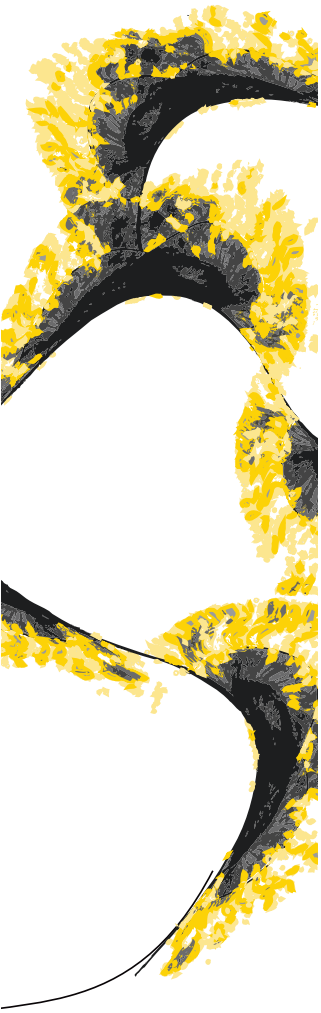


RAM ROBOTICS AND MECHATRONICS

A GENERAL OPTIMIZATION FRAMEWORK FOR SOFT ROBOTIC ACTUATORS WITH ANALYTICAL GRADIENTS

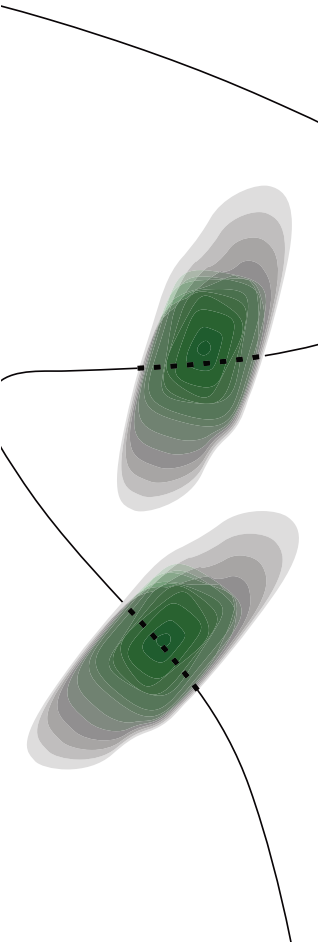
D. (Daniel) Wilmes

MSC ASSIGNMENT

Committee:
dr. ir. M. Abayazid
Y.X. Mak, MSc
dr. C.A. Pérez Arancibia

January, 2022

001RaM2022
Robotics and Mechatronics
EEMathCS
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands



UNIVERSITY OF TWENTE. | TECHMED CENTRE UNIVERSITY OF TWENTE. | DIGITAL SOCIETY INSTITUTE

Abstract

The following thesis ***A general optimization framework for pressure-driven soft robotic actuators with analytical gradients*** is aimed at developing a system that allows defining an arbitrary loss functions in a soft actuators simulation, and optimize towards any parameter in the simulation using analytical gradients. This is motivated by inherent difficulties in modelling and development of both soft robots and their controllers due to the non-linear and time variant properties of the forces acting on them. Furthermore, solutions from comparable literature tend to lack generality and restrict themselves to certain use-cases or offer methods to work around the issues but are more time consuming in return.

A literature study is first conducted related to soft body simulation and optimization procedures to find valid approaches in modelling soft robotic actuators and perform auto-differentiation. After an approach is chosen and implemented, a verification of the simulation is performed by comparing multiple simulations to an established FEM solver. Afterwards the issue of exploding gradients for auto-differentiation is adressed by analysing the mathematical background of the problem and proposing a solution. To demonstrate the use of the developed 3D optimization framework for controller synthesis, a small neural network feed-forward controller is set-up for a pneumatic endoscope actuator model with three pressure chambers and trained using the derived gradients from the simulation. Furthermore, a meta-optimization scheme is presented, where the damping factor of the simulation is split into 50ms time-windows and optimized with the intention of shortening the time until the deformation of the actuator reaches its final state.

The developed system is shown to be able to derive meaningful gradients that can be used to optimize different components of the simulation. The proposed scaling scheme to avoid exploding gradients requires the user to fine-tune a few parameters to get optimal results. The scheme has been shown to produce useful gradients for an exemplary pressure optimization and controller synthesis. In comparison to reinforcement learning, the controller synthesis requires about 1 order of magnitude less iterations steps to converge in addition to more smooth and reduced loss fluctuation over the course of training. The meta-optimization managed to reduce the required number of time-steps by approximately 25% with some caveats. The loss function showed strong signs of being ill-defined but the optimization still succeeded based on the gradients and changes in the damping factors, which implies that more complex and well-behaved formulations have the potential to give better results.

Contents

List of Abbreviations

FEM	Finite Element Method
SOFA	Simulation Open Framework Architecture
GPU	Graphics Processing Unit
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
CAD	Computer Assisted Design
NADAM	Nesterov-Accelerated Adaptive Moment Estimation
ID	Identifier
RaM	Robotics & Mechatronics

List of Symbols

Parameter	Unit	Description
m	kg	Mass
x	m	Position
v	$\mathrm{m\,s^{-1}}$	Velocity
F	N	Force
k	$\mathrm{N\,m^{-1}}$	Spring stiffness
V	$\mathrm{m^3}$	Volume
E	$\mathrm{N\,m^{-2}}$	Gradient of stress-strain function (generalized young's modulus)
l	m	Edge length between two mesh nodes
λ	-	Stretch
B	-	Strain displacement matrix
σ	$\mathrm{N\,m^{-2}}$	Cauchy stress tensor
D	-	Matrix describing the shape of a given tetrahedral element
F	-	Deformation gradient
J	-	Volume ratio
I	-	Invariants of deformation gradient
W	$\mathrm{J\,m^{-3}}$	Strain-energy density
A	$\mathrm{m^2}$	Area
p	$\mathrm{N\,m^{-2}}$	Pressure
n	-	Normal of a given surface
α	-	Volume force scaling factor
β	-	Mass damping scaling factor
C	-	Hyper-elastic material parameter
D	-	Hyper-elastic material volume parameter
e	-	Error
d	m	Displacement
s	-	Gradient scaling factor
z	-	Order of magnitude of a given variable
g	-	Gradient of a given variable for a given function
L	-	Loss function

1 Introduction

1.1 Context

Soft robotic actuators are an emerging technology that are increasingly used in medical [1] [2] [3] and manufacturing contexts [4] [5]. Advantages of using soft robots are the inherent flexibility and adaptability of soft robotics. Particular applications in medical contexts include minimally invasive surgery [2] [3] [6], (semi-) automatic drug delivery [7], diagnosis and biopsy [8] and assistance, rehabilitation and exoskeletal support of practically any body part [9] [10] [11] [12] [13]. Furthermore, soft robotics are considered for support or simulation of mechanical organ parts like the heart [14] or liver [15] because the necessary motion can be replicated by soft robotic actuators.

In regards to manufacturing soft robotics are interesting due to their adaptability and freedom of motion. This potentially allows for a high degree of shape invariance for e.g., grippers [16]. This makes these types of actuators desirable for automation and construction [17].

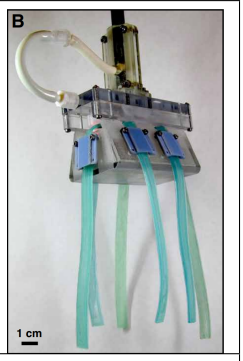
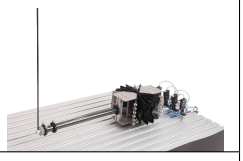
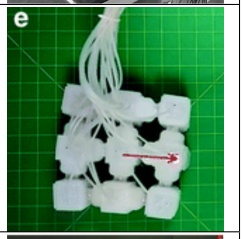
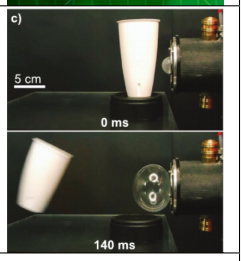
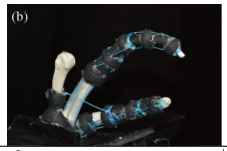
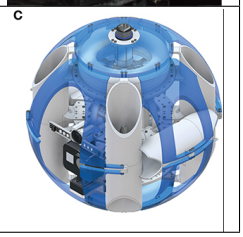
1.2 Problem Statement

Even though soft robotic actuators have a variety of potential areas there are some issues with their development. In particular, because of the complicated underlying mechanics that describe the behaviour of pneumatic or hydraulic soft robotic actuators it is often difficult for humans to intuitively understand the connection between the fluid-cavity shape and motion. This causes the resulting designs to be sub-optimal both in terms of forces and motion [18]. Two problems emerge from this. First, the performance of the robot may suffer because the designer may not know how to optimally shape the robot for the intended motion, and second, more complicated motions require significant development time and trial-and-error approaches despite the freedom that soft robotic actuators theoretically offer in this regard [19]. This issue also extends into the development of controllers because they require the developer to model the complicated dynamics, especially if interaction with an environment is considered, or employ time consuming reinforcement learning methods with an existing robot and simulation. The core issue can therefore be described as the difficulty with the design of soft robotic actuators and their respective controllers. In fact, the development and design part of even simpler actuators are often published in addition or independently of use cases [20] [21] [22] [23]. Furthermore, the designs tend to be arguably simple, partially rigid or bio-mimetic. Table 1.1 summarizes related papers and their approach to modelling.

In some cases this issue can be alleviated through the use of numerical optimization. Still, in comparison to the relevancy and size of soft robotics as a current field of research, optimization is arguably rarely addressed. This is partially due to the novelty of the field but also because performing optimization on soft bodies carries additional problems compared to static structures. Specifically, modelling hyperelastic structures requires more complex material descriptions and large deformations need to be addressed in the discrete model through higher mesh resolution or non-linear finite elements.

In addition, a major problem in modelling is caused by the change of applied forces for pressure-driven actuators since the surfaces where the pressure applies changes over the course of actuation. This makes the common form of linear FEM based on system of equations not applicable as the right hand side containing the forces is time variant. While many of these problems are individually theoretically solved or solvable in some contexts like time variance [32] and hyperelasticity and deformation modelling [33] [34], finding holistic and efficient solutions to these issues is not trivial and FEM programs employ a variety of different strategies to address them [35].

Table 1.1: Different approaches to modelling soft robots from the literature

Author and year	**Technique and Application**	**Limitation**	**Image**
Sinatra et al. 2019 [24]	Nano-fiber-reinforced pressure-driven soft actuators; Gently grabbing marine life	Simple shape and rigid base	
Rateni et al. 2015 [25]	pressure-driven soft actuator; Minimally invasive surgery; Scalable to other use cases	Simple, unoptimized shape	
Hofer et al. 2018 [26]	Inflatable bladders	Simple shape and mostly rigid	
Margheri et al. 2012 [27]	Pressure-driven soft actuator; Aimed at achieving same dexterity as octopus	Bio-mimetic from octopus	
Zou et al. 2018 [28]	Pressure-driven modular soft actuators; Exploring unstructured environments	Bio-mimetic from caterpillar	
Baumgartner et al. 2020 [29]	Pressure-driven elastomer exploiting mechanical instability; safe, high-speed soft grippers	Bio-mimetic from Venus flytrap	
Culha et al. 2016 [30]	Correct bones, elastic ligaments and antagonistic tendons; Large motion range and strong grippers	Bio-mimetic from human finger	
Plum et al. 2020 [31]	Soft shell to minimize damage to the robot with rigid actuation; complex repair, exploration and analysis under water	Bio-inspired outer shell	

Furthermore, an issue that's wholly unaddressed as far as the author is aware is the computational efficiency of these procedures. This is an issue as optimization is inherently a computationally heavy process and the issues that are added when performing it on soft robotic actuators accentuate this problem many times fold. This is intuitive because if every discrete element is more complex and the forces are time variant then every iteration of an optimization requires several times more operations. This is a provably relevant issue since computational optimization of static structure topology optimizations is still not considered a solved problem despite decades of research [36] [37].

1.3 Review of related works

Only a minority of soft robotics enjoy some form of mathematical optimization either through a higher-level model of their behaviour [38] [39] or specifically developed topology optimization strategies [40] [41] [42].

1.3.1 Approaches to optimizations

The approach of [38] consists of an analytical model describing the developed pneumatic actuator shapes as a cantilever with spring-like properties and two different optimization methods, namely a swarm intelligence optimization called Firefly-algorithm [43] and a reinforcement learning agent [44]. These ideas were verified with a FEM simulation applying the same optimizations and later validated by 3D printing the actuators and testing them. The results show that both the analytical model optimization as well as the FEM optimized shape outperform the empirically designed one. In this case, the measure of performance was the decoupled motions in the vertical and horizontal directions upon applying pressure. The advantage of the analytical model was the drastically reduced number of iterations necessary with the Firefly-algorithm. It required about 10 iterations while the FEM model took about 400 iterations. That being said, such analytical higher-level models are not generated but designed and require the researcher to model the system using simpler and understood components like springs and dampers. This is an issue for development as not every shape can easily be simplified in this manner.

The strategies employed by [39] and [42] are more generally applicable and intend to automate the design process of soft robotics by formulating FEM-based topology optimization problems and solving them. In particular, [39] uses an FEM approach that is solved through the method of moving asymptotes [45] to solve the issue of ever changing applied forces as pressure causes deformation and additionally speed up and stabilize convergence due to the computationally expensive nature of the problem. The authors in [42] develop a general framework for topology optimization for soft robots and solves the mentioned issue of changing forces by using the polygonal surfaces to simulate the forces while the deformation occurs. It also addresses the issue of material hyperelasticity in soft robotics by treating the tetrahedra as nonlinear finite elements to increase accuracy for larger deformations. The framework restricts itself to the 2D case and thus doesn't allow the development of more complex shapes but the results show that this approach may significantly decrease development time and generates previously unknown and highly efficient shapes. The optimization is based on material density by defining a finite 2D domain that is split into many small areas with varying density. The program then changes the density of each region to minimize the error function. A simple bending soft actuator from [42] is shown in Figure 1.1 and shows how even a 2D optimization can produce shapes that a human would arguably not be able to think of. Furthermore, it uses a combination of auto-differentiation and an analytical expression to acquire analytical gradients to improve both performance and quality of convergence. Theoretically this approach could be extended to allow more arbitrary optimization for e.g. gradient based training of neural

network controllers. Another point that is explicitly mentioned in these papers is the ability to define the optimization goal in a somewhat arbitrary manner [39]. The beauty of this is that the shapes can be optimized for a variety of objectives as long as a loss function can be found. Figure 1.1 for example maximised the mechanical advantage, meaning the ratio between output and input force. Consequently it is possible to define goals like following a trajectory or minimizing the directional coupling like in [38].

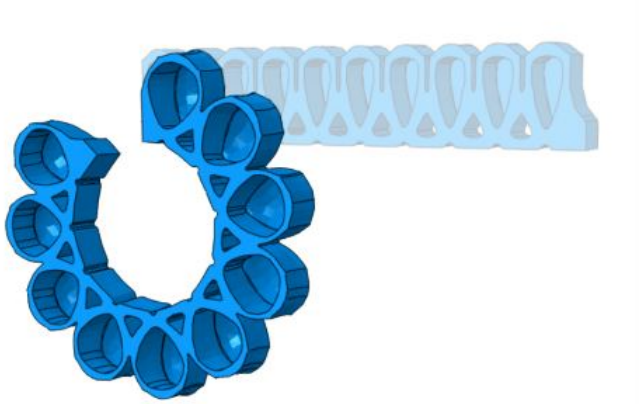


Figure 1.1: Finite element analysis of the developed soft actuator though the proposed topology optimization framework of [42]. Notice the unintuitive teardrop cavities and connections between them which maximise the mechanical advantage and produce a circle.

1.3.2 Soft actuator simulation

While not intended for optimization there exists a framework for real-time soft robotic simulation for iterative shape and controller development called soft-robotics toolkit for Simulation Open Framework Architecture (SOFA) [46] [47]. It implements a large variety of approaches and 30 different forces for simulation at once and is highly optimized. It is based on numerical integration of nodes or surfaces in a mesh, but the approach to the computation of internal forces varies depending on the user's choice. In particular, there are two main ways to compute the internal forces in the model. The first is based on the node edges as springs where the mass, spring stiffness, and (optionally) damping need to be defined. The second method [48] computes the forces from the deformation of tetrahedral finite elements. The forces on each node of the model can be computed as a fraction of the surface forces of all incident faces similar to [49]. Furthermore, SOFA also managed to implement many of its functions partially or entirely on GPU and achieved 212 frames per second on a mesh with 45k tetrahedral elements on a Nvidia GeForce GTX 480 and about 12 frames per second on an Intel Core i7 975 3.33GHz CPU with the same parameters [46].

1.3.3 Gradient based optimization

Regarding general gradient based optimization problems in physical simulations, there have been ideas and implementations to improve both convergence and computational load by calculating analytical derivatives of the cost function towards the simulation parameters to optimize for using differentiable simulations [50] [51] [52]. This is relevant because a numerical physical simulation can be used to alleviate the problem of changing forces at every time step. Furthermore, assuming a framework for auto-differentiation can be found the process of implementation is expected to be quite simple. However, an open problem that is expected to appear in this use case is exploding and vanishing gradients due to large time horizons,

especially considering the non-linearities of the simulation.

1.3.4 Controllers

Another issue in soft robot development is in regards to precise controllers. The two main approaches used to tackle the problem of controlling soft actuators revolve around designing simplified analytical models based on the specific motions of the use case, model-free using neural networks or a hybrid of the two [53]. Each of these has quite severe drawbacks either in terms of quality or development time. In particular, analytical models of soft actuators often suffer from inherent inaccuracies due to being unable to describe the robot as a continuum and simplifying the kinematics. This necessitates additional adaptive abilities to counteract model uncertainties [54]. On the other hand, reinforcement learning approaches can theoretically produce controllers with arguably perfect precision [55] [56]. This is especially true when considering that it is possible to account for any type of non-linearity by including said non-linearities in the neural network of the agent. The downside of reinforcement learning is that it is essentially an automated trial-and-error process where the controller attempts all kinds of inputs and eventually finds out how to achieve the goal. Consequently, the process is quite slow and requires some form of learning environment, which ideally should be the physical robot to be controlled. There have been attempts at work-arounds using a neural network that is supposed to replicate the physical behaviour of a given soft robot by measuring its deformation with a camera and training the network on it. Because the network is differentiable, this effectively gives an approximate differentiable physical simulation of the dynamics. Hence reinforcement learning can be omitted for gradient-learning, which is faster due to adapting the weights directly towards the correct direction [57]. Still, learning based approaches remain time consuming even under with this approach as the robot needs to be build and the model trained before the controller training can even begin.

1.4 Goals and approach

With the main problems stated in Section 1.2 and relevant literature reviewed in Section 1.3 the overarching research question can be defined as

"How can a general optimization framework with analytical gradients for pressure-driven soft robotic actuators be modelled and implemented?"

This main question will be supported by a number of sub-research questions with the purpose of evaluating the developed framework:

- How accurate is the developed simulation compared to established FEM tools in regards to the predicted deformation?
- How can the issue of exploding/vanishing gradients from large time horizons for auto-differentiation be addressed?
- What are the effects of gradient-based training on the speed and reliability of convergence for a soft-robotic controller for an endoscope model?
- What are the effects of performance-targeted meta-optimization of the simulation? Meaning how can the optimization be applied to the speed of the simulation itself?

The approach to answering these questions can be outlined by the following steps. First, a solution to solving the modelling difficulties needs to be found. This is done through the previous literature study to find ways to accurately simulate the dynamics of hyper-elastic materials

and pressure forces. FEM simulations will be used to verify the chosen approach and answer the first research question. Once the simulation is confirmed to give accurate results a robust way to compute analytical gradients through auto-differentiation for optimization needs to be found. With this done it's possible to approach the remaining research questions by first addressing the exploding gradients and later on controller synthesis and meta-optimization.

The second research question about the exploding gradients in non-linear large-time-horizon numerical integration is answered by dedicating a chapter to explaining the mathematical background, how the issue can be approached, the proposed solution and an exemplary optimization. The endoscope controller will be neural network based but with only the input and output layer to get a measure for the capabilities of the framework. Lastly, the meta-optimization will optimize the damping factor for each node for different time-windows to reduce the time it takes to reach the final deformation state.

Consequently, the main contributions of this work compared to previous studies consist of extending and generalizing existing methods to 3D and showing its potential in a number of tests. This is because comparable literature does not utilize auto-differentiation for the entire simulation and hence does not have the same level of generality. Second, the issue of exploding gradients for auto-differentiation of such a non-linear large-time-horizon simulation is an open problem and how it is addressed may be considered a novel contribution. Furthermore, the author is not aware of other works utilizing optimization frameworks for improving the behaviour of the simulation itself, which is referred to as meta-optimization in this document.

Because the research is based on software development, it is useful to define some requirements using the MoSCoW-method. A diagram showing the prioritization is shown in Figure 1.2. The main priorities revolve around the main purpose of the program so that the optimization can at the very least be performed. The "Should"-priorities are important for any type of serious application and freedom for the user but not strictly necessary for the proof of concept of the underlying methods. The "Could"-priorities are advanced functionalities either for convenience or more complex problems that are slightly beyond the scope of this project. Lastly, "Won't"-priorities mostly consist of things that take a lot of time to implement while not contributing much to the project goals.

1.5 Report Structure

Lastly, the chapters of this thesis can be summarized as follows. First this introduction chapter contained the motivation, goals and literature review to find ways to approach the problems. Chapter 2, modelling, explains in mathematical detail how the simulation was chosen to be modelled. Chapter 3, implementation, describes which software resources are used for implementation and the workflow of the implementation. With the theory done, Chapter 4, verification, describes the exact procedure to performing the verification and its results. Because the remaining three research questions are quite different in nature each one has its own dedicated chapter showing and interpreting the obtained results. Consequently, first the issue of exploding gradients for the simulation is addressed in Chapter 5. Afterwards controller synthesis for the endoscope model is performed and evaluated in Chapter 6 and lastly meta-optimization in Chapter 7. The final chapter, Chapter 8, presents the conclusions of the work and gives a number of recommendations for future research.

Figure 1.2: Software prioritization diagram outlining the top-level requirements of the program using the MoSCoW-method

2 Modelling

This chapter will describe and explain the mathematical theory needed to understand the simulation and optimization. As such, it will show how certain physical properties relate to the simulation parameters, how the material is modelled, how the forces are calculated and how the simulation propagates through time.
Before the details are explained, however, a short paragraph will outline the conceptual design decisions resulting from the literature review.

2.1 Conceptual design

First an approach to simulating soft actuators needs to be decided on. Numerical integration will be used to propagate through time, hence the main issues are the calculation of pressure and internal forces. The internal forces will be modeled in two different ways, which will be compared.

1. Damped vertex-edge-springs
2. 3D Element deformation and cauchy stresses

Damped-edge-springs are simpler to implement but are expected to be less accurate than calculating the cauchy stress tensor because they are mere 1D elements with no inherent consideration for surface and volume forces. In comparison, the cauchy tensor is a holistic description of the internal forces inside a finite element. The pressure force then needs to be calculated for each node, which is a solved problem [49] and only requires the pressure, surface areas and normals. The constraints necessary for the simulation are straightforward to implement by fixing the position of a number of individual nodes. The following sections will go into more detail for each of these points.

2.2 Integration scheme

The simulation is based on an explicit numerical solver and uses the Forward-Euler-method to propagate in time [58]. It therefore only requires the calculation of all forces for each time step and integrates them into velocity and position.

The main reasoning to use this simple scheme is the ease of implementation, low computational load and the rather high level of non-linearity and time-dependency of forces. These non-linearities make the usage of implicit solvers much more difficult since they assume the forces don't change between the current configuration and the force equilibrium configuration. Furthermore, explicit solvers are capable of simulating the dynamic behaviour of the object. This may be desirable as quickly inflating and deflating soft robotic actuators will, in reality, not immediately reach force equilibrium but fluctuate dynamically too. Lastly, despite requiring a higher number of integration steps, explicit solvers have an advantage in terms of computation which is the decoupling of all nodes. This has the effect that all velocities and positions of nodes can be integrated independently, which allows full parallelization on GPU [59].

The used explicit scheme is described by a second order differential equation given by:

$$m_n \ddot{x_n} + D_n \dot{x_n} + K_n x_n = F_n \,, \tag{2.1}$$

where m is the mass, D the damping matrix, K the stiffness matrix and F the external forces for vertex n. The vector x contains the position of the vertex and as such its time derivatives are the velocity and acceleration. This differential equation only needs to be solved for the acceleration and then be numerically integrated. There is no need to describe the entire system as a matrix differential equation as the explicit scheme decouples every node. The Forward-Euler-method

then collects the forces at each time step and numerically integrates them to find the values for the next step for each vertex individually:

$$\begin{aligned} v_n(t) &= v_n(t-\Delta t) + \frac{F_n(t-\Delta t)}{m_n} \cdot \Delta t, \\ x_n(t) &= x_n(t-\Delta t) + v_n(t) \cdot \Delta t, \end{aligned} \tag{2.2}$$

where Δt is the time step of the simulation and x, v and F are the position, velocity and total force associated to node n at time t and $t-\Delta t$, respectively [58].

The following sections in this chapter will explain how each component of Equation 2.2 is calculated.

2.3 Internal forces

As mentioned in Section 2.1 the first approach to the internal forces calculation is based on treating the edges of the tetrahedral model as damped springs. This invites the question how the spring stiffness and damping factors are calculated. This springmesh is simple but arguably a somewhat naive way to describe the internal forces because it assumes all appearing forces can be broken down into the behaviour of a 1D spring. This is why the second implemented approach to the internal forces is based on the deformation of the tetrahedral elements which allows a more hollistic (and thus accurate) description of the internal stresses. Both of these methods will be tested later on.

2.3.1 Springmesh

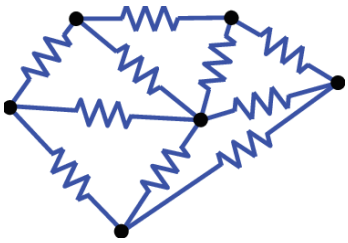


Figure 2.1: A exemplary spring mesh. Each edge is a damped spring applying force to both connected nodes. Taken from [60]

Springmeshes are commonly used for modelling soft tissues owing to ease of implementation [61]. Because the spring stiffness will determine the steady state position of the nodes due to the force balance between pressure and spring forces, it is crucially important to derive the spring stiffness from the material properties. Furthermore, this is the point where the hyperelasticity of the soft materials needs to be addressed. Both of these problems can be solved in the same breath by using the stress-strain curve of a given material [62]. The strain of each edge can be calculated through the theoretical area covered. Mathematically, this can be expressed as

$$k_i(\lambda, t) = E(\lambda)\frac{V_0}{l_0^2}, \tag{2.3}$$

where k_i is the spring stiffness for edge i, $E(\lambda)$ the gradient of the stress-strain curve (similar to Young's modulus but generalized to any curve) at stretch λ, V_0 the initial volume of all tetrahedra adjacent to the edge and l_0 the initial length [63]. This means that the spring stiffness is not a constant but depends on the gradient of the stress-strain curve and needs to be recalculated each time step. Note that this effectively puts the Poisson's ratio $\nu = 0$ but this issue can be worked around by adding a volume conservation force as explained in Section 2.5 and from certain assumptions of the material model as explained in the next paragraph.

The internal forces for each node can then be calculated by iterating through all edges and adding the reaction forces towards both nodes with equal magnitude and opposing signs.

$$f_{\text{internal},i} = \Delta l \cdot k_i \tag{2.4}$$

For the two nodes 1 and 2 of edge i with

$$\Delta l = \left\| x_{i,1} - x_{i,2} \right\| - \left\| x_{i,1,0} - x_{i,2,0} \right\|, \tag{2.5}$$

which is the difference in length between the current position and the non-deformed position. The total internal force can then be summed up over all edges that contain the node. Because of this it makes more sense to iterate over all edges and add the same force with opposing signs to both nodes for the implementation.

The only remaining question is how exactly the strain-energy-derivative $E(\epsilon)$ is calculated as all forms of hyper-elastic strain energy functions require the stretch in all three dimensions, while these spring stiffnesses merely represent 1D truss-elements. The simplest solution is to assume incompressible uniaxial tension which reduces the number of independent stretch-variables to one [64]. Since there are a number of different hyper-elastic strain-energy formulations the most general way to describe this is to say that under uniaxial strain the stretches λ_k with $k = 1,2,3$ succumb to the condition

$$\lambda_1 \lambda_2 \lambda_3 = 1 \tag{2.6}$$

for incompressibility which leads to

$$\lambda = \lambda_1, \quad \lambda_2 = \lambda_3 = \frac{1}{\sqrt{\lambda}} \tag{2.7}$$

This in turn can be used to derive the strain-energy and its derivative from any hyper-elastic model with just a single known stretch [64]. The problem is that this assumes incompressibility which is expected to cause errors because not all materials relevant for soft robotics fall into this category. In the above paragraph it was mentioned how the Poisson's ratio is effectively $\nu = 0$ because of the lack of volume forces but with this material assumption it becomes $\nu \approx 0.5$ and can be influenced using the mentioned volume forces in Section 2.5.

2.3.2 Element deformation

A much less assumption-struck but more complicated formulation of the internal forces is based on the deformation of the tetrahedral elements by calculating all internal stresses. Now instead of 1D-elements it's possible to consider 3 dimensions, which opens some options regarding compressibility and direction of strain. The base equation describing the forces on all 4 nodes of a tetrahedron can be described by the matrix equation:

$$f = B^T \sigma_{\text{voigt}} V, \tag{2.8}$$

where B^T is the transpose of the strain-displacement matrix, σ_{voigt} the Cauchy stress in Voigt-notation and V the volume of the element in question [65]. The force vector f has 12 entries describing the $x-$,$y-$ and $z-$force-components of the 4 nodes. Calculating the strain-displacement matrix [66] and the volume [67] is trivial and only requires the positions of the 4 nodes. Naturally, the forces on each node need to be summed over all elements containing that node.

Cauchy stress calculation from hyperelastic models

Calculating the stress in an element from a given hyperelastic material model requires the invariants of the deformation gradient matrix and applying these to the derivative of the chosen

stress-strain function [68]. The deformation gradient is defined by the transformation of the tetrahedral shape between the initial and current configuration [69]

$$\begin{aligned} F_{\text{def}} &= D_t D_0^{-1} \\ D_0 &= [x_{1,0} - x_{4,0}, x_{2,0} - x_{4,0}, x_{3,0} - x_{4,0}] \\ D_t &= [x_{1,t} - x_{4,t}, x_{2,t} - x_{4,t}, x_{3,t} - x_{4,t}] \end{aligned} \tag{2.9}$$

The first index of x refers to the 4 nodes associated to the given element and the second index the time, meaning D_0 is a constant matrix that can be reused at every time-step. From this a number of values need to be derived. The formulation of the stress will allow slight compressibility as defined in [68] since most hyperelastic materials, which are most often used for soft robots [70], fall into this category. The first value is

$$J = \|F_{\text{def}}\| \tag{2.10}$$

which is the volume ratio of the element, defined by the current volume over the initial volume [68]. Now because near incompressibility / slight compressibility is assumed, the deformation matrix and consequently the invariants need to be scaled using J to comply with the compressible formulation of the material models, because it decomposes the isochoric and volumetric part of the deformation [71]. This does not enforce volume conservation but is a method to ease computation due to the internal stresses growing extremely quickly when an incompressible material is put under pressure. This will cause problems during numerical integration because the motions per step become excessively large.

Using this method, it is possible to simulate nearly incompressible and compressible materials under the condition that volume forces are described by a decoupled volume energy term due to the decomposition of the forces.

$$\bar{F}_{\text{def}} = J^{-1/3} F_{\text{def}} \tag{2.11}$$

From here on, variables denoted with a bar refer to the adapted ones for slight compressibility.

$$\bar{B}_{\text{def}} = \bar{F}_{\text{def}} \bar{F}_{\text{def}}^T \tag{2.12}$$

$\bar{B}_{\text{def}}$ is the left Cauchy deformation tensor which can conveniently be used to calculate the needed invariants.

$$\begin{aligned} \bar{I}_1 &= \text{Tr}(B) \\ \bar{I}_2 &= \frac{1}{2}((\text{Tr}(B))^2 - \text{Tr}(B^2)) \end{aligned} \tag{2.13}$$

Now these invariants and the volume J can be used to find the Cauchy stress given a material model $W(\bar{I}_1, \bar{I}_2, J)$, which will be detailed in Section 2.8.

$$\sigma = \frac{2}{J}\frac{\partial W}{\partial \bar{I}_1}(\bar{B}_{\text{def}} - \frac{1}{3} I_1 1) + \frac{2}{J}\frac{\partial W}{\partial \bar{I}_2}(\bar{I}_1 \bar{B}_{\text{def}} - \bar{B}_{\text{def}}^2 + \frac{2}{3}\bar{I}_2) + \frac{\partial W}{\partial J} 1 \tag{2.14}$$

Equation 2.14 is intentionally split by the variables of $W(\bar{I}_1, \bar{I}_2, J)$ such that it becomes obvious which components drop out depending on the limitations of the material model. The straight 1 refers to the identity matrix of the surrounding dimension. Now the resulting stress tensor only needs to be written in Voigt notation and can be used to calculate the forces using Equation 2.8.

2.4 Pressure forces

Calculating the pressure force is more straightforward than the internal force. In this case, only the surface normals and areas need to be calculated. Because the nodes that make up the

corners of each triangle attain the force and not the surface itself, the force on each surface is divided by 3 for each node, effectively distributing the total force of each surface to each of its vertices.

$$\vec{F_{\mathrm{p},i}} = \frac{1}{3} \cdot p \sum A \cdot \vec{n}_{\mathrm{A}}, \tag{2.15}$$

where $\vec{F_{\mathrm{p},i}}$ is the pressure force at node i, p the pressure, A the area of a triangle the node is part of and $\vec{n}_{\mathrm{A}}$ the outward normal of the triangle so that the force pushes outside. The summation is performed over all pressure surface triangles the node is part of.

2.5 Volume conservation

In literature it appears to be uncommon to use a more complex expression than Equation 2.3 that involves the Poisson's ratio. This is because the authors in [63] showed that the resulting edge stiffness may be negative if the shape is too obtuse, thus making its usage somewhat unsafe because it's not clear if such a negative spring stiffness is physically plausible. Furthermore, guaranteeing triangles and tetrahedra of sufficient quality has other negative side effects like enforcing unnecessarily small shapes, causing higher computational load.

A workaround can be implemented through the use of a volume conservation force [72]. The idea is to calculate the initial volume of each tetrahedra and add a force towards its barycenter to each node depending on its current volume. Mathematically speaking this can be expressed as

$$\vec{F_{\mathrm{V},n}} = \alpha \cdot (V_{\mathrm{current}} - V_0) \cdot \frac{\vec{x_{\mathrm{BC}}} - \vec{x_n}}{\| \vec{x_{\mathrm{BC}}} - \vec{x_n} \|}, \tag{2.16}$$

where $\vec{F_{\mathrm{V},n}}$ is the volume conservation force for node n, α a scaling factor, V_{current} the current volume, V_0 the initial volume, $\vec{x_{\mathrm{BC}}}$ the barycenter of the associated tetrahedra and $\vec{x_n}$ the current position of node n [72]. Hence the loop needs to iterate over all tetrahedra and add an accumulating force to each of the four nodes. The downside of this method is that it may not exactly represent the compressibility of a certain material by relating α to its volume energy coefficients.

Note that this volume conservation force has a neat side effect where it acts as a damping force since it makes the nodes push towards their centers at identical volume while the pressure and internal forces cause deformation in the opposing directions. This makes the addition of a separate damping force potentially unnecessary.

2.6 Damping force

It is necessary to use some form of damping to reduce the dynamic fluctuations and simulate some form of internal friction. The used damping formulation is mass-based-damping because it gives a direct relation between the damping factor and the de-acceleration of a given node due to cancelling out the mass during integration [73]. This means the damping force for a node is given by

$$\vec{F_{\mathrm{d},\,i}} = -\vec{v}_i \cdot m_i \cdot \beta \tag{2.17}$$

for node i with velocity $\vec{v}$, mass m and the damping factor β.

2.7 Constraints

For the explicit integration to function correctly, some boundary conditions in the form of positional constraints are necessary. This is easily realized during integration by setting the velocity of the boundary nodes to zero before integrating. The boundary nodes are defined through an axis-aligned bounding box (AABB) [74]. The AABB is user defined and all nodes inside of it are considered fixed.

2.8 Material models

The implemented hyper-elastic material models are Yeoh and Mooney-Rivlin. Both are quickly described by their general compressible strain-energy function [75]

$$W_{\text{Yeoh}}(\bar{I}_1, J) = \sum_{i=1}^{n} C_{i0}(\bar{I}_1 - 3)^i + \sum_{k=1}^{n} \frac{1}{D_{k1}}(J-1)^{2k} \tag{2.18}$$

and

$$W_{\text{Mooney-Rivlin}}(\bar{I}_1, \bar{I}_2, J) = \sum_{p,q=0}^{N} C_{pq}(\bar{I}_1 - 3)^p(\bar{I}_2 - 3)^q + \sum_{k=1}^{n} \frac{1}{D_{k1}}(J-1)^{2k} \tag{2.19}$$

where the C's and D's are material parameters that are experimentally derived through curve fitting [75].

3 Implementation

This chapter is dedicated to describe and justify the choice of tools and resources necessary for development. The development necessitates a choice in programming language, libraries and external programs. Furthermore, both the experiments as well as the verification require meshes that can be used. Lastly, it describes how the implementation works step-by-step and in relation to the MoSCoW-diagram of Section 1.4.

3.1 Development tools

Because the developed code should be as accessible as possible to possible users the choice of programming language is Python. Due to being an interpreted scripting language Python alone does not deliver the desired performance for a computationally heavy problem like FEM [76]. A remedy for this is the usage of Taichi [77], which is a compiled programming language that works alongside Python and has almost identical syntax. In addition, the compilation can automatically generate CUDA [78] kernels to utilize the GPU. Furthermore, assuming certain paradigms are followed, Taichi includes an auto-differentiation system that can be used instead of implementing it manually. Hence the combination of Python and Taichi gives the ease of implementation and modifiability of Python and the performance of compiled GPU programs in addition to fully solving the question of how to implement auto-differentiation.

The Python libraries used for development are Numpy, Numpy-stl, pathlib, vedo, scipy, Taichi and Taichi-glsl. While Taichi is technically it's own language it is embedded in Python like a library. Numpy and Numpy-stl is for general array calculations and extensively used in the pre-processing steps of the meshes [79] [80]. Pathlib gives access to the execution path of the files and this allows for path invariance when importing functions from self-made modules [81]. Vedo is used as a visualization tools for the meshes and allows loading of multiple meshes and control through interface objects, such that the motion of the meshes during the simulation can be shown, as well as multiple meshes at the same time [82]. Scipy contains algorithms for scientific computation like n-dimensional interpolation functions useful for verification [83]. Lastly, Taichi-glsl is an extension library with some quality-of-life functions like cross-products, dot-products and smoothing functions.

The program tetgen [84] will be used to tetrahedralize the input meshes. This allows the user to use stl-files as input, which is one of the most common and simple mesh file extensions and virtually every CAD program can export them [85].

Lastly, the author uses a computer with an Intel i7-6700HQ CPU and a Nvidia GTX 965m GPU. This information may be relevant to extrapolate certain performance measures mentioned in the thesis to other systems.

3.2 Meshes

For both the verification and in relation to the research questions, meshes are necessary for simulation and optimization. One of these is an endoscope model that was used in previous research. It's intended to exemplify a somewhat realistic and more complex use case of the program. It contains three pressure chambers with rotational symmetry of 120° to allow a bending motion in the plane defined by it's main axis. An isometric and front view of the mesh can be seen in Figure 3.1. The second mesh is a simple cuboid test mesh modelled by the author with a smaller cuboid as a cavity. The cavity is slightly displaced from the center along the $z-$axis to cause a bending motion instead of simple inflation. The isometric and front view is shown in Figure 3.2.

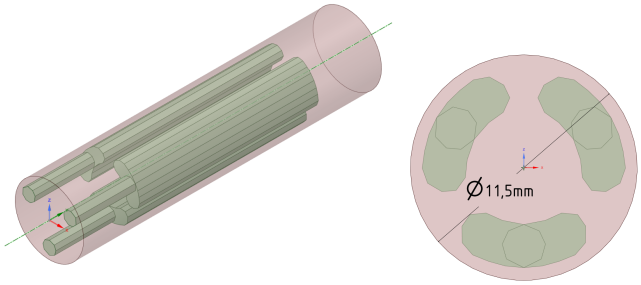


Figure 3.1: Isometric and front view of the endoscope mesh. It contains three pressure cavities (green) that allows the end-effector to bend around the $x-$ and $z-$axis

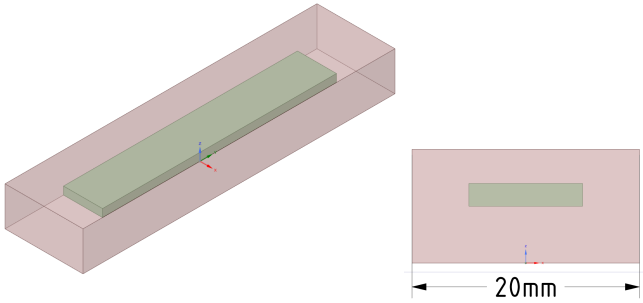


Figure 3.2: Isometric and front view of the test cuboid mesh. It contains one pressure cavity (green) just above its center to bend around the $x-$axis.

3.3 Features and workflow

This section will roughly explain how the implementation works and how it is used. In reference to the MoSCoW-diagram Figure 1.2 the implemented features are the following.

- Soft actuator simulation and gradients of any parameter from any loss function
- Mesh visualization
- 2 Optimization schemes (Gradient descent and NADAM)
- Support for an arbitrary number of pressure chambers
- 2 approaches to internal forces (Spring-mesh and cauchy stress)

The workflow of the program can been shown in a number of block-flow-diagrams and are explained in the following subsections.

3.3.1 Top-level

The top-level diagram is depicted in Figure 3.3. At the highest level the code starts with the definition of the constants that define the simulation. These are put at the beginning of the diagram as they're always necessary and are fittingly defined in the beginning of the Python file. These consist of mesh parameters for the geometry and the material. For the simulation a number of constants for Equation 2.2 - Equation 2.17 are set.

The next step is the mesh pre-processing where the stl-files are tetrahedralized and all necessary mesh data like the pressure chamber surfaces and mesh elements returned to the program. With the pre-processing done the main process fo the code can begin. Because it may be useful for the end-user to just simulate, for instance when testing damping factors, a bool was defined that allows choosing between performing optimization and only simulating. If optimization is chosen more parameters need to be defined for the conditions of the optimization. Afterwards

the optimization loop runs where the simulation including gradient calculation is perfomed and afterwards the parameters to optimize updated. The user has the choice to save the vertex data, face data, optimized parameters and losses of the optimization or simulation. The last step is visualization where all time steps of the final simulation are shown.

3.3.2 Pre-processing

The pre-processing is shown in Figure 3.4. It consists of two broad steps. The first is the combination of the given stl-files and the tetrahedralization. The second is the processing of the resulting tetrahedra-mesh to obtain a number of necessary arrays for the simulation.

The first step is defining two constraint constants for tetgen. Their purpose is controlling the quality and resolution of the resulting tetrahedra mesh. The first processing step is loading the stl-files using numpy-stl. The outer mesh and cavity stl's can be concatenated and saved in a poly-file which is a general polygonal mesh file format that can be used as input to tetgen. It also supports the definition of holes for cavities and different ID's for nodes and faces. These functions can be used to later identify the different faces and ID's for the cavities. This file can then be fed into tetgen with the previously defined constraints to produce the tetrahedralized mesh.

The second part starts with loading the tetgen output which consists of nodes, faces, edges and elements. Afterwards three important data arrays are created by processing the tetrahedra data. Note that this processing does not produce new data but rather reorganizes the available ones into new arrays for performance reasons. The first one is an array containing all adjacent elements for each node for fast calculation of the mass and volume of a given node. The second is a dictionary containing the ID's of the cavity surface faces for each pressure chamber. The key is the chamber ID and the content a list with the ID's. Lastly all data is returned to the main program.

3.3.3 Simulation procedure

The main steps of the simulation procedure consists of the initialization and the simulation loop itself. The first step is determining if the initialization is even necessary because during optimization there is no reason to waste time with initialization.

Because Taichi utilizes the GPU to speed up it's computation it's necessary to explicitly define the variables in taichi-format to use them on the GPU similar to buffers in compute shaders [86]. Furthermore, memory needs to be allocated manually by defining the sizes and data-types of the arrays. After the variables are defined and allocated correctly they can be initialized using the numpy arrays and dictionaries from the pre-processing and the previously defined constants for the simulation. As a final preparation step the taichi functions need to compiled by running them once.

The simulation loop then consists of checking whether the loop is finished and if not sum up the forces for the current time step. The first one is the pressure on the cavity surfaces from Equation 2.15. Afterwards a constant expression is used to determine at compile-time which internal forces model is used and the appropriate equations applied. Lastly the damping forces are added and the forces and velocities integrated. After the pre-defined number of steps is reached the program may calculate the desired gradients assuming optimization is performed.

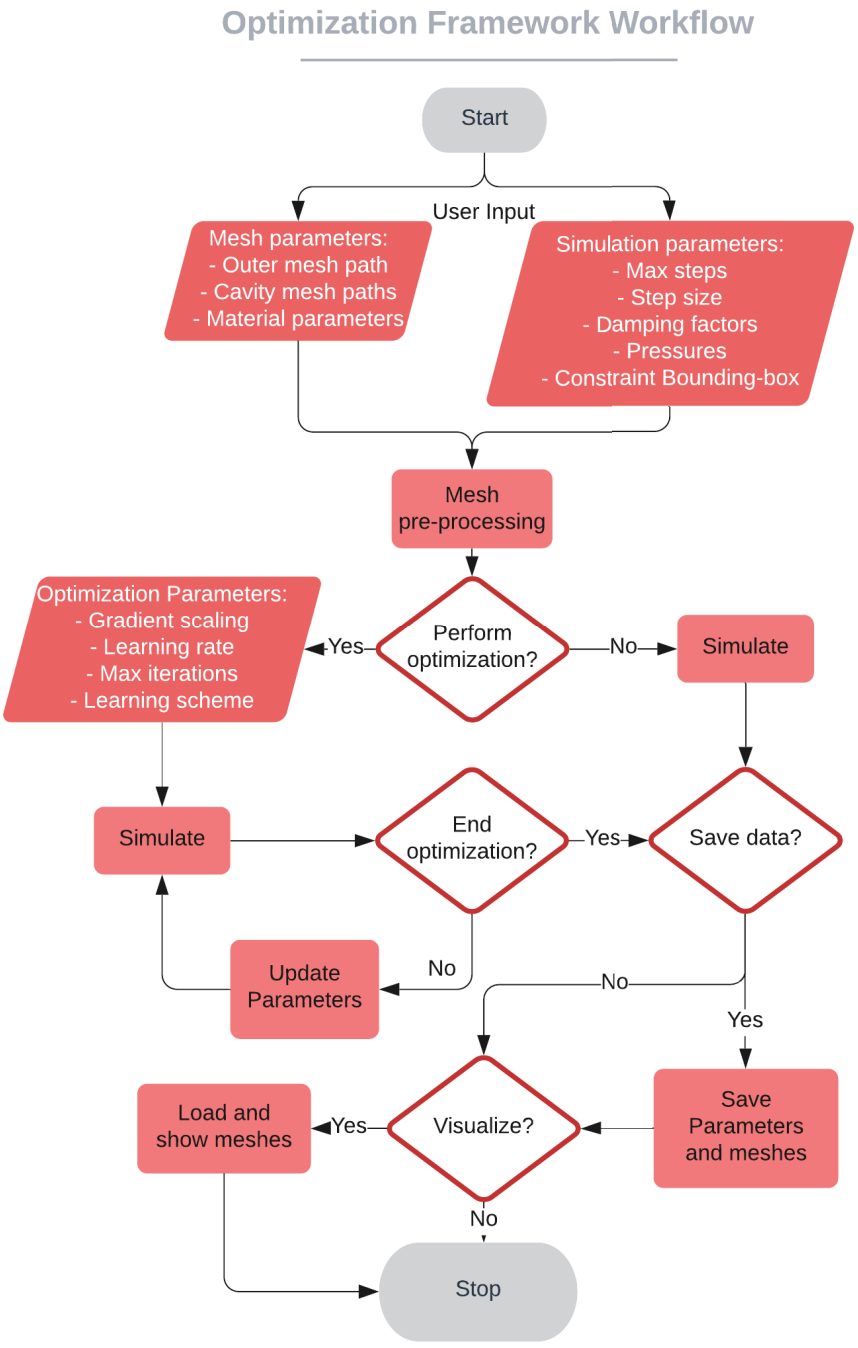


Figure 3.3: Top-level diagram of the developed program. The steps "Mesh pre-processing" and "Simulate" are shown in more detail in Figure 3.4 and Figure 3.5, respectively.

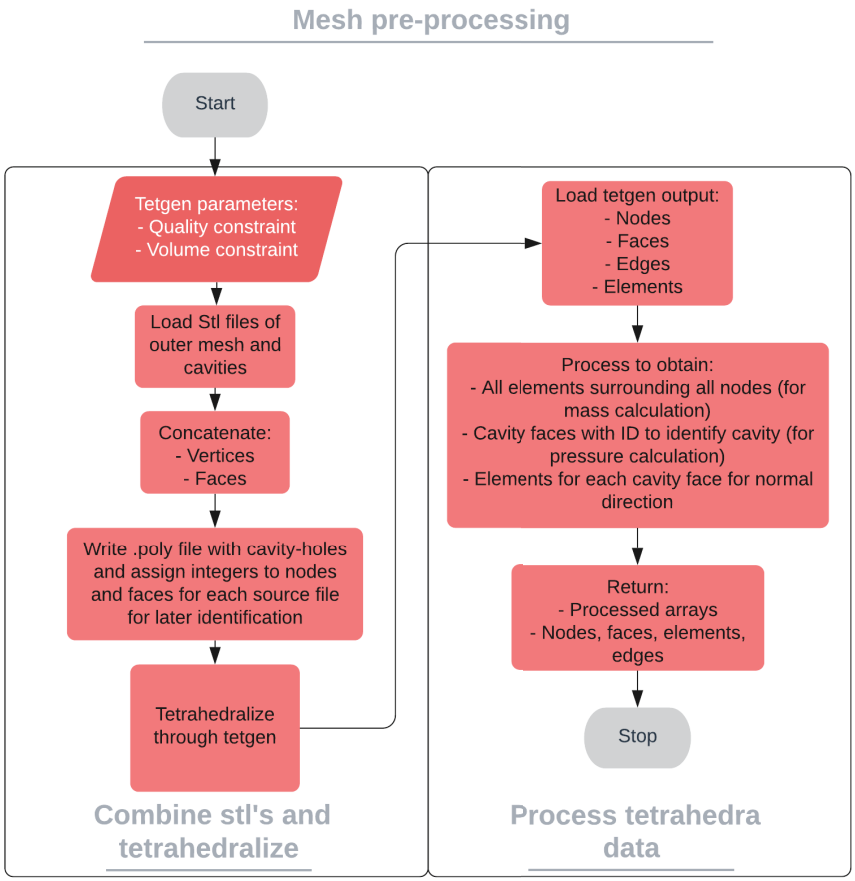


Figure 3.4: Pre-processing steps to obtain all necessary data for the program from stl-files.

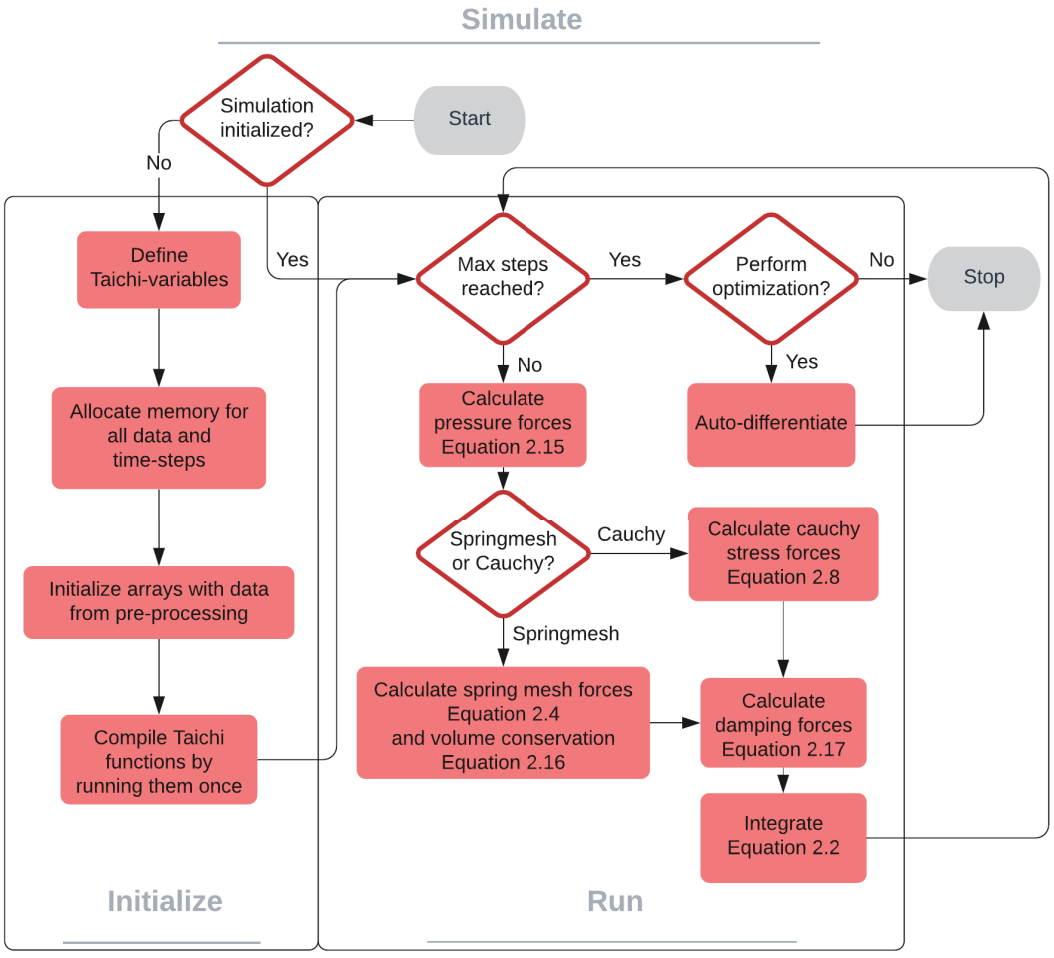


Figure 3.5: Flow diagram showing the steps of the simulation. The initialization is optional to avoid unnecessary steps when repeatedly simulating during optimization.

4 Verification

This chapter is dedicated to explaining and performing the verification for the underlying simulation. It is split into the design of the verification process, the software setup and lastly the results and their interpretation.

4.1 Design

To verify the simulation, Abaqus [87] is used. Abaqus has both implicit and explicit FEM tools for a variety of hyper-elastic materials models, and gives control over meshing behaviour and rank of interpolation inside elements. Furthermore, it allows exporting the stress tensor, strains and displacements on a per-element and per-node basis. This enables a quantitative comparison between the developed simulation and a high-quality solution from Abaqus as an established FEM solver.

The compared data will be the per-node displacement in the bending direction. This is because in an isotropic material checking for one direction should suffice as the other directions don't differ calculation wise. Also, the volume forces influence the bending as well. Because the input meshes for both Abaqus and the simulation are stl-files have different tetrahedralization algorithms, it is necessary to generate an interpolation field from the data to quantitatively compare them. This interpolation is performed using scipy's LinearNDInterpolation function to generate a field from the exported Abaqus data. The field will be evaluated at the node positions of the simulation, which is expected to give sufficiently accurate information for verification.

To verify that the implementation of the pressure and internal forces are correct, a total of 4 sets of data will be compared. This includes the two presented meshes at two different pressures. The displacement error can then be visualized by showing the deformed mesh with a colormap that shows the displacement error for every position on the mesh. This gives an intuitive idea of how accurate the simulation is, what are the minimum and maximum error, and which positions are worst.

Furthermore, a graph showing the error in dependence of the tetrahedralization resolution of the simulation. This gives both an idea of how important the tetrahedralization is as well as strong evidence for or against the implementation. This is because, ideally, the simulation should get closer to Abaqus as the resolution gets closer to its regular tetrahedra.

To be clear about the calculation of the error the following equations sums it up.

$$e_n = |d_z(n) - d_{z,\text{ABAQUS}}(x_n, y_n, z_n)| \tag{4.1}$$

with e_n being the error of node n, $d_z(n)$ the $z-$displacement of node n in the simulation and $d_{z,\text{ABAQUS}}(x_n, y_n, z_n)$ the $z-$displacement in the interpolation field at the coordinates of node n.

4.2 Setup

Because the goal of the verification is to ensure that the underlying calculations of the simulation are correct, it is required to model the system in Abaqus as close as possible to the methods used in the simulation. Because of this, the elements will be linear and maintain the standard settings for everything else. The exact model parameters used are the following.

- Yeoh material parameters for silicone rubber from [88]
  - $C_1 = 0.24162$
  - $C_2 = 0.19977$

- $C_3 = -0.00541$
- $D_1 = 1.0$
- $\alpha = 0.05$ (Section 2.5) by trial and error for the spring-mesh

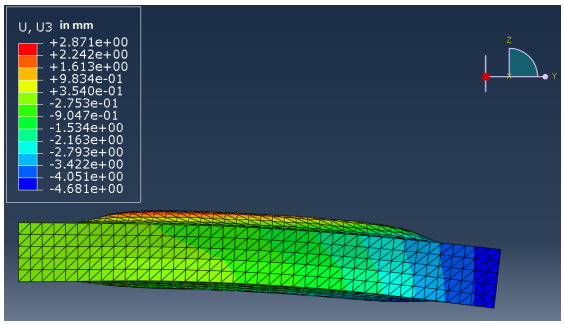
- Meshes
  - Endoscope at pressures $p_1 = 1$kPa and $p_2 = 2$kPa
  - Cuboid at pressures $p_1 = 2$kPa and $p_2 = 3$kPa

Nodes on one side of the mesh are fixed, as boundary conditions. These fixed planes can be seen in Figure 4.1a - Figure 4.4b.

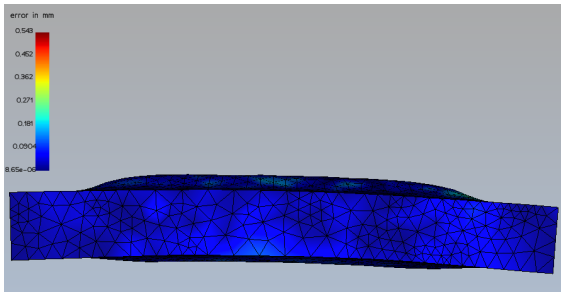
For the simulation, the number of steps varies depending on the pressure because higher pressure need more time to reach steady state. However, since only the final deformation is interesting, this is not crucially important for the verification. The step size is 0.0005 seconds.

4.3 Results and Interpretation

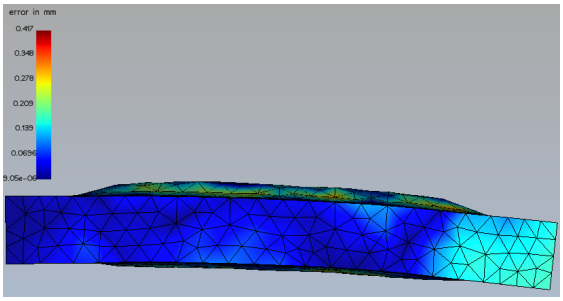
As mentioned in Section 4.1 the results will be presented in a number of 3D plots showing the results of the 4 datasets for both Abaqus and the simulation. The Abaqus results use a colormap for the displacement in $z-$direction while the simulation uses the magnitude error of the $z-$displacement from Equation 4.1 as a colormap for the vertices. The following plots show each case with the Abaqus solution and the error in the developed simulation.



(a) Abaqus, Test Cuboid, 2kPa. Color indicates displacement along $z-$axis.



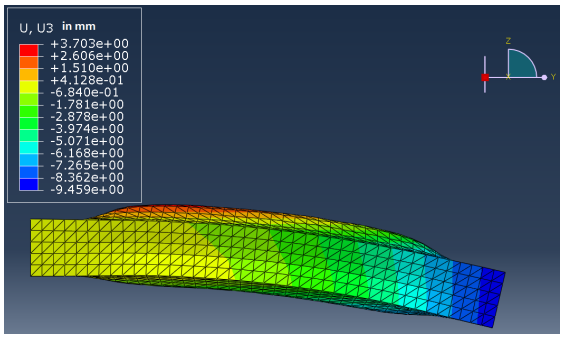
(b) Simulation, Test Cuboid, 2kPa, Cauchy stress forces. Color indicates absolute error to the Abaqus solution.



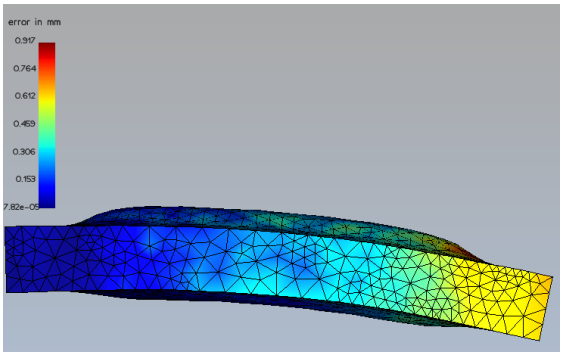
(c) Simulation, Test Cuboid, 3kPa, Spring-mesh. Color indicates absolute error to the Abaqus solution.

Figure 4.1: Abaqus and Simulation results in comparison. Using cauchy stress the vertices at the right end have a error of about 0.04mm, which amounts to < 1% deviation from Abaqus. As a spring-mesh the end-surface has an error of about 0.135mm, amounting to about 2.9%.

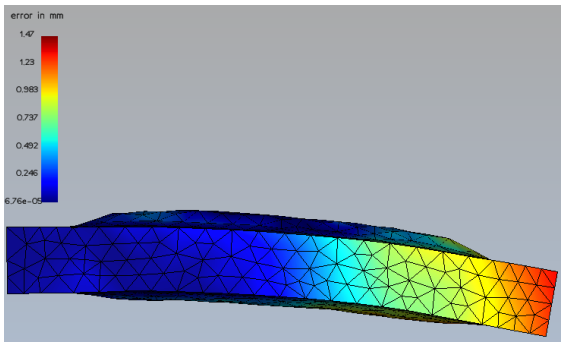
There are a few interesting observations to make here. First of all, when ignoring the quantitative error indicated by the color-maps the solutions all have a similar shape compared to the ground truth of Abaqus. This indicates that the overall behaviour is correctly simulated and there are no major breaking issues in regard to any forces. The graphs of Figure 4.5 strongly



(a) Abaqus, Test Cuboid, 3kPa. Color indicates displacement along $z-$axis.



(b) Simulation, Test Cuboid, 3kPa, Cauchy stress forces. Color indicates absolute error to the Abaqus solution.



(c) Simulation, Test Cuboid, 3kPa, Spring-mesh. Color indicates absolute error to the Abaqus solution.

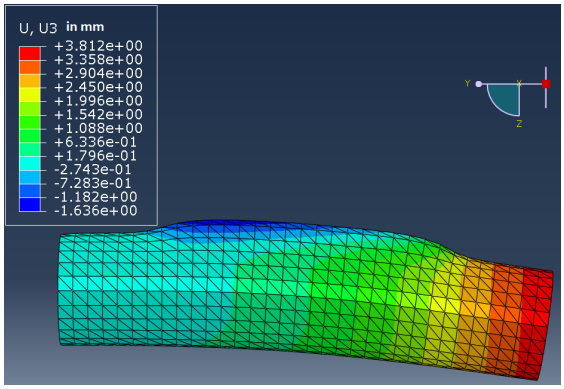
Figure 4.2: Abaqus and Simulation results in comparison. Using cauchy stress the vertices at the right end have an error of about 0.63mm, which amounts to about 6% deviation. As a spring-mesh the end-surface has an error of about 1.47mm, amounting to about 15.5%.

support this claim as the error decreases and seemingly approaches zero with increasing resolution.

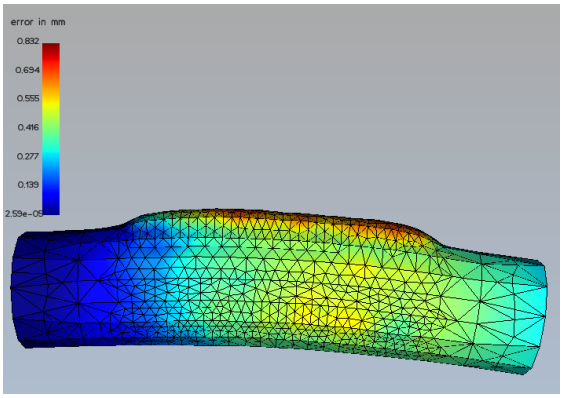
Second, looking at the absolute error, it can be seen that the points of maximum difference are, interestingly, not at the very end of the model where the maximum displacement occurs when using cauchy stress, but rather appear on the bulge of the pressure chambers. It is possible that the bulge has not fully relaxed or reached steady state yet because the author focused on the end-effector standing still at the final time step and ignored the other parts of the mesh. Alternatively it could be caused by higher errors at large strains due to tetrahedralization differences.

Third, it's noteworthy how different the tetrahedralizations are and how similar the solutions are despite it. Especially for such non-linear problems, FEM is infamously dependent on the quality of tetrahedralization and one may argue that the larger tetrahedra are expected to negatively affect the solutions even stronger than the measured errors indicate [89]. Abaqus' tetrahedralization is very regular while tetgen's depends on the constraint factors and mesh geometry. Figure 4.5 shows that these differences become negligible for sufficiently high resolutions.

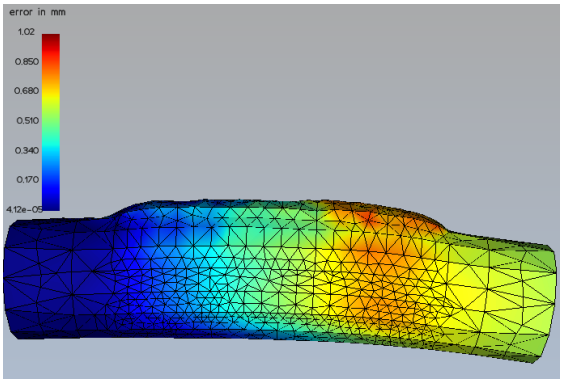
Lastly, the error at the ends of the meshes are all within a few percent deviation under cauchy stress forces. This is evidence that there are no issues with the implementation because mistakes in the underlying calculations are expected to cause more significant issues that cannot be explained through e.g. different tetrahedralizations, lower resolutions or not having reached perfect force equilibrium within the time horizon. Because the spring-mesh is a less physically motivated approach there are more issues. In particular, the volume conservation force has no obvious relation to material parameters, meaning the author had to approximate a value based



(a) Abaqus, Endoscope model, 1kPa. Color indicates displacement along $z-$axis.



(b) Simulation, Endoscope model, 1kPa, Cauchy stress forces. Color indicates absolute error to the Abaqus solution.

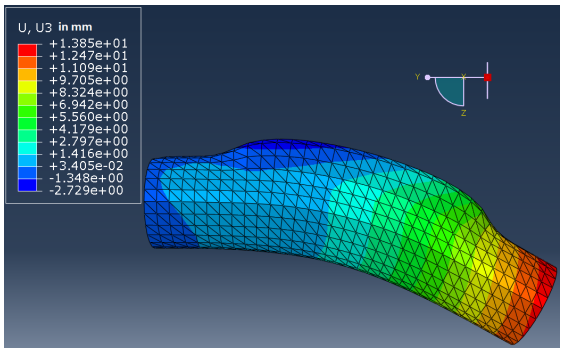


(c) Simulation, Endoscope model, 1kPa, Spring-mesh. Color indicates absolute error to the Abaqus solution.

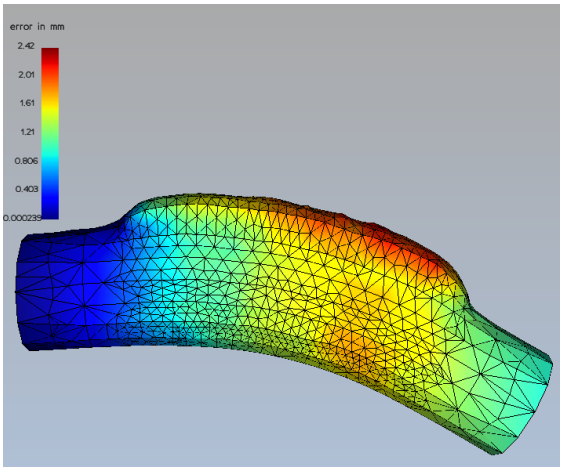
Figure 4.3: Abaqus and Simulation results in comparison. Using cauchy stress the vertices at the right end in cyan have an error of about 0.27mm, which amounts to about 7% deviation. As a spring-mesh the end-surface has an error of about 0.6mm, amounting to about 15.7%.

on trial and error to get as close to the reference solution as possible and yet the accuracy is lower than cauchy stress forces.

Another noteworthy point is the performance of the simulation compared to Abaqus. As it turns out, the simulation appears to be faster than Abaqus and the speed difference between the two internal forces models is negligible. The forward simulation runs above real-time without explicitly focusing on maximizing the performance during development. With a simulated time of about $2-3$ seconds until steady state for the larger deformation of the endoscope, the final state is reached quicker than Abaqus. Even when counting all the preprocessing of the stl-meshes, tetrahedralization and compiling of the GPU functions the simulation is faster than Abaqus when counting from submitting the job to its finish. The time step is 0.0005 seconds with 6000 steps and the total duration of the **entire simulation program** is about 10 seconds. The pure simulation, however, runs faster than real-time as the average frames per second are between 2500 and 6000, depending on the mesh. Interestingly, the main time consumer of the simulation is the compilation of the taichi functions, which takes about 5 seconds. In comparison, when everything is already set up in Abaqus, submitting the job until it is finished with the results available takes 48 seconds. This is a speedup of about 80%. Obviously, Abaqus uses much more sophisticated preprocessing to check the input file, is much more robust, and saves all results in a large file. Still, when comparing the whole workflow, meaning the entire setup of the simulation including meshing, the developed simulation is several orders of magnitude faster because Abaqus requires minutes of preparation.



(a) Abaqus, Endoscope model, 2kPa. Color indicates displacement along $z-$axis.

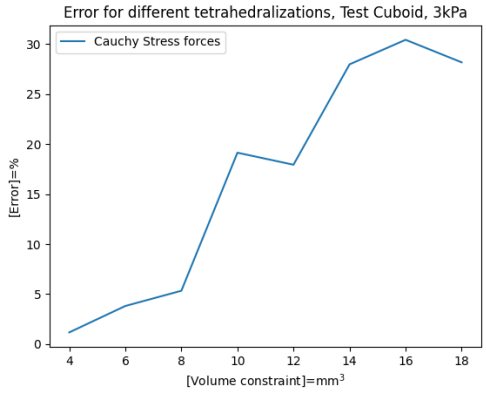


(b) Simulation, Endoscope model, 2kPa. Color indicates absolute error to the Abaqus solution.

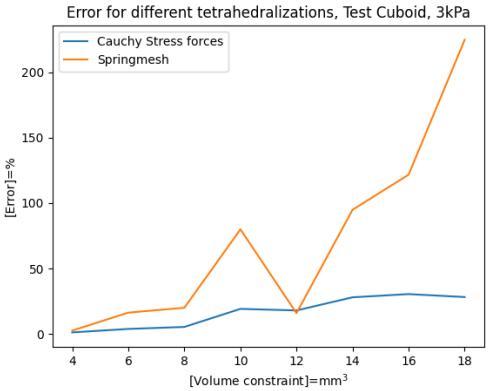
Figure 4.4: Abaqus and Simulation results in comparison. The vertices at the right end in green-blue have an error of about 1mm, which amounts to about 7.2% deviation. The spring-mesh simulation converged but struggled with element inversion and resulting inaccuracies despite varying the damping factor and pressure ramp.

Lastly, the spring-mesh did not manage to produce meaningful results for the 2kPa endoscope run because several elements near the pressure chambers inverted. The author attempted to fix this issue by changing the damping factor and pressure ramp but the issue remained. This implies some stability issues and arguably makes the approach much less useful.

From this comparison, the author interprets these results as sufficient proof that the underlying implementation is correct and the remaining steps of the project can be carried out based on it. Also, because the spring-meshes consistently produced worse results and even struggled with stability the author is going to use cauchy stress forces exclusively for the rest of the thesis.



(a) Error graph for cauchy stress. The graph seems to move towards zero with higher resolution



(b) Error graph for spring mesh and cauchy stress. The spring mesh forces become unstable at a constraint > 8 because the low resolution causes element inversions.

Figure 4.5: Error over tetrahedralization resolution for both internal forces approaches. The volume constraint is an absolute limit for the volume of each tetrahedron and thus directly affects the resolution. Both graphs seem to approach nearly zero error as the resolution increases but the cauchy stress is consistently better and also maintains stability.

5 Exploding gradients

This chapter is dedicated to explaining and showing how the issue of exploding gradients is addressed in the implementation of the framework. This is done by first explaining the mathematical background and context of the problem and how it leads to finding the solution. Afterwards, some limitations that result from this solution are addressed to make it more robust. Lastly, a few exemplary plots of a simple optimization and their learning curves are shown.

5.1 Mathematical background and approach

Exploding gradients in automatic differentiation is a problem that arises due to using the chain rule of differentiation [90]. To concisely explain how this leads to the gradients exploding the following equation shows the general expression of the gradients for the addition and multiplication operator [91].

$$\begin{aligned} c = a + b &\Longrightarrow \partial c = \partial a + \partial b \\ c = a \cdot b &\Longrightarrow \partial c = b \cdot \partial a + a \cdot \partial b \end{aligned} \tag{5.1}$$

Now considering the forward euler integration of Equation 2.2 it becomes clear that integrating over several thousand time-steps results in a chain of multiplication with roughly the same number of links as there are time-steps. Consequently, if many individual links of that chain contain values > 2 then the variable containing the total gradient will grow larger and eventually overflow. As an example, if only the first 100 links in the chain are exactly 2 then the value will be $2^{100} \approx 1.26 \cdot 10^{30}$ which is beyond the capability of a 32-bit floating point number. Even a 64-bit double will fail at $2^{200} \approx 1.6 \cdot 10^{60}$. Hence it is necessary to scale the links to reliably avoid this.

Looking at the literature the issue of exploding or vanishing gradients certainly is not novel. This is because it is not unique to differentiable numerical simulations but rather a consequence of auto-differentiation of large chains of computations and recursions [92] [93] due to using the chain-rule of differentiation. Consequently, the issue also appears in neural networks once they reach a certain size or simply because of recursive layers. Due to the wave of research in this area in the recent years, the issue has been discussed and addressed many times [94] [95] [96]. The problem here is that the gradients of the simulation exist under different conditions compared to neural networks. In particular, neural networks have less recursion compared to the thousands of time steps, different non-linearities and helpful pre-processing like input and weight normalization to put limits on the gradients. Hence the methods used in the literature are not necessarily directly applicable.

Instead it makes more sense to look into what auto-differentiation does to the Euler integration term. If a connection between the values at integration and the gradients at a given time-step can be found, the solution is to scale those accordingly because all simulation parameters depend on this integration, meaning scaling it will affect all parameters. Also note that the used auto-differentation framework of Taichi does not give access to the numerical values of the gradients of each time step. However, it does allow using custom functions and different function arguments during differentiation, meaning the original integration expression can be scaled and then differentiated. The expressions of gradients for different numerical integration forms have been studied before [97]. An interesting observation to make is that the total gradient can be formulated in a numerical integration form in a similar way to the original differential equation. Mathematically speaking this means that the gradient of the velocity term in the integration of the position

$$x_n(t) = x_n(t - \Delta t) + v_n(t) \cdot \Delta t \tag{5.2}$$

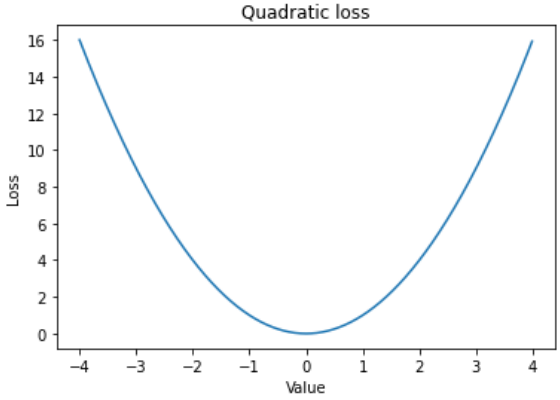


Figure 5.1: Exemplary quadratic loss function to demonstrate importance of gradient magnitude information

can be written as

$$\frac{\partial v_n(t)}{\partial p} = \frac{\partial}{\mathrm{d}t}\left(\frac{\partial x}{dp}\right) \tag{5.3}$$

as shown in [97] and then integrated through time in parallel to the original integration to obtain the gradient. The variable p is any parameter of the simulation for which gradients are desired. While this is not how backwards auto-differentiation is performed algorithmically it implies that this velocity term can be scaled during differentiation to scale the gradients accordingly. This means that the solution to the question is adding a scaling factor s as an argument to the function and using 1.0 during forward simulation and some smaller factor when differentiating.

$$v_n(t) = (v_n(t-\Delta t) + F_n(t)\cdot\Delta t)\cdot s \tag{5.4}$$

This solution seems simplistic at first but has useful mathematical properties. First, it maintains gradient direction, meaning the resulting gradients point in the same direction and can therefore be used, just scaled down. Second, because it is a single linear operation on the gradients of each time step meaning it also maintains magnitude information. To understand what this means consider a quadratic loss function like shown in Figure 5.1. The gradient of the loss function is stronger the further away the value is from the optimum. Because of this, the optimization will naturally converge to the optimum if the size of the value change per iteration is directly proportional to the gradient like in gradient descent. Scaling down the gradient per time step may change the exact magnitude and even the order of magnitude between consecutive gradients but keeps this property overall.

5.2 Issues and implemented remedies

While it is quite easy to avoid exploding gradients with this method, some new problems arise. The first one is that the relation between the total scale and the per time step scaling is not obvious, meaning it is hard to intuitively get a good value. Consequently the total gradient may be useless for the variable to optimize if it is scale is vastly different compared to the variable. For instance, a variable that has its optimum between 1.0 and 2.0 may get gradients somewhere around 10^{-10} with a too small factor s. The solution to this is to first normalize the order of magnitude of the total gradient. Then the order of magnitude of the variable in question and the learning rate can be used to scale it to sensible ranges. To be more clear the following

equation shows the calculation:

$$
\begin{aligned}
z_{\text{g,initial}} &= \begin{cases} 10^{(\lfloor \log_{10} |g_{\text{initial}}| \rfloor)}, & \text{if } |g_{\text{initial}}| > 0.0 \\ 1.0, & \text{otherwise} \end{cases} \\
z_{\text{weight, current}} &= \begin{cases} 10^{(\lfloor \log_{10} |w_{\text{current}}| \rfloor - 1)}, & \text{if } |w_{\text{current}}| > m \\ 0.1 \cdot m, & \text{otherwise} \end{cases} \\
g_{\text{scaled}} &= g \frac{z_{\text{weight, current}}}{z_{\text{g,initial}}},
\end{aligned}
\tag{5.5}
$$

where z are the orders of magnitude of the first gradient g and current variable w in question and g_{scaled} the gradient after the normalization and scaling operation. The constant m determines the minimum rate of change per iteration for a given variable. This scales the gradient to one order of magnitude lower as the variable to be adjusted. Practically, this means that the gradient will be limited by $|g_{\text{scaled}}| \in [0, m)$. The learning rate is not included in Equation 5.5 because it would assume the usage of a specific optimization scheme.

Still, assuming gradient descent, this has the neat side effect of giving the learning rate an intuitive interpretation as the maximum percentage of change per iteration. For instance, a learning rate = 0.1 would imply the value changes by a maximum of 10% per iteration. Depending on the use case this may or may not be a useful quality because different variables may need vastly different learning rates with this approach. Hence, the user is encouraged to adapt this calculation in accordance to their needs. It is also possible to scale them by finding the largest gradient of all variables in question so that relative magnitude information between multiple variables is not lost. The author believes this scaling ability to be a useful property because it gives tighter control over the learning behaviour.

A potential problem of this approach is related to the scaling and the magnitude information of the gradients. Because the gradients at the optimum are 0 regardless of the scaling factor s the differences between subsequent gradients of the simulation are stronger the further away the initial gradients are from 0. Consequently, too small scaling factors s make the usage of Equation 5.5 less applicable as the initial order of magnitude that is used may make the scale of later gradients unusable or even unstable. Hence care must be taken when choosing a factor and ideally the order of magnitude of the unscaled gradients g should be between 1 and 10^3 if possible.

In general, future researchers dealing with this are encouraged to exploit the availability of this gradient information to dynamically scale them to their needs. Considering that there is no information about the ideal scale of the gradients this approach seems to be a solution that is, at least conditionally, able to produce useful gradients. The next section will show how this approach can work with a simple example. The following chapters can be interpreted as further and more complex evidence.

5.3 Exemplary gradients

The example is an optimization of the maximum pressure applied to the cuboid test mesh. The loss function wants the end effector to move 10mm downwards the $z-$axis from the initial position. This is formulated as follows:

$$
L = \sum_{k=1}^{n} (x_{k,t_{\text{max,z}}} - x_{k,0,\text{z}} + 10)^2, \tag{5.6}
$$

where L is the total loss, n the number of nodes at the end surface, x the position of the $k-$th of these nodes at the final time-step t_{max} and $t = 0$ and only its $z-$component. The initial pressure is 1.0kPa (deka-kilo-Pascal due to the units of the model) because from the verification runs it is

clear that this is far from the optimal value. The threshold m is set to 0.1 because this is the order of magnitude of reasonable pressures for this robot from experience with the verification. The gradients are scaled in the exact way it was explained above and a learning rate of 1.0 is used because the optimum is expected to be several times higher than the initial value. The scaling factor s from Equation 5.4 is chosen to be 10^{-3}. The plots will show the deformed optimized mesh, loss function, pressure and the scaled and unscaled gradients.

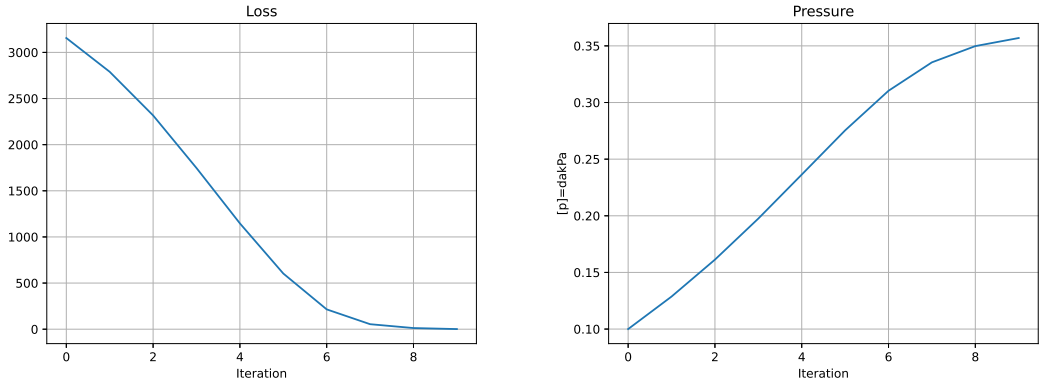


Figure 5.2: Loss function and pressure during optimization. Notice the ideal shape of the loss and pressure. The final loss is 0.88 and the final pressure 36.1kPa

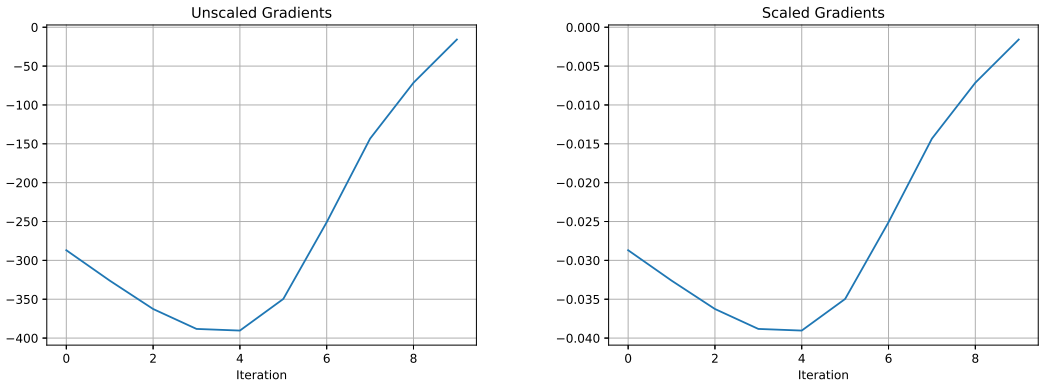


Figure 5.3: Unscaled and scaled gradients for the pressure optimization.

Looking at Figure 5.2 the first observation is the seemingly ideal shape of the learning curve and the low number of iterations needed to reach the global optimum. The loss function converges to nearly zero because the goal is simple and reachable by increasing the pressure. Figure 5.3 shows the gradients both scaled and unscaled. These two figures together give an idea about how well the scaling works. The order of magnitude of the unscaled gradients is far too large to be useful. In the first 3 steps of the optimization the pressure is below $m = 0.1$, the gradients are scaled to 10^{-2}. Afterwards the value is above m but still scaled to 10^{-2} because of Equation 5.5. One may wonder why the gradient becomes stronger during the first few iterations. This is theorized to be caused by the non-linearities of the simulation. In particular, the material model is a third order equation, meaning there is a valley during which an increase in pressure has a stronger effect on the deformation before it gets diminishing returns. Lastly, the effect of the maintained gradient magnitude information is very visible here as the weaker gradients cause convergence without fluctuations towards the end. Figure 5.3 shows how the gradient is positive in the final step, implying that the breaking condition was reached just as the pressure became slightly too large for the exact optimum.

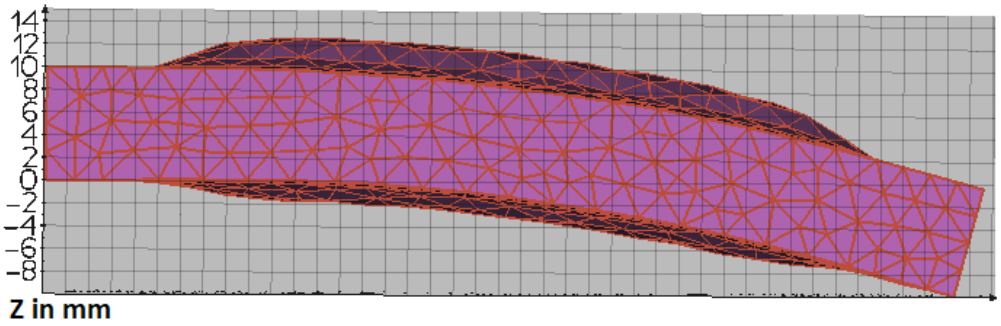


Figure 5.4: Final deformation of the optimized mesh. As can be seen from the axis, the end effector is nearly exactly 10mm lower than the initial position as expected from the loss function.

Figure 5.4 shows the deformed mesh with optimized pressure. It can be seen how the end effector surface seems to be almost exactly 10mm lower than the initial position. This is expected considering the loss of nearly 0.

Despite the seemingly perfect behaviour in this example there are some things to take into account when judging the scheme. First, some prior information about the mesh and reasonable pressures was used to improve the speed of convergence. A lower threshold m or learning rate could have slowed down the process considerably. That being said, finding good values is arguably easy because they are directly related to the change in the loss function, meaning even if nothing is initially known about the optimal values a few test runs over few iterations will quickly give some intuition to what are good values. Lastly, while not implemented in this case, it is theoretically possible to tune m adaptively similar to how certain optimization schemes tune the learning rate [98]. This could be done, for instance, by checking if the minimal rate of change is too large or small by measuring the change in the loss function per iteration.

6 Endoscope controller synthesis

This chapter is dedicated to the third research question regarding controller synthesis. The idea is to use the analytical gradients from the simulation to train a small neural network controller for the endoscope model. Section 6.1 will explain in detail how the controller is trained. Section 6.2 shows the results of the optimization and interpret them. Lastly, Section 6.3 compares the results to competing solutions from the literature and judges the training.

6.1 Controller and training design

The input to the controller is the desired target position in the $x-z-$plane. There are a total of four inputs where each component of the target vector is split at zero, i.e. into its positive and negative components. To avoid confusion, the inputs are calculated as follows:

$$i_1 = \max(0, t_x), \quad i_2 = \min(0, t_x), \quad i_3 = \max(0, t_z), \quad i_4 = \min(0, t_z) \tag{6.1}$$

t_x is the $x-$component of the target vector and t_z the $z-$component accordingly. Each of these inputs is then multiplied with a weight and summed. Afterwards, to avoid instability and vanishing gradients from clamping, a standard logistic function scaled between 0 and 2.0kPa is used to scale the pressure.

$$p(x) = \frac{0.2}{1+e^{-x}} \tag{6.2}$$

Equation 6.1 is necessary because the chambers are aligned in an asymmetrical way meaning the weights for positive and negative values in the target vector need to be different. To be clear about the computation, this means there will be a total of 12 weights from 4 inputs times 3 chambers and Equation 6.2 acts as the activation and output function for the pressure. The architecture can be seen as a graph in Figure 6.1.

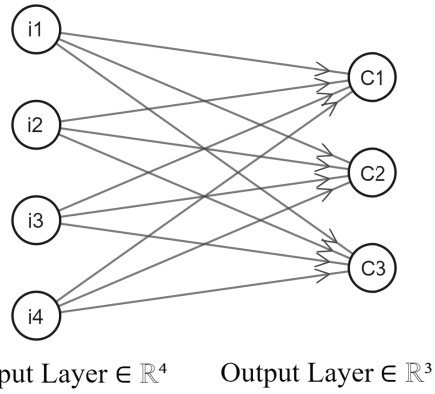


Figure 6.1: Architecture of the controller. The activation function for each chamber C is the sigmoid of Equation 6.2

The network is used to transform the target position into a pressure for the three chambers, meaning it is an open loop controller. The loss function is the squared error between the end effector position, which is the top right surface in Figure 3.1, and the target position at the end of the simulation. The formulation is the same as Equation 5.6 but with $x-$ and $z-$components and a varying distance for each direction. The training will use mini-batches of 4 points each with target points randomly sampled in a circle of 5mm around the initial position in the $x-z-$plane while making sure each batch contains one point from each quadrant in the plane.

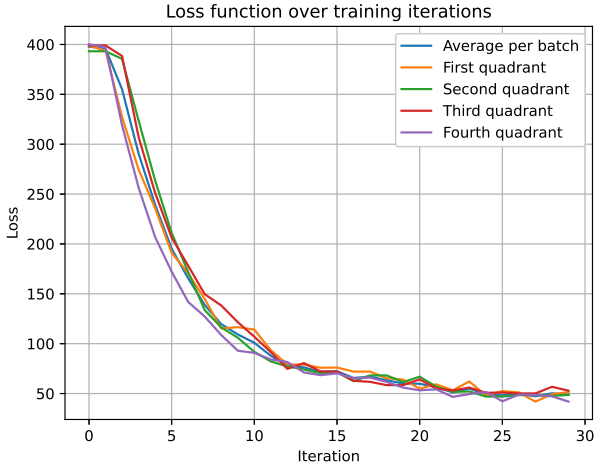


Figure 6.2: Loss for each quadrant in the $x-z-$plane and the average in comparison

The optimization scheme will be stochastic gradient descent because it has been shown to generalize more reliably for certain use cases [99] even if this is not a deep learning scenario.

Because the purpose of this study is to act as a proof of concept for training controllers, the performance of the controller (e.g. its error to the target position) is not the main priority, hence why it only has one simple layer. Instead, the main points of interest are the training speed and convergence behaviour as competing solutions in literature have to rely on reinforcement learning or work-arounds to get gradients for training.

In particular, it is expected that the training will only take a few iterations when using such a simple network and appropriate mini-batches. The time-step is 0.0005 seconds with a total of 3000 steps per iteration which means 1.5 seconds. The learning rate for gradient descent was chosen to be 1.0 with the scaling scheme presented in Chapter 5. The threshold m from Equation 5.5 is 0.1 again. The scale s from Equation 5.4 is set to 10^{-2}.

6.2 Results

Because the training used mini-batches consisting of the 4 quadrants in the $x-z-$plane, the loss function can be shown as an average and for the individual quadrants. This gives insight into the overall training as well as possible issues related to how the optimization was formulated.

Figure 6.2 shows the loss for each quadrant individually and the average. With a few individual steps of exception, the controller improves steadily. As expected the minimum does not appear to be near 0 because the architecture of the controller is as simple as it gets. It can be seen that the controller generalizes well to all 4 quadrants of the target plane because the loss for each quadrant is nearly the same over the course of training. The convergence behaviour seems ideal and the minimal bumps, for instance around iteration 23, are arguably negligible.

The speed of convergence seems nearly ideal. The weights are initialized at 0 which is theorized to be the reason for the very first step being slow. Afterwards, however, the training rapidly converges towards the optimum and slows down towards the end due to the gradients becoming weaker. Even though this case is a lot more complex the loss curve is very similar to the simple example from Chapter 5.

For completion, Figure 6.3 shows the weights of the controller over the training. The following numbering of the chambers presented in Figure 3.1 is used: chamber 1 is the top right, chamber 2 top left and chamber 3 the bottom one.

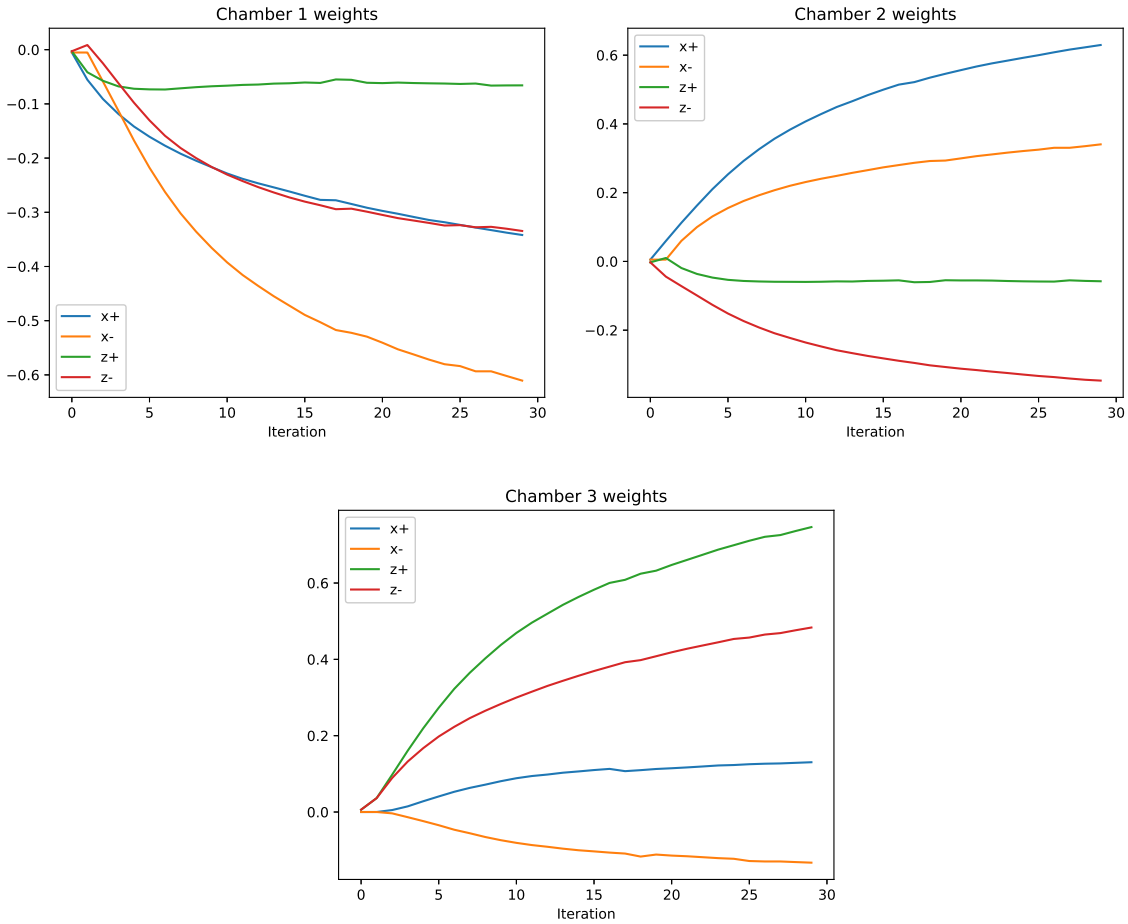


Figure 6.3: Controller weights for the pressure chambers. Chambers 1 and 2 are symmetrical about the $y-z-$plane and it shows in the weights. Only inflating chamber 3 bends in the positive $z-$direction like in the verification case, hence why the weights are easier to interpret. The legend refers to the weights associated to the 4 inputs of Equation 6.1 where $z+$ is the positive $z-$direction etc.

In regards to the training duration, a total of $30 \cdot 4 = 120$ runs of the simulation with gradient calculations were performed in a total of 403 seconds, or about 6 minutes and 43 seconds. This puts the raw average frames per second to about 893 without accounting for any preventable overhead, data saving etc. The reason the performance appears so much worse than during the verification is the gradient calculation and some additional saving and processing procedures.

In particular, the auto-differentiation causes the gradient calculation to be of roughly the same order of computational complexity as the forward simulation, effectively halving the frames per second. Furthermore, there is a little more overhead for saving data each iteration, performing the training etc.

The advantage of using this framework compared to competing approaches in literature is the significantly reduced development time, especially considering

1. a more straightforward development workflow compared to
    - developing an analytical model for analytical controllers
    - Setting up an environment for reinforcement learning and training it using trial-and-error based reward functions
2. with the presented gradient scaling scheme, obtaining functioning gradients does not take much trial-and-error and tuning effort

6.3 Comparison to reinforcement learning

To add weight to these claims 2 comparisons of robots and controllers with similarly complex behaviour and controller structure will be presented here. These are based on reinforcement learning agents and the point is to compare the overall properties of both training schemes qualitatively.

The first benchmark is [100]. The authors developed a reinforcement learning agent with only one hidden layer, which is comparable to the zero-hidden-layer model in this chapter in terms of complexity. The robot is a 3-section honeycomb pneumatic network and has arguably simpler dynamics than the endoscope model due to the honeycomb chambers being a form of inflatable bladder. Because training speed was of concern for the authors the agent was initially trained in a simulation and then deployed to a real robot to continue training. Because of this it's possible to compare only the simulator training session.

Figure 6.4 shows the reward function for said training session. It took over 500 training episodes versus the 30 iterations in the developed framework. The reward function can be interpreted as the inverse of a loss function with some ideal maximum that's approached. Compared to the loss function of the presented controller there are three noticeable differences. First, the large number of steps until some form of convergence is reached. Second, the high frequency fluctuations. And third, the unsteady nature of the improvement with several smaller maxima where the performance seems to decrease over the following episodes, including at the very end.

The high number of episodes necessary is possibly a consequence of the other two points. The high frequency fluctuations is the exploratory nature of reinforcement learning. Because the model is formulated as a sequence of actions taken at every time step the model learns by choosing random ones if it's uncertain and ends up performing worse if bad choices are made. This gives rise to the small fluctuations between episodes. The unsteady improvement is also a consequence of it because not all weight changes of the controller are necessarily correct at every episode.

The second benchmark [101] is a similar reinforcement learning agent but with a Deep Q Network, meaning it's more complex with seven hidden layers in this case. The robot is a cable-

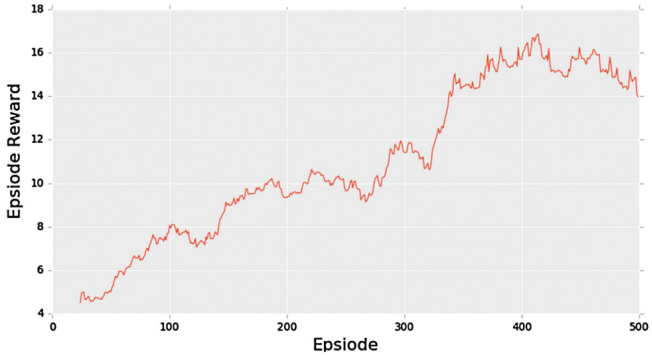
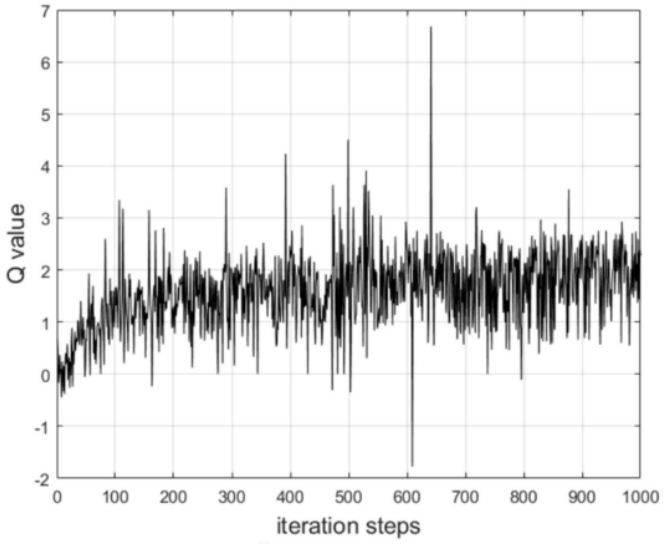


Figure 6.4: Q learning reward function over the course of training in the simulator from [100]. With a comparable controller complexity it takes vastly more steps to reach optimum and the trial-and-error natures introduces high frequency fluctuations which only on average over several iterations improve the performance.

driven tentacle capable of bending in 2D. It is trained in a neural-network-based simulation that learned the dynamics of the robot before training. The reward over the training as well as the robot is visible in Figure 6.5. Even though the structure of the robot is different and the controller more complex the issues mentioned before are virtually identical with fluctuations during training and slow and arguably somewhat unreliable convergence.

(a) Tentacle robot of [101]



(b) Reward function over training from [101]. Similar to the previous learning agent there is strong fluctuations and slow convergence

Figure 6.5: The robot and reward function of the reinforcement learning agent.

Based on these graphs some arguments for and against both training approaches can be made. Training with analytical gradients seems to require about an order of magnitude fewer iterations and the training convergence is much smoother. This comes at the disadvantage of not being able to explore the environment. The slower and less steady training of the reinforcement learning agent is caused by exploration and thus is likely to find a better optimum. Gradient descent is, at least with this formulation, only able to find the closest local optimum, which may not be sufficient for more complex training setups. Hence one may argue that the pure gradient training may not converge to a sufficient optimum. There are some potential solutions to this problem. By exploiting the fast convergence and simulation speed it should be possible to use

particle swarm optimization [102]. This means initializing the weights in a variety of ways and letting several starting points converge to their local optima and then moving all particles towards this point in hopes of finding a sufficient solution. The problem here is that the number of initial solutions grows exponentially with the number of weights in the network, meaning it is not applicable to very complex controllers.

A much better remedy is not viewing pure gradient learning and reinforcement learning as opposites because reinforcement learning also involves finding correct gradients to adjust the weights of the network. While this method has not been used here it should be possible to exploit the availability of analytical gradients and apply them to a reinforcement learning agent to reduce the number of iterations required. This is because the trial-and-error attempts of the agent can be interpreted as a form of calculating numerical gradients by adjusting the weights randomly. Hence the analytical gradients can give insight into the direction of the closest local optimum while still allowing the agent to explore without having to try various random microsteps. Future researchers are encouraged to explore this option.

As a last point, there does not seem to be much quantitative data on total controller development time with competing methods. Judging from the fact that entire papers are dedicated to developing a single controller for specific robots the author claims that this framework could potentially reduce the total development time of soft robotic controllers by several orders of magnitude due to the setup and convergence speed in the simulation when pure gradient descent training is used.

7 Meta-Optimization

This chapter is dedicated to performing speed-targeted meta-optimization and its results. Section 7.1 will explain how the problem is formulated and implemented. Section 7.2 shows the results in a number of plots and interprets them.

7.1 Optimization design

The problem will be approached by using multiple damping factors β from Equation 2.17, each responsible for a 50ms time-window of the simulation. The simulation conditions will be identical to the example of Section 5.3. Except this time it will start with the final optimized pressure of 3.61kPa and 25% fewer maximum steps, meaning the total duration is 25% shorter. This means the total simulation duration went from 4000 steps = 2 seconds to 3000 steps = 1.5 seconds. Now at the end of the simulation the optimization goal of Equation 5.6 is not reached yet. The goal is to optimize the damping factors for each time-window to make the model reach the target despite the shorter duration. To ensure that the damping is not reduced to hit the target at the final step by fluctuating heavily the loss function is slightly adapted.

$$L = \sum_{t=0.99 \cdot t_{\max}}^{t_{\max}} \sum_{k=1}^{n} (x_{k,t,z} - x_{k,0,z} + 10)^2 + (\| v_{k,t} \| \cdot 0.1)^2 \tag{7.1}$$

It now sums the values up over the last percent of time of the simulation and adds the squared magnitude of the velocity to the loss. The scaling of 0.1 makes the weight of both terms roughly the same for the loss. This means the simulation is incentivised to stand still at the target position at the end of the simulation.

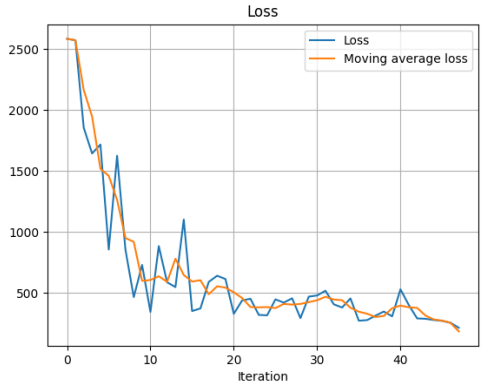
7.2 Results

Similar to Section 5.3 the presented plots are the loss function, a few exemplary β's from different time-windows and the scaled gradients.

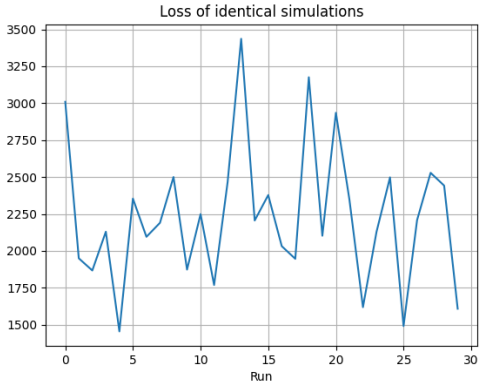
Figure 7.1a shows the loss function and a moving average of it. As can be seen the loss sinks in a much less steady way compared to the previous optimizations. This is theorized to be caused by the fact that changing the damping will slightly change the dynamics. This potentially ends the simulation in a higher loss state (further away from target from a wobbling motion for instance) even though the analytical gradient suggests a lower loss. This is substantiated by the fact that on average the loss consistently sinks at a rate similarly fast to the previous chapters as visible in the moving average. This problem could potentially be solved by a more robust formulation of the problem / loss function, as this is arguably the simplest approach possible. To substantiate this point Figure 7.1b shows the loss of several consecutive simulations runs without adjusting the damping. There is significant variance even though all parameters are identical for all runs, meaning the loss function is not well-behaved for this use case.

That the training worked similarly well as in previous chapters despite the rough loss function can be seen from Figure 7.2a. The damping factors all converge to different target values and shoot right towards them. The shape is very similar to the pressure in Figure 5.2 and the weights in Figure 6.3. The scaled gradients indicate the same as the magnitude of the gradients drops close to zero towards the end.

The final loss is 215, which may give the impression that the goal was not nearly reached. This is a misconception caused by the loss function compared to before. This function loops over 30 time steps and adds the velocity term of the 22 nodes of the end surface, which causes it to be significantly higher despite being quite close to the optimum. Still, it may be argued that the



(a) Loss function and moving average for the meta-optimization. The function is significantly less smooth compared to the optimizations in the previous chapters.



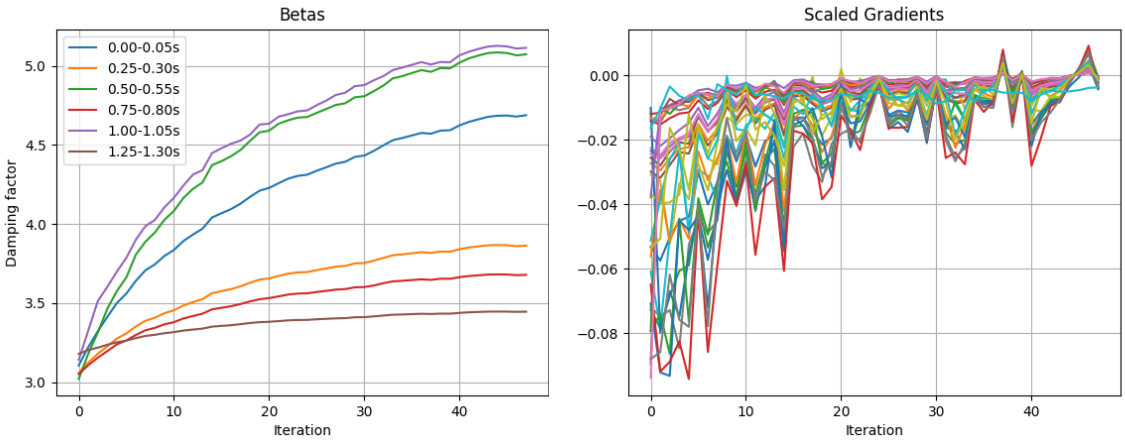
(b) Loss of several consecutive runs without optimization with identical parameters. There are significant differences caused by the ill-behaved loss function.

Figure 7.1: Loss over optimization and for several runs with identical parameters.

simplicity of the problem formulation caused the optimum to be this high and a different form of meta-optimization could give better results.

The loss function and damping factors can be interpreted using information from Chapter 5 and some knowledge regarding the dynamics of the model. The initially high loss function is a combination of the velocity and position error at the end. The given pressure caused the model to reach the target position about half a second later than in this 25% shorter run, meaning at the final time-step it has not quite reached the goal and moves towards it. The damping factors Figure 7.2a are harder to interpret because different time-windows adjust towards seemingly arbitrary factors without any obvious order. This is theorized to be caused by the model trying to affect the settling motion / dynamics such that it moves quicker to reach the target but increase the damping at other points to decrease the final velocity, meaning the individual factors are not ordered intuitively.

Overall, these results imply that the generality of the framework can be exploited to perform meta-optimization. The formulation of the loss function was arguably sub-optimal and it is expected that a more robust approach to either the loss function or meta-optimization as a whole could both improve the convergence and minimum of the loss.



(a) Exemplary β's for certain time-windows over the course of optimization. Notice how their magnitude is not sorted by their time because different points in time require different damping for reaching the target

(b) scaled gradients for all β's. The high number makes individual identification impossible but the weakening magnitude due to reaching the optimum is still visible.

Figure 7.2: A few β's over the course of optimization and the scaled gradients for all

8 Conclusions

Soft robotic actuators are a rapidly growing area of research for medical and manufacturing purposes due to their natural shape adaptation and wide range of possible motions. These properties, however, introduce difficulties in the development of optimal shapes and controllers because their physical behaviour is hard for humans to intuitively predict. This research focused on the development of a general optimization framework with analytical gradients to find a generally working approach to optimal actuator and controller development. The main research question of the thesis was

"How can a general optimization framework with analytical gradients for pressure-driven soft robotic actuators be modelled and implemented?"

This overarching question was split into multiple sub-questions. The first was in regard to modelling the soft robot simulation:

"How accurate is the developed simulation compared to established FEM tools in regards to the predicted deformation?"
Solutions to modelling the forces applied to pressure-driven soft actuators were initially found in a literature review. Two different approaches to modelling the internal forces of hyperelastic bodies have been implemented and verified using Abaqus. The first method is by viewing the edges of the tetrahedral mesh as damped springs. The second one involves calculating the Cauchy stress of each element and applying the resulting forces to each node. The verification shows that both approaches produce deformed shapes roughly similar to Abaqus but the cauchy-stress formulation deviates by only up to 7.2% even for larger deformation while the displacement error of the spring-mesh is up to 15.7% and did not successfully converge at all for one verification case. Additionally, the dependency of the tetrahedralization resolution on one test case was graphed and showed a steady decrease in accuracy with lower resolution but very good results for higher resolutions. The verification also showed that the simulation is faster than Abaqus. Running the entire developed program was about 80% faster than only submitting and running the FEM job in Abaqus, meaning the model- and simulation-setup process in Abaqus is excluded in the comparison.

The second sub-question intended to find a solution to the problem of exploding gradients in large-time-horizon auto-differentiation:

"How can the issue of exploding/vanishing gradients from large time horizons for auto-differentiation be addressed?"
This question has been addressed by analysing the mathematical background of auto-differentiation regarding numerical integration. By finding the general expression of the differentiated Euler-integration, it was discovered that scaling down the velocity term during differentiation the entire gradient of that time step should be scaled down accordingly, regardless of the loss function and variable to differentiate (Equation 5.4). Consequently the exploding gradients should disappear because the link in the chain of the derivatives are scaled down to avoid explosion. Because this produces gradients of arbitrary order of magnitude, an additional normalization scheme has been proposed (Equation 5.5), which semi-automatically scales the resulting gradients to appropriates sizes with a single user defined variable. The approach has been shown to produce meaningful gradients in a test case for optimization the pressure input to a cuboid mesh with the intention of bending it downwards by 10mm.

The third sub-question applied the framework to controller development as a proof of concept and compared the results with methods from the literature:

"What are the effects of gradient-based training on the speed and reliability of convergence for a soft-robotic controller for an endoscope model?"
To answer this question, a simple feed-forward neural network controller for an exemplary endoscope model with 3 pressure chambers has been trained. The network consisted of only the input and output layer and used the target position in the 2D-plane of the main axis of the endoscope as inputs. The results show a steady and smooth learning behaviour until the closest optimum is reached. The weights of the network are shown to directly move towards their ideal values without fluctuations or bumps. Two cases of reinforcement learning for similarly complex models are used as a comparison.

The most important observations are the much higher number of training iterations it takes for them to converge. The developed framework took 30 iterations to fully converge while the most similar comparison took 500. Additionally, the convergence behaviour is much less steady. This causes the reward function to constantly fluctuate slightly and even lets it fall into local minima several times over the course of training. These results indicate that the developed approach to optimization frameworks with analytical gradients could significantly improve both development and training time for controllers.

The final sub-question analysed the effects of optimizing the performance of the underlying simulation rather than the robot or controllers:

"What are the effects of performance-targeted meta-optimization of the simulation? Meaning how can the optimization be applied to the speed of the simulation itself?"
The approach consisted of assigning different damping factors to each 50ms time-window of the simulation and optimizing them to reach the same target position as in the example used for optimizing the input pressure in the second question. The target was a speed increase of 25% percent by reducing the number of max steps by 25%. For this purpose the loss function has been adjusted to incorporate the velocity of the end nodes to ensure the target position is steady. The results show that the damping factors converged into a variety of different values and the goal was reached approximately. While the convergence of the damping factors imply a nearly optimal optimization behaviour the loss function turned out to be ill-behaved. The sub-optimal formulation caused significant variance in the calculated loss from numerical inaccuracy. This resulted in a very jittery loss function over the course of optimization and is expected to negatively affect the optimum and convergence. It is theorized that a more robust approach to meta-optimization could allow larger speed ups and more reliable convergence. While the results are arguably sub-optimal in terms of convergence this at least confirms the ability to perform meta-optimization by exploiting the generality of the framework.

In conclusion, the developed framework has been shown to be able to produce meaningful gradients for different use cases. The issue of exploding gradients is solved conditionally such that a user can easily tune the gradient-scale to usable sizes. The framework appears capable of optimizing both individual parameters of the simulation as well as neural network based controllers with comparatively few iterations and smooth learning behaviour. Furthermore, it was shown that the generality of the framework can be used to improve itself through a meta-optimization of the damping intended to reach the final deformation state quicker.

8.1 Limitations

The main limitations of the developed framework are related to the underlying simulation, in particular:

- Element inversion: The simulation is not robust against element inversion. This is a quite severe limitations in terms of robustness because very fast inflation and movements can quickly cause this. The author theorizes that adding this could improve not only the convergence but even potentially allow for larger time steps and easier damping tuning, be-

cause the velocity of the vertices is less likely to e.g. invert elements or become unstable otherwise.

- Memory allocation: Taichi requires at least all positions and velocities to be saved in a large array covering all nodes and time-steps for auto-differentiation. Because the author did not particularly focus on optimizing memory usage, there are more variables saved in memory than necessary for easier access during development. This memory limits the resolution and size of the used meshes.

- Performance: While the GPU kernels of Taichi in conjunction with the explicit integration scheme are fast compared to established FEM tools, the author did not focus on speed during development. It is very likely that the performance could be improved significantly when fully analyzing the implementation.

- Collision: The simulation does not support any form of collision. This is a rather strong limitation as many forms of pressure-driven soft actuators are based on self-collision. Furthermore, it is necessary to implement continues collision detection with time-of-impact calculations to get functioning gradients because otherwise the discontinues nature of collisions does not work with auto-differentiation.

- Multi-material: Using multiple materials for soft robots could massively increase the possible range of motion as different stiffnesses for different parts of a given robot severely impact its dynamics. This feature was not implemented in this study but is expected to be trivial to implement, as the ability to define a different material for each finite element could be included quickly.

- Interaction with other frameworks: The developed neural network controller was implemented manually by the author rather than through an established library. While it is theorized that it is possible to combine them, this has not been shown.

- Anisotropy: The current implementation only supports isotropic materials. This may not be a problem for soft robots made using casting method, but once other materials are introduced to change the dynamics or the robot is 3D printed, with anisotropy from fused deposition modelling for instance, this becomes a major limitation.

- Real-world effects: Real soft robots succumb to a number of additional non-linearities that are non-trivial to implement in a simulation. Some examples include internal friction and hysteresis. These effects are expected to prevent a direct transition from any simulation result to the real world.

8.2 Recommendations

Based on the findings in this thesis several recommendations for future research can be made.

- Validation: Currently the simulation is verified through Abaqus but an experimental validation is still necessary to confirm the usability of the framework.

- Other domains: This framework was intended for soft robotic actuators, but the gradient scaling scheme is expected to generalize to other explicit integration simulations. This means that, for instance, simulations for electromagnetics, rigid bodies, fluids, or any other domain could utilize this scheme and implement similar optimization frameworks. Naturally, multi-physics implementations are possible as well.

- Practical controllers: Future researchers are encouraged to train a more complex controller with realistic use cases using the methods explored here and report both the training behaviour as well as total development time.

- Topology optimization: A big use-case of this framework could be topology optimization. Unfortunately, the author did not have the time to implement a sophisticated topology optimization scheme within the framework. One way could be mesh generation using voxel-algorithm through a density function defined by the input mesh and optimizing the density for each cell to perform a certain motion or produce certain forces.
- Better Meta-optimization: While the meta-optimization in the thesis arguably worked the formulation of the problem was sub-optimal and succumbed to numerical inaccuracies. It is theorized that a different approach could improve both the maximum possible speed up as well as the convergence behaviour.
- Automatic gradient scaling: The proposed scheme seems to work quite reliably but still requires the user to do some trial-and-error tuning of parameters to find good values. It may be possible to adaptively tune these parameters without user input by making the program run a few test iterations and tune itself.
- Model predictive control: Because the simulation turned out to be able to run faster than real-time it may be possible to use the heavily parallelized modelling approach through CUDA kernels for predictive controllers.
- Better tetrahedralization: While it has been shown that tetgen can produce sufficient meshes it may be argued that there are better alternatives. Specifically, being able to obtain very regular tetrahedral meshes like in Abaqus could improve the accuracy without sacrificing too much simulation speed.

A Replicating results

This appendix outlines the process to replicate the results shown in the thesis. The necessary source code is available to RaM members at https://git.ram.eemcs.utwente.nl/impact/pneumatic_endoscope/topology_optimization. In addition to the following explanation the reader is encouraged to read the short documentation here to better understand the underlying code. The main file "Simulation.py" in the root of the repo also contains elaborate comments to explain the purpose of most variables and functions.

A.1 Verification

The repo contains a folder called "verification" which contains a copy of the main simulation file called "Verification_Simulation.py". It can be used to generate all files that produce the verification results. The file requires the user to set the mesh, pressures and volume constraint as in Chapter 4 and saves the data used for the graphs in the folder. The file needs to be run several times with all the different conditions used in the chapter.

A.1.1 3D plots

The file "Verification.py" computes the interpolation values used as the colormap in the 3D plots. It requires the files from the simulation runs to exist. The 3D views are generated using "verification_plots.py" and the user needs to set which mesh to view below line 77.

A.1.2 Graph

The graphs of Figure 4.5 stem from "Verification_graph.py" and it requires the results from the simulation to exist in the folder as before. Because the graph was only made for a single verification case no changes to the file are necessary.

A.2 Exploding Gradients / Pressure optimization

The repo contains a folder "pressure_opt" containing a copy of the main simulation file and a plotting file. First run the simulation and then the plotting file.

A.3 Controller synthesis

Similar to the pressure optimization there is a folder "controller" with a pre-configured simulation file and plotting file. The simulation file produces all the files necessary for the plotting file.

A.4 Meta-optimization

The folder "meta_opt" contains a pre-configured simulation file and plotting files. As before, the simulation file needs to be run first.

Bibliography

[1] M. Cianchetti, C. Laschi, A. Menciassi, and P. Dario, "Biomedical applications of soft robotics," *Nature Reviews Materials*, vol. 3, 05 2018.

[2] M. Runciman, A. Darzi, and G. P. Mylonas, "Soft robotics in minimally invasive surgery," *Soft robotics*, vol. 6, no. 4, pp. 423–443, 2019.

[3] M. Cianchetti, T. Ranzani, G. Gerboni, T. Nanayakkara, K. Althoefer, P. Dasgupta, and A. Menciassi, "Soft robotics technologies to address shortcomings in today's minimally invasive surgery: the stiff-flop approach," *Soft robotics*, vol. 1, no. 2, pp. 122–131, 2014.

[4] A. Kumar, "Methods and materials for smart manufacturing: Additive manufacturing, internet of things, flexible sensors and soft robotics," *Manufacturing Letters*, vol. 15, pp. 122–125, 2018. Industry 4.0 and Smart Manufacturing.

[5] P. Polygerinos, N. Correll, S. A. Morin, B. Mosadegh, C. D. Onal, K. Petersen, M. Cianchetti, M. T. Tolley, and R. F. Shepherd, "Soft robotics: Review of fluid-driven intrinsically soft devices; manufacturing, sensing, control, and applications in human-robot interaction," *Advanced Engineering Materials*, vol. 19, no. 12, p. 1700016, 2017.

[6] A. Diodato, M. Brancadoro, G. De Rossi, H. Abidi, D. Dall'Alba, R. Muradore, G. Ciuti, P. Fiorini, A. Menciassi, and M. Cianchetti, "Soft robotic manipulator for improving dexterity in minimally invasive surgery," *Surgical innovation*, vol. 25, no. 1, pp. 69–76, 2018.

[7] S. Yim and M. Sitti, "Shape-programmable soft capsule robots for semi-implantable drug delivery," *IEEE Transactions on Robotics*, vol. 28, no. 5, pp. 1198–1202, 2012.

[8] D. Son, H. Gilbert, and M. Sitti, "Magnetically actuated soft capsule endoscope for fine-needle biopsy," *Soft robotics*, vol. 7, no. 1, pp. 10–21, 2020.

[9] P. Polygerinos, Z. Wang, K. C. Galloway, R. J. Wood, and C. J. Walsh, "Soft robotic glove for combined assistance and at-home rehabilitation," *Robotics and Autonomous Systems*, vol. 73, pp. 135–143, 2015.

[10] H. K. Yap, J. H. Lim, F. Nasrallah, and C.-H. Yeow, "Design and preliminary feasibility study of a soft robotic glove for hand function assistance in stroke survivors," *Frontiers in neuroscience*, vol. 11, p. 547, 2017.

[11] C.-Y. Chu and R. M. Patterson, "Soft robotic devices for hand rehabilitation and assistance: a narrative review," *Journal of neuroengineering and rehabilitation*, vol. 15, no. 1, pp. 1–14, 2018.

[12] P. Polygerinos, K. C. Galloway, E. Savage, M. Herman, K. O'Donnell, and C. J. Walsh, "Soft robotic glove for hand rehabilitation and task specific training," in *2015 IEEE international conference on robotics and automation (ICRA)*, pp. 2913–2919, IEEE, 2015.

[13] T. Bützer, O. Lambercy, J. Arata, and R. Gassert, "Fully wearable actuated soft exoskeleton for grasping assistance in everyday activities," *Soft Robotics*, vol. 8, no. 2, pp. 128–143, 2021.

[14] E. T. Roche, M. A. Horvath, I. Wamala, A. Alazmani, S.-E. Song, W. Whyte, Z. Machaidze, C. J. Payne, J. C. Weaver, G. Fishbein, *et al.*, "Soft robotic sleeve supports heart function," *Science translational medicine*, vol. 9, no. 373, 2017.

[15] H. Naghibi, P. A. Costa, and M. Abayazid, "A soft robotic phantom to simulate the dynamic respiratory motion of human liver," in *2018 7th IEEE international conference on biomedical robotics and biomechatronics (Biorob)*, pp. 577–582, IEEE, 2018.

[16] M. Manti, T. Hassan, G. Passetti, N. D'Elia, C. Laschi, and M. Cianchetti, "A bioinspired soft robotic gripper for adaptable and effective grasping," *Soft Robotics*, vol. 2, no. 3, pp. 107–116, 2015.

[17] C. Balaguer, "Nowadays trends in robotics and automation in construction industry: Transition from hard to soft robotics," in *Proceedings of International Symposium on Automation and Robotics in Construction*, Citeseer, 2004.

[18] C. Laschi, J. Rossiter, F. Iida, M. Cianchetti, and L. Margheri, *Soft Robotics: Trends, Applications and Challenges*. Springer, 2017.

[19] B. Mazzolai and M. Cianchetti, "Soft robotics: Technologies and systems pushing the boundaries of robot abilities," *Science Robotics*, vol. 1, 2016.

[20] J. Guo, Y. Sun, X. Liang, J.-H. Low, Y.-R. Wong, V. S.-C. Tay, and C.-H. Yeow, "Design and fabrication of a pneumatic soft robotic gripper for delicate surgical manipulation," in *2017 IEEE International Conference on Mechatronics and Automation (ICMA)*, pp. 1069–1074, IEEE, 2017.

[21] A. D. Marchese, C. D. Onal, and D. Rus, "Autonomous soft robotic fish capable of escape maneuvers using fluidic elastomer actuators," *Soft robotics*, vol. 1, no. 1, pp. 75–87, 2014.

[22] M. Xiloyannis, L. Cappello, D. B. Khanh, S.-C. Yen, and L. Masia, "Modelling and design of a synergy-based actuator for a tendon-driven soft robotic glove," in *2016 6th IEEE International Conference on Biomedical Robotics and Biomechatronics (BioRob)*, pp. 1213–1219, IEEE, 2016.

[23] M. Cianchetti, M. Follador, B. Mazzolai, P. Dario, and C. Laschi, "Design and development of a soft robotic octopus arm exploiting embodied intelligence," in *2012 IEEE International Conference on Robotics and Automation*, pp. 5271–5276, IEEE, 2012.

[24] N. R. Sinatra, C. B. Teeple, D. M. Vogt, K. K. Parker, D. F. Gruber, and R. J. Wood, "Ultra-gentle manipulation of delicate structures using a soft robotic gripper," *Science Robotics*, vol. 4, no. 33, 2019.

[25] G. Rateni, M. Cianchetti, G. Ciuti, A. Menciassi, and C. Laschi, "Design and development of a soft robotic gripper for manipulation in minimally invasive surgery: a proof of concept," *Meccanica*, vol. 50, no. 11, pp. 2855–2863, 2015.

[26] M. Hofer and R. D'Andrea, "Design, modeling and control of a soft robotic arm," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 1456–1463, IEEE, 2018.

[27] L. Margheri, C. Laschi, and B. Mazzolai, "Soft robotic arm inspired by the octopus: I. from biological functions to artificial requirements," *Bioinspiration & biomimetics*, vol. 7, no. 2, p. 025004, 2012.

[28] J. Zou, Y. Lin, C. Ji, and H. Yang, "A reconfigurable omnidirectional soft robot based on caterpillar locomotion," *Soft robotics*, vol. 5, no. 2, pp. 164–174, 2018.

[29] R. Baumgartner, A. Kogler, J. M. Stadlbauer, C. C. Foo, R. Kaltseis, M. Baumgartner, G. Mao, C. Keplinger, S. J. A. Koh, N. Arnold, *et al.*, "A lesson from plants: High-speed soft robotic actuators," *Advanced Science*, vol. 7, no. 5, p. 1903391, 2020.

[30] U. Culha and F. Iida, "Enhancement of finger motion range with compliant anthropomorphic joint design.," *Bioinspiration & biomimetics*, vol. 11 2, p. 026001, 2016.

[31] F. Plum, S. Labisch, and J.-H. Dirks, "Sauv—a bio-inspired soft-robotic autonomous underwater vehicle," *Frontiers in neurorobotics*, vol. 14, p. 8, 2020.

[32] X.-Y. Zhang, Z.-H. Lu, S.-Y. Wu, and Y.-G. Zhao, "An Efficient Method for Time-Variant Reliability including Finite Element Analysis," *Reliability Engineering and System Safety*, vol. 210, no. C, 2021.

[33] R. J. Lapeer, P. D. Gasson, and V. Karri, "A hyperelastic finite-element model of human skin for interactive real-time surgical simulation," *IEEE Transactions on Biomedical Engineering*, vol. 58, no. 4, pp. 1013–1022, 2010.

[34] G. Kluth and B. Després, "Discretization of hyperelasticity on unstructured mesh with a cell-centered lagrangian scheme," *Journal of Computational Physics*, vol. 229, no. 24, pp. 9092–9118, 2010.

[35] M. Mansouri, H. Darijani, and M. Baghani, "On the correlation of fem and experiments for hyperelastic elastomers," *Experimental mechanics*, vol. 57, no. 2, pp. 195–206, 2017.

[36] O. Amir and O. Sigmund, "On reducing computational effort in topology optimization: how far can we go?," *Structural and Multidisciplinary Optimization*, vol. 44, no. 1, pp. 25–29, 2011.

[37] O. Sigmund and K. Maute, "Topology optimization approaches," *Structural and Multidisciplinary Optimization*, vol. 48, no. 6, pp. 1031–1055, 2013.

[38] M. Raeisinezhad, N. G. Pagliocca, B. Koohbor, and M. Trkov, "Design optimization of a pneumatic soft robotic actuator using model-based optimization and deep reinforcement learning," *Frontiers in Robotics and AI*, vol. 8, p. 107, 2021.

[39] S. Nalbach, R. M. Banda, S. Croce, G. Rizzello, D. Naso, and S. Seelecke, "Modeling and design optimization of a rotational soft robotic system driven by double cone dielectric elastomer actuators," *Frontiers in Robotics and AI*, vol. 6, p. 150, 2020.

[40] C.-H. Liu and C.-H. Chiu, "Optimal design of a soft robotic gripper with high mechanical advantage for grasping irregular objects," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 2846–2851, IEEE, 2017.

[41] H. Zhang, M. Y. Wang, F. Chen, Y. Wang, A. S. Kumar, and J. Y. H. Fuh, "Design and development of a soft gripper with topology optimization," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 6239–6244, 2017.

[42] B. Caasenbrood, A. Pogromsky, and H. Nijmeijer, "A computational design framework for pressure-driven soft robots through nonlinear topology optimization," in *2020 3rd IEEE International Conference on Soft Robotics (RoboSoft)*, pp. 633–638, 2020.

[43] X.-S. Yang, "Firefly algorithms for multimodal optimization," in *International symposium on stochastic algorithms*, pp. 169–178, Springer, 2009.

[44] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[45] K. Svanberg, "The method of moving asymptotes—a new method for structural optimization," *International journal for numerical methods in engineering*, vol. 24, no. 2, pp. 359–373, 1987.

[46] 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*, vol. 11. 06 2012.

[47] F. Largilliere, V. Verona, E. Coevoet, M. Sanz-Lopez, J. Dequidt, and C. Duriez, "Real-time control of soft-robots using asynchronous finite element modeling," in *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 2550–2555, 2015.

[48] M. Nesme, Y. Payan, and F. Faure, "Efficient, physically plausible finite elements," 08 2005.

[49] M. Skouras, B. Thomaszewski, P. Kaufmann, A. Garg, B. Bickel, E. Grinspun, and M. Gross, "Designing inflatable structures," *ACM Transactions on Graphics*, vol. 33, pp. 1–10, 07 2014.

[50] J. Degrave, M. Hermans, J. Dambre, *et al.*, "A differentiable physics engine for deep learning in robotics," *Frontiers in neurorobotics*, vol. 13, p. 6, 2019.

[51] X. Lin, H. Zhang, and A. M. Rappe, "Optimization of quantum monte carlo wave functions using analytical energy derivatives," *The Journal of Chemical Physics*, vol. 112, no. 6, pp. 2650–2654, 2000.

[52] A. Treuille, A. McNamara, Z. Popović, and J. Stam, "Keyframe control of smoke simulations," in *ACM SIGGRAPH 2003 Papers*, pp. 716–723, 2003.

[53] T. George Thuruthel, Y. Ansari, E. Falotico, and C. Laschi, "Control strategies for soft robotic manipulators: A survey," *Soft Robotics*, vol. 5, 01 2018.

[54] E. Franco, A. G. Casanovas, F. Rodriguez y Baena, and A. Astolfi, "Model based adaptive control for a soft robotic manipulator," in *2019 IEEE 58th Conference on Decision and Control (CDC)*, pp. 1019–1024, 2019.

[55] T. G. Thuruthel, E. Falotico, F. Renda, and C. Laschi, "Model-based reinforcement learning for closed-loop dynamic control of soft robotic manipulators," *IEEE Transactions on Robotics*, vol. 35, no. 1, pp. 124–134, 2018.

[56] A. Gupta, C. Eppner, S. Levine, and P. Abbeel, "Learning dexterous manipulation for a soft robotic hand from human demonstrations," in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 3786–3793, IEEE, 2016.

[57] J. M. Bern, Y. Schnider, P. Banzet, N. Kumar, and S. Coros, "Soft robot control with a learned differentiable model," in *2020 3rd IEEE International Conference on Soft Robotics (RoboSoft)*, pp. 417–423, 2020.

[58] I. Newton and G. Leibniz, "Euler method,"

[59] F. Harewood and P. McHugh, "Comparison of the implicit and explicit finite element methods using crystal plasticity," *Computational Materials Science*, vol. 39, no. 2, pp. 481–494, 2007.

[60] P. Hammer, N. Vasilyev, D. Perrin, P. del Nido, and R. Howe, "Fast image-based model of mitral valve closure for surgical planning," 11 2021.

[61] Y. Duan, W. Huang, H. Chang, W. Chen, J. Zhou, S. K. Teo, Y. Su, C. K. Chui, and S. K. Y. Chang, "Volume preserved mass–spring model with novel constraints for soft tissue deformation," *IEEE Journal of Biomedical and Health Informatics*, vol. 20, pp. 268–280, 2016.

[62] M. M. Attard and G. W. Hunt, "Hyperelastic constitutive modeling under finite strain," *International Journal of Solids and Structures*, vol. 41, no. 18-19, pp. 5327–5350, 2004.

[63] A. V. Gelder, "Approximate simulation of elastic membranes by triangulated spring meshes," *Journal of Graphics Tools*, vol. 3, no. 2, pp. 21–41, 1998.

[64] M. Sasso, G. Palmieri, G. Chiappini, and D. Amodio, "Characterization of hyperelastic rubber-like materials by biaxial and uniaxial stretching tests based on optical methods," *Polymer Testing*, vol. 27, no. 8, pp. 995–1004, 2008.

[65] M. Nesme, Y. Payan, and F. Faure, "Efficient, physically plausible finite elements," 08 2005.

[66] "Section 12 volume elements." https://academic.csuohio.edu/duffy_s/CVE_512_12.pdf. Accessed: 2021-11-15.

[67] K. Huebner, D. Dewhirst, D. Smith, and T. Byrom, *The Finite Element Method for Engineers*. A Wiley-Interscience publication, Wiley, 2001.

[68] A. Franus, S. Jemioło, and A. Marek, "A slightly compressible hyperelastic material model implementation in abaqus," *Engineering Solid Mechanics*, 03 2020.

[69] G. Irving, J. Teran, and R. Fedkiw, "Tetrahedral and hexahedral invertible finite elements," *Graphical Models*, vol. 68, no. 2, pp. 66–89, 2006.

[70] N. Elango and A. Faudzi, "A review article: investigations on soft materials for soft robot manipulations," *The International Journal of Advanced Manufacturing Technology*, vol. 80, no. 5, pp. 1027–1037, 2015.

[71] C. Horgan and G. Saccomandi, "Constitutive models for compressible nonlinearly elastic materials with limiting chain extensibility," *Journal of Elasticity*, vol. 77, pp. 123–138, 11 2004.

[72] W. Mollemans, F. Schutyser, J. Van Cleynenbreugel, and P. Suetens, "Tetrahedral mass spring model for fast soft tissue deformation," in *International Symposium on Surgery Simulation and Soft Tissue Modeling*, pp. 145–154, Springer, 2003.

[73] K. Ding and L. Ye, "3 - simulation methodology," in *Laser Shock Peening* (K. Ding and L. Ye, eds.), Woodhead Publishing Series in Metals and Surface Engineering, pp. 47–72, Woodhead Publishing, 2006.

[74] P. J. Schneider and D. Eberly, *Geometric Tools for Computer Graphics*. USA: Elsevier Science Inc., 2002.

[75] P. Martins, R. Natal Jorge, and A. Ferreira, "A comparative study of several material models for prediction of hyperelastic properties: Application to silicone-rubber and soft tissues," *Strain*, vol. 42, no. 3, pp. 135–147, 2006.

[76] G. Van Rossum *et al.*, "Python programming language.," in *USENIX annual technical conference*, vol. 41, p. 36, 2007.

[77] Y. Hu, L. Anderson, T.-M. Li, Q. Sun, N. Carr, J. Ragan-Kelley, and F. Durand, "Difftaichi: Differentiable programming for physical simulation," 2020.

[78] NVIDIA, P. Vingelmann, and F. H. Fitzek, "Cuda, release: 10.2.89," 2020.

[79] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, pp. 357–362, Sept. 2020.

[80] "numpy-stl." https://github.com/WoLpH/numpy-stl. Accessed: 2021-11-16.

[81] "Pathlib - object-oriented filesystem paths." https://docs.python.org/3/library/pathlib.html. Accessed: 2021-11-15.

[82] M. Musy, G. Jacquenot, G. Dalmasso, neoglez, R. de Bruin, A. Pollack, F. Claudi, C. Badger, icemtel, B. Sullivan, D. Hrisca, D. Volpatto, N. Schlömer, Z.-Q. Zhou, and ilorevilo, "marcomusy/vedo: 2020.4.2," Nov. 2020.

[83] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[84] H. Si, "Tetgen, a delaunay-based quality tetrahedral mesh generator," *ACM Trans. Math. Softw.*, vol. 41, feb 2015.

[85] M. Szilvśi-Nagy and G. Matyasi, "Analysis of stl files," *Mathematical and computer modelling*, vol. 38, no. 7-9, pp. 945–960, 2003.

[86] R. J. Rost, *OpenGL(R) Shading Language (2nd Edition)*. Addison-Wesley Professional, 2005.

[87] M. Smith, *ABAQUS/Standard User's Manual, Version 6.9*. United States: Dassault Systèmes Simulia Corp, 2009.

[88] P. Martins, R. Natal Jorge, and A. Ferreira, "A comparative study of several material models for prediction of hyperelastic properties: Application to silicone-rubber and soft tissues," *Strain*, vol. 42, pp. 135–147, 08 2006.

[89] Z. Zhao, S. Kuchnicki, R. Radovitzky, and A. Cuitino, "Influence of in-grain mesh resolution on the prediction of deformation textures in fcc polycrystals by crystal plasticity fem," *Acta materialia*, vol. 55, no. 7, pp. 2361–2373, 2007.

[90] L. B. Rall and G. F. Corliss, "An introduction to automatic differentiation," *Computational Differentiation: Techniques, Applications, and Tools*, vol. 89, 1996.

[91] C. C. Margossian, "A review of automatic differentiation and its efficient implementation," *Wiley interdisciplinary reviews: data mining and knowledge discovery*, vol. 9, no. 4, p. e1305, 2019.

[92] B. Hanin, "Which neural net architectures give rise to exploding and vanishing gradients?," *arXiv preprint arXiv:1801.03744*, 2018.

[93] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," in *International conference on machine learning*, pp. 1310–1318, PMLR, 2013.

[94] G. Philipp, D. Song, and J. G. Carbonell, "The exploding gradient problem demystified-definition, prevalence, impact, origin, tradeoffs, and solutions," *arXiv preprint arXiv:1712.05577*, 2017.

[95] A. Kag, Z. Zhang, and V. Saligrama, "Rnns evolving on an equilibrium manifold: A panacea for vanishing and exploding gradients?," *arXiv preprint arXiv:1908.08574*, 2019.

[96] J. Zhang, Q. Lei, and I. Dhillon, "Stabilizing gradients for deep neural networks via efficient svd parameterization," in *International Conference on Machine Learning*, pp. 5806–5814, PMLR, 2018.

[97] P. Eberhard and C. Bischof, "Automatic differentiation of numerical integration algorithms," *Mathematics of Computation*, vol. 68, 11 1997.

[98] A. Lydia and S. Francis, "Adagrad—an optimizer for stochastic gradient descent," *Int. J. Inf. Comput. Sci*, vol. 6, no. 5, 2019.

[99] P. Zhou, J. Feng, C. Ma, C. Xiong, S. Hoi, *et al.*, "Towards theoretically understanding why sgd generalizes better than adam in deep learning," *arXiv preprint arXiv:2010.05627*, 2020.

[100] H. Zhang, R. Cao, S. Zilberstein, F. Wu, and X. Chen, "Toward effective soft robot control via reinforcement learning," in *International Conference on Intelligent Robotics and Applications*, pp. 173–184, Springer, 2017.

[101] Q. Wu, Y. Gu, Y. Li, B. Zhang, S. A. Chepinskiy, J. Wang, A. A. Zhilenkov, A. Y. Krasnov, and S. Chernyi, "Position control of cable-driven robotic soft arm based on deep reinforcement learning," *Information*, vol. 11, no. 6, p. 310, 2020.

[102] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of ICNN'95-international conference on neural networks*, vol. 4, pp. 1942–1948, IEEE, 1995.