat{x}(t_{N-1}))$$

which for notational convenience I will call

$$(\hat{x}_0, \hat{x}_1, \ldots, \hat{x}_{N-1})$$

from here on. We require that these grid-points be evenly spaced, and denote the step-size as $h = t_i - t_{i-1}$.

Remember that we have measurements from only some of these times. Let $\mathcal{I}$ denote the set of indices such that we have a measurement $\tilde{x}(t_i)$ at time i. Again, for notational convenience I will call these measurements $\tilde{x}_i$ from here on. These measurements may contain noise.

We now define two loss functions:[3]

$$L_{\text{data}} = \frac{1}{2}\frac{1}{|\mathcal{I}|}\sum_{i\in\mathcal{I}} (\hat{x}_i - \tilde{x}_i)^2 \tag{2}$$

$$L_{\text{ODE}} = \frac{1}{2}h\sum_{i=1}^{N-2}\left(\hat{\ddot{x}}_i - f\left(\hat{x}_i, \hat{\dot{x}}_i, t_i\right)\right)^2 \tag{3}$$

$$\text{where } \hat{\ddot{x}}_i = h^{-2}\left(\hat{x}_{i-1} - 2\hat{x}_i + \hat{x}_{i+1}\right) \tag{4}$$

$$\text{and } \hat{\dot{x}}_i = h^{-1}\left(\hat{x}_{i+1} - \hat{x}_i\right) \tag{5}$$

Note the range of the sum in (3): we could not extend this to $i = 0$ or $N - 1$, since we would not have the points $\hat{x}_{-1}$ and $\hat{x}_N$ needed to define the second derivative. But this should not concern us. Imagine that our task was to set L_{ODE} to zero. Each term must vanish individually, so the sum gives us $N - 2$ equations in N unknowns $(\hat{x}_0, \ldots, \hat{x}_{N-1})$. For most choices of f we will not be able to solve these equations, but for $N - 2$ equations in N unknowns we expect that, if a solution space exists, it will be 2 dimensional. And that is exactly what we expect for a second-order ODE.

We begin by minimizing L_{data} (an easy task, since this is a convex loss function with constant curvature), and then minimize L_{ODE}. In practice we do this by gradient descent, minimizing

$$L_{\text{total}} = (1 - W_{\text{ODE}})L_{\text{data}} + W_{\text{ODE}}L_{\text{ODE}}$$

and varying W_{ODE} from close to zero at the start training to close to one at the end of training. Schematically, we first proceed to the blue cross (exactly fitting the data) in figure 5, and then to the green curve (the solution space). Due to going via the blue cross, we should arrive at the solid yellow circle (the global minimum of L_{data} on the solution space).

Since we are collapsing the vector onto the lower-dimensional solution space, I will refer to this method as *the collapser.*

2.1 Reducing the stiffness of the optimization problem

After implementing this method I initially tried it on an extremely simple system

$$\ddot{x}(t) = F$$

[3]We could instead use the backward derivative, $\hat{\dot{x}}(t_i) = h^{-1}\left(\hat{x}(t_i) - \hat{x}(t_{i-1})\right)$. I do not think this should make a difference, but I have added an option to do so in my code.

for constant F.

The method described above struggled to produce a good fit to this system at all. Simple gradient descent required incredibly small step sizes ($\sim 10^{-7}$) otherwise NaNs would appear in the solution. Reducing the step size sufficiently that this no longer occurred resulted in solutions that satisfied the ODE, but were a poor fit to the data, even with zero noise.

At first I suspected the system was getting stuck in a local optimum, but this isn't possible - with constant acceleration the discretized system is convex. A little further investigation revealed that the problem was likely due to the condition number[4] of the Hessian of L_{ODE} being very large.[5]

Since this simple version of the problem (constant acceleration) is the optimization of a quadratic function, I switched to `LBFGS`, an approximate quadratic optimizer provided by PyTorch that does not require explicitly computing second order derivative. This partially solved the problem, in that it worked well for small numbers of grid points (10), but still worked very poorly on moderately sized grids (100 points). In the latter case the ODE would typically be satisfied, but the solution would visually be a long way from the sample points.

If we directly compute the Hessian of L_{ODE} for this problem, we find

$$\left[\frac{\partial^2(h^3 L_{\text{ODE}})}{\partial \hat{x}_i \partial \hat{x}_j}\right] = \begin{pmatrix} 1 & -2 & 1 & 0 & 0 & 0 & 0 & \cdots & 0 \\ -2 & 5 & -4 & 1 & 0 & 0 & 0 & \cdots & 0 \\ 1 & -4 & 6 & -4 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & -4 & 6 & -4 & 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \cdots & 0 & 1 & -4 & 6 & -4 & 1 & 0 \\ 0 & \cdots & 0 & 0 & 1 & -4 & 6 & -4 & 1 \\ 0 & \cdots & 0 & 0 & 0 & 1 & -4 & 5 & -2 \\ 0 & \cdots & 0 & 0 & 0 & 0 & 1 & -2 & 1 \end{pmatrix}$$

For 10 dimensions, this matrix has a condition number of

$$\frac{\lambda_{\max}}{\lambda_{\min,\neq 0}} \approx \frac{7.55}{0.0253} \approx 299\,.$$

For 100 dimensions, the condition number is

$$\frac{\lambda_{\max}}{\lambda_{\min,\neq 0}} \approx \frac{8.00}{2.50 \times 10^{-6}} \approx 3.2 \times 10^6\,.$$

At this point I decided to try switching coordinates, defining

$$\begin{aligned} z_0 &= x_0 \\ z_i &= x_i - x_{i-1} \,\text{for}\, i \geq 1 \end{aligned} \tag{6}$$

i.e. we work in gradient space, rather than position space.

In these coordinates the loss becomes

$$L_{\text{ODE}} = \frac{1}{2} h \sum_{i=1}^{N-2} \left(h^{-2}(z_{i+1} - z_i) - F\right)^2$$

[4]The Hessian of L_{ODE} actually has two zero eigenvalues (corresponding to the tangent space of the solution space). When I say 'condition number' I mean the ratio of the largest eigenvalue to the smallest non-zero eigenvalue.

[5]During my time at Layer 6 I used to ask prospective colleagues an interview question about the maximum learning rate they could use on a loss function $L(x, y) = \frac{1}{2}\left(ax^2 + by^2\right)$ for $a \gg b$. I suspect that me spending several hours debugging what amounts to a somewhat obfuscated version of the same problem is down to karma.

and the Hessian becomes

$$\left[\frac{\partial^2(h^3 L_{\mathrm{ODE}})}{\partial z_i \partial z_j}\right] = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & -1 & 0 & 0 & \cdots & 0 \\ 0 & -1 & 2 & -1 & 0 & \cdots & 0 \\ 0 & 0 & -1 & 2 & -1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \cdots & -1 & 2 & -1 & 0 & 0 \\ 0 & \cdots & 0 & -1 & 2 & -1 & 0 \\ 0 & \cdots & 0 & 0 & -1 & 2 & -1 \\ 0 & \cdots & 0 & 0 & 0 & -1 & 1 \end{pmatrix}$$

This is considerably more benign. For 100 dimensions, the condition number of the Hessian with respect to the z coordinates is

$$\frac{\lambda_{\mathrm{max}}}{\lambda_{\mathrm{min},\neq 0}} \approx 4.0 \times 10^3 .$$

In figure 6 we can see that the condition number with this transformation does continue to increase with the number of dimensions (i.e. with finer discretization of the time domain), but is clearly less stiff than the untransformed problem.

In these coordinates, with a grid size of $N = 100$, the optimization process appears to work. I have not tested larger grid sizes yet. One question that I do not have an answer to is whether there is a better coordinate transformation that could be applied. My implementation uses the transformation (6) by default, but does allow for any other linear transformation to be specified.

3 Qualitative results

Does this method (plus the coordinate transformation) actually produce discretizations that both satisfy the ODE and fit the data?

To test this, we generate some data from an underlying ODE[6] and sample a few points from this curve, and add noise. We pass these noisy points, and the ODE (but *not* the initial conditions) to our method. We show in figures 7, 8, 9 and 10 that it does indeed produce good fits to the underlying curves. The results in figure 10 are particularly gratifying, since this ODE does not have an analytic solution.

While these results are gratifying, remember that they are a necessary condition, but not a sufficient one, for this method to be useful to our purposes. We want to evaluate the goodness-of-fit of the ODE to the data. If we pass the *wrong* ODE to the collapser (i.e. one that differs from the ODE used to generate the data), we want to collapser to prefer satisfying the ODE over matching the data. In figure 11 we demonstrate this. We generate data from the same equation (a harmonic oscillator plus quadratic drag) as in figure 10, but give a different ODE to the collapser. The collapser clearly outputs a solution to the simple harmonic oscillator equation, fitted to the data as best it can. The difference shows up quantitatively too: in figure 10 $L_{\mathrm{data}} = 5.9 \times 10^{-3}$, and in figure 11 $L_{\mathrm{data}} = 5.3 \times 10^{-2}$. This is promising, as it suggests that this method can reject ODEs that are a poor fit to the data.

All of the above results were generated using the notebook `demo_notebooks/demo_use_of_libraries.ipynb`.

3.1 Large gaps in timeseries

Recall that in section 1.3 we constructed an example (see figure 3) in which it was possible to fit the data well while satisfying the ODE locally, but not globally. Will our collapser method handle this correctly, or will it build up small violations of the ODE in the region that is far from any sampled datapoint? In figure 12 we run this experiment (see notebook `demo_notebooks/mismatched_oscillators.ipynb` to reproduce this) and find that the collapser behaves as desired: it enforces the ODE, at the cost of fitting the data. In particular, it does not allow any form of 'phase drift' in the gap between observations.

[6]Either using the analytic solution, or the ODE solver from SciPy.

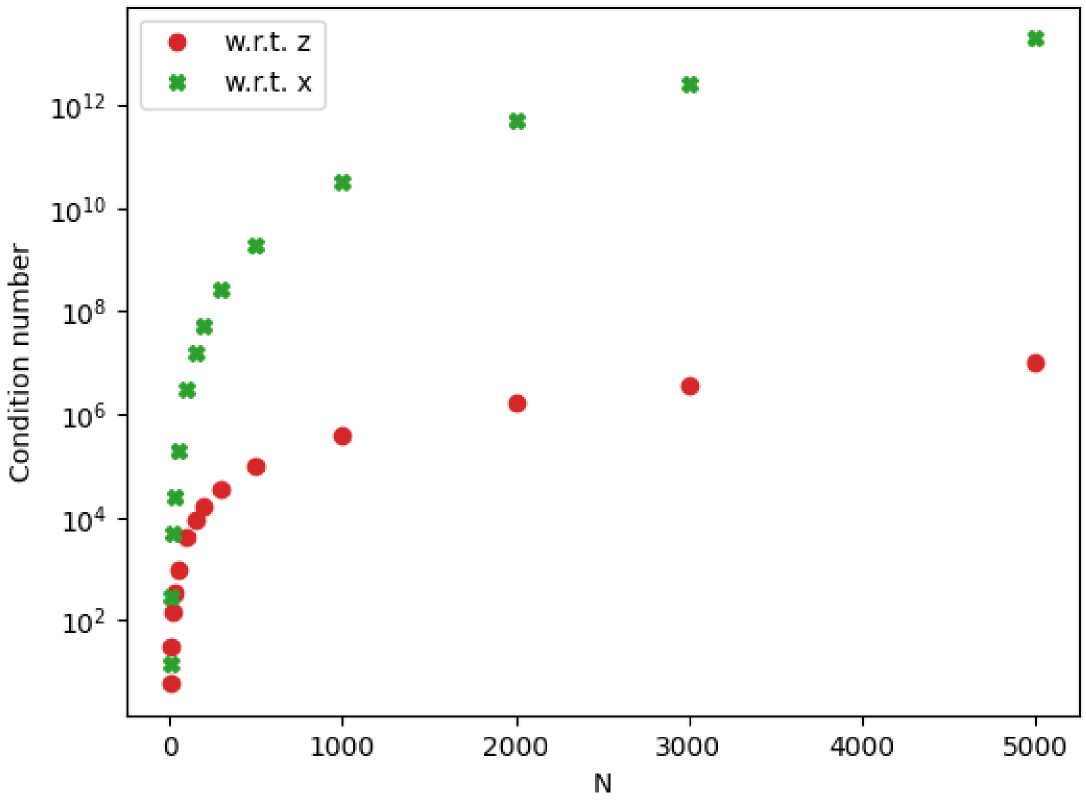

Figure 6: Condition number, $\lambda_{\max}/\lambda_{\min,\neq 0}$, for the Hessian of L_{ODE} with respect to the original variables, $\hat{x}_i$, and the z_i defined by (6).

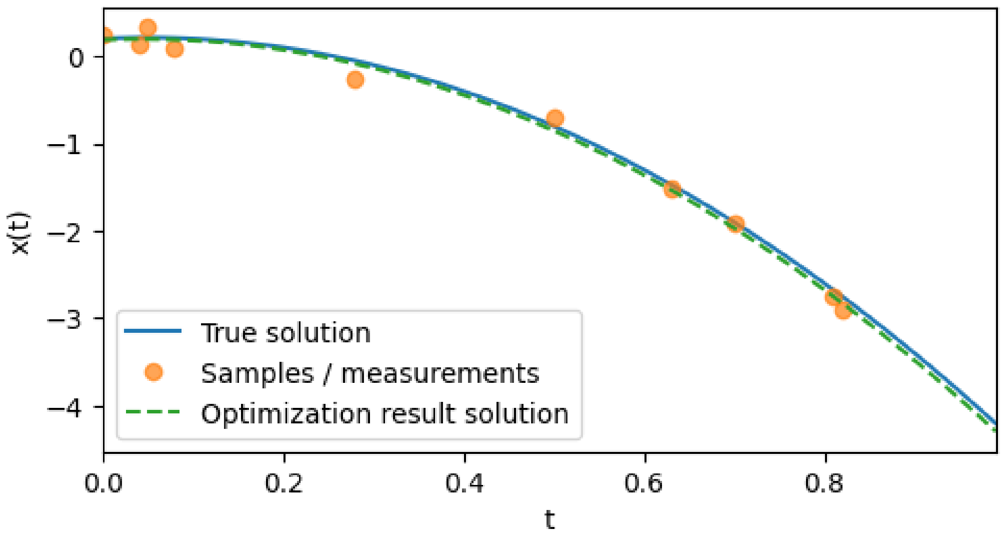

Figure 7: Solution to the ODE $\ddot{x}(t) = -10$ with $x0 = 0.2$ and $\dot{x}(0) = 0.5$ (blue curve), and a small number of samples (orange dots) plus Gaussian noise with $\sigma = 0.1$. The output of the collapser method is shown as the dashed green curve. The collapser has no access to the initial conditions that were used to generate the blue curve.

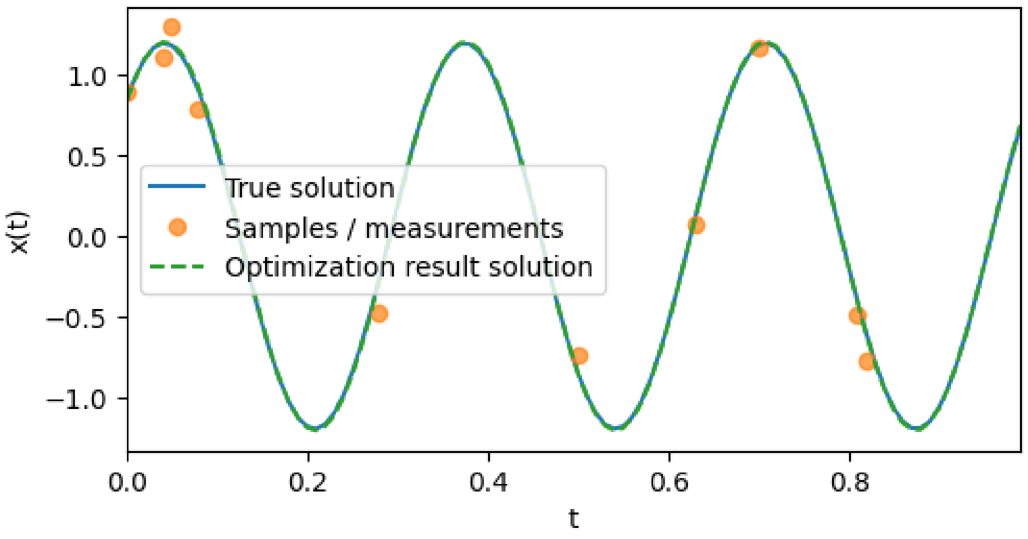

Figure 8: Solution to the ODE $\ddot{x}(t) = -(6\pi)^2 x(t)$ with initial amplitude 1.2 and phase $pi/4$ (blue curve), and a small number of samples (orange dots) plus Gaussian noise with $\sigma = 0.1$. The output of the collapser method is shown as the dashed green curve. The collapser has no access to the initial conditions that were used to generate the blue curve.

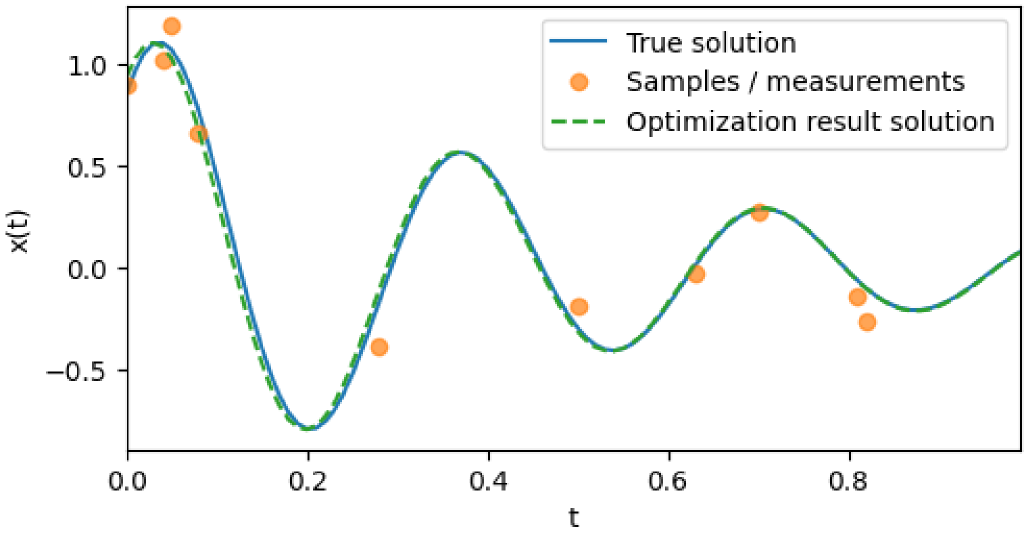

Figure 9: Solution to the ODE $\ddot{x}(t) = -(6\pi)^2 x(t) - 4\dot{x}(t)$ with initial amplitude 1.2 and phase $pi/4$ (blue curve), and a small number of samples (orange dots) plus Gaussian noise with $\sigma = 0.1$. The output of the collapser method is shown as the dashed green curve. The collapser has no access to the initial conditions that were used to generate the blue curve.

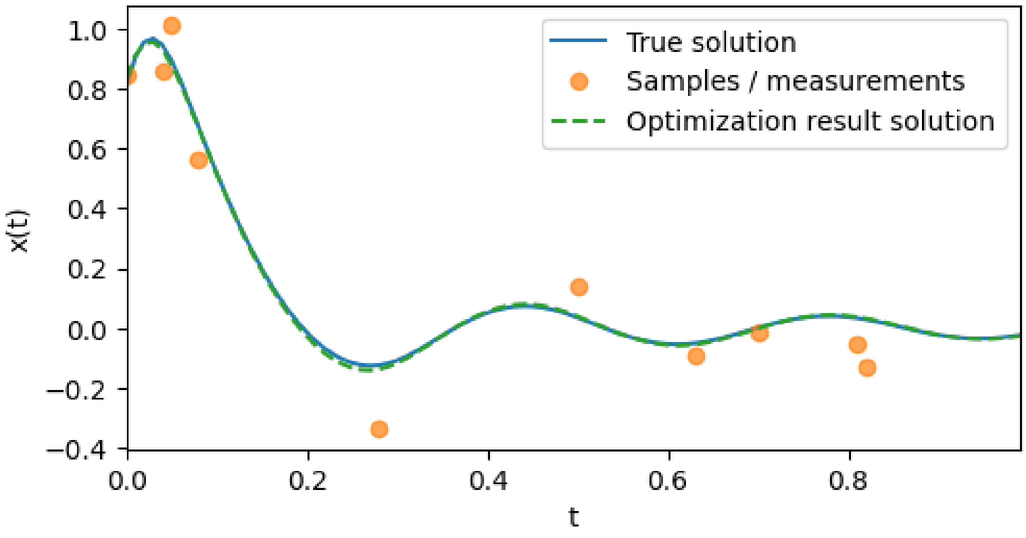

Figure 10: Solution to the ODE $\ddot{x}(t) = -(6\pi)^2 x(t) - 4\dot{x}(t)\,|\dot{x}(t)|$ with $x(0) = 0.8$ and $\dot{x}(0) = 15$ (blue curve), and a small number of samples (orange dots) plus Gaussian noise with $\sigma = 0.1$. The output of the collapser method is shown as the dashed green curve. The collapser has no access to the initial conditions that were used to generate the blue curve.

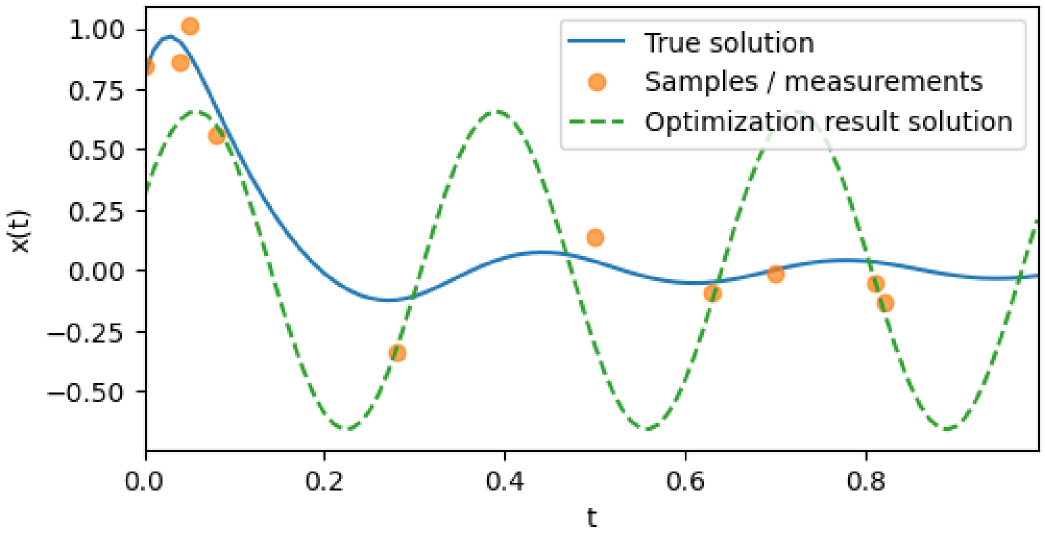

Figure 11: The data generation here is the same as in figure 10. However, the collapser has been passed the ODE $\ddot{x}(t) = -(6\pi)^2 x(t)$. This differs from the ODE used to generate the data, as the quadratic drag term is absent. The output of the collapser method is shown as the dashed green curve, and clearly appears to be a simple harmonic, at the expense of fitting the data.

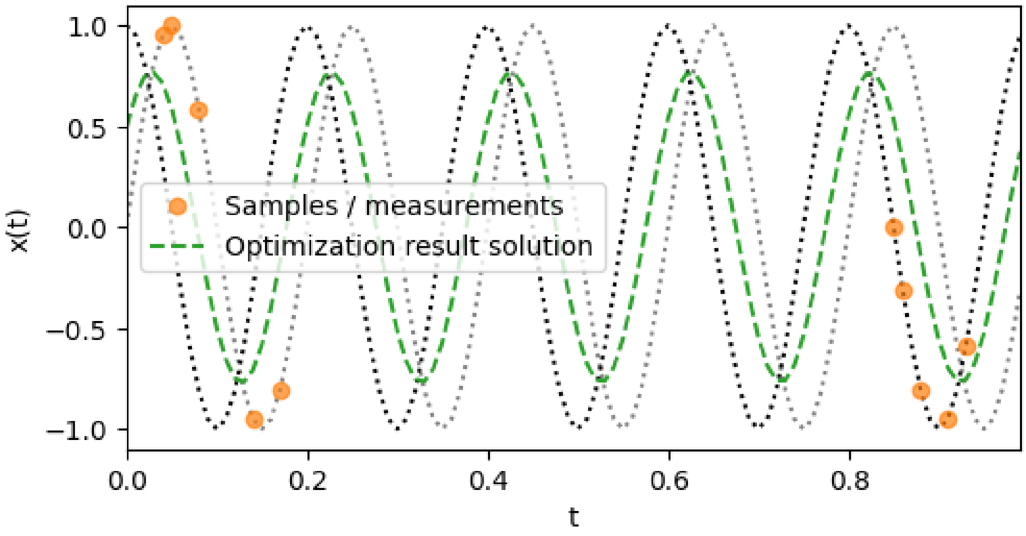

Figure 12: We generate two sets of timeseries from oscillators of the same frequency, but 90° out of phase. We treat these as a single series, with the observations from one sampled at early times and the other at late times. For this combined series, the ODE $\ddot{x}(t) = -\omega^2 x(t)$ should not fit the data. We see that the collapser does indeed correctly enforce the ODE, and does not fit the data well.

3.2 Unstable fixed points

In figure 4 we showed 3 (numerically integrated) solutions for the ODE

$$\ddot{x}(t) = -x(t)^3 + (6\pi/\sqrt{2})^2 x(t) - 3\dot{x}(t)$$

and claimed that it would be very hard for sweep-and-shoot method to discover appropriate initial conditions, and that such a method would falsely reject this ODE as a inconsistent with the data. Can the collapser method find these solutions?

It can, but modifications needed to be made. Using settings that produced the results of section 3, the collapser produced three curves that appeared to obey the ODE, but were very poor fits to that data. Many of them 'fell' into the opposite well compared to their sampled data. Inspecting the evolution of L_{data} and L_{ODE} over the course of the optimization showed that L_{data} did not reach zero, even during the early stages of optimization where $W_{\text{ODE}} = 10^{-2}$. We should in fact be able to fit the data exactly if we ignore the requirement to satisfy the ODE. So I first modified the code to initialize the discretized solution to a linear interpolation between the data points.

Initializing the discretized solution in this manner produced a good result on one of the three timeseries, but the other two still fit the data extremely poorly by falling into the wrong well. Again inspecting the evolution of L_{data} and L_{ODE} showed that L_{data} spiked upwards to an extreme value at the start of the optimization, before decreasing somewhat. This occurred even during the early 'warm-up' period when $W_{\text{ODE}} = 10^{-2}$. I was able to obtain the fits shown in figure 13 by decreasing W_{ODE} to 10^{-4}.

The results of figure 13 do show good fits. The purple solution has been found almost exactly, and the blue has found a solution that matches the data very closely, even though it does not match the underlying solution. The red solution is clearly a little lacking, but still captures something of the correct behaviour. (The final L_{data} for the purple, blue and red curves are 0.011, 0.036 and 5.1 respectively.) In each case the behaviour is qualitatively correct. The results of this section can be produced by the notebook at `/demo_notebooks/quartic_double_well.ipynb`.

However, I believe this should be regarded as only a partial success for the collapser. Manual intervention was required to obtain these solutions. Remember that the motivation for this method is to allow unsupervised evaluation of the goodness-of-fit of an ODE to a dataset, as a component of PySR's search. Nor is the change I made here completely superior: I tried re-running the experiment of figure 10 with these changes. Initializing the discretization by interpolation still resulted in an excellent fit, but changing W_{ODE} to 10^{-4} for the warm-up period caused the optimization to fail, producing NaNs.

More development is needed to make the optimizer widely usable, and I discuss this in section 5.2.

4 Quantitative Results

Recall that the motivation for this method is to take a timeseries, and distinguish between ODEs that are a good and a poor fit to that timeseries. We need the collapser to provide a good fit with low L_{data} when provided with an ODE that fits the data well, but we also need it to produce a high L_{data} when the ODE is unable to fit the data well. We showed an example of this in figures 10 and 11.

To get a more comprehensive picture of how well the collapser can distinguish between candidate ODEs, I ran a series of experiments. I selected 6 systems (ODEs) and for each system I selected 2 or 3 sets of initial conditions. See table 1 for a description of each system. I constructed the solution of each system and initial condition pair (either by knowledge of the analytical solution or using PySR's numerical integrator), and sampled a timeseries of 10 points from each, adding Gaussian noise with $\sigma = 0.1$.[7] This gives me 14 timeseries, and for each of these I know the ODE that generated it.[8] For each timeseries I pass it, along with each possible candidate ODE, to the collapser. What I expect to see is this: the collapser always outputting small L_{ODE} (indicating that it has found a solution to the candidate ODE), and outputting the small L_{data} when the candidate ODE matches the ODE that was used to generate the timeseries.

[7] The times from which the samples are taken are chosen independently for each solution.

[8] I also know the initial conditions, but we won't use those in any way.

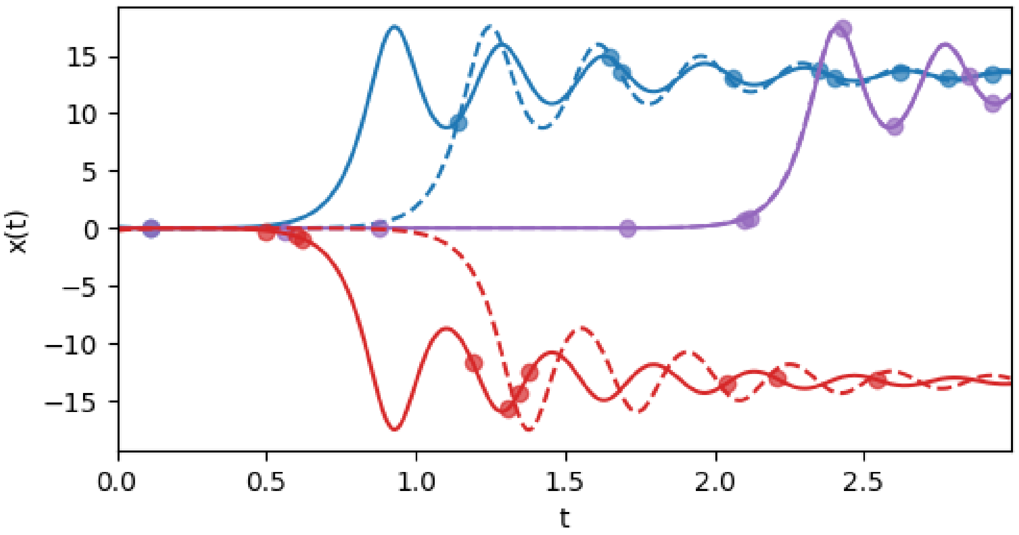

Figure 13: The solid curves show three (numerically integrated) solutions to $\ddot{x}(t) = -x(t)^3 + a^2 x(t) - \nu \dot{x}(t)$ for $a = 6\pi/\sqrt{2}$ and $\nu = 3$. The circles of the same colour show datapoints sampled (with Gaussian noise with $\sigma = 0.1$) from each curve. Each of the three timeseries, along with the ODE, are passed to the collapser. The solutions found by the collapser are shown as dashed curves.

System	Description	ODE	Initial conditions
CA1	Constant (large) acceleration	$\ddot{x}(t) = -10$	$x(0) = 0,\ \dot{x}(0) = 2$
			$x(0) = 2,\ \dot{x}(0) = 0$
CA2	Constant (small) acceleration	$\ddot{x}(t) = -5$	$x(0) = 0,\ \dot{x}(0) = 2$
			$x(0) = 2,\ \dot{x}(0) = 0$
QD1	Quadratic-drag oscillator 1	$\ddot{x}(t) = -(6\pi)^2 x(t) - \dot{x}(t)\,\lvert\dot{x}(t)\rvert$	$x(0) = 0.85,\ \dot{x}(0) = 16$
			$x(0) = 0.85,\ \dot{x}(0) = -16$
QD2	Quadratic-drag oscillator 2	$\ddot{x}(t) = -(4\pi)^2 x(t) - \frac{1}{2}\dot{x}(t)\,\lvert\dot{x}(t)\rvert$	$x(0) = 0.85,\ \dot{x}(0) = 16$
			$x(0) = 0.85,\ \dot{x}(0) = -16$
SHO	Simple harmonic oscillator	$\ddot{x}(t) = -(6\pi)^2 x(t)$	$A_0 = 1.2,\ \phi_0 = \pi/4$
			$A_0 = 0.8,\ \phi_0 = 0$
			$A_0 = 1.2,\ \phi_0 = -\pi/4$
DHO	Simple harmonic oscillator	$\ddot{x}(t) = -(6\pi)^2 x(t) - 4\dot{x}(t)$	$A_0 = 1.2,\ \phi_0 = \pi/4$
			$A_0 = 0.8,\ \phi_0 = 0$
			$A_0 = 1.2,\ \phi_0 = -\pi/4$

Table 1: The systems used in the experiments of section 4.

The results of this experiment are plotted in figure 14. Green symbols denote runs of the collapser where the candidate ODE matched the ODE used to generate the timeseries, and we want these to have a lower L_{data} than the red points (denoting the 'wrong' ODE being passed to the collapser. This is the case for 13 of the 14 timeseries, the one exception being one of the timeseries generated from QD1, our first quadratic-drag oscillator. QD1 has a higher ν than QD2, and it is possible that the reason a simple-harmonic oscillator can be fitted well to it is due to the motion decaying quickly, allowing the SHO to simply be fit with a small amplitude. The other ODEs that the collapser is unable to separate from QD1 in this case is DHO, the damped harmonic oscillator with linear damping. This is at least a system with somewhat similar physics.

On the basis of figure 14, I believe the collapser is a promising method for assessing the goodness-of-fit

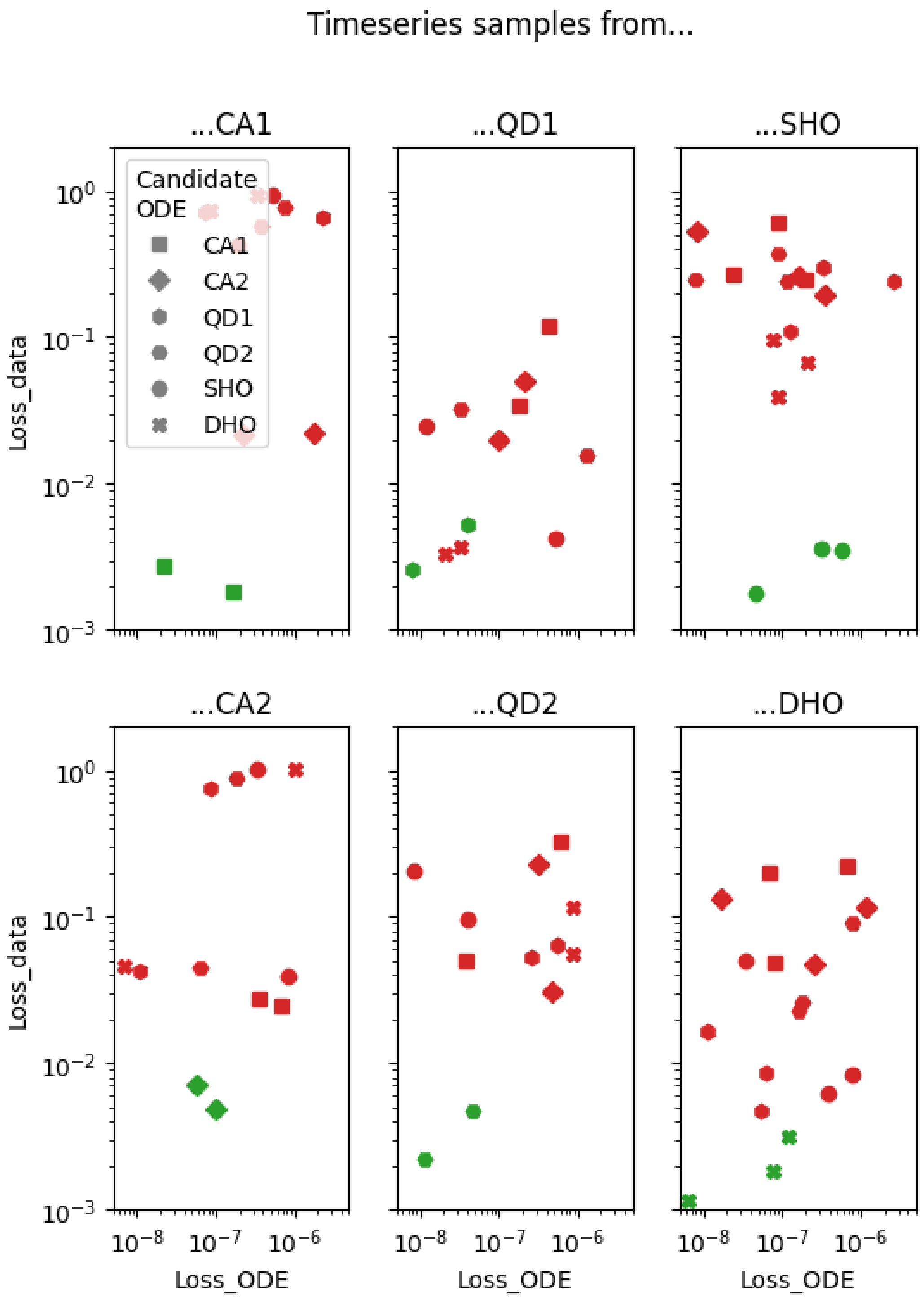

Figure 14: Each panel shows the results of running the collapser with a timeseries generated from the ODE in the title. For example, the top right panel shows results where the timeseries are generated from the second quadratic drag ODE ('QD2'). Within each panel, every point corresponds to running the collapser, with the shape of the point denoting which candidate ODE was passed to it. Note that each symbol type appears either 2 or 3 times, depending on how many timeseries (corresponding to different initial conditions) we generated from the ODE in the title of the panel. Symbols are coloured green if the candidate ODE matches the ODE used to generate the timeseries, and red if it does not.

of a candidate ODE to noisy data in the absence of initial conditions.

The results in this section were produce using the notebook `./demo_notebooks/quant_experiments.ipynb`.

5 Limitations

5.1 Speed

The optimization process is relatively slow, taking between 10 and 30 seconds on my CPU for most of the examples in section 3. Generating the results of section 4 required 84 runs of the collapser (14 timeseries multiplied by 6 candidate ODEs) and took 15 minutes 34 seconds, for an average of 11 seconds per optimization. I doubt using a GPU would give a significant speed-up, given that the configuration space is not very large, and hence the problem is not very parallel.

The slow speed is partly due to using a large number of iterations to ensure convergence to the solution space. This could be avoided by dynamically testing if convergence has occurred, for example by testing for relative changes in L_{ODE}, and terminating once improvements cease. It is possible that I am also not using the best possible optimizer parameters - this is because I don't really understand the details of the LBFGS optimizer, and so I kept the default options.

5.2 Numerical stability

The optimizer parameters I have used here work well for some relatively benign functions (harmonic oscillators, constant acceleration), but had to be manually tuned to solve the case of a trajectory passing very close to an unstable fixed point (see section 3.2). It is also unclear how well this would perform with datasets taking a much larger range of values (e.g. $\tilde{x}(t) \sim 10^6$) or on much larger domains (e.g. $t_{\text{end}} - t_{\text{start}} \sim 10^6$).

As the number of grid points, N, increases (either due to the grid becoming finer or the domain becoming larger for the same resolution) the the condition number of the loss due to the second derivative increases (see figure 6 in section 2.1). It is unclear if there is a better way of addressing this than the transformation described in section 2.1.

I could probably fix up pathological cases in the optimization as I encounter them, using e.g. gradient-clipping, but it would be better to have a good theoretical understanding of what makes the optimization stable or unstable.

6 Extensions

I believe this method could be extended to irregular grids with relative ease. Whether it could also be extended to dynamic grids (c.f. ODE solvers that reduce the step-size in regions of high second-derivative) I am less sure about, but would be a likely be a useful capability for some systems.

Many of the papers I found in section 1.1 consider first order ODEs, potentially in multiple dimensions. This would be a useful extension, and would in fact cover the case of solving second-order ODEs (by treating $\dot{x}$ as a coordinate). I do not think that would be equivalent to this method, since we would now be optimizing the $2N$ dimensional vector

$$\left(\hat{x}(t_0), \hat{\dot{x}}(t_0), \hat{x}(t_1), \hat{\dot{x}}(t_1), \ldots, \hat{x}(t_{N-1}), \hat{\dot{x}}(t_{N-1})\right)$$

whereas the method discussed here involves optimizing an N dimensional vector. It would also require extending this method to allow partial observations, e.g. we observe $x(t)$, but not $\dot{x}(t)$.

Another possibility is that we give the collapser an ODE with unknown parameters. For example, we specify

$$f(\dot{x}(t), x(t), t) = -\omega^2 x(t) - \nu \dot{x}(t)$$

without values for ω and ν. In this example the solution space would have two additional dimensions. We would then minimize the loss with respect to the vector

$$(\omega, \nu, \hat{x}_0, \hat{x}_1, \ldots, \hat{x}_{N-1})$$

I have not implemented this, so I don't know whether or not it will work. One issue might be finding suitable initial values for ω and ν.

References

[1] Pysr issue #568: Physics-informed pysr. https://github.com/MilesCranmer/PySR/discussions/568. Accessed: 2025-07-07.

[2] Pysr issue #732: [feature]: Support for full time-derivative prediction (the right hand side of system of odes) with symbolic regression. https://github.com/MilesCranmer/PySR/issues/732. Accessed: 2025-07-07.

[3] Pysr v1.3.0 change notes. https://github.com/MilesCranmer/PySR/releases/tag/v1.3.0. Accessed: 2025-07-07.

[4] Wikipedia article: Relaxation (iterative method). https://en.wikipedia.org/wiki/Relaxation_(iterative_method). Accessed: 2025-07-07.

[5] Josh Bongard and Hod Lipson. Automated reverse engineering of nonlinear dynamical systems. *Proceedings of the National Academy of Sciences*, 104(24):9943–9948, 2007.

[6] Stéphane d'Ascoli, Sören Becker, Alexander Mathis, Philippe Schwaller, and Niki Kilbertus. Odeformer: Symbolic regression of dynamical systems with transformers. *arXiv preprint arXiv:2310.05573*, 2023.

[7] Hitoshi Iba. Inference of differential equation models by genetic programming. *Information Sciences*, 178(23):4453–4468, 2008.

[8] Gabriel Kronberger, Lukas Kammerer, and Michael Kommenda. Identification of dynamical systems using symbolic regression. In *International Conference on Computer Aided Systems Theory*, pages 370–377. Springer, 2019.