Physics-informed neural networks for a system of discrete masses and springs

Chuah Jonathan,Yu Tz-Ching, Havolli Albias

Laboratory Supervisor: Sharma Vinay from IMOS EPFL laboratory

***Abstract*—Physics-informed neural networks (PINNs) are an exciting and novel approach for solving challenging inverse problems in physics and engineering. In this work, we apply PINNs to predict the forces acting on a system of discrete masses and springs. To do so, we endow our neural networks with knowledge of the physical laws specific to our system, such as Newton's 2nd law. We show that imposing these physics-informed losses and constraints allows the model to effectively learn the system's underlying physics. A preliminary application to parameter identification – computing the system's stiffness matrix – is also demonstrated.**

I. Introduction

Artificial neural networks (ANNs) have been used to solve problems in domains ranging from image classification and computer vision to natural language processing, among many others. More recently, a new promising application domain has emerged: the solution of partial differential equations (PDEs) using ANNs that learn physics, which are referred to as physics-informed neural networks (PINNs). PINNs differ from standard supervised ML problems as they do not train purely on data, but also exploit domain knowledge of the physical systems being solved wherever possible. In addition to the usual regression loss, loss terms with physical meaning are added to the objective function which enforce the trained networks to satisfy physical laws.

Indeed, this new class of ANNs has been developed to embed network models with known equations that governs the physics of a system and exploits the input data as well as currently available techniques of automatic differentiation to compute approximations of all terms involved in the differential equations and the initial and boundary conditions at every point in space and in the simulation time.

A PINN architecture requires the classical elements of feedforward and backpropagation of neural networks like hidden layers, neurons and activation functions. Moreover, the input data are collocation points for fitting the governing differential equations and the training is performed by using optimization techniques. Finally, PINNs are based on the universal approximation theorem. That's a key point as this theorem implies that a neural network can approximate any kind of function and therefore that PINNs can solve any PDEs.

PINNs have been shown to perform well for different types of PDEs, like in fluid [1] or in solid mechanics [2]. However, their utility for solving forward problems is limited as they are generally still slower than traditional numerical methods [3]. Instead, the main research interest in PINNs comes from their potential as computationally efficient *inverse problem* solvers, to infer the "hidden" parameters of a system from raw measurement data. In [2], parameter identification was successfully united with the machine learning concept of *transfer learning* in order to learn the elastic coefficients of a deformed sheet material. We wish to investigate whether the same approach can be used in a dynamical systems setting.

In this paper, we develop, implement and train PINNs in a system composed of discrete masses and springs. We study spring-mass systems because they are relatively simple toy problems but also admit useful extensions to fields such as continuum mechanics and beam deflection theory. After training, we test our PINN on a different setup by changing the number of masses. Finally, we apply the transfer learning techniques of [2] to infer the stiffness coefficients of an unknown spring-mass system.

II. Spring-mass systems

A 2-dimensional discrete spring-mass system comprises a set of N identical point masses of mass m, which are linked by $N-1$ identical springs with stiffness coefficients k and rest lengths L_0 (Figure 1). With the exception of masses $i=0$ and $i=N-1$, which are fixed in place for all t, the masses' motion is described by Newton's 2nd law, which results in a system of coupled ordinary differential equations (ODEs) :

$$m\frac{d\mathbf{x}_i}{dt} = m\mathbf{g} - F(\Delta\mathbf{x}_i) + F(\Delta\mathbf{x}_{i+1})$$

where $i = 1, ..., N-2$ and $\Delta\mathbf{x}_i = \mathbf{x}_i - \mathbf{x}_{i-1}$. The spring force is modeled by Hooke's law:

$$F(\Delta\mathbf{x}_i) = k(\|\Delta\mathbf{x}_i\| - L_0)\frac{\Delta\mathbf{x}_i}{\|\Delta\mathbf{x}_i\|}.$$

Notably, parameters such as k, m, and L_0 are "hidden" in the ODE coefficients and are difficult to infer directly from the trajectory data, $\mathbf{x}_i(t)$ — a classic case of an inverse problem.

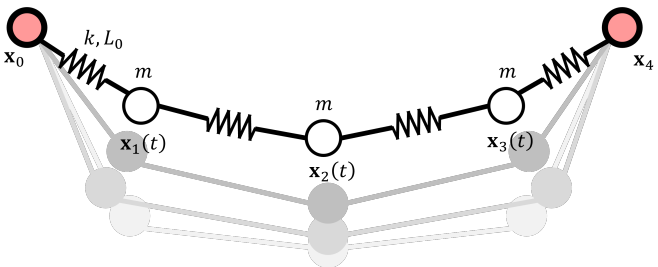

Fig. 1: System of discrete masses & springs with $N = 5$

Like other machine learning models, a PINN needs to be trained against high quality ground truth data. For this project, the training data was obtained using high-fidelity numerical simulation. We first implemented a numerical ODE solver in scipy and chose baseline parameter values of $k = 50\ \mathrm{N\,m^{-1}}$, $L_0 = 7$ m and $m = 1$ kg. Then, system trajectories were computed for a duration T_d of 0-40s with a time step T_s of 0.01s. We obtained trajectories for two different numbers of masses – $N_1 = 5$ and $N_2 = 8$ – which formed the data sets on which our PINNs were trained and tested. Each data set can be represented in matrix form with T_d/T_s rows and $6(N-2)+5$ columns. The first data set S_1 includes 5 masses and therefore is a size 4000x23 matrix. The 1st column represents the elapsed time; the two following represent the position in global coordinates of the left fixed mass ($i = 0$). The next columns represent the positions x and y, the velocities $\dot{x}$ and $\dot{y}$, and accelerations/forces $\ddot{x}$ and $\ddot{y}$ of the movable masses, i.e. $i = 1, 2, 3$. Finally, the last two columns represent the x and y position of the right fixed mass ($i = 4$). The second data S_2 includes 8 masses and similarly forms a 4000x41 matrix. The data set S_1 was used to train the PINNs and S_2 to test it.

III. PINN ARCHITECTURE

The PINN framework offers a lot of flexibility in how machine learning is incorporated into the PDE solution process. For instance, various authors have chosen to model the system's time-series trajectory [4], its Hamiltonian [5], or its state transition matrix [6] as a neural network. For this project, we opt to predict the net force exerted on the i-th mass given its position, $\mathbf{x}_i$, and the positions of its neighboring masses, $\mathbf{x}_{i-1}$ and $\mathbf{x}_{i+1}$, as input. For simplicity, we use a fully-connected network architecture with $\tanh$ activation functions. As an added inductive bias, we also add a "zeroth" layer that takes the input $\{\mathbf{x}_{i-1}, \mathbf{x}_i, \mathbf{x}_{i+1}\}$ and computes $\{\mathbf{x}_i - \mathbf{x}_{i-1}, \mathbf{x}_i - \mathbf{x}_{i+1}\}$, since the forces should only depend on the relative positions of the masses (Figure 2a).

To compute forces on the entire spring-mass chain, we simply run the network on consecutive sequences of 3 masses along the chain, starting from the leftmost mass (Figure 2b). After computing the forces on each mass at the current time step, we numerically integrate the masses' positions and velocities for one time step using scipy's solve_ivp routine with a Dormand-Prince RK45 scheme, in accordance with Newton's 2nd law. In summary, the full neural trajectory solver comprises a small neural network that computes element-wise forces, an aggregation step, and a numerical time integrator.

A. Physics-informed loss functions

Our model's loss contains two parts: a traditional data regression loss, and physics-informed loss (aka PINN loss) that allows physically meaningful constraints to be imposed on the network's output. For the regression loss, we fit the network's force predictions against the ground truth forces, F^*, at every sample in the training set:

$$\mathcal{L}_{data} = \sum_{k \in S} \| F(\mathbf{x}_{i-1}(t_k), \mathbf{x}_i(t_k), \mathbf{x}_{i+1}(t_k)) - F^* \|^2$$

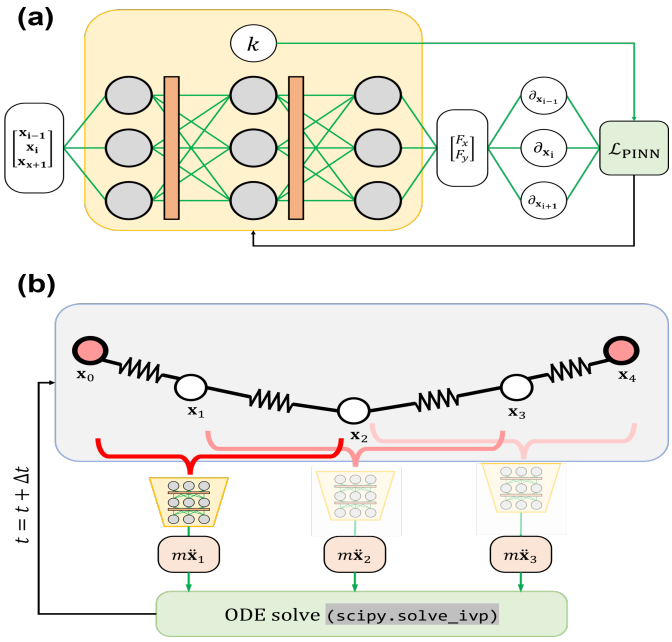

Fig. 2: ML architectures for spring-mass systems. (a) PINN force prediction model, (b) PINN-based trajectory predictor

Meanwhile, the PINN loss comprises three terms. The first two operate on the force Jacobian, $J = \nabla F$, and the relative positions between neighboring masses, $\hat{\mathbf{s}} = \mathbf{x}_i - \mathbf{x}_{i\pm1}/\|\mathbf{x}_i - \mathbf{x}_{i\pm1}\|$. They attempt to impose a fixed eigen-decomposition on J; intuitively, this captures the notion that the spring force always acts along the axis of a spring, and has no orthogonal force component.

$$\mathcal{L}_{PINN} = \sum_{k \in S} \|\hat{\mathbf{s}}^T J \hat{\mathbf{s}} - k\|^2 + \|\hat{\mathbf{s}}^T J \hat{\mathbf{s}}_\perp\| + \|\partial_x F_y - \partial_y F_x\|^2$$

The third term imposes energy conservation indirectly and is inspired by the classical mechanics notion of a conservative force. These forces always preserve a system's total energy: they mediate the inter-conversion of kinetic and potential energy, but will neither add nor dissipate energy from the system. It can also be shown that such forces must be *curl-free* everywhere in the domain, i.e. $\nabla \times F = 0$. By imposing the latter condition as a soft constraint, we encourage the model to find solutions that approximately conserve energy. Finally, the overall loss function $\mathcal{L}$ is simply a weighted sum of $\mathcal{L}_{data}$ and $\mathcal{L}_{PINN}$. For detailed derivations of the loss terms, kindly refer to the appendices A and B.

B. Training and evaluation

We train the PINN model using Pytorch's Adam optimizer with a fixed learning rate of 9e-4 and weight decay of 1e-3. For the regression loss term, we perform a training-validation split according to the trajectory time steps: the network trains on the first 5s of trajectory data, while the time steps in $5\text{s} \leq t \leq 20\text{s}$ are held out for validation. The remaining data from $20\text{s} \leq t \leq 40\text{s}$ is used as the test set. In constrast, the PINN loss is imposed at *collocation points* that are evenly

distributed over the whole system trajectory, since physics should be satisfied everywhere.

Finally, similar to rollouts in reinforcement learning, we apply the network's force predictions recursively to compute time-series trajectories and compare them against the ground truth system trajectories. Starting from the ground truth's initial state, the model computes forces which are integrated to determine the state at the next time step, which is used to compute the next set of forces, and so on. Note that this is a more challenging and robust test of generalizability than the traditional test-val-train evaluation on the force data: the model must re-produce a full-duration trajectory close to the ground truth's while only having access to the initial state.

C. Hyperparameters selection

As mentioned at the end of section I, there is no automatic way to find to find an effective ANN architecture. Therefore, it is necessary to optimize over the hyperparameters in order to find the optimal combination with the least test/train error. For our study, we fix the learning rate ($lr = 9 \cdot 10^{-4}$) and training epochs (6000) and perform a parametric sweep of the hidden layer depth $l = \{3, 5, 7\}$, and neurons/width, $w = \{16, 32, 64\}$. For robustness, we train and evaluate 5 models for each $\{l, w\}$ combination. We report the obtained rollout trajectory mean-squared-error (MSE), averaged across the 5 models, and their spread in Table I.

l \ w	Mean squared error			Standard deviation		
	16	32	64	16	32	64
3	4.7e-03	2.7e-03	6.1e-02	6.2e-03	9.3e-04	6.9e-03
5	1.2e-02	6.6e-03	2.4e-02	7.8e-03	9.8e-03	1.8e-03
7	4.6e-03	7.4e-03	1.1e-02	2.6e-03	4.8e-03	5.2e-03

TABLE I: MSE between the predicted and truth trajectories and STD for different combination of w and l

As we can see in the upper table, there are not huge differences between the $\{l, w\}$ combinations. However, we choose the $\{l, w\} = \{3, 32\}$ combination as it slightly outperforms the other combinations and has the lowest variance among them.

D. Transfer learning and parameter identification

Modifying our network for parameter identification is straightforward. In our Pytorch model, we add an internal representation of the spring stiffness, k_{NN}, and set it as a differentiable parameter (Figure 2a). Conveniently, k_{NN} also appears in the PINN loss and can thus be updated via backpropagation. When trained on the initial data set S_1, we initialize k_{NN} with the true value of $k^* = 50\ \mathrm{N\,m^{-1}}$ and disable gradient updates during training. Then, to test the model's parameter inference ability, we prepare a new data set, S_3, whose true stiffness $k^* = 70\ \mathrm{N\,m^{-1}}$ is unknown to the model. To perform transfer learning, we unfreeze k_{NN}, re-train the S_1-trained model on S_3, and observe whether k_{NN} approaches the correct k^*.

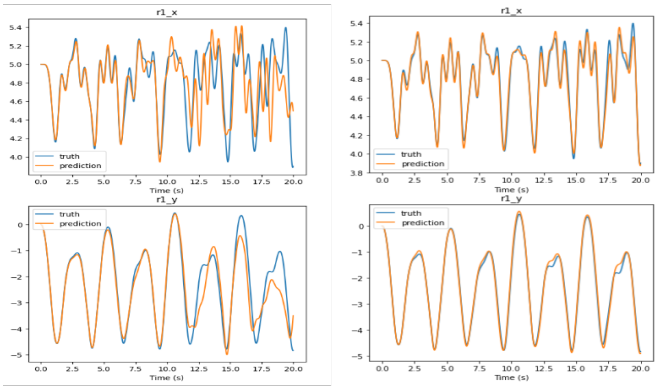

Fig. 3: Predicted trajectories of models trained (a) without PINN loss, (b) with PINN loss

IV. Results and Discussion

A. Effect of PINN loss

To investigate the effect of PINN loss has on the networks, we trained models with and without PINN loss and the corresponding learning curves is shown in Figure 4. Even though the model trained without PINN loss in the objective function, PINN loss decreased over time while only optimizing for data loss. However, the resultant model without PINN loss was not able to recover k value through our previously mentioned formula. We then compared the predicted trajectories for these models Figure 3 where we show the predicted x-axis and y-axis positions of the first masses in these two cases. Without PINN loss, models only has access to forces of the first 5s which are very limited for approximating the target function, so they are prone to overfitting for the unseen time interval. As we can see in the figure, the model still managed to fit the trajectories in the beginning but the trajectories started to deviate from the ground truth as t increased since the error built up over time. On the other hand, with PINN loss added to the loss function, we can clearly observe that the obtained trajectories with the model are much closer to the ground truth trajectories. Therefore, the PINN loss does help networks generalize to points beyond interval where forces are given by enforcing physical constraints.

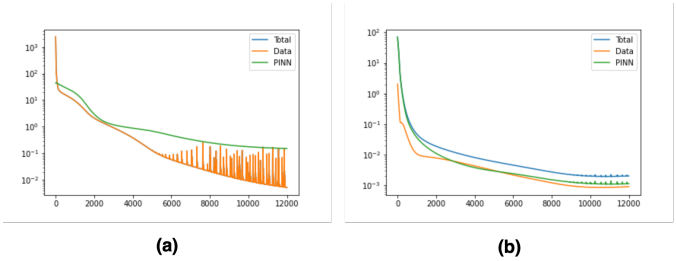

Fig. 4: Learning curves of models trained (a) without PINN loss, (b) with PINN loss

B. Model Result

In this section, we will present and discuss the results of the model trained using our testing set S_2. First of all, we

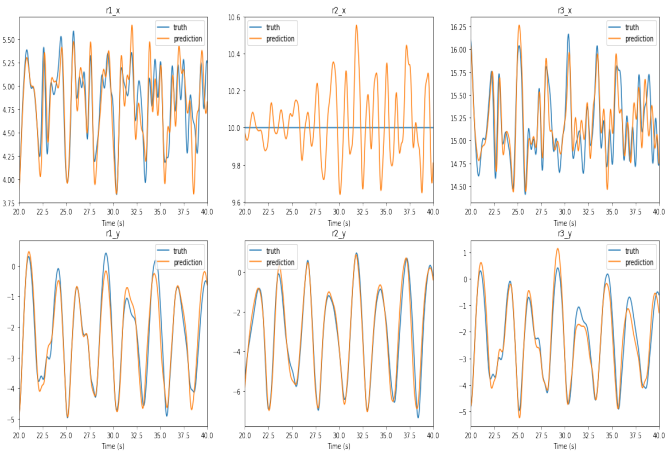

Fig. 5: Predicted Trajectories over validation interval of S_1

examine the trajectories of the resultant model over validation interval $20 \leq t \leq 40$s of S_1 Figure 5. The predicted trajectories of most masses followed the ground truth, except for the third mass which was suppose to be fixed in the system. However, it's important to note that the fluctuation of predicted x-axis position of the third mass is in fact quite low. Finally, we tested our model with the S_2 dataset to see if the model truly learned the dynamics of spring-mass system. The corresponding predicted trajectories are shown in Figure 6. We suspect reason the predicted trajectories didn't work very well could be due to the fact that trajectories of system with 8 masses become more complex and we only use a relatively simple model architecture, and this can be solved by increasing the capacity of the networks.

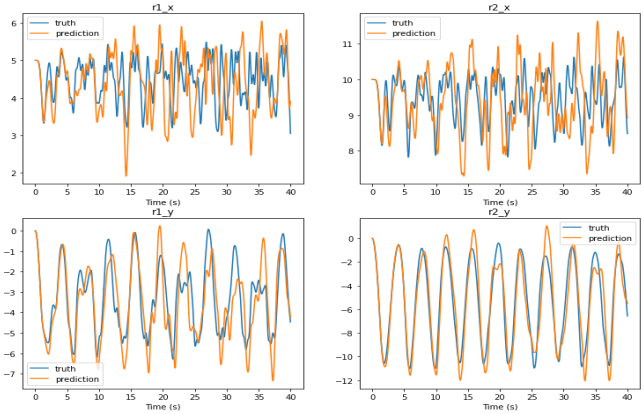

Fig. 6: Predicted Trajectories of 1^{st} and 2^{nd} masses with S_2

C. Parameter identification

As discussed in section III-D, we re-use our trained PINN model for the transfer learning experiment. Figure 7 shows the resulting loss-epoch and k_{NN}-epoch plots; since k_{NN} gradually adapts to $k^* = 70$ during training, it is clear that some degree of parameter inference has occurred. However, convergence is slow and plateaus after 30,000 training epochs, which is comparable to the original model's training duration. Furthermore, we obtained a final value of $k_{NN} = 66.314$, which is still 5% off from k^*.

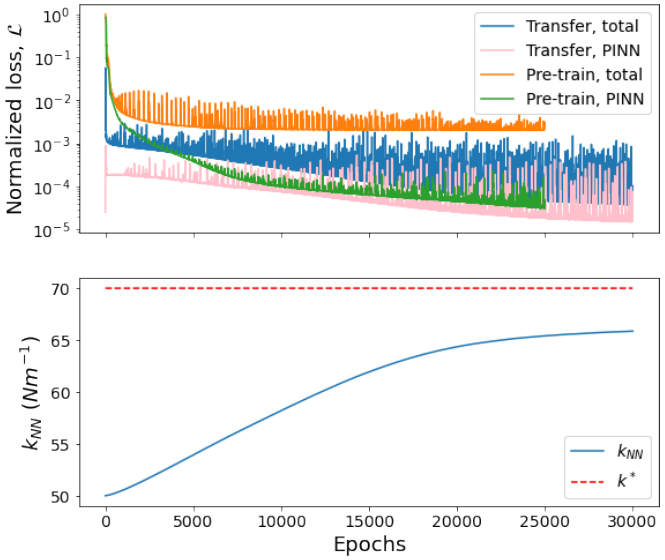

Fig. 7: Learning curves and k_{NN} adaptation during the transfer learning experiment

The long re-training time implies that transfer learning did not occur: the model failed to utilize its prior, pre-existing knowledge and had to re-learn the system from scratch. This result is unsurprising on the one hand, as we do not take any special measures to ensure the model is amenable to transfer learning. We did test the common tactic of freezing all layers but the output layer, but the results were significantly poorer. On the other hand, Haghighat et al. used a similar methodology and were able to perform very efficient and accurate parameter identification [2]. At the same time, our two approaches differ greatly in the problem being solved (PDEs vs. ODEs) and how PINNs are introduced into the problem (forces vs. direct trajectory prediction). Overall, more work is needed to realize parameter identification using our PINN models.

V. Conclusion

In this work, a customized PINN architecture was developed for trajectory simulation of spring-mass systems. Owing to use of physics-informed loss terms, it is able to predict accurate trajectories from little training data, with models trained on as little as 5s of trajectories out of a 40s data set. However, adapting the network to perform parameter identification is still an open problem, with many possible extensions. If stiffness is the parameter of interest, frequency domain or eigenvalue problem-based models may offer more natural methods for determining k. Even with our existing model, however, more careful investigation of transfer learning techniques and architectures may yield better results. It is clear that PINNs are a young and exciting field with many more possibilities still left to explore.

REFERENCES

[1] M. Raissi, P. Perdikaris, and G. Karniadakis, "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations," *Journal of Computational Physics*, 2018.

[2] E. Haghighat, M. Raissi, A. Moure, H. Gomez, and R. James, "A physics-informed deep learning framework for inversion and surrogate modeling in solid mechanics," *Computer methods in applied mechanics and engineering*, 2021.

[3] L. Lu, X. Meng, Z. Mao, and G. E. Karniadakis, "Deepxde: a deep learning library for solving differential equations," *SIAM review*, 2021.

[4] G. E. Karniadakis, I. G. Kevrekidis, L. Lu, P. Perdikaris, S. Wang, and L. Yang, "Physics-informed machine learning," *Nature reviews physics*, 2021.

[5] P. Jin, Z. Zhang, A. Zhu, Y. Tang, and G. E. Karniadakis, "Sympnets: Intrinsic structure-preserving symplectic networks for identifying hamiltonian systems," *Neural Networks*, 2020.

[6] S. L. Brunton, J. L. Proctor, and J. N. Kutz, "Discovering governing equations from data by sparse identification of nonlinear dynamical systems," *Proc. Natl Acad. Sci. USA 113*, 2016.

APPENDIX

A. Spring force direction constraints

Let's look at the analytical form of the spring force, where $\boldsymbol{r}$ is the mass of interest – the one on which the force is exerted – and $\boldsymbol{r}'$ is the mass at the other end of the spring:

$$\boldsymbol{s} = \boldsymbol{r}' - \boldsymbol{r}$$

$$\boldsymbol{F} = k(\|\boldsymbol{s}\| - L)\hat{\boldsymbol{s}} = k\boldsymbol{s} - kL\hat{\boldsymbol{s}}$$

The gradient wrt $\boldsymbol{r}'$ is:

$$\nabla_{\boldsymbol{r}'}\boldsymbol{F} = k\mathbb{I} - kL\,\nabla_{\boldsymbol{r}'}\left[\frac{\boldsymbol{r}' - \boldsymbol{r}}{\|\boldsymbol{r}' - \boldsymbol{r}\|}\right]$$

$$\nabla_{\boldsymbol{r}'}\left[\frac{\boldsymbol{r}'}{\|\boldsymbol{r}'\|}\right] = \frac{1}{\|\boldsymbol{r}'\|^3}\begin{bmatrix} y^2 & -xy \\ -xy & x^2 \end{bmatrix}$$

$$\therefore \nabla_{\boldsymbol{r}'}\boldsymbol{F} = \frac{k}{\|\boldsymbol{s}\|^2}\left(\begin{bmatrix} \|\boldsymbol{s}\|^2 & 0 \\ 0 & \|\boldsymbol{s}\|^2 \end{bmatrix} - \frac{L}{\|\boldsymbol{s}\|}\begin{bmatrix} \Delta y^2 & -\Delta x\Delta y \\ -\Delta x\Delta y & \Delta x^2 \end{bmatrix}\right)$$

With some algebra, we can identify the eigenvector-value pairs of this Jacobian as:

1. Input vectors along s:

$$(\nabla\boldsymbol{F}) * \boldsymbol{s} = k\boldsymbol{s}$$

$$\boldsymbol{v} = \boldsymbol{s}, \qquad \lambda = k$$

2. Input vectors $s^{\perp} = [s_y, -s_x]^T$:

$$(\nabla\boldsymbol{F}) * \boldsymbol{s}^{\perp} = k\left(1 - \frac{L}{\|\boldsymbol{s}\|}\right)\boldsymbol{s}^{\perp}$$

$$\boldsymbol{v} = \boldsymbol{s}^{\perp}, \qquad \lambda = k\left(1 - \frac{L}{\|\boldsymbol{s}\|}\right)$$

Finally – and here is where some uncertainty creeps in – we can try to force $\nabla\boldsymbol{F}$ to have this eigen-decomposition by applying the following constraints:

3. $\boldsymbol{s} \cdot (\nabla F * \boldsymbol{s}) = \boldsymbol{s} \cdot k\boldsymbol{s} = k$
4. $\boldsymbol{s} \cdot (\nabla F * s^{\perp}) = k\left(1 - \frac{L}{\|\boldsymbol{s}\|}\right)(\boldsymbol{s} \cdot \boldsymbol{s}^{\perp}) = 0.$

If points [1-2] hold, then [3-4] must be true; but does the converse apply?

B. Energy conservation constraint

It's difficult to apply an energy conservation constraint directly with our chosen network architecture for two reasons:

1) We operate on positions and forces; velocity information is missing, and the energies are "one derivative up" from the forces ($\boldsymbol{F} = -\nabla\phi$)
2) Our "three-at-a-time" method of parsing the masses is strictly 'local', and we'd have to aggregate the networks' outputs over the full mass chain to get global energy information

This begs the question: can we impose energy conservation locally, and using the forces only? It turns out we can, using our knowledge that all the forces at play (springs, gravity) are **conservative forces**. By definition, conservative forces are force fields which are expressible as the gradient of a scalar potential ($\boldsymbol{F} = \nabla\phi$). As a consequence, $\boldsymbol{F}$ necessarily has the property that it is also **irrotational**[1], i.e. $\nabla \times \boldsymbol{F} = \boldsymbol{0}$. To demonstrate, consider the sum of forces acting on the center mass:

$$\boldsymbol{F} = k(\boldsymbol{r_A} - \boldsymbol{r}) + k_B(\boldsymbol{r_B} - \boldsymbol{r}) + m\boldsymbol{g}$$

$$\nabla_{\boldsymbol{r}} \times \boldsymbol{F} = \begin{bmatrix} \partial_x \\ \partial_y \\ \partial_z \end{bmatrix} \times \begin{bmatrix} k_A(x_A - x) + k_B(x_B - x) \\ k_A(y_A - y) + k_B(y_B - y) - mg \\ 0 \end{bmatrix}$$

$$= \begin{bmatrix} 0 \\ 0 \\ \partial_x[k_A(y_A - y) + k_B(y_B - y) - mg] - \partial_y[k_A(x_A - x) + k_B(x_B - x)] \end{bmatrix}$$

$$= \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

We therefore have a new constraint! $\nabla_{\boldsymbol{r}} \times \boldsymbol{F} = \boldsymbol{0}$ for conservative vector fields, or $\partial_x F_y - \partial_y F_x = 0$. Note that this is the same thing as saying the force Jacobian, $\nabla_{\boldsymbol{r}}\boldsymbol{F}$, should be a symmetric matrix[2]. The constraint is thus:

$$\nabla \times \boldsymbol{F} = 0 \Rightarrow \partial_{r_x}F_y - \partial_{r_y}F_x = 0, \quad \text{or equivalently,} \quad \|(\nabla_{\boldsymbol{r}}\boldsymbol{F})^T - \nabla_{\boldsymbol{r}}\boldsymbol{F}\|_f = 0.$$

On the other hand, it's not clear that we can use the divergence information (diagonal entries of $\nabla\boldsymbol{F}$) so easily. The gravity field $m\boldsymbol{g}$ is divergence-free because gravity is an inverse-square force and subject to Gauss' law + divergence theorem. OTOH, spring forces are *linear*, and their divergence is instead a constant, uniform value everywhere. Using $\boldsymbol{F} = k\boldsymbol{r}$,

$$\nabla \cdot \boldsymbol{F} = \nabla \cdot k\boldsymbol{r} = \begin{bmatrix} \partial_x \\ \partial_y \\ \partial_z \end{bmatrix} \cdot k\begin{bmatrix} x \\ y \\ z \end{bmatrix} = 3k \text{ (in 3D)}$$

[1] The curl of the gradient of any scalar field is always zero: $\nabla \times \nabla\phi \equiv \boldsymbol{0}$
[2] To be precise, the Jacobian wrt the center mass's position, so $\nabla_{r_0}F$.

$$\oint_V 3k\,dV = \oint_S \boldsymbol{F} \cdot \hat{\boldsymbol{n}}\,dS \quad \text{(divergence theorem)}$$

For a spherical volume,

$$3k\oint_V dV = \|\boldsymbol{F}\|\oint_S dS$$

$$\frac{4}{3}\pi r^3 * 3k = 4\pi r^2\|\boldsymbol{F}\|$$

$$\Rightarrow \|\boldsymbol{F}\| = kr \text{ (as expected)}$$

If we know that the divergence of the spring forces is a constant, as shown below:

$$\nabla \cdot \boldsymbol{F} = \begin{bmatrix} \partial_x \\ \partial_y \\ \partial_z \end{bmatrix} \cdot \begin{bmatrix} k_A(x_A - x) + k_B(x_B - x) \\ k_A(y_A - y) + k_B(y_B - y) - mg \\ 0 \end{bmatrix}$$

$$= (-k_A - k_B) + (-k_A - k_B)$$

$$= \text{const.}$$

Then we could possibly stipulate that the *second* derivatives of F_x, F_y are zero. This simply says that the spring forces must be linear in $\boldsymbol{r}$, i.e. $F = kr^n, n > 1$ is not a possible solution for the force law. However, this constraint isn't generalizable to other types of forces, unlike the previous curl constraint.