A COMPARATIVE STUDY FOR MEMBRANE SIMULATION TECHNIQUES WITH PINNS

Giulio Fedeli, Lorenzo Cirone, Francesco Lucchese, Giacomo Garufi
Sapienza University of Rome
{fedeli.1873677, cirone.1930811, lucchese.1559785, garufi.1750327}@studenti.uniroma1.it

ABSTRACT

Simulating the behavior of deformable materials is crucial in many fields, from guiding surgical tools during operations to helping mechanical engineers during the design stage.
For a long time, the go-to method for solving the complex equations behind these simulations has been the Finite Element Method, but recent advances in deep learning offer a new way to tackle physics-based problems.

In this report, we propose a solution to simulate these type of materials using a deep learning approach, where a neural network is trained to learn the non-linear relationship between boundary conditions and the resulting displacement field. In addition we propose a comparison between our solution and some gui-based existing alternatives.

The desired result is to be able to simulate real time interaction of soft materials with virtual robotic objects and possibly including the use of haptic interfaces for the users' interaction.

The ultimate goal is to propose a tentative common computational protocol in order to spark the conversation around its necessity and utility allowing for better compatibility for any future development of related work.

1 Introduction

In many scientific fields simulations are at the core of the practice and the advancement of the work related to them. Concerning medical applications of engineering solutions this is even more crucial, this is why we tasked ourselves with the search for an efficient and effective method that could serve the purpose of simulating the interaction with human membranes. These can range from our skin to the walls of our blood vessels to the cellular membranes, each with their specific mechanical properties.

A tool that could simulate such a vast range of membranes and allow for live interaction during a 3D simulation environment is yet to be readily available. Its main characteristics would need to include the ability to perform kinematics analysis for the inclusion interactive inputs of simulated manipulators (e.g. human hands, surgical tools, surgical robots), the inclusion of a physics simulation engine to calculate the material's response and finally the ability to change a wide range of parameters of the membranes in order to customise them to the user's needs.

The most notable framework for soft material simulation in a 3D interactive environment is represented by the software Simulation Open Framework Architecture (SOFA). This is a C++ library that allows for the decomposition of complex simulators into components designed independently and organized in a scenegraph data structure. Its main advantage is the efficiency in computing the physics underlying its simulation, yet at a great cost of precision and accuracy of response.

The state of the art in materials simulations and more specifically membranes is represented by numerical methods such as the Finite Elements Method (FEM). These are vastly used in science to solve partial differential equations (PDE) on complex domains, for which analytical solutions are not possible. PDEs are the foundation upon which any model of a mechanically responsive material can be produced. Computing the non-linear deformation of mechanical structures is one of the fields which deals with such equations and uses FEM to approximate the solution. The main benefits of the FEM are its accuracy and well-grounded mathematical foundations.

However when the problem complexity increases, the combination of high resolution meshes and non-linear constitutive laws usually leads to computation times that are too high for certain applications and the off the shelf programs don't allow for live interactive simulations but instead they limit themselves to the calculations of the material deformation. In order to obtain the expected results we had to find a method to transpose the problem to an interactive environment like SOFA without losing the precision of the numerical simulations granted by FEM.

For this reason we decided to look into the combination of an existing such environment while studying a solution on how to transpose the numerical problem. Our proposed solution therefore relies on a new field that shows promising results: Deep-learning. Deep-learning is an area of machine learning that has demonstrated a strong ability at extracting high-level representations of the relation between a given input and its corresponding output as opposed to task specific algorithms. In other words, given enough inputs, such techniques can approximate the relation to the corresponding outputs without any prior knowledge.

Our main objective here is to leverage the advantages of deep learning methods (in particular the ability to learn complex relations between inputs and outputs of a model) and the solid scientific foundations of the FEM to obtain solutions of nonlinear elasticity problems. Since multiple FEM strategies have been proposed over the last decades, we have compared our deep learning approach with some Softwares that incorporates these techniques and discusses their limits compared to our expectations.

2 Traditional and Alternative Methods

The main problems when simulating skin concerns computation time and accuracy level, these parameters cannot be improved at the same time; improving one brings to the worsening of the other. In the medical simulation field it is difficult to choose which score is preferred since they are both vital for a good human-like simulation.
Diverse methods have been developed to tackle this problem: Finite Element Method guarantees a high accuracy level that is due to a high computational cost which necessarily has the drawback of extremely high simulating times.
Some variants of this method have also been proposed by [1] to try and close this gap.

The **precomputation-based** FEM is the most popular variation of FEM. This approach leverages the relationship between mechanical forces and deformations, which are precomputed using accurate FEM models that incorporate full physical and biomechanical characteristics.
These precomputed results are used to train an approximate model. To create this model, a database of accurate FEM simulation outcomes is constructed beforehand. The accuracy and speed of the simulated model are influenced by the types of approximation techniques used, including linear/nonlinear regression functions and machine learning methods.

The formulation-adapted FEM involves mathematically modifying FEM formulations with other modeling approaches. One such approach is the **linearized FEM** (L-FEM), which linearizes the kinematic behavior of the object being simulated to the first order of approximation over a specific time period. This results in a simplified and faster-executing FEM model derived from the reduced kinematics. However, due to this simplification, L-FEM is only suitable for modeling soft tissues with linear elastic properties.

Another method, known as **matrix system reduction** FEM (MSR-FEM), focuses on computing only the regions of interest rather than the entire model [1]. The **order reduction method** (ORM) was designed to reduce the computational complexity of nonlinear FEM for real-time simulations. Additionally, the **Total Lagrangian** (TL) formulation was applied to a finite element model to enhance computation speed. A variant of this method, the **Total Lagrangian explicit dynamic** FEM (TLED-FEM), was developed for use in image-guided surgery applications and to simulate human tissue deformations.
These methods successfully improved the size and computation speed of the models.

Lastly, the **element-by-element preconditioned conjugate gradient** FEM (EbE PCG-FEM) was developed, combining FEM with a conjugate gradient method. This approach alternates mesh topological computations during runtime iterations, resulting in a model that is faster and requires less memory than the original FEM.

FEM methods rely on generating 3D meshes, which entail high computational costs. To address this, alternative methods that do not require meshing have been proposed [2]:

- **Meshfree-Based Modeling Methods**

  Unlike mesh-based modeling methods, meshfree-based methods use discrete points to represent the continuum and leverage interpolation methods to solve partial differential equations (PDEs). This approach eliminates the need to preprocess all cell elements to estimate global deformations, making meshfree-based techniques much faster and capable of simulating large deformations in real time.

One popular meshfree-based method is the **mass-spring system modeling** (MSM), which has been used to model muscle deformations in real time [3].
Another notable method is the **mass tensor method** (MTM), which approximates the modeled object into a tetrahedral mesh.

- **Hybrid Modeling Methods**

  Hybrid methods have been extensively studied for their ability to combine the strengths of multiple approaches. For instance, while the MTM is fast and suitable for real-time simulation of soft-tissue deformation, it lacks realistic biomechanical characteristics, particularly for nonlinear materials. Conversely, FEM offers realistic simulations of biomechanical behaviors but comes with high computational costs. Combining MTM, FEM, and pre-computed FEM we can simulate deformations in real time and handle realistic cutting and tearing of nonlinear materials.

  Allard et al. introduced the **SOFA framework** [4], a well-known modular and flexible tool for biomedical researchers to develop new soft-tissue deformation models. The framework effectively combines multiple modeling methods to meet various real-time constraint requirements for simulating soft tissues.

3 A Brief Overview of Our Benchmark: SOFA

SOFA (Simulation Open Framework Architecture) is an open-source C++ library tailored for interactive computational medical simulations. It disassembles complex simulators into independently designed components organized within a scenegraph data structure, with each component encapsulating a distinct aspect of the simulation, such as degrees of freedom, forces, constraints, differential equations, main loop algorithms, linear solvers, collision detection algorithms, and interaction devices. [4]

Interactive physical simulations of both rigid and deformable objects require expertise in geometric modeling, computational mechanics, numerical analysis, collision detection, rendering, user interfaces, and haptic feedback. SOFA addresses these needs through a modular and efficient framework, enabling researchers and developers to concentrate on their specialties while leveraging contributions from others.

Simulated objects in SOFA are broken into independent components that describe various features, such as state vectors, mass, forces, constraints, topology, integration schemes, and solving processes. Beyond this modularity, objects can be further decomposed into specialized models optimized for specific computations. Typically, a physical object in SOFA has three models: an internal model with degrees of freedom, mass, and constitutive laws; a collision model with contact geometry; and a visual model with detailed geometry and rendering parameters. [4]

Handling collision detection efficiently is crucial, as it can become a bottleneck when many primitives are in contact. SOFA implements several approaches, including calculating distances between geometric primitives, points in distance fields, colliding meshes using ray-tracing, and intersection volumes using images. A separate collision model, with its own topology and geometry, adapts models to different collision algorithms. For example, the TriangleModel component interfaces with collision detection on triangular mesh surfaces.

Mesh data, such as material stiffness and nodal masses, are stored in components spread throughout the simulation tree. These are simple arrays with contiguous memory storage and fast access times, important for real-time simulation, though they require renumbering when elements are removed. The consistency of these arrays with topological changes is automated, with specific structures for vertices, edges, triangles, quads, tetras, and hexas.

Collision detection in SOFA is divided into phases, each in a different component and managed by a CollisionPipeline component. Potentially colliding objects have collision geometry based on independent DOFs. The broad phase identifies pairs of colliding bounding volumes, and the narrow phase determines pairs of geometric primitives and contact points, which are passed to the contact manager to create contact interactions based on customizable rules. [4]

For high computational performance, SOFA supports GPU-based computations. The scene-graph design allows components like ODE and linear solvers to be used on both CPU and GPU models.

The main interest of interactive simulation is that the user can modify the course of the computations in real-time. This is essential for surgical simulation : during a training procedure, when a virtual medical instrument comes into contact with some models of a soft-tissue, instantaneous deformations must be computed. This visual feedback of the contact can be enhanced by haptic rendering so that the surgeon can really feel the contact. [4]

There are two main issues for a platform like SOFA for providing haptics: the first is that haptic forces need to be computed at 1kHz whereas real-time visual feedback (without haptic) is obtained at 30Hz. The second is that haptic feedback could artificially add some energy inside the simulation that creates instabilities, if the control is not passive. SOFA addresses this with two approaches: **Virtual Coupling** and **Constraint-based rendering**.

- **Virtual Coupling**: the coupling of a haptic device is bidirectional: the user applies some motions or some forces on the device and this device, in return, applies forces and/or motions to the user. The majority of the haptic devices propose a Impedance coupling: the position of the device is provided by the API and this API asks for force values from the application. A very simple scheme of coupling, presented in 1, could have been used. In this direct coupling case, the simulation would play the role of a controller in an open loop.

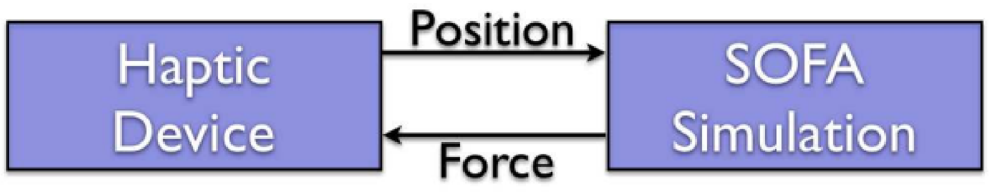

Figure 1: Direct coupling.

- **Constraint-based rendering**: A novel way of dealing with haptic rendering for medical simulation has been proposed in the context of SOFA. The approach deals with the mechanical interactions using appropriate force and/or motion transmission models named compliant mechanisms. These mechanisms are formulated as a constraint-based problem that is solved in two separate threads running at different frequencies. The first thread processes the whole simulation including the soft-tissue deformations, whereas the second one only deals with computer haptics. With this approach, it is possible to describe the specific behavior of various medical devices while relying on a unified method for solving the mechanical interactions between deformable objects and haptic rendering.

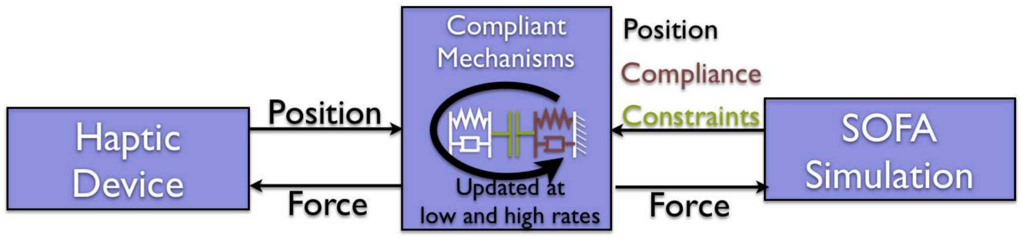

Figure 2: Compliant mechanisms technique. The simulation shares the mechanical compliance of the objects and the constraints between them. The constraint response is being computed at low rate within the simulation and at high rates within a separate haptic thread. A 6-DoF damped spring is still used to couple the position of the device to its position in the simulation.

3.1 Virtual Surgical Tools and Force-Feedback Devices

Virtual surgical tools, in combination with force-feedback devices, significantly enhance the realism and flexibility of surgical simulations by facilitating detailed interactions between the user and the simulated models.
These virtual instruments transfer control signals from external real devices to the simulation model and provide feedback on the biomechanical parameters calculated within the simulation back to the haptic devices. High-speed data transmission between the simulated models and the devices is crucial to ensure realistic visualization and haptic feedback. [5]

Force-feedback devices act as input/output interfaces, enabling users to interact with virtual surgical tools. When interactions are detected by the virtual tool, the simulation model responds by calculating haptic forces in real-time during each simulation iteration. These computed forces are then relayed back to the device, allowing users to experience tactile sensations that mimic interactions with real soft tissues.

In practical applications, a 7-degree-of-freedom (DOF) haptic device can be used in conjunction with a surgical tool fixed at its end, providing a highly realistic simulation environment. For example, the commercial PHANTOM haptic device [6] offers 3 DOF force feedback and 6 DOF position and orientation capabilities, making it suitable for haptic surgery simulations. This device allows surgeons to feel realistic force feedback and positional information during simulated procedures.

Moreover, the SensAble PHANTOM Desktop haptic device [6] has been developed with a pen-sized handle connected to a robot arm with flexible, engine-forced joints to simulate a virtual surgical scalpel. This setup provides precise and realistic feedback for delicate surgical tasks. Similarly, a virtual laparoscopic grasper controlled by a Xitact IHP haptic device has been utilized to simulate laparoscopic procedures. [5]

Overall, these advancements in virtual surgical tools and haptic feedback devices create highly immersive and realistic training environments for surgeons, improving their skills and preparedness for real-life surgical procedures.

4 Experimenting with FEM

For our study we decided to focus on two of the better known softwares for FEM analysis. These are COMSOL Multiphysics and Ansys, they are usually implemented in mechanical and material engineering problems and represent a great baseline for the behaviour we want to reproduce.

For both simulations we performed the test on a squared section of skin tissue of 25 cm^2 to which a force of absolute value 1N was applied. For a more detailed description of the parameters used in our simulations please refer to the table below 4. The sides of the squares are fixed boundaries and the force is applied on a single spot, more precisely at the center of the square.

Skin Density	1109 kg/m^3
Young's Modulus	1,5 MPa
Poisson's Ratio	0,45
Membrane Area	25 cm^2
Skin Thickness	3 mm
Force	1 N

After the application of the force an elastic response is expected, resulting in an oscillating membrane.

4.1 Comsol Multiphysics

COMSOL Multiphysics [7] is a software program used for simulating designs, devices and processes in various engineering and scientific fields.

Some key features:

- **Multiphysics and single-physics**: You can model situations with multiple interacting physical effects or focus on a single type of physics.
- **Complete workflow**: The software provides a unified environment for every step of the simulation process, from defining the geometry to analyzing the results.
- **Physics-based approach**: It uses real-world physics principles to build the simulations.

COMSOL achieves this through a combination of features:

- **Finite element method**: This is a numerical technique for solving complex problems by dividing them into smaller pieces.
- **Partial differential equations (PDEs)**: COMSOL allows you to define the relevant PDEs for your simulation.
- **Add-on modules**: These provide specialized functionalities for specific areas like electromagnetics, fluid dynamics, or heat transfer.

A key feature that distinguishes COMSOL is the possibility to specify the material as skin tissue on top of already indicating the mechanical features of the material when setting up the simulation. For further reference please check appendix A.1.

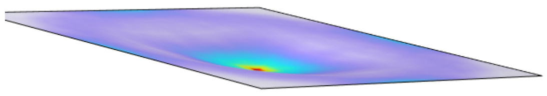

Figure 3: Membrane on Comsol Multiphysics

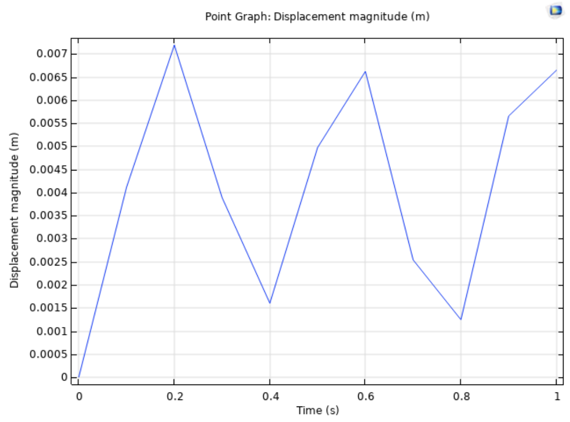

Figure 4: Evolution of the Central Point

4.2 Ansys

Ansys [8] is a software suite used for engineering simulation. It's a powerful toolset used to virtually test and analyze the behavior of structures, materials, fluids, and other engineering systems before building a physical prototype.

There are several key areas where Ansys shines:

- **Finite Element Analysis (FEA)**: This is a numerical technique for solving complex engineering problems by dividing them into smaller, more manageable pieces. Ansys offers a variety of FEA tools for structural analysis, thermal analysis, fluid dynamics, and more.
- **Multiphysics Simulation**: Ansys can handle situations where multiple physical phenomena (like stress, heat, and fluid flow) occur simultaneously, providing a more comprehensive picture of a system's behavior.
- **Optimization**: Ansys can be used to optimize designs by iteratively simulating different configurations and identifying the one that meets performance goals with the least weight, material usage, or other factors.

Ansys is made of several moduli and in particular we used Ansys Mechanical which is a popular FEA tool for analyzing the structural behavior of components under various loads. It's useful for tasks like predicting stress, strain, and deformation in bridges, buildings, and machine parts. Given the great freedom of parameters settings this was greatly suitable for our application on human skin.

For further reference please check appendix A.2.

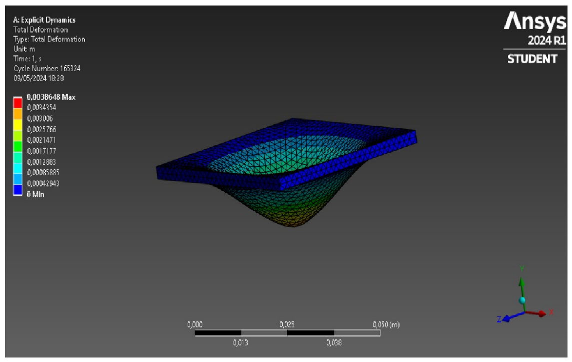

Figure 5: Deformed Membrane on Ansys Mechanical

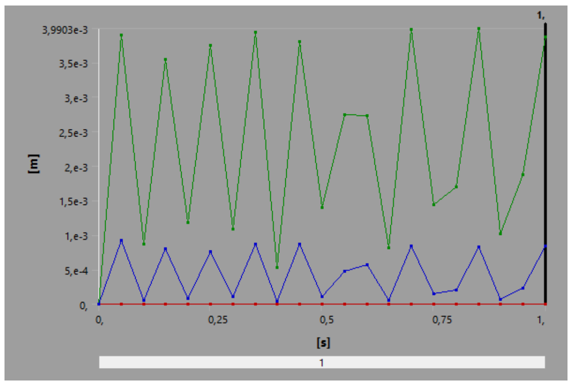

Figure 6: Max, min and average value of the overall displacement

4.3 Comsol vs Ansys

The overall behaviour of the two tests is comparable and somewhat consistent. Both programs show an isomorphic behaviour, consistent with a force applied to a single point and then released. This triggers an oscillatory response also consistent with what was expected. Nevertheless some fundamental issues arise when comparing the two.
Even if both simulations were run with the exact same parameters there are some differences in the obtained results.

The maximum displacement obtained in Ansys is almost half of the one obtained in COMSOL. Moreover the oscillations produced by the applied force differ in frequency and in amplitude. For further comparison check the table below4.3. The result of such comparison is that the risk of mixed outcomes and unreliable simulations calls for a proper model of human membranes and more specifically a model that lets the user set the parameters of the physics behind it as freely as possible, directly having access to the definition of the PDE and their solution engine.

Analysis Comparison	Ansys	Comsol
Maximum Displacement [mm]	3,9	7,0
Maximum Von Mises Stress [MPa]	0,152	0,410

5 Our Solution: Empowering CoppeliaSim with PINNS

In light of what shown in section 2, 3 and 4 we needed a framework that maintains some key features of SOFA such as its interactive nature, its ease of computation, its predisposition for haptic device integration and its ODE solution engine able to run on both CPU and GPU, all the while improving its precision in representation and setting a common framework to ease integration and interoperability. Moreover we would like a more robotic oriented environment given the importance of soft materials in the field of medical robotics.

The obvious choice that met all the desired requirements is CoppeliaSim of which a brief outline can be found in paragraph 5.3. This needs to be paired with a powerful yet light enough computational engine able to achieve the prefixed objective of improvement without loss of flexibility granted by SOFA.

Our attention was brought to the world of Neural Networks and their ability of precisely crunching big amount of data once trained, giving precise enough answers at the expense of little computation cost.

In the next paragraphs we outline our integration process and the theory behind this innovative approach.

5.1 Physics Analysis

Vibrating String Equation String modeled as a sequence of infinitesimal (spheres/elements) of mass m, uniformly distributed along the string, all connected together by flexible segments of infinitesimal length h.
These particles are allowed to move vertically model of the string: *u(x,t)* models the string behavior, with x being the point in which we have a displacement from the equilibrium position and t the time instant.

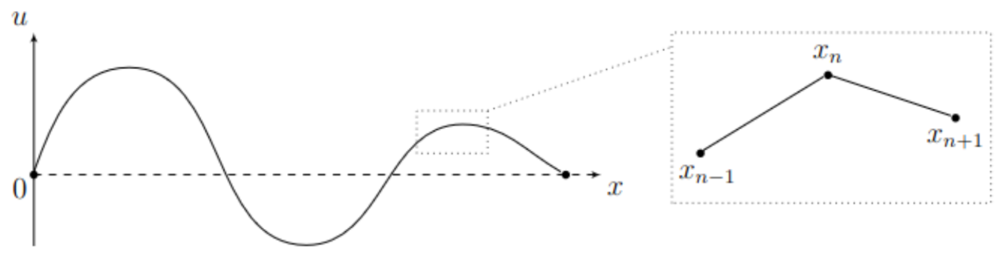

Figure 7: Rope

The PDE for the freely evolving system is as follows:

$$\frac{\partial u(x,t)^2}{\partial t^2} = a^2 \frac{\partial u(x,t)^2}{\partial x^2} \tag{1}$$

where the first element is the acceleration of the string at a certain point x and time t while the second element is the curvature of the string multiplied by the term $a = \sqrt{\frac{T}{p}}$ which describes the wave propagation speed for small amplitude vibrations and is computed as the horizontal tension of the string T (assumed constant for a fully elastic string due to the small angles assumption) divided by the linear density of the string ρ.

The solution of the equation depends on the initial conditions on position and velocity:

$$\begin{gathered} u(x, t=0) = \omega_1 \\ \tfrac{\partial u(x,t=0)}{\partial t} = \omega_2 \end{gathered} \tag{2}$$

In our case boundary conditions on the extremities of the string were also applied:

$$\begin{gathered} u(x, t=0) = 0 \\ u(x=l, t) = 0 \end{gathered} \tag{3}$$

where l denotes the length of the string.

This case can be solved using separation of variables.
By setting:

- $u(x,t) = X(x)T(t)$
- substituting $u(x,t)$ into the PDE, after derivating twice we obtain $X(x)T''(t) = a^2X''(x)T(t)$
- handling it into $\frac{X''(x)}{a^2T(t)} = \frac{X''(x)}{X(x)}$
- the value of the expression being constant means the two terms must necessarily be both equal to a constant value $K \neq 0$:
  $\frac{X''(x)}{X(x)} = \frac{X''(x)}{a^2T(t)} = K^2$

We can then split the previous equation into two ODEs:

$$\begin{gathered} X'' + K^2X = 0 \\ T'' + a^2K^2T^2 = 0 \end{gathered} \tag{4}$$

whose solutions are of the kind $X(x) = A\cos(Kx) + B\sin(Kx)$ and $T(t) = C\cos(\alpha Kt) + D\sin(\alpha Kt)$ and the general solution of the homogeneous equation is:

$u = [A\cos(Kx) + B\sin(Kx)] \cdot [C\cos(\alpha Kt) + D\sin(\alpha Kt)]$.

The coefficients A and B can be calculated by imposing the boundary conditions.

We assumed $u(0,t) = u(l,t) = 0$ being $u(x,t) = X(x)T(t)$, and we obtain $X(0)T(t) = X(l)T(t) = 0$ which leads to:

$X(0) = A\cos(K*0) + B\sin(K*0) = A*1 + B*0 = 0$ and $X(l) = A\cos(Kl) + B\sin(Kl) = 0$

from which necessarily $A = 0$, $B * sin(Kl) = 0$ with $B \neq 0$, which leads to $K = \pm\frac{n\pi}{l}$ with the positive and negative solutions being equivalent.

The complete equation thus becomes:

$$u = [C\sin(\alpha Kt) + D\cos(\alpha Kt)] \cdot \sin Kx \tag{5}$$

which can be rewritten as an infinite sum of sinusoids as every element of the sum is by itself a solution:

$$u = \sum_{n=1}^{\infty} \left[C_n \cos\left(\frac{n\pi\alpha t}{l}\right) + D_n \sin\left(\frac{n\pi\alpha t}{l}\right)\right] \cdot \sin\left(\frac{n\pi x}{l}\right) \tag{6}$$

where the C_n and D_n coefficients can be found by imposing t=0 over the solution:

$u(x, t=0) = \omega_1 = \sum_{n=1}^{\infty} C_n \sin\left(\frac{n\pi x}{l}\right)$

$\frac{\partial u(x,t=0)}{\partial t} = \omega_2 = \sum_{n=1}^{\infty} D_n \sin\left(\frac{n\pi x}{l}\right)$

from which we obtain:

$$C_n = \frac{2}{l}\int_0^l \omega_1 \sin\left(\frac{n\pi z}{l}\right)dz \tag{7}$$

$$D_n = \frac{2}{n\pi\alpha}\int_0^l \omega_2 \sin\left(\frac{n\pi z}{l}\right)dz \tag{8}$$

and the complete solution:

$$u = \sum_{n=1}^{\infty}\left[\left(\frac{2}{l}\int_0^l \omega_1 \sin\left(\frac{n\pi x}{l}\right)dz\right)\cdot\cos\left(\frac{n\pi\alpha t}{l}\right) + \left(\frac{2}{n\pi\alpha}\int_0^l \omega_2 \sin\left(\frac{n\pi z}{l}\right)dz\right)\cdot\sin\left(\frac{n\pi\alpha t}{l}\right)\right]\cdot\sin\left(\frac{n\pi x}{l}\right) \tag{9}$$

5.2 PINNs

We decided to use Python as our framework so first we focused on transcribing the physical problem. Two approaches were explored, the first one was to finetune a pretrained library and the second was to train a Neural Network from scratch. The main goal of this comparison was to verify the cost to benefit ratio in our specific application of the Neural Network.

5.2.1 Solution Engine: PINNs

In order to properly simulate the elastic rope problem an effective physics computational model must be available, more specifically one that includes the solution of PDEs in both closed and non-closed form. The first hurdle to overcome is the choice of the framework on which said simulation should run, one that allows for flexibility in the parameters setup and can be as generalisable as possible. Moreover it should be easily integrable with other applications, in our case CoppeliaSim.

The choice fell onto Physics Informed Neural Networks or better known as PINNs. They integrate the power of neural networks with the physical constraints of partial differential equations. By embedding the governing equations as additional constraints during training, PINNs can learn from limited data while ensuring that the learned solutions adhere to the underlying physics.
PINNs offer advantages in scenarios with sparse or noisy data, complex geometries, and irregular boundary conditions, providing accurate solutions while reducing computational cost compared to traditional numerical methods. Their implementation can be done through C++ or Python, both of which can be easily input in CoppeliaSim thanks to the ZeroMQ plugins. Moreover PINNs have shown promising results in solving equations in non-closed form demonstrating a high degree of generalisation.

This endeavour goes beyond the scope of this work which will limit itself to providing a suitable simulation in a 3D interactive environment for a closed form solution of elastic ropes and membranes. On top of this the chosen framework allows for seamless integration of any further improvements on the PINNs model, corroborating evidence for the case of our study.

5.2.2 DeepXDE

The first implementation of our neural network was done thanks to the DeepXDE library [9].
DeepXDE is a library for scientific machine learning and physics-informed learning.
It includes the following algorithms:

- physics-informed neural network (PINN)
  - solving different problems
  - improving PINN accuracy
- (physics-informed) deep operator network (DeepONet)
- multifidelity neural network (MFNN)

We will only focus on the first application among the above listed as it's the one needed in order to setup the physics equations solver. Here is a brief schematization of its implementation: First the physics equations are transposed in the

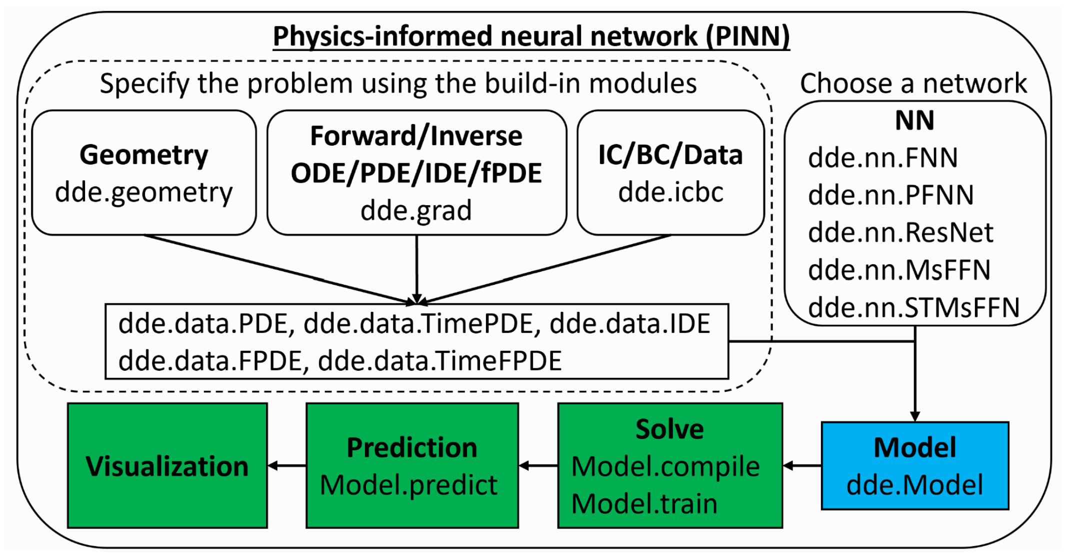

Figure 8: PINNs Overview

Python script, then these are plugged in the definition of the model that will then be fine-tuned to solve them.
A crucial role is played by the choice of the pretrained NN that will structure our model. In our case this is the STMsFNN. Space-Time Multi-Scale Feature Network is a neural network architecture specifically designed for solving partial differential equations (PDEs) involving both spatial and temporal dimensions.

It incorporates space-time awareness by using separate scaling factors for spatial and temporal coordinates. This allows the network to learn features at different scales in both space and time, effectively capturing the interplay between these dimensions in PDEs.
Moreover it leverages multi-scale feature extraction by using multiple fully connected layers with activation functions. It can extract features from the input data at various levels of complexity better representing the underlying physical phenomena governed by the PDE.

The backend on which the library runs can be selected among Tensorflow, Paddle, Pytorch and Jax; in our case Tensorflow was selected. In 4.3 it is shown the parameters configuration.
The model is then trained and saved in order to be later called to solve the equations given to it.

5.2.3 Custom Neural Network

We based our model on the seminal paper "Deep learning models for global coordinate transformations that linearize PDEs" by Craig et al. [10] and published by Cambridge university press
Craig et al. present a novel deep learning framework for tackling nonlinear partial differential equations. Their approach hinges on uncovering a hidden coordinate system where the PDE becomes linear.
This essentially transforms the problem into a more tractable form, enabling efficient analysis and solution strategies.

The key player in this process is a deep autoencoder that is tasked with learning a latent representation of the PDE data, which resides in the hidden coordinate space. This latent space is hypothesised to harbour the linear dynamics of the system, a stark contrast to the nonlinearities present in the original coordinates.
The encoder component of the autoencoder plays a crucial role in achieving linearization. It leverages a residual network architecture. Residual networks excel at learning complex relationships through the use of shortcut connections. In this context, these shortcuts effectively bypass nonlinearities, essentially learning a near-identity transformation. This transformation essentially peels back the layers of complexity, revealing the underlying linear dynamics within the latent space.

The outcome of the encoding process is a Koopman operator matrix (K). This matrix encapsulates the linear relationships between variables in the hidden coordinates. The strength of this approach lies in its data-driven nature. The deep learning model learns the coordinate transformation directly from data, circumventing the need for explicit mathematical manipulations that can be immensely challenging for intricate PDEs. This makes the method particularly appealing for problems where traditional analytical solutions remain elusive. The paper by Craig et al. showcases the efficacy of their method on problems like the Burgers' equation for thermodynamics and the Kuramoto-Sivashinsky equation to model

wave propagation.
In these examples, the deep learning model successfully recovers the known coordinate transformations that linearize the equations, solidifying the validity of the approach. [11, 12]

Before being able to train the network it is necessary to produce the data upon which the network is trained and this can be done by solving randomly generated equations for the problem at hand, in our case the vibrating string.
The inputs given to our system come from a wide range of scenarios and include:

- Square waves
- Sine waves
- White Noise

After discretizing the vibrating string equation we crank the solution generating engine and we obtain a set of .npy files which will constitute the dataset for the network training.
The model architecture is based on a 21 dimensional latent space which corresponds to the number of Koopman eigenfunctions in expansion.

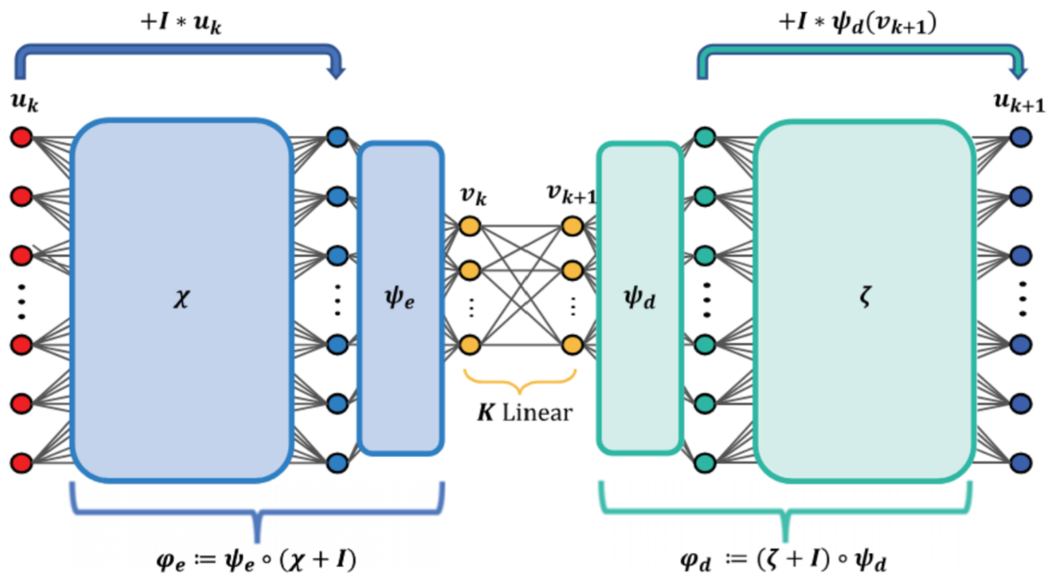

Figure 9: The Network architecture used to find the Koopman eigenfunction. The Network consists of an outer encoder, inner encoder, dynamics matrix K, inner decoder and outer decoder.

The encoder and the decoder are represented by a model composed of 4 hidden layers with ReLu activation function.
The number of inputs to the network is equal to 128.
It should be clear by now that this model represents a big challenge in order to be properly trained and given the machines at our disposal the results were mediocre and obtained at high time cost.
Because of this, we decided to stick to the pretrained library provided by DeepXDE in order to have usable data, even if less task oriented. For future development of our work it would be crucial to being able to test this task and properly compare the performance of the pretrained and the custom networks, given powerful enough machines to run experiments on.

5.2.4 Neural Network Performance Assessment

When training the neural network based on DeepXDE we trained two instances of it in the form of the string case and also the membrane case. In the following section the main parameters and performance of these is briefly summarised. In our final iteration of the application of the DeepXDE based neural network the parameters chosen for the model architecture were as shown in the following tables 5.2.4.

Forced String Model Parameters	
Training Time	1559.670325 s
Num Epochs	20000
Optimizer	Adam
Learning Rate	0.001
Num domain	1440
Num boundary	360
Num Initial	360
Num test	10000
Layer Size	[2] + [100] * 3 + [1]
Num Hidden Layer	3
Activation	tanh

Forced Membrane Model Parameters	
Training Time	2458.752078 s
Num Epochs	20000
Optimizer	Adam
Learning Rate	0.001
Num domain	1440
Num boundary	360
Num Initial	360
Num test	10000
Layer Size	[3] + [100] * 3 + [1]
Num Hidden Layer	3
Activation	tanh

where **num domain** refers to the number of training residual points sampled inside the domain, **num boundaries** is the number of training points sampled on the boundary, **num initial** is the number of training residual points initially sampled, **num test** is the number of points to test the PDE residual. For the **layer size** the syntax is [input] + [hidden units] * [hidden layer] + [output] .

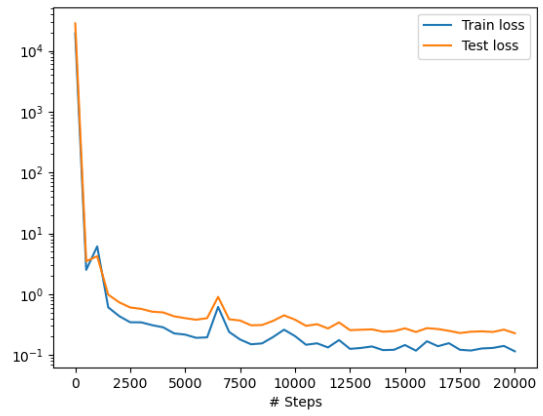

Figure 10: Loss Forced String

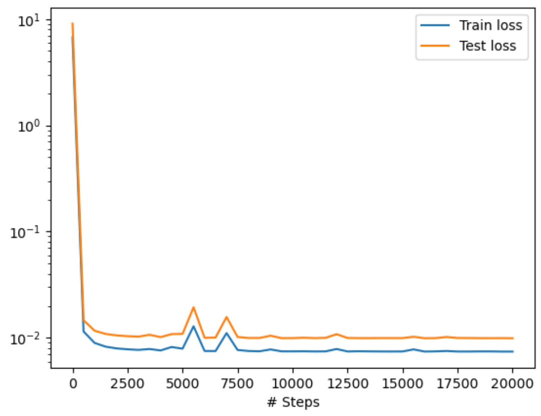

Figure 11: Loss Forced Membrane

The performance of the network training can be seen in the following graphs shown in Figure 10 and 11. We can see the behaviour of the loss function over the training period and its convergence. More specifically two different loss functions are displayed, these are respectively the training loss and the test loss, both of them for both cases of the string and the membrane.

5.2.5 Insight on the String Problem: Closed Form Solution Vs Predicted Solution

A first validation step of the proposed solution is to study the behaviour of the prediction generated by the neural network comparing it to the solution in closed form. This check was performed for the problem of the plucked string. In the following figure 12 we display side by side, from left to right, the predicted solution, then the exact solution and lastly the error. On the x axis we find the position and on the y axis we find time. U(x), i.e. the string's displacement, is shown chromatically.
Notably the predicted and exact solution are visibly similar, this finds confirmation in the right most figure that shows sparse errors of small intensity.

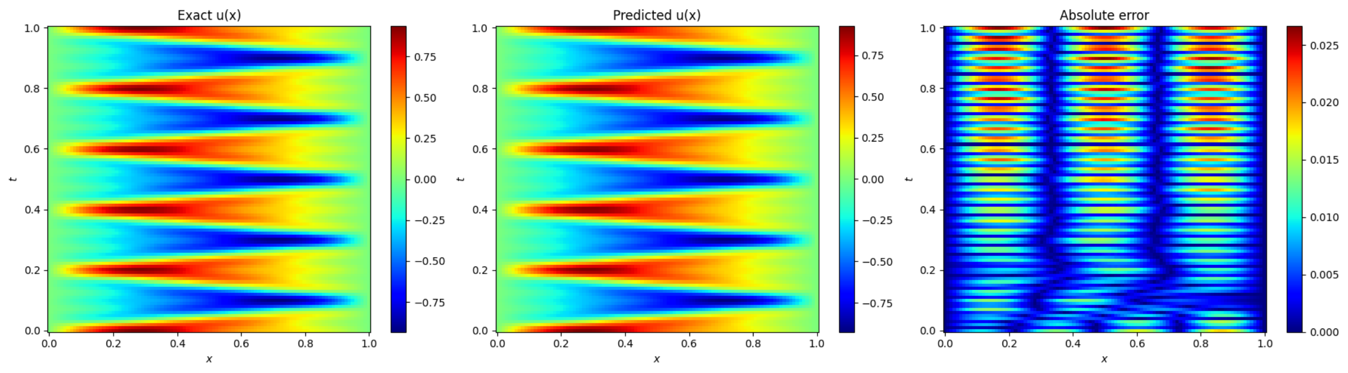

Figure 12: Exact Solution Vs Predicted Solution

5.2.6 Comparison Between FEM and PINNs

In order to give an assessment of comparability between the solutions obtained with FEM and the one obtained by our network we show the map of the maximum displacement across the membrane. We see that in Figure 13 the displacement has a magnitude comparable with the ones obtained with the FEM software. More specifically its value is closest to the result obtained through COMSOL placing it at 8mm of absolute magnitude. It is worth noticing that the distribution of the displacement on the membrane is less regular compared to the one simulated by both FEM applications but this is expected given the nature of the physics used in the PINNs simulation.

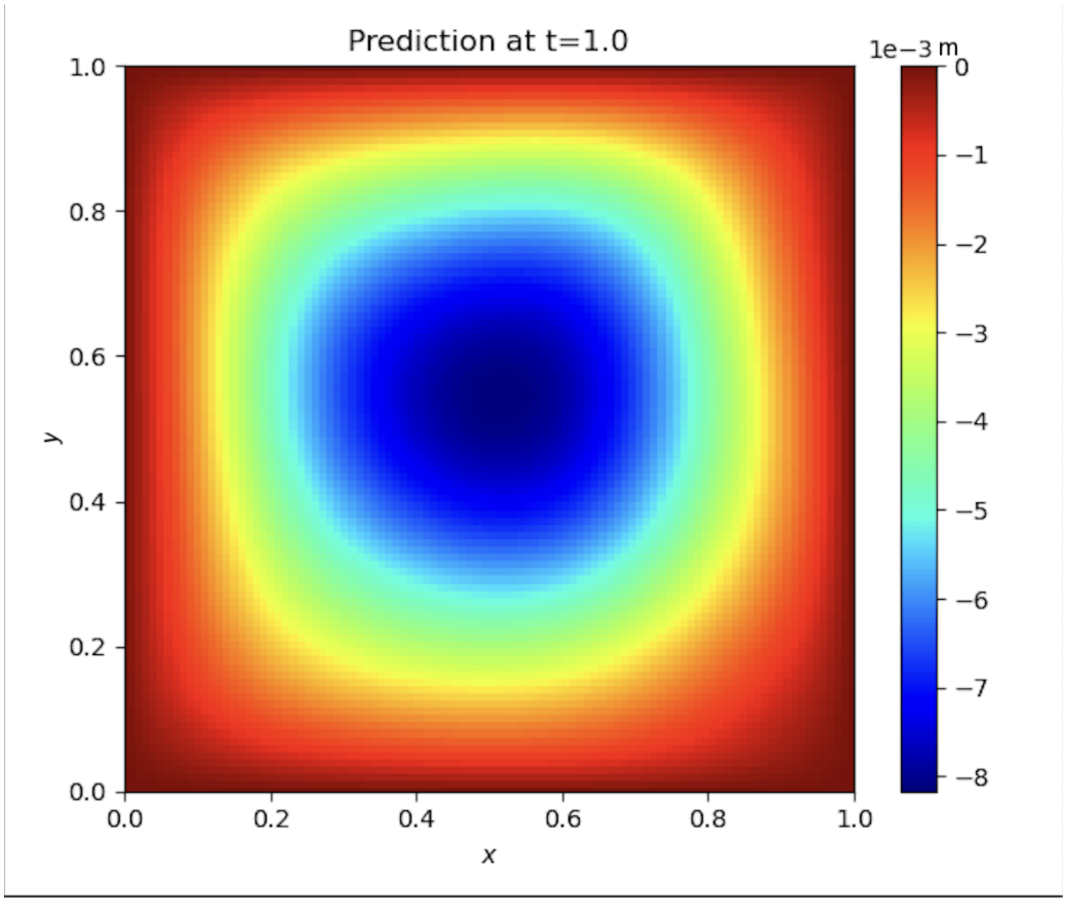

Figure 13: PINNs Membrane Displacement

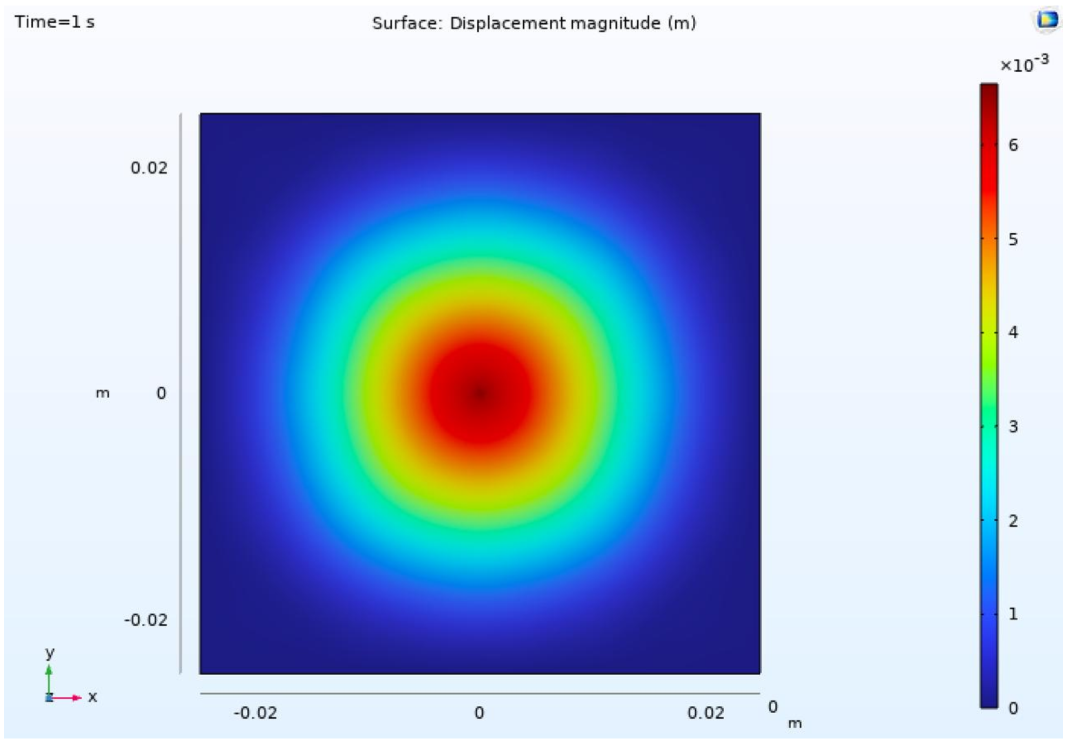

Figure 14: Comsol Membrane Displacement

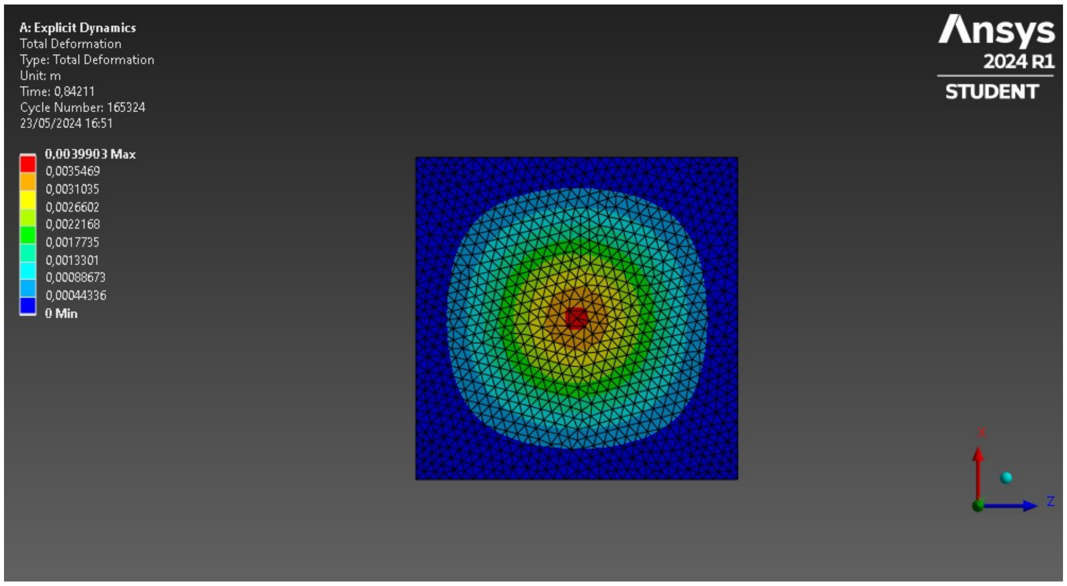

Figure 15: Ansys Membrane Displacement

These results are promising and encourage our hypothesis of applicability of the the proposed solution.

5.3 Integration with CoppeliaSim

CoppeliaSim, formerly known as V-REP, is a robot simulator used in various fields including industry, education, and research. It provides a virtual environment to design, simulate, and test robotic systems before deploying them in the real world.
Its key features are:

- **Comprehensive Environment**: It allows users to create complex robotic models, including various sensors, actuators, and environments.
- **Multiple Physics Engines**: CoppeliaSim utilizes different physics simulation libraries to provide realistic simulations of robot motion and interaction with its surroundings.
- **Control Options**: Robots in CoppeliaSim can be controlled through various methods, including embedded scripts, plugins, external APIs, or a custom user interface.

To integrate our solution (obtained with deepXDE [9]) with CoppeliaSim we use ZeroMQ Remote API that allows to control a simulation (or the simulator itself) from an external application or a remote hardware (e.g. real robot, remote computer, etc.). It offers all API functions also available via a CoppeliaSim script: this includes all regular API functions (i.e. sim.* -type functions), but also all API functions provided by plugins (e.g. simOMPL.*, simUI.*, simIK.*, etc.), if enabled.
Standard continuous elements with mesh features are not suitable in order to properly simulate an elastic response of ropes or membranes.
So clearly a discretization process would be needed to effectively transpose the PINNs output. An acceptable theoretical solution would be to create a series of damped springs connected between each other, this would result in an incredibly heavy computational load resulting in terrible simulation performances, rendering vane any attempt of interaction with the model.

In order to define an object which could be reconducted to a rope we added some spheres with the function sim.createPrimitiveShape(sim.primitiveshape_spheroid) and we set the fixed boundaries on the first and last sphere. The number of spheres which compose the rope represent how much we are discretizing the rope (in terms of finite elements).
After setting up the environment, we load the model of our pre-trained network from a checkpoint and we perform the inference on the position of each sphere on the z-axis.
Since the network is trained on the D'Alambert's Wave Equation in function of time, we obtain a sequence of positions for each sphere of the rope thus we can see the evolution of the rope over time.

The positive result of this process represents the first step in proving the feasibility of our hypothesis allowing us to simulate in CoppeliaSim the behaviour of an elastic rope subject to an external force.
The next step would be to predict the evolution over time of a membrane subject to a known force applied to a known point.

To handle this challenge we first had to adapt the PINNs script to a 2D case, originally tailored only for the mono-dimensional case of the rope. In order to do so we had to redefine the term of the force allowing it to propagate its impulse along more than one dimension, supposing an isotropic behaviour. This would clearly result in a "bell-shaped" distribution along the membrane.

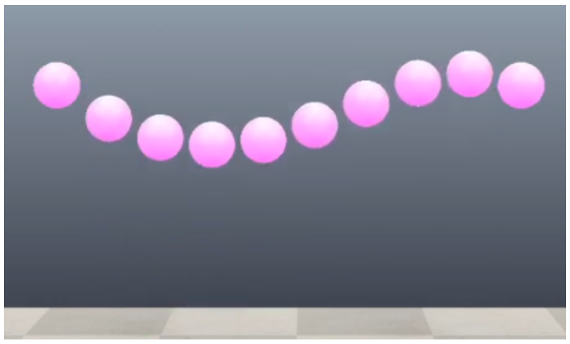
Figure 16: Rope on CoppeliaSim

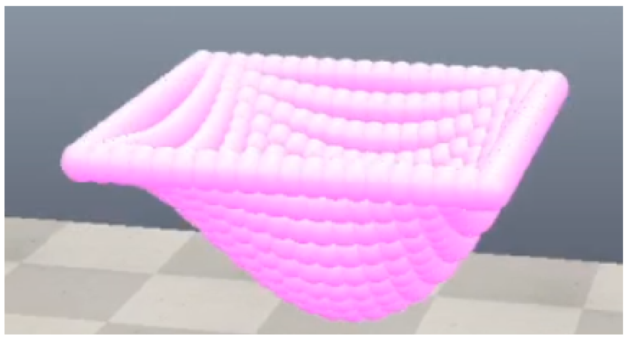
Figure 17: Membrane on CoppeliaSim

6 Conclusion

The goal of this study was to dig into the reality of the tools available for human tissue simulation and to make a case for their inadequacy. First by implementing and testing readily available solutions for FEM simulation we managed to deepen our understanding of the problem and strengthen our conviction that a step forward was critical.

Our work is meant to be a starting point for a new vision in computer simulations leveraging the now easily available and groundbreaking power of machine-learning.
The advantage of such approach is to be able to have direct control over the physics model, down to the core of the equations defining each problem.
On the downside though this comes at the cost of the neural networks limitations, both in time efficiency when it comes to training it and also on the goodness of the results. Though given the current advancements in machine-learning these hurdles are expected to ease greatly over time.

We acknowledge that what we show is purely a starting point and cannot represent a finished product. The next steps to further advance on the trail of our quest are pretty clear in our opinion.
First there's the clear necessity for better exploiting CoppeliaSim for its capabilities since at the moment simply serves as a representation environment. The most notable feature and more specifically the one that pushed us in the direction of using CoppeliaSim is the interactive nature of said simulation environment. The real breakthrough will be when through the simulation the user can interact with the membrane with live response, imitating the real life behaviour of human membranes.
Furthermore we believe that there's value in the custom neural network we ended up discarding because of computational power's constraints, or at least on trying a more custom solution compared to simply using a pretrained model taken from a library.

The field of 3D simulations, more in particular the one with medical application, is an incredibly fascinating field of research and worthy of careful attention for the positive impact it could have on improving our scientific knowledge and finally our overall quality of life.

A FEM Simulations Procedures

A.1 Comsol Multiphysics

To reproduce our simulation on Comsol 6.2:

1. **Define the problem type**: on the home of the program choose model wizard, then 2D, plate and time dependent.
2. **Define the geometry of the problem**: a circle with 1[m] ray centered in (0, 0) with a thickness of 0.01[m] and a point in (0, 0).
3. **Define the constraints**: indicate the whole border of the circle as a fixed constraint.
4. **Apply the force**: define a point load of -1[N] on z-axis applied in the center of the membrane.
5. **Set the membrane material**: we choose skin as material (under the voice Bio-heat materials) and put 1.5e06 as Young's Modulus (which defines the stiffness) and 0.45 as Poisson ratio (which indicates the resistance to the deformation).
6. **Produce the meshes**: COMSOL uses a mesh to discretize the geometry into smaller elements. A finer mesh around the point of force application might be necessary for capturing the details of the deformation.
7. **Simulate**: click on compute under the voice "study" and chose an interval of 1[s] with a discretization interval of 0.1[s].
   COMSOL will solve the PDE equations and provide results like displacement (how much the membrane moves), stress distribution, and strain (deformation of the material).

A.2 Ansys

Skin is not a default ANSYS material, thus we need to start by creating our material using standard values for it which were found in literature. We start by defining the material properties of the membrane that mimic its behavior.

We used:

$$Young\ Modulus = 1.5e^6\ Pa$$

$$Poisson\ Ratio = 0,45$$

$$Density = 1109\ \frac{Kg}{m^3}$$

We continue by creating the geometry of the rectangular membrane within ANSYS which will be of dimensions 50mm x 50mm. We add to the geometry the diagonals lines which will then be used to apply the force at the center of our membrane, where the diagonals intersect.

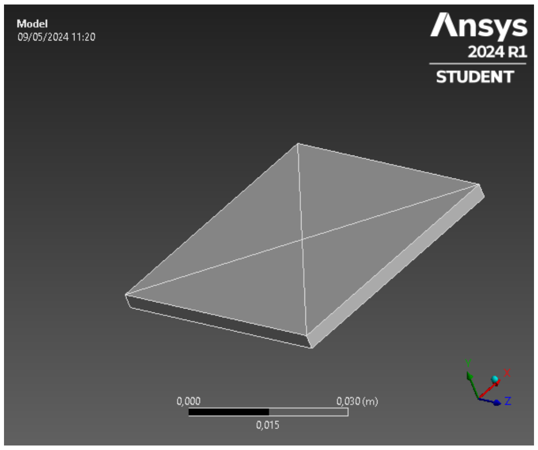

Figure 18: Geometry of the Membrane

We set a thickness of 3mm to simulate at best the skin environment.
Then, we generate a mesh, dividing the geometry into small, finite elements. The meshing process is crucial as it directly affects the accuracy of the simulation results.

We apply boundary conditions to simulate the physical environment. In this case, we apply a force at the center of the membrane to simulate the external force pushing on it. Additionally, we fix the outer edges of the membrane to represent fixed constraints.

After setting up the materials, geometry, and boundary conditions, we define the solver settings. ANSYS Mechanical uses numerical methods to solve the equations governing the behavior of the system, such as the equations of motion and stress equilibrium equations.

Once everything is set up, we run the analysis. ANSYS Mechanical calculates the stress distribution and deformation of the membrane under the applied force. It provides various output results, including von Mises stress, which is a scalar value used to evaluate the maximum equivalent stress in the material, and deformation, which shows how much the membrane has deformed under the applied load.

After the analysis is complete, we analyze the results using ANSYS Mechanical's post-processing tools. This includes visualizing stress contours, deformation plots, and other relevant data to understand how the membrane behaves under the applied load.

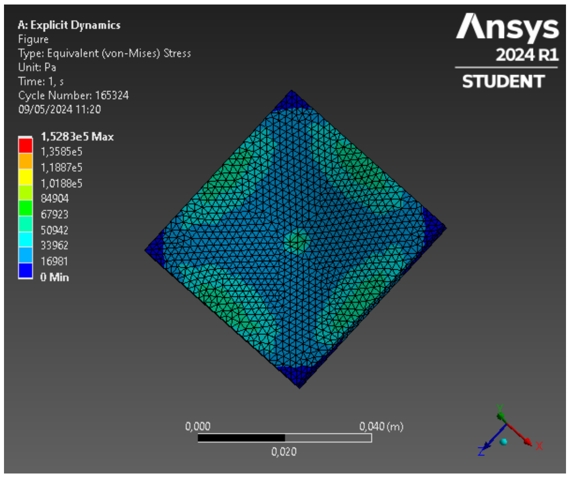

Figure 19: Von Mises Stress of the Membrane

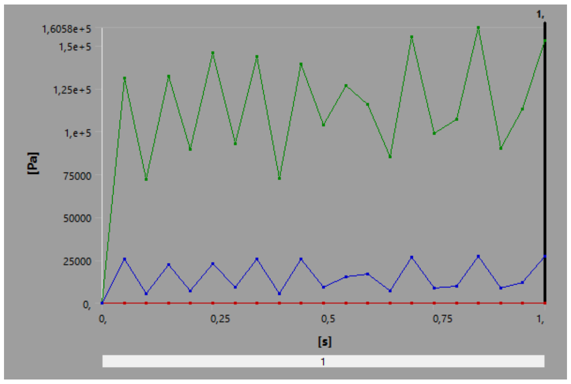

Figure 20: Max, min and average value of the Von Mises Stress

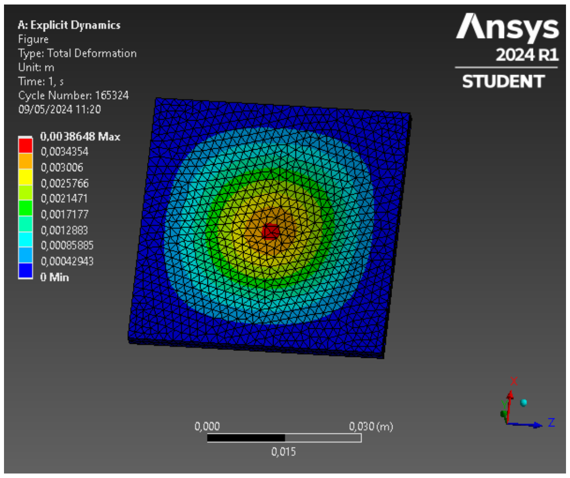

Figure 21: Deformation of the Membrane

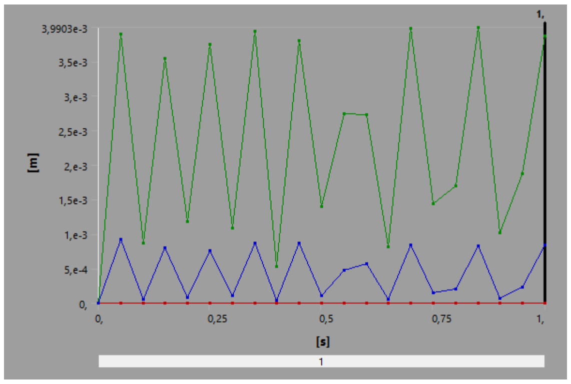

Figure 22: Max, min and average value of the deformation

B PINNs Code

B.1 DeepXDE

The following are the codes used to solve the PDEs, exploiting the framework provided by DeepXDE.

B.1.1 Rope Subject to a Force

```
import deepxde as dde
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import os
import pathlib
from scipy.integrate import quad
from matplotlib.animation import FuncAnimation
import tensorflow.compat.v1 as tf
tf.disable_eager_execution()


A = 2
```

```
ESK2 = 0.1


epochs = 20000


def func(x):
    x, t = np.split(x, 2, axis=1)

    return w1(x)


def w1(x):

    return 0


def w2(z):

    return 0



def go():

    sig = 50
    C = 10
    print(f"Start training C={C} sig={sig}")

    name = f"pinns_string_force"
    script_directory = pathlib.Path.cwd() / f"{name}"
    if not script_directory.exists():
        os.makedirs(script_directory, exist_ok=True)

    model_dir = os.path.join(script_directory, "model")
    if not os.path.exists(model_dir):
        os.makedirs(model_dir)

    def f(sample):
        x = sample[:, 0]
        x_f = 0.8
        height = 1
        t = sample[:, -1]

        alpha = 8.9
        za = -height * tf.exp(-400*((x-x_f)**2)) * (4**alpha * t**(alpha - 1) * (1 - t)**(alpha - 1))
        return za

    tf.reset_default_graph()
    def pde(x, y):
        dy_tt = dde.grad.hessian(y, x, i=1, j=1)
        dy_xx = dde.grad.hessian(y, x, i=0, j=0)

        # Uncomment the following to consider stiffness
        # dy_xxxx = dde.grad.hessian(dy_xx, x, i=0, j=0)
        return dy_tt - C ** 2 * dy_xx  - f(x)# + ESK2 * dy_xxxx


    def plot_animation(c, nume):
```

```
    filename = f"{script_directory}/animation_{nume}.gif"
    dx = 0.01  # Spatial step
    dt = 0.01  # Time step
    L = 1
    t_max = 1  # Maximum time

    # Discretization
    x_values = np.arange(0, L, dx)
    t_values = np.arange(0, t_max, dt)

    p = np.zeros((len(x_values), len(t_values)))

    # Set initial condition
    XX = np.vstack((x_values, np.zeros_like(x_values))).T
    p[:, 0] = c.predict(XX).reshape(len(XX), )

    # Update function using exact solution
    def update(frame):
        x = x_values.reshape(-1, 1)
        t = t_values[frame]
        XX = np.vstack((x_values, np.full_like(x_values, t))).T

        p[:, frame] = c.predict(XX).reshape(len(XX), ).flatten()

        pred_line.set_ydata(p[:, frame])

        plt.xlabel('x')
        plt.ylabel('Amplitude')
        plt.title(f'Wave Equation Animation at t={t:.2f}')
        plt.legend()  # Show legend with labels
        return pred_line

    # Create the animation
    fig, ax = plt.subplots()
    pred_line, = ax.plot(x_values, p[:, 0], label='Predicted')  # New line

    ax.set_ylim(-1, 1)  # Adjust the y-axis limits if needed

    ani = FuncAnimation(fig, update, frames=len(t_values), interval=1.0, blit=True)

    plt.show()

    # Save the animation as a GIF file
    ani.save(filename, writer='imagemagick')


geom = dde.geometry.Interval(0, 1)
timedomain = dde.geometry.TimeDomain(0, 1)
geomtime = dde.geometry.GeometryXTime(geom, timedomain)

# bc = dde.icbc.DirichletBC(geomtime, func, lambda _, on_boundary: on_boundary)
ic_1 = dde.icbc.IC(geomtime, func, lambda _, on_initial: on_initial)
# do not use dde.NeumannBC here, since `normal_derivative` does not work with temporal coordinate.
ic_2 = dde.icbc.OperatorBC(
    geomtime,
    lambda x, y, _: dde.grad.jacobian(y, x, i=0, j=1),
    lambda x, _: np.isclose(x[1], 0),
)
data = dde.data.TimePDE(
    geomtime,
    pde,
    [ic_1, ic_2],
    num_domain=1440,
    num_boundary=360,
    num_initial=360,
    num_test=10000,
)

layer_size = [2] + [100] * 3 + [1]
activation = "tanh"
initializer = "Glorot uniform"
net = dde.nn.STMsFFN(
    layer_size, activation, initializer, sigmas_x=[1], sigmas_t=[1, sig]
)


net.apply_output_transform(lambda x, y: x[:, 0:1] * (1 - x[:, 0:1]) * y)


model = dde.Model(data, net)
model.compile(
    "adam",
    lr=0.001,
    decay=("inverse time", 2000, 0.9),
)
pde_residual_resampler = dde.callbacks.PDEPointResampler(period=1)
early_stopping = dde.callbacks.EarlyStopping(min_delta=1e-6, patience=5000)
losshistory, train_state = model.train(
    iterations=epochs, callbacks=[pde_residual_resampler, early_stopping], display_every=500, model_
)
dde.saveplot(losshistory, train_state, output_dir=f"{script_directory}")
# model.restore(f"{model_dir}/{name}-{epochs}.ckpt", verbose=0)


# Predictions
t = np.linspace(0, 1, num=100)
x = np.linspace(0, 1, num=100)
t, x = np.meshgrid(t, x)
X_star = np.hstack((t.flatten()[:, None], x.flatten()[:, None]))

u_pred = model.predict(X_star)

# Plot
U_pred = u_pred.reshape(100, 100)

# Predictions
plt.figure()
plt.pcolor(t, x, U_pred, cmap='jet')
plt.colorbar()
plt.xlabel('$x$')
plt.ylabel('$t$')
plt.title('Predicted u(x)')
plt.savefig(f"{script_directory}/dde1_{name}.png")

# Convert the list of arrays to a 2D NumPy array
matrix = np.array(losshistory.loss_train)

# Separate the components into different arrays
```

```
    loss_res = matrix[:, 0]
    # loss_bcs = matrix[:, 1]
    loss_u_t_ics = matrix[:, 1]
    loss_du_t_ics = matrix[:, 2]

    l2_error = np.array(losshistory.metrics_test)

    fig = plt.figure(figsize=(6, 5))
    iters = 500 * np.arange(len(loss_res))
    with sns.axes_style("darkgrid"):
        plt.plot(iters, loss_res, label='$\mathcal{L}_{r}$')
        # plt.plot(iters, loss_bcs, label='$\mathcal{L}_{u}$')
        plt.plot(iters, loss_u_t_ics, label='$\mathcal{L}_{u_0}$')
        plt.plot(iters, loss_du_t_ics, label='$\mathcal{L}_{u_t}$')
        plt.plot(iters, l2_error, label='$\mathcal{L}^2 error$')
        plt.yscale('log')
        plt.xlabel('iterations')
        plt.legend(ncol=2)
        plt.tight_layout()
        plt.savefig(f"{script_directory}/dde2_{name}.png")
        plt.show()

    # plot_animation(model, name)

if __name__ == "__main__":

    go()
```

B.1.2 Membrane Subject to a Force

```
import deepxde as dde
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import os
import pathlib
from scipy.integrate import quad
from matplotlib.animation import FuncAnimation
import tensorflow.compat.v1 as tf
tf.disable_eager_execution()


A = 2
ESK2 = 0.1
C = 5

epochs = 20000


def func(x):
    x, y, t = tf.split(x, 3, axis=1)

    return w1(x)


def func2(x):
    x, y, t = tf.split(x, 3, axis=1)
    return w2(x, y)


def w1(x):

    return 0

def w2(x, y):

    return 0


def plot_animation(c, nume):
    filename = f"{nume}/animation.gif"

    dt = 0.01  # Time step
    t_max = 1  # Maximum time

    # Generate meshgrid for x and y values
    x_vals = np.linspace(0, 1, 100)
    y_vals = np.linspace(0, 1, 100)
    x, y = np.meshgrid(x_vals, y_vals)
    t_values = np.arange(0, t_max, dt)

    p = np.zeros((len(x_vals), len(y_vals), len(t_values)))

    # Set initial condition
    XX = np.vstack((x.flatten(), y.flatten(), np.zeros_like(x.flatten()))).T
    p[:, :, 0] = c.predict(XX).reshape(len(x_vals), len(y_vals))

    # Update function using predicted solution
    def update(frame):
        t = t_values[frame]
        XX = np.vstack((x.flatten(), y.flatten(), np.full_like(x.flatten(), t))).T

        p[:, :, frame] = c.predict(XX).reshape(len(x_vals), len(y_vals))

        ax.clear()
        ax.set_xlabel('X-axis')
        ax.set_ylabel('Y-axis')
        ax.set_zlabel('Deformation (Z-axis)')
        ax.set_title(f'Membrane Deformation at t={t:.2f}')

        # Plot the 3D surface without colormap
        surf = ax.plot_surface(x, y, p[:, :, frame], rstride=1, cstride=1, alpha=0.8, antialiased=True)

        # Set fixed z-axis limits
        ax.set_zlim(-1, 1)
        return surf,

    # Create the animation
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')

    ani = FuncAnimation(fig, update, frames=len(t_values), interval=1.0, blit=True)

    # Save the animation as a GIF file
    ani.save(filename, writer='imagemagick')
```

```
def f(sample):
    x = sample[:, 0]
    y = sample[:, 1]
    x_f = 0.8
    y_f = 0.5
    height = 1
    t = sample[:, -1]

    alpha = 8.9
    za = -height * tf.exp(-400*((x-x_f)**2 + (y-y_f)**2)) * (4**alpha * t**(alpha - 1) * (1 - t)**(alpha
    return za

def go():


    print(f"Start training membrane")

    name = f"pinns_membrane_force"
    script_directory = pathlib.Path.cwd() / f"{name}"
    if not script_directory.exists():
        os.makedirs(script_directory, exist_ok=True)

    model_dir = os.path.join(script_directory, "model")
    if not os.path.exists(model_dir):
        os.makedirs(model_dir)

    tf.reset_default_graph()
    def pde(x, z):
        dz_tt = dde.grad.hessian(z, x, i=2, j=2)
        dz_xx = dde.grad.hessian(z, x, i=0, j=0)
        dz_yy = dde.grad.hessian(z, x, i=1, j=1)

        # Uncomment the following to consider stiffness
        # dy_xxxx = dde.grad.hessian(dy_xx, x, i=0, j=0)
        return dz_tt - C ** 2 *(dz_xx+dz_yy) - f(x) # + ESK2 * dy_xxxx


    geom = dde.geometry.Rectangle([0, 0], [1, 1])
    timedomain = dde.geometry.TimeDomain(0, 1)
    geomtime = dde.geometry.GeometryXTime(geom, timedomain)

    # bc = dde.icbc.DirichletBC(geomtime, func, lambda _, on_boundary: on_boundary)
    ic_1 = dde.icbc.IC(geomtime, func, lambda _, on_initial: on_initial)
    # do not use dde.NeumannBC here, since ‘normal_derivative‘ does not work with temporal coordinate.
    ic_2 = dde.icbc.OperatorBC(
        geomtime,
        lambda x, y, _: dde.grad.jacobian(y, x, i=0, j=1) - func2(x),
        lambda x, _: np.isclose(x[1], 0),
    )
    data = dde.data.TimePDE(
        geomtime,
        pde,
        [ic_1, ic_2],
        num_domain=1440,
        num_boundary=360,
        num_initial=360,
        num_test=10000,
    )
```

```
layer_size = [3] + [100] * 3 + [1]
activation = "tanh"
initializer = "Glorot uniform"
net = dde.nn.STMsFFN(
    layer_size, activation, initializer, sigmas_x=[1], sigmas_t=[1, 10]
)


net.apply_output_transform(lambda x, y: x[:, 0:1] * (1 - x[:, 0:1]) * x[:, 1:2] * (1 - x[:, 1:2]) *


model = dde.Model(data, net)
model.compile(
    "adam",
    lr=0.001,
    decay=("inverse time", 2000, 0.9),
)
pde_residual_resampler = dde.callbacks.PDEPointResampler(period=1)
early_stopping = dde.callbacks.EarlyStopping(min_delta=1e-6, patience=5000)
losshistory, train_state = model.train(
    iterations=epochs, callbacks=[pde_residual_resampler, early_stopping], display_every=500, model_
)
dde.saveplot(losshistory, train_state, output_dir=f"{model_dir}")


# Predictions
x = np.linspace(0, 1, num=100)
y = np.linspace(0, 1, num=100)
x, y = np.meshgrid(x, y)
X_0 = np.hstack((x.flatten()[:, None], y.flatten()[:, None], np.zeros_like(x.flatten()[:, None])))
X_025 = np.hstack((x.flatten()[:, None], y.flatten()[:, None], np.full_like(x.flatten()[:, None], 0.
X_05 = np.hstack((x.flatten()[:, None], y.flatten()[:, None], np.full_like(x.flatten()[:, None], 0.5
X_075 = np.hstack((x.flatten()[:, None], y.flatten()[:, None], np.full_like(x.flatten()[:, None], 0.
X_1 = np.hstack((x.flatten()[:, None], y.flatten()[:, None], np.ones_like(x.flatten()[:, None])))

U_0 = model.predict(X_0).reshape(100, 100)
U_025 = model.predict(X_025).reshape(100, 100)
U_05 = model.predict(X_05).reshape(100, 100)
U_075 = model.predict(X_075).reshape(100, 100)
U_1 = model.predict(X_1).reshape(100, 100)


# Predictions
fig = plt.figure(5, figsize=(30, 5))

# Loop over the time steps
for i, (U, t) in enumerate([(U_0, 0), (U_025, 0.25), (U_05, 0.5), (U_075, 0.75), (U_1, 1.0)]):
    plt.subplot(1, 5, i + 1)
    plt.pcolor(x, y, U, cmap='jet')
    plt.colorbar()
    plt.xlabel('$x$')
    plt.ylabel('$t$')
    plt.title(f'Prediction at t={t}')

    # Set axis limits to range from 0 to 1
    plt.xlim(0, 1)
    plt.ylim(0, 1)

plt.tight_layout()
```

```
    plt.savefig(f"{script_directory}/dde1_{name}.png")
    plt.show()

    # Convert the list of arrays to a 2D NumPy array
    matrix = np.array(losshistory.loss_train)

    # Separate the components into different arrays
    loss_res = matrix[:, 0]
    loss_u_t_ics = matrix[:, 1]
    loss_du_t_ics = matrix[:, 2]

    fig = plt.figure(figsize=(6, 5))
    iters = 500 * np.arange(len(loss_res))
    with sns.axes_style("darkgrid"):
        plt.plot(iters, loss_res, label='$\mathcal{L}_{r}$')
        # plt.plot(iters, loss_bcs, label='$\mathcal{L}_{u}$')
        plt.plot(iters, loss_u_t_ics, label='$\mathcal{L}_{u_0}$')
        plt.plot(iters, loss_du_t_ics, label='$\mathcal{L}_{u_t}$')
        # plt.plot(iters, l2_error, label='$\mathcal{L}^2 error$')
        plt.yscale('log')
        plt.xlabel('iterations')
        plt.legend(ncol=2)
        plt.tight_layout()
        plt.savefig(f"{script_directory}/dde2_{name}.png")
        plt.show()

    plot_animation(model, script_directory)

if __name__ == "__main__":
    go()
```

B.2 Integration with CoppeliaSim

B.2.1 Rope

```
from coppeliasim_zmqremoteapi_client import RemoteAPIClient
import numpy as np
import copy
import threading
import os

import pinns_plucked

client = RemoteAPIClient()
sim = client.require('sim')

sim.setStepping(True)

dt = 0.01

sim.loadScene(os.path.abspath("./corda_pinns.ttt"))
sim.setFloatParam(sim.floatparam_simulation_time_step , dt)


spheres = []
initial_pos = []
lx = 1
spheres.append(sim.getObject("/Sphere"))
```

```
initial_pos.append(sim.getObjectPosition(spheres[0]))
j = sim.getObjectPosition(spheres[0])[1]
k = 1.0
# create the other spheres
x = np.linspace(0.0, lx, 100)

pinns_plucked.load_model("./sig50_C10/sig50_C10-40000.ckpt")

for i in x:
    if i == 0.0:
        continue
    s = sim.createPrimitiveShape(sim.primitiveshape_spheroid, [0.05, 0.05, 0.05])
    sim.setObjectPosition(s, [i, j, k])
    sim.setObjectInt32Parameter(s,sim.shapeintparam_static,1)
    sim.setObjectInt32Parameter(s,sim.shapeintparam_respondable,1)
    spheres.append(copy.deepcopy(s))
    initial_pos.append([i, j, k])

sim.startSimulation()

def func():
    if (t := sim.getSimulationTime()) < 10:
        #print(f'Simulation time: {t:.2f} [s]')
        sim.step()
        pred_pos = pinns_plucked.predict(x, [t])
        #print(pred_pos)
        for i in range(len(spheres)):
            new_pos = [initial_pos[i][0], initial_pos[i][1], pred_pos[i]+initial_pos[i][2]]
            sim.setObjectPosition(spheres[i], new_pos)
        threading.Timer(dt, func).start()

    else:
        sim.stopSimulation()
        while (sim.getSimulationState() != sim.simulation_stopped):
            continue
        sim.closeScene()

func()
```

B.2.2 Membrane

```
from coppeliasim_zmqremoteapi_client import RemoteAPIClient
import numpy as np
import copy
import threading
import os

import pinns_membrane_nostro

client = RemoteAPIClient()
sim = client.require('sim')

sim.setStepping(True)

dt = 0.01

sim.loadScene(os.path.abspath("./corda_pinns.ttt"))
sim.setFloatParam(sim.floatparam_simulation_time_step , dt)
```

```
spheres = []
initial_pos = []
lx = 1
ly = 1
spheres.append(sim.getObject("/Sphere"))
initial_pos.append(sim.getObjectPosition(spheres[0]))
j = sim.getObjectPosition(spheres[0])[1]
k = 1.0
# create the other spheres
x = np.linspace(0.0, lx, 20)
y = np.linspace(0.0, ly, 20)

pinns_membrane_nostro.load_model("./pinns_membrane_force/model/model-20000.ckpt")

for j in y:
    for i in x:
        if (i == 0.0) and (j == 0.0):
            continue
        s = sim.createPrimitiveShape(sim.primitiveshape_spheroid, [0.05, 0.05, 0.05])
        sim.setObjectPosition(s, [i, j, k])
        sim.setObjectColor(s, 0, sim.colorcomponent_ambient_diffuse, [50,1,50])
        sim.setObjectInt32Parameter(s, sim.shapeintparam_static,1)
        sim.setObjectInt32Parameter(s, sim.shapeintparam_respondable,1)
        spheres.append(copy.deepcopy(s))
        initial_pos.append([i, j, k])

sim.startSimulation()

def func():
    if (t := sim.getSimulationTime()) < 10:
        #print(f'Simulation time: {t:.2f} [s]')
        sim.step()
        pred_pos = pinns_membrane_nostro.predict(x, y, [t])
        #print(pred_pos)
        for j in range(len(y)):
            for i in range(len(x)):
                new_pos = [initial_pos[(len(y)*i+j)][0], initial_pos[(len(y)*i+j)][1], -10*pred_pos[i,j
                sim.setObjectPosition(spheres[(len(y)*i+j)], new_pos)
        threading.Timer(dt, func).start()

    else:
        sim.stopSimulation()
        while (sim.getSimulationState() != sim.simulation_stopped):
            continue
        sim.closeScene()

func()
```

References

[1] Tan-Nhu Nguyen, Marie-Christine HO BA THO, and Tien-Tuan Dao. A systematic review of real-time medical simulations with soft-tissue deformation: Computational approaches, interaction devices, system architectures, and clinical validations. *Applied Bionics and Biomechanics*, 2020:1–30, 02 2020.

[2] Sarthak Misra, K. T. Ramesh, and Allison M. Okamura. Modeling of tool-tissue interactions for computer-based surgical simulation: A literature review. *Presence: teleoperators and virtual environment*, 17(5):463–491, 2008. October, 2008.

[3] Ehsan Sadraei, Mohamad H. Moazzen, Majid M. Moghaddam, and Faeze Sayad Sijani. Real-time haptic simulation of soft tissue deformation. In *2014 Second RSI/ISM International Conference on Robotics and Mechatronics (ICRoM)*, pages 053–058, 2014.

[4] François Faure, Christian Duriez, Hervé Delingette, Jérémie Allard, Benjamin Gilles, Stéphanie Marchesseau, Hugo Talbot, Hadrien Courtecuisse, Guillaume Bousquet, Igor Peterlík, and Stéphane Cotin. Sofa, a multi-model framework for interactive physical simulation. 2011.

[5] Zachary Pezzementi, Daniel Ursu, Sarthak Misra, and Allison M. Okamura. Modeling realistic tool-tissue interactions with haptic feedback: A learning-based method. In *2008 Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems*, pages 209–215, 2008.

[6] Sarthak Misra. Realistic tool-tissue interaction models for surgical simulation and planning. 01 2009.

[7] **Comsol Multiphysics® Simulation Software Understand, Predict, and Optimize Real-World Designs, Devices, and Processes with Simulation** .

[8] **Ansys Software®, a Suite Used for Engineering Simulation, to Test Designs Before Physically Building Them.** .

[9] **Deepxde, a Library for Scientific Machine Learning and Physics-Informed Learning** .

[10] CRAIG GIN, BETHANY LUSCH, STEVEN L. BRUNTON, and J. NATHAN KUTZ. Deep learning models for global coordinate transformations that linearise pdes. *European Journal of Applied Mathematics*, 32(3):515–539, 2021.

[11] Sifan Wang, Shyam Sankaran, Hanwen Wang, and Paris Perdikaris. An expert's guide to training physics-informed neural networks. *ArXiv*, abs/2308.08468, 2023.

[12] Simone Monaco and Daniele Apiletti. Training physics-informed neural networks: one learning to rule them all? *Results in Engineering*, 18:101023, 03 2023.