May 6th Weekly Report

Erik Bigwood

May 9, 2025

Abstract

Showing work completed up to May 6.

Contents

1 Harmonic Oscillator Demonstration **2**

2 Algorithm improvements **6**
2.1 Calculating $\ddot{q}$ 6
2.2 Calculating gradients 7

3 Derivations **8**
3.1 Lagrangian 8
3.2 Analytical equations of motion 8

A Figures **9**

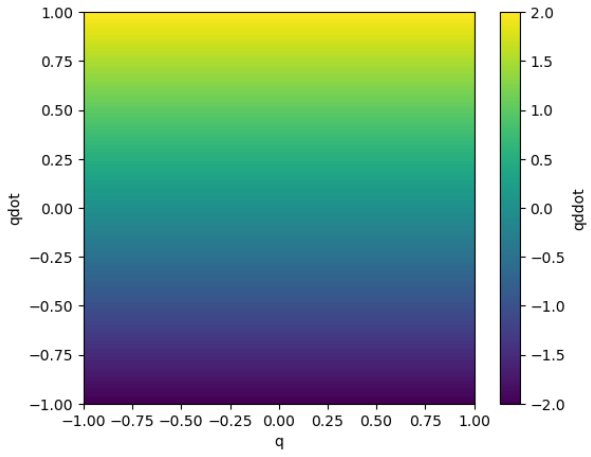
B Bibliography **11**

1 Harmonic Oscillator Demonstration

Basic algorithm See the section "Algorithm improvements" for explanations of the actual code used here.

To begin, we define the problem setup – m, k, n, and the bounds for q and $\dot{q}$. For this demo, I've defined $(m_k) = (2, 4)$ s.t. $k/m = 2$. Define the Lagrangian function, and the methods QDD and QDDv to calculate $\ddot{q}$ at each $(q, \dot{q})$ point. Create n point grids over q and $\dot{q}$ and flatten those grids to $(n^2, 1)$ arrays. Then, create a synthetic dataset with the given (m, k) and store it.

Figure 1: $\ddot{q}$ vs. $(q, \dot{q})$

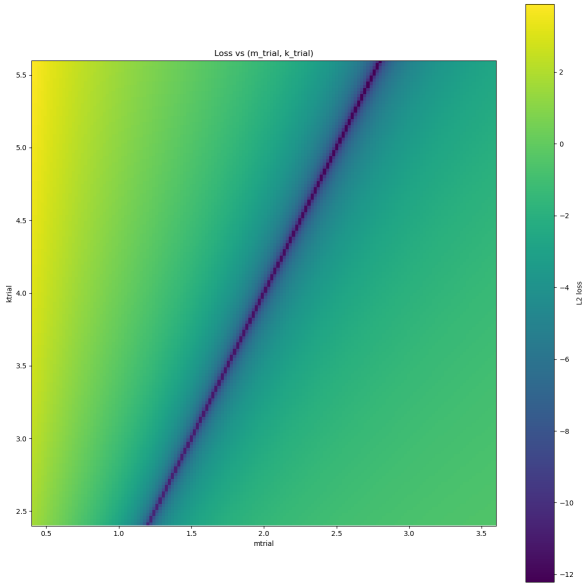


Here I define the loss/error function as the summed L^2 norm of the difference between predicted and true data:

$$loss = \frac{1}{N} \sum_{i=1}^{N} \left(\ddot{q}_{trial} - \ddot{q}_{true} \right)^2 \tag{1}$$

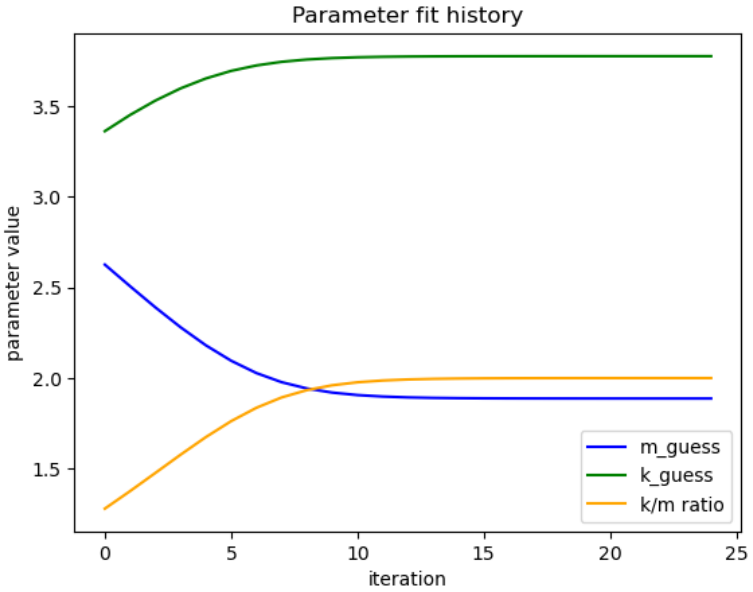
Now,disregard the specified (m, k) values given in the problem setup and calculate the summed L^2 loss function for a grid of trial (m_{trial}, k_{trial}) points. Due to the structure of this Lagrangian, the loss minimum is the line from the origin where $k_{trial}/m_{trial} = k/m$. The loss becomes extremely large as m_{trial} becomes small, which will require some adjustment to the gradient descent algorithm later.

Figure 2: Loss landscape : $\frac{k}{m} = 2$



As a simple demonstration, take a random (m_{trial}, k_{trial}) point and perform gradient descent, shown in 3. The parameters (m_{trial}, k_{trial}) converge to a solution with $k/m = 2$ as expected, but due to the symmetry in the loss landscape, the actual values depend on the initial guess.

Figure 3: Gradient descent parameter fit : $\frac{k}{m} = 2$



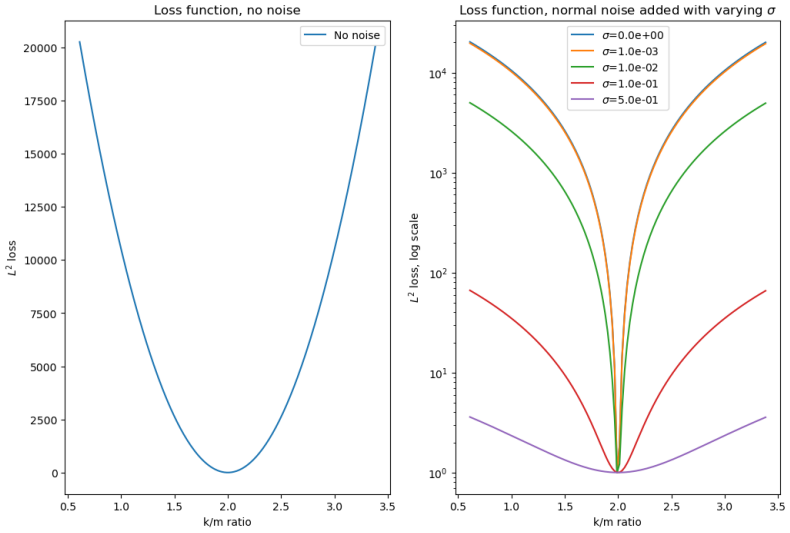
Finally, Fig. 8 contains a plot with the loss landscape and a series of parameter paths for a large number of random initial guesses of (m_{trial}, k_{trial}). As mentioned earlier, the extreme range of loss values is compensated for by including a modification to the learning rate given by:

$$\epsilon = \epsilon_0 e^{-\left(\epsilon_0 \left|\nabla_{m,k} loss\right|\right)^3} \tag{2}$$

This reduces the effective learning rate when $|\nabla_{m,k} loss|$ becomes large, so that the parameters still evolve slowly enough to converge and not skip over the solution space.

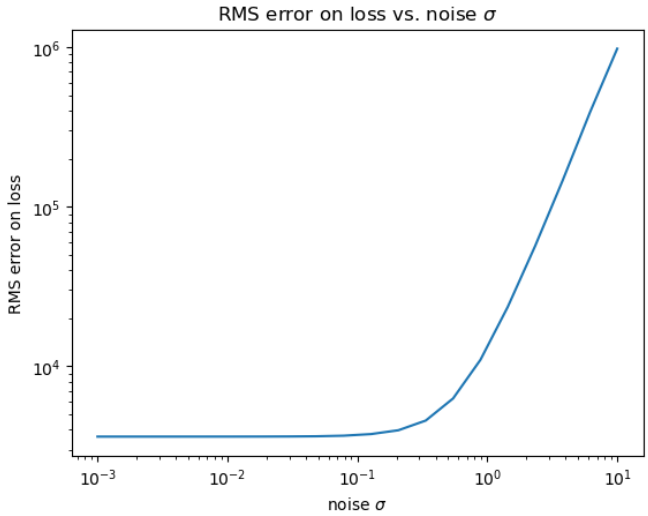
Adding noise Adding noise to the synthetic dataset complicates the situation. In Fig. 4, I've included a plot comparing the loss function on a slice perpendicular to the solution space. On the left we have the unperturbed loss function, and on the right the loss function under a perturbation by normally distributed noise on each point in the $q, \dot{q}$ grids.

Figure 4: Loss vs noise on a slice perpendicular to the solution space



Comparing the RMS error on the loss function caused by perturbation by noise (Fig. 5) leads to a notable trend – $\sigma \leq 10^{-1}$ is effectively undetectable, but increasing noise intensity beyond that threshold leads to a dramatic increase in RMS error. More specifically, the loss function flattens out dramatically, washing out the nicely defined valley of low loss in the solution space. This demands more careful choice of the trial (m_{trial}, k_{trial}) values and learning rate ϵ_0. One potential approach is to evaluate the gradients at a large number of intial guesses and choose ϵ_0 to restrict the gradient descent step size for the largest gradient sampled.

Figure 5: RMS error on loss vs. noise σ on a slice perpendicular to the solution space

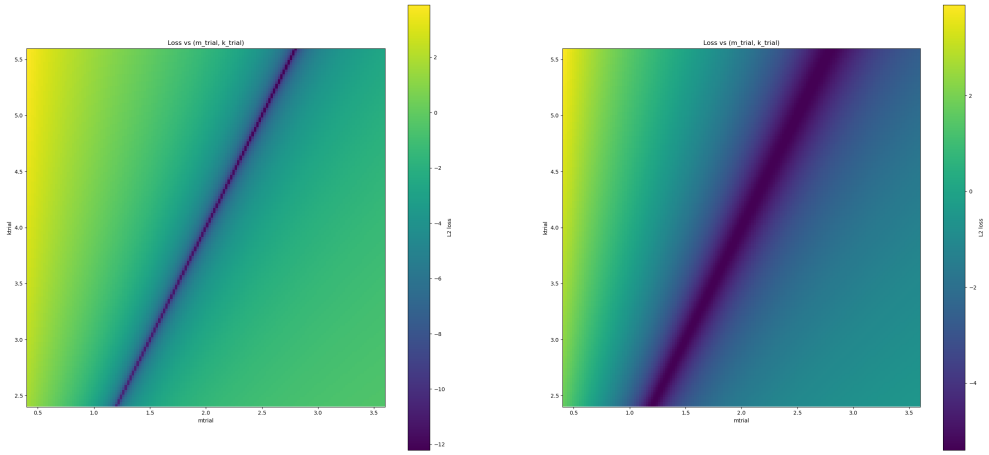


In Fig. 6, I've included density plots of the loss landscapes with and without noise. Including noise broadens the valley of low loss and dramatically reduces the range of the loss function values, decreasing the gradients and slowing down convergence without compensating in the gradient descent algorithm.

Figure 6: Loss landscape with and without noise

(a) Loss landscape, no noise

(b) Loss landscape w/ noise, $\sigma = 0.2$



2 Algorithm improvements

2.1 Calculating $\ddot{q}$

My previous work was based on using Pytorch[2]'s torch.autograd's Hessian and Jacobian to compute $\ddot{q}$ [1]. Notably, these functions calculate *all* derivatives/2nd derivatives at all points, which can be a significant waste of compute. This week I've re-written the core code to utilize the torch.func module. This module includes torch.fun.jacfwd (Jacobian, forward-mode automatic differentiation) which I'm using to calculate derivatives, and torch.func.vmap which is an auto-vectorization method that wraps around a function and handles batching seamlessly. I suspect the torch.func methods are creating the computational graph in the background and tying everything together instead of re-creating it at each iteration.

My older method for calculating $\ddot{q}$ follows roughly this example (the indexing/linear algebra is simplified here by only considering a 1D harmonic oscillator[3]):

```
def L(vec): #Harmonic oscillator Lagrangian density
   #input: array (m,k,q,qdot), returns Lagrangian density
   m, k, q, qdot = vec[0], vec[1], vec[2], vec[3]
   return 0.5*m*qdot**2 - 0.5*k*q**2

def qddot(Lag,m,k,q,qdot):
   #Returns d^2/dt^2 q given Lagrangian L
   v = torch.cat((m,k,q,qdot))
   full_hessian = autograd.functional.hessian(Lag,v)
   full_grad = autograd.functional.jacobian(Lag,v)

   grad_q = full_grad[2]
   hess_q = full_hessian[2,2]
   hess_qdot = full_hessian[3,3]
   hess_q_qdot = full_hessian[2,3]
   qdd = (hess_qdot**(-1))*( grad_q - hess_q_qdot*qdot )
   return qdd

m = tensor([1.])
k = tensor([2.])

out = torch.zeros(n_points**2) #out[i] = qdd(q[i], qdot[i])

#define q,qdot grids
qgrid = torch.linspace(q_min,q_max,n_points)
qdotgrid = torch.linspace(qdot_min,qdot_max,n_points)

for i in range(n_points): #iterate over q
   for j in range(n_points): #iterate over qdot
      q = tensor([qgrid[i]],requires_grad = True) #store iterated q
      qdot = tensor([qdotgrid[j]],requires_grad = True) #store iterated qdot
      qdd = qddot(L2,mold,kold,q,qdot) #calculate qddot
      out[i] = qdd #store qddot
```

After refactoring, the algorithm takes this form:

```
def L(q,qdot,m,k,alpha):
```

```
   return 0.5*m*qdot**2 - 0.5*k*q**2 - alpha*q**2*qdot**2

#define grid of q, qdot points
qgrid = torch.linspace(q_min,q_max,n_points) #(n_points)
qdotgrid = torch.linspace(qdot_min,qdot_max,n_points) #(n_points)

#define flattened q, qdot arrays (n_points^2), (n_points^2)
Q = qgrid.expand([n_points,n_points]).reshape(n_points**2)
QD = qdotgrid.expand([n_points,n_points]).reshape(n_points**2)

#####
# define functions. vmap is used to auto-vectorize

def QDD(Q,QD,m,k,alpha):
   g_q = jacfwd(L,argnums=0)(Q,QD,m,k,alpha) #grad L w.r.t. q
   g_qdot= jacfwd(L,argnums=1)(Q,QD,m,k,alpha) #grad L w.r.t. qdot

   g_q_qdot = jacfwd(jacfwd(L,argnums=1),argnums=0)(Q,QD,m,k,alpha) #d^2L/(dq dqdot)
   g_qdot_qdot = jacfwd(jacfwd(L,argnums=1),argnums=1)(Q,QD,m,k,alpha) #d^2L/(dqdot
       dqdot)
   D = g_qdot_qdot**(-1) #(d^2L/(dqdot dqdot))^-1. in this case, this is 1x1, so
       matrix inversion is just val^-1
   return D*(g_q - g_q_qdot*QD) #return qddot

QDDv = vmap(QDD,in_dims=(0,0,None,None,None)) #vectorize

out = QDDv(Q,QD,m,k,alpha) #calculate and store qddot for each point in flattened
    q,qdot arrays
```

This improvement to the core algorithm results in up to a 3300x speedup for large array sizes! See Fig. 7 Out to about 10,000 points, the v2 algorithm takes roughly the same amount of time per run due to vectorization, so there is no performance hit associated with increasing the number of points up to this threshold. The old method (nested for-loop iteration) takes a constant amount of time per point.

2.2 Calculating gradients

The method for calculating gradients w.r.t. the Lagrangian's parameters has also been improved in the same way. Below I've included the code for performing gradient descent w.r.t. m and k.

```
   # define gradient w.r.t. parameters
   dlossdm = jacfwd(loss,argnums=2) #dloss/dm
   dlossdk = jacfwd(loss,argnums=3) #dloss/dk

   trial_iters = 100 #number of gradient descent iterations
   learning_rate = 1e-1 #\epsilon

   guess_m = 2*box_width*(torch.rand(1)-0.5)+m[0] #random initial guess for m
   guess_k = 2*box_width*(torch.rand(1)-0.5)+k[0] #random initial guess for k
```

```
guess_m_array = torch.zeros(trial_iters) #initialize history array
guess_k_array = torch.zeros(trial_iters) #initialize history array
iter_array = torch.arange(0,trial_iters,1) #array containing iteration numbers

for i in range(trial_iters): #iterate
   guess_m.data += -learning_rate*dlossdm(Q,QD,guess_m,guess_k,alpha,out)
       #gradient descent step
   guess_k.data += -learning_rate*dlossdk(Q,QD,guess_m,guess_k,alpha,out)
       #gradient descent step

   guess_m_array[i] = torch.clone(guess_m) #store updated m
   guess_k_array[i] = torch.clone(guess_k) #store updated k
```

3 Derivations

3.1 Lagrangian

For Lagrangian mechanics[3], we have the action

$$S = \int_{t_0}^{t_f} \mathcal{L}(q_i, \dot{q}_i dt = \int_{t_0}^{t_f} \big(T - U\big) dt \tag{3}$$

From Hamilton's principle we get the Euler-Lagrange equation (derivation from [1]):

$$\frac{d}{dt}\nabla_{\dot{q}}\mathcal{L} = \nabla_q \mathcal{L} \tag{4}$$

Where $(\nabla_{\dot{q}})_i \equiv \frac{\partial}{\partial \dot{q}_i}$. Expanding the time derivative through, we get

$$\big(\nabla_{\dot{q}}\nabla_{\dot{q}}^T\mathcal{L}\big)\ddot{q} + \big(\nabla_q \nabla_{\dot{q}}^T \mathcal{L}\big) = \nabla_q \mathcal{L} \tag{5}$$

$$\ddot{q} = \big(\nabla_{\dot{q}}\nabla_{\dot{q}}^T\mathcal{L}\big)^{-1}\big[\nabla_q \mathcal{L} - \big(\nabla_q \nabla_{\dot{q}}^T \mathcal{L}\big)\dot{q}\big] \tag{6}$$

With $\big(\nabla_q \nabla_{\dot{q}}^T \mathcal{L}\big) = \frac{\partial^2 \mathcal{L}}{\partial q_j \partial \dot{q}_i}$.

3.2 Analytical equations of motion

Harmonic Oscillator Take the harmonic oscillator Lagrangian

$$\mathcal{L} = \frac{1}{2}m\dot{q}^2 - \frac{1}{2}kq^2 \tag{7}$$

Apply the Euler-Lagrange equations [3]:

$$\frac{\partial \mathcal{L}}{\partial q} = \frac{d}{d}\frac{\partial \mathcal{L}}{\partial \dot{q}} \tag{8}$$

$$-kq = \frac{d}{dt}m\dot{q} \tag{9}$$

$$\ddot{q} = -\frac{k}{m}q \tag{10}$$

Appendix A Figures

Figure 7: Old method vs. v2 autovectorization speed comparison

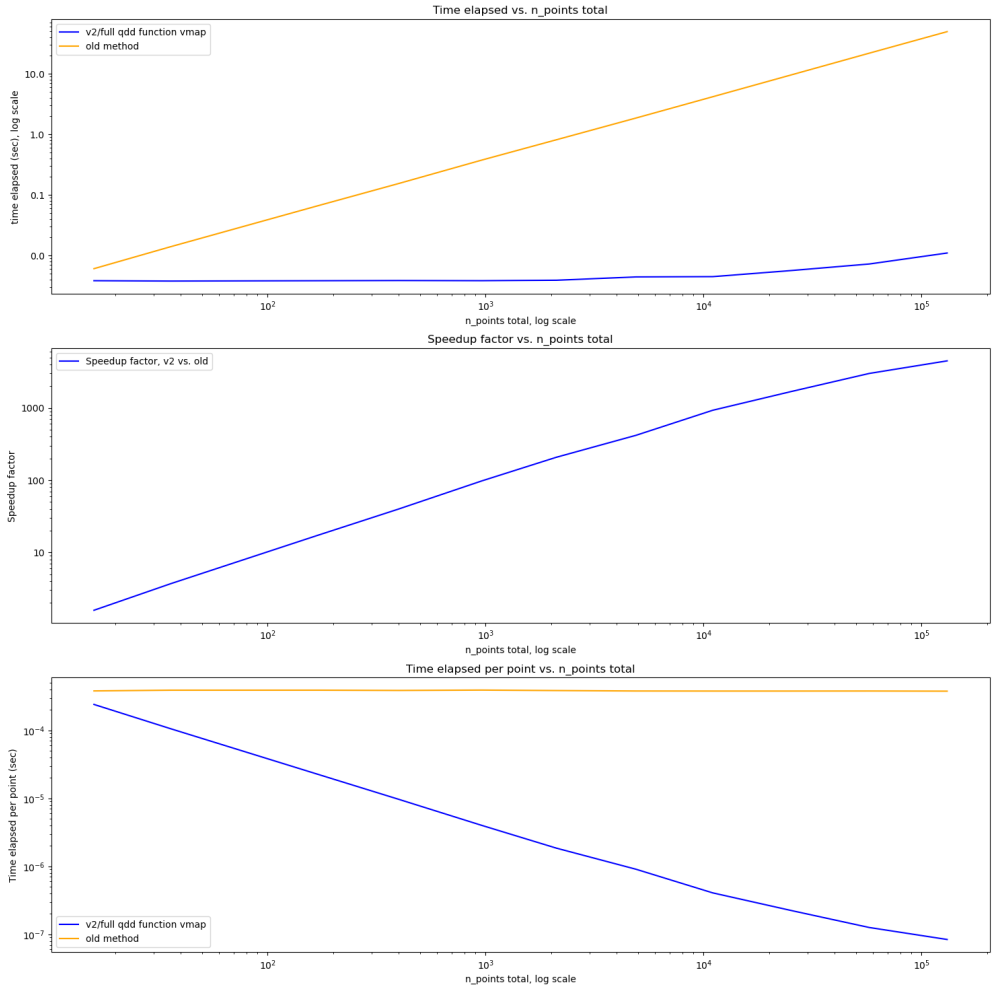
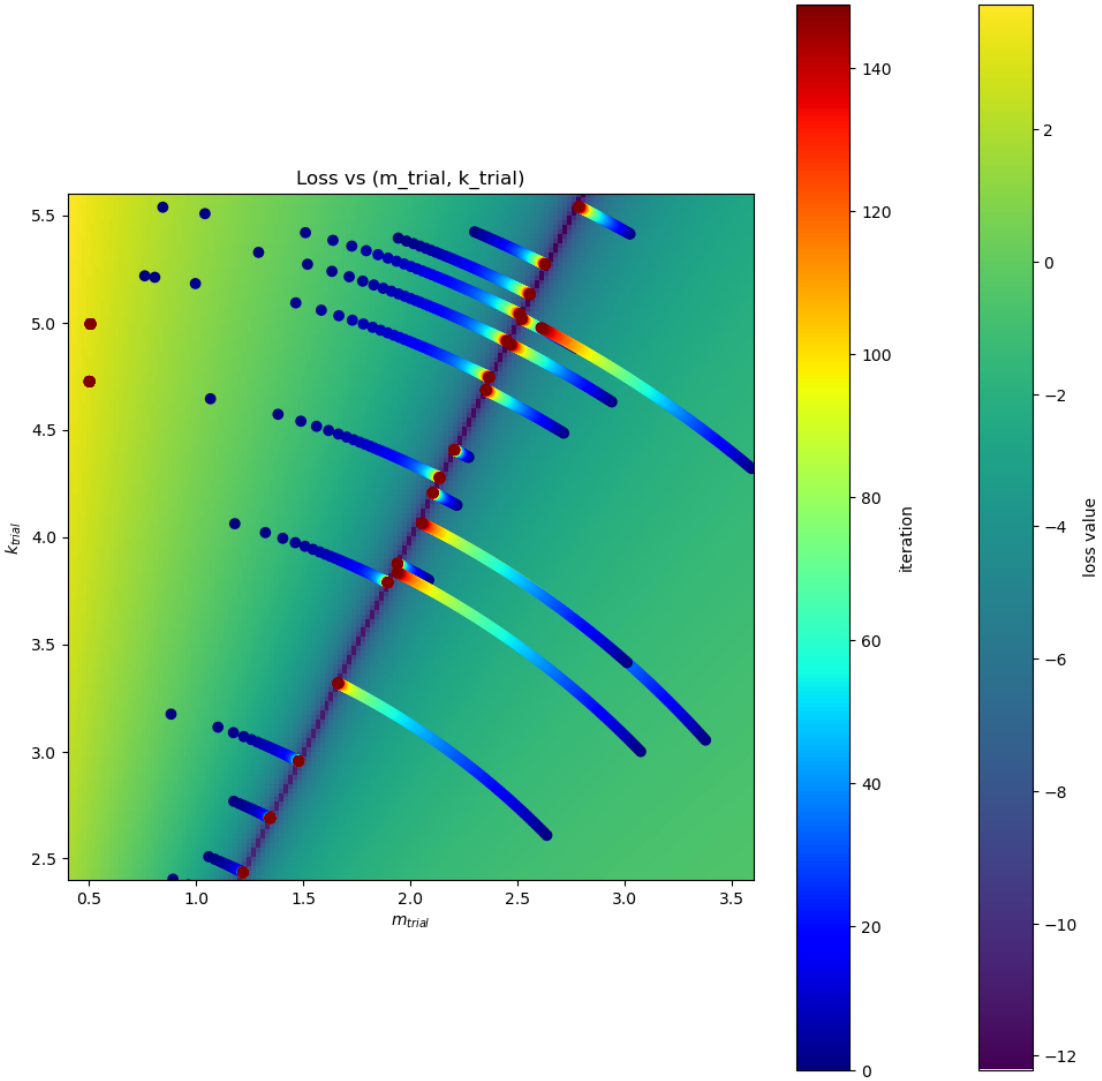


Figure 8: Gradient descent parameter fit for multiple paths : $\frac{k}{m} = 2$



Appendix B Bibliography

References

[1] Miles Cranmer et al. "Lagrangian Neural Networks". In: (May 2020). URL: https://arxiv.org/pdf/2003.04630.pdf.

[2] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: (2019).

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