May 14th Weekly Report

Erik Bigwood

May 14, 2025

Abstract

In total, this week I have extended my work to analyze the physics of two same-charged particles in a harmonic potential. Adopting the exact form of the Lagrangian used to generate the synthetic dataset provided a robust fit to the data, converging well under a suitable loss function (L^2 norm). This method extended partially to cases with the sampling window – large displacements blocked convergence for α but converged well for k, and a narrow window around one of the two fixed points converged exceedingly well.

Adding complexity to the Lagrangian by considering a polynomial in q_0 and q_1 had somewhat more mixed results. Fitting around the origin and in a wide window around a fixed point largely succeeded, but fitting a narrow window around a fixed point and then extending to validate this fit over a wide region failed just about everywhere except in the narrow training window.

To facilitate these more computationally demanding experiments, I have made a couple of improvements to the core algorithm by further leveraging torch.func.vmap.

Contents

1 Two particles in a harmonic potential with Coulomb repulsion, exact Lagrangian

1.1 Basic results

Code setup Here I've created and system with two particles in a 1D harmonic potential (strength k) [4] with a Coulomb potential [2] between them ($\alpha = charge_0 \cdot charge_1$), both masses set to 1. As the dimension of the phase space here is twice as large as the 1D system, and in anticipation of future expansions, I've rewritten the phase-space sampling as a large number of random points instead of a grid.

```
q0_min, q0_max, q1_min, q1_max, = -1, 1, -1, 1
Q0 = (q0_max - q0_min)*(torch.rand(n_points)) + q0_min
Q1 =  (q1_max - q1_min)*(torch.rand(n_points)) + q1_min

qdot_min, qdot_max = -1, 1
qdot_sampling = (qdot_max - qdot_min)*(torch.rand([2,n_points])) + qdot_min
QD0 = qdot_sampling[0,:]
QD1 = qdot_sampling[1,:]
```

Package these into array objects to be passed to later functions:

```
Qv = torch.zeros(n_points,2)
Qv[:,0] = Q0
Qv[:,1] = Q1

QDv = torch.zeros(n_points,2)
QDv[:,0] = QD0
QDv[:,1] = QD1
```

Then calculate the predicted $\ddot{q}$ values in a single call:

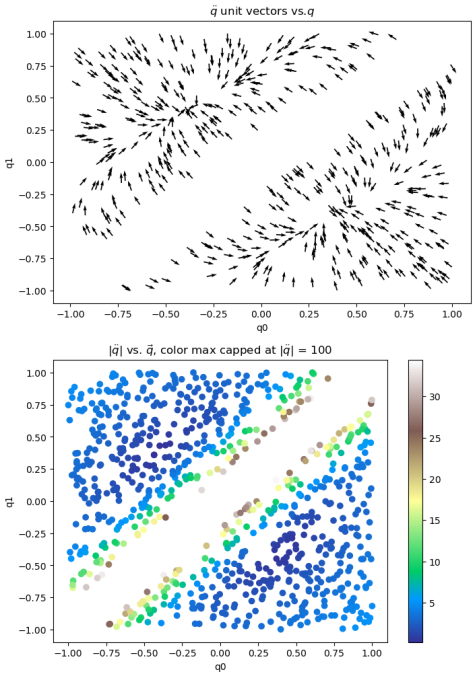
```
out = QDDv(Qv,QDv,m,k,alpha)
outmag = (out[:,0]**2 + out[:,1]**2)**.5
```

1.1.1 Results

In Fig. 1, I've included a quiver plot of the unit vectors $\ddot{q}/|\ddot{q}|$ and a scatter plot colored by $|\ddot{q}|$. Since the band where $|q_0 - q_1|$ is small produces very large generalized forces, I've removed any points in it to reduce numerical instability later in the fitting algorithm. These points produce a much larger contribution to the loss, they contribute to extremely large gradients along the α axis of parameter space that largely wash out the contribution from k and delay convergence under gradient descent.

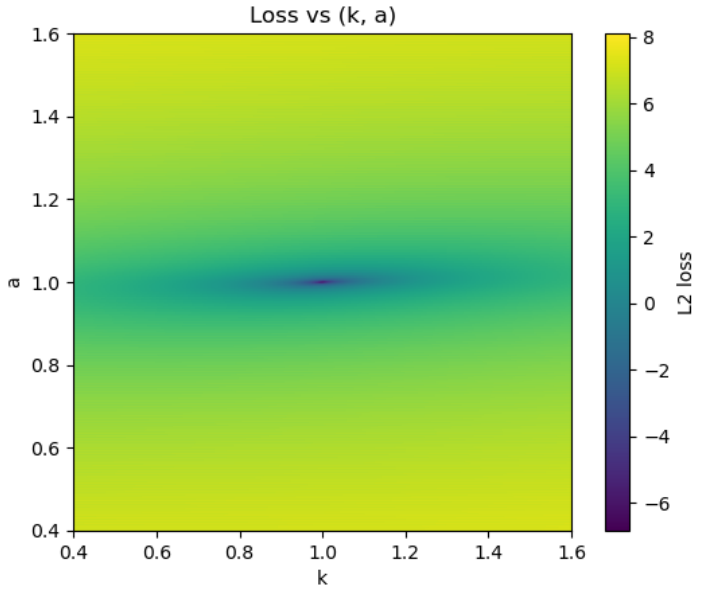
In the quiver plot, we see the exluded band of extreme $\ddot{q}$ as well as a pair of points where $\ddot{q}$ goes to zero. These are the points in (q_0, q_1) configuration space such that the harmonic and Coulomb forces balance and the particles are not accelerated. Away from this region, $\ddot{q}$ becomes dominated by the the harmonic potential.

Figure 1: Quiver and scatter plots of sampled points and associated $\ddot{q}$



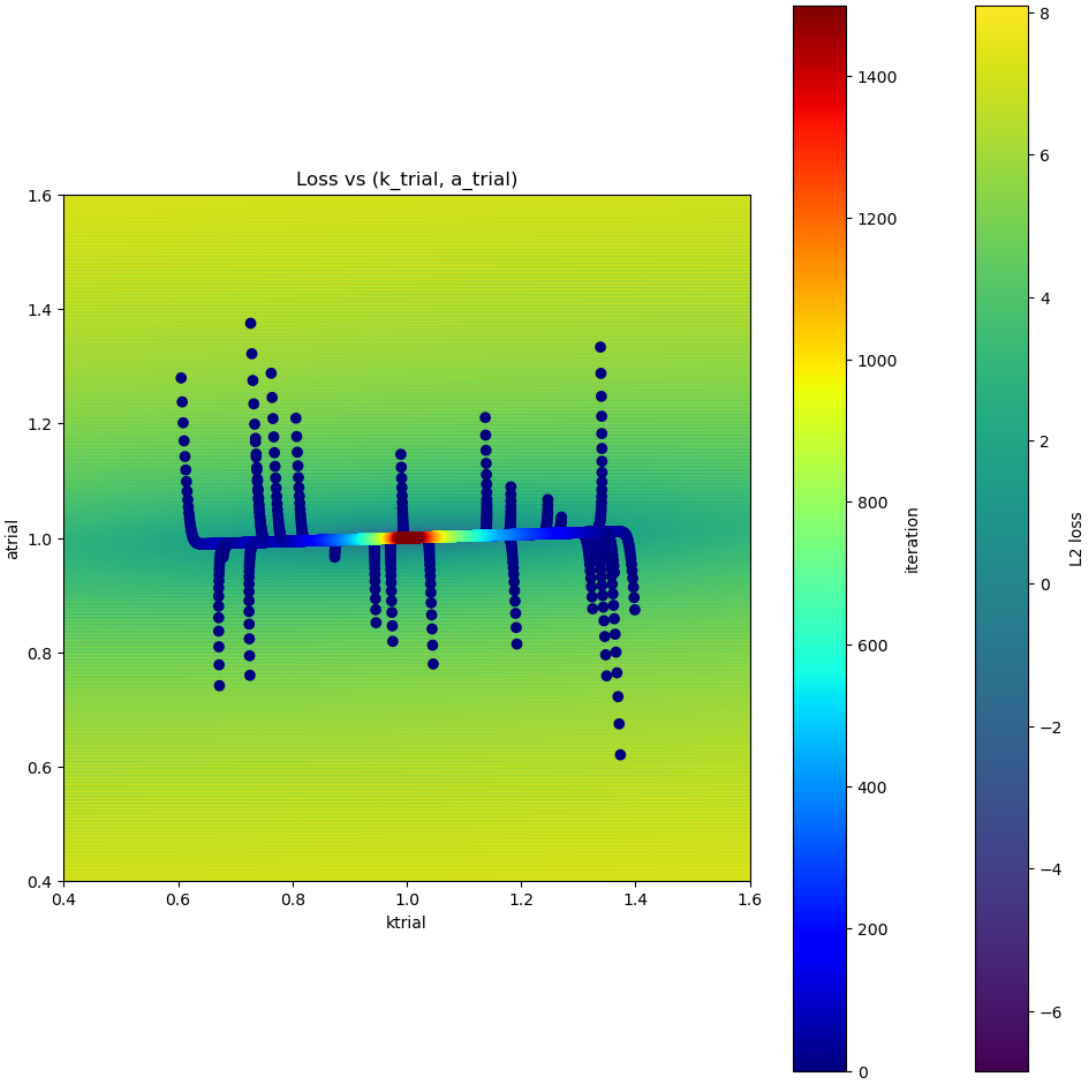
Now calculating the loss landscape around the true (k, α) parameters, we see a deep well of low loss in Fig. 2, with the caveat of mismatched gradients between k and α directions.

Figure 2: Loss landscape vs. (k, α)



Performing gradient descent on this landscape leads to convergence on the small valley of low loss as expected, but this convergence is slow compared to the 1D case due to the mismatch of gradient magnitude between the k and α directions. See Fig. 3. In general, gradient descent in this type of situation will tend to optimize the variable with the larger gradient first (and quickly), and then slowly optimize the other. In future updates I will try using more modern optimizers like 'Adam'[3] or 'AdaGrad'[1] that purport to perform better in these situations.

Figure 3: Gradient descent on loss landscape vs. (k, α)



In Fig. 18, I've plotted both the loss (18a) and parameter fit (18b). As expected, loss monotonically decreases and does not appear to saturate (running further iterations

would likely decrease loss even further). For the parameter fit, I've plotted the ratio of the experimental (l/α) and the true k/α, showing clean convergence to the true parameters.

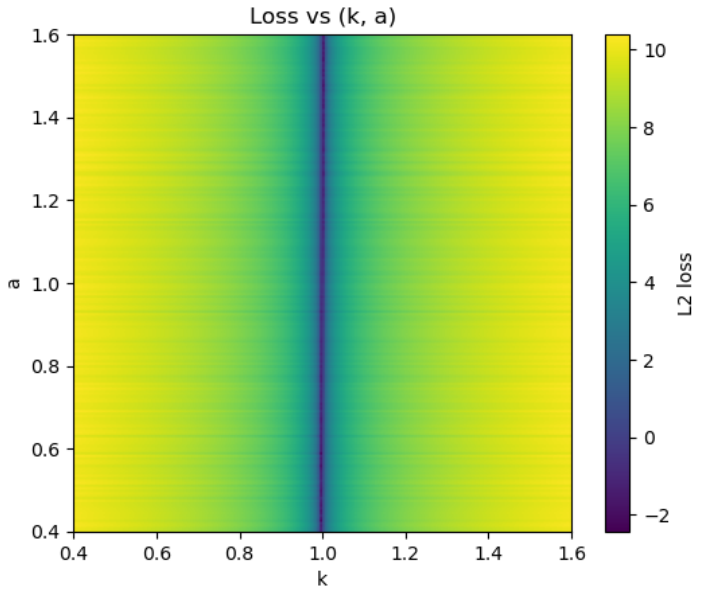
1.2 Changing sampling areas

1.2.1 Far from the origin

Not all experimental situations will have such clear coverage over regions of phase space with strong influence from interactions. Displacing the sampled region of phase space away from makes the sampled $\ddot{q}$ dominated by the harmonic potential and reduces the gradient in the α direction.

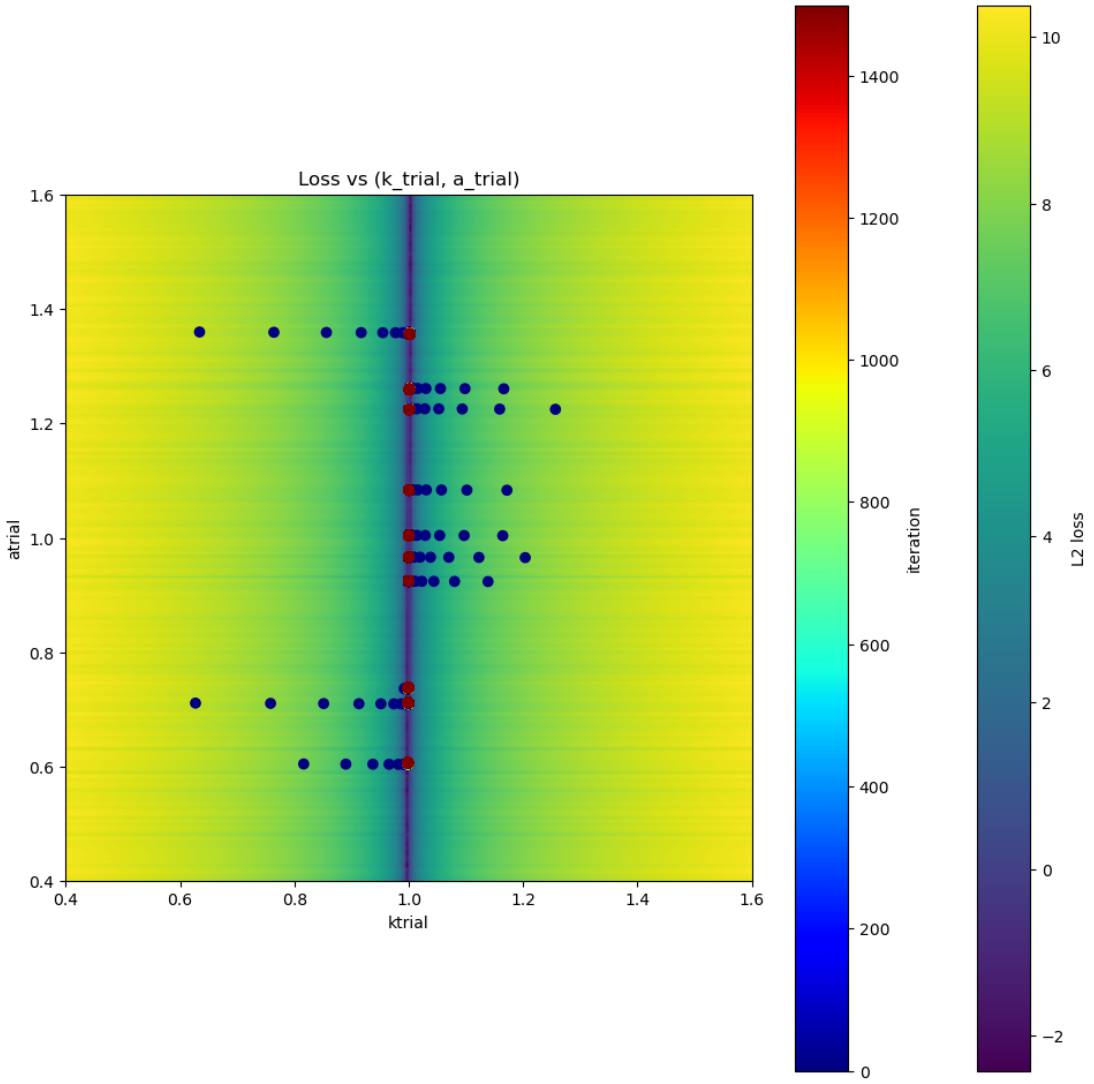
This displacement increases the magnitude of the k gradient and dramatically reduces the magnitude of the α gradient, as expected by the scaling of the harmonic and Coulomb potentials under displacement. Fig. 4 contains a plot of the loss landscape, which notably has some unexpected roughness that persists over large ranges of the number of sampled phase space points.

Figure 4: Loss landscape vs. (k, α), sampling displaced from the origin



Performing gradient descent on this displaced dataset is far less successful – only k converges to the correct value in Fig. 5. α only converges a very small amount due to the mismatch between gradient magnitudes, addressing this would require tuning the learning rate to compensate for this difference and reducing the landscape roughness. I tried tuning the algorithm to dramatically increase the learning rate as the loss value decreases, but this only resulted the system getting trapped in local minima caused by the roughness. Fig. 20 contains plots of the loss (20a) and k/α (20b) performance trends, further indicating the optimizer getting trapped in local minima along the $k = 1$ line.

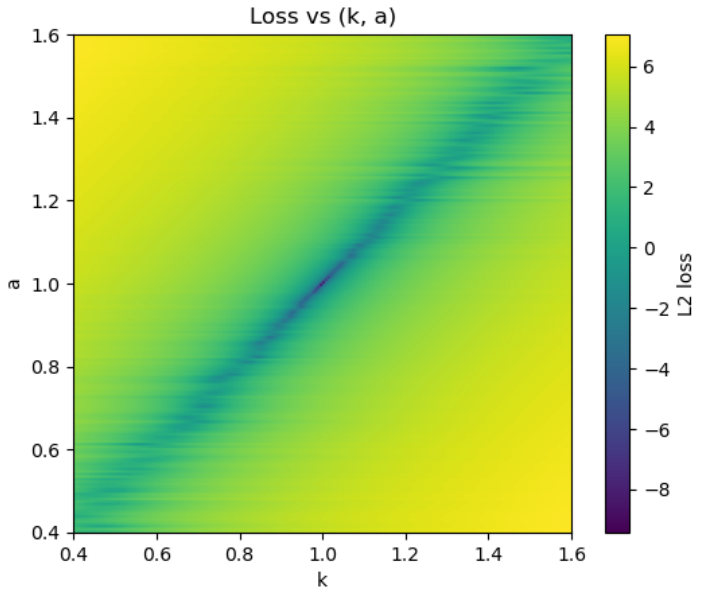
Figure 5: Gradient descent on loss landscape vs. (k, α), displaced from the origin



1.2.2 Near a fixed point

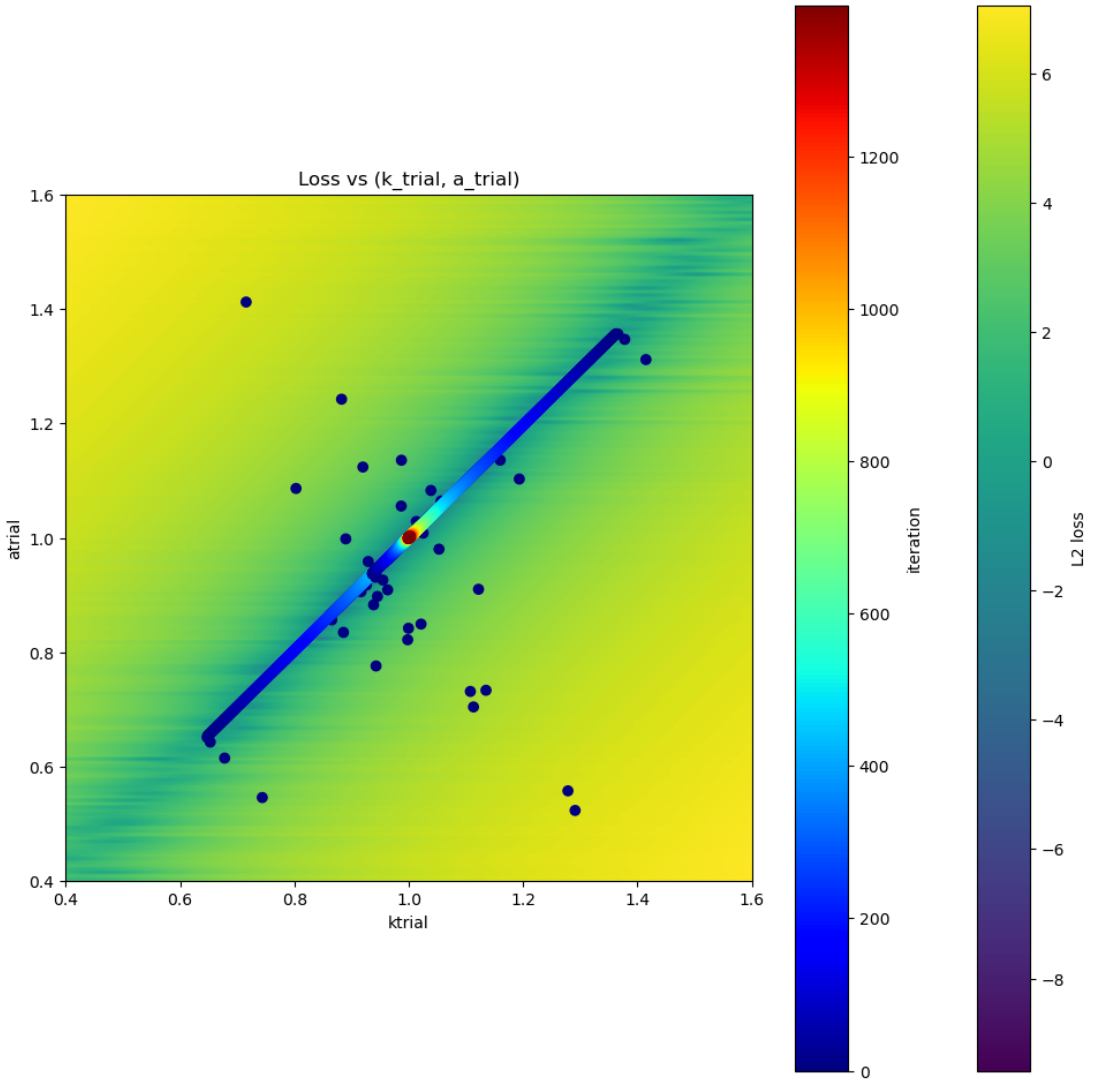
Near one of the fixed points where the harmonic and Coulomb forces balance, the story changes a bit. Fig. 21 shows the sampled points and associated $\ddot{q}$. The loss landscape (Fig. 6) shows a strong loss well, but some additional roughness that appears to be worse with fewer sampled points.

Figure 6: Loss landscape vs. (k, α), sampling near a fixed point



This roughness does not preclude proper convergence, as gradient descent performs reasonably well. In this setup we still see that the magnitude of the gradients are mismatched, but this time the system converges to the $k = \alpha$ line and then slowly optimizes towards the true parameters. The performance parameters are plotted in Fig. 22, with both loss (22a) and k/α (22b) indicating extremely robust convergence to the true solution. As seen before, this optimization proceeds in two phases – first, the system falls onto the $k = \alpha$ line, and then falls towards the low loss point at the true parameter values.

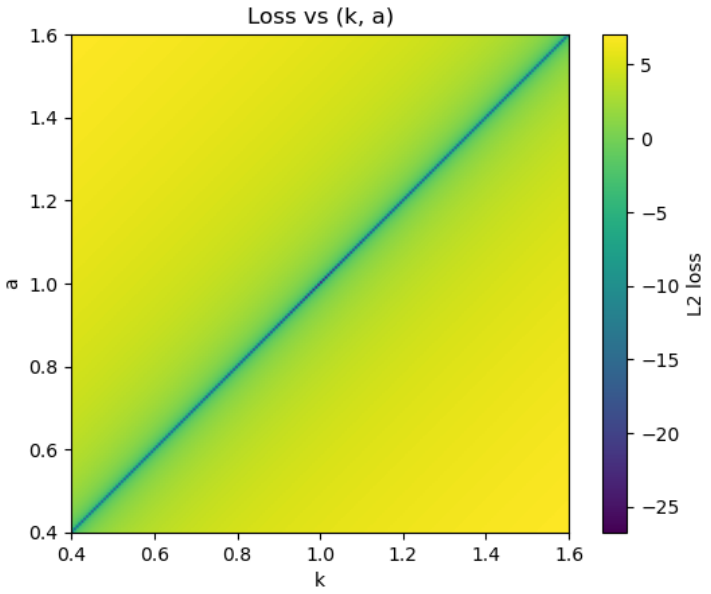
Figure 7: Gradient descent on loss landscape vs. (k, α), near a fixed point



Interestingly, zooming the sampling far into this fixed point region reduces the loss landscape roughness (Fig. 8) by a large degree, and deepens the low-loss valley to the lowest loss value ($\approx 10^{-25}$) seen in any run so far.

Comparing the quiver plots in Fig. 24 for the zoomed-out (sampling region 0.1 units wide, 24a) and zoomed-in (sampling region 0.0002 units wide, 24b) sampling around the fixed points seems to show very similar behavior across these different scales. In further updates I may explore this further, with special attention on any possible mapping to other physical systems, and possible scaling symmetries.

Figure 8: Loss landscape vs. (k, α), sampling zoomed in on a fixed point



2 Two particles in a harmonic potential with Coulomb repulsion, polynomial Lagrangian

2.1 Problem setup

As before, I generate a synthetic dataset from the exact Lagrangian with known parameters, then disregard those known parameters for fitting. Here I move to a more complicated Lagrangian that is a polynomial in powers of products of the coordinates q and quadratic in $\dot{q}$ as follows:

$$\mathcal{L} = \frac{1}{2}\dot{q}^2 + \sum_{m=0}^{n_{particles}} \sum_{n=0}^{n_{particles}} \left[\sum_{i=1}^{N} \sum_{j=1}^{N} c_{m,n,i,j} q_m^i q_n^j \right] \tag{1}$$

Where q_m is the m-th component of q (in this 1D case, the value of q for the m-th particle) and $c_{m,n,i,j}$ is the parameter tensor/weight for the (m,n,i,j)-th term in this polynomial. For implementation in this 1D, 2 particle system, the code actually discards the initial double sum over (m,n) in favor of summing over the three configurations $(m,n) = (0,0)$, $(0,1)$, and $(1,1)$ as the $(0,1)$ case is equivalent to $(1,0)$. This then reduces to the following where $\mathcal{M}$ is the set containing those three cases:

$$\mathcal{L} = \frac{1}{2}\dot{q}^2 + \sum_{(m,n)\in\mathcal{M}} \left[\sum_{i=1}^{N} \sum_{j=1}^{N} c_{m,n,i,j} q_m^i q_n^j \right] \tag{2}$$

Unfortunately, this becomes rather large and compute-intensive quickly as the number of terms N is increased. So far I have only explored out to $N = 8$ on an nVidia T4 GPU

provided by Google Colab. Training this Lagrangian presents additional challenges relative to training a Lagrangian with the same structure as the true one, especially with interactions like this Coulomb potential. The Coulomb potential term $\alpha(\frac{1}{|q_0-q_1|})$ expands as a binomial series[5]. Taking the $q_0 = 1$ case, this becomes (within a suitable radius of convergence $|q_1| < 1$):

$$\left|\frac{1}{q_0 - q_1}\right| \to \left|(1 - q_1)^{-1}\right| = \left|\sum_{k=0}^{\infty} \frac{(1)_k}{k!} q_1^k\right| = \left|1 + q_1 + q_1^2 + q_1^3 + ...\right| \tag{3}$$

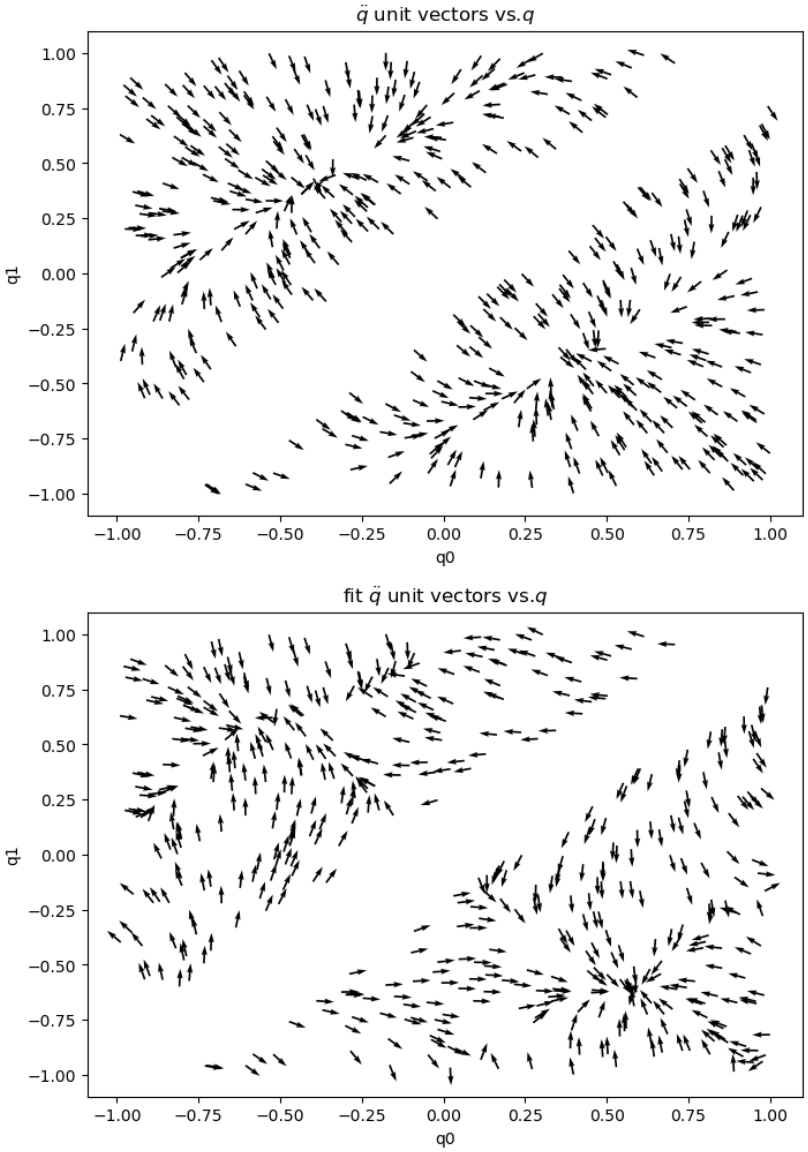
This expression contains terms of arbitrarily high order, so no finite polynomial will perfectly fit this system. Instead, an approximation for the true Lagrangian is optimized by the algorithm, minimizing the prediction loss at each sample point.

2.2 Results

First runs Take as a first example a training run on this polynomial Lagrangian from a dataset with $m = 1$, $k = 4$, and $\alpha = 1$. The loss history/profile is shown in Fig. 25, and converges acceptable, although dramatically slower than the exact form Lagrangian due to the vastly increased number of parameters.

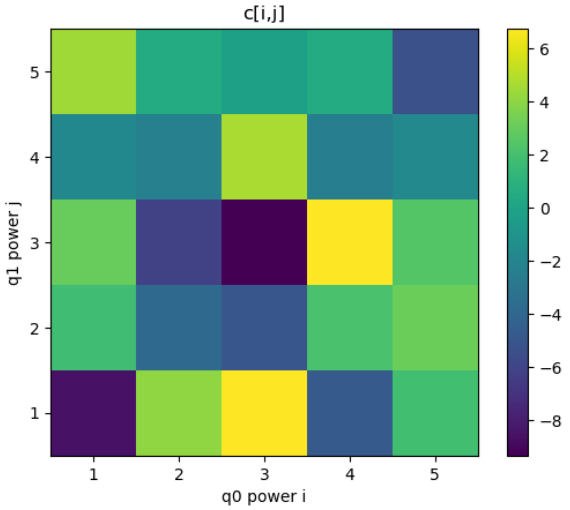
This training resulted in (Fig. 9) a learned Lagrangian that fits the dataset reasonably well – it has fixed representing the equilibra from the harmonic and Coulomb forces balancing, and matches the rough overall trends. There are, however, distinct errors in this run, such as fit arrows pointing outwards at the borders of the sampled region instead of in as in the dataset.

Figure 9: Sampled points and trained polynomial Lagrangian results, run 1



The learned parameters can be visualized via a density plot, where each region corresponds to a specific term in the polynomial. In Fig. 10, I've plotted the polynomial coefficients for only the coupling terms $q_0^i \cdot q_1^j$. We see approximate symmetry across the $i = j$ diagonal, and in general runs tend to converge to a symmetric matrix, as we would expect given the coupling term's symmetry in the arguments.

Figure 10: Polynomial coupling term coefficients, run 1



Alternate form The previous parameter object $c_{m,n,i,j}$ is a $(3\times n_{powers}\times n_{powers})$ element tensor containing a large number of duplicate parameters in the $(m = n)$ slices. To reduce this redundancy and simplify the fitting to just the coupling terms, I define a new (but related) Lagrangian retaining the original values for m and k :

$$\mathcal{L} = \frac{1}{2}\dot{q}^2 - \frac{1}{2}kq^2 - \left[\sum_{i=1}^{N}\sum_{j=1}^{N} c_{i,j}q_0^i q_1^j\right] \tag{4}$$

This bracketed term now contains all of the couplings between the particles and only contains $(n_{powers} \times n_{powers})$ elements to optimize, but neglects any higher powers of individual q_i values (to be addressed later). The convergence performance of this version is similar (Fig. 26), though slightly faster per iteration due to the lower number of parameters. This also produces a result with a good qualitative match to the data set (Fig. 11), but with the same caveats about errors at the boundaries. For this run, I've restricted the sampling to one side of the $q_0 = q_1$ line for visual clarity.

Figure 11: Sampled points and trained polynomial Lagrangian results, run 2

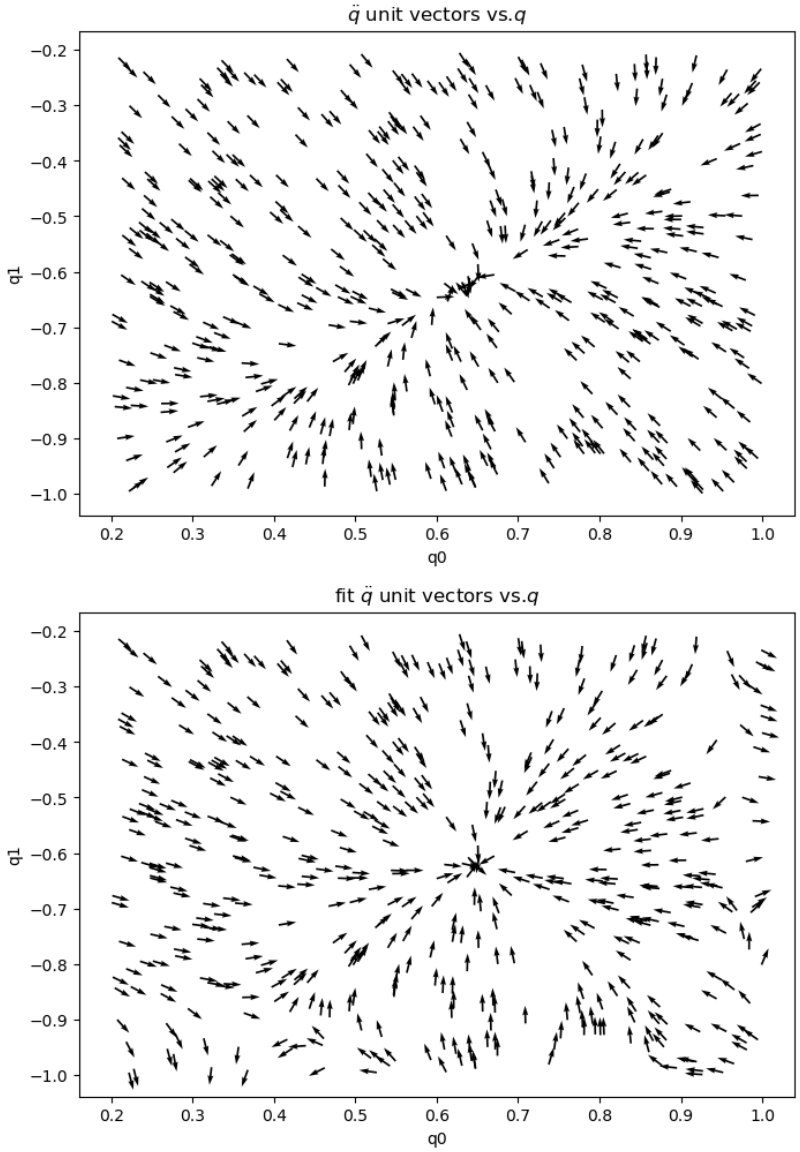
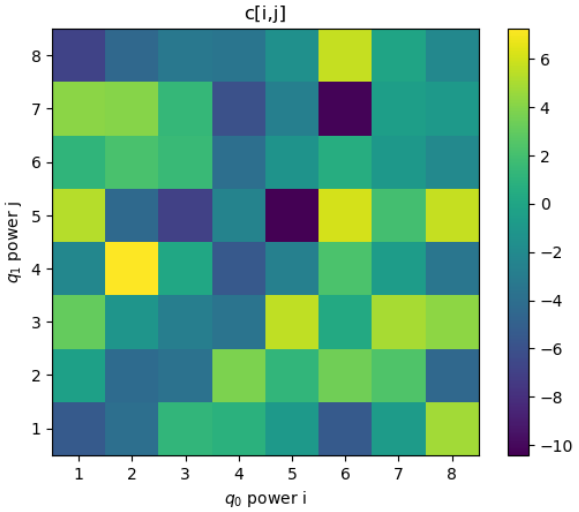


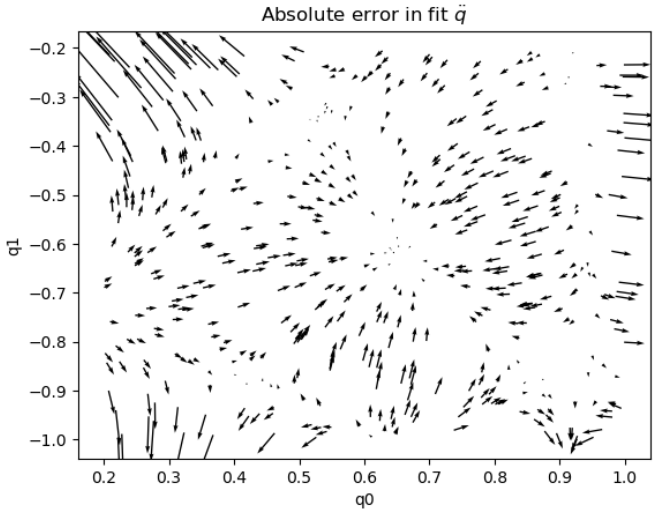
Figure 12: Polynomial coupling term coefficients, run 2



The parameters (Fig. 12) learned in this run are not as symmetric as run 1, and interestingly contain much more contribution from the q_0^8 terms.

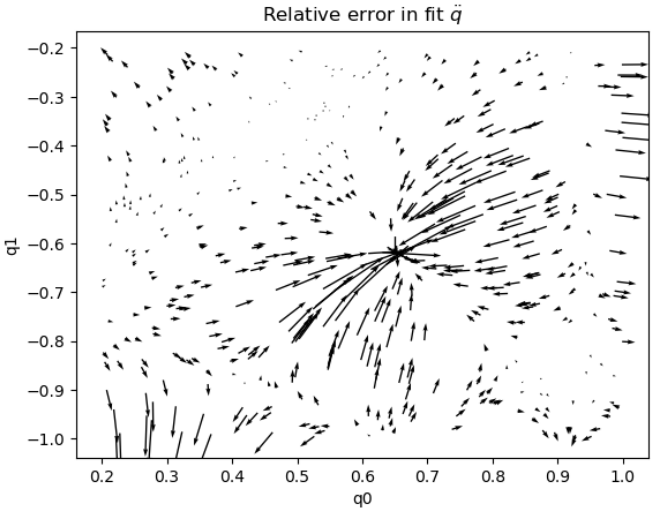
In Fig. 13 I've plotted the absolute errors between dataset $\ddot{q}$ and fit $\ddot{q}$. Near the fixed point (center of frame), the error becomes very small, but the error tends to increase near the boundaries of the sampling region. The error becomes especially large in the upper-left region where the separation between the charges is smallest – here the generalized force $\ddot{q}$ blows up and is not fully captured. This is to be expected as we are truncating the true coupling term, and its magnitude becomes very large in this region.

Figure 13: Sampled points vs. trained polynomial Lagrangian error, run 2



The relative errors (Fig. 14) shows different behavior – there is far less increase for the $q_0 \approx q_1$ region in the top left, but the area near (but not exactly at) the fixed point (center) has much more considerable errors.

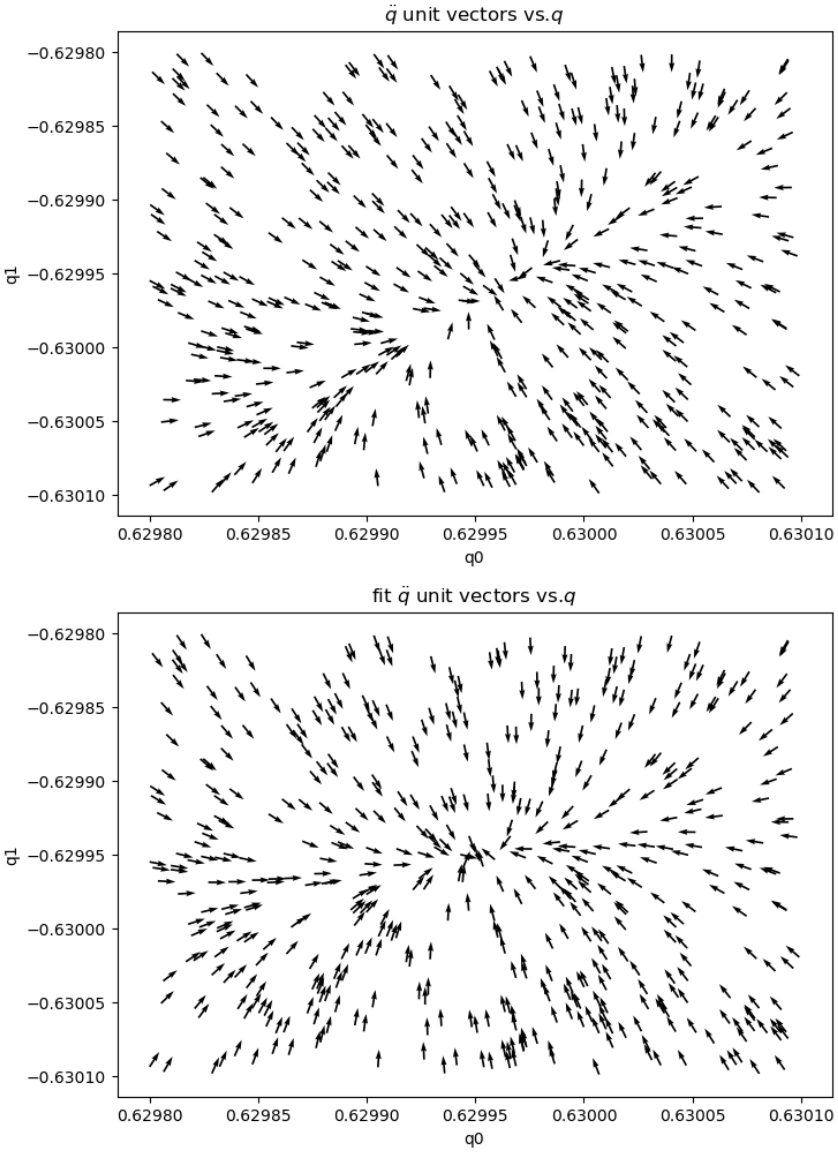
Figure 14: Sampled points and trained polynomial Lagrangian results, run 2



2.2.1 Changing the sampling vs. validation window

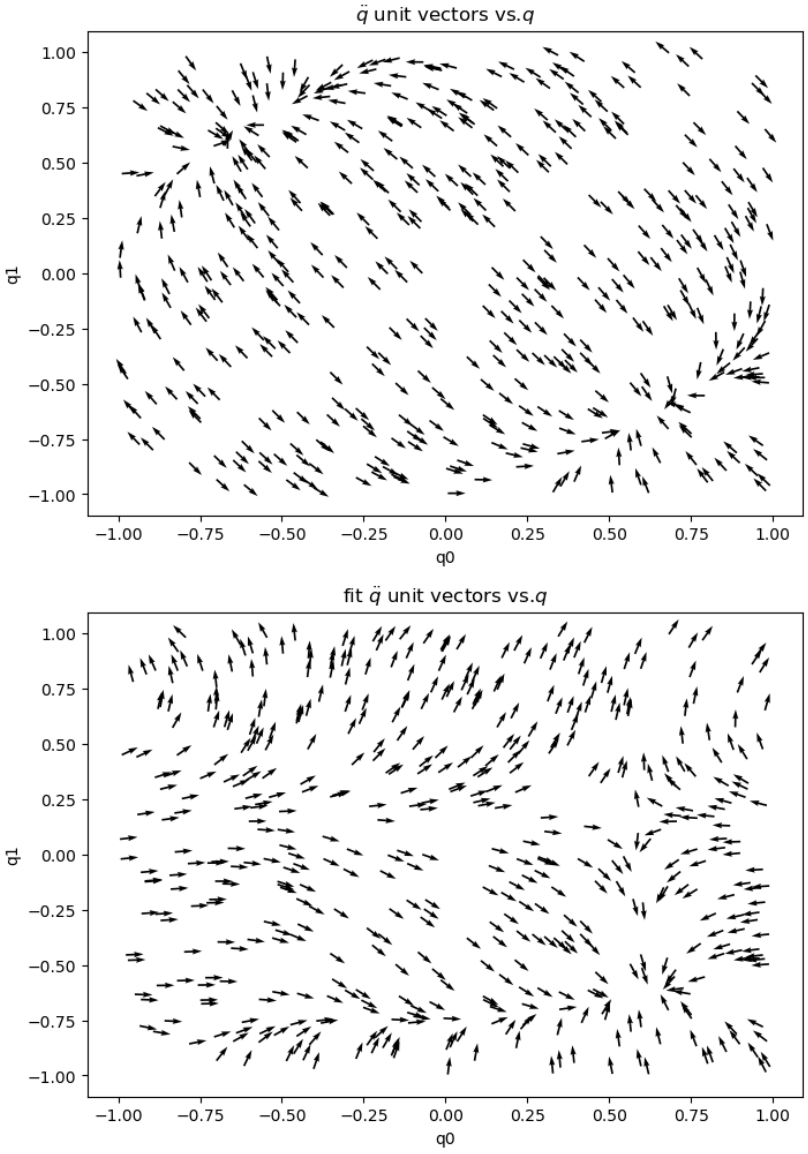
I now restrict the sampling points provided to the algorithm to an exceedingly narrow range (0.0003 units wide compared to previous windows of ≈ 1 unit) centered around one of the two fixed points. As mentioned before, at these scales the calculated $\ddot{q}$ unit verctor field appears independent of window size. To explore how well this system can extrapolate to unknown data, I fit the polynomial Lagrangian to this narrow window and then compare its predictions over a wider validation window to the true Lagrangian's results. Notably, this converges much more unsteadily, tending to halt by increasing loss after a long epoch near what appears to be a local minimum (Fig. 27). The results for the fit over the sampling window are at surface level fairly good, although the pinched line from lower left to upper right is rotated to be horizontal in this fit (Fig. 15).

Figure 15: Sampled points and trained polynomial Lagrangian results, run 3 (narrow sampling window)



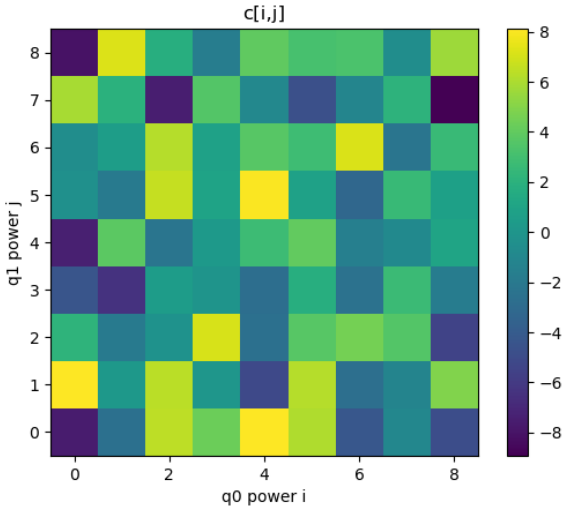
Expanding the sampling window to a validation window covering both fixed points show how poor this fit truly is – despite matching well qualitatively on the narrow window, Fig. 16 shows that away from the training region, the fit completely fails and produces results that are nothing like the true $\ddot{q}$. Compared to the exact form of the Lagrangian that I fit before, this polynomial appears massively undercontrained when the sampling is inadequate.

Figure 16: Sampled points and trained polynomial Lagrangian results, run 3 (narrow sampling window)



This is to some extent the expected behavior – for this run the parameter tensor $c_{i,j}$ has $9^2 = 81$ independent elements! I've included the $q^i_m \cdot q^0_n$ terms to allow the system to fit those terms as well. In a future update I will fit the $\ddot{q}$ values for the true Lagrangian to see what kind of form they truly follow in the fixed point regions.

Figure 17: Polynomial coupling term coefficients, run 2



3 Algorithm improvements

3.1 Plotting loss landscapes

Previously, to plot an $n \times n$ density plot image, the code iterated via two nested for-loops. This does not take advantage of the full vectorization potention of PyTorch, so I've rewritten it to vectorize along one dimension.

To begin, produce an extended version of the dataset by iteratively concatenating (slow, but workable since the main loss calculation is slow):

```
n_param_points = 1 #number of copies needed

dataset_expanded = dataset
for i in range(1,n_param_points):
   dataset_expanded = torch.cat((dataset_expanded,dataset))
```

This same process is repeated for the $(q, \dot{q})$ sets and the parameter grid for α. Then, redefine the loss function to use the unvectorized function $\ddot{q}$, then wrap it via torch.func.vmap:

```
def loss(q,qdot,m,k,alpha,dataset):
   test_qdd = QDD(q,qdot,m,k,alpha)
   return torch.sum(( (test_qdd - dataset) )**2 )#/(dataset.size()[0]**2)

lossv = vmap(loss,in_dims=(0,0,None,None,0,0))
```

This new function lossv takes in an expanded dataset $(\vec{q}, \dot{\vec{q}}, m, k, \vec{\alpha})$ (where the vector symbol indicates the expanded dataset), and operates on all elements via PyTorch's built-in vectorization. This produces a dramatic speedup – for 200×200 pixel images, about 25x.

3.2 Gradient calculations

A more effiient formatting for gradient calculations is rewriting the Lagrangian function and all of the higher-order functions as follows:

```
def L(q,qdot,param):
   return ...

def QDD(q,qdot,param):
   ...
   return qdd

def loss(q,qdot,param,dataset):
   return sum( (qdd-dataset)**2 )/dataset.size**2

parameter_gradient = jacrev(loss,argnums=(2))
```

Here I'm collecting all of the parameters in one tensor object and passing it, instead of passing individually named parameters like m, k, α, etc. The parameter_gradient function now returns a tensor of the same size and shape as the parameter tensor containing the gradient of each parameter, so the gradient descent algorithm simplifies to:

```
parameters = torch.randn([size_0,size_1])

for i in range(n_iterations):
   loss_history[i] = loss(Q,QD,parameters)
   parameters.data -= learning_rate*parameter_gradient(Q,QD,parameters)
```

This makes the algorithm more structure-agnostic, simplifying the addition of more complicated Lagrangians like polynomials, neural nets, etc.

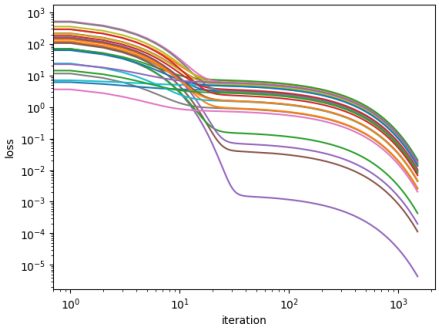
Appendix A Figures

A.1 Exact Lagrangian form

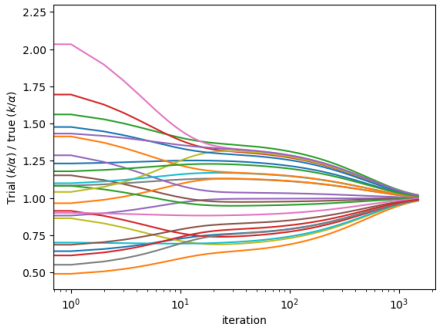
A.1.1 Basic results

Figure 18: Gradient descent performance profile

(a) Loss vs. iterations



(b) (k_{exp}/α_{exp}) / (k_{true}/α_{true}) vs. iteration



A.1.2 Sampling far from the origin

Figure 19: Sampled points and associated $\ddot{q}$, for (q_0, q_1) displaced from the origin

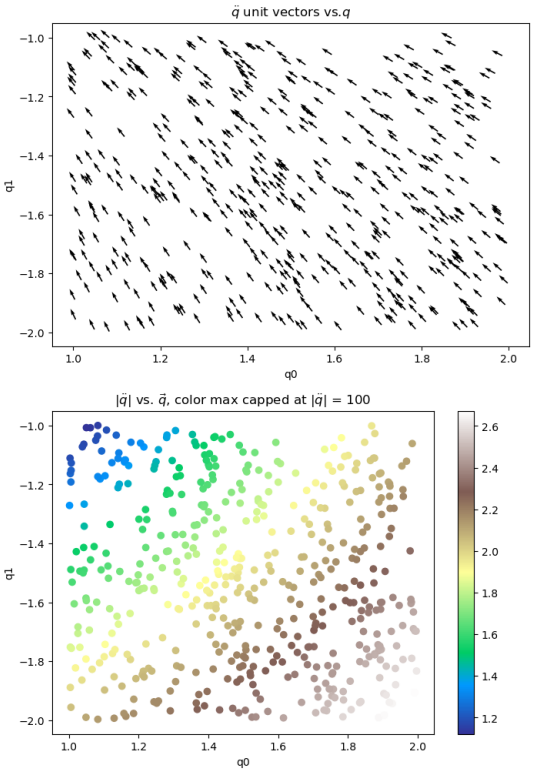
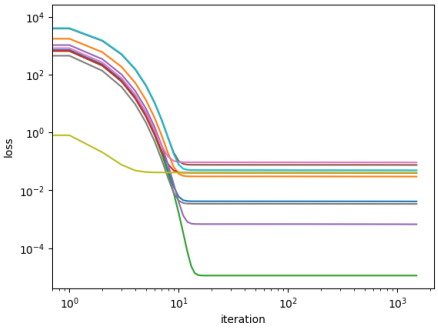
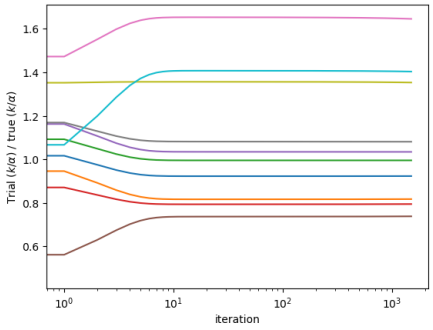


Figure 20: Gradient descent performance profile, displaced from the origin

(a) Loss vs. iterations



(b) (k_{exp}/α_{exp}) / (k_{true}/α_{true}) vs. iteration



A.1.3 Sampling near a fixed point

Figure 21: Sampled points and associated $\ddot{q}$, for (q_0, q_1) near a fixed point

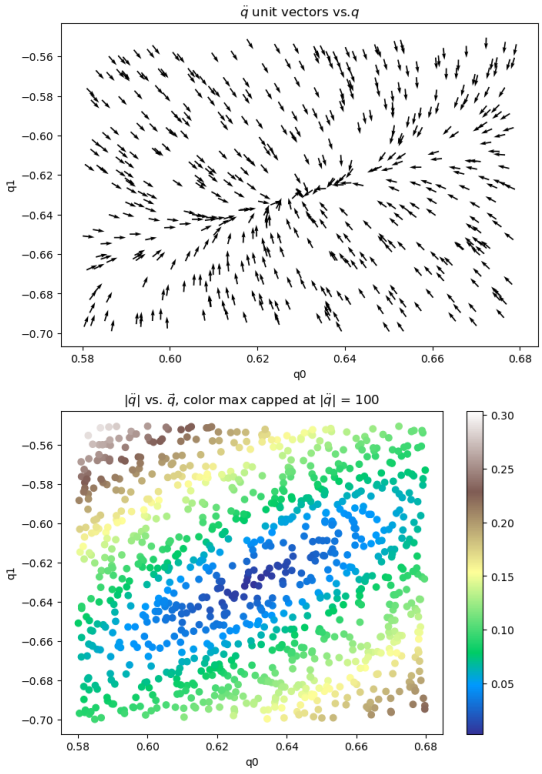
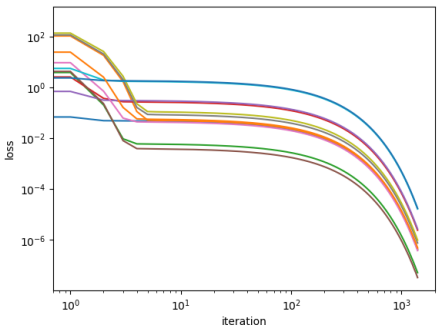
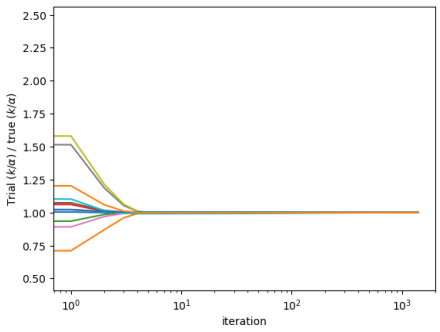


Figure 22: Gradient descent performance profile, near a fixed point

(a) Loss vs. iterations



(b) (k_{exp}/α_{exp}) / (k_{true}/α_{true}) vs. iteration



A.1.4 Sampling very close to a fixed point

Figure 23: Sampled points and associated $\ddot{q}$, for (q_0, q_1) near a fixed point

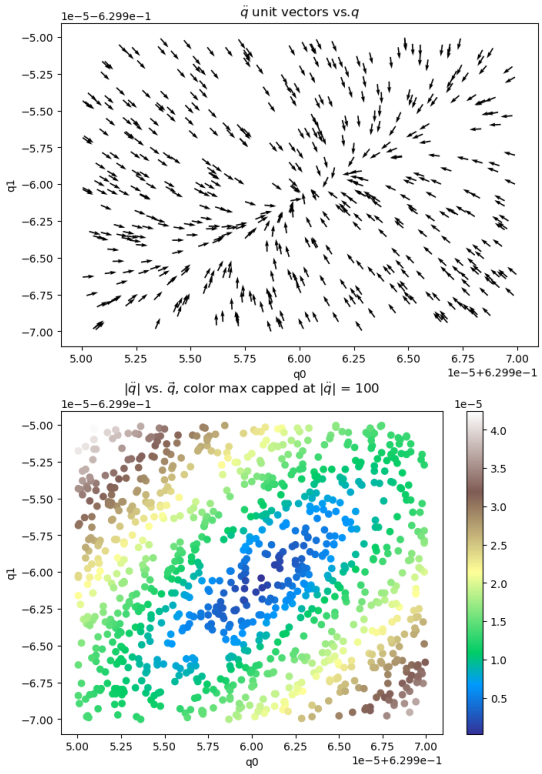
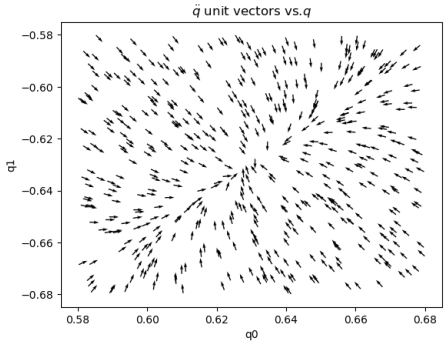
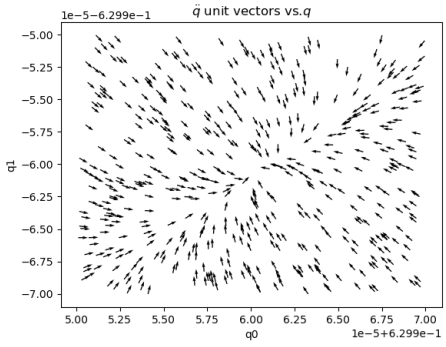


Figure 24: Comparison of sampling near a fixed point for a large range of sampling area sizes

(a) Box width = 0.1



(b) Box width = 0.0002



A.2 Polynomial Lagrangian

Figure 25: Training run 1: Loss vs. iteration

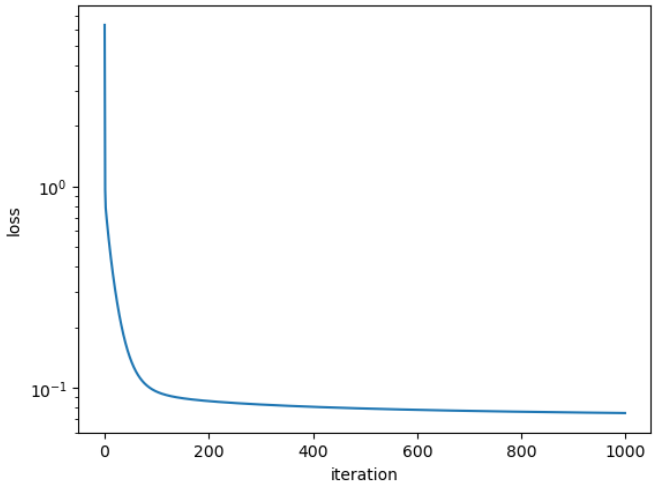


Figure 26: Training run 2: Loss vs. iteration

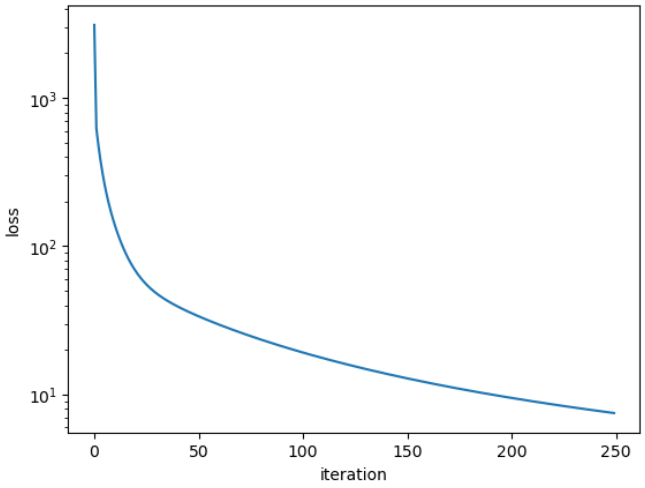
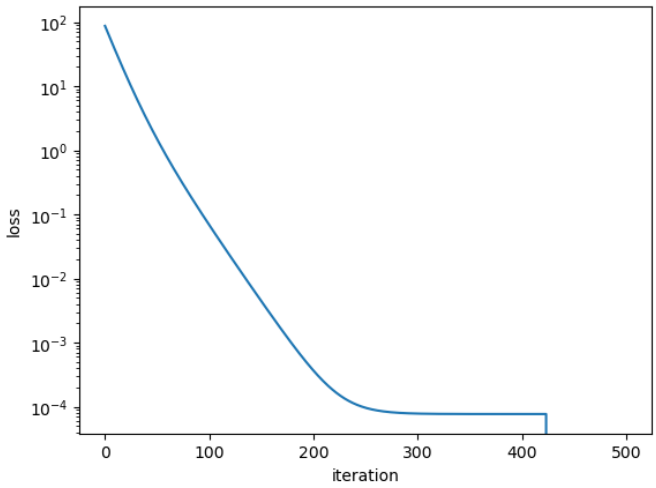


Figure 27: Training run 3 (narrow sampling window): Loss vs. iteration



Appendix B Bibliography

References

[1] John Duchi and Yoram Singer. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization * Elad Hazan". In: *Journal of Machine Learning Research* 12 (2011), pp. 2121–2159.

[2] David Griffiths. *INTRODUCTION TO ELECTRODYNAMICS*. Ed. by Jim Smith, Martha Steele, and Corinne Benson. 4th. Pearson Education, Inc., 2013. ISBN: [“–]Duchi2011}.

[3] Diederik P. Kingma and Jimmy Lei Ba. "Adam: A Method for Stochastic Optimization". In: *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings* (Dec. 2014). URL: https://arxiv.org/abs/1412.6980v9.

[4] John R Taylor. *Classical Mechanics*. Ed. by Lee Young. 1st ed. University Science Books, 2005.

[5] Eric W. Weisstein. "Binomial Series". In: *MathWorld* (Apr. 2025). URL: https://mathworld.wolfram.com/BinomialSeries.html.