Numerical approximation of blow forming hyperelastic 3D printed preforms

Boswerger, M.G.I.

Award date:
2018

SOURCE OF YOUR TECHNOLOGY

Numerical Approximation of Blow Forming Hyperelastic 3D Printed Preforms

Esp 405
5633 AJ Eindhoven
The Netherlands

P +31 (0)40 75 16 116
E info@siouxlime.nl
I www.siouxlime.nl

Marloes Boswerger

Begeleider Technische Universiteit Eindhoven:
B. van der Linden

Begeleiders LIME:
B.J. van der Linden
T. van Opstal

Contents

1 Introduction

1.1 Project Origin

Art Officials is a creative consultancy bureau. One of their projects is called Aera Fabrica, in which hollow 3D printed plastic preforms are heated and inflated to several times their original size. After cooling down, the inflated preform retains its shape. As the final shape can be quite different from its original shape, it is difficult to predict the final shape in advance. Art Officials approached LIME with the question whether they could simulate this process. They wanted to know what preform shape they need to print, given a desired final shape. This is a difficult problem. Just predicting the final shape given the preform shape is not trivial, but would already provide valuable information. Printing preforms can take anywhere from 20 minutes to several days, so a simulation may save a lot of time and effort. Besides, to eventually solve the backward problem, a forward model is needed. In this thesis we will model the inflation process and build a simulation to predict the final shape given a preform shape.

1.2 Blow Forming Process

The current process is as follows. Using CAD software, a preform is designed that can be read and printed by a 3D printer. Each design includes the same nozzle, for attachment to an air compressor. After being printed, the preform is attached to the air compressor and held in warm water at a temperature above the material's glass transition temperature (Tg). When the preform is warm, the preform is removed from the water and the air compressor is turned on. The air pressure causes the preform to inflate and at the same time the material cools down. At some point the inflation stops, and the end form does not shrink back to its original shape when the pressure is removed due to the temperature dependent elasticity of the material. When heated again, the plastic form shrinks back to a form similar (but not exactly equal) to its preform shape.

Even though a lot of shapes can be made, there are limitations to what can be printed. The preform is printed in a process called fused filament fabrication (FFF) or Fused Deposition Modeling (FDM). The filament is a wire of plastic material. This filament is heated in the nozzle of the printer and the molten material is deposited onto a print bed. Layer by layer, the preform is built. In FFF, it is not possible to print shapes that have little support below them. As a rule of thumb, angles may not be less steep than 45 degrees measured from the horizontal axis and without support only a few mm can be bridged. Since the preforms have to be hollow to be inflated, this is a limitation for the shapes. Furthermore, sometimes the printer malfunctions in some way: the nozzle is blocked, the filament is damaged, the print did not stick to the print bed, not enough molten material flows out, etc. This affects the print quality and may mean that the preform contains a hole or a weak spot, where it may break during blow forming. Besides limitations in printing, in the current setting we are not sure the shape has uniform temperature before being inflated. This may mean that the preform does not inflate as intended, and that locations with a higher temperature inflate more. Finally, the pressure is increased by hand after the preform is removed from the water. How much pressure is needed and possible for the preform is a matter of guesswork and feeling, so the strain rate and final pressure are different every time a preform is inflated.

For the present work, we assume that the preform was printed correctly, which means that the preform has no obvious defects. We also ignore the effects of temperature and strain rate of the material during the blow forming process.

1.3 Complicated Materials

To be able to effectively inflate preforms, the material they are made of needs to have the right properties. The first is that the material needs to be available as filament for 3D printing. Ideally, it is hard and not too flexible at room temperature. Furthermore, the material needs to be a thermoplastic, meaning that the material becomes moldable above a certain temperature and solidifies upon cooling. In practice this means that our material can be inflated while above that temperature and does not change shape after cooling when the pressure is removed. This temperature should lie below 100 degrees Celsius, because of the current blow forming process. This means that the glass transition temperature, T_g, should not be too high. This is the temperature below which certain materials are relatively hard and brittle, like glass, and above which they are in a soft, flexible, or rubbery state. Note that this is not the same their melting temperature.

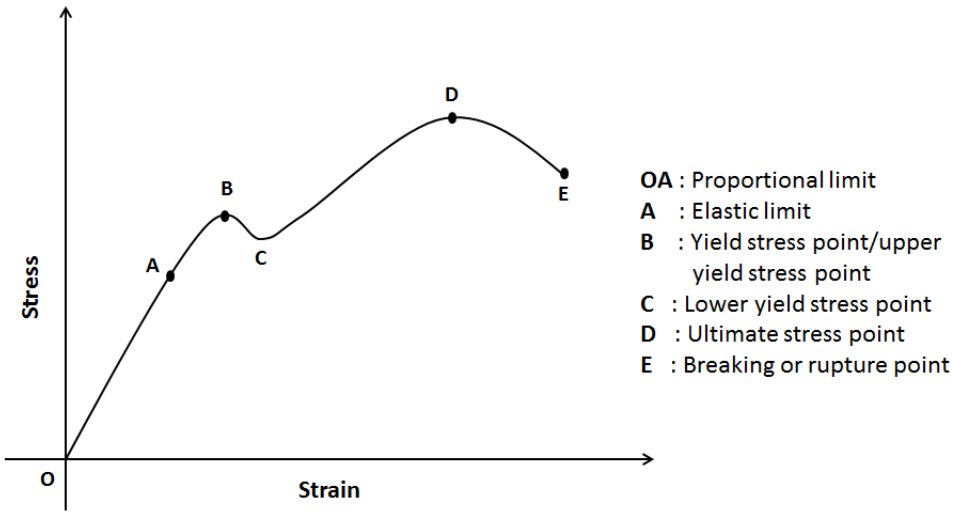


Figure 1 *Example stress-strain curve [1].*

The final property we discuss is its stress-strain curve. See Figure 1 for an example. Young's modulus is the slope of the linear part from the origin to point A. When inflating a preform, the dip at C may be wider. So for little to no extra (or even less) stress, the material still deforms. The part between points C and D is important, it is called strain hardening. Note that it is also possible that the slope of the curve never becomes negative, but merely decreases before strain hardening starts. Furthermore, the strain hardening does not have to be linear. Finally, not the entire curve may be visible in stress strain curves from experiments. The experiment may end before the breaking or rupture point or the material may not even exhibit strain hardening and break at e.g. point B.

To illustrate the usefulness of strain hardening for inflation, we consider two spherical balloons, balloon A and B. Suppose balloon A is made of a material that does not exhibit strain hardening. So after point C on the curve in Figure 1, the curve continues downwards with a negative slope. Furthermore, we suppose balloon B does exhibit strain hardening, and that its stress-strain curve looks exactly like Figure 1. Finally, both balloons have the same weak spot where the material is thinner. When we inflate these two balloons with exactly the same pressure, both will start to inflate most at this weak spot. At some point, the strain passes point C. For balloon A, this means that it will keep inflating at the weak spot until it ruptures there. For balloon B, it means that after this, the weak spot becomes tougher than the rest of the balloon, meaning that now the rest of the balloon inflates as well. So for inflating balloons, strain hardening means that imperfections, such as thinner walls, do not cause the balloon to rupture there as soon as they would without strain hardening. Similarly, for blow forming, strain hardening means that the preform does not rupture as easily due to imperfections which were introduced into the material by 3D printing.

However, Figure 1 is a simplification. In reality, stress is represented by a symmetric tensor, which depends on a tensor that describes the strain. On top of that, real materials complicate matters further, see Section 2.3.2 for more details. Hence, accurately modeling the material properties is a difficult task. We would like to be able to use a data-driven approach, to avoid this task. However, for existing methods, the stress-strain curve is an important property of a material and has a large influence on the inflation process. As such, it is important to take into account in our physical model.

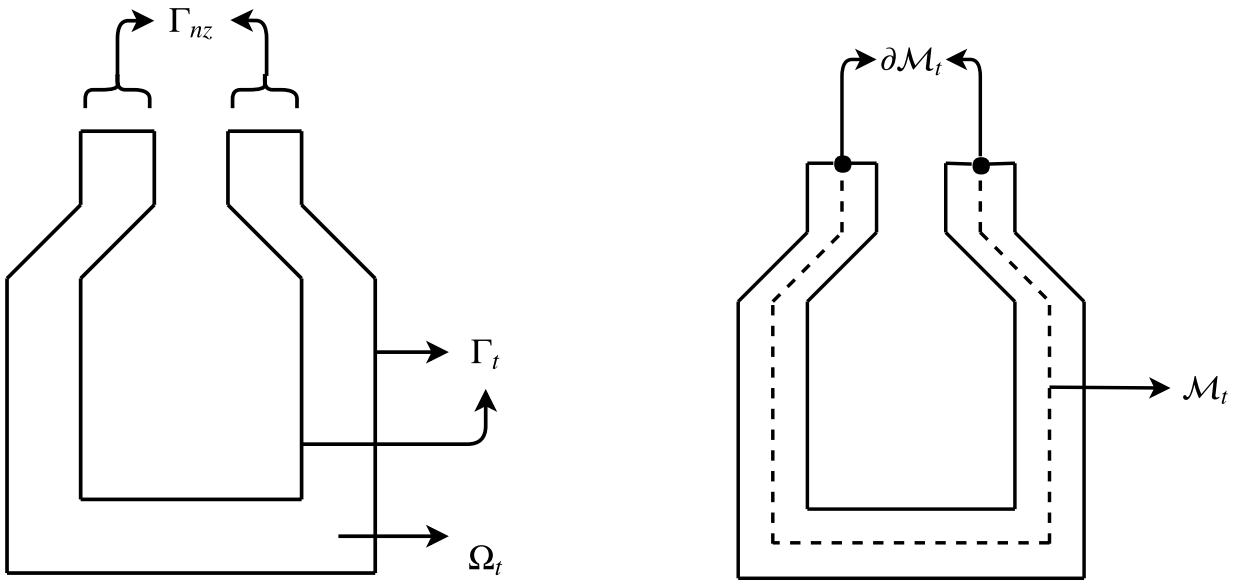
1.4 Summary

In the first part, the physics of the blow forming process and the material models will be explained. There are several material models to capture hyperelastic materials, but their material parameters are generally not available. Furthermore, these parameters depend on e.g. temperature, which is not taken into account by these models.

The first method that is used to simulate blow forming is the Finite Element Method (FEM). While FEM ensures convergence to the solution and certainty about the order of the error, it does not provide any room to use an alternative to the material model, or to easily incorporate experimental data. So with the second method, a Mass-Spring System (MSS), a step towards data-driven models is taken. An MSS is a fast and easy way to simulate a membrane. However, it depends on heuristics, requires too much manual intervention and uses a small number of neighbors.

Smoothed Particle Hydrodynamics (SPH) seems to solve these problems. It has a solid physical basis and at the same time it allows the usage of information about the topology. As the preform is modeled as a manifold of dimension $n-1$ in $\mathbb{R}^n$, the SPH kernel function is modified. SPH shows that a local formulation based on neighbors can work. However, SPH still uses a material model to find the elastic response of the material.

Finally, Neural Networks (NN) are investigated. Using NN to solve partial differential equations proves unsatisfactory, as it is required to train a network per problem. To find out whether NN can effectively extract geometrical information and because an important part of SPH is the distance between particles, a NN is trained to compute the geodesic distance on curves and surfaces. Finally, a formulation to use NN locally to compute the elastic material response is given.



(a) *The original preform and its boundaries.*

(b) *The mid plane and its boundary.*

Figure 2 *Geometry of a preform explained using a cross section of the preform.*

2 Physics of Hyperelastic Blow Forming

2.1 Geometry

We consider a time-dependent, open domain $\Omega_t \subset \mathbb{R}^n,\ n \in \{2,3\}, t \in (0,T)$ with boundary $\partial\Omega_t$. This domain denotes the entire preform. For a preform, the boundary consists of two parts. One is the nozzle, which is fixed, Γ_{nz}. The other part is the rest of the boundary, the moving boundary, Γ_t. See Figure 2a for an illustration. With $\underline{\Omega}$, we denote the initial configuration, before inflation.

Let $\underline{\mathcal{M}}$ denote the mid-plane of the initial configuration $\underline{\Omega}$. The boundary $\partial\underline{\mathcal{M}}$ is equal to the part of $\underline{\mathcal{M}}$ coinciding with Γ_{nz}: $\partial\underline{\mathcal{M}} = \underline{\mathcal{M}} \cap \Gamma_{nz}$. See Figure 2b for an illustration. We note that Γ_{nz} may be empty: $\Gamma_{nz} = \emptyset$, so then $\underline{\mathcal{M}}$ does not have a boundary. Of course, this is not true in practice, as all preforms have a nozzle, but using a domain without boundary has its advantages, especially in SPH. Let $\mathcal{M}_t$ denote the configuration at time t and $\mathbf{x}$ a point on $\mathcal{M}_t$, so $\mathbf{x} : (0,T) \to \mathbb{R}^n$. $\mathbf{x}(t)$ is the position of material point $\mathbf{X} \in \underline{\mathcal{M}}$ at time t. We abbreviate a point on $\mathcal{M}_t$ configuration as $\mathbf{x}(t) = \mathbf{x}_t$. As we are interested in a plane, we may work with a parametrization: $\mathbf{s} : S \to \mathcal{M}_t$ of the mid-plane, where $S \subset \mathbb{R}^{n-1}$. We do the same for the reference configuration $\underline{\mathbf{s}} : S \to \underline{\mathcal{M}}$, such that $\mathbf{x}_t$ is equal to the composition $\mathbf{s}_t \circ \underline{\mathbf{s}}^{-1}$. We define $S := \mathbf{s}_t^{-1}(\mathcal{M}_t)$. Regarding notation, domains which are underlined denote the initial, reference, or undeformed domain. Often, the subscript t is dropped, and we refer to $\mathcal{M}$ as the current configuration.

2.2 Governing Equation

During inflation, we only take two kinds of physical forces into account. The first kind is pressure forces, due to the air pressure being higher on the inside than on the outside. The second kind is elastic forces, due to the deformation of the material. Depending on the method we use (FEM, MSS, SPH), different formulations of the same problem are convenient. Essentially, we always try to find a solution where the pressure and elastic forces cancel, resulting in static equilibrium.

The conservation equation for momentum for solid mechanics problems is [2]:

$$\ddot{\mathbf{x}} = \frac{1}{\rho}\nabla \cdot \mathbf{P} + \mathbf{F}_e/m, \tag{1}$$

where ρ is the density, m is the mass, $\mathbf{P}$ is the first Piola-Kirchhoff stress tensor, and $\mathbf{F}_e$ are external forces, like the pressure forces. We assume that the pressure is uniform and works perpendicular to the surface. $\mathbf{P}$ is calculated using the deformation gradient and the second Piola-Kirchhoff stress tensor. The second Piola-Kirchhoff stress tensor is denoted by $\mathbf{S}$ and is a function of the deformation. How $\mathbf{S}$ is computed depends on the choice of the constitutive equation for the behavior of the material. See Section 2.3.1 for more details on material models. The deformation gradient tensor is

$$\mathbf{F} = \frac{d\mathbf{x}}{d\mathbf{X}} = \frac{d(\mathbf{u} + \mathbf{X})}{d\mathbf{X}} = \frac{d\mathbf{u}}{d\mathbf{X}} + \mathbf{I}, \tag{2}$$

where $\mathbf{x}$ and $\mathbf{X}$ denote the current and reference configuration, respectively. The first Piola-Kirchhoff stress tensor is:

$$\mathbf{P} = \mathbf{F}\mathbf{S}. \tag{3}$$

For the FEM analysis, the governing equation is formulated in terms of the potential energy, which also determines the constitutive behavior of the material. However, it still comes down to an equilibrium of pressure and elastic forces. See Section 3 for the formulation. Finally, for the MSS analysis, the tensor formulation is a bit excessive. We can reduce Equation (1) to

$$\ddot{\mathbf{x}} = \mathbf{F}_e/m, \tag{4}$$

where $\mathbf{F}_e$ is the sum of pressure and spring forces. See Section 4 for the computation of these forces.

2.3 Material Models

In the previous section, we use the second Piola-Kirchhoff stress tensor to describe the stress in the material due to a given deformation. There are a couple of material models to compute $\mathbf{S}$. Of course, the choice for a material model should be based on observations about the material we are actually modeling. So in the following section we state some models for (hyper)elastic materials, after which we discuss the actual material properties of the preforms.

2.3.1 Hyperelastic Material Models

The first Piola-Kirchhoff stress tensor is used to compute the second Piola-Kirchhoff stress tensor. It can be defined in terms of the Green-Lagrange strain tensor, or more commonly, by defining the strain energy density function $\Psi(I_1, I_2, I_3)$ using invariants of the right or left Cauchy-Green (also called Finger) deformation tensor. The right Cauchy-Green deformation tensor and Green-Lagrange strain are, respectively

$$\mathbf{C} = \mathbf{F}^T\mathbf{F}, \tag{5}$$

$$\mathbf{E} = \frac{1}{2}(\mathbf{C} - \mathbf{I}). \tag{6}$$

The first three invariants of $\mathbf{C}$ are [3]

$$I_1 = \mathrm{Tr}(\mathbf{C}), \quad I_2 = \frac{1}{2}\left(I_1^2 - \mathrm{Tr}(\mathbf{C}^T\mathbf{C})\right), \quad I_3 = \det(\mathbf{C}). \tag{7}$$

Sometimes, the principal stretches are used. They are denoted by $\lambda_1, \lambda_2, \lambda_3$ and are the square root of the eigenvalues of $\mathbf{C}$. The principal stretches are related to the invariants in the following way:

$$I_1 = \lambda_1^2 + \lambda_2^2 + \lambda_3^2, \quad I_2 = \lambda_1^2\lambda_2^2 + \lambda_2^2\lambda_3^3 + \lambda_3^2\lambda_1^2, \quad I_3 = \lambda_1^2\lambda_2^2\lambda_3^2. \tag{8}$$

Having obtained the strain energy density function $\Psi(I_1, I_2, I_3)$, the second Piola-Kirchhoff stress is

$$\mathbf{S} = \frac{\partial\Psi}{\partial\mathbf{E}} = 2\frac{\partial\Psi}{\partial\mathbf{C}} = 2\frac{\partial\Psi}{\partial I_1}\frac{\partial I_1}{\partial\mathbf{C}} + 2\frac{\partial\Psi}{\partial I_2}\frac{\partial I_2}{\partial\mathbf{C}} + 2\frac{\partial\Psi}{\partial I_3}\frac{\partial I_3}{\partial\mathbf{C}}. \tag{9}$$

There are many models for isotropic elastic materials. When the stress-strain relationship depends on the deformation they are called hyperelastic. Usually, these models are nonlinear. A non-exhaustive list follows here, generally in order of publication.

1. Saint Venant-Kirchhoff (SVK) [4, p.130]:

$$\mathbf{S} = \lambda \operatorname{Tr}(\mathbf{E})\mathbf{I} + 2\mu\mathbf{E}, \tag{10}$$

where λ and μ are material parameters known as the Lamé parameters. SVK is a linear hyperelastic model and is valid for small deformations. It is also easy to compute and uses just two material parameters.

2. The Mooney model (1940) [5]

$$\Psi(I_1, I_2) = c_1(I_1 - 3) + c_2(I_2 - 3), \tag{11}$$

where c_1 and c_2 are material parameters. It was formulated in terms of principal stretches by Mooney and Rivlin formulated it in terms of invariants [3, 6]. Hence, it is sometimes called the Mooney-Rivlin model. It is valid up to moderate strains.

3. The Generalized Mooney-Rivlin Model (1948) [3, 6]

$$\Psi(I_1, I_2) = \sum_{i=0,j=0}^{\infty} c_{ij}(I_1 - 3)^i(I_2 - 3)^j, \tag{12}$$

where c_{ij} are material parameters and $c_{00} = 0$. It is valid for moderate strains. Many models are actually simple cases of the Generalized Mooney-Rivlin model. In practice, the infinite sum is cut off depending on the required accuracy. Its main drawback is the high number of material parameters.

4. The Neo-Hookean model (1948)[6]:

$$\Psi(I_1) = C_1(I_1 - 3) = \frac{1}{2}NKT(I_1 - 3), \tag{13}$$

where C_1 is a material constant, N is the chain density of the polymer per unit volume, K is the Boltzmann constant and T is the absolute temperature. It is valid for low and moderate strains.

5. The Hart-Smith model [7] (1966) actually defines the *derivatives* of the strain energy density function:

$$\frac{\partial\Psi(I_1, I_2)}{\partial I_1} = Ge^{k_1(I_1-3)^2}, \quad \frac{\partial\Psi(I_1, I_2)}{\partial I_2} = G\frac{k_2}{I_2}, \tag{14}$$

where k_1, k_2 and G are material parameters. It is capable of reproducing the deformation regimes seen in Figure 4.

6. The Ogden Model (1972) [8]

$$\Psi(\lambda_1, \lambda_2, \lambda_3) = \sum_{i=1}^{N} \frac{\mu_i}{\alpha_i}(\lambda_1^{\alpha_i} + \lambda_2^{\alpha_i} + \lambda_3^{\alpha_i} - 3), \tag{15}$$

where μ_i and α_i are material parameters. They should fulfill two conditions: $\mu_i\alpha_i > 0$ and $\sum \mu_i\alpha_i = \mu$, where μ is the second Lamé parameter (or shear modulus). It is often used for large strains and shows good agreement with experimental data.

Many models contain a factor with a -3 in it somewhere. This is due to the fact that in an undeformed domain, $I_1 = I_2 = 3$ and Ψ should be zero. Also, the material parameters are sometimes available for a certain material, but usually have to be determined experimentally.

It should be noted, that the above is only valid for three dimensions. If the spatial dimension is two, the invariants are

$$I_1 = \operatorname{Tr}(\mathbf{C}), \quad I_2 = \det(\mathbf{C}). \tag{16}$$

However, the material models listed, except for Saint Venant-Kirchhoff, above assume the spatial dimension is three. In the FEM analysis, we encounter $d = 2$, see Section 3. However, it assumes the Saint Venant-Kirchhoff material model, so it does not cause any trouble there.

If the spatial dimension is three, but we model the preform as a surface, the material parameters need to be adjusted. Normally, they are in the same units as the stress, N/m^2. However, in a surface stress is actually in N/m, because we ignore the thickness direction. To incorporate this, we have to multiply the material parameters with the thickness of the preform.

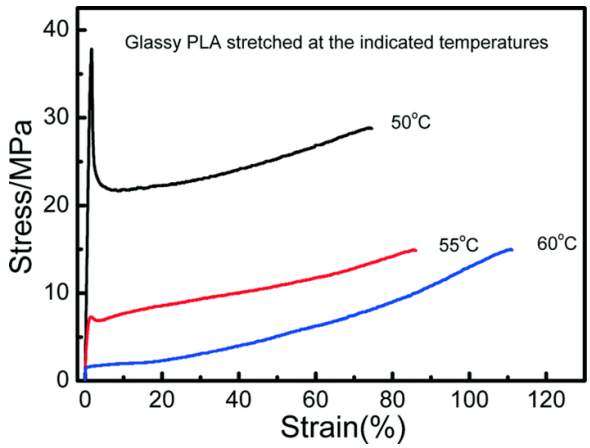
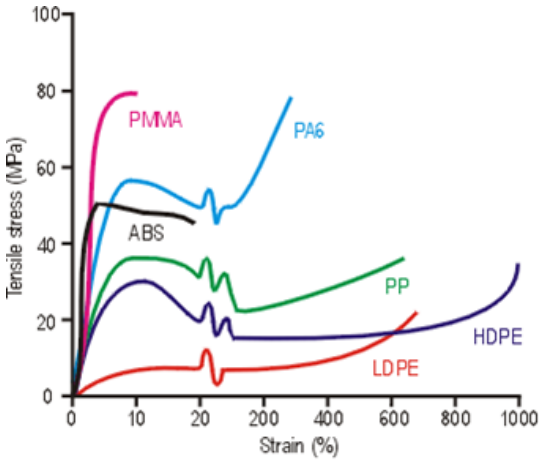
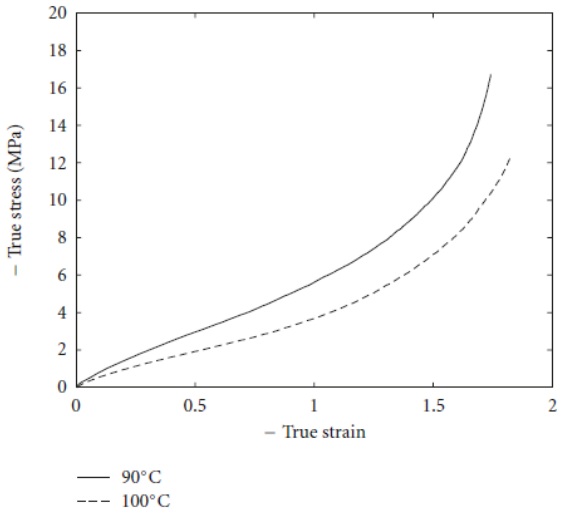


Figure 3 *Stress-strain curve for PLA at temperatures below T_g [9].*



(a) *Stress-strain curve for several plastics [10].*



(b) *Stress-strain curve for PET at two different temperatures [11].*

Figure 4 *Stress-strain curves.*

2.3.2 Real Materials

The material used before LIME got involved in the project was poly lactid acid (PLA). When the temperature of a piece of PLA is around its glass transition temperature, it can be seen that PLA is an elastic material: after deformation it returns to its original size and shape. Furthermore, by looking at its stress-strain curve in Figure 3, we see that the relationship between stress and strain is non-linear. We also see that PLA exhibits a little strain hardening, but it may not be enough to prevent the material from rupturing after starting to inflate at a weak point. This means that PLA might not be the best thermoplastic for the blow forming process. Hence, we also investigated other thermoplastics. See Figure 4a for the stress-strain curves of PMMA, PA6, ABS, PP, HDPE and LDPE and Figure 4b for PET. PMMA and ABS do not have strain hardening and are thus not suitable for our application. The other materials do have strain hardening and are candidates for the blow forming process. While PET does not exhibit a clear linear part, nor a dip in the stress, it does exhibit strain hardening and is already used in blow molding processes. So it looks like a suitable candidate.

We summarize the properties of the discussed materials in Table 1. Note that these are not all available thermoplastics. Some have been excluded because of unfavorable properties (for example very high temperature resistance or release of toxic fumes when melted).

Material	Thermoplastic	Available as filament	Strain hardening	T_g [°C]
PMMA (polymethyl methacrylate)	✓	✓	×	85 – 165
ABS (Acrylonitrile butadiene styrene)	✓	✓	×	88 – 128
PA (polyamide)	✓	✓	✓	44 – 56
PLA (polylactic acid)	✓	✓	?	60 – 65
PC (Polycarbonate)	✓	✓	✓	142 – 205
POM (Polyoxymethylene)	✓	✓	×	-18 – -8
PEEK (Polyetherether ketone)	✓	✓	×	143 – 199
PE (Polyethylene)	✓	✓	?	-25 – -15
PET (Polyethylene terephthalate)	✓	✓	✓	68 – 80
PP (Polypropylene)	✓	✓	?	-25 – -15
PS (Polystyrene)	✓	✓	×	74 – 110
PVC (Polyvinyl chloride)	✓	✓	×	75 – 105

Table 1 *Table containing relevant material properties [12]. A ✓ means that the material has the property and a × means that it does not. ? means that sources are conflicted on whether or not that property is present.*

We also carried out a couple of experiments with PP, PLA, PET and PA. The conclusion was that PLA was by far the easiest material to print with. PLA also performed well even with holes in the preform. PET looked promising, but seemed to have a slightly too high T_g to be effectively inflated in room temperature. PP and PA were difficult to print as well as inflate. From a practical perspective, we conclude from the experiments that PLA seems to be the best choice of material with the current process.

Finally, Table 1 lists materials that are *classes* of polymers, and generally do not represent one single material. So even though we have been able to find some information about strain hardening, it might not represent every material in this class well. Furthermore, there are many suppliers of filament [14, 15], which all have their own variations on the same materials [16, 17], or even their own variants within a certain plastic [16, 18]. All with their own material properties. Even the color of the material effects ultimate tensile strength, yield strength and maximum strain [19]. Besides these differences, batches of the same material bought at one supplier also differ [20]. The way PLA is stored and air humidity [21], and the age [22] influence it before and after printing. Finally, the conditions under which the material is printed, like nozzle temperature, printing speed, and layer height effect the final properties [19, 23].

Another difficulty, is that the material models need material parameters that are to be determined experimentally. While typical values or ranges for quantities like Poisson's ratio and Young modulus are available for most materials, they are measured at room temperature. So to obtain these parameters at the temperatures that the material is used, we would have to carry out some experiments or find it in literature, latter of which is difficult to find. What we could find is shown in Figure 5: not only is Poisson's ration of HDPE temperature dependent, but also strain rate dependent. Neither of these two effects is taken into account in our physical model.

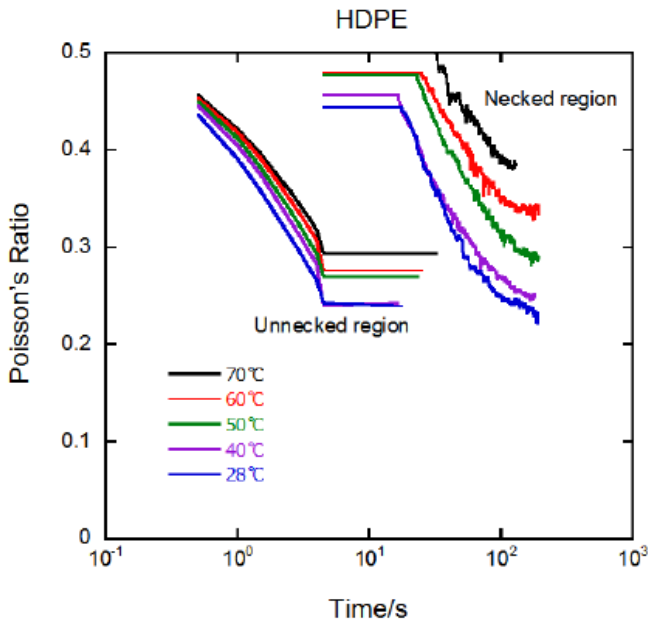


Figure 5 *Temperature dependence of Poisson's ratio of a HDPE specimen plotted under elongation time for a constant elongation speed. [13].*

3 A FEM Approach to Solving Blow Forming

In this section we simulate inflation of a preform using FEM. We start with FEM, because it is often used to simulate deformation of elastic solids [24, 25]. As an example related to inflation, Skouras et al. compute a balloon that, when inflated, approximates a target shape as closely as possible [26]. They use a FEM formulation and show the stress-strain curves for several material models and their fit with experimental data.

We consider the classical Kirchhoff-Love shell model, which uses the Saint Venant-Kirchhoff material law for the constitutive behavior. The fundamental underlying assumption is that internal loads transverse to a reference surface are negligible. This assumption causes the normal and director to coincide. In the present work, the model will merely be posed and not derived. For more details, for instance on the parametrization dependence, see [27] and references therein. We use an univariate beam model to show some simple results.

In this section we use a different notation from the rest of the thesis, because the flexural rigidity is not easily expressed in terms of the Green-Lagrange strain tensor. Indices i and j take on values $0, \dots, n-1$ and indices denoted with Greek letters range from 0 to $n-2$. We also adhere to the Einstein summation convention. Finally, indices following a comma denote derivatives. Tildes are used to indicate that quantities are in parameter domain.

3.1 Internal Energy Formulation

Following the derivation in [27, pp. 22-23], we obtain an expression containing the potential energy on the left hand side and pressure forces on the right hand side:

$$\tilde{W}'(\mathbf{s}; \tilde{\rho}) = \tilde{F}(t; \tilde{\rho}), \tag{17a}$$

with initial condition

$$\mathbf{s}_0 = \tilde{\chi}_0 := \chi_0 \circ \underline{\mathbf{s}}. \tag{17b}$$

In this case we also drop the t subscript. Finally, we also have a boundary condition:

$$\mathbf{s} = \underline{\mathbf{s}} \text{ on } \partial\underline{\Omega}. \tag{17c}$$

The expression for the internal energy of a Kirchhoff-Love shell is [27]

$$\begin{aligned} \tilde{W}(\mathbf{s}) &:= \int_{\underline{\Omega}} \tilde{\Xi}^{\alpha\beta\gamma\delta} \left(\tilde{\varepsilon}_{\alpha\beta}(\mathbf{s}) \tilde{\varepsilon}_{\gamma\delta}(\mathbf{s}) + \epsilon \tilde{\kappa}_{\alpha\beta}(\mathbf{s}) \tilde{\kappa}_{\gamma\delta}(\mathbf{s}) \right) \circ \underline{\mathbf{s}}^{-1} d\mu, \\ &= \int_{\Sigma} \tilde{\Xi}^{\alpha\beta\gamma\delta} \left(\tilde{\varepsilon}_{\alpha\beta}(\mathbf{s}) \tilde{\varepsilon}_{\gamma\delta}(\mathbf{s}) + \epsilon \tilde{\kappa}_{\alpha\beta}(\mathbf{s}) \tilde{\kappa}_{\gamma\delta}(\mathbf{s}) \right) |\underline{\mathbf{z}}| d\mu, \end{aligned} \tag{18}$$

where the membrane and bending strains are

$$\tilde{\varepsilon}_{\alpha\beta}(\mathbf{s}) := \frac{1}{2} (\underline{\mathbf{s}}_{i,\alpha} \underline{\mathbf{s}}_{i,\beta} - \mathbf{s}_{i,\alpha} \mathbf{s}_{i,\beta}), \tag{19a}$$

$$\tilde{\kappa}_{\alpha\beta}(\mathbf{s}) := \underline{\mathbf{s}}_{i,\alpha} \underline{\nu}_{i,\beta} - \mathbf{s}_{i,\alpha} \nu_{i,\beta}. \tag{19b}$$

The membrane strain describes the behavior of the mid-plane of the surface we are interested in, while the bending strain describes the behavior resulting from the thickness direction. The director $\tilde{\nu}$ is given by

$$\tilde{\nu}_i := \mathbf{z}_i / |\mathbf{z}|, \tag{20}$$

where

$$\mathbf{z}_i := \epsilon_{ijk} \mathbf{s}_{j,0} \mathbf{s}_{k,1}, \text{ with } \epsilon_{ijk} = (i-j)(j-k)(k-i)/2, \tag{21}$$

and

$$\tilde{\Xi}^{\alpha\beta\gamma\delta} := \upsilon \underline{a}^{\alpha\beta} \underline{a}^{\gamma\delta} + \frac{1}{2}(1-\upsilon)(\underline{a}^{\alpha\gamma} \underline{a}^{\beta\delta} + \underline{a}^{\alpha\delta} \underline{a}^{\beta\gamma}), \tag{22}$$

in which $\underline{a}^{\alpha\beta}$ is implicitly given by

$$\underline{a}^{\alpha\beta} \underline{\mathbf{s}}_{i,\beta} \underline{\mathbf{s}}_{i,\gamma} = \delta_{\alpha\gamma}. \tag{23}$$

Lastly, υ and ϵ are material parameters, with υ Poisson's ratio and $\epsilon = \frac{\delta^2}{12l^2}$ (where we can take $l = 1$), the flexural rigidity. The definition of ϵ comes from nondimensionalization and δ is the thickness of the shell. Furthermore, $\tilde{\kappa}_{\alpha,\beta}$ can be rewritten as $-\underline{\mathbf{s}}_{i,\alpha\beta}\tilde{\nu}_i + \mathbf{s}_{i,\alpha\beta}\tilde{\nu}_i$ with the product rule.

In (17a) we need $\tilde{W}'$, so assuming existence of the derivative we get

$$\tilde{W}'(\mathbf{s};\tilde{\rho}) = \int_\Sigma \tilde{\Xi}^{\alpha\beta\gamma\delta}\left(\tilde{\varepsilon}_{\alpha\beta}(\mathbf{s})\tilde{\varepsilon}'_{\gamma\delta}(\mathbf{s};\tilde{\rho}) + \epsilon\tilde{\kappa}_{\alpha\beta}(\mathbf{s})\tilde{\kappa}'_{\gamma\delta}(\mathbf{s};\tilde{\rho})\right)|\underline{z}|d\mu, \tag{24}$$

where the variations of the bending and membrane strains are, respectively,

$$\tilde{\varepsilon}'_{\alpha\beta}(\mathbf{s};\tilde{\rho}) := -\frac{1}{2}(\tilde{\rho}_{i,\alpha}\mathbf{s}_{i,\beta} - \mathbf{s}_{i,\alpha}\tilde{\rho}_{i,\beta}), \tag{25a}$$

$$\tilde{\kappa}'_{\alpha\beta}(\mathbf{s};\tilde{\rho}) := \tilde{\rho}_{i,\alpha\beta}\tilde{\nu}_i(\mathbf{s}) + \mathbf{s}_{i,\alpha\beta}\tilde{\nu}'_i(\mathbf{s};\tilde{\rho}). \tag{25b}$$

Finally, to complete (17a) we use the following as load on the surface:

$$\tilde{F}(t,\tilde{\rho}) = -\omega\int_\Sigma |D\mathbf{s}_t|(p\circ\mathbf{s}_t)(\mathbf{n}\circ\mathbf{s}_t)\tilde{\rho}\,d\mu, \tag{26}$$

which is a traction proportional to the pressure p and induces a load in the normal direction $\mathbf{n}$. Note that both $\mathbf{n}$ and p depend explicitly on the current configuration. ω is a nondimensionalization parameter and is chosen to equal $\frac{2(1-\upsilon^2)}{E\delta}$. Lastly, we let χ_0 be the identity, such that $\mathbf{s}_0 = \underline{\mathbf{s}}$.

3.2 Univariate Planar Case

The above equations can be used to get a 2D model [27]. In 3D space, we assume the solution $\mathbf{s}$ to be constant in one direction and hence, we only have to consider the solution in a plane. We align the coordinate axes such that the first two basis vectors span this plane and the third basis vector is normal to it. This way, the solution is zero in the normal direction. Since $\mathbf{s}$ is a vector, we can summarize this by:

$$\mathbf{s}_2 = 0, \quad \mathbf{s}_{i,1} = \delta_{i2}. \tag{27}$$

So $\mathbf{s} = (\mathbf{s}_0 \;\; \mathbf{s}_1 \;\; 0)$. The spatial gradient becomes:

$$D\mathbf{s} = \mathbf{s}_{i,\alpha} = \begin{pmatrix}\mathbf{s}_{0,0} & \mathbf{s}_{1,0} & \mathbf{s}_{2,0}\end{pmatrix} = \begin{pmatrix}\mathbf{s}_{0,0} & \mathbf{s}_{1,0} & 0\end{pmatrix}. \tag{28}$$

The normal to the solution in the plane becomes:

$$\mathbf{z} = \frac{\mathbf{s}_{,0}\times\mathbf{s}_{,1}}{|\mathbf{s}_{,0}\times\mathbf{s}_{,1}|} = \frac{(\mathbf{s}_{0,0},\mathbf{s}_{1,0},0)\times(0,0,1)}{|(\mathbf{s}_{0,0},\mathbf{s}_{1,0},0)\times(0,0,1)|} = \frac{(\mathbf{s}_{1,0},-\mathbf{s}_{0,0},0)}{|(\mathbf{s}_{1,0},-\mathbf{s}_{0,0},0)|}. \tag{29}$$

We only have one parametrization variable, so we can always choose a parametrization such that $|D\underline{\mathbf{s}}| = 1$, and all indices denoted by a Greek letter are actually equal to 0, hence $\underline{a}^{\alpha\beta}$ becomes

$$\begin{aligned}
&\underline{a}^{\alpha\beta}\underline{\mathbf{s}}_{i,\beta}\underline{\mathbf{s}}_{i,\gamma} = \delta_{\alpha\gamma},\\
&\underline{a}^{\alpha\beta}\underline{\mathbf{s}}_{i,\beta}\underline{\mathbf{s}}_{i,\gamma} = 1,\\
&\underline{a}^{\alpha\beta} = (\underline{\mathbf{s}}_{i,\beta}\underline{\mathbf{s}}_{i,\gamma})^{-1},\\
&\underline{a}^{\alpha\beta} = |D\underline{\mathbf{s}}|^{-2} = 1.
\end{aligned}$$

This also reduces the expression for $\tilde{\Xi}^{\alpha\beta\gamma\delta}$:

$$\begin{aligned}
\tilde{\Xi}^{\alpha\beta\gamma\delta} &:= \upsilon\underline{a}^{\alpha\beta}\underline{a}^{\gamma\delta} + \frac{1}{2}(1-\upsilon)(\underline{a}^{\alpha\gamma}\underline{a}^{\beta\delta} + \underline{a}^{\alpha\delta}\underline{a}^{\beta\gamma})\\
&= \upsilon + \frac{1}{2}(1-\upsilon)(2)\\
&= 1.
\end{aligned}$$

When we assume the reference configuration is unstretched and flat $-D^2\underline{\mathbf{s}}_i = 0$. The membrane and bending strains become:

$$\tilde{\varepsilon}_{\alpha\beta}(\mathbf{s}) := \frac{1}{2}(\underline{\mathbf{s}}_{i,\alpha}\underline{\mathbf{s}}_{i,\beta} - \mathbf{s}_{i,\alpha}\mathbf{s}_{i,\beta}) = \frac{1}{2}(1 - |D\mathbf{s}|^2),$$
$$\tilde{\kappa}_{\alpha\beta}(\mathbf{s}) := -\underline{\mathbf{s}}_{i,\alpha\beta}\underline{\tilde{\nu}}_i + \mathbf{s}_{i,\alpha\beta}\tilde{\nu}_i = -D^2\underline{\mathbf{s}}_i\underline{\tilde{\nu}}_i + D^2\mathbf{s}_i\tilde{\nu}_i = \tilde{\nu}\cdot D^2\mathbf{s}.$$

Substituting these into the expression for the internal energy of the Kirchhoff-Love shell, we obtain the univariate planar case:

$$\tilde{W}(\mathbf{s}) = \frac{1}{2}\int_\Sigma (|D\mathbf{s}|^2 - 1)^2 + \epsilon(\tilde{\nu}\cdot D^2\mathbf{s})^2\, d\mu, \tag{31}$$

and its Fréchet derivative

$$\tilde{W}'(\mathbf{s};\tilde{\rho}) = \int_\Sigma (|D\mathbf{s}|^2 - 1)D\mathbf{s}D\tilde{\rho} + \epsilon D^2\mathbf{s}D^2\tilde{\rho}\, d\mu. \tag{32}$$

In the expression for $\tilde{W}'$ a linear approximation of the bending stiffness is taken: $(D^2\mathbf{s})^2 = (\tilde{\nu}\cdot D^2\mathbf{s})^2 + (\tilde{t}\cdot D^2\mathbf{s})^2$, with $\tilde{t}$ the tangential unit vector. The second term is assumed to be negligible.

Putting this together with Equation (26) gives us the variational statement

$$\int_\Sigma (|D\mathbf{s}|^2 - 1)^2 D\mathbf{s}D\tilde{\rho} + \epsilon D^2\mathbf{s}D^2\tilde{\rho}\, d\mu = -\omega\int_\Sigma |D\mathbf{s}|(p\circ\mathbf{s})(n\circ\mathbf{s})\tilde{\rho}\, d\mu, \tag{33a}$$

with initial condition

$$\mathbf{s}_0 = \tilde{\chi}_0 := \underline{\mathbf{s}}. \tag{33b}$$

3.3 FEM Results

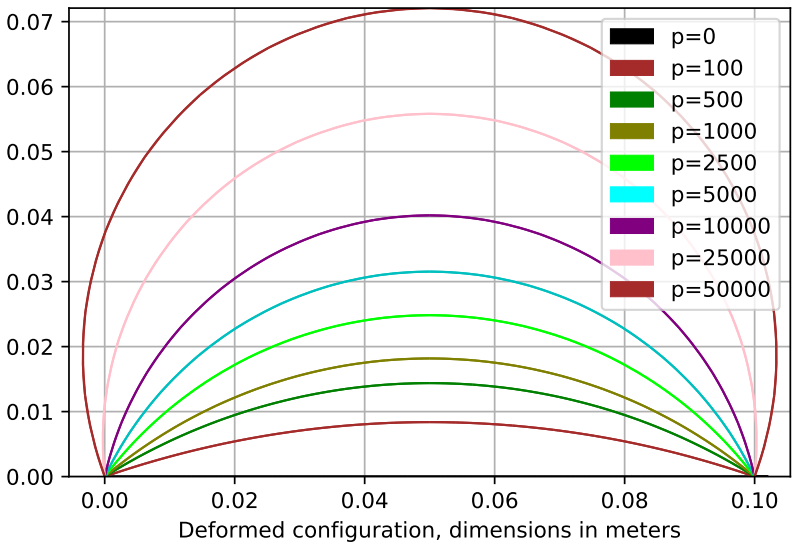


Figure 6 *Deformed configurations for $p \in \{0, 100, 500, 1000, 2500, 5000, 10000, 25000, 50000\}$. The initial configuration is a straight line.*

In computations, we use pseudo time-stepping, as implemented by the nutils package [28]. It is a method that combines Newton and time stepping to solve a nonlinear problem and to find the steady-state solution. The reference configuration, $\underline{\Omega}$, is an arc fixed at both endpoints, such that the displacement at the boundary, $\partial\underline{\Omega}$, in both x and y direction is zero. This causes the boundary of $\underline{\Omega}$ and Ω to coincide.

$$\mathbf{s}(u) = \underline{\mathbf{s}}(u), \text{ for u on } \partial\Omega. \tag{34}$$

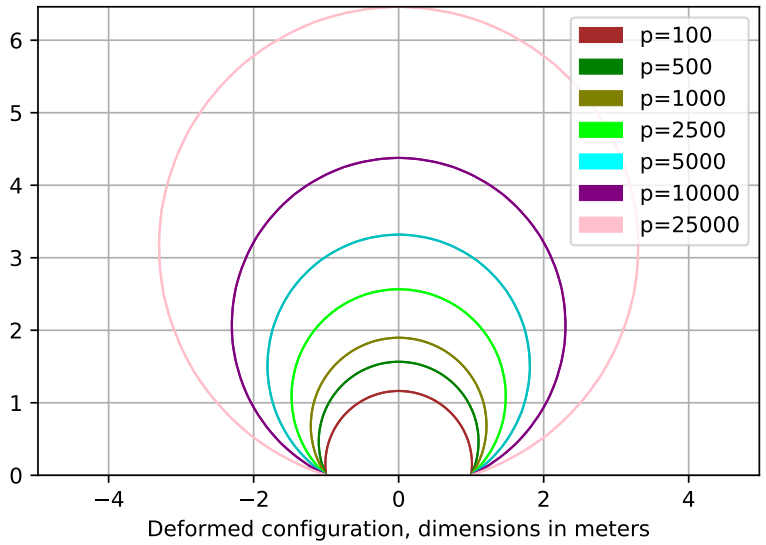


Figure 7 *Deformed configurations for $p \in \{100, 500, 1000, 2500, 5000, 10000, 25000\}$. The initial configuration is half a circle.*

As reference material, we look at a balloon. We use $h = 3 \cdot 10^{-4} m$, giving us a flexural rigidity of $\epsilon = 7.5 \cdot 10^{-9}$, and $\upsilon = 1/2$ and $E = 2 \cdot 10^6$ Pa give us $\omega = p\, 2.5 \cdot 10^{-3}$ for some pressure p in N/m^2.

We start with $d = 2$, where we inflate two different initial configurations: a straight line and a half of a circle. The pressure can be seen as a load on the underside of the line, working upwards. The results for the former can be seen in Figure 6 and the latter in Figure 7.

Besides introducing a load on the surface, we can also carry out a tensile test. For this problem, the reference configuration is a line fixed at the left endpoint and at the right boundary, we prescribe a displacement Δl to the right. So the boundary conditions are

$$\mathbf{s}_0(t) = \underline{\mathbf{s}}_0 = 0, \tag{35}$$
$$\underline{\mathbf{s}}_1 = l, \tag{36}$$
$$\mathbf{s}_1 = l + \Delta l. \tag{37}$$

The stretch is equal to $|D\mathbf{s}|$ and from (33a) we conclude that the (magnitude of the) tension is $|D\mathbf{s}|^2 - 1$. Plotting these against each other after performing tensile tests with several values for d gives us Figure 8. We see that the values from the computations match what is expected from the theory.

While FEM is an often used method, we argue that it is not what we currently need. Convergence to the solution of the partial differential equation is a nice property. However, because our material behavior is difficult to accurately describe with a material model, FEM gives a solution that describes a different material than the one we truly have. The finite element formulation also is not amenable to a data-driven approach, as we can only compute the stresses with the use of a material model. We can, however, still use FEM as a validation tool for other methods to compare results when e.g. the geometry and material model are the same between the two methods.

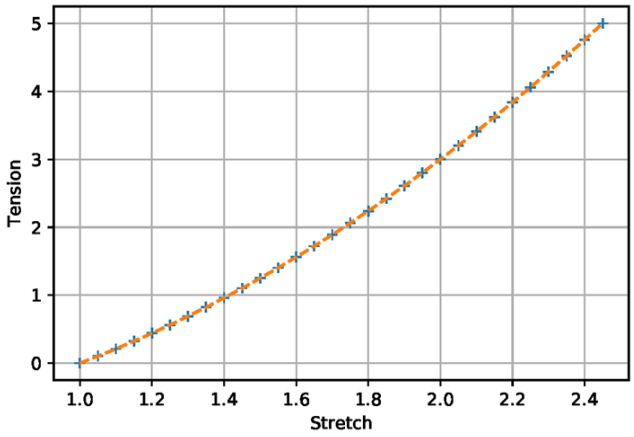


Figure 8 *Tensile test for the univariate Kirchhoff-Love model. The dotted line is the expected stretch from the theory and the crosses are computed values.*

4 Towards Data Driven Models: a Spring System

In order to be able to eventually use data for the solution of the blow forming, we turn to Mass Spring Systems. They use a local formulation and use information about neighboring vertices to compute stresses. MSS have been used to animate deformable surface due to their simplicity and speed [29, 30]. Often, they are used to model cloth [31, 32], sometimes even for real-time applications, e.g., games [33].

We introduce a triangular mesh as discretization of $\underline{\mathcal{M}}$. Let V be the set of vertices in this mesh and E be the set of edges, with each $e \in E$ such that $e = \{u, v\}$, for $u, v \in V$. Similarly, let F be the set of faces, such that each $f \in F : f = \{u, v, w\}$, for $u, v, w \in V$. We assume that the vertices in f are named anticlockwise. We denote the location of vertex v in $\mathbb{R}^3$ by $\mathbf{x}_v$. We call a face or edge adjacent to a vertex, if the vertex is included in one of the edges of that face or is one of the vertices making up the edge. Similary, a face is adjacent to an edge if both of the vertices of the edge are adjacent to the face.

4.1 Physics

We model each edge as a massless spring with a function, $\sigma(\varepsilon) : \mathbb{R} \to \mathbb{R}$, whose graph is based on the stress-strain curve. Given the initial, current and previous position of each vertex, we can compute all forces present in the system, and combined with the mass in each vertex, we can use Verlet integration to find the new position of the grid.

The governing equation, Equation (4), should hold in each vertex, so for one vertex v we have:

$$\ddot{\mathbf{x}}_v(t) = \mathbf{F}_v(\mathbf{x}_v(t))/m_v, \tag{38}$$

where t is the time, $\mathbf{x}_v$ is the position vector, m_v is the mass, and $\mathbf{F}_v$ is a vector, the sum of all forces acting on v, which depends on the location of v and the time. We can rewrite the above equation to:

$$\ddot{\mathbf{x}}_v(t) = \mathbf{A}_v, \tag{39}$$

where $\mathbf{A}_v$ is a vector representing the acceleration of particle v. We will use Verlet integration to discretize time, because it provides good numerical stability and is computationally inexpensive. Let Δt be the time step and let $t_n = n\Delta t$ be the times at which we compute the new position of the grid. Finally, let $\mathbf{x}_v^n$ be the position of vertex v at time step n. To get the second order derivative of $\mathbf{x}_v$, we use the central difference twice:

$$\ddot{\mathbf{x}}_v(t) \doteq \frac{\frac{\mathbf{x}_v^{n+1} - \mathbf{x}_v^n}{\Delta t} - \frac{\mathbf{x}_v^n - \mathbf{x}_v^{n-1}}{\Delta t}}{\Delta t} = \mathbf{A}_v^n, \tag{40}$$

where $\mathbf{A}_v^n$ is the acceleration in vertex v due to forces. Rewriting gives us:

$$\mathbf{x}_v^{n+1} = 2\mathbf{x}_v^n - \mathbf{x}_v^{n-1} + \Delta t^2 \mathbf{A}_v^n. \tag{41a}$$

with

$$\mathbf{x}_v^1 = \mathbf{x}_v^0 = \mathbf{X}_v. \tag{41b}$$

In the following, the superscript n is only used when it is needed to distinguish between time steps. Most quantities need to be recomputed every time step, so it is assumed they are at time step n and the notation is dropped.

As we need all forces in the system, we start with the pressure forces. Let p be the magnitude of the pressure and a the area of the surface where the pressure works on, then the pressure force is

$$\mathbf{F}_p = pa\mathbf{n}. \tag{42}$$

We can only compute the pressure forces on faces, but we need it in vertices. Hence, we assign areas of faces to vertices and use those areas to compute the pressure forces in the vertices. Let a_f be the area belonging to face f, then

$$a_v = \sum_{f \in N_{fv}} a_f/3, \tag{43}$$

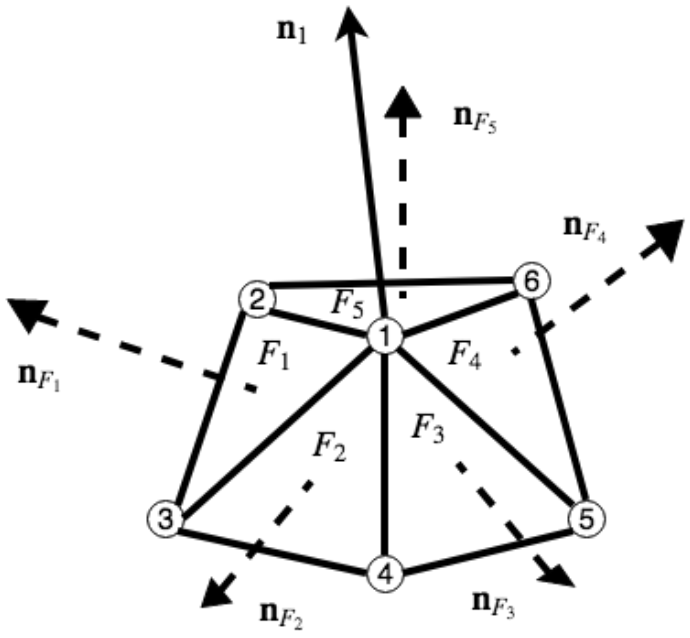


Figure 9 *Drawing of a part of a mesh, showing the normal vector $\mathbf{n}_{F_i}$ to faces F_i and the normal vector $\mathbf{n}_1$ in vertex 1.*

where N_{fv} is the set of indices of faces adjacent to vertex v. We call a_v and a_f the vertex and face area, respectively. To see that the sum of all vertex areas is equal to the sum of all face areas, consider the following. For every face, there are three vertices that belong to that face. So when looking at N_{fv} for all $v \in V$, every face appears three times. Hence

$$\sum_{v \in V} a_v = \sum_{v \in V} \sum_{f \in N_{fv}} a_f/3 = 3 \sum_{f \in F} a_f/3 = \sum_{f \in F} a_f. \tag{44}$$

We now have the magnitude of the pressure force, which leaves the direction. Each face has a normal, $\mathbf{n}_f$. For face $f = \{u, v, w\}$ it is computed as follows

$$\mathbf{n}_f = \frac{(v-u) \times (w-u)}{\|(v-u) \times (w-u)\|}. \tag{45}$$

See Figure 9 for a drawing of the faces and normals in a small part of a mesh. In each vertex, we define the normal as the average of the adjacent face normals:

$$\mathbf{n}_v \doteq \frac{\sum_{f \in N_f} \mathbf{n}_f}{|\sum_{f \in N_f} \mathbf{n}_f|}. \tag{46}$$

This allows us to compute the pressure force in vertex v:

$$\mathbf{F}_{p,v} = p a_v \mathbf{n}_v. \tag{47}$$

For the spring force in vertex v, we need the initial and current lengths of all edges containing v. Denote the length of an edge $e = \{u, v\}$ by $l_0^e = \|\mathbf{x}_v^0 - \mathbf{x}_u^0\|$ and $l_n^e = \|\mathbf{x}_v^n - \mathbf{x}_u^n\|$ respectively. For each edge containing v, we compute the Cauchy strain:

$$\varepsilon_e^n = \frac{l_n^e - l_0^e}{l_0^e} = \frac{l_n^e}{l_0^e} - 1. \tag{48}$$

Using a function σ allows us to compute the stress resulting from the deformation: $\sigma(\varepsilon_e)$. For linear springs we have Hooke's Law stating that $\sigma(\varepsilon) = k\varepsilon$, where k is Young's modulus, which is a constant describing the stiffness of the spring. In Figure 10 it is the steepness of the straight line. In the same figure, several other options to describing the stress-strain relationship of springs can be seen. Note that these are fictitious and do not represent an actual material.

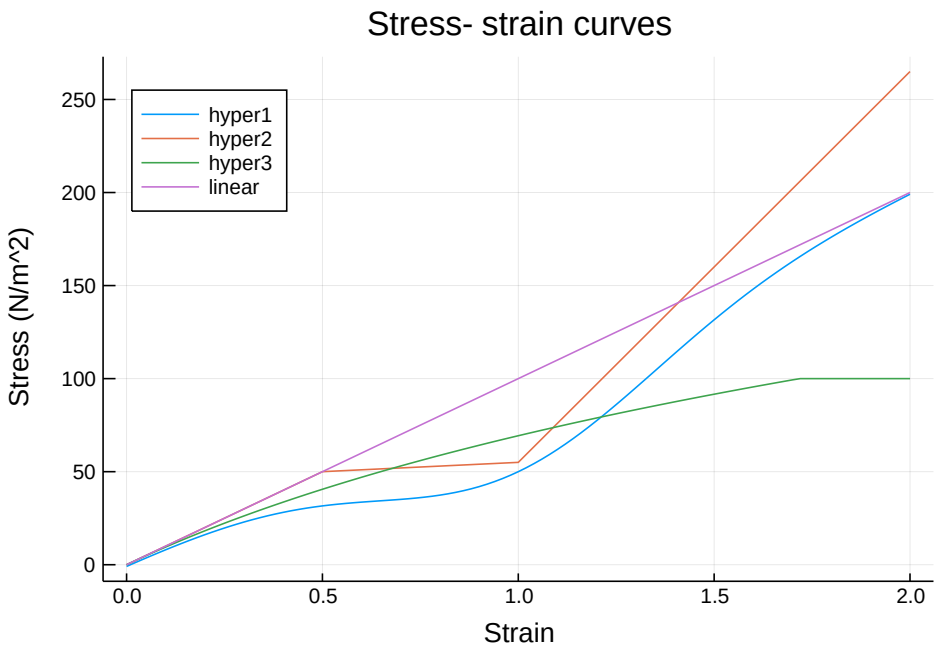


Figure 10 *Example stress-strain curves for a spring. The functions are*

hyper1: $\sigma(\varepsilon) = k(\varepsilon - 0.5e^{-4(\varepsilon-1)^2})$, *hyper2:* $\sigma(\varepsilon) = k\begin{cases}\varepsilon, & \text{if } 0 < \varepsilon < 0.1,\\ \varepsilon - 0.9(\varepsilon - 0.5), & \text{if } 0.1 \le \varepsilon < 1,\\ \varepsilon - 0.9(\varepsilon - 0.5) + 2(\varepsilon - 1), & \text{if } \varepsilon \ge 1,\end{cases}$

hyper3: $\sigma(\varepsilon) = k\begin{cases}\log(\varepsilon + 1), & \text{if } \varepsilon < e - 1,\\ 1, & \text{if } \varepsilon \ge e - 1,\end{cases}$ *linear:* $\sigma(\varepsilon) = k\varepsilon$,

In all cases shown here, $k = 10$.

We also need the direction of these stress forces. The direction of the stress in edge $e = \{u, v\}$ is:

$$\mathbf{d}_e = \mathbf{d}_{u,v} = \frac{\mathbf{x}_v - \mathbf{x}_u}{\|\mathbf{x}_v - \mathbf{x}_u\|}. \tag{49}$$

Finally, we need the cross sectional area belonging to edge e, as $\sigma(\varepsilon)$ is a force per unit area. As e is an edge of a triangle, it is not immediately clear what a_e should be. For a beam, it would be its width times height. The height corresponds to the thickness of the material and is denoted by δ. As we work with triangles, the width varies, so we compute an average width instead. Let e be the edge we want to compute the spring force in. To compute the average width belonging to e, we compute the width of a rectangle with area one third of the triangle and with the same length as e, for all triangles adjacent to e. So the width of an edge e is

$$w_e = \sum_{f \in N_{fe}} \frac{1}{3} \frac{a_f}{\|\mathbf{x}_{v_2} - \mathbf{x}_{v_1}\|}, \tag{50}$$

where N_{fe} is the set of indices of faces adjacent to edge e. The cross sectional area is then w_e times δ:

$$a_e = w_e \delta. \tag{51}$$

Thus, the force in edge e resulting from the spring is:

$$\mathbf{F}_{s,e} = \mathbf{d}_e \sigma(\varepsilon_e) a_e, \tag{52}$$

and the spring force in vertex v is the sum over the spring forces in the edges surrounding v:

$$\mathbf{F}_{s,v} = \sum_{e \in N_{ev}} \mathbf{F}_{s,e}, \tag{53}$$

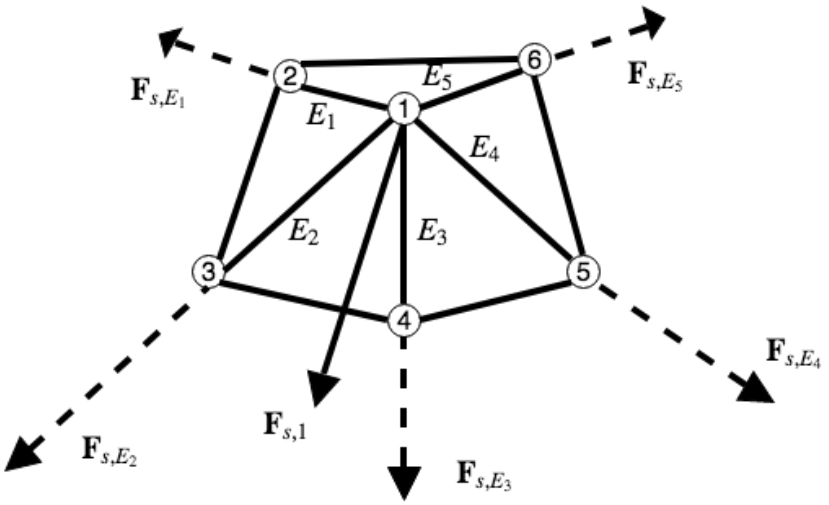


Figure 11 *Drawing of a part of a mesh, showing the spring force $\mathbf{F}_{s,E_i}$ in edge i, as well as the resulting spring force $\mathbf{F}_{s,1}$ in vertex 1.*

where N_{ev} is the set of edges adjacent to vertex v. It is important to note that the direction of $\mathbf{d}_e$ is $\mathbf{x}_u - \mathbf{x}_v$ when computing $\mathbf{F}_{s,v}$ and $\mathbf{x}_v - \mathbf{x}_u$ for $\mathbf{F}_{s,v}$.

Our physical model described in Section 2.2, does not include any dissipative forces. So using only pressure and spring forces results in a system that takes very long stabilize. Also, in practice the pressure is slowly increased when inflating a preform, instead of immediately putting a large pressure on it. So we introduce a fictitious friction force in each vertex. This force depends on the velocity, because friction forces oppose the direction of the vertex' movement. As the velocity for the final position of the grid is zero, this friction force eventually disappears and does not influence the final position of the vertices. For the friction forces, we need the previous and current position of the vertices. We estimate the velocity vector:

$$\mathbf{u}_v \doteq \frac{\mathbf{x}_v^n - \mathbf{x}_v^{n-1}}{\Delta t}, \tag{54}$$

and use it to compute the friction force:

$$\mathbf{F}_f = -\mu \mathbf{u}_v, \tag{55}$$

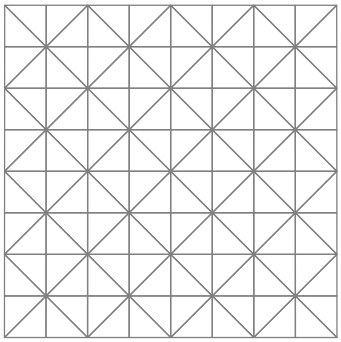
where μ is a constant friction coefficient. The choice of μ depends on the geometry and p. Values between 0.5 and 0.01 seem to work well. Finally, we can compute the mass belonging to a vertex by using a_v when we have the density of the material, ρ:

$$m_v = a_v \delta \rho. \tag{56}$$

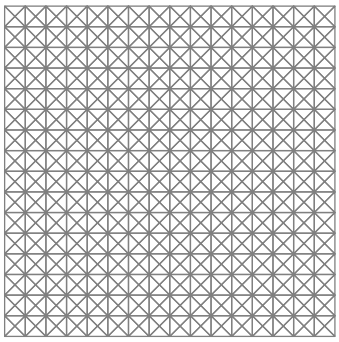
So we obtain for the acceleration in vertex v:

$$\mathbf{A}_v^n = \mathbf{F}_s^n + \mathbf{F}_p^n + \mathbf{F}_f^n. \tag{57}$$

The stopping criterion we use is: $\|\mathbf{F}\| = \|\mathbf{F}_s + \mathbf{F}_p + \mathbf{F}_f\| < \epsilon$.



(a) *The original mesh, m_0.*



(b) *Mesh m_1, which is equal to m_0 with each triangle subdivided into 8 smaller triangles.*

Figure 12 *The meshes used.*

4.2 Results

We show some results of inflating several meshes. Here, we assume that during inflation, the volume of the preform domain Ω_t does not change and that the thickness of the material is uniform. So the material is incompressible. This means that $a_f^n \delta^n$ is constant for all $f \in F$. So we only have to compute the volume of a face once, and then use that to compute a_e^n for all $e \in E$. Let δ^0 denote the initial thickness of the preform, then the volume of a face f is

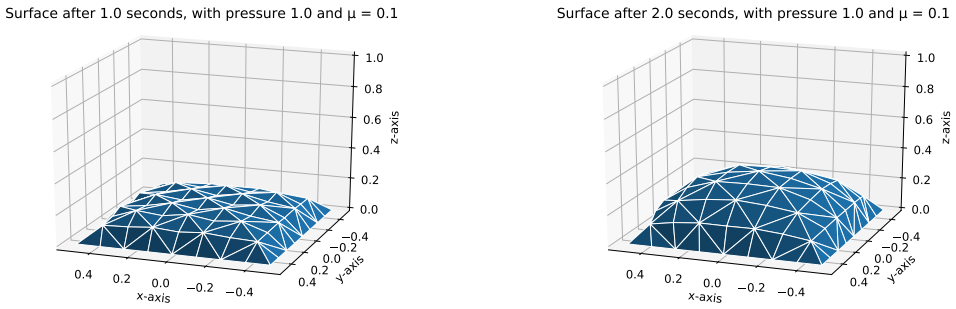
$$v_f = \delta^n a_f^n = \delta^0 a_f^0. \tag{58}$$

Then we obtain the cross sectional area at time step n as follows:

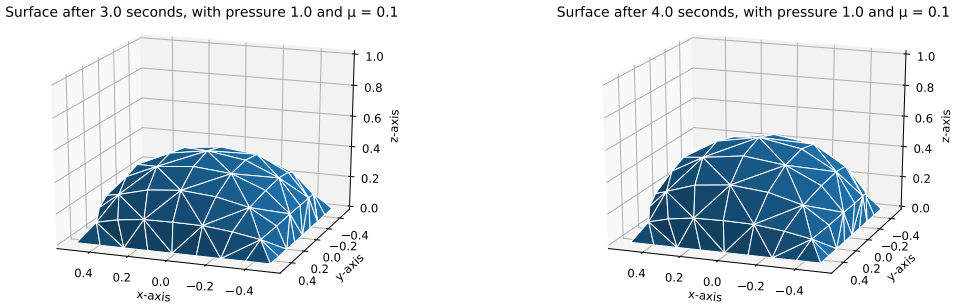
$$a_e^n = \delta^n w_e^n = \sum_{f \in N_{fe}} \frac{1}{3} \frac{\delta^n a_f^n}{l_e^n} = \sum_{f \in N_{fe}} \frac{1}{3} \frac{v_f^n}{l_e^n} = \sum_{f \in N_{fe}} \frac{1}{3} \frac{v_f}{l_e^n} \tag{59}$$

which means that we do not need to know δ^n, but only δ_0.

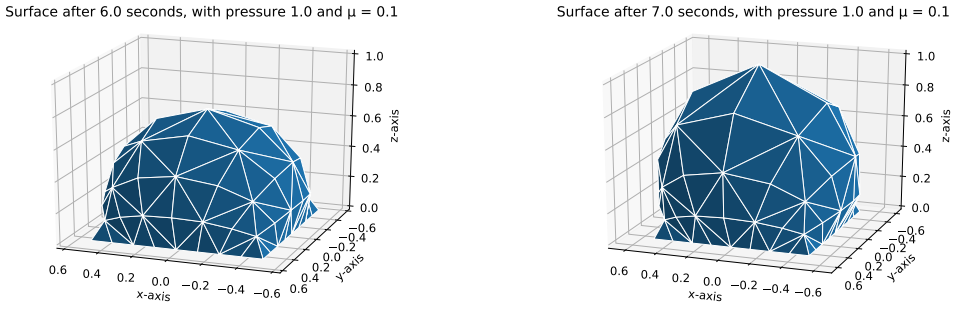
For the same pressure, we compute the effect on two different meshes on $[-0.5, 0.5] \times [-0.5, 0.5]$: see Figure 12 for the graph of the two meshes. The finer mesh (Figure 12b) is the result of a subdivision of the first mesh (Figure 12a). Each triangle is subdivided into eight smaller ones. We call these meshes m_0 and m_1. The boundary vertices are fixed and do not move. We take a fictional material with: $\delta = 0.02\ m$, $k = 100\ N/m^2$, $\rho = 1\ kg/m^3$. See Figure 13 and 14 for the results with a linear spring. We see that both inflate to something similar to a sphere.



(a) $n = 1000$. **(b)** $n = 2000$.

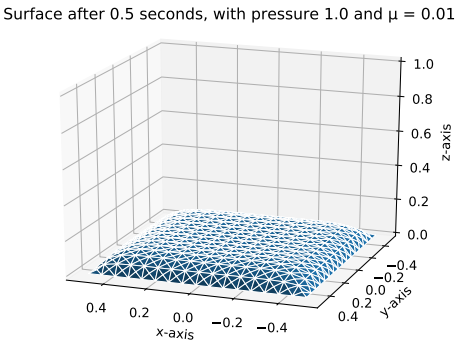


(c) $n = 3000$. **(d)** $n = 4000$.

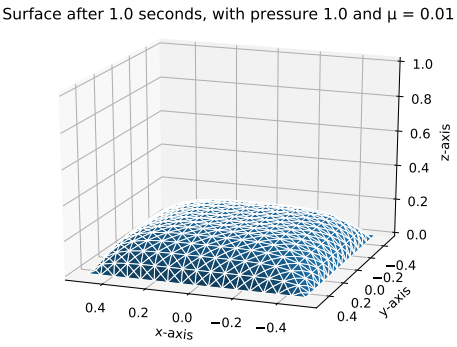


(e) $n = 5000$. **(f)** $n = 7000$.

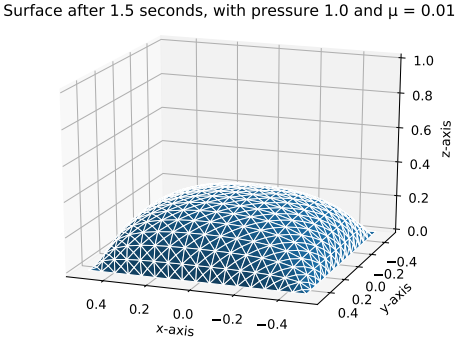
Figure 13 *Inflation process of m_0 with $\Delta t = 0.001$, $p = 1$, $\mu = 0.1$, $k = 100$, $\rho = 1$ and $\delta = 0.02m$.*



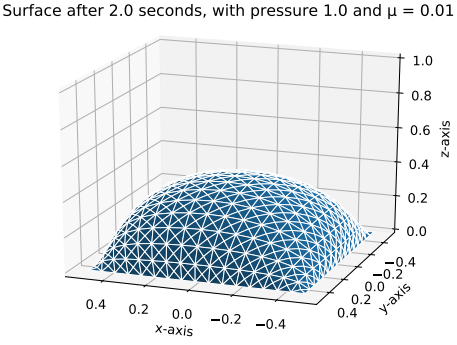
(a) $n = 500$.



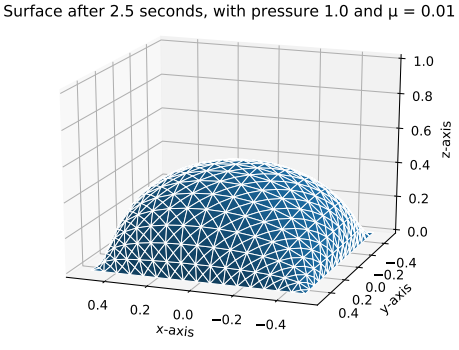
(b) $n = 1000$.



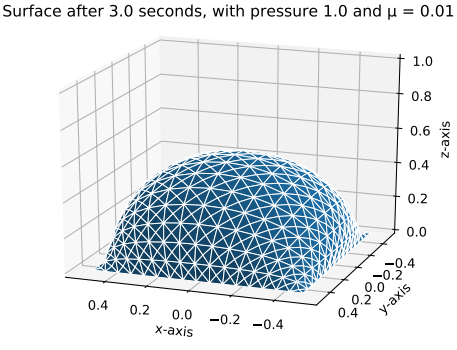
(c) $n = 1500$.



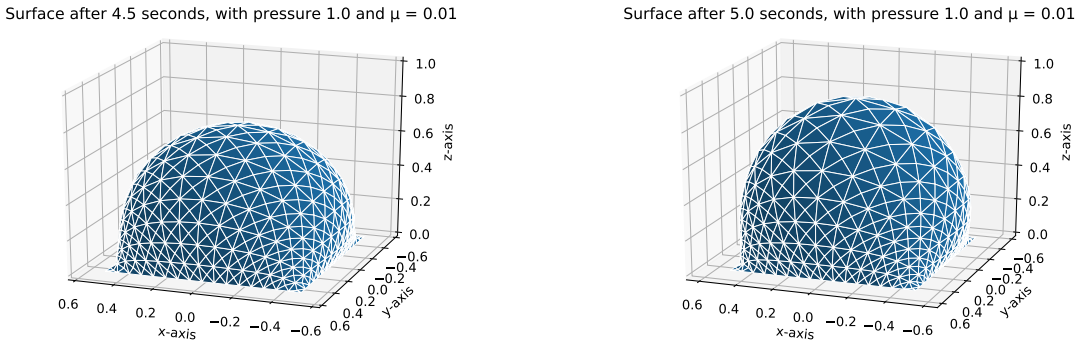
(d) $n = 2000$.



(e) $n = 2500$.



(f) $n = 3000$.



(g) $n = 4500$. **(h)** $n = 5000$.

Figure 14 *Inflation process of m_1 with $\Delta t = 0.001$, $p = 1N/m^2$, $\mu = 0.01$, $k = 100$, $\rho = 1$, and $\delta = 0.02m$.*

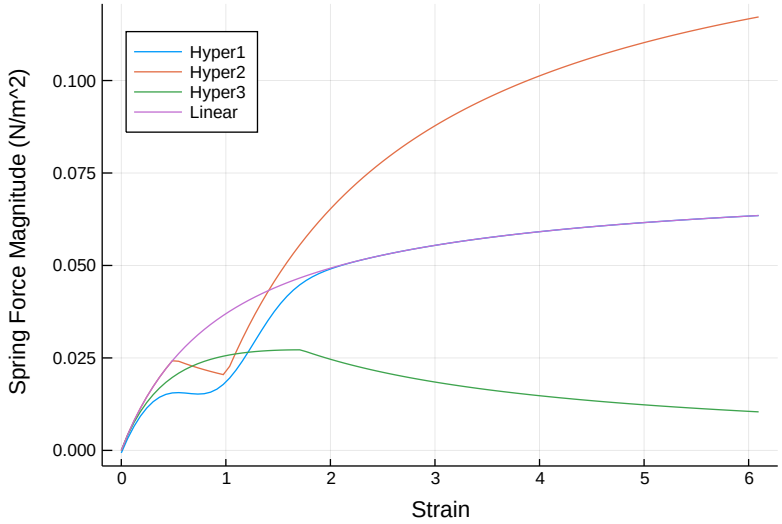


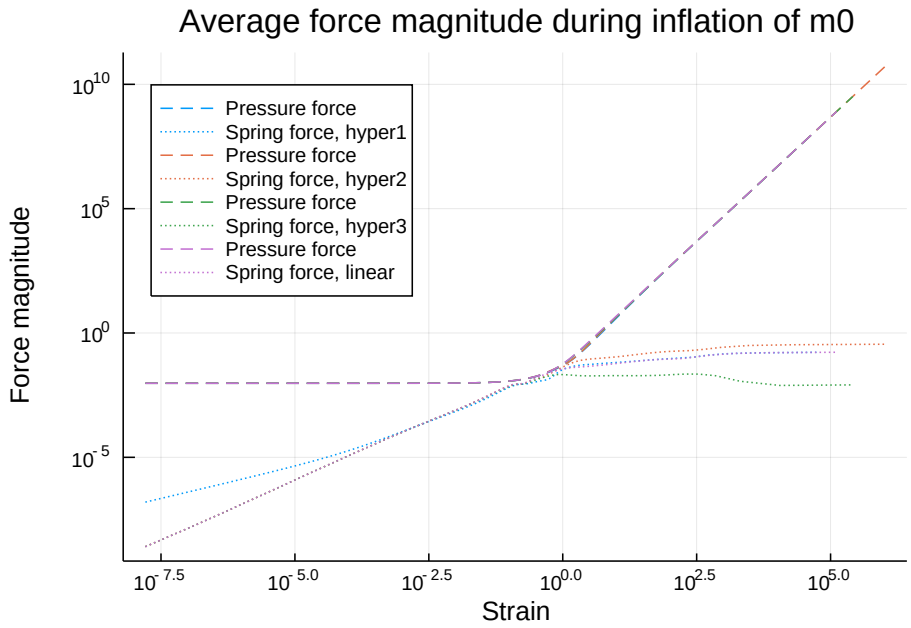
Figure 15 *$\mathbf{F}_{s,e}$ as a function of the strain. The stress functions are the ones in given in Figure 10 with $k = 100$.*

However, they also never stop inflating. An explanation is found when looking at the expression for a_e. It decreases with increasing edge length, resulting in a non-linear spring force magnitude, even though the springs themselves are linear. We show this by looking at the spring force as a function of ε in an edge, e. According to Equation (52), the magnitude of the spring force in e is equal to

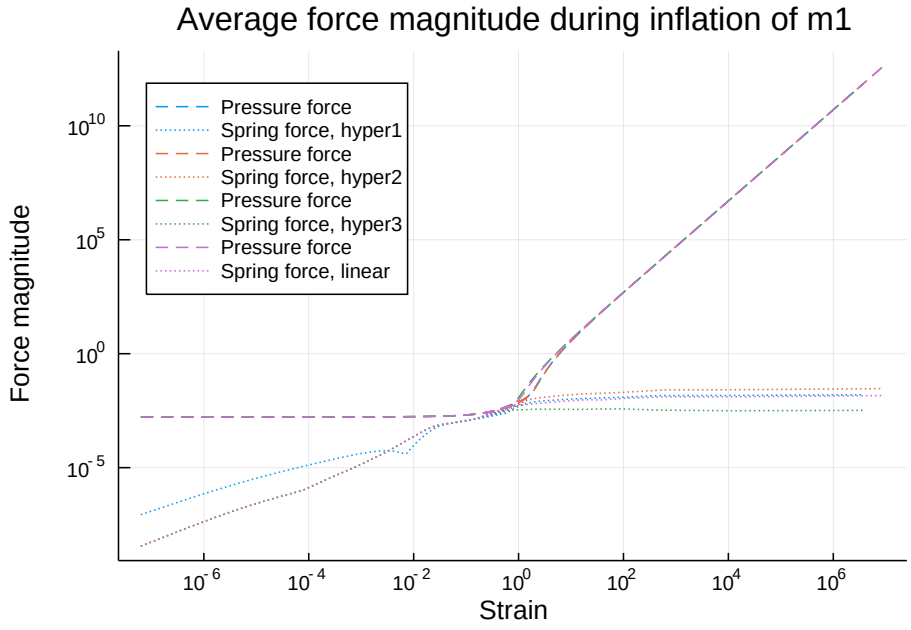
$$\|\mathbf{F}_{s,e}\| = \sigma(\varepsilon_e^n)a_e^n = \sum_{f \in N_{fe}} \sigma(\varepsilon_e^n)\frac{1}{3}\frac{v_f}{l_e^n}. \tag{60}$$

So while σ may be increasing for increasing ε_e^n, the second part, $\frac{v_f}{l_e^n}$, is certainly decreasing for increasing ε_e^n.

As an example, we take $v_f = 0.000156525$, as that is the value for all faces in m_0, and $l_e^0 = 0.14$, as that is the average edge length before inflation in m_0. See Figure 15 for a graph of the spring force magnitude in an edge as a function of the strain. For comparison, the figure also shows the same graph for other springs.

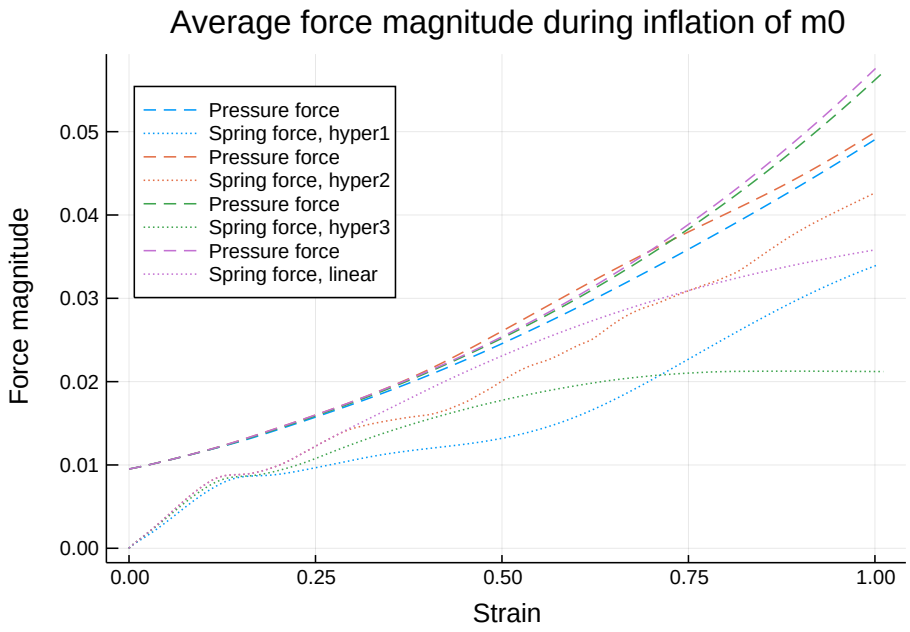


(a) *Average pressure and spring force magnitude of* m_0.

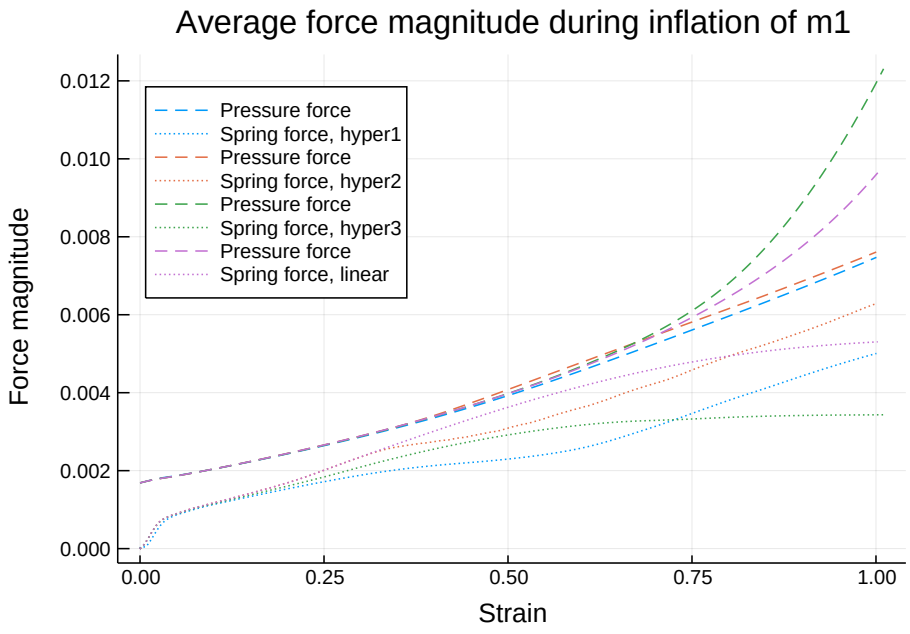


(b) *Average pressure and spring force magnitude of* m_1.

Figure 16 *Average pressure and spring force magnitude as a function of the strain, taken during the inflation of the mesh, with* $k = 100$, $p = 1$, $\Delta t = 0.001$, $\mu = 0.01$, *and* $\delta = 0.02m$.



(a) *Average pressure and spring force magnitude of m_0.*



(b) *Average pressure and spring force magnitude of m_1.*

Figure 17 *The same as Figure 16, with strain between 0 and 1.*

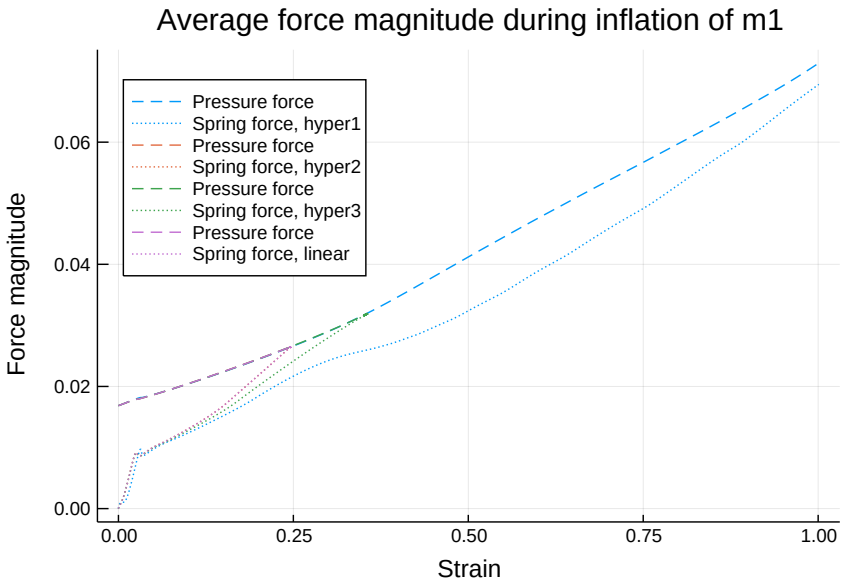


Figure 18 *Average pressure and spring force magnitude as a function of the strain, taken during the inflation of mesh m_1, with $k = 1500$, $p = 1$, $\Delta t = 0.001$, $\mu = 0.01$, and $\delta = 0.02m$.*

We can see that, while we use a linear spring, we do not end up with a linear force at all. This holds true even for the vertex spring force, which is the sum of the edge forces. To compare, the pressure force magnitude for a vertex is

$$\|\mathbf{F}_{p,v}\| = pa_v, \tag{61}$$

so it scales linearly with the area of the vertex. Of course, in general a_v does not scale linearly with the strain of its edges. However, during inflation p is fixed, while a_v increases. So $\|\mathbf{F}_{p,v}\|$ is monotonically increasing, while $\|\mathbf{F}_{s,v}\|$ may even be decreasing. The result is that the inflation never stops if there exists an $N \in \mathbb{N}$ such that $\|\mathbf{F}^n_{p,v}\| > \|\mathbf{F}^n_{s,v}\|$ for all $n > N$. See Figure 16, which illustrates the above by showing the average pressure and spring force magnitudes as a function of the strain for both meshes. In Figure 17 the same values are shown, but for a smaller range of the strain. It clearly shows that the pressure force is always larger than the spring force.

A possible remedy is the case where Young's modulus is higher, which can be seen in Figure 18. In this case, the pressure and spring force magnitude are equal at some point, which implies static equilibrium. However, we also see that we are stuck with small strains. Another option is that the stress functions do not exhibit enough strain hardening, so we design a new one, see Figure 19. It is a cubic function and in Figure 20 we can see that for the same conditions as in Figure 16, static equilibrium is reached with this spring. It also shows that the friction force indeed disappears in that case. Finally, we observe that the hyper4 spring allows a much greater strain before the pressure and spring forces cancel.

Finally, our assumption about the conservation of the volume also contributes to the above phenomenon. And even though not all materials conserve their volume under stress, the thickness does in general decrease with increasing stress. So we conclude that the material needs strain hardening to avoid this problem, or have a high Young's modulus.

Since we can obtain solutions with force equilibrium, we could plot a mesh after inflation to show the effect of pressure. However, for a square mesh like m_1 with fixed boundaries, the maximum height is a good indication of its shape. Hence, we only look at the maximum height of m_1 after inflation as a function of the pressure. In Figure 21 we can see the results. We see that the choice of mesh, m_0 or m_1 differs little. The difference that is present, can be explained by the fact that m_1 has more faces. One third of the triangle faces at the boundary of m_0 is not used, whereas m_1 does use a part of this area. So the pressure force is a little bit higher for m_1. Furthermore, we see that not all springs have a value for the height for all pressures. This is caused by the fact that static equilibrium is not reached for these pressures. Lastly, there is a sharp jump visible for the hyper1, hyper2 and hyper4 springs. This is caused by the negative gradient of the stress-strain function.

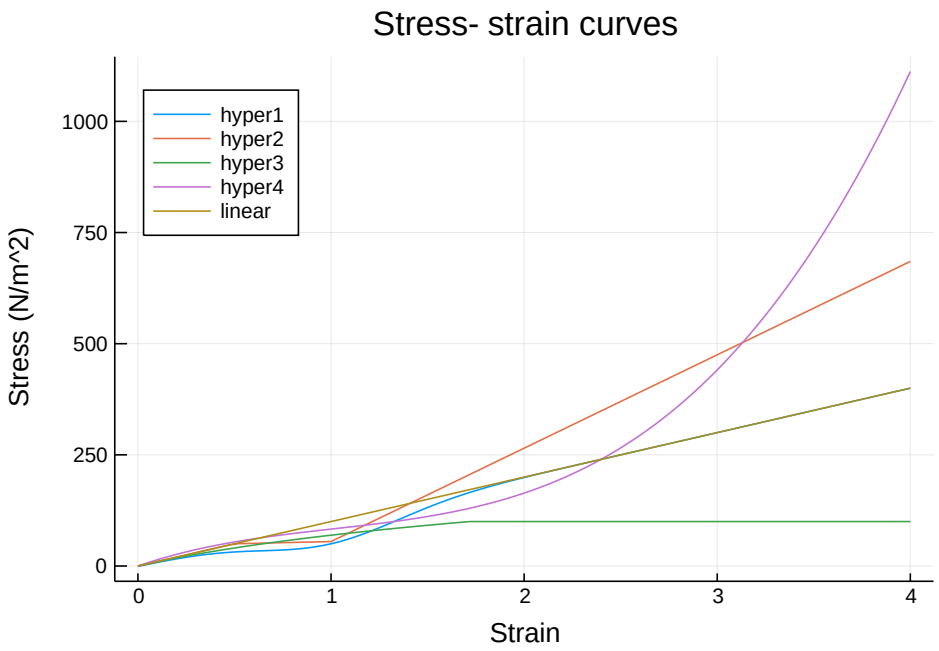


Figure 19 *Example stress-strain curves for a spring. The functions hyper 4 function is*

$$\sigma(\varepsilon) = k\varepsilon(\frac{1}{3}\varepsilon^2 - \varepsilon + \frac{3}{2}), \text{ with } k = 10.$$

Of course, we also want to see the effect on a more interesting shape. To this end, we use a 3D test model called Spot, see Figure 22. In Figures 23, 24, 25, 26, and 27 Spot is inflated with different springs. Note that none of these actually reach static equilibrium and keep inflating.

Qualitatively, the difference between the springs is visible at the legs, nose bridge and tail. Spot with the hyper3 springs is more round, while with hyper4 springs it retains more of its original shape. With the linear spring it is in between the two and the hyper2 spring differs little from the linear spring. As before, the hyper4 spring allows for a much larger inflation. With the hyper1 spring, Spot's shape differs qualitatively the most from the other springs. An explanation lies in the fact that at the start of inflation, the areas of the head and body are closer to each other in area. So when the dip in the stress-strain curve starts, the head also suddenly inflates quite a lot. Another interesting feature is visible in Figure 26e. There are a couple of faces on Spot's side that have increased more in area than the surrounding faces. Hence, the thickness there has decreased much more. In Figure 26f we see that this place inflates much more than the rest, which is as expected from the theory.

The advantages of an MSS are clear. It is a quick way to simulate the inflation of preforms and gives an intuition of the effects of different stress-strain curves on the final shape of a mesh. However, there are also disadvantages. It takes quite a bit of tweaking to get results with, e.g., the friction coefficient. It is also difficult to have realistic elastic responses and relies on many heuristics, e.g. in the computation of the vertex areas from face areas and edge cross sectional areas. Furthermore, while it is a local approach, it uses a small number of neighbors. Finally, we use fictitious springs. While it is possible to use the Kawabata Evaluation System to measure mechanical properties for fabrics, [34], the only option here is to use experimental data to model the stress-strain curve. Hence, we conclude that an MSS is also a less than ideal tool to simulate the inflation of preforms.

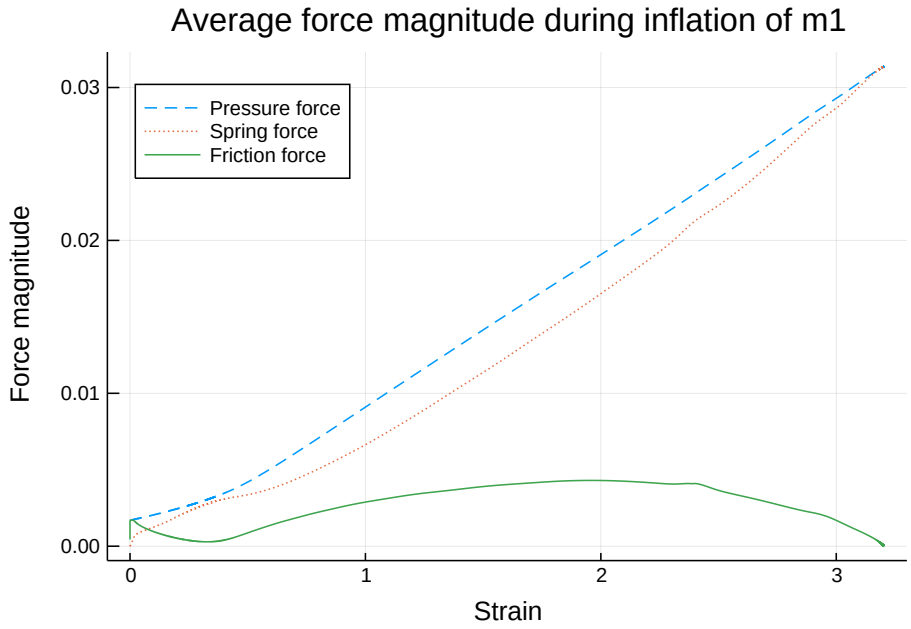


Figure 20 *Average pressure, spring and friction force magnitude as a function of the strain with hyper4 spring, taken during the inflation of mesh m_1, with $k = 100$, $p = 1$, $\Delta t = 0.001$, $\mu = 0.01$, and $\delta = 0.02m$.*

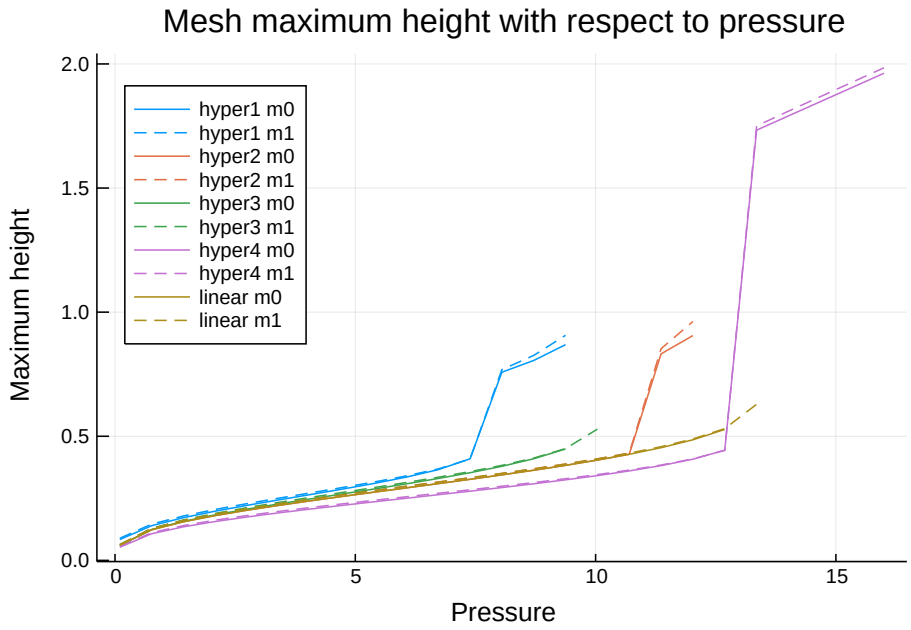


Figure 21 *The maximum height of a mesh at static equilibrium as a function of the pressure, with $k = 1500$, $\delta = 0.02$, $\mu = 0.02$ and $\Delta t = 0.001$.*

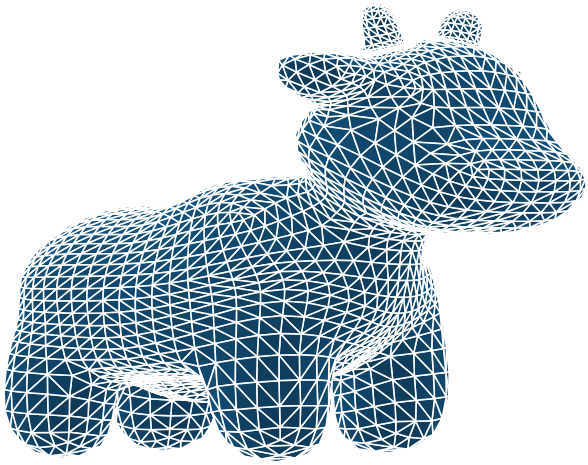
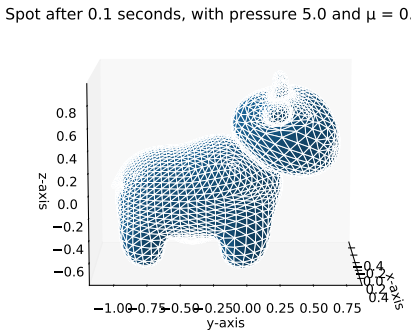
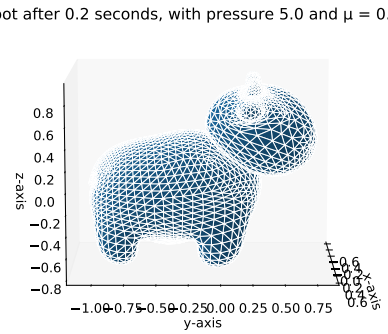


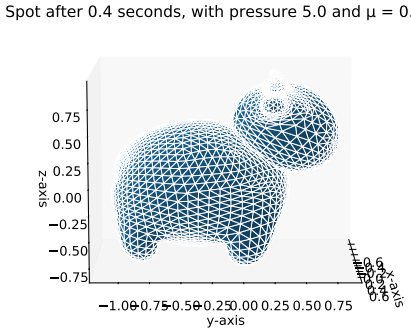
Figure 22 *3D test model, Spot.*



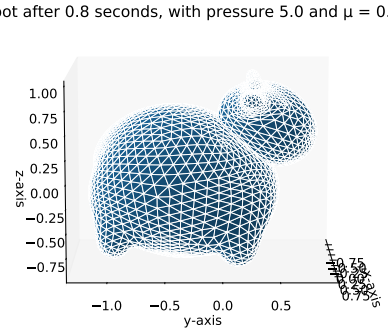
(a) $n = 1000$.



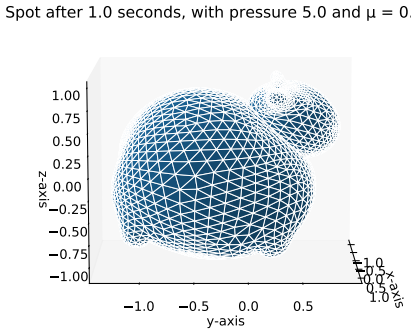
(b) $n = 2000$.



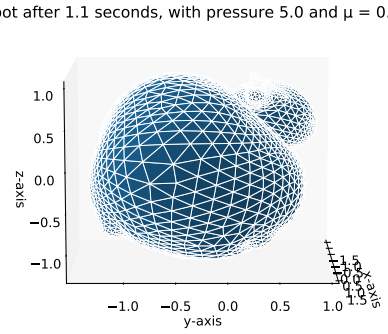
(c) $n = 4000$.



(d) $n = 8000$.

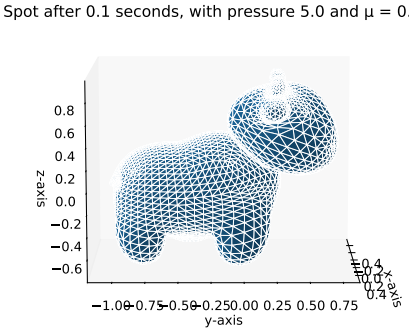


(e) $n = 10000$.

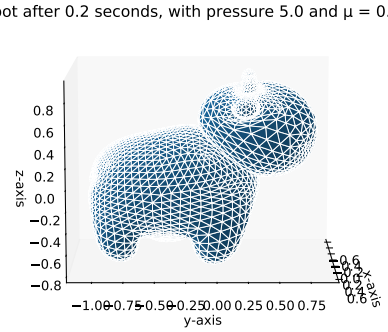


(f) $n = 11000$.

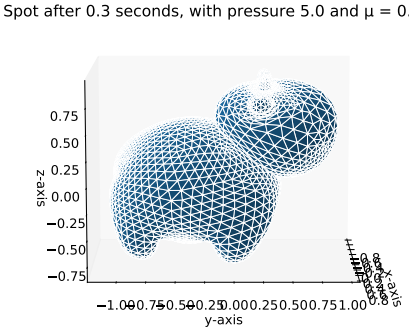
Figure 23 *Inflation process of Spot with linear spring and $\Delta t = 0.0001$, $p = 5N/m^2$, $\mu = 0.01$, $k = 500$, $\rho = 10$, and $\delta = 0.02m$.*



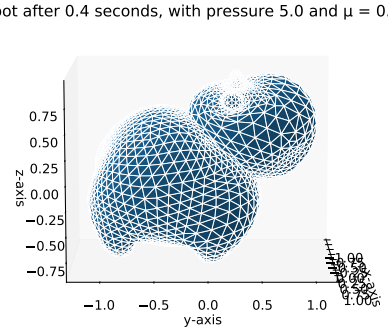
(a) $n = 1000$.



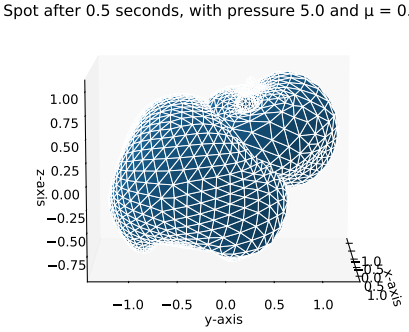
(b) $n = 2000$.



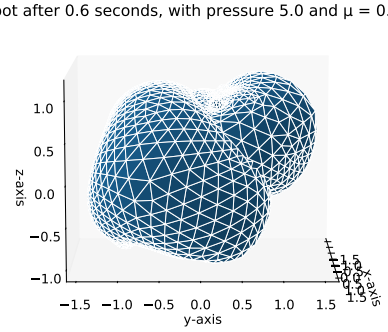
(c) $n = 3000$.



(d) $n = 4000$.

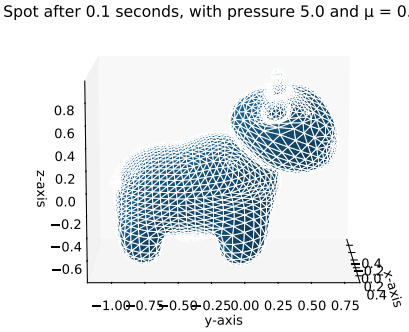


(e) $n = 5000$.

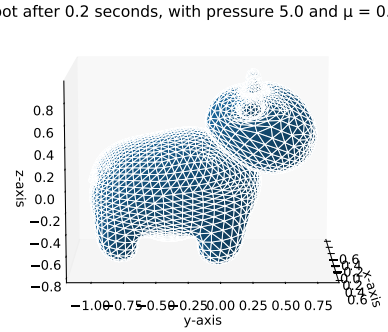


(f) $n = 6000$.

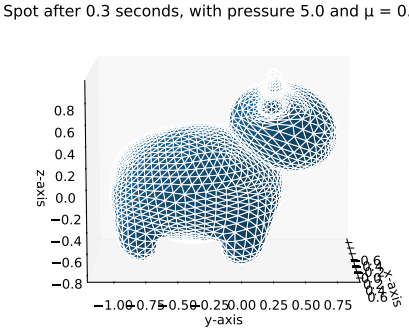
Figure 24 *Inflation process of Spot with hyper1 spring and* $\Delta t = 0.0001$, $p = 5N/m^2$, $\mu = 0.001$, $k = 500$, $\rho = 10$, *and* $\delta = 0.02m$.



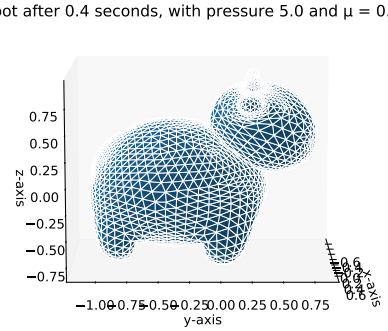
(a) $n = 1000$.



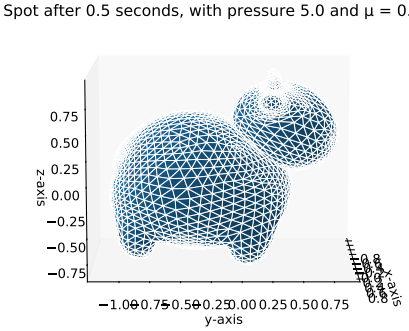
(b) $n = 2000$.



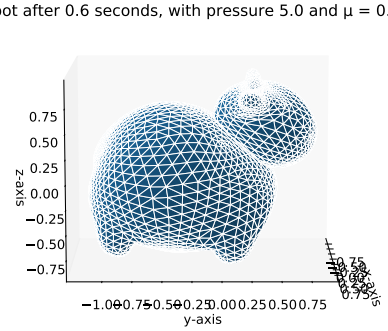
(c) $n = 3000$.



(d) $n = 4000$.

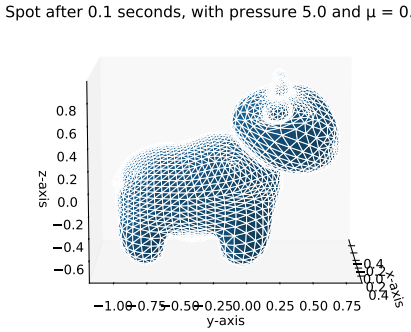


(e) $n = 5000$.

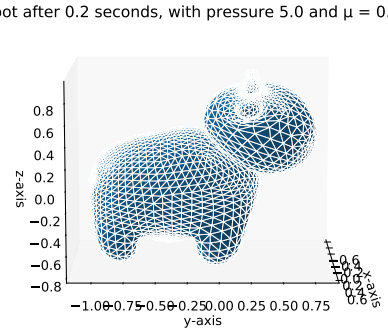


(f) $n = 6000$.

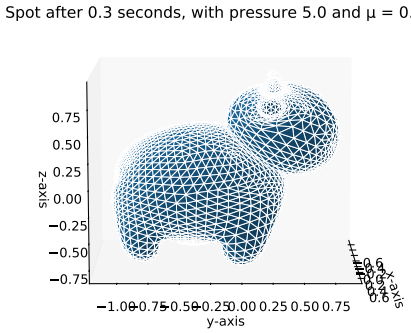
Figure 25 *Inflation process of Spot with hyper2 spring and* $\Delta t = 0.0001$, $p = 5N/m^2$, $\mu = 0.001$, $k = 500$, $\rho = 10$, *and* $\delta = 0.02m$.



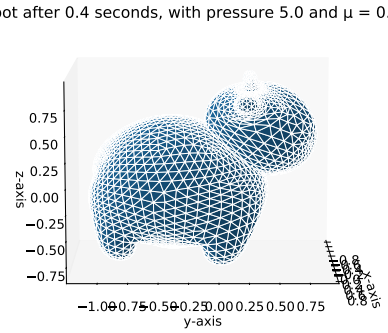
(a) $n = 1000$.



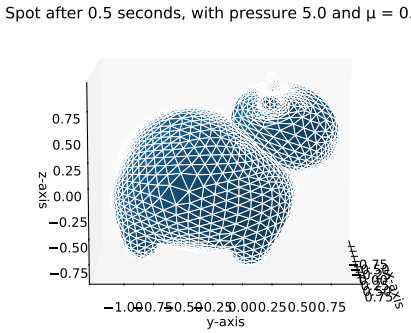
(b) $n = 2000$.



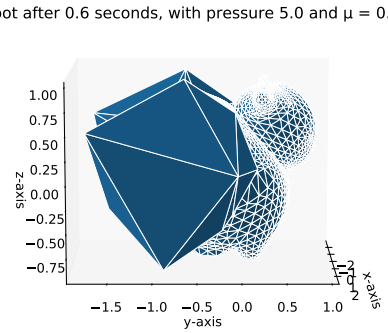
(c) $n = 3000$.



(d) $n = 3750$.

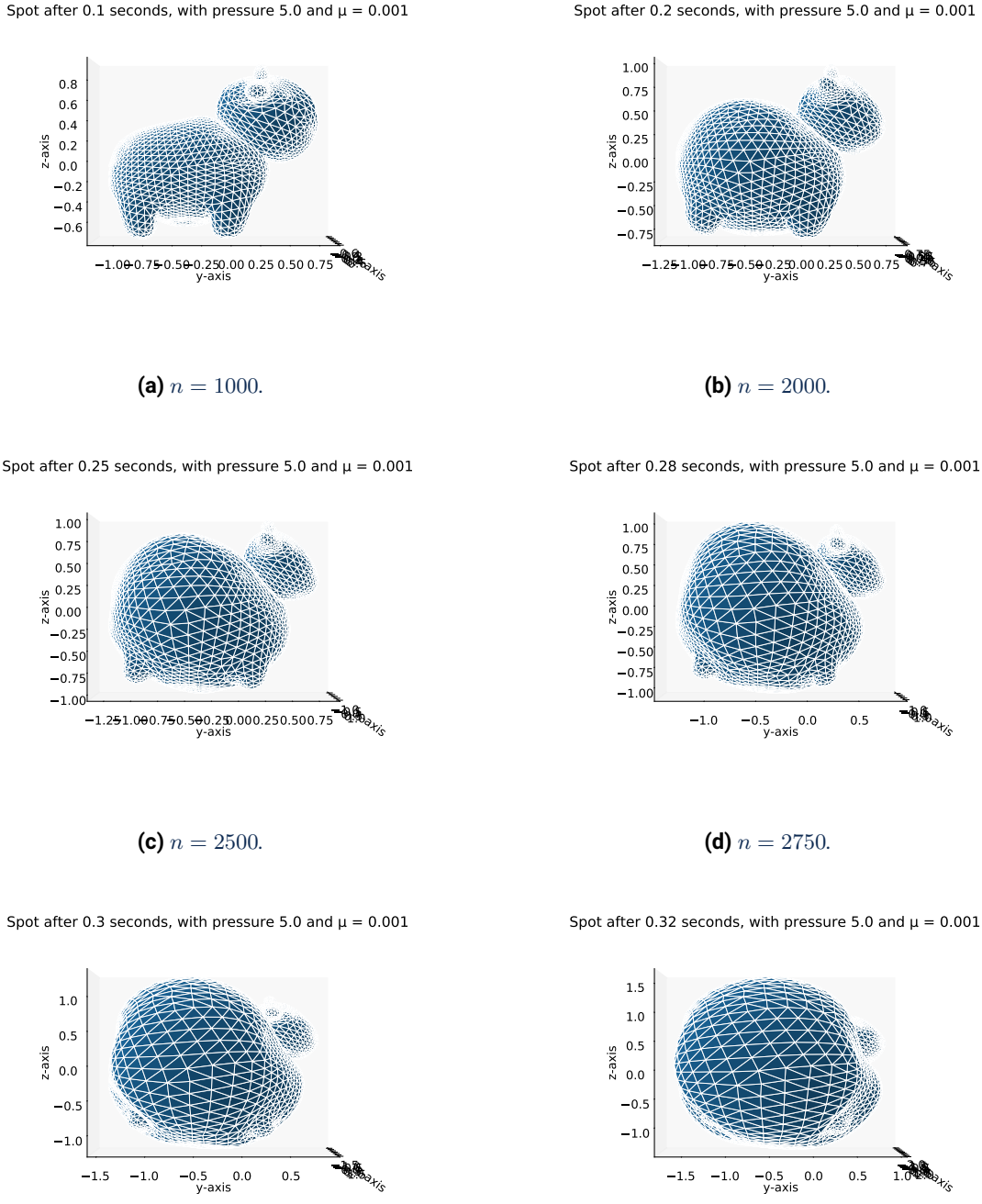


(e) $n = 5000$.



(f) $n = 6000$.

Figure 26 *Inflation process of Spot with hyper3 spring and $\Delta t = 0.0001$, $p = 5N/m^2$, $\mu = 0.001$, $k = 500$, $\rho = 10$, and $\delta = 0.02m$. In Figure 26e, there are a couple of faces on Spot's side that have increased in area more than others. In Figure 26f, this effect is even larger, because the pressure forces scale with area.*



(a) $n = 1000$.

(b) $n = 2000$.

(c) $n = 2500$.

(d) $n = 2750$.

(e) $n = 3000$.

(f) $n = 3250$.

Figure 27 *Inflation process of Spot with hyper4 spring and* $\Delta t = 0.0001$, $p = 5N/m^2$, $\mu = 0.001$, $k = 500$, $\rho = 10$, *and* $\delta = 0.02m$.

5 Smoothed Particle Hydrodynamics

In Section 4 we saw how we could heuristically make a particle system act like an inflatable membrane. The connection to the underlying physics needed to be taken care of manually, while we would prefer to describe a model from first principles. In this section we use SPH to do exactly this. We also note that SPH still uses a material model as described in Section 2.3.1. It is, however, a step closer to a data-driven formulation, as it uses a larger number of neighbors than the MSS. Also, it serves to show that a local approach using information from neighboring particles can work.

5.1 Overview of Literature

SPH is a mesh-free method to discretize physical models by particle interactions. It is used in many disciplines, among which astrophysics and fluid simulations. Its main advantage is that it does not need a grid to compute spatial derivatives, but uses neighboring particles combined with a smoothing kernel to estimate and smooth these. Because of this, it can handle large deformations, lack of boundaries and topological changes.

Smoothed Particle Hydrodynamics was originally developed for astro-physics by Lucy [35] and Gingold and Monaghan [36]. It is often used for fluid simulations [37, 38], but has also been used for solid dynamics. The first was by Libersky and Petschek [39], where they modeled the impact of a rod on a surface. As it is a mesh-free method, it was often used to model impacts and fracture [40–45]. Other applications include metal extrusion [46], simulating lava flow [47], plastic deformation in metals [48], and elastic mechanics [2]. Finally, elastic solids and fluids are sometimes combined [49, 50].

While SPH has certain advantages, there are also some drawbacks. It can suffer from tensile instability [51, 52]. Several remedies have been proposed, such as certain particle distributions and introducing an artificial stress [53, 54]. Also, SPH is often not consistent [55]. To resolve it, the kernel function or its gradient can be modified [56–60]. Finally, boundaries also pose problems, due to a reduced number of neighbors for particles near the boundary. This can usually be solved by using ghost particles [61, 62].

5.2 The SPH Formulation

To obtain a smooth approximation of the function $f : \mathbb{R}^d \to \mathbb{R}$, a mollifier as introduced by Friedrichs [63] may be used:

$$\langle f(x) \rangle = \int_{\mathbb{R}^d} f(y)\phi(x-y)\, dy, \tag{62}$$

where $\phi : \mathbb{R}^d \to \mathbb{R}$ is a smooth function. We can also obtain a smooth approximation of its derivative, in a weak sense, in a similar fashion:

$$\langle \frac{\partial f(x)}{\partial x_\alpha} \rangle = \int_{\mathbb{R}^d} f(y) \frac{\partial \phi(x-y)}{\partial x_\alpha}\, dy, \tag{63}$$

where α is an index ranging over 1 to d. In SPH, these integrals are discretized and ϕ is replaced by a function W_h called the smoothing function, smoothing kernel or kernel function. The parameter $h \in (0, \infty)$ is called the smoothing length.

In SPH, a system is represented by a number of particles, i, each of which has a fixed mass, m_i, and location in Euclidean space, $\mathbf{x}_i$. A particle has no defined size and may vary wildly in volume between different applications. If we are interested in any other scalar field than the mass, we use the neighboring particles to get a smoothed approximation. Let $f(\mathbf{x})$ and $\nabla f(\mathbf{x})$ denote this scalar field and its gradient respectively, then:

$$\langle f(\mathbf{x}_i) \rangle = \sum_{j \in H_i} \frac{m_j}{\rho_j} f(x_j) W(\mathbf{x}_i - \mathbf{x}_j, h), \tag{64}$$

$$\langle \nabla f(\mathbf{x}_i) \rangle = \sum_{j \in H_i} \frac{m_j}{\rho_j} f(x_j) \nabla_i W(\mathbf{x}_i - \mathbf{x}_j, h), \tag{65}$$

where ∇_i denotes the gradient with respect to $\mathbf{x}_i$, and H_i is the set containing the indices of neighboring particles. Whether two particles are neighbors, depends on the choice of smoothing kernel and the smoothing length. For

any scalar field $f(\mathbf{x})$, we use f_i as shorthand notation when we mean $f(\mathbf{x}_i)$. The smoothing kernel can be written in the form [64]:

$$W(\mathbf{x}, h) = \frac{c_d}{h^d} w(q), \quad q := \frac{\|\mathbf{x}\|}{h}, \quad c_d^{-1} := \int_{\mathbb{R}^d} w \, dx. \tag{66}$$

The gradient of $W(\mathbf{x}, h)$ is given by

$$\nabla_i W(\mathbf{x}) = \frac{c_d}{h^{d+1}} \frac{\mathbf{x}}{\|\mathbf{x}\|} \frac{\partial w}{\partial q}. \tag{67}$$

Ideally, the smoothing kernel satisfies the following properties [64]:

1. infinite smoothness, $W \in C^\infty$,
2. compact support, $W(\tilde{q}) = 0$ for some $\tilde{q} \in \mathbb{R}_{>0}$,
3. scaling to unity, $\int_{\mathbb{R}^d} W d\mathbf{x} = 1$,
4. convergence to the Dirac delta function, $W(\mathbf{x}, h \to 0) = \delta(\mathbf{x})$,
5. radial symmetry, $W(\mathbf{x}, h) = W(\|\mathbf{x}\|, h)$.

The correction factor c_d in Equation (66) serves to scale $\int_{\mathbb{R}^d} W(\mathbf{x}, h) dx$ to unity. Often, smoothing kernels do not satisfy all properties. A common choice for the smoothing function used to be the cubic B-spline,

$$w(q) = \begin{cases} \frac{2}{3} - q^2 + \frac{1}{2} q^3 & \text{for } 0 \leq q < 1, \\ \frac{1}{6}(2 - q)^3 & \text{for } 1 \leq q < 2, \\ 0 & \text{for } q \geq 2, \end{cases} \tag{68a}$$

with derivative

$$\frac{\partial w}{\partial q} = \begin{cases} -2q + \frac{3}{2} q^2 & \text{for } 0 \leq q < 1, \\ -\frac{1}{2}(2 - q)^2 & \text{for } 1 \leq q < 2, \\ 0 & \text{for } q \geq 2. \end{cases} \tag{68b}$$

In this case c_d is 1, $15/(7\pi)$ and $2/(3\pi)$ for one, two and three dimensions, respectively [65]. It has been shown that the Wendland kernels result in better convergence [66]. The expression for the kernels depends on the dimension. We use the C^4 kernel, which we will refer to as the Wendland kernel, and have for $n = 1$

$$w(q) = \begin{cases} (1 - \frac{1}{2} q)^5 (1 + \frac{5}{2} q + 2q^2) & \text{for } 0 \leq q < 2, \\ 0 & \text{for } q \geq 2, \end{cases} \tag{69a}$$

with derivative

$$\frac{\partial w}{\partial q} = \begin{cases} -7q(1 - \frac{1}{2} q)^4 (\frac{1}{2} + q) & \text{for } 0 \leq q < 2, \\ 0 & \text{for } q \geq 2, \end{cases} \tag{69b}$$

and for $n = 2, 3$:

$$w(q) = \begin{cases} (1 - \frac{1}{2} q)^6 (1 + 3q + \frac{35}{6} q^2) & \text{for } 0 \leq q < 2, \\ 0 & \text{for } q \geq 2, \end{cases} \tag{70a}$$

with derivative

$$\frac{\partial w}{\partial q} = \begin{cases} \frac{7}{6} q(1 - \frac{1}{2} q)^5 (1 - 20q) & \text{for } 0 \leq q < 2, \\ 0 & \text{for } q \geq 2. \end{cases} \tag{70b}$$

For the Wendland kernel we have c_d is $\frac{3}{4}$, $9/(4\pi)$ and $495/(256\pi)$ for one, two and three dimensions, respectively [66].

Both the cubic B-spline kernel and the Wendland kernel are not in C^∞. A kernel that is infinitely smooth, but does not have compact support, is the Gaussian kernel:

$$w(q) = e^{-(3q/2)^2} \tag{71a}$$

with derivative

$$\frac{\partial w}{\partial q} = -3qe^{-(3q/2)^2}. \tag{71b}$$

Its lack of compact support is the reason it is not often used, because it requires us to take all vertices into account as neighbors. However, if we can cut it off and make it zero for $q \geq 6$. The contributions of particles that far away is much smaller than machine precision, as $w(6) = \frac{1}{e^{324}} \approx 1.9 \cdot 10^{-141}$.

As a shorthand notation, we will write W_{ij} instead of $W(\mathbf{x}_i - \mathbf{x}_j, h)$.

5.3 Equations

From [2], we get the following smoothed approximations for $\mathbf{F}$ and $\frac{d\mathbf{v}_i}{dt}$ at particle i, which we need in the governing Equation (1):

$$\mathbf{F}_i = \sum_{j \in H_i} \frac{m_j}{\rho_j}(\mathbf{u}_j - \mathbf{u}_i) \otimes \nabla_i W_{ij} + \mathbf{I}, \tag{72}$$

$$\frac{d\mathbf{v}_i}{dt} = \sum_{j \in H_i} m_j \left(\frac{\mathbf{P}_i}{\rho_i^2} + \frac{\mathbf{P}_j}{\rho_j^2}\right)\nabla_i W_{ij} + (\mathbf{F}_p + \mathbf{F}_\mu)/m_i, \tag{73}$$

where $\mathbf{F}_p$ and $\mathbf{F}_\mu$ are pressure and artificial friction forces, and $\otimes$ denotes the Kronecker product. Equation (73) uses a different form than expected from (65), as this form conserves linear and angular momentum [53]. The derivation can also be found in [53]. As in Section 4, we add the artificial friction term, because we did not include any dissipative terms in our physical model.

To compute the mass density, we use Equation (64):

$$\langle \rho_i \rangle = \sum_{j \in H_i} m_j \frac{\rho_j}{\rho_j} W(\mathbf{x}_i - \mathbf{x}_j, h) = \sum_{j \in H_i} m_j W(\mathbf{x}_i - \mathbf{x}_j, h). \tag{74}$$

External forces are applied vertex-wise, and the acceleration is computed by dividing these by the vertex mass. The only forces we account for are pressure forces and artificial friction. As before, the pressure is assumed to be uniform and working in the normal direction:

$$\mathbf{F}_{p,i} = p a_i \mathbf{n}_i, \tag{75}$$

where p is the pressure, and a_i and $\mathbf{n}_i$ are the area and normal belonging to particle i, respectively. The artificial friction depends on the current velocity of the particle and is defined as:

$$\mathbf{F}_{\mu,i} = -\mu \mathbf{v}_i = -\mu \frac{\mathbf{x}_i^t - \mathbf{x}_i^{t-1}}{\Delta t}, \tag{76}$$

where μ is the friction parameter. The velocity is discretized with backward Euler, because it computationally inexpensive and ensures that $\mathbf{F}_{\mu,i} = 0$ when particle i has not moved for two consecutive time steps. As in Section 4, we use the acceleration to compute the displacement with Verlet Integration. With respect to the maximum time step, we use the following criterion [67]:

$$\Delta t \leq \frac{1}{4} \min_i \left(\sqrt{\frac{m_i h}{3\|\mathbf{F}_s + \mathbf{F}_p\|}} \right). \tag{77}$$

Simply substituting the stress and deformation gradient tensor calculated using SPH into the governing equation, while still using the external forces as separate terms, is not standard practice. However, we argue that it is quite logical to do so in our case. Since we are simulating inflation and not, e.g., a fluid, we are certain that particles that start out near each other, for example as neighbors, will remain neighbors during the entire simulation. As there is no flow, the topology does not change. Hence, we can use this information to compute normal vectors, areas and thus pressure forces. Additionally, we also need the distance between particles. In order to accurately compute these distances, we need a mesh. See Sections 5.5, 5.6, and especially 5.6.4 for more information.

5.3.1 Vertex Quantities

For a triangular mesh, we have vertices, edges and faces. We identify the particles as used in SPH with the vertices of the mesh. Computing the normals, mass, and areas for faces needs to be done each time step, but is relatively straightforward. Let $f = \{u, v, w\}$ denote a face spanned by vertices u, v, and w, and assume the vertices are named in an anticlockwise manner. Then

$$a_f = \frac{1}{2}\|(v-u)\times(w-u)\|, \tag{78}$$

$$\mathbf{n}_f = \frac{(v-u)\times(w-u)}{\|(v-u)\times(w-u)\|}, \tag{79}$$

$$m_f = \tilde{\rho}\delta a_f, \tag{80}$$

where $\tilde{\rho}$ is the density and δ the thickness of the material. $\tilde{\rho}$ is the same for all vertices and is *not* the same as the densities computed with Equation (74). We derive the quantities at a vertex using the values at the adjacent faces, N_{fi}:

$$a_i \doteq \sum_{f\in N_{fi}} w_n a_f, \tag{81}$$

$$\mathbf{n}_i \doteq \frac{\sum_{f\in N_{fi}} a_f \mathbf{n}_f}{\|\sum_{f\in N_{fi}} a_f \mathbf{n}_f\|}, \tag{82}$$

$$m_i \doteq a_i \tilde{\rho} h, \tag{83}$$

where w_n is a weighting factor depending on the spatial dimension: $w_n = \frac{1}{n}$. A smoothed normal can be obtained by

$$\langle \mathbf{n}_i \rangle = \sum_{j\in H_i} \frac{m_j}{\rho_j} \mathbf{n}_j W_{ij}. \tag{84}$$

For the two dimensional case, we only have vertices and edges. Again we identify the particles and the vertices. Let $e = \{v, w\}$ be an edge. Then the edge length, normal and mass are

$$l_e = \|v - w\|, \tag{85}$$

$$\mathbf{n}_e = \frac{(w_2 - v_2, v_1 - w_1)}{\|(w_2 - v_2, v_1 - w_1)\|}, \tag{86}$$

$$m_e = \tilde{\rho}\delta l_e. \tag{87}$$

Similar to the three dimensional case, the quantities at the vertices are derived using the values at the adjacent edges, N_{ei}:

$$l_i \doteq \sum_{e\in N_{ei}} w_2 l_e, \tag{88}$$

$$\mathbf{n}_i \doteq \frac{\sum_{e\in N_{ei}} l_e \mathbf{n}_e}{\|\sum_{e\in N_{ei}} l_e \mathbf{n}_e\|}, \tag{89}$$

$$m_i \doteq l_i \tilde{\rho}\delta. \tag{90}$$

The smoothed normal is obtained the exact same way as before.

Finally, if we have a function $f : \mathbb{R}^n \to \mathbb{R}$ such that $\mathcal{M} = \{\mathbf{x} \in \mathbb{R}^n | f(\mathbf{x}) = 0\}$, we can obtain the normal vector for a particle as follows:

$$\mathbf{n}_i = \nabla f(\mathbf{x}_i). \tag{91}$$

The advantage is that we wouldn't need a mesh to compute the normal vectors. The disadvantage is that it is difficult to find such a function.

5.3.2 Smoothing Function Correction

One of the shortcomings of SPH is that it cannot accurately approximate even a constant function when particles are unevenly spaced or located near boundaries. The conditions for zeroth- and first order consistency (or completeness) of the SPH approximation for a function are [2, 55]:

$$\sum_{j=1}^{N} \frac{m_j}{\rho_j} W_{ij} = 1, \tag{92a}$$

$$\sum_{j=1}^{N} \frac{m_j}{\rho_j} W_{ij} \mathbf{X}_j = \mathbf{X}_i, \tag{92b}$$

and for its derivative:

$$\sum_{j=1}^{N} \frac{m_j}{\rho_j} \nabla_j W_{ij} = \mathbf{0}, \tag{93a}$$

$$\sum_{j=1}^{N} \frac{m_j}{\rho_j} \mathbf{X}_j \otimes \nabla_j W_{ij} = \mathbf{I}. \tag{93b}$$

In order to fulfill first order completeness of the gradient, a correction matrix is computed. It is computed as follows [2, 56]:

$$\tilde{\nabla}_i W_{ij} = \mathbf{M}_i \nabla_i W_{ij}, \tag{94}$$

where $\mathbf{M}_i$ is such that

$$\sum_{i \in H_i} V_j (\mathbf{X}_j - \mathbf{X}_i) \otimes \tilde{\nabla}_i W_{ij} = (\sum_{i \in H_i} V_j (\mathbf{X}_j - \mathbf{X}_i) \otimes \nabla_i W_{ij}) \mathbf{M}_i^T = I. \tag{95}$$

The corrected equations become:

$$\mathbf{F}_i = \sum_{j \in H_i} \frac{m_j}{\rho_j} (\mathbf{u}_j - \mathbf{u}_i) \otimes \tilde{\nabla}_i W_{ij} + \mathbf{I}, \tag{96}$$

$$\frac{d\mathbf{v}_i}{dt} = \sum_{j \in H_i} m_j (\frac{\mathbf{P}_i}{\rho_i^2} + \frac{\mathbf{P}_j}{\rho_j^2}) \tilde{\nabla}_i W_{ij} + (\mathbf{F}_p - \mathbf{F}_\mu)/m_i, \tag{97}$$

where $\mathbf{M}_i$ needs to be recomputed every time step.

5.4 Adaptive Smoothing Lengths and Densities

As we work with shapes that expand during the simulation, the smoothing length gets relatively smaller to the size of the shape. With enough force pushing particles away, at some point they will not have any neighbors left, resulting in a density equal to zero. Here, look at methods to make the smoothing length and the density of the particle dependent on each other. Instead of just one h for all particles, we get $h(\mathbf{x}_i)$.

It is possible to couple the smoothing length to the density [53, 64]:

$$h(\mathbf{x}_i) = \eta \left(\frac{m_i}{\rho_i} \right)^{1/d}. \tag{98}$$

This uses the definition of the volume around a particle. η is a constant, and usually chosen such that $1.2 \leq \eta \leq 1.5$. Together with Equation (74), we have a system of equations to solve.

Another option is to take Equation (98) and combine it with different definitions for the density. We look at a solution from [61], which takes care of the density deficiency at boundaries. First, they compute the densities as in (74). Then they use these ρ_j as input to

$$\tilde{\rho}_i = \frac{\sum_{j \in H_i} m_j W_{ij}}{\sum_{j \in H_i} m_j / \rho_j W_{ij}}. \tag{99}$$

Finally, [2] provides the following option

$$\rho_i = \frac{\rho_o J_0}{J}, \tag{100}$$

where ρ_0 and J_0 are the initial mass density and Jacobian determinant and the latter can be computed as

$$J = \det(\mathbf{F}). \tag{101}$$

It is not immediately obvious, but this definition of ρ_i also depends on h_i, by the dependence of ∇W on h_i.

For all options, we can use a root finding algorithm to find ρ_i and h_i. Also, due to the dependence of ρ_i on h_i, we obtain a correction factor [64, p. 73]:

$$\omega_i = 1 - \sum_{j \in H_i} \frac{\partial W(\mathbf{x}_i - \mathbf{x}_j, h_i)}{\partial h_i} \frac{\partial h_i}{\partial \rho_i} m_j \tag{102}$$

Equations (72) and (73) with the correction become:

$$\mathbf{F}_i = \sum_{j \in H_i} \frac{m_j}{\rho_j} \left(\mathbf{u}_j \otimes \nabla_i W_{ij}(h_j) - \mathbf{u}_i \otimes \nabla_i W_{ij}(h_i) \right) + \mathbf{I}, \tag{103}$$

$$\frac{d\mathbf{v}_i}{dt} = \sum_{j \in H_i} \left(\frac{\mathbf{P}_i}{\rho_i^2 \omega_t} \nabla_i W_{ij}(h_i) + \frac{\mathbf{P}_j}{\rho_j^2 \omega_j} \nabla_i W_{ij}(h_j) \right) m_j + (\mathbf{F}_p - \mathbf{F}_\mu)/m_i. \tag{104}$$

5.5 Neighbor Search

SPH requires the distances between particles as input for the kernel function. These have to be recomputed each time step, as generally all particles can move to any part of the domain. A naive way to do this, is by computing for each vertex the distances to all other vertices, resulting in an operation of $\mathcal{O}(n_v^2)$. There are many ways to reduce this, but we will explain two often used methods, the cell-linked list (CLL) and the Verlet list (VL).

In CLL the computational domain is divided in cells larger than the smoothing length. Each particle belongs to one cell and each time step, an array is created which contains the cell index for each particle. When searching for neighbors, only the particles in the same and neighboring cells are considered. So for each vertex, it is easy to find *potential* neighbors. In 2 dimensions, only 9 cells need to be searched and in 3 dimensions 27 cells.

When using VL, the domain is again subdivided into cells with a length larger than the smoothing length. The array containing the cell index for each particle is also created again. However, after this step, the *real* neighbors of each vertex are stored in an array by searching the adjacent cells for vertices within the smoothing length.

The main difference between the two, is that during computation of the forces, in VL only particles actually contributing to these forces are considered, whereas in CLL also particles are considered that in the end do not contribute. For VL, it is also possible to not only take the actual neighbors, but also particles that are a little bit further away. This way, the list can be kept for several time steps, during which no neighbor search has to be executed. Domínguez et al. [68] analyzed the efficiency and memory requirements of the two algorithms combined with different ways of storing the location in the grid of the particles, as well as an improved VL method. They found that for both methods the same storage algorithm worked best. It consists of storing the particle indices in an array such that indices of particles in one cell are grouped together. Another array contains at index i the first index of

a particle in cell i. Furthermore regarding the efficiency, VL required much more memory. They recommend their improved version only when running a parallel code, as the runtime improvement of 10% does not outweigh the additional memory needed. Finally, they find that reordering the particles such that adjacent particles are are close in memory improves performance for both CLL and VL.

In the present work, we use neither, since the topology does not change. So we start with a list of all (possible) neighbors, and keep it during the entire inflation period. We obtain the list of neighbors by doing one search where we store all neighbors within 2.5 times the smoothing length. Because we compute the distance to all neighbors in the list every time step, our neighbor search can be seen as a variation on the VL, where we keep a list of potential neighbors during the entire simulation. See Section 5.6.4 for more details on how the distance computation is done.

5.6 SPH on a Manifold

As mentioned in the first paragraph of Section 5, SPH is usually used to describe particle interactions in e.g. fluid simulations. There are two important differences between those applications and ours. The first is that we have a surface with a fixed topology; neighboring particles in the initial configuration stay neighbors during the entire simulation. Secondly, if we let n denote the spatial dimension, then we model the preform as a surface of $n-1$ dimensions. The third difference is the function to measure distances between particles. The standard is the Euclidean distance, which measures the length of a straight line between particles. However, since surfaces can be curved, the Euclidean distance might severely underestimate the geodesic distance between two points on the surface. The result is that some particles effect on each other is larger than it would have been with the geodesic distance. This section makes adjustments to the standard SPH formulation as stated in Section 5.2 to account for these differences.

5.6.1 SPH Formulation

Let $\mathcal{M} \subset \mathbb{R}^n$ be the middle surface of the preform. Suppose the function $\Phi : S \to \mathcal{M}$ mapping every point $\mathbf{s} \in S \subset \mathbb{R}^{n-1}$ to a point $\mathbf{x} \in \mathcal{M}$ exists. Denote the function measuring the distance between two points on the surface by $d(\mathbf{s})$. Instead of $W(\mathbf{x})$, the kernel function becomes

$$W(\mathbf{s}, h) = \frac{c_{n-1}}{h^{n-1}} w(q), \quad \text{where now } q = \frac{d(\mathbf{s})}{h}. \tag{105}$$

Of course, we also need the gradient of W. Let $\frac{\partial d(\mathbf{s})}{\partial \mathbf{s}}$ denote the gradient of $d(\mathbf{s})$, then the gradient is the following vector on S

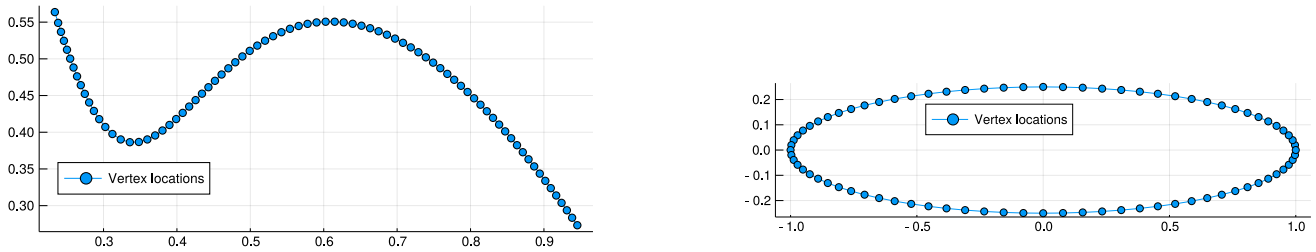
$$\begin{aligned} \nabla W(\mathbf{s}, h) &= \frac{\partial W(\mathbf{s}, h)}{\partial \mathbf{s}} = \frac{c_{n-1}}{h^{n-1}} \frac{\partial w(q)}{\partial \mathbf{s}} = \frac{c_{n-1}}{h^{n-1}} \frac{\partial w}{\partial q} \frac{\partial q}{\partial d(\mathbf{s})} \frac{\partial d(\mathbf{s})}{\partial \mathbf{s}} \\ &= \frac{c_{n-1}}{h^n} \frac{\partial w}{\partial q} \frac{\partial d(\mathbf{s})}{\partial \mathbf{s}}. \end{aligned} \tag{106}$$

What we actually want is the gradient of $W(\mathbf{s}, h)$ with respect to the coordinates in the space that $\mathcal{M}$ is in, the same space where the acceleration and deformation gradient tensor live. We denote this gradient by ∇_x and the one in S by ∇_s. We get

$$\begin{aligned} \nabla_x W(\mathbf{s}, h) &= \frac{\partial W(\mathbf{s}, h)}{\partial \mathbf{x}} = \frac{c_{n-1}}{h^{n-1}} \frac{\partial w(q)}{\partial \mathbf{x}} = \frac{c_{n-1}}{h^{n-1}} \frac{\partial w}{\partial q} \frac{\partial q}{\partial d(\mathbf{s})} \frac{\partial d(\mathbf{s})}{\partial \mathbf{s}} \frac{\partial \mathbf{s}}{\partial \mathbf{x}} \\ &= \frac{c_{n-1}}{h^n} \frac{\partial w}{\partial q} \frac{\partial d(\mathbf{s})}{\partial \mathbf{s}} \frac{\partial \mathbf{s}}{\partial \mathbf{x}}. \end{aligned} \tag{107}$$

This presents a problem, because $\frac{\partial \mathbf{s}}{\partial \mathbf{x}} = \frac{\partial \Phi^{-1}(\mathbf{x})}{\partial \mathbf{x}}$ does not necessarily exist for all $\mathbf{x} \in \mathbb{R}^n$, because $\Phi^{-1}(\mathbf{x})$ does not exist for all $\mathbf{x} \in \mathbb{R}^n$. However, $\frac{\partial \mathbf{x}}{\partial \mathbf{s}} = \frac{\partial \Phi(\mathbf{s})}{\partial \mathbf{s}}$ does exist for all $\mathbf{s} \in S$. We can use the Moore-Penrose inverse to obtain values for $\frac{\partial \mathbf{s}}{\partial \mathbf{x}}$. We denote the Moore-Penrose inverse of a matrix A, by A^+, then

$$\nabla_x W(\mathbf{s}, h) = \frac{c_{n-1}}{h^n} \frac{\partial w}{\partial q} \frac{\partial d(\mathbf{s})}{\partial \mathbf{s}} \frac{\partial \mathbf{s}}{\partial \mathbf{x}} \approx \frac{c_{n-1}}{h^n} \frac{\partial w}{\partial q} \frac{\partial d(\mathbf{s})}{\partial \mathbf{s}} \frac{\partial \mathbf{x}}{\partial \mathbf{s}}^+. \tag{108}$$



(a) *An open discrete curve.*

(b) *A closed discrete curve.*

Figure 28 *Two examples of a discrete curve $\mathcal{M} \in \mathbb{R}^2$.*

The Moore-Penrose inverse can be computed in a number of ways. Let A be an $l \times k$ matrix. If $k = 1$, we compute the Moore-Penrose inverse of a vector as follows:

$$A = \begin{pmatrix} a_1 & a_2 & \dots & a_n \end{pmatrix}^T,$$
$$A^+ = \frac{1}{A^T A} A^T. \tag{109}$$

If $n = 2$, then A is a 3×2 matrix. If it has linearly independent columns or rows, we can easily compute A^+:

$$A^+ = (A^T A)^{-1} A^T, \quad \text{independent columns,} \tag{110a}$$
$$A^+ = A^T (A A^T)^{-1}, \quad \text{independent rows.} \tag{110b}$$

Finally, if neither the rows nor columns are independent, then AA^T nor $A^T A$ is invertible. In that case, we can use the singular value decomposition of A. If $A = U\Sigma V^T$, then $A^+ = V\Sigma^+ U^T$. Σ is an $n \times n$ diagonal matrix and the elements of Σ^+ are the reciprocal values of the non-zero diagonal entries of Σ.

5.6.2 Gradient of the Manifold's Kernel in 2D

Of course, given a mesh in $\mathbb{R}^n$, we do not explicitly have the parametrization Φ. If $n = 2$, we can use a function giving us the distance between two points to make a parametrization. So let $\mathcal{M}$ be a discrete curve in $\mathbb{R}^2$ with n_v vertices. See Figure 28 for an example of an open and closed discrete curve. Suppose we have a distance function $\tilde{d} : \mathcal{M} \times \mathcal{M} \to \mathbb{R}$, with which we can compute distances between two given points in $\mathcal{M}$. We now fix one point and call it $\mathbf{x}_1$. If $\mathcal{M}$ is an open curve, choose one of the two end points and number the other points. If $\mathcal{M}$ is a closed curve, choose any point to call $\mathbf{x}_1$ and number the rest in a clockwise fashion. Then we make the parametrization as follows

$$s_1 = \tilde{d}(\mathbf{x}_1) = \tilde{d}(\mathbf{x}_1, \mathbf{x}_1) = 0, \tag{111a}$$
$$s_i = \tilde{d}(\mathbf{x}_i) = \tilde{d}(\mathbf{x}_i, \mathbf{x}_1). \tag{111b}$$

We know that $s_i \geq 0,\ \forall i \in \{1, 2, \dots, n_v\}$, because of the way we chose $\mathbf{x}_1$ and numbered the points. We can now also make a distance function $d : S \to \mathbb{R}$. We do have to distinguish between closed and open curves. Let L denote the total length of the curve. Then the shortest distance between two points i and j in S is

$$d(s) = \begin{cases} |s|, & \text{if } S \text{ is open,} \\ \min\left(|s|, L - |s|\right) & \text{if } S \text{ is closed,} \end{cases} \tag{112a}$$
$$d(s_i - s_j) = \begin{cases} |s_i - s_j|, & \text{if } S \text{ is open,} \\ \min\left(|s_i - s_j|, L - |s_i - s_j|\right) & \text{if } S \text{ is closed.} \end{cases} \tag{112b}$$

The derivatives are

$$\frac{\partial d(s)}{\partial s} = \frac{s}{|s|}, \text{ if } S \text{ is open,} \tag{112c}$$

$$\frac{\partial d(s)}{\partial s} = \begin{cases} \frac{s}{|s|}, \text{ if } |s| \leq \frac{L}{2}, \\ -\frac{s}{|s|}, \text{ if } |s| > \frac{L}{2}, \end{cases} \text{ if } S \text{ is closed.} \tag{112d}$$

For the closed curve, the two options for the shortest distance correspond to "walking" over the curve from vertex i to j in a clockwise and anti-clockwise manner.

The kernel function is:

$$W(s,h) = \frac{c_1}{h^1} w(q), \quad \text{where } q = \frac{d(s)}{h}, \tag{113}$$

and its gradient:

$$\nabla_x W(s,h) = \frac{c_1}{h^2} \frac{\partial w}{\partial q} \frac{\partial d(s)}{\partial s} \frac{\partial \mathbf{x}}{\partial s}^+, \tag{114a}$$

so the only thing left is an expression for $\frac{\partial \mathbf{x}}{\partial s}^+$. When we need to know the value of $\nabla_{x,i} W(\mathbf{x})$, we have the locations of two vertices: $\mathbf{x}_i$ and $\mathbf{x}_j$. So we use these to approximate the derivative of $\mathbf{x}_i$ with respect to s_i:

$$\frac{\partial \mathbf{x}_i}{\partial s_i} \approx \frac{\mathbf{x}_i - \mathbf{x}_j}{s_i - s_j}.$$

Then the Moore-Penrose inverse is:

$$\frac{\partial \mathbf{x}}{\partial s}^+ = \frac{s_i - s_j}{\|\mathbf{x}_i - \mathbf{x}_j\|^2} (\mathbf{x}_i - \mathbf{x}_j).$$

Thus

$$\nabla_x W(s_i - s_j, h) = \frac{\sigma c_1}{h^2} \frac{|s_i - s_j|}{\|\mathbf{x}_i - \mathbf{x}_j\|^2} (\mathbf{x}_i - \mathbf{x}_j) \frac{\partial w}{\partial q} \tag{115}$$

where $\sigma = 1$ if $\mathcal{M}$ is open and if $\mathcal{M}$ is closed $\sigma = 1$ if $|s_i - s_j| \leq \frac{L}{2}$ and $\sigma = -1$ if $|s_i - s_j| > \frac{L}{2}$ and $q = d(s_i - s_j)/h$. We note that the distance between particles usually is much smaller than the total length of the curve, so σ should practically never be -1. Finally, we can write $\tilde{d}(\mathbf{x}_j)$ instead of s_j, so we get an expression in terms of $\mathbf{x}$ only:

$$\nabla_x W(\mathbf{x}_i - \mathbf{x}_j, h) = \frac{\sigma c_1}{h^2} \frac{|\tilde{d}(\mathbf{x}_i) - \tilde{d}(\mathbf{x}_j)|}{\|\mathbf{x}_i - \mathbf{x}_j\|^2} (\mathbf{x}_i - \mathbf{x}_j) \frac{\partial w}{\partial q} \tag{116}$$

where $q = d(\tilde{d}(\mathbf{x}_i) - \tilde{d}(\mathbf{x}_j))/h$.

5.6.3 Gradient of the Manifold's Kernel in 3D

For $\mathcal{M}$ a surface in $\mathbb{R}^3$, we cannot use a distance function to make a parametrization as we did in the two dimensional case. Hence we turn to an algorithm that can make a parametrization for us. It is called As-Rigid-As-Possible parametrization (ARAP) [69]. It makes a parametrization of a triangular mesh, where it strives to preserve shape and size as much as possible. Let $\Phi : \mathcal{M} \to S$ be the mapping given bij the ARAP parametrization. Then for each $\mathbf{x} \in \mathcal{M}$, we have a corresponding $\mathbf{s} \in S : \mathbf{s} = \Phi(\mathbf{x})$.

Like before, we need an expression for the distance function. On S, we have

$$d(\mathbf{s}) = \|\mathbf{s}\|, \quad \text{and} \quad \frac{\partial d(\mathbf{s})}{\partial \mathbf{s}} = \frac{\mathbf{s}}{\|\mathbf{s}\|}. \tag{117}$$

However, suppose we also have $d : \mathcal{M} \times \mathcal{M} \to \mathbb{R}$, giving us the geodesic distance between two points on $\mathcal{M}$. If the mapping Φ is isometric, it preserves distances. Then $\|\mathbf{s}_i - \mathbf{s}_j\| = d(\mathbf{x}_j, \mathbf{x}_i)$. This was the case in Section 5.6.2. If

the mapping is not isometric, but conformal, then the angles between intersecting curves on $\mathcal{M}$ are preserved on S. Locally, it also introduces a uniform stretch around a point. So given a circle on $\mathcal{M}$, a conformal map maps it to a circle on S with a different radius. This stretch can be represented by a constant number: the conformal factor [38]. It is denoted by λ^2 and differs per point on the surface. It can be computed at vertex i as follows:

$$\lambda_i = \frac{1}{n_i} \sum_{e_{ij} \in \mathcal{M}} \frac{\|e_{ij}\|}{\|e'_{ij}\|}, \tag{118}$$

where e_{ij} denotes an edge between vertex i and j in $\mathcal{M}$ and e'_{ij} an edge between vertex i and j in S. n_i is the number of incident edges of vertex i. The influence of this stretching, is that particles' interactions on each other get over- or underestimated when we use a global smoothing length. [38] states a way to correct the smoothing length per vertex:

$$h_i = \frac{h}{\lambda_i}. \tag{119}$$

However, as we assume we have the geodesic distance on $\mathcal{M}$, we can use this distance with the global smoothing length

$$q = \frac{d(\mathbf{s}_i)}{h_i} = \frac{\lambda_i d(\mathbf{s}_i)}{h} = \frac{d(\Phi^{-1}(\mathbf{s}_i))}{h}. \tag{120}$$

So the kernel function is:

$$W(\mathbf{s}, h) = \frac{c_2}{h^2} w(q), \quad \text{where } q = \frac{d(\Phi^{-1}(\mathbf{s}))}{h}. \tag{121}$$

The gradient of the kernel function is:

$$\nabla_x W(\mathbf{s}, h) = \frac{c_2}{h^3} \frac{\partial w}{\partial q} \frac{\partial d(\mathbf{s})}{\partial \mathbf{s}} \frac{\partial \Phi^{-1}(\mathbf{x})}{\partial \mathbf{x}}. \tag{122}$$

Again it is true the when we need to know the value of $\nabla_{x,i} W(\mathbf{x})$, we are looking at two vertices: $\mathbf{x}_i$ and $\mathbf{x}_j$. So we approximate the Jacobian of the derivative of Φ with respect to $\mathbf{s}$:

$$\frac{\partial \mathbf{x}}{\partial \mathbf{s}} = \begin{pmatrix} \frac{\partial \mathbf{x}^1}{\partial \mathbf{s}^1} & \frac{\partial \mathbf{x}^1}{\partial \mathbf{s}^2} \\ \frac{\partial \mathbf{x}^2}{\partial \mathbf{s}^1} & \frac{\partial \mathbf{x}^2}{\partial \mathbf{s}^2} \\ \frac{\partial \mathbf{x}^3}{\partial \mathbf{s}^1} & \frac{\partial \mathbf{x}^3}{\partial \mathbf{s}^2} \end{pmatrix} \approx \begin{pmatrix} \frac{\mathbf{x}_i^1 - \mathbf{x}_j^1}{\mathbf{s}_i^1 - \mathbf{s}_j^1} & \frac{\mathbf{x}_i^1 - \mathbf{x}_j^1}{\mathbf{s}_i^2 - \mathbf{s}_j^2} \\ \frac{\mathbf{x}_i^2 - \mathbf{x}_j^2}{\mathbf{s}_i^1 - \mathbf{s}_j^1} & \frac{\mathbf{x}_i^2 - \mathbf{x}_j^2}{\mathbf{s}_i^2 - \mathbf{s}_j^2} \\ \frac{\mathbf{x}_i^3 - \mathbf{x}_j^3}{\mathbf{s}_i^1 - \mathbf{s}_j^1} & \frac{\mathbf{x}_i^3 - \mathbf{x}_j^3}{\mathbf{s}_i^2 - \mathbf{s}_j^2} \end{pmatrix}, \tag{123}$$

where the exponent denotes the i^{th} component of $\mathbf{s}$ or $\mathbf{x}$. Since neither the rows or the columns of $\frac{\partial \mathbf{x}}{\partial \mathbf{s}}$ are linearly independent, we have to use the singular value decomposition to compute the the Moore-Penrose inverse. The gradient of the kernel is

$$\nabla_{x,i} W(\mathbf{s}_i - \mathbf{s}_j, h) = \frac{c_2}{h^3} \frac{\mathbf{s}_i - \mathbf{s}_j}{\|\mathbf{s}_i - \mathbf{s}_j\|} \begin{pmatrix} \frac{\mathbf{x}_i^1 - \mathbf{x}_j^1}{\mathbf{s}_i^1 - \mathbf{s}_j^1} & \frac{\mathbf{x}_i^1 - \mathbf{x}_j^1}{\mathbf{s}_i^2 - \mathbf{s}_j^2} \\ \frac{\mathbf{x}_i^2 - \mathbf{x}_j^2}{\mathbf{s}_i^1 - \mathbf{s}_j^1} & \frac{\mathbf{x}_i^2 - \mathbf{x}_j^2}{\mathbf{s}_i^2 - \mathbf{s}_j^2} \\ \frac{\mathbf{x}_i^3 - \mathbf{x}_j^3}{\mathbf{s}_i^1 - \mathbf{s}_j^1} & \frac{\mathbf{x}_i^3 - \mathbf{x}_j^3}{\mathbf{s}_i^2 - \mathbf{s}_j^2} \end{pmatrix}^+ \frac{\partial w}{\partial q} \tag{124}$$

where $\mathbf{s}_i - \mathbf{s}_j$ is a 1×2 row vector and $q = \frac{d(\mathbf{x}_i, \mathbf{x}_j)}{h}$.

The parametrization also effects the computation of the density. Recall Equation (74):

$$\langle \rho_i \rangle = \sum_{j \in H_i} m_j W(\mathbf{x}_i - \mathbf{x}_j, h). \tag{74, revisited}$$

Here, it is assumed that W lives on $\mathcal{M}$. However, we only have the smoothing kernel as a function of $\mathbf{s}$. This means that we have to adjust the equation to account for the fact that we want ρ_i on $\mathcal{M}$. We can use the conformal factor to obtain ρ_i on $\mathcal{M}$ [38]:

$$\langle \rho_i \rangle = \sum_{j \in H_i} \frac{m_j}{\lambda_j^2} W(\mathbf{s}_i - \mathbf{s}_j, h). \tag{125}$$

It is important to remark that this density still is for a surface. So while it is corrected for differing length measures on $\mathcal{M}$ and S, it is a quantity in kg/m^2. If $\tilde{\rho}$ is the material density (in kg/m^3), and we assume an incompressible material model is used, we can retrieve the thickness in a vertex as follows

$$\delta_i = \frac{\rho_i}{\tilde{\rho}}. \tag{126}$$

Finally, we note that the ARAP parametrization algorithm that we use expects a mesh with disk topology as input. So for every vertex, we create a list of neighboring vertices, at a distance of at most three times the smoothing length, and the faces that belong to these vertices. This submesh will be used as input for the ARAP parametrization algorithm. In this case, it might mean that edges present in $\mathcal{M}$ are not in S. For the computation of λ, this means that these edges are discarded from the computation.

5.6.4 Geodesic Distance Computation

For the formulation of SPH on a manifold, we use a function d to compute the geodesic distance, which is the length of a geodesic or shortest path between two points. Would we use the Euclidian distance, which measures a straight line, as measure for the distance between vertices, then we underestimate the true distance between two vertices. The result is that $q = \frac{\|\mathbf{x}_i - \mathbf{x}_j\|}{h}$ is smaller as well, and that more particles contribute to, e.g., the density of particle i.

There are several algorithms available to compute shortest distances on meshes. One option is to use Dijkstra's algorithm for weighted graphs. However, this will result in an overestimation of the distance, as the shortest path rarely goes over the edges between vertices. Then there are a couple of often used algorithms, that solve the eikonal equation,

$$|\nabla \phi(x)| = 1/f(x), \quad x \in \Omega, \tag{127a}$$
$$\text{subject to } \phi(x) = q(x), \; x \in \partial\Omega. \tag{127b}$$

$\phi(x)$ can be interpreted as the minimal amount of time required to travel from x to $\partial\Omega$, with travel speed $f(x)$ and exit-time penalty $q(x)$. The geodesic distance function is the special case where $f(x) = 1$ and $q(x) = 0$. Note that $\partial\Omega$, called the source, is not necessarily the actual boundary of Ω, but the set containing the points to which the distance is needed.

The first is MMP, introduced by Mitchell, Mount and Papadimitriou in 1987 [70]. An implementation was published later [71], which presents an approximate $\mathcal{O}(n_v \log(n_v))$ version. To compute the shortest distance, a wave is propagated from the source over the surface. The first time this wave hits a point, x, on the surface, the point is labeled with the time, $d(x)$, at which it received the wave. This time is the minimum distance from the source to x. The algorithm uses a finite difference equation to solve the eikonal equation and does not reuse information, so even if the geometry does not change, the algorithm takes just as long as when it does change. It also requires a triangulated domain.

The next two method is fast marching [72]. The methods considers a boundary (i.e. the source) moving in its normal direction with a known speed function. If the boundary crosses a point x, the time, and hence the distance, is known. The algorithm is similar to Dijkstra's in structure. It also needs a grid. Another often used algorithm is the fast sweeping method [73]. It uses an upwind difference scheme to discretize the equation and uses Gauss-Seidel iterations to solve the equation. It needs a rectangular grid.

A more recent algorithm is the heat method [74]. It separates solving the eikonal equation into two parts. First it finds the direction in which the distance increases and then it computes the actual distance. It does not require a regular grid or mesh and is faster than fast marching. It also computes the distance from the source to all other points by solving one linear system. Finally, it can reuse information for a given mesh, as it used a matrix in the first step that is the same for any vertex. So it is possible to factorize that matrix and reuse it for all other vertices to solve the first step. See Figure 29 for the 3D model Spot as an example. While a mesh is not required, we do use it, because it is the easiest way to implement the algorithm.

Besides solving the eikonal equation, another option is to fit a shape on which it is easier to compute distances to a set of points. The shape is generally a sphere or ellipse. Then the distance between two points is equal to the distance between their projections on the known shape.

We note that the two dimensional case is much simpler. Let $\mathcal{M} \subset \mathbb{R}^2$ be a discrete curve. We can simply define the geodesic distance between vertex i and j as the sum of Euclidean distances of the vectors between the vertices:

$$d(x_i, x_j) = \begin{cases} \sum_{l=i}^{j-1} \|x_{l+1} - x_l\| & \text{if } i < j, \\ \sum_{l=j}^{i-1} \|x_l - x_{l+1}\| & \text{if } i > j. \end{cases} \tag{128}$$

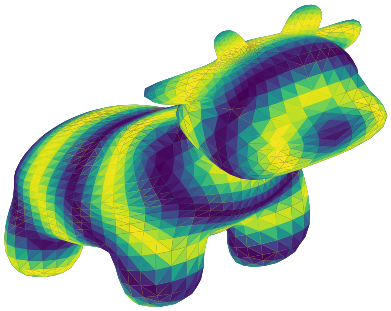


Figure 29 *Distances on Spot to a point on its nose, as computed with the heat method [74]. The colors indicate isolines, where points on the same colored circle have the same distance to the point on the nose.*

5.7 Results

In this section we present the results of the simulations. We compare FEM, SPH, and the Young-Laplace equation to each other. We show 2D and 3D results of SPH, and finally we investigate the NN for distance computations. We assume that the preform material behaves like a balloon, so we use the properties listed in Table 2.

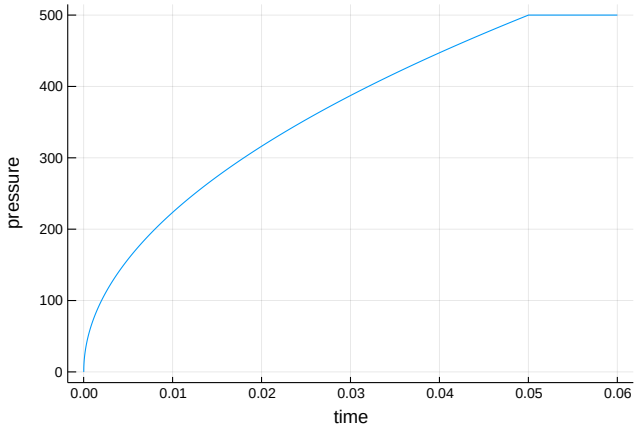
5.7.1 2D Results

We use the Saint Venant-Kirchhoff material law, as it is easy to implement in the two dimensional case and we know the material parameters.

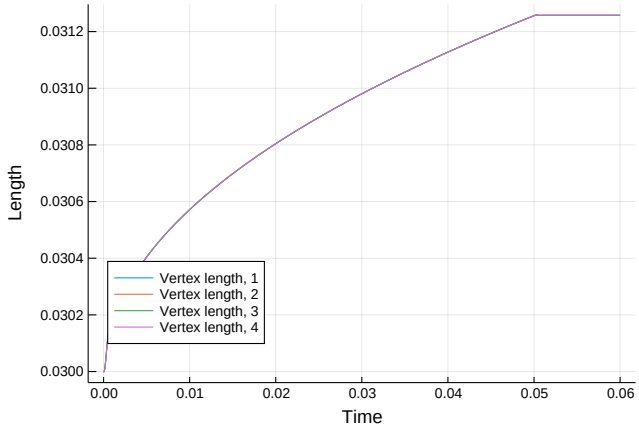
First, we show some results of inflating a circle with radius 1, see Figure 30. We slowly increase the pressure, as can be seen in Figure 30a. For four different points on the circle, we plot the magnitude of the forces, as well as the vertex length. They can be seen in Figures 30b and 30c. We also plot the final shape. The reason that it seems like the values for only one point were plotted, is because they are actually identical for all points. The final shape is still a circle, but with a slightly bigger radius.

Quantity	Symbol	Value	Unit
Young's Modulus	E	$1 \cdot 10^6$	Pa
Poisson's ratio	ν	$.48$	1
1^{st} Lamé parameter	λ	$8.11 \cdot 10^6$	Pa
2^{nd} Lamé parameter	μ	$9.62 \cdot 10^5$	Pa
Thickness	δ	$2.54 \cdot 10^{-4}$	m

Table 2 *Table containing relevant material properties for a silicone rubber balloon [75].*



(a) *Pressure with respect to the time.*



(b) *Length of the vertices.*

When we try exactly the same thing with the cubic B-spline kernel, the result is quite disastrous, as can be seen in Figure 31. We see in Figure 31c that sometime after 0.03 s, the pressure and elastic forces start to increase and decrease. A little bit later, all forces start to heavily fluctuate. When we look at the particle configuration, we see that it looks nothing like a circle anymore. We look at the configuration before everything goes awry in Figure 32 and there we can see that the particles have moved towards each other.

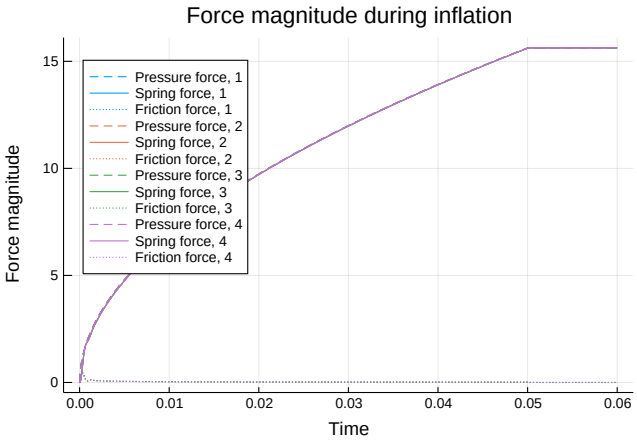
A possible explanation is that we have found the tensile instability often seen in SPH [54, 76]. Swegle et al. carried out a stability analysis and found that the tensile instability depends on the sign of the product of the stress, T, times the second derivative of the kernel function, W'' [77]. The condition for unstable growth is $W''T > 0$. Importantly, the instability is not caused by the time integration algorithm, nor are there stress or strain thresholds for the onset of the instability.

Whether or not we use the correction from Equation (94), does not make a difference for the instability. In Figures 34 and 35 the results of inflating an ellipse with the Wendland kernel can be seen, respectively without and with the correction applied. As it does not seem to resolve the tensile instability, we do not apply it from here on. Additionally, we only use the Wendland kernel from here on, as it suffers less from the instability.

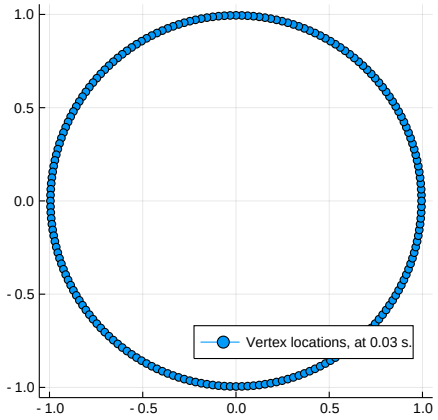
We can also show that the instability is indeed independent of the pressure, stress, and strain. We do so by showing that for a higher pressure than used in Figure 34, the elastic forces are higher, as well as the strains, while the instability sets in later. This can be seen in Figure 36.

We also look at the effect of using an adaptive smoothing length. We solve the system of Equations 74 and 98 to find h_i and ρ_i. In the subfigures of Figure 37 it can be seen that not only does it not help for stability when inflating a circle, it makes matters worse, because we previously were able to inflate this configuration without any trouble.

We conclude that inflating anything to static equilibrium is difficult. We can however, stop the process before the tensile instability sets in. Also, by looking at Figure 36d and 39d we see that at approximately static equilibrium, both the ellipse and squircle resemble a circle, as expected.

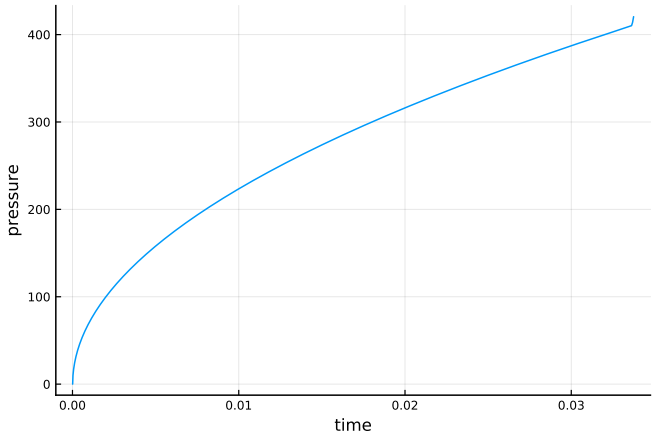


(c) *Force magnitude in four points, which all coincide in this case.*

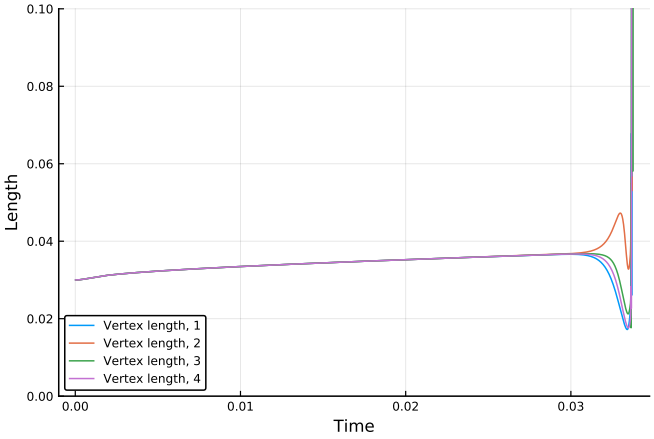


(d) *Final shape.*

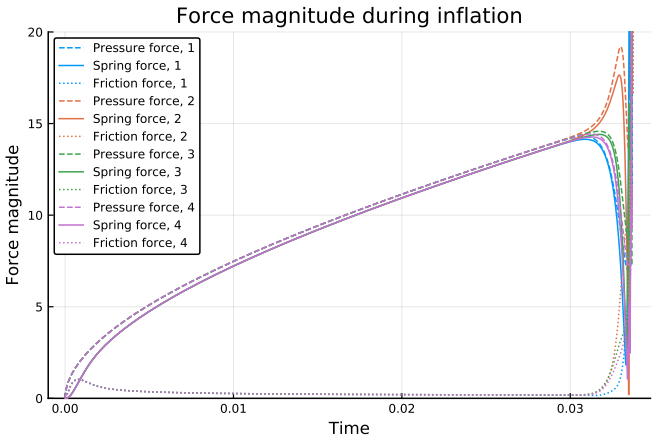
Figure 30 *The result of inflating a circle with the Wendland kernel. Parameters are:* $\Delta t = 1 \cdot 10^{-}5$, $\mu = 0.04$, *and* $h = 0.4$.



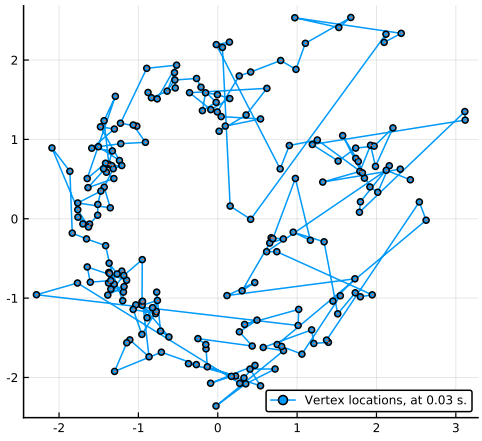
(a) *Pressure with respect to the time.*



(b) *Length of the vertices.*



(c) *Force magnitude in four points.*



(d) *Final shape.*

Figure 31 *The result of inflating a circle with the cubic B-spline kernel. Parameters are:* $\Delta t = 1 \cdot 10^{-}5$, $\mu = 0.04$.

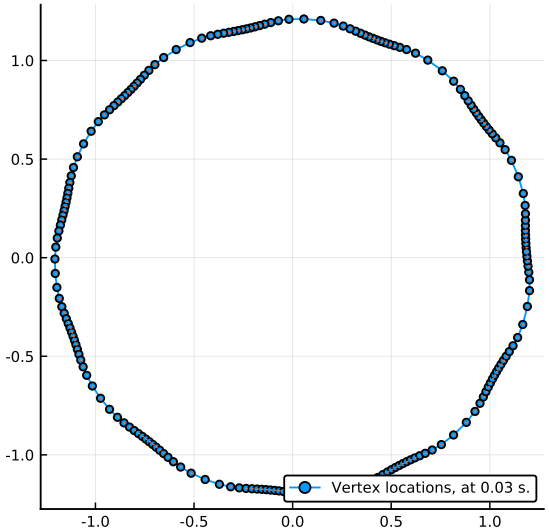


Figure 32 *The configuration of the circle during inflation as $t = 0.033s$ before the fluctuations in the forces.*

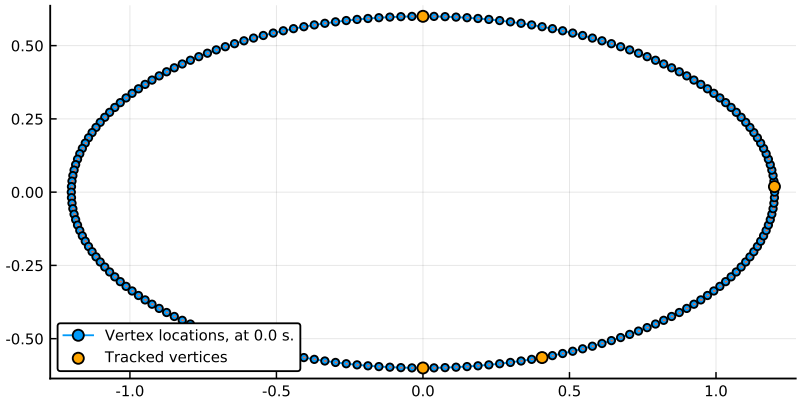
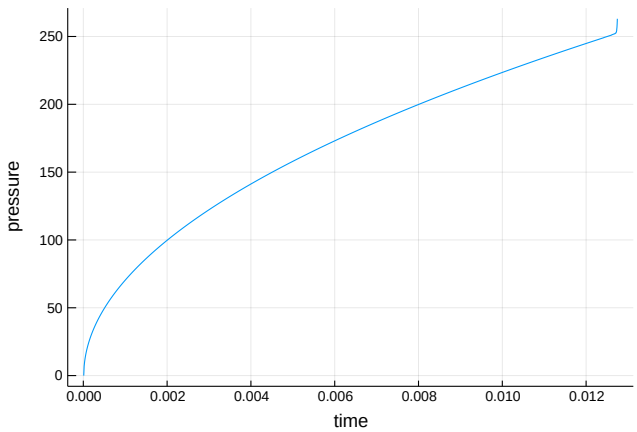
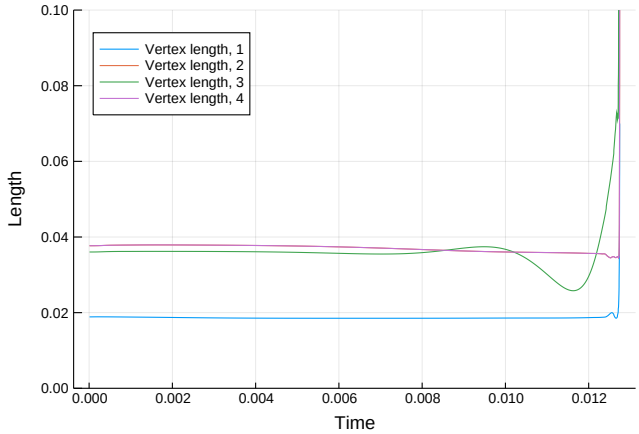


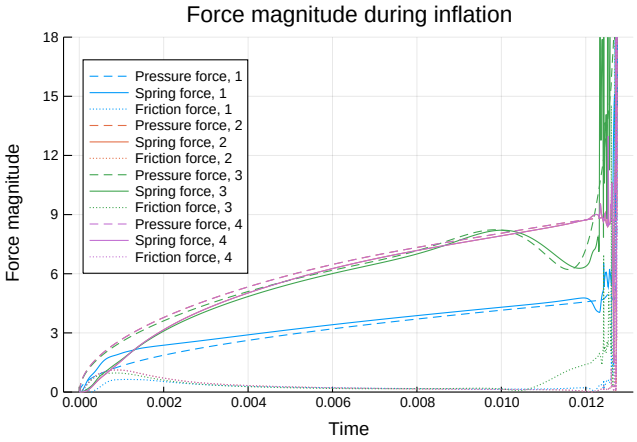
Figure 33 *The initial configuration of the ellipse, with locations of the tracked vertices indicated. Vertex 1 is the rightmost point, the rest is numbered anti-clockwise.*



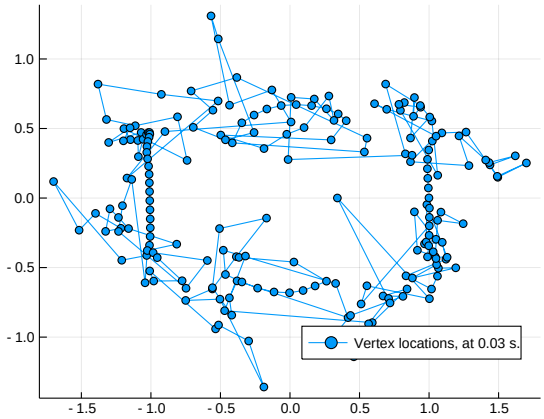
(a) *Pressure with respect to the time.*



(b) *Length of the vertices.*

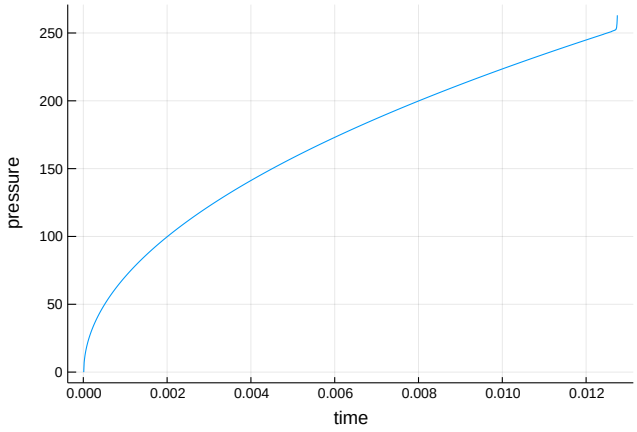


(c) *Force magnitude in four points.*

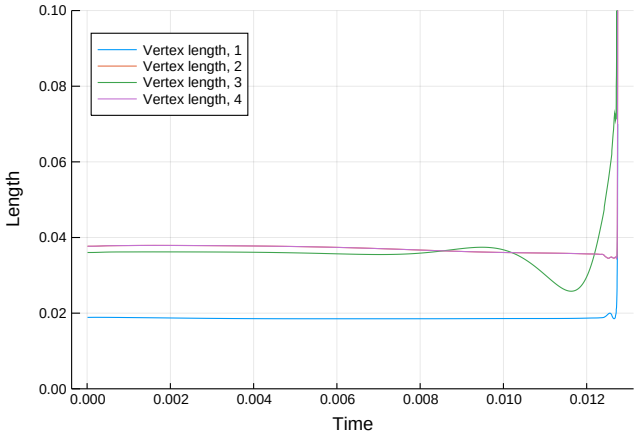


(d) *Final shape.*

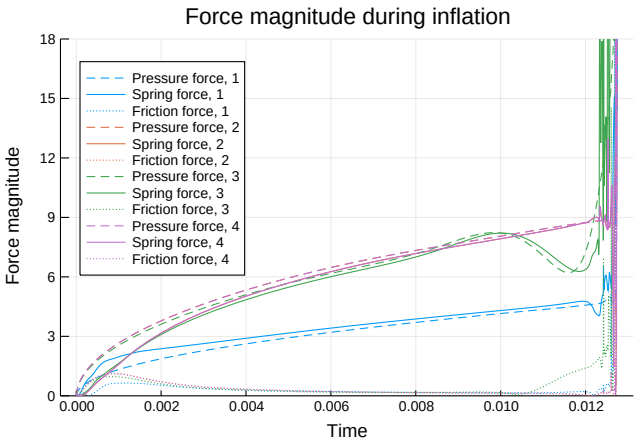
Figure 34 *The result of inflating an ellipse with the Wendland kernel. Parameters are:* $\Delta t = 1 \cdot 10^{-}5$, $\mu = 0.04$, *and* $h = 0.4$.



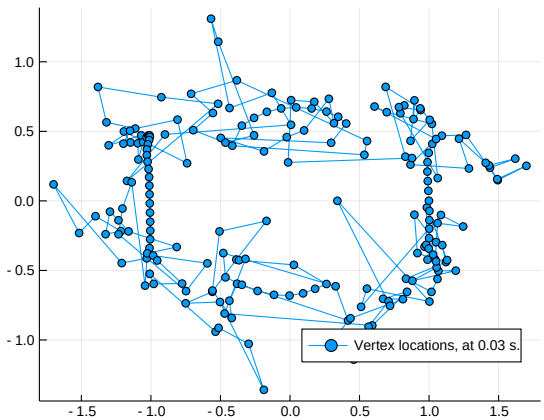
(a) *Pressure with respect to the time.*



(b) *Length of the vertices.*

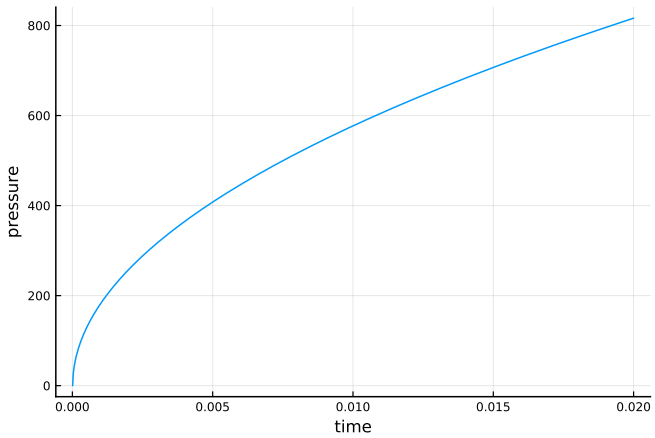


(c) *Force magnitude in four points.*

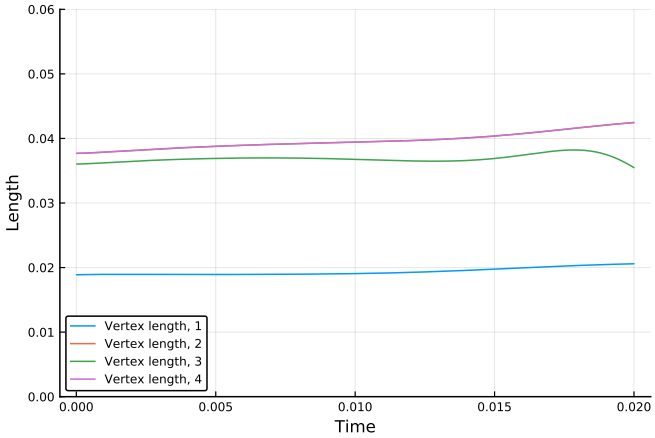


(d) *Final shape.*

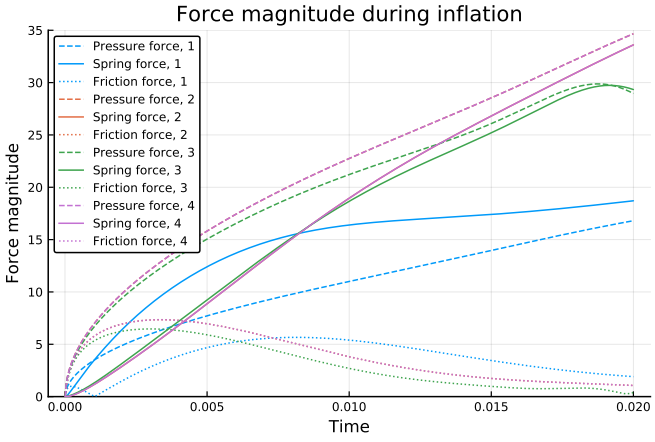
Figure 35 *The result of inflating an ellipse with the Wendland kernel and correction matrix from Equation (94). Parameters are:* $\Delta t = 1 \cdot 10^{-}5$, $\mu = 0.04$, *and* $h = 0.4$.



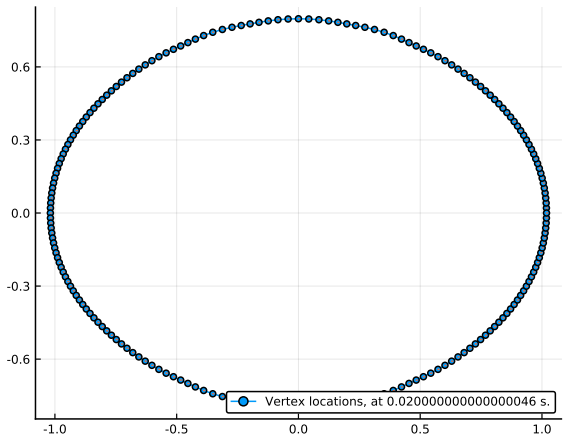
(a) *Pressure with respect to the time.*



(b) *Length of the vertices.*

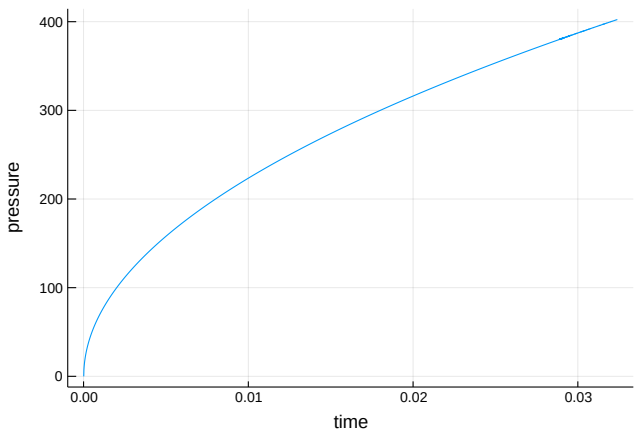


(c) *Force magnitude in four points.*

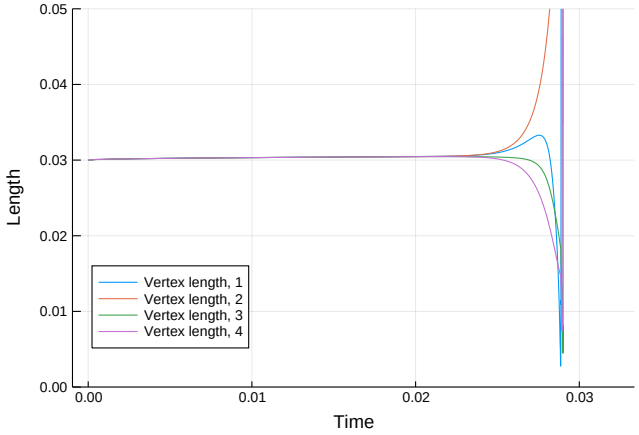


(d) *Final shape.*

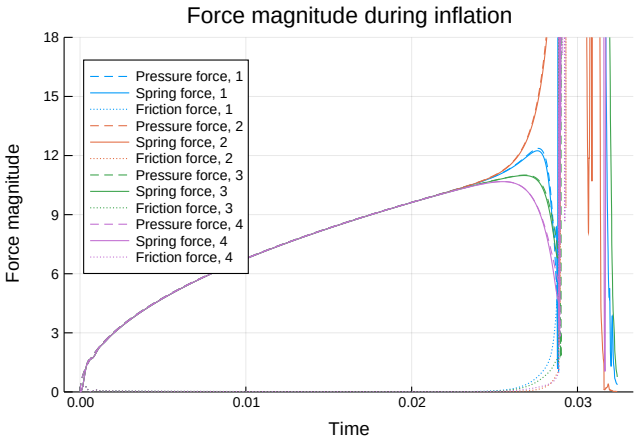
Figure 36 *The result of inflating an ellipse with the Wendland kernel and an adaptive smoothing length. Parameters are:* $\Delta t = 1 \cdot 10^{-}5$, $\mu = 0.04$.



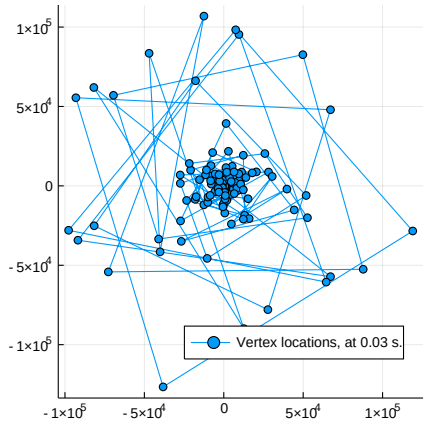
(a) *Pressure with respect to the time.*



(b) *Length of the vertices.*



(c) *Force magnitude in four points.*



(d) *Final shape.*

Figure 37 *The result of inflating a circle with the Wendland kernel and an adaptive smoothing length. Parameters are:* $\Delta t = 1 \cdot 10^{-}5$, $\mu = 0.4$.

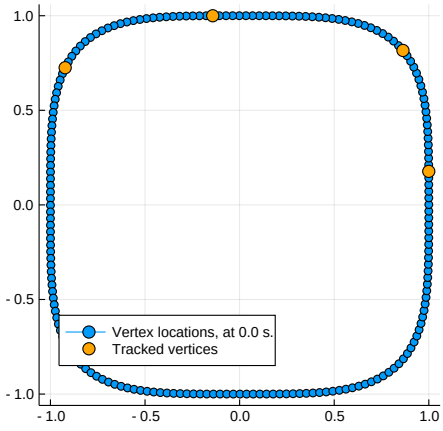
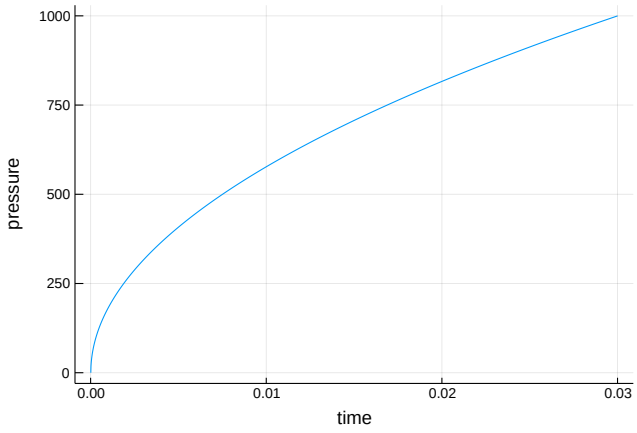


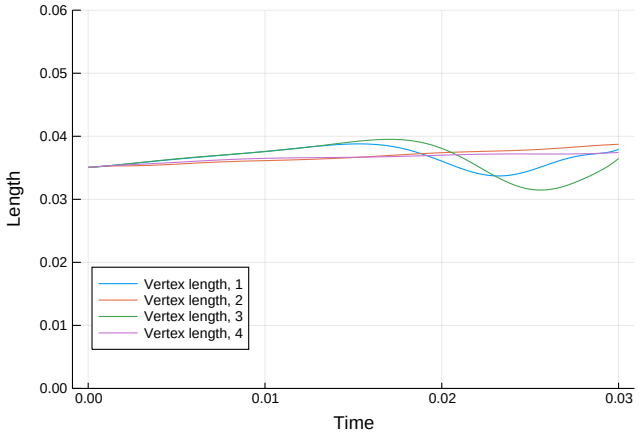
Figure 38 *The initial configuration of the squircle, with locations of the tracked vertices indicated. Vertex 1 is the rightmost point, the rest is numbered anti-clockwise. A parametrization for a squircle centered around the origin is:*

$$x(t) = |\cos(t)|^{0.5} r \mathit{sgn}(\cos(t)),$$
$$y(t) = |\sin(t)|^{0.5} r \mathit{sgn}(\sin(t)),$$

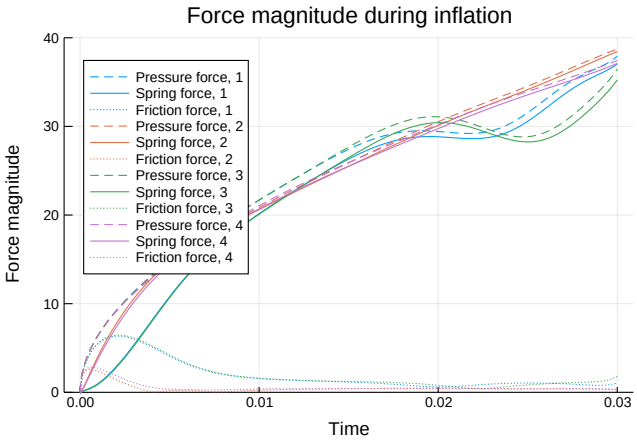
where r *is the radius of the minor radius of the squircle. In this case,* $r = 1$.



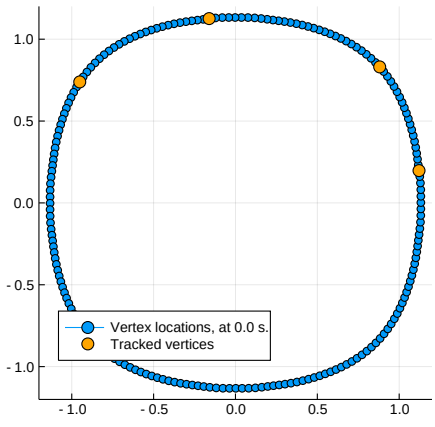
(a) *Pressure with respect to the time.*



(b) *Length of the vertices.*



(c) *Force magnitude in four points.*



(d) *Shape at* $t = 0.025s$.

Figure 39 *The result of inflating a squircle with the Wendland kernel. Parameters are:* $\Delta t = 1 \cdot 10^{-}5$, $\mu = 0.4$.

5.7.2 3D Results

For the three dimensional case, we show some examples with the Wendland kernel, without adaptive smoothing and no correction for the kernel gradient. The result of the inflation of a sphere can be seen in Figure 41. We see in Figure 41c that the spring forces increase quite slowly at the start, and then suddenly increase, quite unlike the two dimensional case. Comparing Figure 41 and 42 shows that it happens for both small and large pressure. In both cases, the simulation becomes unstable. It is difficult to say whether this is the tensile instability again, or whether the ARAP parametrization plays some role in this effect. So not only is it impossible to inflate something until static equilibrium, but we also do not have an approximately static case.

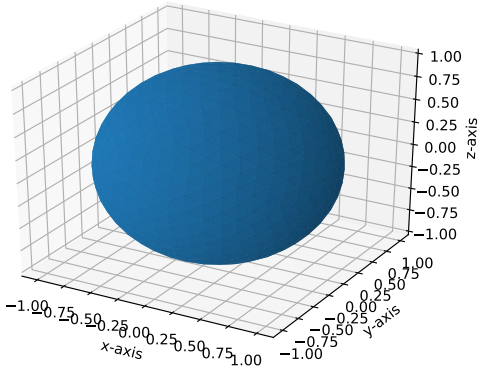
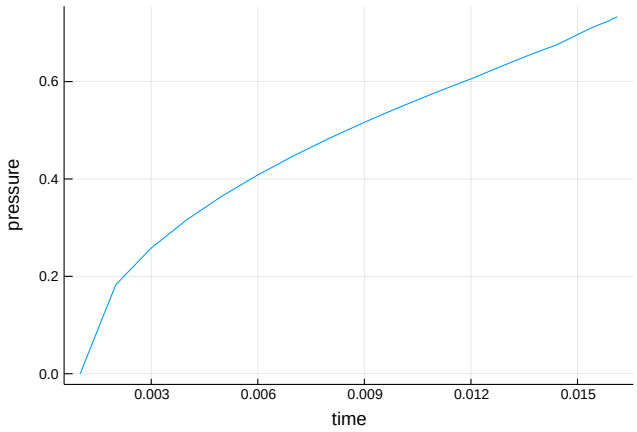
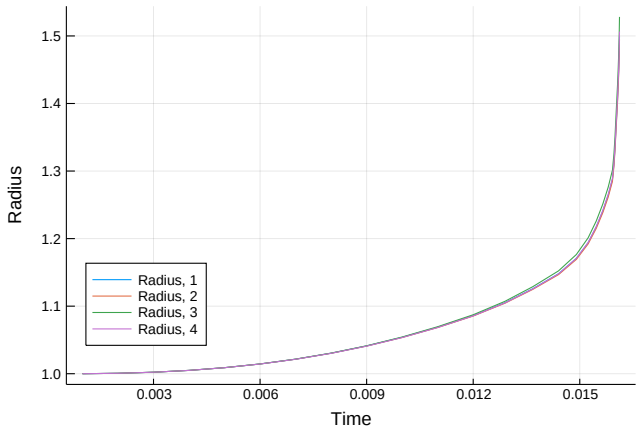


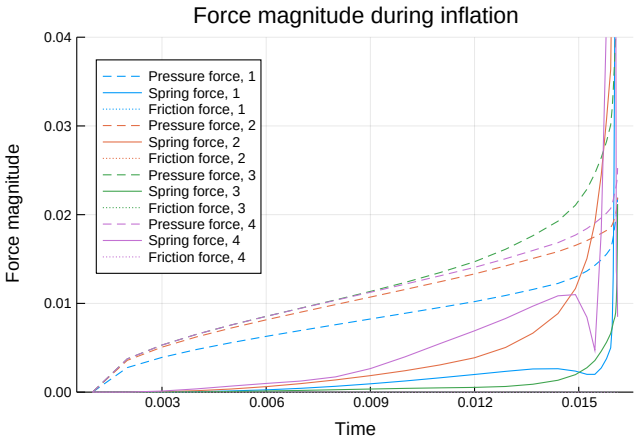
Figure 40 *The initial configuration of the sphere. It has radius 1 and 642 vertices.*



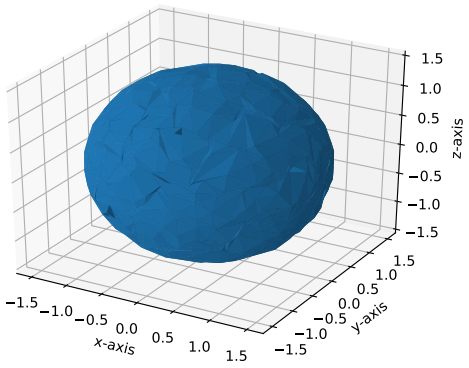
(a) *Pressure with respect to the time.*



(b) *Radius of the four vertices.*

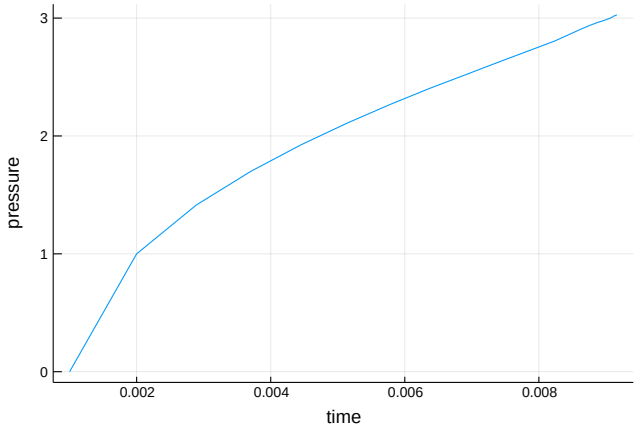


(c) *Force magnitude in four points.*

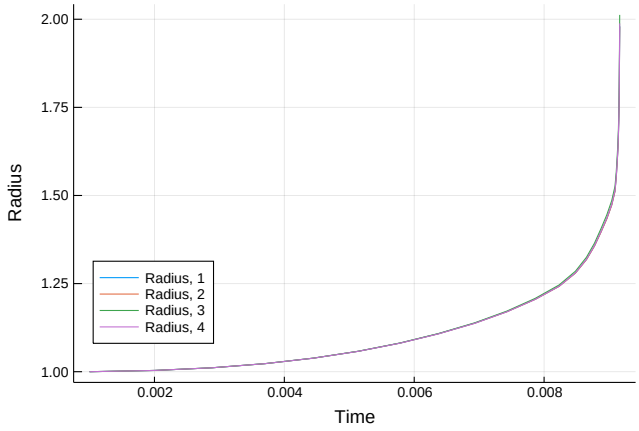


(d) *Final shape.*

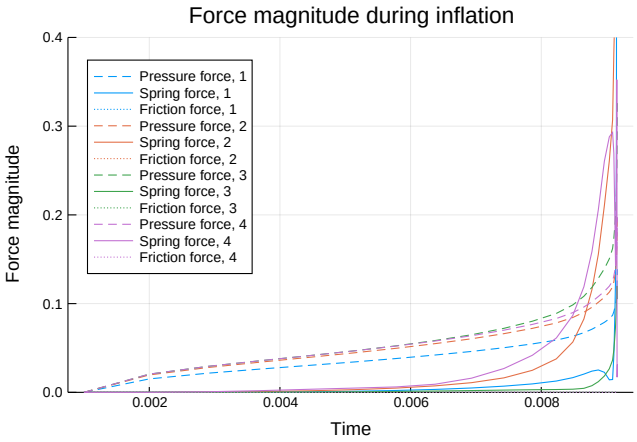
Figure 41 *The result of inflating a sphere with the Wendland kernel. Parameters are: $\Delta t = 1 \cdot 10^{-}5$, $\mu = 0.0$, and $h = 0.2$.*



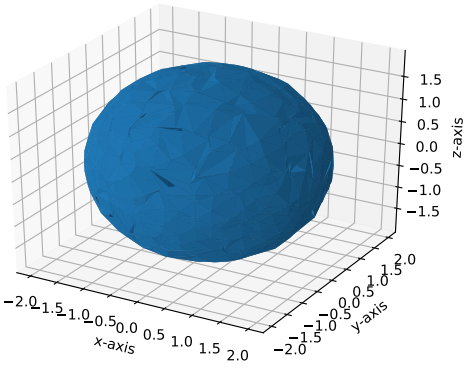
(a) *Pressure with respect to the time.*



(b) *Radius of the four vertices.*



(c) *Force magnitude in four points.*



(d) *Final shape.*

Figure 42 *The result of inflating a sphere with the Wendland kernel. Parameters are: $\Delta t = 1 \cdot 10^{-}5$, $\mu = 0.0$, and $h = 0.2$.*

6 Applying Neural Networks to Blow Forming

In Section 5 we saw that SPH uses neighboring particles to find smoothed approximations for e.g. the deformation gradient tensor. However, the elastic response still requires a material model. As we want to find a method that is data-driven, we investigate Neural Networks, which are a way to obtain an approximation to an unknown function using data. We will use NN to solve partial differential equations, namely the governing equation and the eikonal equation. We also investigate the ability of NN to extract information from a geometry, namely computing the geodesic distance. Before going into details, we will first give an explanation of how NN work, as well as a short overview of NN in engineering. We conclude Section 6 with an observation that connects NN and SPH.

6.1 Introduction to Neural Networks

6.1.1 Neurons, Layers, and Networks

Neural Networks are systems comprised of highly interconnected processing elements in a certain architecture. The term “neural” comes from the fact that neural networks were inspired by the biological structure of the brain [78]. In the brain, as soon as a neuron receives a high enough signal, it releases a signal. This signal is propagated to other neurons, which may also release a signal. The simplest mathematical implementation of a biological neuron is called a perceptron, McCulloch-Pitts neuron, or linear threshold gate [78, p.67]. It takes binary input values, assigns weights to them and returns a binary value depending on the weighted sum:

$$p(\mathbf{x}) = \begin{cases} 1 & \text{if } \sum_i w_i x_i > b, \\ 0 & \text{if } \sum_i w_i x_i \leq b. \end{cases}$$

The threshold value b is called the bias and says something about how difficult it is for the signal to propagate. Usually, it is written as follows:

$$p(\mathbf{x}) = \begin{cases} 1 & \text{if } \sum_i w_i x_i - b > 0, \\ 0 & \text{if } \sum_i w_i x_i - b \leq 0. \end{cases} \tag{129}$$

The weights, or parameters, w_i say something about how important their corresponding input is. In practice, not only perceptrons are used and in general the output is the result of a function of the sum of the weighted input:

$$o(\mathbf{x}) = \sigma(\sum_i w_i x_i - b), \tag{130}$$

where σ is called the activation function. In the case of a perceptron, σ was the binary step function, but there are more possibilities: see Figure 43 for some examples. For clarity, only one type of sigmoidal, or s-shaped, function is shown in the figure, but other options exist besides the logistic function. These include the hyperbolic tangent and the arctangent. Sigmoidal functions are essentially smooth versions of the step function. The choice of activation function can greatly influence the ability of an NN to learn. In particular, their derivative plays an important role during the backpropagation process. This will be explained in Section 6.1.2.

Of course, a single neuron, or node, could be used to base a decision on. However, a network can combine multiple neurons to produce a more sophisticated output. In Figure 44 a simple NN is shown. Every circle represents a neuron. The NN has several input neurons, two hidden layers and one output neuron and each layer is fully connected with the next layer. The neurons in the hidden layers do the same as in the input layer: each one returns a function of the weighted input values. As they are not immediately visible given the input or output of the network, they are called hidden. The number of hidden layers and their sizes may vary. The input to an NN is often a vector, but tensors of any size are possible, as long as everything fits in memory together with all the weights of the NN. The output of the NN is usually a scalar or a vector.

In Figure 44, each neuron is connected to all neurons of the next layer: they are fully connected or dense layers. However, other kinds of layers exist as well. We call a certain sequence of layers, their size and activation functions, combined with the way the layers are connected, the network architecture. The choice for a given architecture depends on the kind of input that the network gets and the desired output. The NN in Figure 44 is called a feedforward neural network (FNN), or more specifically, a multilayer perceptron (MLP) [78]. It can have arbitrarily many input and output nodes. Note that even though a MLP contains the name perceptron, that the neurons do not necessarily

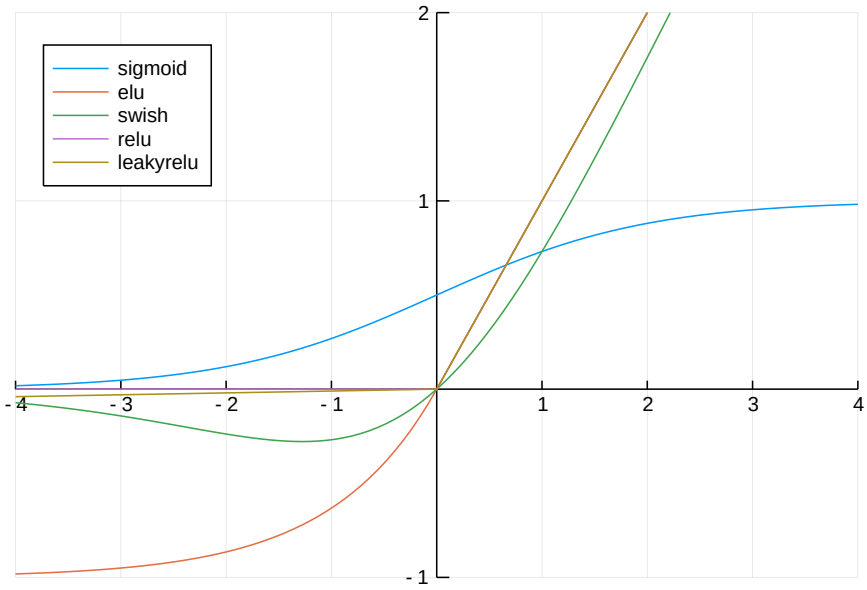


Figure 43 *A couple of often used activation functions. The functions are*

$$\textit{sigmoid: } \sigma(x) = \frac{1}{1+e^{-x}}, \quad \textit{swish [79]: } \sigma(x) = \frac{\alpha x}{1+e^{-\alpha x}}, \quad \textit{ReLU [80]: } \sigma(x) = max(0,x),$$

$$\textit{Leaky ReLU [81]: } \sigma(x) = \begin{cases} \alpha x, & \textit{if } x < 0, \\ x & \textit{if } x, \geq 0, \end{cases} \quad \textit{eLU [82]: } \sigma(x) = \begin{cases} (e^x - 1), & \textit{if } x < 0, \\ x, & \textit{if } x \geq 0, \end{cases}$$

where α is a trainable parameter or can be chosen in advance [79, 83]. Shown are leakyrelu with $\alpha = 0.01$, and swish, and elu with $\alpha = 1$.

need to be perceptrons; any activation function is possible. The name “feedforward” is based on the fact that no information from any part of the network is used as input in a previous layer. Neural networks which do are called recurrent neural networks (RNN) [78, pp. 337-354]. Two examples are long short-term memory (LSTM) [85], and gated recurrent units (GRU) [86]. More information about types of layers, architectures and activation functions can be found in [87]. As we only use convolutional layers besides dense layers, that is the only additional type of layer we will treat here. Neural networks using convolution in at least one of their layers are called Convolutional Neural Networks (CNN), introduced in the “Neocognitron” [88].

As mentioned before, dense layers can process a vector of input values. However, some inputs are not logically represented by a vector. Suppose we have a 28×28 image with RBG color values. When converting this to a vector, we get an input length of 2352 values. While an option, this is not necessarily the most logical one, as spatial information is lost. This is where convolutional layers come into play [87, pp. 326-335]. Convolutional layers work as a kind of filter on multidimensional data. They smooth out noise, much like the kernel function in Equation (62). For ease of reading we repeat it here:

$$\langle f(x) \rangle = \int_{\mathbb{R}^d} f(y)\phi(x-y)\, dy. \tag{62, revisited}$$

In terms of Neural Networks, $f(y)$ is referred to as the input and the function ϕ, the kernel, is called a filter. The output is often called a feature map. As before, we only have discrete input values, so we discretize Equation (62):

$$\langle f(x) \rangle = \sum_j f(y_j)\phi(x-y_j). \tag{131}$$

We can also use convolution for multiple dimensions:

$$\langle f(x_m, x_n) \rangle = \sum_i \sum_j f(y_i, y_j)\phi(x_m - y_i, x_n - y_j). \tag{132}$$

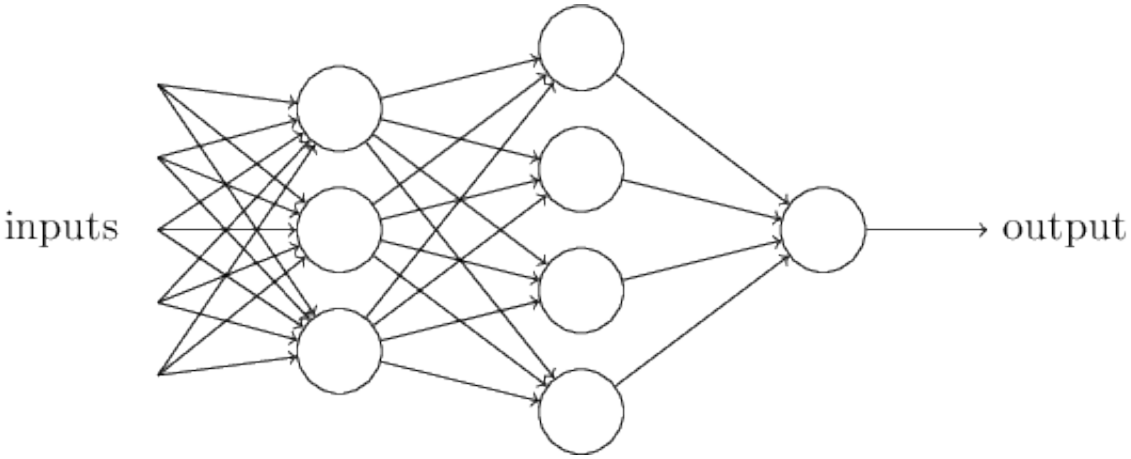


Figure 44 *A simple neural network with two hidden layers [84].*

Since convolution is commutative, we may also write:

$$\langle f(x_m, x_n)\rangle = \sum_i \sum_j f(x_m - y_i, x_n - y_j)\phi(y_i, y_j). \tag{133}$$

In neural network applications, the input and kernel are represented by tensors, where the elements of the filter are weights to be learned by the network. Let K and I be the tensor representing the filter and the discrete input, respectively. For a two dimensional example, we obtain that the output, F, of the convolution is:

$$F(m,n) = \sum_i \sum_j I(m-i, n-j)K(i,j), \tag{134}$$

where i, j, m and n denote indices. Often, the implementation of a convolution is not truly a convolution, but a cross-correlation:

$$F(m,n) = \sum_i \sum_j I(m+i, n+j)K(i,j). \tag{135}$$

The main difference is that cross-correlation is not commutative. However, as a convolutional layer is rarely used on its own, this does not pose a problem. We will refer to both as convolution.

As an example to make this more concrete, suppose our input, I, is an image of 5×5 pixels in grayscale, and that we have a 2×3 matrix K representing the kernel:

$$I = \begin{pmatrix} I_{0,0} & I_{0,1} & I_{0,2} & I_{0,3} & I_{0,4} \\ I_{1,0} & I_{1,1} & I_{1,2} & I_{1,3} & I_{1,4} \\ I_{2,0} & I_{2,1} & I_{2,2} & I_{2,3} & I_{2,4} \\ I_{3,0} & I_{3,1} & I_{3,2} & I_{3,3} & I_{3,4} \\ I_{4,0} & I_{4,1} & I_{4,2} & I_{4,3} & I_{4,4} \end{pmatrix} \tag{136a}$$

$$K = \begin{pmatrix} K_{0,0} & K_{0,1} & K_{0,2} \\ K_{1,0} & K_{1,1} & K_{1,2} \end{pmatrix}. \tag{136b}$$

In Equation (136a), two places with the locations of the elements with which the kernel is multiplied are highlighted. The two highlighted convolutions of I and K are:

$$F(1,0) = \sum_{i=0}^{1}\sum_{j=0}^{2} I_{i+1,j}K_{i,j} = I_{1,0}K_{0,0} + I_{1,1}K_{0,1} + I_{1,2}K_{0,2} + I_{2,0}K_{1,0} + I_{2,1}K_{1,1} + I_{2,2}K_{1,2},$$

$$F(2,2) = \sum_{i=0}^{1}\sum_{j=0}^{2} I_{i+2,j+2}K_{i,j} = I_{2,2}K_{0,0} + I_{2,3}K_{0,1} + I_{2,4}K_{0,2} + I_{3,2}K_{1,0} + I_{3,3}K_{1,1} + I_{3,4}K_{1,2}.$$

The kernel K can be thought of as a kind of window, sliding over the input and filtering information. In the example, we only have one filter, but usually each convolutional layer consists of multiple filters. In images, a convolutional layer can find oriented edges, curves or certain colors. These are the features of the images and the reason that the output of the convolution is often called a feature map. The advantage of using these filters, is that their parameters are re-used, as opposed to dense layers, whose parameters are used only once per input. This makes it easier to work with higher dimensional data.

A convolutional layer usually has three more parameters to set: padding, stride and dilation. Padding has to do with the fact that the output of a convolution has a smaller size than the input. If it is desired that the output has the same size as the initial input, it is possible to pad the input with a border of zeros. This is often done in image analysis. Stride is the step that the kernel makes between convolutions. A stride of n means that we move the kernel n columns or rows between convolutions. Note that care needs to be taken when choosing n. In our example, if we choose a stride of 3, the kernel does not "fit" into I anymore. Finally, dilation is an expansion of the kernel operator [89]. In stead of keeping the parameters together, there is room between them. Let l denote the dilation, then the new convolution becomes

$$F_l(m,n) = \sum_i \sum_j I(m+(l+1)i, n+(l+1)j)K(i,j). \tag{137}$$

So normal convolution has a dilation of zero. As an example, suppose we have a dilation of 1 and use I and K as before. We get

$$F(1,0) = \sum_{i=0}^{1}\sum_{j=0}^{2} I_{(l+1)i+1,(l+1)j}K_{i,j} = I_{1,0}K_{0,0} + I_{1,2}K_{0,1} + I_{1,4}K_{0,2} + I_{3,0}K_{1,0} + I_{3,2}K_{1,1} + I_{3,4}K_{1,2}. \tag{138}$$

Dilation and stride can both be used to reduce the number of parameters, whereas only dilation can be used to connect information on multiple scales.

6.1.2 Training: Loss Functions and Backpropagation

Given an NN, it should be clear that it is capable of approximating a function. However, with random initial weights the network will not approximate the function we look for at all. So we need a way to adjust the weights of the neurons in the network. In order to do so, we need some kind of measure of how "good" its output is compared to what we want. This is where the loss or cost function, backpropagation and, usually, a lot of data come in. We start with the former.

Let $N : \mathbb{R}^n \to \mathbb{R}^n$ denote a feed forward neural network. We assume we have two sets X and Y, both of size n. They contain, respectively, input values for the network and the corresponding true values of the function we are trying to approximate. An element $\mathbf{y} \in Y$ is called a target or answer vector. So for a corresponding pair $\mathbf{x}_i \in X$ and $\mathbf{y}_i \in Y$, we want $N(\mathbf{x_i}) = \hat{\mathbf{y}} \approx \mathbf{y}_i$. A loss or cost function is a function giving us an indication in what measure this is actually true. Some examples are

1. Mean Squared Error (MSE) or L_2 norm: $l(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{n}\sum_{i=1}^{n}(\hat{\mathbf{y}}_i - \mathbf{y}_i)^2$,
2. Mean Absolute Error (MAE) or L_1 norm: $l(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{n}\sum_{i=1}^{n}\|\hat{\mathbf{y}}_i - \mathbf{y}_i\|$,
3. Cross Entropy for binary classification: $l(\mathbf{y}, \hat{\mathbf{y}}) = -\frac{1}{n}\sum_{i=1}^{n}\mathbf{y}_i\log(\hat{\mathbf{y}}_i) + (1-\mathbf{y})\log(1-\hat{\mathbf{y}}_i)$, where $\mathbf{y}_i$ is the i^{th} element of binary vector $\mathbf{y}$ and $\hat{\mathbf{y}}$ is a vector with elements between zero and one.

The choice of the loss function depends on the application. For example, cross entropy is used for classification networks whose output is a probability value between 0 and 1. The choice also depends on the data. In some applications, it is possible that the input data contains a lot of outliers. Since the L_2 norm is more sensitive to outliers, these values will influence the NN more and the L_1 norm might be more appropriate. In the end, we want a small value for the loss function, as that means that we have $N(\mathbf{x_i}) = \hat{\mathbf{y}} \approx \mathbf{y}_i$.

We will minimize the loss function by using gradient descent with backpropagation to calculate the derivatives of the NN. Backpropagation was introduced by Rumelhart [90]. In the simplest form, each parameter is updated with gradient descent:

$$w' = w - \lambda\frac{\partial l(\mathbf{y}, \hat{\mathbf{y}})}{\partial w}, \tag{139}$$

where l is the loss function, w is the current value of the parameter, w' is the new value, and $\mathbf{y},\ \hat{\mathbf{y}}$ are the target vector and output of the NN respectively. λ is called the learning rate. Computing $N(\mathbf{x_i}) = \hat{\mathbf{y}}$ is called forward propagation, as it corresponds with the natural order of feeding input to the network and receiving the output. Updating the weights is called backpropagation, as we go from the output and the loss function's derivative back to all the weights. When we say we train the network, we mean that we are predicting, backpropagating derivatives and updating the networks' weights.

Although it is not immediately clear from the notation, $\frac{\partial l(\mathbf{y},\hat{\mathbf{y}})}{\partial w}$ depends on the choice of activation function. One particular problem that sometimes occurs is called the "vanishing gradient problem", where during training the gradient become so small that the weights are not updated anymore [91]. The activation functions that suffer from this property are the sigmoidal functions as well as deep networks. It is easily seen by looking at Figure 43 that this can happen for values of $|x| > 3$, since there the sigmoid function's gradient is small. Conversely, exploding gradients can also be a problem [91].

$\frac{\partial l}{\partial w}$ can be computed with symbolic, numeric or automatic differentiation. Additionally, they can be derived manually, but this is a time consuming and error-prone endeavor. Numeric differentiation can be highly inaccurate, while symbolic differentiation can lead to long, cryptic expressions and "expression swell" [92]. Automatic differentiation uses the chain rule and the fact that computer programs are essentially a sequence elementary operations and functions, e.g., addition, division, $\sin$, $\log$, etc.

There are three variants of gradient descent and they correspond to the amount of predicted values that we use: $\hat{\mathbf{y}}$. If we first compute $\hat{\mathbf{y}}$ for *all* available data and then update the weights, it is called batch gradient descent [93]. This can be very slow and intractable for datasets that do not fit into memory. Batch gradient descent always converges a local minimum. Stochastic gradient descent (SGD), or online gradient descent, is at the other end of the spectrum, where the weights are updated after just one prediction [93]. As the gradient is based on only one prediction, it produces a high variance in the loss function. As opposed to batch gradient descent, SGD can jump between local minima. Finally, mini-batch gradient descent is in between the two and updates the weights after a subset of the training samples [93]. While the word batch seems to refer to the entire dataset, it is more commonly used to describe the mini-batches for mini-batch gradient descent. Furthermore, besides the three variants of gradient descent, there are also some different flavors. These include, from generally worse to better performance, gradient descent with momentum [78, 94], Nesterov Accelerated Gradient [95], Adagrad [96], ADADELTA [97], and ADAM [98]. See also [87, Chapter 8.5] for more information on how Adagrad, RMSProp, and ADAM. Besides training the network in batches, the dataset is often fed into the network multiple times. The number of times this happens is called the number of epochs. One forward and backward propagation of a batch is called an iteration. So if we have 1000 samples, and a batch size of 200, it takes 5 iterations to complete one epoch.

The batch size, learning rate, size of the network layers, and the padding, stride, and dilation, of convolutional layers, etc. are all called hyperparameters. The term parameters is already used to describe the NN's weights, which cannot be chosen, whereas the hyperparameters can.

Finally, we would like to spend a few words on data. Assuming a dataset with labeled data is available, not all of it is used in training the network. It is split into a training and testing part. The former is used to train the network, while the latter is used to validate its performance. While a rule of thumb for a test/train split is to use 70-80% for training and 30-20% for validation, there are more options [99–101]. A common thread is that holding out too much for validation leaves too little data for training and the reverse, that using too much for training does not allow us to assess the trained model well. Finally, it is often advantageous to normalize the data to have zero mean and variance 1 [102].

6.1.3 Overview of Applications

Usually, NN are used to learn patterns in large amounts of data. What these patterns and data are can vary greatly. It can be in natural language processing to translate human languages [103], image recognition [104] or even learning the game Go [105].

Besides these more common applications, NN are also used in more traditional engineering fields. It has been used to represent the Finite Element Method (FEM), by placing neural units on the FEM elements and nodes, and

connecting them according to the system equation in discrete form [106]. Instead of updating the weights, they update the input to decrease the output error. In another paper, a neural network, Monte Carlo simulation and FEM are combined to predict elastic properties of fabrics [107]. First, they construct a parametric geometrical model of a textile and use FEM to compute the solution to a boundary value problem. This process is automated and used to generate samples using Monte Carlo simulation. These samples are then used as training data for an NN, which afterwards is able to predict elastic properties. Finally, a use of NN in solid mechanics is the modeling of constitutive behavior [108]. As the relationship between stress and strain is not linear, NN are suited to learn it. A difficulty in this application is obtaining enough data to train the network. Other examples include predicting maximum spring back for plate material [109], coupling proportional-integral-derivative control with a neural network for roll bending [110], and predicting deflections of concrete beams [111]. We summarize that in these kind of applications, the NN is often used as an addition to existing techniques and usually not the only tool used. Another area of research is to solve partial differential equations for simulations. Usually, the partial differential equation is used in the loss function, so the training is unsupervised [112–114].

The above examples show that NN are useful and are capable of approximating a wide range of functions. Besides these practical examples, it is important to note that there have also been some theoretical results explaining why NN work. The most important of which is the universal approximation theorem. It states that an arbitrary function defined on $[0, 1]$ can be arbitrarily well uniformly approximated by a multilayer feed-forward neural network with one hidden layer using neurons with arbitrary activation functions in the hidden layer and a linear neuron in the output layer [115]. It does not touch upon the learnability of the parameters of the NN though.

6.2 Solving Partial Differential Equations Using Neural Networks

Usually, an NN needs a large dataset in order to train it. However, the loss function does not necessarily need the data, it merely needs some expression to minimize. We will use the method explained in [114], which does exactly this.

We use the following definition of a differential equation:

$$G(\mathbf{x}, \Psi(\mathbf{x}), \nabla\Psi(\mathbf{x}), \nabla^2\Psi(\mathbf{x})) = 0, \quad \mathbf{x} \in D, \tag{140}$$

subject to boundary conditions (BCs), where $\mathbf{x} = (x_1, \ldots, x_n) \in \mathbb{R}^d$, $D \subset \mathbb{R}^d$, denotes the definition domain with boundary ∂D, and $\Psi(\mathbf{x})$ the function to be computed. The domain is discretized into a set of points $\hat{D}$ and we obtain the discretized equation:

$$G(\mathbf{x}_i, \Psi(\mathbf{x}_i), \nabla\Psi(\mathbf{x}_i), \nabla^2\Psi(\mathbf{x}_i)) = 0, \quad \mathbf{x}_i \in \hat{D}, \tag{141}$$

subject to the BCs. To formulate this in terms of neural networks, let $\Psi_t(\mathbf{x}, \mathbf{p})$ denote a trial solution with parameters $\mathbf{p}$. We obtain the following minimization problem

$$\min_{\mathbf{p}} \sum_{\mathbf{x}_i \in \hat{D}} G\left(\mathbf{x}_i, \Psi_t(\mathbf{x}_i, \mathbf{p}), \nabla\Psi_t(\mathbf{x}_i, \mathbf{p}), \nabla^2\Psi_t(\mathbf{x}_i, \mathbf{p})\right)^2, \tag{142}$$

which is subject to the constraints imposed by the BCs. The trial solution contains the neural network and is written as

$$\Psi_t(\mathbf{x}, \mathbf{p}) = A(\mathbf{x}) + F(\mathbf{x})N(\mathbf{x}, \mathbf{p}), \tag{143}$$

where $A(\mathbf{x})$ satisfies the boundary conditions, $F(\mathbf{x})$ is zero when $x \in \partial D$ and $N(\mathbf{x}, \mathbf{p})$ is a single-output feedforward neural network. This way, the neural network does not interfere with the BCs and the trial solution always satisfies them. It also means that $\Psi : \mathbb{R}^n \to \mathbb{R}$.

6.2.1 Solving the Eikonal Equation

In Section 5.6.4, it is explained that the solution to the eikonal equation is the geodesic distance function. Since the eikonal equation is a first order partial differential equation, we will formulate it such that we can find an approximation to its solution with a neural network. The eikonal equation is given by

$$|\nabla\phi(\mathbf{x})| = 1, \ \ x \in D, \tag{144a}$$

$$\text{subject to } \phi(\mathbf{x}) = 0, \ x \in \Gamma, \tag{144b}$$

where Γ is not necessarily the boundary of D, but the set containing the points to which the geodesic distance is needed. We write the trial solution as

$$\begin{aligned}\Psi_t(\mathbf{x}) &= A(\mathbf{x}) + F(\mathbf{x})N(\mathbf{x},\mathbf{p}) \\ &= F(\mathbf{x})N(\mathbf{x},\mathbf{p}),\end{aligned} \tag{145}$$

where $F(\mathbf{x})$ should be zero if $\mathbf{x} \in \partial D$ and $A(\mathbf{x}) = 0$, as $\Psi_t(\mathbf{x}) = 0$ for $x \in \partial D$. We rewrite the first order partial differential equation to

$$G\left(\mathbf{x}, \Psi_t(\mathbf{x},\mathbf{p}), \nabla\Psi_t(\mathbf{x},\mathbf{p}), \nabla^2\Psi_t(\mathbf{x},\mathbf{p})\right) = G\left(\nabla\Psi_t(\mathbf{x},\mathbf{p})\right) = |\nabla\Psi_t(\mathbf{x})| - 1, \tag{146}$$

and thus the minimization problem on the discretized domain is

$$\min_{\mathbf{p}} \sum_{\mathbf{x}_i \in \hat{D}} \left(|\nabla\Psi_t(\mathbf{x}_i)| - 1\right)^2. \tag{147}$$

In the minimization (147), the gradient of the trial solution is present. This means that we need the gradient of $F(\mathbf{x})N(\mathbf{x},\mathbf{p})$:

$$\begin{aligned}\nabla\Psi_t(\mathbf{x}_i) &= \nabla\left(F(\mathbf{x})N(\mathbf{x},\mathbf{p})\right), \\ &= \nabla F(\mathbf{x})N(\mathbf{x},\mathbf{p}) + F(\mathbf{x})\nabla N(\mathbf{x},\mathbf{p}).\end{aligned} \tag{148}$$

An expression for the k^{th} derivative of $N(\mathbf{x},\mathbf{p})$ with respect to component j of $\mathbf{x} \in \mathbb{R}^n$, in the case of a multilayer perceptron with n input units, one hidden layer with H sigmoid units and one output unit, is provided in [114]:

$$\frac{\partial^k N}{\partial x_j^k} = \sum_{i=1}^{H} v_i w_{ij}^k \sigma^{(k)}(z_i), \tag{149}$$

where $z_i = \sum_{j=1}^{n} w_{ij}x_j + b_i$. w_{ij} and b_i, respectively, are the weight from input node j to sigmoid unit i and the bias of that same sigmoid unit. v_i is the weight of sigmoid unit i to the output node. $\sigma^{(k)}$ is the k^{th} order derivative of the sigmoid function.

We test whether the above works for a simple example. Let $D = [0,10] \times [0,10]$, and $\Gamma = x_\gamma$. Since $D \subset \mathbb{R}^2$, we have two input nodes for the neural network. The trial solution needs a function F such that $F(x_\gamma) = 0$. We take $F(x) = \|x - x_\gamma\|$. So then the trial solution is

$$\Psi_t(\mathbf{x}) = \|x - x_\gamma\| N(\mathbf{x},\mathbf{p}). \tag{150}$$

We note that this should be a very easy function to approximate for the NN, as it should be $N(\mathbf{x},\mathbf{p}) = 1$ for all $\mathbf{x}$. Let w be the matrix containing the weights from node j to sigmoid unit i and let v be the vector containing the weights from the sigmoid units to the output node. Then

$$\nabla N(\mathbf{x},\mathbf{p}) = \sum_{i=1}^{H} \begin{pmatrix} v_i w_{i1} \\ v_i w_{i2} \end{pmatrix} \sigma_i^{(1)}(z_i),$$

where $z_i = w_{i1}x_1 + w_{i2}x_1 + b_i$ and the first derivative of the sigmoid function is

$$\sigma^{(1)}(x) = \sigma(x)(1 - \sigma(x)).$$

The gradient of F is

$$\nabla F(\mathbf{x}) = \frac{\mathbf{x} - \mathbf{x}_\gamma}{\|\mathbf{x} - \mathbf{x}_\gamma\|}.$$

So we obtain for the gradient of the trial solution:

$$\nabla\Psi_t(\mathbf{x}_i) = \frac{\mathbf{x} - \mathbf{x}_\gamma}{\|\mathbf{x} - \mathbf{x}_\gamma\|} N(\mathbf{x},\mathbf{p}) + \|\mathbf{x} - \mathbf{x}_\gamma\| \nabla N(\mathbf{x},\mathbf{p}).$$

We train and test the network with 4000 and 1000 random points in D respectively and take $x_\gamma = (5, 5)$. We use SGD with a learning rate of 0.01. We choose to use a simple NN with two input units, 10 neurons in the hidden layer and one output node as in [114]. In Figure 45 the average loss value per epoch can be seen. In Figure 46a, the result of the trial solution can be seen with respect to the actual distance, whereas in Figure 46b the prediction of the network can be seen. We have found that the trial solution returns either only negative or only positive numbers, so we take the absolute value. The reason lies in the fact that the eikonal equation can be minimal for either one. Since the test and train set both follow the line $x = y$ closely, and the spread is actually quite low, we conclude that an NN can indeed learn the solution to the eikonal equation.

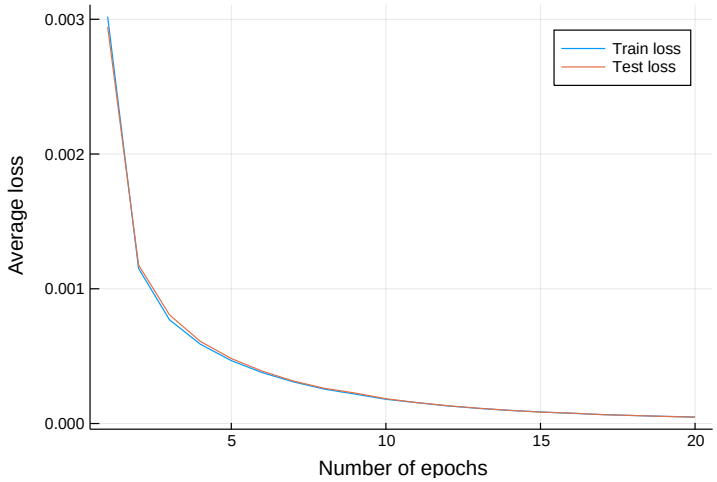


Figure 45 *Average loss value after each epoch during training.*

To be able to use this technique for a curve in $\mathbb{R}^2$, we would have to make an adjustment. Namely, we have to take the gradient of ϕ on the curve, so

$$|\nabla_S \phi(\mathbf{x})| = 1, \;\; x \in \mathcal{M},$$
$$\text{subject to } \phi(\mathbf{x}) = 0, \; x \in \Gamma.$$

However, we would get a network that is trained on one specific domain and would give accurate results only for that domain. To be able to effectively use an NN for distance computations, we need something that can handle an arbitrary curve. Hence, we do not continue this line of research.

6.2.2 Inflation

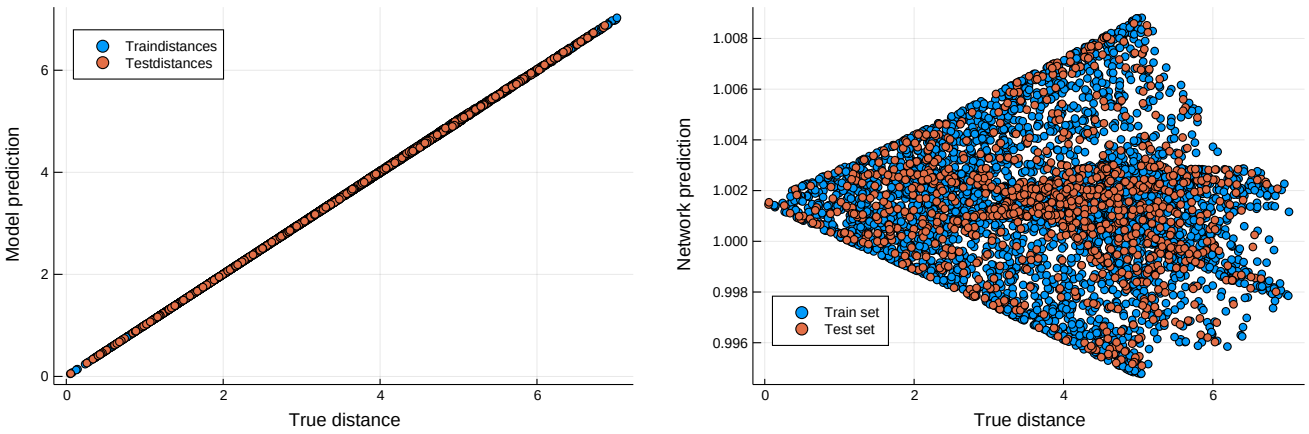
As a note, instead of using the neural network to find a distance function, we could also use it to immediately solve a partial differential equation describing our problem. So we would have to write Equation (1) in the form of Equation (140). To do so, we first need to define which function we are trying to approximate. As $\mathbf{P}$ ultimately depends on the displacement via the deformation gradient tensor, we define Ψ to be the displacement. So Ψ is a function of $\mathbf{X}$, such that $\mathbf{x} = \mathbf{X} + \Psi(\mathbf{X})$. Note that the deformation gradient tensor, is now equal to the Jacobian matrix of Ψ plus the identity:

$$\mathbf{F} = J_\Psi + \mathbf{I}. \tag{151}$$

As $\Psi(\mathbf{X})$ is a vector containing x and y values, we have to adjust the loss function a bit. We obtain the following minimization problem:

$$\min_{\mathbf{P}} \sum_{\mathbf{x}_i \in D} \| - \frac{1}{\rho_i} \nabla \cdot \mathbf{P}_i + \mathbf{F}_{p,i}/m_i \|^2. \tag{152}$$

As boundary condition, we fix one point to its initial location. Initial experimentation with a small neural network with one hidden layer with 10 neurons and small number of vertices showed that sometimes the NN did seem to learn the solution of the PDE. One of the tries that seemed to produce resonable results is shown in Figure 47.



(a) *Absolute value of the distance that the trial solution predicts as a function of the true distance.*

(b) *Prediction of the network. Note the small range of the y-axis.*

Figure 46 *Results of the trial solution and network prediction after training.*

However, training took quite a bit of time due to the computation of second derivatives. For aforementioned figure, training took over 30 minutes. Furthermore, the loss decreased very slowly for the problem even in two dimensional space. To say nothing of finding a learning rate and stresses going to infinity. Finally, we would not be able to reuse anything. We would have to train a network for each preform we want to inflate. For these reasons, this line of research was not investigated further either.

6.3 Neural Networks with Data: Curves

As the results in previous section did not show enough promise, we continue with a more traditional approach to NN: use a lot of data to train the network. We start with curves in two dimensional space and then continue with surfaces in three dimensional space.

6.3.1 Data

The input data is generated using Bézier curves. A Bézier curve is defined as follows [116]:

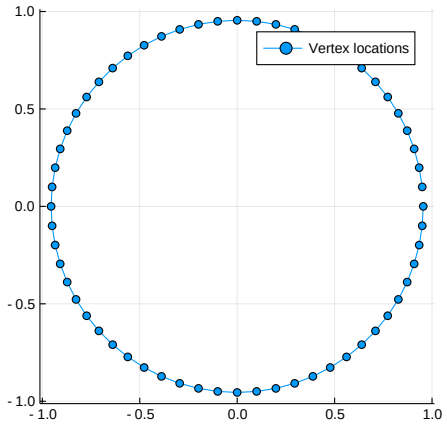
$$\mathbf{B}(t) = \sum_{i=0}^{n} \binom{n}{i} (1-t)^{n-i} t^i \mathbf{P}_i, \tag{153}$$

where $\mathbf{P}_0, \ldots, \mathbf{P}_n$ are control points and $0 \leq t \leq 1$. We use cubic Bézier curves, which are made using four random control points in $[0,1] \times [0,1]$. Note that generally not all control points lie on the curve. See Figure 48 for an example. We sample the curve such that we end up with approximately evenly spaced points. The number of points that we sample per curve is variable, so that the length between two consecutive vertices varies between curves. After making a set of curves, a fixed number, k, of consecutive points is taken from each curve. k will also be the number of input nodes in the neural network. We translate this subcurve such that its middle point is the origin. The length of a curve is computed by summing the Euclidean distances between consecutive points. The matrix containing the x and y coordinates as columns will be the input to the network.

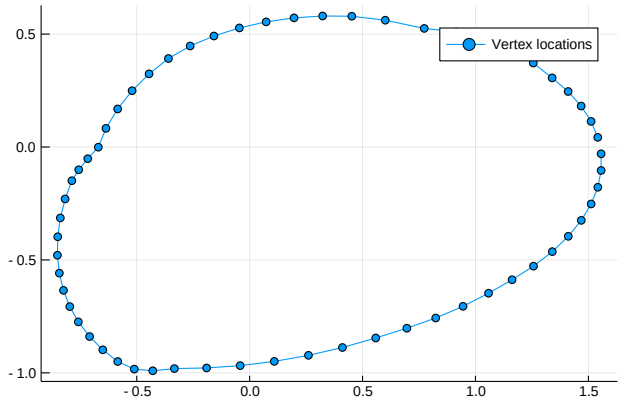
6.3.2 Network Architecture and Complexity

Let k be the number of points of which we have the x and y coordinates. Then the input, $\mathbf{x}$, is a $k \times 2$ matrix. Let n_{dense} and n_{conv} denote integer values. The network architecture is

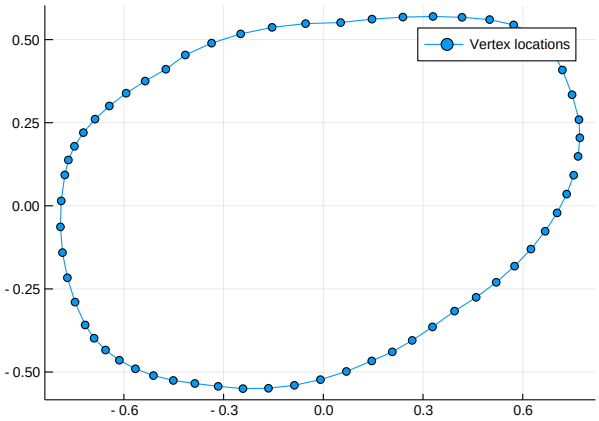
- a dense layer of size (k, n_{dense}),
- one convolutional layer with one filter of size $(n_{\text{conv}}, 2)$,
- one dense layer of size $(n_{\text{dense}} - n_{\text{conv}} - 1, 1)$.



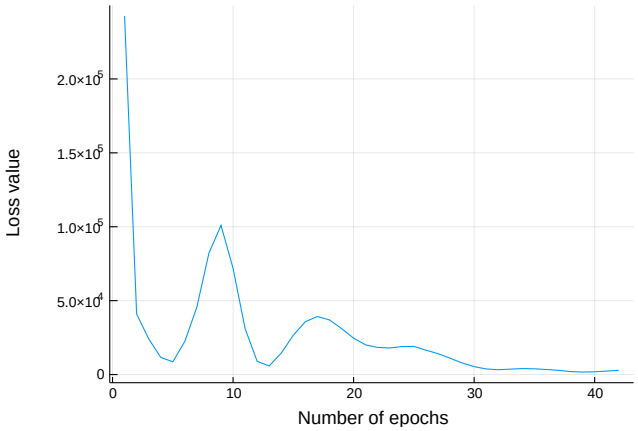
(a) *Initial configuration.*



(b) *Initial guess of the trial solution.*



(c) *Configuration after 42 epochs according to the trial solution.*



(d) *Loss value after each epoch during training.*

Figure 47 *Results of the NN with Equation (152) as loss function and with pressure = 1. The configuration has 60 points.*

The first dense layer and the convolutional layer use the relu activation function, while the final dense layer uses the identity function. We choose the relu, because it does not suffer from the vanishing gradient problem and generally gives good results. The reasoning for the architecture, is that the first dense layer makes sure that all information about the curve can be present in all the nodes used by the convolutional layer. The convolution is needed to go from a $n_{\text{dense}} \times 2$ input to a $(n_{\text{dense}} - n_{\text{conv}} - 1) \times 1$ output. The final dense layer is used to give us one number representing the distance.

Now we know what the network looks like, we can say something about the complexity of the network. A dense layer is comprised of a matrix multiplication, an addition and an activation function. Let $n \times m$ be the size of the input matrix and $m \times l$ be the size of the weight matrix of the dense layer. Then the matrix multiplication has complexity $\mathcal{O}(nml)$. Adding the biases and applying the activation function are both $\mathcal{O}(nl)$. So a dense layer has complexity $\mathcal{O}(nml + 2nl) = \mathcal{O}(nml)$.

For a convolutional layer, obtaining the complexity is a bit more complicated, as it depends on the size of the input tensor, whether we use padding, and which stride and dilation we choose. Also, it is comprised of several matrix multiplications and may have multiple filters. We assume that the input is a matrix here, as that is also the case we use. So let $n \times m$ be the size of the input matrix and $p \times l$ of the filters with $p < n$ and $l < m$. Let f denote the

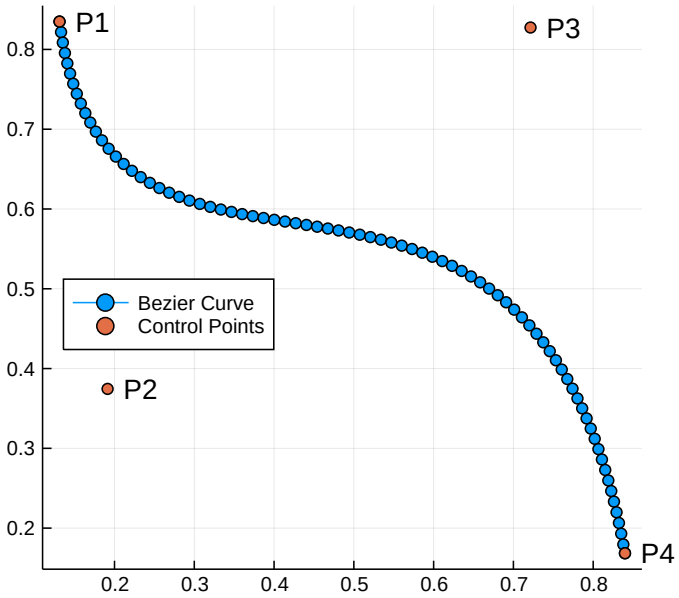


Figure 48 *Bezier curve with controlpoints.*

number of filters. For each filter, the output width and height are [117]

$$W = \frac{n - p - 2 * P_1}{S_1} + 1, \tag{154a}$$

$$H = \frac{m - l - 2 * P_2}{S_2} + 1, \tag{154b}$$

where P_i and S_i are the pad and stride parameters. In our case, $P_i = 0$ and $S_i = 1$ for all i, so we get $W = n - p + 1$ and $H = m - l + 1$. This means that we have to execute $(n - p + 1)(m - l + 1)$ convolutions. One convolution is $\mathcal{O}(lp)$ multiplications and $\mathcal{O}(lp - 1)$ additions, so in total $\mathcal{O}(lp)$. Finally, adding the biases and applying the activation function are both $\mathcal{O}((n-p+1)(m-l+1))$. Hence, we have that one convolutional layer has complexity $\mathcal{O}((n - p + 1)(m - l + 1)lpf)$.

Using the above, we can compute the complexity of our network architecture. The terms containing m in the above two paragraphs simplify, as we always have m = 2. The first dense layer is $\mathcal{O}(kn_{\text{dense}})$, the convolutional layer $\mathcal{O}((n_{\text{dense}} - n_{\text{conv}} + 1)(2 - 2 + 1)2n_{\text{conv}}) = \mathcal{O}(n_{\text{dense}}n_{\text{conv}} - n_{\text{conv}}^2)$ and the final dense layer $\mathcal{O}((n_{\text{dense}} - n_{\text{conv}} + 1)^2) = \mathcal{O}(n_{\text{dense}}^2 + n_{\text{conv}}^2)$. We also know that $n_{\text{conv}} < n_{\text{dense}}$, because a convolutional filter cannot be larger than the input, so $n_{\text{conv}} = \mathcal{O}(n_{\text{dense}})$. We conclude that the neural network has time complexity $\mathcal{O}(n_{\text{dense}}^2 + kn_{\text{dense}})$.

6.3.3 Results

Here we show the results of training an NN for distance computations on curves. We use the ADAM optimizer [98]. The loss function that we use is the MSE. For the curves, we have chosen $k = 21$. For particles with less than 19 other particles between them, it is possible to interpolate and add a few extra points.

We start with a simple plot of the loss with respect to the number of batches. In Figure 49, we see that the loss decreases with an increasing number of batches, and plateaus after around 60 batches (or 7.5 epochs). We also see that our test and train set are very similar. This is no surprise, as they are both made with random Beziér curves, as explained in Section 6.3.1. However, we do not know whether the hyperparameters that we have chosen there result in the lowest loss. Hence, we look at the loss value during training of the NN, with respect to n_{dense}, n_{conv}, and the learning rate. We note that we trained and restarted the NN as often as necessary to obtain something that made sensible predictions. One local minimum that the NN learned especially often, was the average length from the training set.

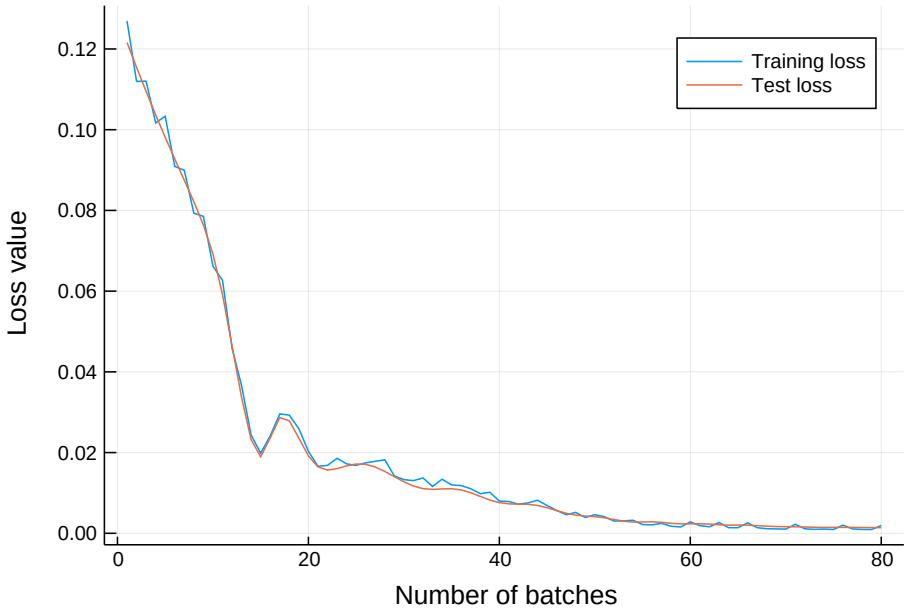
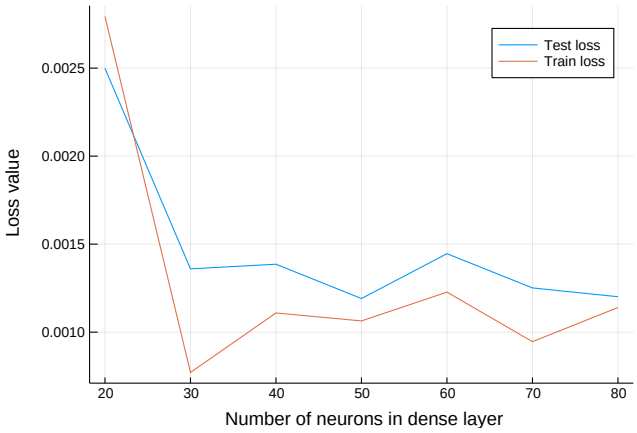


Figure 49 *Plot of the loss as a function of the number of batches, for 20 epochs with a batch size of 1000. Parameters are:* $\lambda = 0.01$, $n_{dense} = 50$, $n_{conv} = 5$,

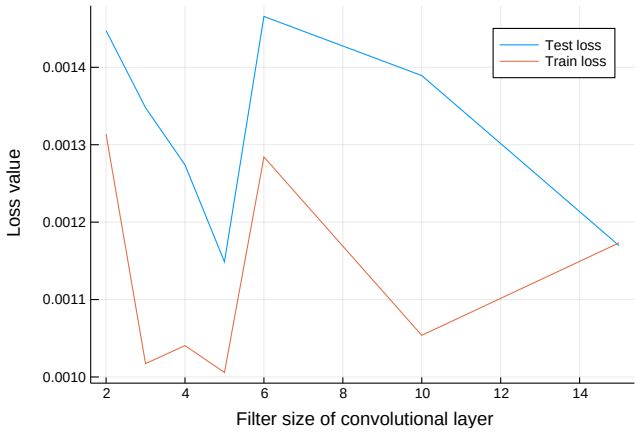
From Figure 50a, we conclude that 80 neurons seems to be a sensible choice for the number of neurons in the dense layer, because the loss is low for the test set. The values for the loss on the train set is also quite similar. However, in Section 6.3.2 we concluded that the number of neurons in the dense layer had a large effect on the time complexity. So we choose $n_{\text{dense}} = 50$ in the following, at the risk of overfitting slightly more than we would with 80 neurons. For the size of the filter in the convolutional layer we look at Figure 50b. We choose $n_{\text{conv}} = 5$, as larger results in a higher or equally high loss. Finally, we look at Figure 50d and set the learning rate at 0.01.

With the parameters described in the previous paragraph, we train an NN for 10 epochs and look at how well it performs. The network we train has a final loss value of 0.00113 and 0.00086 for the test and train set respectively and a total of 1158 parameters. We note that we actually trained networks with other values for the hyperparameters and obtained similar loss values after training for sufficiently long. A difference we observed, was that the network seemed to get stuck in local minima less often.

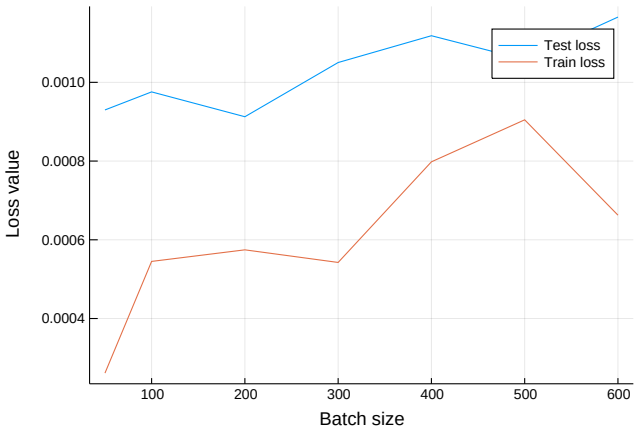
See Figure 52 for plots of the NN's predictions with respect to the true distance, as well as the Euclidean distance. While the Euclidean distance never overestimates, the NN has a less wide spread around the true distance. While simply summing over the Euclidean distances of edges between vertices is more efficient, this shows that NN can approximate the geodesic distance function.



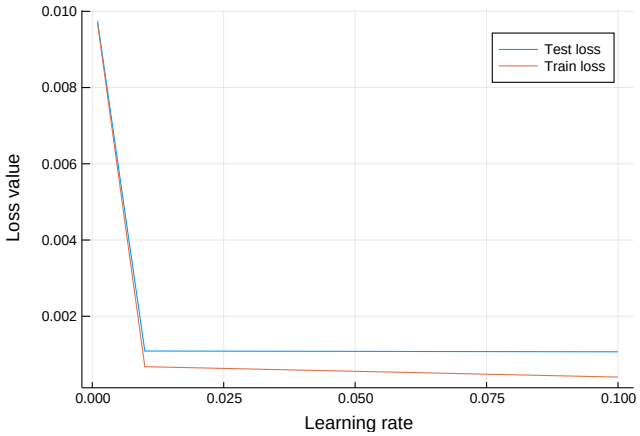
(a) *Plot of the loss as a function of the number of neurons in the dense layer after training for 20 epochs with a batch size of 1000. Parameters are:* $\lambda = 0.01$*, and* $n_{conv} = 5$*.*



(b) *Plot of the loss as a function of the filter size in the convolutional layer after training for 20 epochs with a batch size of 1000. Parameters are:* $\lambda = 0.01$*, and* $n_{dense} = 50$*.*



(c) *Plot of the loss as a function of the batch size after training for 20 epochs. Parameters are:* $\lambda = 0.01$*,* $n_{conv} = 5$ *and* $n_{dense} = 50$*.*



(d) *Plot of the loss as a function of the learning rate after training for 20 epochs with batch size 200. Parameters are:* $n_{conv} = 5$ *and* $n_{dense} = 50$*.*

Figure 50 *The loss of the NN during training, with respect to several hyperparameters.*

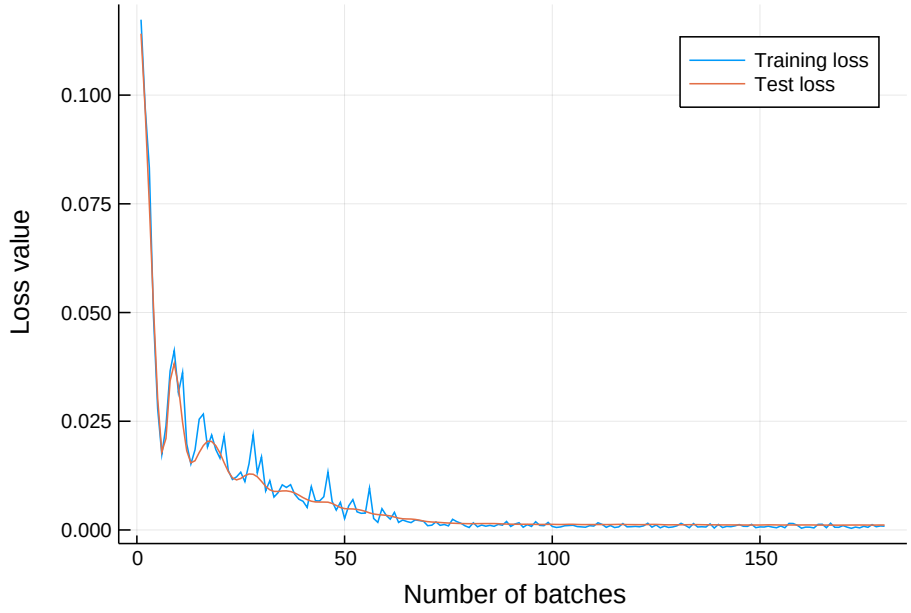
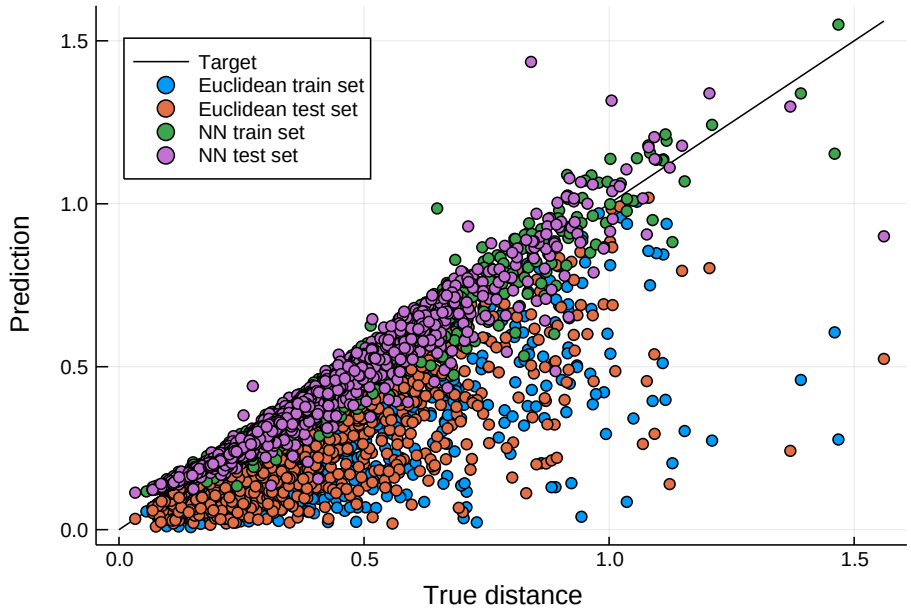
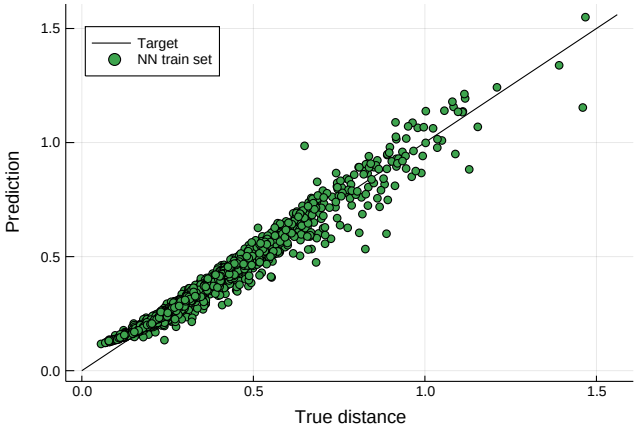


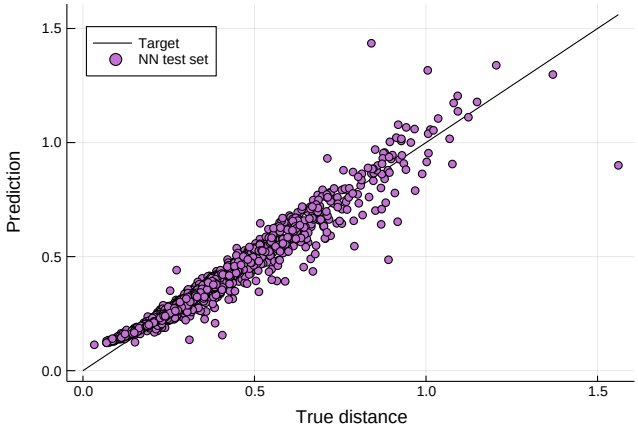
Figure 51 *Plot of the loss as a function of the number of batches, for 10 epochs with a batch size of 200. Parameters are:* $\lambda = 0.01$, $n_{dense} = 50$, $n_{conv} = 5$,



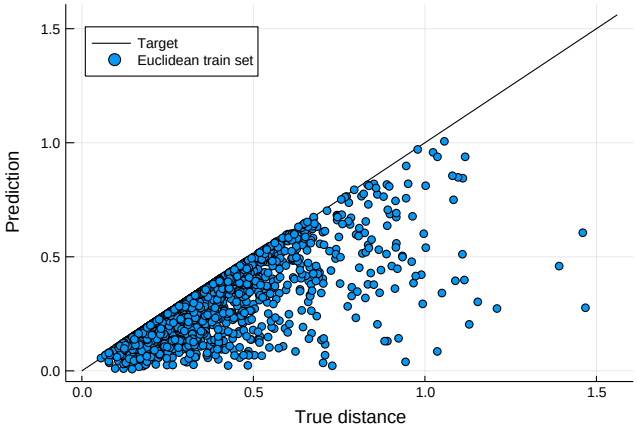
(a) *Euclidean distance and model prediction for test and train sets.*



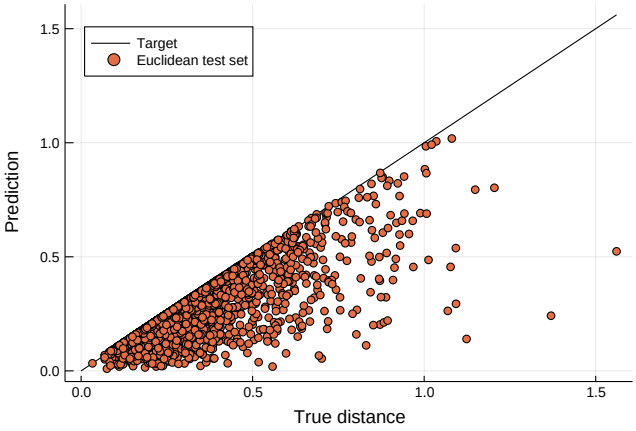
(b) *NN prediction on the train set.*



(c) *NN prediction on the test set.*



(d) *Euclidean distance on the train set.*



(e) *Euclidean distance on the test set.*

Figure 52 *Results of training the NN for distance computations on curves. We show the NN's predictions, as well as the results of the Euclidean distance on both the train and test set. The Euclidean distance is always an underestimate of the true distance, while the NN has a much smaller spread around the target. There is little difference between the test and train set.*

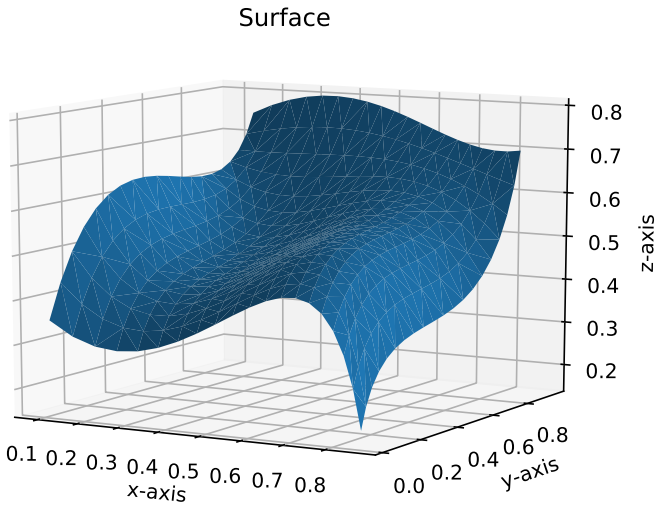


Figure 53 *Bezier surface.*

6.4 Neural Networks with Data: Surfaces

6.4.1 Data

Similar to the 2D case, we can use Beziér surfaces, which are defined as [116]:

$$\mathbf{B}(t,s) = \sum_{i=0}^{n}\sum_{j=0}^{m} B_i^n(t)B_j^m(s)\mathbf{P}_{i,j}, \tag{155}$$

where $B_i^n(t) = \binom{n}{i}(1-t)^{n-i}t^i$, $\mathbf{P}_{i,j} \in \mathbb{R}^3$ are control points and $0 \leq t,s \leq 1$. We use cubic Bézier surfaces, so we have sixteen control points. Again, the control points generally do not lie on the surface. We want to avoid the surface from self-intersecting, so we divide the unit square up into sixteen equally sized smaller squares. Each control point's x and y value is taken randomly in the small square, and their z value is between 0 and 1. The surface is sampled with t and s in a rectangular grid. See Figure 53 for an example of such a Beziér surface.

For each surface, multiple pairs of points are chosen between which the distance is calculated using the heat method from [74]. Let u, v be such a pair. After computing the distance between them, a subsurface is made with a fixed number of points, k. k also determines the size of the input of the neural network. This subsurface contains at least the shortest path over the edges between u and v, as found by Dijkstra's algorithm. We translate the subsurface such that u is placed in the origin. We make a matrix where the first three columns contain x, y and z coordinates and the rest of the columns contain the vertex adjacency matrix of the subsurface. Then we switch the rows such that u and v are always on the first and second row respectively. This way, the network gets connectivity information as well as locations in space.

6.4.2 Network Architecture and Complexity

Let k be the number of points in the subsurface, then the input to the neural network is a $k \times (k+3)$ matrix. The first three columns are the x, y and z coordinates, the rest of the matrix is the vertex adjacency matrix.

The network architecture is as follows:

- one dense layer of size (k, n_{dense1}),

- second dense layer of size $(n_{\text{dense1}}, n_{\text{dense2}})$,
- a convolutional layer with f_1 filters of size $(n_{\text{conv1}}, k+3)$,
- a second convolutional layer with f_1 filters of size $(n_{\text{conv2}}, 1)$,
- final dense layer $(n_{\text{dense2}} - n_{\text{conv1}} - n_{\text{conv2}} + 2, 1)$.

All layers, except for the last one use the relu activation function, while the last layer uses the identity function. The reasoning behind this choice is similar to the 2D one. We need convolutions to lower the number of dimensions of our output. The reason we add a dense and convolutional layer, is that we expect this to be a more difficult task. In image analyses it is very common to have many more convolutional layers, so if the size of the input is not too large, this NN will stay reasonably sized [104].

For each layer, we compute the complexity:

- $\mathcal{O}(k(k+3)n_{\text{dense1}})$,
- $\mathcal{O}(n_{\text{dense2}}n_{\text{dense1}}(k+3))$,
- $\mathcal{O}((n_{\text{dense2}} - n_{\text{conv1}} + 1)n_{\text{conv1}}(k+3)f_1)$,
- $\mathcal{O}((n_{\text{dense2}} - n_{\text{conv1}} - n_{\text{conv2}} + 2)n_{\text{conv2}}f_1)$,
- $\mathcal{O}((n_{\text{dense2}} - n_{\text{conv1}} - n_{\text{conv2}} + 2)^2)$.

We know that $n_{\text{conv1}} < n_{\text{dense2}}$, so $n_{\text{conv1}} = \mathcal{O}(n_{\text{dense2}})$ and $n_{\text{conv2}} < n_{\text{dense2}} - n_{\text{conv1}} + 1$, so $n_{\text{conv2}} = \mathcal{O}(n_{\text{dense2}} - n_{\text{conv1}})$. We conclude that $\mathcal{O}(n_{\text{dense2}} - n_{\text{conv1}} - n_{\text{conv2}} + 2) = \mathcal{O}(n_{\text{dense2}} + n_{\text{conv1}} + n_{\text{dense2}} + n_{\text{conv1}}) = \mathcal{O}(n_{\text{dense2}})$. Thus the complexity of the NN for surface geodesics is $\mathcal{O}(kf_1 n_{\text{dense2}}^2 + k^2 n_{\text{dense1}} + k n_{\text{dense1}} n_{\text{dense2}})$.

6.4.3 Results

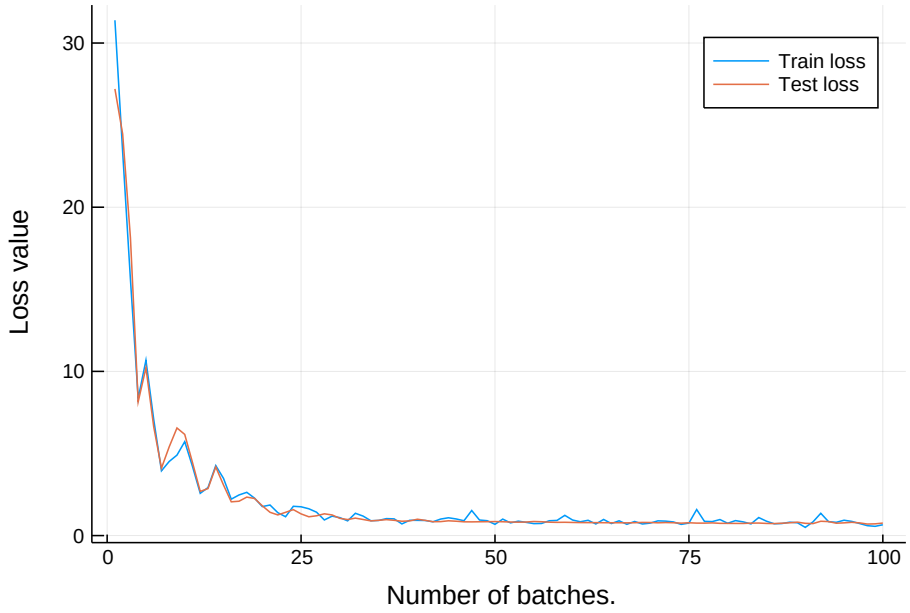
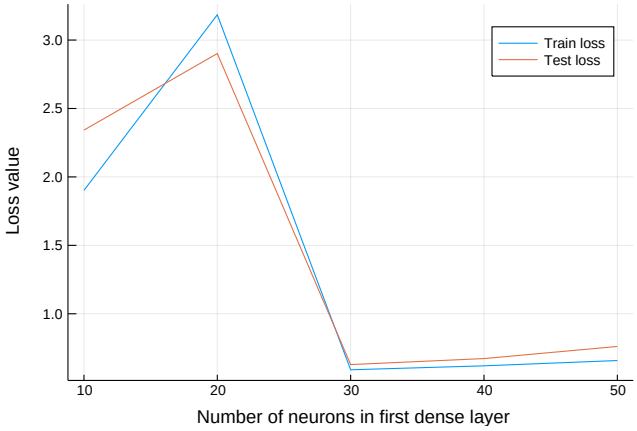
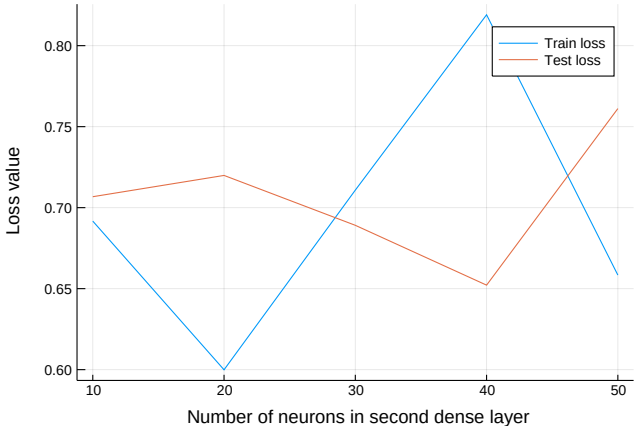


Figure 54 *Plot of the loss as a function of the number of batches, for 5 epochs with a batch size of 200. Parameters are:* $\lambda = 0.01$, $n_{dense1} = 50$, $n_{dense2} = 50$, $n_{conv1} = 10$, $n_{conv1} = 10$, *and* $f = 5$.

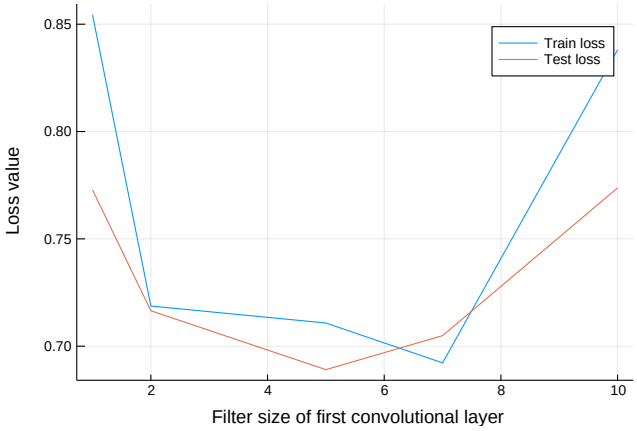
Here we show the results of training an NN for distance computations on surfaces. Again we use the ADAM optimizer and the MSE as loss function. For surfaces, choosing k is less simple than is the case for curves. On one hand, the input size depends on k^2, as well as the time complexity, so we want k to be small. On the other hand, we would like this NN to be usable for SPH and a too small number inhibits this use. As a trade-off between these two considerations, we chose $k = 50$.



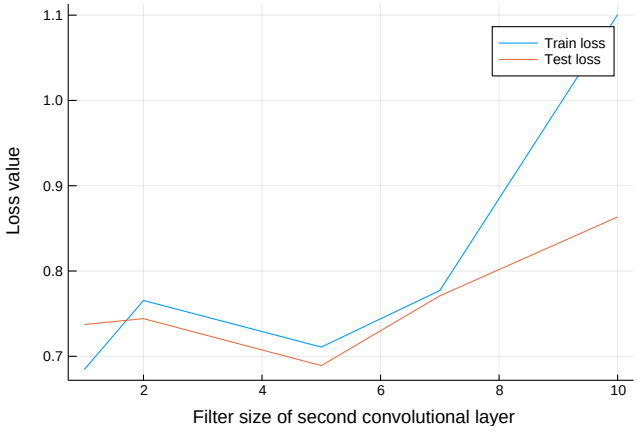
(a) *Plot of the loss as a function of the number of neurons in the first dense layer. Parameters are:* $\lambda = 0.01$, $n_{dense2} = 50$, $n_{conv1} = 5$, $n_{conv2} = 5$.



(b) *Plot of the loss as a function of the number of neurons in the second dense layer. Parameters are:* $\lambda = 0.01$, $n_{dense1} = 30$, $n_{conv1} = 5$, $n_{conv2} = 5$.



(c) *Plot of the loss as a function of the filter size in the first convolutional layer. Parameters are:* $\lambda = 0.01$, $n_{dense1} = 30$, $n_{dense2} = 30$, $n_{conv2} = 5$, *and* $f = 5$.

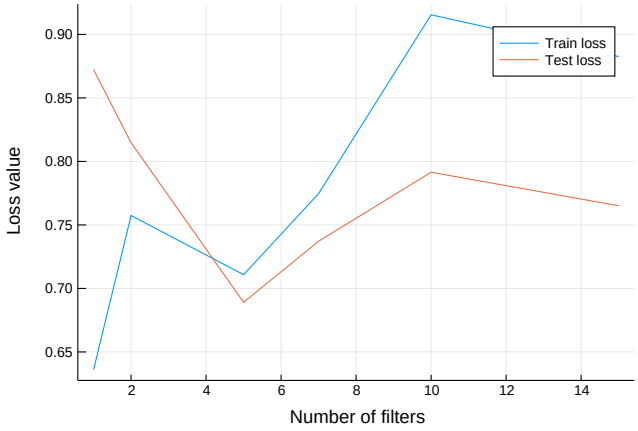


(d) *Plot of the loss as a function of the filter size in the second convolutional layer. Parameters are:* $\lambda = 0.01$, $n_{dense1} = 30$, $n_{dense2} = 30$, $n_{conv1} = 5$, *and* $f = 5$.

As in the 2D case, we start with a simple plot of the loss with respect to the number of batches. We use the hyperparameters that we found in the previous section: $n_{\text{dense1}} = n_{\text{dense2}} = 50$, $n_{\text{conv1}} = n_{\text{conv2}} = 5$, $\lambda = 0.01$, and a batch size of 200. The number of filters, f, we start with is 5. In Figure 54, we see that the loss decreases with an increasing number of batches, and plateaus before 50 batches (or 2.5 epochs). Again, the test and train set are very similar, for the same reason as before. To see whether different hyperparameters result in a lower loss, we look at the loss value during training of the NN. We will vary n_{dense1}, n_{dense2}, n_{conv1} and n_{conv2}. We note that we had to restart training a lot, as the NN got stuck in local minima more often than in the 2D case.

We start with the number of neurons in the first dense layer. In Figure 55a we see that the loss suddenly drops for 30 neurons. As we know from Section 6.4.2 that the time complexity depends heavily on the size of the first layer, we prefer to keep is as small as possible. So we choose $n_{\text{dense1}} = 30$. For the number of neurons in the second layer we look at Figure 55b and see that its size seems to not matter that much. Again, we want n_{dense2} to be small. We choose $n_{\text{dense1}} = 30$, because we do not seem to be overfitting there, while not having a too big value either. We continue with the convolutional layers. See Figure 55c for the filter size in the first convolutional layer. It is quite clear that $n_{\text{conv1}} = 5$ looks like the best choice. The loss for the filter size of the second convolutional layer can be seen in Figure 55d. Again, $n_{\text{conv2}} = 5$ has the lowest test loss value.

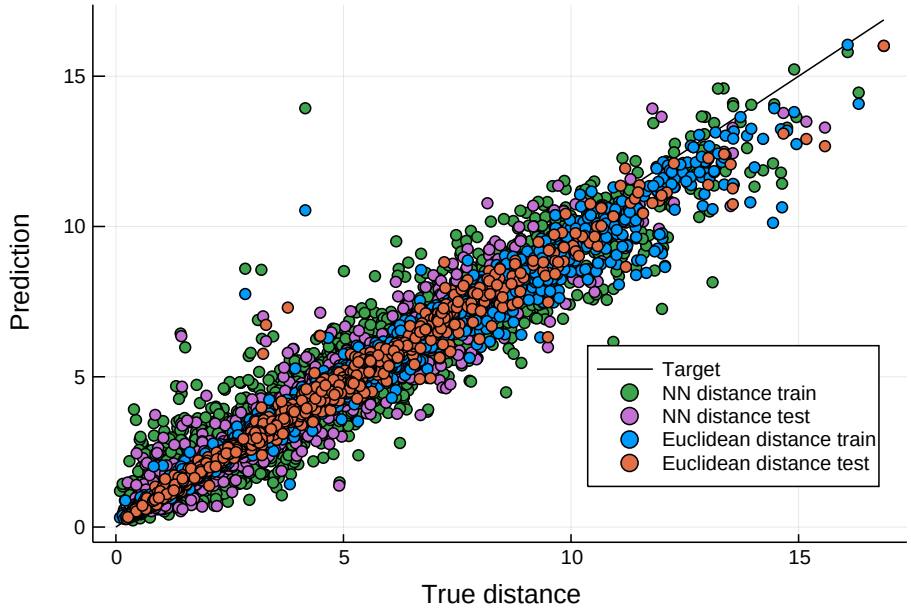
We train a network with the values of the hyperparameters as found in the previous paragraph. The result is an NN with a final train and test loss of 0.612 and 0.684, respectively, and 3839 parameters. See Figure 56 for plots of



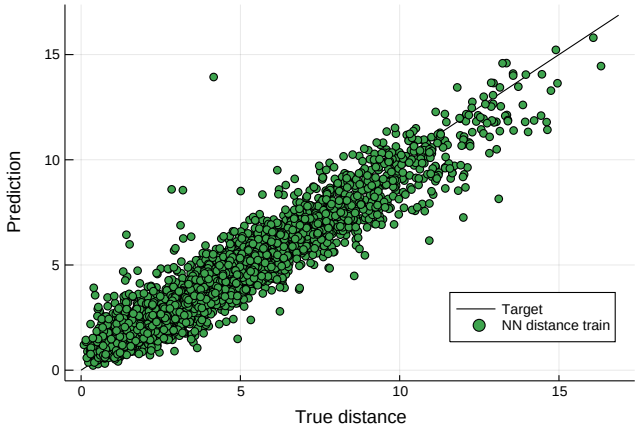
(e) *Plot of the loss as a function of number of filters in the convolutional layers. Parameters are:* $\lambda = 0.01$, $n_{dense1} = 30$, $n_{dense2} = 30$, $n_{conv1} = 5$ *and* $n_{conv2} = 5$.

Figure 55 *The loss of the NN after training for 5 epochs with a batch size of 200, with respect to several hyperparameters.*

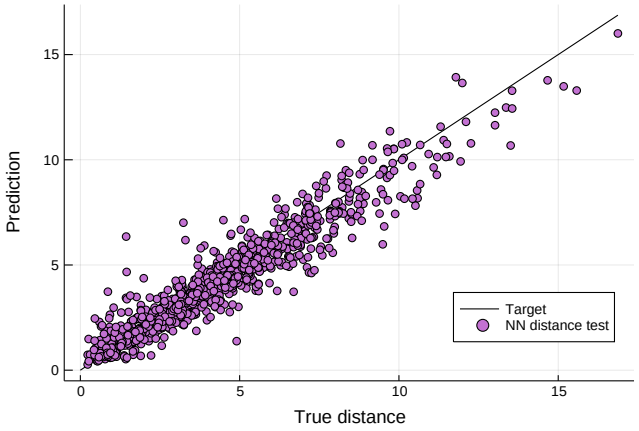
the NN's predictions with respect to the true distance, as well as the Euclidean distance. We see that the spread for the NN predictions is a lot larger than in the 2D case. Furthermore, we now also see some points for which the NN's prediction is lower than the Euclidean distance. Above all, an interesting observation is that there appear to be points for which the Euclidean distance is *higher* than the true distance, which is impossible. The explanation for this is that we computed the true distances with the heat method, which returns an approximation to the true distance. So our truth vector does not contain the actual geodesic distance. Additionally, the fact that the Euclidean distance is such a good approximation of the geodesic distance, seems to suggest that our data contains many pairs of points between which the geodesic is approximately a straight line.



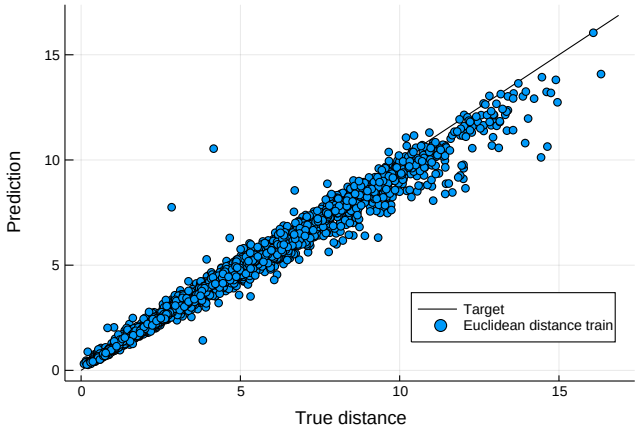
(a) *Euclidean distance and model prediction for test and train sets.*



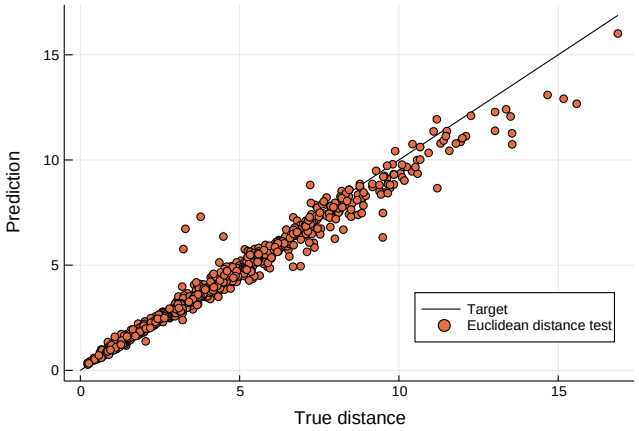
(b) *NN prediction on the train set.*



(c) *NN prediction on the test set.*



(d) *Euclidean distance on the train set.*



(e) *Euclidean distance on the test set.*

Figure 56 *Results of training the NN for distance computations on curves. We show the NN's predictions, as well as the results of the Euclidean distance on both the train and test set. The Euclidean distance is not always an underestimate of the true distance here, because the true distances are actually approximations generated by the heat method. The NN has a larger spread around the target than the Euclidean distance has. There is little difference between the test and train set.*

6.5 Neural Networks for Approximate Computing

We end this section with a couple of observations about SPH and NN. Recall that the output of an NN is a nonlinear function that depends on the input. A single node, with e.g. a ReLU activation function, in the first layer is not that complicated, while a neuron further down the NN is. Its input is already the result of a nonlinear function. So suppose we have an NN with several layers and with a single output neuron, which has a linear activation function and no bias. We can write the output as follows

$$N(\mathbf{x}) = \sum_j w_j \Psi(x_j). \tag{156}$$

Now, we look back at the smoothed approximations of the deformation gradient tensor and the acceleration:

$$\mathbf{F}_i = \sum_{j \in H_i} \frac{m_j}{\rho_j} (\mathbf{u}_j - \mathbf{u}_i) \otimes \nabla_i W_{ij} + \mathbf{I}, \tag{72, revisited}$$

$$\frac{d\mathbf{v}_i}{dt} = \sum_{j \in H_i} m_j \left(\frac{\mathbf{P}_i}{\rho_i^2} + \frac{\mathbf{P}_j}{\rho_j^2}\right) \nabla_i W_{ij} + (\mathbf{F}_p + \mathbf{F}_\mu)/m_i. \tag{73, revisited}$$

As the first Piola Kirchhoff stress tensor depends on the deformation in a nonlinear way, as discussed in Section 2.3.1, $\frac{d\mathbf{v}_i}{dt}$ is the result of a nonlinear function. Considering the above and the fact that it has previously been shown that NN are capable of predicting concrete deflection [111], it is not a stretch to suppose that we can employ an NN to determine the deformation of a preform. We want an NN that does so locally, so for a given node, a fixed number of additional neighbors are used as input. This allows the NN to take derived quantities like curvature into account, as these locally determine shape. The NN would depend on the current and initial (local) configuration. We would use the NN to predict the acceleration due to the stress and get the following formulation:

$$\frac{d\mathbf{v}_i}{dt} = \Psi({}^i\mathbf{x}, {}^i\mathbf{X}) + (\mathbf{F}_p + \mathbf{F}_\mu)/m_i, \tag{157}$$

where ${}^i\mathbf{x}$ and ${}^i\mathbf{X}$ are matrices containing the k nearest neighbors of $\mathbf{x}_i$ in the current and initial configuration respectively.

Training the network could happen based on real world data, possible augmented with data from simulations. The advantage of using experimental data is that this route avoids modeling difficult material behavior. Additionally, an NN might be much faster than conventional methods in producing results after training.

7 Comparison between SPH, FEM and Young-Laplace

In this section, our SPH implementation is applied to a sphere. We compare the results for small strains to the analytical solution of the Young-Laplace equation, and for both small and large strains to a FEM analysis.

7.1 Young-Laplace Equation

The Young-Laplace equation describes the pressure difference between two static fluids, due to surface tension or wall tension, the latter of which is only valid for thin walls. Under the assumption that the preform is a thin membrane and neglecting any bending stiffness, we can use the Young-Laplace equation to find an analytical solution to the inflation of a sphere, as it allows us to relate the pressure difference, stress and the shape to each other. The equation is

$$\Delta p = \gamma \left(\frac{1}{R_1} + \frac{1}{R_2} \right), \tag{158}$$

where Δp is the pressure difference, γ is the wall tension and R_i are the principal radii of curvature. Since we look at a sphere, $R_1 = R_2$ and we get

$$\Delta p = \frac{2\gamma}{R}. \tag{159}$$

Let R_0 denote the initial radius of the sphere. We can obtain the wall tension by computing the stress in the sphere and using the thickness, δ. Let θ be the central angle between two points on the sphere with radius R_0. The geodesic distance between the two points is $l_0 = \theta R_0$. If the radius increases with ΔR, the new distance becomes $l_\Delta = \theta(R_0 + \Delta R)$. This means that the Cauchy strain is

$$\varepsilon = \frac{l_\Delta}{l_0} - 1 = \frac{\theta(R_0 + \Delta R)}{\theta R_0} - 1 = \frac{\Delta R}{R_0}, \tag{160}$$

which is independent of the original choice of the two points. Hence, the stress in a point on the sphere is independent of location. We compute the wall tension by multiplying the stress with the thickness [8]:

$$\gamma = \sigma(\varepsilon)\delta. \tag{161}$$

This means that we can compute ΔR for a given Δp and R_0:

$$\Delta p = \frac{2\gamma}{R} = \frac{2\delta\sigma(\frac{\Delta R}{R_0})}{R_0 + \Delta R}. \tag{162}$$

For small strains, we assume that the stress is linearly elastic, so $\sigma(\varepsilon) = k\varepsilon$, and that the thickness δ is constant:

$$\Delta p = \frac{2\delta k \frac{\Delta R}{R_0}}{R_0 + \Delta R}, \tag{163}$$

which we rewrite to

$$\Delta R = \frac{\Delta p R_0^2}{2\delta k - \Delta p R_0}. \tag{164}$$

Sadly, due to the tensile instability in the three dimensional SPH case, we can only compare two dimensional problems. We also get a slightly different Young-Laplace equation, because there is only one radius of curvature:

$$\Delta p = \frac{\gamma}{R_1}. \tag{165}$$

So the expression for the increase in radius becomes

$$\Delta R = \frac{\Delta p R_0^2}{\delta k - \Delta p R_0}. \tag{166}$$

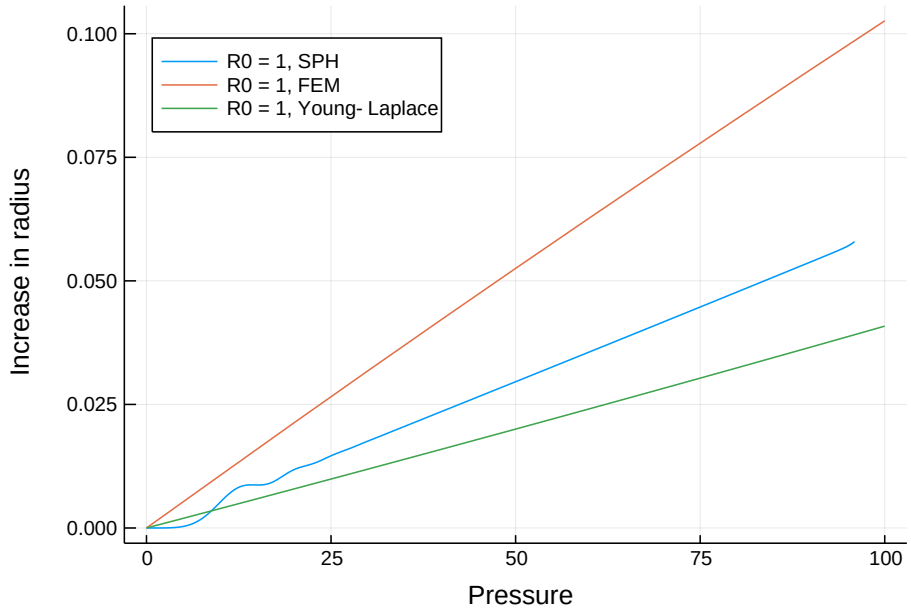


Figure 57 *Result of Equation (166) with the parameters of Table 1 for* $R_0 \in \{0.015, 0.02, 0.025\}$.

7.2 Comparison

To compare the two methods, we will use a silicone rubber balloon's material parameters. They are listed in Table 2, where the Lamé parameters are computed with Young's modulus and Poisson's ratio and rounded to three digits:

$$\lambda = \frac{E\nu}{(1+\nu)(1-2\nu)}, \tag{167}$$

$$\mu = \frac{E}{2(1+\nu)}. \tag{168}$$

For SPH and FEM we use the Saint Venant-Kirchhoff material law. To find a value for k, the spring constant needed for the Young-Laplace equation, we consider the following. The second Piola-Kirchhoff stress tensor is

$$\mathbf{S} = \lambda \mathrm{Tr}(\mathbf{E}) + 2\mu\mathbf{E} = \begin{pmatrix} \lambda(\mathbf{E}_{11} + \mathbf{E}_{22}) + 2\mu\mathbf{E}_{11} & 2\mu\mathbf{E}_{12} \\ 2\mu\mathbf{E}_{21} & \lambda(\mathbf{E}_{11} + \mathbf{E}_{22}) + 2\mu\mathbf{E}_{22} \end{pmatrix} \tag{169}$$

However, this is for a surface in two dimensions. We are actually dealing with a curve, a one dimensional object. Hence, $\mathbf{S}_{12} = \mathbf{S}_{21} = \mathbf{S}_{22} = 0$ and we take $k = \lambda + 2\mu$.

In Figure 57 we can see the increase of the radius as a function of the pressure difference as computed with the Young-Laplace equation, FEM and SPH, for a circle with initial radius 1. We see that the radius increases linear with the pressure for all three methods. For SPH, the radius at first seems to oscillate, but this is the result of the pressure and elastic forces not being balanced there yet. The three methods do not actually agree on the result of inflation. The reason may be that Equation (166) does not take into account the changing thickness of the material during inflation, resulting in a lower radius increase. Similarly, while the density of SPH is computed each time step, the values for the Lamé parameters are fixed during computation. Results more similar to FEM might be obtained by doing so.

8 Conclusion and Discussion

In this thesis, we investigated the numerical approximation of inflating hyperelastic 3D printed preforms. We show a two dimensional example computed with FEM. While neural networks are sometimes used to fit parameters for material models with experimental data, FEM does not allow us to easily use this data in another way to improve the material model. Next, we find that MSS are a quick and easy way to simulate the inflation of a membrane, and give an intuition about the effects of different types of springs on the inflation. However, they require too much manual intervention and are based on heuristics. Additionally, they use a small amount of neighboring vertices, the amount of which varies per vertex.

So we use SPH as a middle way, because it has a solid physical basis and is not dependent on a mesh. Since we model the preforms as manifolds of one dimension less than the spatial dimension, we adapt the SPH formulation. In particular, we find a way to express the gradient of the kernel, which lives on the manifold, with respect to the spatial variables. We also use the geodesic distance instead of the Euclidean to compute the distance between particles, in order to not overestimate the importance of particles.

We obtain results for two and three dimensional configurations. In two dimensions we see that the cubic B-spline kernel and using an adaptive smoothing length suffer more from what seems to be the tensile instability than the Wendland kernel and a fixed smoothing length. We also find that correcting the gradient does not affect the results. In the three dimensional case, inflating the preform becomes even more difficult, even for a simple sphere. A comparison between the Young-Laplace equation, FEM and SPH shows that they are all linear for small strains, but do not agree on the increase in radius of a circle with respect to the pressure.

Since we eventually want to remove the need to use material models, we investigate the use of Neural Networks. We have shown that we can solve partial differential equations. Solving the eikonal or governing equation using NN, however, reuses little information and training is too slow to effectively use this. The NN for distance computations which were trained with data worked well. In the case of curves, it approximated the true distance quite well. For surfaces we found that the network was able to learn the distance, but the Euclidean distance seemed to better approximate the true distance. There are two possible explanations. The first is the algorithm with which the truth vector is generated, and the second is that the geodesics between the points on the test and train surfaces are approximately straight lines. Still, we are convinced that, given more accurate data, a NN is capable of approximating the geodesic distance function.

Finally, we compare the SPH formulation and a NN formulation, where the aim of the NN formulation is to avoid material models altogether. The NN would be trained on experimental data, and be used to predict the acceleration due to the deformation.

References

[1] P. Mishra. *Stress Strain Curve: Relationship, Diagram and Explanation*. [Online, accessed Februari 15, 2018]. 2016. URL: http://www.mechanicalbooster.com/2016/09/stress-strain-curve-relationship-diagram-explanation.html.

[2] Z. Dai et al. "Dual-Support Smoothed Particle Hydrodynamics for Elastic Mechanics". In: (Mar. 2017). DOI: arXiv:1703.07209v1. URL: physics.comp-ph.

[3] R. S. Rivlin. "Large Elastic Deformations of Isotropic Materials. IV. Further Developments of the General Theory". In: *Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences* 241.835 (1948), pp. 379–397. ISSN: 00804614. URL: http://www.jstor.org/stable/91391.

[4] "Chapter 3 Elastic Materials and their Constitutive Equations". In: *Mathematical Elasticity Volume I: Three-Dimensional Elasticity*. Ed. by Philippe G. Ciarlet. Vol. 20. Studies in Mathematics and Its Applications. Elsevier, 1988, pp. 89–136. DOI: https://doi.org/10.1016/S0168-2024(08)70060-2. URL: http://www.sciencedirect.com/science/article/pii/S0168202408700602.

[5] M. Mooney. "A Theory of Large Elastic Deformation". In: *Journal of Applied Physics* 11.9 (1940), pp. 582–592. DOI: 10.1063/1.1712836. URL: https://doi.org/10.1063/1.1712836.

[6] R. S. Rivlin. "Large Elastic Deformations of Isotropic Materials. I. Fundamental Concepts". In: *Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences* 240.822 (1948), pp. 459–490. ISSN: 00804614. URL: http://www.jstor.org/stable/91430.

[7] L. J. Hart-Smith. "Elasticity parameters for finite deformations of rubber-like materials". In: *Zeitschrift für angewandte Mathematik und Physik ZAMP* 17.5 (Sept. 1966), pp. 608–626. ISSN: 1420-9039. DOI: 10.1007/BF01597242. URL: https://doi.org/10.1007/BF01597242.

[8] R. W. Ogden. "Large Deformation Isotropic Elasticity - On the Correlation of Theory and Experiment for Incompressible Rubberlike Solids". In: *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences* 326.1567 (1972), pp. 565–584. ISSN: 00804630. URL: http://www.jstor.org/stable/77930.

[9] Chengbo Zhou et al. "Deformation and structure evolution of glassy poly(lactid acid) below the glass transition temperature". In: *CrystEngComm* 17.30 (2015), pp. 5651–5663. DOI: 10.1039/C5CE00669D.

[10] S. Varatharajan et al. "Design and Analysis of single disc machine top and bottom cover". In: *International Journal of Scientific and Engineering Research* 2.8 (2011).

[11] G. Chandrasekaran and R.B. Dupaix. "Mechanical Behavior of a series of copolyester blends near glass transition: monotonic and load-hold behavior in compression". In: *International Journal of Polymer Science* 2012 (2012). DOI: 10.1155/2012/245205.

[12] Cambridge University Engineering Department. *Materials Data Book*. 2018. URL: http://www-mdp.eng.cam.ac.uk/web/library/enginfo/cueddatabooks/materials.pdf (visited on 09/04/2018).

[13] K. Nitta and M. Yamana. "Poisson's Ratio and Mechanical Nonlinearity Under Tensile Deformation in Crystalline Polymers". In: (Mar. 2012).

[14] Ultimaker. *Ultimaker Products*. 2018. URL: https://ultimaker.com/en/products/materials (visited on 08/28/2018).

[15] MakerPoint. *Maker Point Products*. 2018. URL: https://www.makerpoint.nl/nl/3d-printen/materials-supplies/filament/ (visited on 08/28/2018).

[16] Ultimaker. *Technical data sheet PLA*. 2017. URL: https://ultimaker.com/download/67934/TDS%20PLA%20v3.011.pdf (visited on 08/28/2018).

[17] MakerPoint. *Maker Point PLA material properties*. 2018. URL: https://www.makerpoint.nl/nl/makerpoint-pla-750gr-traffic-black.html (visited on 08/28/2018).

[18] Ultimaker. *Technical data sheet Tough PLA*. 2018. URL: https://ultimaker.com/download/74024/TDS%20Tough%20PLA%20v1.001.pdf (visited on 08/28/2018).

[19] Ben Wittbrodt and Joshua M. Pearce. "The effects of PLA color on material properties of 3-D printed components". In: *Additive Manufacturing* 8 (2015), pp. 110–116. ISSN: 2214-8604. DOI: https://doi.org/10.1016/j.addma.2015.09.006. URL: http://www.sciencedirect.com/science/article/pii/S2214860415000494.

[20] A. M. Tothill et al. "Fabrication and Optimisation of a Fused Filament 3D-printed Microfluidic Platform". In: *Journal of Micromechanics and Microengineering* 27.3 (2017), p. 035018. URL: http://stacks.iop.org/0960-1317/27/i=3/a=035018.

[21] G.H. Yew et al. "Water absorption and enzymatic degradation of poly(lactic acid)/rice starch composites". In: *Polymer Degradation and Stability* 90.3 (2005), pp. 488–500. ISSN: 0141-3910. DOI: https://doi.org/10.

1016/j.polymdegradstab.2005.04.006. URL: http://www.sciencedirect.com/science/article/pii/S0141391005001618.

[22] R. Acioli-Moura and X.S. Sun. "Thermal degradation and physical aging of poly(lactic acid) and its blends with starch". In: *Polymer Engineering & Science* 48.4 (2008), pp. 829–836. ISSN: 0032-3888. DOI: 10.1002/pen.21019. URL: https://browzine.com/articles/51689050.

[23] 3D Matter. *What is the influence of infill %, layer height and infill pattern on my 3D prints?* 2015. URL: http://my3dmatter.com/influence-infill-layer-height-pattern/ (visited on 08/28/2018).

[24] Eftychios Sifakis and Jernej Barbic. "FEM Simulation of 3D Deformable Solids: A Practitioner's Guide to Theory, Discretization and Model Reduction". In: *ACM SIGGRAPH 2012 Courses*. SIGGRAPH '12. Los Angeles, California: ACM, 2012, 20:1–20:50. ISBN: 978-1-4503-1678-1. DOI: 10.1145/2343483.2343501. URL: http://doi.acm.org/10.1145/2343483.2343501.

[25] Andrew Nealen et al. "Physically Based Deformable Models in Computer Graphics". In: *Computer Graphics Forum* 25.4 (), pp. 809–836. DOI: 10.1111/j.1467-8659.2006.01000.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2006.01000.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2006.01000.x.

[26] Mélina Skouras et al. "Computational Design of Rubber Balloons". In: *Computer Graphics Forum* 31.2pt4 (), pp. 835–844. DOI: 10.1111/j.1467-8659.2012.03064.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2012.03064.x.

[27] T.M. van Opstal. "Numerical methods for inflatables with multiscale geometries". Eindhoven University of Technology, 2013, 181 p. ISBN: 978-94-61919-36-6.

[28] Gertjan van Zwieten et al. *Nutils*. Feb. 2016. DOI: 10.5281/zenodo.822381. URL: https://doi.org/10.5281/zenodo.822381.

[29] Demetri Terzopoulos, John Platt, and Kurt Fleischer. "Heating and melting deformable models". In: *The Journal of Visualization and Computer Animation* 2.2 (), pp. 68–73. DOI: 10.1002/vis.4340020208. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/vis.4340020208.

[30] Gavin Miller and Andrew Pearce. "Globular dynamics: A connected particle system for animating viscous fluids." In: *Computers & Graphics* 13.3 (1989), pp. 305–309.

[31] David Baraff and Andrew Witkin. "Large Steps in Cloth Simulation". In: *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '98. New York, NY, USA: ACM, 1998, pp. 43–54. ISBN: 0-89791-999-8. DOI: 10.1145/280814.280821. URL: http://doi.acm.org/10.1145/280814.280821.

[32] Robert Bridson, Ronald Fedkiw, and John Anderson. "Robust treatment of collisions, contact and friction for cloth animation". In: *ACM Transactions on Graphics (ToG)* 21.3 (2002), pp. 594–603.

[33] T. Liu et al. "Fast Simulation of Mass-spring Systems". In: *ACM Trans. Graph.* 32.6 (Nov. 2013), 214:1–214:7. ISSN: 0730-0301. DOI: 10.1145/2508363.2508406. URL: http://doi.acm.org.dianus.libr.tue.nl/10.1145/2508363.2508406.

[34] Sueo Kawabata and Masako Niwa. "Fabric performance in clothing and clothing manufacture". In: *Journal of the Textile Institute* 80.1 (1989), pp. 19–50.

[35] LB Lucy. "Numerical Approach to the Testing of Fusion Process". In: *The Astronomical Journal* 88 (1977), pp. 1013–1024.

[36] R. A. Gingold and J. J. Monaghan. "Smoothed particle hydrodynamics: theory and application to non-spherical stars". In: *Monthly Notices of the Royal Astronomical Society* 181.3 (1977), pp. 375–389. DOI: 10.1093/mnras/181.3.375. eprint: /oup/backfile/content_public/journal/mnras/181/3/10.1093/mnras/181.3.375/2/mnras181-0375.pdf. URL: http://dx.doi.org/10.1093/mnras/181.3.375.

[37] Gui-Rong Liu and Moubin B Liu. *Smoothed particle hydrodynamics: a meshfree particle method*. World Scientific, 2003.

[38] A Bargteil and M van de Panne. "Smoothed Particle Hydrodynamics on Triangle Meshes". In: *Eurographics/ACM SIGGRAPH Symposium on Computer Animation* (2011), pp. 1–9.

[39] L. D. Libersky and A. G. Petschek. "Smooth particle hydrodynamics with strength of materials". In: *Advances in the free-Lagrange method including contributions on adaptive gridding and the smooth particle hydrodynamics method*. Springer, 1991, pp. 248–257.

[40] Zhiyi Chen et al. "Numerical simulation of large deformation in shear panel dampers using smoothed particle hydrodynamics". In: *Engineering Structures* 48 (2013), pp. 245–254. ISSN: 0141-0296. DOI: https://doi.org/10.1016/j.engstruct.2012.09.008. URL: http://www.sciencedirect.com/science/article/pii/S0141029612004865.

[41] p.W. Cleary. “Elastoplastic deformation during projectile–wall collision”. In: *Applied Mathematical Modelling* 34.2 (2010), pp. 266–283. ISSN: 0307-904X. DOI: https://doi.org/10.1016/j.apm.2009.04.004. URL: http://www.sciencedirect.com/science/article/pii/S0307904X09001164.

[42] C. Giacomuzzo et al. “SPH evaluation of out-of-plane peak force transmitted during a hypervelocity impact”. In: *International Journal of Impact Engineering* 35.12 (2008). Hypervelocity Impact Proceedings of the 2007 Symposium, pp. 1534–1540. ISSN: 0734-743X. DOI: https://doi.org/10.1016/j.ijimpeng.2008.07.070. URL: http://www.sciencedirect.com/science/article/pii/S0734743X08001619.

[43] M. B. Liu, G. R. Liu, and K. Y. Lam. “Adaptive smoothed particle hydrodynamics for high strain hydrodynamics with material strength”. In: *Shock Waves* 15.1 (Mar. 2006), pp. 21–29. ISSN: 1432-2153. DOI: 10.1007/s00193-005-0002-1. URL: https://doi.org/10.1007/s00193-005-0002-1.

[44] T. Rabczuk and J. Eibl. “Simulation of high velocity concrete fragmentation using SPH/MLSPH”. In: *International Journal for Numerical Methods in Engineering* 56.10 (2003), pp. 1421–1444. DOI: 10.1002/nme.617. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/nme.617. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/nme.617.

[45] Timon Rabczuk, Josef Eibl, and Lothar Stempniewski. “Numerical analysis of high speed concrete fragmentation using a meshfree Lagrangian method”. In: *Engineering Fracture Mechanics* 71.4 (2004). Fracture and Damage Mechanics, pp. 547–556. ISSN: 0013-7944. DOI: https://doi.org/10.1016/S0013-7944(03)00032-8. URL: http://www.sciencedirect.com/science/article/pii/S0013794403000328.

[46] Mahesh Prakash and Paul W. Cleary. “Modelling highly deformable metal extrusion using SPH”. In: *Computational Particle Mechanics* 2.1 (May 2015), pp. 19–38. ISSN: 2196-4386. DOI: 10.1007/s40571-015-0032-0. URL: https://doi.org/10.1007/s40571-015-0032-0.

[47] D. Stora et al. “Animating Lava Flows”. In: *Graphics Interface (GI’99) Proceedings*. Kingston, Ontario, Canada, 1999, pp. 203–210. URL: https://hal.inria.fr/inria-00510066.

[48] E. Patino Narino et al. “Implementation of the Smoothed Particle Hypdrodynamics Method to Solve Plastic Deformation in Metals”. In: (May 2014), pp. 4593–4606.

[49] A. Peer et al. “An Implicit SPH Formulation for Incompressible Linearly Elastic Solids”. In: *Computer Graphics Forum* 37.6 (2017), pp. 135–148. DOI: 10.1111/cgf.13317. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.13317. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13317.

[50] Nadir Akinci et al. “Coupling elastic solids with smoothed particle hydrodynamics fluids”. In: *Computer Animation and Virtual Worlds* 24.3-4 (2013), pp. 195–203. DOI: 10.1002/cav.1499. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cav.1499. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cav.1499.

[51] J.P. Gray, J.J. Monaghan, and R.P. Swift. “SPH elastic dynamics”. In: *Computer Methods in Applied Mechanics and Engineering* 190.49 (2001), pp. 6641–6662. ISSN: 0045-7825. DOI: https://doi.org/10.1016/S0045-7825(01)00254-7. URL: http://www.sciencedirect.com/science/article/pii/S0045782501002547.

[52] J.W. Swegle et al. “An analysis of smoothed particle hydrodynamics”. In: (Mar. 1994). DOI: 10.2172/10159839.

[53] D.J. Price. “Smoothed particle hydrodynamics and magnetohydrodynamics”. In: *Journal of Computational Physics* 231.3 (2012). Special Issue: Computational Plasma Physics, pp. 759–794. ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2010.12.011. URL: http://www.sciencedirect.com/science/article/pii/S0021999110006753.

[54] J.J. Monaghan. “SPH without a Tensile Instability”. In: *Journal of Computational Physics* 159.2 (2000), pp. 290–311. ISSN: 0021-9991. DOI: https://doi.org/10.1006/jcph.2000.6439. URL: http://www.sciencedirect.com/science/article/pii/S0021999100964398.

[55] T. Belytschko et al. “Meshless methods: An overview and recent developments”. In: *Computer Methods in Applied Mechanics and Engineering* 139.1 (1996), pp. 3–47. ISSN: 0045-7825. DOI: https://doi.org/10.1016/S0045-7825(96)01078-X. URL: http://www.sciencedirect.com/science/article/pii/S004578259601078X.

[56] J. Bonet and T.-S.L. Lok. “Variational and momentum preservation aspects of Smooth Particle Hydrodynamic formulations”. In: *Computer Methods in Applied Mechanics and Engineering* 180.1 (1999), pp. 97–115. ISSN: 0045-7825. DOI: https://doi.org/10.1016/S0045-7825(99)00051-1. URL: http://www.sciencedirect.com/science/article/pii/S0045782599000511.

[57] Gary A. Dilts. “Moving-least-squares-particle hydrodynamics—I. Consistency and stability”. In: *International Journal for Numerical Methods in Engineering* 44.8 (), pp. 1115–1155. DOI: 10.1002/(SICI)1097-0207(19990320)44:8<1115::AID-NME547>3.0.CO;2-L. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/%28SICI%291097-0207%2819990320%2944%3A8%3C1115%3A%3AAID-NME547%3E3.0.CO%3B2-L.

[58] Y. Krongauz and T. Belytschko. “Consistent pseudo-derivatives in meshless methods”. In: *Computer Methods in Applied Mechanics and Engineering* 146.3 (1997), pp. 371–386. ISSN: 0045-7825. DOI: https://doi.org/10.1016/S0045-7825(96)01234-0. URL: http://www.sciencedirect.com/science/article/pii/S0045782596012340.

[59] Larry D. Libersky et al. “Recent improvements in SPH modeling of hypervelocity impact”. In: *International Journal of Impact Engineering* 20.6 (1997). Hypervelocity Impact, pp. 525–532. ISSN: 0734-743X. DOI: https://doi.org/10.1016/S0734-743X(97)87441-6. URL: http://www.sciencedirect.com/science/article/pii/S0734743X97874416.

[60] J. K. Chen, J. E. Beraun, and T. C. Carney. “A corrective smoothed particle method for boundary value problems in heat conduction”. In: *International Journal for Numerical Methods in Engineering* 46.2 (), pp. 231–252. DOI: 10.1002/(SICI)1097-0207(19990920)46:2<231::AID-NME672>3.0.CO;2-K.

[61] P.W. Randles and L.D. Libersky. “Smoothed Particle Hydrodynamics: Some recent improvements and applications”. In: *Computer Methods in Applied Mechanics and Engineering* 139.1 (1996), pp. 375–408. ISSN: 0045-7825. DOI: https://doi.org/10.1016/S0045-7825(96)01090-0. URL: http://www.sciencedirect.com/science/article/pii/S0045782596010900.

[62] Hagit Schechter and Robert Bridson. “Ghost SPH for Animating Water”. In: *ACM Trans. Graph.* 31.4 (July 2012), 61:1–61:8. ISSN: 0730-0301. DOI: 10.1145/2185520.2185557. URL: http://doi.acm.org/10.1145/2185520.2185557.

[63] K.O. Friedrichs. “The Identity of Weak and Strong Extensions of Differential Operators”. In: *Transactions of the American Mathematical Society* 55.1 (Jan. 1944), pp. 132–151. URL: http://jstor.org/stable/19910143.

[64] I. Zisis. “From Continuum Mechanics to Smoothed Particle Hydrodynamics for shocks through inhomogeneous media”. PhD thesis. Eindhoven: Technische Universiteit Eindhoven, 2017. URL: https://research.tue.nl/en/publications/from-continuum-mechanics-to-smoothed-particle-hydrodynamics-for-s.

[65] J.J. Monaghan and J.C. Lattanzio. “A Refined Particle Method for Astrophysical Problems”. In: *Astronomy and astrophysics* 149 (1985), pp. 135–143.

[66] Walter Dehnen and Hossam Aly. “Improving convergence in smoothed particle hydrodynamics simulations without pairing instability”. In: *Monthly Notices of the Royal Astronomical Society* 425.2 (2012), pp. 1068–1082.

[67] Alexandre M Tartakovsky et al. “Smoothed particle hydrodynamics and its applications for multiphase flow and reactive transport in porous media”. In: *Computational Geosciences* 20.4 (2016), pp. 807–834.

[68] J.M. Domínguez et al. “Neighbour Lists in Smoothed Particle Hydrodynamics”. In: *International Journal for Numerical Methods in Fluids* (2010). DOI: 10.1002/fld.2481.

[69] Ligang Liu et al. “A Local/Global Approach to Mesh Parameterization”. In: *Computer Graphics Forum* 27.5 (), pp. 1495–1504. DOI: 10.1111/j.1467-8659.2008.01290.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2008.01290.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2008.01290.x.

[70] J.S.B. Mitchell, D.M. Mount, and C.H. Papadimitriou. “The Discete Geodesic Problem”. In: *SIAM J. Comput.* 16.4 (Aug. 1987), pp. 647–668.

[71] V. Surazhsky et al. “Fast Exact and Approximate Geodesics on Meshes”. In: *ACM Trans. Graph.* 24.3 (July 2005), pp. 553–560. ISSN: 0730-0301. DOI: 10.1145/1073204.1073228. URL: http://doi.acm.org.dianus.libr.tue.nl/10.1145/1073204.1073228.

[72] J. A. Sethian. “A fast marching level set method for monotonically advancing fronts”. In: *Proceedings of the National Academy of Sciences* 93.4 (Feb. 1996), pp. 1591–1595. URL: http://www.pnas.org/content/93/4/1591.abstract.

[73] H. Zhao. “A fast sweeping method for eikonal equations”. In: *Mathematics of Computation* 74 (2005), pp. 603–627. URL: http://adsabs.harvard.edu/cgi-bin/nph-bib%5C_query?bibcode=2005MaCom..74..603Z.

[74] Keenan Crane, Clarisse Weischedel, and Max Wardetzky. “Geodesics in Heat”. In: *CoRR* abs/1204.6216 (2012). arXiv: 1204.6216. URL: http://arxiv.org/abs/1204.6216.

[75] AZoM. *Silicone Rubber*. 2001. URL: https://www.azom.com/properties.aspx?ArticleID=920 (visited on 09/04/2018).

[76] Leonardo Di G Sigalotti and Hender López. “Adaptive kernel estimation and SPH tensile instability”. In: *Computers & Mathematics with Applications* 55.1 (2008), pp. 23–50.

[77] JW Swegle, DL Hicks, and SW Attaway. “Smoothed particle hydrodynamics stability analysis”. In: *Journal of computational physics* 116.1 (1995), pp. 123–134.

[78] Ke-Lin Du and Madisetti NS Swamy. *Neural networks and statistical learning*. Springer Science & Business Media, 2013.
[79] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. "Searching for Activation Functions". In: *CoRR* abs/1710.05941 (2017). arXiv: 1710.05941. URL: http://arxiv.org/abs/1710.05941.
[80] Jitendra Malik and Pietro Perona. "Preattentive texture discrimination with early vision mechanisms". In: *J. Opt. Soc. Am. A* 7.5 (May 1990), pp. 923–932. DOI: 10.1364/JOSAA.7.000923. URL: http://josaa.osa.org/abstract.cfm?URI=josaa-7-5-923.
[81] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. "Rectifier nonlinearities improve neural network acoustic models". In: *Proc. icml*. Vol. 30. 1. 2013, p. 3.
[82] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. "Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)". In: *CoRR* abs/1511.07289 (2015). arXiv: 1511.07289. URL: http://arxiv.org/abs/1511.07289.
[83] Kaiming He et al. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification". In: *The IEEE International Conference on Computer Vision (ICCV)*. Dec. 2015.
[84] M.A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015.
[85] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.
[86] Kyunghyun Cho et al. "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation". In: *CoRR* abs/1406.1078 (2014). arXiv: 1406.1078. URL: http://arxiv.org/abs/1406.1078.
[87] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.
[88] Kunihiko Fukushima and Sei Miyake. "Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Visual Pattern Recognition". In: *Competition and Cooperation in Neural Nets*. Ed. by Shun-ichi Amari and Michael A. Arbib. Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, pp. 267–285. ISBN: 978-3-642-46466-9.
[89] Fisher Yu and Vladlen Koltun. "Multi-Scale Context Aggregation by Dilated Convolutions". In: *CoRR* abs/1511.07122 (2015). arXiv: 1511.07122. URL: http://arxiv.org/abs/1511.07122.
[90] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. "Learning representations by back-propagating errors". In: *nature* 323.6088 (1986), p. 533.
[91] Jürgen Schmidhuber. "Deep learning in neural networks: An overview". In: *Neural Networks* 61 (2015), pp. 85–117. ISSN: 0893-6080. DOI: https://doi.org/10.1016/j.neunet.2014.09.003. URL: http://www.sciencedirect.com/science/article/pii/S0893608014002135.
[92] Atilim Gunes Baydin et al. "Automatic differentiation in machine learning: a survey." In: *Journal of machine learning research* 18.153 (2017), pp. 1–153.
[93] Yoshua Bengio. "Practical recommendations for gradient-based training of deep architectures". In: *CoRR* abs/1206.5533 (2012). arXiv: 1206.5533. URL: http://arxiv.org/abs/1206.5533.
[94] Ning Qian. "On the momentum term in gradient descent learning algorithms". In: *Neural Networks* 12.1 (1999), pp. 145–151. ISSN: 0893-6080. DOI: https://doi.org/10.1016/S0893-6080(98)00116-6. URL: http://www.sciencedirect.com/science/article/pii/S0893608098001166.
[95] Yurii Nesterov. "A method for unconstrained convex minimization problem with the rate of convergence O (1/kˆ 2)". In: *Doklady AN USSR*. Vol. 269. 1983, pp. 543–547.
[96] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive subgradient methods for online learning and stochastic optimization". In: *Journal of Machine Learning Research* 12.Jul (2011), pp. 2121–2159.
[97] Matthew D Zeiler. "ADADELTA: an adaptive learning rate method". In: *arXiv preprint arXiv:1212.5701* (2012).
[98] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *CoRR* abs/1412.6980 (2014). arXiv: 1412.6980. URL: http://arxiv.org/abs/1412.6980.
[99] Richard R. Picard and Kenneth N. Berk. "Data Splitting". In: *The American Statistician* 44.2 (1990), pp. 140–147. DOI: 10.1080/00031305.1990.10475704. URL: https://www.tandfonline.com/doi/abs/10.1080/00031305.1990.10475704.
[100] Sylvain Arlot, Alain Celisse, et al. "A survey of cross-validation procedures for model selection". In: *Statistics surveys* 4 (2010), pp. 40–79.
[101] Z Reitermanova. "Data splitting". In: *WDS*. Vol. 10. 2010, pp. 31–36.
[102] Yann A. LeCun et al. "Efficient BackProp". In: *Neural Networks: Tricks of the Trade: Second Edition*. Ed. by Grégoire Montavon, Geneviève B. Orr, and Klaus-Robert Müller. Berlin, Heidelberg: Springer Berlin Heidelberg,

2012, pp. 9–48. ISBN: 978-3-642-35289-8. DOI: 10.1007/978-3-642-35289-8_3. URL: https://doi.org/10.1007/978-3-642-35289-8_3.

[103] K. Ahmed, N.S. Keskar, and R. Socher. "Weighted Transformer Network for Machine Translation". In: (2017). URL: https://arxiv.org/pdf/1711.02132.pdf.

[104] B. Zoph et al. "Learning Transferable Architectures for Scalable Image Recognition". In: (2018). URL: https://arxiv.org/pdf/1707.07012.pdf.

[105] D. Silver, K. Simonyan, and J. Schrittwieser. "Mastering the Game of Go Without Human Knowledge". In: *Nature* 550 (2017), pp. 354–359. DOI: 10.1038/nature24270. URL: http://www.gwern.net/docs/rl/2017-silver.pdf.

[106] Jun Takeuchi and Yukio Kosugi. "Neural network representation of finite element method". In: *Neural Networks* 7.2 (1994), pp. 389–395. ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(94)90031-0. URL: http://www.sciencedirect.com/science/article/pii/0893608094900310.

[107] G. Balokas, S. Czichon, and R. Rolfes. "Neural network assisted multiscale analysis for the elastic properties prediction of 3D braided composites under uncertainty". In: *Composite Structures* 183 (2018), pp. 550–562. ISSN: 0263-8223. DOI: https://doi.org/10.1016/j.compstruct.2017.06.037.. URL: http://www.sciencedirect.com/science/article/pii/S0263822317302416.

[108] S. Jung and J. Ghaboussi. "Neural network constitutive model for rate-dependent materials". In: *Computers and Structures* 84.15 (2006), pp. 955–963. ISSN: 0045-7949. DOI: https://doi.org/10.1016/j.compstruc.2006.02.015. URL: http://www.sciencedirect.com/science/article/pii/S0045794906000563.

[109] Shaojuan Su, Yong Hu, and Wang C Fang. "Numerical Simulation and Neural Network Prediction the Cold Bending Spring back for Ship Hull Plate". In: *Open Automation and Control Systems Journal* 6 (2014), pp. 181–187.

[110] Chun-yu JIA et al. "Modeling and Simulation of Hydraulic Roll Bending System Based on CMAC Neural Network and PID Coupling Control Strategy". In: *Journal of Iron and Steel Research, International* 20.10 (2013), pp. 17–22. ISSN: 1006-706X. DOI: https://doi.org/10.1016/S1006-706X(13)60170-3. URL: http://www.sciencedirect.com/science/article/pii/S1006706X13601703.

[111] Mateusz Kaczmarek and Agnieszka Szymańska. "Application of artificial neural networks to predict the deflections of reinforced concrete beams". In: *Studia Geotechnica et Mechanica* 38.2 (2016), pp. 37–46. URL: https://content.sciendo.com/view/journals/sgem/38/2/article-p37.xml.

[112] Modjtaba Baymani, Asghar Kerayechian, and Sohrab Effati. "Artificial neural networks approach for solving stokes problem". In: *Applied Mathematics* 1.04 (2010), p. 288.

[113] J. Tompson et al. "Accelerating Eulerian Fluid Simulation With Convolutional Networks". In: *CoRR* abs/1607.03597 (2016). arXiv: 1607.03597. URL: http://arxiv.org/abs/1607.03597.

[114] I.E. Lagaris, A. Likas, and D.I. Fotiadis. "Artificial Neural Networks for Solving Ordinary and Partial Differential Equations". In: (1997). URL: https://arxiv.org/pdf/physics/9705023.pdf.

[115] B. Csaji. "Approximation with Artificial Neural Networks". MA thesis. Eötvös Loránd University, Hungary, 2001. DOI: 10.1.1.101.2647. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.101.2647&rep=rep1&type=pdf.

[116] "Geometrical Design: Bézier Curves and Surfaces". In: *An introduction to Scientific Computing*. Ed. by I. Danaila et al. Springer, 2006, pp. 193–212. ISBN: 978-0-387-30889-0.

[117] *CS231n Convolutional Neural Networks for Visual Recognition, Stanford CS class*. 2014. URL: http://cs231n.github.io/convolutional-networks/ (visited on 09/08/2018).

[118] Jeff Bezanson et al. "A Fresh Approach to Numerical Computing". In: *SIAM Review* 59 (2017), pp. 65–98. DOI: 10.1137/141000671. URL: http://julialang.org/publications/julia-fresh-approach-BEKS.pdf.

[119] M. Innes. "Flux: Elegant Machine Learning with Julia". In: *Journal of Open Source Software* (2018). DOI: 10.21105/joss.00602.

[120] Alec Jacobson, Daniele Panozzo, et al. *libigl: A simple C++ geometry processing library*. http://libigl.github.io/libigl/. 2017.

Appendix

In the following pages, the code written for the thesis is presented. Everything, except for the FEM analysis, is written in Julia version 0.6.3 [118]. Especially since Julia recently released version 1.0, we list here which packages were used together with their version number:

Flux for the neural networks, version 0.5.3 [119],

libigl for the ARAP parametrization [120],

StaticArrays, version 0.7.2,

Roots, version 0.6.0,

Plots, version 0.17.4,

PyCall, version 1.17.1,

PyPlot, version 2.6.0,

BSON, version 0.1.3,

SimpleWeightedGraphs, version 0.4.1,

LightGraphs, version 0.12.0.

Not all packages are needed to run one file. At the start of the document, all packages used are listed after the keyword “using”. As a final note, there is quite a bit of code duplication. The reason is to keep dependencies per application as light as possible. So, e.g., the Mass-Spring System code only depends on PyPlot and StaticArrays and not also PyCall.

For the FEM analysis, we used Python version 3.6 with the package nutils, version 2.0 [28].

A Mass-Spring System

Listing 1 *Demo file using BlowLogicSpring.jl.*

```
println("@@@ STARTING")

include("BlowLogicSpring.jl")
using BlowLogicSpring
bls = BlowLogicSpring
using FileIO

ρ = 10.
k = 500
h = .02

# Creation of spring types

linear = bls.Linear(ρ, k)
hyper1 = bls.Hyper1(ρ, k)
hyper2 = bls.Hyper2(ρ, k)
hyper3 = bls.Hyper3(ρ, k)
hyper4 = bls.Hyper4(ρ, k)

spring = hyper4

# Creation via filename
filename = "../data/teddy2"
teddy = bls.BlowSim{typeof(spring)}(filename, spring, h, zerobased = false);
# Or via loading vertices and faces in another way:
plane = load("../data/plane_with_crosslinks.obj")
V = [Array(vertex) for vertex in plane.vertices]
F = [Array{Int64,1}(face) for face in plane.faces]
plane = bls.BlowSim{typeof(spring)}(V, F, spring, h)

plane2 = load("../data/plane_with_crosslinks_fine.obj")
V = [Array(vertex) for vertex in plane2.vertices]
F = [Array{Int64,1}(face) for face in plane2.faces]
plane2 = bls.BlowSim{typeof(spring)}(V, F, spring, h)
```

```
spot = load("../data/spot_triangulated.obj")
V = [Array(vertex) for vertex in spot.vertices]
F = [Array{Int64,1}(face) for face in spot.faces]
spot = bls.BlowSim{typeof(spring)}(V, F, spring, h)

# Parameters
mesh = deepcopy(spot)
pressure = .5
μ = .01 # artificial friction
timestep = 0.001
j = 0
Niter = 1000

for i = 1:Niter
    stop, fp, fs, ff, ε = bls.inflate!(mesh, pressure, μ, timestep)
    if any(isnan, mesh.V)
        println("Found NaN, quitting inflation at iteration $j")
        break
    end
    if all(stop)
        println("Stopping inflation, due to force balance, at iteration $j.")
        break
    end
    j +=1
end

# Uncomment and use at own risk: Makie does not work for me
# bls.showmesh(teddy.V, elasticteddy.F)
bls.showplot(mesh.V, mesh.F, p = pressure, μ = μ, dt = timestep, nsteps = j)

bls.showplot(mesh.V, mesh.F, p = pressure, μ = μ, dt = timestep, nsteps = j,
                meshname = "mesh2", save = true, spring = spring)
```

Listing 2 *The main Mass-Spring System code.*

```
module BlowLogicSpring

using StaticArrays
# Plotting. Makie does not work currently for me.
# using Makie, Colors, GeometryTypes
using PyPlot

const Vec = SVector{3,Float64}
const Edge = SVector{2, Int}
const Face = SVector{3, Int}

const VectorList = Array{Vec,1}
const EdgeList = Array{Edge,1}
const FaceList = Array{Face,1}
const ScalarList = Array{Float64,1}

const IntList = Array{Int64,1}
const IntIntList = Array{Array{Int64,1},1}

#import Base.isless

function edge_lt(e1 :: Edge, e2 :: Edge)
    e1[1] < e2[1] || (e1[1] == e2[1] && e1[2] < e2[2])
end

function load_vertices(filename :: String)
    # Read as matrix
    Vmat = readdlm(filename)
    # Convert to VectorList
    [Vec(Vmat[i,:]) for i in 1:size(Vmat,1)]
end

function load_faces(filename :: String; zerobased :: Bool = true)
    # Read as matrix and convert to 1 based numbering if needed
    Fmat = readdlm(filename) + 1 * zerobased
    # Convert to FaceList
    [Face(Fmat[i,:]) for i in 1:size(Fmat,1)]
end

function ordered_edge(v1 :: Int, v2 :: Int)
    Edge(minmax(v1, v2)...)
end
```

```
function extract_edges(F :: FaceList)
    E = EdgeList()
    for f in F
        push!(E, ordered_edge(f[1], f[2]))
        push!(E, ordered_edge(f[2], f[3]))
        push!(E, ordered_edge(f[3], f[1]))
    end

    # Return without duplicates
    unique(sort(E, lt=edge_lt), 1)
end

"""Create an array of arrays containing at index v the indices of the faces
adjacent to v"""
function build_face_map(nv :: Int, F :: FaceList)
    fmap = [IntList() for k in 1:nv]

    for (k, f) in enumerate(F)
        push!(fmap[f[1]], k)
        push!(fmap[f[2]], k)
        push!(fmap[f[3]], k)
    end

    fmap
end

"""
Compute the edge maps: for each vertex, finds its adjacent edges. This function
returns two maps: one with edges going out of the vertex (positive direction)
and one incident to the vertex (negative direction)
"""
function build_edge_maps(nv :: Int, E :: EdgeList)
    emap_pos = [IntList() for k in 1:nv]
    emap_neg = [IntList() for k in 1:nv]

    for (k,e) in enumerate(E)
        push!(emap_pos[e[1]], k)
        push!(emap_neg[e[2]], k)
    end

    emap_pos, emap_neg
end

"""
Compute the edge to face map. Given the index of an edge, returns the faces
adjacent to that edge.
"""
function build_edge_facemap(E :: EdgeList, fmap :: IntIntList)
    edge_fmap = [IntList() for k in 1:length(E)]
    for (k, e) in enumerate(E)
        # The faces adjacent to an edge are the shared faces of its vertices
        edge_fmap[k] = intersect(fmap[e[1]], fmap[e[2]])
    end
    edge_fmap
end

"""
Method to compute boundary vertices of a mesh. Returns boundary vertex indices.
Returns empty arrays if the mesh is watertight.
"""
function computeboundary(E :: EdgeList, edge_fmap :: IntIntList)
    boundary_vertices = IntList()
    for (k, faces) in enumerate(edge_fmap)
        len = length(faces)
        # If the edge is adjacent to just once face, it is on the boundary
        if len == 1
            push!(boundary_vertices, E[k][1], E[k][2])
        elseif len == 0 || len == 2
            continue
        else
            error("Edge $k shares more than two faces: $faces.")
        end
    end
    sort(unique(boundary_vertices))
end

abstract type SpringMaterial
end

struct Linear <: SpringMaterial
    ρ :: Float64     # Density
    k :: Float64     # Spring constant
    name :: String
```

```
    Linear(ρ, k) = new(ρ, k, "linear")
end

struct Hyper1 <: SpringMaterial
    ρ :: Float64     # Density
    k :: Float64     # Spring constant
    name :: String
    Hyper1(ρ, k) = new(ρ, k, "hyper1")
end

struct Hyper2 <: SpringMaterial
    ρ :: Float64     # Density
    k :: Float64     # Spring constant
    name :: String
    Hyper2(ρ, k) = new(ρ, k, "hyper2")
end

struct Hyper3 <: SpringMaterial
    ρ :: Float64     # Density
    k :: Float64     # Spring constant
    name :: String
    Hyper3(ρ, k) = new(ρ, k, "hyper3")
end

struct Hyper4 <: SpringMaterial
    ρ :: Float64     # Density
    k :: Float64     # Spring constant
    name :: String
    Hyper4(ρ, k) = new(ρ, k, "hyper4")
end

# Hooke's law and get functions for material parameters
stress(mat :: Linear, ε :: Float64) = mat.k * ε
stress(mat :: Hyper1, ε :: Float64) = mat.k * (ε - .5 * exp(-4. * (ε - 1.)^2))
stress(mat :: Hyper2, ε :: Float64) =
    mat.k * (ε - .9 * (ε - .5)*(ε > .5) + 2. * (ε - 1.)*(ε > 1.))
stress(mat :: Hyper3, ε :: Float64) =
    mat.k * (log(ε + 1)*(ε < e - 1.) + (ε ãĽě e - 1.))
stress(mat :: Hyper4, ε :: Float64) = mat.k * (.33*ε^3 - 1.*ε^2 + 1.5*ε)
density(material :: SpringMaterial) = material.ρ
springconstant(material :: SpringMaterial) = material.k

mutable struct BlowSim{T<:SpringMaterial}

    material :: T

    # Number of vertices, faces and edges
    nv :: Int64
    nf :: Int64
    ne :: Int64

    # Convenient representation of the mesh
    V :: VectorList
    F :: FaceList
    E :: EdgeList

    # Lookup tables
    fmap :: IntIntList
    emap_pos :: IntIntList
    emap_neg :: IntIntList
    edge_fmap :: IntIntList
    boundary :: IntList
    interior :: IntList

    #
    # Derived geometric quantities (normals, areas, lengths)
    #

    Nf :: VectorList
    Af :: ScalarList

    Te :: VectorList
    Le :: ScalarList

    Nv :: VectorList
    Av :: ScalarList

    Vf :: ScalarList

    #
    # Dynamic parameters
    #
    ρ :: Float64
```

```
    Le0 :: ScalarList    # Edge lengths at rest
    M :: ScalarList      # Vertex mass
    U :: VectorList      # Vertex velocity
    Vold :: VectorList   # Vertex initial location

    function BlowSim{T}(V :: VectorList, F :: FaceList,
                        material :: T, h :: Float64) where T <: SpringMaterial
        self = new()

        self.material = material
        self.ρ = density(material)

        # Load mesh
        self.V = V
        self.nv = length(self.V)
        self.F = F
        self.nf = length(self.F)
        self.E = extract_edges(self.F)
        self.ne = length(self.E)

        # Create look up tables
        self.fmap = build_face_map(self.nv, self.F)
        self.emap_pos, self.emap_neg = build_edge_maps(self.nv, self.E)
        self.edge_fmap = build_edge_facemap(self.E, self.fmap)
        self.boundary = computeboundary(self.E, self.edge_fmap)
        self.interior = setdiff(1:self.nv, self.boundary)

        # Pre-allocate space for derived gemoetrical information
        self.Nf = VectorList(self.nf) # face normals
        self.Af = ScalarList(self.nf) # Face areas

        self.Te = VectorList(self.ne) # edge tangentials
        self.Le = ScalarList(self.ne) # edge lengths

        self.Nv = VectorList(self.nv) # vertex normals
        self.Av = ScalarList(self.nv) # vertex areas

        # Compute derived geometric variables
        update_geometry!(self)

        # Compute mass, initial velocity, old position and face initial volume
        self.M = self.Av .* self.ρ * h
        self.U = zeros(Vec, self.nv)
        self.Vold = deepcopy(self.V)
        self.Vf = h * self.Af

        # Fix start lengths
        self.Le0 = deepcopy(self.Le)

        return self
    end

    function BlowSim{T}(filename :: String, material :: T, h :: Float64;
                        zerobased :: Bool = true) where T
        # Read from file
        V = load_vertices(filename * "-V.txt")
        F = load_faces(filename * "-F.txt", zerobased = zerobased)
        BlowSim{T}(V, F, material, h)
    end

    function BlowSim{T}(V :: Array{Array{Float32,1},1},
                        F :: Array{Array{Int64,1},1},
                        material, h :: Float64) where T
        V = [Vec(vertex) for vertex in V]
        F = [Face(face) for face in F]
        BlowSim{T}(V, F, material, h)
    end
end

function update_faces!(b :: BlowSim)
    for (k, f) in enumerate(b.F)
        v :: Vec = b.V[f[2]] - b.V[f[1]]
        w :: Vec = b.V[f[3]] - b.V[f[1]]

        b.Nf[k] = cross(v, w)

        mag = norm(b.Nf[k])
        b.Nf[k] /= mag
        b.Af[k] = 0.5 * mag
    end
end

function update_edges!(b :: BlowSim)
```

```
    for (k, e) in enumerate(b.E)
        b.Te[k] = b.V[e[2]] - b.V[e[1]]
        b.Le[k] = norm(b.Te[k])
        b.Te[k] /= b.Le[k]
    end
end

function update_vertices!(b :: BlowSim)
    for k = 1:b.nv
        Ns = Vec(0.0, 0.0, 0.0)
        As = 0.0

        for f in b.fmap[k]
            As += b.Af[f]
            Ns += b.Af[f] * b.Nf[f]
        end

        b.Av[k] = As / 3.0
        b.Nv[k] = Ns / norm(Ns)
    end
end

"""
Function to approximate the width belonging to an edge. Here, it assumes
that the volume of a face doet not change during inflation.
"""
function edge_area(b :: BlowSim)
    Ae = ScalarList(b.ne)
    nsamples = 20
    for (k, e) in enumerate(b.E)
        Ae[k] = 0.
        # The width is the height of a rectangle with area equal to 1/3
        # of the area of the faces, divided by the edge length
        Ae[k] = (sum(b.Vf[b.edge_fmap[k]])) / (3. * b.Le[k])
    end
    Ae
end

distanceline(v :: Vec, p :: Vec, d :: Vec) = norm(v - p - dot(v - p, d)*d)

function update_geometry!(b :: BlowSim)
    update_faces!(b)
    update_edges!(b)
    update_vertices!(b)
end

function inflate!(b :: BlowSim, pressure :: Float64, μ :: Float64, dt :: Float64
    )

    # Update normals etc
    update_geometry!(b)

    # Spring forces in each edge
    ε :: ScalarList = b.Le ./ b.Le0 - 1.0
    Ae :: ScalarList = edge_area(b)
    σ :: ScalarList = stress.(b.material, ε)

    # dt squared
    dt2 = dt * dt

    # stopping criterion
    stop = Array{Bool, 1}(length(b.interior))
    fill!(stop, false)

    # To compute average forces and strain during inflation
    fp, fs, ff, strain = 0., 0., 0., 0.

    @inbounds for (k, v) in enumerate(b.interior)
        # Pressure force
        Fp = pressure * b.Nv[v] * b.Av[v]

        # Collect spring force from edges to vertices
        Fs = Vec(0.0, 0.0, 0.0)
        for ep in b.emap_pos[v]
            t = b.Te[ep]
            t *= σ[ep] * Ae[ep]
            Fs += t
        end
        for en in b.emap_neg[v]
            t = b.Te[en]
            t *= σ[en] * Ae[en]
            Fs -= t
        end
```

```
375
376         # Friction forces
377         Ff = -μ .* b.U[v]
378
379         # Acceleration
380         a = (Fp + Fs + Ff) / b.M[v]
381
382         # Verlet update
383         Vnew = 2.0 * b.V[v] - b.Vold[v] + a * dt2
384
385         # Averages
386         fp += norm(Fp)
387         fs += norm(Fs)
388         ff += norm(Ff)
389
390         # Swap V and Vold
391         b.Vold[v] = b.V[v]
392         b.V[v] = Vnew
393
394         # Speed update
395         b.U[v] = (b.V[v] - b.Vold[v]) ./ dt
396
397         # Track whether we can stop yet
398         if norm(Fp + Fs + Ff) < 1e-13
399             stop[k] = true
400         elseif norm(b.Vold[v] - Vnew) < 1e-13
401             stop[k] = true
402         end
403     end
404     stop, fp/b.nv, fs/b.nv, ff/b.nv, sum(ε) / b.ne
405 end
406
407 """ Plotting functions """
408 # function showmesh(V :: VectorList, F :: FaceList)
409 #     nv = length(V)
410 #     nf = length(F)
411 #     V = [Point3f0(V[i]) for i in 1:nv]
412 #     F = [GLTriangle(F[i]) for i in 1:nf]
413 #     mesh = GLNormalMesh(V, F)
414 #
415 #     scene = Scene(color=:white)
416 #     Makie.wireframe(mesh, linewidth=2f0, color=:gray)
417 #     Makie.mesh(mesh, color=:darkgray)
418 #
419 #     pos = map(mesh.vertices, mesh.normals) do p, n
420 #        p => p .+ (normalize(n) .* 4f0)
421 #     end
422 #     # linesegment(pos, color=RGBA(0.5,0.5,1.0,0.8))
423 #
424 #     center!(scene)
425 #     scene
426 # end
427
428 function showplot(VL :: VectorList, F :: FaceList;
429     spring :: SpringMaterial = Linear(1., 10.), p = 0., μ = 0.1, dt = 0.001,
430     nsteps = 0, meshname = "mesh", save :: Bool = false)
431     F = [f[i]-1 for f in F, i=1:3] # minus 1, because PyPlot is Python
432     xs = ScalarList(length(VL))
433     ys = ScalarList(length(VL))
434     zs = ScalarList(length(VL))
435     for (k, v) in enumerate(VL)
436       xs[k] = v[1]
437       ys[k] = -v[3]
438       zs[k] = v[2]
439     end
440     fig = figure()
441     plot_trisurf(xs, ys, zs, triangles = F, edgecolor = "white", linewidth =.5)
442     #, color="white", alpha=0., edgecolor = "gray")
443     s = round(nsteps*dt, 2)
444     title("Spot after $s seconds, with pressure $p and μ = $μ")
445     xlabel("x-axis")
446     ylabel("y-axis")
447     zlabel("z-axis")
448     # axis("off")
449
450     # gca()[:set_zlim](0., 1.)
451     gca()[:view_init](elev=10., azim=0.)
452     gca()[:grid](false)
453     k = spring.k
454     if save
455         savefig(meshname*"_$p"*"_$μ"*"_$dt"*"_$nsteps"*spring.name*"_$k.pdf")
456     end
457     show()
```

```
458 end
459
460 end # module BlowLogicSpring
```

B Smoothed Particle Hydrodynamics

B.1 2D Code

Listing 3 *Demo file using SPH2D.jl*

```
1  println("@@@ STARTING")
2
3  include("SPH2D.jl")
4
5  using SPH2D
6  using Plots
7
8  L = 6.
9  nv = 200
10 ngv = 40
11 h = 0.4
12 # Can choose between circle, square, ellipse, squircle, circlearc, parabola and
       line.
13 shape = "squircle"
14 rho = 1.0
15
16 kernelname = "wendland"
17 fixedlength = SPH2D.Fixed(h, kernelname)
18 adaptivelength = SPH2D.Adaptive(nv, kernelname)
19
20 λ = 8.11e6
21 μ = 9.6e5
22 δ = 2.54e-4
23 balloon =  SPH2D.LinearElastic(rho, δ*λ, δ*μ)
24
25 smoothinglength = fixedlength
26 material = balloon
27
28 println("--- Warming up")
29 b = SPH2D.BlowSim{typeof(material), typeof(smoothinglength)}(
30     nv, ngv, L, δ, material, smoothinglength, shape);
31 println();
32 SPH2D.inflate!(b, 10., 0.00001, 0.02)
33
34 print("--- Creation:")
35 b = SPH2D.BlowSim{typeof(material), typeof(smoothinglength)}(
36     nv, ngv, L, δ, material, smoothinglength, shape);
37 print("--- Plot:")
38 # Throws a method error the first time, second time runs fine.
39 @time SPH2D.showplot(b.V, [b.V-b.V], b.nv, 0., L)
40 # savefig("finalsquircle_p1000_ms3000_fixedlength2.pdf")
41 # [b.a .* b.M + b.fp, b.ff]
42 # b.a .* b.M, b.fp, b.ff
43 # [b.Nv]
44
45 timestep = 0.00001
46 mu = 0.4
47 nsteps = 500
48 pmax = 1000.
49 maxsteps = 3000
50 totaltime = 0.
51 dt = 0.
52 totalsteps = 0
53 T = maxsteps * timestep
54
55 v = Int[1, 20, 50, 75]
56 Fp = [Float64[] for i = 1:length(v)]
57 Fs = [Float64[] for i = 1:length(v)]
58 Ff = [Float64[] for i = 1:length(v)]
59 Lv = [Float64[] for i = 1:length(v)]
60 P = Float64[]
61 ds = Float64[]
62 Lv0 = deepcopy(b.Lv)
63
64 println("---$nsteps steps, maximum pressure: $pmax, timestep: $timestep, mu: $mu
       ")
```

```
@time for m = 1:nsteps
    pressure = pmax^2 * min(1, sum(ds) / T)
    push!(P, pressure)
    try
        dt = SPH2D.inflate!(b, sqrt(pressure), timestep, mu)
        push!(ds, dt)
    catch err
        if isa(err, ArgumentError)
            println(ArgumentError)
            println("Quit loop at iteration "*string(m))
            break
        end
    end

    for i = 1:length(v)
        push!(Fp[i], norm(b.fp[v[i]]))
        push!(Fs[i], norm(b.a[v[i]] * b.M[v[i]]))
        push!(Ff[i], norm(b.ff[v[i]]))
        push!(Lv[i], b.Lv[v[i]])
    end

    if any(isnan, b.V)
        println("NaN found, quit loop at iteration $m.")
        break
    end
    if m % 100 == 1
        # Plot the vectors which are the result of the pressure and elastic
        # forces and the artificial friction, scaled to the length of the
        # elastic forces, to not get crazy big vectors and small shapes.
        SPH2D.showplot(b.V, [b.a .* b.M + b.fp, b.ff], b.nv, sum(ds), L)
        println("At iteration $m.")
    end
    totalsteps += 1
end

# Plot forces
c = palette(:default)[1:length(v)]

times = cumsum(ds)
p1 = plot(title = "Force magnitude during inflation", xlabel = "Time",
    ylabel = "Force magnitude", legend = :topleft, ylims = (0.,40.))
n1 = 0
n2 = 1
n3 = length(times)
for i = 1:length(v)
    plot!(times[n2:end-n1], Fp[i][n2:n3-n1], label = "Pressure force, $i",
    line = :dash, c = c[i])
    plot!(times[n2:end-n1], Fs[i][n2:n3-n1], label = "Spring force, $i",
    line =:solid, c = c[i])
    plot!(times[n2:end-n1], Ff[i][n2:n3-n1], label = "Friction force, $i",
    line =:dot, c = c[i])
end
display(p1)

p2 = plot(times, sqrt.(P)[1:n3], xlabel = "time", ylabel ="pressure", legend =
     false)

p3 = plot(xlabel = "Time", ylabel = "Length", legend = :bottomleft, ylims = (0.,
     0.06))
for i=1:length(v)
    plot!(times[n2:end-n1], Lv[i][n2:n3-n1], label = "Vertex length, $i",
    line = :solid, c = c[i])
end
display(p3)
```

Listing 4 *Main SPH code.*

```
module SPH2D

using StaticArrays
using Roots
using Plots
using Interpolations

const ε = 1e-10

const Scalar = Float64
const Index = Int

```

```
const Vec = SVector{2, Scalar}
const Edge = SVector{2, Index}

const SparMat = SparseMatrixCSC{Scalar, Int64}
const Mat = SMatrix{2,2,Float64,4}
const MMat = MMatrix{2,2,Float64,4}
const MatList = Array{Mat, 1}
const MMatList = Array{MMat, 1}

const ScalarList = Array{Scalar,1}
const VectorList = Array{Vec,1}
const EdgeList = Array{Edge,1}
const ScalarMap = Array{ScalarList,1}
const VectorMap = Array{VectorList,1}

const IndexList = Array{Index,1}
const IndexMap = Array{IndexList,1}

""" Function to decide which shape vertices to use. """
function load_vertices(shape :: String, args...; kwargs...)
    if shape == "line"
        return load_vertices_line(args...; kwargs...)
    elseif shape == "parabola"
        return load_vertices_parabola(args...; kwargs...)
    elseif shape == "circlearc"
        return load_vertices_circlearc(args...; kwargs...)
    elseif shape == "circle"
        return load_vertices_circle(args...)
    elseif shape == "square"
        return load_vertices_square(args...)
    elseif shape == "ellipse"
        return load_vertices_ellipse(args...)
    elseif shape == "squircle"
        return load_vertices_squircle(args...)
    else
        error("No such shape implemented.")
    end
end

"""
Functions to create an array with vertex locations. Vertices are evenly spaced
where possible. If the shape is unconnected, standard 10 ghost vertices are
placed at both ends. In the case of a line, if ngv is uneven, (ngv-1)/2 are
placed on the left and (ngv+1)/2 on the right. If the shape is connected at the
first and last vertex, L is the circumference. Also, the vertices can start at
any point, but they need to go anti-clockwise from there, due to the
computation of the normal vectors later on. Depending on the exact shape, the
number of vertices returned may vary from the number specified.
"""
function load_vertices_line(nv :: Index, L :: Scalar; ngv :: Index = 10)
    n = nv + ngv
    spacing = L / (nv - 1)
    [Vec((i-1)*spacing - spacing * fld(ngv, 2), 0.0) for i in 1:n]
end

function load_vertices_parabola(nv :: Index, L :: Scalar; ngv :: Index = 10)
    n = nv + ngv
    # Note that the length returned is > L
    x = linspace(- 0.5 * L, 0.5 * L, n)
    [Vec(x[i], -x[i]^2) for i in 1:n]
end

function load_vertices_circlearc(nv :: Index, L :: Scalar; ngv :: Index = 10)
    # cospi and sinpi calculate their arugments times π.
    n = nv + ngv
    spacing = 1 / nv
    # factor (0.5 * L / π) to get a circle with circumference âĽĹ L.
    [Vec( -(0.5 * L / π) * cospi(spacing * (i - 1 - ngv * 0.5)),
          (0.5 * L / π) * sinpi(spacing * (i - 1 - ngv * 0.5))) for i in 1:n+1]
end

function load_vertices_circle(nv :: Index, L :: Scalar)
    # cospi and sinpi calculate their arugments times π.
    spacing = 2 / nv
    [Vec(  (0.5 * L / π) * cospi(spacing * i),
           (0.5 * L / π) * sinpi(spacing * i)) for i in 1:nv]
end

function load_vertices_ellipse(nv :: Index, L :: Scalar)
    b = L*0.1
    a = L*0.2
    spacing = 2 / nv
    [Vec(  a * cospi(spacing * i), b * sinpi(spacing * i)) for i in 1:nv]
```

```
end

function interp1(xpt, ypt, x; method=:linear, extrapvalue=nothing)

    if extrapvalue == nothing
        y = zeros(x)
        idx = trues(x)
    else
        y = extrapvalue*ones(x)
        idx = (x .>= xpt[1]) .& (x .<= xpt[end])
    end

    if method == :linear
        intf = Interpolations.interpolate((xpt,), ypt, Gridded(Linear()))
        y[idx] = intf[x[idx]]

    elseif method == :cubic
        itp = interpolate(ypt, BSpline(Cubic(Natural())), OnGrid())
        intf = scale(itp, xpt)
        y[idx] = [intf[xi] for xi in x[idx]]
    end

    return y
end

"""Make the discrete curve unit speed"""
function unit_speedify(points; closed=true)
    # Repeat first point if we deal with a closed loop
    if closed
        p = vcat(points, points[1:1])
    else
        p = points
    end

    # Compute the ordinate
    s = [0; cumsum(norm.(diff(p)))]

    # Compute equidistant ordinate mesh...
    si = linspace(minimum(s), maximum(s), length(s))

    # ... and get the coordinates there
    xi = interp1(s, [el[1] for el in p], si)
    yi = interp1(s, [el[2] for el in p], si)

    # Remove added point
    if closed
        xi = xi[1:end-1]
        yi = yi[1:end-1]
    end

    return [Vec(el) for el in zip(xi,yi)]
end


function load_vertices_squircle(nv :: Index, L :: Scalar)
    spacing = 2 / nv
    p = [Vec(sqrt(abs(cospi(spacing * i))) * sign(cospi(spacing * i)),
        sqrt(abs(sinpi(spacing * i))) * sign(sinpi(spacing * i))) for i in 1:nv]
    return unit_speedify(p)
end#

function load_vertices_square(nv :: Index, L :: Scalar)
    # Creates four edges of the square, each with length L / 4
    l = L / 4.0
    # the total number of vertices returned will be 4 * floor(nv/4).
    n = fld(nv, 4)
    spacing = l / n
    V1 = [Vec(l / 2.0 - (i-1) * spacing, l / 2.0) for i in 1:n]
    V2 = [Vec(- l / 2.0, l / 2.0 - (i-1) * spacing) for i in 1:n]
    V3 = [Vec(- l / 2.0 + (i-1) * spacing, - l / 2.0) for i in 1:n]
    V4 = [Vec(l / 2.0, -l / 2.0 + (i-1) * spacing) for i in 1:n]
    append!(V1, V2)
    append!(V1, V3)
    append!(V1, V4)
end

""" Loads the edges, depending on the shapetype (connected or not). """
function load_edges(shapetype :: String, nv :: Index)
    E = [Edge([i, i+1]) for i in 1:nv-1]
    if shapetype == "unconnected"
        return E
    elseif shapetype == "connected"
        push!(E, Edge([nv, 1]))
```

```
        return E
    else
        error("No such shape implemented.")
    end
end

# Abstract type to hold material parameters
abstract type SpringMaterial
end

# Saint Venant-Kirchhoff material
struct LinearElastic <: SpringMaterial
    rho :: Scalar
    λ :: Scalar
    μ :: Scalar
end

density(material :: SpringMaterial) = material.rho

lambda(material :: LinearElastic) = material.λ
mu(material :: LinearElastic) = material.μ

"""
Computes the second Piola-Kirchhoff stress tensor for linear elastic material
     models.
"""
function stress(material :: LinearElastic, F :: MMat)
    E = 0.5 * (F * F' -  eye(Mat))
    material.λ * trace(E) * eye(Mat) + 2.0 * material.μ * E
end

# Abstract type to hold parameters for the kernel function
abstract type Kernel
end

struct Fixed <: Kernel
    h :: Scalar  # smoothing length
    α :: Scalar  # dimensional scaling factor
    func :: Function # kernel function
    gradfunc :: Function # gradient of kernel function

    function Fixed(h :: Scalar, name :: String)
        if name == "cubicBspline"
            self = new(h, 1., cubicBspline, grad_cubicBspline)
        elseif name == "wendland"
            self = new(h, 4./3., wendland, grad_wendland)
        else
            error("no such kernel function")
        end
    end
end

struct Adaptive <: Kernel
    h :: ScalarList # smoothing length
    α :: Scalar  # dimensional scaling factor
    func :: Function # kernel function
    gradfunc :: Function # gradient of kernel function

    function Adaptive(nv :: Index, name :: String)
        if name == "cubicBspline"
            h = ScalarList()
            self = new(h, 1.0, cubicBspline, grad_cubicBspline)
        elseif name == "wendland"
            h = ScalarList()
            self = new(h, 3./4., wendland, grad_wendland)
        else
            error("no such kernel function")
        end
    end
end

smoothinglength(kernel :: Fixed, nv :: Index) = kernel.h
smoothinglength(kernel :: Adaptive, nv :: Index) = [0.0 for i = 1:nv]
kernelcorrection(kernel :: Kernel) = kernel.α     # Gets the value

""" Cubic Bspline kernel function. """
function cubicBspline(h :: Scalar, α :: Scalar, dist :: Scalar)
    s = dist / h
    if s >= 2.0
        return 0.0
    elseif 0.0 <= s < 1.0
        return α / h * ( 2.0/3.0 - s^2 + 0.5*s^3)
    else
```

```
261         return α / h * (2.0 - s)^3 / 6.0
262     end
263 end
264 
265 """
266 Gradient of cubic B spline kernel function on the line, so in 1 dimension.
267 """
268 function grad_cubicBspline(h :: Scalar, α :: Scalar, xi :: Vec, xj :: Vec, dist
        :: Scalar)
269     s = dist / h
270     if s >= 2.0 || s < ε
271         return Vec(0.0, 0.0)
272     elseif 0.0 < s < 1.0
273         return α / (h^2 * norm(xj-xi)^2) * dist * (-2.0*s + 1.5*s^2) * (xi - xj)
274     else 1.0 <= s < 2.0
275         return α / (h^2 * norm(xj-xi)^2) * dist * -(2.0 - s)^2 * 0.5 * (xi - xj)
276     end
277 end
278 
279 """ Wendland kernel """
280 function wendland(h :: Scalar, α :: Scalar, dist :: Scalar)
281     q = dist / h
282     if q >= 2.
283         return 0.
284     else
285         return α / h * (1. - .5 * q)^5 * (1. + 2.5 * q + 2. * q^2)
286     end
287 end
288 
289 """ Gradient of the wendland kernel """
290 function grad_wendland(h :: Scalar, α :: Scalar, xi :: Vec, xj :: Vec, dist ::
        Scalar)
291     q = dist / h
292     if q >= 2.0 || q < ε
293         return Vec(0.0, 0.0)
294     else
295         dwdq = - 7. * q * (1. - .5 * q)^4 * (.5 + q)
296         return α / (h^2 * norm(xj-xi)^2) * dist * (xi - xj) * dwdq
297     end
298 end
299 
300 """
301 Distance between two points on the line, given by the absolute value of the
302 difference of the values at v and ov in the list containing the cumulative sums
303 of the distances. For shapes of which the first and last vertex are connected,
304 this function assumes that the last entry of LS contains circumference
305 of the shape.
306 """
307 function distance_connected(LS :: ScalarList, v :: Index, ov :: Index)
308     min(abs(LS[v] - LS[ov]), LS[end] - abs(LS[v] - LS[ov]))
309 end
310 
311 function distance_unconnected(LS :: ScalarList, v :: Index, ov :: Index)
312     abs(LS[v] - LS[ov])
313 end
314 
315 """ Takes a matrix and changes its entries such that it is an idenity matrix """
316 function eye!(A :: MMat)
317     n, m = size(A)
318     for j = 1:m
319         for i = 1:n
320             @inbounds A[i,j] = 1.0(i == j)
321         end
322     end
323 end
324 
325 """
326 Returns for each vertex i the indices of the neighbors within 2.5*h.
327 """
328 function build_knn_map(h :: Scalar, V :: VectorList, nv :: Index,
329         distance :: Function, LS :: ScalarList)
330     knnmap = [IndexList() for k in 1:nv]
331 
332     for i = 1:nv
333         for j = (i+1):nv
334             if distance(LS, i, j) < 2.5 * h
335                 push!(knnmap[i], j)
336                 push!(knnmap[j], i)
337             end
338         end
339     end
340     knnmap
341 end
```

```
342
343 """
344 Returns for each vertex i the indices of the neighbors within a quarter of the
345     total length, since h is actually unknown at this point.
346 """
347 function build_knn_map(h :: ScalarList, V :: VectorList, nv :: Index,
348         distance :: Function, LS :: ScalarList)
349     knnmap = [IndexList() for k in 1:nv]
350
351     for i = 1:nv
352         for j = (i+1):nv
353             if distance(LS, i, j) < .25 * LS[end]
354                 push!(knnmap[i], j)
355                 push!(knnmap[j], i)
356             end
357         end
358     end
359     knnmap
360 end
361
362 """
363 Compute the edge maps: for each vertex, finds its two adjacent edges.
364 This function returns two maps: one with edges going out of
365 the vertex (clockwise) and one incident to the vertex
366 (anti-clockwise).
367 """
368 function build_edge_maps(nv :: Index, E :: EdgeList, shapetype :: String)
369     emap_pos = IndexList()
370     emap_neg = IndexList()
371
372     for e in E
373         push!(emap_pos, e[2])
374         push!(emap_neg, e[1]-1)
375     end
376
377     if shapetype == "connected"
378         emap_neg[1] = E[end][1]
379     elseif shapetype == "unconnected"
380         # The fist point sees itself as previous element, and the last one
381         # itself as the next element.
382         emap_neg[1] = 1
383         push!(emap_neg, nv - 1)
384         push!(emap_pos, nv)
385     else
386         error("No such shapetype implemented.")
387     end
388
389     emap_pos, emap_neg
390 end
391
392 """
393 Struct containing all relevant information of a mesh during and after inflation,
394 e.g., material parameters, vertex locations, number of vertices, density, etc.
395 Also contains the function to initialize the struct.
396 """
397 mutable struct BlowSim{T<:SpringMaterial, K<:Kernel}
398
399     # Kernel and material parameters, as well as the shape and distance function
400     kernel :: K
401     material :: T
402     shapetype :: String
403     distance :: Function
404
405     # Number of (ghost)vertices and edges
406     ngv :: Index
407     nv :: Index
408     ne :: Index
409
410     # Index of first and last moving particle
411     start_at :: Index
412     end_at ::Index
413
414     # Convenient representation of the mesh
415     V :: VectorList
416     E :: EdgeList
417
418     # Lookup tables
419     emap_pos :: IndexList
420     emap_neg :: IndexList
421     knnmap :: IndexMap
422     dual_knnmap :: IndexMap
423
424     #
```

```
    # Derived geometric quantities (normals, areas, lengths, cot of angles)
    #

    Ne :: VectorList
    Te :: VectorList
    Le :: ScalarList
    LSe :: ScalarList

    Nv :: VectorList
    Nvs :: VectorList
    Lv :: ScalarList

    # Forces, used for plotting
    a :: VectorList
    fp :: VectorList
    ff :: VectorList

    #
    # Dynamic parameters
    #
    l :: Scalar           # Length of domain
    h :: Union{Scalar, ScalarList}           # Smoothing length
    η :: Scalar           # Parameter used in computation of ρ_i and h_i
    α :: Scalar           # Correction factor kernel
    rho :: Scalar         # Density
    δ :: Scalar           # thickness

    R :: ScalarList       # Vertex mass density
    R0 :: ScalarList      # Inital vertex mass density
    M :: ScalarList       # Vertex mass
    U :: VectorList       # Vertex displacement
    Vold :: VectorList    # Vertex old position
    Vnew :: VectorList    # Vertex new position
    V0 :: VectorList      # Vertex initial position

    kernelgrad_hv :: VectorMap   # Gradients of the kernel function, with h(v)
    kernelgrad_hov :: VectorMap  # Gradients of the kernel function, with h(ov)
    kernelvals :: ScalarMap      # All relevant values of the kernel function
    ω :: ScalarList              # Dissipative term for stability

    P :: MatList          # First P-K stress tensor for each vertex
    CM :: MatList         # Gradient correction matrix

    Fdef :: MMatList      # Deformation gradient for each vertex
    eye :: Mat

    # Initialize all quantities
    function BlowSim{T, K}(
        nv :: Index,
        ngv :: Index,
        l :: Scalar,
        δ :: Scalar,
        material :: T,
        kernel :: K,
        shape :: String,
        ) where {T, K}

        self = new()

        # Find material and kernel parameters
        self.material = material
        self.kernel = kernel

        if shape in ["circle", "square", "ellipse", "squircle"]
            self.shapetype = "connected"
            self.distance = distance_connected
            self.ngv = 0
        elseif shape in ["line", "parabola", "circlearc"]
            self.shapetype = "unconnected"
            self.distance = distance_unconnected
            self.ngv = ngv
        else
            println("No such shape implemented.")
        end

        # Load mesh
        self.V = load_vertices(shape, nv, l; ngv = self.ngv)
        self.nv = length(self.V)

        self.E = load_edges(self.shapetype, self.nv)
        self.ne = length(self.E)

        self.l = l
```

```
508         self.rho = density(material)
509         self.h = smoothinglength(kernel, self.nv)
510         self.η = 1.3
511         self.α = kernel.α
512         self.δ = δ
513
514         # Indices of first and last moving particle. End points are fixed
515         # for the line, so do not have to loop over ghost vertices.
516         # For the circle, this reduces to v = 1:b.nv, since b.ngv = 0.
517         self.start_at = fld(self.ngv, 2) + 1
518         self.end_at = self.nv - cld(self.ngv, 2)
519
520         # Create look up table for edges
521         self.emap_pos, self.emap_neg = build_edge_maps(self.nv, self.E, self.
                shapetype)
522
523         # Pre-allocate space for derived gemoetrical information
524         self.Ne = VectorList(self.ne)  # edge normals
525         self.Te = VectorList(self.ne)  # edge tangentials
526         self.Le = ScalarList(self.ne)  # edge lengths
527         self.LSe = ScalarList(self.ne) # edge lengths cumulative sum
528
529         self.a = zeros(VectorList(self.nv))  # acceleration due to elastic
                forces
530         self.fp = zeros(VectorList(self.nv)) # pressure forces
531         self.ff = zeros(VectorList(self.nv)) # friction forces
532
533         self.Nv = VectorList(self.nv)  # vertex normals
534         self.Nvs = VectorList(self.nv) # smooth vertex normals
535         self.Lv = ScalarList(self.nv)  # vertex lengths
536
537         self.ω = ScalarList(self.nv)   # Dissipative term for stability
538
539         # Compute derived geometric variables
540         update_geometry!(self)
541
542         # Create neighbormap
543         self.knnmap = build_knn_map(self.h, self.V, self.nv, self.distance, self
                .LSe)
544
545         # Compute mass, initial velocity, and old and start position
546         self.M = self.Lv .* self.rho .* self.δ
547         self.U = zeros(Vec, self.nv)
548         self.Vold = deepcopy(self.V)
549         self.Vnew = deepcopy(self.V)
550         self.V0 = deepcopy(self.V)
551         self.kernelgrad_hv = [VectorList(length(self.knnmap[k])) for k = 1:self.
                nv]
552         self.kernelgrad_hov = [VectorList(length(self.knnmap[k])) for k = 1:self
                .nv]
553         self.kernelvals = [ScalarList(length(self.knnmap[k])) for k = 1:self.nv]
554
555         # Compute first Piola-Kirchhoff stress tensor, density and
556         # gradient correction matrix for each vertex
557         self.P = MatList(self.nv)
558         for v = 1:self.nv
559             self.P[v] = zeros(Mat)
560         end
561
562         self.CM = zeros(MatList(self.nv))     # Gradient correction matrices
563         self.Fdef = zeros(MMatList(self.nv))  # Deformation gradient
564         self.eye = eye(Mat)
565
566         self.R = ScalarList(self.nv)
567
568         update_densities!(self)
569         update_sph!(self)
570         return self
571     end
572 end
573
574 """ Updates the edge normals, areas, tangentials and lengths. """
575 function update_edges!(b :: BlowSim)
576     for (k, e) in enumerate(b.E)
577         v :: Vec = b.V[e[1]]
578         w :: Vec = b.V[e[2]]
579
580         b.Ne[k] = Vec(w[2] - v[2], v[1] - w[1])
581         if (b.shapetype == "unconnected") b.Ne[k] = -b.Ne[k] end
582         b.Ne[k] /= norm(b.Ne[k])
583
584         # Edge tangentials and lengths
585         b.Te[k] = v - w
```

```
        b.Le[k] = norm(b.Te[k])
        # Add 0.0 at the front, so that LSe[end] - LSe[1] = LSe[end]
        b.LSe = unshift!(cumsum(b.Le), 0.0)
    end
end

""" Updates vertex normals and lengths. """
function update_vertices!(b :: BlowSim)
    if b.shapetype == "unconnected"
        from = 2
        to = b.nv-1
        b.Lv[1] = 0.5 * b.Le[1]
        b.Lv[end] = 0.5 * b.Le[end]
        # Approximate the normal in the last vertices with the edge normals
        b.Nv[1] = b.Ne[1]
        b.Nv[end] = b.Ne[end]
    else
        from = 1
        to = b.nv
    end
    for k = from:to
        # adjacent edge index, clockwise direction for circle, left for line.
        p = b.emap_neg[k]
        b.Lv[k] = 0.5 * (b.Le[p] + b.Le[k])
        Ns = b.Le[p] * b.Ne[p] + b.Le[k] * b.Ne[k]
        b.Nv[k] = Ns / norm(Ns)
    end
end

""" Function which is zero for the optimal value of ρ """
function find_ρandh(b :: BlowSim, v :: Index, ρ :: Scalar)
    sumWij = 0.0
    for ov in b.knnmap[v]
        d = b.distance(b.LSe, v, ov)
        h = b.η * b.M[v] / ρ
        sumWij += b.M[ov] * b.kernel.func(h, b.α, d)
    end
    sumWij - ρ
end

"""
Updates the density of each vertex and the gradients of the kernel function
each vertex with its neighbors and the deformation gradient tensor.
"""
function update_densities!(b :: BlowSim{T,K}) where {T <: SpringMaterial, K <:
     Fixed}
    # Computation of densities and kernel gradient for each vertex
    for v = 1:b.nv
        b.R[v] = 0.0
        for (k, ov) in enumerate(b.knnmap[v])
            d :: Scalar = b.distance(b.LSe, v, ov)
            b.kernelvals[v][k] = b.kernel.func(b.h, b.α, d)
            b.R[v] += b.M[ov] * b.kernelvals[v][k]
            b.kernelgrad_hv[v][k] = b.kernel.gradfunc(b.h, b.α, b.V[v], b.V[ov],
                 d)
            b.kernelgrad_hov[v][k] = b.kernelgrad_hv[v][k]
        end
    end
end

function update_densities!(b :: BlowSim{T,K}) where {T <: SpringMaterial, K <:
     Adaptive}
    # Computation of densities and kernel gradient for each vertex
    for v = 1:b.nv
        # Need to have a function that only depends on 1 variable
        g(ρ) = find_ρandh(b, v, ρ)
        # Use previous value of b.R[v] as inital guess
        b.R[v] = find_zero(g, (ε, 3.0), Bisection())
        if b.R[v] ãĽđ ε
            error("Too small density at vertex: ", v)
        end

        # Update h[v]
        b.h[v] = b.η * b.M[v] / b.R[v]

        for (k, ov) in enumerate(b.knnmap[v])
            d :: Scalar = b.distance(b.LSe, v, ov)
            b.kernelvals[v][k] = b.kernel.func(b.h[v], b.α, d)
            b.kernelgrad_hv[v][k] = b.kernel.gradfunc(b.h[v], b.α, b.V[v], b.V[
                 ov], d)
            b.kernelgrad_hov[v][k] = b.kernel.gradfunc(b.h[ov], b.α, b.V[v], b.V
                 [ov], d)
        end
```

```
    end
end

""" Computes the deformation gradient tensor. """
function deformationgradient!(v :: Index, b :: BlowSim)
    eye!(b.Fdef[v])
    for (k, ov) in enumerate(b.knnmap[v])
        b.Fdef[v] += b.M[ov] / b.R[ov] * (kron(b.U[ov]', b.kernelgrad_hov[v][k])
            -
        kron(b.U[v]', b.kernelgrad_hv[v][k]))
    end
    if any(isnan, b.Fdef) error("NaN Fdef at vertex: $v") end
end

"""
Updates the first Piola-Kirchhoff stress tensor, smoothes the
normal vectors and computes the deformation gradient tensor. Also computes
correction matrices for all vertices (to ensure that the gradient
fulfills first order completenes).
"""
function update_sph!(b :: BlowSim)
    for v = 1:b.nv
        deformationgradient!(v, b)
        # First Piola-Kirchhoff stress tensor using Green Lagrange Strain
        b.P[v] = b.Fdef[v] * stress(b.material, b.Fdef[v])

        # Smoothing of the normal vector and computation of dissipative term
        n = Vec(0.0,0.0)
        b.ω[v] = 1.0
        for (k, ov) in enumerate(b.knnmap[v])
            # normal
            n += b.M[ov] / b.R[ov] * b.Nv[ov] * b.kernelvals[v][k]

            # dissipative term
            dhdr = - b.η * b.M[v] / b.R[v]^2
            dWdh = dW(b, v, ov, k)
            b.ω[v] -= b.M[ov] * dhdr * dWdh
        end
        b.Nvs[v] = n / norm(n)

        # Correction matrices
        # b.CM[v] = zeros(Mat)
        # for (k,ov) in enumerate(b.knnmap[v])
        #     b.CM[v] += b.M[ov] / b.R[ov] * (
        #     kron(b.kernelgrad_hov[v][k]', b.V0[ov]) -
        #     kron(b.kernelgrad_hv[v][k]', b.V0[v]) )
        # end
        # b.CM[v] = inv(b.CM[v])
        # # If b.CM[v] was singular, set correction matrix to identity matrix
        # if true || any(isnan, b.CM[v])
        #     b.CM[v] = eye(Mat)
        # end
        #
        # # Adjust kernel gradient
        # for i = 1:length(b.knnmap[v])
        #     b.kernelgrad_hv[v][i] = b.CM[v] * b.kernelgrad_hv[v][i]
        #     b.kernelgrad_hov[v][i] = b.CM[v] * b.kernelgrad_hov[v][i]
        # end
    end
end

function dW(b :: BlowSim{T,K}, v :: Index, ov :: Index, k :: Index
    ) where {T <: SpringMaterial, K <: Fixed}
    d = b.distance(b.LSe, v, ov)
    s = d / b.h
    if s > 2. || s < ε
        dwdq = 0.
    else
        dwdq = -2.5 * (1. - .5 * s)^4 * (1. + 2.5 * s + 2. * s^2) +
            (1. - .5 * s)^5 * (2.5 + 2. * s)
    end
    return -b.kernel.α / b.h * (b.kernelvals[v][k] + dwdq / d)
end

function dW(b :: BlowSim{T,K}, v :: Index, ov :: Index, k :: Index
    ) where {T <: SpringMaterial, K <: Adaptive}
    d = b.distance(b.LSe, v, ov)
    s = d / b.h[v]
    if s > 2. || s < ε
        dwdq = 0.
    else
        dwdq = -2.5 * (1. - .5 * s)^4 * (1. + 2.5 * s + 2. * s^2) +
            (1. - .5 * s)^5 * (2.5 + 2. * s)
```

```
746         end
747         return -b.kernel.α / b.h[v] * (b.kernelvals[v][k] + dwdq / d)
748     end
749
750     """ Collects update functions regarding to geometrical quantities. """
751     function update_geometry!(b :: BlowSim)
752         update_edges!(b)
753         update_vertices!(b)
754     end
755
756     """ Collects all update functions, geometrical and SPH-related. """
757     function update_all!(b :: BlowSim)
758         # Order is important! E.g. update_vertices! before update_densities!, so
759         # that the current (and not previous) normals are used to smooth.
760         update_edges!(b)
761         update_vertices!(b)
762         update_densities!(b)
763         update_sph!(b)
764     end
765
766     """
767     Executes a Stormer-Verlet integration step, given a BlowSim, pressure,
768     time step, and friction coefficient.
769     """
770     function inflate!(b :: BlowSim, pressure :: Scalar, dt :: Scalar, mu :: Scalar)
771         # Update normals etc
772         update_all!(b)
773
774         # Time step for comparison
775         dt_f = 48. * dt^2
776
777         for v = b.start_at:b.end_at
778             # Acceleration using equation of motion, due to elastic forces.
779             b.a[v] = Vec(0.0, 0.0)
780             b.fp[v] = Vec(0.0, 0.0)
781             b.ff[v] = Vec(0.0, 0.0)
782
783             for (k, ov) in enumerate(b.knnmap[v])
784                 # Stress contribution
785                 b.a[v] += (b.P[v] / (b.ω[v] * b.R[v]^2) * b.kernelgrad_hv[v][k] +
786                 b.P[ov] / (b.ω[v] * b.R[ov]^2) * b.kernelgrad_hov[v][k]) * b.M[ov]
787             end
788
789             # Pressure contribution
790             b.fp[v] = pressure * b.Lv[v] * b.Nvs[v]
791
792             current_cfl = cfl(b, v)
793             if current_cfl < dt_f
794                 dt_f = current_cfl
795             end
796         end
797
798         # adjust timestep, and compute dt squared and inverse
799         dt = .25 / sqrt(3.) * sqrt(dt_f)
800         dt2 = dt * dt
801         dtinv = 1.0 / dt
802
803         # Compute everything needing dt and execute verlet update
804         for v = b.start_at:b.end_at
805             # Friction contribution
806             b.ff[v] = -mu * (b.V[v] - b.Vold[v]) * dtinv
807
808             a = (b.fp[v] + b.ff[v]) / b.M[v]
809
810             # Verlet update
811             b.Vnew[v] = 2.0 * b.V[v] - b.Vold[v] + (b.a[v] + a) * dt2
812         end
813
814         # Swap V and Vold
815         b.Vold = deepcopy(b.V)
816         b.V = deepcopy(b.Vnew)
817
818         # Displacement update, used in update_stress!
819         b.U = b.V - b.V0
820         dt
821     end
822
823     function cfl(b :: BlowSim{T,K}, v :: Index) where {T <: SpringMaterial, K <:
          Fixed}
824         (b.M[v] * b.h) / norm(b.a[v] * b.M[v] + b.fp[v])
825     end
826     function cfl(b :: BlowSim{T,K}, v :: Index) where {T <: SpringMaterial, K <:
          Adaptive}
```

```
    (b.M[v] * b.h[v]) / norm(b.a[v] * b.M[v] + b.fp[v])
end


""" Function to plot the inflated shape. """
function showplot(V :: VectorList, fs :: VectorList, fp :: VectorList,
                  ff ::VectorList,
                  nv :: Index, s :: Scalar, l :: Scalar)
    X = [v[1] for v in V]
    Y = [v[2] for v in V]

    # Also plot pressure, friction and elastic forces.
    Fs = Array{Float64,2}(nv, 2)
    Fp = Array{Float64,2}(nv, 2)
    Ff = Array{Float64,2}(nv, 2)
    for v = 1:nv
        for j = 1:2
            Fs[v,j] = fs[v][j]
            Fp[v,j] = fp[v][j]
            Ff[v,j] = ff[v][j]
       end
   end
   p = plot(X, Y, shape = :circle,
       # Options if you only want markers, without borders and no lines
       # linewidth = 0,
       # markersize = 2,
       # markerstrokealpha = 0
       label = "Vertex locations, at $s s.",
       legend = :bottomright,
       # xlims = (-1.5, 1.5),
       # ylims = (-1.5, 1.5)
       )
   quiver!(X, Y, quiver = (Fs[:,1], Fs[:,2]), color = :blue)
   quiver!(X, Y, quiver = (Ff[:,1], Ff[:,2]), color = :green)
   quiver!(X, Y, quiver = (Fp[:,1], Fp[:,2]), color = :red,
                  aspect_ratio = :equal)
    annotate!([(-2,-1.5,text("Pressure forces", :left, :red, 8)),
               (-2,-1.7,text("Elastic forces", :left, :blue, 8)),
               (-2,-1.9,text("Friction forces", :left, :green, 8))])

    display(p)
end

function showplot(V :: VectorList, VM :: VectorMap,
                  nv :: Index, s :: Scalar, l :: Scalar)
    X = [v[1] for v in V]
    Y = [v[2] for v in V]

    # Also plot pressure, friction and elastic forces.
    n = length(VM)
    F = [Array{Float64, 2}(nv, 2) for i = 1:n]

    for (i, VL) in enumerate(VM)
        for v = 1:nv
            for j = 1:2
                F[i][v,j] = VL[v][j]
            end
        end
    end

    p = plot(X, Y, shape = :circle,
       # Options if you only want markers, without borders and no lines
       # linewidth = 0,
       # markersize = 2,
       # markerstrokealpha = 0
       label = "Vertex locations, at $s s.",
       aspect_ratio = :equal,
       legend = :bottomleft,
       # xlims = (-1.5, 1.5),
       # ylims = (-1.5, 1.5)
       )

    for i = 1:n
        quiver!(X, Y, quiver = (F[i][:,1], F[i][:,2]))
    end

    tracked = Int[1, 20, 50, 75]
    xs = [V[i][1] for i in tracked]
    ys = [V[i][2] for i in tracked]
    scatter!(xs, ys, label = "Tracked vertices", c = :orange, markersize = 6)

    display(p)
end
```

```
910
911 end # module SPH2D
```

B.2 3D Code

Listing 5 *Demo file using BlowLogicSPH.jl*

```
1  println("@@@ STARTING")
2
3  include("BlowLogicSPH.jl")
4
5  # push!(LOAD_PATH, "./data/")
6  using BlowLogicSPH
7  sph = BlowLogicSPH # alias
8  using FileIO
9
10 #--- Initialization
11 E = 1.e6
12 nu = .48
13 λ = E*nu / (1+nu) / (1-2*nu)
14 μ =  E / (2 - 2*nu)
15 ρ = 1.1
16 δ = 2.54e-4
17
18 steel = sph.LinearElastic(ρ, δ*λ, δ*μ)
19
20 h = .2
21 kernel = "wendland"
22 fixedlength = sph.Fixed(h, kernel)
23
24 # via .txt file
25 filename = "sphere"
26 zerobased = filename == "cube"
27 b = sph.BlowSim{sph.LinearElastic, sph.Fixed}(
28     "./data/"*filename, steel, δ, fixedlength, zerobased = zerobased);
29 # or .obj
30 sphere = load("./data/sphere.obj")
31 R0 = 1. # sphere has radius 1, but we can adjust it
32 V = [Float64(vertex[j])*R0 for vertex in sphere.vertices, j = 1:3]
33 F = [Int(face[j]) for face in sphere.faces, j =1:3]
34 s = sph.BlowSim{sph.LinearElastic, sph.Fixed}(V, F,steel, δ, fixedlength)
35
36 #--- Actual inflation
37 mesh = deepcopy(s); # copying is much faster than making it again
38
39 timestep = 0.001
40 mu = 0.
41 nsteps = 10
42 pmax = 10.
43 maxsteps = 300
44 totaltime = 0.
45 dt = 0.
46 totalsteps = 0
47 T = maxsteps * timestep
48
49 v = Int[1, 20, 50, 75]
50 Fp = [Float64[] for i = 1:length(v)]
51 Fs = [Float64[] for i = 1:length(v)]
52 Ff = [Float64[] for i = 1:length(v)]
53 Le = [Float64[] for i = 1:length(v)]
54 R = [Float64[] for i = 1:length(v)]
55 P = Float64[]
56 ds = Float64[]
57 Le0 = deepcopy(mesh.Le)
58
59 println("---$nsteps steps, maximum pressure: $pmax, timestep: $timestep, mu: $mu
       ")
60 @time for k = 1:nsteps
61     if k % 1 == 0; println("At iteration $k.") end
62
63     pressure = pmax^2 * min(1, sum(ds) / T)
64     push!(P, pressure)
65
66     dt = sph.inflate!(mesh, sqrt(pressure), timestep, mu)
67     push!(ds, dt)
68
69     for i = 1:length(v)
70         push!(Fp[i], norm(mesh.fp[v[i]]))
```

```
        push!(Fs[i], norm(mesh.a[v[i]] * mesh.M[v[i]]))
        push!(Ff[i], norm(mesh.ff[v[i]]))
        push!(Le[i], mesh.Le[v[i]])
        push!(R[i], norm(mesh.V[v[i]]))
    end

    if any(isnan, mesh.V)
        println("NaN found, quit loop at iteration $m.")
        break
    end

    radius = sum([R[i][end] for i = 1:length(v)]) / length(v)
    fpress = sum([Fp[i][end] for i = 1:length(v)]) / length(v)
    fel = sum([Fs[i][end] for i = 1:length(v)]) / length(v)
    ffric = sum([Ff[i][end] for i = 1:length(v)]) / length(v)
    println("Average radius: $radius, Fp: $fpress, Fs: $fel],
        Fm: $ffric, dt: $dt")
    totalsteps += 1
end

#--- Plotting

# mesh
sph.showplot(mesh.V, mesh.F)

# forces
using Plots
# Plot forces
c = palette(:default)[1:length(v)]

times = cumsum(ds)
p1 = plot(title = "Force magnitude during inflation", xlabel = "Time",
    ylabel = "Force magnitude", legend = :topleft, ylims = (0.,.2))
n1 = 0
n2 = 1
n3 = length(times)
for i = 1:length(v)
    plot!(times[n2:end-n1], Fp[i][n2:n3-n1], label = "Pressure force, $i",
    line = :dash, c = c[i])
    plot!(times[n2:end-n1], Fs[i][n2:n3-n1], label = "Spring force, $i",
    line =:solid, c = c[i])
    plot!(times[n2:end-n1], Ff[i][n2:n3-n1], label = "Friction force, $i",
    line =:dot, c = c[i])
end
display(p1)

p2 = plot(times, sqrt.(P)[1:n3], xlabel = "time", ylabel ="pressure", legend =
     false)

p3 = plot(xlabel = "Time", ylabel = "Length", legend = :bottomleft, ylims = (0.,
     0.15))
for i=1:length(v)
    plot!(times[n2:end-n1], Le[i][n2:n3-n1], label = "Vertex length, $i",
    line = :solid, c = c[i])
end
display(p3)

p4 = plot(xlabel = "Time", ylabel = "Radius", legend = :bottomleft, ylims = (0.,
     1.6))
for i=1:length(v)
    plot!(times[n2:end-n1], R[i][n2:n3-n1], label = "Radius, $i",
    line = :solid, c = c[i])
end
display(p4)

# savefig(p1, "forces_bol_p100_ms300_2.pdf")
# savefig(p2, "pressure_bol_p100_ms300_2.pdf")
# savefig(p3, "vertexlength_bol_p100_ms300_2.pdf")
# savefig(p4, "radius_bol_p100_ms300_2.pdf")



# Show original and current mesh using Makie # Does not work currently

# if any(isnan, b.V)
#     println("NaN found")
# else
#     sph.showtwomeshes(b.V, b.F, b.nv, b.nf, original.V, original.F,
#                                                 original.nv, original.
    nf)
# end

# Show only current mesh using Makie
```

```
# if any(isnan, b.V)
#     println("NaN found")
# else
#     sph.showmesh(b.V, b.F, b.nv, b.nf)
# end
```

Listing 6 *Main SPH code.*

```
module BlowLogicSPH

using StaticArrays
using PyPlot
using PyCall

@pyimport pyigl as igl
@pyimport numpy as np
# using GeometryTypes, Makie, Colors

const Scalar = Float64
const Index = Int

const Vec3 = SVector{3, Scalar}
const Vec2 = SVector{2, Scalar}
const Edge = SVector{2, Index}
const Face = SVector{3, Index}

const SparMat = SparseMatrixCSC{Scalar, Int64}
const Mat = SMatrix{3,3,Float64,9}
const MMat = MMatrix{3,3,Float64,9}
const MatList = Array{Mat, 1}

const ScalarList = Array{Scalar,1}
const Vector2List = Array{Vec2,1}
const VectorList = Array{Vec3,1}
const EdgeList = Array{Edge,1}
const FaceList = Array{Face,1}
const VectorMap = Array{VectorList,1}

const IndexList = Array{Index,1}
const IndexMap = Array{IndexList,1}
#import Base.isless

""" Conversion from Eigen to Julia matrices. """
function e2j(V :: PyCall.PyObject)
    if pyrepr(pytypeof(V)) == "<class 'pyigl.eigen.MatrixXd'>"
        # Convert to Python array, to Julia is done automatically by PyCall
        t = np.array(V, dtype = "float64")
        return t
    elseif pyrepr(pytypeof(V)) == "<class 'pyigl.eigen.MatrixXi'>"
        # Convert to Python array, to Julia is done automatically by PyCall
        t = np.array(V, dtype = "int64") + 1
        return t
    else
        error("Numerical type not supported.")
    end
end

""" Conversion from Julia to Eigen matrices. """
j2e(V :: Array{Float64, 2}) = igl.eigen[:MatrixXd](V)
function j2e(F :: Array{Int64, 2})
    F = convert(Array{Int32, 2}, F) - Int32(1)
    igl.eigen[:MatrixXi](F)
end
j2e(V :: VectorList) = igl.eigen[:MatrixXd](arr2mat(V))
j2e(F :: FaceList) = j2e(arr2mat(F))

""" Conversion between Array of StaticArrays and Matrix. """
arr2mat(V :: Union{FaceList, VectorList}) =  [v[i] for v in V, i = 1:3]
function mat2vec(V :: Matrix)
    if size(V, 2) == 3
        return [Vec3(V[i,:]) for i in 1:size(V,1)]
    elseif size(V, 2) == 2
        return [Vec2(V[i,:]) for i in 1:size(V,1)]
    else
        error("No such vector defined.")
    end
end
mat2face(F :: Matrix) = [Face(F[i,:]) for i in 1:size(F,1)]

```

```
""" Load vertices from a file. """
function load_vertices(filename :: String)
    # Read as matrix
    Vmat = readdlm(filename)
    # Convert to VectorList
    [Vec3(Vmat[i,:]) for i in 1:size(Vmat,1)]
end

""" Load faces from a file. Assumes faces are given zero indexed. """
function load_faces(filename :: String; zerobased :: Bool = true)
    # Read as matrix and convert to 1 based numbering if needed
    Fmat = readdlm(filename) + 1 * zerobased
    # Convert to FaceList
    [Face(Fmat[i,:]) for i in 1:size(Fmat,1)]
end

""" Orders an edge, such that the smallest element comes first. """
ordered_edge(v1 :: Index, v2 :: Index) = Edge(minmax(v1, v2)...)

"""
Extract all edges to a list from a list of Faces. Returns a list of Edges.
"""
function extract_edges(F :: FaceList)
    E = EdgeList()
    for f in F
        push!(E, ordered_edge(f[1], f[2]))
        push!(E, ordered_edge(f[2], f[3]))
        push!(E, ordered_edge(f[3], f[1]))
    end
    # Return without duplicates
    unique(sort(E, lt=edge_lt), 1)
end

""" Lexical ordering of edges. """
edge_lt(u :: Edge, w :: Edge) = u[1] < w[1] || (u[1] == w[1] && u[2] < w[2])

"""
Compute the edge maps: for each vertex, finds its adjacent edges.
"""
function build_edge_map(nv :: Int, E :: EdgeList)
    emap = [IndexList() for k in 1:nv]

    for (k,e) in enumerate(E)
        push!(emap[e[1]], k)
        push!(emap[e[2]], k)
    end

    emap
end

"""
Create an array of arrays containing at index v the indices of the Faces
adjacent to v.
"""
function build_face_map(nv :: Index, F :: FaceList)
    fmap = [IndexList() for k in 1:nv]

    for (k, f) in enumerate(F)
        push!(fmap[f[1]], k)
        push!(fmap[f[2]], k)
        push!(fmap[f[3]], k)
    end
    fmap
end

# Abstract type to hold material parameters
abstract type SpringMaterial
end

struct LinearElastic <: SpringMaterial
    ρ :: Scalar
    λ :: Scalar
    μ :: Scalar
end

density(material :: LinearElastic) = material.ρ
lambda(material :: LinearElastic) = material.λ
mu(material :: LinearElastic) = material.μ

""" Computes the second Piola-Kirchhoff stress tensor. """
function stress(material :: LinearElastic, E :: Mat)
    material.λ * trace(E) * eye(Mat) + 2.0 * material.μ * E
end
```

```
155
156 # Abstract type to hold parameters for the kernel function
157 abstract type Kernel
158 end
159
160 mutable struct Fixed <: Kernel
161     h :: Scalar  # smoothing length
162     α :: Scalar  # dimensional scaling factor
163     func :: Function # kernel function
164     gradfunc :: Function # gradient of kernel function
165
166     function Fixed(h :: Scalar, name :: String)
167         if name == "cubicBspline"
168             self = new(h, 1., cubicBspline, grad_cubicBspline)
169         elseif name == "wendland"
170             self = new(h, 1., wendland, grad_wendland)
171         else
172             error("no such kernel function")
173         end
174     end
175 end
176
177 smoothinglength(fixed :: Fixed) = fixed.h
178 kernelcorrection(kernel :: Kernel) = kernel.α
179
180 """ Cubic Bspline kernel function. """
181 function cubicBspline(h :: Scalar, α :: Scalar, dist :: Scalar)
182     q = dist / h
183     if q >= 2.
184         return 0.
185     elseif 0. <= q < 1.
186         return α / h^2 * ( 2. / 3. - q^2 + .5*q^3)
187     else
188         return α / h^2 * (2. - q)^3 / 6.
189     end
190 end
191
192 """ Gradient of cubic B spline kernel function. """
193 function grad_cubicBspline(h :: Scalar, α :: Scalar, si :: Vec2, sj :: Vec2,
194         xi :: Vec3, xj :: Vec3, dist :: Scalar)
195     u = si - sj
196     q = dist / h
197     if q >= 2.0 || q âĹď 1e-10
198         return Vec3(0.0, 0.0, 0.0)
199     elseif 1e-10 < q < 1.0
200         âĹĆxâĹĆs = repmat(xi - xj, 1, 2)
201         âĹĆxâĹĆs[:,1] /= u[1]
202         âĹĆxâĹĆs[:,2] /= u[2]
203         âĹĆsâĹĆx = pinv(âĹĆxâĹĆs)
204         return α / (h^3 * norm(u)) * u' * âĹĆsâĹĆx * (-2.0 + 1.5 * q)*q
205     else 1.0 <= q < 2.0
206         âĹĆxâĹĆs = repmat(xi - xj, 1, 2)
207         âĹĆxâĹĆs[:,1] /= u[1]
208         âĹĆxâĹĆs[:,2] /= u[2]
209         âĹĆsâĹĆx = pinv(âĹĆxâĹĆs)
210         return α / (h^3 * norm(u)) * u' * âĹĆsâĹĆx * -(2.0 - q)^2 * 0.5
211     end
212 end
213
214 """ Wendland kernel """
215 function wendland(h :: Scalar, α :: Scalar, dist :: Scalar)
216     q = dist / h
217     if q >= 2.
218         return 0.
219     else
220         return α * (1. - .5 * q)^5 * (1. + 2.5 * q + 2. * q^2)
221     end
222 end
223
224 """ Gradient of the wendland kernel """
225 function grad_wendland(h :: Scalar, α :: Scalar, si :: Vec2, sj :: Vec2,
226         xi :: Vec3, xj :: Vec3, dist :: Scalar)
227     u = si - sj
228     q = dist / h
229     if q >= 2.0 || q < 1e-10
230         return Vec3(0.0, 0.0, 0.0)
231     else
232         âĹĆxâĹĆs = repmat(xi - xj, 1, 2)
233         âĹĆxâĹĆs[:,1] /= u[1]
234         âĹĆxâĹĆs[:,2] /= u[2]
235         âĹĆsâĹĆx = pinv(âĹĆxâĹĆs)
236         âĹĆwâĹĆq = - 7. * q * (1. - .5 * q)^4 * (.5 + q)
237         return α / (h^3 * norm(u)) * dist * u' * âĹĆsâĹĆx * âĹĆwâĹĆq
```

```
    end
end

""" Takes a matrix and changes its entries such that it is an idenity matrix """
function eye!(A::MMat)
    for j = 1:3
        for i = 1:3
            @inbounds A[i,j] = 1.0(i == j)
        end
    end
end

"""
Struct containing all relevant information of a mesh during and after inflation,
e.g., material parameters, vertex locations, number of vertices, density, etc.
Also contains the function to initialize the struct.
"""
mutable struct BlowSim{T<:SpringMaterial, K<:Kernel}

    # Kernel and material parameters
    kernel :: K
    material :: T

    # Number of vertices, faces and edges
    nv :: Index
    nf :: Index
    ne :: Index

    # Convenient representation of the mesh
    V :: VectorList
    F :: FaceList
    E :: EdgeList

    # Lookup tables
    emap :: IndexMap
    fmap :: IndexMap
    F_subinds :: IndexMap
    V_subinds :: IndexMap

    #
    # Derived geometric quantities (normals, areas, lengths, cot of angles)
    #

    Nf :: VectorList
    Af :: ScalarList

    Te :: VectorList
    Le :: ScalarList

    Nv :: VectorList
    Av :: ScalarList

    # Forces, used for plotting
    a :: VectorList
    fp :: VectorList
    ff :: VectorList

    #
    # Dynamic parameters
    #

    h :: Scalar           # Smoothing length
    α :: Scalar           # Correction factor kernel
    ρ :: Scalar           # Density
    δ :: Scalar           # Thickness
    λ :: Scalar           # First Lame parameter
    μ :: Scalar           # Second Lame parameter

    R :: ScalarList       # Vertex density
    M :: ScalarList       # Vertex mass
    U :: VectorList       # Vertex displacement
    Vold :: VectorList    # Vertex old position
    Vnew :: VectorList    # Vertex new position
    V0 :: VectorList      # Vertex initial position

    kernelgrad :: VectorMap        # All gradients of the kernel function

    P :: MatList          # First P-K stress tensor for each vertex
    Fdef :: MMat          # Deformation gradient of a point

    eye :: Mat

    #
```

```
    # Necessary information needed for geodesic distance
    #

    A :: SparMat
    An :: SparMat
    cots :: Matrix{Scalar}
    nLC :: SparMat

    m :: Scalar
    lfac :: SparMat
    fact_nLC :: Base.SparseArrays.CHOLMOD.Factor{Float64}

    gradu :: VectorList
    divx :: ScalarList

    # Initialize all quantities
    function BlowSim{T, K}(
        V :: VectorList,
        F :: FaceList,
        material :: T,
        δ :: Scalar,
        kernel :: K,
        ) where {T, K}

        self = new()

        # Find material and kernel parameters
        self.material = material
        self.kernel = kernel

        self.ρ = density(material)
        self.δ = δ
        self.μ = mu(material)
        self.λ = lambda(material)
        self.h = smoothinglength(kernel)
        self.α = kernelcorrection(kernel)

        # Load mesh
        self.V = V
        self.nv = length(self.V)

        self.F = F
        self.nf = length(self.F)

        self.E = extract_edges(self.F)
        self.ne = length(self.E)

        # Create look up tables
        self.emap = build_edge_map(self.nv, self.E)
        self.fmap = build_face_map(self.nv, self.F)

        # Pre-allocate space for derived gemoetrical information
        self.Nf = VectorList(self.nf) # face normals
        self.Af = ScalarList(self.nf) # Face areas

        self.Te = VectorList(self.ne) # edge tangentials
        self.Le = ScalarList(self.ne) # edge lengths

        self.Nv = VectorList(self.nv) # vertex normals
        self.Av = ScalarList(self.nv) # vertex areas

        self.a = zeros(VectorList(self.nv))   # acceleration due to elastic
             forces
        self.fp = zeros(VectorList(self.nv))  # pressure forces
        self.ff = zeros(VectorList(self.nv))  # friction forces

        # Pre-allocate space for geodesic distance

        self.A = spzeros(self.nv, self.nv)    # cotangents of angles
        self.An = spzeros(self.nv, self.nv)   # mass diagonal materix
        self.cots = zeros(self.nf, 3)         # cotangents of angles per face
        self.nLC = spzeros(self.nv, self.nv)  # negative laplace opterator
        self.lfac = spzeros(self.nv, self.nv) # Backward Euler matrix
        self.gradu = VectorList(self.nf)      # gradient of heat u
        self.divx = ScalarList(self.nf)       # integrated divergence of u
        self.m = 1.

        # Compute derived geometric variables
        update_geometry!(self)
        update_edges!(self)
        update_cotangents!(self)
        laplaceoperator!(self)
        nLCfactor!(self, self.m)
```

```
403
404         self.V_subinds, self.F_subinds = build_submeshmaps(self)
405
406         # Compute mass, initial velocity, and old and start position
407         self.M = self.Av .* self.ρ .* self.δ
408         self.U = zeros(Vec3, self.nv)
409         self.Vold = deepcopy(self.V)
410         self.Vnew = deepcopy(self.V)
411         self.V0 = deepcopy(self.V)
412         self.kernelgrad = [VectorList(length(self.V_subinds[k])) for k = 1:self.
                nv]
413
414         # Compute first Piola-Kirchhoff stress tensor, density and
415         # gradient correction matrix for each vertex
416         self.P = MatList(self.nv)
417         for v = 1:self.nv
418             self.P[v] = zeros(Mat)
419         end
420
421         self.R = ScalarList(self.nv)
422         update_density_grads!(self)
423
424         self.Fdef = eye(MMat) # Deformation gradient
425         self.eye = eye(Mat)
426
427         update_sph!(self)
428
429         return self
430     end
431
432     function BlowSim{T, K}(
433         filename :: String,
434         material :: T,
435         δ :: Scalar,
436         kernel :: K;
437         zerobased :: Bool = true,
438         ) where {T, K}
439
440         # Load mesh
441         V = load_vertices(filename * "-V.txt")
442         F = load_faces(filename * "-F.txt", zerobased = zerobased)
443
444         return BlowSim{T, K}(V, F, material, δ, kernel)
445     end
446
447     function BlowSim{T, K}(
448         Vmat :: Matrix{Scalar},
449         Fmat :: Matrix{Index},
450         material :: T,
451         δ :: Scalar,
452         kernel :: K
453         ) where {T, K}
454
455         0 âĹĹ Fmat ? zerobased = true : zerobased = false
456         # Convert to Vector- and FaceList
457         V = [Vec3(Vmat[i,:]) for i = 1:size(Vmat, 1)]
458         F = [Face(Fmat[i,:]) for i = 1:size(Fmat, 1)] + 1 * zerobased
459
460         return BlowSim{T, K}(V, F, material, δ, kernel)
461     end
462
463     # Initialize only geometric quantities
464     function BlowSim{T, K}(V :: VectorList, F :: FaceList, k :: Index) where {T,
            K}
465         self = new()
466
467         # Load mesh
468         self.V = V
469         self.nv = length(self.V)
470
471         self.F = F
472         self.nf = length(self.F)
473
474         self.E = extract_edges(self.F)
475         self.ne = length(self.E)
476
477         # Create look up tables
478         self.fmap = build_face_map(self.nv, self.F)
479
480         # Pre-allocate space for derived gemoetrical information
481         self.Nf = VectorList(self.nf) # face normals
482         self.Af = ScalarList(self.nf) # Face areas
483
```

```
        self.Te = VectorList(self.ne) # edge tangentials
        self.Le = ScalarList(self.ne) # edge lengths

        self.Nv = VectorList(self.nv) # vertex normals
        self.Av = ScalarList(self.nv) # vertex areas

        self.a = zeros(VectorList(self.nv))   # acceleration due to elastic
             forces
        self.fp = zeros(VectorList(self.nv))  # pressure forces
        self.ff = zeros(VectorList(self.nv))  # friction forces

        # Pre-allocate space for geodesic distance

        self.A = spzeros(self.nv, self.nv)    # cotangents of angles
        self.An = spzeros(self.nv, self.nv)   # mass diagonal materix
        self.cots = zeros(self.nf, 3)         # cotangents of angles per face
        self.nLC = spzeros(self.nv, self.nv)  # negative laplace opterator
        self.lfac = spzeros(self.nv, self.nv) # Backward Euler matrix
        self.gradu = VectorList(self.nf)      # gradient of heat u
        self.divx = ScalarList(self.nf)       # integrated divergence of u
        self.m = 0.0

        # Compute derived geometric variables
        update_geometry!(self)

        # Compute mass, initial velocity, and old and start position
        self.M = self.Av .* self.ρ
        self.U = zeros(Vec3, self.nv)
        self.Vold = deepcopy(self.V)
        self.Vnew = deepcopy(self.V)
        self.V0 = deepcopy(self.V)

        self.eye = eye(Mat)

        return self
    end

    function BlowSim{T, K}(
        filename :: String,
        k :: Index;
        zerobased :: Bool = true) where {T, K}
        # Load mesh
        V = load_vertices(filename * "-V.txt")
        F = load_faces(filename * "-F.txt", zerobased = zerobased)

        return BlowSim{T,K}(V, F, k)
    end

    function BlowSim{T, K}(
        Vmat :: Matrix{Scalar},
        Fmat :: Matrix{Index},
        k :: Index;
        zerobased :: Bool = true) where {T, K}
        # Convert to Vector- and FaceList
        V = [Vec3(Vmat[i,:]) for i = 1:size(Vmat, 1)]
        F = [Face(Fmat[i,:]) for i = 1:size(Fmat, 1)] + 1 * zerobased

        return BlowSim{T,K}(V, F, k)
    end
end

"""
Given an index, finds surrounding faces and vertices in a radius of 2.5*h.
Returns indices of these vertices and faces.
"""
function submesh(b :: BlowSim, v :: Index; update :: Bool = false)
    verts = IndexList()

    # Find vertices closer than 2.5*h to v
    dists = geodesicdistance(b, [v], update = update)
    for ov = 1:b.nv
        if dists[ov] â‰¤ 2.5*b.h
            push!(verts, ov)
        end
    end

    # Find faces the verts are in
    faces = unique(vcat(b.fmap[verts]...))
    # Find vertices not in verts yet, so that all faces are complete.
    verts = unique([f[i] for f in b.F[faces], i = 1:3])

    verts, faces
end
```

```
566
567 function build_submeshmaps(b :: BlowSim)
568     V_subinds =  [IndexList() for k in 1:b.nv]
569     F_subinds =  [IndexList() for k in 1:b.nv]
570     for i = 1:b.nv
571         V_subinds[i], F_subinds[i] = submesh(b, i)
572     end
573     V_subinds, F_subinds
574 end
575
576 """ Collects some update functions. """
577 function update_geometry!(b :: BlowSim)
578     update_faces!(b)
579     update_vertices!(b)
580 end
581
582 """ Collects all update functions. """
583 function update_all!(b :: BlowSim)
584     update_density_grads!(b)
585     update_faces!(b)
586     update_vertices!(b)
587     update_sph!(b)
588 end
589
590 """ Updates the face normals and areas. """
591 function update_faces!(b :: BlowSim)
592     for (k, f) in enumerate(b.F)
593         v :: Vec3 = b.V[f[2]] - b.V[f[1]]
594         w :: Vec3 = b.V[f[3]] - b.V[f[1]]
595
596         b.Nf[k] = cross(v, w)
597
598         mag = norm(b.Nf[k])
599         b.Nf[k] /= mag
600         b.Af[k] = 0.5 * mag
601     end
602 end
603
604 """ Updates vertex normals and areas. """
605 function update_vertices!(b :: BlowSim)
606     for k = 1:b.nv
607         Ns = Vec3(0.0, 0.0, 0.0)
608         As = 0.0
609
610         for f in b.fmap[k]
611             As += b.Af[f]
612             Ns += b.Af[f] * b.Nf[f]
613         end
614
615         b.Av[k] = As / 3.0
616         b.Nv[k] = Ns / norm(Ns)
617     end
618 end
619
620 """
621 Updates the density of each vertex and the gradients of each vertex with
622 its neighbors.
623 """
624 function update_density_grads!(b :: BlowSim)
625     d = geodesicdistance(b, [1], update=true)
626     for v = 1:b.nv
627         # Compute parametrization
628         V_uv, F_uv = parametrization(b.V[b.V_subinds[v]], b.F[b.F_subinds[v]])
629         i = find(x-> x == v, b.V_subinds[v])[1]
630         b.R[v] = 0.0
631
632         # Compute density and gradient of the kernel function
633         for (k, ov) in enumerate(b.V_subinds[v])
634             # conformal factor of ov
635             λov = conformalfactor(b, V_uv, ov, k, b.V_subinds[v])
636             b.R[v] += b.M[ov] / λov^2 * b.kernel.func(b.h, b.α, d[ov])
637             if v == ov
638                 b.kernelgrad[v][k] = Vec3(0., 0., 0.)
639                 continue
640             end
641             if any(iszero, V_uv[i] - V_uv[k])
642                 error("si = sj, divide by zero coming up.")
643             end
644             b.kernelgrad[v][k] = b.kernel.gradfunc(b.h, b.α, V_uv[i], V_uv[k],
645                                                    b.V[v], b.V[ov], d[ov])
646         end
647         if v âĽă b.nv
648             d = geodesicdistance(b, [v+1], update=false)
```

```
        end
    end
end

""" Computes the deformation gradient tensor. """
function deformationgradient!(v :: Index, b :: BlowSim)
    eye!(b.Fdef)
    for (k, ov) in enumerate(b.V_subinds[v])
        b.Fdef += b.M[ov] / b.R[ov] * kron((b.U[ov] - b.U[v])',
            b.kernelgrad[v][k])
    end
    if any(isnan, b.Fdef) error("NaN Fdef at vertex: $v") end
end

"""
Updates the first Piola-Kirchhoff stress tensor, smoothes the
normal vectors and computes the deformation gradient tensor.
"""
function update_sph!(b :: BlowSim)
    for v = 1:b.nv
        deformationgradient!(v, b)
        # First Piola-Kirchhoff stress tensor using Green Lagrange Strain
        b.P[v] = b.Fdef * stress(b.material, (b.Fdef * b.Fdef' -  b.eye)*0.5)

        # Smoothing the normal vector
        # n = Vec3(0.0,0.0)
        # for (k, ov) in enumerate(b.V_subinds[v])
        #     n += b.M[ov] / b.R[ov] * b.Nv[ov] * b.kernelvals[v][k]
        # end
        # b.Nvs[v] = n / norm(n)
    end
end

"""
Executes a Stormer-Verlet integration step, given a BlowSim, pressure,
time step, and friction coefficient.
"""
function inflate!(b :: BlowSim, pressure :: Scalar, dt :: Scalar, mu :: Scalar)
    # Update normals etc
    update_all!(b)

    # Time step for comparison
    dt_f = 48. * dt^2

    for v = 1:b.nv
        # Acceleration using equation of motion, includes pressure and
        # elastic forces.
        b.a[v] = Vec3(0.0, 0.0, 0.0)

        for (k, ov) in enumerate(b.V_subinds[v])
            # Stress contribution
            b.a[v] += (b.P[v] / b.R[v]^2 + b.P[ov] / b.R[ov]^2) *
                b.kernelgrad[v][k] * b.M[ov]
        end

        # Pressure contribution
        b.fp[v] = pressure * b.Av[v] * b.Nv[v]

        current_cfl = cfl(b, v)
        if current_cfl < dt_f
            dt_f = current_cfl
        end
    end

    # adjust timestep, and compute dt squared and inverse
    dt = .25 / sqrt(3.) * sqrt(dt_f)
    dt2 = dt * dt
    dtinv = 1.0 / dt

    # Compute everything needing dt and execute verlet update
    for v = 1:b.nv
        # Friction contribution
        b.ff[v] = -mu * (b.V[v] - b.Vold[v]) * dtinv

        a = (b.fp[v] + b.ff[v]) / b.M[v]

        # Verlet update
        b.Vnew[v] = 2.0 * b.V[v] - b.Vold[v] + (b.a[v] + a) * dt2
    end

    # Swap V and Vold
    b.Vold = deepcopy(b.V)
    b.V = deepcopy(b.Vnew)
```

```
732
733     # Displacement update, used in update_stress!
734     b.U = deepcopy(b.V - b.V0)
735
736     dt
737 end
738
739 function cfl(b :: BlowSim{T,K}, v :: Index) where {T <: SpringMaterial, K <:
        Fixed}
740     (b.M[v] * b.h) / norm(b.a[v] * b.M[v] + b.fp[v])
741 end
742
743 function rename(F :: Matrix{Int})
744     for (k,v) = enumerate(sort(unique(F[:])))
745         if v == k
746             nothing
747         else
748             F[find(x -> x==v, F)] = k
749         end
750     end
751     F
752 end
753
754 function rename(F :: FaceList)
755     F = [f[i] for f in F, i=1:3]
756     Fr = rename(F)
757     [Face(Fr[i,:]) for i=1:size(Fr,1)]
758 end
759
760 """
761 Find the conformal factor of vertex v. V_uv is an ARAP parametrization,
762 v is the index in b.V en i is the index in V_uv.
763 """
764 function conformalfactor(b :: BlowSim, V_uv :: Vector2List,
765                          v :: Index, i :: Index, subinds :: IndexList)
766     n = length(b.emap[v])
767     λv = 0.
768     for (k, edge) in enumerate(b.emap[v])
769         ov = symdiff(b.E[edge], v)[1]            # find other vertex in edge
770         j = find(x-> x == ov, subinds)           # find index of ov in V_uv
771         # if ov and thus the edge is not present in V_uv, continue
772         if isempty(j) ||  ov âĹĽ subinds
773             n -= 1
774             continue
775         else
776             j = j[1]
777         end
778         λv += b.Le[edge] / norm(V_uv[i] - V_uv[j])
779     end
780     λv / n
781 end
782
783 """ ARAP parametrization from libigl. """
784 function parametrization(V :: VectorList, F :: FaceList)
785     # Rename F, such that it does not index into V at i > length(V)
786     F = arr2mat(F)
787     F = rename(F)
788
789     # Convert to Eigen matrices
790     V = j2e(V)
791     F_uv = j2e(F)
792
793     # Make Eigen matrices needed for the parametrization computations
794     bnd = igl.eigen[:MatrixXi]()            # indices of boundary vertices
795     bnd_uv = igl.eigen[:MatrixXd]()         # position of vertices on UV plane
796     initial_guess = igl.eigen[:MatrixXd]()  # intial solution
797     b = igl.eigen[:MatrixXi][:Zero](0, 0)   # boundary indices (of V)
798     bc = igl.eigen[:MatrixXd][:Zero](0, 0)  # boundary values
799     V_uv = igl.eigen[:MatrixXd]()           # UV coordinates
800
801     # Compute the initial solution for ARAP (harmonic parametrization)
802     igl.boundary_loop(F_uv, bnd)
803     if isempty(bnd)
804         error("Boundary is empty, no disk topology. Set smoothing length lower."
              )
805     end
806     igl.map_vertices_to_circle(V, bnd, bnd_uv)
807     igl.harmonic(V, F_uv, bnd, bnd_uv, 1, initial_guess)
808
809     V_uv = igl.eigen[:MatrixXd](initial_guess)  # important, make a copy of it!
810
811     # Add dynamic regularization to avoid to specify boundary conditions
812     arap_data = igl.ARAPData()
```

```
    arap_data[:with_dynamics] = true

    # Initialize ARAP
    arap_data[:max_iter] = 100
    igl.arap_precomputation(V, F_uv, 2, b, arap_data) # 2 for solving in 2D

    # Solve arap using the harmonic map as initial guess
    igl.arap_solve(bc, arap_data, V_uv)

    mat2vec(e2j(V_uv)), mat2face(e2j(F_uv))
end

"""
Minimum geodesic distance from each vertex in surface to any vertex in
"verts" using the algorithm from 'geodesics in heat', Crane et al, 2012. m is
a smoothing parameter, with larger values smoothing more. Returns a ScalarList
of distances. Implementation of this, including functions used in this one,
follows the one found at
https://github.com/gallantlab/pycortex/blob
/79350a6d6df8025f12a7ecfa1bd8fc5246322d65/cortex/polyutils.py.
"""
function geodesicdistance(b :: BlowSim, verts :: IndexList;
                          m :: Scalar = 1.0, update :: Bool = false)
    # Update the factorization of the negative Laplace operator if m differs
    # or if update is needed (e.g. every time step)
    if b.m âĹă m
        b.m = m
        update_edges!(b)
        update_cotangents!(b)
        laplaceoperator!(b)
        nLCfactor!(b, m)
    elseif update
        update_edges!(b)
        update_cotangents!(b)
        laplaceoperator!(b)
        nLCfactor!(b, m)
    end

    # I. Find u, the heat values

    # Set initial heat values
    u0 = zeros(b.nv)
    u0[verts] = 1.0
    # Solve system using factorization
    u = b.fact_nLC \ u0

    # II. Use u to find phi, the distances

    # Compute the surface gradient of u
    surfacegradient!(b, u)
    # Normalized negative gradient of u
    X = -b.gradu ./ norm.(b.gradu)
    # Integrated divergence of X
    divx!(b, X)
    # Solve system for the distances
    φ = b.nLC \ b.divx
    φ - minimum(φ)
end

"""
Returns the factorization of the negative Laplace operator.
"""
function nLCfactor!(b :: BlowSim, m :: Scalar)
    # Time of heat evolution
    t = m * (sum(b.Le) / b.ne)^2
    # Backward Euler matrix
    b.lfac = spdiagm(b.Av, 0) - t * b.nLC
    b.fact_nLC = ldltfact(b.lfac)
    nothing
end

""" Updates the edge tangentials and lengths. """
function update_edges!(b :: BlowSim)
    for (k, e) in enumerate(b.E)
        b.Te[k] = b.V[e[2]] - b.V[e[1]]
        b.Le[k] = norm(b.Te[k])
    end
end

"""
Updates the values of the cotangents of the angles between vectors and the
matrix used to normalize it.
"""
```

```
function update_cotangents!(b :: BlowSim)
    for (i, f) in enumerate(b.F)
        b.cots[i,1] = cot(b.V[f[2]] - b.V[f[1]], b.V[f[3]] - b.V[f[1]])
        b.cots[i,2] = cot(b.V[f[3]] - b.V[f[2]], b.V[f[1]] - b.V[f[2]])
        b.cots[i,3] = cot(b.V[f[1]] - b.V[f[3]], b.V[f[2]] - b.V[f[3]])
    end
end

""" Cotangens of angle between two vectors. """
cot(a :: Vec3, b :: Vec3) =  dot(a,b) / norm(cross(a,b))

""" To compute the negative Laplace operator. """
function laplaceoperator!(b :: BlowSim)
    fill!(b.A, 0.)
    for (i, f) in enumerate(b.F)
        b.A[f[2], f[3]] += 0.5 * b.cots[i, 1]
        b.A[f[3], f[2]] += 0.5 * b.cots[i, 1]
        b.A[f[3], f[1]] += 0.5 * b.cots[i, 2]
        b.A[f[1], f[3]] += 0.5 * b.cots[i, 2]
        b.A[f[1], f[2]] += 0.5 * b.cots[i, 3]
        b.A[f[2], f[1]] += 0.5 * b.cots[i, 3]
    end
    b.An = spdiagm(squeeze(sum(b.A,1),1), 0)

    # Negative laplace operator
    b.nLC = b.A - b.An
    nothing
end

"""
For a given scalar-valued function across vertices, returns the
gradients in x, y, and z direction at each vertex.
"""
function surfacegradient!(b :: BlowSim, u :: ScalarList)
    for (k,f) in enumerate(b.F)
        fe12 = cross(b.Nf[k], b.V[f[2]] - b.V[f[1]])
        fe23 = cross(b.Nf[k], b.V[f[3]] - b.V[f[2]])
        fe31 = cross(b.Nf[k], b.V[f[1]] - b.V[f[3]])
        b.gradu[k] = (fe12 * u[f[3]] + fe23 * u[f[1]] + fe31 * u[f[2]]) / (2.0 *
              b.Af[k])
    end
end

"""
Computes the integrated divergence of X at each vertex.
"""
function divx!(b, X :: VectorList)
    b.divx = zeros(b.nv)
    for (i, v) in enumerate(b.V)
        for j in b.fmap[i]
            # Find the two indices in the face not equal to i, and their index
            # in the face
            ind = findfirst(b.F[j], i)
            k, l = deleteat!([b.F[j].data...], ind)
            n, m = deleteat!([1,2,3], ind)
            # Compute the integrated divergence
            b.divx[i] += 0.5 * (b.cots[j, m] * dot((b.V[k] - b.V[i]), X[j]) +
                                b.cots[j, n] * dot((b.V[l] - b.V[i]), X[j]))
        end
    end
end

# """
# Function to show a mesh using Makie and GeometryTypes, given a list of
# vertices and faces, and the number of vertices and faces.
# """
# function showmesh(V :: VectorList, F :: FaceList, nv :: Index, nf :: Index)
#     V = [Point3f0(V[i]) for i in 1:nv]
#     F = [GLTriangle(F[i]) for i in 1:nf]
#     mesh = GLNormalMesh(V, F)
#
#     scene = Scene(color=:white)
#     wireframe(mesh, linewidth=2f0, color=:gray)
#     Makie.mesh(mesh, color=:darkgray)
#
#     pos = map(mesh.vertices, mesh.normals) do p, n
#        p => p .+ (normalize(n) .* 4f0)
#     end
#     linesegment(pos, color=RGBA(0.5,0.5,1.0,0.8))
#
#     center!(scene)
#     scene
# end
```

```
978
979  # """
980  # Function to show a two meshes as wireframes using Makie and GeometryTypes,
981  # given a list of vertices and faces, and the number of vertices and faces for
982  # both meshes. It assumes the second mesh is smaller and colors besides the
983  # edges also the faces.
984  # """
985  # function showtwomeshes(V1 :: VectorList, F1 :: FaceList, nv1 :: Index, nf1 ::
         Index,
986  #                        V2 :: VectorList, F2 :: FaceList, nv2 :: Index, nf2 ::
         Index)
987  #     V1 = [Point3f0(V1[i]) for i in 1:nv1]
988  #     F1 = [GLTriangle(F1[i]) for i in 1:nf1]
989  #     mesh1 = GLNormalMesh(V1, F1)
990  #     V2 = [Point3f0(V2[i]) for i in 1:nv2]
991  #     F2 = [GLTriangle(F2[i]) for i in 1:nf2]
992  #     mesh2 = GLNormalMesh(V2, F2)
993  #
994  #     scene = Scene(color=:white)
995  #     wireframe(mesh1, linewidth=2f0, color=:blue)
996  #
997  #     wireframe(mesh2, linewidth=2f0, color=:gray)
998  #     Makie.mesh(mesh2, color=:white)
999  #
1000 #     pos = map(mesh1.vertices, mesh1.normals) do p, n
1001 #        p => p .+ (normalize(n) .* 4f0)
1002 #     end
1003 #     linesegment(pos, color=RGBA(0.5,0.5,1.0,0.8))
1004 #
1005 #     center!(scene)
1006 #     scene
1007 # end
1008
1009 function showplot(VL :: VectorList, F :: FaceList)
1010     F = [f[i]-1 for f in F, i=1:3] # minus 1, because PyPlot is Python
1011     xs = ScalarList(length(VL))
1012     ys = ScalarList(length(VL))
1013     zs = ScalarList(length(VL))
1014     for (k, v) in enumerate(VL)
1015       xs[k] = v[1]
1016       ys[k] = v[2]
1017       zs[k] = v[3]
1018     end
1019     fig = figure()
1020     plot_trisurf(xs, ys, zs, triangles = F)
1021     xlabel("x-axis")
1022     ylabel("y-axis")
1023     zlabel("z-axis")
1024     show()
1025 end
1026
1027 end # module BlowLogicSPH
```

C Neural Networks

C.1 Solving Partial Differential Equations

Listing 7 *Demo file using Eikonal.jl.*

```
1  println("@@@ STARTING")
2
3  include("Eikonal.jl")
4
5  using Eikonal
6  using Flux
7  using Flux.Tracker
8  using Plots
9  Eik = Eikonal # alias
10
11 # Make data
12 # Domain is [0., 10]x[0., 10]
13 L = 10.
14 nx = 5000           # total number of points
15 data = [[L*rand(), L*rand()] for i = 1:nx]
16
17 # Split in test and train set
```

```
shuffled_data = shuffle(data[:])
ntrain = Int(nx * .8)
x_train = shuffled_data[:][1:ntrain]
x_test = shuffled_data[:][ntrain+1:end]

# Plot the points
scatter([x[1] for x in x_train], [x[2] for x in x_train], label = "Trainset")
scatter!([x[1] for x in x_test], [x[2] for x in x_test], label = "Testset")

# Boundary point
x_γ = [5., 5.]

# Make NN
ndense = 10         # Number of units in hidden layer
λ = 0.001           # Learning rate

dist = Eik.Distance(ndense, λ)
opt = SGD(params(dist.m), dist.λ)

epochs = 20
Trainloss = [Float64[] for i=1:epochs]
Testloss = [Float64[] for i=1:epochs]

for i = 1:epochs
    shuffled_x = shuffle!(x_train)
    trloss = 0.0
    teloss = 0.0
    for x in shuffled_x
        back!(Eik.loss(x, x_γ, dist.m))
        # Check if any of the gradients of the weights was NaN. If so, set to
        # zero, so that the NaN does not propagate to the weights and biases.
        # This means that this value of x does not change the weights, and is
        # essentially skipped.
        for p in params(dist.m)
            if any(isnan, p.grad) p.grad .= 0 end
        end
        opt()
    end
    # test loss is average per epoch
    for xt in x_test
        push!(Testloss[i], Flux.Tracker.data(Eik.loss(xt, x_γ, dist.m)))
    end
    for xt in x_train
        push!(Trainloss[i], Flux.Tracker.data(Eik.loss(xt, x_γ, dist.m)))
    end

    if i % 1 == 0
        trloss = sum(Trainloss[i])
        teloss = sum(Testloss[i])
        println("Epoch $i, current train loss is $trloss, test loss is $teloss."
            )
    end
end

# Plot average loss per epoch
trloss = [sum(Trainloss[i])/length(x_train) for i = 1:epochs]
teloss = [sum(Testloss[i])/length(x_test) for i = 1:epochs]

plot(trloss, xlabel = "Number of epochs", ylabel = "Average loss", label = "
     Train loss")
plot!(teloss, label = "Test loss")

# Plot distances of NN as a function of the true distance
traindistances = [Flux.Tracker.data(Eik.Ψt(x_train[i], x_γ, dist.m))[1]
    for i = 1:length(x_train)]
testdistances = [Flux.Tracker.data(Eik.Ψt(x_test[i], x_γ, dist.m))[1]
    for i = 1:length(x_test)]
truetrain = [norm(x_γ - x_train[i]) for i = 1:length(x_train)]
truetest = [norm(x_γ - x_test[i]) for i = 1:length(x_test)]

plot(xlabel = "True distance", ylabel = "Model prediction", legend = :topleft)
scatter!(truetrain, abs.(traindistances), label = "Traindistances")
scatter!(truetest, abs.(testdistances), label = "Testdistances")

# Plot of network output
trainΨ = [Flux.Tracker.data(dist.m(x_train[i]))[1] for i = 1:length(x_train)]
testΨ = [Flux.Tracker.data(dist.m(x_test[i]))[1] for i = 1:length(x_test)]

plot(xlabel = "True distance", ylabel = "Network prediction", legend = :
     bottomleft)
scatter!(truetrain, abs.(trainΨ),  label = "Train set")
scatter!(truetest, abs.(testΨ),  label = "Test set")
```

Listing 8 *Neural Network to solve the eikonal equation main code.*

```
module Eikonal

using Flux
using Flux.Tracker

mutable struct Distance
    ndense :: Int
    m :: Flux.Chain
    λ :: Float64

    function Distance(ndense :: Int, λ :: Float64)
        self = new()
        self.ndense = ndense    # Number of units in hidden layer
        self.λ = λ              # Learning rate
        self.m = Chain(         # Network
          Dense(2, ndense, σ),
          Dense(ndense, 1))

        return self
    end
end

"""
Function satisfying the boundary condition, A, and F, a function that is zero
on the boundary. Note that the boundary does not neccesarily mean the boundary
of the domain, but the set of points to which we wish to know the distance.
"""
A(x, x_γ) = 0.
âĹĞA(x, x_γ) = 0.0

F(x, x_γ) = norm(x - x_γ)
âĹĞF(x, x_γ) = norm(x - x_γ) > 1e-10 ? (x - x_γ) / norm(x - x_γ) : x - x

""" First derivative of sigmoid """
âĹĞσ(x) = σ(x) * (1.0 - σ(x))

""" Trial solution """
Ψt(x, x_γ, m) = A(x, x_γ) + F(x, x_γ) * m(x)

""" Gradient of the neural network """
function âĹĞm(x, m)
    z = m[1].W * x + m[1].b
    m[2].W * (âĹĞσ.(z) .* m[1].W)
end

""" Gradient of trial solution """
âĹĞΨt(x, x_γ, m) = âĹĞA(x, x_γ) + âĹĞF(x, x_γ) * m(x)[1] + F(x, x_γ) * âĹĞm(x, m
    )'

""" Loss function """
function loss(x, x_γ, m)
    dΨt = âĹĞΨt(x, x_γ, m)
    normâĹĞΨt = dΨt[1]^2 + dΨt[2]^2
    # If x = x_γ, the loss function does not work.
    if norm(x - x_γ) < 1e-10
        return normâĹĞΨt
    else
        return normâĹĞΨt - 2.0 * sqrt(normâĹĞΨt) + 1
    end
end

end # module Eikonal
```

Listing 9 *Demo file using BlowNN.jl.*

```
println("@@@ STARTING")

include("BlowNN.jl")

using BlowNN
using Flux
using Flux: @epochs, throttle
using Plots
using BSON: @load, @save
bnn = BlowNN

# Material parameters
```

```
rho = 1.0
λ = 10.0
μ = 10.0
# Can choose between circle, square, ellipse, circlearc, parabola and line.
shape = "circle";
nv = 60
ngv = 20
L = 6.0

# NN parameters
η = 1e-8              # Learning rate
n = 10                      # Number of sigmoid units
# Can choose between relu and sigmoid
activationfunction = "relu";
pressure = 1.0             # pressure

# Creation
material = bnn.LinearElastic(rho, λ, μ);
model = bnn.Model(n, η, 1, activationfunction);

# Load/save weights
@load "zeroinit.bson" weights
Flux.loadparams!(b.model.NN, weights)
# weights = Tracker.data.(params(b.model.NN));
# @save "zeroinit.bson" weights

# Create geometry
b = bnn.BlowSim{typeof(material)}(
    nv, ngv, L, material, shape, model, pressure);
print("--- Plot initial:")
bnn.showplot(b.V0, [], b.nv, L)
savefig("initialPDEinflate.pdf")

print("--- Plot NN initial guess:")
bnn.showplot(b.Vold, [], b.nv, L)
savefig("initialguessPDEinflate.pdf")

opt = Momentum(params(b.model.NN), b.model.η)
# SGD(Flux.params(b.model.NN), b.model.η)
# ADAM(params(b.model.NN), b.model.η)
# Nesterov(params(b.model.NN), b.model.η)
#
Loss = Float64[]
epochs = 10

@time for i = 1:epochs
    Flux.back!(bnn.loss(b))
    # Check if any of the gradients of the weights was NaN. If so, set to
    # zero, so that the NaN does not propagate to the weights and biases.
    # This means that this value of x does not change the weights, and is
    # essentially skipped.
    for p in Flux.params(b.model.NN)
        if any(isnan, p.grad) p.grad .= 0 end
    end
    opt()
    push!(Loss, Flux.Tracker.data(bnn.loss(b)))
    bnn.update_all!(b)

    if i % 1 == 0
        println("Epoch ", i, ", mean loss is ", bnn.loss(b) / b.nv)
        bnn.showplot(b.V, [b.a .* b.M, b.fp], b.nv, L)
    end
end

# Plotting
bnn.showplot(V, [], b.nv, L)

# loss
plot(Loss, xlabel = "Number of epochs", ylabel = "Loss value", legend = false)
savefig("PDEinflateloss.pdf")
```

Listing 10 *Neural Nerwork to solve governing equation main code.*

```
module BlowNN

using Flux
using Flux.Tracker
using StaticArrays
using Plots
```

```
7
8  const ε = 1e-10
9
10 const Scalar = Float64
11 const Index = Int
12 const TrReal = Flux.Tracker.TrackedReal{Scalar}
13
14 const ScalarList = Array{Scalar,1}
15 const IndexList = Array{Index,1}
16
17 const Param = TrackedArray
18 const Edge = SVector{2, Index}
19 const EdgeList = Array{Edge,1}
20
21 const ParamList = Array{Param, 1}
22 const ParamMap = Array{ParamList, 1}
23 const TrRealList = Array{TrReal, 1}
24 const TrRealMap = Array{TrRealList, 1}
25
26 ParamList(nv :: Index, A :: Array{Scalar,1}) = ParamList([param(A) for i = 1:nv]
       )
27 TrRealList(nv :: Index, a :: Scalar) = TrRealList([a for i= 1:nv])
28
29 """ Function to decide which shape vertices to use. """
30 function load_vertices(shape :: String, args...; kwargs...)
31     if shape == "line"
32         return load_vertices_line(args...; kwargs...)
33     elseif shape == "parabola"
34         return load_vertices_parabola(args...; kwargs...)
35     elseif shape == "circlearc"
36         return load_vertices_circlearc(args...; kwargs...)
37     elseif shape == "circle"
38         return load_vertices_circle(args...)
39     elseif shape == "square"
40         return load_vertices_square(args...)
41     elseif shape == "ellipse"
42         return load_vertices_ellipse(args...)
43     else
44         error("No such shape implemented.")
45     end
46 end
47
48 """
49 Functions to create an array with vertex locations.
50 If the shape is unconnected, standard 10 ghost vertices are placed at both ends.
51 In the case of a line, if ngv is uneven, (ngv-1)/2 are placed on the left and
52 (ngv+1)/2 on the right. If the shape is connected at the first and last vertex,
53 L is the circumference. Also, the vertices can start at any point, but they
54 need to go anti-clockwise from there, due to the computation of the normal
55 vectors later on. Depending on the exact shape, the number of vertices
56 returned may vary from the number specified. If a unit speed parametrization is
57 easy to make, the length is L and vertices are evenly spaced.
58 """
59 function load_vertices_line(nv :: Index, L :: Scalar; ngv :: Index = 20)
60     n = nv + ngv
61     spacing = L / (nv - 1)
62     [param([(i-1)*spacing - spacing * fld(ngv, 2), 0.0]) for i in 1:n]
63 end
64
65 function load_vertices_parabola(nv :: Index, L :: Scalar; ngv :: Index = 10)
66     n = nv + ngv
67     # Note that the length returned is > L
68     x = linspace(- 0.5 * L, 0.5 * L, n)
69     [param([x[i], -x[i]^2]) for i in 1:n]
70 end
71
72 function load_vertices_circlearc(nv :: Index, L :: Scalar; ngv :: Index = 10)
73     # cospi and sinpi calculate their arugments times π.
74     n = nv + ngv
75     spacing = 1 / nv
76     # factor (0.5 * L / π) to get a circle with circumference âĽĹ L.
77     [param( [-(0.5 * L / π) * cospi(spacing * (i - 1 - ngv * 0.5)),
78           (0.5 * L / π) * sinpi(spacing * (i - 1 - ngv * 0.5))]) for i in 1:n+1]
79 end
80
81 function load_vertices_circle(nv :: Index, L :: Scalar)
82     # cospi and sinpi calculate their arugments times π.
83     spacing = 2 / nv
84     [param( [(0.5 * L / π) * cospi(spacing * i),
85            (0.5 * L / π) * sinpi(spacing * i)]) for i in 1:nv]
86 end
87
88 function load_vertices_ellipse(nv :: Index, L :: Scalar)
```

```
    b = L*0.1
    a = L*0.2
    spacing = 2 / nv
    [param( [a * cospi(spacing * i), b * sinpi(spacing * i)]) for i in 1:nv]
end

function load_vertices_square(nv :: Index, L :: Scalar)
    # Creates four edges of the square, each with length L / 4
    l = L / 4.0
    # the total number of vertices returned will be 4 * floor(nv/4).
    n = fld(nv, 4)
    spacing = l / n
    V1 = [param([l / 2.0 - (i-1) * spacing, l / 2.0]) for i in 1:n]
    V2 = [param([- l / 2.0, l / 2.0 - (i-1) * spacing]) for i in 1:n]
    V3 = [param([- l / 2.0 + (i-1) * spacing, - l / 2.0]) for i in 1:n]
    V4 = [param([l / 2.0, -l / 2.0 + (i-1) * spacing]) for i in 1:n]
    append!(V1, V2)
    append!(V1, V3)
    append!(V1, V4)
end

""" Loads the edges, depending on the shapetype (connected or not). """
function load_edges(shapetype :: String, nv :: Index)
    E = [Edge([i, i+1]) for i in 1:nv-1]
    if shapetype == "unconnected"
        return E
    elseif shapetype == "connected"
        push!(E, Edge([nv, 1]))
        return E
    else
        error("No such shape implemented.")
    end
end

"""
Compute the edge maps: for each vertex, finds its two adjacent edges.
This function returns two maps: one with edges going out of
the vertex (clockwise) and one incident to the vertex
(anti-clockwise).
"""
function build_edge_maps(nv :: Index, E :: EdgeList, shapetype :: String)
    emap_pos = IndexList()
    emap_neg = IndexList()

    for e in E
        push!(emap_pos, e[2])
        push!(emap_neg, e[1]-1)
    end

    if shapetype == "connected"
        emap_neg[1] = E[end][1]
    elseif shapetype == "unconnected"
        # The fist point sees itself as previous element, and the last one
        # itself as the next element.
        emap_neg[1] = 1
        push!(emap_neg, nv - 1)
        push!(emap_pos, nv)
    else
        error("No such shapetype implemented.")
    end

    emap_pos, emap_neg
end

# Abstract type to hold material parameters
abstract type SpringMaterial end

# Saint Venant-Kirchhoff material
struct LinearElastic <: SpringMaterial
    rho :: Scalar
    λ :: Scalar
    μ :: Scalar
end

density(material :: SpringMaterial) = material.rho
lambda(material :: LinearElastic) = material.λ
mu(material :: LinearElastic) = material.μ

mutable struct Model
    ndense :: Index
    η :: Scalar
    x0 :: Index
    NN :: Chain
```

```
    af :: Function
    âĹĞaf :: Function
    âĹĞâĹĞaf :: Function

    function Model(ndense :: Index, η :: Scalar, x0 :: Index, name :: String)
        self = new()
        self.ndense = ndense    # Number of units in hidden layer
        self.η = η              # Learning rate
        self.x0 = x0
        if name == "sigmoid"
            self.af = σ
            self.âĹĞaf = âĹĞσ
            self.âĹĞâĹĞaf = âĹĞâĹĞσ
        elseif name == "relu"
            self.af = relu
            self.âĹĞaf = âĹĞrelu
            self.âĹĞâĹĞaf = âĹĞâĹĞrelu
        else
            error("No such activation function implemented.")
        end
        self.NN = Chain(         # Neural network architecture
          Dense(3, ndense, self.af),
          Dense(ndense, 2))
        self
    end
end

# Activation functions' derivatives
âĹĞσ(x) = σ(x) * (1.0 - σ(x))
âĹĞâĹĞσ(x) = âĹĞσ(x) * (1.0 - 2.0 * σ(x))

âĹĞrelu(x) = x < 0.0 ? 0.0 : 1.0
âĹĞâĹĞrelu(x) = 0.0

"""
Computes the second Piola-Kirchhoff stress tensor for linear elastic and
hyperelastic constitutive material models.
"""
function stress(material :: LinearElastic, F :: TrackedArray)
    E = 0.5 * (F' * F -  eye(2))
    material.λ * trace(E) * eye(2) + 2.0 * material.μ * E
end

"""
Distance between two points on the line, given by the absolute value of the
difference of the values at v and ov in the list containing the cumulative sums
of the distances. For shapes of which the first and last vertex are connected,
this function assumes that the last entry of LS contains circumference
of the shape.
"""
function distance_connected(LS :: TrRealList, v :: Index, ov :: Index)
    Flux.Tracker.data(min(abs(LS[v] - LS[ov]), LS[end] - abs(LS[v] - LS[ov])))
end

function distance_unconnected(LS :: TrRealList, v :: Index, ov :: Index)
    Flux.Tracker.data(abs(LS[v] - LS[ov]))
end

"""
Struct containing all relevant information of a mesh during and after inflation,
e.g., material parameters, vertex locations, number of vertices, density, etc.
Also contains the function to initialize the struct.
"""
mutable struct BlowSim{T<:SpringMaterial}
    # Neural Network and its parameters
    model :: Model

    # Material parameters, as well as the shape and distance function
    material :: T
    shapetype :: String
    distance :: Function

    # Number of (ghost)vertices and edges
    ngv :: Index
    nv :: Index
    ne :: Index

    # Index of first and last moving particle
    start_at :: Index
    end_at ::Index

    # Convenient representation of the mesh
    V :: ParamList
```

```
    E :: EdgeList

    # Lookup tables
    emap_pos :: IndexList
    emap_neg :: IndexList

    #
    # Derived geometric quantities (normals, areas, lengths, cot of angles)
    #

    Ne :: ParamList
    Te :: ParamList
    Le :: TrRealList
    LSe :: TrRealList

    Nv :: ParamList
    Nvs :: ParamList
    Lv :: TrRealList

    # Forces, used for plotting
    a :: ParamList
    fp :: ParamList

    #
    # Dynamic parameters
    #
    l :: Scalar           # Length of domain
    rho :: Scalar         # Density
    p :: Scalar           # Pressure

    M :: ScalarList       # Vertex mass
    Vold :: ParamList     # Vertex position initial guess
    V0 :: ParamList       # Vertex initial position

    P :: ParamList    # First P-K stress tensor for each vertex
    Fdef :: ParamList # Deformation gradient for each vertex

    x0 :: Param

    # Initialize all quantities
    function BlowSim{T}(
        nv :: Index,
        ngv :: Index,
        l :: Scalar,
        material :: T,
        shape :: String,
        model :: Model,
        p :: Scalar,
        ) where {T}

        self = new()
        # NN
        self.model = model

        # Find material and kernel parameters
        self.material = material

        if shape in ["circle", "square", "ellipse"]
            self.shapetype = "connected"
            self.distance = distance_connected
            self.ngv = 0
        elseif shape in ["line", "parabola", "circlearc"]
            self.shapetype = "unconnected"
            self.distance = distance_unconnected
            self.ngv = ngv
        else
            println("No such shape implemented.")
        end

        # Load mesh
        self.V = load_vertices(shape, nv, l; ngv = self.ngv)
        self.nv = length(self.V)
        self.V0 = deepcopy(self.V)

        self.E = load_edges(self.shapetype, self.nv)
        self.ne = length(self.E)

        self.l = l
        self.rho = density(material)
        self.p = p

        # Indices of first and last moving particle. End points are fixed
        # for the line, so do not have to loop over ghost vertices.
```

```
        # For the circle, this reduces to v = 1:b.nv, since b.ngv = 0.
        self.start_at = fld(self.ngv, 2) + 1
        self.end_at = self.nv - cld(self.ngv, 2)

        # Create look up tables
        self.emap_pos, self.emap_neg = build_edge_maps(self.nv, self.E, self.
            shapetype)

        # Pre-allocate space for derived gemoetrical information
        self.Ne = ParamList(self.nv, [0.0, 0.0])  # edge normals
        self.Te = ParamList(self.nv, [0.0, 0.0])  # edge tangentials
        self.Le = TrRealList(self.ne, 0.0)        # edge lengths
        self.LSe = ScalarList(self.ne)            # edge lengths cumulative sum

        self.a = ParamList(self.nv, [0.0, 0.0])  # acceleration due to elastic
            forces
        self.fp = ParamList(self.nv, [0.0, 0.0]) # pressure forces

        self.Nv = ParamList(self.nv, [0.0, 0.0])  # vertex normals
        self.Lv = TrRealList(self.nv, 0.0)  # vertex lengths

        # Compute derived geometric information
        update_geometry!(self)

        # Compute mass
        self.M = ScalarList([Flux.Tracker.data(l) for l in self.Lv]) .* self.rho
        # BC
        self.x0 = self.V[self.model.x0]

        # Pre-allocate room for first Piola-Kirchhoff stress tensor and the
        # deformation gradient tensor for each vertex
        self.P = [Flux.param([0.0 0.0; 0.0 0.0]) for i = 1:self.nv]
        self.Fdef = [Flux.param([1.0 0.0; 0.0 1.0]) for i = 1:self.nv]

        update_all!(self)
        self.Vold = deepcopy(self.V)

        return self
    end
end

# BC function and its gradient

""" Function that is zero on the boundary """
F(x, x0) = (x[1] - x0[1])^2 + (x[2] - x0[2])^2

âĹĞF(x, x0) = 2.0 * (x - x0)

# Jacobian, divergence etc of trial solution and neural network

""" Trial solution, in this case, the displacement. """
Ψt(x, b :: BlowSim) = F(x, b.x0) * b.model.NN([x.data..., b.p])

""" Jacobian of the trial solution. """
âĹĞΨt(x, b :: BlowSim) = kron(âĹĞF(x, b.x0)', b.model.NN([x.data..., b.p])) +
                                            (F(x, b.x0) * âĹĞm(x, b))[1:2,1:2]

""" Divergence of the Jacobian of the trial solution """
function divâĹĞΨt(x, b :: BlowSim)
    dF = âĹĞF(x, b.x0)
    dm = âĹĞm(x, b)
    f = F(x, b.x0)
    ddm = δ2m(x, b)
    ddmm = δ2mixedm(x, b)
    bm = b.model.NN([x.data..., b.p])

    d1 = 2.0 * bm + 2.0 * dF .* dm[1:3:4] + f * ddm[1:3:4]
    d2 = dF .* dm[4:-3:1] + dF[2:-1:1] .* dm[2:1:3] + f * ddmm[2:-1:1]

    return d1 + d2
end

""" Jacobian of the neural network. """
function âĹĞm(x, b :: BlowSim)
    x = [x.data..., b.p]
    z = b.model.NN[1].W * x + b.model.NN[1].b
    b.model.NN[2].W * (b.model.âĹĞaf.(z) .* b.model.NN[1].W)
end

""" Second derivatives of the neural network. """
function δ2m(x, b :: BlowSim)
    x = [x.data..., b.p]
    z = b.model.NN[1].W * x + b.model.NN[1].b
```

```
    b.model.NN[2].W * (b.model.âĹĞâĹĞaf.(z) .* b.model.NN[1].W.^2)
end

""" Mixed second derivative of the NN w.r.t. x and y, not p """
function δ2mixedm(x, b :: BlowSim)
    x = [x.data..., b.p]
    z = b.model.NN[1].W * x + b.model.NN[1].b
    b.model.NN[2].W * (b.model.âĹĞâĹĞaf.(z) .* b.model.NN[1].W[:,1] .* b.model.
        NN[1].W[:,2])
end

""" Divergence of the first Piola-Kirchhoff stress tensor """
function âĹĞP(v, b :: BlowSim)
    S = stress(b.material, b.Fdef[v])
end

""" Divergence of second Piola-Kirchhoff stress tensor """
function âĹĞS(x, b :: BlowSim)
    ddm = δ2m(x, b)
    dF = âĹĞF(x, b.x0)
    dm = âĹĞm(x, b)
    f = F(x, b.x0)
    ddmm = δ2mixedm(x, b)
    jacΨt = âĹĞΨt(x, b)

    âĹĞE = ((ddm[:,1] + ddm[:,2])' * jacΨt)' + jacΨt' * divâĹĞΨt(x, b)
    âĹĞtrE = ddm[1:3:4] + dF .* dm[4:-3:1] + dF[2:-1:1] .* dm[2:1:3] + f * ddmm[
        2:-1:1]
    return b.material.λ * âĹĞtrE + 2.0 * b.material.μ * âĹĞE
end

""" Loss function """
function loss(b :: BlowSim)
    loss = 0.0

    for v = 1:b.nv
        b.a[v] -= b.a[v]

        # Divergence of first Piola-Kirchhoff stress tensor
        b.a[v] -= b.Fdef[v] * âĹĞS(b.V[v], b) +
                        (divâĹĞΨt(b.V[v], b)' * stress(b.material, b.Fdef[v]))'

        # Pressure contribution
        b.fp[v] = b.p * b.Lv[v] * b.Nv[v]

        loss += norm(b.a[v] + b.fp[v] / b.M[v])^2
    end
    loss
end

""" Collects all update functions """
function update_all!(b :: BlowSim)
    update_geometry!(b)
    update_positions!(b)
    deformationgradient!(b)
end

""" Collects update functions regarding to geometrical quantities. """
function update_geometry!(b :: BlowSim)
    update_edges!(b)
    update_vertices!(b)
end

""" Updates vertex positions according to the NN. """
function update_positions!(b :: BlowSim)
    for v = 1:b.nv
        b.V[v] = b.V0[v] + Ψt(b.V0[v], b)
    end
end

""" Updates the edge normals, areas, tangentials and lengths. """
function update_edges!(b :: BlowSim)
    for (k, e) in enumerate(b.E)
        v = b.V[e[1]]
        w = b.V[e[2]]

        b.Ne[k] = Param([w.data[2] - v.data[2], v.data[1] - w.data[1]])
        if (b.shapetype == "unconnected") b.Ne[k] = -b.Ne[k] end
        b.Ne[k] /= norm(b.Ne[k])

        # Edge tangentials and lengths
        b.Te[k] = v - w
        b.Le[k] = norm(b.Te[k])
```

```
        # Add 0.0 at the front, so that LSe[end] - LSe[1] = LSe[end]
        b.LSe = unshift!(cumsum(b.Le), 0.0)
    end
end

""" Updates vertex normals and lengths. """
function update_vertices!(b :: BlowSim)
    if b.shapetype == "unconnected"
        from = 2
        to = b.nv-1
        b.Lv[1] = 0.5 * b.Le[1]
        b.Lv[end] = 0.5 * b.Le[end]
        # Approximate the normal in the last vertices with the edge normals
        b.Nv[1] = b.Ne[1]
        b.Nv[end] = b.Ne[end]
    else
        from = 1
        to = b.nv
    end
    for k = from:to
        # adjacent edge index, clockwise direction for circle, left for line.
        p = b.emap_neg[k]
        b.Lv[k] = 0.5 * (b.Le[p] + b.Le[k])
        Ns = b.Le[p] * b.Ne[p] + b.Le[k] * b.Ne[k]
        b.Nv[k] = Ns / norm(Ns)
    end
end

""" Computes the deformation gradient tensor. """
function deformationgradient!(b :: BlowSim)
    for v = 1:b.nv
        b.Fdef[v] -= b.Fdef[v]           # Cannot set to zero immediately
        b.Fdef[v] += [1.0 0.0; 0.0 1.0]  # identity matrix
        b.Fdef[v] += âĹĞΨt(b.V[v], b)      # Jacobian of displacement
    end
 end

 """ Function to plot the inflated shape. """
 function showplot(V, VM, nv :: Index, l :: Scalar)
     X = [Flux.Tracker.data(v[1]) for v in V]
     Y = [Flux.Tracker.data(v[2]) for v in V]

     # Also plot pressure, friction and elastic forces.
     n = length(VM)
     F = [Array{Float64, 2}(nv, 2) for i = 1:n]

     for (i, VL) in enumerate(VM)
         for v = 1:nv
             for j = 1:2
                 F[i][v,j] = Flux.Tracker.data(VL[v][j])
             end
         end
     end

     p = plot(X, Y, shape = :circle,
        # Options if you only want markers, without borders and no lines
        # linewidth = 0,
        # markersize = 2,
        # markerstrokealpha = 0
        label = "Vertex locations",
        aspect_ratio = :equal,
       # xlim=(-2,2), ylim = (-2, 2),
        legend = :topright)

     for i = 1:n
         quiver!(X, Y, quiver = (F[i][:,1], F[i][:,2]))
     end

     display(p)
 end

end # module BlowNN
```

C.2 2D Code

Listing 11 *Demo file using DistNN2D.jl.*

```
println("@@@ STARTING")
```

```
2
3  include("DistNN2D.jl")
4
5  using DistNN2D
6  using Flux
7  using Plots
8
9  dnn2 = DistNN2D # alias
10
11 #
12 ### First make a Model containing a NN, then train this NN on data
13 #
14
15 # NN parameters
16 λ = 0.01              # Learning rate
17 n_dense = 50          # Number of sigmoid units in first hidden layer
18 n_conv = 5            # Number of sigmoid units in second hidden layer
19 k = 21                # Total number of points in curve
20 af = relu             # Activation function, (leaky)relu, elu, sigmoid, or swish
21 model = dnn2.Model(n_dense, n_conv, λ, k, af);
22 opt = ADAM(params(model.NN), model.λ)
23
24 # Load/save weights
25 # dnn2.loadweights("dnn2_model", model)
26 # dnn2.saveweights("dnn2_model", model)
27
28 #
29 ### Read and shuffle test and train datasets
30 #
31 rs = nothing
32 Xtrain, Ytrain = dnn2.readandshuffle("Xtrain", "Ytrain", randomseed = rs);
33 Xtest, Ytest = dnn2.readandshuffle("Xtest", "Ytest", randomseed = rs);
34
35 # More data sets
36 Xdata_huge, Ydata_huge = dnn2.readandshuffle("Xtrain_huge", "Ytrain_huge",
       randomseed = rs);
37
38 Xcircle_r05, Ycircle_r05 = dnn2.readandshuffle("Xcircle_r05", "Ycircle_r05");
39 Xcircle_r1, Ycircle_r1 = dnn2.readandshuffle("Xcircle_r1", "Ycircle_r1");
40 Xcircle_r2, Ycircle_r2 = dnn2.readandshuffle("Xcircle_r2", "Ycircle_r2");
41 Xcircle_r3, Ycircle_r3 = dnn2.readandshuffle("Xcircle_r3", "Ycircle_r3");
42 Xellipse, Yellipse = dnn2.readandshuffle("Xellipse_1_025", "Yellipse_1_025");
43 Xsine, Ysine = dnn2.readandshuffle("Xsine", "Ysine");
44 Xsine05, Ysine05 = dnn2.readandshuffle("Xsine05", "Ysine05");
45
46 #
47 ### Train model
48 #
49
50 epochs = 10
51 batchsize = 200
52 args = (Xtrain, Ytrain, Xtest, Ytest, model, epochs, batchsize, opt)
53 @time trainloss, testloss = dnn2.train!(args..., verbose = .01)
54
55 #
56 ### Plot results
57 #
58 p = plot(Trainloss[:], label = "Training loss", xlabel = "Number of batches")
59 plot!(Testloss[:], label = "Test loss", ylabel = "Loss value")
60
61 # Plot distances of NN as a function of the true distance
62 traindistances = [Flux.Tracker.data(model.NN(x))[1] for x in Xtrain]
63 testdistances = [Flux.Tracker.data(model.NN(x))[1] for x in Xtest]
64 euclideandistancestrain = [norm(x[1,:] - x[end,:]) for x in Xtrain]
65 euclideandistancestest = [norm(x[1,:] - x[end,:]) for x in Xtest]
66
67 c = palette(:default)[1:4]
68
69 plot(xlabel = "True distance", ylabel = "Prediction", legend = :topleft)
70 m = max(maximum(Ytest), maximum(Ytrain))
71 plot!([0,m], [0, m], label = "Target", c = :gray)
72 scatter!(Ytrain, euclideandistancestrain, label = "Euclidean train set", c = c[1
       ])
73 scatter!(Ytest, euclideandistancestest, label = "Euclidean test set",  c = c[2])
74 scatter!(Ytrain, traindistances, label = "NN train set",  c = c[3])
75 scatter!(Ytest, testdistances, label = "NN test set",  c = c[4])
```

Listing 12 *Code to generate Beziér curves.*

```
println("@@@ STARTING")

include("DistNN2D.jl")

using DistNN2D
using Plots
dnn2 = DistNN2D

### Create datasets
"""
Function to write to file easily. In filenamex, the input data is written.
In Xtrain, every row contains one input vector. First k x values, followed
by k y values. It starts at the first vertex + k, and puts writes those. Then
it skips k vertices etc.
"""
function writexydata(Xtrain :: Array{dnn2.Geometry, 1}, xfilename :: String,
     yfilename :: String)
    iox = open("../data/"*xfilename*".txt", "w")
    ioy = open("../data/"*yfilename*".txt", "w")
    for X in Xtrain
        for v = X.start_at:X.k:X.end_at
            x = X.V[X.knnmap[v]]
            middlex = repmat(X.V[v]', X.k*2+1)
            x = hcat(x...)' - middlex

            d1 = X.distance(X.LSe, v, X.knnmap[v][1])
            d2 = X.distance(X.LSe, v, X.knnmap[v][end])

            writedlm(iox, [x])
            writedlm(ioy, [d1 d2])
        end
    end
    close(iox)
    close(ioy)
end

""" Given a set of points and a value n, returns x and y values of n points
on the bezier curve (of order length of points array) defined by points. """
function bezier(pts :: Array{Array{Float64, 1},1}, n :: Int;
                t :: Array{Float64,1} = Array{Float64,1}(0))
    vals = zeros(n, 2)
    N = length(pts)-1
    if isempty(t) t = collect(linspace(0., 1., n)) end

    for j = 1:n
        l = 0.1
        for (i, p) in enumerate(pts)
            vals[j, :] += binomial(N, i-1) * (1-t[j])^(N-i+1) * t[j]^(i-1) * p
        end
    end
    vals[:, 1], vals[:, 2]
end

function unitbezier(pts :: Array{Array{Float64, 1},1}, n :: Int)
    x, y = bezier(pts, 100)
    L = 0.
    # find approximate length of curve
    for i = 1:100-1
        L += norm([x[i+1], y[i+1]] - [x[i], y[i]])
    end
    t = collect(linspace(0., 1., n))
    N = length(pts)-1
    if N == 3
        l = L/(n-1)
        for j = 2:n-1
            v1 = t[j-1]^2 * (-3.*pts[1] + 9.*pts[2] - 9.*pts[3] + 3.*pts[4])
            v2 = t[j-1] * (6.*pts[1] - 12.*pts[2] + 6.*pts[3])
            v3 = -3.*pts[1] + 3.*pts[2]
            t[j] = t[j-1] + l / norm(v1 + v2 + v3)
        end
        # Scale t to be between 0 and 1
        t /= maximum(t)
    end
    bezier(pts, n, t = t)
end

# Parameters
k = 10
ngv = k
nv = 40
spacing = 2.0 / nv


```

```
# Some possible shapes
# Lines
line = dnn2.Geometry(21, ngv, k, "line", 1.0);
x = collect(linspace(0.0, 4.0, 80))
sine05 = dnn2.Geometry(x, 0.5*sinpi.(x), ngv, k, "unconnected");
sine2 = dnn2.Geometry(x, sinpi.(x) + 3.0, ngv, k, "unconnected");
pts = [rand(2), rand(2), rand(2), rand(2)]
x1, y1 = bezier(pts, 51)
bezier1 = dnn2.Geometry(x1, y1, ngv, k, "unconnected")
x2, y2 = unitbezier(pts, 80)
bezier2 = dnn2.Geometry(x2, y2, ngv, k, "unconnected")


# Connected shapes
circle0 = dnn2.Geometry(40, 0, k, "circle", 1.*π);
circle1 = dnn2.Geometry(40, 0, k, "circle", 2.*π);
circle2 = dnn2.Geometry(40, 0, k, "circle", 4.*π);
circle3 = dnn2.Geometry(40, 0, k, "circle", 6.*π);
ellipse_1_025 = dnn2.Geometry(80, 0, k, "ellipse", [1.0, 0.25]);
xe3 = [2.0 * cospi(spacing * i) + 3.0 for i in 1:nv]
ye3 = [0.5 * sinpi(spacing * i) - 1.0 for i in 1:nv]
ellipse3 = dnn2.Geometry(xe3, ye3, 0, k, "connected");
R = [0.5 * i for i = 1:5]
circles = [dnn2.Geometry(40, 0, k, "circle", 2.0 * π * r) for r in R]


# Create train data from random bezier curves
btrain = Array{dnn2.Geometry,1}(0)
for i = 1:50000
    n = rand(20:80) # Random number of points, to get random lengths
    x, y = unitbezier([rand(2), rand(2), rand(2), rand(2)], n)
    push!(btrain, dnn2.Geometry(x, y, ngv, k, "unconnected"))
end

traindata = btrain
writexydata(traindata, "Xtrain_huge", "Ytrain_huge")

# Create test set

btest = Array{dnn2.Geometry,1}(0)
for i = 1:1000
    n = rand(20:80) # Random number of points, to get random lengths
    x, y = unitbezier([rand(2), rand(2), rand(2), rand(2)], n)
    push!(btest, dnn2.Geometry(x, y, ngv, k, "unconnected"))
end

testdata = btest
writexydata(testdata, "Xtest", "Ytest")

writexydata([circle0], "Xcircle_r05", "Ycircle_r05")
writexydata([circle1], "Xcircle_r1", "Ycircle_r1")
writexydata([circle2], "Xcircle_r2", "Ycircle_r2")
writexydata([circle3], "Xcircle_r3", "Ycircle_r3")
writexydata([ellipse_1_025], "Xellipse_1_025", "Yellipse_1_025")
writexydata([sine05], "Xsine05", "Ysine05")
```

Listing 13 *Neural Network for distance on curves main code.*

```
module DistNN2D

using Flux
using Flux.Tracker
using BSON: @load, @save
using StaticArrays
using Plots

const ε = 1e-10

const Scalar = Float64
const Index = Int
const TrReal = Flux.Tracker.TrackedReal{Scalar}

const ScalarList = Array{Scalar,1}
const MatList = Array{Array{Scalar, 2}, 1}
const IndexList = Array{Index,1}
const IndexMap = Array{IndexList, 1}

const Vec = SVector{2, Scalar}
const Edge = SVector{2, Index}
```

```
22
23  const EdgeList = Array{Edge,1}
24  const VectorList = Array{Vec,1}
25
26  """ Function to decide which shape vertices to use. """
27  function load_vertices(shape :: String, args...; kwargs...)
28      if shape == "line"
29          return load_vertices_line(args...; kwargs...)
30      elseif shape == "parabola"
31          return load_vertices_parabola(args...; kwargs...)
32      elseif shape == "circlearc"
33          return load_vertices_circlearc(args...; kwargs...)
34      elseif shape == "circle"
35          return load_vertices_circle(args...)
36      elseif shape == "square"
37          return load_vertices_square(args...)
38      elseif shape == "ellipse"
39          return load_vertices_ellipse(args...)
40      else
41          error("No such shape implemented.")
42      end
43  end
44
45  function load_vertices(x :: ScalarList, y :: ScalarList)
46      [Vec([x[i], y[i]]) for i in 1:length(x)]
47  end
48
49  """
50  Functions to create an array with vertex locations.
51  If the shape is unconnected, standard 10 ghost vertices are placed at both ends.
52  In the case of a line, if ngv is uneven, (ngv-1)/2 are placed on the left and
53  (ngv+1)/2 on the right. If the shape is connected at the first and last vertex,
54  L is the circumference. Also, the vertices can start at any point, but they
55  need to go anti-clockwise from there, due to the computation of the normal
56  vectors later on. Depending on the exact shape, the number of vertices
57  returned may vary from the number specified. If a unit speed parametrization is
58  easy to make, the length is L and vertices are evenly spaced.
59  """
60  function load_vertices_line(nv :: Index, L :: Scalar; ngv :: Index = 4)
61      n = nv + 2 * ngv
62      spacing = L / (nv - 1.0)
63      x = [(i-1)*spacing - spacing * fld(ngv, 2) for i in 1:n]
64      y = [0.0 for i in 1:n]
65      return x, y
66  end
67
68  function load_vertices_parabola(nv :: Index, L :: Scalar; ngv :: Index = 4)
69      n = nv + 2 * ngv
70      # Note that the length returned is > L
71      x = collect(linspace(- 0.5 * L, 0.5 * L, n))
72      y = x.^2
73      return x, y
74  end
75
76  function load_vertices_circlearc(nv :: Index, L :: Scalar; ngv :: Index = 4)
77      # cospi and sinpi calculate their arugments times π.
78      n = nv + 2 * ngv
79      spacing = 1.0 / nv
80      # factor (0.5 * L / π) to get a circle with circumference âĹĹ L.
81      x = [-(0.5 * L / π) * cospi(spacing * (i - 1 - ngv * 0.5)) for i in 1:n+1]
82      y = [ (0.5 * L / π) * sinpi(spacing * (i - 1 - ngv * 0.5)) for i in 1:n+1]
83      return x, y
84  end
85
86  function load_vertices_circle(nv :: Index, L :: Scalar)
87      # cospi and sinpi calculate their arugments times π.
88      spacing = 2.0 / nv
89      x = [(0.5 * L / π) * cospi(spacing * i) for i in 1:nv]
90      y = [(0.5 * L / π) * sinpi(spacing * i) for i in 1:nv]
91      return x, y
92  end
93
94  function load_vertices_ellipse(nv :: Index, L :: Scalar)
95      b = L*0.1
96      a = L*0.2
97      spacing = 2.0 / nv
98      x = [a * cospi(spacing * i) for i in 1:nv]
99      y = [b * sinpi(spacing * i) for i in 1:nv]
100     return x, y
101 end
102
103 function load_vertices_ellipse(nv :: Index, a :: Scalar, b :: Scalar)
104     spacing = 2.0 / nv
```

```
    x = [a * cospi(spacing * i) for i in 1:nv]
    y = [b * sinpi(spacing * i) for i in 1:nv]
    return x, y
end

function load_vertices_square(nv :: Index, L :: Scalar)
    # Creates four edges of the square, each with length L / 4
    l = L / 4.0
    # the total number of vertices returned will be 4 * floor(nv/4).
    n = fld(nv, 4)
    spacing = l / n
    x1 = [l / 2.0 - (i-1) * spacing  for i in 1:n]
    y1 = [l / 2.0 for i in 1:n]
    x2 = [- l / 2.0 for i in 1:n]
    y2 = [l / 2.0 - (i-1) * spacing for i in 1:n]
    x3 = [- l / 2.0 + (i-1) * spacing for i in 1:n]
    y3 = [- l / 2.0 for i in 1:n]
    x4 = [l / 2.0 for i in 1:n]
    y4 = [-l / 2.0 + (i-1) * spacing for i in 1:n]

    return vcat(x1, x2, x3, x4), vcat(y1, y2, y3, y4)
end

function make_shape(shape :: String; ngv :: Index = 4)
    if shape in ["connected", "circle", "square", "ellipse"]
        return ["connected", distance_connected, 0]
    elseif shape in ["unconnected", "line", "parabola", "circlearc"]
        return ["unconnected", distance_unconnected, ngv]
    else
        error("No such shape implemented.")
    end
end

""" Loads the edges, depending on the shapetype (connected or not). """
function load_edges(shapetype :: String, nv :: Index)
    E = [Edge([i, i+1]) for i in 1:nv-1]
    if shapetype == "unconnected"
        return E
    elseif shapetype == "connected"
        push!(E, Edge([nv, 1]))
        return E
    else
        error("No such shapetype implemented.")
    end
end

"""
Compute the edge maps: for each vertex, finds its two adjacent edges.
This function returns two maps: one with edges going out of
the vertex (clockwise) and one incident to the vertex
(anti-clockwise).
"""
function build_edge_maps(nv :: Index, E :: EdgeList, shapetype :: String)
    emap_pos = IndexList()
    emap_neg = IndexList()

    for e in E
        push!(emap_pos, e[2])
        push!(emap_neg, e[1]-1)
    end

    if shapetype == "connected"
        emap_neg[1] = E[end][1]
    elseif shapetype == "unconnected"
        # The fist point sees itself as previous element, and the last one
        # itself as the next element.
        emap_neg[1] = 1
        push!(emap_neg, nv - 1)
        push!(emap_pos, nv)
    else
        error("No such shapetype implemented.")
    end

    emap_pos, emap_neg
end

"""
Function to find the neighbors of vertex v in the right order and return a list
of lists with neighbors.
"""
function knn(k :: Index, nv :: Index, shapetype :: String; ngv :: Index = 4)
    if shapetype == "connected"
        return knn_connected(k, nv)
```

```
    elseif shapetype == "unconnected"
        return knn_unconnected(k, nv; ngv = ngv)
    else
        error("No such shapetype.")
    end
end

function knn_connected(k :: Index, nv :: Index)
    knnmap = [IndexList() for k = 1:nv]
    for v = 1:k
        knnmap[v] = [(nv-1 + v - l) % nv + 1 for l = k:-1:1]
        push!(knnmap[v], v)
        append!(knnmap[v], [(nv+1 + v + l) % (nv+1) for l = 1:k])
    end
    for v = k+1:nv-k
        knnmap[v] = [(nv+1 + v - l) % (nv+1) for l = k:-1:1]
        push!(knnmap[v], v)
        append!(knnmap[v], [(nv+1 + v + l) % (nv+1) for l = 1:k])
    end
    for v = (nv-k):nv
        knnmap[v] = [(nv+1 + v - l) % (nv+1) for l = k:-1:1]
        push!(knnmap[v], v)
        append!(knnmap[v], [(nv-1 + v + l) % nv + 1 for l = 1:k])
    end
    knnmap
end

function knn_unconnected(k :: Index, nv :: Index; ngv :: Index = 4)
    knnmap = [IndexList() for k = 1:nv]
    n = cld(ngv, 2)
    for v = (n+1):nv-n
        knnmap[v] = [(nv-1 + v - l) % nv + 1 for l = k:-1:1]
        push!(knnmap[v], v)
        append!(knnmap[v], [(nv+1 + v + l) % (nv+1) for l = 1:k])
    end
    knnmap
end

"""
Struct containing all relevant information of a mesh during and after inflation,
e.g., material parameters, vertex locations, number of vertices, density, etc.
Also contains the function to initialize the struct.
"""
mutable struct Geometry
    # Shape and distance function
    shapetype :: String
    distance :: Function

    # Number of (ghost)vertices and edges
    ngv :: Index
    nv :: Index
    ne :: Index

    # Index of first and last moving particle
    start_at :: Index
    end_at ::Index

    # Convenient representation of the mesh
    V :: VectorList
    E :: EdgeList

    # Lookup tables
    emap_pos :: IndexList
    emap_neg :: IndexList
    knnmap :: IndexMap

    #
    # Derived geometric quantities (normals, areas, lengths, cot of angles)
    #

    Te :: VectorList
    Le :: ScalarList
    LSe :: ScalarList

    #
    # Dynamic parameters
    #
    args :: Union{Scalar, ScalarList} # Shape parameters
    k :: Index                        # Number of neighbors

    # Initialisation functions
    function Geometry(
        x :: ScalarList,
```

```
        y :: ScalarList,
        ngv :: Index,
        k :: Index,
        shapetype :: String,
        )

        if length(x) âĹă length(y)
            error("x and y should be the same length")
        end

        self = new()

        self.shapetype, self.distance, self.ngv = make_shape(
            shapetype, ngv = ngv)

        # Load mesh
        self.V = load_vertices(x, y)
        self.nv = length(self.V)

        self.E = load_edges(self.shapetype, self.nv)
        self.ne = length(self.E)
        self.k = k

        # Indices of first and last interesting vertex; do not have to loop
        # over ghost vertices. For connected shapes, this reduces to
        # v = 1:d.nv, since d.ngv = 0.
        self.start_at = self.ngv + 1
        self.end_at = self.nv - self.ngv

        # Create look up tables
        self.emap_pos, self.emap_neg = build_edge_maps(
            self.nv, self.E, self.shapetype)
        self.knnmap = knn(self.k, self.nv, self.shapetype)

        # Pre-allocate space for derived gemoetrical information
        self.Te = VectorList(self.nv)        # edge tangentials
        self.Le = ScalarList(self.ne)        # edge lengths
        self.LSe = ScalarList(self.ne)       # edge lengths cumulative sum

        # Compute derived geometric information
        update_edges!(self)

        return self
    end

    function Geometry(
        nv :: Index,
        ngv :: Index,
        k :: Index,
        shape :: String,
        args,
        )
        x, y = load_vertices(shape, nv, args...; ngv = ngv)
        shapetype, _, _ = make_shape(shape, ngv = ngv)
        self = Geometry(x, y, ngv, k, shapetype)

        return self
    end
end

""" Updates the edge normals, areas, tangentials and lengths. """
function update_edges!(d :: Geometry)
    for (k, e) in enumerate(d.E)
        v = d.V[e[1]]
        w = d.V[e[2]]

        # Edge tangentials and lengths
        d.Te[k] = v - w
        d.Le[k] = norm(Tracker.data(d.Te[k]))
        # Add 0.0 at the front, so that LSe[end] - LSe[1] = LSe[end]
        d.LSe = unshift!(cumsum(d.Le), 0.0)
    end
end

"""
Distance between two points on the line, given by the absolute value of the
difference of the values at v and ov in the list containing the cumulative sums
of the distances. For shapes of which the first and last vertex are connected,
this function assumes that the last entry of LS contains circumference
of the shape.
"""
function distance_connected(LS :: ScalarList, v :: Index, ov :: Index)
    Flux.Tracker.data(min(abs(LS[v] - LS[ov]), LS[end] - abs(LS[v] - LS[ov])))
```

```
354 end
355
356 function distance_unconnected(LS :: ScalarList, v :: Index, ov :: Index)
357     Flux.Tracker.data(abs(LS[v] - LS[ov]))
358 end
359
360 mutable struct Model
361     n_dense :: Index
362     n_conv :: Index
363     λ :: Scalar
364     NN :: Chain
365     k :: Index
366     af :: Function
367
368     function Model(n_dense :: Index, n_conv :: Index, λ :: Scalar,
369             k :: Index, af :: Function)
370         self = new()
371         self.n_dense = n_dense         # Number of units in hidden dense layer
372         self.n_conv = n_conv           # Size of filter in conv layer
373         self.λ = λ                     # Learning rate
374         self.k = k                     # Total number of input points
375         self.af = af                   # Activation function
376         self.NN = Chain(               # Neural network architecture
377             Dense(k, n_dense, af),
378             x -> reshape(x, (size(x)...,1,1)),
379             Conv((n_conv,2), 1=>1, af),
380             x -> squeeze(x, (2,3,4)),
381             Dense(n_dense - n_conv + 1, 1, identity))
382         self
383     end
384 end
385
386 """ Loss function """
387 function loss(X, Y, model :: Model)
388     loss = 0.0
389     n = length(X)
390     if n âĽă length(Y); error("DimensionMismatch in test and train data") end
391     for i = 1:n
392         loss += Flux.mse(model.NN(X[i]), Y[i])
393     end
394     loss / n
395 end
396
397 """
398 Function to train on given data, which is assumed to be a list of matrices.
399 Also, evaluates every epoch whether the loss was NaN or equal to the previous
400 loss. Returns arrays with train and test loss values.
401 """
402 function train!(Xtrain :: MatList, Ytrain :: ScalarList, Xtest :: MatList,
403                 Ytest :: ScalarList, model :: Model, epochs :: Index,
404                 batchsize :: Index, opt; verbose = 0.1)
405     Xset, Yset, n_batches = batches_size(Xtrain, Ytrain, batchsize)
406     trainloss = zeros(n_batches, epochs)
407     testloss = zeros(n_batches, epochs)
408
409     for i = 1:epochs
410         # Train the network per batch, batches are shuffled between epochs
411         for (k,j) in enumerate(shuffle(1:length(Xset)))
412             args = (Xset[j], Yset[j], Xtest, Ytest, model, opt)
413             trloss, teloss = train!(args...)
414             trainloss[k,i] = trloss
415             testloss[k,i] = teloss
416         end
417
418         #  Test for NaNs and network not learning at the start of training
419         if isnan(trainloss[end, i])
420             printloss(i, trainloss[end, i], testloss[end, i])
421             break
422         end
423         if i == 4 && trainloss[i-2] == trainloss[i-1] == trainloss[i]
424             println("Network not learning, quitting training and returning
                    losses.")
425             printloss(i, trainloss[i], testloss[i])
426             break
427         end
428
429         # Print losses
430         if i % ceil(Int, epochs*verbose) == 0
431             printloss(i, trainloss[end, i], testloss[end, i])
432         end
433     end
434     trainloss, testloss
435 end
```

```
436
437 """ Train a batch and return the train and test loss. """
438 function train!(Xtrain :: MatList, Ytrain :: ScalarList, Xtest :: MatList,
439                 Ytest :: ScalarList, model :: Model, opt)
440
441     Flux.back!(loss(Xtrain, Ytrain, model)) # Calculate gradients
442     opt()                                   # Update weights
443
444     # Loss values
445     trainloss = Flux.Tracker.data(loss(Xtrain, Ytrain, model))
446     testloss = Flux.Tracker.data(loss(Xtest, Ytest, model))
447
448     trainloss, testloss
449 end
450
451 printloss(i :: Index, trainloss, testloss) =
452     println("Epoch $i, train loss is $trainloss, and test loss is $testloss")
453
454 """ For a given array of input points, returns the model's predictions """
455 function evaluate(X :: MatList, model :: Model)
456     ymodel = ScalarList(length(X))
457     for (k, x) in enumerate(X)
458         ymodel[k] = Flux.Tracker.data(model.NN(x))[1]
459     end
460     ymodel
461 end
462
463 # Functions to handle data sets
464 """ Splits the data in the given amount of batches or batchsizes. If the length
465 of the given list of matrices is not divisible by the number of batches or
466 the batch size, the returned list contains more batches and/or the last batch
467 is smaller than the rest. """
468 function batches_number(X :: MatList, Y :: ScalarList, nb :: Index)
469     N = length(X)
470     s = fld(N, nb)
471     r = 1:s:N
472     n_batches = length(r)
473     Xset = Array{MatList}(n_batches)
474     Yset = Array{ScalarList}(n_batches)
475
476     for (k,i) in enumerate(r)
477         if i == r[end]
478             Xset[k] = X[i:end]
479             Yset[k] = Y[i:end]
480         else
481             Xset[k] = X[i:i+s-1]
482             Yset[k] = Y[i:i+s-1]
483         end
484     end
485     Xset, Yset, n_batches
486 end
487
488 function batches_size(X :: MatList, Y :: ScalarList, s :: Index)
489     N = length(X)
490     r = 1:s:N
491     n_batches = length(r)
492     Xset = Array{MatList}(n_batches)
493     Yset = Array{ScalarList}(n_batches)
494
495     for (k,i) in enumerate(r)
496         if i == r[end]
497             Xset[k] = X[i:end]
498             Yset[k] = Y[i:end]
499         else
500             Xset[k] = X[i:i+s-1]
501             Yset[k] = Y[i:i+s-1]
502         end
503     end
504     Xset, Yset, n_batches
505 end
506
507 """
508 Permutes the given dataset. Can give a randomseed as input value, to be able to
509 reproduce results.
510 """
511 function permutations(X :: MatList, Y :: ScalarList;
512     randomseed :: Union{Void, Int} = nothing)
513     if randomseed ãĽă nothing
514         srand(randomseed)
515     end
516     p = randperm(length(X))
517     X[p], Y[p]
518 end
```

```
519
520 """ Change matrix to list of matrices. """
521 function Xtomatrices(X :: Array{Float64, 2})
522     n = Int(size(X, 1) / 2)
523     m = size(X, 2)
524     Xarray = MatList(m)
525     for j = 1:m
526         mat = zeros(n, 2)
527         mat[:, 1] = X[1:n, j]
528         mat[:, 2] = X[n+1:end, j]
529         Xarray[j] = mat
530     end
531     Xarray
532 end
533
534 """ Change matrix to vector. """
535 function Ytovectors(Y :: Array{Float64, 2})
536     n = size(Y, 1)
537     Yvec = ScalarList(n)
538     for i = 1:n
539         Yvec[i] = Y[i, 1] + Y[i, 2]
540     end
541     Yvec
542 end
543
544 """ Read data from file, change type and return shuffled. """
545 function readandshuffle(xname :: String, yname :: String;
546     randomseed :: Union{Void, Int} = nothing)
547     # Read
548     Xtraindata = readdlm("../data/"*xname*".txt")';
549     Ytraindata = readdlm("../data/"*yname*".txt");
550     # Convert to the right kind of data
551     Xtrain = Xtomatrices(Xtraindata)
552     Ytrain = Ytovectors(Ytraindata)
553     # Shuffle
554     permutations(Xtrain, Ytrain, randomseed = randomseed)
555 end
556
557 """ Functions to load weights of a model """
558 function loadweights(filename :: String, model :: Model)
559     @load filename*".bson" weights
560     Flux.loadparams!(model.NN, weights)
561 end
562
563 function loadweights(filename :: String, model :: Chain)
564     @load filename*".bson" weights
565     Flux.loadparams!(model, weights)
566 end
567
568 """ Functions to save weights of a model """
569 function saveweights(filename :: String, model :: Model)
570     weights = Tracker.data.(params(model.NN));
571     @save filename*".bson" weights
572 end
573
574 function saveweights(filename :: String, model :: Chain)
575     weights = Tracker.data.(params(model));
576     @save filename*".bson" weights
577 end
578
579  """ Functions to plot the shape. """
580  # May return MethodError: no method matching setcharheight(::Float64) the first
581  # time it is run. Running it again does show the plot.
582 function showplot(X :: Array{Scalar, 1}, Y :: Array{Scalar, 1})
583     p = plot(X, Y, shape = :circle,
584         # Options if you only want markers, without borders and no lines
585         # linewidth = 0,
586         # markersize = 2,
587         # markerstrokealpha = 0
588         label = "Vertex locations",
589         aspect_ratio = :equal,
590         # xlim=(-2,2), ylim = (-2, 2),
591         legend = :topright)
592     p
593 end
594
595  function showplot(V :: VectorList)
596      X = [Flux.Tracker.data(v[1]) for v in V]
597      Y = [Flux.Tracker.data(v[2]) for v in V]
598      p = showplot(X, Y)
599      display(p)
600  end
601
```

```
function showplot(Vlist :: Array{VectorList, 1})
    p = plot()
    for V in Vlist
        X = [Flux.Tracker.data(v[1]) for v in V]
        Y = [Flux.Tracker.data(v[2]) for v in V]

        plot!(X, Y, shape = :circle,
            # Options if you only want markers, without borders and no lines
            # linewidth = 0,
            # markersize = 2,
            # markerstrokealpha = 0
            label = "Vertex locations",
            aspect_ratio = :equal,
            # xlim=(-2,2), ylim = (-2, 2),
            legend = :topright)
       end
   display(p)
end

function showplot(Geomlist :: Array{Geometry, 1})
    p = plot()
    for geom in Geomlist
        X = [Flux.Tracker.data(v[1]) for v in geom.V]
        Y = [Flux.Tracker.data(v[2]) for v in geom.V]

        plot!(X, Y, shape = :circle,
            # Options if you only want markers, without borders and no lines
            # linewidth = 0,
            # markersize = 2,
            # markerstrokealpha = 0
            label = "Vertex locations",
            aspect_ratio = :equal,
            # xlim=(-2,2), ylim = (-2, 2),
            legend = :topright)
       end
   display(p)
end

function plotdata(X :: Array{Matrix{Scalar}, 1})
    p = plot(legend = false)
    n = length(X)
    for i = 1:n
        x = X[i][:,1]
        y = X[i][:,2]
        scatter!(x, y, aspect_ratio = :equal)
    end
    display(p)
end

end # module DistNN2D
```

C.3 3D Code

Listing 14 *Demo file using DistNN3D.jl.*

```
println("@@@ STARTING")

include("DistNN3D.jl")

using DistNN3D

using Flux
using Plots
dnn = DistNN3D # alias

#
### Read and shuffle test and train datasets
#
rs = nothing

Xdata, Ydata = dnn.readandshuffle("Xbsurfs", "Ybsurfs", randomseed = rs);
Xset, Yset = dnn.permutations(Xdata[1:5000], Ydata[1:5000], randomseed = rs);

# Split data in test and train data (20-80)
p = 0.2
n = floor(Int, length(Xset) * p)
Xtest = Xset[1:n];
Ytest = Yset[1:n];
```

```
Xtrain = Xset[n+1:end];
Ytrain = Yset[n+1:end];

#
### Make NN model
#

# NN parameters
λ = 0.01             # Learning rate
n_dense1 = 30        # Number of units in 1st dense layer
n_dense2 = 50        # Number of units in 2nd dense layer
n_conv1 = 5          # Length of 1st convolutional filter
n_conv2 = 5          # Length of 2nd convolutional filter
k = 50               # Number of input points
af = relu            # Activation function, (leaky)relu, elu, sigmoid, or swish
model = dnn.Model(n_dense1, n_dense2, n_conv1, n_conv2, λ, k, af);
opt = ADAM(Flux.params(model.NN), model.λ)

# Load/save weights
# dnn.loadweights("dnn3d_", model)
# dnn.saveweights("dnn3d_001_30_50_5_5", model)

#
### Train NN
#

epochs = 5
batchsize = 200

args = (Xtrain, Ytrain, Xtest, Ytest, model, epochs, batchsize, opt);
@time trainloss, testloss = dnn.train!(args..., verbose = 0.01);

#
### Plot results
#

plot(trainloss[:], label = "Train loss", xlabel = "Number of batches.")
plot!(testloss[:], label = "Test loss", ylabel = "Loss value")

# savefig("testtrainloss3D.pdf")

nd1 =[10, 20, 30, 40, 50]

plot(xlabel = "Number of neurons in first dense layer", ylabel = "Loss value")
plot!(nd1, trainlossnd1, label = "Train loss")
plot!(nd1, testlossnd1, label = "Test loss")

# savefig("nd13Dloss.pdf")

# Convenient function to plot model prediction, true value and Euclidean norm
function compareandplot(v :: Int; X :: dnn.MatList = Xtest,
        Y :: dnn.ScalarMap = Ytest)
    ymodel = Flux.Tracker.data(model.NN(X[v]))
    line = norm(X[v][1,1:3] - X[v][2,1:3])
    println("Model: ", ymodel, ", true value:", Y[v], " norm: ", line)
    dnn.showplot(X[v][:,1:3])
end

function compareresults(X :: dnn.MatList, Y :: dnn.ScalarMap, model :: dnn.Model
    )
    nv = length(X)
    d_euclidean = zeros(nv)
    d_network = zeros(nv)
    d_true = zeros(nv)
    for i = 1:nv
        v1 = X[i][1,1:3]
        v2 = X[i][2,1:3]
        d_euclidean[i] = norm(v1 - v2)
        d_network[i] = Flux.Tracker.data(model.NN(X[i]))[1]
        d_true[i] = Y[i][1]
    end
    m = maximum(max.(d_network, d_true, d_euclidean))
    plot([0,m], [0,m], label="Target", xlabel = "True distance",
    ylabel = "Prediction", legend=:bottomright, c=:lightgray)

    scatter!(d_true, d_network, label = "NN distance", markerstrokewidth = .5,
    c = :deepskyblue)
    scatter!(d_true, d_euclidean, label = "Euclidean distance", c = :orange,
    markerstrokewidth = .5)
end
```

Listing 15 *Code to generate Beziér surfaces.*

```
println("@@@ STARTING")

include("DistNN3D.jl")

using DistNN3D
using FileIO
dnn3 = DistNN3D # alias

#
### Part to make data
#

# Read vertices and faces from .txt file
sphere = dnn3.Geometry("./data/sphere", zerobased = false);
piramide = dnn3.Geometry("./data/piramide", zerobased = false);
plane = dnn3.Geometry("./data/plane", zerobased = false);

# Or from other filetypes
spot = load("./data/spot_triangulated.obj")
V = [dnn3.Vec(vertex[1], vertex[3], vertex[2]) for vertex in spot.vertices]
F = [dnn3.Face(face) for face in spot.faces]
spot = dnn3.Geometry(V, F)

### Create datasets with Bezier surfaces
scale = 1.0
cpts = dnn3.controlpoints(scale, randomseed = nothing)
T = collect(linspace(0., 1., 20))
S = collect(linspace(0., 1., 20))

bpoints = dnn3.beziersurface(T, S, cpts)
bfaces = dnn3.makefacelist(length(T), length(S))
bpoints = [dnn3.Vec(b) for b in bpoints[:]]
bsurf = dnn3.Geometry(bpoints, bfaces)

bsurf = dnn3.makebeziersurface(T, S, cpts)
bsurfs = dnn3.makerandombeziersurfaces(3, 15)

#
### Part to show what can be done with the data
#

# Plot
dnn3.showplot(sphere.V, sphere.F);
dnn3.showplot(bsurfs[3].V, bsurfs[3].F)

# Compute distances to point v
v = 1  # the point to which we want distances.
dists = dnn3.geodesicdistance(spot, [v])

# Plot distance with isolines.
dnn3.plotdistance(spot, 60)

# Make submesh of at least size minpoints
minpoints = 50
subspot = dnn3.makesubmesh(spot, v, minpoints = minpoints);
subbsurf = dnn3.makesubmesh(bsurf, v, minpoints = minpoints);

dnn3.showplot(subspot[1:2]...)
dnn3.showplot(subbsurf[1:2]...)

# Make the submesh a certain size
V, F, U = subspot;
resV, resF, resU = dnn3.resizemesh(V, F, U, 50);
dnn3.showplot(resV, resF)

# Make a Geometry from a submesh and plot it
submesh = dnn3.Geometry(points, faces)
dnn3.showplot(submesh.V, submesh.F)

#
### Part to write data
#

# Write data to file, such that it can be used for training a NN
dnn3.writexydata([sphere], "Xsphere", "Ysphere", npoints = 50)
dnn3.writexydata([spot], "Xspot", "Yspot", npoints = 50)
dnn3.writexydata([piramide], "Xpiramide", "Ypiramide", npoints = 50)
dnn3.writexydata([plane], "Xplane", "Yplane", npoints = 50)

bsurfs = dnn3.makerandombeziersurfaces(200, 15)
dnn3.writexydata(bsurfs, "Xbsurfs", "Ybsurfs", npoints = 50)
```

Listing 16 *Neural Network for distance on surfaces main code.*

```
module DistNN3D

using StaticArrays
using Flux
using Flux.Tracker
using BSON: @load, @save
using SimpleWeightedGraphs
using LightGraphs
using PyPlot

const Scalar = Float64
const Index = Int

const Vec = SVector{3, Scalar}
const Edge = SVector{2, Index}
const Face = SVector{3, Index}

const SparMat = SparseMatrixCSC{Scalar, Int64}
const Mat = SMatrix{3,3,Scalar,9}
const MMat = MMatrix{3,3,Scalar,9}
const MatList = Array{SMatrix{50,53,Float64,2650}, 1}

const ScalarList = Array{Scalar,1}
const VectorList = Array{Vec,1}
const EdgeList = Array{Edge,1}
const FaceList = Array{Face,1}
const VectorMap = Array{VectorList,1}

const IndexList = Array{Index,1}
const IndexMap = Array{IndexList,1}
const ScalarMap = Array{ScalarList,1}

#
### Data part
#

""" Lexical ordering of edges. """
function edge_lt(e1 :: Edge, e2 :: Edge)
    e1[1] < e2[1] || (e1[1] == e2[1] && e1[2] < e2[2])
end

""" Load vertices from a file. """
function load_vertices(filename :: String)
    # Read as matrix
    Vmat = readdlm(filename)
    # Convert to VectorList
    [Vec(Vmat[i,:]) for i in 1:size(Vmat,1)]
end

""" Load faces from a file. Assumes faces are given zero indexed. """
function load_faces(filename :: String; zerobased :: Bool = true)
    # Read as matrix and convert to 1 based numbering if needed
    Fmat = readdlm(filename) + 1 * zerobased
    # Convert to FaceList
    [Face(Fmat[i,:]) for i in 1:size(Fmat,1)]
end

""" Orders an edge, such that the smallest element comes first. """
ordered_edge(v1 :: Index, v2 :: Index) = Edge(minmax(v1, v2)...)

"""
Extract all edges to a list from a list of Faces. Returns a list of Edges.
"""
function extract_edges(F :: FaceList)
    E = EdgeList()
    for f in F
        push!(E, ordered_edge(f[1], f[2]))
        push!(E, ordered_edge(f[2], f[3]))
        push!(E, ordered_edge(f[3], f[1]))
    end
    # Return without duplicates
    unique(sort(E, lt=edge_lt), 1)
end

"""
Create an array of arrays containing at index v the indices of the Faces
adjacent to v.
"""
function build_face_map(nv :: Index, F :: FaceList)
    fmap = [IndexList() for k in 1:nv]

```

```
    for (k, f) in enumerate(F)
        push!(fmap[f[1]], k)
        push!(fmap[f[2]], k)
        push!(fmap[f[3]], k)
    end
    fmap
end

""" Maximum in a list of SArray """
function facemaximum(F :: FaceList)
    m = 0
    for f in F
        if maximum(f) > m
            m = maximum(f)
        end
    end
    m
end

""" Bernstein polynomials """
Bern1(t) = (1-t)^3
Bern2(t) = 3. * t * (1-t)^2
Bern3(t) = 3. * t^2 * (1-t)
Bern4(t) = t^3

""" For one value t, compute a point on a bezier curve. """
function evalbeziercurve(t :: Float64, cp :: Array{Array{Float64,1},1})
    cp[1] * Bern1(t) + cp[2] * Bern2(t) + cp[3] * Bern3(t) + cp[4] * Bern4(t)
end

""" For one pair (t,s), compute a point on a bezier surface. """
function evalbeziersurface(t :: Float64, s :: Float64, cp :: Array{Array{Float64
    ,1},2})
    curvepts = Array{Array{Float64, 1},1}(4)
    for i = 1:4
        curvepts[i] = evalbeziercurve(t, cp[i, :])
    end
    evalbeziercurve(s, curvepts)
end

""" For a given list parametrization values (T and S), and a set of 16
control points, return a bezier surface. """
function beziersurface(T :: ScalarList, S :: ScalarList, cp :: Array{Array{
    Float64,1},2})
    surfpts = Array{Array{Float64,1}, 2}(length(T), length(S))
    for (l, t) in enumerate(T), (k, s) in enumerate(S)
        surfpts[l, k] = evalbeziersurface(t, s, cp)
    end
    surfpts
end

""" To make 16 controlpoints that are placed in a 4x4 grid, with x and y values
not overlapping. """
function controlpoints(scale :: Scalar; randomseed :: Union{Void, Int} = nothing
    )
    if randomseed âĽă nothing
        srand(randomseed)
    end
    cpts = Array{Array{Float64, 1}, 2}(4, 4)
    for j = 1:4, i = 1:4
        x = rand() / 4. + (i - 1) / 4.
        y = rand() / 4. + (j - 1) / 4.
        cpts[i,j] = scale * [x, y, rand()]
    end
    cpts
end

""" Makes a list of faces, given the amount of x and y points. """
function makefacelist(n :: Int, m :: Int)
    # In the rectangular grid, there are (n-1)(m-1) squares, so twice as many
    # triangles
    k = 2 * (n - 1) * (m - 1)
    F = FaceList(k)
    sizes = (n, m)
    for j = 1:m-1, i = 1:n-1
        # Index of the face
        index = 2*sub2ind((n-1, m-1), i, j)
        # Indices of the triangles
        i1 = sub2ind(sizes, i, j)
        i2 = sub2ind(sizes, i+1, j)
        i3 = sub2ind(sizes, i, j+1)
        i4 = sub2ind(sizes, i+1, j+1)

```

```
        # Choose randomly which two triangles to make
        if rand(1:2) == 1
            F[index - 1] = Face([i1, i2, i3])
            F[index] = Face([i3, i2, i4])
        else
            F[index - 1] = Face([i1, i4, i3])
            F[index] = Face([i1, i2, i4])
        end
    end
    F
end

""" Returns a geometry of a bezier surface. """
function makebeziersurface(T :: ScalarList, S :: ScalarList,
                           cpts :: Matrix{Array{Float64,1}})
    bpoints = beziersurface(T, S, cpts)
    bfaces = makefacelist(length(T), length(S))
    bpoints = [Vec(b) for b in bpoints[:]]
    Geometry(bpoints, bfaces)
end

""" Return a list of geometries of bezier surfaces in [0,s]ÃÛ[0,s]ÃÛ[0,s] with
s âĹĹ [0.1, 10]. """
function makerandombeziersurfaces(nsurfs :: Int, npoints :: Int;
        randomseed :: Union{Void, Int} = nothing)
    if randomseed âĹă nothing
        srand(randomseed)
    end
    T = collect(linspace(0., 1., npoints))
    S = collect(linspace(0., 1., npoints))
    geomlist = Array{Geometry, 1}(nsurfs)
    for i = 1:nsurfs
        cpts = controlpoints(rand(5.:15.))
        geomlist[i] = makebeziersurface(T, S, cpts)
    end
    geomlist
end

function edgeinface(edge :: Edge, face :: Face)
    if length(setdiff(face, edge)) == 1
        return true
    else
        return false
    end
end

function findboundaryedges(emap :: IndexMap)
    inds = Int[]
    for (i, facelist) in enumerate(emap)
        if length(facelist) == 1
            push!(inds, i)
        end
    end
    inds
end


""" Returns a list with at index i, the face(s) that edge i is in. """
function build_edge_map(E :: EdgeList, F :: FaceList)
    edgemap =[IndexList() for i = 1:length(E)]
    for (l, face) in enumerate(F)
        edges = EdgeList(3)
        edges[1] = ordered_edge(face[1], face[2])
        edges[2] = ordered_edge(face[2], face[3])
        edges[3] = ordered_edge(face[1], face[3])

        for edge in edges
            if edgeinface(edge, face)
                ind = find(x-> x == edge, E)[1]
                push!(edgemap[ind], l)
            end
        end
    end
    edgemap
end

"""
Struct containing all relevant geometrical information of a mesh, e.g., vertex
locations, number of vertices, edge lengths, normals, etc. Also contains the
function to initialize the struct.
"""
mutable struct Geometry

```

```
    # Number of vertices, faces and edges
    nv :: Index
    nf :: Index
    ne :: Index

    # Convenient representation of the mesh
    V :: VectorList
    F :: FaceList
    E :: EdgeList

    # Lookup table
    fmap :: IndexMap

    # Geometrical information
    Te :: VectorList
    Le :: ScalarList

    Nf :: VectorList
    Af :: ScalarList

    Nv :: VectorList
    Av :: ScalarList

    #
    # Necessary information needed for geodesic distance
    #

    A :: SparMat
    An :: SparMat
    cots :: Matrix{Scalar}
    nLC :: SparMat

    m :: Scalar
    lfac :: SparMat
    fact_nLC :: Base.SparseArrays.CHOLMOD.Factor{Float64}

    gradu :: VectorList
    divx :: ScalarList

    # Initialize all quantities
    function Geometry(filename :: String; zerobased :: Bool = true)
        # Load mesh from file
        V = load_vertices(filename * "-V.txt")
        F = load_faces(filename * "-F.txt", zerobased = zerobased)

        Geometry(V, F)
    end

    function Geometry(V :: VectorList, F :: Matrix{Index})
        Fr = rename(F)
        F = [Face(Fr[i,:]) for i=1:size(Fr,1)]

        Geometry(V, F)
    end

    function Geometry(V :: VectorList, F :: FaceList)
        self = new()

        self.V = V
        self.nv = length(self.V)

        if facemaximum(F) > self.nv
            F = rename(F)
        end

        self.F = F
        self.nf = length(self.F)

        self.E = extract_edges(self.F)
        self.ne = length(self.E)

        # Create look up table
        self.fmap = build_face_map(self.nv, self.F)

        # Pre-allocate space for geometrical information
        self.Te = VectorList(self.ne) # edge tangentials
        self.Le = ScalarList(self.ne) # edge lengths

        self.Nf = VectorList(self.nf) # face normals
        self.Af = ScalarList(self.nf) # Face areas

        self.Nv = VectorList(self.nv) # vertex normals
        self.Av = ScalarList(self.nv) # vertex areas
```

```
328
329         # Pre-allocate space for geodesic distance
330         self.A = spzeros(self.nv, self.nv)    # cotangents of angles
331         self.An = spzeros(self.nv, self.nv)   # mass diagonal materix
332         self.cots = zeros(self.nf, 3)         # cotangents of angles per face
333         self.nLC = spzeros(self.nv, self.nv)  # negative laplace opterator
334         self.lfac = spzeros(self.nv, self.nv) # Backward Euler matrix
335         self.gradu = VectorList(self.nf)      # gradient of heat u
336         self.divx = ScalarList(self.nf)       # integrated divergence of u
337         self.m = 0.0
338
339         # Compute lengths, cotangents etc.
340         update_geometry!(self)
341
342         return self
343     end
344 end
345
346 function update_geometry!(g :: Geometry)
347     update_edges!(g)
348     update_faces!(g)
349     update_vertices!(g)
350     update_cotangents!(g)
351 end
352
353 """ Updates the edge tangentials and lengths. """
354 function update_edges!(g :: Geometry)
355     for (k, e) in enumerate(g.E)
356         g.Te[k] = g.V[e[2]] - g.V[e[1]]
357         g.Le[k] = norm(g.Te[k])
358     end
359 end
360
361 """ Updates the face normals and areas. """
362 function update_faces!(g :: Geometry)
363     for (k, f) in enumerate(g.F)
364         v :: Vec = g.V[f[2]] - g.V[f[1]]
365         w :: Vec = g.V[f[3]] - g.V[f[1]]
366
367         g.Nf[k] = cross(v, w)
368
369         mag = norm(g.Nf[k])
370         g.Nf[k] /= mag
371         g.Af[k] = 0.5 * mag
372     end
373 end
374
375 """ Updates vertex normals and areas. """
376 function update_vertices!(g :: Geometry)
377     for k = 1:g.nv
378         Ns = Vec(0.0, 0.0, 0.0)
379         As = 0.0
380
381         for f in g.fmap[k]
382             As += g.Af[f]
383             Ns += g.Af[f] * g.Nf[f]
384         end
385
386         g.Av[k] = As / 3.0
387         g.Nv[k] = Ns / norm(Ns)
388     end
389 end
390
391 """
392 Updates the values of the cotangents of the angles between vectors and the
393 matrix used to normalize it.
394 """
395 function update_cotangents!(g :: Geometry)
396     for (i, f) in enumerate(g.F)
397         g.cots[i, 1] = cot(g.V[f[2]] - g.V[f[1]], g.V[f[3]] - g.V[f[1]])
398         g.cots[i, 2] = cot(g.V[f[3]] - g.V[f[2]], g.V[f[1]] - g.V[f[2]])
399         g.cots[i, 3] = cot(g.V[f[1]] - g.V[f[3]], g.V[f[2]] - g.V[f[3]])
400     end
401 end
402
403 """ Cotangent of angle between two vectors. """
404 cot(a :: Vec, b :: Vec) =  dot(a,b) / norm(cross(a,b))
405
406 """
407 Given a geometry, returns a subset of the points and faces which certainly
408 contains vertex v. Faces are complete, there are no points in a face that are
409 not returned. Returns the vertices and faces of the submesh, as wel as the list
410 of indices of the vertices in the original mesh.
```

```
"""
function makesubmesh(g :: Geometry, v :: Index; minpoints :: Index = 50)
    # Return the mesh if it already has the right size, or if it is smaller
    if g.nv â‰¥ minpoints
        return g.V, g.F, []
    end
    W = [v]
    U = [v]
    F = IndexMap(0)
    while length(U) < minpoints
        faces = g.fmap[W]            # Look up ajacent faces
        append!(F, faces)            # Add to list of all used faces
        pts = g.F[vcat(faces...)]    # Find the points of these faces
        W = setdiff(vcat(pts...), U) # Add new points to W, for next loop
        U = append!(U, W)
    end
    U = sort(U)
    T = IndexList(0)
    for f in F append!(T, f) end     # Make list of triangles
    g.V[U], g.F[unique(T)], U
end

"""
Given a geometry, returns a subset of the points and faces which certainly
contains the vertices v and w. Faces are complete, there are no points in a face
that are not returned. Returns the vertices and faces of the submesh, as wel as
the list of indices of the vertices in the original mesh.
"""
function makesubmesh(g :: Geometry, v :: Index, w :: Index; minpoints :: Index =
      50)
    # Return the mesh if it already has the right size, or if it is smaller
    if g.nv â‰¥ minpoints
        return g.V, g.F, []
    end
    # Find shortest path with weighted edges
    A = weightedadjacency(g.V, g.F)
    graph = SimpleWeightedGraph(A)
    path = enumerate_paths(dijkstra_shortest_paths(graph, v), w)
    F = IndexMap(0)
    # Start by adding all faces adjacent to the path
    W, U = path, path

    faces = g.fmap[W]                # Look up ajacent faces
    append!(F, faces)                # Add to list of all used faces
    pts = g.F[vcat(faces...)]        # Find the points of these faces
    W = setdiff(vcat(pts...), U)     # Add new points to W, for first loop
    U = append!(U, W)
    while length(U) < minpoints
        faces = g.fmap[W]
        append!(F, faces)
        pts = g.F[vcat(faces...)]
        W = setdiff(vcat(pts...), U)
        U = append!(U, W)
    end
    U = sort(U)
    T = IndexList(0)
    for f in F append!(T, f) end # Make list of triangles
    g.V[U], g.F[unique(T)], U
end

"""
Removes points until either the mesh has the prescribed size, or there are
no obvious points to remove anymore. Returns the smaller lists of
points, faces and indices.
"""
function resizemesh(V :: VectorList, F :: FaceList, U :: IndexList,
                    npoints :: Index)
    while length(U) > npoints
        V, F, U, removed = removepoints(V, F, U, npoints)
        if length(removed) == 0
            break # Also quit when no obvious points are left to remove
        end
    end
    V, F, U
end

"""
Removes points until npoints is reached or no obvious points are left to
remove. Returns the smaller lists of points, faces and indices.
"""
function removepoints(V :: VectorList, FL :: FaceList, U :: IndexList,
                    npoints :: Index)
    # Return the mesh if it already has the right size
```

```
    if length(V) == npoints
        return V, FL, U, []
    end
    F = [f[i] for f in FL, i=1:3]
    nremove = 0
    toremove = Int[]
    # Find indices of vertices that are at the boundary (= in one or two faces)
    for u in U
        c = count(i -> i == u, F)
        if c == 1 || c == 2
            push!(toremove, u)
            nremove += 1
        end
        if length(U) - nremove == npoints # stop if enough points are removed
            break
        end
    end

    todelete = Int[]
    for (k, f) in enumerate(FL)
        if any(k-> k âĹĹ toremove, f)
            push!(todelete, k)
        end
    end

    # Remove points
    inds = find(elt -> eltâĹĹtoremove, U)
    deleteat!(V, inds)
    deleteat!(U, inds)
    deleteat!(FL, todelete)

    # Remove points that are not in any face
    F = [f[i] for f in FL, i=1:3]
    rm = Int[]
    for (k, u) in enumerate(U)
        if u âĹŃ F
            push!(rm, k)
        end
    end
    deleteat!(V, rm)
    deleteat!(U, rm)

    V, FL, U, toremove
end

"""
Minimum geodesic distance from each vertex in surface to any vertex in
"verts" using the algorithm from 'geodesics in heat', Crane et al, 2012. `m` is
a smoothing parameter: larger values of `m` will smooth & regularize the
distance computation. Smaller values of `m` do not return a more accurate
distance! See section 3.2.4 of the paper. Returns a ScalarList of distances.
Implementation of this, including functions used in this one, follows the one
found at https://github.com/gallantlab/pycortex/blob
/79350a6d6df8025f12a7ecfa1bd8fc5246322d65/cortex/polyutils.py.
"""
function geodesicdistance(g :: Geometry, verts :: IndexList;
                          m :: Scalar = 1.0, update :: Bool = false)
    # Update the factorization of the negative Laplace operator if m differs
    # or if update is needed
    if g.m âĹă m
        g.m = m
        update_geometry!(g)
        laplaceoperator!(g)
        nLCfactor!(g, m)
    elseif update
        update_geometry!(g)
        laplaceoperator!(g)
        nLCfactor!(g, m)
    end

    # I. Find u, the heat values

    # Set initial heat values
    u0 = zeros(g.nv)
    u0[verts] = 1.0
    # Solve system using factorization
    u = g.fact_nLC \ u0

    # II. Use u to find phi, the distances

    # Compute the surface gradient of u
    surfacegradient!(g, u)
    # Normalized negative gradient of u
```

```
    X = -g.gradu ./ norm.(g.gradu)

    # Integrated divergence of X
    divx!(g, X)
    # Solve system for the distances
    φ = g.nLC \ g.divx
    φ - minimum(φ)
end

""" Returns the factorization of the negative Laplace operator. """
function nLCfactor!(g :: Geometry, m :: Scalar)
    # Time of heat evolution
    t = m * (sum(g.Le) / g.ne)^2
    # Backward Euler matrix
    g.lfac = spdiagm(g.Av, 0) - t * g.nLC
    g.fact_nLC = ldltfact(g.lfac)
    nothing
end

""" To compute the negative Laplace operator. """
function laplaceoperator!(g :: Geometry)
    fill!(g.A, 0.)
    for (i, f) in enumerate(g.F)
        g.A[f[2], f[3]] += 0.5 * g.cots[i, 1]
        g.A[f[3], f[2]] += 0.5 * g.cots[i, 1]
        g.A[f[3], f[1]] += 0.5 * g.cots[i, 2]
        g.A[f[1], f[3]] += 0.5 * g.cots[i, 2]
        g.A[f[1], f[2]] += 0.5 * g.cots[i, 3]
        g.A[f[2], f[1]] += 0.5 * g.cots[i, 3]
    end
    g.An = spdiagm(squeeze(sum(g.A,1),1), 0)

    # Negative laplace operator
    g.nLC = g.A - g.An
    nothing
end

"""
For a given scalar-valued function across vertices, returns the gradients in
x, y, and z direction at each vertex.
"""
function surfacegradient!(g :: Geometry, u :: ScalarList)
    for (k,f) in enumerate(g.F)
        fe12 = cross(g.Nf[k], g.V[f[2]] - g.V[f[1]])
        fe23 = cross(g.Nf[k], g.V[f[3]] - g.V[f[2]])
        fe31 = cross(g.Nf[k], g.V[f[1]] - g.V[f[3]])
        g.gradu[k] = (fe12 * u[f[3]] + fe23 * u[f[1]] + fe31 * u[f[2]]) / (2.0 *
              g.Af[k])
    end
end

""" Computes the integrated divergence of X at each vertex. """
function divx!(g :: Geometry, X :: VectorList)
    g.divx = zeros(g.nv)
    for (i, v) in enumerate(g.V)
        for j in g.fmap[i]
            # Find the two indices in the face not equal to i, and their index
            # in the face
            ind = findfirst(g.F[j], i)
            k, l = deleteat!([g.F[j].data...], ind)
            n, m = deleteat!([1,2,3], ind)
            # Compute the integrated divergence
            g.divx[i] += 0.5 * (g.cots[j, m] * dot((g.V[k] - g.V[i]), X[j]) +
                                g.cots[j, n] * dot((g.V[l] - g.V[i]), X[j]))
        end
    end
end

"""
Functions to rename the vertices in the list of faces, such that they go
from 1 to the maximum number of unique vertices in F.
"""
function rename(F :: Matrix{Int})
    for (k, v) = enumerate(sort(unique(F[:])))
        if v == k
            nothing
        else
            F[find(x -> x==v, F)] = k
        end
    end
    F
end
```

```
function rename(F :: FaceList)
    F = [f[i] for f in F, i=1:3]
    Fr = rename(F)
    [Face(Fr[i,:]) for i=1:size(Fr,1)]
end

""" Get x, y, and z values from a VectorList. """
function xyzlist(VL :: VectorList)
    xs = ScalarList(length(VL))
    ys = ScalarList(length(VL))
    zs = ScalarList(length(VL))
    for (k, v) in enumerate(VL)
      xs[k] = v[1]
      ys[k] = v[2]
      zs[k] = v[3]
    end
    xs, ys, zs
end

""" Functions to plot a surface, when faces are known. """
function showplot(VL :: VectorList, F :: Matrix{Index})
    xs, ys, zs = xyzlist(VL)
    fig = figure()
    plot_trisurf(xs, ys, zs, triangles = F)
    xlabel("x-axis")
    ylabel("y-axis")
    zlabel("z-axis")
    show()
end

function showplot(VL :: VectorList, FL :: FaceList)
    F = [f[i] for f in FL, i=1:3]
    F = rename(F)
    showplot(VL, F .- 1) # minus 1, because PyPlot is Python
end

""" Function to plot a surface, when faces are unknown. """
function showplot(VL :: Matrix{Scalar})
    xs = VL[:,1]
    ys = VL[:,2]
    zs = VL[:,3]

    fig = figure()
    scatter3D(xs[3:end], ys[3:end], zs[3:end], color = "blue")
    scatter3D(xs[1], ys[1], zs[1], color = "red")
    scatter3D(xs[2], ys[2], zs[2], color = "orange")

    xlabel("x-axis")
    ylabel("y-axis")
    zlabel("z-axis")
    show()
end

""" Plot surface with colors indicating distance to given vertex. """
function plotdistance(g :: Geometry, i :: Int)
    F = [f[i] for f in g.F, i=1:3]
    F = rename(F)
    F = F .- 1 # minus 1, because PyPlot is Python
    n = length(g.V)
    xs, ys, zs = xyzlist(g.V)
    dists = geodesicdistance(g, [i])
    colors = cospi.(10. * dists / maximum(dists))
    facecolors = zeros(length(g.F))
    for (j,face) in enumerate(g.F)
        facecolors[j] = (colors[face[1]] + colors[face[2]] + colors[face[3]]) /
             3.
    end

    fig = figure()
    plot_trisurf(xs, ys, zs, triangles = F)
    ax = fig[:gca](projection = "3d")
    collec = ax[:plot_trisurf](xs, ys, zs, triangles = F, cmap = ColorMap("
         viridis_r"))
    matplotlib[:cm][:ScalarMappable][:set_array](collec, facecolors)

    # xlabel("x-axis")
    # ylabel("y-axis")
    # zlabel("z-axis")
    axis("off")
    ax[:grid](false)
    show()
end
```

```
"""
Function to write to file easily. In filenamex, the input data is written.
In Xtrain, every row contains one input matrix. The first three columns
are x, y and z values of the points and the other part is the vertex
adjacency matrix.
"""
function writexydata(geometries :: Array{Geometry, 1}, xfilename :: String,
                     yfilename :: String; npoints :: Index = 50)
    iox = open("./data/"*xfilename*".txt", "w")
    ioy = open("./data/"*yfilename*".txt", "w")
    for g in geometries
        emap = build_edge_map(g.E, g.F)
        points = shuffle(1:g.nv)
        n = fld(g.nv, 2)

        for i = 1:n
            # pick two points between which we want to know the distance
            v1 = points[i]
            v2 = points[i+n]
            # Compute distance from v1 to v2
            distances = []
            try distances = geodesicdistance(g, [v1]) end
            # If geodesicdistance() fails, skip to next point
            if isempty(distances) continue end

            V, F, U = makesubmesh(g, v1, v2)
            V, F, U = resizemesh(V, F, U, npoints)
            index1 = find(x -> x == g.V[v1], V)
            index2 = find(x -> x == g.V[v2], V)

            if length(U) âĽă npoints continue end # skip if not the right size
            if isempty(index1) continue end  # skip if vertex v1 was removed
            if isempty(index2) continue end  # skip if vertex v2 was removed

            # Translate surface to put vertex v1 at (0,0,0)
            for (k, vertex) in enumerate(V)
                V[k] = Vec([vertex.data...] - g.V[v1])
            end

            A = adjacency(F)
            m = size(A,1)
            X = zeros(m, m+3)
            for (k, p) in enumerate(V)
                X[k, 1:3] = p
            end
            X[:,4:end] = A

            # Make sure that the points between which we want to know the
            # distances are placed at the top of X and distances
            index1 = find(elt -> elt==v1, U)[1]
            index2 = find(elt -> elt==v2, U)[1]

            X = swaprows(X, 1, index1)
            U = swaprows(U, 1, index1)
            X = swaprows(X, 2, index2)
            U = swaprows(U, 2, index2)

            writedlm(iox, [X])
            writedlm(ioy, distances[v2])
        end
    end
    close(iox)
    close(ioy)
end

""" Given a matrix and the indices of two rows, swap the two rows. """
function swaprows(A :: Matrix{Scalar}, v1 :: Index, v2 :: Index)
    a = A[v2, :]
    b = A[v1, :]
    A[v2, :], A[v1, :] = b, a
    A
end

function swaprows(U :: Array{T, 1}, v1 :: Index, v2 :: Index) where T <: Number
    a = U[v1]
    b = U[v2]
    U[v2], U[v1] = b, a
    U
end

""" Make the vertex adjacency matrix. """
function adjacency(F :: FaceList)
    n = length(F)
```

```
    F = [f[i] for f in F, i=1:3]
    F = rename(F)
    m = maximum(F)
    A = zeros(Int, m, m)
    for i = 1:n
        f = F[i,:]
        A[f[1], f[2]], A[f[2], f[1]] = 1, 1
        A[f[1], f[3]], A[f[3], f[1]] = 1, 1
        A[f[2], f[3]], A[f[3], f[2]] = 1, 1
    end
    A
end

""" Make the weighted adjacency matrix. """
function weightedadjacency(V :: VectorList, F :: FaceList)
    n = length(F)
    F = [f[i] for f in F, i=1:3]
    F = rename(F)
    m = maximum(F)
    A = spzeros(Scalar, m, m)
    for i = 1:n
        f = F[i,:]
        d12 = norm(V[f[1]] - V[f[2]])
        d13 = norm(V[f[1]] - V[f[3]])
        d23 = norm(V[f[2]] - V[f[3]])
        A[f[1], f[2]], A[f[2], f[1]] = d12, d12
        A[f[1], f[3]], A[f[3], f[1]] = d13, d13
        A[f[2], f[3]], A[f[3], f[2]] = d23, d23
    end
    A
end

#
### NN part
#

mutable struct Model
    n_dense1 :: Index
    n_dense2 :: Index
    n_conv1 :: Index
    n_conv2 :: Index
    λ :: Scalar
    NN :: Chain
    k :: Index
    af :: Function

    function Model(n_dense1 :: Index, n_dense2 :: Index, n_conv1 :: Index,
                   n_conv2 :: Index, λ :: Scalar, k :: Index, af :: Function)
        self = new()
        self.n_dense1 = n_dense1          # Number of units in 1st dense layer
        self.n_dense2 = n_dense2          # Number of units in 2nd dense layer
        self.n_conv1 = n_conv1             # Size of filter in 1st conv layer
        self.n_conv2 = n_conv2             # Size of filter in 2nd conv layer
        self.λ = λ                        # Learning rate
        self.k = k                        # Number of input units
        self.af = af
        self.NN = Chain(                  # Neural network architecture
            Dense(k, n_dense1, af),
            Dense(n_dense1, n_dense2, af),
            x -> reshape(x, (size(x)...,1,1)),
            Conv((n_conv1, k + 3), 1=>5, af),
            Conv((n_conv2, 1), 5=>1, af),
            x -> squeeze(x, (2,3,4)),
            Dense(n_dense2 - n_conv1 - n_conv2 + 2, 1, identity))
        self
    end
end

""" Loss function """
function loss(X, Y, model :: Model)
    loss = 0.0
    n = length(X)
    if n âĽă length(Y) error("DimensionMismatch in test and train data") end
    for i = 1:n
        loss += Flux.mse(model.NN(X[i]), Y[i])
    end
    loss / n
end

"""
Function to train on given data, which is assumed to be a list of matrices.
Also, evaluates every epoch whether the loss was NaN of equal to the previous
loss. Returns arrays with train and test loss values.
```

```
"""
function train!(Xtrain :: MatList, Ytrain :: ScalarMap, Xtest :: MatList,
                Ytest :: ScalarMap, model :: Model, epochs :: Index,
                batchsize :: Index, opt; verbose :: Scalar = 0.1)
    # Make batches
    Xset, Yset, n_batches = batches_size(Xtrain, Ytrain, batchsize)
    trainloss = zeros(n_batches, epochs)
    testloss = zeros(n_batches, epochs)

    for i = 1:epochs
        # Train the network per batch
        for (k,j) in enumerate(1:length(Xset))
            args = (Xset[j], Yset[j], Xtest, Ytest, model, opt)
            trloss, teloss = train!(args...)
            trainloss[k, i] = trloss
            testloss[k, i] = teloss

            # Print losses
            if i % ceil(Int, epochs*verbose) == 0
                printloss(i, k, trloss, teloss)
            end
        end

        #  Test for NaNs and network not learning at the start of training
        if isnan(trainloss[i])
            printloss(i, trainloss[i], testloss[i])
            break
        end
        if i == 4 && testloss[end, i-2] == testloss[end, i-1] == testloss[end, i
            ]
            println("Network not learning, quitting training and returning
                losses.")
            printloss(i, n_batches, trainloss[end,i], testloss[end,i])
            break
        end
        # Make new batches
        Xset, Yset, n_batches = batches_size(permutations(Xtrain, Ytrain)...,
            batchsize)
    end
    trainloss, testloss
end

function train!(Xtrain :: MatList, Ytrain :: ScalarMap, Xtest :: MatList,
                Ytest :: ScalarMap, model :: Model, opt)

    Flux.back!(loss(Xtrain, Ytrain, model)) # Calculate gradients
    opt()                                   # Update weights

    # Loss values
    trainloss = Flux.Tracker.data(loss(Xtrain, Ytrain, model))
    testloss = Flux.Tracker.data(loss(Xtest, Ytest, model))

    trainloss, testloss
end

function printloss(i :: Index, j :: Index, trainloss, testloss)
    println("Epoch $i, batch $j: train loss is $trainloss, test loss is $
        testloss")
end


""" For a given array of input points, returns the model's predictions """
function evaluate(X :: MatList, model :: Model)
    ymodel = ScalarMap(length(X))
    for (k, x) in enumerate(X)
        ymodel[k] = Flux.Tracker.data(model.NN(x))
    end
    ymodel
end

"""
Splits the data in the given amount of batches or in batches of the given size.
If the length of the given list of matrices is not divisible by the number of
batches or the batch size, the returned list contains more batches and/or the
last batch is smaller than the rest.
"""
function batches_number(X :: MatList, Y :: ScalarMap, nb :: Index)
    N = length(X)
    s = fld(N, nb)
    r = 1:s:N
    n_batches = length(r)
    Xset = Array{MatList}(n_batches)
    Yset = Array{}(n_batches)
```

```
984
985     for (k,i) in enumerate(r)
986         if i == r[end]
987             Xset[k] = X[i:end]
988             Yset[k] = Y[i:end]
989         else
990             Xset[k] = X[i:i+s-1]
991             Yset[k] = Y[i:i+s-1]
992         end
993     end
994     Xset, Yset, n_batches
995 end
996
997 function batches_size(X :: MatList, Y :: ScalarMap, s :: Index)
998     N = length(X)
999     r = 1:s:N
1000    n_batches = length(r)
1001    Xset = Array{MatList}(n_batches)
1002    Yset = Array{ScalarMap}(n_batches)
1003
1004    for (k,i) in enumerate(r)
1005        if i == r[end]
1006            Xset[k] = X[i:end]
1007            Yset[k] = Y[i:end]
1008        else
1009            Xset[k] = X[i:i+s-1]
1010            Yset[k] = Y[i:i+s-1]
1011        end
1012    end
1013    Xset, Yset, n_batches
1014 end
1015
1016 """ Permutes the given dataset. """
1017 function permutations(X :: MatList, Y :: ScalarMap;
1018     randomseed :: Union{Void, Int} = nothing)
1019     if randomseed âĽă nothing
1020         srand(randomseed)
1021     end
1022     p = randperm(length(X))
1023     X[p], Y[p]
1024 end
1025
1026 function Xtomatrices(X :: Array{Float64, 2}, k :: Index)
1027     n= size(X,1)
1028     Xarray = MatList(n)
1029     m = k + 3
1030     for i = 1:n
1031         Xarray[i] = reshape(X[i,:], (k, m))
1032     end
1033     Xarray
1034 end
1035
1036 function Ytovectors(Y :: Array{Float64, 2})
1037     n = size(Y, 1)
1038     Yvec = ScalarMap(n)
1039     for i = 1:n
1040         Yvec[i] = Y[i,:]
1041     end
1042     Yvec
1043 end
1044
1045 function readandshuffle(xname :: String, yname :: String;
1046     randomseed :: Union{Void, Int} = nothing)
1047     # Read
1048     Xtraindata = readdlm("./data/"*xname*".txt");
1049     Ytraindata = readdlm("./data/"*yname*".txt");
1050     k = 50
1051     # Convert to the right kind of data
1052     Xtrain = Xtomatrices(Xtraindata, k)
1053     Ytrain = Ytovectors(Ytraindata)
1054     permutations(Xtrain, Ytrain, randomseed = randomseed)
1055     Xtrain, Ytrain
1056 end
1057
1058 """ Functions to load weights of a model """
1059 function loadweights(filename :: String, model :: Model)
1060     @load filename*".bson" weights
1061     Flux.loadparams!(model.NN, weights)
1062 end
1063
1064 function loadweights(filename :: String, model :: Chain)
1065     @load filename*".bson" weights
1066     Flux.loadparams!(model, weights)
```

```
end

""" Functions to save weights of a model """
function saveweights(filename :: String, model :: Model)
    weights = Tracker.data.(params(model.NN));
    @save filename*".bson" weights
end

function saveweights(filename :: String, model :: Chain)
    weights = Tracker.data.(params(model));
    @save filename*".bson" weights
end

end # module DistNN3D
```

D FEM

Listing 17 *Code for FEM analysis.*

```
# -*- coding: utf-8 -*-
"""
Model that contains a non-linear elastic planar 2D model using a parametrization
based on the Kirchhoff-Love shell. The structure is an inflatable line, fixed
at the left and right endpoints.
"""
import sys
sys.path.insert(0, r'C:\Users\s126359\Documents\TUe\Stage\nutils')
from nutils import mesh, function, cli, solver, plot, log
import numpy as np

def main(
        nelems: 'number of elements' = 16,
        length: 'length of the parametrization' = .1,
        pressure: 'pressure' = 10,
        epsilon: 'flexural rigidity' = 7.5e-9,
        w: 'w' = '0.0025',
        ext: 'extension' = 0,
        timestep: 'time step' = 1/24,
        degree: 'polynomial degree' = 2,
        solvetol: 'solver tolerance' = 1e-10,
        referenceconfig: 'reference configuration' = 'line',
        figures: 'create figures' = True,
        ttest: 'tensile test' = False,
        klshell: 'Kirchhoff-Love shell' = True,
        lhs0: 'previous guess' = None,
    ):
    """Main method of the module, which computes the deformed domain."""

    # Create domain and geometry
    if referenceconfig != 'line':
        nodes = np.linspace(-np.pi, np.pi, nelems+1)
    else:
        nodes = np.linspace(0, length, nelems+1)
    domain, geom = mesh.rectilinear([nodes])

    # Create namespace and populate it
    ns = function.Namespace(default_geometry_name='x0')
    ns.x0 = geom
    ns.basis = domain.basis('spline', degree=degree).vector(2)
    ns.u_i = 'basis_ni ?lhs_n'
    ns.p = pressure
    ns.epsilon = epsilon
    ns.w = w
    ns.ext = ext
    # Parametrization such that the left boundary coincides with the zero value
    # of the parametrization variable.
    if referenceconfig == 'line':
        ns.refc = function.stack([
            ns.x0[0],
            0
            ])
    elif referenceconfig == 'circle':
        ns.refc = -function.TrigNormal(-ns.x0[0])

    ns.x_i = 'refc_i + u_i'
    ns.rotx0 = 'x_1,0'
```

```
    ns.rotx1 = '-x_0,0'

    # Construct residual
    inertia = domain.integral('basis_nl x_l' @ ns, geometry=ns.x, degree=degree)
    if klshell:
        res = domain.integral(
            '(x_i,j x_i,j - 1) x_l,k basis_nl,k' @ ns, geometry=ns.x0,
            degree=degree)
    else: # Linear elastic string
        res = domain.integral(
                '(1 - sqrt(x_i,j x_i,j)^(-1)) x_l,k basis_nl,k' @ ns,
                geometry=ns.x0, degree=degree)
    res += domain.integral(
            'epsilon x_i,jj basis_ni,ll'@ ns, geometry=ns.x0, degree=degree)
    # Pressure
    res += domain.integral(
           'w p (basis_n0 rotx0 + rotx1 basis_n1)' @ ns, geometry=ns.x0, degree=
                degree)

    # Construct dirichlet boundary conditions
    if not ttest:
        sqr = domain.boundary['left,right'].integral(
            'u_k u_k' @ ns, geometry=ns.x0, degree=degree)
    if ttest:
        sqr = domain.boundary['left'].integral(
            'u_k u_k' @ ns, geometry=ns.x0, degree=degree)
        if not (np.abs(ext) < 1e-6):
            sqr += domain.boundary['right'].integral('(u_0 - ext)^2' @ ns,
                           geometry=ns.x0, degree=degree)
    cons = solver.optimize('lhs', sqr, droptol=1e-15)

    # Find lhs such that res == 0 and substitute this lhs in the namespace
#    lhs = solver.solve(solver.newton('lhs', res, constrain=cons, lhs0=lhs0),
     solvetol)
#    ns |= dict(lhs=lhs)

    lhs0 = domain.project('refc_i' @ ns, onto=ns.basis, geometry=ns.x0, degree=
         degree)
    lhs = solver.solve(solver.pseudotime('lhs', residual=res, inertia=inertia,
                       lhs0=lhs0, timestep=timestep, constrain=cons), solvetol)
    ns |= dict(lhs=lhs)

    stretch = 'sqrt(x_i,j x_i,j)' @ ns
    if klshell:
        tension = 'x_i,j x_i,j - 1' @ ns
    else:
        tension = '1 - sqrt(x_i,j x_i,j)^(-1)' @ ns
    t, s = domain.elem_eval([tension, stretch], ischeme='bezier1', separate=True
          )
    t, s = np.array(t).flatten(), np.array(s).flatten()

    # plot initial configuration, solution and normal
    if figures:
        points = domain.elem_eval(ns.x, ischeme='bezier9', separate=True)
        refpoints = domain.elem_eval(ns.refc, ischeme='bezier9', separate=True)
#        rotpoints1, rotvals1 = domain.elem_eval([ns.x0, ns.rotx1], ischeme='
     bezier9', separate=True)
#        rotpoints0, rotvals0 = domain.elem_eval([ns.x0, ns.rotx0], ischeme='
     bezier9', separate=True)
        with plot.PyPlot('referenceconfiguration', index=nelems) as plt:
            plt.mesh(refpoints, cmap='jet', tight=True,edgewidth=1)
            plt.grid(True)
            ax = plt.gca()
            ax.set_axisbelow(True)
            plt.xlabel('Reference configuration, dimensions in meters')
            plt.savefig('undeformed_k-l_balloon.pdf')
            plt.show()
        with plot.PyPlot('solution', index=nelems) as plt:
            plt.mesh(points, cmap='jet', tight=False, edgewidth=1.5)
            plt.grid(True)
            ax = plt.gca()
            ax.set_axisbelow(True)
            plt.xlabel('Deformed balloon, dimensions in meters')
            plt.savefig('deformed_k-l_balloon.pdf')
            plt.show()
#        with plot.PyPlot('pressure', index=nelems) as plt:
#            plt.mesh(rotpoints0, rotvals0, tight=False)
#            plt.xlim(np.amin(rotpoints0), np.amax(rotpoints0))
#            plt.ylim(np.amin(rotvals0), np.amax(rotvals0))
#            plt.grid(True)
#            ax = plt.gca()
#            ax.set_axisbelow(True)
#            plt.show()
```

```
#        with plot.PyPlot('pressure', index=nelems) as plt:
#            plt.mesh(rotpoints1, rotvals1, tight=False)
#            plt.xlim(np.amin(rotpoints1), np.amax(rotpoints1))
#            plt.ylim(np.amin(rotvals1), np.amax(rotvals1))
#            plt.grid(True)
#            ax = plt.gca()
#            ax.set_axisbelow(True)
#            plt.show()

    return points, cons, lhs, t, s


def tensiletest(
        nelems: 'number of elements' = 16,
        length: 'length of the parametrization' = 1.,
        epsilon: 'flexural rigidity' = 1e-5,
        w: 'w' = '1',
        ext: 'extension' = .01,
        degree: 'polynomial degree' = 2,
        solvetol: 'solver tolerance' = 1e-10,
        figures: 'create figures' = False,
        ntimes: 'number of times extended' = 5,
        klshell: 'Kirchhoff-Love shell' = True,
    ):
    lhs0=None
    tension = np.zeros(nelems*ntimes)
    stretch = np.zeros(nelems*ntimes)
    for itimes in log.range('stretch', ntimes):
        log.user('Stretch numer {}/{}'.format(itimes, ntimes-1))
        p0 = main(length=length, pressure=0, ext=ext*itimes,
                referenceconfig='line', figures=figures, ttest=True,
                klshell=klshell, lhs0=lhs0)[2:5]
        lhs0, t1, s1 = p0
        tension[(itimes*nelems):(nelems*(itimes+1))] = t1
        stretch[(itimes*nelems):(nelems*(itimes+1))] = s1
    x = np.linspace(np.min(stretch), np.max(stretch), nelems+1)

    # plot stress strain points with expected curves
    with plot.PyPlot('stretch-tension', index=ntimes) as plt:
        plt.plot(stretch, tension, label='computed', marker='+', linestyle='')
        if klshell:
            plt.plot(x,x**2-1, linestyle='--', label='x^2-1')
        else:
            plt.plot(x,1-x**(-1), linestyle='-.', label='1-x^(-1)')
        plt.grid(True)
        ax = plt.gca()
        ax.set_axisbelow(True)
        plt.xlabel('Stretch')
        plt.ylabel('Tension')
        plt.savefig('tension-stretch.pdf')
        plt.legend()
        plt.show()
    return tension, stretch


if __name__ == '__main__':
  cli.choose(main, tensiletest)
```