Numerical tours of continuum mechanics using FEniCS

Release master

Jeremy Bleyer

Mar 12, 2021

Contents

Contents:

CHAPTER 1

Introduction

Welcome to these **Numerical Tours of Computational Mechanics with FEniCS**.

1.1 What is it about ?

These numerical tours will introduce you to a wide variety of topics in computational continuum and structural mechanics using the finite element software FEniCS. Many covered topics can be considered as standard and will help the reader in getting started with FEniCS using solid mechanics examples.

Other topics will also be more exploratory and will reflect currently investigated research topics, illustrating the versatility of FEniCS.

The full set of demos can be obtained from the *COmputational MEchanics Toolbox* (COMET) available at https://gitlab.enpc.fr/jeremy.bleyer/comet-fenics.

A new set of demos illustrating how to couple FEniCS with the *MFront* code generator have been added. They are based on the `mgis.fenics` module of the MFrontGenericInterfaceSupport (MGIS) project. A general introduction of the package is available here and the demos source files can be found here. This project has been realized in collaboration with Thomas Helfer (CEA, thomas.helfer@cea.fr).

1.2 Citing and license

If you find these demos useful for your research work, please consider citing them using the following Zenodo DOI :

```
@manual{bleyer2018numericaltours,
title={Numerical Tours of Computational Mechanics with {FE}ni{CS}},
DOI={10.5281/zenodo.1287832},
howpublished = {https://comet-fenics.readthedocs.io},
publisher={Zenodo},
```

(continues on next page)

(continued from previous page)

```
author={Jeremy Bleyer},
year={2018}}
```


The MGIS project can be cited through the following paper

```
@article{Helfer2020,
 doi = {10.21105/joss.02003},
 url = {https://doi.org/10.21105/joss.02003},
 year = {2020},
 publisher = {The Open Journal},
 volume = {5},
 number = {48},
 pages = {2003},
 author = {Thomas Helfer and Jeremy Bleyer and Tero Frondelius and
           Ivan Yashchuk and Thomas Nagel and Dmitri Naumov},
 title = {The `MFrontGenericInterfaceSupport` project},
 journal = {Journal of Open Source Software}
}
```

1.3 How do I get started ?

You can find instructions on how to install FEniCS on the FEniCS project website http://fenicsproject.org. In the following numerical tours, we will use the Python interface for the different FEniCS scripts. These demos are compatible with FEniCS 2018.1.0 but many should work with older versions after minor changes.

FEniCS is also distributed along with an important number of documented or undocumented examples, some of them will be revisited in these tours but do not hesitate over looking at other interesting examples.

In the following, we will assume that readers possess basic knowledge of FEniCS commands. In particular, we advise you to go first through the documentation and tutorials https://fenicsproject.org/tutorial/ if this is not the case.

1.4 About the author

Jeremy Bleyer is a researcher in Solid and Structural Mechanics at Laboratoire Navier, a joint research unit of Ecole Nationale des Ponts et Chaussées, IFSTTAR and CNRS (UMR 8205).

email: jeremy.bleyer@enpc.fr


École des Ponts
ParisTech
IFSTTAR
cnrs

CHAPTER 2

Linear problems in solid mechanics

Contents:

2.1 2D linear elasticity

2.1.1 Introduction

In this first numerical tour, we will show how to compute a small strain solution for a 2D isotropic linear elastic medium, either in plane stress or in plane strain, in a tradtional displacement-based finite element formulation. The corresponding file can be obtained from `2D_elasticity.py`.

See also:

Extension to 3D is straightforward and an example can be found in the *Modal analysis of an elastic structure* example.

We consider here the case of a cantilever beam modeled as a 2D medium of dimensions $L \times H$. Geometrical parameters and mesh density are first defined and the rectangular domain is generated using the `RectangleMesh` function. We also choose a criss-crossed structured mesh:

```
from dolfin import *

L = 25.
H = 1.
Nx = 250
Ny = 10
mesh = RectangleMesh(Point(0., 0.), Point(L, H), Nx, Ny, "crossed")
```

2.1.2 Constitutive relation

We now define the material parameters which are here given in terms of a Young's modulus E and a Poisson coefficient ν. In the following, we will need to define the constitutive relation between the stress tensor $\boldsymbol{\sigma}$ and the strain tensor $\boldsymbol{\varepsilon}$.

Let us recall that the general expression of the linear elastic isotropic constitutive relation for a 3D medium is given by:

$$\boldsymbol{\sigma} = \lambda \text{tr}(\boldsymbol{\varepsilon})\mathbf{1} + 2\mu\boldsymbol{\varepsilon} \tag{2.1}$$

for a natural (no prestress) initial state where the Lamé coefficients are given by:

$$\lambda = \frac{E\nu}{(1+\nu)(1-2\nu)}, \quad \mu = \frac{E}{2(1+\nu)} \tag{2.2}$$

In this demo, we consider a 2D model either in plane strain or in plane stress conditions. Irrespective of this choice, we will work only with a 2D displacement vector $\boldsymbol{u} = (u_x, u_y)$ and will subsequently define the strain operator `eps` as follows:

```
def eps(v):
    return sym(grad(v))
```

which computes the 2x2 plane components of the symmetrized gradient tensor of any 2D vectorial field. In the plane strain case, the full 3D strain tensor is defined as follows:

$$\boldsymbol{\varepsilon} = \begin{bmatrix} \varepsilon_{xx} & \varepsilon_{xy} & 0 \\ \varepsilon_{xy} & \varepsilon_{yy} & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

so that the 2x2 plane part of the stress tensor is defined in the same way as for the 3D case (the out-of-plane stress component being given by $\sigma_{zz} = \lambda(\varepsilon_{xx} + \varepsilon_{yy})$.

In the plane stress case, an out-of-plane strain component ε_{zz} must be considered so that $\sigma_{zz} = 0$. Using this condition in the 3D constitutive relation, one has $\varepsilon_{zz} = -\dfrac{\lambda}{\lambda+2\mu}(\varepsilon_{xx} + \varepsilon_{yy})$. Injecting into (2.1), we have for the 2D plane stress relation:

$$\boldsymbol{\sigma} = \lambda^* \text{tr}(\boldsymbol{\varepsilon})\mathbf{1} + 2\mu\boldsymbol{\varepsilon}$$

where $\boldsymbol{\sigma}, \boldsymbol{\varepsilon}, \mathbf{1}$ are 2D tensors and with $\lambda^* = \dfrac{2\lambda\mu}{\lambda+2\mu}$. Hence, the 2D constitutive relation is identical to the plane strain case by changing only the value of the Lamé coefficient λ. We can then have:

```
E = Constant(1e5)
nu = Constant(0.3)
model = "plane_stress"

mu = E/2/(1+nu)
lmbda = E*nu/(1+nu)/(1-2*nu)
if model == "plane_stress":
    lmbda = 2*mu*lmbda/(lmbda+2*mu)

def sigma(v):
    return lmbda*tr(eps(v))*Identity(2) + 2.0*mu*eps(v)
```

Note: Note that we used the variable name `lmbda` to avoid any confusion with the lambda functions of Python

We also used an intrinsic formulation of the constitutive relation. Example of constitutive relation implemented with a matrix/vector engineering notation will be provided in the *Orthotropic linear elasticity* example.

2.1.3 Variational formulation

For this example, we consider a continuous polynomial interpolation of degree 2 and a uniformly distributed loading $\boldsymbol{f} = (0, -f)$ corresponding to the beam self-weight. The continuum mechanics variational formulation (obtained from the virtual work principle) is given by:

$$\text{Find } \boldsymbol{u} \in V \text{ s.t. } \int_\Omega \boldsymbol{\sigma}(\boldsymbol{u}) : \boldsymbol{\varepsilon}(\boldsymbol{v}) d\Omega = \int_\Omega \boldsymbol{f} \cdot \boldsymbol{v} d\Omega \quad \forall \boldsymbol{v} \in V$$

which translates into the following FEniCS code:

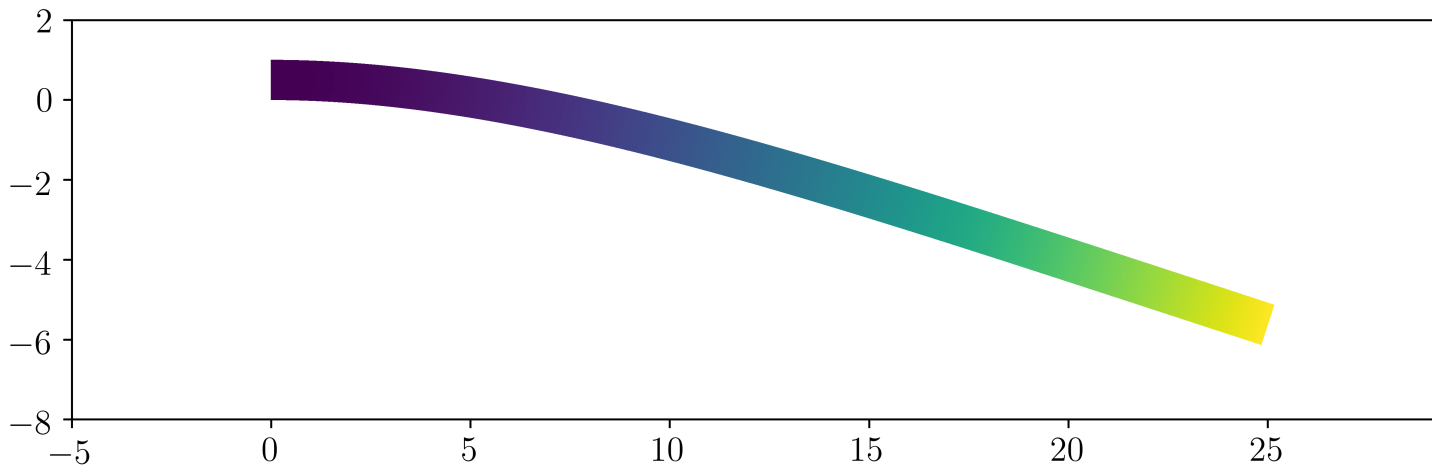
```
rho_g = 1e-3
f = Constant((0, -rho_g))

V = VectorFunctionSpace(mesh, 'Lagrange', degree=2)
du = TrialFunction(V)
u_ = TestFunction(V)
a = inner(sigma(du), eps(u_))*dx
l = inner(f, u_)*dx
```

2.1.4 Resolution

Fixed displacements are imposed on the left part of the beam, the `solve` function is then called and solution is plotted by deforming the mesh:

```
def left(x, on_boundary):
    return near(x[0], 0.)

bc = DirichletBC(V, Constant((0.,0.)), left)

u = Function(V, name="Displacement")
solve(a == l, u, bc)

plot(1e3*u, mode="displacement")
```

The (amplified) solution should look like this:



2.1.5 Validation and post-processing

The maximal deflection is compared against the analytical solution from Euler-Bernoulli beam theory which is here $w_{beam} = \frac{qL^4}{8EI}$:

```
print("Maximal deflection:", -u(L,H/2.)[1])
print("Beam theory deflection:", float(3*rho_g*L**4/2/E/H**3))
```

One finds $w_{FE} = 5.8638\text{e-}3$ against $w_{beam} = 5.8594\text{e-}3$ that is a 0.07% difference.

The stress tensor must be projected on an appropriate function space in order to evaluate pointwise values or export it for Paraview vizualisation. Here we choose to describe it as a (2D) tensor and project it onto a piecewise constant function space:

```
Vsig = TensorFunctionSpace(mesh, "DG", degree=0)
sig = Function(Vsig, name="Stress")
sig.assign(project(sigma(u), Vsig))
print("Stress at (0,H):", sig(0, H))
```

Fields can be exported in a suitable format for vizualisation using Paraview. VTK-based extensions (.pvd,.vtu) are not suited for multiple fields and parallel writing/reading. Prefered output format is now .xdmf:

```
file_results = XDMFFile("elasticity_results.xdmf")
file_results.parameters["flush_output"] = True
file_results.parameters["functions_share_mesh"] = True
file_results.write(u, 0.)
file_results.write(sig, 0.)
```

2.2 Orthotropic linear elasticity

2.2.1 Introduction

In this numerical tour, we will show how to tackle the case of orthotropic elasticity (in a 2D setting). The corresponding file can be obtained from `orthotropic_elasticity.py`.

We consider here the case of a square plate perforated by a circular hole of radius R, the plate dimension is $2L \times 2L$ with $L \gg R$ Only the top-right quarter of the plate will be considered. Loading will consist of a uniform traction on the top/bottom boundaries, symmetry conditions will also be applied on the correponding symmetry planes. To generate the perforated domain we use here the `mshr` module and define the boolean "*minus*" operation between a rectangle and a circle:

```
from dolfin import *
from mshr import *

L, R = 1., 0.1
N = 50 # mesh density

domain = Rectangle(Point(0.,0.), Point(L, L)) - Circle(Point(0., 0.), R)
mesh = generate_mesh(domain, N)
```

2.2.2 Constitutive relation

Constitutive relations will be defined using an engineering (or Voigt) notation (i.e. second order tensors will be written as a vector of their components) contrary to the *2D linear elasticity* example which used an intrinsic notation. In the material frame, which is assumed to coincide here with the global (Oxy) frame, the orthotropic constitutive law writes $\boldsymbol{\varepsilon} = \mathbf{S}\boldsymbol{\sigma}$ using the compliance matrix $\mathbf{S}$ with:

$$\begin{Bmatrix} \varepsilon_{xx} \\ \varepsilon_{yy} \\ 2\varepsilon_{xy} \end{Bmatrix} = \begin{bmatrix} 1/E_x & -\nu_{xy}/E_x & 0 \\ -\nu_{yx}/E_y & 1/E_y & 0 \\ 0 & 0 & 1/G_{xy} \end{bmatrix} \begin{Bmatrix} \sigma_{xx} \\ \sigma_{yy} \\ \sigma_{xy} \end{Bmatrix}$$

with E_x, E_y the two Young's moduli in the orthotropy directions, ν_{xy} the in-plane Poisson ration (with the following relation ensuring the constitutive relation symmetry $\nu_{yx} = \nu_{xy}E_y/E_x$) and G_{xy} being the shear modulus. This relation needs to be inverted to obtain the stress components as a function of the strain components $\boldsymbol{\sigma} = \mathbf{C}\boldsymbol{\varepsilon}$ with $\mathbf{C} = \mathbf{S}^{-1}$:

```
Ex, Ey, nuxy, Gxy = 100., 10., 0.3, 5.
S = as_matrix([[1./Ex,-nuxy/Ex,0.],[-nuxy/Ex,1./Ey,0.],[0.,0.,1./Gxy]])
C = inv(S)
```

Note: Here we used the `inv` opertor to compute the elasticity matrix $\mathbf{C}$. We could also have computed analytically the inverse relation. Note that the `inv` operator is implemented only up to 3x3 matrices. Extension to the 3D case yields 6x6 matrices and therefore requires either analytical inversion or numerical inversion using Numpy for instance (assuming that the material parameters are constants).

We define different functions for representing the stress and strain either as second-order tensor or using the Voigt engineering notation:

```
def eps(v):
    return sym(grad(v))
def strain2voigt(e):
    """e is a 2nd-order tensor, returns its Voigt vectorial representation"""
    return as_vector([e[0,0],e[1,1],2*e[0,1]])
def voigt2stress(s):
    """
    s is a stress-like vector (no 2 factor on last component)
    returns its tensorial representation
    """
    return as_tensor([[s[0], s[2]],
                      [s[2], s[1]]])
def sigma(v):
    return voigt2stress(dot(C, strain2voigt(eps(v))))
```

2.2.3 Problem position and resolution

Different parts of the quarter plate boundaries are now defined as well as the exterior integration measure `ds`:

```
class Top(SubDomain):
    def inside(self, x, on_boundary):
        return near(x[1], L) and on_boundary
class Left(SubDomain):
    def inside(self, x, on_boundary):
        return near(x[0], 0) and on_boundary
class Bottom(SubDomain):
```

(continues on next page)

(continued from previous page)

```
    def inside(self, x, on_boundary):
        return near(x[1], 0) and on_boundary

# exterior facets MeshFunction
facets = MeshFunction("size_t", mesh, 1)
facets.set_all(0)
Top().mark(facets, 1)
Left().mark(facets, 2)
Bottom().mark(facets, 3)
ds = Measure('ds', subdomain_data=facets)
```

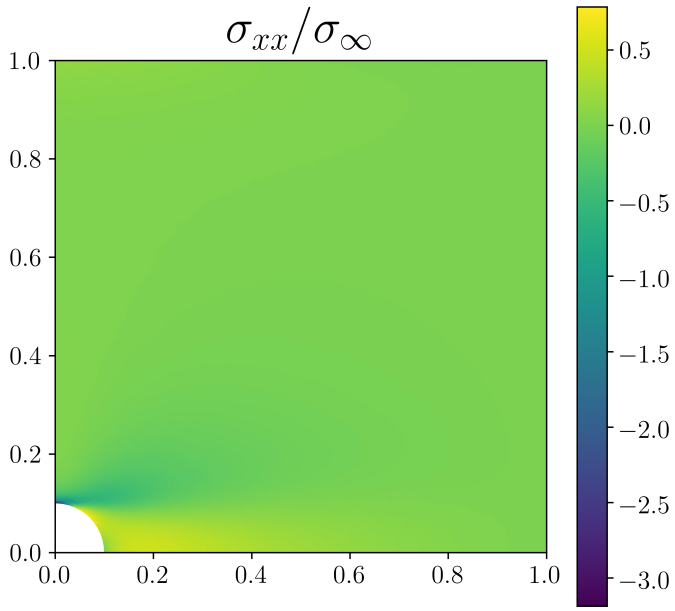
We are now in position to define the variational form which is given as in *2D linear elasticity*, the linear form now contains a Neumann term corresponding to a uniform vertical traction σ_∞ on the top boundary:

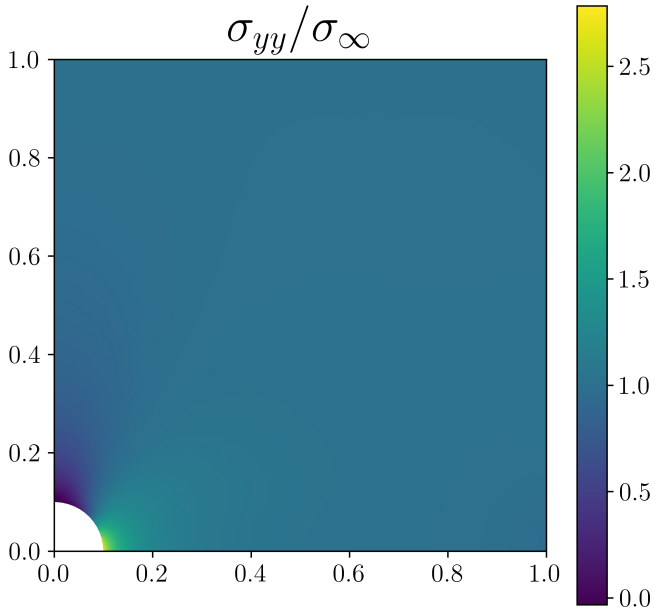
```
# Define function space
V = VectorFunctionSpace(mesh, 'Lagrange', 2)

# Define variational problem
du = TrialFunction(V)
u_ = TestFunction(V)
u = Function(V, name='Displacement')
a = inner(sigma(du), eps(u_))*dx

# uniform traction on top boundary
T = Constant((0, 1e-3))
l = dot(T, u_)*ds(1)
```

Symmetric boundary conditions are applied on the `Bottom` and `Left` boundaries and the problem is solved:

```
# symmetry boundary conditions
bc = [DirichletBC(V.sub(0), Constant(0.), facets, 2),
      DirichletBC(V.sub(1), Constant(0.), facets, 3)]

solve(a == l, u, bc)

import matplotlib.pyplot as plt
p = plot(sigma(u)[1, 1]/T[1], mode='color')
plt.colorbar(p)
plt.title(r"$\sigma_{yy}$",fontsize=26)
plt.show()
```

The σ_{xx} and σ_{yy} components should look like that:





The corresponding files can be obtained from:

- Jupyter Notebook: `axisymmetric_elasticity.ipynb`
- Python script: `axisymmetric_elasticity.py`

2.3 Axisymmetric formulation for elastic structures of revolution

In this numerical tour, we will deal with axisymmetric problems of elastic solids. We will consider a solid of revolution around a fixed axis (Oz), the loading, boundary conditions and material properties being also invariant with respect

to a rotation along the symmetry axis. The solid cross-section in a plane θ = cst will be represented by a two-dimensional domain ω for which the first spatial variable (`x[0]` in FEniCS) will represent the radial coordinate r whereas the second spatial variable will denote the axial variable z.

2.3.1 Problem position

We will investigate here the case of a hollow hemisphere of inner (resp. outer) radius R_i (resp. R_e). Due to the revolution symmetry, the 2D cross-section corresponds to a quarter of a hollow cylinder.

[9]:
```
from __future__ import print_function
from dolfin import *
from mshr import *
import matplotlib.pyplot as plt
%matplotlib notebook

Re = 11.
Ri = 9.
rect = Rectangle(Point(0., 0.), Point(Re, Re))
domain = Circle(Point(0., 0.), Re, 100) - Circle(Point(0., 0.), Ri, 100)
domain = domain - Rectangle(Point(0., -Re), Point(-Re, Re)) \
                - Rectangle(Point(0., 0.), Point(Re, -Re))

mesh = generate_mesh(domain, 40)
plot(mesh)

class Bottom(SubDomain):
    def inside(self, x, on_boundary):
        return near(x[1], 0) and on_boundary
class Left(SubDomain):
    def inside(self, x, on_boundary):
        return near(x[0], 0) and on_boundary
class Outer(SubDomain):
    def inside(self, x, on_boundary):
        return near(sqrt(x[0]**2+x[1]**2), Re, 1e-1) and on_boundary

facets = MeshFunction("size_t", mesh, 1)
facets.set_all(0)
Bottom().mark(facets, 1)
Left().mark(facets, 2)
Outer().mark(facets, 3)
ds = Measure("ds", subdomain_data=facets)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

2.3.2 Definition of axisymmetric strains

For axisymmetric conditions, the unkown displacement field is of the form:

$$\boldsymbol{u} = u_r(r,z)\boldsymbol{e}_r + u_z(r,z)\boldsymbol{e}_z \quad (2.3)$$

As a result, we will work with a standard VectorFunctionSpace of dimension 2. The associated strain components are however given by:

$$\boldsymbol{\varepsilon} = \begin{bmatrix} \partial_r u_r & 0 & (\partial_z u_r + \partial_r u_z)/2 \\ 0 & u_r/r & 0 \\ (\partial_z u_r + \partial_r u_z)/2 & 0 & \partial_z u_z \end{bmatrix}_{(\boldsymbol{e}_r, \boldsymbol{e}_\theta, \boldsymbol{e}_z)} \tag{2.4}$$

$(\partial_z u_r + \partial_r u_z)/20$
$0(\partial_z u_r + \partial_r u_z)/2$
$\partial_z u_z$

$(\boldsymbol{e}_r, \boldsymbol{e}_\theta, \boldsymbol{e}_z)$

The previous relation involves explicitly the radial variable r, which can be obtained from the SpatialCoordinate `x[0]`, the strain-displacement relation is then defined explicitly in the `eps` function.

> **Note**: we could also express the strain components in the form of a vector of size 4 in alternative of the 3D tensor representation implemented below.

[10]:
```
x = SpatialCoordinate(mesh)

def eps(v):
    return sym(as_tensor([[v[0].dx(0), 0, v[0].dx(1)],
                          [0, v[0]/x[0], 0],
                          [v[1].dx(0), 0, v[1].dx(1)]]))

E = Constant(1e5)
nu = Constant(0.3)
mu = E/2/(1+nu)
lmbda = E*nu/(1+nu)/(1-2*nu)
def sigma(v):
    return lmbda*tr(eps(v))*Identity(3) + 2.0*mu*eps(v)
```

2.3.3 Resolution

The rest of the formulation is similar to the 2D elastic case with a small difference in the integration measure. Indeed, the virtual work principle reads as:

$$\text{Find } \boldsymbol{u} \in V \text{ s.t. } \int_\Omega \boldsymbol{\sigma}(\boldsymbol{u}) : \boldsymbol{\varepsilon}(\boldsymbol{v}) d\Omega = \int_{\partial\Omega_T} \boldsymbol{T} \cdot \boldsymbol{v} dS \quad \forall \boldsymbol{v} \in V$$

where $\boldsymbol{T}$ is the imposed traction on some part $\partial\Omega_T$ of the domain boundary.

In axisymmetric conditions, the full 3D domain Ω can be decomposed as $\Omega = \omega \times [0; 2\pi]$ where the interval represents the θ variable. The integration measures therefore reduce to $d\Omega = d\omega \cdot (r d\theta)$ and $dS = ds \cdot (r d\theta)$ where dS is the surface integration measure on the 3D domain Ω and ds its counterpart on the cross-section boundary $\partial\omega$. Exploiting the invariance of all fields with respect to θ, the previous virtual work principle is reformulated on the cross-section only as follows:

Find $\boldsymbol{u} \in V$ s.t. $\int_\omega \boldsymbol{\sigma}(\boldsymbol{u}) : \boldsymbol{\varepsilon}(\boldsymbol{v}) r d\omega = \int_{\partial\omega_T} \boldsymbol{T} \cdot \boldsymbol{v} r ds \quad \forall \boldsymbol{v} \in V$

where the 2π constants arising from the integration on θ have been cancelled on both sides. As a result, the bilinear and linear form are similar to the plane 2D case with the exception of the additional r term in the integration measures.

The final formulation is therefore pretty straightforward. Since a uniform pressure loading is applied on the outer boundary, we will also need the exterior normal vector to define the work of external forces form.

[11]:
```
n = FacetNormal(mesh)
p = Constant(10.)

V = VectorFunctionSpace(mesh, 'CG', degree=2)
du = TrialFunction(V)
u_ = TestFunction(V)
a = inner(sigma(du), eps(u_))*x[0]*dx
l = inner(-p*n, u_)*x[0]*ds(3)

u = Function(V, name="Displacement")
```

First, smooth contact conditions are assumed on both $r = 0$ (`Left`) and $z = 0$ (`Bottom`) boundaries. For this specific case, the solution corresponds to the classical hollow sphere under external pressure with a purely radial displacement:

$$u_r(r) = -\frac{R_e^3}{R_e^3 - R_i^3}\left((12\nu)r + (1+\nu)\frac{R_i^3}{2r^2}\right)\frac{p}{E}, \quad u_z = 0$$

[12]:
```
bcs = [DirichletBC(V.sub(1), Constant(0), facets, 1),
       DirichletBC(V.sub(0), Constant(0), facets, 2)]
solve(a == l, u, bcs)
print("Inwards radial displacement at (r=Re, theta=0): \
{:1.7f} (FE) {:1.7f} (Exact)".format(-u(Re, 0.)[0], float(Re**3/(Re**3-Ri**3)*((1-
↪2*nu)*Re+(1+nu)*Ri**3/2/Re**2)*p/E)))
print("Inwards radial displacement at (r=Ri, theta=0): \
{:1.7f} (FE) {:1.7f} (Exact)".format(-u(Ri, 0.)[0], float(Re**3/(Re**3-Ri**3)*((1-
↪2*nu)*Ri+(1+nu)*Ri/2)*p/E)))
```

```
Inwards radial displacement at (r=Re, theta=0): 0.0018375 (FE) 0.0018387 (Exact)
Inwards radial displacement at (r=Ri, theta=0): 0.0020879 (FE) 0.0020894 (Exact)
```

[13]:
```
plt.figure()
plot(100*u, mode="displacement")
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

The second loading case corresponds to a fully clamped condition on $z = 0$, the vertical boundary remaining in smooth contact.

[14]:
```
bcs = [DirichletBC(V, Constant((0., 0.)), facets, 1),
       DirichletBC(V.sub(0), Constant(0), facets, 2)]
solve(a == l, u, bcs)
plt.figure()
plot(mesh, linewidth=0.2)
plot(200*u, mode="displacement")
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

The corresponding files can be obtained from:

- Jupyter Notebook: `thermoelasticity.ipynb`
- Python script: `thermoelasticity.py`

2.4 Linear thermoelasticity (weak coupling)

2.4.1 Introduction

In this tour, we will solve a linear thermoelastic problem. In permanent regime, the temperature field is uncoupled from the mechanical fields whereas the latter depend on the temperature due to presence of thermal strains in the thermoelastic constitutive relation. This situation can be described as *weak* thermomechanical coupling. *Full* thermomechanical coupling for a transient evolution problem is treated *in this tour*.

In the present setting, the temperature field can be either given as a `Constant` or an `Expression` throughout the domain or obtained as the solution of a steady-state heat (Poisson) equation. This last case will be implemented in this tour.

2.4.2 Problem position

We consider the case of a rectangular 2D domain of dimensions $L \times H$ fully clamped on both lateral sides and subjected to a self-weight loading. The bottom side is subjected to a uniform temperature increase of $\Delta T = +50°C$ while the top and lateral boundaries remain at the initial temperature T_0. The geometry and boundary regions are first defined.

[1]:
```
from dolfin import *
from mshr import *
import matplotlib.pyplot as plt
%matplotlib notebook

L, H = 5, 0.3
mesh = RectangleMesh(Point(0., 0.), Point(L, H), 100, 10, "crossed")

def lateral_sides(x, on_boundary):
    return (near(x[0], 0) or near(x[0], L)) and on_boundary
def bottom(x, on_boundary):
    return near(x[1], 0) and on_boundary
def top(x, on_boundary):
    return near(x[1], H) and on_boundary
```

Because of the weak coupling discussed before, the thermal and mechanical problem can be solved separately. As a result, we don't need to resort to Mixed FunctionSpaces but can just define separately both problems.

2.4.3 Resolution of the thermal problem

The temperature is solution to the following equation $\text{div}(k\nabla T) = 0$ where k is the thermal conductivity (here we have no heat source). Since k is assumed to be homogeneous, it will not influence the solution. We therefore obtain a standard Poisson equation without forcing terms. Its formulation and resolution in FEniCS is quite standard with the temperature variation ΔT as the main unknown.

> **Note**: We needed to define the linear form `LT` in order to use the standard `solve(aT == LT, T, bcT)` syntax for linear problems. We could also have equivalently defined the residual `res = dot(grad(T), grad(T_))*dx` and used the nonlinear-like syntax `solve(res == 0, T, bcT)`.

[2]:
```
VT = FunctionSpace(mesh, "CG", 1)
T_, dT = TestFunction(VT), TrialFunction(VT)
Delta_T = Function(VT, name="Temperature increase")
aT = dot(grad(dT), grad(T_))*dx
LT = Constant(0)*T_*dx

bcT = [DirichletBC(VT, Constant(50.), bottom),
       DirichletBC(VT, Constant(0.), top),
       DirichletBC(VT, Constant(0.), lateral_sides)]
solve(aT == LT, Delta_T, bcT)
plt.figure()
p = plot(Delta_T, mode="contour")
plt.colorbar(p)
plt.show()
```

```
Calling FFC just-in-time (JIT) compiler, this may take some time.
Calling FFC just-in-time (JIT) compiler, this may take some time.
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

2.4.4 Mecanical problem

The linearized thermoelastic constitutive equation is given by:

$$\boldsymbol{\sigma} = \mathbb{C} : (\boldsymbol{\varepsilon} - \alpha(T - T_0)\mathbf{1}) = \lambda \text{tr}(\boldsymbol{\varepsilon})\mathbf{1} + 2\mu\boldsymbol{\varepsilon} - \alpha(3\lambda + 2\mu)(T - T_0)\mathbf{1}$$

where λ, μ are the Lamé parameters and α is the thermal expansion coefficient. As regards the current problem, the last term corresponding to the thermal strains is completely known. The following formulation can thus be generalized to any kind of known initial stress or eigenstrain state such as pre-stress or phase changes.

The weak formulation for the mechanical problem builds upon the presentation of the *2D linear elasticity* tour. The main difference lies in the presence of the temperature term in the constitutive relation which introduces a linear term with respect to the TestFunction `u_` when writing the work of internal forces `inner(sigma(du, Delta_T), eps(u_))*dx`. As a result, the bilinear form is extracted using the left-hand side `lhs` function whereas the thermal

strain term, acting as a loading term, is extracted using the right-hand side `rhs` function and added to the work of external forces when defining the linear form `LM`.

[3]:
```
E = Constant(50e3)
nu = Constant(0.2)
mu = E/2/(1+nu)
lmbda = E*nu/(1+nu)/(1-2*nu)
alpha = Constant(1e-5)

f = Constant((0, 0))

def eps(v):
    return sym(grad(v))
def sigma(v, dT):
    return (lmbda*tr(eps(v))- alpha*(3*lmbda+2*mu)*dT)*Identity(2) + 2.0*mu*eps(v)

Vu = VectorFunctionSpace(mesh, 'CG', 2)
du = TrialFunction(Vu)
u_ = TestFunction(Vu)
Wint = inner(sigma(du, Delta_T), eps(u_))*dx
aM = lhs(Wint)
LM = rhs(Wint) + inner(f, u_)*dx

bcu = DirichletBC(Vu, Constant((0., 0.)), lateral_sides)

u = Function(Vu, name="Displacement")
```

First, the self-weight loading is deactivated, only thermal stresses are computed.

[4]:
```
solve(aM == LM, u, bcu)

plt.figure()
p = plot(1e3*u[1],title="Vertical displacement [mm]")
plt.colorbar(p)
plt.show()
plt.figure()
p = plot(sigma(u, Delta_T)[0, 0],title="Horizontal stress [MPa]")
plt.colorbar(p)
plt.show()
```

```
Calling FFC just-in-time (JIT) compiler, this may take some time.
Calling FFC just-in-time (JIT) compiler, this may take some time.
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
Calling FFC just-in-time (JIT) compiler, this may take some time.
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
Calling FFC just-in-time (JIT) compiler, this may take some time.
```

We now take into account the self-weight.

[5]:
```
rho_g = 2400*9.81e-6
f.assign(Constant((0., -rho_g)))
solve(aM == LM, u, bcu)
```

(continues on next page)

(continued from previous page)

```
plt.figure()
p = plot(1e3*u[1],title="Vertical displacement [mm]")
plt.colorbar(p)
plt.show()
plt.figure()
p = plot(sigma(u, Delta_T)[0, 0],title="Horizontal stress [MPa]")
plt.colorbar(p)
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[ ]:

The corresponding files can be obtained from:

- Jupyter Notebook: `thermoelasticity_transient.ipynb`
- Python script: `thermoelasticity_transient.py`

2.5 Thermo-elastic evolution problem (full coupling)

2.5.1 Introduction

In this tour, we will solve a transient thermoelastic evolution problem in which both thermo-mechanical fields are fully coupled, we will however assume that the evolution is quasi-static and will hence neglect inertial effects. Note that a staggered approach could also have been adopted in which one field is calculated first (say the temperature for instance) using predicted values of the other field and similarly for the other field in a second step (see for instance *[FAR91]*).

Static elastic computation with thermal strains is treated in the LinearThermoelasticity tour.

2.5.2 Problem position

The problem consists of a quarter of a square plate perforated by a circular hole. Thermo-elastic properties are isotropic and correspond to those of aluminium (note that stress units are in MPa, distances in m and mass in kg). Linearized thermo-elasticity will be considered around a reference temperature of $T_0 = 293\,\mathrm{K}$. A temperature increase of $\Delta T = +10°\mathrm{C}$ will be applied on the hole boundary. Symmetry conditions are applied on the corresponding symmetry planes and stress and flux-free boundary conditions are adopted on the plate outer boundary.

We first import the relevant modules and define the mesh and material parameters (see *the next section* for more details on the parameters).

[1]:
```
from dolfin import *
from mshr import *
import numpy as np
import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
%matplotlib notebook

L = 1.
R = 0.1
N = 50  # mesh density

domain = Rectangle(Point(0., 0.), Point(L, L)) - Circle(Point(0., 0.), R, 100)
mesh = generate_mesh(domain, N)

T0 = Constant(293.)
DThole = Constant(10.)
E = 70e3
nu = 0.3
lmbda = Constant(E*nu/((1+nu)*(1-2*nu)))
mu = Constant(E/2/(1+nu))
rho = Constant(2700.)      # density
alpha = 2.31e-5  # thermal expansion coefficient
kappa = Constant(alpha*(2*mu + 3*lmbda))
cV = Constant(910e-6)*rho  # specific heat per unit volume at constant strain
k = Constant(237e-6)  # thermal conductivity
```

We now define the relevant FunctionSpace for the considered problem. Since we will adopt a monolithic approach i.e. in which both fields are coupled and solved at the same time, we will need to resort to a Mixed FunctionSpace for both the displacement $\boldsymbol{u}$ and the temperature variation $\Theta = T - T_0$. For an introduction on the use of Mixed FunctionSpace, check out the FEniCS tutorials on the mixed Poisson equation or the Stokes problem. Let us just point out that the constructor using `MixedFunctionSpace` has been deprecated since version 2016.2 (see this post). In the following code, `MixedElement([Vue, Vte])` could also be replaced by `Vue * Vte`.

[2]:
```
Vue = VectorElement('CG', mesh.ufl_cell(), 2) # displacement finite element
Vte = FiniteElement('CG', mesh.ufl_cell(), 1) # temperature finite element
V = FunctionSpace(mesh, MixedElement([Vue, Vte]))

def inner_boundary(x, on_boundary):
    return near(x[0]**2+x[1]**2, R**2, 1e-3) and on_boundary
def bottom(x, on_boundary):
    return near(x[1], 0) and on_boundary
def left(x, on_boundary):
    return near(x[0], 0) and on_boundary

bc1 = DirichletBC(V.sub(0).sub(1), Constant(0.), bottom)
bc2 = DirichletBC(V.sub(0).sub(0), Constant(0.), left)
bc3 = DirichletBC(V.sub(1), DThole, inner_boundary)
bcs = [bc1, bc2, bc3]
```

Dirichlet boundary conditions must be defined from the full FunctionSpace `V` using the appropriate subspaces that is `V.sub(0)` for the displacement (and `.sub(0)` or `.sub(1)` for the corresponding x/y component) and `V.sub(1)` for the temperature. Note also that in the following, we will in fact work with the temperature variation $\Theta = T - T_0$ as a field unkwown instead of the total temperature. Hence, the boundary condition on the hole boundary reads indeed as $\Delta T = +10°\text{C}$.

> **Note:** The definition of the `inner_boundary` region used the syntax `near(expr, value, tolerance)` to define the region where `expr = value` up to the specified `tolerance`. The given tolerance is quite large given that `mshr` generates a faceted approximation of a Circle (here using 100 segments). Mesh nodes may therefore not lie exactly on the true circle.

2.5.3 Variational formulation and time discretization

The linearized thermoelastic constitutive equations are given by:

$$\boldsymbol{\sigma} = \mathbb{C} : (\boldsymbol{\varepsilon} - \alpha(T - T_0)\mathbf{1}) = \lambda \text{tr}(\boldsymbol{\varepsilon})\mathbf{1} + 2\mu\boldsymbol{\varepsilon} - \kappa(T - T_0)\mathbf{1}$$

$$\rho s = \rho s_0 + \frac{\rho C_\varepsilon}{T_0}(T - T_0) + \kappa \text{tr}(\boldsymbol{\varepsilon})$$

- λ, μ the Lamé coefficients
- ρ material density
- α thermal expansion coefficient
- $\kappa = \alpha(3\lambda + 2\mu)$
- C_ε the specific heat at constant strain (per unit of mass).
- s (resp. s_0) the entropy per unit of mass in the current (resp. initial configuration)

These equations are completed by the equilibrium equation which will later be expressed in its weak form (virtual work principle) and the linearized heat equation (without source terms):

$$\rho T_0 \dot{s} + \text{div}\boldsymbol{q} = 0$$

where the heat flux is related to the temperature gradient through the isotropic Fourier law: $\boldsymbol{q} = -k\nabla T$ with k being the thermal conductivity. Using the entropy constitutive relation, the weak form of the heat equation reads as:

$$\int_\Omega \rho T_0 \dot{s}\widehat{T} d\Omega - \int_\Omega \boldsymbol{q} \cdot \nabla\widehat{T} d\Omega = -\int_{\partial\Omega} \boldsymbol{q} \cdot \boldsymbol{n}\widehat{T} dS \quad \forall \widehat{T} \in V_T$$

$$\int_\Omega \left(\rho C_\varepsilon \dot{T} + \kappa T_0 \text{tr}(\dot{\boldsymbol{\varepsilon}})\right)\widehat{T} d\Omega + \int_\Omega k\nabla T \cdot \nabla\widehat{T} d\Omega = \int_{\partial\Omega} k\partial_n T\widehat{T} dS \quad \forall \widehat{T} \in V_T$$

with V_T being the FunctionSpace for the temperature field.

The time derivatives are now replaced by an implicit Euler scheme, so that the previous weak form at the time increment $n+1$ is now:

$$\int_\Omega \left(\rho C_\varepsilon \frac{T-T_n}{\Delta t} + \kappa T_0 \text{tr}\left(\frac{\boldsymbol{\varepsilon}-\boldsymbol{\varepsilon}_n}{\Delta t}\right)\right)\widehat{T}d\Omega + \int_\Omega k\nabla T\cdot\nabla\widehat{T}d\Omega = \int_{\partial\Omega} k\partial_n T\widehat{T}dS \quad \forall\widehat{T}\in V_T \qquad (1)$$

where T and $\boldsymbol{\varepsilon}$ correspond to the *unknown* fields at the time increment $n+1$. For more details on the time discretization of the heat equation, see also the Heat equation FEniCS tutorial.

In addition to the previous thermal weak form, the mechanical weak form reads as:

$$\int_\Omega (\lambda \text{tr}(\boldsymbol{\varepsilon})\mathbf{1} + 2\mu\boldsymbol{\varepsilon} - \kappa(T-T_0)\mathbf{1}) : \nabla^s\widehat{\boldsymbol{v}}\ \text{d}\Omega = W_{ext}(\widehat{\boldsymbol{v}}) \quad \forall\widehat{\boldsymbol{v}}\in V_U \qquad (2)$$

where V_U is the displacement FunctionSpace and W_{ext} the linear functional corresponding to the work of external forces.

The solution of the coupled problem at $t=t_{n+1}$ is now $(\boldsymbol{u}_{n+1}, T_{n+1}) = (\boldsymbol{u}, T) \in V_U \times V_T$ verifying (1) and (2). These two forms are implemented below with zero right-hand sides (zero Neumann BCs for both problems here). One slight modification is that the temperature unknown T is replaced by the temperature variation $\Theta = T - T_0$ which appears naturally in the stress constitutive relation.

> **Note**: We will later make use of the `lhs` and `rhs` functions to extract the corresponding bilinear and linear forms.

[3]:
```
U_ = TestFunction(V)
(u_, Theta_) = split(U_)
dU = TrialFunction(V)
(du, dTheta) = split(dU)
Uold = Function(V)
(uold, Thetaold) = split(Uold)


def eps(v):
    return sym(grad(v))


def sigma(v, Theta):
    return (lmbda*tr(eps(v)) - kappa*Theta)*Identity(2) + 2*mu*eps(v)


dt = Constant(0.)
mech_form = inner(sigma(du, dTheta), eps(u_))*dx
therm_form = (cV*(dTheta-Thetaold)/dt*Theta_ +
              kappa*T0*tr(eps(du-uold))/dt*Theta_ +
              dot(k*grad(dTheta), grad(Theta_)))*dx
form = mech_form + therm_form
```

2.5.4 Resolution

The problem is now solved by looping over time increments. Because of the typical exponential time variation of temperature evolution of the heat equation, time steps are discretized on a non-uniform (logarithmic) scale. Δt is therefore updated at each time step. Note that since we work in terms of temperature variation and not absolute temperature all fields can be initialized to zero, otherwise T would have needed to be initialized to the reference temperature T_0.

[ ]:
```
Nincr = 100
t = np.logspace(1, 4, Nincr+1)
Nx = 100
x = np.linspace(R, L, Nx)
T_res = np.zeros((Nx, Nincr+1))
U = Function(V)
for (i, dti) in enumerate(np.diff(t)):
    print("Increment " + str(i+1))
    dt.assign(dti)
    solve(lhs(form) == rhs(form), U, bcs)
    Uold.assign(U)
    T_res[:, i+1] = [U(xi, 0.)[2] for xi in x]
```

At each time increment, the variation of the temperature increase Θ along a line $(x, y = 0)$ is saved in the `T_res` array. This evolution is plotted below. As expected, the temperature gradually increases over time, reaching eventually a uniform value of $+10°$C over infinitely long waiting time.

[5]:
```
plt.figure()
plt.plot(x, T_res[:, 1::Nincr//10])
plt.xlabel("$x$-coordinate along $y=0$")
plt.ylabel("Temperature variation $\Theta$")
plt.legend(["$t={:.0f}$".format(ti) for ti in t[1::Nincr//10]], ncol=2)
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

The final stresses and temperature variation are plotted using `matplotlib`. Note that we can add colorbar and change the plot axis limit.

[6]:
```
u, Theta = split(U)

plt.figure()
p = plot(sigma(u, Theta)[1, 1], title="$\sigma_{yy}$ stress near the hole")
plt.xlim((0, 3*R))
plt.ylim((0, 3*R))
plt.colorbar(p)
plt.show()

plt.figure()
p = plot(Theta, title="Temperature variation")
plt.xlim((0, 3*R))
plt.ylim((0, 3*R))
plt.colorbar(p)
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

2.5.5 References

[FAR91]: Farhat, C., Park, K. C., & Dubois-Pelerin, Y. (1991). An unconditionally stable staggered algorithm for transient finite element analysis of coupled thermoelastic problems. Computer Methods in Applied Mechanics and Engineering, 85(3), 349-365. https://doi.org/10.1016/0045-7825(91)90102-C

[ ]:

2.6 Modal analysis of an elastic structure

2.6.1 Introduction

This program performs a dynamic modal analysis of an elastic cantilever beam represented by a 3D solid continuum. The eigenmodes are computed using the **SLEPcEigensolver** and compared against an analytical solution of beam theory. The corresponding file can be obtained from `cantilever_modal.py`.

The first four eigenmodes of this demo will look as follows:

The first two fundamental modes are on top with bending along the weak axis (left) and along the strong axis (right), the next two modes are at the bottom.

2.6.2 Implementation

After importing the relevant modules, the geometry of a beam of length $L = 20$ and rectangular section of size $B \times H$ with $B = 0.5, H = 1$ is first defined:

```
from dolfin import *
import numpy as np

L, B, H = 20., 0.5, 1.

Nx = 200
Ny = int(B/L*Nx)+1
Nz = int(H/L*Nx)+1

mesh = BoxMesh(Point(0.,0.,0.),Point(L,B,H), Nx, Ny, Nz)
```

Material parameters and elastic constitutive relations are classical (here we take $\nu = 0$) and we also introduce the material density ρ for later definition of the mass matrix:

```
E, nu = Constant(1e5), Constant(0.)
rho = Constant(1e-3)

# Lame coefficient for constitutive relation
mu = E/2./(1+nu)
lmbda = E*nu/(1+nu)/(1-2*nu)
```

(continues on next page)

(continued from previous page)

```
def eps(v):
    return sym(grad(v))
def sigma(v):
    dim = v.geometric_dimension()
    return 2.0*mu*eps(v) + lmbda*tr(eps(v))*Identity(dim)
```

Standard FunctionSpace is defined and boundary conditions correspond to a fully clamped support at $x = 0$:

```
V = VectorFunctionSpace(mesh, 'Lagrange', degree=1)
u_ = TrialFunction(V)
du = TestFunction(V)


def left(x, on_boundary):
    return near(x[0],0.)

bc = DirichletBC(V, Constant((0.,0.,0.)), left)
```

The system stiffness matrix $[K]$ and mass matrix $[M]$ are respectively obtained from assembling the corresponding variational forms:

```
k_form = inner(sigma(du),eps(u_))*dx
l_form = Constant(1.)*u_[0]*dx
K = PETScMatrix()
b = PETScVector()
assemble_system(k_form, l_form, bc, A_tensor=K, b_tensor=b)

m_form = rho*dot(du,u_)*dx
M = PETScMatrix()
assemble(m_form, tensor=M)
```

Matrices $[K]$ and $[M]$ are first defined as PETSc Matrix and forms are assembled into it to ensure that they have the right type. Note that boundary conditions have been applied to the stiffness matrix using `assemble_system` so as to preserve symmetry (a dummy `l_form` and right-hand side vector have been introduced to call this function).

Modal dynamic analysis consists in solving the following generalized eigenvalue problem $[K]\{U\} = \lambda[M]\{U\}$ where the eigenvalue is related to the eigenfrequency $\lambda = \omega^2$. This problem can be solved using the `SLEPcEigenSolver`.

```
eigensolver = SLEPcEigenSolver(K, M)
eigensolver.parameters['problem_type'] = 'gen_hermitian'
eigensolver.parameters['spectral_transform'] = 'shift-and-invert'
eigensolver.parameters['spectral_shift'] = 0.
```

The problem type is specified to be a generalized eigenvalue problem with Hermitian matrices. By default, SLEPc computes the largest eigenvalues. Here we instead look for the smallest eigenvalues (they should all be real). A spectral transform is therefore performed using the keyword `shift-invert` i.e. the original problem is transformed into an equivalent problem with eigenvalues given by $\dfrac{1}{\lambda - \sigma}$ instead of λ where σ is the value of the spectral shift. It is therefore much easier to compute eigenvalues close to σ i.e. close to $\sigma = 0$ in the present case. Eigenvalues are then transformed back by SLEPc to their original value λ.

We now ask SLEPc to extract the first 6 eigenvalues by calling its solve function and extract the corresponding eigenpair (first two arguments of `get_eigenpair` correspond to the real and complex part of the eigenvalue, the last two to the real and complex part of the eigenvector):

```
N_eig = 6   # number of eigenvalues
print("Computing {} first eigenvalues...".format(N_eig))
eigensolver.solve(N_eig)

# Exact solution computation
from scipy.optimize import root
from math import cos, cosh
falpha = lambda x: cos(x)*cosh(x)+1
alpha = lambda n: root(falpha, (2*n+1)*pi/2.)['x'][0]

# Set up file for exporting results
file_results = XDMFFile("modal_analysis.xdmf")
file_results.parameters["flush_output"] = True
file_results.parameters["functions_share_mesh"] = True

# Extraction
for i in range(N_eig):
    # Extract eigenpair
    r, c, rx, cx = eigensolver.get_eigenpair(i)

    # 3D eigenfrequency
    freq_3D = sqrt(r)/2/pi

    # Beam eigenfrequency
    if i % 2 == 0: # exact solution should correspond to weak axis bending
        I_bend = H*B**3/12.
    else:          #exact solution should correspond to strong axis bending
        I_bend = B*H**3/12.
    freq_beam = alpha(i/2)**2*sqrt(E*I_bend/(rho*B*H*L**4))/2/pi

    print("Solid FE: {0:8.5f} [Hz]   Beam theory: {1:8.5f} [Hz]".format(freq_3D, freq_
↪beam))

    # Initialize function and assign eigenvector
    eigenmode = Function(V,name="Eigenvector "+str(i))
    eigenmode.vector()[:] = rx
```

The beam analytical solution is obtained using the eigenfrequencies of a clamped beam in bending given by $\omega_n = \alpha_n^2\sqrt{\dfrac{EI}{\rho S L^4}}$ where $S = BH$ is the beam section, I the bending inertia and α_n is the solution of the following nonlinear equation:

$$\cos(\alpha)\cosh(\alpha) + 1 = 0$$

the solution of which can be well approximated by $(2n+1)\pi/2$ for $n \geq 3$. Since the beam possesses two bending axis, each solution to the previous equation is associated with two frequencies, one with bending along the weak axis ($I = I_{\text{weak}} = HB^3/12$) and the other along the strong axis ($I = I_{\text{strong}} = BH^3/12$). Since $I_{\text{strong}} = 4I_{\text{weak}}$ for the considered numerical values, the strong axis bending frequency will be twice that corresponsing to bending along the weak axis. The solution α_n are computed using the `scipy.optimize.root` function with initial guess given by $(2n+1)\pi/2$.

With `Nx=400`, we obtain the following comparison between the FE eigenfrequencies and the beam theory eigenfrequencies :

Mode	Eigenfrequencies	
#	Solid FE [Hz]	Beam theory [Hz]
1	2.04991	2.01925
2	4.04854	4.03850
3	12.81504	12.65443
4	25.12717	25.30886
5	35.74168	35.43277
6	66.94816	70.86554

2.7 Time-integration of elastodynamics equation

This demo is implemented in a single Python file, `demo_elastodynamics.py`, which contains both the variational forms and the solver.

This demo shows how to perform time integration of transient elastodynamics using the generalized-α method [ERL2002]. In particular it demonstrates how to:

- formulate mass and damping forms of the elastodynamics equation
- implement the generalized-α method and its influence on the solution
- perform efficient computation of stresses using `LocalSolver`

The deformed structure evolution over time along with the axial stress will look as follows:

2.7.1 Introduction and elastodynamics equation

The elastodynamics equation combine the balance of linear momentum:

$$\nabla \cdot \sigma + \rho b = \rho \ddot{u}$$

where u is the displacement vector field, $\ddot{u} = \partial^2 u / \partial t^2$ is the acceleration, ρ the material density, b a given body force and σ the stress tensor which is related to the displacement through a constitutive equation. In the case of isotropic linearized elasticity, one has:

$$\sigma = \lambda \text{tr}(\varepsilon) \mathbb{1} + 2\mu\varepsilon$$

where $\varepsilon = (\nabla u + (\nabla u)^T)/2$ is the linearized strain tensor, $\mathbb{1}$ is the identity of second-rank tensors and $\lambda = \dfrac{E\nu}{(1+\nu)(1-2\nu)}, \mu = \dfrac{E}{2(1+\nu)}$ are the Lame coefficients given as functions of the Young modulus E and the Poisson ratio ν.

The weak form is readily obtained by integrating by part the balance equation using a test function $v \in V$ with V being a suitable function space that satisfies the displacement boundary conditions:

$$\int_\Omega \rho \ddot{u} \cdot v \, \mathrm{d}x + \int_\Omega \sigma(u) : \varepsilon(v) \, \mathrm{d}x = \int_\Omega \rho b \cdot v \, \mathrm{d}x + \int_{\partial\Omega} (\sigma \cdot n) \cdot v \, \mathrm{d}s \quad \text{for all } v \in V$$

The previous equation can be written as follows:

$$\text{Find } u \in V \text{ such that } m(\ddot{u}, v) + k(u, v) = L(v) \quad \text{for all } v \in V$$

where m is the symmetric bilinear form associated with the mass matrix and k the one associated with the stiffness matrix.

After introducing the finite element space interpolation, one obtains the corresponding discretized evolution equation:

$$\text{Find } \{u\} \in \mathbb{R}^n \text{ such that } \{v\}^T[M]\{\ddot{u}\} + \{v\}^T[K]\{u\} = \{v\}^T\{F\} \quad \text{for all } \{v\} \in \mathbb{R}^n$$

which is a generalized n-dof harmonic oscillator equation.

Quite often in structural dynamics, structures do not oscillate perfectly but lose energy through various dissipative mechanisms (friction with air or supports, internal dissipation through plasticity, damage, etc.). Dissipative terms can be introduced at the level of the constitutive equation if these mechanisms are well known but quite often it is not the case. Dissipation can then be modeled by adding an *ad hoc* damping term depending on the structure velocity $\dot{u}$ to the previous evolution equation:

$$\text{Find } u \in V \text{ such that } m(\ddot{u}, v) + c(\dot{u}, v) + k(u, v) = L(v) \quad \text{for all } v \in V$$

The damping form will be considered here as bilinear and symmetric, being therefore associated with a damping matrix $[C]$.

Rayleigh damping

When little is known about the origin of damping in the structure, a popular choice for the damping matrix, known as *Rayleigh damping*, consists in using a linear combination of the mass and stiffness matrix $[C] = \eta_M[M] + \eta_K[K]$ with two positive parameters η_M, η_K which can be fitted against experimental measures for instance (usually by measuring the damping ratio of two natural modes of vibration).

2.7.2 Time discretization using the generalized-α method

We now introduce a time discretization of the interval study $[0; T]$ in $N+1$ time increments $t_0 = 0, t_1, \ldots, t_N, t_{N+1} = T$ with $\Delta t = T/N$ denoting the time step (supposed constant). The resolution will make use of the generalized-α method which can be seen as an extension of the widely used Newmark-β method in structural dynamics. As an implicit method, it is unconditionally stable for a proper choice of coefficients so that quite large time steps can be used. It also allows for high frequency dissipation and offers a second-order accuracy, i.e. in $O(\Delta t^2)$.

The method consists in solving the dynamic evolution equation at intermediate time between t_n and t_{n+1} as follows:

$$[M]\{\ddot{u}_{n+1-\alpha_m}\} + [C]\{\dot{u}_{n+1-\alpha_f}\} + [K]\{u_{n+1-\alpha_f}\} = \{F(t_{n+1-\alpha_f})\}$$

with the notation $X_{n+1-\alpha} = (1-\alpha)X_{n+1} + \alpha X_n$. In addition, the following approximation for the displacement and velocity at t_{n+1} are used:

$$\begin{aligned}
\{u_{n+1}\} &= \{u_n\} + \Delta t\{\dot{u}_n\} + \frac{\Delta t^2}{2}\left((1-2\beta)\{\ddot{u}_n\} + 2\beta\{\ddot{u}_{n+1}\}\right) \\
\{\dot{u}_{n+1}\} &= \{\dot{u}_n\} + \Delta t\left((1-\gamma)\{\ddot{u}_n\} + \gamma\{\ddot{u}_{n+1}\}\right)
\end{aligned}$$

It can be seen that these are the relations of the Newmark method. The latter is therefore obtained as a particular case when $\alpha_f = \alpha_m = 0$.

The problem can then be formulated in terms of unkown displacement at t_{n+1} with:

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

After plugging into the evolution and rearranging the known and unknown terms, one obtains the following system to solve:

$$\begin{aligned}
[\bar{K}]\{u_{n+1}\} = \{F(t_{n+1-\alpha_f})\} &- \alpha_f[K]\{u_n\} \\
&- [C](c_1\{u_n\} + c_2\{\dot{u}_n\} + c_3\{\ddot{u}_n\}) - [M](m_1\{u_n\} + m_2\{\dot{u}_n\} + m_3\{\ddot{u}_n\})
\end{aligned}$$

where:

- $[\bar{K}] = [K] + c_1[C] + m_1[M]$
- $c_1 = \dfrac{\gamma(1-\alpha_f)}{\beta\Delta t}$
- $c_2 = 1 - \gamma(1-\alpha_f)/\beta$
- $c_3 = \Delta t(1-\alpha_f)(1-\dfrac{\gamma}{2\beta})$
- $m_1 = \dfrac{(1-\alpha_m)}{\beta\Delta t^2}$
- $m_2 = \dfrac{(1-\alpha_m)}{\beta\Delta t}$
- $m_3 = 1 - \dfrac{1-\alpha_m}{2\beta}$

Once the linear system has been solved for $\{u_{n+1}\}$, the new velocity and acceleration are computed using the previous formulae.

Popular choice of parameters

The most popular choice for the parameters is: $\alpha_m, \alpha_f \leq 1/2$ and $\gamma = \dfrac{1}{2} + \alpha_m - \alpha_f$, $\beta = \dfrac{1}{4}\left(\gamma + \dfrac{1}{2}\right)^2$ which ensures unconditional stability, optimal dissipation and second-order accuracy.

2.7.3 Implementation

We consider a rectangular beam clamped at one end and loaded by a uniform vertical traction at the other end. After importing the relevant modules, the mesh and subdomains for boundary conditions are defined:

```
from dolfin import *
import numpy as np
import matplotlib.pyplot as plt

# Form compiler options
parameters["form_compiler"]["cpp_optimize"] = True
parameters["form_compiler"]["optimize"] = True

# Define mesh
mesh = BoxMesh(Point(0., 0., 0.), Point(1., 0.1, 0.04), 60, 10, 5)

# Sub domain for clamp at left end
def left(x, on_boundary):
    return near(x[0], 0.) and on_boundary

# Sub domain for rotation at right end
def right(x, on_boundary):
    return near(x[0], 1.) and on_boundary
```

Material parameters for the elastic constitutive relation, the material density ρ for the mass matrix and the two parameters defining the Rayleigh damping η_M, η_K (initially zero damping is considered but this value can be changed) are now defined:

```
# Elastic parameters
E  = 1000.0
nu = 0.3
mu    = Constant(E / (2.0*(1.0 + nu)))
lmbda = Constant(E*nu / ((1.0 + nu)*(1.0 - 2.0*nu)))

# Mass density
rho = Constant(1.0)

# Rayleigh damping coefficients
eta_m = Constant(0.)
eta_k = Constant(0.)
```

Parameters used for the time discretization scheme are now defined. First, the four parameters used by the generalized-α method are chosen. Here, we used the optimal dissipation and second-order accuracy choice for β and γ, namely $\beta = \frac{1}{4}\left(\gamma + \frac{1}{2}\right)^2$ and $\gamma = \frac{1}{2} + \alpha_m - \alpha_f$ with $\alpha_m = 0.2$ and $\alpha_f = 0.4$ ensuring unconditional stability:

```
# Generalized-alpha method parameters
alpha_m = Constant(0.2)
alpha_f = Constant(0.4)
gamma   = Constant(0.5+alpha_f-alpha_m)
beta    = Constant((gamma+0.5)**2/4.)
```

We also define the final time of the interval, the number of time steps and compute the associated time interval between two steps:

```
# Time-stepping parameters
T       = 4.0
Nsteps  = 50
dt = Constant(T/Nsteps)
```

We now define the time-dependent loading. Body forces are zero and the imposed loading consists of a uniform vertical traction applied at the `right` extremity. The loading amplitude will vary linearly from 0 to $p_0 = 1$ over the time interval $[0; T_c = T/5]$, after T_c the loading is removed. For this purpose, we used the following JIT-compiled `Expression`. In particular, it uses a conditional syntax using operators `?` and `:`

```
p0 = 1.
cutoff_Tc = T/5
# Define the loading as an expression depending on t
p = Expression(("0", "t <= tc ? p0*t/tc : 0", "0"), t=0, tc=cutoff_Tc, p0=p0,␣
↪degree=0)
```

A standard vectorial P^1 FunctionSpace is now defined for the displacement, velocity and acceleration fields. We also define a tensorial DG-0 FunctionSpace for saving the stress field evolution:

```
# Define function space for displacement, velocity and acceleration
V = VectorFunctionSpace(mesh, "CG", 1)
# Define function space for stresses
Vsig = TensorFunctionSpace(mesh, "DG", 0)
```

Test and trial functions are defined and the unkown displacement (corresponding to $\{u_{n+1}\}$ for the current time step) will be represented by the Function `u`. Displacement, velocity and acceleration fields of the previous increment t_n will respectively be represented by functions `u_old`, `v_old` and `a_old`:

```
# Test and trial functions
du = TrialFunction(V)
u_ = TestFunction(V)
# Current (unknown) displacement
u = Function(V, name="Displacement")
# Fields from previous time step (displacement, velocity, acceleration)
u_old = Function(V)
v_old = Function(V)
a_old = Function(V)
```

We now use a `MeshFunction` for distinguishing the different boundaries and mark the right extremity using an `AutoSubDomain`. The exterior surface measure `ds` is then defined using the boundary subdomains. Simple Dirichlet boundary conditions are also defined at the left extremity:

```
# Create mesh function over the cell facets
boundary_subdomains = MeshFunction("size_t", mesh, mesh.topology().dim() - 1)
boundary_subdomains.set_all(0)
force_boundary = AutoSubDomain(right)
force_boundary.mark(boundary_subdomains, 3)

# Define measure for boundary condition integral
dss = ds(subdomain_data=boundary_subdomains)

# Set up boundary condition at left end
zero = Constant((0.0, 0.0, 0.0))
bc = DirichletBC(V, zero, left)
```

Python functions are now defined to obtain the elastic stress tensor σ (linear isotropic elasticity), the bilinear mass and stiffness forms as well as the damping form obtained as a linear combination of the mass and stiffness forms (Rayleigh damping). The linear form corresponding to the work of external forces is also defined:

```
# Stress tensor
def sigma(r):
    return 2.0*mu*sym(grad(r)) + lmbda*tr(sym(grad(r)))*Identity(len(r))

# Mass form
def m(u, u_):
    return rho*inner(u, u_)*dx

# Elastic stiffness form
def k(u, u_):
    return inner(sigma(u), sym(grad(u_)))*dx

# Rayleigh damping form
def c(u, u_):
    return eta_m*m(u, u_) + eta_k*k(u, u_)

# Work of external forces
def Wext(u_):
    return dot(u_, p)*dss(3)
```

Functions for implementing the time stepping scheme are also defined. `update_a` returns $\{\ddot{u}_{n+1}\}$ as a function of the variables at the previous increment and of the new displacement $\{u_{n+1}\}$. The function accepts a keyword `ufl` so that the expressions involved can be used with UFL representations if `True` or with array of values if `False` (we will make use of both possibilities later). In particular, the time step `dt` and time-stepping scheme parameters are either `Constant` or floats depending on the case. Function `update_v` does the same but for the new velocity $\{\dot{u}_{n+1}\}$ as a function of the previous variables and of the new acceleration. Finally, function `update_fields` performs the

final update at the end of the time step when the new displacement $\{u_{n+1}\}$ has effectively been computed. In this context, the new acceleration and velocities are computed using the vector representation of the different fields. The variables keeping track of the values at the previous increment are now assigned the new values computed for the current increment:

```
# Update formula for acceleration
# a = 1/(2*beta)*((u - u0 - v0*dt)/(0.5*dt*dt) - (1-2*beta)*a0)
def update_a(u, u_old, v_old, a_old, ufl=True):
    if ufl:
        dt_ = dt
        beta_ = beta
    else:
        dt_ = float(dt)
        beta_ = float(beta)
    return (u-u_old-dt_*v_old)/beta_/dt_**2 - (1-2*beta_)/2/beta_*a_old

# Update formula for velocity
# v = dt * ((1-gamma)*a0 + gamma*a) + v0
def update_v(a, u_old, v_old, a_old, ufl=True):
    if ufl:
        dt_ = dt
        gamma_ = gamma
    else:
        dt_ = float(dt)
        gamma_ = float(gamma)
    return v_old + dt_*((1-gamma_)*a_old + gamma_*a)

def update_fields(u, u_old, v_old, a_old):
    """Update fields at the end of each time step."""

    # Get vectors (references)
    u_vec, u0_vec  = u.vector(), u_old.vector()
    v0_vec, a0_vec = v_old.vector(), a_old.vector()

    # use update functions using vector arguments
    a_vec = update_a(u_vec, u0_vec, v0_vec, a0_vec, ufl=False)
    v_vec = update_v(a_vec, u0_vec, v0_vec, a0_vec, ufl=False)

    # Update (u_old <- u)
    v_old.vector()[:], a_old.vector()[:] = v_vec, a_vec
    u_old.vector()[:] = u.vector()
```

The system variational form is now built by expressing the new acceleration $\{\ddot{u}_{n+1}\}$ as a function of the TrialFunction `du` using `update_a`, which here works as a UFL expression. Using this new acceleration, the same is done for the new velocity using `update_v`. Intermediate averages using parameters α_m, α_f of the generalized- α method are obtained with a user-defined fuction `avg`. The weak form evolution equation is then written using all these quantities. Since the problem is linear, we then extract the bilinear and linear parts using `rhs` and `lhs`:

```
def avg(x_old, x_new, alpha):
    return alpha*x_old + (1-alpha)*x_new

# Residual
a_new = update_a(du, u_old, v_old, a_old, ufl=True)
v_new = update_v(a_new, u_old, v_old, a_old, ufl=True)
res = m(avg(a_old, a_new, alpha_m), u_) + c(avg(v_old, v_new, alpha_f), u_) \
       + k(avg(u_old, du, alpha_f), u_) - Wext(u_)
a_form = lhs(res)
L_form = rhs(res)
```

Alternatively, the use of `derivative` can be made for non-linear problems for instance or one can also directly formulate the system to solve, involving the modified stiffness matrix $[\bar{K}]$ and the various coefficients introduced earlier.

Since the system matrix to solve is the same for each time step (constant time step), it is not necessary to factorize the system at each increment. It can be done once and for all and only perform assembly of the varying right-hand side and backsubstitution to obtain the solution much more efficiently. This is done by defining a `LUSolver` object and asking for reusing the matrix factorization:

```
# Define solver for reusing factorization
K, res = assemble_system(a_form, L_form, bc)
solver = LUSolver(K, "mumps")
solver.parameters["symmetric"] = True
```

We now initiate the time stepping loop. We will keep track of the beam vertical tip displacement over time as well as the different parts of the system total energy. We will also compute the stress field and save it, along with the displacement field, in a `XDMFFile`. The option *flush_ouput* enables to open the result file before the loop is finished, the `function_share_mesh` option tells that only one mesh is used for all functions of a given time step (displacement and stress) while the `rewrite_function_mesh` enforces that the same mesh is used for all time steps. These two options enables writing the mesh information only once instead of $2N_{steps}$ times:

```
# Time-stepping
time = np.linspace(0, T, Nsteps+1)
u_tip = np.zeros((Nsteps+1,))
energies = np.zeros((Nsteps+1, 4))
E_damp = 0
E_ext = 0
sig = Function(Vsig, name="sigma")
xdmf_file = XDMFFile("elastodynamics-results.xdmf")
xdmf_file.parameters["flush_output"] = True
xdmf_file.parameters["functions_share_mesh"] = True
xdmf_file.parameters["rewrite_function_mesh"] = False
```

The time loop is now started, the loading is first evaluated at $t = t_{n+1-\alpha_f}$. The corresponding system right-hand side is then assembled and the system is solved. The different fields are then updated with the newly computed quantities. Finally, some post-processing is performed: stresses are computed and written to the result file and the tip displacement and the different energies are recorded:

```
def local_project(v, V, u=None):
    """Element-wise projection using LocalSolver"""
    dv = TrialFunction(V)
    v_ = TestFunction(V)
    a_proj = inner(dv, v_)*dx
    b_proj = inner(v, v_)*dx
    solver = LocalSolver(a_proj, b_proj)
    solver.factorize()
    if u is None:
        u = Function(V)
        solver.solve_local_rhs(u)
        return u
    else:
        solver.solve_local_rhs(u)
        return

for (i, dt) in enumerate(np.diff(time)):

    t = time[i+1]
```

(continues on next page)

(continued from previous page)

```
    print("Time: ", t)

    # Forces are evaluated at t_{n+1-alpha_f}=t_{n+1}-alpha_f*dt
    p.t = t-float(alpha_f*dt)

    # Solve for new displacement
    res = assemble(L_form)
    bc.apply(res)
    solver.solve(K, u.vector(), res)


    # Update old fields with new quantities
    update_fields(u, u_old, v_old, a_old)

    # Save solution to XDMF format
    xdmf_file.write(u, t)

    # Compute stresses and save to file
    local_project(sigma(u), Vsig, sig)
    xdmf_file.write(sig, t)

    p.t = t
    # Record tip displacement and compute energies
    u_tip[i+1] = u(1., 0.05, 0.)[1]
    E_elas = assemble(0.5*k(u_old, u_old))
    E_kin = assemble(0.5*m(v_old, v_old))
    E_damp += dt*assemble(c(v_old, v_old))
    # E_ext += assemble(Wext(u-u_old))
    E_tot = E_elas+E_kin+E_damp #-E_ext
    energies[i+1, :] = np.array([E_elas, E_kin, E_damp, E_tot])
```

Note that in the above, the stresses are computed using a `LocalSolver` through the `local_project` function. Since the stress function space is a DG-0 space, the projection on this space can be performed element-wise in a very efficient manner. We therefore take advantage of the `LocalSolver` functionality which is precisely dedicated to such situations. Since this projection is performed at each time step, the savings in terms of computing time can be quite important.

As regards the computation of the various energies, the elastic and kinetic energies are respectively given by:

$$E_{elas} = \int_\Omega \frac{1}{2}\sigma(u) : \varepsilon(u)\,\mathrm{d}x$$

$$E_{kin} = \int_\Omega \frac{1}{2}\rho\dot{u}\cdot\dot{u}\,\mathrm{d}x$$

which are readily computed from the respective stiffness and mass forms `k` and `m` and the current displacement and velocity. The energy related to damping is computed from the corresponding dissipation term $\mathcal{D} = c(\dot{u},\dot{u})$ and integrated over time:

$$E_{damp} = \int_0^T \mathcal{D}\,\mathrm{d}t$$

As for the work developed by the external forces, the contribution to the energy is added at each time step. Finally, the total energy of the sytem is given by:

$$E_{tot} = E_{elas} + E_{kin} + E_{damp} - E_{ext}$$

When the time evolution loop is finished, the evolution of the tip displacement as well as the different contributions of the energy are plotted as functions of time:

```
# Plot tip displacement evolution
plt.figure()
plt.plot(time, u_tip)
plt.xlabel("Time")
plt.ylabel("Tip displacement")
plt.ylim(-0.5, 0.5)
plt.show()

# Plot energies evolution
plt.figure()
plt.plot(time, energies)
plt.legend(("elastic", "kinetic", "damping", "total"))
plt.xlabel("Time")
plt.ylabel("Energies")
plt.ylim(0, 0.0011)
plt.show()
```

2.7.4 Analyzing the results

We first consider the case of zero Rayleigh damping $\eta_M = \eta_K = 0$. In this case, it can be observed that the evolution of the total energy depends on the choice of the time-stepping scheme parameters. With $\alpha_m = \alpha_f = 0$, we recover the Newmark-β method with $\beta = 0.25, \gamma = 0.5$. This scheme is known for being conservative. This can be observed (figure-left) in the constant total energy for $t \geq T_c$ when the loading is removed. On the contrary, for non zero alpha parameters, e.g. $\alpha_m = 0.2, \alpha_f = 0.4$, it can be observed (figure-right) that the energy is decreasing during this phase, indicating numerical damping. For both cases, the scheme is unconditionally stable. Moreover, these differences vanish when reducing the time step.

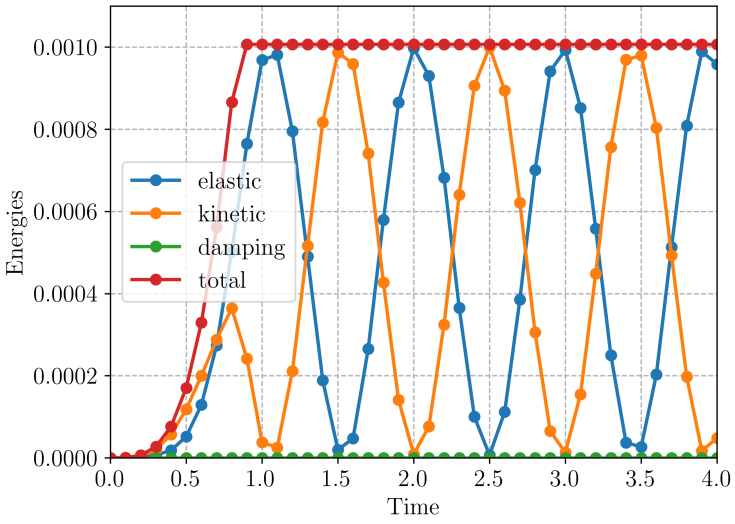


Fig. 1: Newmark-β method $\alpha_m = \alpha_f = 0$

For non-zero Rayleigh damping $\eta_M = \eta_K = 0.01$, the total energy including viscous dissipation tends to oscillate around a constant value, with oscillations vanishing for decreasing time steps.

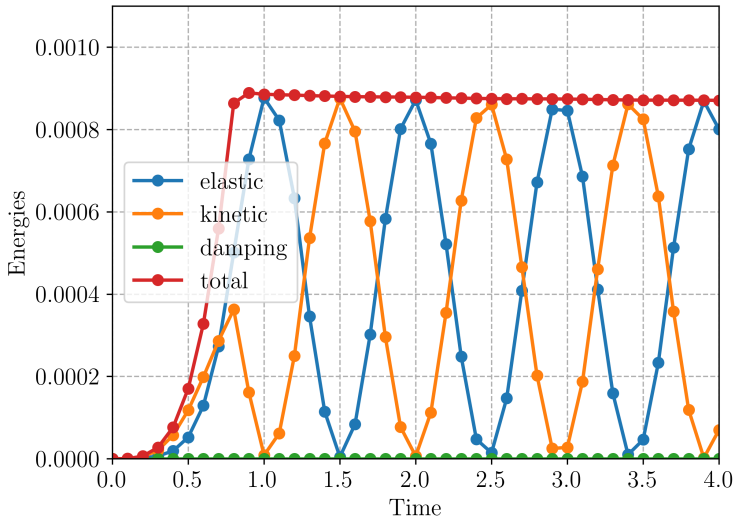
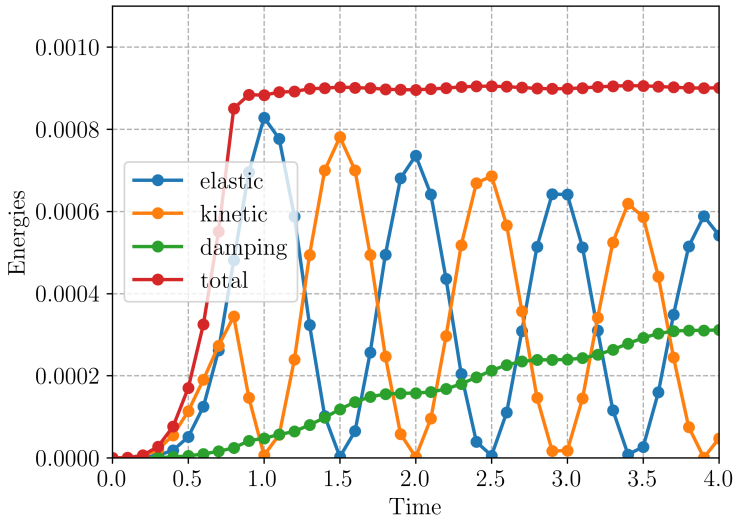


Fig. 2: Generalized-α with $\alpha_m = 0.2, \alpha_f = 0.4$



2.7.5 References

CHAPTER 3

Homogenization of heterogeneous materials

Contents:

The corresponding files can be obtained from:

- Jupyter Notebook: `periodic_homog_elas.ipynb`
- Python script: `periodic_homog_elas.py`

3.1 Periodic homogenization of linear elastic materials

3.1.1 Introduction

This tour will show how to perform periodic homogenization of linear elastic materials. The considered 2D plane strain problem deals with a skewed unit cell of dimensions $1 \times \sqrt{3}/2$ consisting of circular inclusions (numbered 1) of radius R with elastic properties (E_r, ν_r) and embedded in a matrix material (numbered 0) of properties (E_m, ν_m) following an hexagonal pattern. A classical result of homogenization theory ensures that the resulting overall behavior will be isotropic, a property that will be numerically verified later.

Complete sources including mesh files can be obtained from `periodic_homog_elas.zip`.

We suggest reading first the *2D linear elasticity* tour if you are not familiar with implementation of elastic materials.

[1]:
```
from __future__ import print_function
from dolfin import *
import numpy as np
import matplotlib.pyplot as plt
%matplotlib notebook

a = 1.          # unit cell width
b = sqrt(3.)/2. # unit cell height
c = 0.5         # horizontal offset of top boundary
```

(continues on next page)

(continued from previous page)

```
R = 0.2         # inclusion radius
vol = a*b       # unit cell volume
# we define the unit cell vertices coordinates for later use
vertices = np.array([[0, 0.],
                     [a, 0.],
                     [a+c, b],
                     [c, b]])
fname = "hexag_incl"
mesh = Mesh(fname + ".xml")
subdomains = MeshFunction("size_t", mesh, fname + "_physical_region.xml")
facets = MeshFunction("size_t", mesh, fname + "_facet_region.xml")
plt.figure()
plot(subdomains)
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Remark: `mshr` does not allow to generate a meshed domain with perfectly matching vertices on opposite boundaries as would be required when imposing periodic boundary conditions. For this reason, we used a `Gmsh`-generated mesh.

3.1.2 Periodic homogenization framework

The goal of homogenization theory consists in computing the apparent elastic moduli of the homogenized medium associated with a given microstructure. In a linear elastic setting, this amounts to solving the following auxiliary problem defined on the unit cell $\mathcal{A}$:

$$\begin{cases} \operatorname{div} \boldsymbol{\sigma} = \mathbf{0} & \text{in } \mathcal{A} \\ \boldsymbol{\sigma} = \mathbb{C}(\boldsymbol{y}) : \boldsymbol{\varepsilon} & \text{for } \boldsymbol{y} \in \mathcal{A} \\ \boldsymbol{\varepsilon} = \boldsymbol{E} + \nabla^s \boldsymbol{v} & \text{in } \mathcal{A} \\ \boldsymbol{v} & \text{is } \mathcal{A}\text{-periodic} \\ \boldsymbol{T} = \boldsymbol{\sigma} \cdot \boldsymbol{n} & \text{is } \mathcal{A}\text{-antiperiodic} \end{cases}$$

in $\mathrm{A}\boldsymbol{\sigma} = \mathrm{C}(\boldsymbol{y}){:}\boldsymbol{\varepsilon}$
in $\mathrm{A}\boldsymbol{v}$ for $\boldsymbol{y} \in \mathcal{A}\boldsymbol{\varepsilon} = \boldsymbol{E} + \nabla^s \boldsymbol{v}$
is A-antiperiodic is $\mathcal{A}$-periodic$\boldsymbol{T} = \boldsymbol{\sigma} \cdot \boldsymbol{n}$

where $\boldsymbol{E}$ is the **given** macroscopic strain, $\boldsymbol{v}$ a periodic fluctuation and $\mathbb{C}(\boldsymbol{y})$ is the heterogeneous elasticity tensor depending on the microscopic space variable $\boldsymbol{y} \in \mathcal{A}$. By construction, the local microscopic strain is equal on average to the macroscopic strain: $\langle \boldsymbol{\varepsilon} \rangle = \boldsymbol{E}$. Upon defining the macroscopic stress $\boldsymbol{\Sigma}$ as the microscopic stress average: $\langle \boldsymbol{\sigma} \rangle = \boldsymbol{\Sigma}$, there will be a linear relationship between the auxiliary problem loading parameters $\boldsymbol{E}$ and the resulting average stress:

$$\boldsymbol{\Sigma} = \mathbb{C}^{hom} : \boldsymbol{E}$$

where $\mathbb{C}^{hom}$ represents the apparent elastic moduli of the homogenized medium. Hence, its components can be computed by solving elementary load cases corresponding to the different components of $\boldsymbol{E}$ and performing a unit cell average of the resulting microscopic stress components.

Total displacement as the main unknown

The previous problem can also be reformulated by using the total displacement $\boldsymbol{u} = \boldsymbol{E} \cdot \boldsymbol{y} + \boldsymbol{v}$ as the main unknown with now $\boldsymbol{\varepsilon} = \nabla^s \boldsymbol{u}$. The periodicity condition is therefore equivalent to the following constraint:

$$\boldsymbol{u}(\boldsymbol{y}^+) - \boldsymbol{u}(\boldsymbol{y}^-) = \boldsymbol{E} \cdot (\boldsymbol{y}^+ - \boldsymbol{y}^-)$$

where $\boldsymbol{y}^\pm$ are opposite points on the unit cell boundary related by the periodicity condition. This formulation is widely used in solid mechanics FE software as it does not require specific change of the problem formulation but just adding tying constraints between some degrees of freedom.

This formulation is however not easy to translate in FEniCS. It would indeed require introducing Lagrange multipliers defined on some part of the border only, a feature which does not seem to be available at the moment.

Periodic fluctuation as the main unknown

Instead, we will keep the initial formulation and consider the periodic fluctuation $\boldsymbol{v}$ as the main unknown. The periodicity constraint on $\boldsymbol{v}$ will be imposed in the definition of the associated FunctionSpace using the `constrained_domain` optional keyword. To do so, one must define the periodic map linking the different unit cell boundaries. Here the unit cell is 2D and its boundary is represented by a parallelogram of vertices `vertices` and the corresponding base vectors `a1` and `a2` are computed. The right part is then mapped onto the left part, the top part onto the bottom part and the top-right corner onto the bottom-left one.

[2]:
```
# class used to define the periodic boundary map
class PeriodicBoundary(SubDomain):
    def __init__(self, vertices, tolerance=DOLFIN_EPS):
        """ vertices stores the coordinates of the 4 unit cell corners"""
        SubDomain.__init__(self, tolerance)
        self.tol = tolerance
        self.vv = vertices
        self.a1 = self.vv[1,:]-self.vv[0,:] # first vector generating periodicity
        self.a2 = self.vv[3,:]-self.vv[0,:] # second vector generating periodicity
        # check if UC vertices form indeed a parallelogram
        assert np.linalg.norm(self.vv[2, :]-self.vv[3, :] - self.a1) <= self.tol
        assert np.linalg.norm(self.vv[2, :]-self.vv[1, :] - self.a2) <= self.tol

    def inside(self, x, on_boundary):
        # return True if on left or bottom boundary AND NOT on one of the
        # bottom-right or top-left vertices
        return bool((near(x[0], self.vv[0,0] + x[1]*self.a2[0]/self.vv[3,1], self.
↪tol) or
                     near(x[1], self.vv[0,1] + x[0]*self.a1[1]/self.vv[1,0], self.
↪tol)) and
                     (not ((near(x[0], self.vv[1,0], self.tol) and near(x[1], self.
↪vv[1,1], self.tol)) or
                     (near(x[0], self.vv[3,0], self.tol) and near(x[1], self.vv[3,1],␣
↪self.tol)))) and on_boundary)

    def map(self, x, y):
        if near(x[0], self.vv[2,0], self.tol) and near(x[1], self.vv[2,1], self.tol):
↪# if on top-right corner
```

(continues on next page)

(continued from previous page)

```
            y[0] = x[0] - (self.a1[0]+self.a2[0])
            y[1] = x[1] - (self.a1[1]+self.a2[1])
        elif near(x[0], self.vv[1,0] + x[1]*self.a2[0]/self.vv[2,1], self.tol): # if␣
↪on right boundary
            y[0] = x[0] - self.a1[0]
            y[1] = x[1] - self.a1[1]
        else:   # should be on top boundary
            y[0] = x[0] - self.a2[0]
            y[1] = x[1] - self.a2[1]
```

We now define the constitutive law for both phases:

[3]:
```
Em = 50e3
num = 0.2
Er = 210e3
nur = 0.3
material_parameters = [(Em, num), (Er, nur)]
nphases = len(material_parameters)
def eps(v):
    return sym(grad(v))
def sigma(v, i, Eps):
    E, nu = material_parameters[i]
    lmbda = E*nu/(1+nu)/(1-2*nu)
    mu = E/2/(1+nu)
    return lmbda*tr(eps(v) + Eps)*Identity(2) + 2*mu*(eps(v)+Eps)
```

3.1.3 Variational formulation

The previous problem is very similar to a standard linear elasticity problem, except for the periodicity constraint which has now been included in the FunctionSpace definition and for the presence of an eigenstrain term $\boldsymbol{E}$. It can easily be shown that the variational formulation of the previous problem reads as: Find $\boldsymbol{v} \in V$ such that:

$$F(\boldsymbol{v}, \widehat{\boldsymbol{v}}) = \int_{\mathcal{A}} (\boldsymbol{E} + \nabla^s \boldsymbol{v}) : \mathbb{C}(\boldsymbol{y}) : \nabla^s \widehat{\boldsymbol{v}} \, \mathrm{d}\Omega = 0 \quad \forall \widehat{\boldsymbol{v}} \in V$$

The above problem is not well-posed because of the existence of rigid body translations. One way to circumvent this issue would be to fix one point but instead we will add an additional constraint of zero-average of the fluctuation field v as is classically done in homogenization theory. This is done by considering an additional vectorial Lagrange multiplier λ and considering the following variational problem (see the pure Neumann boundary conditions FEniCS demo for a similar formulation): Find $(\boldsymbol{v}, \boldsymbol{\lambda}) \in V \times \mathbb{R}^2$ such that:

$$\int_{\mathcal{A}} (\boldsymbol{E} + \nabla^s \boldsymbol{v}) : \mathbb{C}(\boldsymbol{y}) : \nabla^s \widehat{\boldsymbol{v}} \, \mathrm{d}\Omega + \int_{\mathcal{A}} \boldsymbol{\lambda} \cdot \widehat{\boldsymbol{v}} \, \mathrm{d}\Omega + \int_{\mathcal{A}} \widehat{\boldsymbol{\lambda}} \cdot \boldsymbol{v} \, \mathrm{d}\Omega = 0 \quad \forall (\widehat{\boldsymbol{v}}, \widehat{\boldsymbol{\lambda}}) \in V \times \mathbb{R}^2$$

Which can be summarized as:

$$a(\boldsymbol{v},\widehat{\boldsymbol{v}}) + b(\boldsymbol{\lambda},\widehat{\boldsymbol{v}}) + b(\widehat{\boldsymbol{\lambda}},\boldsymbol{v}) = L(\widehat{\boldsymbol{v}}) \quad \forall (\widehat{\boldsymbol{v}},\widehat{\boldsymbol{\lambda}}) \in V \times \mathbb{R}^2$$

This readily translates into the following FEniCS code:

[4]:
```
Ve = VectorElement("CG", mesh.ufl_cell(), 2)
Re = VectorElement("R", mesh.ufl_cell(), 0)
W = FunctionSpace(mesh, MixedElement([Ve, Re]), constrained_
↪domain=PeriodicBoundary(vertices, tolerance=1e-10))
V = FunctionSpace(mesh, Ve)

v_,lamb_ = TestFunctions(W)
dv, dlamb = TrialFunctions(W)
w = Function(W)
dx = Measure('dx')(subdomain_data=subdomains)

Eps = Constant(((0, 0), (0, 0)))
F = sum([inner(sigma(dv, i, Eps), eps(v_))*dx(i) for i in range(nphases)])
a, L = lhs(F), rhs(F)
a += dot(lamb_,dv)*dx + dot(dlamb,v_)*dx
```

We have used a general implementation using a sum over the different phases for the functional `F`. We then used the `lhs` and `rhs` functions to respectively extract the corresponding bilinear a and linear L forms.

3.1.4 Resolution

The resolution of the auxiliary problem is performed for elementary load cases consisting of uniaxial strain and pure shear sollicitations by assigning unit values of the correspnonding E_{ij} components. For each load case, the average stress $\boldsymbol{\Sigma}$ is computed components by components and the macroscopic stiffness components $\mathbb{C}^{hom}$ are then printed.

[5]:
```
def macro_strain(i):
    """returns the macroscopic strain for the 3 elementary load cases"""
    Eps_Voigt = np.zeros((3,))
    Eps_Voigt[i] = 1
    return np.array([[Eps_Voigt[0], Eps_Voigt[2]/2.],
                     [Eps_Voigt[2]/2., Eps_Voigt[1]]])
def stress2Voigt(s):
    return as_vector([s[0,0], s[1,1], s[0,1]])

Chom = np.zeros((3, 3))
for (j, case) in enumerate(["Exx", "Eyy", "Exy"]):
    print("Solving {} case...".format(case))
    Eps.assign(Constant(macro_strain(j)))
    solve(a == L, w, [], solver_parameters={"linear_solver": "cg"})
    (v, lamb) = split(w)
    Sigma = np.zeros((3,))
    for k in range(3):
        Sigma[k] = assemble(sum([stress2Voigt(sigma(v, i, Eps))[k]*dx(i) for i in␣
↪range(nphases)]))/vol
    Chom[j, :] = Sigma

print(np.array_str(Chom, precision=2))
```

```
Solving Exx case...
Solving Eyy case...
Solving Exy case...
[[ 6.56e+04  1.74e+04 -2.10e-02]
 [ 1.74e+04  6.56e+04 -3.96e-02]
 [-2.45e-02 -4.21e-02  2.41e+04]]
```

It can first be verified that the obtained macroscopic stiffness is indeed symmetric and that the corresponding behaviour is quasi-isotropic (up to the finite element discretization error). Indeed, if $\lambda^{hom} = \mathbb{C}_{xxyy}$ and $\mu^{hom} = \mathbb{C}_{xyxy}$ we have that $\mathbb{C}_{xxxx} \approx \mathbb{C}_{yyyy} \approx \mathbb{C}_{xxyy} + 2\mathbb{C}_{xyxy} = \lambda^{hom} + 2\mu^{hom}$.

> **Note:** The macroscopic stiffness is not exactly symmetric because we computed it from the average stress which is not stricly verifying local equilibrium on the unit cell due to the FE discretization. A truly symmetric version can be obtained from the computation of the bilinear form for a pair of solutions to the elementary load cases.

[6]:
```
lmbda_hom = Chom[0, 1]
mu_hom = Chom[2, 2]
print(Chom[0, 0], lmbda_hom + 2*mu_hom)
```

```
65570.19577047735 65570.25538998275
```

We thus deduce that $E^{hom} = \mu^{hom}\dfrac{3\lambda^{hom} + 2\mu^{hom}}{\lambda^{hom} + \mu^{hom}}$ and $\nu^{hom} = \dfrac{\lambda^{hom}}{2(\lambda^{hom} + \mu^{hom})}$ that is:

[7]:
```
E_hom = mu_hom*(3*lmbda_hom + 2*mu_hom)/(lmbda_hom + mu_hom)
nu_hom = lmbda_hom/(lmbda_hom + mu_hom)/2
print("Apparent Young modulus:", E_hom)
print("Apparent Poisson ratio:", nu_hom)
```

```
Apparent Young modulus: 58239.72435207217
Apparent Poisson ratio: 0.2101253163282564
```

[8]:
```
# plotting deformed unit cell with total displacement u = Eps*y + v
y = SpatialCoordinate(mesh)
plt.figure()
p = plot(0.5*(dot(Eps, y)+v), mode="displacement", title=case)
plt.colorbar(p)
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

CHAPTER 4

Eigenvalue problems in solid mechanics

Contents:

The corresponding files can be obtained from:

- Jupyter Notebook: `buckling_3d_solid.ipynb`
- Python script: `buckling_3d_solid.py`

4.1 Linear Buckling Analysis of a 3D solid

This demo has been written in collaboration with Eder Medina (Harvard University).

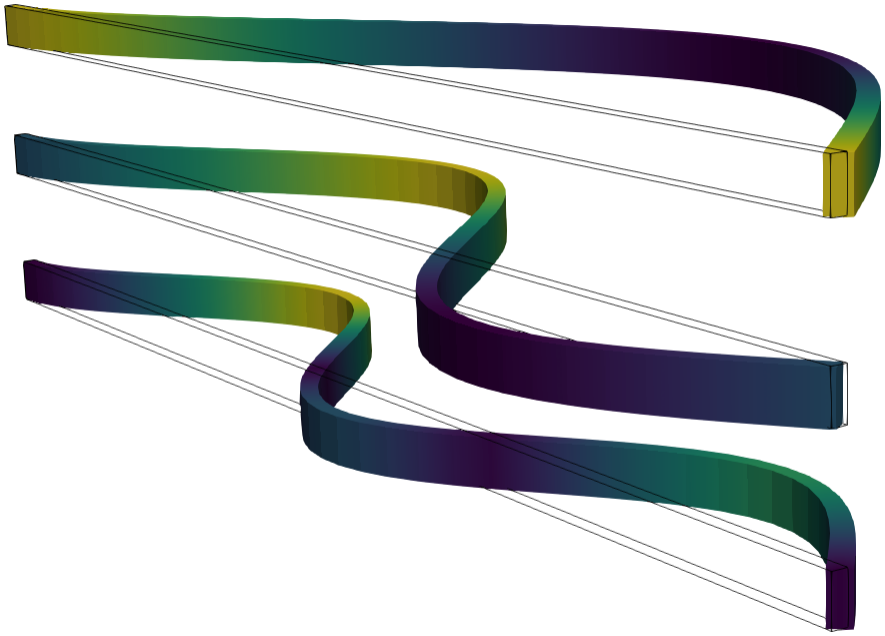
In this numerical tour, we will demonstrate how to compute the buckling modes of a three-dimensional elastic solid under a conservative loading. The critical buckling loads and buckling modes are computed by solving a generalized non-hermitian eigenvalue problem using the `SLEPcEigensolver`. This example is closely related to the Modal analysis of an elastic structure and the *Eulerian buckling of a beam* demos. For more details on the theoretical formulation, the reader can refer to *[NGU00]*.

4.1.1 Geometrically Nonlinear Elasticity

To formulate the buckling problem, we must consider a general geometrically nonlinear framework. More precisely, we will consider the case of small-strain/large displacements that are expected in typical buckling analysis.

We consider the Green-Lagrange strain tensor defined as:

$$e_{ij} = \frac{1}{2}(u_{i,j} + u_{j,i}) + \frac{1}{2}u_{k,i}u_{k,j}$$



Note that this can be rewritten in tensor format as

$$= () + \frac{1}{2}(,)$$

where we isolate the linear $()$ and nonlinear quadratic part $(,) = \nabla\nabla$.

Since we restrict to a small strain assumption, we adopt a standard linear Hookean behaviour for the material material model which can be described by the following quadratic elastic energy density:

$$\psi() = \frac{1}{2} : \mathbb{C} :$$

where $\mathbb{C}$ is the elastic moduli tensor.

We seek to find the underlying equilibrium displacement field as a function of a scalar loading parameter λ under a set of prescribed external surface forces (λ).

4.1.2 Equilibrium and stability conditions

Let us consider the total potential energy functional:

$$\mathcal{E}_{\text{pot}}(;\lambda) = \int_\Omega \psi(())d\Omega - \int_{\Gamma_N} (\lambda) \cdot dS$$

The equilibrium equations are obtained from the first variation which must vanish:

$$\partial_u \mathcal{E}_{\text{pot}}[\delta] = \int_\Omega () : \mathbb{C} : \delta[\delta]d\Omega - \int_{\Gamma_N} (\lambda) \cdot \delta dS$$

where δ is any kinematically admissible perturbation direction and where:

$$\delta[\delta] = (\delta) + \nabla\nabla\delta = (\delta) + (,\delta)$$

The stability conditions of an equilibrium are obtained from the Lejeune-Dirichlet theorem stating that the energy second variation must be positive. Here the second variation bilinear form is given by:

$$\partial_{uu}\mathcal{E}_{\text{pot}}[\delta,\delta] = \int_\Omega (\delta[\delta] : \mathbb{C} : \delta[\delta] + : \mathbb{C} : \delta^2[\delta,\delta])d\Omega$$

where:

$$\delta^2[\delta,\delta] = \delta(,\delta)[\delta] = \nabla\delta\nabla\delta = (\delta,\delta)$$

4.1.3 Bifurcation point

A given equilibrium solution $\lambda^c, ^c = (\lambda^c)$ is a bifurcation point if the second variation vanishes for some δ. λ^c defines the bifurcation load and δ the bifurcation mode. Hence, we have:

$$\int_\Omega (((\delta) + (^c,\delta)) : \mathbb{C} : ((\delta) + (^c,\delta)) + (^c) : \mathbb{C} : (\delta,\delta))d\Omega = 0$$

4.1.4 Linear buckling analysis

When performing a linear buckling analysis, we look for bifurcation points on the fundamental equilibrium branch obtained by a small displacement assumption. As a result, in the above bifurcation equation the quadratic contribution in the first term can be neglected and the Green-Lagrange strain $(^c)$ can be replaced by the linearized strain $(^c)$. Besides, in the small-displacement assumption, if the loading depends linearly on the load parameter λ, the small displacement solution also depends linearly on it. We therefore have $^c = \lambda_0^c$ and the bifurcation condition becomes:

$$\int_\Omega ((\delta) : \mathbb{C} : (\delta) + \lambda^c(_0) : \mathbb{C} : (\delta,\delta))d\Omega = 0$$

After, introducing the inital pre-stress $_0 = \mathbb{C} : (_0)$, the last term can be rewritten as $_0 : (\delta,\delta) = (\nabla\delta)_0(\nabla\delta)$ so that :

$$\int_\Omega ((\delta) : \mathbb{C} : (\delta) + \lambda^c(\nabla\delta)_0(\nabla\delta))d\Omega = 0$$

We recognize here a linear eigenvalue problem where the first term corresponds to the classical linear elasticity bilinear form and the second term is an additional contribution depending on the pre-stressed state $_0$. Transforming this variational problem into matrix notations yields:

$$[\mathbf{K}] + \lambda[\mathbf{K}_G(_0)] = 0$$

where $[\mathbf{K}]$ is the classical linear elastic stiffness matrix and $[\mathbf{K}_G(_0)]$ the so-called *geometrical stiffness* matrix.

4.1.5 FEniCS implementation

Defining the Geometry

Here we will consider a fully three dimensional beam-like structure with length $L = 1$ and of rectangular cross-section with height $h = 0.01$, and width $b = 0.03$.

[1]:
```
from dolfin import *
import matplotlib.pyplot as plt
import numpy as np

L, b, h = 1, 0.01, 0.03
Nx, Ny, Nz = 51, 5, 5

mesh = BoxMesh(Point(0, 0, 0), Point(L, b, h), Nx, Ny, Nz)
```

Solving for the pre-stressed state $_0$

We first need to compute the pre-stressed linear state $_0$. It is obtained by solving a simple linear elasticity problem. The material is assumed here isotropic and we consider a pre-stressed state obtained from the application of a unit compression applied at the beam right end in the X direction while the Y and Z displacement are fixed, mimicking a simple support condition. The beam is fully clamped on its left end.

[2]:
```
E , nu = 1e3, 0.

mu = Constant(E/2*(1+nu))
lmbda = Constant(E*nu/(1+nu)/(1-2*nu))

def eps(v):
    return sym(grad(v))
def sigma(v):
    return lmbda*tr(eps(v))*Identity(v.geometric_dimension())+2*mu*eps(v)

# Compute the linearized unit preload
N0 = 1
T = Constant((-N0, 0, 0))
V = VectorFunctionSpace(mesh, 'Lagrange', degree = 2)
v = TestFunction(V)
du = TrialFunction(V)

# Two different ways to define how to impose boundary conditions
def left(x,on_boundary):
    return near(x[0], 0.)

def right(x, on_boundary):
    return near(x[0], L)

boundary_markers = MeshFunction('size_t', mesh, mesh.topology().dim()-1)
ds = Measure('ds', domain=mesh, subdomain_data=boundary_markers)
AutoSubDomain(right).mark(boundary_markers, 1)

# Clamped Boundary and Simply Supported Boundary Conditions
bcs = [DirichletBC(V, Constant((0, 0, 0)), left),
       DirichletBC(V.sub(1), Constant(0), right),
       DirichletBC(V.sub(2), Constant(0), right)]

# Linear Elasticity Bilinear Form
a = inner(sigma(du), eps(v))*dx

# External loading on the right end
l = dot(T, v)*ds(1)
```

We will reuse our stiffness matrix in the linear buckling computation and hence we supply a `PETScMatrix` and `PETScVector` to the system assembler. We then use the `solve` function on these objects and specify the linear solver method.

[3]:
```
K = PETScMatrix()
f = PETScVector()
assemble_system(a, l, bcs, A_tensor=K, b_tensor=f)

u = Function(V,name = "Displacement")
solve(K, u.vector(), f, "mumps")
```

(continues on next page)

(continued from previous page)

```
# Output the trivial solution
ffile = XDMFFile("output/solution.xdmf")
ffile.parameters["functions_share_mesh"] = True
ffile.parameters["flush_output"] = True
ffile.write(u, 0)
```

Forming the geometric stiffness matrix and the buckling eigenvalue problem

From the previous solution, the prestressed state $_0$ entering the geometric stiffness matrix expression is simply given by `sigma(u)`. After having formed the geometric stiffness matrix, we will call the `SLEPcEigenSolver` for solving a generalized eigenvalue problem $\mathbf{Ax} = \lambda\mathbf{Bx}$ with here $\mathbf{A} = \mathbf{K}$ and $\mathbf{B} = -\mathbf{K}_G$. We therefore include directly the negative sign in the definition of the geometric stiffness form:

[4]:
```
kgform = -inner(sigma(u), grad(du).T*grad(v))*dx

KG = PETScMatrix()
assemble(kgform, KG)

# Zero out the rows to enforce boundary condition
for bc in bcs:
    bc.zero(KG)
```

The buckling points of interest are the smallest critical loads, i.e. the points corresponding to the smallest λ^c.

We now pass some parameters to the SLEPc solver. In particular, we peform a shift-invert transform as discussed in the Eulerian buckling of an elastic beam tour.

[5]:
```
# What Solver Configurations exist?
eigensolver = SLEPcEigenSolver(K, KG)
print(eigensolver.parameters.str(True))
```

```
<Parameter set "slepc_eigenvalue_solver" containing 8 parameter(s) and parameter␣
↪set(s)>

  slepc_eigenvalue_solver  |    type    value    range  access  change
  -------------------------------------------------------------------
  maximum_iterations       |     int  <unset>  Not set       0       0
  problem_type             |  string  <unset>  Not set       0       0
  solver                   |  string  <unset>  Not set       0       0
  spectral_shift           |  double  <unset>  Not set       0       0
  spectral_transform       |  string  <unset>  Not set       0       0
  spectrum                 |  string  <unset>  Not set       0       0
  tolerance                |  double  <unset>  Not set       0       0
  verbose                  |    bool  <unset>  Not set       0       0
```

[6]:
```
eigensolver.parameters['problem_type'] = 'gen_hermitian'
eigensolver.parameters['solver'] = "krylov-schur"
eigensolver.parameters['tolerance'] = 1.e-12
eigensolver.parameters['spectral_transform']  = 'shift-and-invert'
eigensolver.parameters['spectral_shift'] = 1e-2

# Request the smallest 3 eigenvalues
N_eig = 3
print("Computing {} first eigenvalues...".format(N_eig))
eigensolver.solve(N_eig)
```

(continues on next page)

(continued from previous page)

```
print("Number of converged eigenvalues:", eigensolver.get_number_converged())

eigenmode = Function(V)

# Analytical beam theory solution
# F_cr,n = alpha_n^2*EI/L^2
# where alpha_n are solutions to tan(alpha) = alpha
alpha = np.array([1.4303, 2.4509, 3.4709] + [(n+1/2) for n in range(4, 10)])*pi
I = b**3*h/12
S = b*h

for i in range(eigensolver.get_number_converged()):
    # Extract eigenpair
    r, c, rx, cx = eigensolver.get_eigenpair(i)

    critical_load_an = alpha[i]**2*float(E*I/N0/S)/L**2
    print("Critical load factor {}: {:.5f} FE | {:.5f} Beam".format(i+1, r, critical_
↪load_an))

    # Initialize function and assign eigenvector
    eigenmode.vector()[:] = rx
    eigenmode.rename("Eigenmode {}".format(i+1), "")

    ffile.write(eigenmode, 0)
```

```
Computing 3 first eigenvalues...
Number of converged eigenvalues: 3
Critical load factor 1: 0.16821 FE | 0.16826 Beam
Critical load factor 2: 0.49691 FE | 0.49405 Beam
Critical load factor 3: 0.98918 FE | 0.99084 Beam
```

Note that the analysis above is not limited to problems subject to Neumann boundary conditions. One can equivalenty compute the prestress resulting from a displacement control simulation and the critical buckling load will correspond to the critical buckling displacement of the structure.

4.1.6 References

[NGU00]: Nguyen, Q. S. (2000). *Stability and nonlinear solid mechanics*. Wiley.

CHAPTER 5

Nonlinear problems in solid mechanics

Contents:

The corresponding files can be obtained from:

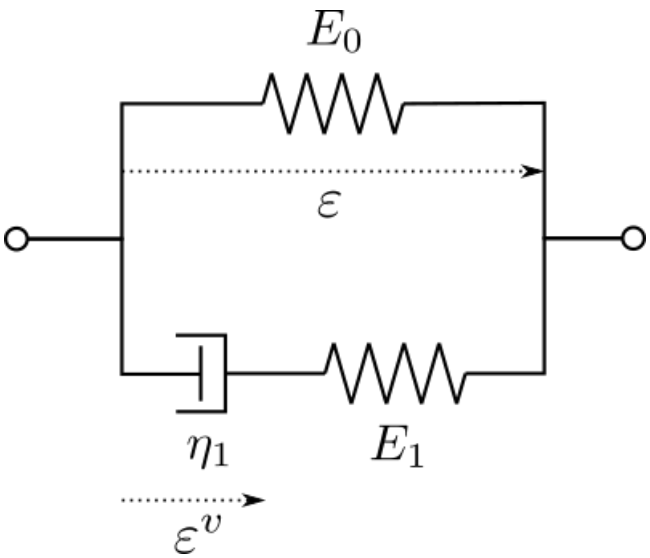
- Jupyter Notebook: `linear_viscoelasticity.ipynb`
- Python script: `linear_viscoelasticity.py`

5.1 Linear viscoelasticity

In this numerical tour, we will explore the formulation of simple linear viscoelastic behaviours such as Maxwell, Kelvin-Voigt or Standard Linear Solid models. The formulation can also be quite easily extended to a generalized Maxwell model.

5.1.1 Constitutive evolution equations

1D rheological formulation



We consider a 1D Linear Standard Solid model consisting of a spring of stiffness E_0 in parallel to a Maxwell arm (spring of stiffness E_1 in serie with a dashpot of viscosity η_1). The uniaxial stress for this rheological model can be decomposed as the sum of a reversible and an irreversible stress:

$$\sigma = E_0\varepsilon + E_1(\varepsilon - \varepsilon^v)$$

whereas the evolution equation for the viscous internal strain is given by:

$$\dot{\varepsilon}^v = \frac{E_1}{\eta_1}(\varepsilon - \varepsilon^v)$$

The extension to a generalized Maxwell model with N internal strains is given by:

$$\sigma = E_0\varepsilon + \sum_{i=1}^{N} E_i(\varepsilon - \varepsilon^{v,i}) \tag{5.1}$$

$$\dot{\varepsilon}^{v,i} = \frac{E_i}{\eta_i}(\varepsilon - \varepsilon^{v,i}) \quad \forall i = 1,..N \tag{5.2}$$

3D generalization

For the 3D case, isotropic viscoelasticity is characterized by two elastic moduli (resp. two viscosities, or equivalently two relaxation times) for each spring (resp. dashpot) element of the 1D model. Here, we will restrict to a simpler case in which one modulus is common to all elements (similar Poisson ratio for all elements), that is:

$$\boldsymbol{\sigma} = E_0 : \boldsymbol{\varepsilon} + E_1 : (\boldsymbol{\varepsilon} - \boldsymbol{\varepsilon}^v) \tag{5.3}$$

$$\dot{\boldsymbol{\varepsilon}}^v = \frac{E_1}{\eta_1} : (\boldsymbol{\varepsilon} - \boldsymbol{\varepsilon}^v) \tag{5.4}$$

where $= \dfrac{\nu}{(1+\nu)(1-2\nu)}\mathbf{1}\otimes\mathbf{1} + \dfrac{1}{1+\nu}\mathbb{I}$ with $\mathbf{1}$ and $\mathbb{I}$ being respectively the 2nd and 4th order identity tensors and ν being the Poisson ratio.

5.1.2 Problem position

We consider here a 2D rectangular domain of dimensions $L \times H$. The constitutive relations are written in plane stress conditions, the unitary stiffness tensor is therefore $= \frac{\nu}{1-\nu^2}\mathbf{1}\otimes\mathbf{1} + \frac{1}{1+\nu}\mathbb{I}$. The boundary conditions consist of symmetry planes on $x = 0$ and $y = 0$ and smooth contact with a plane with imposed vertical displacement on $y = H$ or imposed vertical uniform traction depending on the load case. The solution will therefore be homogeneous in the sample.

[18]:
```
from dolfin import *
from ufl import replace
import numpy as np
import matplotlib.pyplot as plt
%matplotlib notebook

L, H = 0.1, 0.2
mesh = RectangleMesh(Point(0., 0.), Point(L, H), 5, 10)

E0 = Constant(70e3)
E1 = Constant(20e3)
eta1 = Constant(1e3)
nu = Constant(0.)
dt = Constant(0.) # time increment
sigc = 100. # imposed creep stress
epsr = 1e-3 # imposed relaxation strain

def left(x, on_boundary):
    return near(x[0], 0.) and on_boundary
def bottom(x, on_boundary):
    return near(x[1], 0.) and on_boundary
class Top(SubDomain):
    def inside(self, x, on_boundary):
        return near(x[1], H) and on_boundary

facets = MeshFunction("size_t", mesh, 1)
facets.set_all(0)
Top().mark(facets, 1)
ds = Measure("ds", subdomain_data=facets)
```

5.1.3 Time discretization and formulation of the variational problem

Here we will discuss the time discretization of the viscoelastic constitutive equations and the formulation of the global problem using FEniCS.

A first approach

A first approach for formulating the time-discretized equations consists in approximating the viscous strain evolution equation using an implicit backward-Euler scheme at a given time t_{n+1}:

$$\frac{\varepsilon^{v,n+1} - \varepsilon^{v,n}}{\Delta t} \approx \dot{\varepsilon}^{v,n+1} = \frac{E_1}{\eta_1}(\varepsilon^{n+1} - \varepsilon^{v,n+1})$$

which can be rewritten as:

$$\varepsilon^{v,n+1} = \left(1 + \frac{\Delta t E_1}{\eta_1}\right)^{-1} \left(\varepsilon^{v,n} + \frac{\Delta t E_1}{\eta_1}\varepsilon^{n+1}\right)$$

Note: In the 3D case, the previous expression is more involved since one must invert the following 4-th order tensor: $\mathbb{I} + \frac{\Delta t E_1}{\eta_1}$

Introducing $\tau = \eta_1/E_1$ and plugging the previous relation into the stress-strain relation, one obtains:

$$\sigma_{n+1} = (E_0 + E_1)\varepsilon^{n+1} - E_1 \frac{\Delta t/\tau}{1 + \Delta t/\tau}\varepsilon^{n+1} - E_1\varepsilon^{v,n} \tag{5.7}$$

$$= \left(E_0 + \frac{E_1}{1 + \Delta t/\tau}\right)\varepsilon^{n+1} - \frac{E_1}{1 + \Delta t/\tau}\varepsilon^{v,n} \tag{5.8}$$

A possible solution for solving these simple linear viscoelastic behaviours would then be to formulate exactly the problem associated with the previous stress-strain relation, taking into account the modified elasticity tensor depending on the time step Δt and the value of the previous viscous strain. As a result, one would have a pure displacement problem. After solving for u at a given time step, the new viscous strain would have to be updated using the previous relations.

A mixed approach

One problematic aspect of the previous approach is that it requires inverting the modified elasticity tensor which poses no problem in the present case but may not be possible for more general non-linear behaviours for instance. We hence propose a more general approach which amounts to solving both the displacement-problem and the viscous strain update in a monolothic way. The formulation can be derived from an incremental variational principle by defining the following incremental potential:

$$\mathcal{E} = \int_\Omega w(\boldsymbol{\varepsilon}, \dot{\boldsymbol{\varepsilon}}^v) d\Omega + \Delta t \int_\Omega \phi(\dot{\boldsymbol{\varepsilon}}^v) d\Omega - W_{ext}(\boldsymbol{u})$$

where $W_{ext}(\boldsymbol{u})$ is the work of external forces and the strain energy density $w(\boldsymbol{\varepsilon}, \boldsymbol{\varepsilon}^v)$ and the dissipation potential $\phi(\dot{\boldsymbol{\varepsilon}}^v)$ of the Linear Standard Solid model are respectively given by:

$$w(\boldsymbol{\varepsilon}, \boldsymbol{\varepsilon}^v) = \frac{1}{2}\boldsymbol{\varepsilon} : (E_0) : \boldsymbol{\varepsilon} + \frac{1}{2}(\boldsymbol{\varepsilon} - \boldsymbol{\varepsilon}^v) : (E_1) : (\boldsymbol{\varepsilon} - \boldsymbol{\varepsilon}^v)$$

$$\phi(\dot{\boldsymbol{\varepsilon}}^v) = \frac{1}{2}\eta_1 \dot{\boldsymbol{\varepsilon}}^v : \dot{\boldsymbol{\varepsilon}}^v$$

Introducing the backward-Euler approximation

$$\dot{\boldsymbol{\varepsilon}}^v \approx \frac{\boldsymbol{\varepsilon}^{v,n+1} - \boldsymbol{\varepsilon}^{v,n}}{\Delta t}$$

into the dissipation potential, the new total and viscous strain variables are obtained as the solution to the following minimization problem:

$$\min_{\boldsymbol{\varepsilon}^{n+1},\,\boldsymbol{\varepsilon}^{v,n+1}} \mathcal{E}_{n+1} = \int_\Omega w(\boldsymbol{\varepsilon}^{n+1}, \boldsymbol{\varepsilon}^{v,n+1}) d\Omega + \Delta t \int_\Omega \phi\left(\frac{\boldsymbol{\varepsilon}^{v,n+1} - \boldsymbol{\varepsilon}^{v,n}}{\Delta t}\right) d\Omega$$

which depends on the knowledge of the previous viscous strain $\boldsymbol{\varepsilon}^{v,n}$.

It can be shown that the optimality conditions of the previous minimization problem correspond exactly to the relations of the first approach.

In the FEniCS implementation, we use a CG1 interpolation for the displacement and a DG0 interpolation for the viscoelastic strain. We then define appropriate functions and form the previous incremental potential. We use the `derivative` function for automatic differentiation. Since the considered problem is linear, we also extract the corresponding bilinear and linear parts of the potential derivative.

[19]:
```
Ve = VectorElement("CG", mesh.ufl_cell(), 1)
Qe = TensorElement("DG", mesh.ufl_cell(), 0)
W = FunctionSpace(mesh, MixedElement([Ve, Qe]))
w = Function(W, name="Variables at current step")
(u, epsv) = split(w)
w_old = Function(W, name="Variables at previous step")
(u_old, epsv_old) = split(w_old)
w_ = TestFunction(W)
(u_, epsv_) = split(w_)
dw = TrialFunction(W)

def eps(u):
    return sym(grad(u))
def dotC(e):
    return nu/(1+nu)/(1-nu)*tr(e)*Identity(2) + 1/(1+nu)*e
def sigma(u, epsv):
    return E0*dotC(eps(u)) + E1*dotC(eps(u) - epsv)
def strain_energy(u, epsv):
    e = eps(u)
    return 0.5*(E0*inner(e,dotC(e)) + E1*inner(e-epsv, dotC(e-epsv)))
def dissipation_potential(depsv):
    return 0.5*eta1*inner(depsv, depsv)

Traction = Constant(0.)
incremental_potential = strain_energy(u, epsv)*dx \
                        + dt*dissipation_potential((epsv-epsv_old)/dt)*dx \
                        - Traction*u[1]*ds(1)
F = derivative(incremental_potential, w, w_)
form = replace(F, {w: dw})
```

5.1.4 Solutions of elementary tests

In the following, we will consider three elementary tests of viscoelastic behaviour, namely a relaxation, creep and recovery test. We implement a function for setting up the boundary conditions and loading parameters for these three different tests and perform the time integration. We keep track of the average vertical stress and strain states (the problem is such that all fields are uniform in the sample) and compare the evolutions with analytical solutions given below. Parameters have been taken as $E_0 = 70$ GPa, $E_1 = 20$ GPa, $\eta_1 = 1$ GPa.s.

[20]:
```
dimp = Constant(H*epsr) # imposed vertical displacement instead
bcs = [DirichletBC(W.sub(0).sub(0), Constant(0), left),
       DirichletBC(W.sub(0).sub(1), Constant(0), bottom),
       DirichletBC(W.sub(0).sub(1), dimp, facets, 1)]

def viscoelastic_test(case, Nsteps=50, Tend=1.):
    # Solution fields are initialized to zero
    w.interpolate(Constant((0.,)*6))

    # Define proper loading and BCs
    if case in ["creep", "recovery"]: # imposed traction on top
        Traction.assign(Constant(sigc))
        bc = bcs[:2] # remove the last boundary conditions from bcs
        t0 = Tend/2. # traction goes to zero at t0 for recovery test
    elif case == "relaxation":
        Traction.assign(Constant(0.)) # no traction on top
        bc = bcs

    # Time-stepping loop
    time = np.linspace(0, Tend, Nsteps+1)
    Sigyy = np.zeros((Nsteps+1, ))
    Epsyy = np.zeros((Nsteps+1, 2))
    for (i, dti) in enumerate(np.diff(time)):
        if i>0 and i % (Nsteps/5) == 0:
            print("Increment {}/{}".format(i, Nsteps))
        dt.assign(dti)
        if case == "recovery" and time[i+1] > t0:
            Traction.assign(Constant(0.))
        w_old.assign(w)
        solve(lhs(form) == rhs(form), w, bc)
        # get average stress/strain
        Sigyy[i+1] = assemble(sigma(u, epsv)[1, 1]*dx)/L/H
        Epsyy[i+1, 0] = assemble(eps(u)[1, 1]*dx)/L/H
        Epsyy[i+1, 1] = assemble(epsv[1, 1]*dx)/L/H

    # Define analytical solutions
    if case == "creep":
        if float(E0) == 0.:
            eps_an = sigc*(1./float(E1)+time/float(eta1))
        else:
            Estar = float(E0*E1/(E0+E1))
            tau = float(eta1)/Estar
            eps_an = sigc/float(E0)*(1-float(Estar/E0)*np.exp(-time/tau))
        sig_an = sigc + 0*time
    elif case == "relaxation":
        if float(E1) == 0.:
            sig_an = epsr*float(E0) + 0*time
        else:
            tau = float(eta1/E1)
```

(continues on next page)

(continued from previous page)

```
            sig_an = epsr*(float(E0) + float(E1)*np.exp(-time/tau))
        eps_an = epsr + 0*time

    elif case == "recovery":
        Estar = float(E0*E1/(E0+E1))
        tau = float(eta1)/Estar
        eps_an = sigc/float(E0)*(1-float(E1/(E0+E1))*np.exp(-time/tau))
        sig_an = sigc + 0*time
        time2 = time[time > t0]
        sig_an[time > t0] = 0.
        eps_an[time > t0] = sigc*float(E1/E0/(E0+E1))*(np.exp(-(time2-t0)/tau)
                                                       - np.exp(-time2/tau))

    # Plot strain evolution
    plt.figure()
    plt.plot(time, 100*eps_an, label="analytical solution")
    plt.plot(time, 100*Epsyy[:, 0], '.', label="FE solution")
    plt.plot(time, 100*Epsyy[:, 1], '--', linewidth=1, label="viscous strain")
    plt.ylim(0, 1.2*max(Epsyy[:, 0])*100)
    plt.xlabel("Time")
    plt.ylabel("Vertical strain [\%]")
    plt.title(case.capitalize() + " test")
    plt.legend()
    plt.show()

    # Plot stress evolution
    plt.figure()
    plt.plot(time, sig_an, label="analytical solution")
    plt.plot(time, Sigyy, '.', label="FE solution")
    plt.ylim(0, 1.2*max(Sigyy))
    plt.ylabel("Vertical stress")
    plt.xlabel("Time")
    plt.title(case.capitalize() + " test")
    plt.legend()
    plt.show()
```

Relaxation test

First we consider a relaxation test in which the top surface is instantaneously displaced and maintained fixed, generating a uniform and constant uniaxial strain state $\varepsilon_{yy} = \varepsilon_r$. Resolving the 1D evolution equation for this test gives:

$$\sigma_{yy}(t) = E_0\varepsilon_r + E_1\varepsilon_r \exp(-t/\tau) \quad \text{with } \tau = \frac{\eta_1}{E_1}$$

The analytical solution is well reproduced by the FE solution as seen below. The instantaneous stress being $\sigma_{yy}(t = 0^+) = (E_0 + E_1)\varepsilon_r$ whereas the long term stress is being given by $\sigma_{yy}(t = \infty) = E_0\varepsilon_r$.

[21]:
```
viscoelastic_test("relaxation")
```

```
Increment 10/50
Increment 20/50
Increment 30/50
Increment 40/50
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Creep test

We now consider a creep test in which a uniform vertical traction of intensity σ_c is suddenly applied and maintained constant. The integration of the evolution equation gives in this case:

$$\varepsilon_{yy}(t) = \frac{\sigma_c}{E_0}\left(1 - \frac{E_1}{E_0 + E_1}\exp(-t/\tau)\right) \quad \text{with } \tau = \frac{\eta_1(E_0 + E_1)}{E_0 E_1}$$

Again, the solution is well reproduced. The instantaneous strain is $\varepsilon_{yy}(t = 0^+) = \dfrac{\sigma_c}{E_0 + E_1}$ whereas the long term strain is being given by $\varepsilon_{yy}(t = \infty) = \dfrac{\sigma_c}{E_0}$.

[22]:
```
viscoelastic_test("creep")
```

```
Increment 10/50
Increment 20/50
Increment 30/50
Increment 40/50
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Recovery test

Finally, we consider again the same creep test as before up to $t = t_0 = 0.5$, after which the imposed traction is suddenly removed. The solution is the same as before during the first stage whereas during the second stage it is given by:

$$\varepsilon_{yy}(t) = \frac{\sigma_c E_1}{E_0(E_0 + E_1)}\left(\exp(-(t - t_0)/\tau) - \exp(-t/\tau)\right) \quad \text{with } \tau = \frac{\eta_1(E_0 + E_1)}{E_0 E_1}$$

[23]:
```
viscoelastic_test("recovery")
```

```
Increment 10/50
Increment 20/50
Increment 30/50
Increment 40/50
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

5.1.5 Relaxation and creep tests for a Maxwell model

We give here the solutions for a Maxwell model which is obtained from the degenerate case $E_0 = 0$. We recover that the strain evolves linearly with time for the creep test.

[24]:
```
E0.assign(Constant(0.))
viscoelastic_test("relaxation")
viscoelastic_test("creep")
```

```
Increment 10/50
Increment 20/50
Increment 30/50
Increment 40/50
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
Increment 10/50
Increment 20/50
Increment 30/50
Increment 40/50
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

5.1.6 Relaxation and creep tests for a Kelvin-Voigt model

We give here the solutions for a Kelvin-Voigt model which is obtained from the degenerate case $E_1 = \infty$. We recover that the stress is constant for the relaxation test.

[25]:
```
E0.assign(Constant(70e3))
E1.assign(Constant(1e10))
viscoelastic_test("relaxation")
viscoelastic_test("creep")
```

```
Increment 10/50
Increment 20/50
Increment 30/50
Increment 40/50
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
Increment 10/50
Increment 20/50
Increment 30/50
Increment 40/50
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

The corresponding files can be obtained from:

- Jupyter Notebook: `penalty.ipynb`
- Python script: `penalty.py`

5.2 Hertzian contact with a rigid indenter using a penalty approach

In this numerical tour, we explore the formulation of frictionless contact between a rigid surface (the indenter) and an elastic domain, representing an infinite half-space for the present case. Contact will be solved using a penalty formulation, that is small interpenetration between the solid and the indenter will be authorized. Prerequisites for this tour are the formulation of an linear elasticity problem and the resolution of a general nonlinear variational problem. See for instance:

- https://comet-fenics.readthedocs.io/en/latest/demo/elasticity/2D_elasticity.py.html
- https://fenicsproject.org/olddocs/dolfin/latest/python/demos/hyperelasticity/demo_hyperelasticity.py.html

The elastic (isotropic E, ν) solid will be represented by a 3D cubic domain of unit dimension and contact will take place on its top surface $z = 0$, centered at $x = y = 0$. Symmetry conditions will be applied on the $x = 0$ and $y = 0$ surfaces whereas the bottom surface $z = -1$ will be fully fixed. If contact appears on a small region of extent $a \ll 1$, the problem can then be considered to be a good approximation of contact on a semi-infinite domain.

The rigid indenter will not be explictly modeled (in particular it will not be meshed) but instead its distance with respect to the solid top surface will be given as an `Expression`. We will also consider that the indenter radius R is sufficiently large with respect to the contact region characteristic size a so that the spherical surface can be approximated by a parabola. In this case, the distance between such an indenter and the top surface can be written as:

$$h(x, y) = h_0 + \frac{1}{2R}(x^2 + y^2)$$

where h_0 is the initial gap between both surfaces at $x = y = 0$. Obviously, if $h_0 > 0$ there is no contact between both surfaces, contact appears only if $h_0 = -d < 0$ where d will be the indenter depth inside the surface. This classical problem admits the following known analytical solution *[1,2]*:

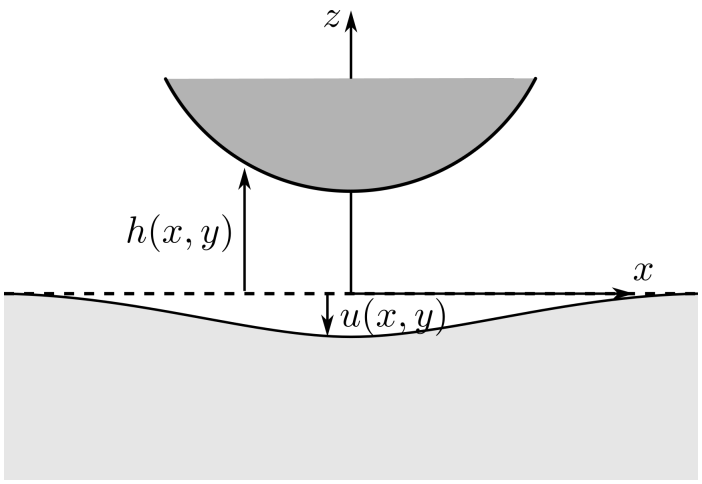


Fig. 1: Hertzian contact problem of a rigid spherical indenter on a semi-infinite domain

- the contact area is of circular shape and radius $a = \sqrt{Rd}$
- the force exerted by the indenter onto the surface is $F = \frac{4}{3}\frac{E}{1-\nu^2}ad$
- the pressure distribution on the contact region is given by $p(r) = p_0\sqrt{1-(r/a)^2}$ where $p_0 = 3F/(2\pi a^2)$ is the maximal pressure

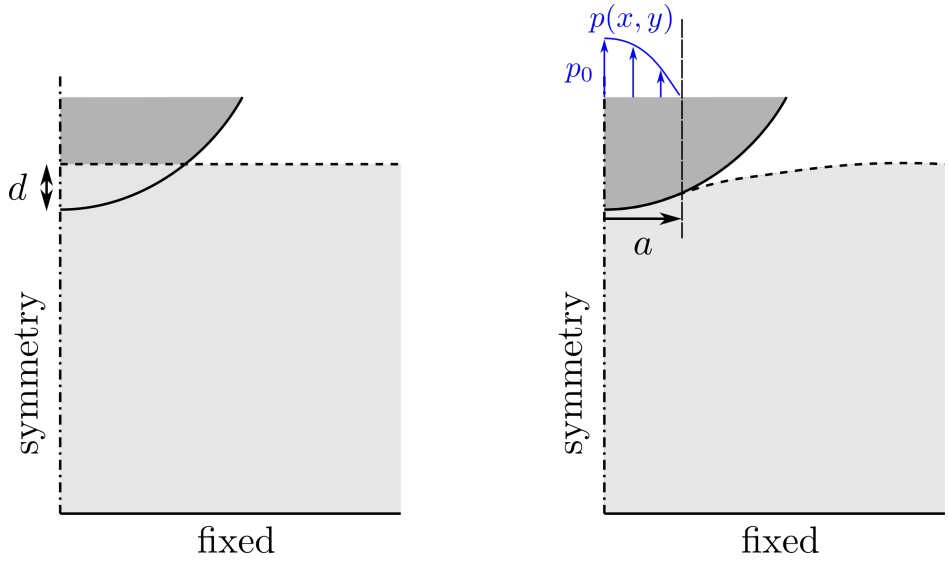


Fig. 2: Indent at depth d (left) and contact solution (right)

5.2.1 Contact problem formulation and penalty approach

The unilateral contact (Signorini) condition on the top surface Γ writes as:

$$g \geq 0, p \geq 0, g \cdot p = 0 \text{ on } \Gamma$$

where $g = h - u$ is the gap between the obstacle surface and the solid surface and $p = -\sigma_{zz}$ is the pressure. One of the most simple way to solve approximately this contact condition consists in replacing the previous complementary

conditions by the following penalized condition:

$$p = k\langle -g \rangle_+ = k\langle u - h \rangle_+$$

where $\langle x \rangle_+ = (|x| + x)/2$ is the positive part (Mackauley bracket) and k is a large penalizing stiffness coefficient. With the previous relation, the pressure will be positive but a small negative gap will be authorized.

5.2.2 Implementation

First, a unit cubic mesh is defined and some mapping is applied on the mesh nodes in order to refine the element size near the contact area (around $x = y = z = 0$) and change to a $[0; 1] \times [0; 1] \times [-1; 0]$ domain.

[15]:
```
from dolfin import *
import numpy as np

N = 30
mesh = UnitCubeMesh.create(N, N, N//2, CellType.Type.hexahedron)
# mesh size is smaller near x=y=0
mesh.coordinates()[:, :2] = mesh.coordinates()[:, :2]**2
# mesh size is smaller near z=0 and mapped to a [-1;0] domain along z
mesh.coordinates()[:, 2] = -mesh.coordinates()[:, 2]**2
```

The top surface is defined as a `SubDomain` and corresponding exterior facets are marked as 1. Functions for imposition of Dirichlet BC on other boundaries are also defined.

[16]:
```
class Top(SubDomain):
    def inside(self, x, on_boundary):
        return near(x[2], 0.) and on_boundary
def symmetry_x(x, on_boundary):
        return near(x[0], 0) and on_boundary
def symmetry_y(x, on_boundary):
        return near(x[1], 0) and on_boundary
def bottom(x, on_boundary):
        return near(x[2], -1) and on_boundary

# exterior facets MeshFunction
facets = MeshFunction("size_t", mesh, 2)
facets.set_all(0)
Top().mark(facets, 1)
ds = Measure('ds', subdomain_data=facets)
```

The obstacle shape $h(x, y)$ is now defined and a small indentation depth d is considered. Standard function spaces are then defined for the problem formulation and output field exports. In particular, gap and pressure will be saved later. Finally, the `DirichletBC` are defined.

[17]:
```
R = 0.5
d = 0.02
obstacle = Expression("-d+(pow(x[0],2)+pow(x[1], 2))/2/R", d=d, R=R, degree=2)

V = VectorFunctionSpace(mesh, "CG", 1)
V2 = FunctionSpace(mesh, "CG", 1)
V0 = FunctionSpace(mesh, "DG", 0)

u = Function(V, name="Displacement")
du = TrialFunction(V)
u_ = TestFunction(V)
```

(continues on next page)

(continued from previous page)

```
gap = Function(V2, name="Gap")
p = Function(V0, name="Contact pressure")

bc =[DirichletBC(V, Constant((0., 0., 0.)), bottom),
     DirichletBC(V.sub(0), Constant(0.), symmetry_x),
     DirichletBC(V.sub(1), Constant(0.), symmetry_y)]
```

The elastic constants and functions for formulating the weak form are defined. A penalization stiffness `pen` is introduced and the penalized weak form

$$\text{Find } u \text{ such that } \int_\Omega \sigma(u) : \varepsilon(v) d\Omega + k_{pen} \int_\Gamma \langle u - h \rangle_+ v dS = 0$$

is defined. The corresponding Jacobian `J` is also computed using automatic differentiation.

[18]:
```
E = Constant(10.)
nu = Constant(0.3)
mu = E/2/(1+nu)
lmbda = E*nu/(1+nu)/(1-2*nu)
def eps(v):
    return sym(grad(v))
def sigma(v):
    return lmbda*tr(eps(v))*Identity(3) + 2.0*mu*eps(v)
def ppos(x):
    return (x+abs(x))/2.

pen = Constant(1e4)
form = inner(sigma(u), eps(u_))*dx + pen*dot(u_[2], ppos(u[2]-obstacle))*ds(1)
J = derivative(form, u, du)
```

A non-linear solver is now defined for solving the previous problem. Due to the 3D nature of the problem with a significant number of elements, an iterative Conjugate-Gradient solver is chosen for the linear solver inside the Newton iterations. Note that choosing a large penalization parameter deteriorates the problem conditioning so that solving time will drastically increase and can eventually fail.

[19]:
```
problem = NonlinearVariationalProblem(form, u, bc, J=J)
solver = NonlinearVariationalSolver(problem)
solver.parameters["newton_solver"]["linear_solver"] = "cg"
solver.parameters["newton_solver"]["preconditioner"] = "ilu"

solver.solve()
```

[19]:
```
(8, True)
```

As post-processing, the gap is computed (here on the whole mesh) as well as the pressure. The maximal pressure and the total force resultant (note the factor 4 due to the symmetry of the problem) are compared with respect to the analytical solution. Finally, XDMF output is performed.

[20]:
```
p.assign(-project(sigma(u)[2, 2], V0))
gap.assign(project(obstacle-u[2], V2))

a = sqrt(R*d)
F = 4/3.*float(E)/(1-float(nu)**2)*a*d
p0 = 3*F/(2*pi*a**2)
print("Maximum pressure FE: {0:8.5f} Hertz: {1:8.5f}".format(max(np.abs(p.vector().
↪get_local())), p0))
print("Applied force    FE: {0:8.5f} Hertz: {1:8.5f}".format(4*assemble(p*ds(1)), F))
```

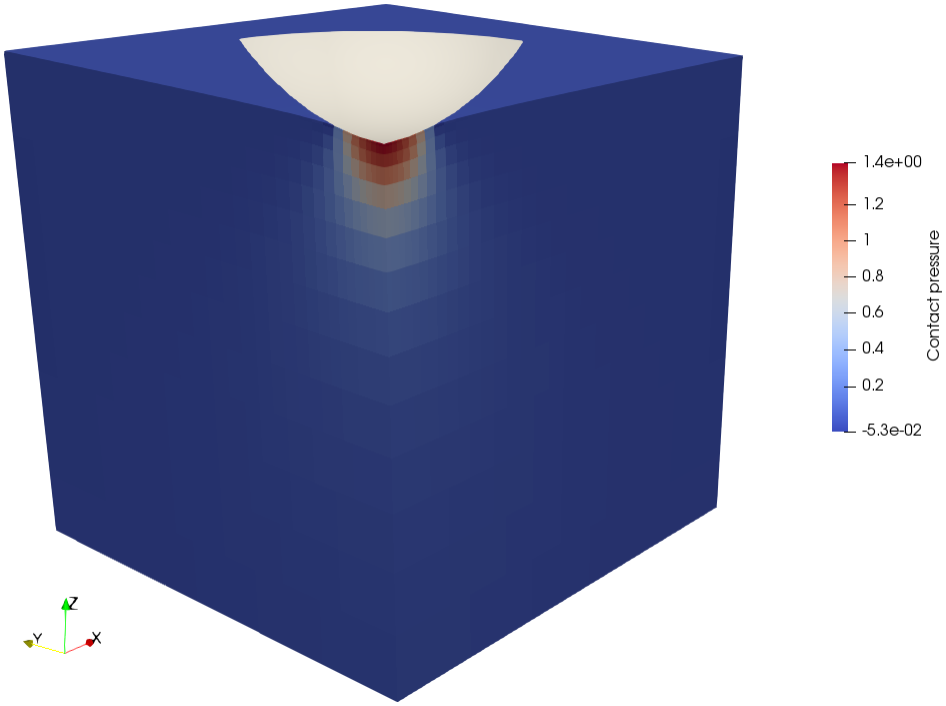
(continues on next page)

(continued from previous page)

```
file_results = XDMFFile("contact_penalty_results.xdmf")
file_results.parameters["flush_output"] = True
file_results.parameters["functions_share_mesh"] = True
file_results.write(u, 0.)
file_results.write(gap, 0.)
file_results.write(p, 0.)
```

```
Maximum pressure FE:  1.42763 Hertz:  1.39916
Applied force    FE:  0.03206 Hertz:  0.02930
```

Maximum pressure over the contact surface and total applied force by the indenter are computed and compared against the analytical solution. Comparison between both solutions shows good agreement although the quality of the results is quite dependent on the chosen penalty sitffness. Here is what the solution looks like.



5.2.3 References

[1] K.L., Johnson, Contact Mechanics, Cambridge University Press, 1987

[2] S.P. Timoshenko, J.N. Goodier. Theory of Elasticity, MacGraw Hill International Book, 1982

[ ]:

5.3 Elasto-plastic analysis of a 2D von Mises material

5.3.1 Introduction

This example is concerned with the incremental analysis of an elasto-plastic von Mises material. The structure response is computed using an iterative predictor-corrector return mapping algorithm embedded in a Newton-Raphson global loop for restoring equilibrium. Due to the simple expression of the von Mises criterion, the return mapping procedure is completely analytical (with linear isotropic hardening). The corresponding sources can be obtained from `vonMises_plasticity.zip`. Another implementation of von Mises plasticity can also be found at https://bitbucket.org/fenics-apps/fenics-solid-mechanics as well as in the numerical tour *Elasto-plastic analysis implemented using the MFront code generator*.

We point out that the 2D nature of the problem will impose keeping track of the out-of-plane ε^p_{zz} plastic strain and dealing with representations of stress/strain states including the zz component. Note also that are not concerned with the issue of volumetric locking induced by incompressible plastic deformations since quadratic triangles in 2D is enough to mitigate the locking phenomenon.

The plastic strain evolution during the cylinder expansion will look like this:

5.3.2 Problem position

In FEniCS 2017.2, the FEniCS Form Compiler `ffc` now uses `uflacs` as a default representation instead of the old `quadrature` representation. However, using `Quadrature` elements generates some bugs in this representation and we therefore need to revert to the old representation. Deprecation warning messages are also disabled. See this post and the corresponding issue for more information.:

```
from dolfin import *
import numpy as np
parameters["form_compiler"]["representation"] = 'quadrature'
import warnings
from ffc.quadrature.deprecation import QuadratureRepresentationDeprecationWarning
warnings.simplefilter("once", QuadratureRepresentationDeprecationWarning)
```

The material is represented by an isotropic elasto-plastic von Mises yield condition of uniaxial strength σ_0 and with isotropic hardening of modulus H. The yield condition is thus given by:

$$f(\boldsymbol{\sigma}) = \sqrt{\frac{3}{2}\boldsymbol{s} : \boldsymbol{s}} - \sigma_0 - Hp \leq 0$$

where p is the cumulated equivalent plastic strain. The hardening modulus can also be related to a tangent elastic modulus $E_t = \dfrac{EH}{E+H}$.

The considered problem is that of a plane strain hollow cylinder of internal (resp. external) radius R_i (resp. R_e) under internal uniform pressure q. Only a quarter of cylinder is generated using `Gmsh` and converted to `.xml` format.

```
# elastic parameters
E = Constant(70e3)
nu = Constant(0.3)
lmbda = E*nu/(1+nu)/(1-2*nu)
mu = E/2./(1+nu)
sig0 = Constant(250.)  # yield strength
Et = E/100.  # tangent modulus
H = E*Et/(E-Et)  # hardening modulus
```

(continues on next page)

(continued from previous page)

```
Re, Ri = 1.3, 1.   # external/internal radius
mesh = Mesh("thick_cylinder.xml")
facets = MeshFunction("size_t", mesh, "thick_cylinder_facet_region.xml")
ds = Measure('ds')[facets]
```

Function spaces will involve a standard CG space for the displacement whereas internal state variables such as plastic strains will be represented using a **Quadrature** element. This choice will make it possible to express the complex non-linear material constitutive equation at the Gauss points only, without involving any interpolation of non-linear expressions throughout the element. It will ensure an optimal convergence rate for the Newton-Raphson method. See Chapter 26 of the FEniCS book. We will need Quadrature elements for 4-dimensional vectors and scalars, the number of Gauss points will be determined by the required degree of the Quadrature element (e.g. `degree=1` yields only 1 Gauss point, `degree=2` yields 3 Gauss points and `degree=3` yields 6 Gauss points (note that this is suboptimal)):

```
deg_u = 2
deg_stress = 2
V = VectorFunctionSpace(mesh, "CG", deg_u)
We = VectorElement("Quadrature", mesh.ufl_cell(), degree=deg_stress, dim=4, quad_
↪scheme='default')
W = FunctionSpace(mesh, We)
W0e = FiniteElement("Quadrature", mesh.ufl_cell(), degree=deg_stress, quad_scheme=
↪'default')
W0 = FunctionSpace(mesh, W0e)
```

Note: In older versions, it was possible to define Quadrature function spaces directly using `FunctionSpace(mesh, "Quadrature", 1)`. This is no longer the case since FEniCS 2016.1 (see this issue). Instead, Quadrature elements must first be defined by specifying the associated degree and quadrature scheme before defining the associated FunctionSpace.

Various functions are defined to keep track of the current internal state and currently computed increments:

```
sig = Function(W)
sig_old = Function(W)
n_elas = Function(W)
beta = Function(W0)
p = Function(W0, name="Cumulative plastic strain")
u = Function(V, name="Total displacement")
du = Function(V, name="Iteration correction")
Du = Function(V, name="Current increment")
v = TrialFunction(V)
u_ = TestFunction(V)
```

Boundary conditions correspond to symmetry conditions on the bottom and left parts (resp. numbered 1 and 3). Loading consists of a uniform pressure on the internal boundary (numbered 4). It will be progressively increased from 0 to $q_{lim} = \frac{2}{\sqrt{3}}\sigma_0 \log\left(\frac{R_e}{R_i}\right)$ which is the analytical collapse load for a perfectly-plastic material (no hardening):

```
bc = [DirichletBC(V.sub(1), 0, facets, 1), DirichletBC(V.sub(0), 0, facets, 3)]


n = FacetNormal(mesh)
q_lim = float(2/sqrt(3)*ln(Re/Ri)*sig0)
loading = Expression("-q*t", q=q_lim, t=0, degree=2)
```

(continues on next page)

(continued from previous page)

```
def F_ext(v):
    return loading*dot(n, v)*ds(4)
```

5.3.3 Constitutive relation update

Before writing the variational form, we now define some useful functions which will enable performing the constitutive relation update using a return mapping procedure. This step is quite classical in FEM plasticity for a von Mises criterion with isotropic hardening and follow notations from [BON2014]. First, the strain tensor will be represented in a 3D fashion by appending zeros on the out-of-plane components since, even if the problem is 2D, the plastic constitutive relation will involve out-of-plane plastic strains. The elastic consitutive relation is also defined and a function `as_3D_tensor` will enable to represent a 4 dimensional vector containing respectively xx, yy, zz and xy components as a 3D tensor:

```
def eps(v):
    e = sym(grad(v))
    return as_tensor([[e[0, 0], e[0, 1], 0],
                      [e[0, 1], e[1, 1], 0],
                      [0, 0, 0]])
def sigma(eps_el):
    return lmbda*tr(eps_el)*Identity(3) + 2*mu*eps_el
def as_3D_tensor(X):
    return as_tensor([[X[0], X[3], 0],
                      [X[3], X[1], 0],
                      [0, 0, X[2]]])
```

The return mapping procedure consists in finding a new stress $\boldsymbol{\sigma}_{n+1}$ and internal variable p_{n+1} state verifying the current plasticity condition from a previous stress $\boldsymbol{\sigma}_n$ and internal variable p_n state and an increment of total deformation $\Delta\boldsymbol{\varepsilon}$. An elastic trial stress $\boldsymbol{\sigma}_{\text{elas}} = \boldsymbol{\sigma}_n + \mathbf{C}\Delta\boldsymbol{\varepsilon}$ is first computed. The plasticity criterion is then evaluated with the previous plastic strain $f_{\text{elas}} = \sigma_{\text{elas}}^{eq} - \sigma_0 - Hp_n$ where $\sigma_{\text{elas}}^{eq} = \sqrt{\frac{3}{2}\boldsymbol{s} : \boldsymbol{s}}$ with the deviatoric elastic stress $\boldsymbol{s} = \operatorname{dev}\boldsymbol{\sigma}_{\text{elas}}$. If $f_{\text{elas}} < 0$, no plasticity occurs during this time increment and $\Delta p, \Delta\boldsymbol{\varepsilon}^p = 0$.

Otherwise, plasticity occurs and the increment of plastic strain is given by $\Delta p = \dfrac{f_{\text{elas}}}{3\mu + H}$. Hence, both elastic and plastic evolution can be accounted for by defining the plastic strain increment as follows:

$$\Delta p = \frac{\langle f_{\text{elas}} \rangle_+}{3\mu + H}$$

where $\langle \star \rangle_+$ represents the positive part of $\star$ and is obtained by function `ppos`. Plastic evolution also requires the computation of the normal vector to the final yield surface given by $\boldsymbol{n}_{\text{elas}} = \boldsymbol{s}/\sigma_{\text{elas}}^{eq}$. In the following, this vector must be zero in case of elastic evolution. Hence, we multiply it by $\dfrac{\langle f_{\text{elas}} \rangle_+}{f_{\text{elas}}}$ to tackle both cases in a single expression. The final stress state is corrected by the plastic strain as follows $\boldsymbol{\sigma}_{n+1} = \boldsymbol{\sigma}_{\text{elas}} - \beta\boldsymbol{s}$ with $\beta = \dfrac{3\mu}{\sigma_{\text{elas}}^{eq}}\Delta p$. It can be observed that the last term vanishes in case of elastic evolution so that the final stress is indeed the elastic predictor.

```
ppos = lambda x: (x+abs(x))/2.
def proj_sig(deps, old_sig, old_p):
    sig_n = as_3D_tensor(old_sig)
    sig_elas = sig_n + sigma(deps)
    s = dev(sig_elas)
    sig_eq = sqrt(3/2.*inner(s, s))
    f_elas = sig_eq - sig0 - H*old_p
    dp = ppos(f_elas)/(3*mu+H)
```

(continues on next page)

(continued from previous page)

```
    n_elas = s/sig_eq*ppos(f_elas)/f_elas
    beta = 3*mu*dp/sig_eq
    new_sig = sig_elas-beta*s
    return as_vector([new_sig[0, 0], new_sig[1, 1], new_sig[2, 2], new_sig[0, 1]]), \
           as_vector([n_elas[0, 0], n_elas[1, 1], n_elas[2, 2], n_elas[0, 1]]), \
           beta, dp
```

Note: We could have used conditionals to write more explicitly the difference between elastic and plastic evolution.

In order to use a Newton-Raphson procedure to resolve global equilibrium, we also need to derive the algorithmic consistent tangent matrix given by:

$$\mathbf{C}_{\text{tang}}^{\text{alg}} = \mathbf{C} - 3\mu\left(\frac{3\mu}{3\mu+H} - \beta\right)\boldsymbol{n}_{\text{elas}} \otimes \boldsymbol{n}_{\text{elas}} - 2\mu\beta\mathbf{DEV}$$

where $\mathbf{DEV}$ is the 4th-order tensor associated with the deviatoric operator (note that $\mathbf{C}_{\text{tang}}^{\text{alg}} = \mathbf{C}$ for elastic evolution). Contrary to what is done in the FEniCS book, we do not store it as the components of a 4th-order tensor but it will suffice keeping track of the normal vector and the β parameter related to the plastic strains. We instead define a function computing the tangent stress $\boldsymbol{\sigma}_{\text{tang}} = \mathbf{C}_{\text{tang}}^{\text{alg}}\boldsymbol{\varepsilon}$ as follows:

```
def sigma_tang(e):
    N_elas = as_3D_tensor(n_elas)
    return sigma(e) - 3*mu*(3*mu/(3*mu+H)-beta)*inner(N_elas, e)*N_elas-
↪2*mu*beta*dev(e)
```

5.3.4 Global problem and Newton-Raphson procedure

We are now in position to derive the global problem with its associated Newton-Raphson procedure. Each iteration will require establishing equilibrium by driving to zero the residual between the internal forces associated with the current stress state `sig` and the external force vector. Because we use Quadrature elements a custom integration measure must be defined to match the quadrature degree and scheme used by the Quadrature elements:

```
metadata = {"quadrature_degree": deg_stress, "quadrature_scheme": "default"}
dxm = dx(metadata=metadata)

a_Newton = inner(eps(v), sigma_tang(eps(u_)))*dxm
res = -inner(eps(u_), as_3D_tensor(sig))*dxm + F_ext(u_)
```

The consitutive update defined earlier will perform nonlinear operations on the stress and strain tensors. These nonlinear expressions must then be projected back onto the associated Quadrature spaces. Since these fields are defined locally in each cell (in fact only at their associated Gauss point), this projection can be performed locally. For this reason, we define a `local_project` function that use the `LocalSolver` to gain in efficiency (see also *Efficient projection on DG or Quadrature spaces*) for more details:

```
def local_project(v, V, u=None):
    dv = TrialFunction(V)
    v_ = TestFunction(V)
    a_proj = inner(dv, v_)*dxm
    b_proj = inner(v, v_)*dxm
    solver = LocalSolver(a_proj, b_proj)
    solver.factorize()
    if u is None:
```

(continues on next page)

(continued from previous page)

```
        u = Function(V)
        solver.solve_local_rhs(u)
        return u
    else:
        solver.solve_local_rhs(u)
        return
```

Note: We could have used the standard `project` if we are not interested in optimizing the code. However, the use of Quadrature elements would have required telling `project` to use an appropriate integration measure to solve the global L^2 projection that occurs under the hood. This would have needed either redefining explicitly the projection associated forms (as we just did) or specifiying the appropriate quadrature degree to the form compiler as follows `project(sig_, W, form_compiler_parameters={"quadrature_degree":deg_stress})`

Before defining the Newton-Raphson loop, we set up the output file and appropriate FunctionSpace (here piecewise constant) and Function for output of the equivalent plastic strain since XDMF output does not handle Quadrature elements:

```
file_results = XDMFFile("plasticity_results.xdmf")
file_results.parameters["flush_output"] = True
file_results.parameters["functions_share_mesh"] = True
P0 = FunctionSpace(mesh, "DG", 0)
p_avg = Function(P0, name="Plastic strain")
```

We now define the global Newton-Raphson loop. We will discretize the applied loading using `Nincr` increments from 0 up to 1.1 (we exclude zero from the list of load steps). A nonlinear discretization is adopted to refine the load steps during the plastic evolution phase. At each time increment, the system is assembled and the residual norm is computed. The incremental displacement `Du` is initialized to zero and the inner iteration loop performing the constitutive update is initiated. Inside this loop, corrections `du` to the displacement increment are computed by solving the Newton system and the return mapping update is performed using the current total strain increment `deps`. The resulting quantities are then projected onto their appropriate FunctionSpaces. The Newton system and residuals are reassembled and this procedure continues until the residual norm falls below a given tolerance. After convergence of the iteration loop, the total displacement, stress and plastic strain states are updated

```
Nitermax, tol = 200, 1e-8  # parameters of the Newton-Raphson procedure
Nincr = 20
load_steps = np.linspace(0, 1.1, Nincr+1)[1:]**0.5
results = np.zeros((Nincr+1, 2))
for (i, t) in enumerate(load_steps):
    loading.t = t
    A, Res = assemble_system(a_Newton, res, bc)
    nRes0 = Res.norm("l2")
    nRes = nRes0
    Du.interpolate(Constant((0, 0)))
    print("Increment:", str(i+1))
    niter = 0
    while nRes/nRes0 > tol and niter < Nitermax:
        solve(A, du.vector(), Res, "mumps")
        Du.assign(Du+du)
        deps = eps(Du)
        sig_, n_elas_, beta_, dp_ = proj_sig(deps, sig_old, p)
        local_project(sig_, W, sig)
        local_project(n_elas_, W, n_elas)
        local_project(beta_, W0, beta)
```

(continues on next page)

(continued from previous page)

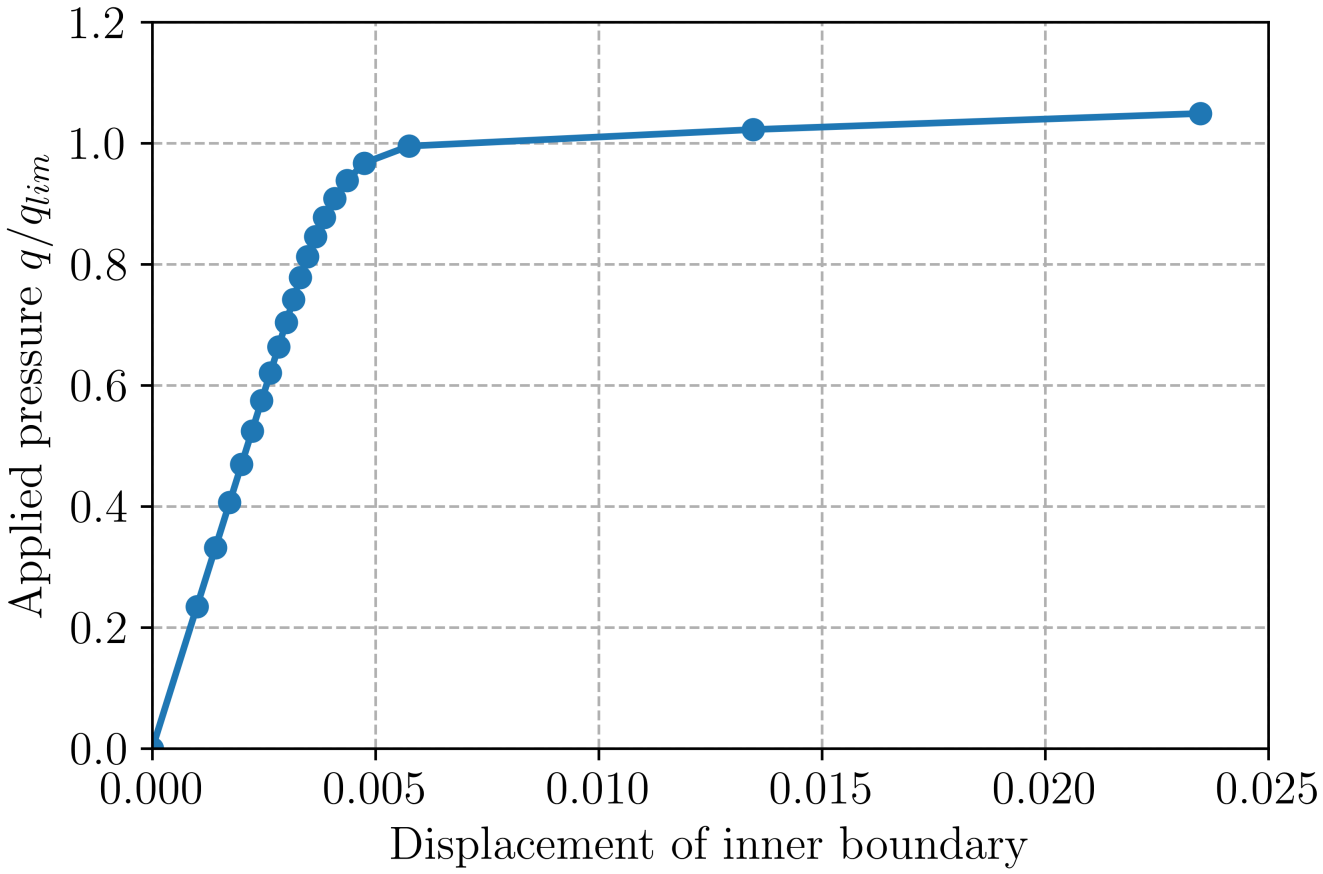
```
        A, Res = assemble_system(a_Newton, res, bc)
        nRes = Res.norm("l2")
        print("    Residual:", nRes)
        niter += 1
    u.assign(u+Du)
    sig_old.assign(sig)
    p.assign(p+local_project(dp_, W0))
```

5.3.5 Post-processing

Inside the incremental loop, the displacement and plastic strains are exported at each time increment, the plastic strain must first be projected onto the previously defined DG FunctionSpace. We also monitor the value of the cylinder displacement on the inner boundary. The load-displacement curve is then plotted:

```
    file_results.write(u, t)
    p_avg.assign(project(p, P0))
    file_results.write(p_avg, t)
    results[i+1, :] = (u(Ri, 0)[0], t)

import matplotlib.pyplot as plt
plt.plot(results[:, 0], results[:, 1], "-o")
plt.xlabel("Displacement of inner boundary")
plt.ylabel(r"Applied pressure $q/q_{lim}$")
plt.show()
```

The load-displacement curve looks as follows:



It can also be checked that the analytical limit load is also well reproduced when considering a zero hardening modulus.

5.3.6 References

5.4 Elasto-plastic analysis implemented using the *MFront* code generator

This numerical tour has been written in collaboration with **Thomas Helfer** (thomas.helfer@cea.fr), MFront's main developper.

5.4.1 Introduction

This example is concerned with the incremental analysis of an elasto-plastic von Mises material. The behaviour is implemented using the code generator tool *MFront* [HEL2015] (see http://tfel.sourceforge.net). The behaviour integration is handled by the *MFrontGenericInterfaceSupport* project (see https://github.com/thelfer/MFrontGenericInterfaceSupport) which allows to load behaviours generated by *MFront* and handle the behaviour integration. Other information concerning FEniCS binding in *MFront* can be found at https://thelfer.github.io/mgis/web/FEniCSBindings.html, in particular binding through the C++ interface and inspired by the Fenics Solid Mechanics project is discussed.

The considered example is exactly the same as the pure FEniCS tutorial *Elasto-plastic analysis of a 2D von Mises material* so that both implementations can be compared. Many implementation steps are common to both demos and

will not be discussed again here, the reader can refer to the previous tutorial for more details. The sources can be downloaded from `plasticity_mfront.zip`.

Let us point out that a pure FEniCS implementation was possible **only for this specific constitutive law**, namely von Mises plasticity with isotropic linear hardening, since the return mapping step is analytical in this case and can be expressed using simple UFL operators. The use of *MFront* can therefore make it possible to consider more complex material laws. Indeed, one only has to change the names of the behaviour and library generated by *MFront* to change the material behaviours, which can be arbitraly complex (although limited to small strains). This will be illustrated in forthcoming tutorials. Interested users may have a look at the examples of the *MFront* gallery to have a small overview of *MFront* abilities: http://tfel.sourceforge.net/gallery.html

5.4.2 Prerequisites

In order to run this numerical tour, you must first install the TFEL library on which *MFront* is built as well as MGIS (*MFrontGenericInterfaceSupport*). Please note that development versions are used for now.

To proceed with the installation, we provide the following installation shell script `install.sh`. Before running it, please install the necessary prerequisites mentioned in the script, e.g. on Ubuntu run:

```
> sudo apt-get install cmake libboost-all-dev g++ gfortran
> sudo apt-get install git libqt5svg5-dev qtwebengine5-dev
> sudo apt-get install python3-matplotlib
```

Once the installation script finished installing TFEL and MGIS, you can use MGIS by running the following command:

```
> source <PREFIX>/codes/mgis/master/install/env.sh
```

in which `<PREFIX>` is the installation directory you have chosen.

As recalled later in the tutorial, the MFront behaviour file must first be compiled before running this Python script. The compilation command is:

```
> mfront --obuild --interface=generic IsotropicLinearHardeningPlasticity.mfront
```

5.4.3 Problem position

The present implementation will heavily rely on `Quadrature` elements to represent previous and current stress states, internal state variables (cumulated equivalent plastic strain p in the present case) as well as components of the tangent stiffness matrix. The use of `quadrature` representation is therefore still needed to circumvent the known issue with `uflacs`:

```
from __future__ import print_function
from dolfin import *
import numpy as np
parameters["form_compiler"]["representation"] = 'quadrature'
import warnings
from ffc.quadrature.deprecation import QuadratureRepresentationDeprecationWarning
warnings.simplefilter("once", QuadratureRepresentationDeprecationWarning)
```

In order to bind *MFront* behaviours with FEniCS Python interface we will first load the *MFrontGenericInterfaceSupport* Python package named `mgis` and more precisely the `behaviour` subpackage:

```
import mgis.behaviour as mgis_bv
```

As before, we load a `Gmsh` generated mesh representing a portion of a thick cylinder:

```
Re, Ri = 1.3, 1.   # external/internal radius
mesh = Mesh("thick_cylinder.xml")
facets = MeshFunction("size_t", mesh, "thick_cylinder_facet_region.xml")
ds = Measure('ds')[facets]
```

We now define appropriate function spaces. Standard CG-space of degree 2 will still be used for the displacement whereas various Quadrature spaces are considered for:

- stress/strain-like variables (represented here as 4-dimensional vector since the zz component must be considered in the plane strain plastic behaviour)
- scalar variables for the cumulated plastic strain
- the consistent tangent matrix represented here as a tensor of shape 4x4

As in the previous tutorial a `degree=2` quadrature rule (i.e. 3 Gauss points) will be used. In the end, the total number of Gauss points in the mesh is retrieved as it will be required to instantiate *MFront* objects (note that it can be obtained from the dimension of the Quadrature function spaces or computed from the number of mesh cells and the chosen quadrature degree):

```
deg_u = 2
deg_stress = 2
stress_strain_dim = 4
V = VectorFunctionSpace(mesh, "CG", deg_u)
# Quadrature space for sigma
We = VectorElement("Quadrature", mesh.ufl_cell(), degree=deg_stress,
                   dim=stress_strain_dim, quad_scheme='default')
W = FunctionSpace(mesh, We)
# Quadrature space for p
W0e = FiniteElement("Quadrature", mesh.ufl_cell(), degree=deg_stress, quad_scheme=
↪'default')
W0 = FunctionSpace(mesh, W0e)
# Quadrature space for tangent matrix
Wce = TensorElement("Quadrature", mesh.ufl_cell(), degree=deg_stress,
                    shape=(stress_strain_dim, stress_strain_dim),
                    quad_scheme='default')
WC = FunctionSpace(mesh, Wce)

# get total number of gauss points
ngauss = W0.dim()
```

Various functions are defined to keep track of the current internal state (stresses, current strain estimate, cumulative plastic strain and cpnsistent tangent matrix) as well as the previous displacement, current displacement estimate and current iteration correction:

```
# Define functions based on Quadrature spaces
sig = Function(W, name="Current stresses")
sig_old = Function(W)
Eps1 = Function(W, name="Current strain increment estimate at the end of the end step
↪")
Ct = Function(WC, name="Consistent tangent operator")
p = Function(W0, name="Cumulative plastic strain")
p_old = Function(W0)

u  = Function(V, name="Displacement at the beginning of the time step")
u1 = Function(V, name="Current displacement estimate at the end of the end step")
du = Function(V, name="Iteration correction")
```

(continues on next page)

(continued from previous page)

```
v = TrialFunction(V)
u_ = TestFunction(V)
```

5.4.4 Material constitutive law definition using *MFront*

We now define the material. First let us make a rapid description of some classes and functions introduced by the *MFrontGenericInterface* project that will be helpful for this tutorial:

- the `Behaviour` class handles all the information about a specific *MFront* behaviour. It is created by the `load` function which takes the path to a library, the name of a behaviour and a modelling hypothesis.
- the `MaterialDataManager` class handles a bunch of integration points. It is instantiated using an instance of the `Behaviour` class and the number of integration points[1]. The `MaterialDataManager` contains various interesting members:
  - `s0`: data structure of the `MaterialStateManager` type which stands for the material state at the beginning of the time step.
  - `s1`: data structure of the `MaterialStateManager` type which stands for the material state at the end of the time step.
  - `K`: a `numpy` array containing the consistent tangent operator at each integration points.
- the `MaterialStateManager` class describe the state of a material. The following members will be useful in the following:
  - `gradients`: a numpy array containing the value of the gradients at each integration points. The number of components of the gradients at each integration points is given by the `gradients_stride` member.
  - `thermodynamic_forces`: a numpy array containing the value of the thermodynamic forces at each integration points. The number of components of the thermodynamic forces at each integration points is given by the `thermodynamic_forces_stride` member.
  - `internal_state_variables`: a numpy array containing the value of the internal state variables at each integration points. The number of internal state variables at each integration points is given by the `internal_state_variables_stride` member.
- the `setMaterialProperty` and `setExternalStateVariable` functions can be used to set the value a material property or a state variable respectively.
- the `update` function updates an instance of the `MaterialStateManager` by copying the state `s1` at the end of the time step in the state `s0` at the beginning of the time step.
- the `revert` function reverts an instance of the `MaterialStateManager` by copying the state `s0` at the beginning of the time step in the state `s1` at the end of the time step.
- the `integrate` function triggers the behaviour integration at each integration points. Various overloads of this function exist. We will use a version taking as argument a `MaterialStateManager`, the time increment and a range of integration points.

In the present case, we compute a plane strain von Mises plasticity with isotropic linear hardening. The material behaviour is implemented in the `IsotropicLinearHardeningPlasticity.mfront` file which must first be compiled to generate the appropriate librairies as follows:

```
> mfront --obuild --interface=generic IsotropicLinearHardeningPlasticity.mfront
```

We can then setup the `MaterialDataManager`:

[1] Note that an instance of MaterialDataManager keeps a reference to the behaviour

```
# Defining the modelling hypothesis
h = mgis_bv.Hypothesis.PlaneStrain
# Loading the behaviour
b = mgis_bv.load('src/libBehaviour.so','IsotropicLinearHardeningPlasticity',h)
# Setting the material data manager
m = mgis_bv.MaterialDataManager(b, ngauss)
# elastic parameters
E = 70e3
nu = 0.3
# yield strength
sig0 = 250.
Et = E/100.
# hardening slope
H = E*Et/(E-Et)

for s in [m.s0, m.s1]:
    mgis_bv.setMaterialProperty(s, "YoungModulus", E)
    mgis_bv.setMaterialProperty(s, "PoissonRatio", nu)
    mgis_bv.setMaterialProperty(s, "HardeningSlope", H)
    mgis_bv.setMaterialProperty(s, "YieldStrength", sig0)
    mgis_bv.setExternalStateVariable(s, "Temperature", 293.15)
```

Boundary conditions and external loading are defined as before along with the analytical limit load solution:

```
bc = [DirichletBC(V.sub(1), 0, facets, 1), DirichletBC(V.sub(0), 0, facets, 3)]


n = FacetNormal(mesh)
q_lim = float(2/sqrt(3)*ln(Re/Ri)*sig0)
loading = Expression("-q*t", q=q_lim, t=0, degree=2)

def F_ext(v):
    return loading*dot(n, v)*ds(4)
```

5.4.5 Global problem and Newton-Raphson procedure

Before writing the global Newton system, we first define the strain measure function `eps_MFront` consistent with the *MFront* conventions (see http://tfel.sourceforge.net/tensors.html for details). We also define the custom quadrature measure and the projection function onto Quadrature spaces:

```
def eps_MFront(v):
    e = sym(grad(v))
    return as_vector([e[0, 0], e[1, 1], 0, sqrt(2)*e[0, 1]])

metadata = {"quadrature_degree": deg_stress, "quadrature_scheme": "default"}
dxm = dx(metadata=metadata)

def local_project(v, V, u=None):
    """
    projects v on V with custom quadrature scheme dedicated to
    FunctionSpaces V of `Quadrature` type

    if u is provided, result is appended to u
    """
    dv = TrialFunction(V)
```

(continues on next page)

(continued from previous page)

```
    v_ = TestFunction(V)
    a_proj = inner(dv, v_)*dxm
    b_proj = inner(v, v_)*dxm
    solver = LocalSolver(a_proj, b_proj)
    solver.factorize()
    if u is None:
        u = Function(V)
        solver.solve_local_rhs(u)
        return u
    else:
        solver.solve_local_rhs(u)
        return
```

The bilinear form of the global problem is obtained using the consistent tangent matrix `Ct` and the *MFront* strain measure, whereas the right-hand side consists of the residual between the internal forces associated with the current stress state `sig` and the external force vector.

```
a_Newton = inner(eps_MFront(v), dot(Ct, eps_MFront(u_)))*dxm
res = -inner(eps_MFront(u_), sig)*dxm + F_ext(u_)
```

Before defining the Newton-Raphson loop, we set up the output file and appropriate FunctionSpace (here piecewise constant) and Function for output of the equivalent plastic strain since XDMF output does not handle Quadrature elements:

```
file_results = XDMFFile("plasticity_results.xdmf")
file_results.parameters["flush_output"] = True
file_results.parameters["functions_share_mesh"] = True
P0 = FunctionSpace(mesh, "DG", 0)
p_avg = Function(P0, name="Plastic strain")
```

The tangent stiffness is also initialized with the elasticity matrix:

```
it = mgis_bv.IntegrationType.PredictionWithElasticOperator
mgis_bv.integrate(m, it, 0, 0, m.n);
Ct.vector().set_local(m.K.flatten())
Ct.vector().apply("insert")
```

The main difference with respect to the pure FEniCS implementation of the previous tutorial is that *MFront* computes the current stress state and stiffness matrix (`integrate` method) based on the value of the total strain which is computed from the total displacement estimate `u1`. The associated strain is projected onto the appropriate Quadrature function space so that its array of values at all Gauss points is given to *MFront* via the `m.s1.gradients` attribute. The flattened array of stress and tangent stiffness values are then used to update the current stress and tangent stiffness variables. The cumulated plastic strain is also retrieved from the `internal_state_variables` attribute (p being the last column in the present case). At the end of the iteration loop, the material behaviour and the previous displacement variable are updated:

```
Nitermax, tol = 200, 1e-8  # parameters of the Newton-Raphson procedure
Nincr = 20
load_steps = np.linspace(0, 1.1, Nincr+1)[1:]**0.5
results = np.zeros((Nincr+1, 2))
for (i, t) in enumerate(load_steps):
    loading.t = t
    A, Res = assemble_system(a_Newton, res, bc)
    nRes0 = Res.norm("l2")
    nRes = nRes0
```

(continues on next page)

(continued from previous page)

```
    u1.assign(u)
    print("Increment:", str(i+1))
    niter = 0
    while nRes/nRes0 > tol and niter < Nitermax:
        solve(A, du.vector(), Res, "mumps")
        # update the current estimate of the displacement at the end of the time step
        u1.assign(u1+du)
        # compute the current estimate of the strain at the end of the
        # time step using `MFront` conventions
        local_project(eps_MFront(u1), W, Eps1)
        # copy the strain values to `MGIS`
        m.s1.gradients[:, :] = Eps1.vector().get_local().reshape((m.n, stress_strain_
↪dim))
        # integrate the behaviour
        it = mgis_bv.IntegrationType.IntegrationWithConsistentTangentOperator
        mgis_bv.integrate(m, it, 0, 0, m.n);
        # getting the stress and consistent tangent operator back to
        # the FEniCS world.
        sig.vector().set_local(m.s1.thermodynamic_forces.flatten())
        sig.vector().apply("insert")
        Ct.vector().set_local(m.K.flatten())
        Ct.vector().apply("insert")
        # retrieve cumulated plastic strain values
        p.vector().set_local(m.s1.internal_state_variables[:, -1])
        p.vector().apply("insert")
        # solve Newton system
        A, Res = assemble_system(a_Newton, res, bc)
        nRes = Res.norm("l2")
        print("    Residual:", nRes)
        niter += 1
    # update the displacement for the next increment
    u.assign(u1)
    # update the material
    mgis_bv.update(m)

    # postprocessing results
    file_results.write(u, t)
    p_avg.assign(project(p, P0))
    file_results.write(p_avg, t)
    results[i+1, :] = (u(Ri, 0)[0], t)

import matplotlib.pyplot as plt
plt.plot(results[:, 0], results[:, 1], "-o")
plt.xlabel("Displacement of inner boundary")
plt.ylabel(r"Applied pressure $q/q_{lim}$")
plt.show()
```

Note: Note that we defined the cumulative plastic strain variable p in FEniCS only for post-processing purposes. In fact, FEniCS deals only with the global equilibrium whereas *MFront* manages the history of internal state variables, so that this variable would not have been needed if we were not interested in post-processing it.

5.4.6 Results and future works

We can verify that the convergence of the Newton-Raphson procedure is extremely similar between the *MFront*-based implementation and the pure FEniCS one, the same number of iterations per increment is obtained along with close values of the residual.

Total computing time took approximately:

- 5.9s for the present *MFront* implementation against
- 6.8s for the previous FEniCS-only implementation

Several points need to be mentioned regarding this implementation efficiency:

- MGIS can handle parallel integration of the constitutive law which has not been used for the present computation
- the present approach can be improved by letting MGIS reuse the memory already allocated by FEniCS which will reduce information transfer times and memory consumption
- extension to large strains is a work in progress
- this FEniCS/MGIS coupling will make it possible, in a near future, to test in a rapid manner generalized constitutive laws (higher-order theories, phase-field) and/or multiphysics couplings

5.4.7 References

which has been used for its initialization: the user must ensure that this behaviour outlives the instance of the MaterialDataManager, otherwise memory corruption may occur.

CHAPTER 6

Documented demos coupling FEniCS with MFront

These demos are based on the `mgis.fenics` module of the MFrontGenericInterfaceSupport (MGIS) project. A general introduction of the package is available here. This project has been realized in collaboration with Thomas Helfer (CEA, thomas.helfer@cea.fr).

Contents:

CHAPTER 7

Beams

Contents:

The corresponding files can be obtained from:

- Jupyter Notebook: `beam_buckling.ipynb`
- Python script: `beam_buckling.py`

7.1 Eulerian buckling of a beam

In this numerical tour, we will compute the critical buckling load of a straight beam under normal compression, the classical Euler buckling problem. Usually, buckling is an important mode of failure for slender beams so that a standard Euler-Bernoulli beam model is sufficient. However, since FEniCS does not support Hermite elements ensuring C^1-formulation for the transverse deflection, implementing such models is not straightforward and requires using advanced DG formulations for instance, see the `fenics-shell` implementation of the Love-Kirchhoff plate model or the FEniCS documented demo on the biharmonic equation.

As a result, we will simply formulate the buckling problem using a Timoshenko beam model.

7.1.1 Timoshenko beam model formulation

We first formulate the stiffness bilinear form of the Timoshenko model given by:

$$k((w,\theta),(\widehat{w},\widehat{\theta})) = \int_0^L EI\frac{d\theta}{dx}\frac{d\widehat{\theta}}{dx}dx + \int_0^L \kappa\mu S\left(\frac{dw}{dx}-\theta\right)\left(\frac{d\widehat{w}}{dx}-\widehat{\theta}\right)dx$$

where $I = bh^3/12$ is the bending inertia for a rectangular beam of width b and height h, $S = bh$ the cross-section area, E the material Young modulus and μ the shear modulus and $\kappa = 5/6$ the shear correction factor. We will use a P^2/P^1 interpolation for the mixed field (w,θ).

For issues related to shear-locking and reduced integration formulation, we refer to the *Reissner-Mindlin plate with Quadrilaterals* tour.

[5]:
```
from dolfin import *
import numpy as np
import matplotlib.pyplot as plt
%matplotlib notebook

L = 10.
thick = Constant(0.03)
width = Constant(0.01)
E = Constant(70e3)
nu = Constant(0.)

EI = E*width*thick**3/12
GS = E/2/(1+nu)*thick*width
kappa = Constant(5./6.)


N = 100
mesh = IntervalMesh(N, 0, L)

U = FiniteElement("CG", mesh.ufl_cell(), 2)
T = FiniteElement("CG", mesh.ufl_cell(), 1)
V = FunctionSpace(mesh, U*T)

u_ = TestFunction(V)
du = TrialFunction(V)
(w_, theta_) = split(u_)
(dw, dtheta) = split(du)


k_form = EI*inner(grad(theta_), grad(dtheta))*dx + \
         kappa*GS*dot(grad(w_)[0]-theta_, grad(dw)[0]-dtheta)*dx
l_form = Constant(1.)*u_[0]*dx
```

As in the *Modal analysis of an elastic structure* tour, a dummy linear form `l_form` is used to call the `assemble_system` function which retains the symmetric structure of the associated matrix when imposing boundary conditions. Here, we will consider clamped conditions on the left side $x = 0$ and simple supports on the right side $x = L$.

[6]:
```
def both_ends(x, on_boundary):
    return on_boundary
def left_end(x, on_boundary):
    return near(x[0], 0) and on_boundary

bc = [DirichletBC(V.sub(0), Constant(0.), both_ends),
      DirichletBC(V.sub(1), Constant(0.), left_end)]

K = PETScMatrix()
assemble_system(k_form, l_form, bc, A_tensor=K)
```

[6]:
```
(<dolfin.cpp.la.PETScMatrix at 0x7f454e3dbbf8>,
 <dolfin.cpp.la.Vector at 0x7f454e3dbf10>)
```

7.1.2 Construction of the geometric stiffness matrix

The buckling analysis amounts to solving an eigenvalue problem of the form:

$$(\mathbf{K} + \lambda \mathbf{K_G})\mathbf{U} = 0$$

in which the geometric stiffness matrix $\mathbf{K_G}$ depends (linearly) on a prestressed state, the amplitude of which is represented by λ. The eigenvalue/eigenvector $(\lambda, \mathbf{U})$ solving the previous generalized eigenproblem respectively correspond to the critical buckling load and its associated buckling mode. For a beam in which the prestressed state correspond to a purely compression state of intensity $N_0 > 0$, the geometric stiffness bilinear form is given by:

$$k_G((w,\theta),(\widehat{w},\widehat{\theta})) = -\int_0^L N_0 \frac{dw}{dx}\frac{d\widehat{w}}{dx} dx$$

which is assembled below into the `KG` `PETScMatrix` (up to the negative sign).

[7]:
```
N0 = Constant(1e-3)
kg_form = N0*dot(grad(w_), grad(dw))*dx
KG = PETScMatrix()
assemble(kg_form, tensor=KG)
for bci in bc:
    bci.zero(KG)
```

Note that we made use of the `zero` method of `DirichletBC` making the rows of the matrix associated with the boundary condition zero. If we used instead the `apply` method, the rows would have been replaced with a row of zeros with a 1 on the diagonal (as for the stiffness matrix `K`). As a result, we would have obtained an eigenvalue equal to 1 for each row with a boundary condition which can make more troublesome the computation of eigenvalues if they happen to be close to 1. Replacing with a full row of zeros in `KG` results in infinite eigenvalues for each boundary condition which is more suitable when looking for the lowest eigenvalues of the buckling problem.

7.1.3 Setting and solving the eigenvalue problem

Up to the negative sign cancelling from the previous definition of `KG`, we now formulate the generalized eigenvalue problem $\mathbf{KU} = -\lambda \mathbf{K_G}\mathbf{U}$ using the `SLEPcEigenSolver`. The only difference from what has already been discussed in the dynamic modal analysis numerical tour is that buckling eigenvalue problem may be more difficult to solve than modal analysis in certain cases, it is therefore beneficial to prescribe a value of the spectral shift close to the critical buckling load.

[8]:
```
eigensolver = SLEPcEigenSolver(K, KG)
eigensolver.parameters['problem_type'] = 'gen_hermitian'
eigensolver.parameters['spectral_transform'] = 'shift-and-invert'
eigensolver.parameters['spectral_shift'] = 1e-3
eigensolver.parameters['tolerance'] = 1e-12

N_eig = 3   # number of eigenvalues
print("Computing {} first eigenvalues...".format(N_eig))
eigensolver.solve(N_eig)

# Exact solution computation
from scipy.optimize import root
from math import tan
falpha = lambda x: tan(x)-x
alpha = lambda n: root(falpha, 0.99*(2*n+1)*pi/2.)['x'][0]

plt.figure()
```

(continues on next page)

(continued from previous page)

```
# Extraction
print("Critical buckling loads:")
for i in range(N_eig):
    # Extract eigenpair
    r, c, rx, cx = eigensolver.get_eigenpair(i)

    critical_load_an = alpha(i+1)**2*float(EI/N0)/L**2
    print("Exact: {0:>10.5f}  FE: {1:>10.5f}  Rel. gap {2:1.2f}%%".format(
           critical_load_an, r, 100*(r/critical_load_an-1)))

    # Initialize function and assign eigenvector (renormalize for plotting)
    eigenmode = Function(V,name="Eigenvector "+str(i))
    eigenmode.vector()[:] = rx/np.max(np.abs(rx.get_local()))

    plot(eigenmode.sub(0), label="Buckling mode "+str(i+1))

plt.ylim((-1.2, 1.2))
plt.legend()
plt.show()
```

```
Computing 3 first eigenvalues...
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
Critical buckling loads:
Exact:    0.31800  FE:    0.31805  Rel. gap 0.01%%
Exact:    0.93995  FE:    0.94033  Rel. gap 0.04%%
Exact:    1.87267  FE:    1.87415  Rel. gap 0.08%%
```

Above, we compared the computed FE critical loads with the known analytical value for the Euler-Bernoulli beam model and the considered boundary conditions given by:

$$F_n = (\alpha_n)^2 \frac{EI}{L^2} \quad \text{with } \alpha_n \text{ solutions to } \tan(\alpha) = \alpha$$

In particular, it can be observed that the displacement-based FE solution overestimates the exact buckling load and that the error increases with the order of the buckling load.

The corresponding files can be obtained from:

- Jupyter Notebook: `beams_3D.ipynb`
- Python script: `beams_3D.py`

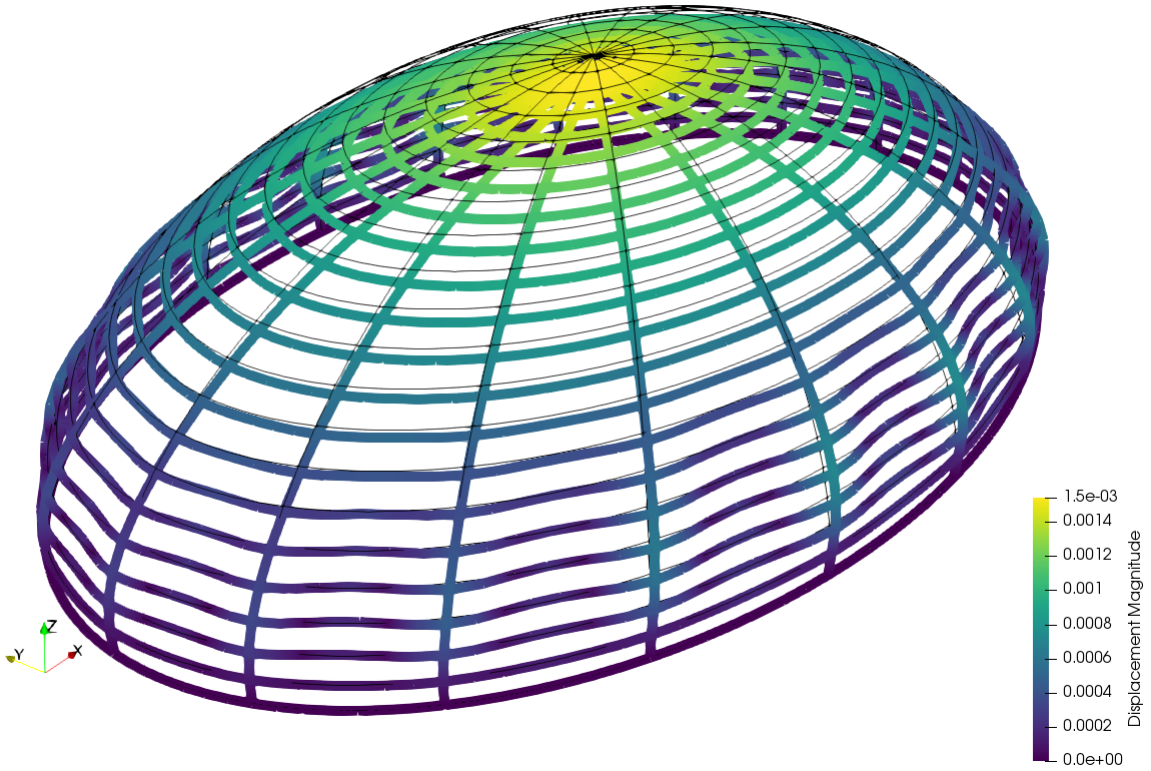
7.2 Elastic 3D beam structures

This tour explores the formulation of 3D beam-like elastic structures in FEniCS. One particularity of this tour is that the mesh topology is 1D (beams) but is embedded in a 3D ambient space. We will also show how to define a local frame containing the beam axis direction and two perpendicular directions for the definition of the cross-section geometrical properties. This tour is illustrated on a shell-like structure made of beams and is available in `.xdmf` format.

We also refer to the following tours for similar problems on beams or plates: Eulerian buckling of a beam or Reissner-Mindlin plate with Quadrilaterals

The complete source files including mesh files can be obtained from `beams_3D.zip`.

The final deformed structure (under self-weight) should look like this:



7.2.1 Variational formulation: beam kinematics and consititutive equations

The variational formulation for 3D beams requires to distinguish between motion along the beam direction and along both perpendicular directions. These two directions often correspond to principal directions of inertia and distinguish between stong and weak bending directions. The user must therefore specify this local orientation frame throughout the whole structure. In this example, we will first compute the vector t tangent to the beam direction, and two perpendicular directions a_1 and a_2. a_1 will always be perpendicular to t and the vertical direction, while a_2 will be such that (t, a_1, a_2) is a direct orthonormal frame.

In the following, we will consider a Timoshenko beam model with St-Venant uniform torsion theory for the torsional part. The beam kinematics will then be described by a 3D displacement field $\boldsymbol{u}$ and an independent 3D rotation field $\boldsymbol{\theta}$, that is 6 degrees of freedom at each point. (u_t, u_1, u_2) will respectively denote the displacement components in the section local frame (t, a_1, a_2). Similarly, θ_1 and θ_2 will denote the section rotations about the two section axis whereas θ_t is the twist angle.

The beam strains are given by (straight beam with constant local frame):

- the **normal strain**: $\delta = \dfrac{du_t}{ds}$
- the **bending curvatures**: $\chi_1 = \dfrac{d\theta_1}{ds}$ and $\chi_2 = \dfrac{d\theta_2}{ds}$
- the **shear strains**: $\gamma_1 = \dfrac{du_1}{ds} - \theta_2$ and $\gamma_2 = \dfrac{du_2}{ds} + \theta_1$.
- the **torsional strain**: $\omega = \dfrac{d\theta_t}{ds}$.

where $\dfrac{dv}{ds} = \nabla v \cdot t$ is the tangential gradient. Associated with these generalized strains are the corresponding generalized stresses:

- the **normal force** N
- the **bending moments** M_1 and M_2
- the **shear forces** Q_1 and Q_2
- the **torsional moment** M_T

The beam constitutive equations (assuming no normal force/bending moment coupling and that a_1 and a_2 are the principle axis of inertia) read as:

$$\begin{Bmatrix} N \\ Q_1 \\ Q_2 \\ M_T \\ M_1 \\ M_2 \end{Bmatrix} = \begin{bmatrix} ES & 0 & 0 & 0 & 0 & 0 \\ 0 & GS_1 & 0 & 0 & 0 & 0 \\ 0 & 0 & GS_2 & 0 & 0 & 0 \\ 0 & 0 & 0 & GJ & 0 & 0 \\ 0 & 0 & 0 & 0 & EI_1 & 0 \\ 0 & 0 & 0 & 0 & 0 & EI_2 \end{bmatrix} \begin{Bmatrix} \delta \\ \gamma_1 \\ \gamma_2 \\ \omega \\ \chi_1 \\ \chi_2 \end{Bmatrix}$$

where S is the cross-section area, E the material Young modulus and G the shear modulus, $I_1 = \int_S x_2^2 dS$ (resp. $I_2 = \int_S x_1^2 dS$) are the bending second moment of inertia about axis a_1 (resp. a_2), J is the torsion inertia, S_1 (resp. S_2) is the shear area in direction a_1 (resp. a_2).

The 3D beam variational formulation finally reads as: Find $(\boldsymbol{u}, \boldsymbol{\theta}) \in V$ such that:

$$\int_S (N\widehat{\delta} + Q_1\widehat{\gamma}_1 + Q_2\widehat{\gamma}_2 + M_T\widehat{\omega} + M_1\widehat{\chi}_1 + M_2\widehat{\chi}_2) dS = \int \boldsymbol{f} \cdot \widehat{\boldsymbol{u}} dS \quad \forall (\widehat{\boldsymbol{u}}, \widehat{\boldsymbol{\theta}}) \in V$$

where we considered only a distributed loading $\boldsymbol{f}$ and where $\widehat{\delta}, \ldots, \widehat{\chi}_2$ are the generalized strains associated with test functions $(\widehat{\boldsymbol{u}}, \widehat{\boldsymbol{\theta}})$.

7.2.2 FEniCS implementation

After first loading the shell mesh file in `.xdmf` format, we will use the UFL `Jacobian` function which computes the transformation Jacobian between a reference element (interval here) and the current element. For the present case, the Jacobian is of shape (3,1). Transforming it into a vector of unit length will give us the local tangent vector t.

[1]:
```
from dolfin import *
from ufl import Jacobian, diag
from plotting import plot

mesh = Mesh()
filename = "shell.xdmf"
f = XDMFFile(filename)
f.read(mesh)

def tangent(mesh):
    t = Jacobian(mesh)
    return as_vector([t[0,0], t[1, 0], t[2, 0]])/sqrt(inner(t,t))

t = tangent(mesh)
```

We now compute the section local axis. As mentioned earlier, a_1 will be perpendicular to t and the vertical direction $e_z = (0, 0, 1)$. After normalization, a_2 is built by taking the cross product between t and a_1, a_2 will therefore belong to the plane made by t and the vertical direction.

[2]:
```
ez = as_vector([0, 0, 1])
a1 = cross(t, ez)
a1 /= sqrt(dot(a1, a1))
a2 = cross(t, a1)
a2 /= sqrt(dot(a2, a2))
```

We now define the material and geometrical constants which will be used in the constitutive relation. We consider the case of a rectangular cross-section of width b and height h in directions a_1 and a_2. The bending inertia will therefore be $I_1 = bh^3/12$ and $I_2 = hb^3/12$. The torsional inertia is $J = \beta hb^3$ with $\beta \approx 0.26$ for $h = 3b$. Finally, the shear areas are approximated by $S_1 = S_2 = \kappa S$ with $\kappa = 5/6$.

[3]:
```
thick = Constant(0.3)
width = thick/3
E = Constant(70e3)
nu = Constant(0.3)
G = E/2/(1+nu)
rho = Constant(2.7e-3)
g = Constant(9.81)

S = thick*width
ES = E*S
EI1 = E*width*thick**3/12
EI2 = E*width**3*thick/12
GJ = G*0.26*thick*width**3
kappa = Constant(5./6.)
GS1 = kappa*G*S
GS2 = kappa*G*S
```

We now consider a mixed $\mathbb{P}_1/\mathbb{P}_1$-Lagrange interpolation for the displacement and rotation fields. The variational form is built using a function `generalized_strains` giving the vector of six generalized strains as well as a function `generalized_stresses` which computes the dot product of the strains with the above-mentioned constitutive matrix (diagonal here). Note that since the 1D beams are embedded in an ambient 3D space, the gradient operator has shape (3,), we therefore define a tangential gradient operator `tgrad` by taking the dot product with the local tangent vector t.

Finally, similarly to Reissner-Mindlin plates, shear-locking issues might arise in the thin beam limit. To avoid this, reduced integration is performed on the shear part $Q_1\widehat{\gamma}_1 + Q_2\widehat{\gamma}_2$ of the variational form using a one-point rule.

[6]:
```
Ue = VectorElement("CG", mesh.ufl_cell(), 1, dim=3)
W = FunctionSpace(mesh, Ue*Ue)

u_ = TestFunction(W)
du = TrialFunction(W)
(w_, theta_) = split(u_)
(dw, dtheta) = split(du)

def tgrad(u):
    return dot(grad(u), t)
def generalized_strains(u):
    (w, theta) = split(u)
    return as_vector([dot(tgrad(w), t),
                      dot(tgrad(w), a1)-dot(theta, a2),
                      dot(tgrad(w), a2)+dot(theta, a1),
                      dot(tgrad(theta), t),
                      dot(tgrad(theta), a1),
                      dot(tgrad(theta), a2)])
def generalized_stresses(u):
```

(continues on next page)

(continued from previous page)

```
    return dot(diag(as_vector([ES, GS1, GS2, GJ, EI1, EI2])), generalized_strains(u))

Sig = generalized_stresses(du)
Eps =  generalized_strains(u_)

dx_shear = dx(scheme="default",metadata={"quadrature_scheme":"default", "quadrature_
↪degree": 1})
k_form = sum([Sig[i]*Eps[i]*dx for i in [0, 3, 4, 5]]) +␣
↪(Sig[1]*Eps[1]+Sig[2]*Eps[2])*dx_shear
l_form = Constant(-rho*S*g)*w_[2]*dx
```

```
Calling FFC just-in-time (JIT) compiler, this may take some time.
Calling FFC just-in-time (JIT) compiler, this may take some time.
Calling FFC just-in-time (JIT) compiler, this may take some time.
```

Clamped boundary conditions are considered at the bottom $z = 0$ level and the linear problem is finally solved.

[7]:
```
def bottom(x, on_boundary):
    return near(x[2], 0.)
bc = DirichletBC(W, Constant((0, 0, 0, 0, 0, 0)), bottom)

u = Function(W)
solve(k_form == l_form, u, bc)
```

```
Calling FFC just-in-time (JIT) compiler, this may take some time.
Calling FFC just-in-time (JIT) compiler, this may take some time.
Calling FFC just-in-time (JIT) compiler, this may take some time.
Calling FFC just-in-time (JIT) compiler, this may take some time.
```

Matplotlib functions cannot plot functions defined on a 1D mesh embedded in a 3D space so that we output the solution fields to XDMF format for vizualisation with Paraview for instance. We also export the bending moments by projecting them on a suitable function space.

[8]:
```
ffile = XDMFFile("shell-results.xdmf")
ffile.parameters["functions_share_mesh"] = True
v = u.sub(0, True)
v.rename("Displacement", "")
ffile.write(v, 0.)
theta = u.sub(1, True)
theta.rename("Rotation", "")
ffile.write(theta, 0.)

V1 = VectorFunctionSpace(mesh, "CG", 1, dim=2)
M = Function(V1, name="Bending moments (M1,M2)")
Sig = generalized_stresses(u)
M.assign(project(as_vector([Sig[4], Sig[5]]), V1))
ffile.write(M, 0)

ffile.close()
```

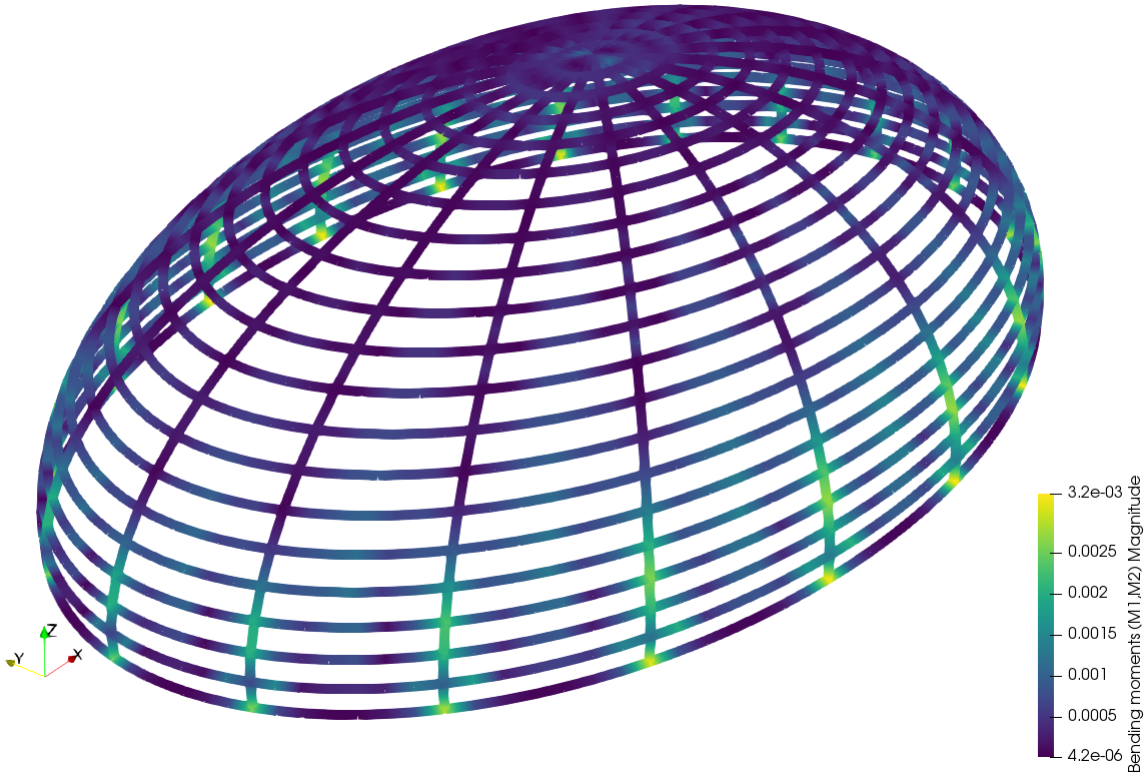
```
Calling FFC just-in-time (JIT) compiler, this may take some time.
Calling FFC just-in-time (JIT) compiler, this may take some time.
Calling FFC just-in-time (JIT) compiler, this may take some time.
Calling FFC just-in-time (JIT) compiler, this may take some time.
Calling FFC just-in-time (JIT) compiler, this may take some time.
Calling FFC just-in-time (JIT) compiler, this may take some time.
Calling FFC just-in-time (JIT) compiler, this may take some time.
Calling FFC just-in-time (JIT) compiler, this may take some time.
```

(continues on next page)

(continued from previous page)

```
Calling FFC just-in-time (JIT) compiler, this may take some time.
```

Bending moment amplitude over the structure would look like this:



CHAPTER 8

Plates

Contents:

8.1 Reissner-Mindlin plate with Quadrilaterals

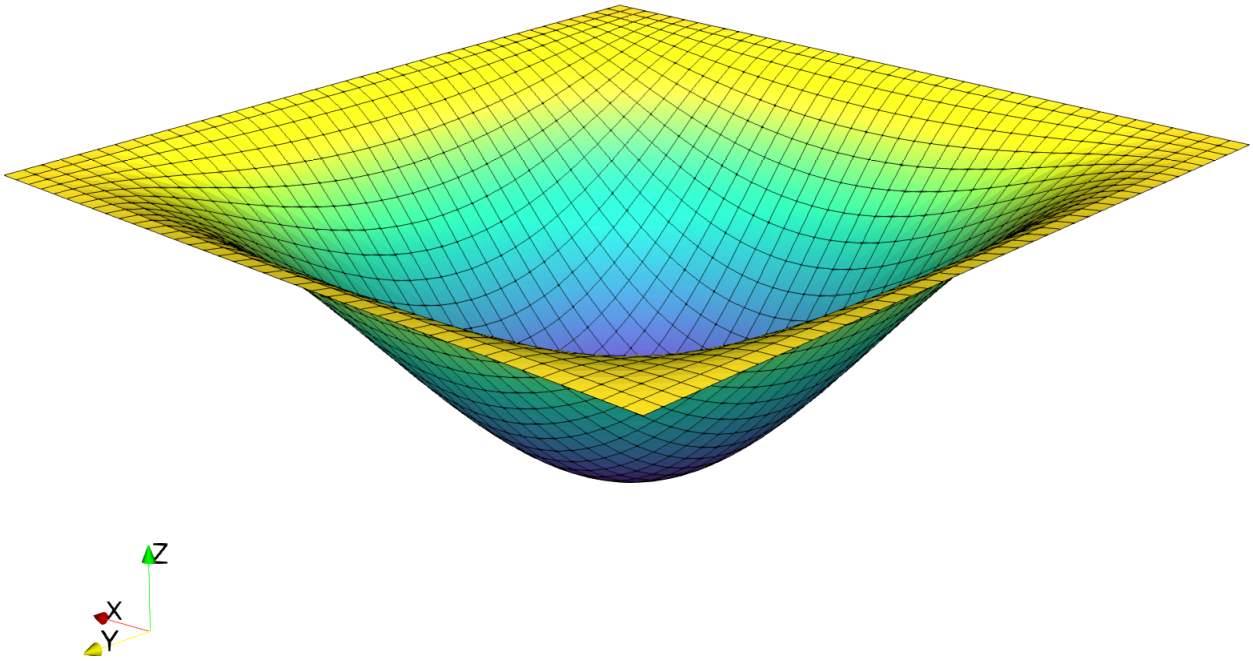
8.1.1 Introduction

This program solves the Reissner-Mindlin plate equations on the unit square with uniform transverse loading and fully clamped boundary conditions. The corresponding file can be obtained from `reissner_mindlin_quads.py`.

It uses quadrilateral cells and selective reduced integration (SRI) to remove shear-locking issues in the thin plate limit. Both linear and quadratic interpolation are considered for the transverse deflection w and rotation $\underline{\theta}$.

Note: Note that for a structured square grid such as this example, quadratic quadrangles will not exhibit shear locking because of the strong symmetry (similar to the criss-crossed configuration which does not lock). However, perturbating the mesh coordinates to generate skewed elements suffice to exhibit shear locking.

The solution for w in this demo will look as follows:



8.1.2 Implementation

Material parameters for isotropic linear elastic behavior are first defined:

```
from dolfin import *

E = Constant(1e3)
nu = Constant(0.3)
```

Plate bending stiffness $D = \dfrac{Eh^3}{12(1-\nu^2)}$ and shear stiffness $F = \kappa G h$ with a shear correction factor $\kappa = 5/6$ for a homogeneous plate of thickness h:

```
thick = Constant(1e-3)
D = E*thick**3/(1-nu**2)/12.
F = E/2/(1+nu)*thick*5./6.
```

The uniform loading f is scaled by the plate thickness so that the deflection converges to a constant value in the thin plate Love-Kirchhoff limit:

```
f = Constant(-thick**3)
```

The unit square mesh is divided in $N \times N$ quadrilaterals:

```
N = 50
mesh = UnitSquareMesh.create(N, N, CellType.Type.quadrilateral)
```

Continuous interpolation using of degree $d = \texttt{deg}$ is chosen for both deflection and rotation:

```
deg = 1
We = FiniteElement("Lagrange", mesh.ufl_cell(), deg)
Te = VectorElement("Lagrange", mesh.ufl_cell(), deg)
V = FunctionSpace(mesh, MixedElement([We, Te]))
```

Clamped boundary conditions on the lateral boundary are defined as:

```
def border(x, on_boundary):
    return on_boundary

bc =  [DirichletBC(V, Constant((0., 0., 0.)), border)]
```

Some useful functions for implementing generalized constitutive relations are now defined:

```
def strain2voigt(eps):
    return as_vector([eps[0, 0], eps[1, 1], 2*eps[0, 1]])
def voigt2stress(S):
    return as_tensor([[S[0], S[2]], [S[2], S[1]]])
def curv(u):
    (w, theta) = split(u)
    return sym(grad(theta))
def shear_strain(u):
    (w, theta) = split(u)
    return theta-grad(w)
def bending_moment(u):
    DD = as_tensor([[D, nu*D, 0], [nu*D, D, 0],[0, 0, D*(1-nu)/2.]])
    return voigt2stress(dot(DD,strain2voigt(curv(u))))
def shear_force(u):
    return F*shear_strain(u)
```

The contribution of shear forces to the total energy is under-integrated using a custom quadrature rule of degree $2d-2$ i.e. for linear ($d=1$) quadrilaterals, the shear energy is integrated as if it were constant (1 Gauss point instead of 2x2) and for quadratic ($d=2$) quadrilaterals, as if it were quadratic (2x2 Gauss points instead of 3x3):

```
u = Function(V)
u_ = TestFunction(V)
du = TrialFunction(V)

dx_shear = dx(metadata={"quadrature_degree": 2*deg-2})

L = f*u_[0]*dx
a = inner(bending_moment(u_), curv(du))*dx + dot(shear_force(u_), shear_
↪strain(du))*dx_shear
```

We then solve for the solution and export the relevant fields to XDMF files

```
solve(a == L, u, bc)

(w, theta) = split(u)

Vw = FunctionSpace(mesh, We)
Vt = FunctionSpace(mesh, Te)
ww = u.sub(0, True)
ww.rename("Deflection", "")
tt = u.sub(1, True)
tt.rename("Rotation", "")
```

(continues on next page)

(continued from previous page)

```
file_results = XDMFFile("RM_results.xdmf")
file_results.parameters["flush_output"] = True
file_results.parameters["functions_share_mesh"] = True
file_results.write(ww, 0.)
file_results.write(tt, 0.)
```

The solution is compared to the Kirchhoff analytical solution:

```
print("Kirchhoff deflection:", -1.265319087e-3*float(f/D))
print("Reissner-Mindlin FE deflection:", -min(ww.vector().get_local())) # point␣
↪evaluation for quads
                                                                        # is not␣
↪implemented in fenics 2017.2
```

For $h = 0.001$ and 50 quads per side, one finds $w_{FE} = 1.38182\text{e-}5$ for linear quads and $w_{FE} = 1.38176\text{e-}5$ for quadratic quads against $w_{\text{Kirchhoff}} = 1.38173\text{e-}5$ for the thin plate solution.

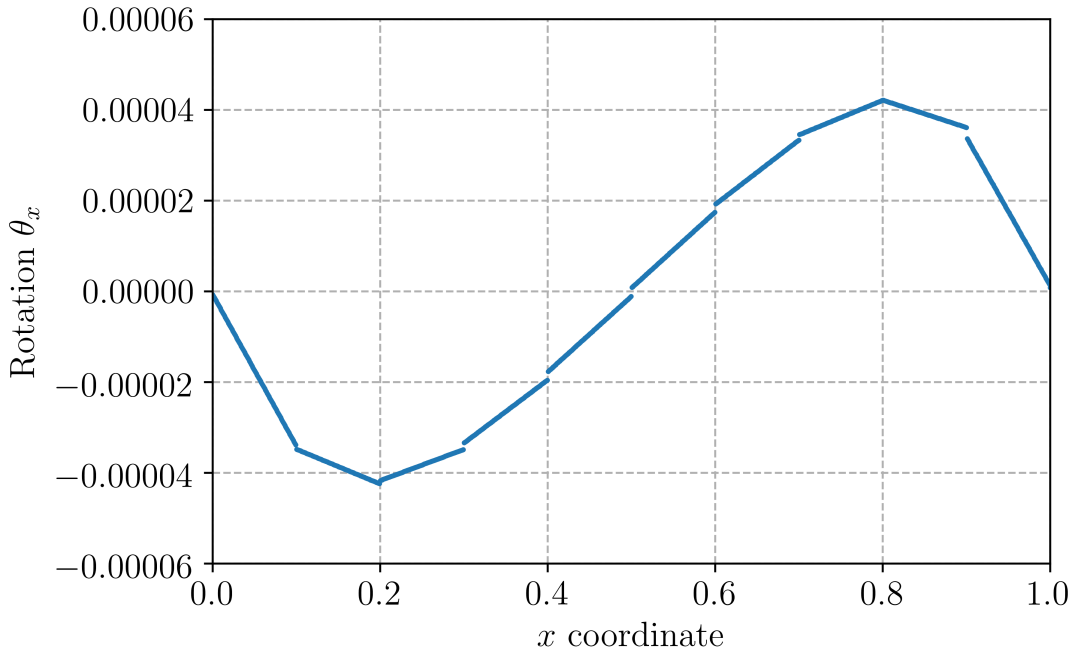
8.2 Reissner-Mindlin plate with a Discontinuous-Galerkin approach

8.2.1 Introduction

This program solves the Reissner-Mindlin plate equations on the unit square with uniform transverse loading and clamped boundary conditions. The corresponding file can be obtained from `reissner_mindlin_dg.py`.

It uses a Discontinuous Galerkin interpolation for the rotation field to remove shear-locking issues in the thin plate limit. Details of the formulation can be found in [HAN2011].

The solution for θ_x on the middle line of equation $y = 0.5$ will look as follows for 10 elements and a stabilization parameter $s = 1$:



8.2.2 Implementation

Material properties and loading are the same as in *Reissner-Mindlin plate with Quadrilaterals*:

```
from dolfin import *

E = Constant(1e3)
nu = Constant(0.3)
thick = Constant(1e-2)
D = E*thick**3/(1-nu**2)/12.
F = E/2/(1+nu)*thick*5./6.
f = Constant(-thick**3)
```

The unit square mesh is here divided in triangles and we get the facet MeshFunction for the integration measure $\mathrm{d}s$:

```
N = 40
mesh = UnitSquareMesh(N, N)
facets = MeshFunction("size_t", mesh, 1)
facets.set_all(0)
ds = Measure("ds", subdomain_data=facets)
```

Continuous interpolation using of degree 2 is chosen for the deflection w whereas the rotation field $\underline{\theta}$ is discretized using discontinuous linear polynomials:

```
We = FiniteElement("Lagrange", mesh.ufl_cell(), 2)
Te = VectorElement("DG", mesh.ufl_cell(), 1)
V = FunctionSpace(mesh, MixedElement([We, Te]))
```

Clamped boundary conditions on the lateral boundary are defined as:

```
def border(x, on_boundary):
    return on_boundary

bc =  [DirichletBC(V.sub(0), Constant(0.), border)]
```

Standard part of the variational form is the same (without full integration):

```
def strain2voigt(eps):
    return as_vector([eps[0, 0], eps[1, 1], 2*eps[0, 1]])
def voigt2stress(S):
    return as_tensor([[S[0], S[2]], [S[2], S[1]]])
def curv(u):
    (w, theta) = split(u)
    return sym(grad(theta))
def shear_strain(u):
    (w, theta) = split(u)
    return theta-grad(w)
def bending_moment(u):
    DD = as_tensor([[D, nu*D, 0], [nu*D, D, 0],[0, 0, D*(1-nu)/2.]])
    return voigt2stress(dot(DD,strain2voigt(curv(u))))
def shear_force(u):
    return F*shear_strain(u)

u = Function(V)
u_ = TestFunction(V)
du = TrialFunction(V)
```

(continues on next page)

(continued from previous page)

```
L = f*u_[0]*dx
a = inner(bending_moment(u_), curv(du))*dx + dot(shear_force(u_), shear_strain(du))*dx
```

We then add the contribution of jumps in rotation across all internal facets plus a stabilization term involing a user-defined parameter s:

```
n = FacetNormal(mesh)
h = CellVolume(mesh)
h_avg = (h('+')+h('-'))/2
stabilization = Constant(10.)

(dw, dtheta) = split(du)
(w_, theta_) = split(u_)

a -= dot(avg(dot(bending_moment(u_), n)), jump(dtheta))*dS + dot(avg(dot(bending_
↪moment(du), n)), jump(theta_))*dS \
   - stabilization*D/h_avg*dot(jump(theta_), jump(dtheta))*dS
```

Because of the clamped boundary conditions, we also need to add the corresponding contributions of the external facets (the imposed rotation is zero on the boundary so that no term arise in the linear functional):

```
a -= dot(dot(bending_moment(u_), n), dtheta)*ds + dot(dot(bending_moment(du), n),␣
↪theta_)*ds \
   - 2*stabilization*D/h*dot(theta_, dtheta)*ds
```

We then solve for the solution and export the relevant fields to XDMF files

```
solve(a == L, u, bc)

(w, theta) = split(u)

Vw = FunctionSpace(mesh, We)
Vt = FunctionSpace(mesh, Te)
ww = u.sub(0, True)
ww.rename("Deflection", "")
tt = u.sub(1, True)
tt.rename("Rotation", "")

file_results = XDMFFile("RM_DG_results.xdmf")
file_results.parameters["flush_output"] = True
file_results.parameters["functions_share_mesh"] = True
file_results.write(ww, 0.)
file_results.write(tt, 0.)
```

The solution is compared to the Kirchhoff analytical solution:

```
print("Kirchhoff deflection:", -1.265319087e-3*float(f/D))
print("Reissner-Mindlin FE deflection:", -ww(0.5, 0.5))
```

For $h = 0.001$ and 50 elements per side, one finds $w_{FE} = 1.38322\text{e-}5$ against $w_{\text{Kirchhoff}} = 1.38173\text{e-}5$ for the thin plate solution.

8.2.3 References

The corresponding files can be obtained from:

- Jupyter Notebook: `reissner_mindlin_CR.ipynb`
- Python script: `reissner_mindlin_CR.py`

8.3 Locking-free Reissner-Mindlin plate with Crouzeix-Raviart interpolation

This tour explores a new discretization strategy for Reissner-Mindlin plates. It is known that such thick plate discretizations face the shear-locking issue in the thin plate limit. Different strategies can be adopted to alleviate this issue such as:

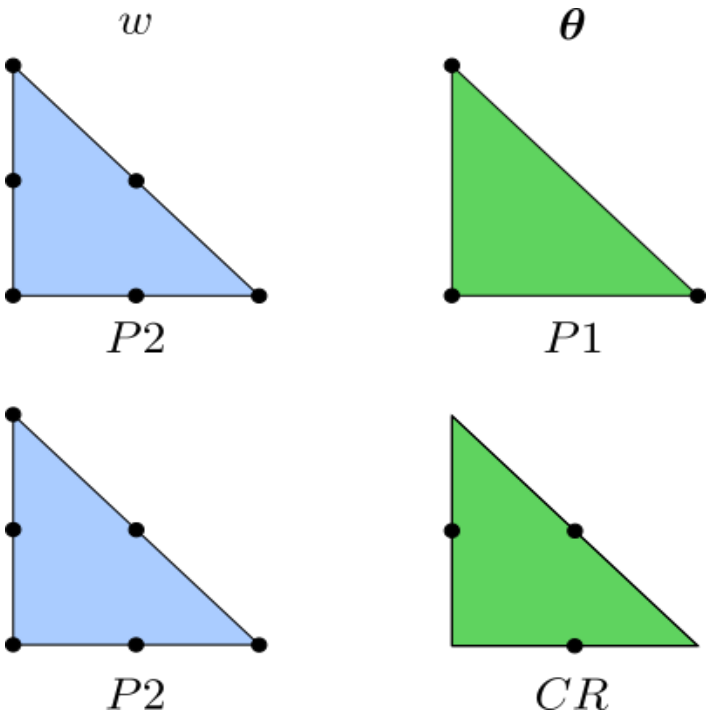
- reduced integration (see Reissner-Mindlin plate with Quadrilaterals)
- Discontinuous Galerkin methods (see Reissner-Mindlin plate with a Discontinuous-Galerkin approach)
- mixed methods (see the `fenics-shells` demo using the Duran-Liberman element)
- and many more...

Often such methods can be quite cumbersome to implement, even more with FEniCS. In this tour, we investigate another discretization which does not seem to have been widely studied in the literature, except in *[CAM03]* and subsequent works from the same group of authors where it has been used for a shell element. Note that Crouzeix-Raviart elements have already been used in Reissner-Mindlin plate models such as in *[ARN89]* but with a different discretization strategy than the present one.

The discretization that we will explore is extremely simple:

- P^2 Lagrange continuous interpolation for the plate deflection w
- Crouzeix-Raviart (CR) interpolation for the rotation vector $\boldsymbol{\theta}$

We recall that the CR interpolation corresponds to a piecewise linear interpolation with continuity satisfied only at the mid-points of the element edges. In this sense, the element is non-conforming, akin to the DG approach mentioned before. However, the rotation field is not fully discontinuous and it seems that numerical stabilization of jump terms as in DG methods is not necessary. In the following, we will also compare the P2/CR interpolation results with a continuous P2/P1 interpolation which is known to exhibit shear-locking without any specific treatment.



8.3.1 FEniCS implementation

The implementation is very similar to the previously mentioned tours except that no reduced integration or jump terms have to be included in the formulation. Again, we will investigate a square plate under uniform loading and clamped supports.

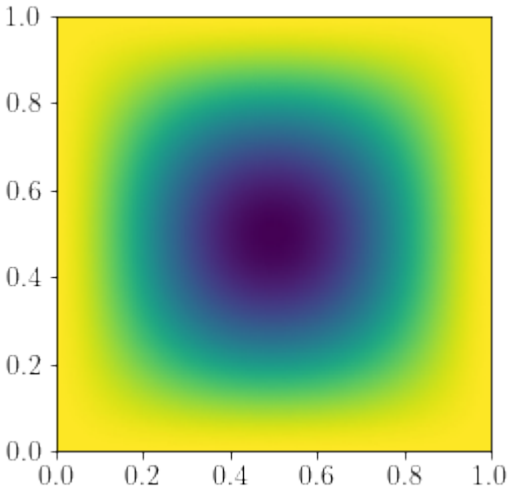
The complete source files including convergence results can be obtained from `reissner_mindlin_CR.zip`.

[1]:
```
from dolfin import *
import matplotlib.pyplot as plt

# material parameters
thick = Constant(1e-3)
E = Constant(10)
nu = Constant(0.3)

# bending stiffness
D = E*thick**3/(1-nu**2)/12.
# shear stiffness
F = E/2/(1+nu)*thick*5./6.

# uniform transversal load
f = Constant(-1e3*thick**3)

# Useful function for defining strain energy
def strain2voigt(eps):
    return as_vector([eps[0, 0], eps[1, 1], 2*eps[0, 1]])
def voigt2stress(S):
    return as_tensor([[S[0], S[2]], [S[2], S[1]]])
def curv(u):
    (w, theta) = split(u)
    return sym(grad(theta))
```

(continues on next page)

(continued from previous page)

```
def shear_strain(u):
    (w, theta) = split(u)
    return theta-grad(w)
def bending_moment(u):
    DD = as_tensor([[D, nu*D, 0], [nu*D, D, 0],[0, 0, D*(1-nu)/2.]])
    return voigt2stress(dot(DD,strain2voigt(curv(u))))
def shear_force(u):
    return F*shear_strain(u)


# Unit square mesh NxN elements
N = 100
mesh = UnitSquareMesh(N, N, "crossed")

# Definition of function space for U:displacement, T:rotation
Ue = FiniteElement("Lagrange", mesh.ufl_cell(), 2)
Te = VectorElement("CR", mesh.ufl_cell(), 1)
V = FunctionSpace(mesh, MixedElement([Ue, Te]))

# Definition of boundary conditions
def border(x, on_boundary):
    return on_boundary
bc =  [DirichletBC(V,Constant((0.,)*3), border)]


# Functions
u = Function(V)
(w,theta) = split(u)
u_ = TestFunction(V)
du = TrialFunction(V)

# Linear and bilinear forms
L = f*u_[0]*dx
a = inner(bending_moment(u_),curv(du))*dx + dot(shear_force(u_),shear_strain(du))*dx

# Solver
solve(a == L, u, bc, solver_parameters={"linear_solver":"mumps"})

# Post-process
#(w,theta) = split(u)
w,theta = u.split(deepcopy=True)
plot(w, mode="color")
plt.show()

w_LK = 1.265319087e-3*float(-f/D)
print("Reissner-Mindlin deflection:", -w(0.5, 0.5))
print("Love-Kirchhoff deflection:  ", w_LK)
```



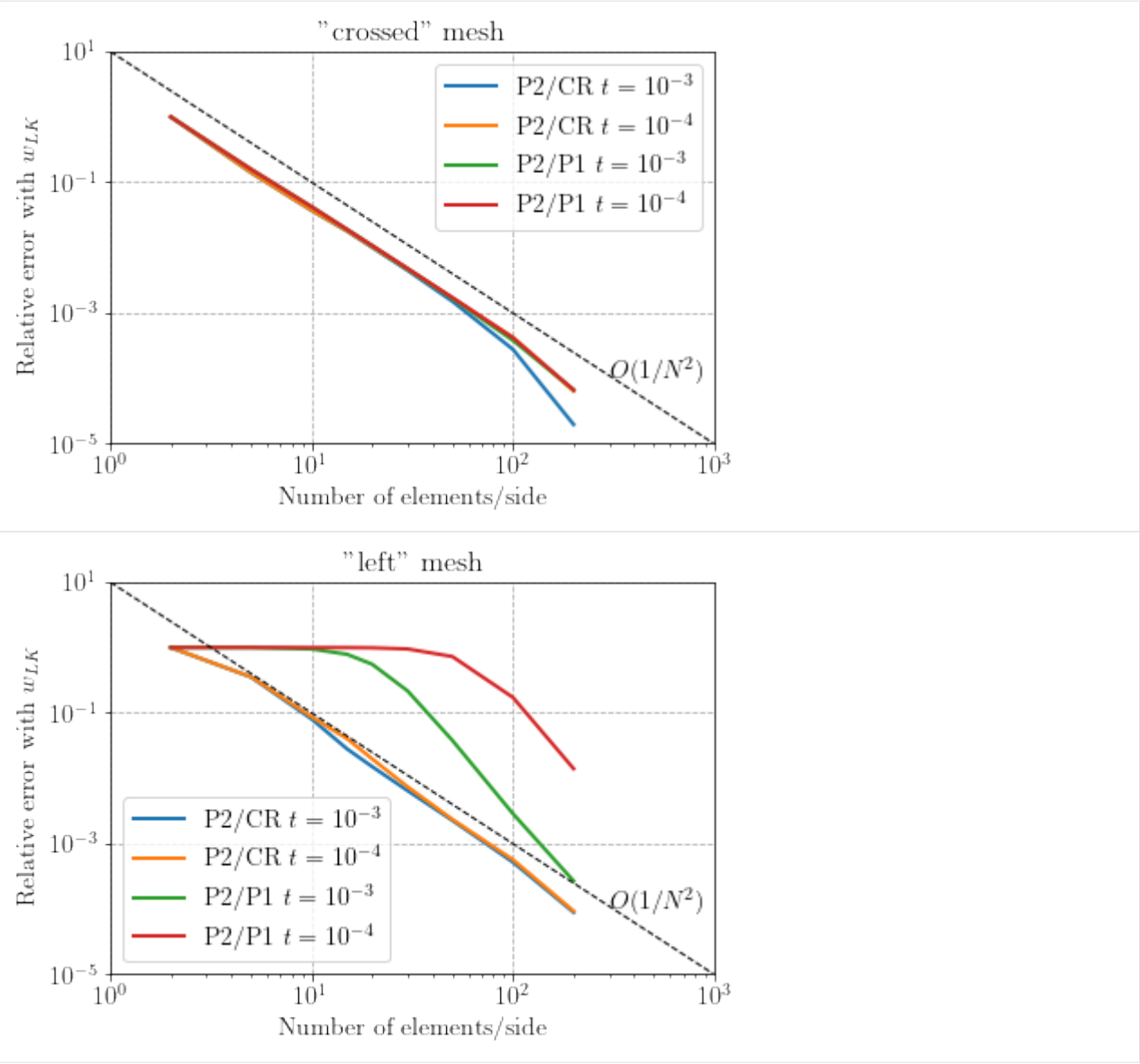

```
Reissner-Mindlin deflection: 1.381343203499173
Love-Kirchhoff deflection:   1.3817284430040004
```

8.3.2 Convergence results

We compare the convergence behaviour of the `P2/CR` vs a standard `P2/P1` discretization, on either a `"crossed"` or a `"left"` mesh.

[2]:
```
import numpy as np

labels = [r"P2/CR $t=10^{-3}$", r"P2/CR $t=10^{-4}$",
          r"P2/P1 $t=10^{-3}$", r"P2/P1 $t=10^{-4}$"]
for mesh_type in ["crossed", "left"]:
    data = np.loadtxt("reissner_mindlin_CR_{}.csv".format(mesh_type),
                      delimiter=",", skiprows=1)
    N = data[:, 0]
    res = np.abs(data[:, 1:]/w_LK-1)
    for (i, label) in enumerate(labels):
        plt.loglog(N, res[:, i], label=label)
    plt.loglog([1, 1e3], [1e1, 1e-5], "--k", linewidth=1)
    plt.annotate("$O(1/N^2$)", xy=[3e2, 1e-4], fontsize=14)
    plt.legend()
    plt.ylim(1e-5, 1e1)
    plt.title("\"{}\" mesh".format(mesh_type), fontsize=16)
    plt.xlabel("Number of elements/side")
    plt.ylabel("Relative error with $w_{LK}$")
    plt.show()
```

```
findfont: Font family ['serif'] not found. Falling back to DejaVu Sans.
```



As expected, the `P2/P1` discretization locks in the thin plate limit for the `"left"`, whereas locking does not appear for the `"crossed"` mesh. On the contrary the `P2/CR` discretization does not lock and exhibits the same quadratic convergence for both types of mesh.

8.3.3 References

[ARN89] Arnold, D. N., & Falk, R. S. (1989). A uniformly accurate finite element method for the Reissner–Mindlin plate. SIAM Journal on Numerical Analysis, 26(6), 1276-1290.

[CAM03] Campello, E. M. B., Pimenta, P. M., & Wriggers, P. (2003). A triangular finite shell element based on a fully nonlinear shell formulation. Computational Mechanics, 31(6), 505-518.

CHAPTER 9

Topology optimization of structures

Contents:

The corresponding files can be obtained from:

- Jupyter Notebook: `simp_topology_optimization.ipynb`
- Python script: `simp_topology_optimization.py`

9.1 Topology optimization using the SIMP method

This numerical tour investigates the optimal design of an elastic structure through a topology optimization approach. It consists in finding the optimal distribution of a material in a computational domain which minimizes the compliance (or, equivalently, maximizes the stiffness) of the resulting structure under a fixed volume fraction constraint.

For instance, in the bottom animation, the optimal material density (a continuous field $\theta(x) \in [0; 1]$) of a cantilever beam evolves with the topoology optimization algorithm iterations. The first 20 iterations correspond to a "non-penalized" distribution where intermediate densities (i.e. $0 < \theta(x) < 1$) are allowed. However, such a *gray-level* distribution is hard to realize in practice and one would like to obtain a *black-and-white* design of material/void distribution. This is what is achieved in the last iterations where some penalty is applied to intermediate densities, exhibiting finally a truss-like design. Note that the final design is not a global optimum but only a local one. Hence, different designs can well be obtained if changing the initial conditions or the mesh density for instance.

The present tour relies on one of the most widely implemented approach in topology optimization, namely the Solid Isotropic Material with Penalisation (SIMP) method developed by Bendsoe and Sigmund *[BEN03]*. See also the monograph *[ALL07]* and Gregoire Allaire's class on *Optimal Design of Structures* at Ecole Polytechnique. In particular, this tour refers to lesson 8 and is the direct adaptation of the SIMP Topology Optimization example of the CMAP group toolbox written in Freefem++.

This tour should also work in parallel.

9.1.1 Theory

Compliance optimization on binary shapes

Let us consider a computational domain D in which we seek an elastic material domain Ω ($D \setminus \Omega$ representing void) so that the compliance under prescribed loading is minimal when subjected to a fixed volume constraint $\text{Vol}(\Omega) = \eta \text{Vol}(D)$. Replacing the shape Ω by its characteristic function $\chi(x) \in \{0,1\}$, the previous problem can be formulated as the following optimization problem:

$$\begin{array}{ll} \min\limits_{\chi} & \displaystyle\int_D \boldsymbol{f} \cdot \boldsymbol{u}(\chi) \,\text{dx} + \int_{\Gamma_N} \boldsymbol{T} \cdot \boldsymbol{u}(\chi) \,\text{dS} \\ \text{s.t.} & \int_D \chi(x) \,\text{dx} = \eta \text{Vol}(D) \\ & \chi(x) \in \{0,1\} \end{array}$$

where $\boldsymbol{u}(\chi)$ is the solution of the following $\chi(x)-$dependent elasticity problem:

$$\begin{aligned} \text{div}\,\boldsymbol{\sigma} + \boldsymbol{f} &= 0 \quad \text{in } D \qquad (9.1) \\ \boldsymbol{\sigma} &= \chi(x)\mathbb{C} : \nabla^s \boldsymbol{u} \quad \text{in } D \qquad (9.2) \\ \boldsymbol{\sigma} \cdot \boldsymbol{n} &= \boldsymbol{T} \quad \text{on } \Gamma_N \qquad (9.3) \\ \boldsymbol{u} &= 0 \quad \text{on } \Gamma_D \qquad (9.4) \end{aligned}$$

where $\boldsymbol{f}$ is a body force, $\boldsymbol{T}$ corresponds to surface tractions on Γ_N and $\mathbb{C}$ is the elastic material stiffness tensor. Owing to the elastic nature of the problem, the compliance can also be expressed using the elastic stress energy density:

$$\int_D \boldsymbol{f} \cdot \boldsymbol{u}(\chi) \,\text{dx} + \int_{\Gamma_N} \boldsymbol{T} \cdot \boldsymbol{u}(\chi) \,\text{dS} = \int_D \boldsymbol{\sigma}(\chi) : (\chi(x)\mathbb{C})^{-1} : \boldsymbol{\sigma}(\chi) \,\text{dx} = \int_D \nabla^s \boldsymbol{u}(\chi) : (\chi(x)\mathbb{C}) : \nabla^s \boldsymbol{u}(\chi) \,\text{dx}$$

so that the above optimization problem can be reformulated as:

$$\begin{array}{ll} \min\limits_{\chi, \boldsymbol{u}} & \displaystyle\int_D \boldsymbol{\sigma}(\chi) : (\chi(x)\mathbb{C})^{-1} : \boldsymbol{\sigma}(\chi) \,\text{dx} \\ \text{s.t.} & \text{div}\,\boldsymbol{\sigma} + \boldsymbol{f} = 0 \quad \text{in } D \\ & \boldsymbol{\sigma} \cdot \boldsymbol{n} = \boldsymbol{T} \quad \text{on } \Gamma_N \\ & \int_D \chi(x) \,\text{dx} = \eta \text{Vol}(D) \\ & \chi(x) \in \{0,1\} \end{array}$$

Continuous relaxation and SIMP penalization

The binary constraint $\chi(x) \in \{0,1\}$ makes the above problem extremely difficult to solve so that a classical remedy consists in relaxing the constraint by a continuous constraint $\theta(x) \in [0,1]$ where $\theta(x)$ will approximate the characteristic function $\chi(x)$ by a gray-level continuous function taking values between 0 and 1.

However, in order to still obtain a final binary density distribution, the binary modulus $\chi(x)\mathbb{C}$ will be replaced by $\theta(x)^p\mathbb{C}$ with $p > 1$ in order to penalize intermediate densities, yielding the so-called *SIMP formulation*:

$$\begin{array}{lll} \min\limits_{\theta, \boldsymbol{\sigma}} J_p(\theta, \boldsymbol{\sigma}) = \min\limits_{\theta, \boldsymbol{\sigma}} & = \displaystyle\int_D \boldsymbol{\sigma} : (\theta(x)^{-p}\mathbb{C}^{-1}) : \boldsymbol{\sigma} \,\text{dx} \\ \text{s.t.} & \text{div}\,\boldsymbol{\sigma} + \boldsymbol{f} = 0 \quad \text{in } D \\ & \boldsymbol{\sigma} \cdot \boldsymbol{n} = \boldsymbol{T} \quad \text{on } \Gamma_N \\ & \int_D \theta(x) \,\text{dx} = \eta \text{Vol}(D) \\ & 0 \leq \theta(x) \leq 1 \end{array}$$

Optimization of the SIMP formulation

Unfortunately jointly minimizing J_p over $(\theta, \boldsymbol{\sigma})$ is hard to do in practice since this functional is non-convex (except if $p = 1$). However, it is convex over each variable θ and $\boldsymbol{\sigma}$ when fixing the other variable. This makes alternate

minimization an attractive method for finding a local optimum. Besides, minimizing directly for a fixed value of $p > 1$ does not work well in practice. A better solution consists in performing alternate minimization steps and progressively increase p from 1 to a maximum value (typically 3 or 4) using some heuristic (see later).

Iteration $n + 1$ of the algorithm, knowing a previous pair $(\theta_n, \boldsymbol{\sigma}_n)$ and a given value of the penalty exponent p_n, therefore consists in:

- minimizing $J_{p_n}(\theta_n, \boldsymbol{\sigma})$ yielding $\boldsymbol{\sigma}_{n+1} = (\theta_n)^{p_n}\mathbb{C} : \nabla^s \boldsymbol{u}_{n+1}$
- minimizing $J_{p_n}(\theta, \boldsymbol{\sigma}_{n+1})$ yielding θ_{n+1}
- potentially update the value of the exponent using some heuristic $p_n \to p_{n+1}$

Both minimization problems are quite easy to solve since the first one is nothing else than a heterogeneous elastic problem with some known modulus $(\theta_n(x))^{p_n}\mathbb{C}$.

The second problem can be rewritten using the Lagrange multiplier λ corresponding to the toal volume constraint, as follows:

$$\min_{\theta\in[0;1]} \int_D \theta(x)^{-p_n} e(\boldsymbol{\sigma}_{n+1}) \,\mathrm{dx} + \lambda\left(\int_D \theta(x)\,\mathrm{dx} - \eta \mathrm{Vol}(D)\right)$$

where $e(\sigma_{n+1}) = \boldsymbol{\sigma}_{n+1} : \mathbb{C}^{-1} : \boldsymbol{\sigma}_{n+1} = (\theta_n)^{p_n}\boldsymbol{\sigma}_{n+1} : \nabla^s \boldsymbol{u}_{n+1}$. The optimality conditions for this problem yields the following explicit and local condition:

$$-p_n \theta_{n+1}^{-p_n-1} e(\sigma_{n+1}) + \lambda = 0$$

which along with the $[\theta_{min}; 1]$ constraint gives:

$$\theta_{n+1} = \min\left\{1; \max\left\{\theta_{min}; \left(\frac{p_n e(\sigma_{n+1})}{\lambda}\right)^{1/(p_n+1)}\right\}\right\}$$

where we replaced the 0 constraint by a minimum density value $\theta_{min} > 0$ to avoid degeneracy issue with void material.

Note that the above expression assumes that λ is known. Its value is found by satsifying the volume constraint $\int_D \theta(x)\,\mathrm{dx} = \eta \mathrm{Vol}(D)$.

9.1.2 FEniCS implementation

We first define some parameters of the algorithm. The most important ones concern the number of total alternate minimizations (AM) `niter` and the parameters controlling the exponent update strategy:

- `nitern`p corresponds to the number of first (AM) for which $p = 1$. These are non-penalized iterations yielding a diffuse gray-level field $\theta(x)$ at convergence (note that we solve this convex problem with AM iterations although one could use a dedicated convex optimization solver).

- `pmax` is the maximum value taken by the penalty exponent p
- `exponent_update_frequency` corresponds to the minimum number of AM iterations between two consecutive updates of the exponent

[8]:
```
%matplotlib notebook
from dolfin import *
import matplotlib.pyplot as plt
import numpy as np

# Algorithmic parameters
niternp = 20 # number of non-penalized iterations
niter = 80 # total number of iterations
pmax = 4         # maximum SIMP exponent
exponent_update_frequency = 4 # minimum number of steps between exponent update
tol_mass = 1e-4 # tolerance on mass when finding Lagrange multiplier
thetamin = 0.001 # minimum density modeling void
```

We now define the problem parameters, in particular the target material density ($\eta = 40\%$ here). The mesh consists of a rectangle of dimension 4×1, clamped on its left side and loaded by a uniformly distributed vertical force on a line of length 0.1 centered around the center of the right side.

Finally, we initialized the SIMP penalty exponent to $p = 1$ and initialized also the density field and the Lagrange multiplier λ.

[9]:
```
# Problem parameters
thetamoy = 0.4 # target average material density
E = Constant(1)
nu = Constant(0.3)
lamda = E*nu/(1+nu)/(1-2*nu)
mu = E/(2*(1+nu))
f = Constant((0, -1)) # vertical downwards force

# Mesh
mesh = RectangleMesh(Point(-2, 0), Point(2, 1), 50, 30, "crossed")
# Boundaries
def left(x, on_boundary):
    return near(x[0], -2) and on_boundary
def load(x, on_boundary):
    return near(x[0], 2) and near(x[1], 0.5, 0.05)
facets = MeshFunction("size_t", mesh, 1)
AutoSubDomain(load).mark(facets, 1)
ds = Measure("ds", subdomain_data=facets)

# Function space for density field
V0 = FunctionSpace(mesh, "DG", 0)
# Function space for displacement
V2 = VectorFunctionSpace(mesh, "CG", 2)
# Fixed boundary condtions
bc = DirichletBC(V2, Constant((0, 0)), left)

p = Constant(1) # SIMP penalty exponent
exponent_counter = 0 # exponent update counter
lagrange = Constant(1) # Lagrange multiplier for volume constraint

thetaold = Function(V0, name="Density")
thetaold.interpolate(Constant(thetamoy))
coeff = thetaold**p
```

(continues on next page)

(continued from previous page)

```
theta = Function(V0)

volume = assemble(Constant(1.)*dx(domain=mesh))
avg_density_0 = assemble(thetaold*dx)/volume # initial average density
avg_density = 0.
```

We now define some useful functions for formulating the linear elastic variational problem.

[10]:
```
def eps(v):
    return sym(grad(v))
def sigma(v):
    return coeff*(lamda*div(v)*Identity(2)+2*mu*eps(v))
def energy_density(u, v):
    return inner(sigma(u), eps(v))

# Inhomogeneous elastic variational problem
u_ = TestFunction(V2)
du = TrialFunction(V2)
a = inner(sigma(u_), eps(du))*dx
L = dot(f, u_)*ds(1)
```

We now define the function for updating the value of θ. Note that this function will be called many times in each step since we are finding λ through a dichotomy procedure. For this reason, we make use of the *local_project* function (see *Elasto-plastic analysis of a 2D von Mises material* and *Efficient projection on DG or Quadrature spaces*) for fast projection on local spaces such as DG0 for the present case.

[11]:
```
def local_project(v, V):
    dv = TrialFunction(V)
    v_ = TestFunction(V)
    a_proj = inner(dv, v_)*dx
    b_proj = inner(v, v_)*dx
    solver = LocalSolver(a_proj, b_proj)
    solver.factorize()
    u = Function(V)
    solver.solve_local_rhs(u)
    return u
def update_theta():
    theta.assign(local_project((p*coeff*energy_density(u, u)/lagrange)**(1/(p+1)),␣
↪V0))
    thetav = theta.vector().get_local()
    theta.vector().set_local(np.maximum(np.minimum(1, thetav), thetamin))
    theta.vector().apply("insert")
    avg_density = assemble(theta*dx)/volume
    return avg_density
```

We now define a function for finding the correct value of the Lagrange multiplier λ. First, a rough bracketing of λ is obtained, then a dichotomy is performed in the interval `[lagmin,lagmax]` until the correct average density is obtained to a certain tolerance.

[12]:
```
def update_lagrange_multiplier(avg_density):
    avg_density1 = avg_density
    # Initial bracketing of Lagrange multiplier
    if (avg_density1 < avg_density_0):
        lagmin = float(lagrange)
        while (avg_density < avg_density_0):
            lagrange.assign(Constant(lagrange/2))
```

(continues on next page)

(continued from previous page)

```
            avg_density = update_theta()
        lagmax = float(lagrange)
    elif (avg_density1 > avg_density_0):
        lagmax = float(lagrange)
        while (avg_density > avg_density_0):
            lagrange.assign(Constant(lagrange*2))
            avg_density = update_theta()
        lagmin = float(lagrange)
    else:
        lagmin = float(lagrange)
        lagmax = float(lagrange)

    # Dichotomy on Lagrange multiplier
    inddico=0
    while ((abs(1.-avg_density/avg_density_0)) > tol_mass):
        lagrange.assign(Constant((lagmax+lagmin)/2))
        avg_density = update_theta()
        inddico += 1;
        if (avg_density < avg_density_0):
            lagmin = float(lagrange)
        else:
            lagmax = float(lagrange)
    print("   Dichotomy iterations:", inddico)
```

Finally, the exponent update strategy is implemented:

- first, $p = 1$ for the first `niternp` iterations
- then, p is increased by some amount which depends on the average gray level of the density field computed as $g = \frac{1}{\text{Vol(D)}} \int_D 4(\theta - \theta_{min})(1-\theta)\,\text{dx}$, that is $g = 0$ is $\theta(x) = \theta_{min}$ or 1 everywhere and $g = 1$ is $\theta = (\theta_{min}+1)/2$ everywhere.
- Note that p can increase only if at least `exponent_update_frequency` AM iterations have been performed since the last update and only if the compliance evolution falls below a certain threshold.

[13]:
```
def update_exponent(exponent_counter):
    exponent_counter += 1
    if (i < niternp):
        p.assign(Constant(1))
    elif (i >= niternp):
        if i == niternp:
            print("\n Starting penalized iterations\n")
        if ((abs(compliance-old_compliance) < 0.01*compliance_history[0]) and
            (exponent_counter > exponent_update_frequency) ):
            # average gray level
            gray_level = assemble((theta-thetamin)*(1.-theta)*dx)*4/volume
            p.assign(Constant(min(float(p)*(1+0.3**(1.+gray_level/2)), pmax)))
            exponent_counter = 0
            print("   Updated SIMP exponent p = ", float(p))
    return exponent_counter
```

Finally, the global loop for the algorithm is implemented consisting, at each iteration, of the elasticity problem resolution, the corresponding compliance computation, the update for θ and its associated Lagrange multiplier λ and the exponent update procedure.

[ ]:
```
u = Function(V2, name="Displacement")
old_compliance = 1e30
ffile = XDMFFile("topology_optimization.xdmf")
ffile.parameters["flush_output"]=True
ffile.parameters["functions_share_mesh"]=True
compliance_history = []
for i in range(niter):
    solve(a == L, u, bc, solver_parameters={"linear_solver": "cg", "preconditioner":
↪"hypre_amg"})
    ffile.write(thetaold, i)
    ffile.write(u, i)

    compliance = assemble(action(L, u))
    compliance_history.append(compliance)
    print("Iteration {}: compliance =".format(i), compliance)

    avg_density = update_theta()

    update_lagrange_multiplier(avg_density)

    exponent_counter = update_exponent(exponent_counter)

    # Update theta field and compliance
    thetaold.assign(theta)
    old_compliance = compliance
```

The final density is represented as well as the convergence history of the compliance. One can note that the final compliance obtained after the first non-penalized iterations is smaller than the final one. This initially obtained topology is therefore more optimal than the final one, although it is made of large diffuse gray regions (see XMDF outputs or the animation at the beginning of the tour) contrary to the final one which is close to being binary.

[15]:
```
plot(theta, cmap="bone_r")
plt.title("Final density")
plt.show();

plt.figure()
plt.plot(np.arange(1, niter+1), compliance_history)
ax = plt.gca()
ymax = ax.get_ylim()[1]
plt.plot([niternp, niternp], [0, ymax], "--k")
plt.annotate(r"$\leftarrow$ Penalized iterations $\rightarrow$", xy=[niternp+1,␣
↪ymax*0.02], fontsize=14)
plt.xlabel("Number of iterations")
plt.ylabel("Compliance")
plt.title("Convergence history", fontsize=16)
plt.show();
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

9.1.3 References

[ ]:

CHAPTER 10

Tips and Tricks

Contents:

The corresponding files can be obtained from:

- Jupyter Notebook: `mass_lumping.ipynb`
- Python script: `mass_lumping.py`

10.1 Lumping a mass matrix

Explicit dynamics simulations require the usage of lumped mass matrices i.e. diagonal mass matrices for which inversion can be done explicitly.

We show how to do this for P1 and P2 Lagrange elements. The former case can be done quite easily whereas the latter case requires a special integration scheme which is not present by default in FEniCS.

10.1.1 P1 elements

First method

This problem has been already tackled in the past in old forum posts, see for instance here.

[24]:
```
from dolfin import *
import numpy as np
mesh = UnitSquareMesh(1, 1)
V1 = FunctionSpace(mesh, "CG", 1)
v = TestFunction(V1)
u = TrialFunction(V1)
```

(continues on next page)

(continued from previous page)

```
mass_form = v*u*dx
mass_action_form = action(mass_form, Constant(1))

M_consistent = assemble(mass_form)
print("Consistent mass matrix:\n", np.array_str(M_consistent.array(), precision=3))

M_lumped = assemble(mass_form)
M_lumped.zero()
M_lumped.set_diagonal(assemble(mass_action_form))
print("Lumped mass matrix:\n", np.array_str(M_lumped.array(), precision=3))
```

```
Consistent mass matrix:
 [[0.083 0.042 0.042 0.   ]
 [0.042 0.167 0.083 0.042]
 [0.042 0.083 0.167 0.042]
 [0.    0.042 0.042 0.083]]
Lumped mass matrix:
 [[0.167 0.    0.    0.   ]
 [0.    0.333 0.    0.   ]
 [0.    0.    0.333 0.   ]
 [0.    0.    0.    0.167]]
```

For explicit dynamics simulation, the mass matrix can then be manipulated using the diagonal vector, to compute for instance $M^{-1}w$ where w is some Function.

[25]:
```
M_vect = assemble(mass_action_form)
w = Function(V1)
iMw = Function(V1)
iMw.vector().set_local(w.vector().get_local()/M_vect.get_local())
```

Second method

The above trick consists in summing all rows and affecting the corresponding mass to the corresponding diagonal degree of freedom. E.g. for element T with number of vertices n, the diagonal lumped mass i is obtained as:

$$\overline{M}_i = \sum_j M_{ij} = \sum_j \int_T N_i(x) N_j(x) \text{ dx} = \int_T N_i(x) \text{ dx} = \frac{|T|}{n}$$

for linear shape functions.

We can obtain the same result by integrating the mass form with a "vertex" scheme (i.e. quadrature points are located at the vertices of the simplex with equal weights $\frac{|T|}{n}$:

$$\begin{aligned} M_{ij} &= \int_T N_i(x) N_j(x) \text{ dx('vertex')} \qquad (10.2) \\ &= \sum_{k=1}^{n} \frac{|T|}{n} N_i(x_k) N_j(x_k) = \sum_{k=1}^{n} \frac{|T|}{n} \delta_{ik} \delta_{jk} = \frac{|T|}{n} \delta_{ij} \qquad (10.3) \end{aligned}$$

(1)

[26]:
```
M_lumped2 = assemble(v*u*dx(scheme="vertex", metadata={"degree":1, "representation":
↪"quadrature"}))
print("Lumped mass matrix ('vertex' scheme):\n", np.array_str(M_lumped2.array(),␣
↪precision=3))
```

```
Lumped mass matrix ('vertex' scheme):
 [[0.167 0.    0.    0.   ]
 [0.    0.333 0.    0.   ]
 [0.    0.    0.333 0.   ]
 [0.    0.    0.    0.167]]
```

```
/home/bleyerj/.local/lib/python3.6/site-packages/ffc/jitcompiler.py:234:␣
↪QuadratureRepresentationDeprecationWarning:
*** ======================================================== ***
*** FFC: quadrature representation is deprecated! It will ***
*** likely be removed in 2018.2.0 release. Use uflacs     ***
*** representation instead.                               ***
*** ======================================================== ***
  issue_deprecation_warning()
```

10.1.2 P2 elements

The problem with the first method is that we obtain a singular matrix for P2 elements:

[27]:
```
V2 = FunctionSpace(mesh, "CG", 2)
v = TestFunction(V2)
u = TrialFunction(V2)

mass_form = v*u*dx
mass_action_form = action(mass_form, Constant(1))

M_consistent = assemble(mass_form)
print("Consistent mass matrix:\n", np.array_str(M_consistent.array(), precision=3))

M_lumped = assemble(mass_form)
M_lumped.zero()
M_lumped.set_diagonal(assemble(mass_action_form))
print("Lumped mass matrix:\n", np.array_str(M_lumped.array(), precision=3))
```

```
Consistent mass matrix:
 [[ 0.017 -0.003 -0.003 -0.011  0.     0.     0.     0.     0.   ]
 [-0.003  0.033 -0.006  0.    -0.011  0.    -0.003 -0.011  0.   ]
 [-0.003 -0.006  0.033  0.     0.    -0.011 -0.003  0.    -0.011]
 [-0.011  0.     0.     0.178  0.044  0.044 -0.011  0.044  0.044]
 [ 0.    -0.011  0.     0.044  0.089  0.044  0.     0.     0.   ]
 [ 0.     0.    -0.011  0.044  0.044  0.089  0.     0.     0.   ]
 [ 0.    -0.003 -0.003 -0.011  0.     0.     0.017  0.     0.   ]
 [ 0.    -0.011  0.     0.044  0.     0.     0.     0.089  0.044]
 [ 0.     0.    -0.011  0.044  0.     0.     0.     0.044  0.089]]
Lumped mass matrix:
 [[0.    0.    0.    0.    0.    0.    0.    0.    0.   ]
 [0.    0.    0.    0.    0.    0.    0.    0.    0.   ]
 [0.    0.    0.    0.    0.    0.    0.    0.    0.   ]
 [0.    0.    0.    0.333 0.    0.    0.    0.    0.   ]
 [0.    0.    0.    0.    0.167 0.    0.    0.    0.   ]
 [0.    0.    0.    0.    0.    0.167 0.    0.    0.   ]
 [0.    0.    0.    0.    0.    0.    0.    0.    0.   ]
```

(continues on next page)

(continued from previous page)

```
 [0.    0.    0.    0.    0.    0.    0.    0.167 0.   ]
 [0.    0.    0.    0.    0.    0.    0.    0.    0.167]]
```

We can however adapt the second method by considering a generalized "vertex" quadrature scheme with quadrature points located at the element degrees of freedom points i.e. at vertices and facets mid-points for P2 elements. In this case, we still have (10.3). To do this we need to implement a custom quadrature scheme called `"lumped"` based on the following this discussion.

This has been implemented in the file `lumping_scheme.py` for triangles and tetrahedra.

[28]:
```
import lumping_scheme

M_lumped = assemble(v*u*dx(scheme="lumped", degree=2))
print("Lumped mass matrix ('lumped' scheme):\n", np.array_str(M_lumped.array(),␣
↪precision=3))
```

```
Lumped mass matrix ('lumped' scheme):
 [[0.083 0.    0.    0.    0.    0.    0.    0.    0.   ]
 [0.    0.167 0.    0.    0.    0.    0.    0.    0.   ]
 [0.    0.    0.167 0.    0.    0.    0.    0.    0.   ]
 [0.    0.    0.    0.167 0.    0.    0.    0.    0.   ]
 [0.    0.    0.    0.    0.083 0.    0.    0.    0.   ]
 [0.    0.    0.    0.    0.    0.083 0.    0.    0.   ]
 [0.    0.    0.    0.    0.    0.    0.083 0.    0.   ]
 [0.    0.    0.    0.    0.    0.    0.    0.083 0.   ]
 [0.    0.    0.    0.    0.    0.    0.    0.    0.083]]
```

The corresponding files can be obtained from:

- Jupyter Notebook: `computing_reactions.ipynb`
- Python script: `computing_reactions.py`

10.2 Computing consistent reaction forces

One often needs to compute a resulting reaction force on some part of the boundary as post-processing of a mechanical resolution.

Quite often, such a reaction will be computed using the stress field associated with the computed displacement. We will see that this may lead to slight inaccuracies whereas another more consistent approach using the virtual work principle is more consistent.

10.2.1 A cantilever beam problem

We reuse here a simple 2D small strain elasticity script of a rectangular domain of dimensions $L \times H$ representing a cantilever beam clamped on the left hand-side and under uniform body forces $\boldsymbol{f} = (f_x, f_y)$. P^2 Lagrange elements are used for the displacement discretization.

For the sake of illustration, we are interested in computing the horizontal and vertical reaction forces R_x and R_y on the left boundary as well as the resulting moment M_z around the out-of-plane direction. In the present simple case

using global balance equations, they are all given explicitly by:

$$R_x = \int_{x=0} \boldsymbol{T} \cdot \boldsymbol{e}_x = \int_{x=0} (-\sigma_{xx}) dS = -f_x \cdot L \cdot H$$
$$R_y = \int_{x=0} \boldsymbol{T} \cdot \boldsymbol{e}_y = \int_{x=0} (-\sigma_{xy}) dS = -f_y \cdot L \cdot H$$
$$M_z = \int_{x=0} (\vec{\boldsymbol{OM}} \times \boldsymbol{T}) \cdot \boldsymbol{e}_z = \int_{x=0} (y\sigma_{xx}) dS = f_y \cdot \frac{L^2}{2} \cdot H$$

[6]:
```
from dolfin import *
import matplotlib.pyplot
%matplotlib notebook

L = 5.
H = 1.
Nx = 20
Ny = 5
mesh = RectangleMesh(Point(0., 0.), Point(L, H), Nx, Ny, "crossed")
plot(mesh)

E = Constant(1e5)
nu = Constant(0.3)
mu = E/2/(1+nu)
lmbda = E*nu/(1+nu)/(1-2*nu)


def eps(v):
    return sym(grad(v))


def sigma(v):
    return lmbda*tr(eps(v))*Identity(2) + 2.0*mu*eps(v)


fx = 0.1
fy = -1.
f = Constant((fx, fy))

V = VectorFunctionSpace(mesh, 'Lagrange', degree=2)
du = TrialFunction(V)
u_ = TestFunction(V)
a = inner(sigma(du), eps(u_))*dx
l = inner(f, u_)*dx


def left(x, on_boundary):
    return near(x[0], 0.)


facets = MeshFunction("size_t", mesh, 1)
AutoSubDomain(left).mark(facets, 1)
ds = Measure("ds", subdomain_data=facets)
bc = DirichletBC(V, Constant((0., 0.)), facets, 1)

u = Function(V, name="Displacement")
solve(a == l, u, bc)
plot(u[1])
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[6]:
```
<matplotlib.tri.tricontour.TriContourSet at 0x7fcc98623a90>
```

10.2.2 First method: using the post-processed stress

The first, and most widely used, method for computing the above reactions relies on the stress field computed from the obtained displacement `sigma(u)` and perform `assemble` over the left boundary (measure `ds(1)`). Unfortunately, this procedure does not ensure an exact computation as seen below. Indeed, the stress field, implicitly known only at the quadrature points only is extended to the structure boundary and does not satisfy global equilibrium anymore.

[11]:
```
x = SpatialCoordinate(mesh)

print("Horizontal reaction Rx = {}".format(
    assemble(-sigma(u)[0, 0]*ds(1))))
print("              (analytic = {})".format(-L*H*fx))
print("-"*50)
print("Vertical reaction Ry = {}".format(
    assemble(-sigma(u)[0, 1]*ds(1))))
print("            (analytic = {})".format(-L*H*fy))
print("-"*50)
print("Bending moment Mz = {}".format(assemble(-x[1]*sigma(u)[0, 0]*ds(1))))
print("         (analytic = {})".format(H*L**2/2*fy))
print("-"*50)
print("\n")
```

```
Horizontal reaction Rx = -0.48578099336058145
              (analytic = -0.5)
--------------------------------------------------
Vertical reaction Ry = 4.514399579992231
           (analytic = 5.0)
--------------------------------------------------
Bending moment Mz = -12.057524729246653
         (analytic = -12.5)
--------------------------------------------------


```

10.2.3 Second method: using the work of internal forces

The second approach relies on the virtual work principle (or weak formulation) which writes in the present case:

$$\int_\Omega (\boldsymbol{u}) : \nabla^s \boldsymbol{v} = \int_\Omega \boldsymbol{f} \cdot \boldsymbol{v} + \int_{\partial\Omega_N} \boldsymbol{T} \cdot \boldsymbol{v} + \int_{\partial\Omega_D} \boldsymbol{T} \cdot \boldsymbol{v} \quad \forall \boldsymbol{v} \in V$$

in which $\boldsymbol{v}$ does not necessarily satisfy the Dirichlet boundary conditions on $\partial\Omega_D$.

The solution u is precisely obtained by enforcing the Dirichlet boundary conditions on $\boldsymbol{v}$ such that:

$$\int_\Omega (\boldsymbol{u}) : \nabla^s \boldsymbol{v} = \int_\Omega \boldsymbol{f} \cdot \boldsymbol{v} + \int_{\partial\Omega_N} \boldsymbol{T} \cdot \boldsymbol{v} \quad \forall \boldsymbol{v} \in V \text{ and } \boldsymbol{v} = 0 \text{ on } \partial\Omega_D$$

Defining the **residual**:

$$Res(v) = \int_\Omega (\boldsymbol{u}) : \nabla^s \boldsymbol{v} - \int_\Omega \boldsymbol{f} \cdot \boldsymbol{v} - \int_{\partial\Omega_N} \boldsymbol{T} \cdot \boldsymbol{v} = a(\boldsymbol{u}, \boldsymbol{v}) - \ell(\boldsymbol{v})$$

we have that $Res(v) = 0$ if $\boldsymbol{v} = 0$ on $\partial\Omega_D$.

Now, if $\boldsymbol{v} \neq 0$ on $\partial\Omega_D$, say, for instance, $\boldsymbol{v} = (1, 0)$ on $\partial\Omega_D$, we have that:

$$Res(v) = \int_{\partial\Omega_D} \boldsymbol{T} \cdot \boldsymbol{v} = \int_{\partial\Omega_D} \boldsymbol{T}_x = \int_{\partial\Omega_D} -\sigma_{xx} = R_x$$

Similarly, we obtain the vertical reaction R_y by considering $\boldsymbol{v} = (0, 1)$ and the bending moment M_z by considering $\boldsymbol{v} = (y, 0)$.

As regards implementation, the residual is defined using the action of the bilinear form on the displacement solution: `residual = action(a, u) - l`. We then define boundary conditions on the left boundary and apply them to an empty Function to define the required test field v. We observe that the computed reactions are now exact.

[10]:
```
residual = action(a, u) - l

v_reac = Function(V)
bcRx = DirichletBC(V.sub(0), Constant(1.), facets, 1)
bcRy = DirichletBC(V.sub(1), Constant(1.), facets, 1)
bcMz = DirichletBC(V.sub(0), Expression("x[1]", degree=1), facets, 1)

bcRx.apply(v_reac.vector())
print("Horizontal reaction Rx = {}".format(assemble(action(residual, v_reac))))
print("              (analytic = {})".format(-L*H*fx))
print("-"*50)
v_reac.interpolate(Constant((0., 0.)))

bcRy.apply(v_reac.vector())
print("Vertical reaction Ry = {}".format(assemble(action(residual, v_reac))))
print("            (analytic = {})".format(-L*H*fy))
print("-"*50)
v_reac.interpolate(Constant((0., 0.)))

bcMz.apply(v_reac.vector())
print("Bending moment Mz = {}".format(assemble(action(residual, v_reac))))
print("         (analytic = {})".format(H*L**2/2*fy))
print("-"*50)
```

```
Horizontal reaction Rx = -0.49999999999492606
              (analytic = -0.5)
--------------------------------------------------
Vertical reaction Ry = 5.000000000715704
            (analytic = 5.0
--------------------------------------------------
Bending moment Mz = -12.750000002550555
          analytic = -12.5
--------------------------------------------------
```

In construction...

10.3 Efficient projection on DG or Quadrature spaces

For projecting a Function on a DG or Quadrature space, that is a space with no coupling between elements, the projection can be performed element-wise. For this purpose, using the LocalSolver is much faster than performing a global projection:

```
metadata={"quadrature_degree": deg}
def local_project(v,V):
    dv = TrialFunction(V)
    v_ = TestFunction(V)
    a_proj = inner(dv,v_)*dx(metadata=metadata)
    b_proj = inner(v,v_)*dx(metadata=metadata)
    solver = LocalSolver(a_proj,b_proj)
    solver.factorize()
    u = Function(V)
    solver.solve_local_rhs(u)
    return u
```

Local factorizations can be cached if projection is performed many times.

CHAPTER 11

Useful packages

Here is a list of useful packages graviting around the FEniCS ecosystem:

11.1 FEniCS add-ons

- fenics-shells: a FEniCS library for simulating thin structures
- multiphenics: easy prototyping of multiphysics problems in FEniCS
- RBniCS: reduced order modelling techniques for parametrized problems in FEniCS
- fenics_optim: a Convex Optimization interface in FEniCS
- PFIBS: a Parallel FEniCS Implementation of Block Solvers

Bibliography

[ERL2002] Silvano Erlicher, Luca Bonaventura, Oreste Bursi. The analysis of the Generalized-alpha method for non-linear dynamic problems. Computational Mechanics, Springer Verlag, 2002, 28, pp.83-104, doi:10.1007/s00466-001-0273-z

[BON2014] Marc Bonnet, Attilio Frangi, Christian Rey. *The finite element method in solid mechanics.* McGraw Hill Education, pp.365, 2014

[HEL2015] Helfer, Thomas, Bruno Michel, Jean-Michel Proix, Maxime Salvo, Jérôme Sercombe, and Michel Casella. 2015. *Introducing the Open-Source Mfront Code Generator: Application to Mechanical Behaviours and Material Knowledge Management Within the PLEIADES Fuel Element Modelling Platform.* Computers & Mathematics with Applications. <https://doi.org/10.1016/j.camwa.2015.06.027>.

[HAN2011] Peter Hansbo, David Heintz, Mats G. Larson, A finite element method with discontinuous rotations for the Mindlin-Reissner plate model, *Computer Methods in Applied Mechanics and Engineering*, 200, 5-8, 2011, pp. 638-648, https://doi.org/10.1016/j.cma.2010.09.009.