Data-driven computational biomechanics using deep neural networks : application to augmented surgery

Alban Odot

Université de Strasbourg

UNIVERSITÉ DE STRASBOURG

msii ÉCOLE DOCTORALE

ÉCOLE DOCTORALE MSII 269

Mathématiques, Sciences de l'Information et de l'Ingénieur

Institut national de recherche en sciences et technologies du numérique

Thèse présentée par :

Alban ODOT

soutenue le : 7 novembre 2023

pour obtenir le grade de : **Docteur de l'Université de Strasbourg**

Discipline/ Spécialité : **Informatique**

Data-driven computational biomechanics using Deep Neural Networks – Application to augmented surgery

THÈSE dirigée par :

Dr Stéphane Cotin	Directeur de recherche INRIA, Strasbourg

MEMBRES DU JURY :

Rapporteurs :

Dr Maud MARCHAL	Professeur des Universités, INSA Rennes
Dr Jérémie Dequidt	Professeur des Universités, Université de Lille

Examinateurs :

Dr Stéphane BORDAS	Professeur des Université, Université du Luxembourg
Dr Florence Zara	Maître de conférence, Université Claude Bernard Lyon 1

Université de Strasbourg

Alban ODOT

iCUBE

Data-driven computational biomechanics using Deep Neural Networks – Application to augmented surgery

Résumé

Cette thèse aborde le problème de la simulation des tissus mous pour les applications de réalité augmentée dans l'assistance à la chirurgie du foie. En particulier, nous mettons en œuvre un pipeline de recalage non rigide pour générer des déformations interactives d'une représentation virtuelle du foie des patients. Les méthodes traditionnelles de calcul de déformations réalistes ne peuvent pas fonctionner à une fréquence interactive (60 images par seconde) avec des données spécifiques au patient. Récemment, des chercheurs ont utilisé des réseaux neuronaux artificiels pour calculer des déformations réalistes d'objets virtuels similaires en deux millisecondes avec précision et une relative stabilité. Nous proposons d'utiliser une approche similaire, utilisant cependant une architecture de réseau neuronal artificiel différente offrant une précision égale tout en étant dix fois plus rapide. Avec cette proposition, nous présentons également une nouvelle méthode pour générer un ensemble de données qui minimise les entrées de l'utilisateur tout en maintenant le contrôle sur le contenu grâce à une analyse des propriétés mécaniques de l'objet. En outre, nous montrons qu'il est possible d'améliorer la fiabilité du réseau neuronal artificiel en utilisant sa prédiction comme initialisation de l'algorithme de Newton-Rapshon utilisé par les méthodes traditionnelles. A l'aide de nos contributions précédentes, nous construisons un pipeline de recalage non rigide en utilisant le cadre du contrôle optimal et l'algorithme de rétropropagation. Ce pipeline effectue le calcul plusieurs ordres de grandeur plus rapidement que les méthodes traditionnelles au prix d'un bruit de reconstruction de contrôle. Enfin, nous construisons un second pipeline de recalage non rigide en mettant en œuvre un moteur physique de corps mou entièrement différentiable qui est plus lent que les réseaux neuronaux artificiels mais plus flexible dans le type de contrôles, fiable et précis.

Mots clés : Méthode aux éléments finis, Apprentissage profond, Solveur differentiable, Recalage de forme, Contrôle optimal.

Université de Strasbourg

Alban ODOT

iCUBE

Data-driven computational biomechanics using Deep Neural Networks – Application to augmented surgery

Abstract

This thesis addresses the problem of soft tissue simulation for augmented reality applications in liver surgery assistance. In particular, we are implementing a non-rigid registration pipeline to generate interactive deformations of a patient-specific liver virtual representation. Traditional methods to compute realistic deformations cannot run at an interactive framerate (60 frames per second) with patient-specific data. Recently, researchers have used artificial neural networks to compute realistic deformations of resembling virtual objects in two milliseconds with accuracy and a relative stability. We propose using a similar approach but with a different artificial neural network architecture having equal precision while being ten times faster. With this proposition, we also present a new method to generate a dataset that minimizes user inputs but maintains control over the content using an analysis of the mechanical properties of the object. Furthermore, we show that it is possible to improve the reliability of the artificial neural network by using its prediction as the initialization of the Newton-Rapshon algorithm used by the traditional methods. Using our previous contributions, we build a non-rigid registration pipeline using the optimal control framework and the backpropagation algorithm. This pipeline performs the computation multiple orders of magnitude faster than traditional methods at the cost of control reconstruction noise. Finally, we build a second non-rigid registration pipeline by implementing a fully differentiable soft-body physics engine that is slower than artificial neural networks but more flexible in the type of controls, reliable and precise.

Key words : Finite element method, Deep learning, Differentiable solver, Non-rigid registration, Optimal control.

ACKNOWLEDGMENTS

J'aimerais remercier mon directeur de thèse, Stéphane Cotin, qui m'a permis de réaliser ce projet de thèse, et par la même occasion un rêve d'enfant. Merci Stéphane pour ton soutient, tes sages conseils et ta patience.

Je voudrais aussi exprimer ma gratitude envers les membres du jury de mon comité de thèse: Jérémie Dequidt, Maud Marchal, Stéphane Bordas et Florence Zara. Je tiens à remercier plus spécialement les rapporteurs Maud Marchal et Jérémie Dequidt. Avoir eût cette thèse lue et rapportée par deux experts de renoms est un honneur.

Je tiens à remercier Hugo Talbot et Jean-Nicolas Brunet pour m'avoir si bien intégré dans l'équipe à mon arrivée, puis pour votre disponibilité durant le reste de celle-ci.

Merci à Paul Baksic, mon ~~camarade de thèse~~ ami, pour tous ces bons moments, fous rires et conseils.

Je voudrais remercier Virginie Marec, pour son appui incommensurable durant ces deux dernières années et l'aide que tu m'as apporté lors de la rédaction et la relecture de ce manuscrit.

Merci aussi à mes parents et grand-parents pour leur soutien tout au long de mes études et plus spécialement dans le supérieur.

Finalement, j'aimerais remercier les enseignants qui ont cru en moi et m'ont redonné goût à l'école: Mme Yap, Mme Colomier, M. Laissac

CONTENTS

Contents i

Notations iii

1 Introduction 1
- 1.1 Motivation . . . 2
- 1.2 Numerical model . . . 4
- 1.3 Partial surface shape matching . . . 7
- 1.4 Objectives and scientific contribution of this thesis . . . 9

2 Finite element method 13
- 2.1 Mesh definition . . . 14
- 2.2 Isoparametric elements . . . 16
- 2.3 Tensors and quantities transformations . . . 18
- 2.4 Hyper-elastic material . . . 22
- 2.5 Strong and weak forms . . . 25
- 2.6 Solving the equation . . . 27

3 Deep learning 29
- 3.1 Generalities . . . 30
- 3.2 Core components of an artificial neural network . . . 34
- 3.3 Architectures . . . 36
- 3.4 Loss function . . . 39
- 3.5 Network optimisation . . . 40
- 3.6 Example of training process . . . 43

4 Fast and accurate deformations using deep learning 49
- 4.1 Dataset generation . . . 55
- 4.2 Toward faster simulations using artificial neural networks . . . 60

5 Hybrid solver **69**
5.1 Newton method 70
5.2 Artificial neural network and solver 72

6 Optimal control for augmented surgery **79**
6.1 Context 81
6.2 Shape matching 84

7 Lastest optimisation tool: Differentiable simulation **103**
7.1 DiffEn : A differentiable solver based on energy 105
7.2 Results and future works 118

8 Conclusion **135**
8.1 Summary and achievements 135
8.2 Outlook and futur work 137

9 Résumé en français **139**

Bibliography **161**

List of Figures **173**

List of Tables **179**

NOTATIONS

Acronymes

AI	Artificial intelligence
ANN	Artificial Neural Network
AR	Augmented Reality
CNN	Convolutional Neural Network
DOFS	Degrees Of Freedom
FEA	Finite Element Analysis
FEM	Finite Element Method
GNN	Graph Neural Network
ML	Machine Learning
MLP	MultiLayers Perceptron
TRE	Target Registration Error

CHAPTER 1

INTRODUCTION

1.1 Motivation 2
1.2 Numerical model 4
1.3 Partial surface shape matching 7
1.4 Objectives and scientific contribution of this thesis 9

1.1 Motivation

According to the World Health Organization (WHO), nearly twenty million new cancers were diagnosed in 2020. This disease can affect any body part, but the main affected parts are the breast, lungs, and colon. Among other factors, one-third of deaths from cancer are due to excessive consumption of alcohol and tobacco, high-fat mass index, lack of vegetable intake, and physical activity. Thanks to the recent advances in medications and medical procedures, many cancers can be cured if detected early. Nevertheless, all cancers are not equal regarding treatment response and death rate.

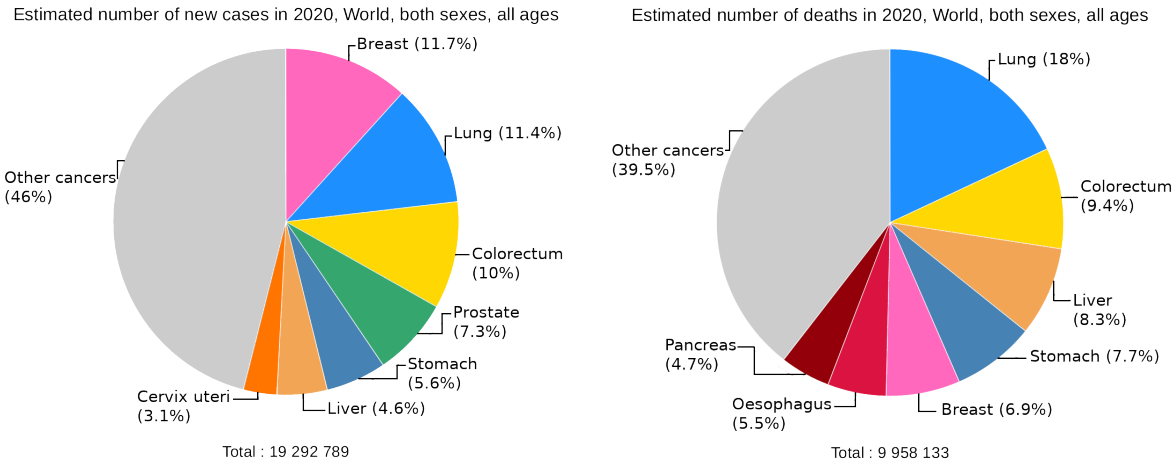


Figure 1.1: Pie-charts displaying worldwide newly diagnosed (left) and deaths (right) by cancer type in 2020 based on World Health Organization data. Note that there are no necessary intersections in populations between figures since people appearing in the second figure could have been diagnosed multiple years prior.

Liver cancers are especially deadly, as shown in Figure 1.1, where they account for approximately one in twenty diagnoses but represent one in twelve deaths induced by cancer. The most common primary liver cancers (originating from the liver) are hepatocellular carcinoma and cholangiocarcinoma. Such pathologies are usually treated via liver resection using either open abdominal or laparoscopic surgery.

During open abdominal surgery, a single long incision, or a laparotomy, is made to gain access to the abdominal cavity. This approach allows direct visualization of the organs within the abdomen and is most commonly used for more complex surgical procedures.

In laparoscopic surgery, the entire procedure is carried out through small incisions that resemble keyholes while the surgeon watches the images transmitted by a camera on a screen. Specially designed instruments are used to perform liver resection, and robotic arms can also be utilized. Although movement may be somewhat restricted, laparoscopic

or robotic techniques are frequently used for less complex liver resections. With advancements in surgical techniques, however, more intricate procedures such as major liver resections or living donor liver resections are now feasible through a laparoscopic or robotic approach.

During the manipulation, surgeons must remember and project patient-specific data such as tumor locations and sizes, blood vessels, and arteries in an environment that is constantly moving due to respiration and blood flow. On top of this tracking which already requires much focus, they have to perform precise and complex surgical acts while commanding the whole operation room.

One idea to facilitate the task and reduce the surgeon's mental load would be to enhance the surgeon's visual perception of the patient using augmented reality (AR). This can be achieved by combining a live video feed from the laparoscope (a small camera inserted into the patient's body through a small incision) with virtual images and data overlaid. This can include 3D models of the patient's anatomy, surgical instructions, and real-time data such as vital signs.

Figure 1.2: Illustration of a laparoscopic liver biopsy [21].

One of the main benefits of AR in laparoscopic surgery is that it allows the surgeon to see inside the patient's body more intuitively and naturally, which can help to improve accuracy and precision during the procedure. 3D images and virtual overlay can also help to enhance the visualization of complex anatomy and internal structures, which can be challenging to see with traditional laparoscopic techniques.

AR technology can also assist surgeons with tracking and visualization of surgical instruments, guides, and implants within the body, which can help to reduce the time of surgery and complications.

Finally, AR can also improve communication and collaboration between surgical teams, with the ability to share images and data and even remotely operate surgical instruments.

When overlaying 3D models of the patient's anatomy during surgery, one has to consider the deformation induced by the surgeon and reproduce them on the 3D models. The complexity arises in detecting deformations and preserving the internal properties of the organs. Such technology would require the ability to compute in real-time complex deformations of organs from partial surface observations, thus, asking two important questions :

1. How to compute the deformations of detailed objects in real-time?
2. How to process partial surface data from the laparoscope video feed to compute the corresponding deformations?

The following sections present our choices to answer these questions, which the results in the subsequent chapters will further support.

1.2 Numerical model

This section provides an introductory answer to the first question by placing the thesis work in its context.

Picking the proper set of equations is essential to correctly model a physical phenomenon. In our case, the liver is an object with a defined rest shape and can undergo deformations that we assume are non-damaging for the organ. This places us in the elasticity framework and, more specifically, nonlinear elasticity with a set of equations detailed in Chapter 2. The simulation of such objects is called "Soft body simulation" and can be achieved using multiple techniques such as mass-spring systems, finite element analysis (FEA), and many others.

Simulations that focus on realism use a subfield of the FEA called finite element methods (FEM). The Finite Element Method is a powerful numerical technique widely used to solve various physical problems in various fields, such as solid mechanics, fluid dynamics, heat transfer, and electromagnetism. FEM can simulate the behavior of elastic materials, such as cloth, rubber, and skin, under different physical conditions.

The simulated object is represented by many interconnected elements, known as finite elements. A set of nodes and edges defines these elements, and each element has a specific set of physical properties, such as density, stiffness, and mass.

The simulation process involves the calculation of the quantities acting on each element based on the object's properties and the environment's physical conditions, such as gravity, collisions, and temperature. These quantities are then used to update the position and velocity of each element, creating a realistic and dynamic simulation of the object's behavior.

Such simulations are used in various applications, including animation and cloth simulation. It is also used in engineering and manufacturing to simulate the behavior of soft materials such as rubber and in medical simulation to simulate the behavior of human tissue.

FEM-based soft body simulations can be computationally intensive, and the accuracy of the simulation depends on the quality of the underlying finite element model and the

computational resources available. However, recent advances in computing power and the development of more efficient algorithms have made it possible to create highly realistic and accurate simulations of soft bodies.

One of the critical aspects of this method is that it can handle various types of boundary conditions and works with both linear and nonlinear systems. Nonlinear systems are handy when dealing with complex problems using elastic, plastic, or viscoelastic materials.

This versatility also applies to the type of system. It can handle static and time-dependent problems on complex geometries and topologies, making it useful for various engineering problems.

The accuracy of the solution can be improved by refining the mesh and increasing the number of elements. Since the method is not defined for specific elements, it can be applied to large-scale problems with millions of degrees of freedom. Developing specialized algorithms, hardware, and parallel computing has enabled solving large, complex systems with high computational efficiency and scalability.

However, all of these advantages come with their associated drawbacks.

The previously mentioned mesh refining for accurate solutions relies on excellent mesh generation. One of the first steps in FEM is to divide the system into small, simple, and manageable finite elements. This process, known as mesh generation, can be challenging, especially for complex geometries and topologies. Creating an accurate and efficient mesh is crucial for the accuracy and convergence of the FEM solution.

Furthermore, FEM solutions are usually obtained by iteratively refining the mesh and solving the equations on each element. This process can be sensitive to the initial conditions, the choice of numerical solvers, and the mesh quality. Ensuring the solution converges to the correct solution can be challenging, especially for nonlinear and time-dependent problems.

FEM can be computationally intensive, requiring significant computational power. This can be incredibly challenging for large and complex systems or simulations requiring real-time performance. In real-world applications, most models require dealing with large-scale problems with millions of degrees of freedom. These can be computationally demanding, and ensuring that the algorithms are efficient and scalable is essential.

Complex problems are often assumed to be represented by nonlinear systems, but such modeling can be challenging. Nonlinear systems can have multiple solutions, and the FEM solution may not be unique or may not converge to the correct solution. This is partly because solutions are based on assumptions and approximations, and it is essential to validate and verify the solutions against experimental data or analytical solutions. This can be challenging, especially for complex systems and simulations that involve multiple

physical phenomena.

Having discussed the pros and cons of the finite element method for simulating soft bodies, we will now discuss the role of deep learning in realistic physical simulations.

1.2.1 Fast computation using deep learning

The main idea behind using deep learning in physics simulation is to use artificial neural networks (ANN) to learn the underlying physical laws of a system from data and then use this knowledge to make predictions about the system's behavior under different conditions.

Deep learning can be used to model complex physical systems, such as fluid dynamics, granular materials, and soft body dynamics, that can be difficult to simulate using traditional physics-based methods. This is done by training neural networks on large amounts of data generated from simulations or experiments and then using the trained networks to predict the system's behavior.

One of the main benefits of using deep learning in physics simulation is that it can alleviate some limitations of the finite element method. It can learn from real-world data, making the simulation more accurate and realistic. This can be achieved by changing simulation parameters such as element-wise properties or boundary conditions.

Furthermore, as seen, the finite element method is versatile and precise, but introducing complexity in simulations significantly increases computation time. This increase in computation time is usually due to a large number of solver iterations or a poorly constrained problem. One could use an artificial neural network to predict the result of a simulation and feed this result to a classic simulation. The fed simulation only corrects minor imperfections due to the neural network approximations, thus requiring a relatively low number of solver iterations. This idea has been successfully developed in this thesis and shows drastic improvement in computation time (Chapter 5).

It is important to note that deep learning in physics simulation is still an active area of research and is still in its early stages. There are challenges, such as the need for large amounts of high-quality data, interpretability, and generalization of the models. Additionally, it is essential to ensure that the predictions made by the neural network are physically meaningful and consistent with the laws of physics. This can be achieved by adding physics-based terms in the learning policy, which we will discuss in the Chapter 4 Despite these challenges, combining deep learning and physics simulation has shown great promise in creating more realistic and accurate simulations of various physical systems. It has the potential to revolutionize how we simulate and understand complex physical phenomena and is expected to have a wide range of applications in fields such as en-

gineering, science, and computer graphics. However, more research is needed to fully understand its capabilities and limitations and address the above mentioned challenges.

1.3 Partial surface shape matching

This section provides an introductory answer to the second question by placing the thesis work in its context.

Successful surgery usually follows detailed pre-operative planning. This planning is achieved via tomographic imaging of the zone of interest, creating detailed 3D models of the patient's organ, arteries, tumors, and other specificities. Adding a volumic element to this existing mesh puts us in a context where we can simulate organ deformations using the finite-element method, as presented in the previous section. This part is already interesting since it could be automatized without adding any cost or delay to the surgery.

During surgery, the liver undergoes significant deformations due to the practician manipulations. We want to be able to deform the 3D model to match its current real shape. This way, we can overlap them on the screen and keep realistic track of the points of interest. One way to observe the real liver is to use the camera's video feed in laparoscopic surgery. Using already existing techniques [97], we can convert the observations into partial surfaces 3D point-cloud.

We now have two important pieces of data: the organ at rest and the partial surface reconstruction of the same deformed organ.

How to use the video to deform the 3D model? Our answer is in three steps:

1. Generate a 3D point cloud from the video feed

2. A rigid registration where we fit the undeformed organ into the point cloud.

3. An elastic registration that deforms the 3D model to fit the observed data.

While the first and second steps are open research topics, we will focus on the last one that fits the rest of the ideas developed in this thesis.

Helping the surgeon to visualize internal structures during surgery requires computing a displacement field of the 3D model that fits the observed data. This fitting raise multiple challenges.

The first one is realism; to provide helpful information, the displacement of each model node is as close as possible to reality, which requires much computation.

The second one is real-time. Interactivity between the surgery and the simulation requires the simulation to run smoothly at a decent framerate ($\approx$60 frames per second). This

constraint opposes the first one, which will require a trade-off since, for each frame, we have to compute the three previously mentioned steps. Though we are not expecting to meet the real-time requirement for the whole process in this work, we should remember that involved procedures should be as fast as possible.

The third challenge is that a point cloud of a part of the surface is insufficient to identify a unique solution. Consequently, other information, such as physical hypotheses, should be added before we start thinking about registration accuracy. The chosen model and methods should result from a trade-off between the physical acceptability of the solution and the efficiency required by real-time execution.

The problem of organ registration in augmented surgery is familiar; so far, two main methods exist.

The first is inspired by the Iterative Closest Point algorithm [7] (ICP). A term is added to the minimized equation that can be understood as fictitious forces between the observed data and the liver model. In the case of Haouchine et al. [39], these forces are generated using linear spring; Plantefève et al. [93] proposed to change these springs to an attractive nonlinear force.

The second method relies on the formulation of an inverse problem. For example, a displacement could be imposed on the posterior face of the liver. At the same time, the anterior is observed by a camera as done by Rucker et al. [102] or by Heiselman et al. [41]. The latter also adds a displacement constraint on the liver ligaments. Both works use linear elasticity, significantly reducing computation time since it can be preprocessed. It is interesting to note that Özgür et al. [87] proposed to take into account the effect of gravity in pre- and per-operative computation and pneumoperitoneum pressure in intra-operative conditions to determine the final shape of the liver.

All of these methods lead to accurate results. However, they are too specific to a given problem and are often tailored for linear elasticity, which does not accurately model organs under essential deformations.

Recently, Mestdagh et al. [78] proposed a third method to formulate the optimization problem in the generic optimal control framework. This generic framework makes it easy to incorporate additional pre- or intra-operative data in the computation. Their results are presented on a model with nonlinear elasticity properties, which is an improvement compared to other methods. Furthermore, they implement it using the adjoint method, which allows for a naturally integrated neural network, as presented in chapter 6. For these reasons, we decided to continue with this approach, where we mostly worked on improving the computation speed using an artificial neural network, as we will see in Chapter 6.

In the next section, we present the contributions of this thesis.

1.4 Objectives and scientific contribution of this thesis

This thesis aims to propose new approaches to real-time numerical simulation to meet the expectations in the field of per-operative assistance. As a result, the surgeon can see the organ's internal structures, such as blood vessels or tumors, during the operation.

Achieving these goals requires overcoming many challenges. In particular, realistic numerical models of the patient must be developed while remaining compatible with real-time computing constraints. In the context of per-operative assistance, organs do not show an important dynamic because they do not bounce, fall or spin and, thus, are accurately represented by static simulations. The finite element method is one of the most widely used techniques for predicting the deformation of human tissue. However, this method involves complex calculations resulting in computation times incompatible with the constraints of the applications listed above.

These methods are based on simplifying behavioral law or specific numerical strategies. In this thesis, we propose the use of Machine Learning techniques. This idea fits very well in the context of image-guided surgery since it is possible to collect much information during an intervention and thus continuously learn to improve the modeling. Today, learning techniques, particularly Deep Learning, have shown spectacular results in computer vision or image processing. However, apart from very recent and still preliminary work, few results have been demonstrated in biomechanics.

This thesis continues the previous works of the team (based on deep neural networks) to model an organ's biomechanical behavior according to nonlinear behavioral laws to guide surgeons during interventions where it is essential to take into account movements and deformations of the anatomy. Thus we propose to replace the classical steps of experimentation, modeling, and simulation, typically used in biomechanics, with a single step capable of generating a solution based on a set of experimental and synthetic data.

This thesis has three main objectives.

The first is to give a better understanding of artificial neural network training and improve it using the finite element method in real-time soft body simulation.

The second one consists in improving the computation time of the FEM using artificial neural network.

The last one is to provide a new way to achieve the required surface matching in augmented reality coupled with FEM simulation.

Contributions are split into two parts, code and scientific contributions.

Scientific contributions :

1. The journal paper "DeepPhysics: a physics aware deep learning framework for real-

time simulation"[84] has three contributions :

- A novel method to sample the deformation space of an object using the eigen-decomposition of the tangent stiffness matrix.
- A formulation of the loss function based on the relative value of the force residual in the system.
- An improved Newton-Raphson scheme using a neural network prediction as an initial guess.

2. Conference paper "Real-time elastic partial shape matching using a neural network-based adjoint method"[85] has two contributions :

   - Forward simulation using an artificial neural network in the optimal control framework.
   - Adjoint method using the backpropagation through the ANN.

3. Book chapter "Deep learning for real-time computational biomechanics"[76] offers a comparison between two architectures for real-time computational biomechanics.

Code contributions:

1. The DeepPhysX project provides Python packages allowing users to interface their numerical simulations with learning algorithms easily.

2. The DiffEn project provides a fully differentiable soft-body physics engine based on energy formulation.

This thesis is split into seven chapters starting with the introduction (Introduction).

The second one (Finite element method) focuses on developing tools and notations to understand the finite-element method.

The third chapter (Deep learning) is in the same spirit but on the subject of deep learning.

The fourth chapter (Fast and accurate deformations using deep learning) contains our first contribution and answers the first question about how to improve the computation speed of the finite-element method.

The fifth chapter (Hybrid solver) is in the continuity of the fourth and presents our contribution to improving the robustness of the neural network prediction.

The sixth chapter (Optimal control for augmented surgery) presents our answer to the second question on how to process partial surface data to compute 3D model deformations.

The seventh chapter (Lastest optimisation tool: Differentiable simulation) presents our latest results with the development of a fully differentiable soft-body physics engine.

We will finish with a conclusion on the overall work of the thesis and the openings it created.

CHAPTER 2

FINITE ELEMENT METHOD

2.1 Mesh definition 14
2.2 Isoparametric elements 16
2.3 Tensors and quantities transformations 18
2.4 Hyper-elastic material 22
2.5 Strong and weak forms 25
2.6 Solving the equation 27

In this thesis we discuss numerical models that are derived from basic elasticity principles originating from the field of continuum mechanics. Using some well-known mechanical concepts we are then able to compute the deformation of an object subject to external load. One of these concepts is the displacement function.

The displacement function returns a vector between the initial and terminating positions of a deformed point of the object for any point inside the object. Researcher shave been studying different kind of elastic object through the years and have established relationship between the applied forces and the object responses. These relationships are formulated as function of the derivative of the displacement functions. A set of Partial Differential Equation (PDE) must be solved to get an explicit representation of the deformation.

This deformation can be seen as the balance between internal elastic forces of the object and the external forces exerted on it. In reality, it works the other way around where the object deforms to store potential elastic energy in order to regain its initial shape when the load is removed. Yet, to the equilibrium state this potential energy is equal to the external energy applied. This is why solving the displacement function from PDE yields the displacement of any points from the rest position to its deformed position when the balance of energy is reached.

In this chapter we present the mathematical framework behind these concepts. We will start by defining what is a mesh and how we interpolate quantities in its elements. Then we will present the different tensors associated with the problem and how they are combined to construct the system of PDE we will solve. Finally, we will present a common formulation to solve such set of equation using the Newton-Rapshon method.

2.1 Mesh definition

Most continuous problems cannot be solved directly and thus require an approximation. This approximation usually uses a variational approach. One formulation of such variational methods is the Galerkin Method. They convert a continuous PDE in its integral (weak) form into a discrete problem by applying linear constraints determined by a finite set of basis functions. If this sentence might be blurry right now, it should be clearer by the end of this chapter. Basically, this says that Galerkin's methods give a way to solve continuous problems by discretizing them into smaller elements. A mesh describes a continuous surface or volume into smaller discrete elements on which we know exists a way to approximate the solution of the governing PDE.

A mesh is a set of nodes and edges that approximate the real continuous shape of

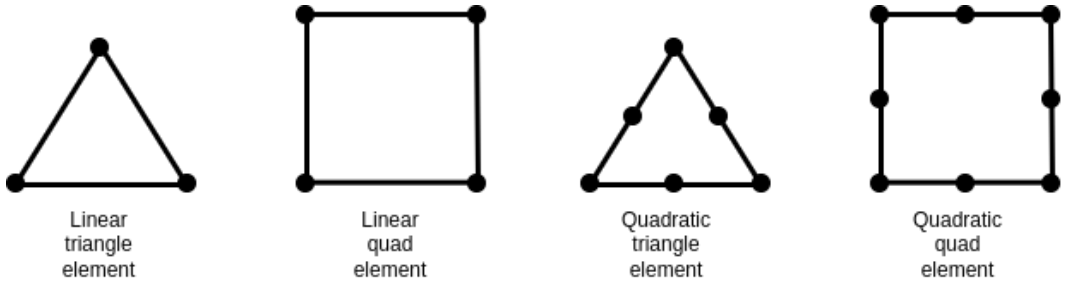


Figure 2.1: Reference surface elements. Quadratic elements have more integration points (black dots) than linear, thus, generating bigger systems to solve.

an object. These edges and nodes are arranged in elements on which we will perform computations. Meshing a two- or three-dimensional domain is a complicated task and remains a research subject today. Multiple meshing techniques exist to fit as many regular polygons as possible in a given shape. They all try to minimize the loss or gain of volume without introducing too many irregular polygons. In the context of finite element methods, these polygons are called elements. Before we continue and define mechanical properties on meshes, let us have a rapid introduction to the type of elements used in FEM.

Simulations usually use triangular or quadrilateral elements when dealing with 2D objects such as cloth and paper. Each type of element has its pros and cons. Regarding geometrical correctness, triangles are usually more accurate but provide less stable simulations, which is the opposite of the quadrilateral elements. A third type of element, called shell element, is used to model objects primarily 2D (understand very thin) but poorly represented by 2D elements. For example, the eye sclera requires shell elements to model its behaviour[18, 112] correctly.

Objects too thick to be represented by shell elements are considered volumic. Usually made of tetrahedral or hexahedral elements, the volumic version of triangular and quadrilateral elements. They have the same properties as their lower dimensional counterparts in convergence and representativeness.

It exists meshes called hybrid meshes that are composed of multiple types of elements. Usually, tetrahedra (resp. triangles) mesh regions with high spatial variations, and hexahedra (resp. quadrilaterals) mesh mostly flat regions. This way, one benefits from representativeness and simulation stability at the cost of code complexity.

In the context of this thesis, we focus on the simulation of homogeneous volumic meshes. For simplicity and to avoid redundancies, the following sections and subsections will only present examples and figures dealing with tetrahedral elements. The theory remains the same for all the previously presented element types.

Nodes and degrees of freedom

Each element of a mesh is composed of nodes which very often are the vertex of the element.

The Gauss node is a broader type of node that includes geometrical nodes. Gauss points are also called integration points because numerical integration is carried out in these points. In our case, each Gauss node represents three degrees of freedom.

Hence, an object $\mathbf{X}$ is represented by its Gauss points $\mathbf{X}_i$:

$$\mathbf{X} = \begin{pmatrix} \mathbf{X}_1 \\ \mathbf{X}_2 \\ \cdots \\ \mathbf{X}_n \end{pmatrix}$$

Where n is the number of degrees of freedom. We denote $\mathbf{X}$ the reference position of the object and $\mathbf{x}$ the same object subject to a displacement $\mathbf{u}$ such that $\mathbf{x} = \mathbf{X} + \mathbf{u}$. We denote $\mathbf{X}_i$, $\mathbf{x}_i$, $\mathbf{u}_i$ the i-th node of the respective vectors. In the next section, we present how we interpolate mechanical properties inside an element function called the shape function.

2.2 Isoparametric elements

In the context of elastic deformations, the finite element method is based on approximating a continuous displacement function $\hat{\mathbf{u}}$ by interpolating the function $\mathbf{u}$ on nodes $\mathbf{u}_i$. We define the nodal displacement field as the difference between corresponding degrees of freedom on the deformed and initial state $\mathbf{u}_i = \mathbf{x}_i - \mathbf{X}_i$. Thus we have:

$$\hat{\mathbf{u}} \approx \mathbf{u} = \sum_{i=1}^{ndofs} \mathcal{N}_i(\mathbf{X}) \cdot \mathbf{u}_i \tag{2.1}$$

Where $\mathcal{N}_i(\mathbf{X})$ is the shape function (interpolation function) associated with the node i. Shape functions represent the element-wise weights associated with each node and allow the computation of the field's evolution in the integration domain by interpolating the nodal quantities. To ensure stable convergence of the elements, shape functions must satisfy four essential conditions.

Shape functions must:

1. Ensure continuity and unicity of the element field. The approximated field must be

continuous and differentiable on the whole element. Given a set of nodes, this field is unique since a material point can only move in a single direction.

2. Ensure continuity and unicity on the boundary of the element. Displacement compatibility must be ensured on the boundaries. This way, neighboring elements remain neighbors and have coherent displacements.

3. Model constant strain state. All normal and shearing strains have a fixed value everywhere in the element.

4. Model rigid body motions. When a rigid translation or rotation is applied to an element, it must not generate any deformation. A rigid body motion can only introduce null deformations.

The formulation of the shape function is essential in the quality and convergence of the simulation. The usual example given to illustrate shape functions uses Lagrange's interpolation polynomials.

A particular case of shape functions is found when we use isoparametric elements to discretize the domain. Using this concept, the same shape functions can interpolate the field variables (e.g., displacement field) and the geometry (e.g., position vector field) inside an element. Thus, if we have an element e having n_e dofs we can modify (2.1) to become:

$$\mathbf{X}_e(\xi) = \sum_{i=1}^{n_e} \mathcal{N}_i(\xi) \cdot \mathbf{X}_i \tag{2.2}$$

$$\mathbf{x}_e(\xi) = \sum_{i=1}^{n_e} \mathcal{N}_i(\xi) \cdot \mathbf{x}_i \tag{2.3}$$

Where $\xi = (a, b, c)$ is called the local coordinates vector. These coordinates represent the relative position of a point within a reference element Ω_e. Therefore, using the shape functions of an isoparametric element, one can map the positions of a point in the reference element in either the initial or deformed element. Shape functions being invertible, one can also compute the inverse transformation.

When dealing with physical phenomena, one usually wants to compute the variations of physical properties. This is achieved by computing gradients and jacobians. Here, the quantity of interest is the gradient of the displacement. The displacement gradient at any point inside the element can be approximated using the displacement of the dofs and the gradient of their shape functions with respect to the material points.

$$\nabla_X \mathbf{u}_e = \sum_{i=1}^{n_e} \mathbf{u}_i \otimes \nabla_X \mathcal{N}_i \tag{2.4}$$

Where $\otimes$ is the classic tensor product (here, second order). Transforming the gradient from local to material coordinate can be achieved by taking the jacobian of the transformation given at (2.2) and (2.3). This is obtained by differentiating $\mathbf{X}_e$ and $\mathbf{x}_e$ with respect to the local coordinate ξ.

$$\mathbf{J}_e = \frac{d\mathbf{X}_e}{d\xi} = \sum_{i=1}^{n_e} \mathbf{X}_i \otimes \nabla_\xi \mathcal{N}_i \tag{2.5}$$

$$\mathbf{j}_e = \frac{d\mathbf{x}_e}{d\xi} = \sum_{i=1}^{n_e} \mathbf{x}_i \otimes \nabla_\xi \mathcal{N}_i \tag{2.6}$$

Where $\nabla_\xi \mathcal{N}_i$ is the gradient with respect to ξ of the scalar valued shape function $\mathcal{N}_i$. This gives us the gradient of the shape functions with respect to the material coordinates:

$$\nabla_{\mathbf{X}} \mathcal{N}_i = \mathbf{J}_e^{-1} \cdot \nabla_\xi \mathcal{N}_i \tag{2.7}$$

This leads us to be able to compute the displacement gradient at any point on the reference element using the following:

$$\nabla_X \mathbf{u}_e = \sum_{i=1}^{n_e} \mathbf{u}_i \otimes (\mathbf{J}_e^{-1} \cdot \nabla_\xi \mathcal{N}_i) \tag{2.8}$$

The term $\mathbf{J}_e^{-1} \cdot \nabla_\xi \mathcal{N}_i$ only depends on reference material coordinates and can therefore be precomputed for each mesh.

2.3 Tensors and quantities transformations

The problem is computing a body's deformation at rest subject to external forces. This thesis will cover nonlinear deformations since they are required to model our application cases. We start by defining the notations, and we continue by presenting the different quantities and tensors.

Deformation tensor

Let Ω^0 be the initial domain and $\mathbf{X}$ a material point of the domain. Through time Ω^0 moves and deforms in space, and this new domain is named Ω. Therefore, the material point $\mathbf{X}$ in Ω^0 is now situated at $\mathbf{x}$ in Ω.

As such, elements of length dX, area dA or volume dV in Ω^0 are transformed in dx, da et dv in Ω. The transformation from the initial domain to the deformed one is

given by:

$$dx_i = \frac{\partial x_i}{\partial X_j} dX_j$$

We observe that $\frac{\partial x_i}{\partial X_j}$ describe a tensor known as the deformation tensor noted $\mathbf{F}$. Its inverse is given by $\frac{\partial X_i}{\partial x_j}$.

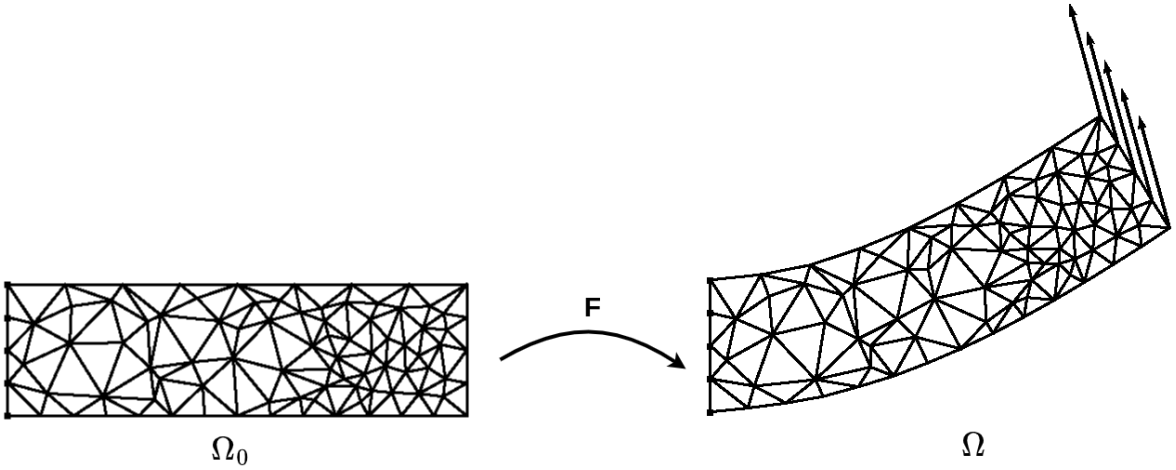


Figure 2.2: The domain Ω^0 is deformed by the external forces (green arrows) applied to its surface. The deformation gradient tensor $\mathbf{F}$ give the transformation from Ω^0 to Ω.

One usually computes displacement $\mathbf{u} = \mathbf{x} - \mathbf{X}$ and expresses the deformation gradient using the displacement gradient.

$$\boxed{\mathbf{F} = \mathbf{I} + \nabla_X \mathbf{u}} \tag{2.9}$$

Where ∇_X means that the gradient is computed with respect to X.

Volume of a deformed element

The initial state of a volume element can be written as the triple product $d\mathbf{X_1} \cdot (d\mathbf{X_2} \times d\mathbf{X_3})$ where $d\mathbf{X_i} = dX_i \mathbf{e_i}$, with $\mathbf{e_i}$ being unitary vectors. We have $d\mathbf{x}_i = \mathbf{F} \cdot d\mathbf{X_i} = \frac{\partial \mathbf{x}}{\partial X_i} dX_i$ such that :

$$d\mathbf{x}_1 \cdot (d\mathbf{x}_2 \times d\mathbf{x}_3) = \frac{\partial \mathbf{x}}{\partial X_1} \cdot (\frac{\partial \mathbf{x}}{\partial X_2} \times \frac{\partial \mathbf{x}}{\partial X_3}) dX_1 dX_2 dX_3$$

Where the right-hand side triple product is the determinant of the transformation. This way, one can write the volume change equation as:

$$d\mathbf{x}_1 \cdot (d\mathbf{x}_2 \times d\mathbf{x}_3) = J d\mathbf{X_1} \cdot (d\mathbf{X_2} \times d\mathbf{X_3})$$

Or in terms of volume:

$$\boxed{dv = det(\mathbf{F})\, dV = J dV} \tag{2.10}$$

Area of a deformed element

Since $dv = JdV$ we have

$$d\mathbf{x} \cdot d\mathbf{a} = Jd\mathbf{X} \cdot d\mathbf{A}$$

where $d\mathbf{A}$ is an oriented area element $d\mathbf{A} = \mathbf{N}dA$ and its normal $\mathbf{N}$ on the initial domain written as $d\mathbf{a} = \mathbf{n}\,da$ and $\mathbf{n}$ respectively on the deformed domain.

Given $d\mathbf{x} = \mathbf{F} \cdot d\mathbf{X}$

$$(\mathbf{F} \cdot d\mathbf{X}) \cdot d\mathbf{a} = Jd\mathbf{X} \cdot d\mathbf{A} \quad \text{or} \quad d\mathbf{X} \cdot (\mathbf{F}^T \cdot d\mathbf{a}) = Jd\mathbf{X} \cdot d\mathbf{A}$$

for all $d\mathbf{X}$. We obtain the Nanson formula that gives the relationship between the initial and deformed oriented area element.

$$\boxed{d\mathbf{a} = J(\mathbf{X})\mathbf{F}^{-T} \cdot d\mathbf{A} \quad \text{or} \quad \mathbf{n}\,da = J(\mathbf{X})\mathbf{F}^{-T} \cdot \mathbf{N}\,dA} \tag{2.11}$$

Green-Lagrange tensor

The Green-Lagrange deformation tensor usually noted $\mathbf{E}$ is defined by:

$$\boxed{\mathbf{E} = \frac{1}{2}(\mathbf{C} - \mathbf{I})} \tag{2.12}$$

Where $\mathbf{C} = \mathbf{F}^T \cdot \mathbf{F}$ is the Cauchy-Green tensor.

Using the polar decomposition theorem, a tensor $\mathbf{T}$ can be decomposed into a rotation tensor $\mathbf{R}$ and an elongation tensor $\mathbf{U}$ such that $\mathbf{R}^T\mathbf{R} = \mathbf{I}$ and $\mathbf{U}$ symmetric positive-definite. The other way around is also true with a tensor $\mathbf{V}$ that has the same properties as $\mathbf{U}$.

$$\mathbf{T} = \mathbf{R} \cdot \mathbf{U} = \mathbf{V} \cdot \mathbf{R}$$

Since $\mathbf{F}$ is invertible, $\mathbf{C}$ is symmetric and positive-definite such that it can be diagonalized.

$$\mathbf{C} = \mathbf{F}^T \cdot \mathbf{F} = \mathbf{Q} \cdot \Lambda^2 \cdot \mathbf{Q}^T$$

Where $\mathbf{Q}$ is orthogonal, Λ is a diagonal tensor containing the square roots of the eigenvalues of $\mathbf{C}$.

Let $\mathbf{U} = \mathbf{Q} \cdot \Lambda \cdot \mathbf{Q}^T$, since $\mathbf{U}^2 = \mathbf{Q} \cdot \Lambda^2 \cdot \mathbf{Q}^T = \mathbf{F}^T \cdot \mathbf{F}$ then $\mathbf{U}$ is symmetric positive-definite. Let $\mathbf{R} = \mathbf{F} \cdot \mathbf{U}^{-1}$, $\mathbf{R}$ is orthogonal by construction since $\mathbf{R}^T \cdot \mathbf{R} = \mathbf{U}^{-T} \cdot (\mathbf{F}^T \cdot \mathbf{F}) \cdot \mathbf{U}^{-1} = \mathbf{U}^{-T} \cdot \mathbf{U}^2 \cdot \mathbf{U}^{-1} = \mathbf{I}$ since $\mathbf{U}$ is symetric.

Thus, the $\mathbf{C}$ tensor can be rewritten as $\mathbf{U}^2$ and the Green-Lagrange tensor as

$$\boxed{\mathbf{E} = \frac{1}{2}(\mathbf{U}^2 - \mathbf{I})}$$

Showing the Green-Lagrange tensor is invariant by rotation since it only depends on U.

Finally, we can express the Green-Lagrange tensor using displacements.

$$\mathbf{E} = \frac{1}{2}(\mathbf{F}^T \cdot \mathbf{F} - \mathbf{I}) = \frac{1}{2}((\mathbf{I} + \nabla_X \mathbf{u})^T \cdot (\mathbf{I} + \nabla_X \mathbf{u}) - \mathbf{I})$$

Which gives

$$\mathbf{E} = \frac{1}{2}(\nabla_X \mathbf{u} + \nabla_X^T \mathbf{u} + \nabla_X \mathbf{u} \cdot \nabla_X^T \mathbf{u}) \tag{2.13}$$

We can neglect the quadratic term when dealing with small deformations and thus fall back on linear elasticity.

$$\boxed{\mathbf{E} \approx \frac{1}{2}(\nabla_X \mathbf{u} + \nabla_X^T \mathbf{u})} \tag{2.14}$$

Cauchy and Piola-Kirchhoff tensors

One of the most important tensors is the tensor of constraint σ acting on the deformed configuration Ω. The geometry being in constant variation makes it hard to evaluate such a tensor directly. To do so, we will fall back on the initial domain Ω^0 and define the transformation tensor between domains.

Let $d\mathbf{b}$ be an element of force acting on a surface element da with normal $\mathbf{n}$ on Ω. We define **Cauchy stress tensor** as :

$$\boxed{d\mathbf{b} = \sigma \cdot \mathbf{n}\ da \quad \text{or} \quad \frac{d\mathbf{b}}{da} = \sigma \cdot \mathbf{n}} \tag{2.15}$$

Writing this equation with respect to the initial domain introduces the **first Piola-Kirchhoff stress tensor**.

$$\boxed{d\mathbf{b} = \mathbf{P} \cdot \mathbf{N}\, dA \quad \text{or} \quad \frac{d\mathbf{b}}{dA} = \mathbf{P} \cdot \mathbf{N}} \tag{2.16}$$

Here we have an element of force $d\mathbf{b}$ acting on a surface element da with normal $\mathbf{n}$ on Ω. Setting everything on Ω^0 requires formulating a fictive element of force $d\mathbf{f_0} = \mathbf{F}^{-1} \cdot d\mathbf{b}$ that act on the initial area element dA. Doing this introduces the **second Piola-**

Kirchhoff stress tensor

$$d\mathbf{f_0} = \mathbf{S}\cdot\mathbf{N}\,dA \quad \text{or} \quad \frac{d\mathbf{f_0}}{dA} = \mathbf{S}\cdot\mathbf{N} \tag{2.17}$$

$$\mathbf{P} = \mathbf{F}\cdot\mathbf{S} \tag{2.18}$$

When comparing equations, we can see the similarities between σ on Ω and $\mathbf{S}$ on Ω^0. This relation can be quickly established knowing the definitions of the second Piola-Kirchhoff stress tensor 2.17 and the Nanson formulae 2.11. We have :

$$d\mathbf{b} = \mathbf{F}\cdot\mathbf{S}\cdot\mathbf{N}\,dA \qquad \text{from 2.17}$$

$$d\mathbf{b} = \sigma\cdot\mathbf{n}\,da = \sigma\cdot(J\,\mathbf{F}^{-T}\cdot(\mathbf{N}\,dA)) \qquad \text{from 2.11}$$

Subtracting the first from the second, we obtain:

$$(J\,\sigma\cdot\mathbf{F}^{-T} - \mathbf{F}\cdot\mathbf{S})\cdot(\mathbf{N}\,dA) = 0 \qquad \text{for all } \mathbf{N}\,dA$$

Thus we have :

$$\sigma = \frac{1}{J}\mathbf{F}\cdot\mathbf{S}\cdot\mathbf{F}^T \quad \text{or} \quad \mathbf{S} = J\,\mathbf{F}^{-1}\cdot\sigma\cdot\mathbf{F}^{-T} \tag{2.19}$$

which is equivalent to :

$$\sigma = \frac{1}{J}\mathbf{P}\cdot\mathbf{F}^T \quad \text{or} \quad \mathbf{P} = J\,\sigma\cdot\mathbf{F}^{-T} \tag{2.20}$$

We have defined all the essential tensors associated with the finite element method that we will use throughout this thesis. We will now discuss how we combine these different tensors to model the material properties of different objects. We do so by presenting the mathematical formulations of well-studied materials that can represent a wide range of objects.

2.4 Hyper-elastic material

As presented in equation 2.14, Green-Lagrange is a tensor that can represent linear elasticity. Linear elastic is called such because the stress-strain relation is linear. In other words, the response of an object (strain) is proportional to the external force applied (stress). This behavior is realistic under the assumption of small deformations and/or for materials such as metals before they reach their yielding point, which changes their

behavior from elastic to plastic (permanent) deformation. Therefore, linear elasticity is commonly used in fields such as civil engineering.

In this thesis, we deal with biological tissues undergoing important deformations not correctly represented by linear elasticity [66]. One common approach when modeling biological tissues is to use hyper-elastic materials. For example, Gao et al. [32] use a hyper-elastic material to model the chordae tendinae (tendons holding the heart valves). Furthermore, Roan et al. [99] use a hyper-elastic model to estimate the nonlinear material properties of liver tissue. The liver being our main subject of study, we chose to use hyper-elastic material to compute our simulations. This section will present two hyper-elastic materials, Saint-Venant-Kirchhoff and Neo-Hook, and one incompressible hyper-elastic material.

In the case of a hyper-elastic material, the second Piola-Kirchhoff tensor comes from deformation energy Ψ .

$$\mathbf{S} = \frac{\partial \Psi}{\partial \mathbf{E}} = 2\frac{\partial \Psi}{\partial \mathbf{C}}$$

We can express potential Ψ depending on $\mathbf{C}$ as a function of its invariants.

$$\begin{cases} I_1 &= tr(\mathbf{C}) \\ I_2 &= \frac{1}{2}(I_1^2 - tr(\mathbf{C}\cdot\mathbf{C})) \\ I_3 &= det(\mathbf{C}) \end{cases} \tag{2.21}$$

We can now write

$$\Psi = \Psi(I_1, I_2, I_3)$$

Thus we have :

$$\mathbf{S} = 2\frac{\partial \Psi}{\partial \mathbf{C}} = 2(\frac{\partial \Psi}{\partial I_1}\frac{\partial I_1}{\partial \mathbf{C}} + \frac{\partial \Psi}{\partial I_2}\frac{\partial I_2}{\partial \mathbf{C}} + \frac{\partial \Psi}{\partial I_3}\frac{\partial I_3}{\partial \mathbf{C}})$$

Which is equivalent to:

$$\mathbf{S} = 2(\frac{\partial \Psi}{\partial I_1}\mathbf{I} + \frac{\partial \Psi}{\partial I_2}(I_1\mathbf{I} - \mathbf{C}) + \frac{\partial \Psi}{\partial I_3}I_3\mathbf{C}^{-1}) \tag{2.22}$$

Saint-Venant-Kirchhoff

Saint-Venant-Kirchhoff is the big deformation generalization of the linear elasticity. Energy potential and second Piola-Kirchhoff is formulated as :

$$\Psi = \frac{1}{2}\lambda(tr(\mathbf{E}))^2 + \mu tr(\mathbf{E}^2)$$

$$\mathbf{S} = \frac{\partial \Psi}{\partial \mathbf{E}} = \lambda tr(\mathbf{E})\mathbf{I} + 2\mu\mathbf{E}$$

Neo-hookean

The other classic hyper-elastic model is the neo-Hookean material. Potential energy is formulated as :

$$\Psi = \frac{\mu}{2}(I_1 - 3) + \frac{\lambda}{8}(lnI_3)^2 - \frac{\mu}{2}lnI_3$$

This leads to the following:

$$\mathbf{S} = \frac{\partial \Psi}{\partial \mathbf{E}} = \mu(\mathbf{I} - \mathbf{C}^{-1}) + \frac{\lambda}{2}(lnI_3)\mathbf{C}^{-1}$$

Mooney-Rivlin incompressible

The formulation of certain hyper-elastic materials is quite complex, and therefore, for readability, we introduce new tensors:

$$\hat{\mathbf{F}} = det(\mathbf{F})^{-\frac{1}{3}}\mathbf{F} = J^{-\frac{1}{3}}\mathbf{F}$$

Which gives the Cauchy-Green tensor:

$$\hat{\mathbf{C}} = \hat{\mathbf{F}}^T \cdot \hat{\mathbf{F}} = J^{\frac{2}{3}}\hat{\mathbf{C}}$$

The two first invariant of $\hat{\mathbf{C}}$ are :

$$\begin{cases} J_1 &= tr(\hat{\mathbf{C}}) \\ J_2 &= \frac{1}{2}(J_1^2 - tr(\hat{\mathbf{C}} \cdot \hat{\mathbf{C}})) \end{cases} \tag{2.23}$$

In this model, the penalty term is formulated as $U(J) = \frac{1}{2}k(J-1)^2$ where k is the incompressibility module.

We introduce a new term that corresponds to the pressure p

$$p = -\frac{1}{3} tr(\sigma) = -\frac{1}{3} tr(\mathbf{S} \cdot \mathbf{C}) = -\frac{dU}{dJ}$$

The potential has the same formulation where we only add a penalty term which only depends on J.

$$\Psi = c_1(J_1 - 3) + c_2(J_2 - 3) + U(J) \tag{2.24}$$

$$\mathbf{S} = 2(c_1 \frac{\partial J_1}{\partial \mathbf{C}} + c_2 \frac{\partial J_2}{\partial \mathbf{C}}) - pJ\mathbf{C}^{-1} \tag{2.25}$$

$$\boxed{\mathbf{S} = 2c_1 I_3^{-\frac{1}{3}}(\mathbf{I} - \frac{1}{3} I_1 \mathbf{C}^{-1}) + 2c_2 I_3^{-\frac{2}{3}}(I_1 \mathbf{I} - \mathbf{C} - \frac{2}{3} I_2 \mathbf{C}^{-1}) - pJ\mathbf{C}^{-1}} \tag{2.26}$$

We obtain the incompressible neo-hookean model by setting $\mu = 2c_1$ and $c_2 = 0$

$$\boxed{\mathbf{S} = \mu I_3^{-\frac{1}{3}}(\mathbf{I} - \frac{1}{3} I_1 \mathbf{C}^{-1}) - pJ\mathbf{C}^{-1}} \tag{2.27}$$

2.5 Strong and weak forms

Physical simulations of deformable objects require that a set of laws be defined on the different kinematic quantities of the system. Here we define a physical system where the known quantities are: external load (forces), topology (elements), geometry (positions), material, and scalar quantities associated with it. The single unknown quantity is the displacement field and its temporal derivatives. These laws constrain the displacement field and its temporal derivative using the previously presented tensors stemming from the known quantities. In continuum mechanics, these rules are called the balance of mass, the balance of linear momentum (Cauchy's first law of motion), the balance of angular momentum (Cauchy's second law of motion) and the balance of energy (first law of thermodynamics).

We will now present the strong form of these balance equations. To simplify notation we define the usual $\dot{\mathbf{x}} = \frac{d\mathbf{x}}{dt}$ as the velocity, $\ddot{\mathbf{x}} = \frac{d\dot{\mathbf{x}}}{dt}$ as the acceleration, and $\rho(\mathbf{X}, t) = \rho_t$ the mass density at $\mathbf{X}$. The rules can be defined either in their Lagrangian formulation (deformed state) or Eulerian (initial state):

Law	Eulerian description	Lagrangian description
Balance of mass	$\dot{\rho} = \rho \nabla \cdot \dot{\mathbf{x}}$	$\rho_0 = J\rho_t$
Balance of linear momentum	$\rho\ddot{\mathbf{x}} = \rho\mathbf{b} + \nabla_x \cdot \sigma$	$\rho_0\ddot{\mathbf{x}} = \rho_0\mathbf{b} + \nabla_X \cdot \mathbf{P}$
Balance of angular momentum	$\sigma = \sigma^T$	$\mathbf{P}\mathbf{F}^T = \mathbf{F}\mathbf{P}^T$
Balance of energy	$\rho\dot{\mathbf{u}} - \rho r = \sigma \cdot \mathbf{d} - \nabla_x \mathbf{q}$	$\rho_0\dot{\mathbf{x}} - \rho_0 R = \mathbf{S} \cdot \frac{\mathrm{d}\mathbf{E}}{\mathrm{dt}} - \nabla_X \mathbf{Q}$

Where the heat flux vector $\mathbf{q}$ (respectively $\mathbf{Q}$) and the heat source r (respectively R) have been introduced for the Eulerian (respectively Lagrangian) configuration. The vector $\mathbf{b}$ represents the external load. In this thesis, we only consider the material with constant density, represented by the second Piola-Kirchhoff stress tensor, hence, assuming the conservation of mass and angular momentum. We only consider simulation without heat; hence, the balance of energy is equivalent to the balance of linear momentum, which is the only law remaining.

Weak formulation

In the finite element method, we replace the exact solution of the system $\hat{\mathbf{u}}$ by a piecewise continuous approximation $\mathbf{u}$. Inserting $\mathbf{u}$ in the balance of linear momentum yields :

$$\rho_0\ddot{\mathbf{x}} - \nabla_X \cdot \mathbf{P}(\mathbf{u}) - \rho_0\mathbf{b} = R \tag{2.28}$$

Where R is the residual from the approximation error. Using a test function $\mathbf{w}$ and integrating over the initial domain, one can minimize the error R. The resulting equation is called the weak formulation of (2.28), and residual R is minimized in a weak sense.

$$\int_{\Omega^0} \mathbf{P} : \nabla_X \mathbf{w}\, dv - \int_{\Omega^0} \rho_0\ddot{\mathbf{x}} \cdot \mathbf{w}\, dv - \int_{\Omega^0} \rho_0\mathbf{b} \cdot \mathbf{w}\, dv - \int_{\partial\Omega_T} \mathbf{t} \cdot \mathbf{w}\, da = 0 \tag{2.29}$$

subject to

$$\mathbf{u} = \mathbf{u}_d \quad \text{on} \quad \partial\Omega_D$$

where $\partial\Omega_D \cup \partial\Omega_T = \partial\Omega^0$ the boundary of Ω^0. $\mathbf{t}$ and $\mathbf{u}_d$ are respectively the natural and essential conditions.

Discretized quantities

This thesis does not deal with how things are discretized; therefore, this part will not be covered. We only give the resulting discretized weak formulation.

Internal Virtual work We define internal virtual work as the first term of the weak formulation. Its discretized formulation is written as:

$$\int_{\Omega^0} \mathbf{P} : \nabla_X \mathbf{w} \, dv = \int_{\Omega^0} \mathbf{FS} : \nabla_X \mathbf{w} \, dv = \mathbf{R}(\mathbf{u}) \cdot \mathbf{w} \tag{2.30}$$

Where $\mathbf{R}$ are the body's internal forces.

Virtual inertia We define virtual inertia as the second term of the weak formulation. Its discretized formulation is written as:

$$\int_{\Omega^0} \rho_0 \ddot{\mathbf{x}} \cdot \mathbf{w} \, dv = \mathbf{M}\ddot{\mathbf{x}} \cdot \mathbf{w} \tag{2.31}$$

Where $\mathbf{M}$ is the constant mass matrix of the system.

Virtual load work We define the virtual load work as the third and fourth terms of the weak formulation. Its discretized formulation is written as:

$$\int_{\Omega^0} \rho_0 \mathbf{b} \cdot \mathbf{w} \, dv + \int_{\partial\Omega_T} \mathbf{t} \cdot \mathbf{w} \, da = \mathbf{B} \cdot \mathbf{w} + \mathbf{T} \cdot \mathbf{w} \tag{2.32}$$

Where $\mathbf{B}$ is the body forces matrix, and $\mathbf{T}$ is the traction force matrix.

Finally, combining (2.30), (2.31), (2.32) and (2.29) we have:

$$\boxed{(\mathbf{M}\ddot{\mathbf{x}} - \mathbf{R}(\mathbf{u}) - \mathbf{B} - \mathbf{T}) \cdot \mathbf{w} = 0} \tag{2.33}$$

Often a damping matrix $\mathbf{D}$ is introduced in the equation to model the damping effect in a structure undergoing dynamic motion. Which since $\mathbf{x} = \mathbf{X} + \mathbf{u}$ and $\mathbf{w}$ are arbitrary can be written :

$$\boxed{(\mathbf{M}\ddot{\mathbf{u}} - \mathbf{D}\dot{\mathbf{u}} - \mathbf{R}(\mathbf{u}) - \mathbf{B} - \mathbf{T}) \cdot \mathbf{w} = 0} \tag{2.34}$$

subject to

$$\mathbf{u} = \mathbf{u}_d \quad \text{on} \quad \partial\Omega_D$$

This formulation leads to a system of the number of dofs equations and the number of dofs unknown that we will solve in the following sub-section.

2.6 Solving the equation

The equation (2.34) presented previously can be solved in multiple ways. A review of these methods can be found in [118].

Using an implicit scheme requires solving a system of nonlinear equations. One of the most popular technics is the Newton-Raphson (NR) iterative algorithm. In this thesis, we only deal with static simulation (no time dependency); thus, we will present the algorithm in this context, although it easily extends to dynamic simulations.

Static simulation is achieved by simply dropping all the time-dependent terms of (2.34), which gives us:

$$\mathbf{R}(\mathbf{u}) = \mathbf{b} \tag{2.35}$$

where $\mathbf{b} = \mathbf{B} + \mathbf{T}$ is the vector of the external forces applied to the body. Here the terms $\mathbf{R}(\mathbf{u})$ is nonlinear in $\mathbf{u}$ due to the formulation of the Green-Lagrange tensor $\mathbf{E}$. Thus, we need to solve iteratively until a residual quantity is minimized. In the NR algorithm, each iteration linearizes the problem before solving it and accumulating the solution into the displacement. Thus we have the taylor series development of (2.35) at $\mathbf{u}$:

$$\mathbf{R}(\mathbf{u} + d\mathbf{u}) = \mathbf{b} + \mathbf{R}(\mathbf{u}) + \mathscr{J}(\mathbf{R}(\mathbf{u})) \cdot d\mathbf{u} + \mathbf{r}(\mathbf{u}) \tag{2.36}$$

Where $\mathscr{J}(\mathbf{R}(\mathbf{u}))$ is the tangent stiffness matrix usually named $\mathbf{K}(\mathbf{u})$ and $\mathbf{r}$ the residual vector of the Taylor approximation.

Dropping the residual term of the previous equation leads to the following linear system that has to be solved for each iteration k:

$$\mathbf{K}(\mathbf{u}^k) \cdot d\mathbf{u}^k = \mathbf{b} + \mathbf{R}(\mathbf{u}^k) \tag{2.37}$$

$$\mathbf{u}^{k+1} = \mathbf{u}^k + d\mathbf{u}^k \tag{2.38}$$

The final solution is considered computed when the user-specified criterion is satisfied. Usually, the user criterion is either a threshold on the absolute/relative length of the incremental displacement or a threshold on the absolute/relative force equilibrium.

With this, we have presented the different tools needed to understand the numerical aspect of this thesis. The next chapter provides the tools to understand the different deep learning concepts used and developed during this thesis.

CHAPTER 3

DEEP LEARNING

3.1 Generalities 30
3.2 Core components of an artificial neural network 34
3.3 Architectures 36
3.4 Loss function 39
3.5 Network optimisation 40
3.6 Example of training process 43

3.1 Generalities

Artificial intelligence (AI) is currently one of the hottest buzzwords in the technology industry, and for a good reason. The last few years have seen several innovations and advancements that have previously been solely in the realm of science fiction slowly transform into reality. Due to its overuse in the media, the words "artificial intelligence" or "AI" have lost parts of their meaning. In reality, it refers to a program that can perform tasks that require human or animal intelligence. This field of research was founded on the idea that one could describe the human brain so precisely that it could be implemented on a computer [70]. As the human brain has multiple functions, AI has multiple subfields centered around tasks that, when combined, could create a general artificial intelligence. A general intelligence could address arbitrary problems and have human-like intelligence with all the ethical and existential risks. Various subfields, such as reasoning, planning, and natural language processing, have concrete applications in our daily lives. Web search engines, recommendation systems, vocal assistants, and self-driving cars are examples of artificial intelligence applications at the service of society. Most of the applications presented require dealing with billions of data points in order to start having a grasp of the tasks. Processing and inferring links between all these data required the development of a class of methods called machine learning (ML).

Machine learning is a subpart of artificial intelligence. It regroups methods that leverage data to improve task performance on abstract concepts. For example, in image processing, a cat is an abstract concept that would be almost impossible to describe as a list of instructions. However, we can use ML tools to create an AI that can detect cats in pictures with human-like accuracy, if not better. All the previously mentioned subfields use advanced ML tools to either satisfy current tasks or improve the quality of the next task. Thus, ML has many applications and is a powerful tool for representing abstract concepts. It builds a model or function based on sample data, referred to throughout this manuscript as training data or sample, to predict or decide without being explicitly programmed. We usually split ML methods into three categories corresponding to learning paradigms.

Supervised learning

Machine learning aims to build a model from a data set containing the features (inputs) and the corresponding labels (ground truth). It infers a function from labeled data that maps inputs to outputs. Each time the model is presented with an input, it produces an output evaluated against the label by a function. This function represents the problem

and how close the labels are to each other. Supervised learning includes classification, regression, and active learning.

As its name indicates, classification is classifying inputs according to an arbitrary criterion. For example, one can create a binary classifier (a model that classifies) that detects whether a cat is present in a picture.

Problems where value estimation, such as temperature or wind, falls under the hood of regression problems. They consist of fitting patterns and behaviors of data points to approximate the function that generated it.

Finally, active learning is a case of semi-supervised learning where the model can query for certain inputs to improve its training.

Unsupervised learning

Unsupervised learning involves machine learning algorithms that analyze unlabeled data to identify patterns, relationships, and structures. Unlike supervised learning, it does not rely on predefined output labels and instead extracts meaningful information directly from the data. The techniques employed in unsupervised learning include clustering, dimensionality reduction, anomaly detection, and association mining.

Clustering algorithms group similar data points together based on their properties, enabling the identification of natural clusters or subgroups.

Dimensionality reduction techniques simplify data by reducing the number of features or variables while preserving essential information.

Anomaly detection algorithms pinpoint abnormal instances, highlighting outliers or deviations from expected patterns.

Association mining algorithms uncover interesting patterns or associations in transactional data, aiding in market basket analysis or recommendation systems.

Unsupervised learning finds practical applications in customer segmentation, exploratory data analysis, and preprocessing. It provides valuable insights, reveals hidden structures, and facilitates data-driven decision-making without relying on labeled examples or predefined outcomes.

Reinforcement learning

Reinforcement learning is a subfield of machine learning that focuses on training agents to make optimal decisions in dynamic environments through interaction and feedback. In reinforcement learning, an agent learns by trial and error, receiving feedback as rewards or penalties based on its actions. The goal of reinforcement learning is to maximize

the cumulative reward over time, guiding the agent to make better decisions to achieve long-term objectives.

The agent interacts with an environment, taking action and receiving feedback through rewards or punishments. Reinforcement learning employs a reward signal, which quantifies the desirability of an agent's actions based on the environment's objective. The agent learns through exploration and exploitation, exploring new actions and exploiting known ones to maximize rewards. It has applications in various domains, including robotics, games, autonomous vehicles, and recommendation systems.

One key component of these learning paradigms is that they all use models to deal with data. So far, the term model has been swept under the rug by associating it with the term function, which is true but needs to be more explicit. In the following subsection, we discuss the historical evolution of models used in machine learning and how it led us to deep learning.

Brief history and model evolution

Machine learning relies on the concept of models. A model represents an abstract learned function using a dataset. This training aims to represent the data using the model, to either classify or extrapolate from similar data. From a historical point of view, the first model, perceptron, was invented in 1943 by McCulloch and Pitts [71] and implemented by Frank Rosenblatt in 1957 at the Cornell laboratory.

The perceptron 3.1 is a binary classifier that consists of a single cell or neuron. Its architecture and behavior are based on its biological counterpart. This mathematical

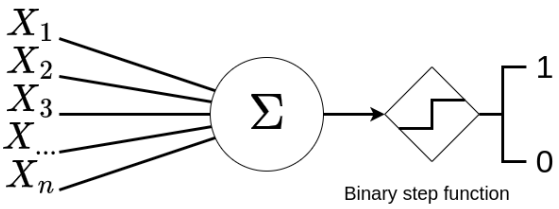


Figure 3.1: Schematic of the Perceptron

function works in three steps: First, it takes an input multiplied by a vector of learnable weights. Second, the resulting scalar value is translated up or down using an additive bias. Third, a step function of a desired threshold is applied. By construction, a perceptron has several limitations. It can only learn linearly separable functions, thus, cannot classify data that contain more than two classes. As exposed in a book titled Perceptrons [79] by Marvin Minsky and Seymour Papert, this model type cannot represent simple logic such as the XOR function. These shortcomings can be solved by setting multiple perceptrons

end-to-end and creating a multilayer perceptron (MLP). From here, the notion of deep learning appears and designates a model composed of two or more inner/hidden layers. Through the years, a plethora of network architectures have been invented in order to solve specific problems. One of the more remarkable ones is the convolutional neural network by Kunihiko Fukushima [31]. It consists of formulating the problem such that the learning parameters are the weights of the convolution masks. This leads to great recognition task results and makes the network position invariant.

With this increased complexity, one has to find the perfect set of parameters to represent its data. While it could be feasible when dealing with a handful of parameters, the task becomes almost impossible if we add multiple layers. As for every task in computer science, people quickly realized they would need an efficient algorithm that could do the job for them, which led to the creation of the backpropagation algorithm.

In 1960, Henry J. Kelley published "Gradient Theory of Optimal Flight Paths" [48], which proposed the basics of continuous backpropagation. Two years later, Stuart Dreyfus proposed a more straightforward version based only on the chain rule. At this stage, the algorithm is still long and inefficient. In 1970 a Finnish master's student going by the name Seppo Linnainmaa [61] proposed the final version that is still in use today. Unfortunately, the idea was not applied to neural networks until 1985 when Rumelhart, Williams, and Hinton [103] show that error propagation could lead to interesting distribution representation in neural networks. Finally, in 1989, Yann LeCun combined a convolutional neural network with backpropagation to recognize handwritten digits. This first use settled backpropagation as the algorithm of choice to train ANN. Modern backpropagation, or the reverse mode of automatic differentiation, is at the core of every deep-learning framework and training.

For the next ten years, most of the improvements came from the hardware with the development of graphics processing units (GPUs). Signal processing can now be highly parallelized on the GPU, leading to 1,000-fold improvements in computation times. People can now use more profound and complex models for the same training time, leading to better results. This improvement reached a limit around the year 2,000. Deeper artificial neural networks exposed the vanishing gradient problem. The upper layers were not learning features formed in the lower layers. This problem only affected artificial neural networks that use gradient-based learning methods such as backpropagation. Certain activation functions have been pointed out as the source of the problem. It concerns the functions that condense the input range into a small output range. This mapped a large area of input over an extremely small range. In these areas, significant input variations will be reduced to a small output perturbation; thus, the gradient is near zero, resulting in a vanishing gradient. One way to avoid that is to use other activation functions, such as

Rectified Linear Units (ReLU), that have a gradient of either one or zero according to the input sign.

In 2010, a breakthrough in training deep learning models was achieved by developing a technique called "Dropout" which prevents overfitting during training. In 2012, a team at the University of Toronto trained a deep learning model called AlexNet [54] to recognize objects in images and won the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) with a margin much larger than the previous state-of-the-art. This achievement demonstrated the power of deep learning for image recognition and sparked a renewed interest in the field. In 2013, Google researchers proposed an architecture called Inception, which demonstrated that using convolutional layers of different sizes makes it possible to improve the performance of deep neural networks on image classification tasks. In the following years, the field continued to grow and evolve with the development of new architectures such as ResNet, which introduced the concept of residual connections, and transformer architecture, which revolutionized the field of natural language processing.

In recent years, the field has seen significant advancements in computer vision and natural language processing, with models such as U-Net for image segmentation, BERT for natural language understanding, and GPT-3 for natural language generation. Deep learning has also been applied in other fields, such as healthcare, finance, and transportation. Currently, the field is focusing on developing new architectures and algorithms to improve the performance and interpretability of deep learning models. Overall, deep learning is a rapidly evolving field that has already had a significant impact and is likely to continue shaping the way we interact with and make decisions based on data in the future.

3.2 Core components of an artificial neural network

Definition

In artificial intelligence, an artificial neural network is an organized set of artificial neurons allowing complex problem-solving such as computer vision, text generation, and many other tasks. An important number of tunable/learnable parameters characterizes this specific class of learning algorithm. It exists a wide variety of network architectures. We will present a couple of them in this section.

Artificial neuron

We call an artificial neuron a mathematical function conceived as a model of biological neurons. These functions are the building block of every artificial neuron network. As

their biological counterpart, they can have multiple input connections, process them, and produce an output that can be spread to multiple other neurons. Each artificial neuron has one or multiple parameters called weights that can be fine-tuned to improve its response to an input signal.

Therefore, a neurons processing an input signal x composed of N channels and producing an output y can be written :

$$y = \sum_{i=1}^{N} w_i \cdot x_i = \mathbf{w} \cdot \mathbf{x} \tag{3.1}$$

This linear function values 0 for a null input signal and thus cannot represent much. For this reason, we add a bias term to the previous equation (3.3):

$$y = (\sum_{i=1}^{N} w_i \cdot x_i) + b = \mathbf{w} \cdot \mathbf{x} + b \tag{3.2}$$

The composition of linear functions only results in linear functions. To model complex problems, one has to introduce nonlinearity. This is achieved using activation functions.

Activation function

Activation functions are the backbone of artificial neural network representativity. They allow for complex responses to an input signal. The complete formulation of an artificial neuron is then written:

$$y = \Phi((\sum_{i=1}^{N} w_i \cdot x_i) + b) = \Phi(\mathbf{w} \cdot \mathbf{x} + b) \tag{3.3}$$

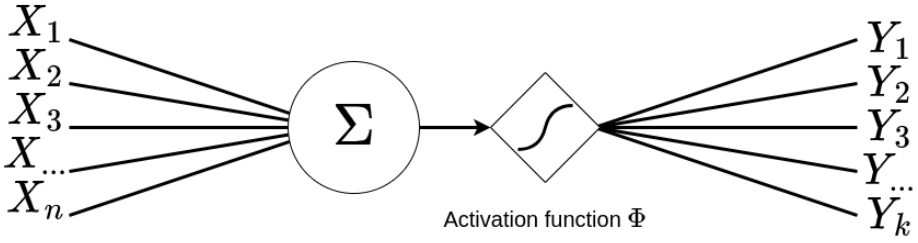


Figure 3.2: Schematic of a neuron with an activation function Φ

It has various activation functions. We will present the most used ones:

Sigmoid : Standard sigmoid function.

$$\Phi(x) = \frac{1}{1+e^{-x}} \tag{3.4}$$

$$\nabla\Phi(x) = \frac{e^x}{(1+e^x)^2} = \Phi(x)(1-\Phi(x)) \tag{3.5}$$

Simple to evaluate and easy to differentiate, this function was at the basis of all deep learning algorithms for a long time. Later it is the reason behind the vanishing gradient problem presented in the introduction 1.

Rectifier Linear Unit (ReLU) : This function is the positive part of its argument. It has proven efficient at dealing with the previously mentioned vanishing gradient problem, thus, allowing training deeper and more powerful ANN.

$$\Phi(x) = \begin{cases} x & \text{if } x > 0, \\ 0 & \text{otherwise.} \end{cases} \tag{3.6}$$

$$\nabla\Phi(x) = \begin{cases} 1 & \text{if } x > 0, \\ 0 & \text{otherwise.} \end{cases} \tag{3.7}$$

tanh : The tanh is often used in natural language processing where it plays a key role in the memory management of Long Short-Term Memory ANN. It also has the advantage of behaving like the sigmoid but preserves the sign of the inputs.

$$\Phi(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{3.8}$$

$$\nabla\Phi(x) = 1 - \Phi(x)^2 \tag{3.9}$$

These three functions are the top three choices for activation functions as they can easily be applied to all the ANN architectures. In the following subsection, we present different types of architectural paradigms.

3.3 Architectures

Neural network architectures play an important role in data processing and learning. More often than not, each architecture is problem-specific, but we can find similarities from network to network. The global architecture aggregates the standard method/layers that we will present.

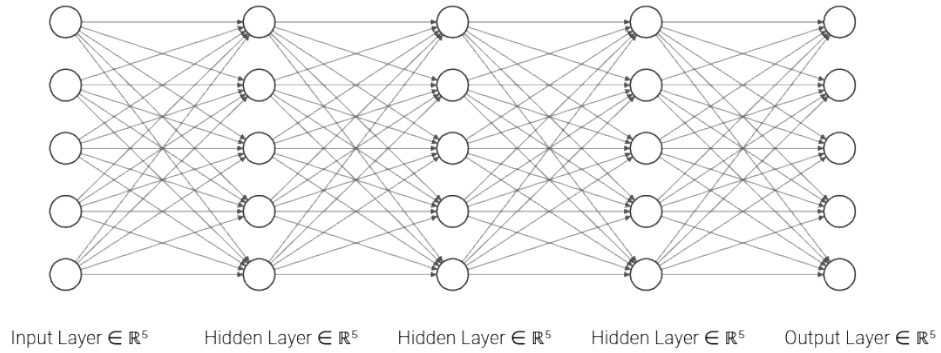


Figure 3.3: Schematic of an MLP with three hidden layers, each with five neurons. There are no restrictions on the number of neurons per layer.

Multi Layer Perceptron

The Multilayer Perceptron 3.3 or MLP is the most basic and older form of neural network. It consists of a single fully connected layer that processes data as presented in (3.3), which, when considering the whole layer, can be written as:

$$\mathbf{x}_k = \Phi_k(\mathbf{W}_k \cdot \mathbf{x}_{k-1} + \mathbf{b}_k) \tag{3.10}$$

Where $\mathbf{x}_{k-1}$ is the output of the $(k-1)^{th}$ layer and $\mathbf{x}_k, \Phi_k, \mathbf{W}_k, \mathbf{b}_k$ are respectively the output, activation function, weights and biases of the current layer. This architecture is fast and has been proven to be a universal approximator, meaning that given enough width and depth, this network can learn anything.

One main drawback is the quadratic growth in the number of parameters. This causes problems in terms of space and computation and the amount of data needed to learn without over-fitting on the dataset. A network is said to be over-fitting when the network performances on the evaluation data are terrible compared to the performance on the training data. This is often because the network has too many parameters compared to the number of training examples. Therefore, it tunes its weights and biases to perfectly fit the training data compromising the overall performances on unseen data. As an image, one can think about polynomial interpolation where you need $N+1$ data points to fit a polynomial of degree N. Reducing the data points will cause the polynomial to poorly approximate the real $N+1$ data points. One way to limit overfitting is to randomly freeze weights during the training phase, thus, artificially reducing the number of training parameters without changing the input/output sizes. The other way, which usually costs either time or money, often both, is to create a bigger dataset, thus reducing the weight ratio per sample.

Convolutional neural network

One of the most known architectures is the convolutional neural network or CNN. It has been introduced to solve image recognition tasks [31] and has since been applied to almost every class of problems. As the name suggests, a convolution happens. This operation replaces the classic agglomeration of the MLP with a more local one.

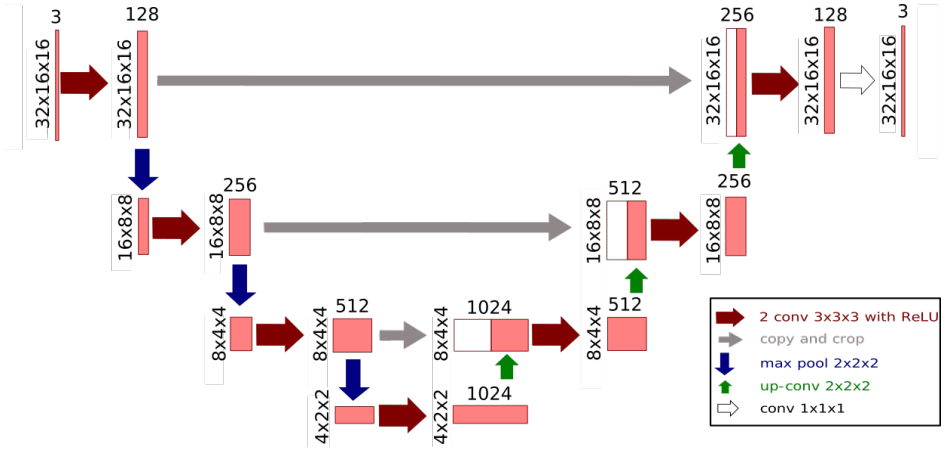


Figure 3.4: Schematic of a famous CNN known as U-net due to its shape

The weights are disposed in a learnable mask or kernel of fixed size (usually relatively small, three to eleven cells wide) and are convoluted with the image/data grid. A single convolution layer can have multiple kernels dealing with possibly different grid channels. This local agglomeration reduces the number of parameters since the same weights are used for the whole grid. Convolutional layers are considered useful to summarize the presence of features in data. Feature detection usually relies on local translation invariance, as no matter where the cat is in the picture, we must detect it. As convolutions are local, the resulting outputs are sensitive to the feature's location in the input. One way to reduce this sensitivity is to downsample the output. This down-sampling is usually the result of a pooling policy. The two main pooling methods or policies are mean and max pooling which respectively summarizes the average presence of a feature and the most activated presence of a feature. While this operation reduces the dimensions of the problem, we can augment it using the transposed convolution to go back to the desired output shape. Thus, we are not constrained to dealing with outputs that are smaller than the inputs.

All of this is well-defined because every computation of a CNN happens on a grid. This grid is also the main limitation since CNN is not usable when dealing with unstructured data such as mesh, point cloud, or even simple graphs.

Graph neural network

This relatively new technic was introduced in "A new model for learning in graph domains" [34]. The proposed architecture, now called graph neural network (GNN), uses the topology or underlying link between data points. This explicit formulation removes the need for the grid and the convolution operation while keeping the local information flow since a graph node only communicates with its direct neighbors. This local dataflow can be a problem, but different operations or structural choices exist that can bypass such constraints. To discuss these choices, we must present GNN dataflow.

First, each node of the graph gathers all the neighboring values. Second, the aggregate of data using an aggregation function (symmetric function since data order is not guaranteed). Third, the pooled message is passed through an update function.

The first one is the introduction of message passing. As the name suggests, consists of multiple calls of the first and second steps. Thus, for example, after two rounds of message passing, you obtain the transformed data from the neighbors of your direct neighbors and so on for subsequent calls. This is interesting but tends to average local data, leading to a tradeoff between data flow and accuracy.

The second introduces a fictive global node connected to all the others. Data from one side of the graph can now travel to the other side in a single step.

Both methods are not exclusive and lead to great improvement in output quality.

It is almost impossible to create an exhaustive list of architectures as the different archetypes are often put together to create a solution to a problem.

One can see the architecture as a comprehensive image of the data flow and computations that append during a prediction. This data flow represents the generic form of the function we are trying to model while we learn the value of the parameters in these computations.

Learning these parameters is achieved using a learning policy or loss function that computes for a given input data how far the prediction of the network is from reality. In the following sections, we present the notion of the loss function and how we can use its evaluation to modify the parameters of the network to learn.

3.4 Loss function

Basics of loss functions

Until here we presented the part of the process that learns (the networks). We will now discuss the learning policy also called the loss function. Loss functions measure how well

the ANN is performing for a given input. In its simpler version the function has two variables, the network output and the ground-truth or desired output. The distance between the data points is measured and is then processed to update the neural network weights. More often than not loss functions are quadratic. This is an important property of the loss function. During training the gradient of the error is used ot update the weights. Linear functions have constant gradient that do not take into account the error. When differentiated, Their quadratic cousins are proportional to the loss, which positively affect the weights variations.

The most famous loss function for regression tasks is the Mean Squared Error (MSE).

$$MSE(\mathbf{x},\mathbf{y}) = \frac{1}{N}\sum_{i=1}^{N}(x_i - y_i)^2 \tag{3.11}$$

Where $\mathbf{x}$ is the ground-truth and $\mathbf{y}$ is the network output. It is independent of the size of the elements, fully represents the data and is quadratic. For the remaining of this thesis we denote $\mathscr{L}$ an arbitrary loss that fits the context.

When optimizing for a problem one might want to optimize for multiple criteria. This can be achieved by having a final loss, a weighted sum of the different criteria. The formulation of the weights can be very simple and using constant weights or more sophisticated with softmax function to consider the variation rate of the different members.

3.5 Network optimisation

Back Propagation

Backpropagation is used to compute the derivative of the loss function with respect to the learnable parameters of the network. The loss function used for all the following demonstrations will be the MSE (3.11), but it holds for any function that respects the following two assumptions.

The first one is that the cost function can be written as an average. We need this assumption because backpropagation computes the partial derivatives for each training sample and average over a batch.

The second assumption we make about the loss is that it is written as a function of the output from the neural network. This will allow us to use the chain rule to differentiate through the network.

The backpropagation is based on four main equations. The first one computes the

error in the output layer:

$$\epsilon^N = \frac{\partial \mathscr{L}}{\partial \Phi^N} \frac{\partial \Phi^N}{\partial x^N} \tag{3.12}$$

This formulation is natural as the first term on the right-hand side computes the variation of loss as a function of the last activation function (usually passthrough when dealing with regression problems). The second term gives the variation of the activation function as a function of the output. Every part of this equation is easily computed as evaluating the network gives x^N thus a small overhead is required to compute $\frac{\partial \Phi^N}{\partial x^N}$. Given that the loss $\mathscr{L}$ is quadratic we have $\frac{\partial \mathscr{L}}{\partial \Phi^N} = \alpha \times (x^N - y)$ where α is an arbitrary constant. Thus, when computing the loss, we also compute its derivative.

The second important equation gives the error in terms of the error in the next layer.

$$\epsilon^k = ((\mathbf{W}^{k+1})^T \epsilon^{k+1}) \frac{\partial \Phi^k}{\partial x^k} \tag{3.13}$$

When we apply the transpose weight matrix, we can think intuitively of this as moving the error back through the network, looking at the error at the output of the k^{th} layer. Using (3.12) and (3.13), we can now compute the error of any layer in the ANN.

The third important equation gives us the rate of change in the loss with respect to the bias.

$$\frac{\partial \mathscr{L}}{\partial b^k} = \epsilon^k \tag{3.14}$$

Which is trivial to compute using (3.12) and (3.13).

Finally, The rate of change of the loss with respect to any weight in the network is given by:

$$\frac{\partial \mathscr{L}}{\partial \mathbf{W}^k} = \Phi^{k-1} \times \epsilon^k \tag{3.15}$$

As a consequence of the last equation, a small activation lead to slow learning.

Using these four equations, one can write the backpropagation algorithm as follows: When looking at step 3, we compute the error backward, which explains the name.

Algorithm 1: Backpropagation algorithm

1 **Feedforward** : Compute x^N from x^1 using the neural network.;
2 **Evaluate error** : $\epsilon^N = \frac{\partial \mathscr{L}}{\partial \Phi^N} \frac{\partial \Phi^N}{\partial x^N}$;
3 **Backpropagate error** : For $k \in \{N-1, ..., 2\}$ $\epsilon^k = ((\mathbf{W}^{k+1})^T \epsilon^{k+1}) \frac{\partial \Phi^k}{\partial x^k}$;
4 **Gradient** : Compute $\frac{\partial \mathscr{L}}{\partial b^k} = \epsilon^k$ and $\frac{\partial \mathscr{L}}{\partial \mathbf{W}^k} = \Phi^{k-1} \times \epsilon^k$;

At this point, we computed the sensibility of the loss according to both weights and

biases. The gradient value is essential to learn but not enough. We can improve the last step to not only compute the gradient but also use it to update the weights and biases. There are multiple optimization methods to do so, but we will present the Gradient and Adam gradient descent, which will be used in this thesis.

As the name suggests, the gradient descent algorithm uses the gradient of a function f to minimize its value. Basically, one step of the algorithm can be written as:

$$X_{n+1} = X_n - \gamma \nabla_X f(X_n) \tag{3.16}$$

Where γ is an arbitrary coefficient called the learning rate. It can be difficult to choose a γ that can fit the network response at any point in the training and for any batch of data. The Adam algorithm (short for Adaptive Moment Estimation) introduces a modified learning rate computed from the average and second moment of the gradient. The algorithm is looped over each training batch t.

Algorithm 2: Adam algorithm

$m_t = \beta_1 m_{t-1} + (1-\beta_1)\frac{\partial \mathscr{L}}{\partial \mathbf{W}}$;
$v_t = \beta_2 v_{t-1} + (1-\beta_2)(\frac{\partial \mathscr{L}}{\partial \mathbf{W}})^2$;
$M_t = \frac{m_t}{(1-\beta_1^t)}$;
$V_t = \frac{v_t}{(1-\beta_2^t)}$;
$\mathbf{W}_t = \mathbf{W}_{t-1} - \gamma \frac{M_t}{(\sqrt{v_t}+\rho)}$;

With $m_0 = 0$ and $v_0 = 0$, $\gamma = 0.001$ the learning rate, and $(\beta_1, \beta_2) = (0.9, 0.999)$. β_1, β_2 are tunable parameters called the forgetting factors for gradients and second moments of gradients, respectively, that indicate how much we trust the new data compared to the previous ones. The higher the values, the higher the smoothness of the descent, but the less the new gradient is considered.

Implementing this within the previously presented algorithm 1, we obtain the full backpropagation algorithm 3:

Algorithm 3: Optimisation algorithm

Feedforward : Compute x^N from x^1 using the neural network.;
Evaluate error : $\epsilon^N = \frac{\partial \mathscr{L}}{\partial \Phi^N}\frac{\partial \Phi^N}{\partial x^N}$;
Backpropagate error : For $k \in \{N-1, ..., 2\}$ $\epsilon^k = ((\mathbf{W}^{k+1})^T \epsilon^{k+1})\frac{\partial \Phi^k}{\partial x^k}$;
Gradient : Compute $\frac{\partial \mathscr{L}}{\partial b^k} = \epsilon^k$ and $\frac{\partial \mathscr{L}}{\partial \mathbf{W}^k} = \Phi^{k-1} \times \epsilon^k$;
Update : Adam([$\frac{\partial \mathscr{L}}{\partial b}$, $\frac{\partial \mathscr{L}}{\partial \mathbf{W}}$]);

We obtain a full training loop if we repeat this algorithm for each training sample. Data are usually grouped in batches to average error over multiple samples.

We have presented the different parts composing a deep-learning framework so far. Starting by defining what neurons are and how they are assembled to create a layer of neurons. We then explained how layers and activation functions are combined to create a deep-learning model. Further down the line, we presented how we can evaluate the output of a model using a loss function. Finally, how using the value of this evaluation, we train the model by modifying the value of each neuron.

In the next sections, we will be more specific on our use case. We present how we assemble the different building blocks and create a training dataset to train a neural network to learn the relation between external forces and the corresponding displacement field.

3.6 Example of training process

This section presents how to train a neural network to achieve a certain goal. In our case, the goal is to predict a soft-body subject's response to various external forces/loads. We have isolated three main steps (each composed of multiple sub-steps) that define the training process of a neural network.

Creating a dataset that fits the goal The name of the paragraph represents two objectives. The first one, creating a dataset which is a simple thematic collection, is quite easy to achieve. The hard part happens when we try to satisfy the second objective *fitting our goal*. One important aspect of neural networks is to keep in mind that they are good at interpolating properties but are not as good when dealing with extrapolation. For example, a network trained to classify dogs from cats with a dataset that only presents one species of white cats and black dogs will most likely perform well on different white cats and black dogs but perform poorly when dealing with black cats or white dogs. In this case, we say that the representativeness of the dataset is biased toward the white cats and black dogs specific species. But we could also lack representativeness due to data being too sparse for the problem. We want our dataset to be representative of our problem or the task we are trying to achieve.

Returning to our problem, the goal is to create a dataset that represents the deformations of an object subject to external forces. For this, we use a simulation framework [28] where we can apply external force on a deformable body and compute the corresponding deformation. The first objective is simple and relies on saving the applied force and its corresponding deformation field. The second objective, as mentioned previously, is a bit more subtle. Simulated objects usually have a purpose, a bridge carries, a propeller pushes, and a rope pulls. It would be interesting to create a dataset of *meaningful* samples

that represent the average use of simulated objects and their most significant deformations. Engineered objects like bridges and propellers are designed to respond to various stress; thus, we know by design where and how to apply external forces. Due to this lack of design, non-engineered objects like organs are a bit harder to handle. Although we have a rough estimation of where and how forces are applied thanks to surgeon knowledge, we have no exact data. This is why creating a meaningful dataset that accurately represents the response of an organ subject to external forces is difficult. We present our take on this subject in Chapter 4.1, where we use a mechanical analysis of the object coupled with expert knowledge to sample the force and deformation space. For the rest of this section, we consider the dataset $\mathscr{D} = (\mathbf{b}, \mathbf{u})$ as being a set of pairs of forces and displacements. We will now discuss how to create a model that suits the task we are trying to solve.

Building a model Building a model is very simple. You take the given predefined layers (fully connected, convolution, max pooling, etc.) and assemble them in a computational graph that defines the data flow through the model. The difficulty appears during the design phase. Both the layer picking and computational graph are defined at the same time as they have a direct impact on each other. The idea is that the data flow represents a certain knowledge about the problem. For example, it is interesting to extract local features with images. One can consider an image as a patchwork of extractable details. Therefore, directly inspecting a whole image doesn't make much sense. Small areas are subject to smaller variations and thus often represent meaningful information like a wheel, a paw, an eye, or a stop sign. One way to extract such data is to apply local operators such as convolutions and reduce them to only keep the most meaningful information. This is one of the reasons why CNN performs so well on image recognition tasks.

In our case, since the object is dense, the information of each point is instantaneously transmitted to every other point. This represents a global process perfectly modeled by dense / fully-connected layers where each neuron is connected to every other neuron. Thus, we use dense / fully-connected layers in our architecture. The architecture will be detailed in the next chapter (Chapter 4).

It is also common practice to split the data flow into multiple separated streams with the idea of specializing each stream in a subfield of the problem. Returning to the image recognition task, one could split its network into three streams, each specialized in dealing with respectively red, blue, and green information in an image. As shown in Figure 3.5, these streams can be created anywhere in the architecture and computed back to a single one anywhere. Stream agglomeration is usually a simple operation like a concatenation or a sum. This aims at helping the decision according to the activation quantities of each stream.

Finally, one last common practice people use to create new architecture is the *shortcut architecture* or *skip connection*.

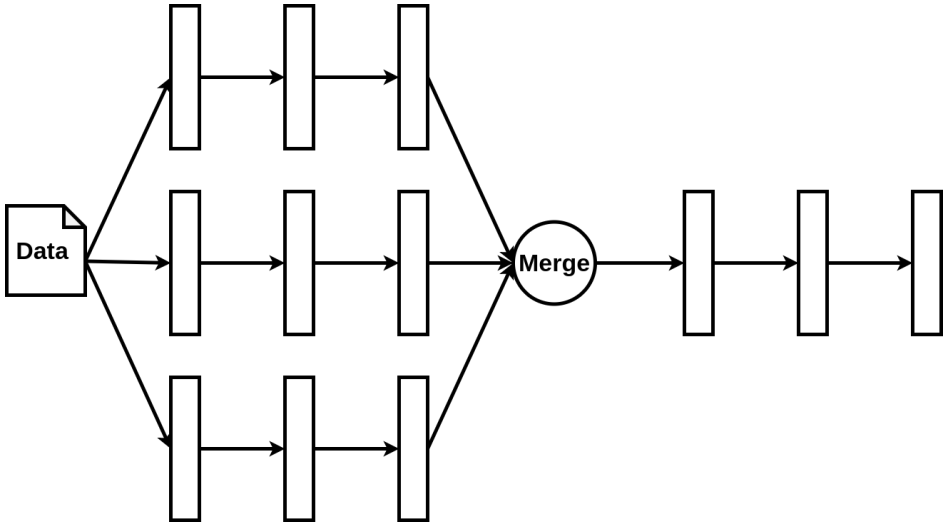


Figure 3.5: Schematic of a 3-way split of the dataflow. A merge operation is computed in the network to share the knowledge extracted by each stream.

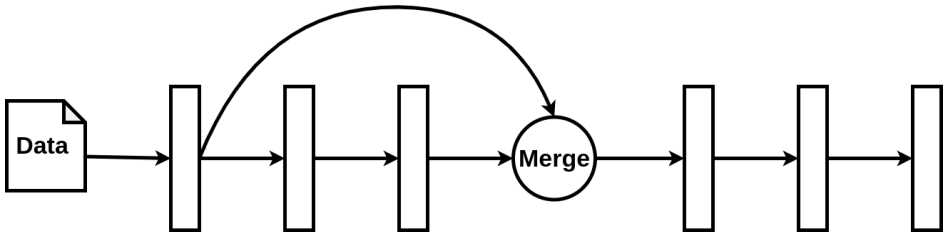


Figure 3.6: Schematic of a shortcut architecture. A merge operation is computed in the network to better propagate data and gradient potential through the network.

As shown in Figure 3.6 one takes the output of layers that appear early in the architecture and merges it with the output of a layer further down the computational graph. This usually helps by improving the gradient flow through the network, thus improving the learning. As presented in the backpropagation algorithm, we use the chain rule. During this operation, we multiply values that are often close to or less than one, and therefore, the resulting value of the gradient is very small. The skip connections help by reducing the number of multiplications in the chain rule, improving the potential learnability of the first layers of the architecture.

As for the split architecture, the merge operation is usually a sum or concatenation.

We now have constructed a good dataset with a model designed to work on our problem. Now, Let's discuss how we train the model to perform the required task.

Training a model We start this paragraph by presenting a figure (Figure 3.7) that displays a general neural network training process. Throughout this paragraph, we will explain and refer to the five steps of the schematic.

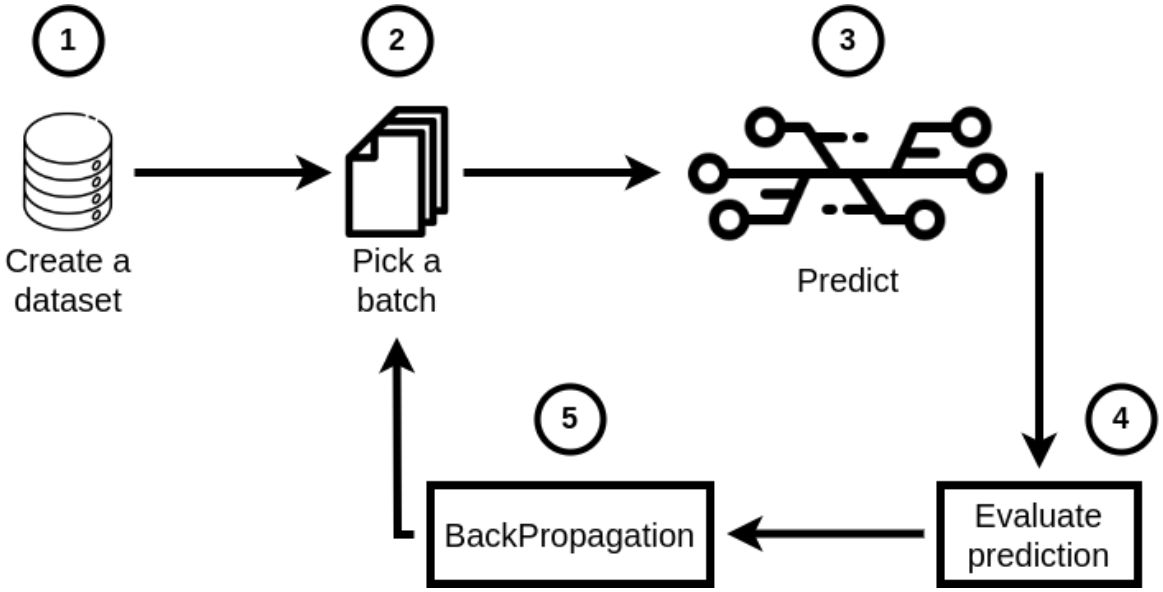


Figure 3.7: Schematic of a general training process.

So far, we have created a dataset and built the model we want to train (Step 1 in Figure 3.7). The last part remaining is the actual training. This paragraph discusses how we use the dataset to teach the chosen model the relation between inputs and outputs.

Training a neural network is based on iterating multiple times over the whole dataset. A single iteration is called an *epoch*. Thus, training is composed of multiple epochs. Each epoch consists of first splitting the dataset into multiple groups of data called batch (Step 2 in Figure 3.7), second feeding each batch to the network (Step 3 in Figure 3.7).

Initially, batches were introduced because one could not load the whole dataset, the neural network, and the gradient of each sample on a single GPU. Picking the right batch size (or, similarly, the number of batches) is an important hyperparameter in deep learning. Studies have been conducted to understand the impact of batch size on training quality. All of them lead to the same point: The performances of a network and the training duration increase when the batch size decreases [49]. Although there are no rules of thumb for selecting which batch size to use, a thousand is considered big, while a batch size of ten is small. Therefore, selecting the correct size is more of a user tradeoff between hardware memory, model capacity, and allocated training time.

Practically, what usually happens is that one will quickly verify that the model can be trained to solve the problem using an important batch size. Once the verification is passed, the model is fine-tuned with a smaller batch size to test its performance.

In our case we use batch sizes between ten and fifty.

As mentioned, a batch is fed to the network, producing a batch of predictions (Step 3 in Figure 3.7). These predictions are evaluated using a loss function such as the mean squared error (Step 4 in Figure 3.7).

Once the network output is evaluated, we can proceed to the optimization part (Step 5 in Figure 3.7). As presented in 3.5, we can now use the backpropagation algorithm and the network evaluation to compute the gradient of the error with respect to each neuron of the network. With the help of this gradient and classic optimization algorithms such as ADAM (Algorithm 2), we can update the value of each neuron. This update is the actual learning part of the algorithm and is then essential.

We repeat steps two to five in Figure 3.7 until the whole dataset is processed. Once the end of the dataset is reached, we slice it into new random batches and process the whole dataset.

The training procedure ended when any set criteria were reached. This criterion ranges from the number of epoch to more complex ones based on the value of the evaluation variation.

In short, a neural network is trained by splitting the dataset into smaller slices called batch (Steps 1 and 2 in Figure 3.7). Each batch is then fed to the network, which results in a batch of prediction (Step 3 in Figure 3.7). This batch of predictions is evaluated using the loss function (Step 4 in Figure 3.7). Using the evaluation, we proceed to run the backpropagation algorithm, which gives us the gradient used to update the neurons of the networks.

With the presentation of the training process, we have now explained all the tools needed to understand this thesis. The next chapter will present our first contribution to dataset generation and how we chose the network architecture used throughout this thesis.

CHAPTER 4

FAST AND ACCURATE DEFORMATIONS USING DEEP LEARNING

4.1 Dataset generation 55
4.2 Toward faster simulations using artificial neural networks 60

As presented, numerical methods such as the Finite Element Method [127] (FEM) are vastly used in science to solve partial differential equations (PDE) on complex domains, for which analytical solutions are not possible. Computing the nonlinear 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, and accurate solutions are sought, the combination of high-resolution meshes and nonlinear constitutive laws usually leads to computation times that are too high for certain applications.

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 very fast yet accurate solutions to nonlinear elasticity problems. Since multiple strategies have been proposed over the last decades to reduce the computation time of FEM, the following paragraphs describe some of these techniques and discuss their limits compared to our expectations.

Domain decomposition As the name suggests, Domain Decomposition [9] relies on a smart decomposition of the initial domain into multiple sub-domains. The associated (simpler) sub-problems are coupled via disjoint or overlapping boundaries. Different algorithms have been proposed to solve such problems. Sub-structuring algorithms such as BNN [65] or FETI [27] can only handle non-overlapping domain decomposition, while others like Schwarz [107] or Lions [62] methods can work on both overlapping and non-overlapping sub-domains. Combined with parallel processing, it is possible to achieve significant speedups while maintaining good convergence properties [38]. However, the optimal gain for Domain Decomposition methods is observed when dealing with very large problems (i.e. with millions of degrees of freedom) [38]. On smaller resolution meshes, or when real-time computation is needed, the speedup obtained by this approach is more limited as the computation of boundary interactions becomes predominant [38]. Since we want to favor fast or real-time computation of nonlinear elastic structures, domain decomposition techniques are not well suited. During the training phase, the ANN, the batch of data, and the gradient of the network output have to fit on the GPU memory simultaneously. With this memory space limitation, we only consider objects up to 15,000 degrees of freedom which makes this method unsuited for our application cases.

Proper Orthogonal Decomposition Another approach to improve computation times is Model Order Reduction. Among this class of methods, Proper Orthogonal Decomposi-

tion (POD) [14] has proved to perform well on a wide variety of problems ranging from fluid dynamics [5, 74] , soft robotics [35], aeronautics [63], optimal design [86] and many others. This approach aims to reduce the number of Degrees of Freedom (DOFs) by analyzing deformation modes and discarding the least significant ones. These modes are obtained via a set of data of P samples that are put together as a matrix $\mathbf{Q}$ of size $N \times P$ where N is the number of DOFs. The eigenvalues and their associated eigenvectors are then computed from the self-adjoint matrix $\mathbf{Q}\mathbf{Q}^T$. When the displacement field is well-behaved, the magnitude of the eigenvalues decreases quickly, showing that the object is mostly characterized by the first few modes of deformation (i.e. the eigenvalues with the highest norm). Follows a step where the lowest eigenvalues (usually anything 10^{-8} times shorter than the magnitude of the first one) are removed. The simulation is computed in the reduced space, thus speeding up the solving of the system. This method relies on a trade-off between accuracy and speed. Multiple factors impact the quality of the simulation and the gain, such as how magnitudes of eigenvalues decrease and the magnitude of the cutoff value. Although it introduces errors in the simulation, the computed reduced space can be used to solve "similar" problems with slight changes in the boundary conditions or model parameters [64, 88] . This allows to reuse the reduced model for a variety of scenarios, therefore reducing the overall computational cost of the method. Artificial neural networks have proven to be resilient to geometry variations; Pfeiffer *et al.* [91] trained a neural network on randomly generated meshes to predict displacement fields. We will demonstrate in this thesis that the presented approach is at least as fast as the POD while providing more flexibility to model parameter variations.

GPU-acceleration Many publications have addressed the problem of computational performance for FE simulations through GPU-accelerated approaches. A parallel implementation can speed up the computation of the local, element-wise, stiffness matrices. These methods aim to solve the global linear system associated with the linearization of the problem.

In the case of an iterative method, such as the conjugate gradient, most of the gain can be achieved by improving the sparse matrix-vector operations. Multiple methods have been explored to implement efficiently these operations on the GPU [3]. Mueller *et al.* [81], or Martínez-Frutos *et al.* [67] use the fact that conjugate gradient iterations can be performed without explicitly assembling the whole matrix. This has the advantage of reducing the memory bandwidth while being fast and stable. For example, Allard *et al.* [2] proposed a cache optimization process for a co-rotated formulation of a linear elastic model. With this method, a mesh composed of 20,000 tetrahedra can be simulated in about 2 ms [2]. However, This approach is too specific to be applied to other materials,

such as hyperelastic ones. In the case of direct solvers, the global matrix assembly is required to compute the decomposition or factorization of the system.

Dziekonski *et al.* [24] and Mueller *et al.* [82] proposed to assemble the matrix directly on the GPU, thus reducing memory transfer and speeding up the assembly. In this case, the method also requires a model-specific algorithm and cannot provide a good combination of heterogeneous CPU/GPU simulations. One could also optimize the construction of the matrices, using very efficient implementation such as the one provided by the Eigen [36] library.

Despite some limitations, GPU-based FEM algorithms provide fast and accurate results. However, as for the POD, GPU-based FEM falls under task-specific algorithms. It de facto encounters the same limitations when using different hyper-elastic laws to represent complex objects.

Physics Informed Neural Network In physics, neural networks have mainly been used to find the relation between input and output data. This approach has multiple obstacles, such as data quantity and data quality. In this context, data are generated using a simulator or acquired from real-world sensors (camera, Pitot tube, heat sensor). Both perspectives have their advantages and drawbacks. A simulation is repeatable and easy to set up but only precise to the user's domain knowledge. Micro variation in domain properties, mesh quality, and boundary conditions definitions are examples of factors that lead to imprecision in the results. On the other hand, acquisitions give a better representation of the real world but are sensitive to the experimental setup and probe quality.

Physical phenomena are represented using partial differential equations (PDE) as presented in 2.5. These sets of equations provide all the necessary knowledge to solve the problem and are an excellent and noise-free source of information.

It exists two main ways of using the PDE to learn. The first one, usually called inductive biases, focuses on designing a neural network architecture that embeds the knowledge of the predictive task. We briefly present examples of widely used architectures. The convolutional neural networks [31], which by construction respect the group of symmetries and patterns found in images. Graph neural networks [33] that are designed to work on unstructured data or data with changing structure. Fourier Neural operators [60] work on frequency transformations and are translations independent. On the specific subject of soft-body mechanics, multiple networks have been designed to satisfy Dirichlet [56, 109], Neumann [72, 108] and Robin boundary conditions [55].

These designs are of interest and, most of the time, complementary to other learning methods. They provide tools that enforce constraints on the results without using com-

plex loss functions. Most of the development of these new architectures has been done in the last five years and is thus an active field of research.

The second approach relies on imposing constraints using the loss function, also called learning bias. This method can be seen as a use case of multi-task learning. The network is trained to predict quantities that satisfy physical properties such as volume, the balance of forces, energy conservation, etc. Examples of this approach are deep Galerkin method [110] and Physics Informed Neural Network (PINN) [17, 94, 126].

Soft penalties offer great flexibility when incorporating domain-specific knowledge into the network. For example, one can enforce the covariance of the prediction of a generative adversarial network [119] to better represent the dataset. It is also used to deal with contacts [92], or even to conserve Lyapunov stability [26].

As said previously, these methods are often used conjointly, where data-driven learning is mixed with a physics-informed paradigm and inductive bias to create a performant network. For example, one can use a low-fidelity model coupled with a multi-fidelity strategy using observational and learning biases (data and physics-informed loss function) to facilitate complex systems learning as done in [30]. Another possibility is to embed a neural network into a traditional numerical method, such as the finite element. This approach was applied to solve problems in multiple fields such as dynamical nonlinear systems [98], subsurface mechanics [58] and more [122].

Results on PINN [45, 50, 122] demonstrate that they are particularly effective in solving ill-posed and inverse problems, whereas the forward and the well-posed problem is better solved using grid-based solvers.

When dealing with incomplete models, recent research [95] proved that finding meaningful solutions to ill-posed problems is possible. The examples usually include problematic inverse and forward problems with no initial boundary conditions or missing parameters in the PDE. Noisy data is also well-supported by PINN with the introduction of Bayesian PINN (B-PINN) [121] to quantify uncertainty.

Uncertainty quantification is important when dealing with the evolution of multiscale and multi-physics systems [68, 113]. Physics-informed learning has proven to work with three sources of uncertainty: physics, data, and the learning model.

The first source refers to an inherently stochastic problem described using stochastic PDE (SPDE). As an example, in [47], the expectation of the energy functional of the PDE over the stochastic variable is used to train a neural network parametrizing the solution of an elliptic SPDE.

The second source of uncertainty comes from noisy data acquisition. The article [121] shows that it is possible to compute an uncertainty bound of the same order of magnitude as the error increases with the noise in the data. Although the question remains on how

to set a systematic prior when using B-PINNS.

The third source of noise refers to limitations in the learning model, which is hard to quantify. Training their network with a probabilistic supervised learning procedure [117] has been able to map the source term and the domain geometry of a PDE to the solution and the uncertainty.

Having a neural network that is resilient to noisy data is important in the context of this thesis. Each patient's history and habits present variations in ligament locations, veins or arteries rigidity, and organ composition. Since preventive surgeries are rarely practiced, patients undergoing surgery are often pathological. Ill organs have a higher variation in mechanical properties due to tissue composition being modified with inflammations, necrosis, clogs, and others. Thus, training a network on data computed considering an average healthy organ can be considered trained on noisy data (noise from simulation and mechanical parameters).

Recently, Moya et al. [80] trained a network to predict the physics of a fluid with a fixed viscosity. Then, using physics-informed learning and a video of a fluid in motion, they show that it is possible to quickly retrain the network to adapt to the new viscosity. Setting this result in our context could lead to being able to train a network on standard organs and quickly retrain at the beginning of surgery using video footage to adapt its parameters. This result interests us since we can reduce the range of variable mechanical parameters in the dataset. Thus, we can train our ANN on sparser models with simpler training policies and only train on a short video feed before the surgery to create a model that produces patient-specific predictions.

Data-driven learning process Data-driven methods are a class of machine learning techniques that can deal with a wide variety of physics-based problems ranging from radio-frequency or microwave modeling [125] to fluid mechanics [53] and solid mechanics [75].

Data are generated from real-world acquisitions [100] or can be computer-generated [16].

Once the data are processed or generated, the inputs are fed to the network, and the outputs are compared to the ground truth via a loss function (usually the MSE). The error gradient is then computed, and finally, the weights and biases of the artificial neural network are updated using an optimization algorithm.

Such an approach is helpful to approximate problems that do not have a mathematical model, such as image animation [43] or problems that do not fit into the PINN framework.

Prior work has been done on simulating hyper-elastic materials using data-driven approaches. Cloth deformation is an important subject in physics-based animation, and Wang *et al.* [115] used measured data to build a piece-wise bending and stretching model to compute the nonlinear dynamic of cloth material. Xu *et al.* [120] following the idea

of Kim *et al.* [51] proposed a technique of mix and match to generate meshes that fit the desired pose. More recently, Santestaben *et al.* [105] used a neural network to generate nonlinear garment wrinkles. For applications involving the deformation of elastic solids, Brunet *et al.* [13] proposed a method to compute hyper-elastic volume deformation of a liver in real-time from a set of Dirichlet boundary conditions. Finally, Meister *et al.* [73] and Mendizabal *et al.* [75] proposed neural networks able to predict the deformation of an elastic structure given variable external forces. The latter approximate static deformations using a Convolutional Neural Network (CNN). This method requires immersing the object within a topological grid to perform convolutions. Neumann boundary conditions are transferred to the grid through mapping. The network then computed the deformation on the grid and applied it to the object using inverse mapping. These methods provide a fast estimation of the displacement field. Their principal drawbacks are related to the data generation process and the limited accuracy of the solution. The number of simulations required to train the network (several hundred simulations per boundary node) becomes time-consuming when the grid resolution is increased, as does the training time. The prediction accuracy is in the range $[10^{-4}; 10^{-3}]$ of the object scale, which is acceptable for some applications but not sufficient for others.

In this chapter, the first section presents how we build our dataset following the idea proposed in [84]. The second section focuses on our choice of ANN used throughout this thesis.

4.1 Dataset generation

The creation of a dataset is the first step of the training of an artificial neural network. This creation is achieved by collecting data but also generating them. For our application, we want to predict an object's deformation subject to an external load. Thus, our goal is to create a dataset that represents the deformations of an object subject to external forces. For this, we use a simulation framework [28] where we can apply external force on a deformable body and compute the corresponding deformation. Creating a dataset is simple and relies on saving the applied force and its corresponding deformation field.

Complexity arises when we want to consider the functionality of an object. It would be interesting to create a dataset of *meaningful* samples representing the average use of simulated objects and their most significant deformations.

Engineered objects are easier to deal with since their design is purpose-oriented. Therefore, we have a good idea of where and how to apply load. This lack of design makes non-engineered objects like organs harder to handle. Thanks to acquired knowledge through

discussions with surgeons and video feedback, we can estimate the zone of interest and type of force applied to organs during surgeries. However, we have no exact data, and thus the sampling of the space of force remains complicated. This is why creating a meaningful dataset that accurately represents the response of an organ subject to external forces is challenging. In this section, we present our take on the subject where we use an eigenvalues analysis of the mechanical properties of the object to sample the force and deformation space. For the rest of this thesis, we consider the dataset $\mathscr{D} = (\mathbf{b}, \mathbf{u})$ as being a set of pairs of forces and displacements.

Two main difficulties arise when creating a dataset that represents an object's deformation.

First difficulty: Number of parameters. Applying an external load to an object requires affecting force values to each degree of freedom of the mesh. In the context of this thesis, we are dealing with meshes composed of two to five thousand nodes, hence, six to fifteen degrees of freedom. One solution to deal with the problem of dimensionality is to apply forces that are constant over the whole object since we only have to consider the X, Y, Z component of the vector field, thus reducing the number of parameters to three. Although applying such force is easy, it rarely represents real scenarios or scenarios of interest. When we want to apply a more complex load to an object, the task quickly becomes a burden. Defining the dofs-wise value over the entire mesh is already problematic at our scale, even though we deal with relatively small meshes compared to other scientific fields. One must define the value of multiple thousand coefficients to generate a single displacement when multiple thousands of displacements must be computed to train an ANN. Therefore, one has to rely on a random sampling and naive approach to generate a force vector and hope it provides interesting deformations, which is neither time nor energy efficient. In this section, we present our take on reducing the dimensionality of the force generation problem while managing to generate a complex force vector field.

Second difficulty: Relation between the load and displacement. When an object is subject to an external load, its response is given by its material and geometry. While the material response is direct and defined by the user, the geometry response can be very complex, even for simple-looking geometry. For example, one can take a rope and apply a twisting force. The rope will start to twist to some point, where it starts to roll on itself, forming what is known as plectoneme. This type of response is unpredictable by the user; therefore, forces resulting in such a response are hard to produce when creating a dataset. While in our case, we do not have such an extreme case, the Dirichlet's boundary condi-

tions of a liver are patient-specific. They can significantly influence the mechanics of the organ. Therefore, producing a standard method to generate a dataset for any given liver is challenging.

The number of nodes, material, and geometry are constitutive of an object and, therefore, cannot be modified without impacting its physics. While we cannot modify them, they embed much information about how an object responds to stress. We can extract information using our knowledge of the mechanics of deformable bodies. In particular, we would like to have two things.

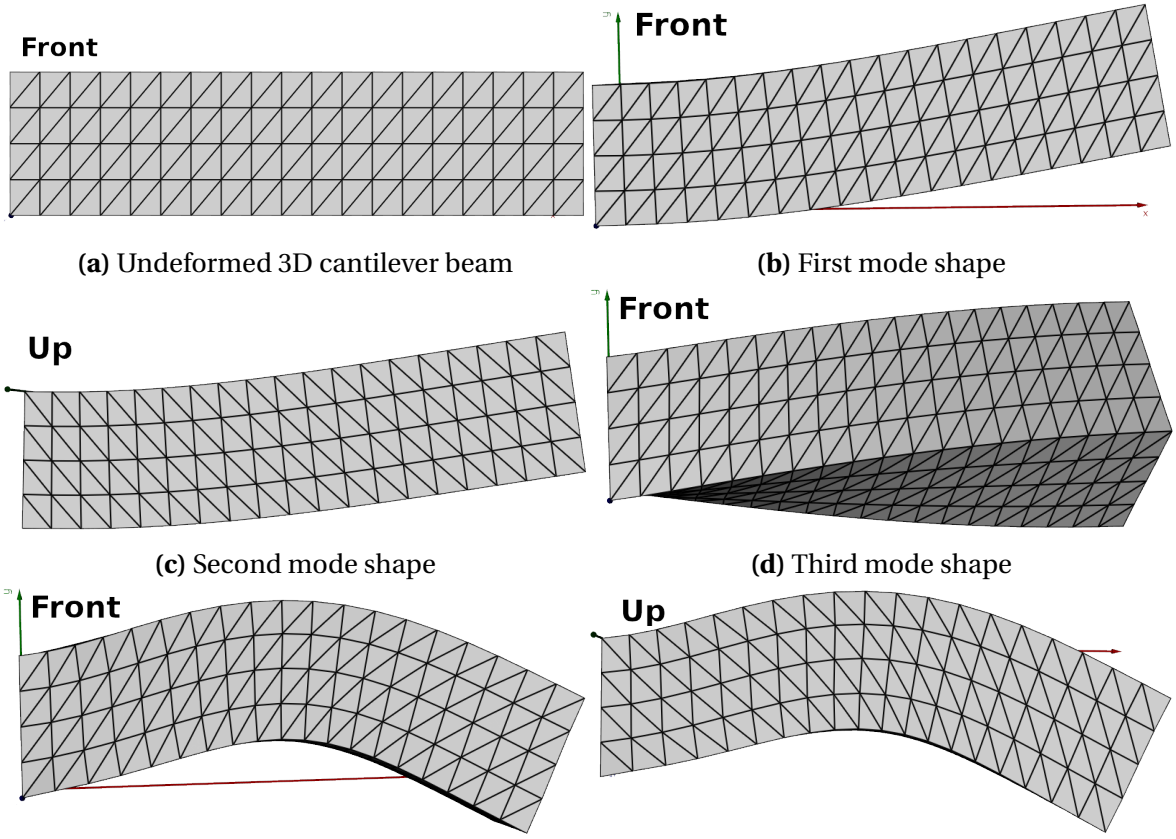
The first is a rough estimation of the displacement generated from stress to know if it is worth computing the full deformation.

The second one is to generate interesting/complex force vectors easily.

The first request corresponds to creating a transformation that allows us to jump from the force to the deformation space. This is achieved by using equation 2.37 where the tangent stiffness matrix represents the transformation from the deformation space ($d\mathbf{u}$) to the force space $\mathbf{b}$. Since we want a rough approximation, we can consider that $\mathbf{K}(\mathbf{0})$ will be our approximation transformation. The nonlinear material response depends, among other factors, on the current deformation state of the object. Thus, using $\mathbf{u} = \mathbf{0}$, we do not bias $\mathbf{K}$ toward any specific response, although for specific use-cases, one could use $\mathbf{K}(\mathbf{u} \neq \mathbf{0})$.

We will use our answer to the first to satisfy the second request and try to extract information from the equation 2.37. First, let us take a step back and present how engineers have been studying systems like equation 2.37 and similar systems.

One common practice in engineering is to study the natural frequencies of vibration. Called eigenvalues analysis, this practice is the most frequent system analysis method. Using this method, the result is double since it gives us both the natural frequencies and shapes of vibrations. These shapes of vibration are called undamped free vibration response of the structure. The total number of frequencies or eigenvalues equals the total number of degrees of freedom in the model. Each eigenvalue/frequency has a corresponding eigenvector/mode shape. Eigenvalues analysis usually only focuses on the first few eigenvalues of the model. This is primarily because the finite element model approximates the real shape. Therefore, it correctly models the lowest spatial frequencies while dropping the higher spatial frequencies. This significantly impacts the values of the eigenvalues, which are inaccurate for the high frequencies. In other words, the first modes are the most common deformations of the object, as shown in Figure 4.1. Furthermore, the mode shapes are normalized to the maximum displacement of the structure.



(a) Undeformed 3D cantilever beam **(b)** First mode shape

(c) Second mode shape **(d)** Third mode shape

(e) Fourth mode shape **(f)** Fifth mode shape

Figure 4.1: A cantilever beam Neo-Hookean material attached by the left face (a) and its five first modes shape (b,c,d,e,f). The complexity of the deformation increases with the index of the mode.

From this, we have two interesting takes. First, we usually analyze a few modes to understand the object's behavior. Most of the information is concentrated in a small portion of the problem which could help us reduce the complexity of the force generation by explicitly targeting this region. The second is that since the mode shapes are normalized, we can interact with them using normalized coefficients.

Going back to our problem we have equation 2.35 where $\mathbf{u} = \mathbf{0}$. We perform an eigen decomposition on $\mathbf{K}(\mathbf{0})$, therefore equation 2.35 becomes :

$$\mathbf{K}(\mathbf{0})d\mathbf{u} = \mathbf{b} + \mathbf{R}(\mathbf{0}) \tag{4.1}$$

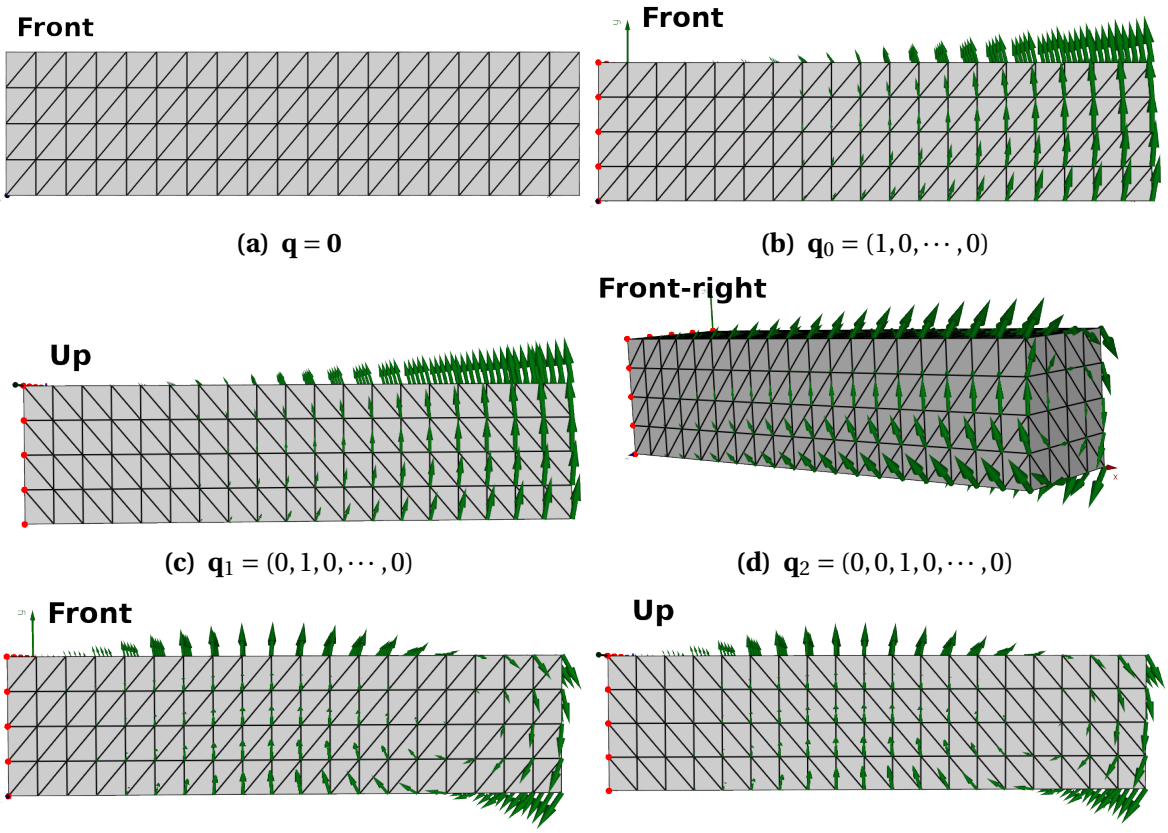
$$\mathbf{K}(\mathbf{0})d\mathbf{u} = \mathbf{b} \quad \text{, no strain at rest,} \quad \mathbf{R}(\mathbf{0}) = \mathbf{0} \tag{4.2}$$

$$\Phi\Lambda\Phi^T d\mathbf{u} = \mathbf{b} \tag{4.3}$$

$$\Phi(\Lambda\Phi^T d\mathbf{u}) = \Phi\mathbf{q} = \mathbf{b} \tag{4.4}$$

We obtain a vector $\mathbf{q}$ that we call control since it controls how much each mode shape

appears in the result. Now we observe that $\Phi\mathbf{q}$ is homogenous to a force. Since Φ is the eigenvector matrix, we can exploit the mode shape to generate forces. Furthermore, as mentioned earlier, we usually only use the first few modes, significantly reducing the complexity of the force generation. In other words, we only need a handful of coefficients to generate complex forces, as shown in Figure 4.2.



(a) $\mathbf{q} = \mathbf{0}$ **(b)** $\mathbf{q}_0 = (1, 0, \cdots, 0)$

(c) $\mathbf{q}_1 = (0, 1, 0, \cdots, 0)$ **(d)** $\mathbf{q}_2 = (0, 0, 1, 0, \cdots, 0)$

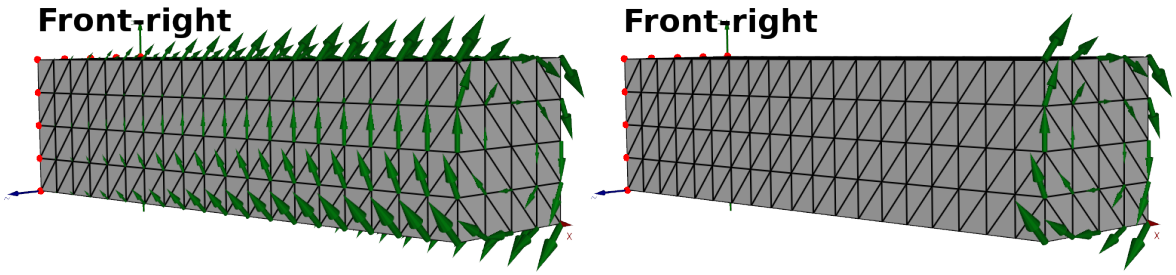
(e) $\mathbf{q}_3 = (0, 0, 0, 1, 0, \cdots, 0)$ **(f)** $\mathbf{q}_4 = (0, 0, 0, 0, 1, 0, \cdots, 0)$

Figure 4.2: Examples of forces we can generate using only the first five modes shape of the cantilever beam presented in Figure 4.1. Here the $\mathbf{q}_i$ are presented as one-hot vectors, but any linear combination of the $\mathbf{q}_i$ can be used to generate a force. We can see the similarities between the forces computed using the vector $\mathbf{q}_i$ and the i-th mode shape in Figure 4.1.

Using the $\Phi\mathbf{q}$ product, we can now generate complex forces using only a handful of coefficients. We call $\bar{\mathbf{q}}$ the selected subset of indices of $\mathbf{q}$. We can now interpolate the coefficient of $\bar{\mathbf{q}}$ between -1 and 1 since we recall that the mode shapes are normalized to the maximum displacement of the structure.

One observation remains. The generated force vectors are dense. While they present interesting patterns, they remain dense. Thankfully, this can be easily solved by applying a binary mask on the force. The mask coefficients are one on the nodes where we want to

apply a force and zero elsewhere. For example if we consider a twisting force applied on the tip of the beam, then the force given by $\mathbf{q}_2$ in Figure 4.2 becomes the one presented in Figure 4.3 :



(a) Force generated using $\bar{\mathbf{q}} = (0,0,1,0,0)$

(b) $\mathbf{q}$ multiplied by a mask which values at one for every node of the tip, zero elsewhere.

Figure 4.3: Example of mask application to create sparser forces. Local forces are usually more representative of object manipulation if we do not consider gravity.

Finally, we can rewrite equation 2.37 as :

$$\mathbf{K}(\mathbf{u}^k + d\mathbf{u}^k) \cdot d\mathbf{u}^{k+1} = m(\Phi\mathbf{q}) + \mathbf{R}(\mathbf{u}^k) \tag{4.5}$$

Where $m(\mathbf{x}) = x \cdot m$ is a function that computes the Hadamard product between a mask m and a vector of matching size x .

To conclude this section, we created a process that allows us to compute complex forces using only a handful of coefficients. This is achieved by using the eigenvalues analysis of the system in equation 2.37 for $\mathbf{u} = \mathbf{0}$. The generated forces are automatically scaled by the matrix of eigenvectors Φ , which allows us to sample the space between minus one and one, reducing the complexity of the generation one step further. Furthermore, we can use a binary mask to apply the force on zones of interest instead of the whole object.

Now that we have created a dataset, we can discuss the model we will train. In the following section, we weigh the pros and cons of different model architectures and explain our choice of architecture for all the following works.

4.2 Toward faster simulations using artificial neural networks

To effectively solve the problem described in Section 2.5 in real-time, we suggest using deep neural networks in a direct FEM-based manner, as demonstrated in [19, 75, 84]. We

plan to train the selected network with FEM-generated data, which we can thoroughly control.

Selecting the appropriate network architecture is crucial and should consider the target application, available computational resources (both hardware and time limitations), and the structure and amount of training data. For instance, in augmented surgery, the time between the preoperative CT scan (which enables reconstructing the organ's geometry) and the intervention is often less than 12 hours. This short period implies that data generation and network training must be accomplished in less than half a day. The stringent time constraint and the desired accuracy for the virtual model make it challenging to identify a suitable network architecture.

This section introduces two distinct architectures for physical simulation: multilayers perceptron (MLP) and convolutional neural networks (CNN). Specifically, we present the U-Mesh framework, a specific type of CNN. We evaluate the performance of these networks in two different scenarios where we first consider a square section beam and, secondly, a liver.

The U-Mesh framework In the following paragraphs, we will introduce the U-Mesh framework [75], a specific type of CNN used for displacement field estimation. This framework is built on a U-Net architecture [101], which gets its name from its U-shaped structure, resulting from a sequence of max-pooling and up-sampling operations. It has an autoencoder architecture (see Figure 4.4) comprising an encoding path that transforms the input into a low-dimensional space and a decoding path that expands it back to its original size. We chose this framework due to its similarity to model order reduction techniques, specifically the Proper Orthogonal Decomposition (POD). The number of singular vectors retained in POD can be related to the depth of the U-Net, which monitors the quality of the predicted displacement [75].

Regular grid structure CNNs have a significant limitation in that they are only applicable to structured matrices. These networks are designed to extract features from matrices with spatially connected values, such as images. The input needs to be encoded in a grid structure, as convolutional filters have a spatial representation of the information. Therefore, to use CNNs for physical quantities of interest, we must encode them on regular grids.

To overcome this limitation, we will use an *immersed-boundary method* to create a regular hexahedral mesh directly from surface geometry. Regular hexahedral meshes offer several advantages, including better convergence properties and stability [4]. The surface mesh is embedded into a regular grid, and only the cells inside the surface or overlapping

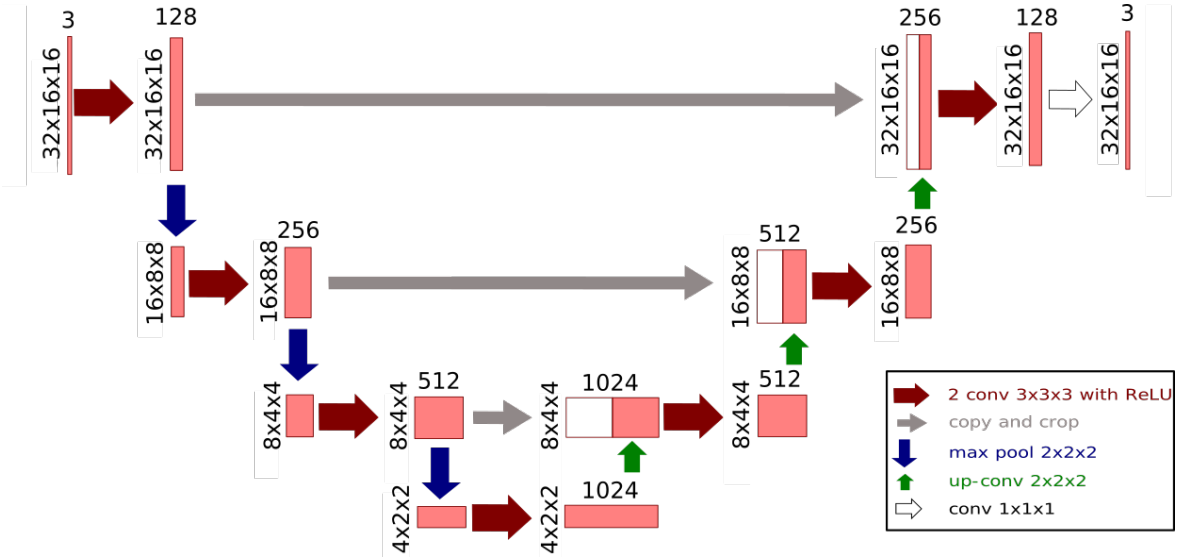


Figure 4.4: Chosen network architecture with three steps and 128 channels in the first layer for an input grid with $28 \times 12 \times 12$ nodes Zero-padding is added in each spatial direction to avoid any loss of information on edge nodes, which leads to a $32 \times 16 \times 16$ shaped tensor At each step of the encoding phase, two padded $3 \times 3 \times 3$ convolutions are applied, followed by a ReLu activation function and a $2 \times 2 \times 2$ max pooling operation (which halves the spatial dimension) Each feature map doubles the number of channels In the decoding phase, upsampling $2 \times 2 \times 2$ transposed convolutions are applied, followed by two padded $3 \times 3 \times 3$ convolutions (doubles spatial dimension and halves the number of channels) The output of each layer is concatenated along matching levels from the encoding to the decoding path.

it are retained to build a sparse grid that will be used as a finite-element mesh. The volume integration of displacement on boundary cells is handled using recursive subdivision, as done in [25].

MLPs do not have the same requirement and can potentially work with any unstructured mesh. However, we will use the same structured meshes for both network architectures presented in this study for clarity and fair comparison.

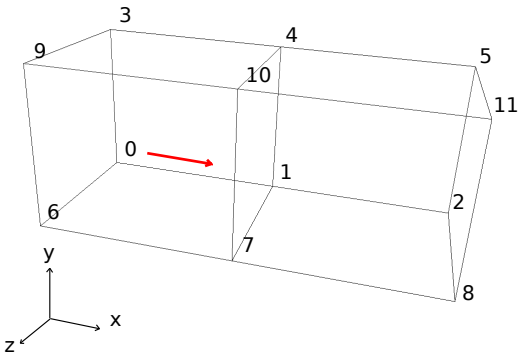
4.2.1 Case study - Cantilever beam

In this section, we consider a cantilever beam of size $1 \times 0.25 \times 0.25\, m^3$ modeled as a Neo-Hookean material from Equation (2.27) with a Young's modulus of 5,000 Pa and a Poisson's ratio of 0.45. It is discretized with 304 linear hexahedral (H8) elements.

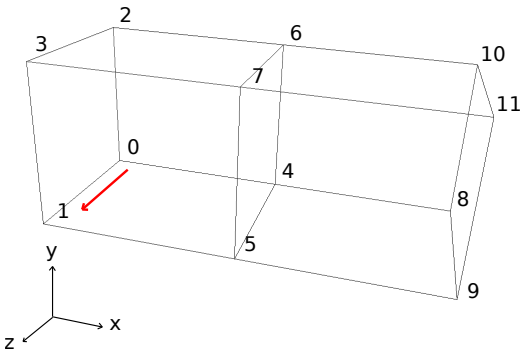
The effect of node ordering

As demonstrated in [19], ordering degrees of freedom in a FEM mesh is critical in training a U-Net. This is because convolution filters extract spatial information from input ten-

sors, and neighboring nodes in the input tensors must be spatially adjacent in the FEM mesh. To reduce the jumps between nodes in a convolution operation, we started node numbering in the smaller dimension axis, as shown in Figure 4.5b, instead of the larger dimension axis, as shown in Figure 4.5a.



(a) Configuration 1: The node numbering starts in the object's principal dimension (here x). Nodes 0, 3, 6, and 9 are fixed. The network predictions are not convincing.



(b) Configuration 2: The node numbering does not start in the object's principal dimension. Here it starts in z. Nodes 0, 1, 2, and 3 (which appear to be consecutive in the input tensor) are fixed. The network predictions are very good.

Figure 4.5: The effect of node ordering on prediction accuracy. For clarity, a beam with only 12 degrees of freedom is represented here.

In this scenario, the deformation of the beam's tip is insufficient due to the significant physical distance between nodes 2 and 3 in Configuration 1's FEM mesh (Figure 4.5a), which is not represented in the input tensors. Furthermore, it should be noted that node 2 has the potential for significant deformations. In contrast, node 3 remains fixed, resulting in a significant difference in the mechanical behavior of these two consecutive nodes in the input tensor. To circumvent this issue, we prefer to adopt the node ordering approach from Configuration 2, as illustrated in Figure 4.5b.

Random forces vs. modal forces

This section compares the innovative and brute-force data generation approaches introduced in Section 4.1. The performance of an MLP with four layers (one input layer, two hidden and output layers) trained on 20,480 samples is presented in Table 4.1. The samples were generated using random force amplitudes (second column) or modal force amplitudes (third column). In both strategies, a random area of variable size on the mesh boundary is selected to apply forces, which improves the network's ability to generalize and reproduce local and global deformations. It is worth noting that only mechanically stable samples are retained. The results indicate that the model-based force generation

approach produces larger deformations on average, resulting in higher prediction accuracy.

	MLP - Random F	MLP - Modal F
e	$2.205 \times 10^{-4} \pm 6.618 \times 10^{-4}\ m$	$1.692 \times 10^{-4} \pm 1.933 \times 10^{-4}\ m$
e_{max}	$0.0116 \pm 0.0293\ m$	$0.0096 \pm 0.0107\ m$
u_2	$1.0070 \pm 1.7846\ m$	$1.7095 \pm 1.6082\ m$

Table 4.1: Two MLP trained for 100 epochs with random or modal force amplitudes. u_2 gives the distribution of the L2 norm of the displacement.

MLP vs U-Mesh

The performance of two networks, namely a four-layer MLP and a three-stepped U-Mesh, were compared in this study. Both networks were trained on 20,480 samples generated using modal forces for 100 epochs. The results are presented in Table 4.2a, which shows the metric distribution over a testing dataset generated using the same strategy as the training data (i.e., modal-based forces). In addition, Table 4.2b presents the metrics over a dataset generated using random forces. The findings demonstrate that, for the beam scenario, the MLP performs slightly better on average than the U-Mesh when tested on data drawn from the same distribution as the training data (Table 4.2a). However, the U-Mesh exhibits better generalization capabilities than the MLP (Table 4.2b). Absolute training times were not reported since the two networks were trained on different machines. However, on similar problems, the MLP has proven to be much faster to train than the U-Mesh. Moreover, the MLP is around ten times faster than the U-Mesh in making predictions, making it the preferred option for this problem.

The FEM solution for this particular problem is computed in approximately 25 ms, within the limits of real-time constraints. Therefore, even though the U-Mesh is ten times faster than the FEM solver and the MLP is 100 times faster, the impact of our approach is negligible in this context. However, in the upcoming section, we will implement the two networks on finer mesh resolutions, where FEM solvers struggle to produce deformations in real-time, even with highly optimized versions of the codes.

4.2.2 Case study - Liver

We explored augmented reality for hepatic surgery to apply our method to a clinical setting. To achieve this, we trained an MLP and a U-Mesh using a real-life human liver geometry, modeled as a Neo-Hookean material according to Equation (2.27). The liver was

	MLP	U-Mesh
e	$1.692 \times 10^{-4} \pm 1.933 \times 10^{-4}$ m	$1.872 \times 10^{-4} \pm 1.410 \times 10^{-4}$ m
e_{max}	0.0096 ± 0.0107 m	0.0103 ± 0.0070 m
PT.	0.26 ± 0.01 ms	2.39 ± 0.05 ms

(a) Performance over a test dataset with samples drawn from the same distribution as training samples. PT stands for prediction times on a laptop equipped with a Quadro M1200.

	MLP	U-Mesh
e	$4.668 \times 10^{-4} \pm 9.897 \times 10^{-4}$ m	$4.056 \times 10^{-4} \pm 5.190 \times 10^{-4}$ m
e_{max}	0.0209 ± 0.0436 m	0.0198 ± 0.0257 m

(b) Generalization capacities, performance over a test dataset with samples drawn from a different distribution than training samples.

Table 4.2: Result of the MLP and U-Mesh on different dataset.

discretized into $3,309$ H8 elements with a length of 0.2 m , Young's modulus of $5,000$ Pa , and a Poisson's ratio of 0.45 . We fixed the mesh nodes that were near the *vena cava*, which serves as an anatomical liver fixation. However, for accurate simulations, it is essential to estimate the attachments of the organ precisely and specifically to the patient as they play a significant role in the accuracy of the simulation [13].

We generated $20,480$ samples by applying up to five random simultaneous forces on the liver's surface, and the data generation process took 19 h41 min50s for a grid resolution of $25 \times 22 \times 21$ nodes. We applied a *data scale factor* as a normalization technique. Node ordering made little difference in this scenario since the fixed boundary conditions were globally centered. Table 4.3 shows the performance of the two networks.

The results obtained from this experiment were satisfactory regarding accuracy and prediction times. The MLP produced more accurate predictions on average than the U-Mesh but had higher errors for some outliers due to its poor generalization capacity. These outliers corresponded to the upper bounds of the testing dataset that were poorly represented during training. The U-Mesh produced a deformation approximately 500 times faster than standard FEM solvers, which takes around $1,500$ ms to compute a solution, and the MLP was about 5,000 times faster.

The results indicate that for small-resolution meshes, the MLP outperforms the U-Mesh regarding prediction accuracy and training speed. This may seem surprising initially, but there are several potential reasons for this. Firstly, the U-Mesh requires a regular grid structure with many zeros, increasing input size without providing additional information. This leads to a more significant performance gap between the two networks

	MLP	U-Mesh
e	$5.879 \times 10^{-6} \pm 3.064 \times 10^{-6}$ m	$7.619 \times 10^{-6} \pm 4.874 \times 10^{-6}$ m
e_{max}	0.0052 ± 0.0031 m	0.0038 ± 0.0020 m
u_2	0.6574 ± 0.3524 m	0.6574 ± 0.3524 m
PT	0.31 ± 0.010 ms	2.61 ± 0.030 ms

Table 4.3: Performance of the MLP and of the U-Mesh trained over a dataset generated with random forces on the liver's surface. Both networks are trained for 100 epochs over 20,480 samples and tested on 100 samples drawn from the same distribution. PT stands for prediction times on a GeForce RTX 3090. u_2 gives the distribution of the L2 norm of the displacement.

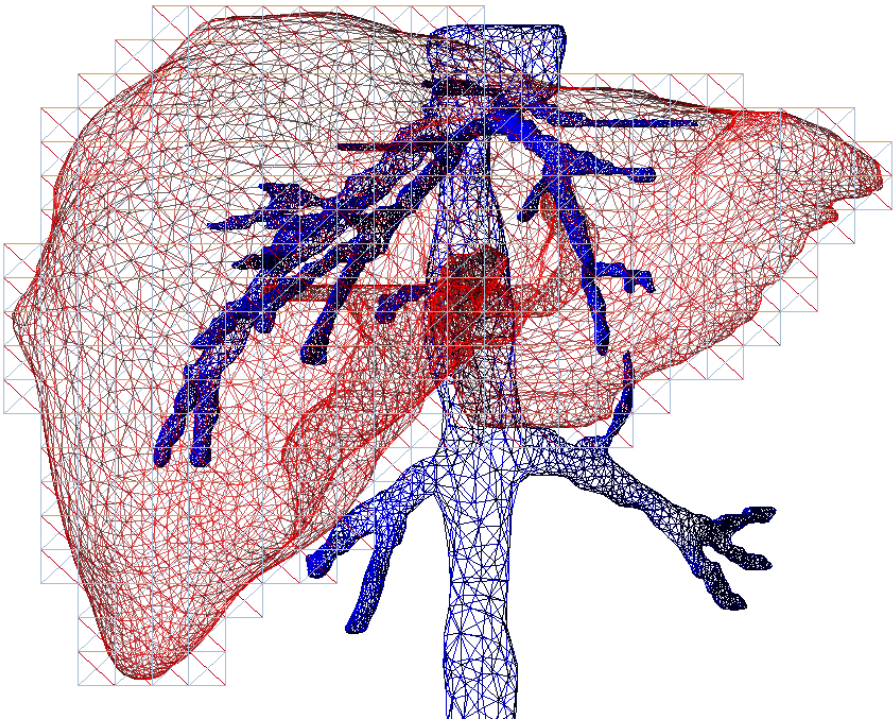


Figure 4.6: Projected view of the liver geometry and its 3,309 H8 mesh. The nodes in the middle of the trunk of the vascular tree (here in blue) are fixed.

in scenarios with more meaningless zeros, such as the liver scenario. A possible solution to reduce this gap is to use a geometry mask in the loss function, which can create constraints on domain knowledge. A more sophisticated solution involving sparse convolutions may be necessary for such cases. Secondly, the U-Mesh uses a reduced latent space to represent the problem, which allows for better generalization but results in slightly lower accuracy for patient-specific scenarios. However, the accuracy can be increased by increasing the size of the reduced latent space, although this will increase

training time and require a trade-off between accuracy and efficiency.

One topic we would like to discuss is the usage of structured meshes. While some may view it as a limitation in certain applications, using regular hexahedral grids combined with immersed-boundary methods for FEM simulation appears to be the most suitable option for numerical convergence to the analytical solution, as mentioned in previous sections. For instance, the works in [23] demonstrate that the Φ -FEM approach systematically outperforms the standard FEM on comparable meshes for structural mechanics simulation. Despite the current success of graph neural networks, we believe that structured meshes are the right research direction to follow when high accuracy is desired. However, graph neural networks can provide significant benefits for animation applications where speed and qualitative accuracy are the primary concerns.

However, structured meshes cannot accurately represent the geometrical complexity of complex shapes such as left atria or the human brain. In such cases, to use the proposed method in this chapter, one can voxelize complex triangular surface meshes by computing signed distance fields as proposed in [91], which enables accurate data encoding in structured grids.

We have presented results for relatively coarse meshes to meet clinical constraints in this chapter. We need to consider strategies that reduce training times and GPU memory load to learn the behavior of finer-resolution meshes. One option could be to encode fine meshes in coarser voxelization grids via barycentric weight functions or hierarchical grids. Although the error of 5.2 mm is acceptable, it was obtained on synthetic samples. When dealing with real, noisy, and sparse data, the error is likely to increase, which means that extra efforts will need to be made to improve the accuracy of the approach.

4.2.3 Conclusion and network of choice

In summary, the MLP and the U-Mesh achieve similar accuracy in the studied scenarios while being orders of magnitude faster than FEM solvers for simulating nonlinear deformations. The MLP is notably, on average, an order of magnitude faster than the U-Mesh. A notable advantage of this approach is the absence of fine-tuning in data generation and training, enabled by an automatic force generation method based on modal analysis and data normalization techniques. A generic numbering strategy was also proposed to optimize CNN learning accuracy.

Each network has its specific strengths, with the MLP being much faster, more accurate, and versatile, whereas the U-Mesh has better generalization capabilities. However, the quadratic growth of the number of parameters needed by the MLP limits its applicability to relatively small problems, as larger meshes quickly exceed the physical limits of

typical hardware.

The primary limitation of this work is the time-consuming data generation process, which can take several hours due to the need for numerous iterations of the Newton-Raphson algorithm to generate nonlinear deformations. One thing that could help to solve or at least reduce the data generation cost could be to use physics-informed machine learning where physical quantities are added to the learning process to better understand each sample; thus, we might be able to limit the number of samples.

Considering this limitation and the properties of the networks, we decided to continue our studies and research with the multilayers perceptron. As stated previously, the training speed is an important factor since, in augmented surgery, we have a short time between data acquisition and the actual surgery. Moreover, it is often easier to take something fast and make it more reliable than doing the opposite. From this perspective, we can add complexity to our network architecture or training to improve its generalization capabilities where we cannot improve the computation speed of the U-Mesh.

The next chapter present our take on the reliability issue coming from the ANN prediction. We will first present the Newton-Raphson algorithm in more detail, as well as its convergence scheme and limitations. Then we will present the contribution of Odot et al. [84] which uses the predictions of the ANN to accelerate the resolution of the problem, which in turn improves the reliability of the solution.

CHAPTER 5

HYBRID SOLVER

5.1 Newton method 70
5.2 Artificial neural network and solver 72

5.1 Newton method

This section presents the general Newton method from a mathematical perspective. The presentation in 2.6 was succinct and applied to soft body mechanics.

Newton's method is a second-order method for convex optimization. We first consider unconstrained smooth convex optimization.

$$\min_x \quad f(x) \tag{5.1}$$

Consider the following quadratic approximation:

$$f(y) \approx f(x) + \nabla f(x)^T (y-x) + \frac{1}{2}(y-x)^T \nabla^2 f(x)(y-x) \tag{5.2}$$

the newton update is obtained by minimizing the above with respect to y. This quadratic approximation is better than the approximation used in gradient descent since it computes more information about the function using the hessian. In the Newton method, we move in the direction of the negative Hessian inverse of the gradient:

$$x^{k+1} = x^k - (\nabla^2 f(x^k))^{-1} \cdot \nabla f(x^k) \tag{5.3}$$

The previous equation does not provide any notion of step size and is thus called the pure Newton's method. As explained by the inverse, Newton's method involves solving linear systems in the hessian.

Given a smooth, convex function f the Newton's decrement is given by:

$$\lambda(x) = \sqrt{\nabla f(x)^T \cdot (\nabla^2 f(x))^{-1} \cdot \nabla f(x)}$$

This decrement is proportional to the difference between $f(x)$ and the minima of the quadratic approximation at x.

$$\begin{aligned} f(x) - \min_x (f(x) + \nabla f(x)^T (y-x) + \frac{1}{2}(y-x)^T \nabla^2 f(x)(y-x)) \\ = f(x) - (f(x) - \frac{1}{2}\nabla f(x)^T \cdot (\nabla^2 f(x))^{-1} \cdot \nabla f(x)) \\ = \frac{1}{2}\lambda(x)^2 \end{aligned}$$

This formulation gives an approximate bound for suboptimality gap $f(x) - f^*$. The bound is not exact since it is computed on the minimum of the quadratic approximation. This is an interesting criterion for backtracking line searches.

We have presented Newton's method and a stopping criterion. We can now discuss the convergence of the method.

5.1.1 Newton method convergence

Let us assume that f is strongly convex with parameter m and twice differentiable and $dom(f) = \mathbb{R}^n$. Let us also assume that ∇f is Lipschitz with parameter L. Let additionally assume that $\nabla^2 f(x)$ is Lipschitz with parameter H. The following results hold for Newton's method with backtracking with parameters α, β and depend on two parameters $\gamma > 0$ and $0 < \eta \leq \frac{m^2}{H}$. Let k_0 be the number of step until $\|\nabla f(x^{k_0+1})\|_2 < \eta \cdot k_0$ with $\eta = \min\{1, 3(1-2\alpha)\} m^2/H$. This inequality will define the two stages of convergence. When $k \leq k_0$ we are in the phase called the damped Newton phase,

$$f(x^k) - f^* \leq f(x^0) - f^* - \gamma k$$

We are guaranteed to decrease the criterion by $\gamma = \alpha\beta^2\eta^2 m/L^2$ in every step. If the function is poorly conditioned, γ is small, and thus, convergence is slow. When $k > k_0$, we enter the pure Newton phase, and the convergence rate is named quadratic.

$$f(x^k) - f^* \leq \frac{2m^3}{H^2}(\frac{1}{2})^{2^{k-k_0+1}}$$

One important fact about this phase is that once we enter the quadratic convergence, we never leave it.

To reach a desired precision ϵ, the number of iterations needed to leave the first phase is:

$$\frac{f(x^0) - f^*}{\gamma}$$

For the second phase using strong convexity, we can prove that the number of iterations required is:

$$\frac{f(x^0) - f^*}{\gamma} + log(log(\epsilon_0/\epsilon))$$

Where $\epsilon_0 = \frac{2m^3}{H^2}$. The $log(log(\epsilon_0/\epsilon))$ convergence term make the phase quadratic. This convergence is only local and guaranteed in the pure Newton's phase.

5.1.2 Failure analysis

We have seen that the Newton method can converge when the assumptions are met. Sometimes it is hard to know if assumptions are all perfectly met when dealing with par-

tial differential equations when we do not know the exact solution to the problem. In this section, we discuss some failures that can happen during the process.

We first discuss iteration points. Starting point problems can be extremely tricky since the method might still diverge even when all the assumptions are met. For example, this can append with functions that approach zero asymptotically when x goes to infinity.

The second type of problem occurs when an iteration point is stationary (the gradient is null). The method will then diverge since we divide by the derivation of f . This also causes a problem when the derivative is negligible since the next iteration will be worse than the current one.

The final problem considering iteration points is that it may enter a cycle. As an example, the function $f(x) = x^3 - 2x + 2$, when starting at 0, will alternate between 0 and 1 without converging. It is said to enter a stable 2-cycle since even a small neighborhood around 0 and 1 will converge toward one of these values and eventually enter the 2-cycle.

Considering problems with the derivative, it could not exist or be discontinuous at the root. Thus, the method diverges. Newton's method is very sensitive to the quality of the derivative.

We have now presented Newton's method and will discuss the proposed contribution that uses a neural network to improve the convergence rate of Newton's method, which in turn improves the reliability of the solution.

5.2 Artificial neural network and solver

As we have seen, Newton's method has interesting convergence properties in the proper context. Our goal here will be to use an artificial neural network to put ourselves in the quadratic convergence phase as soon as possible. The idea is to initialize Newton's method with a prediction of a purposely trained ANN.

To present the contribution, we first explicit the algorithm in the context of the Finite element method applied to soft-body mechanics. The second part will consist of the modified algorithm and a quick discussion of the bottlenecks of the method.

Newton method and Soft-body mechanics

We first need to identify a couple of terms.

Using (5.3) and (2.35) we can rewrite the newton step as:

$$\mathbf{u}^{k+1} = \mathbf{u}^k - \mathbf{K}(\mathbf{u}^k)^{-1} \cdot (\mathbf{R}(\mathbf{u}^k) - \mathbf{b}) \tag{5.4}$$

Where $\mathbf{K}(\mathbf{u})$ is the tangent stiffness matrix. This Newton step presents the computation of two important quantities. The first one is the internal forces $\mathbf{R}(u)$ that are parametrized by the material law and its associated parameters. The second quantity is the inverse tangent stiffness matrix assembled or directly derived from the internal forces.

We define the vector $\mathbf{r}(\mathbf{u})^k = \mathbf{R}(\mathbf{u}^k) - \mathbf{b}$ which represent the residual forces in the system.

Using this, we can write the algorithm as follows:

Algorithm 4: Newton-Raphson algorithm

Data: $k = 0$, $\mathbf{u}^0 = 0$, $d\mathbf{u} = 0$
1 **while** $\|\mathbf{r}(\mathbf{u}^k)\| < \epsilon \; or \; \|d\mathbf{u}\| < \eta$ **do**
2 Compute $\mathbf{R}(\mathbf{u}^k)$
3 Compute $\mathbf{K}(\mathbf{u}^k)$
4 Solve $\mathbf{K}(\mathbf{u}^k) \cdot d\mathbf{u} = \mathbf{r}(\mathbf{u}^k)$
5 $\mathbf{u}^{k+1} = \mathbf{u}^k - d\mathbf{u}$
6 $k = k + 1$
7 **end**

It exists multiple options in order to speed up the Newton-Raphson algorithm. One can improve the computation speed of $line$ 3 using parallel computation as an example. Usually, the bottleneck of the algorithm is at $line$ 4 where most of the time is spent inverting the matrix $\mathbf{K}(\mathbf{u}^k)$, composed of the number of dofs squared coefficients. Multiple approaches have been proposed to improve such computation. The Quasi-Newton-Method [10] create a matrix $\mathbf{B}$ which is an approximation of $\mathbf{K}^{-1}(\mathbf{u}^k)$, recently, Duff *et al.* [22] proposed a new formulation of the LDL^T solver using an *a posteriori* threshold pivoting.

Finally, one can also reduce the number of iterations needed to satisfy the condition on $line$ 1 using a good initial guess $\mathbf{u}^0 = \mathbf{u_p}$. In this work, we choose this approach and use the prediction of the network to set an optimal starting point. This reduces the number of iterations of the algorithm most of the time while also guaranteeing that we obtain a correct solution to our problem, even if the prediction is inaccurate or incorrect. Furthermore, this work does not interfere with previously presented methods and can be used as a complementary upgrade for the algorithm.

The logic of the algorithm remains unchanged. There are multiple scenarios to consider. First, the trivial one where $\mathbf{u}^p = \mathbf{u}^0$, algorithm 5 introduces a slight computational overhead at $line$ 3 but produces the same answer. The second scenario is the one where $\|\mathbf{R}(\mathbf{u}^1)\| <= \|\mathbf{R}(\mathbf{u}^p)\|$ at $line$ 12. In this case, the prediction does not provide any gain to the simulation. This could be due to $\mathbf{b}$ being too different from the training data. The

artificial neural network cannot generalize enough to produce a good answer.

Algorithm 5: Hybrid Newton-Raphson algorithm

Data: $k = 0$, $\mathbf{u}^0 = 0$, $d\mathbf{u} = 0$

1 $\mathbf{u}^p = Prediction(\mathbf{b})$
2 Compute $\mathbf{R}(\mathbf{u}^p)$
3 **if** $\|\mathbf{r}(\mathbf{u}^p)\| < \epsilon$ **then**
4 exit
5 **end**
6 Compute $\mathbf{R}(\mathbf{u}^0)$
7 **while** $\|\mathbf{r}(\mathbf{u}^k)\| < \epsilon \ or \ \|d\mathbf{u}\| < \eta$ **do**
8 Compute $\mathbf{R}(\mathbf{u}^k)$
9 Compute $\mathbf{K}(\mathbf{u}^k)$
10 Solve $\mathbf{K}(\mathbf{u}^k) \cdot d\mathbf{u} = \mathbf{r}(\mathbf{u}^k)$
11 $\mathbf{u}^{k+1} = \mathbf{u}^k - d\mathbf{u}$
12 **if** $k = 0$ and $\|\mathbf{r}(\mathbf{u}^1)\| > \|\mathbf{r}(\mathbf{u}^p)\|$ **then**
13 $\mathbf{u}^1 = \mathbf{u}^p$
14 **end**
15 $k = k + 1$
16 **end**

This can also happen if the force is too small and produces a displacement field in the order of magnitude of the noise generated by the network. The last scenario appears when the condition at either $line\,4$ or $line\,12$ is satisfied. In the best case, when it stops at $line\,4$ we can compute the displacement field in a couple of milliseconds. In the other case, the prediction usually reduces the number of iterations needed to satisfy the condition on $line\,7$, thus speeding up the algorithm compared to its classical version.

We now present the result of this contribution. First, we must show that the network can quickly and accurately predict deformations. Then, we will discuss the gains of the Hybrid Newton-Raphson algorithm.

5.2.1 General results

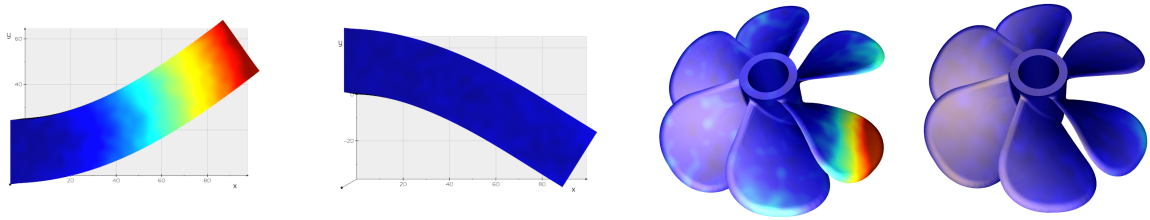
This section discusses the ability of the network to approximate deformations on various geometries with different elastic laws.

This is another important test for the neural network. The different parameters of the 2 test cases are summarised in table 5.1. Both trainings lasted about 12 hours (including dataset generation) on an NVidia TITAN RTX, with 4,095 generated samples each.

Name	#DOFs	material	L [m]	E [Pa]	Time [ms]	e_{mean}	e_{max}	SNR_{min}^{dB}
Beam	12,000	S-V-K	100	4.5×10^3	0.4	8.0×10^{-6}	0.03	8.4
Propeller	12,075	N-H	1.0	2.03×10^{11}	0.4	2.7×10^{-6}	0.01	18.9

Table 5.1: Results of a comparison between FEM simulations and ANN predictions over 100 randomly distributed forces with random amplitudes.

Both models with completely different simulation parameters provide similar mean and max errors over 100 simulations. This, although demonstrated only on two models in this thesis, can argue that the network and its framework provide accurate global and local deformations of a mesh while handling a wide range of simulation parameters. The displacement field is computed in a steady $0.4ms$ up to three orders of magnitude faster than the reference FEM simulation. To compare, the simulation of the deformation of the propeller takes around $500ms$ to compute. One hypothesis to explain the big difference in SNR relies on the amount of near-null deformations seen by both models. The beam has few samples where the displacement of the whole body is almost null and hence is less trained at generating a value close to 0. Where with the propeller, most of the samples require a null or almost null displacement field for most of the points. A proper in-depth analysis of multiple models is required to conclude this hypothesis. Works on this perspective are currently being held.



(a) **(b)** **(c)** **(d)**

Figure 5.1: Examples of large nonlinear elastic deformations predicted by our neural network. The colors represent the node-wise Euclidean distance to the solution of the Newton-Raphson algorithm. For both beams, the color gradient goes from 3×10^{-4}m (blue) to 3×10^{-2}m (red), and for both propellers, the color gradient goes from 3×10^{-5}m (blue) to 2×10^{-3}m (red).

Beams (a) and (b) in Figure 5.1 undergo roughly the same amount of deformation (with a deflection at the tip of $\approx$ 40m) yet lead to very different error values and patterns. Beam (a) has a maximum error of 37mm near the beam's free end, reducing gradually to 54 µm near the fixed end. Beam (b), on the other hand, provides a homogeneous error with a maximum error of 5.0 mm and an average error of 6.6 µm. The same behavior can be observed with the propeller model. Propeller (c) has a similar deformation to (d). It

Name	e_{mean}	e_{max}	SNR^{dB}
Beam (a)	5.4×10^{-5}	0.037	22.2
Beam (b)	6.6×10^{-6}	0.0050	40.2
Propeller (c)	2.82×10^{-6}	0.0036	33.9
Propeller (d)	8.2×10^{-7}	0.0008	42.3

Table 5.2: Error values and SNR of the deformations shown at Figure 5.1. The deformations are highly nonlinear, yet the error values and SNR remain in the range of values displayed in table 5.1.

has an average prediction error of 2.82 µm with a maximum value of 3.6 mm, about four times as much as in the second scenario.

5.2.2 Hybrid Newton-Raphson results

The artificial neural network has proven precise, up to a couple of micrometers on average. Where one could be satisfied with the precision, another may want to ensure the respect of some properties, such as incompressibility or fixed points on the boundaries. As shown in section 5.2, the Hybrid Newton-Raphson algorithm proposes to use the prediction of the network in order to speed up the algorithm. The experiment will compare the speed and solution of the ANN, the classic Newton-Raphson, and its hybrid version. The network trained for the beam at section 5.2.1 is used to obtain the following values.

Solver type	Converged simulations	Prediction picked	Average iterations
Classic	68%	-	9.1
Hybrid	99%	53%	5.0

Table 5.3: Results of comparing the classic Newton-Raphson algorithm and the presented Hybrid Newton-Raphson algorithm over 100 randomly distributed forces with random amplitudes.

These results are computed from a dataset of 100 random external forces. Among them, 50 are within the amplitude range of the training dataset, and the 50 others have an amplitude up to two times bigger. Although 50 have an amplitude within the training bounds, they do not share orientation or location with any training datasets.

The classic Newton-Raphson manages to converge 68% of the time. On average, when it converges, it does so in 9.1 iterations.

The Hybrid Newton-Raphson converges 99% of the time and 100% when the classic version did too. Our algorithm leads to a gain of 45% in terms of convergence. In 53%

of the cases, the solution of the neural network is preferred to the first Newton-Raphson iteration. Over the 100 test samples, the Hybrid Newton-Raphson picked two out of three times the prediction of the network as a better starting point than the result of the first iteration of the Newton-Raphson algorithm. From this point, on average, the algorithm converges in 5 iterations. This shows that from the prediction, the algorithm converges on average in 4 iterations, adding up to 5 to account for the first one discarded when the prediction is picked.

Overall the proposed algorithm converges more often and faster than the classic method while keeping the ordinary convergence properties of the Newton-Raphson algorithm.

We have presented our take on the reliability issue from the ANN prediction. This first use of our ANN is interesting as it leads to performance improvement, but computing object deformation usually comes from a need. In the context of this thesis, the need for such computations comes from augmented surgery. The next chapter presents our take on augmented surgery, specifically augmented laparoscopic surgery. We start by presenting the context and why surgeons would benefit from accurate augmented reality during surgery. We then introduce the contribution of Odot et al. [85], where they use an ANN to perform the non-rigid registration task.

CHAPTER 6

OPTIMAL CONTROL FOR AUGMENTED SURGERY

6.1 Context 81
6.2 Shape matching 84

From the previous chapters, we have shown that we can accurately predict nonlinear deformations of soft bodies such as, but not only organs. Using an artificial neural network allows us to compute deformations in less than a millisecond, which is multiple orders of magnitude faster than the traditional methods for the presented meshes. The deformations are computed in real-time, which is a hard constraint if we want to apply our work in the context of laparoscopic surgery. The real-time requirement emerges from the fact that we cannot interfere with the operation flow by asking the surgeon to wait for our network to predict the deformation, thus, slowing the operation.

The idea of this work is to give tools to introduce augmented reality in the operation room by projecting the organ's internal structures on the surgeons' video feed. Introducing internal 3D information in the display could help the practician by offering a better visualization of the operation and also help the decision-making. This projection requires first extracting the current deformation of the organ from the video feed and second, deforming the internal structure and projecting them on the video feed. Once we have the forces, we know that deforming a liver and its internal structure is a matter of milliseconds using the previously presented methods.

In this section, we focus on the first task: *Compute the current deformation of the organ from the video feed*. Extracting deformations from a video feed requires us to find the rigid registration, extract the surface from the image then compute the deformation. This thesis focuses more on the theoretical aspect of the method than the actual experimental results. Thus, we make two hypotheses that allow us to only focus on retrieving the deformation. The first one is that the rigid registration is already done. The second one is that we can extract a 3D point cloud from RGB images. These two hypotheses are actual research subjects and could be thesis subjects on their own. In order to have a proper understanding of our work, we will study the impact of these two hypotheses on our method.

From the video feed, we are left with a partial 3D surface of the object. This point cloud is interesting because it provides valuable information on the deformation. Using this deformation, we should be able to extract the force that has generated it. This reasoning has been proven working by Mestdagh et al. [78]. In this work, they can retrieve the forces that deform a liver composed of nonlinear material to fit a partial surface point cloud. In other words, they find the forces such that the liver fits the observed data.

Our approach mainly involves speeding up the presented computation using an artificial neural network.

In this section, we first briefly present laparoscopic liver surgery to help contextualize our work. We then briefly introduce the optimal control framework used by Mestdagh et al. [78]. Finally, we present our contribution to the subject.

6.1 Context

6.1.1 Liver resection

Liver resection or hepatectomy is a surgical procedure where part or all of the liver is removed. Partially removed liver can grow to its former size, whereas total liver resection requires a transplant. Due to the extreme regeneration ability of the liver, liver resection is a common practice when dealing with liver disease. Although the operation is often practiced, it remains a complicated surgery due to the density of vessels in the liver, which can cause significant bleeding.

It exists two main methods to perform a liver resection. The oldest one is open abdominal surgery, where a single long incision (Figure 6.1), also known as a laparotomy, is made to gain access to the abdominal cavity. The more recent one is laparoscopic surgery, where the procedure is carried out through small incisions in the abdomen.

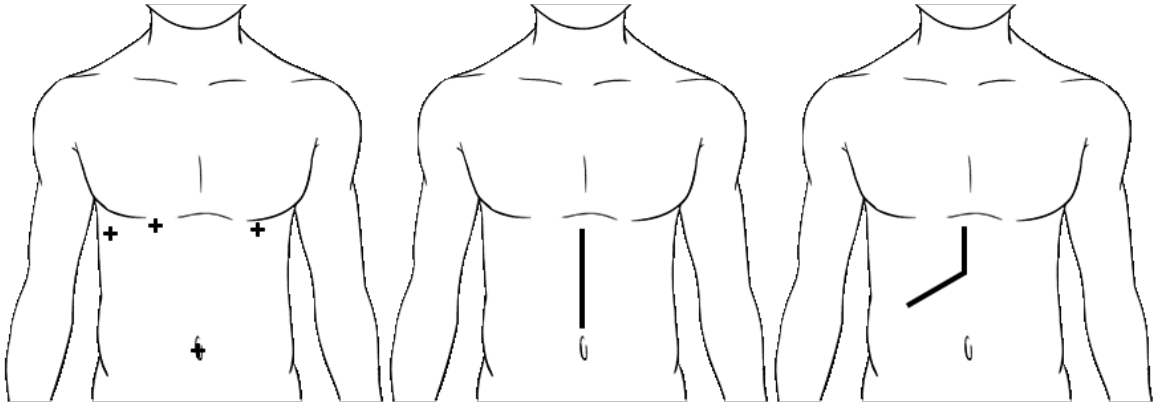


Figure 6.1: example of incisions location and size in laparoscopic surgery (left) compared to two examples of laparotomy (middle and right) performed in open surgery. Incisions vary depending on the surgery and patient-wise characteristics.

The open surgery approach allows direct visualization of the organs within the abdomen and is most commonly used for more complex surgical procedures. In comparison, during laparoscopic surgery, the surgeon watches the images transmitted by a camera attached to a specially designed tool called laparoscope.

The technic has improved since 1991, and the first laparoscopic liver resection [96] (LLR). This method has become the primary curative treatment for liver malignancies since it presents both pre and post-operative benefits with reduced operation times, blood loss, and length of stay [15, 111]. For example, the LLR is recommended as the first line of treatment for 0-A stage hepatocellular carcinoma (the most common type of primary liver cancer) by the Barcelona clinic liver cancer [11]. With advancements in surgical

techniques, more intricate procedures such as major liver resections or living donor liver resections are now feasible through a laparoscopic or robotic approach.

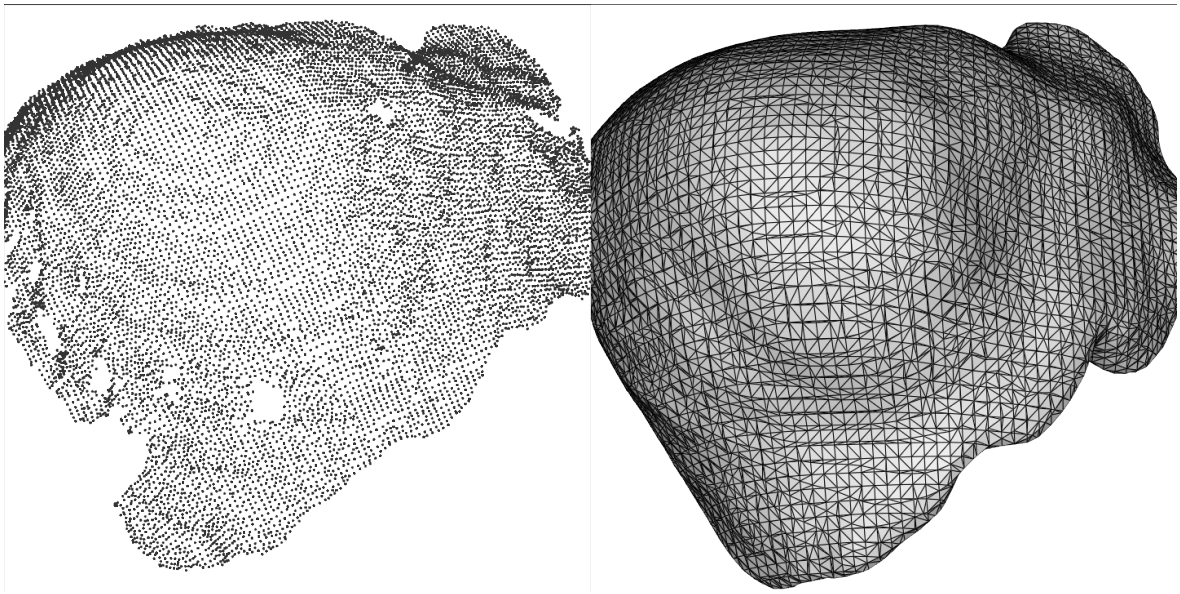
Existing technics allow using the video feed to create a surface mesh of the observed part of the liver. For the rest of this chapter, we will consider the reconstruction and rigid registration already done. The displayed point clouds are generated by randomly sampling the surface of different parts of multiple deformed livers using classic simulation techniques. For clarity, we will call these randomly sampled surfaces "reconstructed surfaces" to fit the arguments.

Random sampling is essential to remove any one-to-one correspondence between the simulated mesh and the reconstructed surface, which would introduce biases in the regression and not correctly reflect reality. As we can see in Figure 6.2, neither the density nor the location of the points match between data and observation. Random sampling is achieved by selecting multiple triangles corresponding to a zone of interest on the mesh. For each triangle, we generate random barycentric coordinates, which gives us a point that is not part of the geometry. An example of such sampling is presented in Figure 6.3.

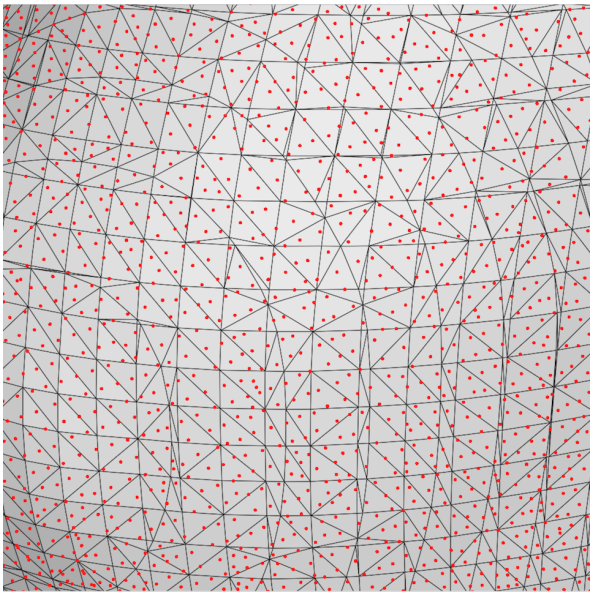
Scholars have tackled the subject of per-operative nonlinear registration multiple times. We present one of the works that is the closest to our method.

Rucker et al. [102] propose to optimize the displacement of a part of the liver that they call *support surface* to fit the observed sparse surface data. To do so, they compute at the same time the rigid and non-rigid registrations of a linear model of the liver. Their rigid registration optimization is done on classical translational and rotational coefficients. The particularity appears with their non-rigid registration; they decided to use the linearity of their model and optimize the coefficient of precomputed deformation basis. Therefore, The resulting deformation is a linear interpolation of their deformation basis, which works since their liver material is linear. This method has the advantage of being relatively fast but requires formulation to use linear material. Linear materials are an important drawback since, during surgery, organs can undergo important deformations, which are poorly modeled by linear materials.

We chose to continue the work of Mestdagh et al.[78] on optimal control using the adjoint method applied to laparoscopic surgery. It has shown promising results on nonlinear material with a key benefit. The key aspect of this work is that it aims at fitting the mesh in the observed data by computing the forces responsible for the corresponding deformation. Forces on which our method is based. As a very brief introduction, the method is composed of two parts. One forward problem uses the forces to compute a displacement, and one adjoint problem uses the geometrical error to the target to compute an increment of force. One flaw of the method is that using nonlinear material requires a Newton-based solver in the forward process, thus, drastically slowing down the compu-



(a) Pointcloud acquired from a RGBD camera next to the corresponding mesh.



(b) Zoom-in : Acquired pointcloud (red) overlapped with the corresponding mesh

Figure 6.2

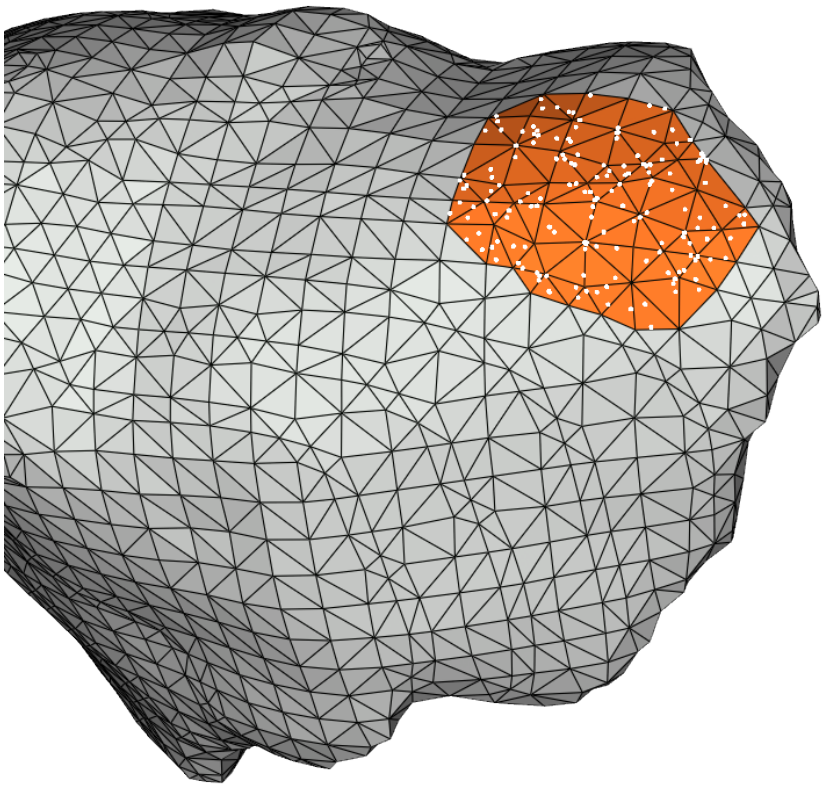


Figure 6.3: example of a mesh with a randomly sampled zone of interest. The mesh appears grey, while the zone of interest is orange, and the randomly selected points appear as white dots.

tation compared to working with a linear problem. This nonlinearity is essential to model the organ correctly but doesn't allows them to fit the real-time requirements. Our work will focus on improving the speed of the forward problem using an artificial neural network. We will see that by using the neural network to its full advantage, we reduce both the forward and adjoint problem computation times.

With this said, using our neural network, we can produce a displacement from a force, and using Mestdagh et al.[78] work, we know how to compute the force of a corresponding displacement. We will first focus our interest on a presentation of the method used by Mestdagh et al.[78] and explain how our contribution fits in this framework. We will then present our result on the matter and study the effect of our hypothesis on our precision.

6.2 Shape matching

6.2.1 Problem setup

Formulating the registration problem in the generic optimal control framework allows more flexibility in the problem and type of data being dealt with. The optimal control

framework offers various generic tools to study and solve the problem.

We now present the problem setup in its integral form to fit the general description of the generic optimal control theory. Following the illustration of the problem in Figure 6.4, we have a body Ω^0 with its constitutive law. The surface of the organ noted $\partial\Omega$ can be split into two parts, $\partial\Omega_D$ and $\partial\Omega_N$, which represent the zones on which the respectively Dirichlet and Neumann boundary conditions are applied. We define the Dirichlet boundary condition at the zone where the hepatic veins enter the liver and where the falciform ligament attaches to the surface of the liver. We do not consider gravity in this experiment, although it can be easily added as a parameter. Finally, we note $\mathbf{u_b}$ the displacement field $\mathbf{u}$ generated by the force distribution $\mathbf{b}$.

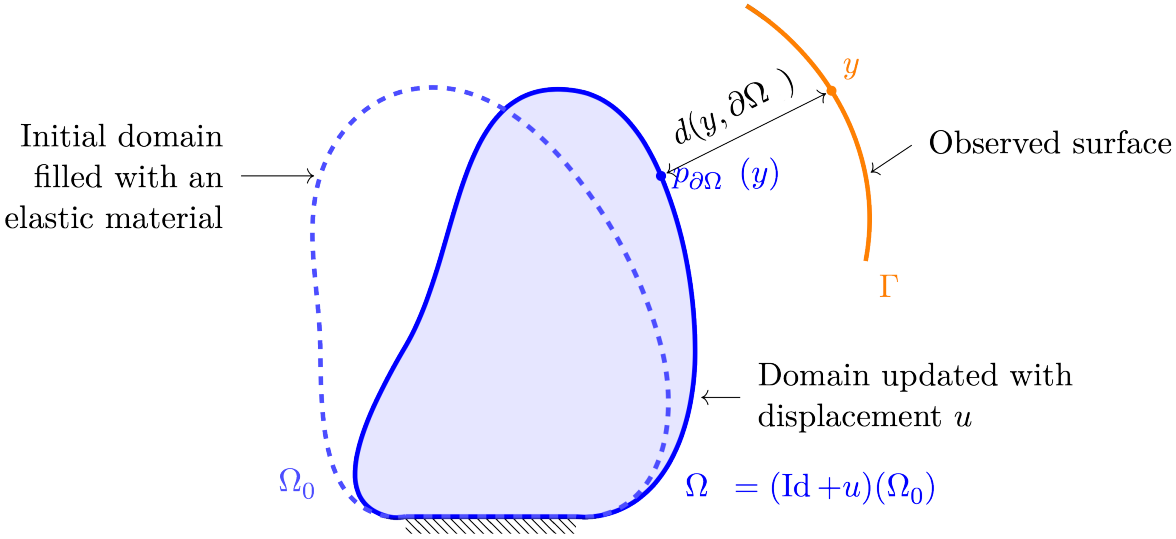


Figure 6.4: Problem setup. The body at rest Ω^0 undergoes a deformation u becoming the deformed body Ω. The distance from a point of the observed surface y to its orthogonal projection on the surface of the body $\partial\Omega$ is noted $d(y,\partial\Omega)$. (Image from Mestdagh et al.[78])

Data and objective function

Although the per-operative configuration is known, the current deformed state must be computed using the observed data. This observed data is considered to be a two-dimensional manifold noted Γ representing a part of the surface of $\partial\Omega$. We can perform a rigid registration using the surgeon's knowledge of the current part of the liver that is being manipulated. This first registration gives us a region noted $S_0 \subset \partial\Omega^0$, which we use to fit the observed data. We consider $S_{\mathbf{u_b}}$ the region S_0 displaced by $\mathbf{u_b}$.

Thus, we consider the registration done when :

$$\Gamma \subset S_{\mathbf{u_b}} \tag{6.1}$$

The displacement $\mathbf{u_b}$ is computed as the minimizer of a functional $J : C(\bar{\Omega}_0) \rightarrow \mathbb{R}$ which goes to zero only when equation 6.1 holds.

We define the functional J :

$$J(\mathbf{u}) = \frac{1}{2}\int_{\Gamma} d^2(y, S_{\mathbf{u}})\, dy \tag{6.2}$$

where $d^2(y, S_{\mathbf{u}}) = min_{x \in S_{\mathbf{u}}} \|x - y\|^2$. In the following part, we formulate the optimal control problem.

Optimal control formulation

We have defined the role of the functional J , which computes the discrepancy between Γ and $S_{\mathbf{u}}$. While we can find trivial solutions to this problem, the role of optimal control is to ensure that the physical properties of the model are respected. The simplest form of the optimal control problem reads :

$$inf_{\mathbf{b} \in \mathscr{B}} J(\mathbf{u_b}) \tag{6.3}$$

Where $\mathbf{b}$ is a set of forces applied on the surface of the body ($\mathscr{B} = L^2(\partial\Omega_N)$), which will be our *control*. When solving equation 6.3 the algorithm will try to find a force $\mathbf{b}$ such that Γ and $S_{\mathbf{u_b}}$ coincide perfectly. The formulation of equation 6.3 does not constrain the set of force in any way. The problem is ill-posed; we need a way to constrain the space of forces to optimize toward a more probable solution. This is achieved by putting external knowledge in the previous equation, such as load location and force distribution. Load location can be added by restricting the optimization domain to a subdomain B such that ($\mathbf{b} \in B \subset \mathscr{B}$), and force distribution can be controlled by a penalty term in the objective function.

One important knowledge we can add to the optimization is the concept of noise. When we reconstruct the surface observed by the laparoscope, we introduce noise in the data. This noise comes from multiple sources, such as the reconstruction method or the camera, and must be considered when we compute the non-rigid registration. Thus letting the algorithm perfectly fit the observation would introduce non-realistic high-frequencies in the model surface. These high frequencies also reduce the computation speed of the method by requiring more solver iterations due to the important fine-tuning of the final steps. In other words, we want a smooth surface minimizing the distance to the reconstructed point cloud without overfitting the observations.

To do so we reformulate the equation 6.3 to :

$$inf_{\mathbf{b}\in\mathscr{B}}\Psi(\mathbf{b}) \quad \text{where} \quad \Psi(\mathbf{b}) = J(\mathbf{u_b}) + G(\mathbf{b}) \tag{6.4}$$

Where the constraint is the equation 2.35 that corresponds to the elastic deformation of a body subject to external forces $\mathbf{b}$. Here $G(\mathbf{b})$ is a penalisation term such as :

$$G(\mathbf{b}) = \frac{\alpha}{2}\int_{\partial\Omega_N} \|\mathbf{b}\| ds \qquad \text{or} \tag{6.5}$$

$$G(\mathbf{b}) = \frac{\alpha}{2}\int_{\partial\Omega_N} \nabla\cdot\mathbf{b}\, ds \tag{6.6}$$

The first one aims to minimize the amplitude of the forces applied to the object. The importance of this criterion in the equation 6.4 is weighted using the term α , which values between zero and one. This is interesting since the algorithm cannot apply important local forces to perfectly fit the point cloud, which could result in the described high frequencies.

The second formulation tries to minimize the divergence of the forces and thus produce a force distribution where the vectors collinear one another. The term α plays the same role as mentioned before. This is interesting since surgeons usually apply local pull or push forces to the surface, which can be considered local divergence-free vector fields.

These regularisation criteria wrap up the first part of the algorithm. We now know how given a control $\mathbf{b}$, we evaluate how well a deformation fits the observed data. The whole idea of optimal control is to find the control that minimizes the equation 6.4 This minimization requires updating the controls and, thus, finding a way to compute the correct updates. In order to compute the update of the control, they chose to use the Adjoint method, as it is a standard method for solving optimal control problems.

Solving equation 6.4 using a first-order optimizer requires computing $\nabla\Psi(\mathbf{b})$. This can be done considering $\mathbf{h}$ an admissible direction in the space of controls (force) and $\mathbf{w}$ its corresponding direction in the space of state (displacement). We recall that $\Psi(\mathbf{b})$ hides the elastic problem, but every evaluation of Ψ requires to solves equation 2.35 first. From this observation, we have that $\mathbf{w}$ is the solution of the tangent system $\nabla\mathbf{R}(\mathbf{u_b})\mathbf{w} = \mathbf{h}$, where $\nabla\mathbf{R}(\mathbf{u_b})$ is the Jacobian of the elastic residual also know as the tangent stiffness matrix. Using this notation, the first order derivation of Ψ becomes :

$$\nabla\Psi(\mathbf{b})^T\mathbf{h} = \nabla J(\mathbf{u})\mathbf{w} + \nabla G(\mathbf{b})\mathbf{h} \tag{6.7}$$

One problem remains, on the right-hand side, we have $\nabla J(\mathbf{u})$ that is expressed in the space of displacement where we would like it to be expressed in the space of controls.

In the adjoint method, the adjoint state $\mathbf{p}$ is used to do just that. We define $\mathbf{p}$ as the solution of the adjoint system :

$$\nabla \mathbf{R}(\mathbf{u_b})\mathbf{p} = \nabla J(\mathbf{u_b}) \tag{6.8}$$

Although we solve a single linear problem (the value of $\mathbf{p}$ do not affect $\mathbf{R}(\mathbf{u_b})$), we cannot precompute $(\nabla \mathbf{R}(\mathbf{u_b}))^{-1}$ in the case of nonlinear materials since the response is not linear. This gives us the following:

$$\nabla J(\mathbf{u_b})\mathbf{w} = \mathbf{p}^T \nabla \mathbf{R}(\mathbf{u_b})\mathbf{w} = \mathbf{p}^T \mathbf{h} \tag{6.9}$$

Thus, we can rewrite equation 6.7 as:

$$\nabla \Psi(\mathbf{b}) = \mathbf{p} + \nabla G(\mathbf{b}) \tag{6.10}$$

Our favorite first-order optimizer can use this gradient to minimize Ψ .

We have now presented all the parts constituting the method proposed by Mestdagh et al. [78]. We now present the algorithm in its entirety. As shown in Figure 6.5, the algorithm starts with an initial guess, usually zero. The corresponding deformation is computed to give a displacement $\mathbf{u}$. This displacement is evaluated using a loss function that computes the distance from the mesh to the observational data. From this evaluation, the adjoint system is built and solved, giving us the increment in force we must apply to minimize the loss of function value.

6.2.2 Our contributions

Our contributions to the subject will improve the algorithm's computational performance. The presented algorithm is subject to two main computational bottlenecks, which can be removed using deep learning techniques. This acceleration is achieved using deep neural networks trained to produce deformations from input forces.

First bottleneck: Simulation of nonlinear material. The first bottleneck we aim to tackle appears in the forward simulation step. This step is the most computationally demanding part of the algorithm. It computes the deformations of an object subject to a control using the finite element method presented in Chapter 2. This can be easily computed in real-time when dealing with linear material. This is done by observing that the left-hand side of the system we are solving in equation 2.37 is composed of only two members, the variable we are solving for and the tangent stiffness matrix $\mathbf{K}(\mathbf{u} + d\mathbf{u})$ The matrix $\mathbf{K}(\mathbf{u} + d\mathbf{u})$ representing the mechanical properties of the object is constant since the material is linear. Therefore, one can precompute the value of $(\mathbf{K}(\mathbf{u}))^{-1}$ and simply right-multiply the

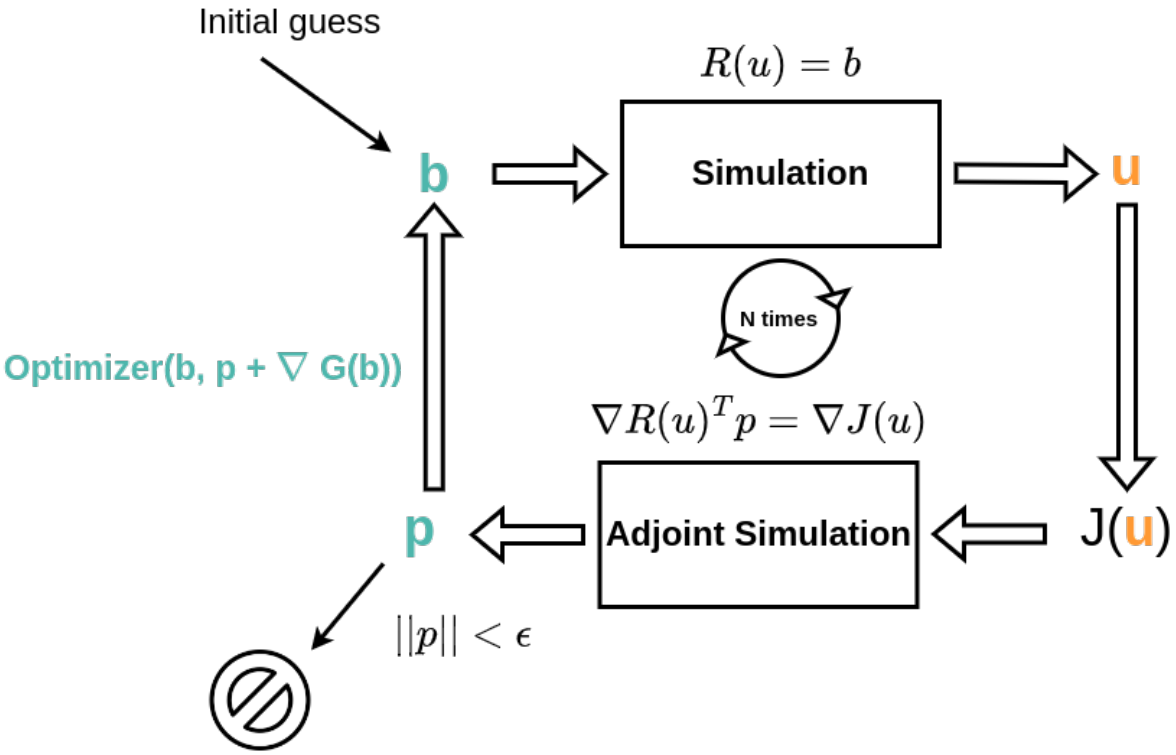


Figure 6.5: Schematic of the algorithm proposed by Mestdagh et al. [78] where $J(\mathbf{u})$ correspond to the function evaluated in equation 6.4.

inverted tangent stiffness matrix to the external force. The simulation results in a single matrix-vector product that can be computed in real-time even for objects composed of hundreds of thousands of degrees of freedom. While this is extremely fast, linear materials do not correctly model biological tissues and are more accurately modeled using nonlinear material.

The bottleneck appears when we try to deal with nonlinear materials. Here, the matrix $\mathbf{K}(\mathbf{u}+d\mathbf{u})$ is no longer constant; thus, we cannot precompute its inverse. Therefore, we have to solve multiple subsequent linear systems where each system will most likely take more time than a $\frac{1}{60}$ of a second (real-time criterion) when dealing with meshes of interest in this thesis. The full resolution of the systems can take multiple seconds and therefore is not usable as such in the context of an operation room.

Our contribution to reducing this bottleneck is to use our artificial neural network to compute the forward simulations. By identification, we can see similarities between the control of this method and our previous work. Both inputs/control are external forces; both outputs are the resulting deformations. We replace the simulation with an ANN that performs the same operations but much faster, as shown in Chapter 4. With this improvement, the cost of a simulation goes from multiple seconds to less than a millisecond. Furthermore, the most important bottleneck of the algorithm now becomes its fastest part.

To keep track of the different improvements of the original algorithm, we transform the previous Figure 6.5 into Figure 6.6:

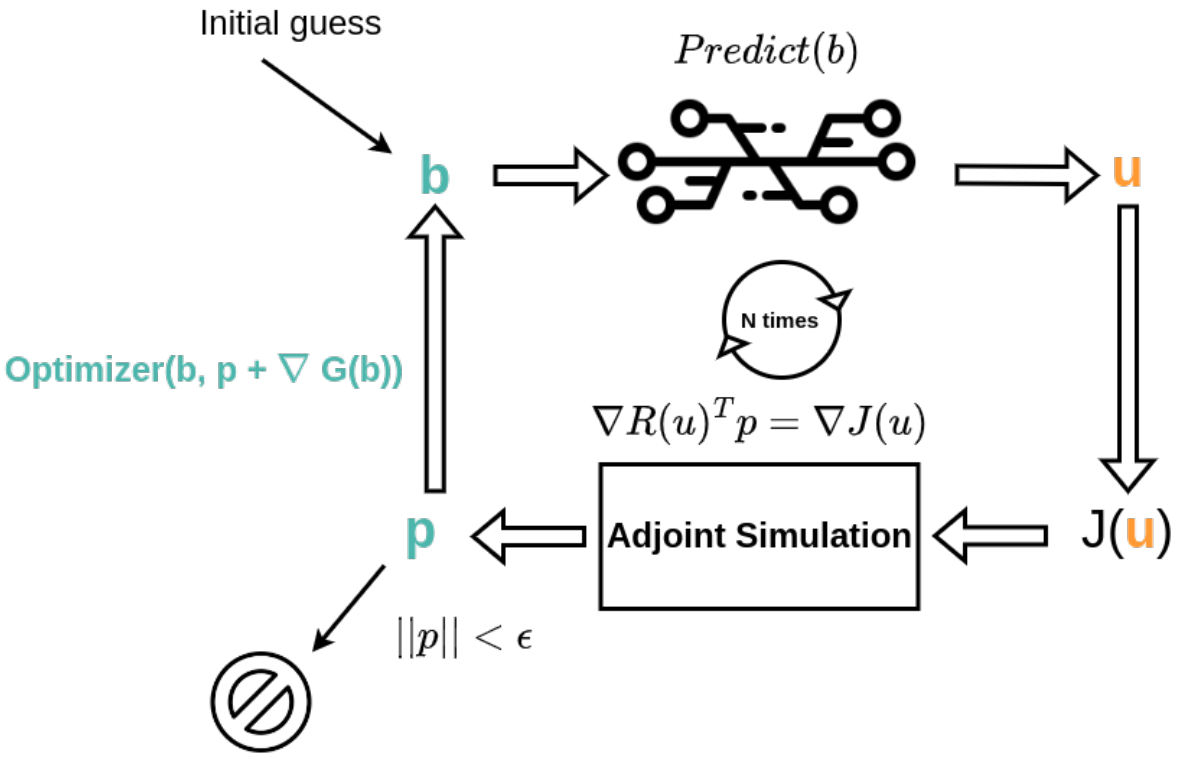


Figure 6.6: Schematic of the algorithm with our solution to remove the first bottleneck. The simulation step has been replaced with an ANN that does the same job.

Introducing a neural network in the loop has advantages over going fast. One of the key advantages is that most deep-learning frameworks use a technique where a program computes a value and a procedure to compute the derivative of that value. This technique is called automatic differentiation and has been key in developing deep learning since one only needs to implement the forward computation to develop new models. The backpropagation algorithm can use the graph given by the automatic differentiation without being explicitly told by the user how to differentiate for each neuron. This remark is of interest since the second bottleneck of the algorithm is the adjoint problem which requires computing derivatives. In the next paragraph, we present our take on improving the speed at which we compute $\nabla\Psi$ to update the control.

Second bottleneck: Adjoint problem Evaluating the adjoint problem is a mandatory step in the algorithm. The step is responsible for computing the gradient of the loss function used to update the control. Gradient computation is achieved by solving the linear system presented in equation 6.8. One term of the loss function ($J(\mathbf{u_b})$) is computed in the space of states, and thus the gradient of this function is expressed in this same state. This

system transforms the gradient of this term ($\nabla J(\mathbf{u_b})$) from the states space to the controls space. Once solved, all the terms are expressed in the control space, and therefore, we can update the control using our favorite first-order optimizer.

We have updated the forward simulation with a neural network that predicts deformations in less than a millisecond for meshes denser than those of interest. We are now stuck with a step that requires solving a linear system to change the base of a gradient. While a single linear solve can be achieved in real-time for decently sized meshes, the algorithm loops multiple times to update the controls. Therefore, we have to solve multiple sequential adjoint problems. The linear solution reduces the algorithm's scalability and speed, which puts us out of the real-time criterion.

In order to speed up the computation, we would like to avoid this system while being able to compute $\nabla\Psi$. Thanks to the neural network added in the loop, we already use a deep-learning framework. Therefore, we can easily use the associated tools, such as automatic differentiation and backpropagation. The backpropagation algorithm presented in section 3.5 relies on the chain rule to compute the gradient of the loss function with respect to any variable involved in the computation. The chain rule does not require solving a linear system but either rely on a graph of computation. This is a significant advantage but also restricts us in the range of tools we can use. In order to keep the chain rule possible, every operation has to be computed in the framework of the neural network (in our case PyTorch). Mainly the loss function has to be implemented using primitives of PyTorch. This said the control is involved in the computation (input of the network), meaning that we can differentiate Ψ with respect to $\mathbf{b}$, thus obtain $\nabla\Psi$. We are leading to our last contribution to the subject, where we replace the adjoint problem with the backpropagation algorithm.

The Figure 6.7 presents our full contribution to the matter where the simulation is replaced by a neural network and the adjoint problem by the backpropagation algorithm.

We have now presented our total contribution and will discuss the results.

6.2.3 Results

To assess the validity of our method, we first consider a toy problem involving a square section beam with 304 hexahedral elements (Figure 6.8). The network is trained using 20,000 pairs $(\mathbf{b}, \mathbf{u_b})$, computed using a Neo-Hookean material law with a Young's modulus $E = 4,500$ Pa and a Poisson's ratio $\nu = 0.49$. We create 10,000 additional synthetic beam deformations, distinct from the training dataset, using the SOFA finite element framework [28]. Figure 6.9 shows three examples of synthetic deformations, along with the sampled point

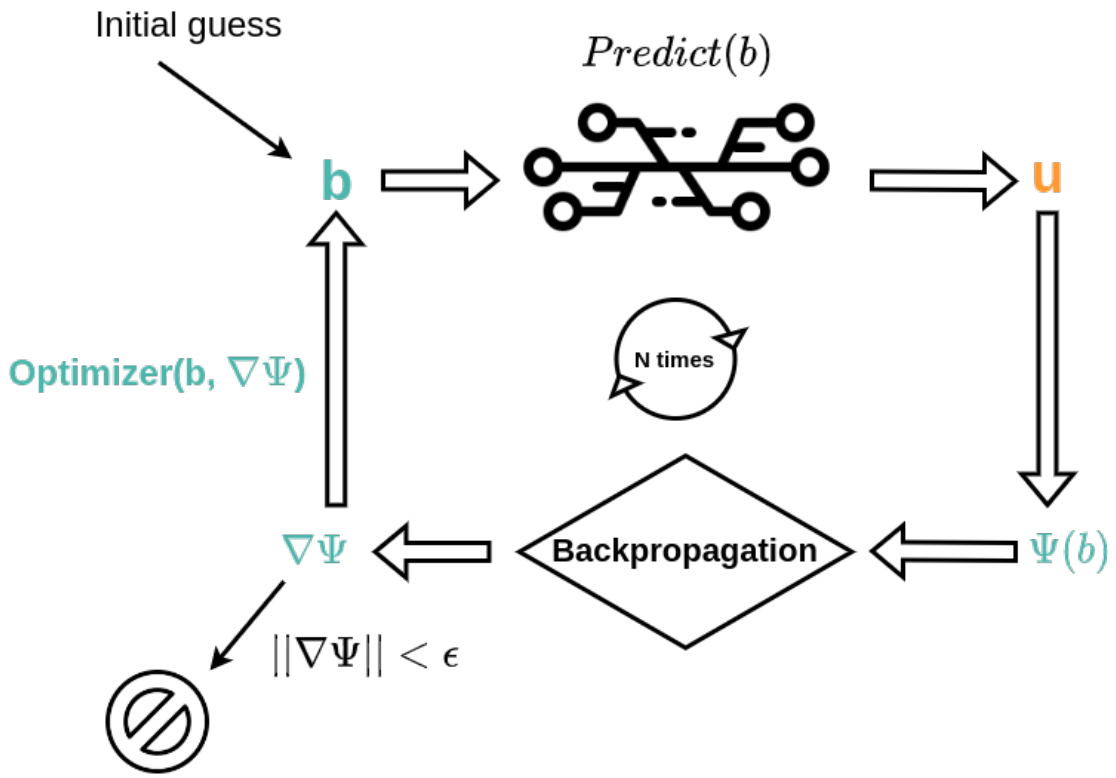


Figure 6.7: Schematic of the algorithm with our solution to remove the second bottleneck. The simulation step has been replaced with an ANN that does the same job, and the backpropagation algorithm has replaced the adjoint system.

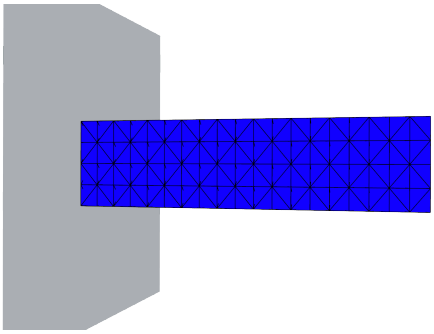
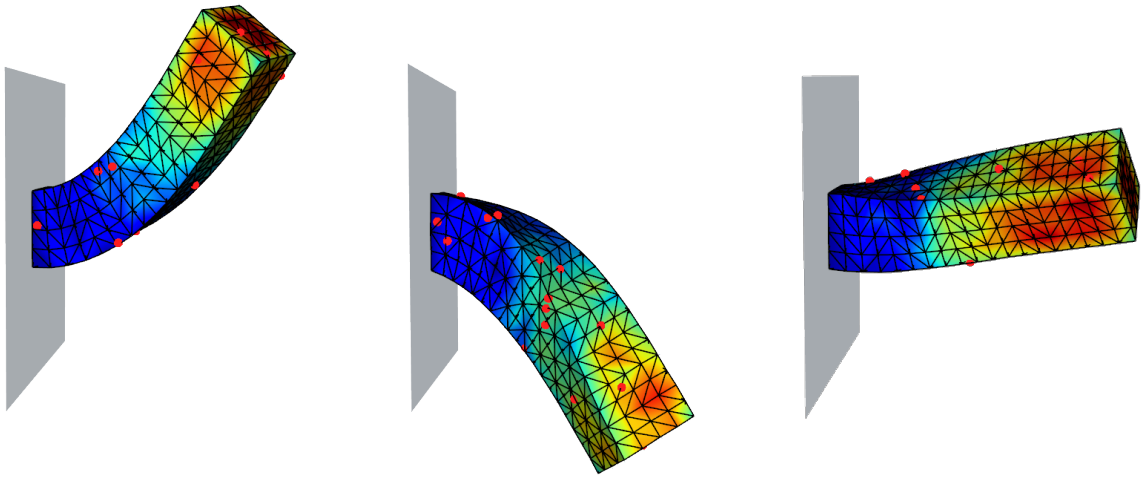


Figure 6.8: Beam used in this section (blue) attached to the grey wall which represents Dirichlet's boundary conditions

clouds. Generated deformations include bending (6.9a), torsion (6.9c), or a combination of them (6.9b). We sample the deformed surface for each deformation to create a point cloud. We then apply our algorithm with a relative tolerance of 10^{-4} on the objective gradient norm. We computed some statistics regarding the performance of our method over a series of 10,000 different scenarios. We obtained the following results: mean registration error: $6 \times 10^{-5} \pm 6.15 \times 10^{-5}$, mean computation time: 48 ms $\pm$19 ms and the mean number of iterations: 27 $\pm$ 11.



(a) TRE: 5.9×10^{-5} , time: 0.07 s, iterations: 13

(b) TRE: 6.6×10^{-5} m, time: 0.09 s, iterations: 15

(c) TRE: 3.4×10^{-5} , time: 0.115 s, iterations: 19

Figure 6.9: Deformations from the test dataset. The red dots represent the target point clouds, and the color map represents the Von Mises stress error of the neural network prediction.

Measurement	Mean	STD	Minimum	Maximum
Registration	5.99×10^{-5}	6.15×10^{-5}	1.30×10^{-7}	6.20×10^{-4}
Computation time	48 ms	19 ms	2 ms	210 ms
Number of iterations	27	11	0	122

Table 6.1: This table gathers statistics for 10,000 test cases and presents registration errors, number of iterations, and computation times (in ms).

Using a FEM solver, each sample of the test dataset took between 1 and 2 seconds to compute. This is primarily due to the complexity of the deformations as shown in 6.9. Such displacement fields require numerous costly Newton-Raphson iterations to reach equilibrium. The neural network provides physical deformations in less than a millisecond regardless of the complexity of the force or resulting deformation, which highly improves the computation time of the method. From our analysis, the time repartition of the different tasks in the algorithm is consistent, even with denser meshes. Network predictions and loss function evaluations represent 10% to 15% each, and gradient computations represent up to the last 80% of the whole optimization process. This allows us to reach an average registration error of 5.37×10^{-5} in less time than it takes to compute a single simulation of the problem using a classic FEM solver. Such error represents an excellent mesh fit in the point cloud as shown in 6.9.

Due to the beam shape symmetry, some point clouds may be compatible with several deformed configurations, resulting in wrong displacement fields returned by the procedure. However, our procedure achieved a satisfying surface matching in each case. These results on a toy scenario prove that our algorithm provides fast and accurate registrations.

In the next part, we apply our method in augmented surgery with the partial surface registration of a liver and show that with no additional computation, our approach produces the forces that generate such displacements with satisfying accuracy.

We now turn to another test case involving a more complex domain. The setting is similar to [78, Sect. 3.2]. In this context, a patient-specific liver mesh is generated from tomographic images, and the objective is to provide augmented reality by registering, in real time, the mesh to the deformed organ. Only a partial point cloud of the visible liver surface can be obtained during the surgery. The contact zones with the surgical instruments can also be estimated.

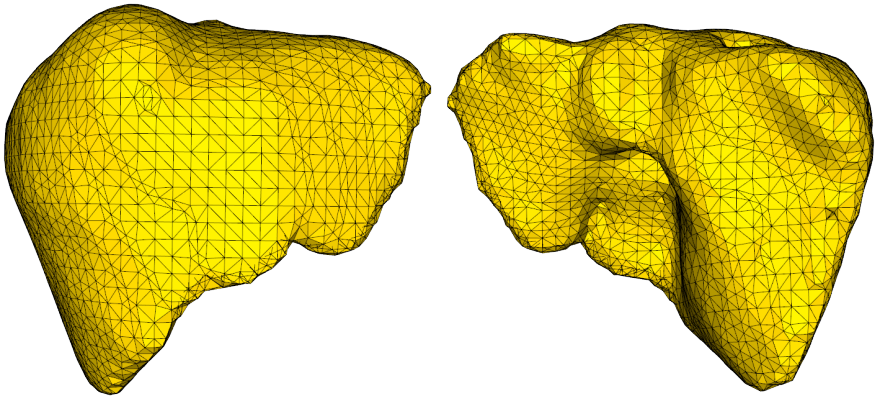


Figure 6.10: Mesh of the liver used in this section. Composed of 3,046 vertices and 10,703 tetrahedral elements, which represents a challenge compared to the one used in 6.9

In our case, the liver mesh contains 3,046 vertices and 10,703 tetrahedral elements. Homogeneous Dirichlet conditions are applied at zones where ligaments hold the liver and at the hepatic vein entry. Like previously, we use a Neo-Hookean constitutive law with $E = 4,500$ Pa and $\nu = 0.49$, and the network is trained on 20,000 force/displacement pairs. We create five series of synthetic deformations by applying a local variable force distributed on a few nodes on the liver mesh boundary. Each series is composed of 50 incremental displacements and the corresponding point clouds. The network-based registration algorithm updates the displacement field and forces between two frames. We also run a standard adjoint method involving the Newton algorithm to compare with our approach. As the same mesh is used for data generation and reconstruction, the Newton-based reconstruction is expected to perform well.

liver partial surface matching for augmented surgery

This subsection presents two relevant metrics: target registration error and computation times. In augmented surgery, applications such as robot-aided or holographic lenses require accurate calibrations that rely on registration. One of the most common metrics in registration tasks is the target registration error (TRE), which is the distance between corresponding markers not used in the registration process. In our case, we work on the synthetic deformation of a liver. Thus, the markers will be the nodes of the deformed mesh.

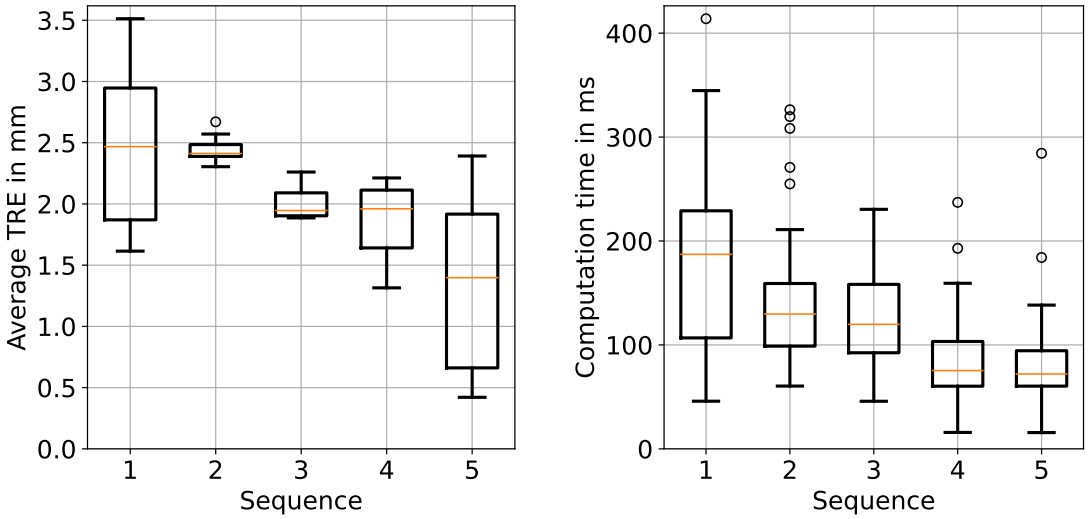


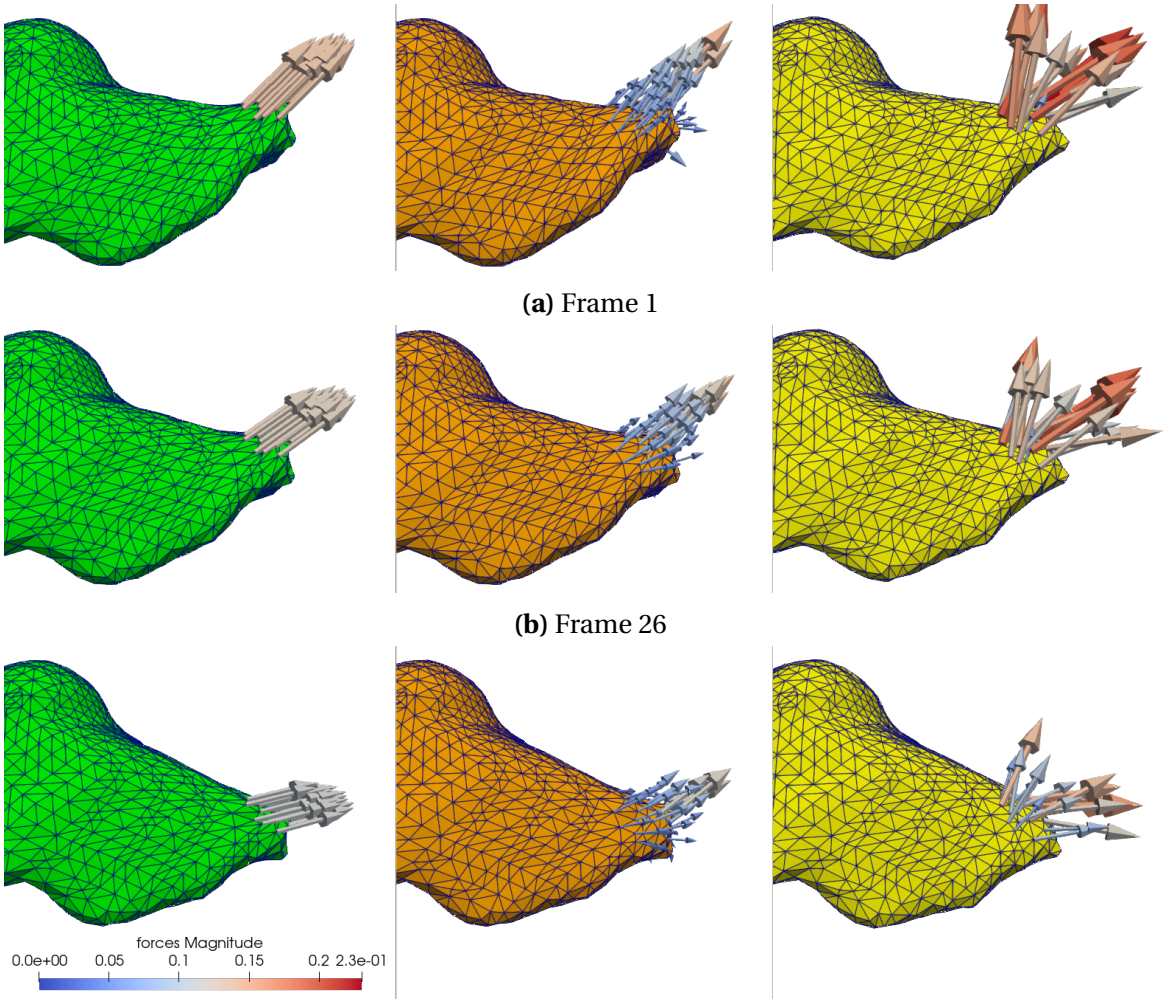
Figure 6.11: Average target registration error and computation times of each sequence.

The five scenarios present similar results with TRE between 3.5 mm and 0.5 mm. Such errors are entirely acceptable and preserve the physical properties of the registered mesh. We point out that the average TRE for the classic method is around 0.1 mm, which shows the impact of the network approximations.

Due to the nonlinearity introduced by the Neo-Hookean material used to simulate the liver, we need multiple iterations to converge toward the target point cloud. Considering the complexity of the mesh, computing a single iteration of the algorithm using a classical solver takes multiple seconds, which leads to an average of 14 minutes per frame. Our proposed algorithm uses a neural network to improve the computation speed of both the hyper-elastic and adjoint problems. The hyper-elastic problem takes around 4 to 5 milliseconds to compute while the adjoint problem replaced by the backpropagation algorithm takes around 11 ms. This leads to significant improvement in convergence speed as seen in 6.11 where we reduce the computation time by a factor of 6,000 on average.

6.2.4 Force estimation for robotic surgery

In the context of liver computer-assisted surgery, the objective is to estimate a force distribution supported by a small zone on the liver boundary. Such a local force is, for instance, applied when a robotic instrument manipulates the organ. In this case, it is critical to estimate the net force magnitude applied by the instrument to avoid damaging the liver.



(a) Frame 1

(b) Frame 26

(c) Frame 50

Figure 6.12: Synthetic liver deformations and force distributions (left), reconstructed deformations and forces using the Newton method (middle) and the network (right) for test case 3.

To represent the uncertainty about the position of the instruments, the reconstructed forces are allowed to be non-zero on a larger support than the original distribution, as shown in Figure 6.13.

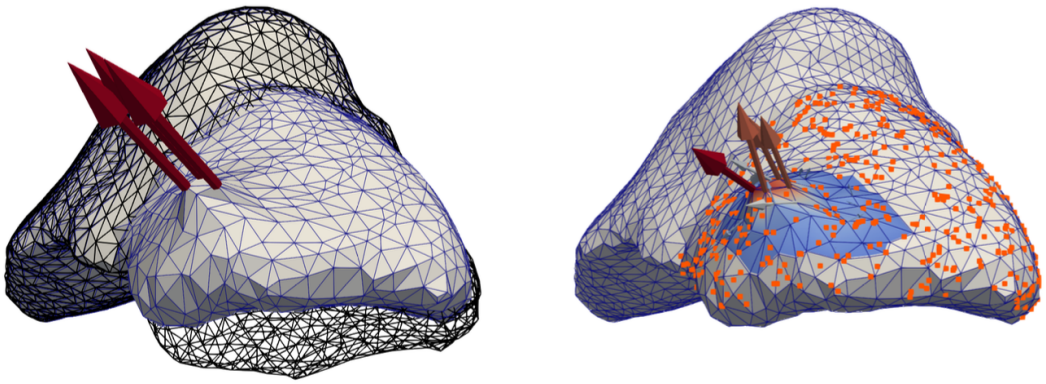


Figure 6.13: On the left-hand side, the original simulation that produced the point cloud used in the registration of the right-hand-side picture. A registration is performed on the right-hand side to match the point cloud. The support in blue is larger than the original zone where forces are applied, yet, the optimization gives forces similar to the original.

Figure 6.12 shows the reference and reconstructed deformations and nodal forces for three frames of the same series. While the Newton-based reconstruction looks similar to the reference one, network-based nodal forces are much noisier. This is primarily due to the network only approximating the hyper-elastic model.

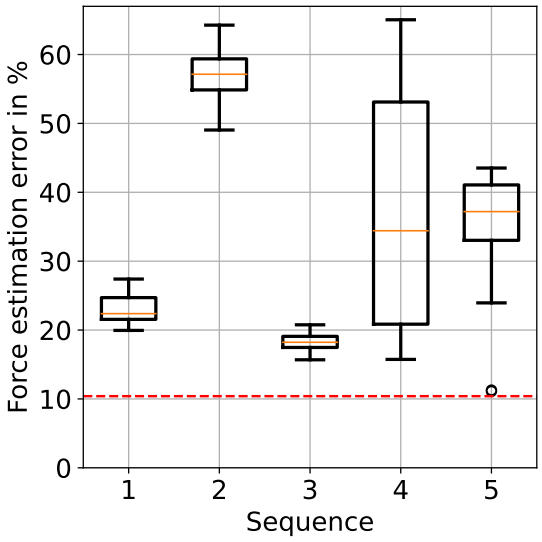


Figure 6.14: Force estimation error of the five sequences using our method, in red the average force reconstruction error with the classical method.

The remarkable improvement in speed comes at the cost of precision. The neural network provides noisy force reconstructions, as shown in 6.12. This is primarily due to prediction errors since the ANN only approximates solutions. These errors also propagate through the backward pass, thus, accumulate in the final solution. Although the force estimation is noisy for most cases, it remains acceptable as displayed in 6.14. The red

dotted line corresponds to the average error obtained with the classical adjoint method (10.04 %). While we are not reaching such value, some sequences, such as 1 and 3, provide good reconstructions. The difference in errors between scenarios is due mainly to training force distribution. This problem can be corrected by adding more data to the dataset, thus providing better coverage of the force and deformation space.

These results show that this algorithm can produce fast and accurate registration at the expense of force reconstruction accuracy. This also shows that the force estimation is not directly correlated to registration accuracy. For example, sequence 1 has the worst TRE but a better force reconstruction than sequence 4.

Finally, these results are computed under the hypothesis of a perfect rigid registration and surface acquisition. We will now discuss the impact of these hypotheses on a single example where we will vary the amplitude of the noise for both parameters.

Impact of the hypothesis

Our presented results were computed assuming perfect rigid registration of the liver and perfect acquisition of the surface. These assumptions can be considered strong thus to finalize our work we have to study the impact of the registration and surface acquisition on the method.

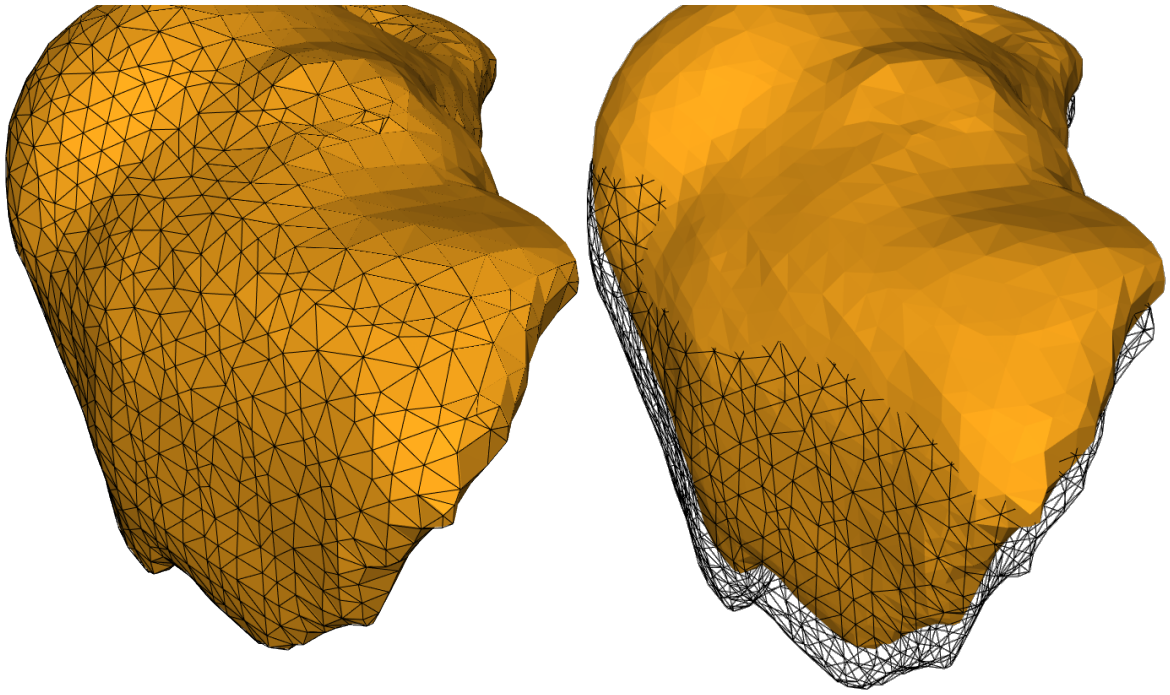
The study will consider a realistic scenario where noise appears in the rigid registration and target. We will then analyze its impact on the method by discussing the fluctuation of the TRE and force reconstruction.

Our first hypothesis rely on the fact that we are able to compute a perfect rigid registration before we start our non-rigid registration process. Since perfect registration do not exist we want to study the impact such error on the final registration and force reconstruction. To do so we consider a transformation matrix $T \in \mathbb{R}^{4\times4}$ as follows:

$$\begin{pmatrix} \mathbf{R} & \mathbf{t} \\ 0 & 1 \end{pmatrix}$$

Where R is a rotation matrix and $\mathbf{t}$ a translation vector. The rotation matrix and translation are generated using an unbiased Gaussian noise with variance between 10^{-4} and 10^{-3}. The transformation matrix T is then multiplied to the position vector of the mesh which will apply a rotation and translation of the geometry. To represent our second hypothesis on target acquisition, we add a Gaussian noise vector with the same parameter as the previous transformation to the target position. We chose this range of variance since this kind of noise represents important error on the registration as shown in Figure 6.15. Fur-

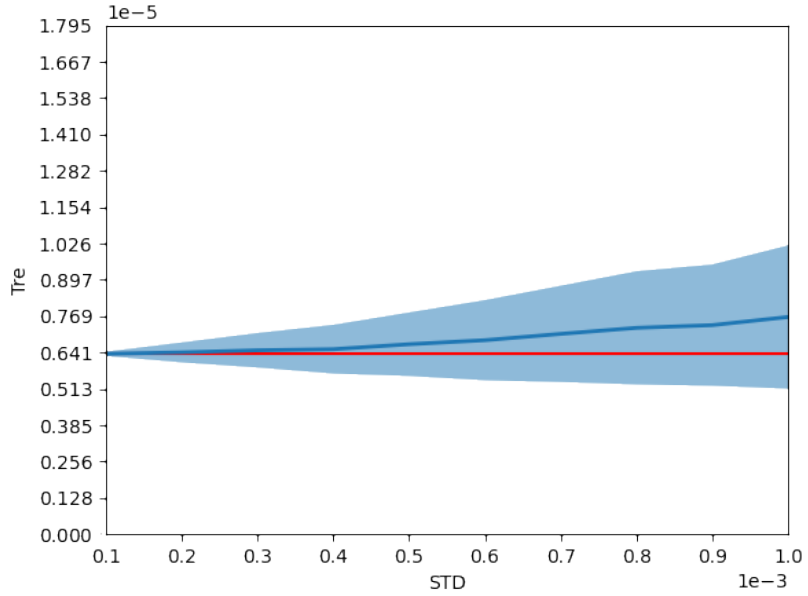
thermore, it represents the range in which we expect this method to be used for realife application cases.



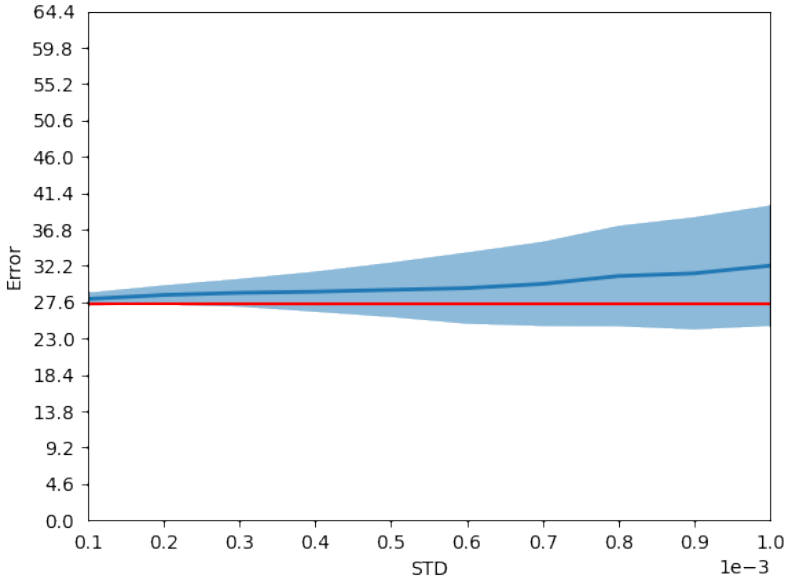
(a) $\sigma = 10^{-4}$ **(b)** $\sigma = 10^{-3}$

Figure 6.15: Two examples of noisy rigid registrations used in this study with in yellow the reference position, in black the noisy rigid registration. The left and right-hand side is generated using an STD of 10^{-4} and 10^{-3} respectively.

This study will compare the TRE and force reconstruction of a single frame where the model is subject to random rotation and translation and target is subject to noise. To give a point of comparison we want to first present these value in the case of perfect rigid registration and target. For this example in the perfect scenario our method reach a TRE of 6.35×10^{-6} and a force reconstruction error of 27.34% . For this study we computed 10,000 registrations. These samples are distributed in 10 classes of uniformly distributed STD ($\{1.0\times10^{-4}, 2.0\times10^{-4}\cdots, 1.0\times10^{-3}\}$). For each class we have drawn 1,000 random rotation matrices, translation and noise vectors with the said STD then proceed to reconstruct the force with the presented method.



(a) Target registration error



(b) Force reconstruction error in %

Figure 6.16: Top : Graph of the target registration error as a function of the STD of the gaussian noise. Bottom : Graph of the error on the force reconstruction as a function of the standard deviation of the gaussian noise. Dark blue represent the mean of the samples, while light blue represent the standard deviation of the measured quantity. The red line represent the value computed on the noise-free ground truth.

Let us first consider the target registration error in Figure 6.16a. The average TRE appears to slowly grow with the noise going from 6.4×10^{-6} to 7.5×10^{-6} . This represent

an increase of approximately 10% when the noise increases by 1000% which tells us that our method is on average resilient to the noise in the expected application range. The standard deviation of the error also increases with the noise growing from 8.13×10^{-8} to 2.53×10^{-6} . While this represents an important relative growth ($\times 31$) at any point in our test the standard deviation is multiple orders of magnitude smaller than the input noise. The average and standard deviation of the target registration error are slightly affected by input noise, yet, our algorithm keeps predicting an acceptable displacement fields. This proves that our method is resilient to the expected noise and thus could be a solution to the initial problem for the nonrigid registration aspect of the problem.

We now focus our attention on the force reconstruction aspect of the method (Figure 6.16b). We can observe a pattern similar to the TRE. The average reconstruction error starts with an error slightly higher than the baseline with 28.0% and slowly increase to 32.2% . Here also our method has a relative growth two orders of magnitude smaller than the noise proving that even for the force reconstruction the average solution is resilient to the tested noise. Finally, the standard deviation of the reconstruction error increases from 0.83% to 7.65% . Such results suggest that the force reconstruction is more prone to producing outliers and therefore might not be applicable in all range of noise.

Overall for acceptable rigid reconstruction noises our method is able to perform an efficient fitting of the observed data with an expected TRE between 6.4×10^{-6} and 1.0×10^{-5} which remain acceptable for most applications. Although the force reconstruction performs well on average this the standard deviation of this quantity is more susceptible to the noise and thus might not be applicable when the data are not clean enough.

This chapter concludes our contributions to this thesis. The next chapter presents unpublished but promising results on a new concept called differentiable solver. We start by presenting what is a differentiable solver and how we implemented it using PyTorch. We finish by presenting the possibilities of this new concept by playing with different optimization parameters.

CHAPTER 7

LASTEST OPTIMISATION TOOL: DIFFERENTIABLE SIMULATION

7.1 DiffEn : A differentiable solver based on energy 105
7.2 Results and future works 118
8.1 Summary and achievements 135
8.2 Outlook and futur work 137

We have presented our approach at combining the optimal control framework with deep learning. We have seen that most of the error on the force reconstruction comes from the fact that the network only approximates the correct solution. Thus, might require a better training that what is already done.

A better training can be achieved by either letting the learning run much longer at the risk of over fitting the dataset or giving the network a better feedback about the prediction errors. We have presented the physics informed loss function that aim at giving information about the physical properties of the prediction thus improving the feedback about the predictions. Yet, similar to the data-driven learning the loss function will most likely not reach zero during training. The optimization of the training process will minimize the supervised terms and the residual terms as much as possible but non-zero contribution can remain. In that sense, the physical constraints only represent soft constraints, without guarantees of minimizing these.

The ease of access to the different derivative of the model is one strong point of such method. On the other side we are constrained by the learned representation regarding the reliability of these derivatives. Also, each derivative requires backpropagation through the full network and as we presented in the previous section this can be time-consuming.

Finally, the setup is simple but quite difficult to control. The model can refine the solution by itself but requires using multiple technics to make sure it is focusing on the regions of interest.

Removing this drawback requires going to the next step and writing a simulation engine using automatic differentiation. This will allow to easily and naturally incorporate deep learning in simulation without only relying on blackbox generated data and a physical loss function. Such simulators are called differentiable simulators, and, in our case, differentiable physics simulator.

The goal of such methods is to use existing numerical method and reimplement them using automatic differentiation framework. This allows to differentiate them with respect to their inputs by running the backpropagation algorithm to let the gradient flow through the simulator / neural network. This as multiple advantage such as improved feedback and generalization, but also being able to solve larger classes of inverse problems very efficiently.

Before we start presenting our results on the subject of differentiable simulators we want to mention that it already exists multiple example of such physics engines. Yet we haven't found one that satisfy our needs. For example, DiffCloth [59] only works for cloth-like objects, REDMAX [116] is designed for articulated bodies and PhiFlow [42] is poorly designed for mesh deformation and would require a lot of workaround. The closest work to our goal is GradSim [46]. This framework allows computing FEM simulations but only

on tetrahedron meshes made of incompressible NeoHookean material which greatly reduce its use cases.

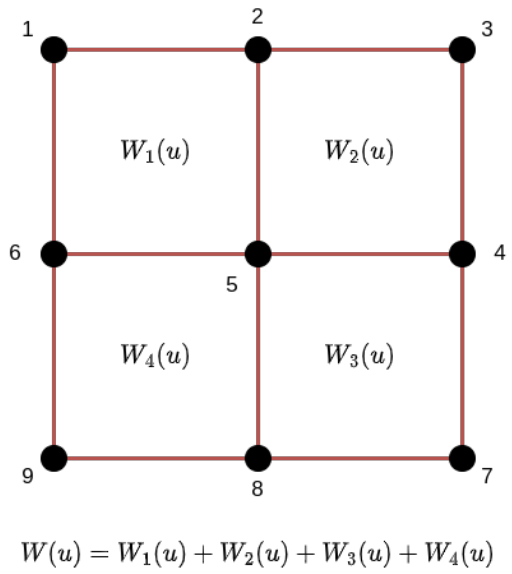
In this section we present unpublished promising results on the development of a FEM differentiable simulator that is based on energy formulation. The simulator developed in PyTorch supports linear and quadratic tetrahedron and hexahedron as well as multiple linear and nonlinear materials. We start by formulating the FEM elastic problem in terms of energy and see how we derive it to fall back on equation 2.35. We then present how we efficiently compute the different members of the equation in PyTorch. Once all members are computed, we present the workflow of the engine and how it optimizes the corresponding variables. Finally, we will present how to use it to solve complex forward and inverse problems.

7.1 DiffEn : A differentiable solver based on energy

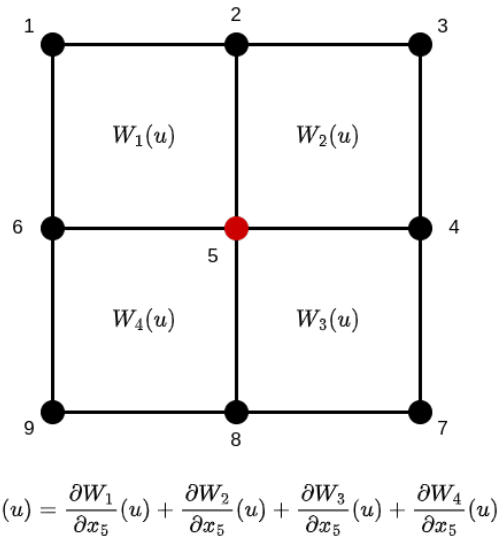
7.1.1 Energy formulation of the elastic problem.

In the FEM framework, the global energy is computed by accumulating local energy over each element. Each local energy is an independent scalar. Therefore, the computation is highly parallelizable and memory efficient, perfect for GPU computation.

These affirmations are not true when discussing internal forces deriving from internal energy. Internal forces are nodal quantities that derivate from an element quantity. As shown in Figure 7.1, their computation requires adding the participation of each element to which the node belongs, thus, creating memory access conflict and reducing the computation speed.



(a) Computation of the global energy.



(b) Computation of the internal force associated with the fifth node.

Figure 7.1: Difference in computation, figure a) computes the global energy while figure b) computes the force for a single point of the mesh. Force computation requires to use scatter operator, which is not vectorizable; hence, slower than simply computing the energy.

Since our engine is written in PyTorch and this framework allows easy computation on the GPU, we preferred to work with energy to maximize the performance. We now present in detail how the elastic problem is formulated in terms of energy and how we can fall back on equation 2.35 from it.

Let us consider a domain $\Omega \subset \mathbb{R}^3$ filled with an elastic material. The global elastic energy or internal elastic energy is computed by accumulating the local elastic energy over the whole material. Namely, the global elastic energy reads:

$$W(\mathbf{u}) = \int_{\Omega} w(\nabla \mathbf{u}) dx \tag{7.1}$$

where w is a function of $\nabla \mathbf{u}$ representing the local elastic energy invariant by any rigid transformation. The deformation energy is minimal when the domain is at rest ($\Omega = \Omega^0$). The energy cost to bring the material from its rest shape to a given displacement $\mathbf{u}$ is given by $W(\mathbf{u}) - W(\mathbf{0})$.

Deformations are the result of external stress applied to the object using forces. Forces summarize various types of interaction by describing the first-order variation of the total energy around a given displacement $\mathbf{u}$. In static elasticity, a force is represented by a linear potential energy. Consider $\mathbf{b}$ a force distribution; the associated linear potential

energy is as follows:

$$-\langle \mathbf{b}, \mathbf{u} \rangle = -\int_{\Omega} \mathbf{b} \cdot \mathbf{u} dx \tag{7.2}$$

The configuration then gives a stable equilibrium that minimizes the system energy. In other word, the displacement $\mathbf{u}$ generated by $\mathbf{b}$ is a solution to the optimization problem :

$$\min_{\mathbf{u} \in \mathscr{U}} W(\mathbf{u}) - \langle \mathbf{b}, \mathbf{u} \rangle \quad \text{subject to} \quad \mathbf{u}_i = 0 \quad \forall \mathbf{u}_i \in \partial\Omega_D \tag{7.3}$$

where $\mathscr{U}$ is the set of admissible displacements. Feasible displacement in static elasticity involves sets of Dirichlet boundary conditions $\partial\Omega_D \subset \partial\Omega$ to ensure the existence of a solution.

We note that equation 7.3 comprises two terms. As the name suggests, the first one, linear potential energy, is linear in $\mathbf{u}$. The second one $W(\mathbf{u})$ can be either linear or quadratic in $\mathbf{u}$ depending on the material used to simulate the object. One key aspect of $W(\mathbf{u})$ is that $\forall \mathbf{u} \in \mathbb{U}, W(\mathbf{u}) \geq 0$. Meaning that the solution $\mathbf{u}$ will be the value at which the derivatives of the minimized function values at zero. Hence, the minimization problem can be written as follows:

$$\min_{\mathbf{u} \in \mathscr{U}} W(\mathbf{u}) - \langle \mathbf{b}, \mathbf{u} \rangle \quad \Longleftrightarrow \tag{7.4}$$

$$\nabla_{\mathbf{u}} W(\mathbf{u}) - \nabla_{\mathbf{u}} (\langle \mathbf{b}, \mathbf{u} \rangle) = \mathbf{0} \quad \Longleftrightarrow \tag{7.5}$$

$$\mathbf{R}(\mathbf{u}) - \mathbf{b} = \mathbf{0} \tag{7.6}$$

This formulation loops us back to equation 2.35. From here, we know how to solve the system with traditional methods.

We will now present how we compute the different members of equation 7.3.

7.1.2 Efficient computation of energy terms

The presented differentiable simulator works using the PyTorch framework. PyTorch allows running computations on the GPU by simply setting a flag to the correct value. GPUs have been developed to process massively parallel computations. In this context, vectorization of the computation is essential since we want to take full advantage of the hardware architecture. Code vectorization is often a complex process that requires a deep understanding of the problem and programming language. In order to have a better understanding of the fully vectorized source code coming after, we have to introduce some Python and PyTorch tricks.

Topological representation in position space One interesting aspect of Python is the notion of slice. A slice is an unordered collection of unique integers. A slice is also a Python object usually used to access array values. As shown in Figure 7.2, the operation returns a subset of the previous array where the values are ordered according to the slice order.

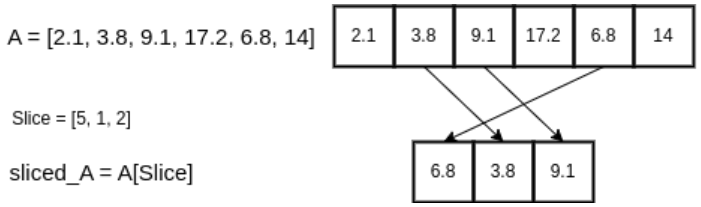


Figure 7.2: Example of a sliced array in Python. Array $sliced\_A$ contains the fifth, first, and second value of array A.

Slices are not limited to one-dimensional collection and can therefore represent a topology. We can use this knowledge to have a topological representation in the position space. By this term, we mean to have an array whose shape is the same as the topology array where instead of indices, we have 3D coordinates as presented in Figure 7.3

This representation is one trick to vectorize the code, but as it often happens, vectorization is a tradeoff between memory and speed. In the given example (Figure 7.3), the values of nodes one and three are duplicated in the array Pos_tri. Therefore, the memory footprint of this representation is quite important compared to the classical $(Position, Topology)$ pair. For example, for a mesh of 40 nodes and 100 triangles, the pair is around 3,600 bytes, while the topological representation in position space weighs around 7,500 bytes. If we consider a hexahedra topology, the weight reaches 19,000 bytes.

We now know that we can build a topological representation of a mesh in the position space by indexing the position array by the topology, but why would we do that? The answer is a vectorization of the energy computation. Each array line is an element we can perform the computation to obtain the local elastic energy. Duplicating the data avoid the random memory access collision and can perform the computation in parallel.

We can now proceed to the next trick, which allows fast, complex vectorized computations.

Einstein notation for vectorized tensor operation Einstein notation or Einstein summation convention is a notational convention that implies summation over a set of indexed terms in a formula. While this is primarily useful when dealing with equations on paper, most mathematical frameworks have implemented a function called *einsum*. This function takes as a parameter a string that describes the operation according to Einstein

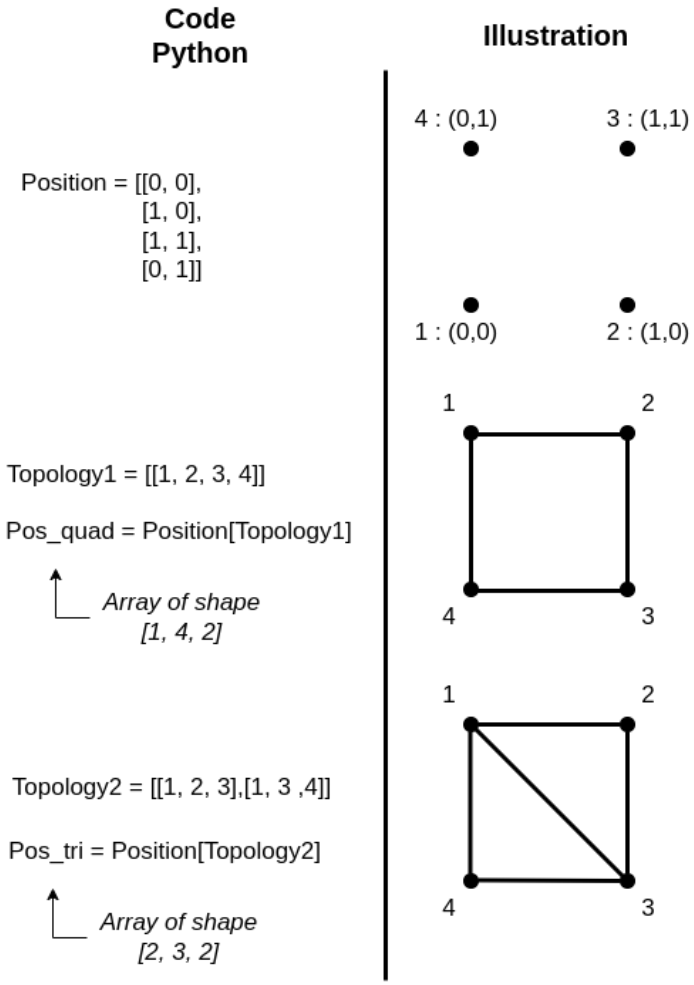


Figure 7.3: Example of two topological representations in position space (Pos_quad and Pos_tri). Indexing the array of positions by the topology creates a topological representation in the position space of the object. We create an array in which the first dimension is the number of elements in the topology, the second is the number of vertex per element, and the third is the number of space dimensions.

notation and a list of corresponding tensors. This explanation might be blurry right now, so let us have a couple of examples.

Assuming that $\mathbf{a}$ and $\mathbf{b}$ are vectors in $\mathbb{R}^n$ and $\mathbf{A}$ and $\mathbf{B}$ are tensors in $\mathbb{R}^{m\times p}$ Figure 7.4 present some basic matrix/vector operations where n, m, p are compatible dimensions.

```
a_inner_b =     einsum('i, i -> ', a, b)       # dot product between a and b
a_outer_b =     einsum('i, j -> ij', a, b)     # outter product between a and b

A_dot_a =       einsum('ij, j -> i', A, a)     # Matrix-vector product between A and a
A_dot_B =       einsum('ij, jk -> ik', A, B.T) # Matrix-Matrix product between A and B transposed.
A_Hadamar_B =   einsum('ij, ij -> ij', A, B)   # Hadamar product between A and B
A_col_prod_b =  einsum('ij, i -> ij', A, b)    # Coefficient-wise multiplication of each column of A by b
A_sum_col =     einsum('ij -> i',A)            # Sum over the colums of the matrix
```

Figure 7.4: Example of the utilization of the enum function. This description of the computation allows for parallel vectorized execution of the operations.

Is the increase in code complexity worth it? Let's compare the execution time of a single matrix-vector product using three different methods. As presented in Figure 7.5, the first consists of implementing the product using two nested for-loops, the second uses the built-in product operator, and the third uses the enum description.

```
import torch


A = torch.rand((1000,1000))
b = torch.rand(1000)
c = torch.zeros(1000)

# Naive product
for i in range(A.shape[0]):
    for j in range(A.shape[1]):
        c[i] += A[i,j] * b[j]

# Built-in matrix-vector product
c2 = A @ b

# einsum version of the matrix-vector product
c3 = torch.einsum('ij, j -> i', A, b)
```

Figure 7.5: The three compared product formulations written using the PyTorch framework.

Given a $1,000 \times 1,000$ matrix, the first method takes around 20 seconds on the GPU. This value seems absurd, but the nested for-loop introduces a lot of synchronization points and data transfer between the CPU and the GPU, dramatically slowing the computation. The built-in method takes a reasonable 9 ms to compute the product, while the enum is four times faster at 2.2 ms per product.

The einsum function is exceptionally versatile in tensor manipulation, as presented in Figure 7.4, where built-in functions are rigid in their behavior. Furthermore, even for simple operations, enum is significantly faster than built-in functions making it a must-have in our code base. Finally, the relative complexity in development only appears during the first few use of the Einstein notation. With a handful of tests, one quickly get used to reading and understanding enum strings.

We now present how we compute the total elastic energy using these two tricks.

Elastic energy In order to compute the total elastic energy, we first have to compute the element-wise elastic energy and then sum over all elements according to equation 7.1. Computing local elastic energy requires information about the local deformation variation, i.e., we need to compute the deformation gradient. We know from equation 2.4 and equation 2.9 that we can use the position and derivative of the shape function to compute

the deformation gradient $\mathbf{F}$.

We have	$\mathbf{F} = \mathbf{I} + \nabla_X \mathbf{u}$
And also	$\nabla_X \mathbf{u}_e = \sum_{i=1}^{n_e} \mathbf{u}_i \otimes \nabla_X \mathcal{N}_i$
Furthermore,	$\mathbf{x} = \mathbf{X} + \mathbf{u}$ and $\mathbf{I} = \mathbf{X} \otimes \nabla_X \mathcal{N}$
Hence we can write,	$\mathbf{F} = \mathbf{x} \otimes \nabla_X \mathcal{N}$

From this mathematical formulation, we can now write the associated Python code using the two presented tricks. We first create a topological representation in our object's position space, then apply the tensor product on the corresponding dimension.

```
pos_topo = position[topology]
F = torch.einsum('tij, tnjk -> tnik', pos_topo.transpose(2,1), dN_dx)
J = det_3x3(F)
```

Figure 7.6: Computation of the deformation gradient using PyTorch.

Using the presented formulation of the isoparametric element in section 2.2, we can precompute dN_dx. The tensor J represents the scale of volume change; a gain of volume will produce a J bigger than one, whereas a loss of volume is smaller than one. The position tensor is transposed to align the dimensions with the ones of dN_dx.

The deformation gradient is used to compute the right Cauchy-Green strain tensor at the Gauss nodes noted $\mathbf{C}$. Its computation is different if we consider both linear and nonlinear material.

```
# Linear material
C = F.transpose(3,2) + F

# Nonlinear material
C = torch.einsum('tnij, tnjk -> tnik', F.transpose(3, 2), F)
```

Figure 7.7: Computation of the right Cauchy-Green strain tensor using PyTorch for linear and nonlinear case.

The right Cauchy-Green strain tensor is used to compute the Green-Lagrange deformation tensor $\mathbf{E}$.

```
# Green-Lagrange
E = 0.5 * (C - I)
```

Figure 7.8: Computation of the right Green-Lagrange deformation tensor using PyTorch.

Finally, using J and $\mathbf{E}$, we can compute the material elastic potential energy W. This response depends on the material used as suggested by equation 2.4 and equation 2.4.

Before presenting the associated code, we want to mention that Saint-Venant-Kirchhoff extends linear material to essential deformations. Therefore, the difference in the formulation appears in the choice of computation for the Cauchy-Green strain tensor.

```
# Linear material and Saint-Venant-Kirchhoff
trE = torch.einsum('tnii -> tn', E)
E2 = torch.einsum('tnij, tnjk -> tnik', E, E)
Psi =  0.5 * l * trE * trE + 2.0 * mu * torch.einsum('tnii -> tn', E2)

# Neo-Hook
lnJ = torch.log(J)
trC = torch.einsum('tnii -> tn', C)
Psi = 0.5 * mu * (trC - 3.) - mu * lnJ + 0.5 * l * lnJ * lnJ
```

Figure 7.9: Computation of the element-wise incomplete elastic potential energy using PyTorch.

Finally, since we work with isoparametric elements, we have to multiply the element-wise incomplete elastic potential energy by Gauss quadrature nodes weights (w) and the determinant of the Jacobian of the Gauss nodes transformations mapping from the elementary space to the world space ($det(J)$). These values are constant during the simulation if the topology remains constant and are therefore precomputed.

```
element_wise_energy = torch.einsum('tn, tn -> tn', wdetJ, sed)
```

Figure 7.10: Computation of the element-wise elastic potential energy tensor using PyTorch.

Finally, the elastic potential energy is given by summing all the contributions.

```
elastic_potential_energy = element_wise_energy.sum()
```

Figure 7.11: Computation of the total elastic potential energy tensor using PyTorch.

We have presented how to compute the first member of the minimization problem. We will now present the second one, the linear potential energy.

Elastic energy As presented in equation 7.2, the linear potential energy is the integration over the whole domain of an element-wise product, a dot product in our discrete space.

The formulation of this potential then becomes:

```
linear_potential_energy = torch.einsum('i,i->',b,u)
```

Figure 7.12: Computation of the linear potential energy tensor using PyTorch.

We have presented both terms of the minimization function; we now present the computation of the Dirichlet conditions.

Dirichlet boundary conditions The Dirichlet boundary condition are defined by a fixed value for a given node. This value can evolve in time, but, in our case, we work with a static equation; hence we have constant Dirichlet conditions.

In order to solve the problem while keeping the chain rule active, we implement them in a soft manner. The squared norm is added to the minimization function. Thus, equation 7.3 becomes:

$$\min_{\mathbf{u}\in\mathscr{U}} \quad W(\mathbf{u}) - \langle \mathbf{b}, \mathbf{u} \rangle + \|\mathbf{u}_i\|_2^2 \qquad \forall \mathbf{u}_i \in \partial\Omega_D \tag{7.7}$$

$$\min_{\mathbf{u}\in\mathscr{U}} \quad \mathscr{F}(\mathbf{u}) \tag{7.8}$$

As for the last term, a squared l2 norm is a dot product, so the error on the boundary conditions is implemented as such:

```
dirichlet_error = torch.einsum('i,i->',u,u)
```

Figure 7.13: Computation of the Dirichlet boundary conditions tensor using PyTorch.

We conclude this section the Dirichlet boundary conditions. We now present how the engine process these quantities to compute deformations.

7.1.3 Engine workflow

The engine has been designed to achieve two main goals. The first one is to be able to compute the solution to soft-body problems for linear and nonlinear material composed of linear or quadratic elements. The second one relies on the first one and imposes the condition that the solution of the soft-body problem must be differentiable using the backpropagation algorithm.

Having its own soft-body simulation engine is an exciting feat. However, another engine with better performance and flexibility is likely to exist. The gain appears in the second goal, where we want the solution to be differentiable. The gain is two-fold; with this condition, we can train a neural network over a simulation since we can backpropagate the error through the network. We also can use methods similar to the one presented in the previous section 6.2 to optimize several simulation parameters to fit observations.

Solving the soft-body problem presented in equation 7.8 requires assembling the different terms. We implemented a manager called *SystemManager* to do so. The manager

handles two entities. The first ones are potentials, which are the minimized quantities. The second ones are the constraints which are used to represent Dirichlet boundary conditions but not only. The system manager runs as presented in Figure 7.14:

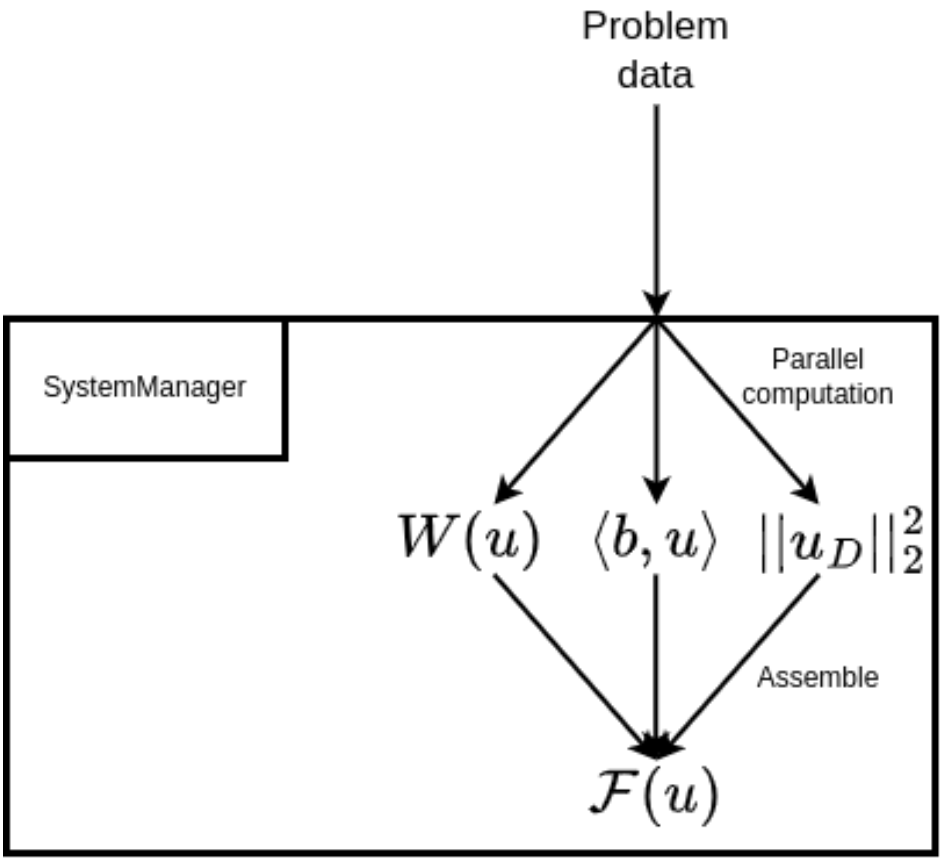


Figure 7.14: Presentation of the SystemManager workflow.

This gives us an evaluation of the system, yet, we want to optimize the displacement field to minimize this function. To do so, we add a step of the Newton-Raphson algorithm after the evaluation as presented in section 2.6. Thus, the schematic of the engine becomes:

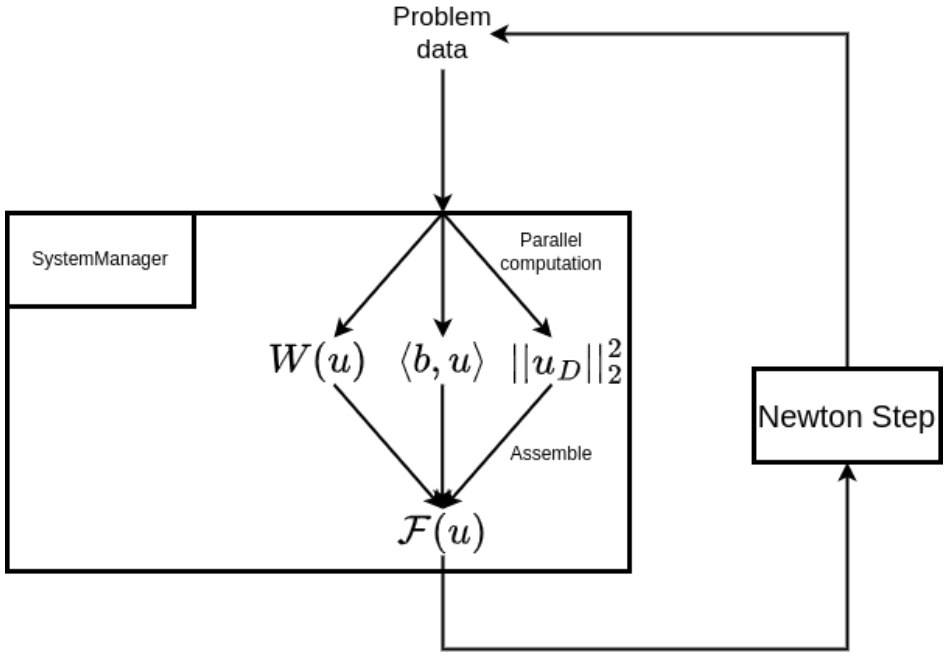


Figure 7.15: Presentation of the differentiable physics engine workflow.

This schematic achieves the first goal of simulating the forward problem and computing the solutions of soft-body simulations. Achieving the second goal is more complicated. The forward problem requires the computation in the *SystemManager* to be differentiable to optimize for the displacement. Having a differentiable solution forces us to implement a differentiable Newton-Raphson algorithm since the resulting displacement field is an algorithm output.

It would be redundant to present the Newton-Raphson code in its entirety since it is implemented as described in section 2.6 In the next paragraph, we present some sticking points when it comes to making the solver differentiable.

Differentiable Newton-Raphson First, we want to mention that the code is inspired by the work of Reuben Feinman [29].

The first sticking point is the computation of the derivatives. Newton-Raphson's steps rely on the computation of the tangent stiffness matrix. This matrix is the derivative of the internal force with respect to the positions. The internal forces are the derivation of the internal elastic energy. Therefore, we need to compute the hessian of the energy with respect to the positions. Doing so requires computing the derivation graph of the energy we are currently differentiating. This is achieved by setting the *create_graph* and *retain_graph* flags to $True$ using PyTorch.

```
f = function_to_minimize(x)
grad_f = autograd.grad(f, x, create_graph=True, retain_graph=True)[0]
```

Figure 7.16: Differentiation of a scalar function (*function_to_minimize*) with respect to the input variable x. The flags *create_graph* and *retain_graph* are set to $True$, meaning that we will be able to differentiate *grad_f* another time to obtain the tangent stiffness matrix.

With this computation, we obtain the internal forces of the domain. The question remains about how to compute the matrix $\mathbf{K}(\mathbf{u})$ from $\mathbf{R}(\mathbf{u})$. The grad function from autograd uses the vector-Jacobian product to compute the gradient of a function. When the function returns a scalar, we do not have to specify the vector in this product since we differentiate the scalar with respect to all the passed variables. When dealing with a vector-valued function, we have to specify which variable of the result we want to differentiate. This is achieved by passing a tensor to the *grad_outputs* variable, its value is by default None, but when vector-valued functions are differentiated, it has to be set.

In our case, we want the first row of $\mathbf{K}(\mathbf{u})$ to represent the gradient of the first coefficient of the internal forces with respect to the positions (i.e. $\frac{d\mathbf{R}(u)_i}{d\mathbf{u}_j}$). Computing the $i-th$ line of $\mathbf{K}(\mathbf{u})$ is done by passing a one-hot vector which values at one at the $i-th$ coefficient. Therefore, we can build the tangent stiffness matrix by passing the identity matrix as the vector in the vector-matrix product. The framework does not accept this. The vector cannot be a matrix, but we can still vectorize the computation using the $vmap$ functional.

The functional $vmap$ takes a function as a parameter and returns a function that accepts parameters with an additional dimension. Meaning that instead of evaluating a function multiple times with a for loop, we can now vectorize the evaluation as shown in Figure 7.17.

```
f = function_to_minimize(x)
grad_f = autograd.grad(f, x, create_graph=True, retain_graph=True)[0]


K_ij = lambda v : autograd.grad(grad_f, x, v, create_graph=True, retain_graph=True)[0]

# Without vmap
K = torch.zeros_like(identity)
for i in range(K.shape[0]):
    K[i,:] += K_ij(identity[i])

#With vmap
K = vmap(K_ij)(identity)
```

Figure 7.17: We use vmap to vectorize the computation of the matrix $\mathbf{K}(\mathbf{u})$. This greatly improves performances compared to the naive for-loop method.

Once again, we set both graph-related flags to true to be able to backpropagate through

the solver once the computation is over.

The second sticking point comes when we want to solve the linear system. Working with energies, we know that $\mathbf{K}(\mathbf{u})$ is a symmetric positive-definite matrix. Therefore, we can compute the Cholesky decomposition and solve the linear problem extremely fast. The Cholesky decomposition and solve can be easily differentiated, as shown by Murray [83]. Differentiating the call is done by PyTorch, that have implemented the corresponding functions *cholesky_ex* and *cholesky_solve*.

With these, we presented how to solve the forward problem, as Figure 7.15 shows, using the Newton-Raphson algorithm, we optimize the displacement field to minimize equation 7.8. Thanks to the multiple tricks presented, the final solution is differentiable. As mentioned, this is important since it means we can optimize other parameters to fit our observations.

The optimization of another parameter is achieved by introducing a computation graph similar to the one presented in Chapter 6. Here the control does not need to be the force; it can be any parameters that are set during the simulation, such as Young's modulus that defines the stiffness of the object or Poisson's ratio that affect the volume change of the objects.

7.1.4 Optimal control using differentiable physics

As presented previously Mestdagh et al. [77] method was mostly limited by computation time. Our contribution has improved this limitation, where we use a neural network instead of a soft-body physics engine to compute the forward pass and the backpropagation instead of the adjoint method. The gain in computation time is significant, but force reconstruction has proven to be noisy. This is primarily due to the neural network only approximating the deformation of the liver. The proposed neural network can be considered a straightforward soft-body physics engine that could handle a single mesh and material and give an approximation of the FEM solution. One key aspect of this engine is that the solution is differentiable, making it possible to optimize for controls, as presented in our results.

We have presented a second physics engine which is more complex and supports any conforming mesh with either linear or quadratic elements and different nonlinear materials. The solution of the engine is also differentiable, meaning that we can swap the neural network for the differentiable engine seamlessly in our contribution. This should lead to an increase in computation time but more pertinent control optimization.

Swapping the engines changes Figure 6.7 to Figure 7.18:

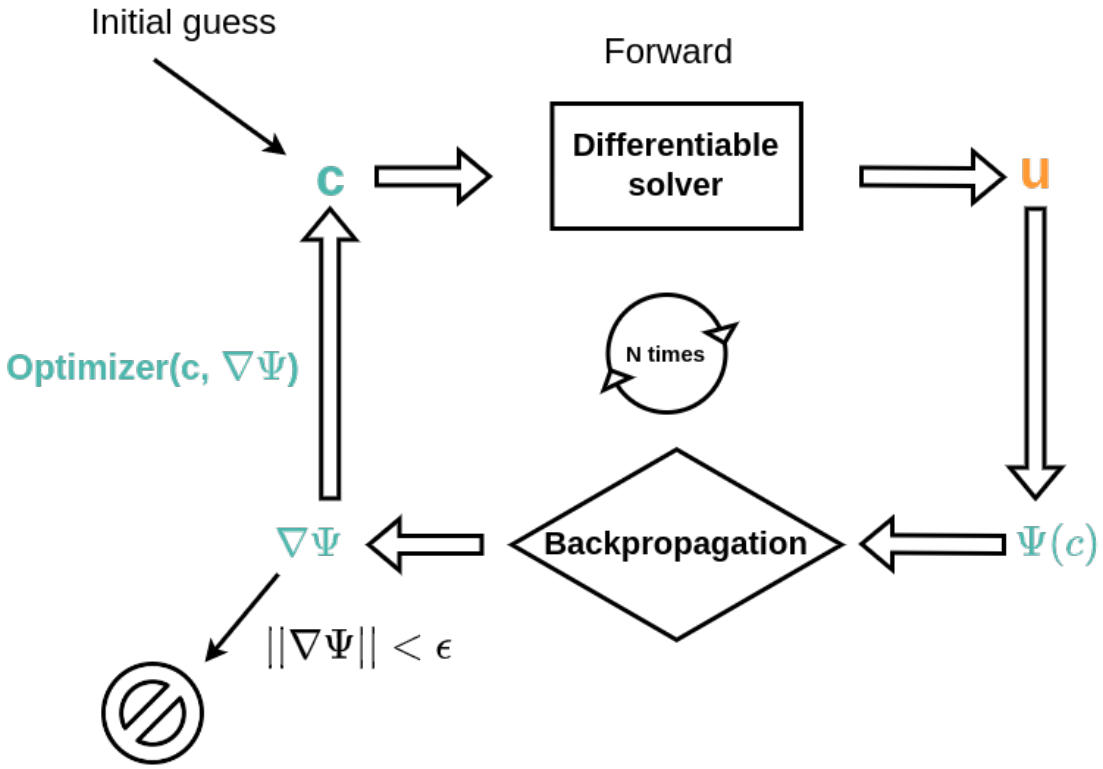


Figure 7.18: Schematic of the algorithm proposed to optimize any simulation parameter c . The neural network of Figure 6.7 has been changed to the presented differentiable solver. Which computes slower but precise deformations according to the finite element framework.

As we can see, most of the computation remains the same, with only the engine part that has been modified. Furthermore, the code is not specific to any particular control. Thus, we can seamlessly optimize the forces, Young's modulus, or Poisson's ratio by setting the correct flag to $True$.

In the following section, we present the type of results we obtain using our differentiable solver. We start by comparing our solution against other FEM solvers and then test our optimization algorithm to retrieve forces, Young's modulus, and Poisson's ratio of simulations.

7.2 Results and future works

After presenting the formulation and implementation of our differentiable solver, we have to prove our claim on the topic.

We will first see if we can compute accurate deformations by comparing ourselves against SOFA Framework [28]. SOFA is a successful physics engine that has been developed since 2006. It gathers 17 years of research in physics simulation and many publica-

tions [6, 12, 20, 106, 124]. It implements multiple physics topics such as solid mechanics, soft-body mechanics [89], fluid dynamics [1], thermodynamic [106]. Given the number of people who have worked and published with the engine, we built trust in the solutions given by its computation and consider it our ground truth.

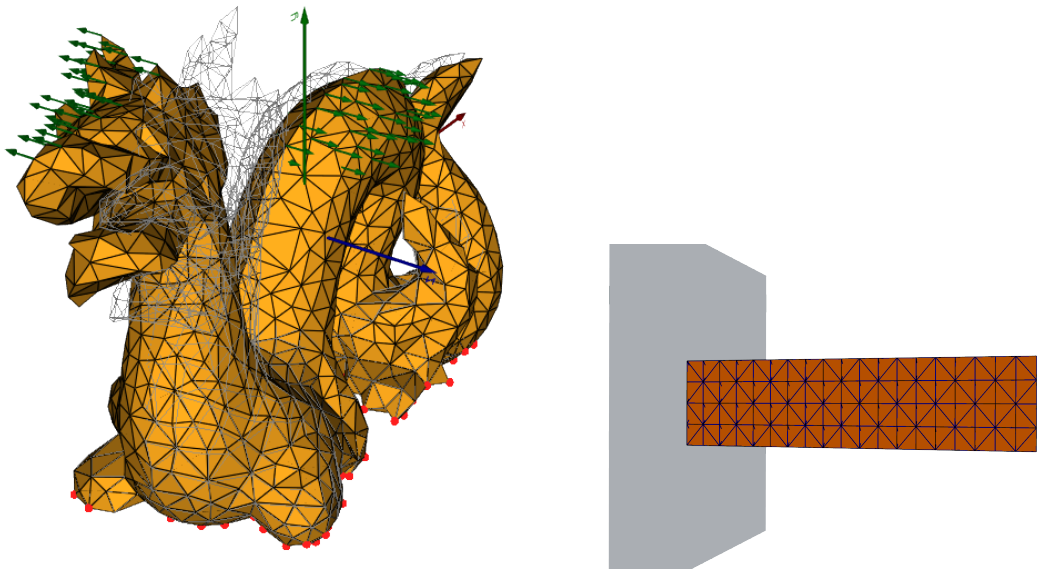
We will then present how the solver performs when used to optimize for simulation parameters. We start by presenting the solver performances on force optimization. In order to get more insight into force optimization, we lead a small study on the effect of multiple parameters, such as initial value and support.

We then proceed to optimize for Young's modulus and Poisson's ratio.

7.2.1 Exactness of the solution

Before optimizing for a given control, we have to ensure that the simulator produces the correct solutions. This verification is done by comparing ourselves against SOFA Framework [28].

The test will consist in computing 1,000 deformations of randomly sampled forces. The deformations will be computed on a square section beam in which width and height measure 0.25m and length 1.0m. Although we are not limited to computing the deformations of a beam (see Figure 7.19), we find this example explicit and intuitive and will use it to present all of our results. The beam is composed of 500 nodes or 1,500 dofs assembled in 304 hexahedra see Figure 7.19.

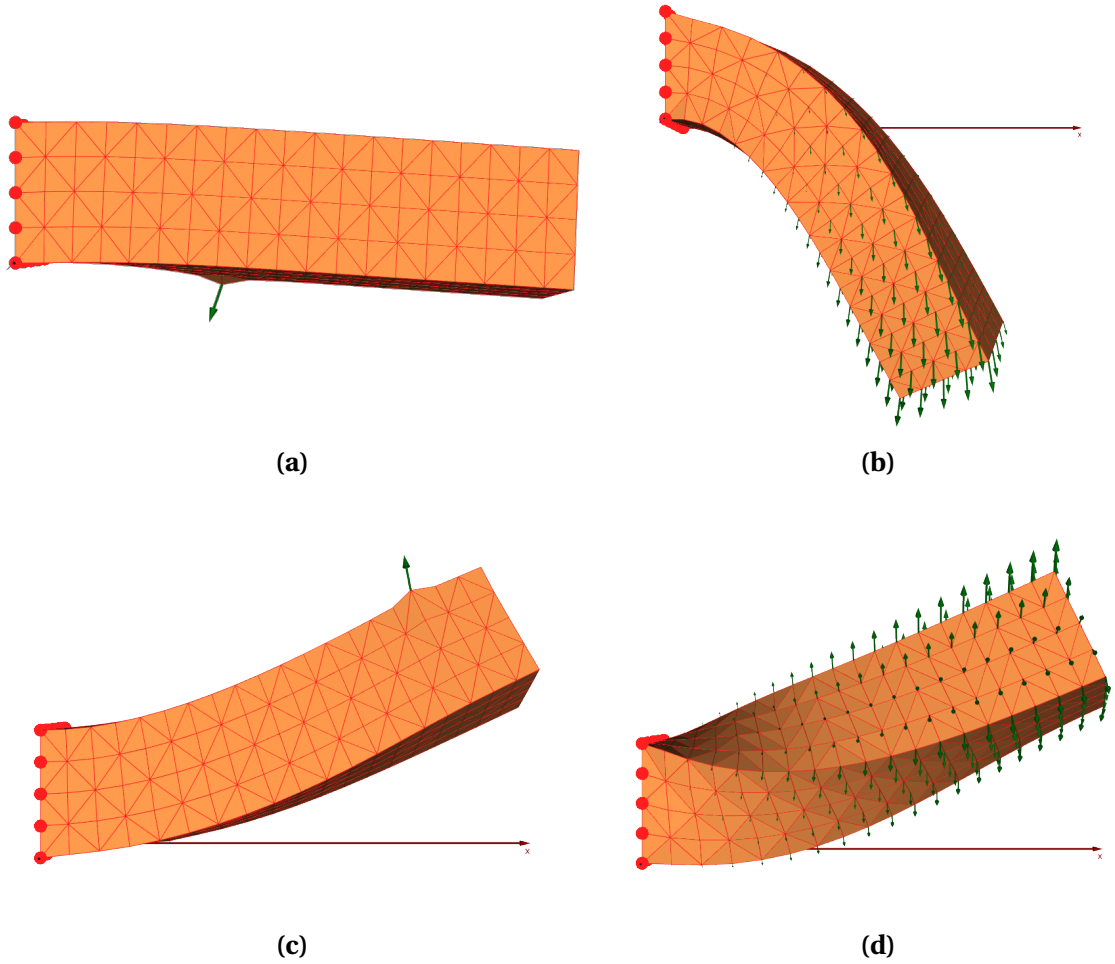


(a) **(b)**

Figure 7.19: On the left-hand side, an example of a deformation computed on a dragon made of tetrahedra. On the right-hand side, the beam used in this section (orange) is attached to the grey wall, representing Dirichlet's boundary conditions. The surface mesh comprises triangles, but the physic is computed on hexahedra.

In terms of material, the beam is filled with a NeoHookean material with Young's modulus of 4,500 and a Poisson's ratio of 0.49. Considering our application case, Yeh et al.[123] showed that the liver Young's modulus values somewhere between 1,000 and 10,000 Pa depending on the strain and fibrosis classification of the organ. Picking 4,500 is a reasonable guess. Finally, Poisson's ratio defines how compressible the object is. With 0, the object has no volume restriction, and with 0.5, the object is incompressible. Although 0.5 is not a value we can input in the computation due to division by 0, we can pick a value that is close to it. Yet, the closer the value is to 0.5, the more unstable the simulation becomes due to dividing by a value close to zero. Organs such as the liver are mostly water, and water is not compressible. Therefore, we chose to go with 0.49, where the simulation can still be complex with a volume that is almost constant.

Before presenting the numbers, we want to display the diversity of the deformations we test in Figure 7.20.



(a) **(b)**

(c) **(d)**

Figure 7.20: Sample of the deformations present in the test dataset. The deformation is heteroclite with bending and twisting from all ranges.

The dataset presents all types of bending varying from low to high amplitude but also different twisting ranges. Some of these deformations are a combination of the two making the simulations even harder to compute. This dataset has been constructed to put our solver to the test, which will enlighten us on its flaws as well as its qualities.

Our comparative study is based on the displacement field. SOFA's Newton solver and ours are set with the same convergence thresholds of 10^{-6} on the norm of the update. Thus, if the object has moved less than a micrometer, we consider the simulation as converged, which at the scale of the simulation ($\approx$ 1 meter) is already strict.

One small detail about the test is that our Newton-Raphson uses a strong-Wolfe line search algorithm to perform an adaptive update of the displacement field, while SOFA Newton-Raphson doesn't. This linear search algorithm dramatically improves the solver stability. On the test dataset, which mainly consists of challenging cases, the solver converges 100% of the time using strong-Wolfe line search and only 20% of the time for the classic update method. This different update pattern slightly changes the final solu-

tion, which is why we expect it to lead to errors in the order of magnitude of the threshold in the results. Constructing the dataset using the SOFA framework, which does not use a strong-Wolfe linear search algorithm, required us to subdivide the input force and apply incremental load. Depending on the subdivision, this incremental load multiplies the computation times by five to ten. While we could have done the same for our solver, it would not bring more information about the exactness of the solution than if we obtained an error in the order of magnitude of the threshold.

We compare and study the differences between the displacement fields produced by the two engines. To do so, we introduce classic error metrics such as the $l2$ norm, node-wise max, mean, and STD error.

Mean $l2$	STD $l2$	Mean node-wise	STD node-wise	Max node-wise
2.6×10^{-6}	1.2×10^{-6}	9.3×10^{-8}	4.5×10^{-8}	2.6×10^{-7}

Figure 7.21: Results of the computation of 1,000 deformations using a square section beam filled with NeoHookean material. As expected, the mean $l2$ is in the range of the threshold.

Over 1,000 deformations, we obtain an error of $2.6 \times 10^{-6} \pm 1.2 \times 10^{-6} m$ which is what we expected. The error is in the order of the threshold with a slight variation given by the strong-Wolfe linear search algorithm.

The results on the node-wise quantities are also satisfying. The node-wise error values are almost negligible with a max error of $2.6 \times 10^{-7} m$.

Now that we have asserted the exactness of the solutions, we can use our differentiable solver to optimize for controls such as force, Young's modulus, and Poisson's ratio. In the next part, we will present our results on optimizing these parameters, starting with the force, then proceed to Young's modulus and Poisson's ratio.

7.2.2 Force as a control

Using the method presented in the previous section (Figure 7.18), we can optimize for a simulation parameter such as the external forces, but not only. In this subsection, we present the results of the optimization of the forces.

We chose to put our work to the test using the previously mentioned dataset. The goal will be to find the forces such that the computed displacement field matches the target. This is achieved using the same approach presented for the liver in Chapter 6 but replacing the neural network by a differentiable solver. We first randomly sample the surface of the deformed mesh to create a point cloud that is used as a target for the optimization. The error to the target is computed using the same metric as the one presented in the said

chapter. We then optimize the forces so the beam fits in the point cloud. The fitting is considered correct if the norm of the gradient of the update is below a certain threshold. In our case, we set this threshold at 10^{-5}.

This study is divided into three parts. We first see if the algorithm can find a solution to fit the target point cloud. We will then study the impact of the initial guess on the computed force. Finally, we will discuss the impact of the support on the reconstructed forces.

Although the two last parts are not the subject of this thesis, we thought they could bring a better understanding of the matter. We limit the presentation to general results since a more in-depth analysis could be a thesis subject.

Force optimization In this paragraph, the support will be the same as the one in the original simulation, and the force is set to $\mathbf{0}$ such that we do not introduce any bias in the optimization. As displayed in Figure 7.22, we can fit the target point cloud by reconstructing a force that generates the corresponding displacement. Since the loss function computes the distance of the point cloud to the surface of the mesh, there is no point-wise identification. We are not certain that the fit corresponds exactly to the original deformation; thus, measuring the force reconstruction doesn't bring pertinent information here.

On average, over 100 forces optimizations, we reach a target registration error of 2.7 ± 10^{-6}. This error is in the same order of magnitude as the exactness of the solution. In conclusion, we are able to compute a force on a given support such that a mesh fits a target point cloud.

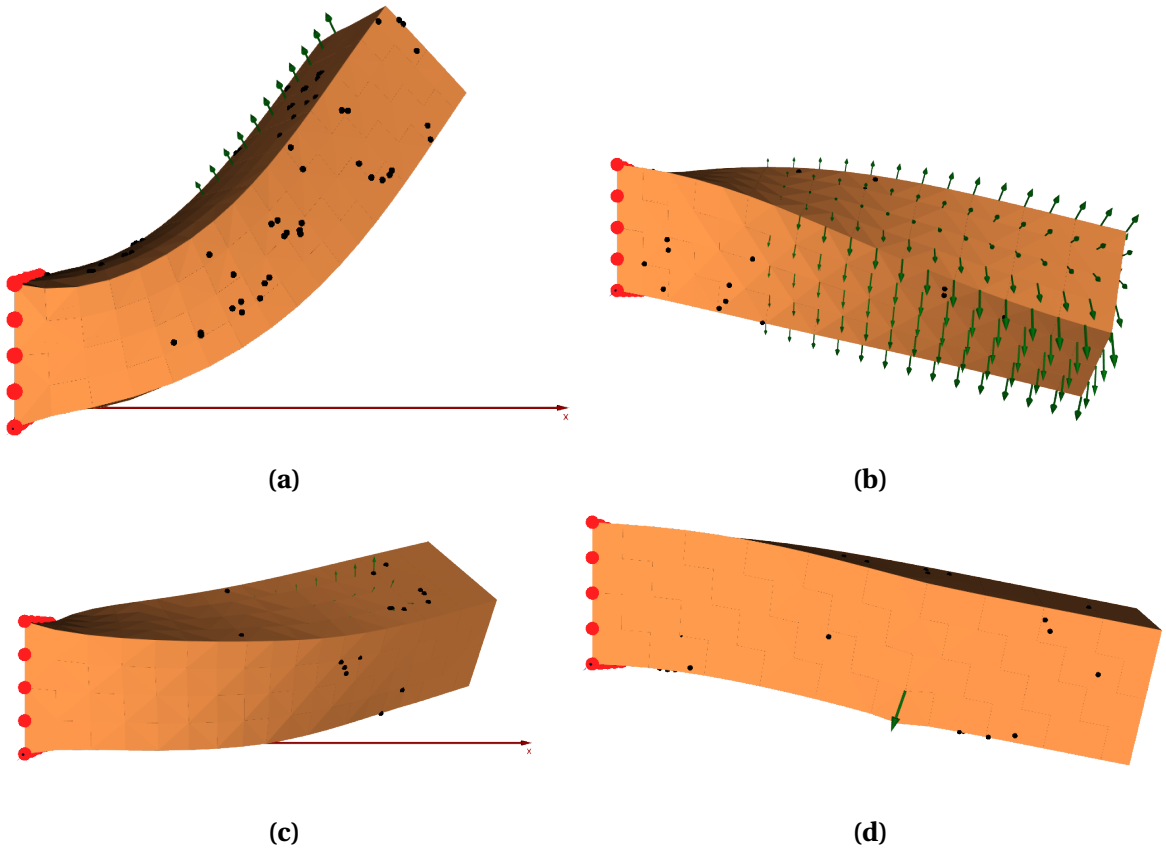


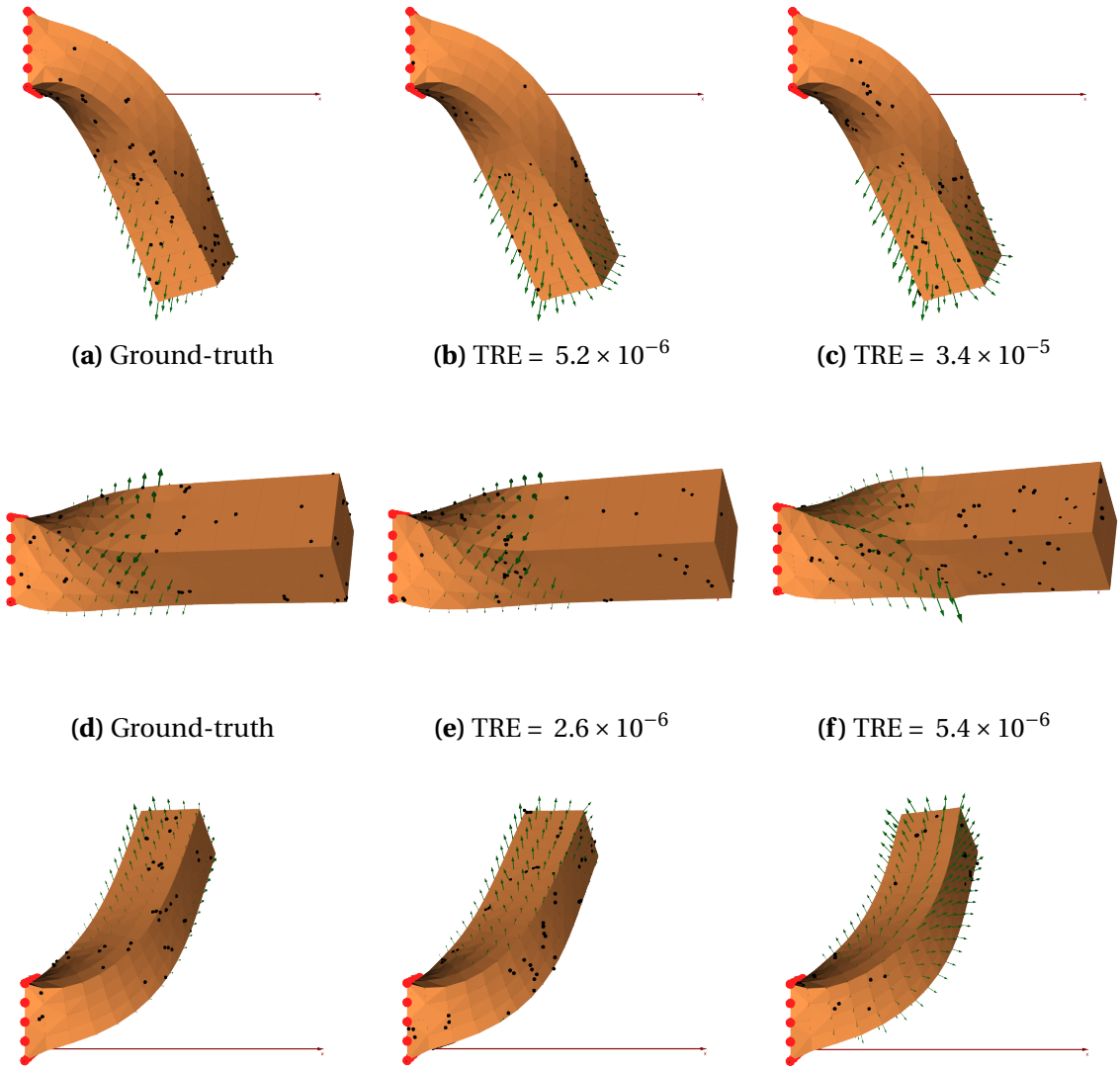
Figure 7.22: Reconstructed forces and resulting deformations. The target point cloud appears in black.

Impact of the initial guess on the reconstruction We previously mentioned the bias that the initial guess of the force vector could introduce. We mean by initial guess the value at which the vector is set before the optimization process.

We will see that this parameter has so much influence that measuring the force reconstruction doesn't make sense. Yet, we can still measure the target registration error to ensure we correctly fit the point cloud.

The test was similar to the one presented in Figure 7.23. Half of the samples were initialized with the opposite of the force, and the other half was initialized with Gaussian noise with parameters $(0, 10^{-2})$.

We tested this over 100 samples from the dataset and found that the TRE has a value similar to the previous test to validate the solution of the solver. The average TRE of 7.4×10^{-6} shows that the solver manages to find an excellent fitting for the observational data but also that the initial guess has a low impact on the TRE. Yet, as we can see in Figure 7.23 despite this excellent fitting the initial value of the force greatly impacts the reconstructed force



(a) Ground-truth **(b)** TRE = 5.2×10^{-6} **(c)** TRE = 3.4×10^{-5}

(d) Ground-truth **(e)** TRE = 2.6×10^{-6} **(f)** TRE = 5.4×10^{-6}

(g) Ground-truth **(h)** TRE = 1.1×10^{-5} **(i)** TRE = 3.8×10^{-5}

Figure 7.23: Example of three force optimization (each row). The left column consists of the groud-truth. The middle column consists of examples where the forces are initialized with Gaussian noise with parameters $(0, 10^{-2})$. The right-hand-side column consists of examples where the forces are initialized with the opposite of the ground truth.

Obtaining a more specific or close-to-ground-truth force reconstruction requires putting as much knowledge as possible into the target registration metric. This can be done by introducing penalization terms with a solution such as the one presented in Chapter 6.

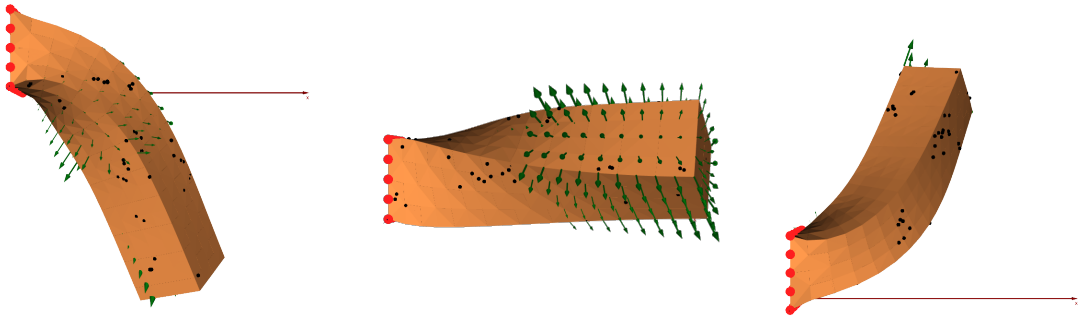
Impact of the support on the reconstruction So far, we have shown that we can fit a point cloud by controlling the force we apply on the object. Even when the initial guess is far from the solution, we can maintain a relatively low TRE.

These tests correspond to a problem where we have perfect knowledge of the location of the applied force but zero knowledge of the force itself. We have shown that even

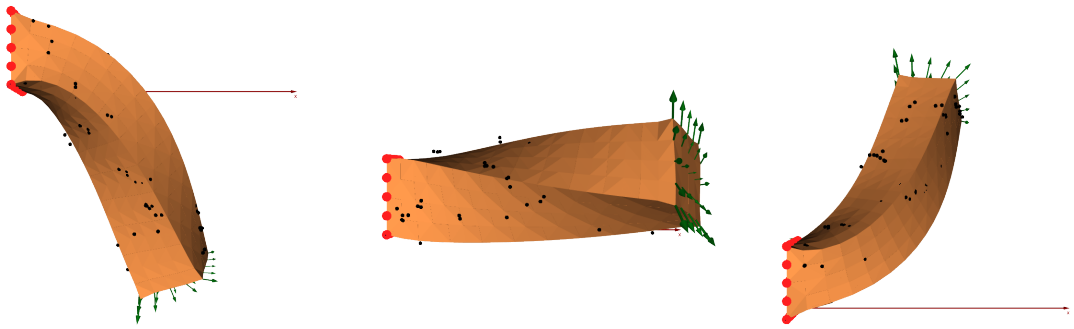
though the force reconstruction is very different, we obtain similar target registration errors.

We now take the opposite hypothesis, where we have a bad knowledge of where the force is applied. To do so, we initialize the force vector to $\mathbf{0}$ and only optimize for a specific location on the object. In this study, the location will be the opposite of the ground-truth support and the tip of the beam.

We now study the impact of the support on the TRE.



(a) TRE = 5.3×10^{-5} **(b)** TRE = 1.4×10^{-4} **(c)** TRE = 1.2×10^{-5}



(d) TRE = 6.4×10^{-5} **(e)** TRE = 5.0×10^{-4} **(f)** TRE = 5.5×10^{-5}

Figure 7.24: Example of three force optimization (each row). The top row consists of examples where the support is the opposite of the original. The bottom row consists of examples where the support is at the tip of the beam.

As presented in Figure 7.24, the choice of support has a lot of impact on the registration. The average TRE is higher than the worst TRE of the previous test over the whole test dataset.

This can be explained by the fact that every support cannot represent every deformation. As an example, if one picks a support that corresponds to a point at the center of the tip of the beam, one will not be able to represent any twisting motion of the said beam since this point, in particular, cannot apply torque on Z-axis. By taking poorly suited support, one will not be able to perform satisfying registration.

Generally speaking, the bigger the support, the better the registration. For the tested sample where the support corresponds to the tip of the beam (illustration in Figure 7.24),

we measure an average TRE three times higher than the one where the support is the opposite of the original support. This can be explained by the solver having not enough variable to play with to fit the observed data.

We used our differentiable solver to optimize the force distribution to fit observational data by setting different support and initial value. We have compared the impact on the TRE of the two parameters and seen than the target registration error is more susceptible to support location than initial value of the force distribution. We now present some results on the Young's modulus and Poisson's ratio optimisation.

7.2.3 Material parameters as a control

We use a similar setup as the one presented in the subsection 7.2.2 The goal will be to find the material parameters such that the computed displacement field matches the one computed by SOFA. The fitting is considered correct if the gradient norm of the update is below a certain threshold arbitrarily fixed at 10^{-5} .

In this study, we consider two material parameters. Since these parameters only impact the material response, the tests we can do are limited. They will mostly consist in setting a first guess far from the actual value and seeing if we can still reach the correct solution. This study's first part focuses on Young's modulus, while the second part focuses on Poisson's ratio.

Although these short studies are not the subject of this thesis and only aim at showing the capabilities of the physics engine, we thought they could bring a better understanding of the influence of these parameters. Therefore, we limit the presentation to general results since a more in-depth analysis could be a thesis subject.

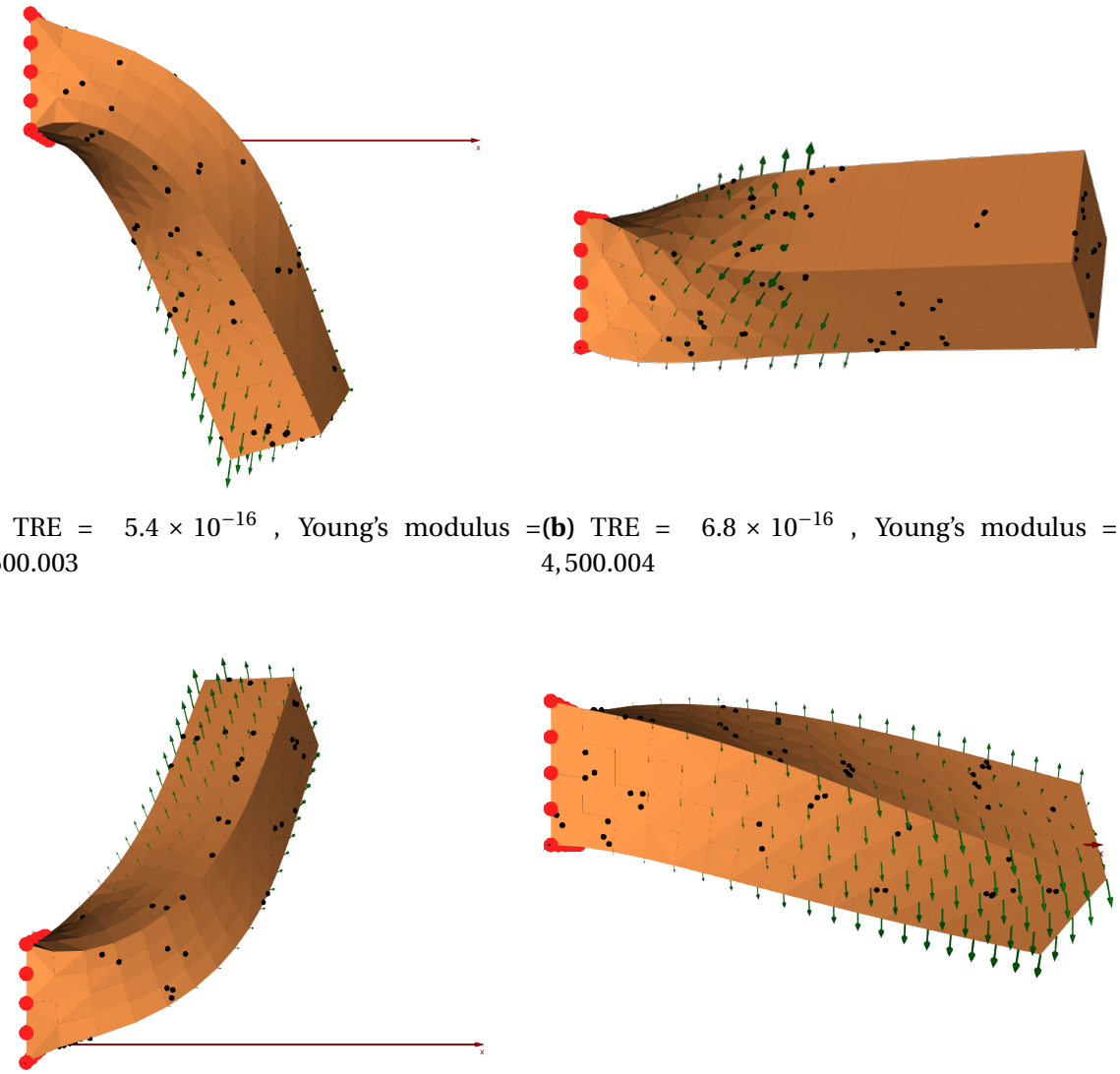
Young's modulus optimization Young's modulus is a positive real value that represents the stiffness of a material. It is measured in Pascal and usually in Giga Pascal. The bigger the value, the stiffer the object. For example, Young's modulus of most metals ranges from ≈100 GPa (gold is 72 GPa) to 700 GPa (Tungsten carbide), wood is approximately 10 GPa, and caoutchouc is around 0.1 GPa.

In this study, Young's modulus is a constant defining homogenous material. Representing more complex materials with non-homogenous stiffness can be achieved by defining element-wise values. This solution is not yet implemented in our solver. Thus, we cannot perform the appropriate tests.

Our test consists in setting a random value between 10^6 and 10^{10} and seeing if the solver is capable of optimizing the value to reach the target point cloud. The real value for the object is 4,500 Pa, which is multiple orders of magnitude smaller than the initial value.

While we always set an initial value bigger than the ground truth, the solver works just as well with an initial value smaller than the ground truth. We estimate that the distance between the initial and ground truth is insufficient if we pick a value between 0 and 4,500 which do not put the solver to the test. Yet, we allow Young's modulus to reach any real positive value during the optimization process. The space of solutions has been restricted by using a constrained optimization solver with a lower boundary set to 10 Pa.

The test has been done on 100 samples; Figure 7.25 presents some results. The presented samples will be the same as the one used in the force estimation study to easily compare the TRE.



(a) TRE = 5.4×10^{-16}, Young's modulus = 4,500.003

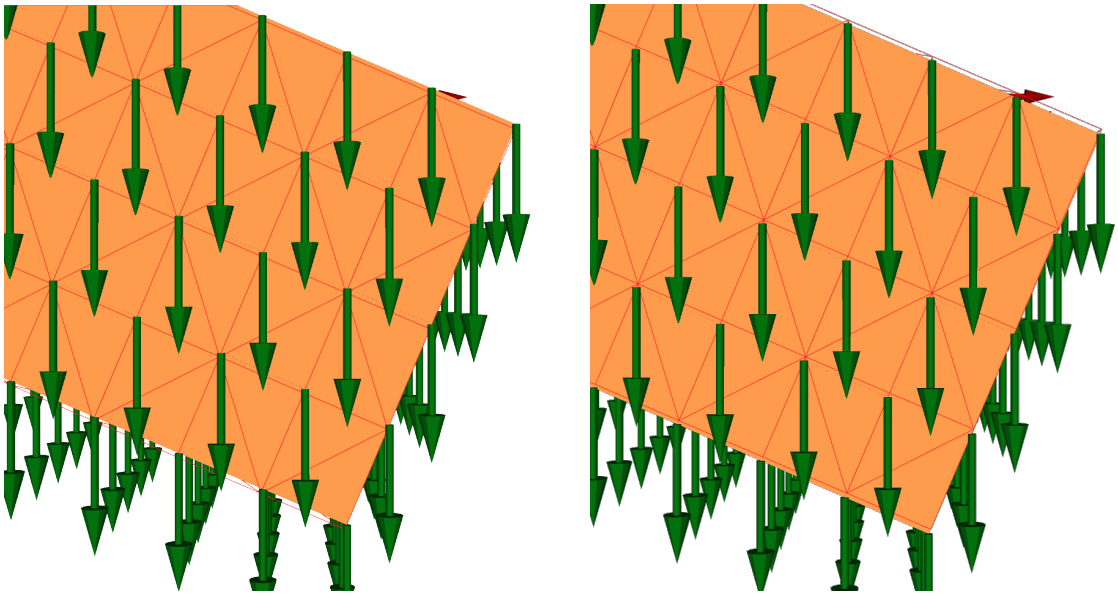
(b) TRE = 6.8×10^{-16}, Young's modulus = 4,500.004

(c) TRE = 6.4×10^{-16}, Young's modulus = 4,500.003

(d) TRE = 2.0×10^{-16}, Young's modulus = 4,500.002

Figure 7.25: Example of four Young's modulus optimization. The TRE is minimal, which represents an excellent fit. Young's modulus is also extremely close to the actual solution.

On average, we obtain a TRE of 4.6×10^{-16}, which can be considered exact up to machine precision. In terms of Young's modulus, we have an average error of 3.1×10^{-3} Pa, which represents an error of 6.9×10^{-5}% and can be considered exact too. As Figure 7.26 displays, with a variation of 1%, the TRE is less than the solver threshold, meaning that the impact on the solution is minimal.



(a) TRE = 7.5×10^{-7}, Young's modulus = 4,500 + 1%

(b) TRE = 7.75×10^{-7}, Young's modulus = 4,500 − 1%

Figure 7.26: Close-up on the tip of the beam. The ground truth with Young's modulus of 4,500 appears in red. The deviation is minimal for an error of 1%, showing that the displacement field is relatively similar for small variations of Young's modulus.

Young's modulus is a coefficient with a range so important that variations of one to two percent can be considered negligible in many application cases. This observation is the total opposite of Poisson's ratio. In the following paragraph, we present our result on optimizing this parameter.

Poisson's ratio optimization The Poisson's ratio is a coefficient that is set between 0 and 0.5 excluded. It represents how much an object is compressible where 0 means that the object has no volume restriction and $0.5 - \epsilon$ the object is totally incompressible. We will see that the coefficient greatly impacts the final solution, even for small variations. Our test consists of setting a random value between 0.2 and 0.499 and seeing if the solver can optimize the value to reach the target point cloud. Optimizing the Poisson's ratio is done using an LBFGS. While the LBFGS is not constrained by default, we optimize the value of a parameter that is projected in the space $]0.2, 0.5[$ using a parametrized logistic function as shown in Figure 7.27

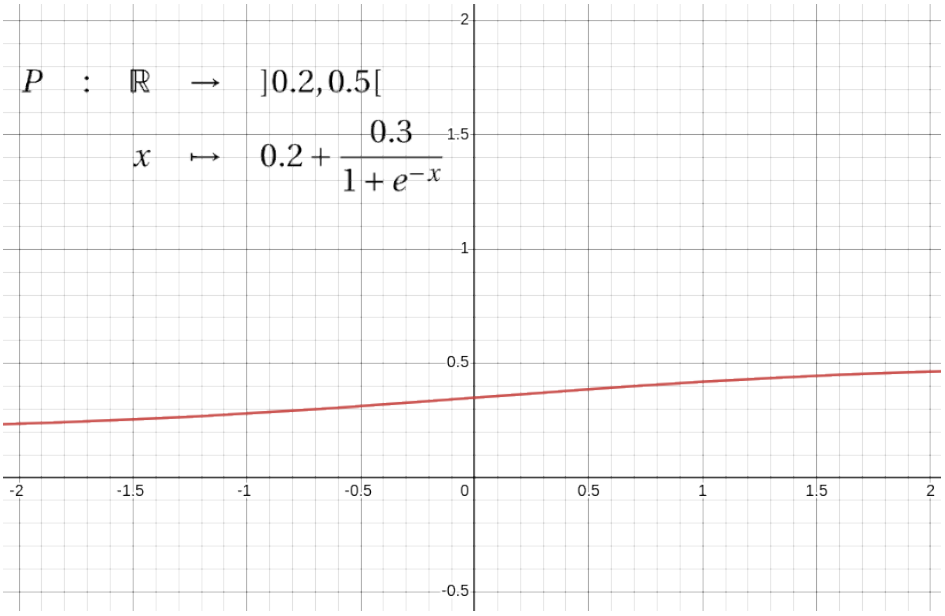
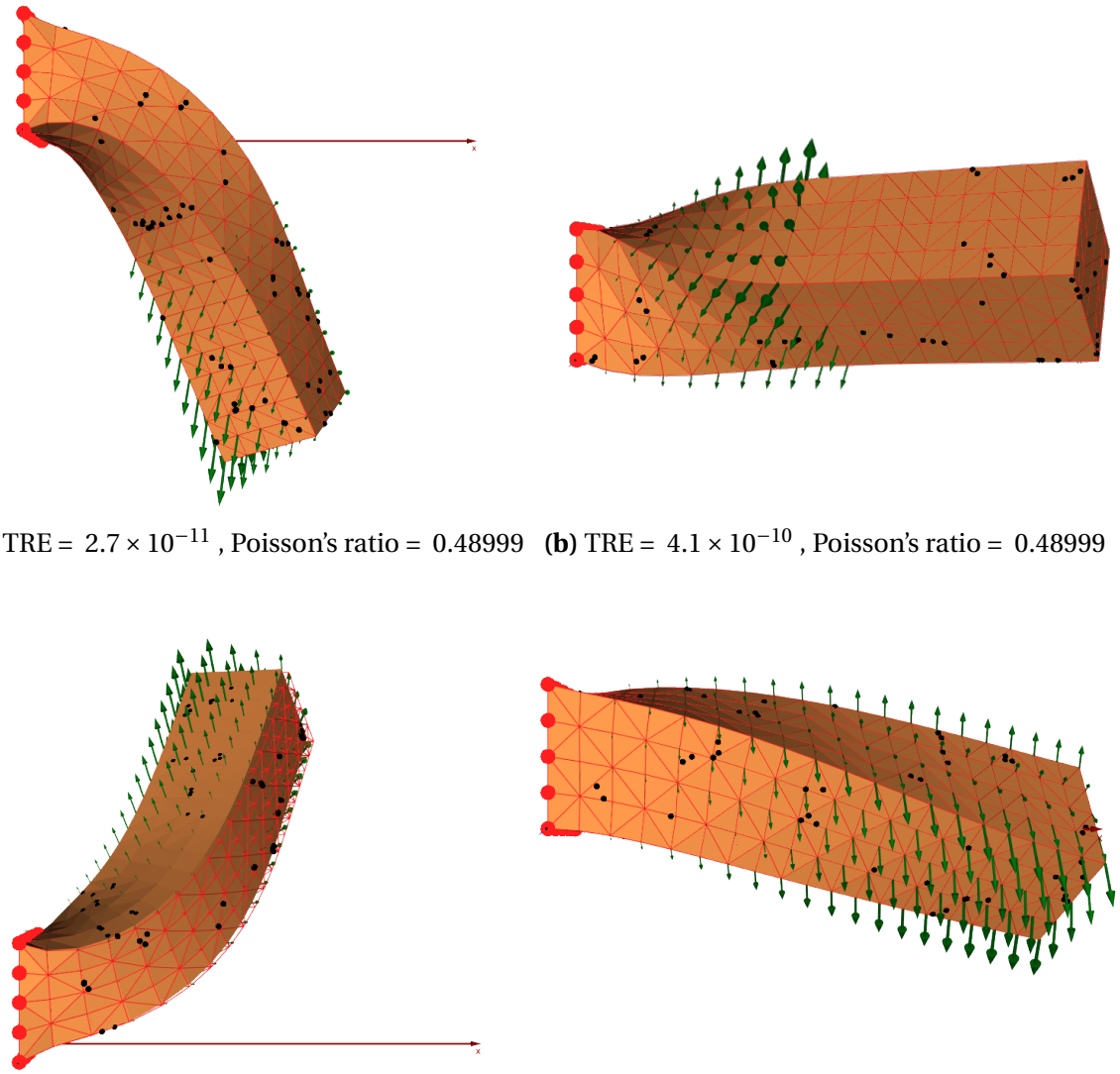


Figure 7.27: The projection P is used to keep the Poisson's ratio in a reasonable value considering our problems. This function is a parametrization of the logistic function. Therefore, it can be easily modified to project in any desired space. The additive constant value defines the minimum value of the space, while the numerator is computed by subtracting the extremes of the space.

This allows us to have more flexibility and optimize any floating point number while still being able to compute deformations. We tried multiple times to use constrained solvers, but the optimization parameter is so sensitive to perturbations that when it did converge, it often took multiple minutes to do so.

This study is done on 100 samples; Figure 7.28 presents some results. Over the 100 tested samples, we obtain a relative error of $0.1\% \pm 1\%$. One observation of the result is that the more complex the deformation, the closer to ground truth the initial value of the Poisson's ratio has to be. For extremely complex deformation such as Figure 7.28a, the parameter's initial value has to be set around $\pm 10\%$ of the original value, or else the FEM solver will not converge. This lack of convergence leads to a critically failing gradient with values reaching 10^{40} ; thus, the whole algorithm fails. For simpler deformation, the algorithm manages to converge with initial values up to 10 times smaller than the ground truth.



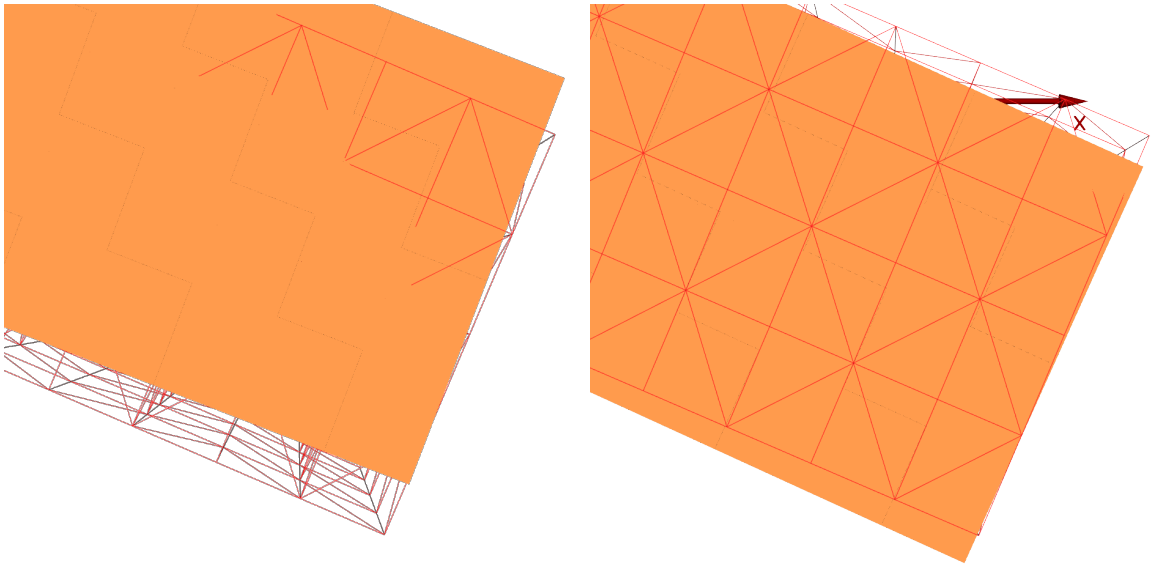
(a) TRE = 2.7×10^{-11} , Poisson's ratio = 0.48999 **(b)** TRE = 4.1×10^{-10} , Poisson's ratio = 0.48999

(c) TRE = 6.3×10^{-5} , Poisson's ratio = 0.48461 **(d)** TRE = 7.1×10^{-11} , Poisson's ratio = 0.49000

Figure 7.28: Example of four Poisson's ratio optimization. The TRE is minimal, which represents an excellent fit. The Poisson's ratio is also extremely close to the actual solution.

Having a low error on this parameter optimization is crucial since it impacts the final result, as shown in Figure 7.29. Estimating the Poisson's ratio with an error of 1% leads to errors a thousand times bigger than the same error on Young's modulus (Figure 7.26). This parameter sensibility is critical since it also affects the stability of the simulation.

With this, we wrap up the differentiable soft-body physics engine presentation. We will now proceed to the conclusion of this work.



(a) TRE = 1.0×10^{-4} , Poisson's ratio = $0.49 + 1\%$ **(b)** TRE = 3.2×10^{-5} , Poisson's ratio = $0.49 - 1\%$

Figure 7.29: Close-up on the tip of the beam. The ground truth with Young's modulus of 4,500 appears in red. The deviation is important for an error of 1%, showing that small variations of Poisson's ratio highly influence the displacement field.

7.2.4 Conclusion on the engine

We developed a differentiable physics engine using PyTorch. The engine can calculate the mechanics of meshed objects with linear or quadratic elements for common topologies, namely triangles, tetrahedra, and hexahedra. Linear and nonlinear materials such as Neo-hookean and Saint-Venant Kirrchhoff are available in the framework. The engine allows the formulation of forward simulations, where we compute the displacement field, and inverse simulations, where we optimize simulation parameters to fit observational data. This is achieved using a differentiable formulation of the solver, allowing us to optimize parameters using the backpropagation algorithm. The mechanic is formulated using an energy minimization problem allowing fast computation of the quantities since the computation of the derivatives is usually computationally demanding. Constraints are softly enforced for the moment, but we plan to change this behavior by manipulating the value of the tangent stiffness matrix. During the development, we emphasized code vectorization and optimization to reduce the computation time of the system state and its gradient.

Although the engine is stable and fulfills all our expectations there is still room for improvement. The calculation time for the tangent stiffness matrix is currently too long for real-time use (the forward simulation of the beam runs at around two frames per second). It is not yet clear whether this is due to the short development time or a limitation of PyTorch.

We have presented the possibilities of the engine using the example of a beam, although it is not limited to a single type of mesh. These examples include the optimization of external forces applied to an object, the optimization of the object's stiffness, and its Poisson's ratio. We briefly discussed the impact of initialization and force support on the quality of the non-rigid registration. Finally, we discussed the sensitivity of forward and inverse simulations to optimize Young's modulus and Poisson's ratio.

We have demonstrated that our engine can calculate forward and inverse simulations by optimizing different simulation parameters.

The next chapter concludes this thesis by recapitulating our achievements and pointing out our contributions' shortcomings. We end this conclusion by opening up on future works that could help resolve part of the shortcomings.

CHAPTER 8

CONCLUSION

8.1 Summary and achievements

This thesis produced contributions in multiple deep learning subdomains at the interface of multiple fields, such as data-driven computational biomechanics, deep learning, and augmented surgery.

Our first contribution [84] is about data generation (Dataset generation). Creating a well-designed dataset is essential to train robust artificial neural networks. We use the tangent stiffness matrix's eigenvalue analysis to study the object's deformation modes. Obtaining the most common deformation of the object is achieved by sampling the first few modes between -1 and 1 (i.e., create a vector where the first few coefficients are between -1 and 1). Once we obtain the deformation of interest, we can compute the corresponding force by a simple matrix-vector product. Therefore, we can produce a force responsible for the linear approximation of the actual deformation using a handful of coefficients. We can now use this force in a simulation to obtain the associated nonlinear deformation.

Although this is not a scientific contribution, our next section focuses on a comparative study between the work of Mendizabal et al. [75] and a much simpler MLP architecture (Toward faster simulations using artificial neural networks). This study compares the result on a cantilever beam and liver mesh. The results indicate that the MLP outperforms U-Mesh for small-resolution meshes regarding prediction accuracy and training speed. Secondly, the U-Mesh uses a reduced latent space to represent the problem, which allows

for better generalization but results in slightly lower accuracy for patient-specific scenarios. However, the accuracy can be increased by increasing the size of the reduced latent space, although this will increase training time and require a trade-off between accuracy and efficiency. Regarding speed, the FEM simulation takes around 1,500 ms to compute a solution. The U-Mesh reduces this time by 500 and the MLP by 5,000. Furthermore, on similar problems, the MLP has proven to be much faster to train than the U-Mesh. Overall, both architectures perform similarly. The U-Mesh is more resilient to outliers, but the MLP performs better on average and is faster to train and predict. Therefore, we decided to continue this thesis with the MLP.

Our second contribution [84] is about the reliability of the ANN prediction (Hybrid solver). As we have seen before, the chosen network performs well on average but is susceptible to outliers. During surgery, we cannot allow the predictions to behave poorly. Therefore, we must create a way to correct them on the fly. In our contribution, we propose initializing Newton's method with a prediction of a purposely trained ANN. This benefits both the network prediction and the Newton-Raphson algorithm. Using this algorithm, we are sure that the neural network's prediction is acceptable, and we put ourselves in the quadratic convergence phase as soon as possible. This way, we maximize quality and speed at the cost of training an ANN. The study shows that the proposed algorithm converges more often and faster than the classic method while keeping the ordinary convergence properties of the Newton-Raphson algorithm. With this, we have a first answer on how to quickly predict nonlinear deformations of a soft body and how to verify and correct these predictions.

Our last contribution [85] uses our previous work to compute the non-rigid registration of an object from surface data (Optimal control for augmented surgery). In particular, we present an application for laparoscopic surgery. This contribution follows the work of Mestdagh et al. [78]. Their work uses optimal control and an adjoint method to retrieve the forces that deform a liver composed of nonlinear material to fit a partial surface point cloud. We identified two main bottlenecks the forward problem and the adjoint problem. We propose to replace the forward problem with an equivalent neural network that takes input forces and outputs a displacement field. Since we now have easy access to automatic differentiation and backpropagation, we propose to replace the adjoint problem with the backpropagation algorithm. This way, we can differentiate easily with respect to the control (force). The study shows that we have similar results over the five practical scenarios. The registration error is acceptable and preserves the physical properties of the registered mesh. Considering the mesh's complexity and the material's nonlinearity, computing a single iteration of the algorithm using a classical solver takes multiple seconds, leading to an average of 14 minutes per frame. Our proposed algorithm uses a neural net-

work to improve the computation speed of both the hyper-elastic and adjoint problems. The hyper-elastic problem takes around 4 to 5 milliseconds to compute while the adjoint problem replaced by the backpropagation algorithm takes around 11 ms . This leads to a significant improvement in convergence speed, where we reduce the computation time by a factor of 6,000 on average. We also show that the method is resilient to the range of noise expected from the hypothesis, making it suitable for this application.

In summary, we have worked throughout the whole deep-learning pipeline to extend the possibilities of deep learning applied to soft-body physics. We used our contributions to improve the non-rigid registration of an object from surface data and showed that it performs well on synthetic laparoscopic data.

8.2 Outlook and futur work

Although considerable advances have been made toward efficient computations of soft body deformations for liver surgery, shortcomings remain. These shortcomings can be classified into generalization and performance improvements of the learning.

The generalization part intervenes when we consider our method as a product. During the time frame between the data acquisition (scanner / MRI) and the surgery, we have around 24 to 48 hours to generate the dataset and train the network, which might not be sufficient. At the moment, our ANN is specific to each mesh. Therefore, we need to create a new dataset and retrain from scratch, which is time and resources demanding. One idea would be to use transfer learning [114] to specialize a generalist pre-trained ANN on each patient. The transfer learning technic proposes to train a network to perform decently well on many tasks. This produces a set of *good weights* that can be easily specialized in one of the previous tasks. The authors claim that the pre-trained network converges faster and better than if we started with random weights. In our case, this would consist in training a network to perform decently well on multiple livers and then retraining it with patient-specific data. Therefore, it would converge faster on fewer data.

For our application, transfer learning raises an architectural challenge. The mesh of the simulated object defines MLP architectures and cannot be modified after the training. A single mesh cannot accurately represent a patient's specific data due to patient-wise variations in Dirichlet boundary conditions (arteries, ligaments). Therefore, we cannot pre-train a single MLP for everyone. However, all hopes are not lost since a neural network architecture called graph neural network has been specially designed to deal with variable graph-like data structures. GNN for soft body physics is a hot topic as multiple publications have recently appeared [40, 69, 104]. Today, GNN has not achieved satisfac-

tory precision for our applications, but this approach is relatively new and already produces promising results [90].

The performance improvements mainly refer to the computation speed of the gradient and hessian, which raise concerns in the penultimate and last chapters. These two items are computed using the automatic differentiation and backpropagation algorithm, which becomes the main limiting factor of the methods. As mentioned in the conclusion of the last chapter, these shortcomings are likely due to short development time or problematic limitations from PyTorch. Using other frameworks such as JAX [8] we have seen a significant reduction in computation time of the gradient and hessian of the system. These promising results require more work in the future since we only implemented the said computation and not the entire differentiable soft-body physics engine. Implementing the remaining parts, such as differentiable Newton-Raphson, could be extremely challenging.

Generalization and performance improvements of the proposed methods promise exciting results from the research and development aspect. The continuation of this thesis would most likely be centered on GNN and efficient differentiable solver. One could enable extreme versatility of the methods via a small sample of patient-specific data, while the second will produce fast and differentiable simulations to train said ANN.

Furthermore, the differentiable solver could be used to optimize more abstract simulation parameters such as the mesh geometry/topology [37] or material of the patient organs [57] to reduce the number of integration points and thus, speeding up the computations.

CHAPTER 9

RÉSUMÉ EN FRANÇAIS

Selon l'Organisation Mondiale de la Santé (OMS), près de vingt millions de nouveaux cancers ont été diagnostiqués en 2020. Cette maladie peut toucher n'importe quelle partie du corps, mais les principales parties affectées sont le sein, les poumons et le côlon. Néanmoins, tous les cancers ne sont pas égaux vis à vis de la réponse au traitement et le taux de mortalité. Les cancers du foie sont particulièrement meurtriers, puisqu'ils représentent environ un diagnostic sur vingt, ainsi qu'un sur douze induit par le cancer. Ces pathologies sont généralement traitées par résection du foie, soit par chirurgie abdominale ouverte, soit par chirurgie laparoscopique.

Grâce aux progrès des techniques chirurgicales, des procédures plus complexes telles que les résections hépatiques majeures ou les résections du foie d'un donneur vivant sont désormais réalisables par une approche laparoscopique ou robotique. En chirurgie laparoscopique, l'ensemble de l'intervention est réalisé par de petites incisions semblable à des trous de serrure, tandis que le chirurgien regarde les images transmises par une caméra sur un écran.

Pendant la manipulation, les chirurgiens doivent mémoriser et projeter des données spécifiques au patient, telles que l'emplacement et la taille des tumeurs, les vaisseaux sanguins et les artères dans un environnement en mouvement constant en raison de la respiration et de la circulation sanguine. En plus de ce suivi qui demande déjà beaucoup de concentration, ils doivent réaliser des actes chirurgicaux précis et complexes tout en commandant l'ensemble de la salle d'opération.

Afin de faciliter la tâche et réduire la charge mentale du chirurgien il serait possible

d'améliorer sa perception visuelle du patient en utilisant la réalité augmentée (RA). Pour ce faire, il est possible combiner un flux vidéo en direct provenant du laparoscope (une petite caméra insérée dans le corps du patient par une petite incision) avec des images et des données virtuelles superposées. Il peut s'agir de modèles 3D de l'anatomie du patient, d'instructions chirurgicales ou de données en temps réel telles comme les signes vitaux.

L'un des principaux avantages de la RA en chirurgie laparoscopique est de permettre au chirurgien de voir à l'intérieur du corps du patient de manière plus intuitive et naturelle, ce qui peut contribuer à améliorer l'exactitude et la précision de l'intervention. Les images 3D et la superposition virtuelle peuvent également contribuer à améliorer la visualisation de l'anatomie complexe et des structures internes, qui peuvent être difficiles à voir avec les techniques laparoscopiques traditionnelles.

Lors de la superposition de modèles 3D de l'anatomie du patient pendant l'opération, il faut tenir compte des déformations induites par le chirurgien et les reproduire sur les modèles 3D. La complexité réside dans la détection des déformations et la préservation des propriétés internes des organes. Une telle technologie nécessiterait la capacité de calculer des déformations complexes des organes en temps réel à partir d'observations partielles de la surface.

Les calculs de déformations complexes peuvent être effectués à l'aide de la méthode aux éléments finis (MEF) et d'un modèle d'organe en 3D. La méthode des éléments finis est une technique numérique puissante largement utilisée pour résoudre des problèmes physiques dans divers domaines, tels que la dynamique des fluides, l'électromagnétisme ou encore la dynamique des corps mous. L'objet simulé est représenté par de nombreux éléments interconnectés, appelés éléments finis, définis comme un ensemble de propriétés et de quantités physiques. Les éléments finis peuvent gérer diverses conditions aux limites et travailler avec des systèmes linéaires et non linéaires. Dans notre contexte, la simulation serait utilisée pour calculer la déformation du foie, qui est constitué de tissus mous, principalement composés d'eau. Cette composition génère une réponse non linéaire à la charge appliquée, créant ainsi un système non linéaire à résoudre.

En raison de sa formulation mathématique, la méthode des éléments finis peut être très gourmande en ressources informatiques, nécessitant une puissance de calcul importante pour produire une solution. Cela peut constituer un défi pour les problèmes finement maillé ou les simulations nécessitant des performances en temps réel comme les nôtres.

L'apprentissage profond est un domaine qui a été principalement conçu pour fournir des solutions rapides à des problèmes complexes. Au cours des dix dernières années, le développement de l'apprentissage profond a permis de s'attaquer à des problèmes complexes à un rythme jamais atteint auparavant. L'idée principale derrière l'utilisation de

l'apprentissage profond dans la simulation physique est d'utiliser des réseaux de neuronnes artificiel (RNA) pour apprendre les lois physiques sous-jacentes d'un système à partir de données, puis d'utiliser ces connaissances pour faire des prédictions sur le comportement du système dans différentes conditions.

L'apprentissage profond peut être utilisé pour modéliser des systèmes physiques complexes, tels que la dynamique des fluides, les matériaux granulaires ou encore la dynamique des corps mous, qui peuvent être difficiles à simuler à l'aide des méthodes traditionnelles basées sur la physique. Pour ce faire, on entraîne des réseaux neuronaux sur de grandes quantités de données générées par des simulations ou des expériences, puis on les utilise pour prédire le comportement du système.

L'un des principaux avantages de l'utilisation de l'apprentissage profond dans la simulation physique est sa capacité à atténuer certaines limites de la méthode des éléments finis. Il peut apprendre à partir de données réelles, ce qui rend la simulation plus précise et plus réaliste. Cela est fait en modifiant les paramètres de simulation tels que les propriétés des éléments ou les conditions aux limites.

Il est important de noter que l'apprentissage profond dans la simulation physique est un nouveau domaine de recherche. De nombreux défis sont à relever, tels que la nécessité de disposer de grandes quantités de données de haute qualité, l'interprétabilité et la généralisation des modèles. En outre, il est essentiel de s'assurer que les prédictions faites par le RNA sont physiquement significatives et cohérentes avec les lois de la physique. Malgré ces difficultés, la combinaison de l'apprentissage profond et de la simulation physique s'est révélée très prometteuse pour créer des simulations plus réalistes et plus précises de divers systèmes physiques. Elle pourrait révolutionner la façon dont nous simulons et comprenons les phénomènes physiques complexes et devrait avoir un large éventail d'applications dans des domaines tels que l'ingénierie, la science et l'infographie. Toutefois, des recherches supplémentaires sont nécessaires afin de comprendre pleinement ses capacités et ses limites et pour relever les défis susmentionnés.

Grâce à la MEF et à l'intelligence artificielle (IA), nous pouvons calculer les déformations complexes des tissus mous en temps réel. Cependant, un aspect essentiel du processus reste la manière dont nous traitons les données d'observation pour les adapter à notre modèle de corps mou.

Pendant la laparoscopie, les données d'observation sont acquises à l'aide du laparoscope. En utilisant des techniques existantes [97], nous pouvons convertir les observations en surfaces partielles de nuages de points 3D. Cependant, le calcul en temps réel de la déformation qui correspond au nuage de points 3D soulève de nombreux défis.

Le premier est le réalisme. Des informations utiles sont fournies lorsque le déplacement calculé est aussi proche que possible de la réalité, ce qui nécessite de nombreux

calculs.

Le deuxième défi est celui du temps réel. L'interactivité entre la chirurgie et la simulation nécessite que la simulation fonctionne de manière fluide à une fréquence d'images décente (≈60 images par seconde). Cette contrainte s'oppose à la première, exigeant le compromis habituel vitesse/précision.

Le troisième défi est qu'un nuage de points d'une partie de la surface est insuffisant pour identifier une solution unique. Par conséquent, d'autres informations, telles que des hypothèses physiques, doivent être ajoutées avant de considérer la précision du recalage. Le modèle et les méthodes choisis doivent résulter d'un compromis entre l'acceptabilité physique de la solution et l'efficacité requise par l'exécution en temps réel.

Récemment, Mestdagh et al. [78] ont proposé une méthode formulant le problème d'optimisation dans le cadre générique du contrôle optimal. Ce cadre générique rend accessible l'incorporation de données pré- ou intra-opératoires supplémentaires dans le calcul. Leurs résultats sont présentés sur un modèle avec des propriétés d'élasticité non linéaires, constituant une avancée par rapport aux méthodes similaires. De plus, ils l'implémentent en utilisant la méthode adjointe, ce qui pourrait permettre de remplacer le solveur traditionnel par un réseau de neurones.

Cette thèse intitulée *Data-driven computational biomechanics using Deep Neural Networks – Application to augmented surgery* donne une première solution à ce problème de chirurgie augmentée pour la résection du foie.

Finite element method

La bonne compréhension de ce travail nécessite une connaissance avancée de la méthode des éléments finis. Nous proposons un premier chapitre sur ce thème afin que cette thèse se suffise à elle-même. La plupart des problèmes définis par une équation aux dérivées partielles n'ont pas de solutions analytiques et nécessitent donc une approximation. Cette approximation est réalisée à l'aide de la méthodes aux éléments finis et un maillage d'éléments finis. Ces éléments peuvent être de plusieurs types : surface (triangle, quadrilatère), coque ou volume (tétraèdre, hexaèdre). Ces éléments sont utilisés pour calculer des quantités mécaniques locales, qui sont ensuite interpolées dans le but de calculer la quantité globale correspondante. Le degré de l'interpolation est défini par celui des éléments (linéaire ou quadratique).La différence entre les formes linéaires et quadratiques n'apparaît que dans le nombre de nœuds d'interpolation. Il existe donc des triangles quadratiques, ainsi que des triangles linéaires.

Cette interpolation est réalisée à l'aide de la fonction de forme, qui constitue la deux-

ième section de ce chapitre. Les fonctions de forme représentent les poids par élément associés à chaque nœud et permettent de calculer l'évolution des quantités mécaniques dans le domaine d'intégration en interpolant les quantités nodales. Nous présentons un cas particulier de fonction de forme appelé fonction de forme isoparamétrique. Le terme isoparamétrique est dérivé de l'utilisation des mêmes fonctions de forme (ou fonctions d'interpolation) pour définir la forme géométrique de l'élément que celles utilisées pour définir les déplacements à l'intérieur de l'élément. En outre, lorsqu'on traite des phénomènes physiques, on souhaite généralement calculer les variations des propriétés physiques. Pour ce faire, on calcule les gradients et les jacobiens. Ici, la quantité d'intérêt est le gradient du déplacement. Le gradient de déplacement en tout point de l'élément peut être approximé en utilisant le déplacement des nœuds et le gradient de leurs fonctions de forme par rapport aux points matériels. La définition de l'élément isoparamétrique étant générique, il n'est pas nécessaire de spécifier comment calculer le jacobien de chaque type d'élément, ce qui facilite grandement le développement d'un moteur de simulation.

Dès lors, nous pouvons interpoler n'importe quelle quantité sur le maillage et calculer la dérivée de cette quantité. Ceci est particulièrement utile puisque la MEF est basée sur le calcul de la dérivée de quantités telles que le champ de déplacement et l'énergie potentielle élastique. Le gradient du champ de déplacement est principalement utilisé pour calculer une quantité essentielle appelée tenseur de déformation. Le tenseur de déformation est à la base de tout calcul dans le cadre de la MEF. Par exemple, il nous permet de calculer le tenseur de déformation de Cauchy-Green, qui nous donne le carré de la variation locale de la distance due à la déformation. Il constitue un invariant intéressant pour calculer les propriétés physiques, comme nous le verrons par la suite. Le tenseur de déformation est également utilisé pour calculer le tenseur de déformation de Green-Lagrange, qui mesure l'écart entre le tenseur de Cauchy-Green et l'identité. Sa formulation varie selon que l'on considère l'élasticité linéaire ou non linéaire. Ce tenseur est particulièrement explicite puisque ses termes diagonaux (respectivement non diagonaux) sont liés à la déformation normale (respectivement de cisaillement). Tous ces tenseurs représentent des quantités géométriques où seul le déplacement est considéré. En utilisant ces tenseurs et les propriétés mécaniques de l'objet, nous pouvons calculer les premier et deuxième tenseurs de contrainte de Piola-Kirchhoff. Ces tenseurs représentent la réponse à la contrainte induite.

Cette réponse est donnée par le matériau associé à l'objet. Le second tenseur des contraintes de Piola-Kirchhoff est la dérivée de la fonction d'énergie par rapport au tenseur de déformation de Green-Lagrange. Chaque matériau est défini par sa fonction d'énergie, qui définit sa réponse à une contrainte donnée. En particulier, la loi des matériaux de Saint-Venant-Kirchhoff, qui étend les matériaux linéaires en utilisant la formulation non

linéaire du tenseur de déformation de Green-Lagrange. Nous souhaitons mentionner la loi des matériaux de Néo-Hookéen, qui nous aidera à modéliser le comportement du foie dans cette thèse. Cette loi est un cas particulier du matériau incompressible de Mooney-Rivlin.

Tous ces tenseurs sont assemblés dans un système représentant un ensemble d'équations d'équilibre. Nous trouvons l'équilibre de la masse, du moment linéaire, du moment angulaire et de l'énergie formulé à l'aide des tenseurs mentionnés précédemment. Ce système est appelé forme forte, où les égalités doivent être satisfaites en chaque point du domaine. Afin de résoudre le système, nous nous appuyons sur la recherche d'une solution continue par morceaux pour le champ de déplacement. C'est pourquoi, nous introduisons une fonction de test qui nous conduit à la forme faible du système, dans laquelle les conditions ne doivent être valables qu'en moyenne. Enfin, nous discrétisons ces quantités et résolvons le système à l'aide des méthodes appropriées, telles que l'algorithme de Newton-Raphson ou un simple optimiseur de gradient conjugué pour les systèmes linéaires.

Après cette brève présentation de la MEF, nous pouvons maintenant passer au deuxième aspect important de cette thèse: l'apprentissage profond. C'est ce sujet que nous abordons dans notre deuxième chapitre (chapitre 3).

Deep learning

L'intelligence artificielle (IA) est actuellement l'un des mots les plus à la mode dans l'industrie technologique, et ce pour une bonne raison. Ces dernières années, plusieurs innovations et avancées qui relevaient auparavant du domaine de la science-fiction se sont lentement transformées en réalité. L'un des principaux acteurs de ces avancés est l'apprentissage automatique qui est une sous-partie de l'intelligence artificielle. Il regroupe des méthodes qui exploitent les données pour améliorer la performance des tâches sur des concepts abstraits. Il construit un modèle ou une fonction sur la base d'un échantillon de données, appelé tout au long de ce manuscrit "données d'apprentissage" ou "échantillon", pour prédire ou décider sans être explicitement programmé. Les méthodes d'apprentissage automatique sont généralement divisées en apprentissage supervisé, non supervisé et par renforcement. Bien qu'initialement elles aient été parfaitement distinctes, la frontière entre elles s'est estompée lorsque les scientifiques ont commencé à former leurs modèles à l'aide de plusieurs paradigmes d'apprentissage simultanément.

L'apprentissage supervisé fait référence au type d'apprentissage dans lequel le modèle voit des paires explicites d'entrées et de sorties. Son travail consiste alors à être capable

de reproduire l'ensemble de données en espérant qu'il interpolera correctement entre les points de données.

L'apprentissage non supervisé est tout l'opposé : le réseau reçoit une entrée et une politique à suivre. Pour une entrée donnée, tel qu'une force, le modèle doit produire un déplacement tel que l'équilibre de la quantité de mouvement linéaire soit respecté.

Enfin, l'apprentissage par renforcement fait référence à l'apprentissage dans lequel le modèle agit sur un environnement donné. Son retour d'information correspond à une mesure de la qualité de son impact sur l'environnement.

Les modèles utilisés par les différents paradigmes d'apprentissage fonctionnent de manière similaire bien que leur architecture soit souvent très différente. Ils sont tous composés de neurones artificiels assemblés en couches. Leur assemblement et les opérations associées définissent le type de couche. La sortie de la couche est ensuite transmise à une fonction d'activation qui introduit une non-linéarité dans la réponse du réseau de neuronnes artificiel. Les fonctions d'activation peuvent par exemple être $tanh, ReLU$ ou encore $sigmoïde$. Une architecture est définit par l'empilement de plusieurs couches. Il n'y a aucune restriction à la définition d'une architecture tant que la sortie de la couche précédente correspond à l'entrée de la couche suivante.

Historiquement parlant, si l'on exclut le traitement du langage naturel qui utilise des mécanismes précis, il existe trois paradigmes d'architectures : le perceptron multicouche (MLP), le réseau neuronal convolutif (CNN) et le réseau de neuronnes en graphe (GNN).

Un perceptron multicouche est un empilement de couches denses ou entièrement connectées où chaque neurone des couches agit comme un perceptron, d'où son nom. Ce type d'architecture est rapide et mathématiquement capable d'approximer n'importe quelle fonction [44]. Ses principaux inconvénients sont la croissance quadratique du nombre de paramètres et la tendance à suradapter les données d'apprentissage. Les perceptrons multicouches ne peuvent accepter que des vecteurs comme entrées. Lorsqu'il s'agit d'images, nous devons aplatir les tableaux de pixels en 2D pour les introduire dans le modèle. Prenons l'exemple d'une petite image en niveaux de gris de taille 64 x 64, aplatie en un vecteur de longueur 4 096. La connexion de deux couches denses composées de 4 096 entrées nécessite la création de 16 777 216 neurones. Cette valeur doit être multipliée par le nombre de couches, ce qui nécessite rapidement un matériel spécialement conçu, doté d'une bande passante et d'une mémoire importantes.

Les réseaux neuronaux convolutifs ont été introduits pour résoudre des tâches de reconnaissance d'images [31] et ont depuis été appliqués à presque toutes les classes de problèmes. Comme son nom l'indique, une convolution se produit. Cette opération remplace l'accumulation globale du MLP par une opération plus locale. Les poids sont disposés dans un masque ou noyau apprenable de taille fixe (généralement relativement

petite, de trois à onze de large) et sont convolués avec la grille d'images/de données. Une seule couche de convolution peut comporter plusieurs noyaux traitant des différents canaux de la grille. Cette agglomération locale réduit le nombre de paramètres puisque les mêmes poids sont utilisés pour l'ensemble de la grille. Les couches de convolution sont considérées comme précieuses pour résumer la présence de caractéristiques dans les données. Le principal inconvénient de cette architecture est qu'elle nécessite des données structurées en grille en raison de la convolution. Cette contrainte correspond parfaitement à son objectif initial, qui est de travailler sur des images, mais lorsqu'il s'agit de données non structurées, nous devons d'abord projeter les propriétés sur une grille. Cette projection introduit beaucoup de bruit dans les données car elle modifie les entrées réelles. De plus l'interpolation n'a parfois aucun sens car le lien entre les points de données peut représenter des concepts abstraits comme l'affinité, qui ne peuvent pas être interpolés de manière adéquate.

Enfin, l'architecture la plus récente que nous souhaitons aborder est le réseau de neuronnes en graphe. Cette technique relativement nouvelle a été introduite dans "A new model for learning in graph domains" [34] mais a été principalement développée au cours des cinq dernières années. L'architecture proposée, désormais appelée réseau de neuronnes en graphe, utilise la topologie ou le lien sous-jacent entre les points de données. Cette formulation explicite supprime la nécessité de la grille et de l'opération de convolution tout en conservant le flux d'informations local puisqu'un nœud de graphe ne communique qu'avec ses voisins directs. C'est intéressant à bien des égards, mais jusqu'à présent, nous n'avons pas été en mesure de produire des résultats ne serait-ce qu'un peu similaires à ceux obtenus à l'aide de CNN ou MLP.

En général, les modèles ont tendance à combiner différents paradigmes pour accomplir leur tâche. On peut considérer l'architecture comme une image complète du flux de données et des calculs qui s'ajoutent au cours d'une prédiction. Ce flux de données représente la forme générale de la fonction que nous essayons de modéliser, tandis que le model apprend la valeur des paramètres dans ces calculs.

L'apprentissage de ces paramètres s'effectue à l'aide d'une politique d'apprentissage ou d'une fonction de coût qui calcule, pour des données d'entrée données, la distance entre la prédiction du réseau et la réalité. Les fonctions de perte mesurent la performance du RNA pour une entrée donnée. Dans sa version la plus simple, la fonction comporte deux variables, la sortie du réseau et la vérité de base ou sortie souhaitée. La distance entre les points de données est mesurée et est ensuite traitée pour mettre à jour les poids du réseau neuronal. Le plus souvent, les fonctions de perte sont quadratiques. Il s'agit d'une propriété importante de la fonction de coût. Au cours de la formation, le gradient de l'erreur est utilisé pour mettre à jour les poids. Les fonctions linéaires ont des gradi-

ents constants qui ne tiennent pas compte de l'erreur. Lorsqu'elles sont différenciées, les fonctions quadratiques sont proportionnelles à l'erreur, ce qui affecte positivement les variations des poids. La fonction de coût la plus connue pour les tâches de régression est l'erreur quadratique moyenne (MSE). Elle est indépendante de la taille des éléments, représente entièrement les données et est quadratique.

Le résultat de cette évaluation est ensuite utilisé comme point de départ de l'algorithme de rétropropagation. La rétropropagation est utilisée pour calculer la dérivée de la fonction de coût par rapport aux paramètres apprenables du réseau. Cette dérivée est utilisée pour mettre à jour les poids/neurones du réseau, créant ainsi la partie apprentissage de l'apprentissage profond. Cet algorithme s'appuie sur quatre équations présentées dans la section 3.5 pour construire la dérivée de la fonction de coût par rapport à chaque neurone de l'ANN. La présentation de l'algorithme de rétropropagation est faite dans la section correspondante mais, malheureusement, elle est très mathématique et ne peut être résumée de manière appropriée qu'à l'aide d'équations.

Enfin, l'apprentissage profond peut être résumé en trois étapes : la création d'un ensemble de données, l'assemblage d'un modèle et la formation. Un aspect important des réseaux neuronaux à garder à l'esprit lors de la création d'un ensemble de données est qu'ils sont bons pour interpoler les propriétés, mais qu'ils pourraient être meilleurs lorsqu'il s'agit d'extrapolation. Un ensemble de données peut être biaisé ou manquer de représentativité, ce qui entraîne une extrapolation encore plus mauvaise. Nous souhaitons que notre ensemble de données représente notre problème ou la tâche que nous essayons d'accomplir. Dans certains cas, l'objectif pourrait être mieux défini, ce qui augmenterait encore la difficulté de la génération de l'ensemble de données.

L'assemblage d'un modèle est une tache simple. Il suffit d'empiler les couches prédéfinies (entièrement connectées, convolution, pooling max, etc.) dans un graphe de calcul définissant le flux de données à travers le modèle. La difficulté apparaît lors de la phase de conception. La sélection des couches et le graphe de calcul sont définis en même temps, car ils ont un impact direct l'un sur l'autre. L'idée est que le flux de données représente une connaissance spécifique du problème. Un flux de données spécifique peut également être réalisé à l'aide d'une architecture fractionnée ou raccourcie (Figure 3.5 et Figure 3.6 respectivement).

Afin d'entrainer un RNA nous itérons plusieurs fois sur l'ensemble des données. Une itération est appelée époques. L'entrainement est donc composée de plusieurs époques. Chaque époques est divisé en lots de taille arbitraire qui sont ensuite donné un par un au réseau. Ensuite, chaque lot est évalué à l'aide de la fonction de coût. Finalement, l'erreur est rétropropagée et les poids sont mis à jour à l'aide d'un optimiseur linéaire, généralement ADAM [52].

Fast and accurate deformations using deep learning

Après avoir présenté tous les outils dont nous avons besoin et leur fonctionnement, le chapitre suivant (chapitre 4) de cette thèse se concentre sur la déformation rapide et précise à l'aide de l'apprentissage profond. Pour obtenir des déformations précises, il faut d'abord disposer d'un bon ensemble de données. Nous commençons cette partie du résumé en présentant une manière plus intuitive de générer un ensemble de données de paires de forces et de déformations. Lors de la construction d'un ensemble de données, la complexité apparaît lors de la prise en compte la fonctionnalité d'un objet. Il serait intéressant de créer un ensemble de données d'échantillons significatifs représentant l'utilisation moyenne d'objets simulés et leurs déformations les plus courantes.

Les objets manufacturés sont plus faciles à traiter car leur conception est orientée vers un but précis. Par conséquent, nous avons une bonne idée de l'endroit et de la manière d'appliquer la charge. A l'inverse cette absence de conception rend les objets non-manufacturé, tels que les organes, plus difficiles à manipuler. Grâce aux connaissances acquises lors de discussions avec les chirurgiens, nous pouvons estimer la zone d'intérêt et le type de force appliquée aux organes pendant les interventions chirurgicales. Cependant, nous ne disposons pas de données exactes et l'échantillonnage de l'espace des forces reste donc une tache complexe.

Deux difficultés principales se posent lors de la création d'un ensemble de données représentant la déformation d'un objet : le nombre de paramètres et la relation entre la charge et le déplacement. Concernant le nombre de paramètres, dans le contexte de cette thèse, nous avons affaire à des maillages composés de deux à cinq mille nœuds, soit six à quinze mille degrés de liberté. L'application d'une charge externe à un objet nécessite d'affecter des valeurs de force à chaque degré de liberté du maillage. Une solution pour traiter le problème de la dimensionnalité consiste à appliquer des forces qui sont constantes sur l'ensemble de l'objet en ne considérant que la composante X, Y, Z du champ vectoriel, réduisant ainsi le nombre de paramètres à trois. Bien que l'application d'une telle force soit facile, elle représente rarement des scénarios réels ou des scénarios intéressants. Définir la valeur des degrés de liberté sur l'ensemble du maillage est déjà problématique à notre échelle. Il faut définir la valeur de plusieurs milliers de coefficients pour générer un seul déplacement alors que plusieurs milliers de déplacements doivent être calculés pour entraîner un RNA. Par conséquent, il faut s'appuyer sur un échantillonnage aléatoire et une approche naïve pour générer un vecteur de force et espérer qu'il produise des déformations intéressantes, ce qui n'est pas efficace en termes de temps et d'énergie.

La deuxième difficulté apparaît lors de l'utilisation de modèles non-manufacturés qui conduisent à des relations complexes entre la charge et le déplacement. Lorsqu'un objet est soumis à une charge externe, sa réponse est donnée par son matériau et sa géométrie. Alors que la réponse du matériau est simple et définie par l'utilisateur, la réponse de la géométrie peut elle être très complexe. De ce fait, une géométrie d'apparence simple telle qu'une corde peut avoir une réponse difficile à prévoire tel que la formation de plectonèmes lorsqu'une torsion est appliquée.

Le nombre de nœuds, le matériau et la géométrie sont constitutifs d'un objet et ne peuvent donc pas être modifiés sans avoir un impact sur sa physique. Bien que nous ne puissions pas les modifier, ils contiennent beaucoup d'informations sur la façon dont un objet réagit aux contraintes. Nous pouvons extraire des informations en utilisant notre connaissance de la mécanique des corps déformables. En particulier, nous voudrions obtenir deux choses.

La première est une estimation approximative du déplacement généré par la contrainte afin de savoir s'il est nécessaire de calculer la déformation totale.

La seconde est de générer facilement des vecteurs de force intéressants/complexes.

Si l'on considère la première demande, la matrice de rigidité tangente contient toutes les informations nécessaires sur la géométrie, la topologie et le matériau de l'objet pour un déplacement donné. Supposons que nous admettions que le comportement de l'objet soumis à des déformations non destructives ne change pas radicalement. Il est alors possible d'utiliser la matrice de rigidité tangente de la forme au repos pour obtenir une approximation linéaire de la déformation finale.

En ce qui concerne la deuxième demande, nous pouvons jouer avec les modes de déformation d'un objet. Une pratique courante en ingénierie consiste à étudier les fréquences naturelles de vibration à l'aide de l'analyse des valeurs propres. Grâce à cette dernière, le résultat est double puisqu'elle permet d'obtenir les les fréquences naturelles ainsi que les formes des vibrations. Ces formes de vibration sont appelées réponses de vibration libre non amorties de la structure ou modes de déformation. L'analyse des valeurs propres se concentre généralement sur les premières valeurs propres du modèle. Cela s'explique principalement par le fait que le modèle d'éléments finis se rapproche de la forme réelle. Par conséquent, il modélise correctement les fréquences spatiales les plus basses tout en négligeant les fréquences spatiales plus élevées. En d'autres termes, les premiers modes correspondent aux déformations les plus courantes de l'objet, comme le montre la Figure 4.1. En outre, les formes de mode sont normalisées par rapport au déplacement maximal de la structure.

En combinant ces deux considérations, nous pouvons utiliser l'analyse des valeurs propres de la matrice de rigidité tangente pour étudier les modes de déformation de

l'objet. En échantillonnant les quatre ou cinq premiers modes avec des coefficients entre -1 et 1, nous pouvons générer les déformations les plus courantes de l'objet. De plus, un simple produit matriciel permet d'associer cette déformation à la force correspondante. Par conséquent, nous pouvons produire la force responsable de l'approximation linéaire de la déformation réelle en utilisant seulement quatre ou cinq de coefficients. Nous pouvons maintenant utiliser cette force dans une simulation pour obtenir la déformation non linéaire associée.

La seconde partie de cette section du résumé mène une étude comparative entre deux architectures de réseaux neuronaux artificiels utilisées pour prédire les déformations non linéaires à partir d'une force d'entrée. La première, utilisée par Mendizabal et al. [75], est un CNN appelé U-Net ou U-Mesh dans notre context, tandis que la seconde est un MLP composé de cinq couches.

Cette étude présente le premier cas de test sur une poutre à section carrée fixée d'un côté par des conditions limites de Dirichlet. Les résultats montrent que, pour le scénario de la poutre, le MLP est légèrement plus performant en moyenne que l'U-Mesh lorsqu'il est testé sur des données tirées de la même distribution que les données d'apprentissage (Tableau 4.2a). Cependant, U-Mesh présente de meilleures capacités de généralisation que le MLP (Tableau 4.2b). Les temps de formation absolus n'ont pas été rapportés car les deux réseaux ont été formés sur des machines différentes. Cependant, sur des problèmes similaires, le MLP s'est avéré beaucoup plus rapide à former que l'U-Mesh. En outre, le MLP est environ dix fois plus rapide que l'U-Mesh pour faire des prédictions, ce qui en fait l'option préférée pour ce problème.

Le deuxième test a été réalisé sur un foie. Les résultats obtenus lors de cette expérience sont satisfaisants en termes de précision et de temps de prédiction. Le MLP a produit des prédictions plus précises en moyenne que l'U-Mesh, mais ses erreurs étaient plus élevées pour certaines valeurs aberrantes en raison de sa faible capacité de généralisation. Ces valeurs aberrantes correspondent aux limites supérieures de l'ensemble de données de test qui ont été mal représentées pendant la formation. L'U-Mesh produit une déformation environ 500 fois plus rapidement que les solveurs MEF standard, qui prennent environ 1 500 ms par solution. Le MLP lui est environ 5 000 fois plus rapide.

Les résultats indiquent que le MLP est plus performant que l'U-Mesh pour les maillages à petite résolution concernant la précision de la prédiction et la vitesse d'apprentissage. Ces résultats, pouvant sembler surprenant de prime abord, peuvent être expliquer de la manière suivante : Premièrement, l'U-Mesh nécessite une structure de grille régulière avec de nombreux zéros, ce qui augmente la taille de l'entrée sans fournir d'informations supplémentaires. Il en résulte un écart de performance plus important entre les deux réseaux dans les scénarios comportant davantage de zéros sans signification, comme

le scénario du foie. Une solution possible pour réduire cet écart consiste à utiliser un masque géométrique dans la fonction de coût, qui peut créer des contraintes sur la connaissance du domaine. Une solution plus sophistiquée impliquant des convolutions éparses peut s'avérer nécessaire dans de tels cas. Deuxièmement, l'U-Mesh utilise un espace latent réduit pour représenter le problème, ce qui permet une meilleure généralisation mais se traduit par une précision légèrement inférieure pour les scénarios spécifiques aux patients. Toutefois, la précision peut être améliorée en augmentant la taille de l'espace latent, bien que cela augmente le temps de formation et nécessite un compromis entre la précision et l'efficacité.

En résumé, le MLP et l'U-Mesh atteignent une précision similaire dans les scénarios étudiés tout en étant des ordres de grandeur plus rapides que les solveurs MEF pour simuler les déformations non linéaires. Le MLP est notamment, en moyenne, un ordre de grandeur plus rapide que l'U-Mesh. Un avantage notable de cette approche est l'absence de réglage fin dans la génération des données et l'entraînement, grâce à une méthode de génération automatique des forces basée sur l'analyse modale et les techniques de normalisation des données. Compte tenu des limites et des propriétés des réseaux, nous avons décidé de poursuivre nos études et nos recherches avec le réseau entièrement connecté.

Hybrid solver

Notre choix de réseau s'est principalement porté sur la prédiction rapide. En effet, il est souvent plus facile de prendre quelque chose de rapide et de le rendre plus fiable que de faire l'inverse. Le quatrième chapitre (chapitre 5) de cette thèse se concentre sur la combinaison de notre réseau de neurones artificiels, qui produit des prédictions rapides, et de l'algorithme de Newton-Raphson, qui produit des résultats fiables.

Nous commençons par une présentation plus approfondie de la méthode de Newton-Raphson. La méthode de Newton ou de Newton-Raphson est une méthode d'optimisation convexe du second ordre. Elle repose sur l'optimisation d'une fonction à l'aide de son approximation quadratique. Cette approximation calcule des informations sur la fonction en utilisant le hessien, ce qui la rend plus pertinente que la descente de gradient classique. Dans la méthode de Newton, nous nous déplaçons vers le Hessien négatif inverse du gradient. Cette opération est répétée jusqu'à ce que le pas soit arbitrairement considéré comme trop petit ou que la fonction soit suffisamment proche de son minimum.

En outre, la convergence de l'algorithme peut être divisée en deux phases. Dans la première phase, appelée phase de Newton amortie, le critère diminue d'une quantité prévis-

ible qui dépend des paramètres de l'algorithme de recherche linéaire à rebours. Après un nombre de pas qui dépend de la simulation, nous satisfaisons un critère qui nous place dans la phase de Newton pure, et le taux de convergence devient quadratique.

Il est parfois difficile de savoir à l'avance si le problème respecte le critère de convergence de la méthode de Newton. Des problèmes surviennent lorsqu'un point d'itération est stationnaire ou que le gradient est presque nul puisque nous multiplions par l'inverse de la dérivée de la fonction. Un dernier problème lié aux points d'itération est que l'algorithme peut entrer dans un cycle, ce qui signifie qu'il ne converge jamais. Les dérivées peuvent également ne pas exister ou être discontinues à la racine. Globalement, la méthode de Newton est très sensible au point de départ et à la qualité de la dérivée.

Notre contribution propose d'utiliser un réseau neuronal artificiel pour se placer dans la phase de convergence quadratique le plus tôt possible. L'idée est d'initialiser la méthode de Newton avec une prédiction d'un réseau neuronal artificiel formé à cet effet. Par conséquent, la prédiction du réseau neuronal est utilisée si elle constitue un meilleur point de départ que le résultat de la première itération. La comparaison est effectuée à l'aide de l'équation minimisée. Bien que la méthode produise généralement des solutions qualitatives, si la force d'entrée est trop différente de l'ensemble de données d'entraînement, elle peut produire des solutions erronées qui seront préjudiciables à l'algorithme. De cette manière, nous nous plaçons toujours dans le meilleur scénario possible en utilisant la supposition initiale qui minimise la fonction.

Cette contribution est testée sur deux cas, l'un utilisant une poutre en porte-à-faux et l'autre une hélice. Pour mettre en évidence la capacité de notre approche à apprendre sur des formes et des propriétés matérielles très différentes, nous avons sélectionné un maillage d'hélice avec une loi d'hyper-élasticité néo-hookéenne et une poutre en porte-à-faux avec une loi d'hyper-élasticité de Saint-Venant-Kirrchhof. Les deux ont environ 12 000 degrés de liberté. En revanche, leurs paramètres de matériaux sont très différents, avec une faible rigidité pour le modèle de poutre et une rigidité très élevée pour l'hélice, afin d'évaluer la validité du processus d'apprentissage et des prédictions. Les dimensions des deux structures ont également été choisies pour être très différentes pour la même raison. La poutre extrêmement souple permet des déplacements de dizaines de mètres à partir de petites forces externes. Ces scénarios extrêmes sont généralement le point faible des réseaux neuronaux artificiels. En outre, compte tenu de sa géométrie, la poutre a tendance à se déformer globalement, même sous l'effet de forces locales. Une prédiction précise exige que le réseau transfère les informations de déformation nécessaires à tous les nœuds. Au contraire, des forces importantes sont nécessaires pour que l'hélice fournisse des déplacements relativement faibles, de l'ordre du centimètre. Cependant, sa géométrie est telle qu'avec des forces locales, elle affiche des déformations au niveau des

pales sans aucun déplacement sur les pales voisines.

Tout d'abord, nous nous assurons que notre réseau neuronal prédit des déformations exactes. Notre ANN obtient des résultats similaires pour les deux modèles, avec une erreur quadratique moyenne (EQM) d'environ 10^{-6}. Nous montrons ainsi que notre apprentissage n'est pas spécifique à un type de problème.

Nous injectons ensuite notre solution telle que proposée par l'algorithme hybride de Newton-Raphson. Sur les 100 échantillons de test, l'algorithme de Newton-Raphson hybride a choisi deux fois sur trois la prédiction du réseau comme meilleur point de départ que le résultat de la première itération de l'algorithme de Newton-Raphson. À partir de ce point, l'algorithme converge en moyenne en 5 itérations. Cela montre qu'à partir de la prédiction, l'algorithme converge en moyenne en 4 itérations, en ajoutant 5 pour tenir compte de la première itération rejetée lorsque la prédiction est choisie.

Dans l'ensemble, l'algorithme proposé converge plus souvent et plus rapidement que la méthode classique tout en conservant les propriétés de convergence ordinaires de l'algorithme de Newton-Raphson.

Optimal control for augmented surgery

Nous avons ainsi une première réponse sur la façon de prédire rapidement les déformations non linéaires d'un corps mou et sur la façon de vérifier et de corriger ces prédictions. Il est maintenant temps de plonger dans l'aspect médical de cette thèse avec le cinquième chapitre (chapitre 6), qui introduit le problème du recalage non rigide.

La résection hépatique ou hépatectomie est une procédure chirurgicale qui consiste à retirer une partie ou la totalité du foie. Le foie partiellement enlevé peut reprendre sa taille initiale, alors que la résection totale du foie nécessite une transplantation. En raison de l'extrême capacité de régénération du foie, la résection hépatique est une pratique courante en cas de maladie du foie. Bien que l'opération soit souvent pratiquée, elle reste une compliquée en raison de la densité des vaisseaux dans le foie, qui peut provoquer des saignements importants.

Il existe deux méthodes principales pour réaliser une résection du foie : la chirurgie abdominale ouverte et la chirurgie laparoscopique. La plus ancienne est la chirurgie abdominale ouverte, qui consiste à pratiquer une longue incision unique, également appelée laparotomie, pour accéder à la cavité abdominale. La plus récente est la chirurgie laparoscopique, où la procédure est effectuée par de petites incisions dans l'abdomen à l'aide d'outils allongés et d'une caméra appelée laparoscope.

Ces travaux visent à fournir des outils permettant d'introduire la réalité augmentée

dans la salle d'opération en projetant les structures internes de l'organe sur le flux vidéo des chirurgiens. L'introduction d'informations 3D internes dans l'affichage pourrait aider le praticien en offrant une meilleure visualisation de l'opération et également aider à la prise de décision. Cette projection nécessite tout d'abord l'extraction de la déformation actuelle de l'organe à partir du flux vidéo et, ensuite, la déformation de la structure interne et sa re-projection sur le flux vidéo.

Dans notre travail, nous nous concentrons sur la première tâche : *Calculer la déformation actuelle de l'organe à partir du flux vidéo.* L'extraction des déformations à partir d'un flux vidéo nécessite de trouver le recalage rigide, d'extraire la surface de l'image puis de calculer la déformation. Cette thèse se concentre davantage sur l'aspect théorique de la méthode que sur les résultats expérimentaux réels. Nous émettons donc deux hypothèses qui nous permettent de nous concentrer uniquement sur le calcul de la déformation. La première est que recalage rigide est déjà effectué. La seconde est que nous pouvons extraire un nuage de points 3D à partir d'images couleur. Ces deux hypothèses sont des sujets de recherche réels et pourraient faire l'objet de thèses à elles seules. Afin de bien comprendre notre travail, nous allons étudier l'impact de ces deux hypothèses sur notre méthode.

A partir du flux vidéo, nous disposons d'une surface 3D partielle de l'objet. Ce nuage de points est intéressant car il fournit des informations précieuses sur la déformation. En utilisant cette déformation, nous devrions être en mesure d'extraire la force qui l'a générée. Ce raisonnement a été prouvé par Mestdagh et al. [78]. Dans ce travail, ils utilisent le contrôle optimal et la méthode adjointe pour récupérer les forces qui déforment un foie composé d'un matériau non linéaire pour s'adapter à un nuage de points de surface partielle. En d'autres termes, ils trouvent les forces telles que le foie déformé correspond aux données observées.

Une fois que nous avons les forces, nous savons que la déformation d'un foie et de sa structure interne est calculée en quelques millisecondes en utilisant les travaux présentées précédemment.

Notre contribution consiste principalement à accélérer les calculs présentés à l'aide d'un réseau neuronal artificiel.

L'algorithme proposé par Mestdagh et al. [78] comprend quatre étapes principales. La première consiste à calculer la déformation du modèle sous l'effet d'une charge externe, également appelée problème à terme. La deuxième étape évalue la distance entre le modèle et les données d'observation (surface reconstruite). La troisième étape utilise la formulation d'un problème adjoint pour calculer le gradient de la charge externe. Enfin, la dernière étape met à jour la charge externe pour réduire la distance entre le modèle et la surface reconstruite. Ces quatre étapes sont répétées jusqu'à ce qu'un critère arbitraire

soit satisfait.

Dans ce processus, nous avons identifié deux goulots d'étranglement principaux : la première étape (problème direct) et la troisième étape (problème adjoint).

Le problème direct est résolu à l'aide de l'approche MEF classique, où un algorithme de Newton est utilisé pour calculer la déformation non linéaire d'un corps mou soumis à une charge externe. Notre contribution à la réduction de ce goulot d'étranglement consiste à utiliser notre réseau neuronal artificiel pour calculer les simulations directes. Par identification, nous pouvons voir des similitudes entre le contrôle de cette méthode et notre travail précédent. Les deux entrées/contrôles sont des forces externes ; les deux sorties sont les déformations résultantes. Nous remplaçons la simulation par un ANN qui effectue les mêmes opérations mais beaucoup plus rapidement, comme le montre le chapitre 4. Grâce à cette amélioration, le coût d'une simulation passe de plusieurs secondes à moins d'une milliseconde.

Le deuxième goulot d'étranglement apparaît lors de l'évaluation du problème adjoint. Cette étape est responsable du calcul du gradient de la fonction de coût utilisée pour mettre à jour la commande. Le calcul du gradient s'effectue en résolvant un système linéaire qui transforme le gradient de la fonction de coût de l'espace des déplacements à l'espace des forces. La solution linéaire réduit l'évolutivité et la vitesse de l'algorithme, ce qui nous écarte du critère de temps réel.

Afin d'accélérer le calcul, nous aimerions éviter ce système tout en étant en mesure de calculer le gradient de l'évaluation de la simulation. Nous utilisons déjà un cadre d'apprentissage profond grâce au réseau neuronal ajouté dans la boucle. Nous pouvons donc facilement utiliser les outils associés, tels que la différenciation automatique et la rétropropagation. L'algorithme de rétropropagation présenté dans la section 3.5 repose sur la règle de la chaîne pour calculer le gradient de la fonction de coût par rapport à toute variable impliquée dans le calcul. La règle de la chaîne ne nécessite pas la résolution d'un système linéaire, mais repose sur un graphique de calcul. Cela signifie que le contrôle doit être impliqué dans le calcul (entrée du réseau), de cette façon nous pouvons différencier le résultat de l'évaluation par rapport à la force, obtenant ainsi $\nabla\Psi$.

Nous avons remplacé le problème direct, qui prend du temps, par un réseau neuronal artificiel et le problème de l'adjoint par l'algorithme de rétropropagation. Nous testons notre méthode sur un exemple fictif de poutre en porte-à-faux et effectuons un test plus approfondi sur cinq scénarios plus réalistes à l'aide d'un modèle de foie.

Concernant le test de la poutre, le calcul de chaque échantillon de l'ensemble de données de test a pris entre 1 et 2 secondes. En moyenne, recalage non rigide a pris 48 millisecondes pour une erreur moyenne de recalage de la cible de 5.37×10^{-5} . Notre méthode produit un recalage non rigide précis et presque en temps réel des matériaux

non linéaires. D'après notre analyse, la répartition temporelle des différentes tâches de l'algorithme est cohérente, même avec des maillages plus denses. Les prédictions du réseau et les évaluations de la fonction de coût représentent chacune entre 10 et 15 % du temps de calcul, alors que l'évaluation du gradient représentent les 70-80% restant de l'ensemble du processus d'optimisation.

Les cinq scénarios présentent des résultats similaires avec des erreur de recalage compris entre 3,5 mm et 0,5 mm. Ces erreurs sont tout à fait acceptables et préservent les propriétés physiques du maillage enregistré. Nous soulignons que l'erreur de recalage moyenne pour la méthode classique est d'environ 0,1 mm, ce qui montre l'impact des approximations du réseau.

En raison de la non-linéarité introduite par le matériau néo-hookéen utilisé pour simuler le foie, nous avons besoin de plusieurs itérations pour converger vers le nuage de points cible. Compte tenu de la complexité du maillage, le calcul d'une seule itération de l'algorithme à l'aide d'un solveur classique prend plusieurs secondes, ce qui se traduit par une moyenne de 14 minutes par image. L'algorithme que nous proposons utilise un réseau neuronal pour améliorer la vitesse de calcul des problèmes hyperélastiques et adjoints. Le calcul du problème hyperélastique prend environ 4 à 5 millisecondes, tandis que le problème adjoint remplacé par l'algorithme de rétropropagation prend environ 11 ms. Cela conduit à une amélioration significative de la vitesse de convergence, comme le montre 6.11 où nous réduisons le temps de calcul par un facteur de 6000 en moyenne.

Enfin, nous examinons l'impact de notre hypothèse sur notre travail. Jusqu'à présent, nous avons considéré une reconstruction parfaite de la surface et un recalage rigide. Dans la suite de notre étude nous appliquons du bruit sur ces deux paramètres pour un échantillon choisi arbitrairement. Les résultats montrent que notre méthode est assez résistante dans la gamme de bruit que nous attendons de l'hypothèse et qu'elle est donc bien adaptée à ce type de problème.

Nous avons présenté notre approche de la combinaison du cadre de contrôle optimal avec l'apprentissage profond. La facilité d'accès aux différentes dérivées du modèle est un point fort d'une telle méthode. D'un autre côté, le sujet de la représentation apprise peu entacher la fiabilité de ces dérivées. De plus, chaque dérivée nécessite une rétropropagation dans l'ensemble du réseau, ce qui peut être chronophage.

Enfin, le recalage est simple mais assez difficile à contrôler. Le modèle peut affiner la solution de lui-même, mais nécessite de multiples techniques pour s'assurer qu'il se concentre sur les régions d'intérêt.

Pour remédier à cet inconvénient, il faut passer à l'étape suivante et écrire un moteur de simulation utilisant la différenciation automatique. Cela nous permettra d'incorporer efficacement et naturellement l'apprentissage profond dans la simulation sans dépendre

uniquement des données générées par la boîte noire et d'une fonction de coût physique. De tels simulateurs sont appelés simulateurs différentiables et, dans notre cas, simulateurs physiques différentiables.

Lastest optimisation tool: Differentiable simulation

Notre dernier chapitre (chapitre 7) porte sur les simulateurs différentiables. Dans ce chapitre, nous présentons des résultats prometteurs non publiés sur le développement d'un simulateur MEF différentiable basé sur la formulation énergétique. Le simulateur développé dans PyTorch prend en charge plusieurs types et degrés d'éléments et de matériaux. Nous commençons par présenter comment nous calculons efficacement les différents membres de l'équation dans PyTorch. Enfin, nous présentons comment résoudre des problèmes direct et inverse similaires à ceux présentés dans le chapitre précédent (chapitre 6).

Dans le cadre de la MEF, l'énergie globale est calculée en accumulant l'énergie locale sur chaque élément. Chaque énergie locale est un scalaire indépendant. Par conséquent, le calcul est hautement parallélisable et efficace en termes de mémoire, ce qui est parfait pour les calculs sur GPU.

Ces affirmations ne sont pas valables lorsque l'on parle de forces internes dérivant de l'énergie interne. Les forces internes sont des quantités nodales qui dérivent d'une quantité élémentaire. Comme notre moteur est écrit en PyTorch et que ce framework permet de calculer facilement sur le GPU, nous avons préféré travailler avec l'énergie pour maximiser les performances.

Le problème est alors formulé comme la minimisation de la différence entre deux énergies potentielles. Celle-ci représente l'énergie potentielle élastique et dépend de la déformation actuelle du modèle. La seconde est l'énergie potentielle linéaire et correspond au produit du déplacement par la charge.

Pour paralléliser efficacement le calcul de chaque membre, nous avons dû introduire une représentation topologique spécifique de l'objet. A la place de contenir les indices des nœuds, le tableau topologique contient directement la position des nœuds associé a chaque éléments. Bien que cela duplique les données, cela supprime le besoin d'une boucle for lente. Toute optimisation a un effet double avec PyTorch puisqu'elle accélère également la rétropropagation.

Le tableau de positions étant plus complexe, nous utilisons la notation d'Einstein pour effectuer nos calculs. Grâce à cette notation, PyTorch permet de formuler rapidement des opérations tensorielles complexes. Ainsi, nous n'écrivons que les indices sur lesquels nous effectuons la multiplication et l'addition dans une chaîne de caractères qui sert de

modèle pour le calcul des tenseurs donnés. Cette notation a également le bon goût d'être extrêmement rapide à écrire et à calculer, même pour des opérations de base telles que la multiplication matrice-vecteur, où elle est plus performante que la fonction par défaut.

Le calcul de l'énergie potentielle élastique se fait en calculant les tenseurs dans l'ordre où nous les avons présentés dans le chapitre 2. Nous commençons par multiplier la position par la dérivée de la fonction de forme. Cela nous donne le gradient de déformation $\mathbf{F}$. Nous multiplions ensuite sa transposée par lui-même, ce qui nous donne le tenseur de déformation de Cauchy-Green $\mathbf{C}$. Nous y soustrayons l'identité, ce qui nous donne le tenseur de déformation de Green-Lagrange $\mathbf{E}$. Nous disposons de toutes les quantités nécessaires pour calculer le second tenseur des contraintes de Piola-Kirrchhoff $\mathbf{S}$, qui donne la réponse du matériau à une déformation donnée. Avec quelques étapes supplémentaires qui introduiraient des équations dans ce résumé, nous aboutissons à une somme de l'énergie par élément qui nous donne l'énergie potentielle élastique totale.

Un simple produit scalaire entre le champ de déplacement et la charge externe permet de calculer l'énergie potentielle linéaire.

Enfin, un aspect essentiel de l'élasticité statique est constitué par les conditions aux limites de Dirichlet, sans lesquelles le système n'a pas de solution. Afin de résoudre le problème tout en conservant la chaîne de dérivation, nous les appliquons ces contraintes de manière douce. Pour se faire nous ajoutons à la fonction de minimisation la norme quadratique du déplacement des nœuds correspondants.

Cela nous donne une évaluation du système. Cependant afin pour minimiser cette fonction nous devons optimiser le champ de déplacement. Pour ce faire, nous ajoutons une étape de l'algorithme de Newton-Raphson après l'évaluation. Nous répétons ensuite ce processus jusqu'à ce qu'un critère soit atteint.

Le moteur a été conçu pour atteindre deux objectifs principaux. Le premier est de pouvoir calculer la solution de problèmes de corps mous pour des matériaux linéaires et non linéaires composés d'éléments linéaires ou quadratiques. Le second s'appuie sur le premier et impose la condition que la solution du problème de corps mou soit différentiable en utilisant l'algorithme de rétropropagation.

Cela signifie que l'algorithme de Newton-Raphson doit être différentiable si nous voulons utiliser le moteur pour effectuer l'optimisation du contrôle comme nous l'avons fait dans le chapitre précédent (chapitre 6). Pour ce faire, nous avons dû implémenter un algorithme de Newton-Raphson différentiable avec une résolution linéaire différentiable et une recherche linéaire à rebours. Par conséquent, nous pouvons brancher notre moteur différentiable à la place du RNA et effectuer le calcul exact, mais ici, la simulation directe est une simulation MEF réelle, et l'algorithme de rétropropagation remplace le problème adjoint.

Comme le moteur n'est pas spécifique et qu'il comporte de multiples variables (maillage, module d'Young, coefficient de Poisson, etc.), nous pouvons effectuer une optimisation en fonction d'un de ces paramètres.

Dans le résultat, nous montrons les capacités du moteur en mettant en évidence les différents paramètres que nous pouvons définir comme contrôles. Comme ces résultats n'ont rien de scientifique, nous ne ferons qu'une brève présentation de l'étude, sans présenter de chiffres ni mener d'étude comparative dans le résumé. Nous commençons par un aperçu de la force comme contrôle de la simulation, où nous discutons de l'impact de la connaissance de l'utilisateur et de la valeur initiale sur le processus d'optimisation. En utilisant notre moteur, nous présentons ensuite l'estimation du module d'Young's et du coefficient de Poisson. Nous laissons le lecteur intéressé sauter à la section correspondante (section 7.2).

En résumé, nous avons construit un moteur de physique des corps mou différentiable qui peut être optimisé pour n'importe quel paramètre de simulation à l'aide du cadre de contrôle optimal et de l'algorithme de rétropropagation. Bien que le moteur soit stable et réponde à toutes nos attentes, il peut être amélioré. Le temps de calcul de la matrice de rigidité tangente est actuellement trop long pour une utilisation en temps réel. Il n'est pas encore clair si cela est dû à la courte durée de développement ou à une limitation de PyTorch.

BIBLIOGRAPHY

[1] J. Allard, H. Courtecuisse, and F. Faure. Implicit FEM and Fluid Coupling on GPU for Interactive Multiphysics Simulation. In M. Elendt, editor, SIGGRAPH Talks, page Article No. 52, Vancouver, Canada, Aug. 2011. ACM.

[2] J. Allard, H. Courtecuisse, and F. Faure. Implicit FEM Solver on GPU for Interactive Deformation Simulation. In W. mei W. Hwu, editor, GPU Computing Gems Jade Edition, Applications of GPU Computing Series, pages 281–294. Elsevier, Nov. 2011.

[3] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC '09, New York, NY, USA, 2009. NVidia, Association for Computing Machinery.

[4] S. E. Benzley, E. Perry, K. Merkley, B. Clark, and G. Sjaardema. A comparison of all hexagonal and all tetrahedral finite element meshes for elastic and elasto-plastic analysis. In In Proceedings, 4th International Meshing Roundtable, pages 179–191, 1995.

[5] G. Berkooz, P. Holmes, and J. L. Lumley. The proper orthogonal decomposition in the analysis of turbulent flows. Annual review of fluid mechanics, 25(1):539–575, 1993.

[6] A. Bernardin, E. Coevoet, P. Kry, S. Andrews, C. Duriez, and M. Marchal. Constraint-based Simulation of Passive Suction Cups. ACM Transactions on Graphics, 42(1):1–14, Feb. 2023.

[7] P. J. Besl and N. D. McKay. Method for registration of 3-d shapes. In Sensor fusion IV: control paradigms and data structures, volume 1611, pages 586–606. Spie, 1992.

[8] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[9] S. Brenner and R. Scott. The mathematical theory of finite element methods, volume 15. Springer Science & Business Media, 2007.

[10] C. G. Broyden. Quasi-newton methods and their application to function minimisation. Mathematics of Computation, 21(99):368–381, 1967.

[11] J. Bruix and M. Sherman. Management of hepatocellular carcinoma. Hepatology, 42(5):1208–1236, 2005.

[12] J.-N. Brunet, V. Magnoux, B. Ozell, and S. Cotin. Corotated meshless implicit dynamics for deformable bodies. In WSCG 2019 - 27th International Conference on Computer Graphics, Visualization and Computer Vis
Pilsen, Czech Republic, May 2019. Západočeská univerzita.

[13] J.-N. Brunet, A. Mendizabal, A. Petit, N. Golse, E. Vibert, and S. Cotin. Physics-based deep neural network for augmented reality during liver surgery. In International Conference on Medical image computing and computer-assisted intervention, pages 137–145. Springer, 2019.

[14] A. Chatterjee. An introduction to the proper orthogonal decomposition. Current science, pages 808–817, 2000.

[15] J. Chen, H. Li, F. Liu, B. Li, and Y. Wei. Surgical outcomes of laparoscopic versus open liver resection for hepatocellular carcinoma for various resection extent. Medicine, 96(12), 2017.

[16] N. Chentanez, M. Macklin, M. Müller, S. Jeschke, and T.-Y. Kim. Cloth and skin deformation with a triangle mesh based convolutional neural network. Computer Graphics Forum, 39(8):123–134, 2020.

[17] E. Cueto and F. Chinesta. Thermodynamics of learning physical phenomena, 2022.

[18] S. Dastjerdi, B. Akgöz, and Ö. Civalek. On the shell model for human eye in glaucoma disease. International Journal of Engineering Science, 158:103414, 2021.

[19] S. Deshpande, J. Lengiewicz, and S. Bordas. Probabilistic deep learning for real-time large deformation simulations. 2021.

[20] R. Dreyfus, Q. Boehler, and B. J. Nelson. A Simulation Framework for Magnetic Continuum Robots. IEEE Robotics and Automation Letters, 7(3):8370 – 8376, June 2022.

[21] Drugs.com. Laparoscopic surgery. https://www.drugs.com/cg/laparoscopic-liver-biopsy.html.

[22] I. Duff, J. Hogg, and F. Lopez. A new sparse ldlˆt solver using a posteriori threshold pivoting. SIAM Journal on Scientific Computing, 42(2):C23–C42, 2020.

[23] M. Duprez and A. Lozinski. ϕ-fem: A finite element method on domains defined by level-sets. SIAM Journal on Numerical Analysis, 58(2):1008–1028, 2020.

[24] A. Dziekonski, P. Sypek, A. Lamecki, and M. Mrozowski. Finite element matrix generation on a gpu. Progress In Electromagnetics Research, 128:249–265, 01 2012.

[25] A. Düster, J. Parvizian, Z. Yang, and E. Rank. The finite cell method for three-dimensional problems of solid mechanics. Computer Methods in Applied Mechanics and Engineering, 197(45):3768–3782, 2008.

[26] N. B. Erichson, M. Muehlebach, and M. W. Mahoney. Physics-informed autoencoders for lyapunov-stable fluid flow prediction, 2019.

[27] C. Farhat, M. Lesoinne, P. LeTallec, K. Pierson, and D. Rixen. Feti-dp: a dual–primal unified feti method—part i: A faster alternative to the two-level feti method. International journal for numerical methods in engineering, 50(7):1523–1544, 2001.

[28] F. Faure, C. Duriez, H. Delingette, J. Allard, B. Gilles, S. Marchesseau, H. Talbot, H. Courtecuisse, G. Bousquet, I. Peterlik, and S. Cotin. SOFA: A Multi-Model Framework for Interactive Physical Simulation. In Soft Tissue Biomechanical Modeling for Computer Assisted Surgery, volume 11 of Studies in Mechanobiology, Tissue Engineering and Biomaterials, pages 283–321. Springer, 2012.

[29] R. Feinman. Pytorch-minimize: a library for numerical optimization with autograd, 2021.

[30] J. B. Freund, J. F. MacArt, and J. Sirignano. Dpm: A deep learning pde augmentation method (with application to large-eddy simulation), 2019.

[31] K. Fukushima and S. Miyake. Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition. In Competition and cooperation in neural nets, pages 267–285. Springer, 1982.

[32] H. Gao, X. Ma, N. Qi, C. Berry, B. E. Griffith, and X. Luo. A finite strain nonlinear human mitral valve model with fluid-structure interaction. International journal for numerical methods in biomedical engineering, 30(12):1597–1613, 2014.

[33] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl. Neural message passing for quantum chemistry, 2017.

[34] M. Gori, G. Monfardini, and F. Scarselli. A new model for learning in graph domains. In Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005., volume 2, pages 729–734. IEEE, 2005.

[35] O. Goury, B. Carrez, and C. Duriez. Real-time simulation for control of soft robots with self-collisions using model order reduction for contact forces. IEEE Robotics and Automation Letters, 6(2):3752–3759, 2021.

[36] G. Guennebaud, B. Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[37] R. Gulakala, B. Markert, and M. Stoffel. Graph neural network enhanced finite element modelling. PAMM, 22(1):e202200306, 2023.

[38] R. Haferssas, P.-H. Tournier, F. Nataf, and S. Cotin. Simulation of soft tissue deformation in real-time using domain decomposition. INRIA, 2019.

[39] N. Haouchine, J. Dequidt, I. Peterlik, E. Kerrien, M.-O. Berger, and S. Cotin. Image-guided simulation of heterogeneous tissue deformation for augmented reality during hepatic surgery. In 2013 IEEE international symposium on mixed and augmented reality (ISMAR), pages 199–208. IEEE, 2013.

[40] J. He, D. Abueidda, S. Koric, and I. Jasiuk. On the use of graph neural networks and shape-function-based gradient computation in the deep energy method. International Journal for Numerical Methods in Engineering, 124(4):864–879, 2023.

[41] J. S. Heiselman, W. R. Jarnagin, and M. I. Miga. Intraoperative correction of liver deformation using sparse surface and vascular features via linearized iterative boundary reconstruction. IEEE transactions on medical imaging, 39(6):2223–2234, 2020.

[42] P. Holl, V. Koltun, K. Um, and N. Thuerey. phiflow: A differentiable pde solving framework for deep learning via physical simulations. In NeurIPS workshop, volume 2, 2020.

[43] A. Holynski, B. L. Curless, S. M. Seitz, and R. Szeliski. Animating pictures with eulerian motion fields. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), pages 5810–5819. University of Washington, June 2021.

[44] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. Neural networks, 2(5):359–366, 1989.

[45] A. D. Jagtap, E. Kharazmi, and G. E. Karniadakis. Conservative physics-informed neural networks on discrete domains for conservation laws: Applications to forward and inverse problems. Computer Methods in Applied Mechanics and Engineering, 365:113028, 2020.

[46] K. M. Jatavallabhula, M. Macklin, F. Golemo, V. Voleti, L. Petrini, M. Weiss, B. Considine, J. Parent-Levesque, K. Xie, K. Erleben, L. Paull, F. Shkurti, D. Nowrouzezahrai, and S. Fidler. gradsim: Differentiable simulation for system identification and visuomotor control. International Conference on Learning Representations (ICLR), 2021.

[47] S. Karumuri, R. Tripathy, I. Bilionis, and J. Panchal. Simulator-free solution of high-dimensional stochastic elliptic partial differential equations using deep neural networks. Journal of Computational Physics, 404:109120, 2020.

[48] H. J. Kelley. Gradient theory of optimal flight paths. Ars Journal, 30(10):947–954, 1960.

[49] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang. On large-batch training for deep learning: Generalization gap and sharp minima. arXiv preprint arXiv:1609.04836, 2016.

[50] E. Kharazmi, Z. Zhang, and G. E. Karniadakis. hp-vpinns: Variational physics-informed neural networks with domain decomposition. Computer Methods in Applied Mechanics and Engineering, 374:113547, 2021.

[51] D. Kim, W. Koh, R. Narain, K. Fatahalian, A. Treuille, and J. F. O'Brien. Near-exhaustive precomputation of secondary cloth effects. ACM Transactions on Graphics (TOG), 32(4):1–8, 2013.

[52] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization, 2017.

[53] D. Kochkov, J. A. Smith, A. Alieva, Q. Wang, M. P. Brenner, and S. Hoyer. Machine learning accelerated computational fluid dynamics. arXiv preprint arXiv:2102.01010, 2021.

[54] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. Commun. ACM, 60(6):84–90, may 2017.

[55] P. L. Lagari, L. Tsoukalas, S. Safarkhani, and I. Lagaris. Systematic construction of neural forms for solving partial differential equations inside rectangular domains, subject to initial, boundary and interface conditions. International Journal on Artificial Intelligence Tools, 29, 05 2020.

[56] I. Lagaris, A. Likas, and D. Fotiadis. Artificial neural networks for solving ordinary and partial differential equations. IEEE Transactions on Neural Networks, 9(5):987–1000, 1998.

[57] M. Lefik and B. A. Schrefler. Artificial neural network as an incremental non-linear constitutive model for a finite element code. Computer methods in applied mechanics and engineering, 192(28-30):3265–3283, 2003.

[58] D. Li, K. Xu, J. M. Harris, and E. Darve. Coupled time-lapse full-waveform inversion for subsurface flow problems using intrusive automatic differentiation. Water Resources Research, 56(8):e2019WR027032, 2020.

[59] Y. Li, T. Du, K. Wu, J. Xu, and W. Matusik. Diffcloth: Differentiable cloth simulation with dry frictional contact. ACM Trans. Graph., 42(1), oct 2022.

[60] Z. Li, N. Kovachki, K. Azizzadenesheli, B. Liu, K. Bhattacharya, A. Stuart, and A. Anandkumar. Fourier neural operator for parametric partial differential equations, 2021.

[61] S. Linnainmaa. The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors. PhD thesis, Master's Thesis (in Finnish), Univ. Helsinki, 1970.

[62] P.-L. Lions. On the schwarz alternating method. iii: a variant for nonoverlapping subdomains. In Third international symposium on domain decomposition methods for partial differential equations, volume 6, pages 202–223. SIAM Philadelphia, PA, 1990.

[63] D. G. MacManus, N. Chiereghin, D. G. Prieto, and P. Zachos. Complex aeroengine intake ducts and dynamic distortion. AIAA Journal, 55(7):2395–2409, 2017.

[64] Y. Maday and E. M. Ronquist. The reduced basis element method: application to a thermal fin problem. SIAM Journal on Scientific Computing, 26(1):240–258, 2004.

[65] J. Mandel. Balancing domain decomposition. Communications in numerical methods in engineering, 9(3):233–241, 1993.

[66] S. Marchesseau, S. Chatelin, and H. Delingette. Nonlinear biomechanical model of the liver. pages 243–265, 2017.

[67] J. Martínez-Frutos, P. J. Martínez-Castejón, and D. Herrero-Pérez. Fine-grained gpu implementation of assembly-free iterative solver for finite element problems. Computers & Structures, 157:9–18, 2015.

[68] Y. M. Marzouk, H. N. Najm, and L. A. Rahn. Stochastic spectral methods for efficient bayesian solution of inverse problems. Journal of Computational Physics, 224(2):560–586, 2007.

[69] M. Maurizi, C. Gao, and F. Berto. Predicting stress, strain and deformation fields in materials and structures with graph neural networks. Scientific Reports, 12(1):21834, 2022.

[70] J. McCarthy, M. L. Minsky, N. Rochester, and C. E. Shannon. A proposal for the dartmouth summer research project on artificial intelligence, august 31, 1955. AI magazine, 27(4):12–12, 2006.

[71] W. Mcculloch and W. Pitts. A logical calculus of ideas immanent in nervous activity. Bulletin of Mathematical Biophysics, 5:127–147, 1943.

[72] K. S. McFall and J. R. Mahan. Artificial neural network method for solution of boundary value problems with exact satisfaction of arbitrary boundary conditions. IEEE Transactions on Neural Networks, 20(8):1221–1233, 2009.

[73] F. Meister, T. Passerini, V. Mihalef, A. Tuysuzoglu, A. Maier, and T. Mansi. Towards fast biomechanical modeling of soft tissue using neural networks. arXiv preprint arXiv:1812.06186, 2018.

[74] M. Mendez, M. Balabane, and J.-M. Buchlin. Multi-scale proper orthogonal decomposition of complex fluid flows. Journal of Fluid Mechanics, 870:988–1036, 2019.

[75] A. Mendizabal, P. Márquez-Neila, and S. Cotin. Simulation of hyper-elastic materials in real-time using deep learning. Medical image analysis, 59:101569, 2020.

[76] A. Mendizabal, A. Odot, and S. Cotin. Chapter 5 - deep learning for real-time computational biomechanics. In F. Chinesta, E. Cueto, Y. Payan, and J. Ohayon, editors, Reduced Order Models for the Biomechanics of Living Organs, Biomechanics of Living Organs, pages 95–126. Academic Press, 2023.

[77] G. Mestdagh. An optimal control formulation for organ registration in augmented surgery. Theses, Université de Strasbourg, Dec. 2022.

[78] G. Mestdagh and S. Cotin. An Optimal Control Problem for Elastic Registration and Force Estimation in Augmented Surgery. In MICCAI 2022 - 25th International Conference on Medical Image Computing and Computer Assisted Singapore, Singapore, Sept. 2022.

[79] M. Minsky and S. Papert. Perceptrons: An Introduction to Computational Geometry. MIT Press, Cambridge, MA, USA, 1969.

[80] B. Moya, A. Badías, D. González, F. Chinesta, and E. Cueto. A thermodynamics-informed active learning approach to perception and reasoning about fluids. Computational Mechanics, pages 1–15, 2023.

[81] E. Mueller, X. Guo, R. Scheichl, and S. Shi. Matrix-free gpu implementation of a preconditioned conjugate gradient solver for anisotropic elliptic pdes. https://arxiv.org/pdf/1302.7193.pdf, 2013.

[82] J. S. Mueller-Roemer and A. Stork. Gpu-based polynomial finite element matrix assembly for simplex meshes. In Computer Graphics Forum, volume 37, pages 443–454. Wiley Online Library, 2018.

[83] I. Murray. Differentiation of the cholesky decomposition. arXiv preprint arXiv:1602.07527, 2016.

[84] A. Odot, R. Haferssas, and S. Cotin. Deepphysics: A physics aware deep learning framework for real-time simulation. International Journal for Numerical Methods in Engineering, 123(10):2381–2398, 2022.

[85] A. Odot, G. Mestdagh, Y. Privat, and S. Cotin. Real-time elastic partial shape matching using a neural network-based adjoint method. In B. Dorronsoro, F. Chicano, G. Danoy, and E.-G. Talbi, editors, Optimization and Learning, pages 137–147, Cham, 2023. Springer Nature Switzerland.

[86] I. Oliveira and A. Patera. Reduced-basis techniques for rapid reliable optimization of systems described by affinely parametrized coercive elliptic partial differential equations. Optimization and Engineering, 8(1):43–65, 2007.

[87] E. Özgür, B. Koo, B. Le Roy, E. Buc, and A. Bartoli. Preoperative liver registration for augmented monocular laparoscopy using backward–forward biomechanical simu-

lation. International journal of computer assisted radiology and surgery, 13:1629–1640, 2018.

[88] H. Park and D. Cho. The use of the karhunen-loeve decomposition for the modeling of distributed parameter systems. Chemical Engineering Science, 51(1):81–98, 1996.

[89] I. Peterlík, C. Duriez, and S. Cotin. Modeling and Real-Time Simulation of a Vascularized Liver Tissue. In MICCAI 2012 - 15th International Conference on Medical Image Computing and Computer-Assis pages 50–57, Nice, France, Oct. 2012. Springer.

[90] T. Pfaff, M. Fortunato, A. Sanchez-Gonzalez, and P. W. Battaglia. Learning mesh-based simulation with graph networks. arXiv preprint arXiv:2010.03409, 2020.

[91] M. Pfeiffer, C. Riediger, J. Weitz, and S. Speidel. Learning soft tissue behavior of organs for surgical navigation with convolutional neural networks. International journal of computer assisted radiology and surgery, 14(7):1147–1155, 2019.

[92] S. Pfrommer, M. Halm, and M. Posa. Contactnets: Learning discontinuous contact dynamics with smooth, implicit representations, 2020.

[93] R. Plantefeve, I. Peterlik, N. Haouchine, and S. Cotin. Patient-specific biomechanical modeling for guidance during minimally-invasive hepatic surgery. Annals of biomedical engineering, 44:139–153, 2016.

[94] M. Raissi, P. Perdikaris, and G. E. Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. Journal of Computational Physics, 378:686–707, 2019.

[95] M. Raissi, A. Yazdani, and G. E. Karniadakis. Hidden fluid mechanics: Learning velocity and pressure fields from flow visualizations. Science, 367(6481):1026–1030, 2020.

[96] H. Reich, F. McGlynn, J. DeCaprio, and R. Budin. Laparoscopic excision of benign liver lesions. Obstetrics and gynecology, 78(5 Pt 2):956–958, 1991.

[97] F. Remondino and S. El-Hakim. Image-based 3d modelling: a review. The photogrammetric record, 21(115):269–291, 2006.

[98] R. Rico-Martinez, J. Anderson, and I. Kevrekidis. Continuous-time nonlinear signal processing: a neural network based approach for gray box identification. In Proceedings of IEEE Workshop on Neural Networks for Signal Processing, pages 596–605. IEEE, 1994.

[99] E. Roan and K. Vemaganti. The nonlinear material properties of liver tissue determined from no-slip uniaxial compression experiments. 2007.

[100] L. V. Romaguera, R. Plantefève, F. P. Romero, F. Hébert, J.-F. Carrier, and S. Kadoury. Prediction of in-plane organ deformation during free-breathing radiotherapy via discriminative spatial transformer networks. Medical image analysis, 64:101754, 2020.

[101] O. Ronneberger, P. Fischer, and T. Brox. U-net: Convolutional networks for biomedical image segmentation. In N. Navab, J. Hornegger, W. M. Wells, and A. F. Frangi, editors, Medical Image Computing and Computer-Assisted Intervention - MICCAI 2015, pages 234–241, Cham, 2015. Springer International Publishing.

[102] D. C. Rucker, Y. Wu, L. W. Clements, J. E. Ondrake, T. S. Pheiffer, A. L. Simpson, W. R. Jarnagin, and M. I. Miga. A mechanics-based nonrigid registration method for liver surgery using sparse intraoperative data. IEEE Transactions on Medical Imaging, 33(1):147–158, 2014.

[103] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[104] Y. Salehi and D. Giannacopoulos. Physgnn: A physics–driven graph neural network based model for predicting soft tissue deformation in image–guided neurosurgery. Advances in Neural Information Processing Systems, 35:37282–37296, 2022.

[105] I. Santesteban, M. A. Otaduy, and D. Casas. Learning-based animation of clothing for virtual try-on. In Computer Graphics Forum, volume 38, pages 355–366. Wiley Online Library, 2019.

[106] N. Schulmann, M. Soltani-Sarvestani, M. De Landro, S. Korganbayev, S. Cotin, and P. Saccomandi. Model-based thermometry for laser ablation procedure using kalman filters and sparse temperature measurements. IEEE Transactions on Biomedical Engineering, 69(9):2839–2849, 2022.

[107] H. A. Schwarz. Au-dessus d'une frontière "u transition par procédure alternée. Z "u rcher et Furrer, 1870.

[108] R. Shekari Beidokhti and A. Malek. Solving initial-boundary value problems for systems of partial differential equations using neural networks and optimization techniques. Journal of the Franklin Institute, 346(9):898–913, 2009.

[109] H. Sheng and C. Yang. PFNN: A penalty-free neural network method for solving a class of second-order boundary-value problems on complex geometries. Journal of Computational Physics, 428:110085, mar 2021.

[110] J. Sirignano and K. Spiliopoulos. Dgm: A deep learning algorithm for solving partial differential equations. Journal of computational physics, 375:1339–1364, 2018.

[111] K. E. Smiley, F. Wuraola, B. O. Mojibola, A. Aderounmu, R. R. Price, and A. O. Adisa. An outcomes-focused analysis of laparoscopic and open surgery in a nigerian hospital. JSLS: Journal of the Society of Laparoscopic & Robotic Surgeons, 27(1), 2023.

[112] J. D. Stitzel, S. M. Duma, J. M. Cormier, I. P. Herring, et al. A nonlinear finite element model of the eye with experimental validation for the prediction of globe rupture. In Sae conference proceedings p, pages 81–102. SAE; 1999, 2002.

[113] A. M. Stuart. Inverse problems: a bayesian perspective. Acta numerica, 19:451–559, 2010.

[114] L. Torrey and J. Shavlik. Transfer learning. In Handbook of research on machine learning applications and trends: algorithms, methods, and techniques, pages 242–264. IGI global, 2010.

[115] H. Wang, J. F. O'Brien, and R. Ramamoorthi. Data-driven elastic models for cloth: modeling and measurement. ACM transactions on graphics (TOG), 30(4):1–12, 2011.

[116] Y. Wang, N. J. Weidner, M. A. Baxter, Y. Hwang, D. M. Kaufman, and S. Sueda. RedMax: Efficient & flexible approach for articulated dynamics. ACM Trans. Graph., 38(4), July 2019.

[117] N. Winovich, K. Ramani, and G. Lin. Convpde-uq: Convolutional neural networks with quantified uncertainty for heterogeneous elliptic partial differential equations on varied domains. Journal of Computational Physics, 394:263–279, 2019.

[118] P. Wriggers. Nonlinear Finite Element Methods, volume 4. 01 2008.

[119] J.-L. Wu, K. Kashinath, A. Albert, D. Chirila, Prabhat, and H. Xiao. Enforcing statistical constraints in generative adversarial networks for modeling chaotic dynamical systems. Journal of Computational Physics, 406:109209, apr 2020.

[120] W. Xu, N. Umetani, Q. Chao, J. Mao, X. Jin, and X. Tong. Sensitivity-optimized rigging for example-based real-time clothing synthesis. ACM Trans. Graph., 33(4):107–1, 2014.

[121] L. Yang, X. Meng, and G. E. Karniadakis. B-pinns: Bayesian physics-informed neural networks for forward and inverse pde problems with noisy data. Journal of Computational Physics, 425:109913, 2021.

[122] Y. Yang, M. Aziz Bhouri, and P. Perdikaris. Bayesian differential programming for robust systems identification under uncertainty. Proceedings of the Royal Society A, 476(2243):20200290, 2020.

[123] W.-C. Yeh, P.-C. Li, Y.-M. Jeng, H.-C. Hsu, P.-L. Kuo, M.-L. Li, P.-M. Yang, and P. H. Lee. Elastic modulus measurements of human liver and correlation with pathology. Ultrasound in medicine & biology, 28(4):467–474, 2002.

[124] Z. Zeng, S. Cotin, and H. Courtecuisse. Real-Time FE Simulation for Large-Scale Problems Using Precondition-Based Contact Resolution and Isolated DOFs Constraints. Computer Graphics Forum, 41(6):418–434, June 2022.

[125] Q.-J. Zhang, K. C. Gupta, and V. K. Devabhaktuni. Artificial neural networks for rf and microwave design-from theory to practice. IEEE transactions on microwave theory and techniques, 51(4):1339–1350, 2003.

[126] Y. Zhu, N. Zabaras, P.-S. Koutsourelakis, and P. Perdikaris. Physics-constrained deep learning for high-dimensional surrogate modeling and uncertainty quantification without labeled data. Journal of Computational Physics, 394:56–81, oct 2019.

[127] O. C. Zienkiewicz, R. L. Taylor, P. Nithiarasu, and J. Zhu. The finite element method, volume 3. McGraw-hill London, 1977.

LIST OF FIGURES

1.1 Pie-charts displaying worldwide newly diagnosed (left) and deaths (right) by cancer type in 2020 based on World Health Organization data. Note that there are no necessary intersections in populations between figures since people appearing in the second figure could have been diagnosed multiple years prior. . 2

1.2 Illustration of a laparoscopic liver biopsy [21]. 3

2.1 Reference surface elements. Quadratic elements have more integration points (black dots) than linear, thus, generating bigger systems to solve. 15

2.2 The domain Ω^0 is deformed by the external forces (green arrows) applied to its surface. The deformation gradient tensor $\mathbf{F}$ give the transformation from Ω^0 to Ω. 19

3.1 Schematic of the Perceptron 32

3.2 Schematic of a neuron with an activation function Φ 35

3.3 Schematic of an MLP with three hidden layers, each with five neurons. There are no restrictions on the number of neurons per layer. 37

3.4 Schematic of a famous CNN known as U-net due to its shape 38

3.5 Schematic of a 3-way split of the dataflow. A merge operation is computed in the network to share the knowledge extracted by each stream. 45

3.6 Schematic of a shortcut architecture. A merge operation is computed in the network to better propagate data and gradient potential through the network. 45

3.7 Schematic of a general training process. 46

4.1 A cantilever beam Neo-Hookean material attached by the left face (a) and its five first modes shape (b,c,d,e,f). The complexity of the deformation increases with the index of the mode. 58

4.2 Examples of forces we can generate using only the first five modes shape of the cantilever beam presented in Figure 4.1. Here the $\mathbf{q}_i$ are presented as one-hot vectors, but any linear combination of the $\mathbf{q}_i$ can be used to generate a force. We can see the similarities between the forces computed using the vector $\mathbf{q}_i$ and the i-th mode shape in Figure 4.1. 59

4.3 Example of mask application to create sparser forces. Local forces are usually more representative of object manipulation if we do not consider gravity. . . . 60

4.4 Chosen network architecture with three steps and 128 channels in the first layer for an input grid with $28 \times 12 \times 12$ nodes Zero-padding is added in each spatial direction to avoid any loss of information on edge nodes, which leads to a $32 \times 16 \times 16$ shaped tensor At each step of the encoding phase, two padded $3 \times 3 \times 3$ convolutions are applied, followed by a ReLu activation function and a $2 \times 2 \times 2$ max pooling operation (which halves the spatial dimension) Each feature map doubles the number of channels In the decoding phase, upsampling $2 \times 2 \times 2$ transposed convolutions are applied, followed by two padded $3 \times 3 \times 3$ convolutions (doubles spatial dimension and halves the number of channels) The output of each layer is concatenated along matching levels from the encoding to the decoding path. 62

4.5 The effect of node ordering on prediction accuracy. For clarity, a beam with only 12 degrees of freedom is represented here. 63

4.6 Projected view of the liver geometry and its 3,309 H8 mesh. The nodes in the middle of the trunk of the vascular tree (here in blue) are fixed. 66

5.1 Examples of large nonlinear elastic deformations predicted by our neural network. The colors represent the node-wise Euclidean distance to the solution of the Newton-Raphson algorithm. For both beams, the color gradient goes from 3×10^{-4}m (blue) to 3×10^{-2}m (red), and for both propellers, the color gradient goes from 3×10^{-5}m (blue) to 2×10^{-3}m (red). 75

6.1 example of incisions location and size in laparoscopic surgery (left) compared to two examples of laparotomy (middle and right) performed in open surgery. Incisions vary depending on the surgery and patient-wise characteristics. . . . 81

6.2 83

6.3 example of a mesh with a randomly sampled zone of interest. The mesh appears grey, while the zone of interest is orange, and the randomly selected points appear as white dots. 84

6.4 Problem setup. The body at rest Ω^0 undergoes a deformation u becoming the deformed body Ω . The distance from a point of the observed surface y to its orthogonal projection on the surface of the body $\partial\Omega$ is noted $d(y,\partial\Omega)$. (Image from Mestdagh et al.[78]) 85

6.5 Schematic of the algorithm proposed by Mestdagh et al. [78] where $J(\mathbf{u})$ correspond to the function evaluated in equation 6.4. 89

6.6 Schematic of the algorithm with our solution to remove the first bottleneck. The simulation step has been replaced with an ANN that does the same job. . 90

6.7 Schematic of the algorithm with our solution to remove the second bottleneck. The simulation step has been replaced with an ANN that does the same job, and the backpropagation algorithm has replaced the adjoint system. 92

6.8 Beam used in this section (blue) attached to the grey wall which represents Dirichlet's boundary conditions 92

6.9 Deformations from the test dataset. The red dots represent the target point clouds, and the color map represents the Von Mises stress error of the neural network prediction. 93

6.10 Mesh of the liver used in this section. Composed of 3,046 vertices and 10,703 tetrahedral elements, which represents a challenge compared to the one used in 6.9 94

6.11 Average target registration error and computation times of each sequence. . . 95

6.12 Synthetic liver deformations and force distributions (left), reconstructed deformations and forces using the Newton method (middle) and the network (right) for test case 3. 96

6.13 On the left-hand side, the original simulation that produced the point cloud used in the registration of the right-hand-side picture. A registration is performed on the right-hand side to match the point cloud. The support in blue is larger than the original zone where forces are applied, yet, the optimization gives forces similar to the original. 97

6.14 Force estimation error of the five sequences using our method, in red the average force reconstruction error with the classical method. 97

6.15 Two examples of noisy rigid registrations used in this study with in yellow the reference position, in black the noisy rigid registration. The left and right-hand side is generated using an STD of 10^{-4} and 10^{-3} respectively. 99

6.16 Top : Graph of the target registration error as a function of the STD of the gaussian noise. Bottom : Graph of the error on the force reconstruction as a function of the standard deviation of the gaussian noise. Dark blue represent the mean of the samples, while light blue represent the standard deviation of the measured quantity. The red line represent the value computed on the noise-free ground truth. 100

7.1 Difference in computation, figure a) computes the global energy while figure b) computes the force for a single point of the mesh. Force computation requires to use scatter operator, which is not vectorizable; hence, slower than simply computing the energy. 106
7.2 Example of a sliced array in Python. Array $sliced\_A$ contains the fifth, first, and second value of array A. 108
7.3 Example of two topological representations in position space (Pos_quad and Pos_tri). Indexing the array of positions by the topology creates a topological representation in the position space of the object. We create an array in which the first dimension is the number of elements in the topology, the second is the number of vertex per element, and the third is the number of space dimensions. 109
7.4 Example of the utilization of the enum function. This description of the computation allows for parallel vectorized execution of the operations. 109
7.5 The three compared product formulations written using the PyTorch framework. 110
7.6 Computation of the deformation gradient using PyTorch. 111
7.7 Computation of the right Cauchy-Green strain tensor using PyTorch for linear and nonlinear case. 111
7.8 Computation of the right Green-Lagrange deformation tensor using PyTorch. . 111
7.9 Computation of the element-wise incomplete elastic potential energy using PyTorch. 112
7.10 Computation of the element-wise elastic potential energy tensor using PyTorch. 112
7.11 Computation of the total elastic potential energy tensor using PyTorch. 112
7.12 Computation of the linear potential energy tensor using PyTorch. 112
7.13 Computation of the Dirichlet boundary conditions tensor using PyTorch. . . . 113
7.14 Presentation of the SystemManager workflow. 114
7.15 Presentation of the differentiable physics engine workflow. 115
7.16 Differentiation of a scalar function (*function_to_minimize*) with respect to the input variable x. The flags *create_graph* and *retain_graph* are set to $True$, meaning that we will be able to differentiate *grad_f* another time to obtain the tangent stiffness matrix. 116

7.17 We use vmap to vectorize the computation of the matrix $\mathbf{K}(\mathbf{u})$. This greatly improves performances compared to the naive for-loop method. 116

7.18 Schematic of the algorithm proposed to optimize any simulation parameter c . The neural network of Figure 6.7 has been changed to the presented differentiable solver. Which computes slower but precise deformations according to the finite element framework. 118

7.19 On the left-hand side, an example of a deformation computed on a dragon made of tetrahedra. On the right-hand side, the beam used in this section (orange) is attached to the grey wall, representing Dirichlet's boundary conditions. The surface mesh comprises triangles, but the physic is computed on hexahedra. 120

7.20 Sample of the deformations present in the test dataset. The deformation is heteroclite with bending and twisting from all ranges. 121

7.21 Results of the computation of 1,000 deformations using a square section beam filled with NeoHookean material. As expected, the mean $l2$ is in the range of the threshold. 122

7.22 Reconstructed forces and resulting deformations. The target point cloud appears in black. 124

7.23 Example of three force optimization (each row). The left column consists of the groud-truth. The middle column consists of examples where the forces are initialized with Gaussian noise with parameters $(0, 10^{-2})$. The right-hand-side column consists of examples where the forces are initialized with the opposite of the ground truth. 125

7.24 Example of three force optimization (each row). The top row consists of examples where the support is the opposite of the original. The bottom row consists of examples where the support is at the tip of the beam. 126

7.25 Example of four Young's modulus optimization. The TRE is minimal, which represents an excellent fit. Young's modulus is also extremely close to the actual solution. 128

7.26 Close-up on the tip of the beam. The ground truth with Young's modulus of 4,500 appears in red. The deviation is minimal for an error of 1%, showing that the displacement field is relatively similar for small variations of Young's modulus. 129

7.27 The projection P is used to keep the Poisson's ratio in a reasonable value considering our problems. This function is a parametrization of the logistic function. Therefore, it can be easily modified to project in any desired space. The additive constant value defines the minimum value of the space, while the numerator is computed by subtracting the extremes of the space. 130
7.28 Example of four Poisson's ratio optimization. The TRE is minimal, which represents an excellent fit. The Poisson's ratio is also extremely close to the actual solution. 131
7.29 Close-up on the tip of the beam. The ground truth with Young's modulus of 4,500 appears in red. The deviation is important for an error of 1%, showing that small variations of Poisson's ratio highly influence the displacement field. 132

LIST OF TABLES

4.1 Two MLP trained for 100 epochs with random or modal force amplitudes. u_2 gives the distribution of the L2 norm of the displacement. 64
4.2 Result of the MLP and U-Mesh on different dataset. 65
4.3 Performance of the MLP and of the U-Mesh trained over a dataset generated with random forces on the liver's surface. Both networks are trained for 100 epochs over 20,480 samples and tested on 100 samples drawn from the same distribution. PT stands for prediction times on a GeForce RTX 3090. u_2 gives the distribution of the L2 norm of the displacement. 66

5.1 Results of a comparison between FEM simulations and ANN predictions over 100 randomly distributed forces with random amplitudes. 75
5.2 Error values and SNR of the deformations shown at Figure 5.1. The deformations are highly nonlinear, yet the error values and SNR remain in the range of values displayed in table 5.1. 76
5.3 Results of comparing the classic Newton-Raphson algorithm and the presented Hybrid Newton-Raphson algorithm over 100 randomly distributed forces with random amplitudes. 76

6.1 This table gathers statistics for 10,000 test cases and presents registration errors, number of iterations, and computation times (in ms). 93