FEniCS Mechanics Documentation

Release 1.0

Miguel A. Rodriguez

Oct 09, 2018

CONTENTS:

CHAPTER

ONE

INTRODUCTION

1.1 Objective

FEniCS Mechanics is a python package developed to facilitate the formulation of computational mechanics simulations and make these simulations more accessible to users with limited programming or mechanics knowledge. This is done by using the tools from the FEniCS Project (www.fenicsproject.org). The FEniCS Project libraries provide tools to formulate and solve variational problems. FEniCS Mechanics is built on top of these libraries specifically for computational mechanics problems.

1.2 Current State

At this point, FEniCS Mechanics can handle problems in fluid mechanics and (hyper)elasticity. Fluid Mechanics problems are formulated in Eulerian coordinates, while problems in elasticity are formulated in Lagrangian coordinates. Further development is required for the implementation of Arbitrary Lagrangian-Eulerian (ALE) formulations. Single domain problems are supported, but interaction problems are not. While the user may provide their own constitutive equations, the following definitions are provided by FEniCS Mechanics:

- Solids:
  - Linear Isotropic Material
  - Neo-Hookean Material
  - Fung-type Material
  - Guccione et al. (1995)
- Fluids:
  - Incompressible Newtonian
  - Incompressible Stokes

See section *Custom Constitutive Equation* for instructions on how to provide a user-defined constitutive equation.

Also note that this package can handle both steady state and time-dependent simulations. Check *Finite Difference Methods*.

1.3 Installation

FEniCS Mechanics requires that version 2016.1.0 or newer of FEniCS be installed. Installation instructions can be found at http://fenicsproject.org/download.

If you are using FEniCS 2016.1.0 or 2016.2.0 in Python 2, you may also install the FEniCS Application CBC-Block found at https://bitbucket.org/fenics-apps/cbc.block. Note that to use CBC-Block, PETSc must be installed with the Trilinos ML package.

Once dependencies have been installed, the user simply has to inform python of the path where FEniCS Mechanics is stored. This could be done in several ways. Here are three possible solutions:

1. Add the directory to the `PYTHONPATH` environment variable:

```
export PYTHONPATH=<path/to/fenicsmechanics>:$PYTHONPATH
```

2. Append the directory to the path in your python script

```
import os
os.path.append("path/to/fenicsmechanics")
```

3. Add a file to the python `dist-packages` directory with the `pth` extension, e.g. `python3.5/dist-packages/additional_packages.pth` with a list of additional directories you want python to check for importable libraries. The content of this file will simply be:

```
path/to/fenicsmechanics
```

   with the specific path on your machine.

CHAPTER

TWO

MATHEMATICAL FORMULATION

A brief review of continuum mechanics, the finite element method for spatial discretization, and finite difference methods for time derivative discretization is presented here. For further details, the reader is referred to Chadwick *[Cha99]* and Hughes *[Hug07]*.

2.1 Continuum Mechanics

The body of interest is assumed to occupy a subregion of three-dimensional Euclidean space at some initial time, t_0, and denote with region by $\mathcal{B}_0$. We refer to $\mathcal{B}_0$ as the reference configuration. The region occupied by the same body at a later time, t, will be denoted by $\mathcal{B}$, and referred to as the current configuration. Solving mechanics problems involves solving for the map (or its derivatives) that transforms the body from $\mathcal{B}_0$ to $\mathcal{B}$. We will denote this map with $\boldsymbol{\varphi} : \mathcal{B}_0 \times \mathbb{R} \to \mathcal{B}$, where $t \in \mathbb{R}$.

The formulation of continuum mechanics that we will use is concerned with the deformation of the body at a local level, which is quantified by the deformation gradient

$$\mathbf{F}(\mathbf{X}, t) = \frac{\partial \boldsymbol{\varphi}}{\partial \mathbf{X}},$$

where $\mathbf{X} \in \mathcal{B}_0$ is the position vector of a material point at time t_0. This tensor is used to define other tensors quantifying different strain measures – all of which can be used to study the kinematics of deformation.

Once the kinematics of the deformation undergone by the body of interest has been mathematically formulated, they can be used to study the relationship between deformation and forces that either cause or result from it. These relations are referred to as dynamics. This leads to the concept of the Cauchy stress tensor, which we will denote by $\mathbf{T}$. The Cauchy stress tensor maps the unit normal vector, $\mathbf{n}$, on any surface within a continuum to the traction vector, $\mathbf{t}$, which is the force per unit area at the point where $\mathbf{T}$ and $\mathbf{n}$ are evaluated. By Newton's second law, and the Reynolds Transport Theorem, we get

$$\int_{\mathcal{B}} \rho \frac{d\mathbf{v}}{dt}\, dv = \int_{\partial\mathcal{B}} \mathbf{t}\, da + \int_{\mathcal{B}} \rho \mathbf{b}\, dv,$$

where $\mathbf{v} = \frac{d}{dt}\mathbf{u}$ is the velocity field of the body, $\mathbf{u}$ is the displacement, $\mathbf{b}$ is the force applied per unit mass, and $\partial\mathcal{B}$ is the boundary of $\mathcal{B}$. Making use of the Cauchy stress tensor, the divergence theorem, and the localization theorem, we get

$$\rho \dot{\mathbf{v}} = \text{div } \mathbf{T} + \rho \mathbf{b},$$

where we have replaced the time derivative by the dot, and div is the divergence operator with respect to the spatial coodinates. FEniCS Mechanics uses this equation to formulate fluid mechanics problems. The equivalent equation in the reference coordinate system is

$$\rho_0 \dot{\mathbf{v}} = \text{Div } \mathbf{P} + \rho_0 \mathbf{b},$$

where $\rho_0 = \rho J$ is the referential mass density, $\mathbf{P} = J\mathbf{T}\mathbf{F}^{-T}$ is the first Piola-Kirchhoff stress tensor, $J = \det \mathbf{F}$ is the Jacobian, and Div is the divergence operator with respect to the referential coordinates, $\mathbf{X}$. This equation is used to formulate solid mechanics problems.

Note that we have not limited the formulation to a specific type of material, e.g. rubber or water, and thus these equations are applicable to all materials that satisfy the continuity assumptions of the classical continuum mechanics formulation. Specific relations between strain tensors and the Cauchy tensor are known as constitutive equations and depend on the material being modelled. For further details on constitutive equations, the reader is referred to Chadwick *[Cha99]*.

2.1.1 Incompressibility Constraint

Many materials are modeled as incompressible. In order to satisfy incompressibility, we must formulate it as a constraint that the displacement and/or velocity must satisfy. This results in an additional equation, making the system over-determined. Thus, we need an additional variable, a Lagrange multiplier, to solve the system. This Lagrange multiplier ends up being the pressure within the material, be it a solid or fluid.

If a material is modeled as incompressible, the stress tensor will now depend on the displacement field for solids, velocity field for fluids, and the pressure for both. I.e.,

$$\mathbf{P} = \mathbf{G}(\mathbf{u}, p)$$

for solids, and

$$\mathbf{T} = \mathbf{H}(\mathbf{v}, p)$$

for fluids. The constraint equation will also depend on the relevant vector field, and the scalar field p. Thus,

$$G(\mathbf{u}, p) = 0$$

or

$$H(\mathbf{v}, p) = 0.$$

The constraint equation for solid materials is of the form

$$G(\mathbf{u}, p) = \phi(\mathbf{u}) - \frac{1}{\kappa} p,$$

where κ is the bulk modulus. For linear materials, we take

$$\phi(\mathbf{u}) = \text{div } \mathbf{u}.$$

On the other hand,

$$\phi(\mathbf{u}) = \frac{1}{J} \ln(J)$$

is the default expression for nonlinear materials.

The incompressibility constraint for fluid mechanics is

$$H(\mathbf{v}, p) = \text{div } \mathbf{v}.$$

2.2 Finite Element Method

The finite element method (FEM) requires a variational form. Thus, we must convert the governing equations from *Continuum Mechanics* to a variational form. This involves taking the dot product of the equation of interest with an arbitrary vector field, $\boldsymbol{\xi}$, using the product rule, and divergence theorem to obtain

$$\int_{\mathcal{B}} \boldsymbol{\xi} \cdot \rho \dot{\mathbf{v}} \, dv + \int_{\mathcal{B}} \frac{\partial \boldsymbol{\xi}}{\partial \mathbf{x}} \cdot \mathbf{T} \, dv = \int_{\mathcal{B}} \boldsymbol{\xi} \cdot \rho \mathbf{b} \, dv + \int_{\Gamma_q} \boldsymbol{\xi} \cdot \bar{\mathbf{t}} \, da,$$

where $\bar{\mathbf{t}}$ is a specified traction on $\Gamma_q \in \partial\mathcal{B}$. Applying the chain rule to the material time derivative of the velocity yields

$$\int_{\mathcal{B}} \boldsymbol{\xi} \cdot \rho \frac{\partial \mathbf{v}}{\partial t} \, dv + \int_{\mathcal{B}} \boldsymbol{\xi} \cdot \frac{\partial \mathbf{v}}{\partial \mathbf{x}} \mathbf{v} \, dv + \int_{\mathcal{B}} \frac{\partial \boldsymbol{\xi}}{\partial \mathbf{x}} \cdot \mathbf{T} \, dv = \int_{\mathcal{B}} \boldsymbol{\xi} \cdot \rho \mathbf{b} \, dv + \int_{\Gamma_q} \boldsymbol{\xi} \cdot \bar{\mathbf{t}} \, da.$$

Note that this is the most general variational form of the governing equation for the balance of momentum, and can thus be used to provide a general formulation for computational mechanics problems. FEniCS Mechanics does just this, and changes the physical meaning of the variables in the above weak form when necessary. E.g., the stress tensor $\mathbf{T}$ is replaced by the first Piola-Kirchhoff stress tensor $\mathbf{P}$ when formulating a solid mechanics problem.

Suppose that the true solution to the variational forms belongs to a function space $\mathcal{F}$. The FEM uses a subspace of $\mathcal{F}$ with finite size to approximate the solution to the boundary value problem. We denote this finite function space by $\mathcal{F}^h$, which is spanned by polynomials of order $n \in \mathbb{N}$, with $n \leq 2$ in most cases. Thus, the approximation of a function $\mathbf{u}$ would be a linear combination of a set of polynomials, i.e.

$$\mathbf{u} = \sum_{i=1}^{N_n} \hat{\mathbf{u}}_i \phi_i(\mathbf{x}),$$

where $\hat{\mathbf{u}}_i$ are the coefficients of the approximation, $\{\phi_i\}_{i=1}^{N_n}$ is the set of basis functions for $\mathcal{F}^h$, and N_n is the cardinality of $\mathcal{F}^h$. We substitute approximations for all functions in the weak form, resulting in a system of ordinary differential equations (ODEs) in which we will solve for $\hat{\mathbf{u}}_i$. We will write this system of ODEs as

$$\begin{aligned} \dot{u} &= v \\ M\dot{v} + C(v) + R(u, v, p) &= F_b(t) + F_q(u, t) \\ G(u, v, p) &= 0, \end{aligned}$$

where each of these terms is the approximation of the terms given in the general variational form provided above. FEniCS Mechanics parses a user-provided dictionary to determine which terms in the above system of ODEs need to be computed and how.

2.3 Finite Difference Methods

Once spatial discretization has been achieved with the FEM, we must discretize the time derivatives in the system of ODEs given above. The time integrators implemented in FEniCS Mechanics are currently single-step methods.

2.3.1 First Order ODEs

Consider the system of ODEs

$$\dot{y} = f(y, t),$$

where $y \in \mathbb{R}^N$ for some $N \in \mathbb{N}$, and $t \in \mathcal{T} \in \mathbb{R}$. A general single-step, single-stage numerical method for approximating the solution to such a differential equation takes the form

$$\frac{y_{n+1} - y_n}{\Delta t} = \theta f(y_{n+1}, t_{n+1}) + (1 - \theta) f(y_n, t_n)$$

Applying the generalized θ-method to the system of ODEs, we get

$$f_1(u_{n+1}, v_{n+1}, p_{n+1}) = 0,$$
$$f_2(u_{n+1}, v_{n+1}, p_{n+1}) = 0,$$

and

$$f_3(u_{n+1}, v_{n+1}, p_{n+1}) = 0,$$

where

$$f_1(u_{n+1}, v_{n+1}, p_{n+1}) = u_{n+1} - \theta \Delta t v_{n+1} - u_n - \Delta(1 - \theta) v_n,$$

$$\begin{aligned} f_2(u_{n+1}, v_{n+1}) = M v_{n+1} + \theta \Delta t \left[C(v_{n+1}) + R(u_{n+1}, v_{n+1}, p_{n+1}) - F_b(t_{n+1}) - F_q(u_{n+1}, t_{n+1}) \right] \\ - M v_n + \Delta t (1 - \theta) \left[C(v_n) + R(u_n, v_n, p_n) - F_b(t_n) - F_q(u_n, t_n) \right], \end{aligned}$$

and f_3 is the discrete variational form of the incompressibility constraint. This discretization is used by `MechanicsProblem` and `MechanicsBlockSolver`, which is only available when FEniCS is installed with python 2, and CBC-Block is also installed. It is also used by `FluidMechanicsProblem` and `FluidMechanicsSolver` without the dependency on displacement, u_n, and without f_1.

2.3.2 Second Order ODEs

Alternatively, one can discretize a system of second-order ODEs. First, we substitute $v = \dot{u}$ into the ODE that results from the balance of linear momentum. This gives

$$M \ddot{u} + R(u, p) = F_b(t) + F_q(u, t),$$

and

$$G(u, p) = 0,$$

where we also made use of the fact that solid mechanics problems are formulated with respect to the reference configuration. For this problem, we use the Newmark scheme to discretize the second-order time derivative. The Newmark scheme is given by

$$\dot{u}_{n+1} = \dot{u}_n + \Delta t \left[(1 - \gamma) \ddot{u}_n + \gamma \ddot{u}_{n+1} \right],$$

and

$$u_{n+1} = u_n + \Delta t \dot{u}_n + \frac{1}{2} (\Delta t)^2 \left[(1 - 2\beta) \ddot{u}_n + 2\beta \ddot{u}_{n+1} \right].$$

We then solve for $\ddot{u}_{n+1}$ in the above equations, and substitute into the ODE given above. Then we solve the system for u_{n+1} (with a nonlinear solver if necessary).

CHAPTER

THREE

CODE STRUCTURE AND USER INTERFACE

This chapter covers the user interface, as well as the internal design of FEniCS Mechanics. The following abbreviations are used in figures below.

Abbreviation	Full Name
BMP	`BaseMechanicsProblem`
IM	`IsotropicMaterial`
MP	`MechanicsProblem`
LIM	`LinearIsoMaterial`
FMP	`FluidMechanicsProblem`
NHM	`NeoHookeMaterial`
SMP	`SolidMechanicsProblem`
AM	`AnisotropicMaterial`
MBS	`MechanicsBlockSolver`
FM	`FungMaterial`
NVS	`NonlinearVariationalSolver`
GM	`GuccioneMaterial`
SMS	`SolidMechanicsSolver`
F	`Fluid`
FMS	`FluidMechanicsSolver`
NF	`NewtonianFluid`
EM	`ElasticMaterial`

3.1 Code Structure

The flow of information within FEniCS Mechanics is shown in Figure Fig. 3.1. First, the user defines the mechanics problem they wish to solve through a python dictionary, which we will refer to as `config`. FEniCS Mechanics then uses this input to define the variational form that is to be solved through the Unified Form Language (UFL) from the FEniCS Project. Note that information provided in `config` is sent to two separate components: problem formulation, and material law. This separation is done to maintain the generality of the governing equation given in *Continuum Mechanics*. In other words, a separate part of the code is responsible for tailoring the governing equations to specific material models, providing a modular structure that better lends itself to the addition of new models.

The forms defined in the problem formulation stage are then used for matrix assembly in order to obtain the numerical solution to the specified problem. All of the terms that need to be defined within `config` are listed and explained in *User Interface*.

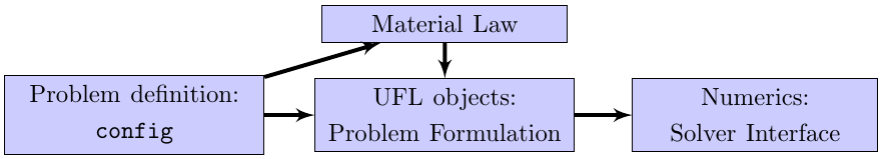

Fig. 3.1: The flow of information within FEniCS Mechanics.

3.1.1 Problem Objects

There are three classes that define the variational form of the computational mechanics problem: `MechanicsProblem`, `SolidMechanicsProblem`, and `FluidMechanicsProblem`. The input, `config`, takes the same structure for all three. All of three classes are derived from `BaseMechanicsProblem`, as is shown in Fig. 3.2. Functions that parse different parts of `config` belong to `BaseMechanicsProblem` since they are common to all mechanics problems. In addition to parsing methods, terms in the variational form of the governing equation are defined in the parent class, as well as any functions that update the state of common attributes for all problems.

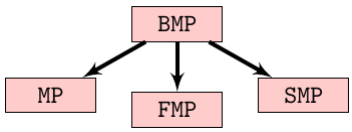

Fig. 3.2: A tree of the different problem classes in FEniCS Mechanics showing their inheritance.

One difference between all three is the time integration scheme used. Specifically, `MechanicsProblem` treats the system of ODEs after FEM discretization as first order. Thus, the system is reduced to a set of first order ODEs for solid mechanics as shown at the end of *Finite Element Method*, and integrated with the method described in *First Order ODEs*. The time integration scheme in `FluidMechanicsProblem` is currently the same without the need for the equation $\dot{u} = v$. On the other hand, `SolidMechanicsProblem` defines the variational form using the Newmark integration scheme. This is a common integrator used for solid mechanics problems.

Another difference between `MechanicsProblem` and the other two problem classes is that `MechanicsProblem` uses separate function space objects from `dolfin` for vector and scalar fields. The other two problem classes use a mixed function space object.

All problem classes are instantiated by providing the python dictionary, `config`, e.g.

```
import fenicsmechanics as fm
# ...
# Code defining 'config'
# ...
problem = fm.MechanicsProblem(config)
```

Full demonstrations of the use of FEniCS Mechanics are given in *Examples*.

3.1.2 Solver Objects

Once the problem object has been created with the `config` dictionary, it is passed to a solver class for instantiation. Like the problem classes, there are three solver classes: `MechanicsBlockSolver`, `SolidMechanicsSolver`, and `FluidMechanicsSolver`. The inheritance of these classes are shown in Fig. 3.3. All three solver classes use

the UFL objects defined by their corresponding problem classes to assemble the resulting linear algebraic system at each iteration of a nonlinear solve. This is repeated for all time steps of the problem if it is time-dependent.

Note that `MechanicsBlockSolver` is a stand-alone class, while `SolidMechanicsSolver` and `FluidMechanicsSolver` are both subclasses of the `NonlinearVariationalSolver` in `dolfin`. This is due to the fact that `MechanicsProblem` uses separate function spaces for the vector and scalar fields involved in the problem, and hence uses CBC-Block to assemble and solve the resulting variational form.

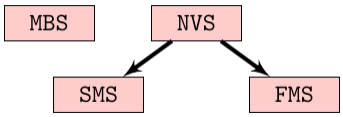

Fig. 3.3: A tree of the different solver classes in FEniCS Mechanics showing their inheritance.

3.1.3 Constitutive Equations

A number of constitutive equations have been implemented in FEniCS Mechanics. All of them can be found in the `materials` sub-package. A list of all constitutive equations included can be seen by executing `fenicsmechanics.list_implemented_materials()`. The inheritance for constitutive equations of solid materials is shown in Fig. 3.4.

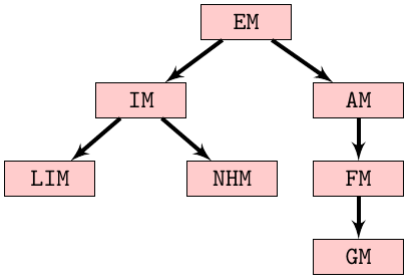

Fig. 3.4: A tree of the different constitutive equations implemented for solid materials in FEniCS Mechanics.

The inheritance for constitutive equations of fluids is shown in Fig. 3.5.

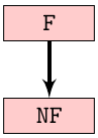

Fig. 3.5: A tree of the constitutive equations implemented for fluids in FEniCS Mechanics.

It can be seen that the classes defining different constitutive equations are grouped in such a way that common functions are defined in parent classes. This is more evident for solid materials. We see in Fig. 3.4 that all classes are derived from the `ElasticMaterial` class. Then, the second level of inheritance separates isotropic and anisotropic materials.

Do note that the user is not limited to the constitutive equations provided in `materials`. An example of providing a user-defined constitutive equation is given in *Custom Constitutive Equation*.

3.2 User Interface

The mechanics problem of interest is specified using a python dictionary referred to as `config`. Within this dictionary, the user provides information regarding the mesh, material properties, and details to formulate the boundary value problem. Each of these are defined as subdictionaries within `config`. Further details are provided below.

3.2.1 Mesh

The mesh subdictionary is where the user will provide all of the information regarding the discretization of the computational domain, and any tags necessary to identify various regions of the boundary. We now provide a list of keywords and their descriptions.

- `mesh_file`: the name of the file containing the mesh information (nodes and connectivity) in a format supported by `dolfin`. If the user is creating a `dolfin.Mesh` object within the same script, they can use the mesh object instead of a file name.
- `boundaries`: the name of the file containing the mesh function to mark different boundaries regions of the mesh. Similarly to `mesh_file`, the user can pass a `dolfin.MeshFunction` object directly if they are creating it within the same script.

3.2.2 Material

The user specifies the constitutive equation they wish to use, as well as any parameters that it requires in the material subdictionary. Below is a list of keywords and their descriptions.

- `type`: The class of material that will be used, e.g. elastic, viscous, viscoelastic, etc.
- `const_eqn`: The name of the constitutive equation to be used. User may provide their own class which defines a material instead of using those implemented in `fenicsmechanics.materials`. For a list of implemented materials, call `fenicsmechanics.list_implemented_materials()`.
- `incompressible`: True if the material is incompressible. An additional weak form for the incompressibility constraint will be added to the problem.
- `density`: Scalar specifying the density of the material.

Additional material parameters depend on the constitutive equation that is used. To see which other values are required, check the documentary of that specific constitutive equation.

3.2.3 Formulation

Details for the formulation of the boundary value problem are provided in the formulation subdictionary. This is where the user provides parameters for the time integration, any initial conditions, the type of finite element to be used, body force to be applied, boundary conditions, etc. A list of keywords and their descriptions is provided below.

- `time`: providing this dictionary is optional. If it is not provided, the problem is assumed to be a steady-state problem.
  - `unsteady`: A boolean value specifying if the problem is time dependent.
  - `dt`: The time step used for the numerical integrator.
  - `interval`: A list or tuple of length 2 specifying the time interval, i.e. `[t0, tf]`.

  - `theta`: The weight given to the current time step and subtracted from the previous, i.e.

$$\frac{dy}{dt} = \theta f(y_{n+1}) + (1-\theta) f(y_n).$$

    Note that $\theta = 1$ gives a fully implicit scheme, while $theta = 0$ gives a fully explicit one. It is optional for the user to provide this value. If it is not provided, it is assumed that $\theta = 1$.

  - `beta`: The β parameter used in the Newmark integration scheme. Note that the Newmark integration scheme is only used by `SolidMechanicsProblem`. Providing this value is optional. If it is not provided, it is assumed that $\beta = 0.25$.

  - `gamma`: The γ parameter used in the Newmark integration scheme. Note that the Newmark integration scheme is only used by `SolidMechanicsProblem`. Providing this value is optional. If it is not provided, it is assumed that $\gamma = 0.5$.

- `initial_condition`: a subdictionary containing initial conditions for the field variables involved in the problem defined. If this is not provided, all initial conditions are assumed to be zero.

  - `displacement`: A `dolfin.Coefficient` object specifying the initial value for the displacement.

  - `velocity`: A `dolfin.Coefficient` object specifying the initial value for the velocity.

  - `pressure`: A `dolfin.Coefficient` object specifying the initial value for the pressure.

- `element`: Name of the finite element to be used for the discrete function space. Currently, elements of the form `p<n>-p<m>` are supported, where `<n>` is the degree used for the vector function space, and `<m>` is the degree used for the scalar function space. If the material is not incompressible, only the first term should be specified. E.g., `p2`.

- `domain`: String specifying whether the problem is to be formulated in terms of Lagrangian, Eulerian, or ALE coordinates. Note that ALE is currently not supported.

- `inverse`: Boolean value specifying if the problem is an inverse elastostatics problem. If value is not provided, it is set to `False`. For more information, see Govindjee and Mihalic *[GM96]*.

- `body_force`: Value of the body force throughout the body.

- `bcs`: A subdictionary of `formulation` where the boundary conditions are specified. If this dictionary is not provided, no boundary conditions are applied and a warning is printed to the screen

  - `dirichlet`: A subdictionary of `bcs` where the Dirichlet boundary conditions are specified. If this dictionary is not provided, no Dirichlet boundary conditions are applied and a warning is printed to the screen.

    * `velocity`: List of velocity values for each Dirichlet boundary region specified. The order must match the order used in the list of region IDs.

    * `displacement`: List of displacement values for each Dirichlet boundary region specified. The order must match the order used in the list of region IDs.

    * `pressure`: List of pressure values for each Dirichlet boundary region specified. The order must match the order used in the list of pressure region IDs.

    * `regions`: List of the region IDs on which Dirichlet boundary conditions for displacement and velocity are to be imposed. These IDs must match those used by the mesh function provided. The order must match that used in the list of values (velocity and displacement).

    * `p_regions`: List of the region IDs on which Dirichlet boundary conditions for pressure are to be imposed. These IDs must match those used by the mesh function provided. The order must also match that used in the list of values (`pressure`).

  - `neumann`: A subdictionary of `bcs` where the Neumann boundary conditions are specified. If this dictionary is not provided, no Neumann boundary conditions are applied and a warning is printed to the screen.

* `regions`: List of the region IDs on which Neumann boundary conditions are to be imposed. These IDs must match those used by the mesh function provided. The order must match the order used in the list of types and values.
* `types`: List of strings specifying whether a `'pressure'`, `'piola'`, or `'cauchy'` is provided for each region. The order must match the order used in the list of region IDs and values.
* `values`: List of values for each Dirichlet boundary region specified. The order must match the order used in the list of region IDs and types.

CHAPTER

FOUR

EXAMPLES

Here, we show five examples of how to use FEniCS Mechanics. The first two are steady-state solid mechanics problems, with the second being an inverse elastostatics formulation. The third example is a time-dependent two-dimensional fluid mechanics problem. The fourth is a steady-state solid mechanics problem that is incrementally loaded. Thus, it is formulated as a time-dependent problem in FEniCS Mechanics. The last example shows the user how to define their own constitutive equation.

4.1 Steady-State Solid Mechanics

4.1.1 Mathematical Formulation

For the first example, we consider a steady-state solid mechanics problem. The domain is the unit square domain $[0,1] \times [0,1]$. The material will be modeled as an incompressible neo-Hookean material, which is given by the strain energy function

$$W = U(J) + \frac{1}{2}\mu \left(\text{tr } \tilde{\mathbf{C}} - 3 \right),$$

where $J = \det \mathbf{F}$ is the determinant of the deformation gradient, and $\tilde{\mathbf{C}} = J^{-2/3}\mathbf{F}^T\mathbf{F}$ is the isochoric component of the Cauchy-Green strain tensor. This leads to the first Piola-Kirchhoff stress tensor

$$\mathbf{P} = -\left[Jp + \frac{1}{3}\mu J^{-2/3} I_1(\mathbf{C}) \right] \mathbf{F}^{-T} + \mu J^{-2/3}\mathbf{F},$$

where we define the pressure as

$$p = -\frac{dU(J)}{dJ}.$$

Additionally, we require that $\mathbf{u}$ and p satisfy the incompressibility constraint given by

$$p - \frac{\kappa}{J} \ln J = 0.$$

Thus, κ is the bulk modulus of the material.

The material is clamped at $x = 0$, and a uniformly distributed load is applied at $x = 1$, i.e. $\bar{\mathbf{p}} = \bar{p}\mathbf{e}_1$, where $\bar{p}$ is constant, as shown in Figure Fig. 4.1. For this particular example, we use material properties representative of natural rubber. Hence, $\mu = 1.5$ MPa, and $\kappa = 10$ GPa. The load applied is $\bar{p} = 1$ MPa. Quadratic and linear finite elements are used for the displacement and pressure fields, respectively.

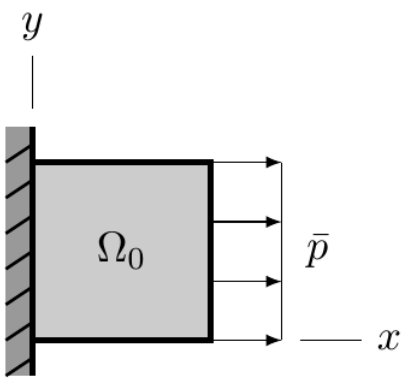

Fig. 4.1: Free body diagram for the loading of a unit square domain filled with natural rubber.

4.1.2 Code

First, import FEniCS Mechanics with the alias `fm`.

```
import fenicsmechanics as fm
```

In this demonstration, we will define three separate dictionaries that will be combined into one later. First, we define the material dictionary.

```
material = {
```

The problem is of ann elastic material.

```
'type': 'elastic',
```

We will be using the neo-Hookean constitutive equation. This model is already implemented in the `materials` submodule, so we can just provide a string.

```
'const_eqn': 'neo_hookean',
```

We must also let FEniCS Mechanics know that we wish to model the material as incompressible.

```
'incompressible': True,
```

Given the formulation above for the incompressibility constraint, we must provide the bulk modulus of the material.

```
'kappa': 10e9, # Pa
```

Also, the shear modulus of the material is provided.

```
    'mu': 1.5e6 # Pa
}
```

The material dictionary is defined all together below.

```
material = {
    'type': 'elastic',
    'const_eqn': 'neo_hookean',
    'incompressible': True,
    'kappa': 10e9, # Pa
```

```
    'mu': 1.5e6 # Pa
}
```

Now, we get the name of the files storing the mesh and the boundary tagging by using the `get_mesh_file_names` function provided. We also define the mesh dictionary by telling FEniCS Mechanics where the files containing the mesh and the boundary tags are located.

```
mesh_file, boundaries = fm.get_mesh_file_names("unit_domain", ret_facets=True,
                                               refinements=[20, 20])
mesh = {
    'mesh_file': mesh_file,
    'boundaries': boundaries
}
```

Last, but not least, is the formulation dictionary used to define the weak form for the problem.

```
formulation = {
```

First, we tell FEniCS Mechanics that we wish to a quadratic finite element for the displacement, and a linear finite element for the pressure.

```
'element': 'p2-p1',
```

Then, we specify that the mathematical formulation should be done in the reference (Lagrangian) configuration.

```
'domain': 'lagrangian',
```

Now, we define the boundary conditions. First up are the Dirichlet boundary conditions.

```
'bcs': {
    'dirichlet': {
```

We will apply a homogeneous boundary condition at $x = 0$.

```
'displacement': [[0.0, 0.0]],
```

To identify this region, we must use the same tags that were provided for the `'boundaries'`. In this case, this region was tagged with the integer 1.

```
    'regions': [1]
},
```

Next, we define the Neumann boundary conditions.

```
'neumann': {
```

We are applying a Cauchy traction of $\mathbf{p} = 10^6\mathbf{e}_1$ at $x = 1$.

```
            'values': [[1e6, 0.0]],
            'regions': [2],
            'types': ['piola']
        }
    }
}
```

The formulation dictionary is defined all together below.

```
formulation = {
    'element': 'p2-p1',
    'domain': 'lagrangian',
    'bcs': {
        'dirichlet': {
            'displacement': [[0.0, 0.0]],
            'regions': [1]
        },
        'neumann': {
            'values': [[1e6, 0.0]],
            'regions': [2],
            'types': ['piola']
        }
    }
}
```

We now combine all three dictionaries into one by the name of `config`.

```
config = {
    'material': material,
    'mesh': mesh,
    'formulation': formulation
}
```

We can create a `SolidMechanicsProblem` object to define the necessary UFL objects for the problem we have defined.

```
problem = fm.SolidMechanicsProblem(config)
```

Last, we define the solver object by passing in the problem object and the name of the file we wish to save the displacement in, and tell FEniCS Mechanics to solve our problem.

```
solver = fm.SolidMechanicsSolver(problem, fname_disp="results/displacement.pvd")
solver.full_solve()
```

The solution is shown in Figure Fig. 4.2.

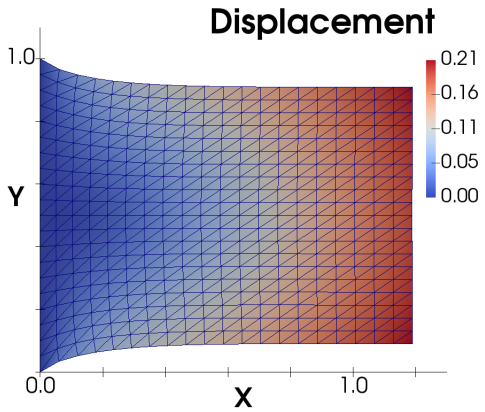

Fig. 4.2: Elongation of a unit square made of natural rubber.

4.2 Inverse Elastostatics

4.2.1 Mathematical Formulation

For the second example, we consider the problem of determining the unloaded configuration of a material given the loaded configuration and the loads that it is undergoing. The loaded geometry desired here is an L-shape as shown in Figure Fig. 4.3.

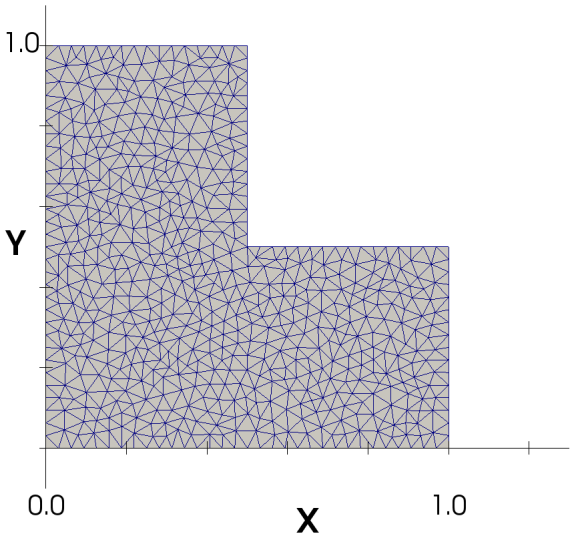

Fig. 4.3: The mesh of the loaded configuration.

In other words, we want the material to take this shape after applying the distributed load given by $\bar{\mathbf{p}} = -10^6$ N/m at $x = 1$, and imposing a homogeneous Dirichlet boundary condition at $y = 1$.

Because this is an inverse elastostatics problem, the Cauchy stress tensor is given by

$$\mathbf{T} = -\left[p + \frac{1}{3}\mu j^{-1/3} i_2\right]\mathbf{I} + \mu j^{5/3}\mathbf{c}^{-1},$$

where $j = \det \mathbf{f}$, $\mathbf{c} = \mathbf{f}^T\mathbf{f}$, and i_2 is the invariant of $\mathbf{f}$ with $\mathbf{f} = \mathbf{F}^{-1}$. Further details of the formulation of an inverse elastostatics problem can be found in Govindjee et al. *[GM96]*.

4.2.2 Code

First, we import FEniCS Mechanics with the alias `fm`.

```
import fenicsmechanics as fm
```

Now we get the name of the files storing the mesh and the boundary tagging by using the `get_mesh_file_names` function provided.

```
mesh_file, boundaries = fm.get_mesh_file_names("lshape", ret_facets=True,
                                               refinements="fine")
```

Next, we start to define the problem defined before. We do this by defining the `config` dictionary all at once. Here we will traverse it line by line:

```
config = {
```

First up is the material subdictionary.

```
'material':
{
```

The problem is of an elastic material. Thus,

```
'type': 'elastic',
```

We will be using the neo-Hookean constitutive equation. This model is already implemented in the `materials` submodule, so we can just provide a string.

```
'const_eqn': 'neo_hookean',
```

We must also let FEniCS Mechanics know that we wish to model the material as incompressible.

```
'incompressible': True,
```

Given the formulation above for the incompressibility constraint, we must provide the bulk modulus of the material.

```
'kappa': 10e9,
```

Also, the shear modulus of the material is provided.

```
   'mu': 1.5e6
},
```

Now that the material has been specified, we tell FEniCS Mechanics where the files containing the mesh and the boundary tags are located. This is where we use the variable `mesh_dir` that we defined before.

```
'mesh': {
    'mesh_file': mesh_file,
    'boundaries': boundaries
},
```

Last, but not least, is the formulation used to define the weak form for the problem.

```
'formulation': {
```

First, we tell FEniCS Mechanics that we wish to linear finite elements for both the displacement and the pressure.

```
'element': 'p1-p1',
```

Then, we specify that the mathematical formulation should be done in the reference (Lagrangian) configuration.

```
'domain': 'lagrangian',
```

We want to use the inverse elastotatics formulation as in *[GM96]*. Thus, we define this as an inverse problem.

```
'inverse': True,
```

Now, we define the boundary conditions. First up are the Dirichlet boundary conditions.

```
'bcs': {
    'dirichlet': {
```

We will apply a homogeneous boundary condition at $y = 1$.

```
'displacement': [[0., 0.]],
```

To identify this region, we must use the same tags that were provided for the `'boundaries'`. In this case, this region was tagged with the integer 1.

```
    'regions': [1]
},
```

Next, we define the Neumann boundary conditions.

```
'neumann': {
```

We are applying a Cauchy traction of $\mathbf{p} = -10^5\mathbf{e}_2$ at $x = 1$.

```
                'values': [[0., -1e5]],
                'regions': [2],
                'types': ['cauchy']
            }
        }
    }
}
```

Below is the definition of the `config` dictionary all together:

```
config = {
    'material':
    {
        'type': 'elastic',
        'const_eqn': 'neo_hookean',
        'incompressible': True,
        'kappa': 10e9,
        'mu': 1.5e6
    },
    'mesh': {
        'mesh_file': mesh_file,
        'boundaries': boundaries
    },
    'formulation': {
        'element': 'p1-p1',
        'domain': 'lagrangian',
        'inverse': True,
        'bcs': {
            'dirichlet': {
                'displacement': [[0., 0.]],
                'regions': [1]
            },
            'neumann': {
                'values': [[0., -1e5]],
                'regions': [2],
                'types': ['cauchy']
            }
        }
    }
}
```

Now, we can create a `SolidMechanicsProblem` object to define the necessary UFL objects for the problem we have defined.

```
problem = fm.SolidMechanicsProblem(config)
```

Last, we define the solver object by passing in the problem object and the name of the file we wish to save the displacement in, and tell FEniCS Mechanics to solve our problem.

```
solver = fm.SolidMechanicsSolver(problem, fname_disp="results/unloaded_config.pvd")
solver.full_solve()
```

This solution is shown in Figure Fig. 4.4.

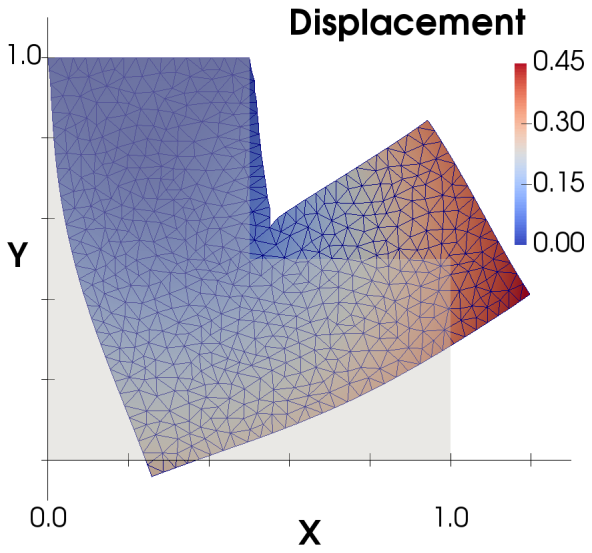

Fig. 4.4: The loaded configuration desired (transparent gray) along with the solution to the inverse elastostatics problem (color contour).

Now, to see how well the formulation of the inverse elastostatics problem works, we will reload the material. First, we import the `ALE` submodule from `dolfin` to move the mesh based on the solution of the *inverse* problem, as well as the `Mesh` class to create a copy of an existing mesh.

```
from dolfin import ALE, Mesh
```

Now, move the mesh and create a deep copy.

```
ALE.move(problem.mesh, problem.displacement)
mesh_copy = Mesh(problem.mesh)
```

Note that we can use the same `config` variable by only changing necessary variables. First, we provide it with the mesh copy.

```
config['mesh']['mesh_file'] = mesh_copy
```

Then, we change the *inverse* flag to `False`, since we are interested in solving the forward problem.

```
config['formulation']['inverse'] = False
```

Now, create new problem and solver objects, and solve the problem.

```
problem = fm.SolidMechanicsProblem(config)
solver = fm.SolidMechanicsSolver(problem, fname_disp="results/loaded_config.pvd")
solver.full_solve()
```

The solution to the reloading problem is shown in Figure Fig. 4.5.

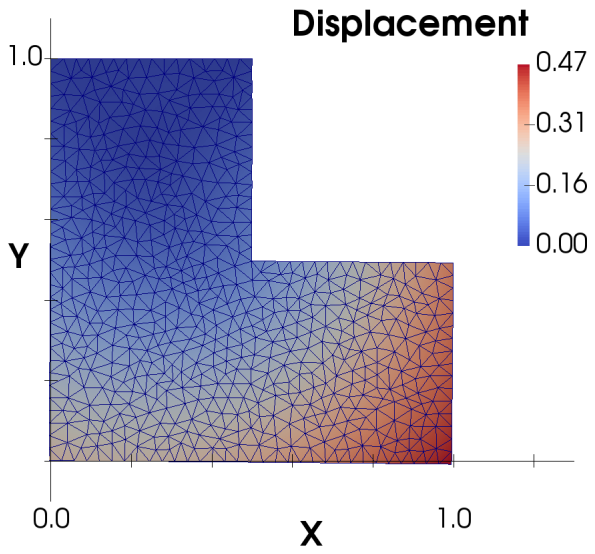

Fig. 4.5: The loaded configuration starting with the solution to the inverse elastostatics problem.

It is clear that reloading the solution to the inverse elastostatics problem yields a geometry that is nearly identical to the desired unloaded configuration.

4.3 Time-dependent Fluid Mechanics

4.3.1 Mathematical Formulation

In this example, we consider fluid flow through a pipe formulated as a two-dimensional problem. The domain is taken to be $[0, 10] \times [0, 1]$. The mesh of the domain is shown in Figure Fig. 4.6.

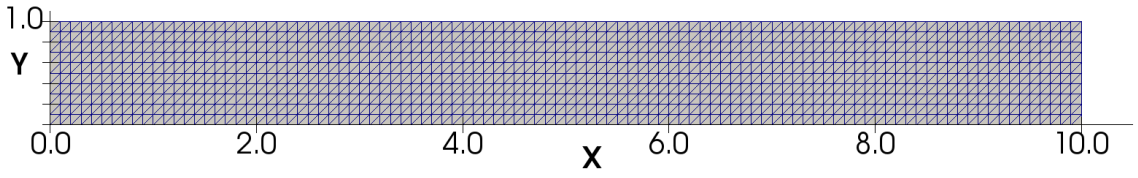

Fig. 4.6: Mesh generated with `dolfin`.

No slip conditions are imposed at $y = 0$ and $y = 1$. Furthermore, a homogeneous Dirichlet boundary condition is imposed for the pressure at $x = 10$. A traction boundary condition,

$$\bar{\mathbf{t}} = -\bar{p}\mathbf{n}$$

with

$$\bar{p} = 1.0 + \sin(2\pi t),$$

is imposed at the inlet ($x = 0$).

For this example, the fluid is assumed to be an incompressible Newtonian fluid, and the density and dynamic viscosity are taken to be $\rho = 1$ $\mu = 0.1$, respectively.

Both the velocity and the pressure are given a zero initial condition. We run the simulation for four cycles, $t_0 = 0$ to $t_f = 4.0$, to see how the flow develops. The time step is taken as $\Delta t = 0.01$, and a fully implicit scheme is used.

In addition to demonstrating how to run a Fluid Mechanics simulation, we show how to generate a mesh and mark the boundaries with tools from `dolfin`, and passing these objects to `fenicsmechanics`.

4.3.2 Code

As before, import `fenicsmechanics` with the alias `fm`.

```
import fenicsmechanics as fm
```

Now, we import `dolfin` with the alias `dlf`, and create a rectangle mesh with 100 and 10 intervals along the x and y directions, respectively.

```
import dolfin as dlf
mesh = dlf.RectangleMesh(dlf.Point(), dlf.Point(10, 1), 100, 10)
```

Next, we create a `MeshFunction` object that we will use to mark the different boundary regions. We tag every facet with 0.

```
boundaries = dlf.MeshFunction("size_t", mesh, 1)
boundaries.set_all(0)
```

Then, we use the `CompiledSubDomain` class to define the inlet, outlet, and no-slip regions of the boundary.

```
inlet = dlf.CompiledSubDomain("near(x[0], 0.0)")
outlet = dlf.CompiledSubDomain("near(x[0], 10.0)")
no_slip = dlf.CompiledSubDomain("near(x[1], 0.0) || near(x[1], 1.0)")
```

Once the different regions are defined, we can mark them with their own integer values for identification.

```
inlet.mark(boundaries, 1)
outlet.mark(boundaries, 2)
no_slip.mark(boundaries, 3)
```

FEniCS Mechanics will accept `dolfin.Mesh` and `dolfin.MeshFunction` objects in place of strings specifying file names. Thus, we define the mesh dictionary with the following:

```
mesh_dict = {
    'mesh_file': mesh,
    'boundaries': boundaries
}
```

Next, we define the material dictionary by choosing the type to be viscous, with the incompressible Newtonian constitutive equation for fluids, and the density and dynamic viscosity values of 1.0 and 0.01, respectively. Note that we can model Stokes' flow by switching `'newtonian'` with `'stokes'`.

```
material_dict = {
    'type': 'viscous',
    'const_eqn': 'newtonian',
    'incompressible': True,
    'density': 1, # kg/m^3
    'mu': 0.01   # Pa*s
}
```

We use quadratic and linear finite elements for the velocity and pressure fields, respectively, and tell FEniCS mechanics to use Eulerian coordinates to formulate the problem. We also specify parameters for the time integrator in the `time` subdictionary.

```
formulation_dict = {
    'element': 'p2-p1',
    'domain': 'eulerian',
    'time': {
        'unsteady': True,
        'interval': [0.0, 4.0],
        'dt': 0.01
    },
```

Within the formulation dictionary, we specify the Dirichlet and Neumann boundary conditions. The Dirichlet boundary conditions are the no-slip conditions, and the zero pressure at the outlet. Note that the regions for pressure are specified under the key `p_regions`. Thus, the Dirichlet boundary conditions for velocity and pressure need not be the same. We also specify the pressure traction at the inlet.

```
    'bcs': {
        'dirichlet': {
            'velocity': [[0.0, 0.0]],
            'regions': [3],
            'pressure': [0.0],
            'p_regions': [2]
        },
        'neumann': {
            'values': ["1.0 + sin(2.0*pi*t)"],
            'regions': [1],
            'types': ['pressure']
        }
    }
}
```

As the other examples, we combine all three dictionaries into one and call it `config`.

```
config = {
    'mesh': mesh_dict,
    'material': material_dict,
    'formulation': formulation_dict
}
```

We then create the problem and solver objects, and ask FEniCS Mechanics to solve the problem we have defined.

```
problem = fm.FluidMechanicsProblem(config)
solver = fm.FluidMechanicsSolver(problem, fname_vel="results/v.pvd",
                                 fname_pressure="results/p.pvd")
solver.full_solve()
```

The velocity field at $t = 4.0$ is shown in the subdomain $[0, 5] \times [0, 1]$ in Figure Fig. 4.7. As expected, we can see a symmetry with respect to the line $y = 0.5$.

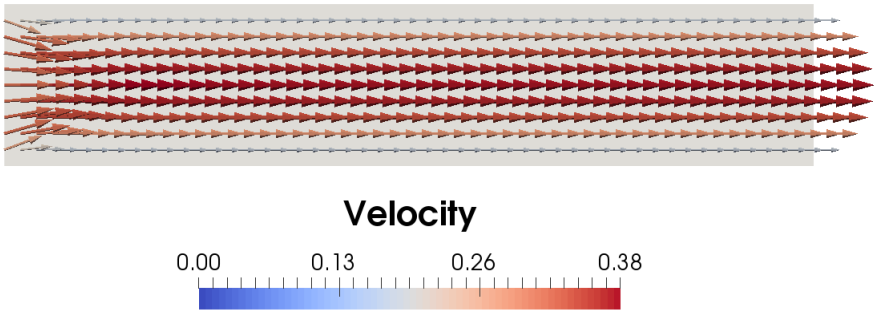

Fig. 4.7: Velocity field at $t = 4.0$ for the subdomain $[0, 5] \times [0, 1]$

4.4 Time-dependent Anisotropic Material

4.4.1 Mathematical Formulation

In this example, we use an ellipsoid as an ideal geometry model for the human heart as is done in Land et al. *[LGA+15]*. This geometry is shown in Figure Fig. 4.8.

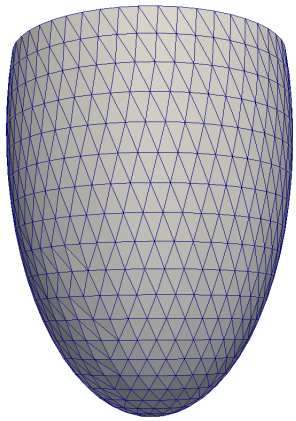

Fig. 4.8: Mesh used as an ideal left ventricle of the human heart.

The cardiac tissue is modeled using the strain energy function proposed by Guccione et al. *[GCM95]*, which is given by

$$W = U(J) + Ce^{Q},$$

where

$$Q = b_f \bar{E}_{11}^2 + b_t \left(\bar{E}_{22}^2 + \bar{E}_{33}^2 + 2\bar{E}_{23}^2 \right) + 2b_{fs} \left(\bar{E}_{12}^2 + \bar{E}_{13}^2 \right),$$

with

$$\bar{\mathbf{E}} = \mathbf{Q}\mathbf{E}\mathbf{Q}^T.$$

Note that $\mathbf{Q}$ is an orthogonal tensor that transforms the components of the Lagrangian strain tensor, $\mathbf{E}$, from the global coordinate system to the local fiber direction frame. For more details, please read the paper by Guccione et al.

4.4.2 Code

Import FEniCS Mechanics with the alias `fm`, and use the function 'get_mesh_file_names' provided to get the name of the file where the mesh, boundaries, and fiber direction information is stored. Note that all of this information is in the same file since this is an HDF5 file.

```
import fenicsmechanics as fm

mesh_file = fm.get_mesh_file_names("ellipsoid", refinements="1000um", ext="h5")
```

In this demonstration, we will define three separate dictionaries that will be combined into one later. First, we define the material dictionary.

```
mat_dict = {
```

We will be using the constitutive equation presented by Guccione et al. *[GCM95]*. This model is already implemented in the `materials` submodule, so we can just provide a string with the name of the model.

```
'const_eqn': 'guccione',
```

The problem is of an elastic material.

```
'type': 'elastic',
```

We must also let FEniCS Mechanics know that we wish to model the material as incompressible.

```
'incompressible': True,
```

Given that we are incrementally solving a steady-state problem, we wish to cancel the inertial term. Hence, we set the density of the material to zero.

```
'density': 0.0,
```

Now, we provide all of the coefficients that are specific to the constitutive equation of choice. See *[GCM95]* for further details on their physical significance.

```
'bt': 1.0,
'bf': 1.0,
'bfs': 1.0,
'C': 10.0,
```

This example involves a material that is not isotropic, and hence requires fiber information. For this, we define a subdictionary within the `material` subdictionary.

```
'fibers': {
```

Here, we provide the name of the files that contain the vector field information necessary to define the stress tensor, as well as the names we wish to give to these fields.

```
'fiber_files': mesh_file,
'fiber_names': [['fib1', 'fib2', 'fib3'],
                ['she1', 'she2', 'she3']],
```

We also tell FEniCS Mechanics that the fiber vector fields are assumed to be constant over each element and that they are provided as mesh functions.

```
    'elementwise': True
    }
}
```

The entire material dictionary is defined all together below.

```
mat_dict = {
    'const_eqn': 'guccione',
    'type': 'elastic',
    'incompressible': True,
    'density': 0.0,
    'bt': 1.0,
    'bf': 1.0,
    'bfs': 1.0,
    'C': 10.0,
    'fibers': {
        'fiber_files': mesh_file,
        'fiber_names': [['fib1', 'fib2', 'fib3'],
                        ['she1', 'she2', 'she3']],
        'elementwise': True
        }
    }
```

We define the mesh dictionary by telling FEniCS Mechanics where the file containing the mesh and the boundary tags are located. Note that this information is stored in a single HDF5 file for this example. FEniCS Mechanics will recognize this and open the file just once for efficiency.

```
mesh_dict = {
    'mesh_file': mesh_file,
    'boundaries': mesh_file
}
```

The last dictionary to define is the formulation dictionary.

```
formulation_dict = {
```

This incremental steady-state problem is being treated as a time-dependent problem and thus requires a `time` subdictionary within the formulation dictionary. Here, we specify the size of the time interval, Δt, and the initial and final time for the simulation.

```
'time': {
    'dt': 0.01,
    'interval': [0., 1.]
},
```

Note that no value was provided for θ, β, γ. Thus, the default values, $\theta = 1$, $\beta = 0.25$, and $\gamma = 0.5$, are used.

Now, we specify a quadratic finite element for the displacement, and a linear finite element for the pressure.

```
'element': 'p2-p1',
```

Then, we specify that the mathematical formulation should be done in the reference (Lagrangian) configuration.

```
'domain': 'lagrangian',
```

Now, we define the boundary conditions. First up are the Dirichlet boundary conditions. We will apply a zero Dirichlet boundary condition at the base of the ideal ventricle. The integer value 10 was used to identify this region.

```
'bcs':{
    'dirichlet': {
        'displacement': [[0., 0., 0.]],
        'regions': [10], # Integer ID for base plane
    },
```

Next, we define the Neumann boundary conditions. We apply a pressure that is incremented from 0 to 10 at the inner wall of the ideal ventricle. The integer value 20 was used to identify this region.

```
        'neumann': {
            'regions': [20], # Integer ID for inner surface
            'types': ['pressure'],
            'values': ['10.0*t']
        }
    }
}
```

The formulation dictionary is defined all together below.

```
formulation_dict = {
    'time': {
        'dt': 0.01,
        'interval': [0., 1.]
    },
    'element': 'p2-p1',
    'domain': 'lagrangian',
    'bcs':{
        'dirichlet': {
            'displacement': [[0., 0., 0.]],
            'regions': [10], # Integer ID for base plane
        },
        'neumann': {
            'regions': [20], # Integer ID for inner surface
            'types': ['pressure'],
            'values': ['10.0*t']
        }
    }
}
```

We now combine all three dictionaries into one by the name of `config`.

```
config = {
    'material': mat_dict,
    'mesh': mesh_dict,
    'formulation': formulation_dict
}
```

We can create a `SolidMechanicsProblem` object to define the necessary UFL objects for the problem we have defined.

```
problem = fm.SolidMechanicsProblem(config)
```

Last, we define the solver object by passing in the problem object and the name of the file we wish to save the displacement in, choose the linear solver we want to use, and tell FEniCS Mechanics to solve our problem.

```
solver = fm.SolidMechanicsSolver(problem, fname_disp='results/displacement.pvd')
solver.set_parameters(linear_solver="superlu_dist")
solver.full_solve()
```

The solution is shown on the right of Figure Fig. 4.9.

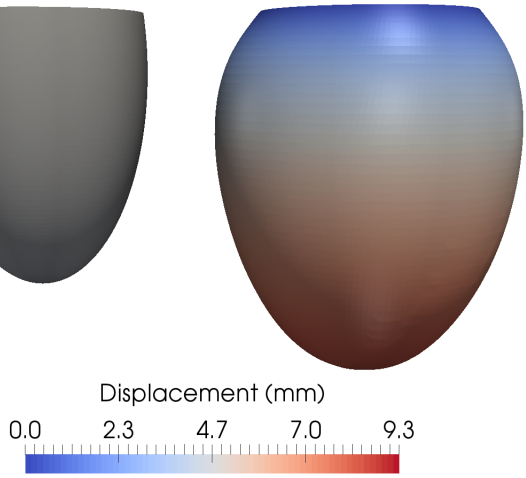

Fig. 4.9: The unloaded (right) and loaded (left) geometries.

The choice of linear solvers, and other solver parameters, depends on your linear algebra backend. To see a list of linear solvers available, use the following:

```
import dolfin as dlf
dlf.list_linear_solver_methods()
```

4.5 Custom Constitutive Equation

In this example, we will not show the full script for running a simulation with FEniCS Mechanics. Instead, we will show the minimal code required when a user wants to provide their own constitutive equation.

Though the constitutive equation for linear isotropic materials has already been implemented in the `materials` submodule, we will use this constitutive equation as an example. In order to use a custom constitutive equation, the user must define it as a python class. Thus, we start with

```
class MyMaterial:
```

The two member functions that are required to make this constitutive model work are `__init__`, and `stress_tensor`. If the material is to be modeled as incompressible, the third member function `incompressibilityConstraint` must also be defined. First, let us look at the constructor function, `__init__`.

```
def __init__(self, inverse=False, **params):
    self._parameters = dict()
    self._parameters.update(params)
    self._inverse = inverse
    self._incompressible = params['incompressible'] \
          if 'incompressible' in params else False

    if 'kappa' not in params:
```

```
        kappa = self._parameters['first_lame'] \
              + 2.0*self._parameters['second_lame']/3.0
        self._parameters.update(kappa=kappa)
```

The parameter `inverse` specifies if the stress tensor is to be formulated for an inverse elastostatics problem. Then, `**params` allows for an arbitrary number of keyword arguments to be passed when initializing an object of this class. The particular keys used to identify the material parameters are up to the user. Here, we used `first_lame` and `second_lame`.

The first line of this method creates a member object by the name `_parameters` as a dictionary object. This is necessary for the way we will be accessing the material parameters in the other functions. We then use the member function of python dictionary objects `update` to include all of the keyword arguments passed when the material object was initialized. We also define a member object `_inverse` that will store the Boolean value of `inverse` used during initialization. We now check if the material was specified as incompressible with a conditional expressions to see if the keyword parameter `incompressible` was provided. Note that we assume that the material is not incompressible if the keyword parameter was not provided. The last three lines check if the bulk modulus was given under the key `kappa`. If not, the bulk modulus is computed based on the first and second Lame parameters, and included in the `_parameters` dictionary.

Next, we must define the stress tensor. Note that the specific stress tensor defined here should be consistent with the formulation used (Eulerian or Lagrangian). Since this is a linear material, Eulerian and Lagrangian formulations are identical. We now define the function.

```
def stress_tensor(self, F, J, p=None):
    import ufl
    dim = ufl.domain.find_geometric_dimension(F)
    la = self._parameters['first_lame']
    mu = self._parameters['second_lame']

    from dolfin import Identity, sym, inv, tr

    I = Identity(dim)
    if self._inverse:
        epsilon = sym(inv(F)) - I
    else:
        epsilon = sym(F) - I

    if self._incompressible:
        T = -p*I + 2.0*mu*epsilon
    else:
        T = la*tr(epsilon)*I + 2.0*mu*epsilon
```

We now traverse this function line by line. Starting with the definition of the function itself.

```
def stress_tensor(self, F, J, p=None):
```

Here, `F` is the deformation gradient, `J` the Jacobian, and `p` the pressure for incompressible problems. Next, we import the `ufl` module from the FEniCS Project to determine the geometric dimension of the problem instead of hard coding this in our definition.

```
import ufl
dim = ufl.domain.find_geometric_dimension(F)
```

Next, we retrieve the material constants for our constitutive equation, λ and μ.

```
la = self._parameters['first_lame']
mu = self._parameters['second_lame']
```

Here, we use key values that are different than those used by the `materials` submodule to demonstrate that the user can use any key values so long as they are consistent. A `KeyError` will be raised if these keyword arguments were not provided when the object was initialized.

Now, we import objects from `dolfin` that we will need to define our constitutive equation.

```
from dolfin import Identity, sym, inv, tr
```

Now we can start to define the stress tensor itself. First, we define the linearized strain tensor in terms of the deformation gradient, which is given by

$$\boldsymbol{\varepsilon} = \mathbf{F} - \mathbf{I} = \mathbf{f}^{-1} - \mathbf{I},$$

where $\mathbf{f}$ is used for inverse elastostatics problems. We must use a conditional state to check which formulation we should use.

```
I = Identity(dim)
if self._inverse:
    epsilon = sym(inv(F)) - I
else:
    epsilon = sym(F) - I
```

Finally, we define the Cauchy stress tensor given by

$$\mathbf{T} = -p\mathbf{I} + 2\mu\boldsymbol{\varepsilon}$$

when the material is incompressible, and

$$\mathbf{T} = \lambda\,(\mathrm{tr}\,\boldsymbol{\varepsilon}) + 2\mu\boldsymbol{\varepsilon}$$

otherwise. This is done with

```
if self._incompressible:
    T = -p*I + 2.0*mu*epsilon
else:
    T = la*tr(epsilon) + 2.0*mu*epsilon
```

and this object is returned.

```
return T
```

The incompressibility condition for a linear material is

$$\phi(\mathbf{u}) - \frac{1}{\kappa}p = 0,$$

where κ is the bulk modulus of the material, and $\phi(\mathbf{u}) = \mathrm{div}\ \mathbf{u}$. The user must return ϕ, and FEniCS Mechanics will formulate the corresponding weak form. This is done with

```
def incompressibilityCondition(self, u):
    from dolfin import div
    return div(u)
```

The only part of the `config` dictionary that will change is the material subdicionary. Thus, we provide an example below.

```
material = {
    'type': 'elastic',
    'const_eqn': MyMaterial,
    'density': 10.0,
    'first_lame': 1e9,
    'second_lame': 1.5e6
}
```

The combined code for the class definition is shown below.

```
class MyMaterial:

    def __init__(self, inverse=False, **params):
        self._parameters = dict()
        self._parameters.update(params)
        self._inverse = inverse
        self._incompressible = params['incompressible'] \
              if 'incompressible' in params else False

        if 'kappa' not in params:
            kappa = self._parameters['first_lame'] \
                  + 2.0*self._parameters['second_lame']/3.0
            self._parameters.update(kappa=kappa)

    def stress_tensor(self, F, J, p=None):
        import ufl
        dim = ufl.domain.find_geometric_dimension(F)
        la = self._parameters['first_lame']
        mu = self._parameters['second_lame']

        from dolfin import Identity, sym, inv, tr

        I = Identity(dim)
        if self._inverse:
            epsilon = sym(inv(F)) - I
        else:
            epsilon = sym(F) - I

        if self._incompressible:
            T = -p*I + 2.0*mu*epsilon
        else:
            T = la*tr(epsilon)*I + 2.0*mu*epsilon

    def incompressibilityCondition(self, u):
        from dolfin import div
        return div(u)
```

The rest of the script to run the simulation can be written as in the other examples where the three dictionaries are combined into one, and the problem and solver objects are created.

CHAPTER

FIVE

APPLICATION PROGRAM INTERFACE

The difference classes in FEniCS Mechanics and member classes are listed here.

5.1 Problem Objects

The problem classes defining the weak form for mechanics problems are listed here.

5.1.1 Base Mechanics Problem

All mechanics problem objects are child classes of `basemechanics.BaseMechanicsProblem`.

class `fenicsmechanics.basemechanics.`**`BaseMechanicsProblem`**(*user_config*)

This is the base class for mechanics problems. Checking validity of the 'config' dictionary provided by users is done at this level since all mechanics problems are derived from this class. The derived classes will then define the variational problem using the FEniCS UFL language.

For details on the format of the `config` dictionary, check the documentation of the FEniCS Mechanics module by executing

```
>>> import fenicsmechanics as fm
>>> help(fm)
```

static **`apply_initial_conditions`**(*init_value*, *function*, *function0=0*)

Assign the initial values to field variables.

Parameters

- **`init_value`** (*`ufl.Coefficient`*, *`dolfin.Expression`*) – A function/expression that approximates the initial condition.
- **`function`** (*`dolfin.Function`*) – The function approximating a field variable in a mechanics problem.
- **`function0`** (*`dolfin.Function`*) – The function approximatinga field variable at the previous time step in a mechanics problem.

Returns

Return type None

`check_and_load_mesh`(*config*)

Check the mesh files provided, and load them. Also, extract the geometrical dimension for later use.

Parameters **config** (*dict*) – Dictionary describing the formulation of the mechanics problem to be simulated. Check the documentation of BaseMechanicsProblem to see the format of the dictionary.

Returns

Return type None

check_bcs (*config*)

Check the boundary conditions provided by the user in the config dictionary.

Parameters **config** (*dict*) – Dictionary describing the formulation of the mechanics problem to be simulated. Check the documentation of BaseMechanicsProblem to see the format of the dictionary.

Returns

Return type None

check_body_force (*config*)

Check if body force is specified. If it is, the type is checked, and converted to a ufl.Coefficient object when necessary. If it is not specified, it is set to None.

Parameters **config** (*dict*) – Dictionary describing the formulation of the mechanics problem to be simulated. Check the documentation of BaseMechanicsProblem to see the format of the dictionary.

Returns

Return type None

check_config (*user_config*)

Check that all parameters provided in 'user_config' are valid based on the current capabilities of the package. An exception is raised when a parameter (or combination of parameters) is (are) found to be invalid. Please see the documentation of fenicsmechanics for detailed information on the required values.

Parameters **user_config** (*dict*) – Dictionary describing the formulation of the mechanics problem to be simulated. Check the documentation of BaseMechanicsProblem to see the format of the dictionary.

Returns **config** – A copy of user_config with possibly new keys that are needed if they were not originally provided.

Return type dict

check_dirichlet (*config*)

Check if the number of parameters for each key in the dirichlet sub-dictionary are equal. If the key 'dirichlet' does not exist, it is added to the dictionary with the value None.

Parameters **config** (*dict*) – Dictionary describing the formulation of the mechanics problem to be simulated. Check the documentation of BaseMechanicsProblem to see the format of the dictionary.

Returns

Return type None

check_domain (*config*)

Check if the domain formulation specified is implemented for the material type.

Parameters **config** (*dict*) – Dictionary describing the formulation of the mechanics problem to be simulated. Check the documentation of BaseMechanicsProblem to see the format of the dictionary.

Returns

Return type None

check_finite_element(*config*)
Check the finite element specified for the numerical formulation of the problem.

Parameters **config**(*dict*) – Dictionary describing the formulation of the mechanics problem to be simulated. Check the documentation of BaseMechanicsProblem to see the format of the dictionary.

Returns

Return type None

check_initial_condition(*config*)
Check if initial condition values were provided. Insert 'None' for the values that were not provided.

Parameters **config**(*dict*) – Dictionary describing the formulation of the mechanics problem to be simulated. Check the documentation of BaseMechanicsProblem to see the format of the dictionary.

Returns

Return type None

check_material_const_eqn(*config*)
Check if the material type and the specific constitutive equation specified in the config dictionary are implemented, unless a class is provided. An exception is raised if an unknown material type and/or constitutive equation name is provided.

Parameters **config**(*dict*) – Dictionary describing the formulation of the mechanics problem to be simulated. Check the documentation of BaseMechanicsProblem to see the format of the dictionary.

Returns

Return type None

check_material_type(*config*)
Check if the material type specified is supported.

Parameters **config**(*dict*) – Dictionary describing the formulation of the mechanics problem to be simulated. Check the documentation of BaseMechanicsProblem to see the format of the dictionary.

Returns

Return type None

check_neumann(*config*)
Check if the number of parameters for each key in the neumann sub-dictionary are equal. If the key 'neumann' does not exist, it is added to the dictionary with the value None.

Parameters **config**(*dict*) – Dictionary describing the formulation of the mechanics problem to be simulated. Check the documentation of BaseMechanicsProblem to see the format of the dictionary.

Returns

Return type None

check_time_params(*config*)
Check the time parameters provided by the user in the config dictionary. Things this function does:

- If the problem is steady, the theta and dt values are set to 1.
- If the problem is unsteady and no value was provided for theta, an exception is raised.
- If the problem is unsteady, and a theta outside of the interval [0, 1] is provided, an exception is raised.

Parameters **config** (*dict*) – Dictionary describing the formulation of the mechanics problem to be simulated. Check the documentation of BaseMechanicsProblem to see the format of the dictionary.

Returns

Return type None

static define_ufl_neumann_form(*regions*, *types*, *values*, *domain*, *mesh*, *boundaries*, *F*, *J*, *xi*)

Define the UFL object representing the variational form of the Neumann boundary.

Parameters

- **regions** (*list, tuple*) – List of the region IDs on which Neumann boundary conditions are to be imposed. These IDs must match those used by the mesh function provided. The order must match the order used in the list of types and values.
- **types** (*list, tuple*) – List of strings specifying whether a 'pressure', 'piola', or 'cauchy' is provided for each region. The order must match the order used in the list of region IDs and values.
- **values** (*list, tuple*) – List of values (dolfin.Constant or dolfin.Expression) for each Dirichlet boundary region specified. The order must match the order used in the list of region IDs and types.
- **domain** (*str*) – String specifying whether the problem is to be formulated in terms of Lagrangian, Eulerian, or ALE coordinates. Note: ALE is currently not supported.
- **mesh** (*dolfin.Mesh*) – Mesh object used to define a measure.
- **boundaries** (*dolfin.MeshFunction*) – Mesh function used to tag different regions of the domain boundary.
- **F** (*ufl object*) – Deformation gradient.
- **J** (*ufl object*) – Determinant of the deformation gradient.
- **xi** (*dolfin.Argument*) – Test function used in variational formulation.

Returns **neumann_form** – The UFL Form object defining the variational term(s) corresponding to the Neumann boundary conditions.

Return type ufl.Form

update_bodyforce_time(*t*, *t0=None*)

Update the time parameter in the body force expression if it depends on time explicitly.

Parameters

- **t** (*float*) – The value to which the user wishes to update the time to.
- **t0** (*float (default None)*) – The previous time value.

Returns

Return type None

update_dirichlet_time(*t*)

Update the time parameter in the Dirichlet BCs that depend on time explicitly.

Parameters **t** (*float*) – The value to which the users wishes to update the time to.

Returns

Return type None

static update_form_time(*form*, *t*)
Update the time of the coefficient objects that depend on it.

Parameters

- **form** (*ufl.Form*) – Variational form for which the time is to be updated.
- **t** (*float*) – The value to which the time is to be updated.

Returns

Return type None

update_neumann_time(*t*, *t0=None*)
Update the time parameter in the Neumann BCs that depend on time explicitly.

Parameters

- **t** (*float*) – The value to which the user wishes to update the time to.
- **t0** (*float (default None)*) – The previous time value.

Returns

Return type None

update_time(*t*, *t0=None*)
Update the time parameter in the BCs that depend on time explicitly. Also, the body force expression if necessary.

Parameters

- **t** (*float*) – The value to which the user wishes to update the time to.
- **t0** (*float (default None)*) – The previous time value.

Returns

Return type None

5.1.2 Mechanics Problem

class fenicsmechanics.mechanicsproblem.**MechanicsProblem**(*user_config*)
This class represents the variational form of a continuum mechanics problem. The specific form and boundary conditions are generated based on definitions provided by the user in a dictionary of sub-dictionaries.

Refer to the documentation of the FEniCS Mechanics package for details on how to define a problem using the `config` dictionary.

```
>>> import fenicsmechanics as fm
>>> help(fm)
```

define_deformation_tensors()
Define kinematic tensors needed for constitutive equations. Tensors that are irrelevant to the current problem are set to 0, e.g. the deformation gradient is set to 0 when simulating fluid flow. Secondary tensors are added with the suffix "0" if the problem is time-dependent. The names of member data added to an instance of the MechanicsProblem class are:

- deformationGradient
- deformationGradient0
- velocityGradient
- velocityGradient0
- jacobian
- jacobian0

define_dirichlet_bcs()
Define a list of Dirichlet BC objects based on the problem configuration provided by the user, and add it as member data under 'dirichlet_bcs'. If no Dirichlet BCs are provided, 'dirichlet_bcs' is set to None.

define_forms()
Define all of the variational forms necessary for the problem specified by the user and add them as member data. The variational forms are those corresponding to the order reduction of the time derivative (for unsteady solid material simulations), the balance of linear momentum, and the incompressibility constraint.

define_function_spaces()
Define the vector (and scalar if incompressible) function spaces based on the degrees specified in config['formulation']['element'], and add them to the instance of MechanicsProblem as member data. If the material is not incompressible, the scalar function space is set to None.

define_functions()
Define the vector and scalar functions necessary to define the problem specified in the 'config' dictionary. Functions that are not needed are set to 0. This method calls a method to define the vector functions and another for the scalar functions.

define_material()
Create an instance of the class that defining the constitutive equation for the current problem and add it as member data under '_material'. All necessary parameters must be included in the 'material' subdictionary of 'config'. The specific values necessary depends on the constitutive equation used. Please check the documentation of the material classes provided in 'fenicsmechanics.materials' if using a built-in material.

define_scalar_functions()
Define the pressure function(s) necessary to define the problem specified in the 'config' dictionary. If the problem is not specified as incompressible, the scalar functions are set to 0. A secondary pressure function with the suffix "0" is also added if the problem is time-dependent to store the pressure values at the previous time step. The names of the member data added to an instance of the MechanicsProblem class are:

- test_scalar
- trial_scalar
- pressure
- pressure0

define_ufl_body_force()
Define the UFL object corresponding to the body force term in the weak form. The function exits if it has already been defined.

define_ufl_convec_accel()
Define the UFL object corresponding to the convective acceleration term in the weak form. The function exits if it has already been defined.

define_ufl_convec_accel_diff()
Define the UFL object corresponding to the Gateaux derivative of the convective acceleration term in the weak form. The function exits if it has already been defined.

define_ufl_equations()
Define all of the variational forms necessary for the problem specified in the 'config' dictionary. This function calls other functions to define the velocity, momentum, and incompressibility equations.

define_ufl_equations_diff()
Differentiate all of the variational forms with respect to appropriate fields variables and add as member data.

define_ufl_incompressibility_equation()
Define the variational form corresponding to the incompressibility constraint and add as member data.

define_ufl_local_inertia()
Define the UFL object corresponding to the local acceleration term in the weak form. The function exits if it has already been defined.

define_ufl_local_inertia_diff()
Define the UFL object that describes the matrix that results from taking the Gateaux derivative of the local acceleration term in the weak form. The function exits if it has already been defined.

define_ufl_momentum_equation()
Define the variational form corresponding to the balance of linear momentum and add as member data.

define_ufl_neumann_bcs()
Define the variational forms for all of the Neumann BCs given in the 'config' dictionary under "ufl_neumann_bcs". If the problem is time-dependent, a secondary variational form is defined at the previous time step with the name "ufl_neumann_bcs0".

define_ufl_neumann_bcs_diff()
Define the derivative(s) of the variational form of the Neumann BCs and add them as member data.

define_ufl_stress_work()
Define the UFL object corresponding to the stress tensor term in the weak form. The function exits if it has already been defined.

define_ufl_stress_work_diff()
Define the UFL object corresponding to the Gateaux derivative of the stress tensor term in the weak form. The function exits if it has already been defined.

define_ufl_velocity_equation()
Define the variational form for the reduction of order equation (equation that relates the velocity and displacement) and add as member data. Note that this is only necessary for time-dependent elastic materials. If this form is not necessary, it is set to 0.

define_vector_functions()
Define the vector functions necessary to define the problem specified in the 'config' dictionary. If the material is elastic, displacement and velocity functions are defined. Secondary displacement and velocity functions are defined with the suffix "0" if the problem is time-dependent to store the previous time step. Functions that are not needed are set to 0. The trial and test functions for the vector function space are also defined by this function. The names of the member data added to the instance of the MechanicsProblem class are:

- `test_vector`
- `trial_vector`
- `displacement`
- `displacement0`
- `velocity`
- `velocity0`

5.1.3 Solid Mechanics Problem

class `fenicsmechanics.solidmechanics.`**SolidMechanicsProblem**(*user_config*)

This class represents the variational form of a solid mechanics problem. The specific form and boundary conditions are generated based on definitions provided by the user in a dictionary of sub-dictionaries.

Refer to the documentation of the FEniCS Mechanics package for details on how to define a problem using the `config` dictionary.

```
>>> import fenicsmechanics as fm
>>> help(fm)
```

define_compressible_functions()

Define functions necessary to formulate a compressible solid mechanics problem. The names of the member data added to the instance of the SolidMechanicsProblem class are:

- `sys_u = ufl_displacement = displacement`: all point to the same displacement function, unlike the incompressible case.
- `sys_du = trial_vector`: trial function for vector function space
- `test_vector`: sub component of mixed test function
- `ufl_pressure = pressure = None`

If problem is unsteady, the following are also added:

- `sys_v0 = ufl_velocity0 = velocity0`
- `sys_a0 = ufl_acceleration0 = acceleration0`
- `sys_u0 = ufl_displacement0 = displacement0`

define_deformation_tensors()

Define kinematic tensors needed for constitutive equations. Secondary tensors are added with the suffix "0" if the problem is time-dependent. The names of member data added to an instance of the SolidMechanicsProblem class are:

- `deformationGradient`
- `deformationGradient0`
- `jacobian`
- `jacobian0`

define_dirichlet_bcs()

Define a list of Dirichlet BC objects based on the problem configuration provided by the user, and add it as member data under 'dirichlet_bcs'. If no Dirichlet BCs are provided, 'dirichlet_bcs' is set to None.

define_forms()

Define all of the variational forms necessary for the problem specified by the user and add them as member data. The variational forms are those corresponding to the balance of linear momentum and the incompressibility constraint.

define_function_assigners()

Create function assigners to update the current and previous time values of all field variables. This is specific to incompressible simulations since the mixed function space formulation requires the handling of the mixed functions and the copies of its subcomponents in a specific manner.

define_function_spaces()

Define the function space based on the degree(s) specified in config['formulation']['element'] and add it as member data. If the problem is specified as incompressible, a mixed function space made up of a vector

and scalar function spaces is defined. Note that this differs from MechanicsProblem since there, the vector and scalar function spaces are defined separately.

define_functions()
Define mixed functions necessary to formulate the problem specified in the 'config' dictionary. The sub-functions are also defined for use in formulating variational forms and saving results separately. Functions that are not needed are set to 0.

define_incompressible_functions()
Define mixed functions necessary to formulate an incompressible solid mechanics problem. The mixed function is also split using dolfin.split (for use in UFL variational forms) and u.split(), where u is a mixed function (for saving solutions separately). The names of the member data added to the instance of the SolidMechanicsProblem class are:

- `sys_u`: mixed function
- `ufl_displacement`: sub component corresponding to displacement
- `displacement`: copy of sub component for writing and assigning values
- `ufl_pressure`: sub component corresponding to pressure
- `pressure`: copy of sub component for writing and assigning values
- `sys_du`: mixed trial function
- `trial_vector`: sub component of mixed trial function
- `trial_scalar`: sub component of mixed trial function
- `test_vector`: sub component of mixed test function
- `test_scalar`: sub component of mixed test function

If problem is unsteady, the following are also added:

- `ufl_velocity0`: sub component corresponding to velocity
- `velocity0`: copy of sub component for writing and assigning values
- `ufl_acceleration0`: sub component corresponding to acceleration
- `acceleration0`: copy of sub component for writing and assigning values
- `sys_u0`: mixed function at previous time step
- `ufl_displacement0`: sub component at previous time step
- `displacement0`: copy of sub component at previous time step
- `ufl_pressure0`: sub component at previous time step
- `pressure0`: copy of sub component at previous time step

define_material()
Create an instance of the class that defines the constitutive equation for the current problem and add it as member data under '_material'. All necessary parameters must be included in the 'material' subdictionary of 'config'. The specific values necessary depends on the constitutive equation used. Please check the documentation of the material classes provided in 'fenicsmechanics.materials' if using a built-in material.

define_ufl_acceleration()
Define the acceleration based on the Newmark integration scheme and add as member data under 'ufl_acceleration'.

define_ufl_body_force()
Define the UFL object corresponding to the body force term in the weak form.

define_ufl_equations()
Define all of the variational forms necessary for the problem specified in the 'config' dictionary, as well as the mixed variational form.

define_ufl_equations_diff()
Differentiate all of the variational forms with respect to appropriate fields variables and add as member data.

define_ufl_local_inertia()
Define the UFL object corresponding to the local acceleration term in the weak form.

define_ufl_neumann_bcs()
Define the variational forms for all of the Neumann BCs given in the 'config' dictionary under "ufl_neumann_bcs". If the problem is time-dependent, a secondary variational form is defined at the previous time step with the name "ufl_neumann_bcs0".

define_ufl_stress_work()
Define the UFL object corresponding to the stress tensor term in the weak form.

5.1.4 Fluid Mechanics Problem

class fenicsmechanics.fluidmechanics.**FluidMechanicsProblem**(*user_config*)
This class represents the variational form of a fluid mechanics problem. The specific form and boundary conditions are generated based on definitions provided by the user in a dictionary of sub-dictionaries.

Refer to the documentation of the FEniCS Mechanics package for details on how to define a problem using the `config` dictionary.

```
>>> import fenicsmechanics as fm
>>> help(fm)
```

define_compressible_functions()
COMPRESSIBLE FLUIDS ARE CURRENTLY NOT SUPPORTED.

define_deformation_tensors()
Define kinematic tensors needed for constitutive equations. Secondary tensors are added with the suffix "0" if the problem is time-dependent. The names of member data added to an instance of FluidMechanicsProblem class are:

- `velocityGradient`
- `velocityGradient0`

define_dirichlet_bcs()
Define a list of Dirichlet BC objects based on the problem configuration provided by the user, and add it as member data under 'dirichlet_bcs'. If no Dirichlet BCs are provided, 'dirichlet_bcs' is set to None.

define_forms()
Define all of the variational forms necessary for the problem specified by the user and add them as member data. The variational forms are those corresponding to the order reduction of the time derivative (for unsteady solid material simulations), the balance of linear momentum, and the incompressibility constraint.

define_function_assigners()
Create function assigners to update the current and previous time values of all field variables. This is specific to incompressible simulations since the mixed function space formulation requires the handling of the mixed functions and the copies of its subcomponents in a specific manner.

define_function_spaces()
Define the function space based on the degree(s) specified in config['formulation']['element'] and add it

as member data. If the problem is specified as incompressible, a mixed function space made up of a vector and scalar function spaces is defined. Note that this differs from MechanicsProblem since there, the vector and scalar function spaces are defined separately.

define_functions()
Define the vector and scalar functions necessary to define the problem specified in the 'config' dictionary. Functions that are not needed are set to 0. This method calls one of two methods to define these functions based depending on whether the fluid is incompressible or not.

define_incompressible_functions()
Define mixed functions necessary to formulate an incompressible fluid mechanics problem. The mixed function is also split using dolfin.split (for use in UFL variational forms) and u.split(), where u is a mixed function (for saving solutions separately). The names of the member data added to the instance of the SolidMechanicsProblem class are:

- `sys_v`: mixed function
- `ufl_velocity`: sub component corresponding to velocity
- `velocity`: copy of sub component for writing and assigning values
- `ufl_pressure`: sub component corresponding to pressure
- `pressure`: copy of sub component for writing and assigning values
- `sys_du`: mixed trial function
- `trial_vector`: sub component of mixed trial function
- `trial_scalar`: sub component of mixed trial function
- `test_vector`: sub component of mixed test function
- `test_scalar`: sub component of mixed test function

If problem is unsteady, the following are also added:

- `sys_v0`: mixed function at previous time step
- `ufl_velocity0`: sub component corresponding to velocity
- `velocity0`: copy of sub component for writing and assigning values
- `ufl_pressure0`: sub component at previous time step
- `pressure0`: copy of sub component at previous time step

define_material()
Create an instance of the class that defines the constitutive equation for the current problem and add it as member data under '_material'. All necessary parameters must be included in the 'material' subdictionary of 'config'. The specific values necessary depends on the constitutive equation used. Please check the documentation of the material classes provided in 'fenicsmechanics.materials' if using a built-in material.

define_ufl_acceleration()
Define the acceleration based on a single step finite difference and add as member data under 'ufl_acceleration'.

ufl_acceleration = (ufl_velocity - ufl_velocity0)/dt,

where dt is the size of the time step.

define_ufl_body_force()
Define the UFL object corresponding to the body force term in the weak form.

define_ufl_convec_accel()
Define the UFL object corresponding to the convective acceleration term in the weak form.

define_ufl_equations()
Define all of the variational forms necessary for the problem specified in the 'config' dictionary, as well as the mixed variational form.

define_ufl_equations_diff()
Differentiate all of the variational forms with respect to appropriate fields variables and add as member data.

define_ufl_local_inertia()
Define the UFL object corresponding to the local acceleration term in the weak form.

define_ufl_neumann_bcs()
Define the variational forms for all of the Neumann BCs given in the 'config' dictionary under "ufl_neumann_bcs". If the problem is time-dependent, a secondary variational form is defined at the previous time step with the name "ufl_neumann_bcs0".

define_ufl_stress_work()
Define the UFL object corresponding to the stress tensor term in the weak form.

5.2 Solver Objects

5.2.1 Mechanics Solver

class fenicsmechanics.mechanicssolver.**MechanicsBlockSolver**(*mechanics_problem*, *fname_disp=None*, *fname_vel=None*, *fname_pressure=None*, *fname_hdf5=None*, *fname_xdmf=None*)
This class assembles the UFL variational forms from a MechanicsProblem object, and calls solvers to solve the resulting (nonlinear) algebraic equations. If the problem is time-dependent, this class loops through the time interval specified in the 'config' dictionary. An LGMRES algorithm from CBC-Block is used for time-dependent problems. For steady problems, the user may choose from the linear solvers available through dolfin. Furthermore, Newton's method is used to solve the resulting algebraic equations.

nonlinear_solve(*lhs*, *rhs*, *bcs*, *nonlinear_tol=1e-10*, *iter_tol=1e-08*, *maxNonlinIters=50*, *maxLinIters=200*, *show=0*, *print_norm=True*, *lin_solver='mumps'*)
Solve the nonlinear system of equations using Newton's method.

Parameters

- **lhs** (*ufl.Form, list*) – The definition of the left-hand side of the resulting linear system of equations.
- **rhs** (*ufl.Form, list*) – The definition of the right-hand side of the resulting linear system of equations.
- **bcs** (*dolfin.DirichletBC, list*) – Object specifying the Dirichlet boundary conditions of the system.
- **nonlinear_tol** (*float (default 1e-10)*) – Tolerance used to terminate Newton's method.
- **iter_tol** (*float (default 1e-8)*) – Tolerance used to terminate the iterative linear solver.
- **maxNonlinIters** (*int (default 50)*) – Maximum number of iterations for Newton's method.

- **maxLinIters** (*int (default 200)*) – Maximum number of iterations for iterative linear solver.
- **show** (*int (default 0)*) – Amount of information for iterative.LGMRES to show. See documentation of this class for different log levels.
- **print_norm** (*bool (default True)*) – True if user wishes to see the norm at every linear iteration and False otherwise.
- **lin_solver** (*str (default "mumps")*) – Name of the linear solver to be used for steady compressible elastic problems. See the dolfin.solve documentation for a list of available linear solvers.

Returns

Return type None

solve(*nonlinear_tol=1e-10*, *iter_tol=1e-08*, *maxNonlinIters=50*, *maxLinIters=200*, *show=0*, *print_norm=True*, *save_freq=1*, *save_initial=True*, *lin_solver='mumps'*)

Solve the mechanics problem defined in the MechanicsProblem object.

Parameters

- **nonlinear_tol** (*float (default 1e-10)*) – Tolerance used to terminate Newton's method.
- **iter_tol** (*float (default 1e-8)*) – Tolerance used to terminate the iterative linear solver.
- **maxNonlinIters** (*int (default 50)*) – Maximum number of iterations for Newton's method.
- **maxLinIters** (*int (default 200)*) – Maximum number of iterations for iterative linear solver.
- **show** (*int (default 0)*) – Amount of information for iterative.LGMRES to show. See documentation of this class for different log levels.
- **print_norm** (*bool (default True)*) – True if user wishes to see the norm at every linear iteration and False otherwise.
- **save_freq** (*int (default 1)*) – The frequency at which the solution is to be saved if the problem is unsteady. E.g., save_freq = 10 if the user wishes to save the solution every 10 time steps.
- **save_initial** (*bool (default True)*) – True if the user wishes to save the initial condition and False otherwise.
- **lin_solver** (*str (default "mumps")*) – Name of the linear solver to be used for steady compressible elastic problems. See the dolfin.solve documentation for a list of available linear solvers.

Returns

Return type None

time_solve(*nonlinear_tol=1e-10*, *iter_tol=1e-08*, *maxNonlinIters=50*, *maxLinIters=200*, *show=0*, *print_norm=True*, *save_freq=1*, *save_initial=True*, *lin_solver='mumps'*)

Loop through the time interval using the time step specified in the MechanicsProblem config dictionary.

Parameters

- **nonlinear_tol** (*float (default 1e-10)*) – Tolerance used to terminate Newton's method.

- **iter_tol** (*float (default 1e-8)*) – Tolerance used to terminate the iterative linear solver.
- **maxNonlinIters** (*int (default 50)*) – Maximum number of iterations for Newton's method.
- **maxLinIters** (*int (default 200)*) – Maximum number of iterations for iterative linear solver.
- **show** (*int (default 0)*) – Amount of information for iterative.LGMRES to show. See documentation of this class for different log levels.
- **print_norm** (*bool (default True)*) – True if user wishes to see the norm at every linear iteration and False otherwise.
- **save_freq** (*int (default int)*) – The frequency at which the solution is to be saved if the problem is unsteady. E.g., save_freq = 10 if the user wishes to save the solution every 10 time steps.
- **save_initial** (*bool (default True)*) – True if the user wishes to save the initial condition and False otherwise.
- **lin_solver** (*str (default "mumps")*) – Name of the linear solver to be used for steady compressible elastic problems. See the dolfin.solve documentation for a list of available linear solvers.

Returns

Return type None

ufl_lhs_rhs()
Return the UFL objects that define the left and right hand sides of the resulting linear system for the variational problem defined by the MechanicsProblem object.

update_soln(*du*)
Update the values of the field variables based on the solution to the resulting linear system.

Parameters **du** (*block.block_vec, dolfin.Vector*) – The solution to the resulting linear system of the variational problem defined by the MechanicsProblem object.

Returns

Return type None

5.2.2 Base Mechanics Solver

The `SolidMechanicsSolver` and `FluidMechanicsSolver` classes inherit from both `dolfin.NonlinearVariationalSolver` and `basemechanics.BaseMechanicsSolver`.

class `fenicsmechanics.basemechanics.`**BaseMechanicsSolver**(*problem*, *fname_pressure=None*, *fname_hdf5=None*, *fname_xdmf=None*)
This is the base class for mechanics solvers making use of the FEniCS mixed function space functionality. Methods common to all mechanics solvers are defined here.

full_solve(*save_freq=1*, *save_initial=True*)
Solve the mechanics problem defined by SolidMechanicsProblem. If the problem is unsteady, this function will loop through the entire time interval using the parameters provided for the Newmark integration scheme.

Parameters

- **save_freq** (*int (default 1)*) – The frequency at which the solution is to be saved if the problem is unsteady. E.g., save_freq = 10 if the user wishes to save the solution every 10 time steps.
- **save_initial** (*bool (default True)*) – True if the user wishes to save the initial condition and False otherwise.

Returns

Return type None

set_parameters (*linear_solver='default'*, *preconditioner='default'*, *newton_abstol=1e-10*, *newton_reltol=1e-09*, *newton_maxIters=50*, *krylov_abstol=1e-08*, *krylov_reltol=1e-07*, *krylov_maxIters=50*)

Set the parameters used by the NonlinearVariationalSolver.

Parameters

- **linear_solver** (*str*) – The name of linear solver to be used.
- **newton_abstol** (*float (default 1e-10)*) – Absolute tolerance used to terminate Newton's method.
- **newton_reltol** (*float (default 1e-9)*) – Relative tolerance used to terminate Newton's method.
- **newton_maxIters** (*int (default 50)*) – Maximum number of iterations for Newton's method.
- **krylov_abstol** (*float (default 1e-8)*) – Absolute tolerance used to terminate Krylov solver methods.
- **krylov_reltol** (*float (default 1e-7)*) – Relative tolerance used to terminate Krylov solver methods.
- **krylov_maxIters** (*int (default 50)*) – Maximum number of iterations for Krylov solver methods.

Returns

Return type None

step()

Compute the solution for the next time step in the simulation. Note that there is only one "step" if the simulation is steady.

update_assign()

This method is meant to update the state of the system and assign to the proper dolfin.Function objects. It is to be overwritten by the specific solver object, and hence will raise an exception at this level.

5.2.3 Solid Mechanics Solver

class fenicsmechanics.solidmechanics.**SolidMechanicsSolver** (*problem*, *fname_disp=None*, *fname_pressure=None*, *fname_hdf5=None*, *fname_xdmf=None*)

This class is derived from the dolfin.NonlinearVariationalSolver to solve problems formulated with SolidMechanicsProblem. It passes the UFL variational forms to the solver, and loops through all time steps if the problem is unsteady. The solvers that are available through this class are the same as those available through dolfin. The

user may use the helper function, 'set_parameters' to set the linear solver used, as well as the tolerances for iterative and nonlinear solves, or do it directly through the 'parameters' member.

static update(*u*, *u0*, *v0*, *a0*, *beta*, *gamma*, *dt*)

Function to update values of field variables at the current and previous time steps based on the Newmark integration scheme:

a = 1.0/(beta*dt^2)*(u - u0 - v0*dt) - (1.0/(2.0*beta) - 1.0)*v0 v = dt*((1.0 - gamma)*a0 + gamma*a) + v0

This particular method is to be used when the function objects do not derive from a mixed function space.

Parameters

- **u** (*dolfin.Function*) – Object storing the displacement at the current time step.
- **u0** (*dolfin.Function*) – Object storing the displacement at the previous time step.
- **v0** (*dolfin.Function*) – Object storing the velocity at the previous time step.
- **a0** (*dolfin.Function*) – Object storing the acceleration at the previous time step.
- **beta** (*float*) – Scalar parameter for the family of Newmark integration schemes. See equation above.
- **gamma** (*float*) – Scalar parameter for the family of Newmark integration schemes. See equation above.
- **dt** (*float*) – Time step used to advance through the entire time interval.

Returns

Return type None

update_assign()

Update the values of the field variables – both the current and previous time step in preparation for the next step of the simulation.

5.2.4 Fluid Mechanics Solver

class fenicsmechanics.fluidmechanics.**FluidMechanicsSolver**(*problem*, *fname_vel=None*, *fname_pressure=None*, *fname_hdf5=None*, *fname_xdmf=None*)

This class is derived from the dolfin.NonlinearVariationalSolver to solve problems formulated with FluidMechanicsProblem. It passes the UFL variational forms to the solver, and loops through all time steps if the problem is unsteady. The solvers that are available through this class are the same as those available through dolfin. The user may use the helper function, 'set_parameters' to set the linear solver used, as well as the tolerances for iterative and nonlinear solves, or do it directly through the 'parameters' member.

update_assign()

Update the values of the field variables – both the current and previous time step in preparation for the next step of the simulation.

5.3 Constitutive Equations

The materials submodule implements some common constitutive equations for users to take advantage of.

5.3.1 Solid Materials

This module contains classes that define the stress tensor for various solid materials. The parent class `ElasticMaterial` is used for all class definitions. Then materials are separated into isotropic and anisotropic using `IsotropicMaterial` and `AnisotropicMaterial`.

Parameters required by each constitutive equation is case-dependent. Thus, the user should check that particular documentation.

class `fenicsmechanics.materials.solid_materials.`**`LinearIsoMaterial`**(*inverse=False*, ***params*)

Return the Cauchy stress tensor based on linear elasticity, i.e. infinitesimal deformations, both compressible and incompressible. The stress tensor is given by

- Compressible: $\mathbf{T} = \lambda \text{tr}(\mathbf{e})\mathbf{I} + 2\mu\mathbf{e}$,
- Incompressible: $\mathbf{T} = -p\mathbf{I} + 2\mu\mathbf{e}$,

where λ and μ are the Lame material parameters, $\mathbf{e} = \text{sym}(\text{grad}(\mathbf{u}))$ where $\mathbf{u}$ is the displacement, and p is the pressure in the case of an incompressible material.

The inverse elastostatics formulation is also supported for this material model. In that case, the only change that must be accounted for is the fact that

$$\mathbf{e} = \text{sym}(\mathbf{f}^{-1}) - \mathbf{I},$$

where $\mathbf{f} = \mathbf{F}^{-1}$ is the deformation gradient from the current to the reference configuration.

At least two of the material constants from the list below must be provided in the `'material'` subdictionary of `config` in addition to the values already listed in the docstring of `fenicsmechanics`. The remaining constants will be computed based on the parameters provided by the user.

Parameters

- **`'la'`** (*`float`*) – The first Lame parameter used as shown in the equations above. Note: providing la and inv_la does not qualify as providing two material constants.
- **`'mu'`** (*`float`*) – The second Lame parameter used as shown in the equations above.
- **`'kappa'`** (*`float`*) – The bulk modulus of the material.
- **`'inv_la'`** (*`float`*) – The reciprocal of the first Lame parameter, la. Note: providing la and inv_la does not qualify as providing two material constants.
- **`'E'`** (*`float`*) – The Young's modulus of the material.
- **`'nu'`** (*`float`*) – The Poisson's ratio of the material.

`incompressibilityCondition`(*u*)

Return the incompressibility condition for a linear material, $p = \kappa \text{div}(\mathbf{u})$.

Parameters **`u`** (*`dolfin.Function, ufl.tensors.ListTensor`*) – The displacement vector.

Returns **Bvol** – UFL object defining the incompressibility condition.

Return type ufl.algebra.Product

`stress_tensor`(*F*, *J*, *p=None*, *formulation=None*)

Return the Cauchy stress tensor for a linear material, namely:

- Compressible: $\mathbf{T} = \lambda \text{tr}(\mathbf{e})\mathbf{I} + 2\mu\mathbf{e}$,
- Incompressible: $\mathbf{T} = -p\mathbf{I} + 2\mu\mathbf{e}$,

Parameters

- **F** (*ufl.algebra.Sum*) – The deformation gradient.
- **J** (*ufl.tensoralgebra.Determinant*) – The Jacobian, i.e. determinant of the deformation gradient. Note that this is not used for this material. It is solely a place holder to conform to the format of other materials.
- **p** (*ufl.Coefficient (default, None)*) – The UFL pressure function for incompressible materials.
- **formulation** (*str (default, None)*) – This input is not used for this material. It is solely a place holder to conform to the format of other materials.

Returns **T** – The Cauchy stress tensor given by the equation above.

Return type ufl.algebra.Sum

class fenicsmechanics.materials.solid_materials.**NeoHookeMaterial**(*inverse=False*, ***params*)

Return the first Piola-Kirchhoff stress tensor based on variations of the strain energy function

$$\psi(\mathbf{C}) = \frac{1}{2}\mu(\text{tr}(\mathbf{C}) - n),$$

where $\mathbf{C} = \mathbf{F}^T\mathbf{F}$, and n is the geometric dimension. For nearly incompressible materials, the total strain energy function is given by

$$W = U(J) + \psi(\mathbf{C}),$$

where $U(J)$ corresponds to the strain energy in response to dilatation of the material with $J = \det(\mathbf{F})$. The two forms of $U(J)$ supported here are

- Square: $U(J) = \frac{1}{2}\kappa(J-1)^2$,
- Log: $U(J) = \frac{1}{2}\kappa(\ln(J))^2$,

where κ is the bulk modulus of the material. This results in the first Piola-Kirchhoff stress tensor given by

$$\mathbf{P} = J\frac{dU}{dJ}\mathbf{F}^{-T} + \mu\mathbf{F}.$$

In the case of an incompressible material, the total strain energy is assumed to be of the form

$$W = U(J) + \psi(\bar{\mathbf{C}}),$$

where $\bar{\mathbf{C}} = J^{-2/n}\mathbf{C}$. Furthermore, the pressure scalar field is defined such that $p = -\frac{dU}{dJ}$. The resulting first Piola-Kirchhoff stress tensor is then

$$\mathbf{P} = -\left[Jp + \frac{1}{n}\mu J^{-2/n}\text{tr}(\mathbf{C})\right]\mathbf{F}^{-T} + \mu J^{-2/n}\mathbf{F}.$$

The inverse elastostatics formulation is also supported for this material model. In that case, the Cauchy stress tensor for compressible material is

$$\mathbf{T} = -j^2\frac{dU}{dj}\mathbf{I} + \mu\mathbf{c}^{-1},$$

and

$$\mathbf{T} = -\left[p + \frac{1}{n}\mu j^{-1/n}i_2\right]\mathbf{I} + \mu j^{5/n}\mathbf{c}^{-1},$$

for incompressible material where $j = \det(\mathbf{f}) = \det(\mathbf{F}^{-1}) = \frac{1}{J}$, $\mathbf{c} = \mathbf{f}^T\mathbf{f}$, i_2 is the second invariant of c, and p is the pressure in the latter case. Note that $\mathbf{f}$ is the deformation gradient from the current configuration to the reference configuration.

At least two of the material constants from the list below must be provided in the `'material'` subdictionary of `config` in addition to the values already listed in the docstring of `fenicsmechanics`.

Parameters

- **`'la'`** (*float*) – The first parameter used as shown in the equations above. Note: providing `la` and `inv_la` does not qualify as providing two material parameters.
- **`'mu'`** (*float*) – The second material parameter used as shown in the equations above.
- **`'kappa'`** (*float*) – The bulk modulus of the material.
- **`'inv_la'`** (*float*) – The reciprocal of the first parameter, `la`. Note: providing `la` and `inv_la` does not qualify as providing two material parameters.
- **`'E'`** (*float*) – The Young's modulus of the material. Note: this is not entirely consistent with the neo-Hookean formulation, but `la` and `mu` will be computed based on the relation between the Young's modulus and the Lame parameters if `E` is given.
- **`'nu'`** (*float*) – The Poisson's ratio of the material. Note: this is not entirely consistent with the neo-Hookean formulation, but `la` and `mu` will be computed based on the relation between the Poisson's ratio and the Lame parameters if `nu` is given.

`strain_energy`(*F*, *J*, *formulation=None*)

Define the total strain energy based on the incompressibility of the material (fully or nearly incompressible, or compressible), defined with respect to the deformation gradient, or by its inverse if the objective is to find the inverse displacement. The strain energy for a compressible material is defined by

$$\begin{aligned} W &= \frac{1}{2}\mu(I_1 - n) + \frac{1}{2}\lambda(\ln(J))^2 - \mu\ln(J) \\ &= \frac{1}{2}\mu\left(\frac{i_2}{i_3} - n\right) + \frac{1}{2}\lambda(\ln(j))^2 - \mu\ln(j), \end{aligned}$$

where I_1 is the first invariant of $\mathbf{C} = \mathbf{F}^T\mathbf{F}$, while i_2 and i_3 are the second and third invariants of $\mathbf{c} = \mathbf{f}^T\,\mathbf{f}$, with $\mathbf{f} = \mathbf{F}^{-1}$. For a (nearly-)incompressible material, the strain energy is defined by

$$\begin{aligned} W &= U(J) + \frac{1}{2}\mu(I_1 - n) \\ &= U(j) + \frac{1}{2}\mu(\frac{i_2}{i_3} - n), \end{aligned}$$

where the invariants are now those of $\bar{\mathbf{C}} = J^{-2/n}\mathbf{C}$ or $\bar{\mathbf{c}} = j^{-2/n}\mathbf{c}$, and $\frac{dU}{dJ} = p$ for fully incompressible material, while $U(J) = \kappa\phi(J)$. The two forms of $\phi(J)$ supported here are

- Square: $\phi(J) = \frac{1}{2}(J-1)^2$,
- Log: $\phi(J) = \frac{1}{2}(\ln(J))^2$.

Parameters

- **`F`** (*ufl.algebra.Sum*) – The (forward or inverse) deformation gradient.
- **`J`** (*ufl.tensoralgebra.Determinant*) – The jacobian, i.e. determinant of the deformation gradient given above.
- **`formulation`** (*str (default, None)*) – The formulation used for the nearly-incompressible materials. Value must either be `'square'` or `'log'`.

Returns **W** – The strain energy defined above.

Return type ufl.algebra.Sum

stress_tensor(*F*, *J*, *p=None*, *formulation=None*)

Return the first Piola-Kirchhoff stress tensor for a neo-Hookean material. The stress tensor is given by

- Compressible:

$$\mathbf{P} = J\frac{dU}{dJ}\mathbf{F}^{-T} + \mu\mathbf{F}$$

- Incompressible:

$$\mathbf{P} = \left[-Jp - \frac{\mu J^{-2/n}}{n}\text{tr}(\mathbf{C})\right]\mathbf{F}^{-T} + \mu J^{-2/n}\mathbf{F},$$

where $\mathbf{F}$ is the deformation gradient, $J = \det(\mathbf{F})$, $\mathbf{C} = \mathbf{F}^T\mathbf{F}$, and μ is a material constant. If the problem is an inverse elastostatics problem, the Cauchy stress tensor is given by

- Compressible:

$$\mathbf{T} = \left[-j^2\frac{dU}{dj} - \frac{\mu j^{-1/n}}{n}i_2\right]\mathbf{I} + \mu j^{-5/n}\mathbf{c}^{-1}$$

- Incompressible:

$$\mathbf{T} = -\left[p + \frac{\mu j^{-1/n}}{n}i_2\right]\mathbf{I} + \mu j^{-5/n}\mathbf{c}^{-1}$$

where $\mathbf{f} = \mathbf{F}^{-1}$, $j = \det(\mathbf{f})$, $\mathbf{c} = \mathbf{f}^T\mathbf{f}$, and n is the geometric dimension.

Parameters

- **F** (*ufl.algebra.Sum*) – The deformation gradient.
- **J** (*ufl.tensoralgebra.Determinant*) – The Jacobian, i.e. determinant of the deformation gradient.
- **p** (*dolfin.Function, ufl.indexed.Indexed (default, None)*) – The pressure for incompressible materials.
- **formulation** (*str (default, None)*) – The formulation used for the strain energy due to dilatation of the material.

Returns **P** – The first Piola-Kirchhoff stress tensor for forward problems and the Cauchy stress tensor for inverse elastostatics problems.

Return type ufl.algebra.Sum

class fenicsmechanics.materials.solid_materials.**FungMaterial**(*mesh*, *inverse=False*, ***params*)

This class defines the stress tensor for Fung type materials which are based on the strain energy function given by

$$W = C\exp(Q),$$

where

$$Q = d_1E_{11}^2 + d_2E_{22}^2 + d_3E_{33}^2 + 2(d_4E_{11}E_{22} + d_5E_{22}E_{33} + d_6E_{11}E_{33}) + d_7E_{12}^2 + d_8E_{23}^2 + d_9E_{13}^2,$$

and C and $d_i, i = 1, ..., 9$ are material constants. The E_{ij} components here are the components of the Lagrangian strain tensor,

$$\mathbf{E} = \frac{1}{2}(\mathbf{F}^T\mathbf{F} - \mathbf{I}),$$

with respect to the orthonormal set $\{\mathbf{e}_1, \mathbf{e}_2,\ \mathbf{e}_3\}$, where $\mathbf{e}_1$ is a fiber direction, $\mathbf{e}_2$ the direction normal to the fiber, and $\mathbf{e}_3 = \mathbf{e}_1 \times \mathbf{e}_2$.

The resulting first Piola-Kirchhoff stress tensor is then

$$\mathbf{P} = \mathbf{P}_{vol} + \mathbf{P}_{iso}$$

with

$$\mathbf{P}_{iso} = J^{-2/n}\mathbf{F}\hat{\mathbf{S}} - \frac{1}{n}J^{-2/n}\text{tr}(\mathbf{C}\hat{\mathbf{S}})\mathbf{F}^{-T},$$

and

$$\hat{\mathbf{S}} = C\exp(Q)\left((d_1E_{11} + d_4E_{22} + d_6E_{33})\text{outer}(\mathbf{e}_1, \mathbf{e}_1) + (d_4E_{11} + d_2E_{22} + d_5E_{33})\text{outer}(\mathbf{e}_2, \mathbf{e}_2) + (d_6E_{11} + d_5E_{22} + d_3E_{33}\right.$$

For compressible materials,

$$\mathbf{P}_{vol} = 2J\kappa(J - \frac{1}{J})\mathbf{F}^{-T},$$

and

$$\mathbf{P}_{vol} = -Jp\mathbf{F}^{-T}$$

for incompressible, where κ is the bulk modulus, and p is the pressure.

The inverse elastostatics formulation for this material is supported. The constitutive equation remains the same, but the Lagrangian strain is defined in terms of the $\mathbf{f} = \mathbf{F}^{-1}$, i.e.

$$\mathbf{E} = \frac{1}{2}((\mathbf{f}\mathbf{f}^T)^{-1} - \mathbf{I}).$$

Furthermore, $\mathbf{F} = \mathbf{f}^{-1}$ is substituted, and the stress tensor returned is the Cauchy stress tensor given by $\mathbf{T} = j\mathbf{P}\mathbf{f}^{-T}$, where $j = \det(\mathbf{f})$.

In addition to the values listed in the docstring of `fenicsmechanics` for the 'material' subdictionary of 'config', the user must provide the values listed below.

Parameters

- **`'fibers'`** (*`dict`*) – See AnisotropicMaterial for details.
- **`'C'`** (*`float`*) – Material constant that can be thought of as "stiffness" in some limiting cases.
- **`'d'`** (*`list, tuple`*) – A list containing the coefficients in the exponent, Q, defined above.
- **`'kappa'`** (*`float`*) – The bulk modulus of the material.

`stress_tensor`(*F*, *J*, *p=None*, *formulation=None*)

Return the first Piola-Kirchhoff stress tensor for forward problems and the Cauchy stress tensor for inverse elastostatics problems. The constitutive equation is shown in the documentation of FungMaterial.

Parameters

- **`F`** (*`ufl.algebra.Sum`*) – The deformation gradient.

- **J** (*ufl.tensoralgebra.Determinant*) – The Jacobian, i.e. the determinant of the deformation gradient.
- **p** (*dolfin.Function, ufl.indexed.Indexed (default, None)*) – The pressure function for incompressible materials.
- **formulation** (*str (default, None)*) – This input is not used for this material. It is solely a place holder to conform to the format of other materials.

Returns **P** – The stress tensor defined in the FungMaterial documentation.

Return type ufl.algebra.Sum

class fenicsmechanics.materials.solid_materials.**GuccioneMaterial**(*mesh*, *inverse=False*, ***params*)

This class defines the stress tensor for Guccione type materials, which are a subclass of Fung-type materials. The constitutive equation is the same as that found in the FungMaterial documentation, but with

$$Q = b_f E_{11}^2 + b_t(E_{22}^2 + E_{33}^2 + 2E_{23}^2) + 2b_{fs}(E_{12}^2 + E_{13}^2),$$

where b_f, b_t, and b_{fs} are the material parameters. The relation between these material constants and the $d_i, i = 1, ..., 9$ can be obtained in a straight-forward manner.

Note that the inverse elastostatics formulation is also supported since this class is derived from FungMaterial.

In addition to the values listed in the documentation for `fenicsmechanics` for the `'material'` subdictionary of 'config', the user must provide the following values:

Parameters

- **'fibers'** (*dict*) – See AnisotropicMaterial for details.
- **'C'** (*float*) – Material constant that can be thought of as "stiffness" in some limiting cases.
- **'bf'** (*float*) – Material "stiffness" in the fiber direction.
- **'bt'** (*float*) – Material "stiffness" in transverse directions.
- **'bfs'** (*float*) – Material rigidity under shear.

strain_energy(*u*, *p=None*)

Return the strain energy of Guccione material defined in the documentation of GuccioneMaterial.

Parameters

- **u** (*dolfin.Funcion, ufl.tensors.ListTensor*) – The displacement of the material.
- **p** (*dolfin.Function, ufl.indexed.Indexed (default, None)*) – The pressure for incompressible materials.

Returns **W** – The strain energy defined in the documentation of GuccioneMaterial.

Return type ufl.algebra.Sum

5.3.2 Fluids

class fenicsmechanics.materials.fluids.**NewtonianFluid**(***params*)

Class defining the stress tensor for an incompressible Newtonian fluid. Currently, only incompressible fluids are supported. The Cauchy stress tensor is given by

$$\mathbf{T} = -p\mathbf{I} + 2\mu\text{Sym}(\mathbf{L}),$$

where p is the pressure, $\mathbf{I}$ is the identity tensor, μ is the dynamic viscosity of the fluid, and $\mathbf{L} = \text{grad}(\mathbf{v})$, where $\mathbf{v}$ is the velocity vector field.

In addition to the values listed in the documentation of `fenicsmechanics` for the `'material'` subdictionary of 'config', the user must provide at least one of the values listed below:

Parameters

- **`'mu'`** (*float*) – Dynamic viscosity of the fluid.
- **`'nu'`** (*float*) – Kinematic viscosity of the fluid.

`stress_tensor`(L, p)

Return the Cauchy stress tensor for an incompressible Newtonian fluid, namely

$$\mathbf{T} = -p\mathbf{I} + 2\mu\text{Sym}(\mathbf{L}),$$

where $\mathbf{L} = \text{grad}(\mathbf{v})$, $\mathbf{I}$ is the identity tensor, p is the hydrostatic pressure, and μ is the dynamic viscosity.

Parameters

- **`L`** (*ufl.differentiation.Grad*) – The velocity gradient.
- **`p`** (*dolfin.Function, ufl.indexed.Indexed*) – The hydrostatic pressure.

Returns **T** – The Cauchy stress tensor defined above.

Return type ufl.algebra.Sum

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[Cha99] Peter Chadwick. *Continuum mechanics: concise theory and problems*. Dover Publications, 1999.

[GM96] Sanjay Govindjee and Paul A. Mihalic. Computational methods for inverse finite elastostatics. *Computer Methods in Applied Mechanics and Engineering*, 136(1-2):47–57, 1996. doi:10.1016/0045-7825(96)01045-6.

[GCM95] Julius M. Guccione, Kevin D. Costa, and Andrew D. Mcculloch. Finite element stress analysis of left ventricular mechanics in the beating dog heart. *Journal of Biomechanics*, 28(10):1167–1177, 1995. doi:10.1016/0021-9290(94)00174-3.

[Hug07] Thomas J. R. Hughes. *The finite element method: linear static and dynamic finite element analysis*. Dover publication, Inc, 2007.

[LGA+15] Sander Land, Viatcheslav Gurev, Sander Arens, Christoph M. Augustin, Lukas Baron, Robert Blake, Chris Bradley, Sebastian Castro, Andrew Crozier, Marco Favino, and et al. Verification of cardiac mechanics software: benchmark problems and solutions for testing active and passive material behaviour. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Science*, 471(2184):20150641, Aug 2015. doi:10.1098/rspa.2015.0641.

PYTHON MODULE INDEX

f

INDEX

F

G

I

L

M

N

S

T

U