FEniCS-Shells

Release 2018.1.0

Aug 31, 2022

Contents

CHAPTER 1

Subpackages

1.1 fenics_shells.analytical package

1.1.1 Submodules

1.1.2 fenics_shells.analytical.lovadina_clamped module

Analytical solution for clamped Reissner-Mindlin plate problem from Lovadina et al.

1.1.3 fenics_shells.analytical.simply_supported module

Analytical solution for simply-supported Reissner-Mindlin square plate under a uniform transverse load.

1.1.4 fenics_shells.analytical.vonkarman_heated module

Analytical solution for elliptic orthotropic von Karman plate with lenticular thickness subject to a uniform field of inelastic curvatures.

`fenics_shells.analytical.vonkarman_heated.`**`analytical_solution`**(*Ai*, *Di*, *a_rad*, *b_rad*)

1.1.5 Module contents

1.2 fenics_shells.common package

1.2.1 Submodules

1.2.2 fenics_shells.common.constitutive_models module

fenics_shells.common.constitutive_models.**psi_M**(*k*, ***kwargs*)
Returns bending moment energy density calculated from the curvature k using:

Isotropic case: .. math:: D = frac{E*t^3}{24(1 - nu^2)} W_m(k, ldots) = D*((1 - nu)*tr(k**2) + nu*(tr(k))**2)

Parameters

- **k** – Curvature, typically UFL form with shape (2,2) (tensor).
- ****kwargs** – Isotropic case: E: Young's modulus, Constant or Expression. nu: Poisson's ratio, Constant or Expression. t: Thickness, Constant or Expression.

Returns UFL form of bending stress tensor with shape (2,2) (tensor).

fenics_shells.common.constitutive_models.**psi_N**(*e*, ***kwargs*)
Returns membrane energy density calculated from e using:

Isotropic case: .. math:: B = frac{E*t}{2(1 - nu^2)} N(e, ldots) = B(1 - nu)e + nu mathrm{tr}(e)I

Parameters

- **e** – Membrane strain, typically UFL form with shape (2,2) (tensor).
- ****kwargs** – Isotropic case: E: Young's modulus, Constant or Expression. nu: Poisson's ratio, Constant or Expression. t: Thickness, Constant or Expression.

Returns UFL form of membrane stress tensor with shape (2,2) (tensor).

fenics_shells.common.constitutive_models.**strain_from_voigt**(*e_voigt*)
Inverse operation of strain_to_voigt.

Parameters

- **sigma_voigt** – UFL form with shape (3,1) corresponding to the strain
- **in Voigt format** (*pseudo-vector*) –

Returns a symmetric stress tensor, typically UFL form with shape (2,2)

fenics_shells.common.constitutive_models.**strain_to_voigt**(*e*)
Returns the pseudo-vector in the Voigt notation associate to a 2x2 symmetric strain tensor, according to the following rule (see e.g. https://en.wikipedia.org/wiki/Voigt_notation),

$$e = \begin{bmatrix} e_{00} & e_{01} \\ e_{01} & e_{11} \end{bmatrix} \quad \rightarrow \quad e_{\text{voigt}} = \begin{bmatrix} e_{00} & e_{11} & 2e_{01} \end{bmatrix}$$

Parameters **e** – a symmetric 2x2 strain tensor, typically UFL form with shape (2,2)

Returns a UFL form with shape (3,1) corresponding to the input tensor in Voigt notation.

fenics_shells.common.constitutive_models.**stress_from_voigt**(*sigma_voigt*)
Inverse operation of stress_to_voigt.

Parameters

- **sigma_voigt** – UFL form with shape (3,1) corresponding to the stress
- **in Voigt format.** (*pseudo-vector*) –

Returns a symmetric stress tensor, typically UFL form with shape (2,2)

fenics_shells.common.constitutive_models.**stress_to_voigt**(*sigma*)

Returns the pseudo-vector in the Voigt notation associate to a 2x2 symmetric stress tensor, according to the following rule (see e.g. https://en.wikipedia.org/wiki/Voigt_notation),

$$\sigma = \begin{bmatrix} \sigma_{00} & \sigma_{01} \\ \sigma_{01} & \sigma_{11} \end{bmatrix} \quad \rightarrow \quad \sigma_{\text{voigt}} = \begin{bmatrix} \sigma_{00} & \sigma_{11} & \sigma_{01} \end{bmatrix}$$

Parameters

- **sigma** – a symmetric 2x2 stress tensor, typically UFL form with shape
- **(2,2)** –

Returns a UFL form with shape (3,1) corresponding to the input tensor in Voigt notation.

1.2.3 fenics_shells.common.energy module

fenics_shells.common.energy.**membrane_bending_energy**(*e*, *k*, *A*, *D*, *B*)

Return the coupled membrane-bending energy for a plate model.

Parameters

- **e** – Membrane strains, UFL or DOLFIN Function of rank (2, 2) (tensor).
- **k** – Curvature, UFL or DOLFIN Function of rank (2, 2) (tensor).
- **A** – Membrane stresses.
- **D** – Bending stresses.
- **B** – Coupled membrane-bending stresses.

fenics_shells.common.energy.**membrane_energy**(*e*, *N*)

Return internal membrane energy for a plate model.

Parameters

- **e** – Membrane strains, UFL or DOLFIN Function of rank (2, 2) (tensor).
- **N** – Membrane stress, UFL or DOLFIN Function of rank (2, 2) (tensor).

Returns UFL form of internal elastic membrane energy for a plate model.

1.2.4 fenics_shells.common.kinematics module

fenics_shells.common.kinematics.**F**(*u*)

Return deformation gradient tensor for non-linear plate model.

Deformation gradient of 2-dimensional manifold embedded in 3-dimensional space.

$$F = I + \nabla u$$

Parameters **u** – displacement field, typically UFL (3,1) coefficient.

Returns a UFL coeffient with shape (3,2)

fenics_shells.common.kinematics.**e**(*u*)
Return membrane strain tensor for linear plate model.

$$e = \frac{1}{2}(\nabla u + \nabla u^T)$$

Parameters **u** – membrane displacement field, typically UFL (2,1) coefficient.

Returns a UFL form with shape (2,2)

fenics_shells.common.kinematics.**k**(*theta*)
Return bending curvature tensor for linear plate model.

$$k = \frac{1}{2}(\nabla \theta + \nabla \theta^T)$$

Parameters **theta** – rotation field, typically UFL (2,1) form or a dolfin Function

Returns a UFL form with shape (2,2)

1.2.5 fenics_shells.common.laminates module

fenics_shells.common.laminates.**ABD**(*E1*, *E2*, *G12*, *nu12*, *hs*, *thetas*)
Return the stiffness matrix of a kirchhoff-love model of a laminate obtained by stacking n orthotropic laminae with possibly different thinknesses and orientations (see Reddy 1997, eqn 1.3.71).

It assumes a plane-stress state.

Parameters

- **E1** – The Young modulus in the material direction 1.
- **E2** – The Young modulus in the material direction 2.
- **G12** – The in-plane shear modulus.
- **nu12** – The in-plane Poisson ratio.
- **hs** – a list with length n with the thicknesses of the layers (from top to bottom).
- **theta** – a list with the n orientations (in radians) of the layers (from top to bottom).

Returns a symmetric 3x3 ufl matrix giving the membrane stiffness in Voigt notation. B: a symmetric 3x3 ufl matrix giving the membrane/bending coupling stiffness in Voigt notation. D: a symmetric 3x3 ufl matrix giving the bending stiffness in Voigt notation.

Return type A

fenics_shells.common.laminates.**F**(*G13*, *G23*, *hs*, *thetas*)
Return the shear stiffness matrix of a Reissner-Midlin model of a laminate obtained by stacking n orthotropic laminae with possibly different thinknesses and orientations. (See Reddy 1997, eqn 3.4.18)

It assumes a plane-stress state.

Parameters

- **G13** – The transverse shear modulus between the material directions 1-3.
- **G23** – The transverse shear modulus between the material directions 2-3.
- **hs** – a list with length n with the thicknesses of the layers (from top to bottom).

- **theta** – a list with the n orientations (in radians) of the layers (from top to bottom).

Returns a symmetric 2x2 ufl matrix giving the shear stiffness in Voigt notation.

Return type *F*

fenics_shells.common.laminates.**NM_T**(*E1*, *E2*, *G12*, *nu12*, *hs*, *thetas*, *DeltaT_0*, *DeltaT_1=0.0*, *alpha1=1.0*, *alpha2=1.0*)

Return the thermal stress and moment resultant of a Kirchhoff-Love model of a laminate obtained by stacking n orthotropic laminae with possibly different thinknesses and orientations.

It assumes a plane-stress states and a temperature distribution in the from

Delta(z) = DeltaT_0 + z * DeltaT_1

Parameters

- **E1** – The Young modulus in the material direction 1.
- **E2** – The Young modulus in the material direction 2.
- **G12** – The in-plane shear modulus.
- **nu12** – The in-plane Poisson ratio.
- **hs** – a list with length n with the thicknesses of the layers (from top to bottom).
- **theta** – a list with the n orientations (in radians) of the layers (from top to bottom).
- **alpha1** – Expansion coefficient in the material direction 1.
- **alpha2** – Expansion coefficient in the material direction 2.
- **DeltaT_0** – Average temperature field.
- **DeltaT_1** – Gradient of the temperature field.

Returns a 3x1 ufl vector giving the membrane inelastic stress. M_T: a 3x1 ufl vector giving the bending inelastic stress.

Return type N_T

fenics_shells.common.laminates.**rotated_lamina_expansion_inplane**(*alpha11*, *alpha22*, *theta*)

Return the in-plane expansion matrix of an orhtropic layer in a reference rotated by an angle theta wrt to the material one. It assumes Voigt notation and plane stress state. (See Reddy 1997, eqn 1.3.71)

Parameters

- **alpha11** – Expansion coefficient in the material direction 1.
- **alpha22** – Expansion coefficient in the material direction 2.
- **theta** – The rotation angle from the material to the desired reference system.

Returns a 3x1 ufl vector giving the expansion matrix in voigt notation.

Return type alpha_theta

fenics_shells.common.laminates.**rotated_lamina_stiffness_inplane**(*E1*, *E2*, *G12*, *nu12*, *theta*)

Return the in-plane stiffness matrix of an orhtropic layer in a reference rotated by an angle theta wrt to the material one. It assumes Voigt notation and plane stress state. (See Reddy 1997, eqn 1.3.71)

Parameters

- **E1** – The Young modulus in the material direction 1.
- **E2** – The Young modulus in the material direction 2.

- **G23** – The in-plane shear modulus
- **nu12** – The in-plane Poisson ratio
- **theta** – The rotation angle from the material to the desired refence system

Returns a 3x3 symmetric ufl matrix giving the stiffness matrix

Return type Q_theta

fenics_shells.common.laminates.**rotated_lamina_stiffness_shear**(*G13*, *G23*, *theta*, *kappa=0.8333333333333334*)

Return the shear stiffness matrix of an orhtropic layer in a reference rotated by an angle theta wrt to the material one. It assumes Voigt notation and plane stress state (see Reddy 1997, eqn 3.4.18).

Parameters

- **G12** – The transverse shear modulus between the material directions 1-2.
- **G13** – The transverse shear modulus between the material directions 1-3.
- **kappa** – The shear correction factor.

Returns a 3x3 symmetric ufl matrix giving the stiffness matrix.

Return type Q_shear_theta

fenics_shells.common.laminates.**z_coordinates**(*hs*)

Return a list with the thickness coordinate of the top surface of each layer taking the midplane as z = 0.

Parameters **hs** – a list giving the thinckesses of each layer ordered from bottom (layer - 0) to top (layer n-1).

Returns

a list of coordinate of the top surface of each layer ordered from bottom (layer - 0) to top (layer n-1)

Return type z

1.2.6 Module contents

1.3 fenics_shells.fem package

1.3.1 Submodules

1.3.2 fenics_shells.fem.CDG module

fenics_shells.fem.CDG.**cdg_energy**(*theta*, *M*, *stabilization*, *mesh*, *bcs_theta=None*, *dS=<Mock id='139667557486224'>*)

Return the continuous/discontinuous terms for a fourth-order plate model.

$$\pi_{cdg} = -\partial_n w \cdot M_n(w) + \frac{1}{2}\frac{\alpha}{|e|}|\partial_n w|^2$$

Parameters

- **theta** – Rotations, UFL or DOLFIN Function of rank (2,) (vector).
- **M** – UFL form of bending moment tensor of rank (2,2) (tensor).

- **stabilization** – a constant or ulf expression providing the stabilization parameter of the continuous/discontinuous formulation. This should be an eximation of the norm of the bending stiffness
- **mesh** – DOLFIN mesh.
- **bcs_theta** (*Optional*) – list of dolfin.DirichletBC for the rotations theta. Defaults to None.
- **dS** – (Optional). Measure on interior facets. Defaults to dolfin.dS.

Returns a dolfin.Form associated with the continuous/discontinuous formulation.

The Kirchhoff-Love plate model is a fourth-order PDE, giving rise to a weak form with solution in Sobolev space $H^2(\Omega)$. Because FEniCS does not currently include support for $H^2(\Omega)$ conforming elements we implement a hybrid continuous/discontinuous approach, allowing the use of Lagrangian elements with reduced regularity requirements.

Description can be found in the paper: G. Engel, K. Garikipati, T. J. R. Hughes, M. G. Larson, L. Mazzei and R. L. Taylor, "Continuous/discontinuous finite element approximations of fourth-order elliptic problems in structural and continuum mechanics with applications to thin beams and plates, and strain gradient elasticity" Comput. Method. Appl. M., vol. 191, no. 34, pp. 3669-3750, 2002.

fenics_shells.fem.CDG.**cdg_stabilization**(*E*, *t*)

Returns the stabilization parameter as the norm of the bending stiffness matrix.

Parameters

- **E** – Young's modulus, Constant or Expression.
- **t** – Thickness, Constant or Expression.

Returns a dolfin.Coefficient providing the stabilization parameter of the continuous/discontinuous formulation.

1.3.3 fenics_shells.fem.assembling module

fenics_shells.fem.assembling.**assemble**(**args*, ***kwargs*)

Pass-through for dolfin.assemble and fenics_shells.projected_assemble.

If the first argument is an instance of ProjectedFunctionSpace it will call fenics_shells.projected_assemble, otherwise it will pass through to dolfin.assemble.

fenics_shells.fem.assembling.**projected_assemble**(*U_P*, *a*, *L*, *bcs=None*, *A=None*, *b=None*, *is_interpolation=False*, *a_is_symmetric=False*, *form_compiler_parameters=None*, *add_values=False*, *finalize_tensor=True*, *keep_diagonal=False*, *backend=None*)

1.3.4 fenics_shells.fem.solving module

fenics_shells.fem.solving.**reconstruct_full_space**(*u_f*, *u_p*, *a*, *L*, *is_interpolation=False*, *a_is_symmetric=False*, *form_compiler_parameters=None*)

Given a Function on a projected space $u_p \in U_P$ and a function in the full space $u_f \in U_F : suchthat : math :$

‘$U_P \subset U_F$, reconstruct the variable u_f on the full space via direct copy of the matching subfunctions shared between U_F and U_P and the local solution of the original problem a == L.

Parameters

- **u_f** – DOLFIN Function on FullFunctionSpace.
- **u_p** – DOLFIN Function on ProjectedFunctionSpace.
- **a** – Bilinear form.
- **L** – Bilinear form.

Returns DOLFIN Function on FullFunctionSpace.

Return type u_f

1.3.5 Module contents

fenics_shells.fem.**projected_assemble**(*U_P*, *a*, *L*, *bcs=None*, *A=None*, *b=None*, *is_interpolation=False*, *a_is_symmetric=False*, *form_compiler_parameters=None*, *add_values=False*, *finalize_tensor=True*, *keep_diagonal=False*, *backend=None*)

fenics_shells.fem.**assemble**(**args*, ***kwargs*)

Pass-through for dolfin.assemble and fenics_shells.projected_assemble.

If the first argument is an instance of ProjectedFunctionSpace it will call fenics_shells.projected_assemble, otherwise it will pass through to dolfin.assemble.

fenics_shells.fem.**reconstruct_full_space**(*u_f*, *u_p*, *a*, *L*, *is_interpolation=False*, *a_is_symmetric=False*, *form_compiler_parameters=None*)

Given a Function on a projected space $u_p \in U_P$ and a function in the full space $u_f \in U_F$: $suchthat$: $math$: ‘$U_P \subset U_F$, reconstruct the variable u_f on the full space via direct copy of the matching subfunctions shared between U_F and U_P and the local solution of the original problem a == L.

Parameters

- **u_f** – DOLFIN Function on FullFunctionSpace.
- **u_p** – DOLFIN Function on ProjectedFunctionSpace.
- **a** – Bilinear form.
- **L** – Bilinear form.

Returns DOLFIN Function on FullFunctionSpace.

Return type u_f

1.4 fenics_shells.functions package

1.4.1 Submodules

1.4.2 fenics_shells.functions.functionspace module

1.4.3 Module contents

1.5 fenics_shells.kirchhoff_love package

1.5.1 Submodules

1.5.2 fenics_shells.kirchhoff_love.forms module

fenics_shells.kirchhoff_love.forms.**theta**(*w*)

Returns the rotations as a function of the transverse displacements according to the Kirchoff-Love kinematics.

..math:: theta = nabla w

Parameters **w** – Transverse displacement field, typically a UFL scalar or a DOLFIN Function

Returns the rotations with shape (2,)

1.5.3 Module contents

Overview

This package contains routines for simulating thin plate structures using the Kirchhoff-Love theory.

Further reading

1.6 fenics_shells.naghdi package

1.6.1 Submodules

1.6.2 fenics_shells.naghdi.kinematics module

fenics_shells.naghdi.kinematics.**G**(*F*)

Returns the stretching tensor (1st non-linear Naghdi strain measure).

$$G = \frac{1}{2}(F^T F - I)$$

Parameters **F** – Deformation gradient.

Returns UFL expression of stretching tensor.

fenics_shells.naghdi.kinematics.**K**(*F*, *d*)

Returns the curvature tensor (2nd non-linear Naghdi strain measure).

$$K = \frac{1}{2}(F^T \nabla d + (\nabla d)^T F^T)$$

Parameters

- **F** – Deformation gradient.
- **d** – Director vector.

Returns UFL expression of curvature tensor.

fenics_shells.naghdi.kinematics.**d**(*theta*)
Director vector.

$$d = \{\sin(\theta_2)\cos(\theta_1), -\sin(\theta_1), \cos(\theta_2)\cos(\theta_1)\}^T$$

Parameters **vector** (*Rotation*) –

Returns UFL expression of director vector.

fenics_shells.naghdi.kinematics.**g**(*F*, *d*)
Returns the shear strain vector (3rd non-linear Naghdi strain measure).

$$g = F^T d$$

Parameters

- **F** – Deformation gradient.
- **d** – Director vector.

Returns UFL expression of shear strain vector.

1.6.3 Module contents

1.7 fenics_shells.reissner_mindlin package

1.7.1 Submodules

1.7.2 fenics_shells.reissner_mindlin.forms module

fenics_shells.reissner_mindlin.forms.**gamma**(*theta*, *w*)
Return shear strain vector calculated from primal variables:

$$\gamma = \nabla w - \theta$$

fenics_shells.reissner_mindlin.forms.**inner_e**(*x*, *y*, *restrict_to_one_side=False*, *quadrature_degree=1*)
The inner product of the tangential component of a vector field on all of the facets of the mesh (Measure objects dS and ds).

By default, restrict_to_one_side is False. In this case, the function will return an integral that is restricted to both sides ('+') and ('-') of a shared facet between elements. You should use this in the case that you want to use the 'projected' version of DuranLibermanSpace.

If restrict_to_one_side is True, then this will return an integral that is restricted ('+') to one side of a shared facet between elements. You should use this in the case that you want to use the *multipliers* version of DuranLibermanSpace.

Parameters

- **x** – DOLFIN or UFL Function of rank (2,) (vector).

- **y** – DOLFIN or UFL Function of rank (2,) (vector).
- **(Optional[bool]** (*restrict_to_one_side*) – Default is False.
- **quadrature_degree** (*Optional[int]*) – Default is 1.

Returns UFL Form.

fenics_shells.reissner_mindlin.forms.**psi_T**(*gamma*, ***kwargs*)

Returns transverse shear energy density calculated using:

Isotropic case: .. math:

```
\psi_T(\gamma, \ldots) = \frac{E \kappa t}{4(1 + \nu)}\gamma**2
```

Parameters

- **gamma** – Shear strain, typically UFL form with shape (2,).
- ****kwargs** – Isotropic case: E: Young's modulus, Constant or Expression. nu: Poisson's ratio, Constant or Expression. t: Thickness, Constant or Expression. kappa: Shear correction factor, Constant or Expression.

Returns UFL expression of transverse shear stress vector.

1.7.3 fenics_shells.reissner_mindlin.function_spaces module

fenics_shells.reissner_mindlin.function_spaces.**DuranLibermanSpace**(*mesh*)

A helper function which returns a FiniteElement for the simulation of the out-of-plane Reissner-Mindlin problem without shear-locking based on the ideas of Duran and Liberman's paper:

R. Durán and E. Liberman, "On mixed finite element methods for the Reissner-Mindlin plate model" Math. Comp., vol. 58, no. 198, pp. 561–573, 1992.

Parameters **mesh** (*dolfin.Mesh*) – a mesh of geometric dimension 2.

Returns fenics_shells.ProjectedFunctionSpace.

fenics_shells.reissner_mindlin.function_spaces.**MITC7Space**(*mesh*, *space_type='multipliers'*)

Warning: Currently not working due to regressions in FFC/FIAT.

A helper function which returns a FunctionSpace for the simulation of the out-of-plane Reissner-Mindlin problem without shear-locking based on the ideas of Brezzi, Bathe and Fortin's paper:

"Mixed-interpolated elements for Reissner–Mindlin plates" Int. J. Numer. Meth. Engng., vol. 28, no. 8, pp. 1787–1801, Aug. 1989. http://dx.doi.org/10.1002/nme.1620280806

In the case that space_type is "multipliers", a dolfin.FunctionSpace will be returned with an additional Lagrange multiplier field to tie the shear strains computer with the primal variables (transverse displacement and rotations) to the independent shear strain variable.

In the case that space_type is "primal", a dolfin.FunctionSpace will be returned with just the primal (transverse displacement and rotation) variables. Of course, any Reissner-Mindlin problem constructed with this approach will be prone to shear-locking.

Parameters

- **mesh** (*dolfin.Mesh*) – a mesh of geometric dimension 2.
- **space_type** (*Optional[str]*) – Can be "primal", or
- **Default is "multipliers".** (*"multipliers".*) –

Returns dolfin.FunctionSpace.

1.7.4 Module contents

Overview

This package contains routines for simulating thin through to moderately thick plate structures using the Reissner-Mindlin theory.

Further reading

1.8 fenics_shells.utils package

1.8.1 Submodules

1.8.2 fenics_shells.utils.Probe module

`fenics_shells.utils.Probe.`**`strip_essential_code`**(*filenames*)

1.8.3 Module contents

1.9 fenics_shells.von_karman package

1.9.1 Submodules

1.9.2 fenics_shells.von_karman.kinematics module

`fenics_shells.von_karman.kinematics.`**`e`**(*u*, *theta*)

Return the membrane strain tensor for the Von-Karman plate model.

$$e(u, theta) = \mathrm{sym}\nabla u + \frac{\theta \otimes \theta}{2}$$

Parameters

- **`u`** – In-plane displacement.
- **`theta`** – Rotations.

Returns UFL form of Von-Karman membrane strain tensor.

1.9.3 Module contents

Overview

This package contains routines for simulating thin plate structures using the von Karman theory.

Further reading

CHAPTER 2

Module contents

fenics-shells is an open-source library that provides a wide range of thin structural models (beams, plates and shells) expressed in the Unified Form Language (UFL) of the FEniCS Project.

CHAPTER 3

Documented demos

3.1 Where to start

We suggest new users to look at the folllowing two demos for having a first idea of the possible approaches and modelling capabilities for linear plate (MITC discretisation) and nonlinear shells (PRSI discretisation):

3.1.1 Clamped Reissner-Mindlin plate under uniform load

This demo is implemented in the single Python file `demo_reissner-mindlin-clamped.py`.

This demo program solves the out-of-plane Reissner-Mindlin equations on the unit square with uniform transverse loading with fully clamped boundary conditions. It is assumed the reader understands most of the functionality in the FEniCS Project documented demos.

Specifically, you should know how to:

- Define a `MixedElement` and a `FunctionSpace` from it.
- Write variational forms using the Unified Form Language.
- Automatically derive Jacobian and residuals using `derivative()`.
- Apply Dirichlet boundary conditions using `DirichletBC` and `apply()`.
- Assemble forms using `assemble()`.
- Solve linear systems using `LUSolver`.
- Output data to XDMF files with `XDMFFile`.

This demo then illustrates how to:

- Define the Reissner-Mindlin plate equations using UFL.
- Define the Durán-Liberman (MITC) reduction operator using UFL. This procedure eliminates the shear-locking problem.

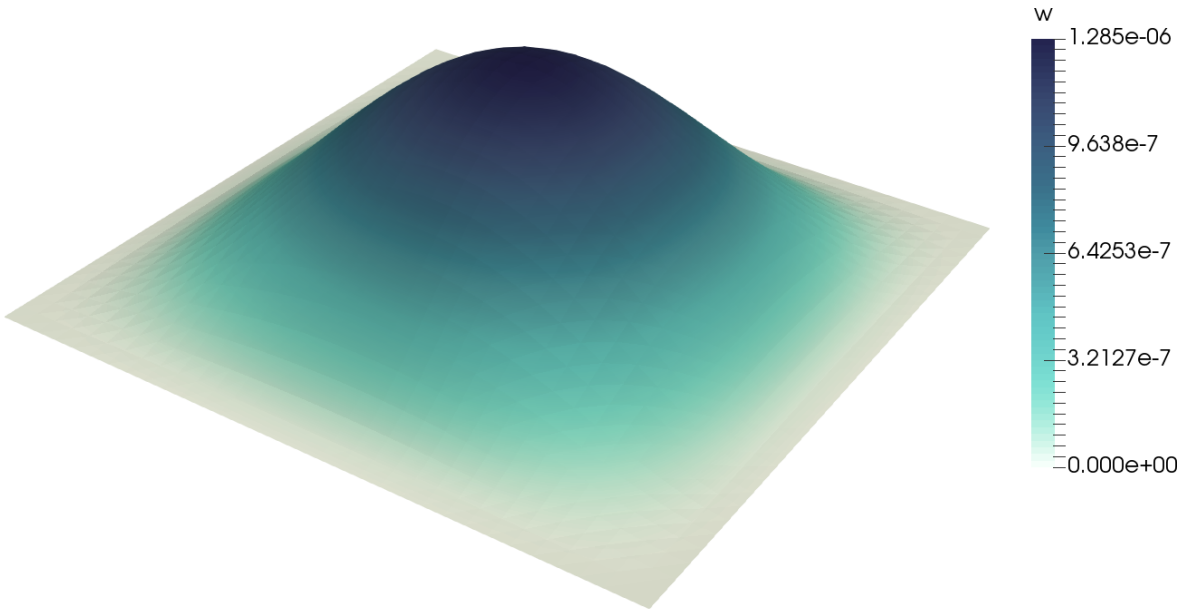


Fig. 3.1: Transverse displacement field w of the clamped Reissner-Mindlin plate problem scaled by a factor of 250000.

- Use `ProjectedFunctionSpace` and `assemble()` in FEniCS-Shells to statically condensate two problem variables and assemble a linear system of reduced size.
- Reconstruct the variables that were statically condensated using `reconstruct_full_space()`.

First the `dolfin` and *`fenics_shells`* modules are imported. The *`fenics_shells`* module overrides some standard methods in DOLFIN, so it should always be `import`-ed *after* `dolfin`:

```
from dolfin import *
from fenics_shells import *
```

We then create a two-dimensional mesh of the mid-plane of the plate $\Omega = [0, 1] \times [0, 1]$:

```
mesh = UnitSquareMesh(32, 32)
```

The Durán-Liberman element for the Reissner-Mindlin plate problem consists of second-order vector-valued element for the rotation field $\theta \in [\mathrm{CG}_2]^2$ and a first-order scalar valued element for the transverse displacement field $w \in \mathrm{CG}_1$, see [1]. Two further auxilliary fields are also considered, the reduced shear strain γ_R, and a Lagrange multiplier field p which ties together the shear strain calculated from the primal variables $\gamma = \nabla w - \theta$ and the reduced shear strain γ_R. Both p and γ_R are are discretised in the space NED_1, the vector-valued Nédélec elements of the first kind. The final element definition is then:

```
element = MixedElement([VectorElement("Lagrange", triangle, 2),
                        FiniteElement("Lagrange", triangle, 1),
                        FiniteElement("N1curl", triangle, 1),
                        FiniteElement("N1curl", triangle, 1)])
```

We then pass our `element` through to the `ProjectedFunctionSpace` constructor. As we will see later in this example, we can project out both the p and NED_1 fields at assembly time. We specify this by passing the argument `num_projected_subspaces=2`:

```
Q = ProjectedFunctionSpace(mesh, element, num_projected_subspaces=2)
```

From `Q` we can then extract the full space `Q_F`, which consists of all four function fields, collected in the state vector $q = (\theta, w, \gamma_R, p)$.

```
Q_F = Q.full_space
```

In contrast the projected space `Q` only holds the two primal problem fields (θ, w).

Using only the *full* function space object `Q_F` we setup our variational problem by defining the Lagrangian of the Reissner-Mindlin plate problem. We begin by creating a `Function` and splitting it into each individual component function:

```
q_ = Function(Q_F)
theta_, w_, R_gamma_, p_ = split(q_)
q = TrialFunction(Q_F)
q_t = TestFunction(Q_F)
```

We assume constant material parameters; Young's modulus E, Poisson's ratio ν, shear-correction factor κ, and thickness t:

```
E = Constant(10920.0)
nu = Constant(0.3)
kappa = Constant(5.0/6.0)
t = Constant(0.001)
```

The bending strain tensor k for the Reissner-Mindlin model can be expressed in terms of the rotation field θ:

$$k(\theta) = \frac{1}{2}(\nabla\theta + \nabla\theta^T)$$

which can be expressed in UFL as:

```
k = sym(grad(theta_))
```

The bending energy density ψ_b for the Reissner-Mindlin model is a function of the bending strain tensor k:

$$\psi_b(k) = \frac{1}{2}D\left((1-\nu)\,\mathrm{tr}\,(k^2) + \nu\,(\mathrm{tr}\,k)^2\right) \qquad D = \frac{Et^3}{12(1-\nu^2)}$$

which can be expressed in UFL as:

```
D = (E*t**3)/(12.0*(1.0 - nu**2))
psi_b = 0.5*D*((1.0 - nu)*tr(k*k) + nu*(tr(k))**2)
```

Because we are using a mixed variational formulation, we choose to write the shear energy density ψ_s is a function of the reduced shear strain vector:

$$\psi_s(\gamma_R) = \frac{E\kappa t}{4(1+\nu)}\gamma_R^2$$

or in UFL:

```
psi_s = ((E*kappa*t)/(4.0*(1.0 + nu)))*inner(R_gamma_, R_gamma_)
```

Finally, we can write out external work due to the uniform loading in the out-of-plane direction:

$$W_{\text{ext}} = \int_\Omega f t^3 \cdot w \,\mathrm{d}x.$$

where $f = 1$ and $\mathrm{d}x$ is a measure on the whole domain. The scaling by t^3 is included to ensure a correct limit solution as $t \to 0$.

In UFL this can be expressed as:

```
f = Constant(1.0)
W_ext = inner(f*t**3, w_)*dx
```

With all of the standard mechanical terms defined, we can turn to defining the numerical Duran-Liberman reduction operator. This operator 'ties' our reduced shear strain field to the shear strain calculated in the primal space. A partial explanation of the thinking behind this approach is given in the *Appendix*.

The shear strain vector γ can be expressed in terms of the rotation and transverse displacement field:

$$\gamma(\theta, w) := \nabla w - \theta$$

or in UFL:

```
gamma = grad(w_) - theta_
```

We require that the shear strain calculated using the displacement unknowns $\gamma = \nabla w - \theta$ be equal, in a weak sense, to the conforming shear strain field $\gamma_R \in \mathrm{NED}_1$ that we used to define the shear energy above. We enforce this constraint using a Lagrange multiplier field $p \in \mathrm{NED}_1$. We can write the Lagrangian of this constraint as:

$$\Pi_R(\gamma, \gamma_R, p) = \int_e (\{\gamma_R - \gamma\} \cdot t) \cdot (p \cdot t) \ \mathrm{d}s$$

where e are all of edges of the cells in the mesh and t is the tangent vector on each edge.

Writing this operator out in UFL is quite verbose, so *fenics_shells* includes a special *all edges* inner product function `inner_e()` to help. However, we choose to write the operation out in full here:

```
dSp = Measure('dS', metadata={'quadrature_degree': 1})
dsp = Measure('ds', metadata={'quadrature_degree': 1})

n = FacetNormal(mesh)
t = as_vector((-n[1], n[0]))

inner_e = lambda x, y: (inner(x, t)*inner(y, t))('+')*dSp + \
                       (inner(x, t)*inner(y, t))('-')*dSp + \
                       (inner(x, t)*inner(y, t))*dsp

Pi_R = inner_e(gamma - R_gamma_, p_)
```

We can now define our Lagrangian for the complete system:

```
Pi = psi_b*dx + psi_s*dx + Pi_R - W_ext
```

and derive our Jacobian and residual automatically using the standard UFL `derivative()` function:

```
dPi = derivative(Pi, q_, q_t)
J = derivative(dPi, q_, q)
```

We now assemble our system using the additional projected assembly in *fenics_shells*.

By passing `Q_P` as the first argument to `assemble()`, we state that we want to assemble a Matrix or Vector from the forms on the `ProjectedFunctionSpace` `Q`, rather than the full FunctionSpace `Q_F`:

```
A, b = assemble(Q, J, -dPi)
```

Note that from this point on, we are working with objects on the `ProjectedFunctionSpace` `Q`. We now apply homogeneous Dirichlet boundary conditions:

```
def all_boundary(x, on_boundary):
    return on_boundary

bcs = [DirichletBC(Q, Constant((0.0, 0.0, 0.0)), all_boundary)]

for bc in bcs:
    bc.apply(A, b)
```

and solve the linear system of equations:

```
q_p_ = Function(Q)
solver = PETScLUSolver("mumps")
solver.solve(A, q_p_.vector(), b)
```

We can now reconstruct the full space solution (i.e. the fields γ_R and p) using the method `reconstruct_full_space()`:

```
reconstruct_full_space(q_, q_p_, J, -dPi)
```

This step is not necessary if you are only interested in the primal fields w and θ.

Finally we output the results to XDMF to the directory `output/`:

```
save_dir = "output/"
theta_h, w_h, R_gamma_h, p_h = q_.split()
fields = {"theta": theta_h, "w": w_h, "R_gamma": R_gamma_h, "p": p_h}
for name, field in fields.items():
    field.rename(name, name)
    field_file = XDMFFile("%s/%s.xdmf" % (save_dir, name))
    field_file.write(field)
```

The resulting `output/*.xdmf` files can be viewed using Paraview.

Appendix

For the clamped problem we have the following regularity for our two fields, $\theta \in [H_0^1(\Omega)]^2$ and $w \in [H_0^1(\Omega)]^2$ where $H_0^1(\Omega)$ is the usual Sobolev space of functions with square integrable first derivatives that vanish on the boundary. If we then take ∇w we have the result $\nabla w \in H_0(\mathrm{rot}; \Omega)$ which is the Sobolev space of vector-valued functions with square integrable rot whose tangential component $\nabla w \cdot t$ vanishes on the boundary. Functions $\nabla w \in H_0(\mathrm{rot}; \Omega)$ are rot free, in that $\mathrm{rot}(\nabla w) = 0$.

Let's look at our expression for the shear strain vector in light of these new results. In the thin-plate limit $t \to 0$, we would like to recover our the standard Kirchhoff-Love problem where we do not have transverse shear strains $\gamma \to 0$ at all. In a finite element context, where we have discretised fields w_h and θ_h we then would like:

$$\gamma(\theta_h, w_h) := \nabla w_h - \theta_h = 0 \quad t \to 0 \; \forall x \in \Omega$$

If we think about using first-order piecewise linear polynomial finite elements for both fields, then we are requiring that piecewise constant functions (∇w_h) are equal to piecewise linear functions (θ_h) ! This is strong requirement, and is the root of the famous shear-locking problem. The trick of the Durán-Liberman approach is recognising that

by modifying the rotation field at the discrete level by applying a special operator R_h that takes the rotations to the conforming space $\mathrm{NED}_1 \subset H_0(\mathrm{rot};\Omega)$ for the shear strains that we previously identified:

$$R_h : H_0^1(\Omega) \to H_0(\mathrm{rot};\Omega),$$

we can 'unlock' the element. With this reduction operator applied as follows:

$$\gamma(\theta_h, w_h) := R_h(\nabla w_h - \theta_h = 0) \quad t \to 0 \; \forall x \in \Omega$$

our requirement of vanishing shear strains can actually hold. This is the basic mathematical idea behind all MITC approaches, of which the Durán-Liberman approach is a subclass

Unit testing

```
def test_close():
    import numpy as np
    assert(np.isclose(w_h((0.5, 0.5)), 1.285E-6, atol=1E-3, rtol=1E-3))
```

References

[1] R. Duran, E. Liberman. On mixed finite element methods for the Reissner-Mindlin plate model. Mathematics of Computation. Vol. 58. No. 198. 561-573. 1992.

3.1.2 Clamped semi-cylindrical Naghdi shell under point load

This demo is implemented in the single Python file `demo_nonlinear-naghdi-cylindrical.py`.

This demo program solves the nonlinear Naghdi shell equations for a semi-cylindrical shell loaded by a point force. This problem is a standard reference for testing shell finite element formulations, see [1]. The numerical locking issue is cured using enriched finite element including cubic bubble shape functions and Partial Selective Reduced Integration [2].

To follow this demo you should know how to:

- Define a `MixedElement` and a `FunctionSpace` from it.
- Write variational forms using the Unified Form Language.
- Automatically derive Jacobian and residuals using derivative().
- Apply Dirichlet boundary conditions using DirichletBC and apply().
- Solve non-linear problems using NonlinearProblem.
- Output data to XDMF files with XDMFFile.

This demo then illustrates how to:

- Define and solve a nonlinear Naghdi shell problem with a curved stress-free configuration given as analytical expression in terms of two curvilinear coordinates.
- Use the PSRI approach to simultaneously cure shear- and membrane-locking issues.

We start with importing the required modules, setting `matplolib` as plotting backend, and generically set the integration order to 4 to avoid the automatic setting of FEniCS which would lead to unreasonably high integration orders for complex forms.

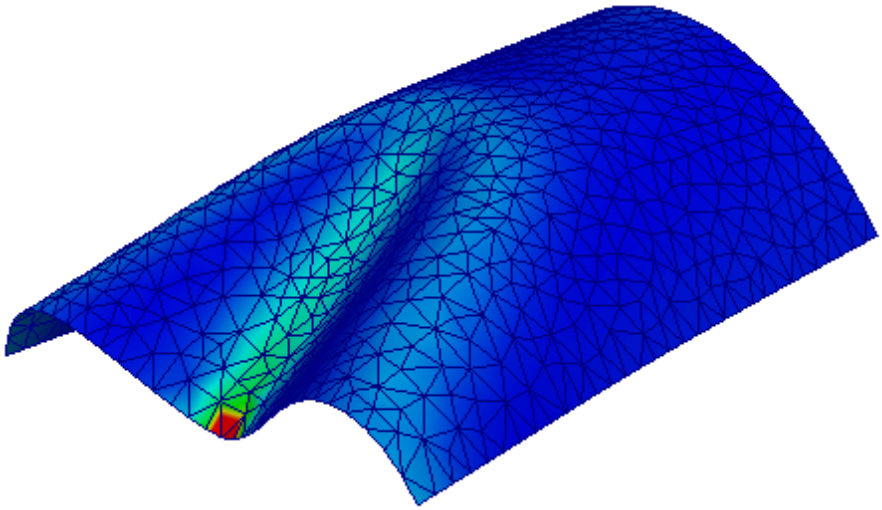


Fig. 3.2: Shell deformed configuration.

```
import os, sys

import numpy as np
import matplotlib.pyplot as plt

from dolfin import *
from ufl import Index
from mshr import *
from mpl_toolkits.mplot3d import Axes3D

parameters["form_compiler"]["quadrature_degree"] = 4

output_dir = "output/"
if not os.path.exists(output_dir):
    os.makedirs(output_dir)
```

We consider a semi-cylindrical shell of radius ρ and axis length L. The shell is made of a linear elastic isotropic homogeneous material with Young modulus E and Poisson ratio ν. The (uniform) shell thickness is denoted by t. The Lamé moduli λ, μ are introduced to write later the 2D constitutive equation in plane-stress:

```
rho = 1.016
L = 3.048
E, nu = 2.0685E7, 0.3
mu = E/(2.0*(1.0 + nu))
lmbda = 2.0*mu*nu/(1.0 - 2.0*nu)
t = Constant(0.03)
```

The midplane of the initial (stress-free) configuration Φ_o of the shell is given in the form of an analytical expression

$$\phi_0 : x \in \omega \subset R^2 \rightarrow \phi_0(x) \in \Phi_o \subset \mathcal{R}^3$$

in terms of the curvilinear coordinates x. In the specific case we adopt the cylindrical coordinates x_0 and x_1 representing the angular and axial coordinates, respectively. Hence we mesh the two-dimensional domain $\omega \equiv [0, L_y] \times [-\pi/2, \pi/2]$.

```
P1, P2 = Point(-np.pi/2., 0.), Point(np.pi/2., L)
ndiv = 21
mesh = generate_mesh(Rectangle(P1, P2), ndiv)
plot(mesh); plt.xlabel(r"$x_0$"); plt.ylabel(r"$x_1$")
plt.savefig("output/mesh.png")
```

We provide the analytical expression of the initial shape as an `Expression` that we represent on a suitable `FunctionSpace` (here P_2, but other are choices are possible):

```
initial_shape = Expression(('r*sin(x[0])','x[1]','r*cos(x[0])'), r=rho, degree = 4)
V_phi =  FunctionSpace(mesh, VectorElement("P", triangle, degree = 2, dim = 3))
phi0 = project(initial_shape, V_phi)
```

Given the midplane, we define the corresponding unit normal as below and project on a suitable function space (here P_1 but other choices are possible):

```
def normal(y):
    n = cross(y.dx(0), y.dx(1))
    return n/sqrt(inner(n,n))

V_normal = FunctionSpace(mesh, VectorElement("P", triangle, degree = 1, dim = 3))
n0 = project(normal(phi0), V_normal)
```

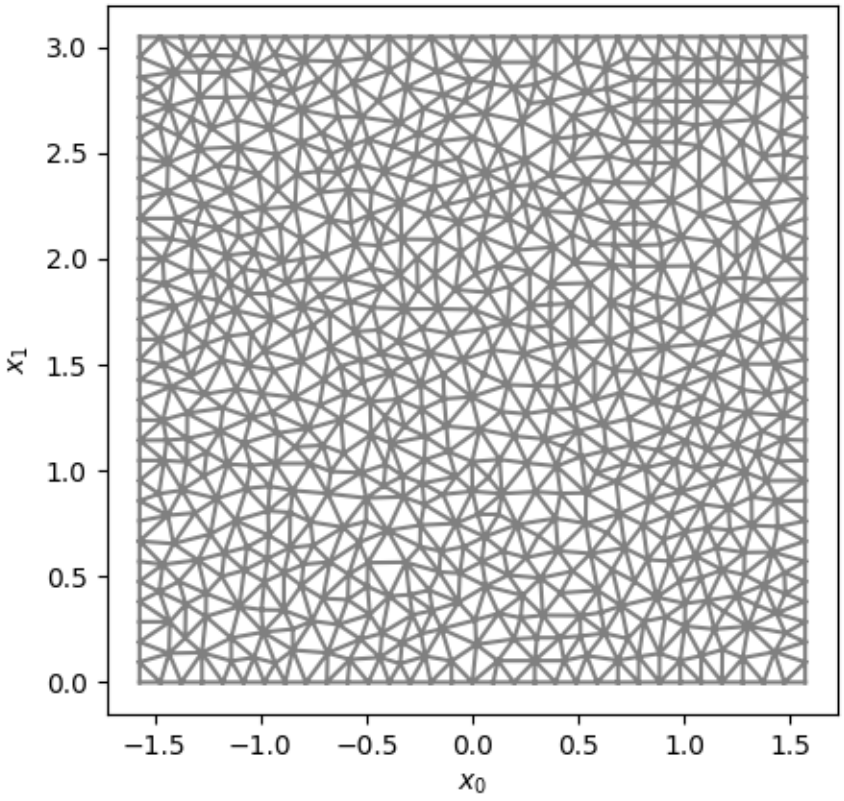


Fig. 3.3: Discretisation of the parametric domain.

The kinematics of the Nadghi shell model is defined by the following vector fields :

- ϕ: the position of the midplane, or the displacement from the reference configuration $u = \phi - \phi_0$:
- d: the director, a unit vector giving the orientation of the microstructure

We parametrize the director field by two angles, which correspond to spherical coordinates, so as to explicitly resolve the unit norm constraint (see [3]):

```
def director(beta):
    return as_vector([sin(beta[1])*cos(beta[0]), -sin(beta[0]),␣
↪cos(beta[1])*cos(beta[0])])
```

We assume that in the initial configuration the director coincides with the normal. Hence, we can define the angles β: for the initial configuration as follows:

```
beta0_expression = Expression(["atan2(-n[1], sqrt(pow(n[0],2) + pow(n[2],2)))",
                               "atan2(n[0],n[2])"], n = n0, degree=4)

V_beta = FunctionSpace(mesh, VectorElement("P", triangle, degree = 2, dim = 2))
beta0 = project(beta0_expression, V_beta)
```

The director in the initial configuration is then written as

```
d0 = director(beta0)
```

We can visualize the shell shape and its normal with this utility function:

```
def plot_shell(y,n=None):
    y_0, y_1, y_2 = y.split(deepcopy=True)
    fig = plt.figure()
    ax = fig.gca(projection='3d')
    ax.plot_trisurf(y_0.compute_vertex_values(),
                    y_1.compute_vertex_values(),
                    y_2.compute_vertex_values(),
                    triangles=y.function_space().mesh().cells(),
                    linewidth=1, antialiased=True, shade = False)
    if n:
        n_0, n_1, n_2 = n.split(deepcopy=True)
        ax.quiver(y_0.compute_vertex_values(),
              y_1.compute_vertex_values(),
              y_2.compute_vertex_values(),
              n_0.compute_vertex_values(),
              n_1.compute_vertex_values(),
              n_2.compute_vertex_values(),
              length = .2, color = "r")
    ax.view_init(elev=20, azim=80)
    plt.xlabel(r"$x_0$")
    plt.ylabel(r"$x_1$")
    plt.xticks([-1,0,1])
    plt.yticks([0,pi/2])
    return ax

plot_shell(phi0, project(d0, V_normal))
plt.savefig("output/initial_configuration.png")
```

In our 5-parameter Naghdi shell model the configuration of the shell is assigned by

- the 3-component vector field u: representing the displacement with respect to the initial configuration ϕ_0:

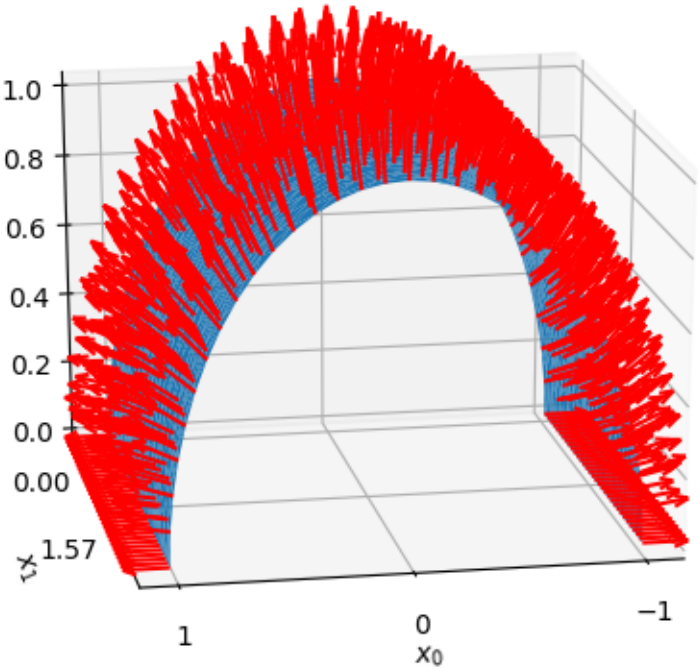


Fig. 3.4: Shell initial shape and normal.

- the 2-component vector field β: representing the angle variation of the director d: with respect to the initial configuration

Following [1], we use a $[P_2 + B_3]$ element for u and a $[CG_2]^2$ element for $beta$, and collect them in the state vector $q = (u, \beta)$:

```
P2 = FiniteElement("Lagrange", triangle, degree = 2)
bubble = FiniteElement("B", triangle, degree = 3)
enriched = P2 + bubble

element = MixedElement([VectorElement(enriched, dim=3), VectorElement(P2, dim=2)])

Q = FunctionSpace(mesh, element)
```

Then, we define `Function`, `TrialFunction` and `TestFunction` objects to express the variational forms and we split them into each individual component function:

```
q_, q, q_t = Function(Q), TrialFunction(Q), TestFunction(Q)
u_, beta_ = split(q_)
```

The gradient of the transformation and the director in the current configuration are given by:

```
F = grad(u_) + grad(phi0)
d = director(beta_ + beta0)
```

With the following definition of the natural metric and curvature

```
a0 = grad(phi0).T*grad(phi0)
b0 = -0.5*(grad(phi0).T*grad(d0) + grad(d0).T*grad(phi0))
```

The membrane, bending, and shear strain measures of the Naghdi model are defined by:

```
e = lambda F: 0.5*(F.T*F - a0)
k = lambda F, d: -0.5*(F.T*grad(d) + grad(d).T*F) - b0
gamma = lambda F, d: F.T*d - grad(phi0).T*d0
```

Using curvilinear coordinates, and denoting by `a0_contra` the contravariant components of the metric tensor $a_0^{\alpha\beta}$ (in the initial curved configuration) the constitutive equation is written in terms of the matrix A below, representing the contravariant components of the constitutive tensor for isotropic elasticity in plane stress (see *e.g.* [4]). We use the index notation offered by UFL to express operations between tensors:

```
a0_contra = inv(a0)
j0 = det(a0)

i, j, l, m = Index(), Index(), Index(), Index()
A_ = as_tensor((((2.0*lmbda*mu)/(lmbda + 2.0*mu))*a0_contra[i,j]*a0_contra[l,m]
                + 1.0*mu*(a0_contra[i,l]*a0_contra[j,m] + a0_contra[i,m]*a0_contra[j,
↪l]))
                ,[i,j,l,m])
```

The normal stress N, bending moment M, and shear stress T tensors are (they are purely Lagrangian stress measures, similar to the so called 2nd Piola stress tensor in 3D elasticity):

```
N = as_tensor(t*A_[i,j,l,m]*e(F)[l,m], [i,j])
M = as_tensor((t**3/12.0)*A_[i,j,l,m]*k(F,d)[l,m],[i,j])
T = as_tensor(t*mu*a0_contra[i,j]*gamma(F,d)[j], [i])
```

Hence, the contributions to the elastic energy density due to membrane, ψ_m, bending, ψ_b, and shear, ψ_s are (they are per unit surface in the initial configuration):

```
psi_m = 0.5*inner(N, e(F))
psi_b = 0.5*inner(M, k(F,d))
psi_s = 0.5*inner(T, gamma(F,d))
```

Shear and membrane locking is treated using the partial reduced selective integration proposed in Arnold and Brezzi [2]. In this approach shear and membrane energy are splitted as a sum of two contributions weighted by a factor α. One of the two contributions is integrated with a reduced integration. While [1] suggests a 1-point reduced integration, we observed that this leads to spurious modes in the present case. We use then 2×2-points Gauss integration for a portion $1 - \alpha$ of the energy, whilst the rest is integrated with a 4×4 scheme. We further refine the approach of [1] by adopting an optimized weighting factor $\alpha = (t/h)^2$, where h is the mesh size.

```
dx_h = dx(metadata={'quadrature_degree': 2})
h = CellDiameter(mesh)
alpha = project(t**2/h**2, FunctionSpace(mesh,'DG',0))

Pi_PSRI = psi_b*sqrt(j0)*dx + alpha*psi_m*sqrt(j0)*dx + alpha*psi_s*sqrt(j0)*dx + (1.
↪0 - alpha)*psi_s*sqrt(j0)*dx_h + (1.0 - alpha)*psi_m*sqrt(j0)*dx_h
```

Hence the total elastic energy and its first and second derivatives are

```
Pi = Pi_PSRI
dPi = derivative(Pi, q_, q_t)
J = derivative(dPi, q_, q)
```

The boundary conditions prescribe a full clamping on the top boundary, while on the left and the right side the normal component of the rotation and the transverse displacement are blocked:

```
up_boundary = lambda x, on_boundary: x[1] <= 1.e-4 and on_boundary
leftright_boundary = lambda x, on_boundary: near(abs(x[0]), pi/2., 1.e-6)  and on_
↪boundary

bc_clamped = DirichletBC(Q, project(q_, Q), up_boundary)
bc_u = DirichletBC(Q.sub(0).sub(2), project(Constant(0.), Q.sub(0).sub(2).collapse()),
↪ leftright_boundary)
bc_beta = DirichletBC(Q.sub(1).sub(1), project(q_[4], Q.sub(1).sub(0).collapse()),␣
↪leftright_boundary)
bcs = [bc_clamped, bc_u, bc_beta]
```

The loading is exerted by a point force applied at the midpoint of the bottom boundary. This is implemented using the `PointSource` in FEniCS and defining a custom `NonlinearProblem`:

```
class NonlinearProblemPointSource(NonlinearProblem):

    def __init__(self, L, a, bcs):
        NonlinearProblem.__init__(self)
        self.L = L
        self.a = a
        self.bcs = bcs
        self.P = 0.0

    def F(self, b, x):
        assemble(self.L, tensor=b)
        point_source = PointSource(self.bcs[0].function_space().sub(0).sub(2),␣
↪Point(0.0, L), self.P)
        point_source.apply(b)
```

```
        for bc in self.bcs:
            bc.apply(b, x)

    def J(self, A, x):
        assemble(self.a, tensor=A)
        for bc in self.bcs:
            bc.apply(A, x)

problem = NonlinearProblemPointSource(dPi, J, bcs)
```

We use a standard Newton solver and setup the files for the writing the results to disk:

```
solver = NewtonSolver()
output_dir = "output/"
file_phi = File(output_dir + "configuration.pvd")
file_energy = File(output_dir + "energy.pvd")
```

Finally, we can solve the quasi-static problem, incrementally increasing the loading from 0 to 2000 N:

```
P_values = np.linspace(0.0, 2000.0, 40)
displacement = 0.*P_values
q_.assign(project(Constant((0,0,0,0,0)), Q))

for (i, P) in enumerate(P_values):
    problem.P = P
    (niter,cond) = solver.solve(problem, q_.vector())

    phi = project(u_ + phi0, V_phi)
    displacement[i] = phi(0.0, L)[2] - phi0(0.0, L)[2]

    phi.rename("phi", "phi")
    file_phi << (phi, P)
    print("Increment %d of %s. Converged in %2d iterations. P:  %.2f, Displ: %.2f"
↪%(i, P_values.size,niter,P, displacement[i]))

    en_function = project(psi_m + psi_b + psi_s, FunctionSpace(mesh, 'Lagrange', 1))
    en_function.rename("Elastic Energy", "Elastic Energy")
    file_energy << (en_function,P)
```

We can plot the final configuration of the shell:

```
plot_shell(phi)
plt.savefig("output/finalconfiguration.png")
```

The results for the transverse displacement at the point of application of the force are validated against a standard reference from the literature, obtained using Abaqus S4R element and a structured mesh of 40×40 elements, see [1]:

```
plt.figure()
reference_Sze = np.array([
    1.e-2*np.array([0., 5.421, 16.1, 22.195, 27.657, 32.7, 37.582, 42.633,
    48.537, 56.355, 66.410, 79.810, 94.669, 113.704, 124.751, 132.653,
    138.920, 144.185, 148.770, 152.863, 156.584, 160.015, 163.211,
    166.200, 168.973, 171.505]),
    2000.*np.array([0., .05, .1, .125, .15, .175, .2, .225, .25, .275, .3,
    .325, .35, .4, .45, .5, .55, .6, .65, .7, .75, .8, .85, .9, .95, 1.])
    ])
plt.plot(-np.array(displacement), P_values, label='fenics-shell %s divisions (AB)'
↪%ndiv)
```

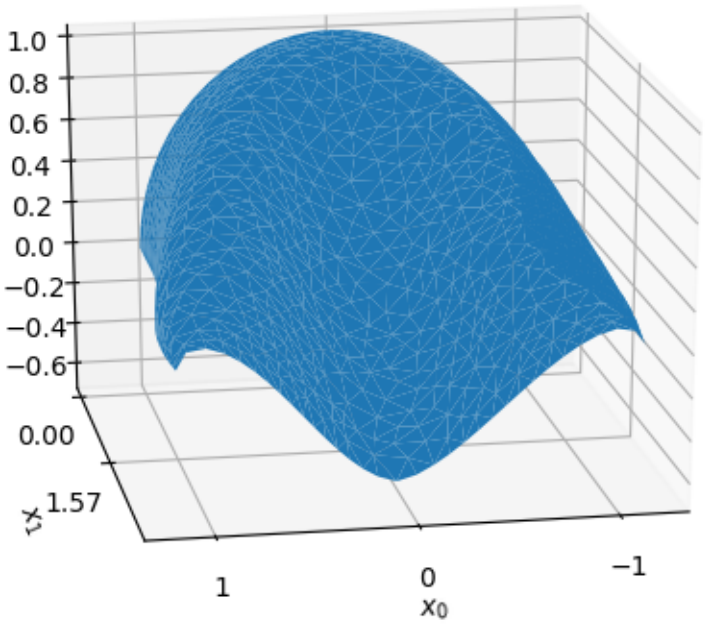


Fig. 3.5: Shell deformed shape.

```
plt.plot(*reference_Sze, "or", label='Sze (Abaqus S4R)')
plt.xlabel("Displacement (mm)")
plt.ylabel("Load (N)")
plt.legend()
plt.grid()
plt.savefig("output/comparisons.png")
```

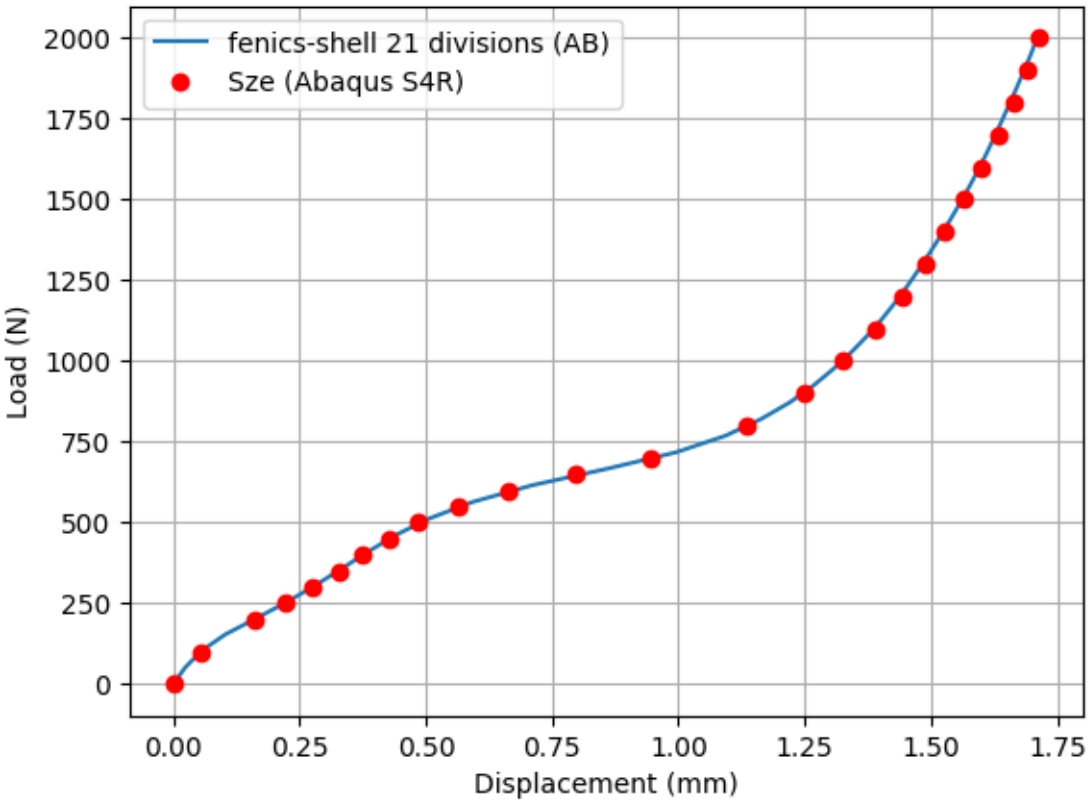


Fig. 3.6: Comparison with reference solution.

References

[1] K. Sze, X. Liu, and S. Lo. Popular benchmark problems for geometric nonlinear analysis of shells. Finite Elements in Analysis and Design, 40(11):1551 – 1569, 2004.

[2] D. Arnold and F.Brezzi, Mathematics of Computation, 66(217): 1-14, 1997. https://www.ima.umn.edu/~arnold//papers/shellelt.pdf

[3] P. Betsch, A. Menzel, and E. Stein. On the parametrization of finite rotations in computational mechanics: A classification of concepts with application to smooth shells. Computer Methods in Applied Mechanics and Engineering, 155(3):273 – 305, 1998.

[4] P. G. Ciarlet. An introduction to differential geometry with applications to elasticity. Journal of Elasticity, 78-79(1-3):1–215, 2005.

3.2 Other demos

Linear Reissner-Mindlin plate problems using the Durán-Liberman reduction operator (MITC) and MITC7 to cure shear-locking:

3.2.1 Simply supported Reissner-Mindlin plate

This demo is implemented in the single Python file `demo_reissner-mindlin-simply-supported.py`.

This demo program solves the out-of-plane Reissner-Mindlin equations on the unit square with uniform transverse loading with simply supported boundary conditions.

The first part of the demo is similar to the demo *Clamped Reissner-Mindlin plate under uniform load*.

```
from dolfin import *
from fenics_shells import *

mesh = UnitSquareMesh(64, 64)

element = MixedElement([VectorElement("Lagrange", triangle, 2),
                        FiniteElement("Lagrange", triangle, 1),
                        FiniteElement("N1curl", triangle, 1),
                        FiniteElement("N1curl", triangle, 1)])

U = ProjectedFunctionSpace(mesh, element, num_projected_subspaces=2)
U_F = U.full_space

u_ = Function(U_F)
theta_, w_, R_gamma_, p_ = split(u_)
u = TrialFunction(U_F)
u_t = TestFunction(U_F)

E = Constant(10920.0)
nu = Constant(0.3)
kappa = Constant(5.0/6.0)
t = Constant(0.0001)

k = sym(grad(theta_))

D = (E*t**3)/(24.0*(1.0 - nu**2))
psi_M = D*((1.0 - nu)*tr(k*k) + nu*(tr(k))**2)

psi_T = ((E*kappa*t)/(4.0*(1.0 + nu)))*inner(R_gamma_, R_gamma_)

f = Constant(1.0)
W_ext = inner(f*t**3, w_)*dx

gamma = grad(w_) - theta_
```

We instead use the *fenics_shells* provided `inner_e()` function:

```
L_R = inner_e(gamma - R_gamma_, p_)
L = psi_M*dx + psi_T*dx + L_R - W_ext

F = derivative(L, u_, u_t)
J = derivative(F, u_, u)
```

```
A, b = assemble(U, J, -F)
```

and apply simply-supported boundary conditions:

```
def all_boundary(x, on_boundary):
    return on_boundary

def left(x, on_boundary):
    return on_boundary and near(x[0], 0.0)

def right(x, on_boundary):
    return on_boundary and near(x[0], 1.0)

def bottom(x, on_boundary):
    return on_boundary and near(x[1], 0.0)

def top(x, on_boundary):
    return on_boundary and near(x[1], 1.0)

# Simply supported boundary conditions.
bcs = [DirichletBC(U.sub(1), Constant(0.0), all_boundary),
       DirichletBC(U.sub(0).sub(0), Constant(0.0), top),
       DirichletBC(U.sub(0).sub(0), Constant(0.0), bottom),
       DirichletBC(U.sub(0).sub(1), Constant(0.0), left),
       DirichletBC(U.sub(0).sub(1), Constant(0.0), right)]

for bc in bcs:
    bc.apply(A, b)
```

and solve the linear system of equations before writing out the results to files in `output/`:

```
u_p_ = Function(U)
solver = PETScLUSolver("mumps")
solver.solve(A, u_p_.vector(), b)
reconstruct_full_space(u_, u_p_, J, -F)

save_dir = "output/"
theta, w, R_gamma, p = u_.split()
fields = {"theta": theta, "w": w, "R_gamma": R_gamma, "p": p}
for name, field in fields.items():
    field.rename(name, name)
    field_file = XDMFFile("%s/%s.xdmf" % (save_dir, name))
    field_file.write(field)
```

We check the result against an analytical solution calculated using a series expansion:

```
from fenics_shells.analytical.simply_supported import Displacement

w_e = Displacement(degree=3)
w_e.t = t.values()
w_e.E = E.values()
w_e.p = f.values()*t.values()**3
w_e.nu = nu.values()

print("Numerical out-of-plane displacement at centre: %.4e" % w((0.5, 0.5)))
print("Analytical out-of-plane displacement at centre: %.4e" % w_e((0.5, 0.5)))
```

Unit testing

```
def test_close():
    import numpy as np
    assert(np.isclose(w((0.5, 0.5)), w_e((0.5, 0.5)), atol=1E-3, rtol=1E-3))
```

3.2.2 Clamped Reissner-Mindlin plate with MITC7

This demo is implemented in a single Python file `demo_reissner-mindlin-mitc7.py`.

This demo program solves the out-of-plane Reissner-Mindlin equations on the unit square with uniform transverse loading with fully clamped boundary conditions. The MITC7 reduction operator is used, instead of the Durán Liberman one as in the *Clamped Reissner-Mindlin plate under uniform load* demo.

We express the MITC7 projection operator in the Unified Form Language. Lagrange multipliers are required on the edge and in the interior of each element to tie together the shear strain calculated from the primal variables and the conforming shear strain field.

Unlike the other Reissner-Mindlin documented demos, e.g *Clamped Reissner-Mindlin plate under uniform load*, where the FEniCS-Shells `assemble()` function is used to eliminate all of the additional degrees of freedom at assembly time, we keep the full problem with all of the auxilliary variables here.

We begin as usual by importing the required modules:

```
from dolfin import *
from ufl import EnrichedElement, RestrictedElement
from fenics_shells import *
```

and creating a mesh:

```
mesh = UnitSquareMesh(32, 32)
```

The MITC7 element for the Reissner-Mindlin plate problem consists of:

- a second-order scalar-valued element for the transverse displacement field $w \in \mathrm{CG}_2$,
- second-order bubble-enriched vector-valued element for the rotation field $\theta \in [\mathrm{CG}_2]^2$,
- the reduced shear strain γ_R is discretised in the space NED_2, the second-order vector-valued Nédélec element of the first kind,
- and two Lagrange multiplier fields p and r which tie together the shear strain calculated from the primal variables $\gamma = \nabla w - \theta$ and the reduced shear strain γ_R. p and r are discretised in the space NED_2 restricted to the element edges and the element interiors, respectively.

The final element definition is:

```
element = MixedElement([FiniteElement("Lagrange", triangle, 2),
                        VectorElement(EnrichedElement(FiniteElement("Lagrange",␣
↪triangle, 2) + FiniteElement("Bubble", triangle, 3))),
                        FiniteElement("N1curl", triangle, 2),
                        RestrictedElement(FiniteElement("N1curl", triangle, 2), "edge
↪"),
                        VectorElement("DG", triangle, 0)])
```

The definition of the bending and shear energies and external work are standard and identical to those in *Clamped Reissner-Mindlin plate under uniform load*:

```
U = FunctionSpace(mesh, element)

u_ = Function(U)
w_, theta_, R_gamma_, p_, r_  = split(u_)
u = TrialFunction(U)
u_t = TestFunction(U)

E = Constant(10920.0)
nu = Constant(0.3)
kappa = Constant(5.0/6.0)
t = Constant(0.001)

k = sym(grad(theta_))
D = (E*t**3)/(24.0*(1.0 - nu**2))
psi_M = D*((1.0 - nu)*tr(k*k) + nu*(tr(k))**2)

psi_T = ((E*kappa*t)/(4.0*(1.0 + nu)))*inner(R_gamma_, R_gamma_)

f = Constant(1.0)
W_ext = inner(f*t**3, w_)*dx
```

We require that the shear strain calculated using the displacement unknowns $\gamma = \nabla w - \theta$ be equal, in a weak sense, to the conforming shear strain field $\gamma_R \in \mathrm{NED}_2$ that we used to define the shear energy above. We enforce this constraint using *two* Lagrange multiplier fields field $p \in \mathrm{NED}_2$ restricted to the edges and $r \in \mathrm{NED}_2$ restricted to the interior of the element. We can write the Lagrangian of this constraint as:

$$L_R(\gamma, \gamma_R, p, r) = \int_e (\{\gamma_R - \gamma\} \cdot t) \cdot (p \cdot t) \ \mathrm{d}s + \int_T \{\gamma_R - \gamma\} \cdot r \ \mathrm{d}x$$

where T are all cells in the mesh, e are all of edges of the cells in the mesh and t is the tangent vector on each edge.

This operator can be written in UFL as:

```
n = FacetNormal(mesh)
t = as_vector((-n[1], n[0]))

gamma = grad(w_) - theta_

inner_e = lambda x, y: (inner(x, t)*inner(y, t))('+')*dS + \
                       (inner(x, t)*inner(y, t))*ds

inner_T = lambda x, y: inner(x, y)*dx

L_R = inner_e(gamma - R_gamma_, p_) + inner_T(gamma - R_gamma_, r_)
```

We set homogeneous Dirichlet conditions for the reduced shear strain, edge Lagrange multipliers, transverse displacement and rotations:

```
def all_boundary(x, on_boundary):
    return on_boundary

bcs = [DirichletBC(U.sub(0), Constant(0.0), all_boundary),
       DirichletBC(U.sub(1), project(Constant((0.0, 0.0)), U.sub(1).collapse()), all_
↪boundary),
       DirichletBC(U.sub(2), Constant((0.0, 0.0)), all_boundary),
       DirichletBC(U.sub(3), Constant((0.0, 0.0)), all_boundary)]
```

Before assembling in the normal way:

```
L = psi_M*dx + psi_T*dx + L_R - W_ext
F = derivative(L, u_, u_t)
J = derivative(F, u_, u)

A, b = assemble_system(J, -F, bcs=bcs)

solver = PETScLUSolver("mumps")
solver.solve(A, u_.vector(), b)

w_, theta_, R_gamma_, p_, r_ = u_.split()
```

Finally we output the results to XDMF to the directory `output/`:

```
save_dir = "output"
fields = {"theta": theta_, "w": w_}
for name, field in fields.items():
    field.rename(name, name)
    field_file = XDMFFile("%s/%s.xdmf"%(save_dir,name))
    field_file.write(field)
```

The resulting `output/*.xdmf` files can be viewed using Paraview.

Unit testing

```
def test_close():
    import numpy as np
    assert(np.isclose(w_((0.5, 0.5)), 1.265E-6, atol=1E-3, rtol=1E-3))
```

Fourth-order Kirchhoff-Love plate problem with a Continuous-Discontinuous Galerkin method:

3.2.3 Clamped Kirchhoff-Love plate

This demo is implemented in a single Python file `demo_kirchhoff-love-clamped.py`.

This demo program solves the out-of-plane Kirchhoff-Love equations on the unit square with uniform transverse loading and fully clamped boundary conditions.

We use the Continuous/Discontinuous Galerkin (CDG) formulation to allow the use of H^1-conforming elements for this fourth-order, or H^2-type problem. This demo is very similar to the Biharmonic equation demo in the main DOLFIN repository, and as such we recommend reading that page first. The main differences are:

- we express the stabilisation term in terms of a Lagrangian functional rather than as a bilinear form,
- the Lagrangian function in turn is expressed in terms of the bending moment and rotations rather than the primal field variable,
- and we show how to place Dirichlet boundary conditions on the first-derivative of the solution (the rotations) using a weak approach.

We begin as usual by importing the required modules:

```
from dolfin import *
from fenics_shells import *
```

and creating a mesh:

```
mesh = UnitSquareMesh(64, 64)
```

We use a second-order scalar-valued element for the transverse displacement field $w \in CG_2$:

```
element_W = FiniteElement("Lagrange", triangle,  2)
W = FunctionSpace(mesh, element_W)
```

and then define `Function`, `TrialFunction` and `TestFunction` objects to express the variational form:

```
w_ = Function(W)
w = TrialFunction(W)
w_t = TestFunction(W)
```

We take constant material properties throughout the domain:

```
E = Constant(10920.0)
nu = Constant(0.3)
t = Constant(1.0)
```

The Kirchhoff-Love model, unlike the Reissner-Mindlin model, is a *rotation-free* model: the rotations θ do not appear explicitly as degrees of freedom. Instead, the rotations of the Kirchhoff-Love model are calculated from the transverse displacement field as:

$$\theta = \nabla w$$

which can be expressed in UFL as:

```
theta = grad(w_)
```

The bending tensor can then be calculated from the derived rotation field in exactly the same way as for the Reissner-Mindlin model:

$$k = \frac{1}{2}(\nabla\theta + (\nabla\theta)^T)$$

or in UFL:

```
k = variable(sym(grad(theta)))
```

The function `variable()` annotation is important and will allow us to take differentiate with respect to the bending tensor `k` to derive the bending moment tensor, as we will see below.

Again, identically to the Reissner-Mindlin model we can calculate the bending energy density as:

```
D = (E*t**3)/(24.0*(1.0 - nu**2))
psi_M = D*((1.0 - nu)*tr(k*k) + nu*(tr(k))**2)
```

For the definition of the CDG stabilisation terms and the (weak) enforcement of the Dirichlet boundary conditions on the rotation field, we need to explicitly derive the moment tensor M. Following standard arguments in elasticity, a stress measure (here, the moment tensor) can be derived from the bending energy density by taking its derivative with respect to the strain measure (here, the bending tensor):

$$M = \frac{\partial \psi_M}{\partial k}$$

or in UFL:

```
M = diff(psi_M, k)
```

We now move onto the CDG stabilisation terms.

Consider a triangulation $\mathcal{T}$ of the domain ω with the set of interior edges is denoted $\mathcal{E}_h^{\text{int}}$. Normals to the edges of each facet are denoted n. Functions evaluated on opposite sides of a facet are indicated by the subscripts $+$ and $-$.

The Lagrangian formulation of the CDG stabilisation term is then:

$$L_{\text{CDG}}(w) = \sum_{E \in \mathcal{E}_h^{\text{int}}} \int_E -[\![\theta]\!] \cdot \langle M \cdot (n \otimes n) \rangle + \frac{1}{2} \frac{\alpha}{\langle h_E \rangle} \langle \theta \cdot n \rangle \cdot \langle \theta \cdot n \rangle \ \mathrm{d}s$$

Furthermore, $\langle u \rangle = \frac{1}{2}(u_+ + u_-)$ operator, $[\![u]\!] = u_+ \cdot n_+ + u_- \cdot n_-$, $\alpha \geq 0$ is a penalty parameter and h_E is a measure of the cell size. We choose the penalty parameter to be on the order of the norm of the bending stiffness matrix $\frac{Et^3}{12}$.

This can be written in UFL as:

```
alpha = E*t**3
h = CellDiameter(mesh)
h_avg = (h('+') + h('-'))/2.0

n = FacetNormal(mesh)

M_n = inner(M, outer(n, n))

L_CDG = -inner(jump(theta, n), avg(M_n))*dS + \
           (1.0/2.0)*(alpha('+')/h_avg)*inner(jump(theta, n), jump(theta, n))*dS
```

We now define our Dirichlet boundary conditions on the transverse displacement field:

```
class AllBoundary(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary

# Boundary conditions on displacement
all_boundary = AllBoundary()
bcs_w = [DirichletBC(W, Constant(0.0), all_boundary)]
```

Because the rotation field θ does not enter our weak formulation directly, we must weakly enforce the Dirichlet boundary condition on the derivatives of the transverse displacement ∇w.

We begin by marking the exterior facets of the mesh where we want to apply boundary conditions on the rotation:

```
facet_function = MeshFunction("size_t", mesh, mesh.geometry().dim() - 1)
facet_function.set_all(0)
all_boundary.mark(facet_function, 1)
```

and then define an exterior facet `Measure` object from that subdomain data:

```
ds = Measure("ds")(subdomain_data=facet_function)
```

In this example, we would like $\theta_d = 0$ everywhere on the boundary:

```
theta_d = Constant((0.0, 0.0))
```

The definition of the exterior facets and Dirichlet rotation field were trivial in this demo, but you could extend this code straightforwardly to non-homogeneous Dirichlet conditions.

The weak boundary condition enforcement term can be written:

$$L_{\mathrm{BC}}(w) = \sum_{E \in \mathcal{E}_h^{\mathrm{D}}} \int_E -\theta_e \cdot (M \cdot (n \otimes n)) + \frac{1}{2} \frac{\alpha}{h_E} (\theta_e \cdot n) \cdot (\theta_e \cdot n) \, \mathrm{d}s$$

where $\theta_e = \theta - \theta_d$ is the effective rotation field, and $\mathcal{E}_h^{\mathrm{D}}$ is the set of all exterior facets of the triangulation $\mathcal{T}$ where we would like to apply Dirichlet boundary conditions, or in UFL:

```
theta_effective = theta - theta_d
L_BC = -inner(inner(theta_effective, n), M_n)*ds(1) + \
        (1.0/2.0)*(alpha/h)*inner(inner(theta_effective, n), inner(theta_effective,␣
↪n))*ds(1)
```

The remainder of the demo is as usual:

```
f = Constant(1.0)
W_ext = f*w_*dx

L = psi_M*dx - W_ext + L_CDG + L_BC

F = derivative(L, w_, w_t)
J = derivative(F, w_, w)

A, b = assemble_system(J, -F, bcs=bcs_w)
solver = PETScLUSolver("mumps")
solver.solve(A, w_.vector(), b)
XDMFFile("output/w.xdmf").write(w_)
```

Unit testing

```
def test_close():
    import numpy as np
    assert(np.isclose(w_((0.5, 0.5)), 1.265E-6, atol=1E-3, rtol=1E-3))
```

Von-Kármán plate problem solved with MITC and PRSI approaches to cure locking:

3.2.4 Buckling of a heated von-Kármán plate

This demo is implemented in the single Python file `demo_von-karman-mansfield.py`.

This demo program solves the von-Kármán equations on a circular plate with a lenticular cross section free on the boundary. The plate is heated, causing it to bifurcate. Bifurcation occurs with a striking shape transition: before the critical threshold the plate assumes a cup-shaped configurations (left); above it tends to a cylindrical shape (right). An analytical solution has been found by Mansfield, see [1].

The standard von-Kármán theory gives rise to a fourth-order PDE which requires the transverse displacement field w to be sought in the space $H^2(\Omega)$. We relax this requirement in the same manner as the Kirchoff-Love plate theory can be relaxed to the Reissner-Mindlin theory, resulting in seeking a transverse displacement field w in $H^1(\Omega)$ and a rotation field θ in $[H^2(\Omega)]^2$. To alleviate the resulting shear- and membrane-locking issues we use Partial Selective Reduced Integration (PSRI), see [2].

To follow this demo you should know how to:

- Define a `MixedElement` and a `FunctionSpace` from it.
- Write variational forms using the Unified Form Language.

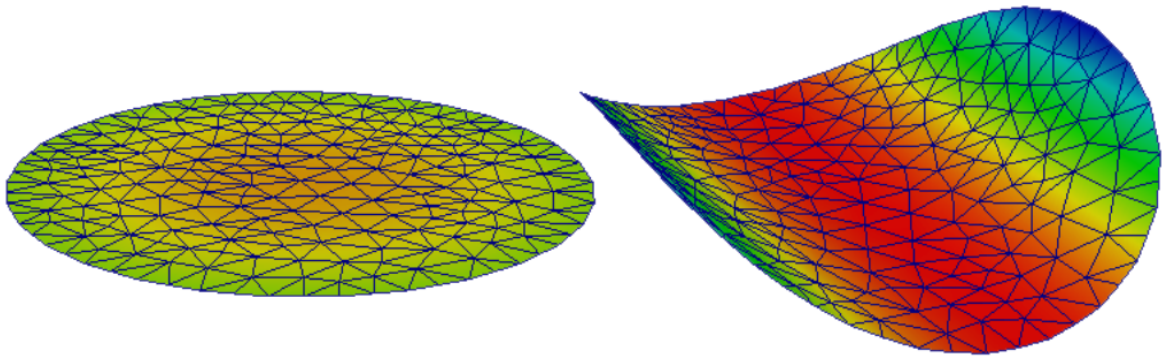


Fig. 3.7: Pre-critical and post-critical plate configuration.

- Automatically derive Jabobian and residuals using `derivative()`.
- Apply Dirichlet boundary conditions using `DirichletBC` and `apply()`.
- Assemble forms using `assemble()`.

This demo then illustrates how to:

- Define the Reissner-Mindlin-von-Kármán plate equations using UFL.
- Use the PSRI approach to simultaneously cure shear- and membrane-locking issues.

We start with importing the required modules, setting `matplolib` as plotting backend, and generically set the integration order to 4 to avoid the automatic setting of FEniCS which would lead to unreasonably high integration orders for complex forms.

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

parameters["form_compiler"]["quadrature_degree"] = 4
```

The mid-plane of the plate is a circular domain with radius $a = 1$. We generate a mesh of the domain using the package `mshr`:

```
a = 1.
ndiv = 8
domain_area = np.pi*a**2

centre = Point(0.,0.)

geom = mshr.Circle(centre, a)
mesh = mshr.generate_mesh(geom, ndiv)
```

The lenticular thinning of the plate can be modelled directly through the thickness parameter in the plate model:

```
t = 1E-2
ts = interpolate(Expression('t*(1.0 - (x[0]*x[0] + x[1]*x[1])/(a*a))', t=t, a=a,␣
↪degree=2), FunctionSpace(mesh, 'CG', 2))
```

We assume the plate is isotropic with constant material parameters, Young's modulus E, Poisson's ratio ν; shear correction factor κ:

```
E = Constant(1.0)
nu = Constant(0.3)
kappa = Constant(5.0/6.0)
```

Then, we compute the (scaled) membrane, A/t^3, bending, D/t^3, and shear, S/t^3, plate elastic stiffnesses:

```
A = (E*ts/t**3/(1. - nu**2))*as_tensor([[1., nu, 0.],[nu, 1., 0.],[0., 0., (1. - nu)/
↪2]])
D = (E*ts**3/t**3/(12.*(1. - nu**2)))*as_tensor([[1., nu, 0.],[nu, 1., 0.],[0., 0.,␣
↪(1. - nu)/2]])
S = E*kappa*ts/t**3/(2*(1. + nu))
```

We use a CG_2 element for the in-plane and transverse displacements u and w, and the enriched element $[CG_1 + B_3]$ for the rotations θ. We collect the variables in the state vector $q = (u, w, \theta)$:

```
P1 = FiniteElement("Lagrange", triangle, degree = 1)
P2 = FiniteElement("Lagrange", triangle, degree = 2)
bubble = FiniteElement("B", triangle, degree = 3)
enriched = P1 + bubble

element = MixedElement([VectorElement(P2, dim=2), P2, VectorElement(enriched, dim=2)])

Q = FunctionSpace(mesh, element)
```

Then, we define `Function`, `TrialFunction` and `TestFunction` objects to express the variational forms and we split them into each individual component function:

```
q_, q, q_t = Function(Q), TrialFunction(Q), TestFunction(Q)
v_, w_, theta_ = split(q_)
```

The membrane strain tensor e for the von-Kármán plate takes into account the nonlinear contribution of the transverse displacement in the approximate form:

$$e(v, w) = \mathrm{sym}\nabla v + \frac{\nabla w \otimes \nabla w}{2}$$

which can be expressed in UFL as:

```
e = sym(grad(v_)) + 0.5*outer(grad(w_), grad(w_))
```

The membrane energy density ψ_m is a quadratic function of the membrane strain tensor e. For convenience, we use our function `strain_to_voigt()` to express e in Voigt notation $e_V = \{e_1, e_2, 2e_{12}\}$:

```
ev = strain_to_voigt(e)
psi_m = 0.5*dot(A*ev, ev)
```

The bending strain tensor k and shear strain vector γ are identical to the standard Reissner-Mindlin model. The shear energy density ψ_s is a quadratic function of the shear strain vector:

```
gamma = grad(w_) - theta_
psi_s = 0.5*dot(S*gamma, gamma)
```

The bending energy density ψ_b is a quadratic function of the bending strain tensor. Here, the temperature profile on the plate is not modelled directly. Instead, it gives rise to an inelastic (initial) bending strain tensor k_T which can be incoporated directly in the Lagrangian:

```
k_T = as_tensor(Expression((("c/imp","0.0"),("0.0","c*imp")), c=1., imp=.999, ␣
↪degree=0))
k = sym(grad(theta_)) - k_T
kv = strain_to_voigt(k)
psi_b = 0.5*dot(D*kv, kv)
```

Note: The standard von-Kármán model can be recovered by substituting in the Kirchoff constraint $\theta = \nabla w$.

Shear- and membrane-locking are treated using the partial reduced selective integration proposed by Arnold and Brezzi, see [2]. In this approach shear and membrane energy are splitted as a sum of two contributions weighted by a factor α. One of the two contributions is integrated with a reduced integration. We use 2×2-points Gauss integration for a portion $1 - \alpha$ of the energy, whilst the rest is integrated with a 4×4 scheme. We adopt an optimized weighting factor $\alpha = (t/h)^2$, where h is the mesh size.:

```
dx_h = dx(metadata={'quadrature_degree': 2})
h = CellDiameter(mesh)
alpha = project(t**2/h**2, FunctionSpace(mesh,'DG',0))

Pi_PSRI = psi_b*dx + alpha*psi_m*dx + alpha*psi_s*dx + (1.0 - alpha)*psi_s*dx_h + (1.
↪0 - alpha)*psi_m*dx_h
```

Then, we compute the total elastic energy and its first and second derivatives:

```
Pi = Pi_PSRI
dPi = derivative(Pi, q_, q_t)
J = derivative(dPi, q_, q)
```

This problem is a pure Neumann problem. This leads to a nullspace in the solution. To remove this nullspace, we fix all the variables in the central point of the plate and the displacements in the x_0 and x_1 direction at $(0, a)$ and $(a, 0)$, respectively:

```
zero_v1 = project(Constant((0.)), Q.sub(0).sub(0).collapse())
zero_v2 = project(Constant((0.)), Q.sub(0).sub(1).collapse())
zero = project(Constant((0.,0.,0.,0.,0.)), Q)

bc = DirichletBC(Q, zero, "near(x[0], 0.) and near(x[1], 0.)", method="pointwise")
bc_v1 = DirichletBC(Q.sub(0).sub(0), zero_v1, "near(x[0], 0.) and near(x[1], 1.)",␣
↪method="pointwise")
bc_v2 = DirichletBC(Q.sub(0).sub(1), zero_v2, "near(x[0], 1.) and near(x[1], 0.)",␣
↪method="pointwise")
bcs = [bc, bc_v1, bc_v2]
```

Then, we define the nonlinear variational problem and the solver settings:

```
init = Function(Q)
q_.assign(init)
problem = NonlinearVariationalProblem(dPi, q_, bcs, J = J)
solver = NonlinearVariationalSolver(problem)
solver.parameters["newton_solver"]["absolute_tolerance"] = 1E-8
```

Finally, we choose the continuation steps (the critical loading `c_cr` is taken from the Mansfield analytical solution [1]):

```
c_cr = 0.0516
cs = np.linspace(0.0, 1.5*c_cr, 30)
```

and we solve as usual:

```
defplots_dir = "output/3dplots-psri/"
file = File(defplots_dir + "sol.pvd")

ls_kx = []
ls_ky = []
ls_kxy = []
ls_kT = []

for count, i in enumerate(cs):
    k_T.c = i
    solver.solve()
    v_h, w_h, theta_h = q_.split(deepcopy=True)
```

To visualise the solution we assemble the bending strain tensor:

```
K_h = project(sym(grad(theta_h)), TensorFunctionSpace(mesh, 'DG', 0))
```

we compute the average bending strain:

```
Kxx = assemble(K_h[0,0]*dx)/domain_area
Kyy = assemble(K_h[1,1]*dx)/domain_area
Kxy = assemble(K_h[0,1]*dx)/domain_area

ls_kT.append(i)
ls_kx.append(Kxx)
ls_ky.append(Kyy)
ls_kxy.append(Kxy)
```

and output the results at each continuation step:

```
v1_h, v2_h = v_h.split()
u_h = as_vector([v1_h, v2_h, w_h])
u_h_pro = project(u_h, VectorFunctionSpace(mesh, 'CG', 1, dim=3))
u_h_pro.rename("q_","q_")
file << u_h_pro
```

Finally, we plot the average curvatures as a function of the inelastic curvature:

```
fig = plt.figure(figsize=(5.0, 5.0/1.648))
plt.plot(ls_kT, ls_kx, "o", color='orange', label=r"$k_{1h}$")
plt.plot(ls_kT, ls_ky, "x", color='red', label=r"$k_{2h}$")
plt.xlabel(r"inelastic curvature $\eta$")
plt.ylabel(r"curvature $k_{1,2}$")
plt.legend()
plt.tight_layout()
plt.savefig("psri-%s.png"%ndiv)
```

Unit testing

```
import pytest
@pytest.mark.skip
def test_close():
    pass
```

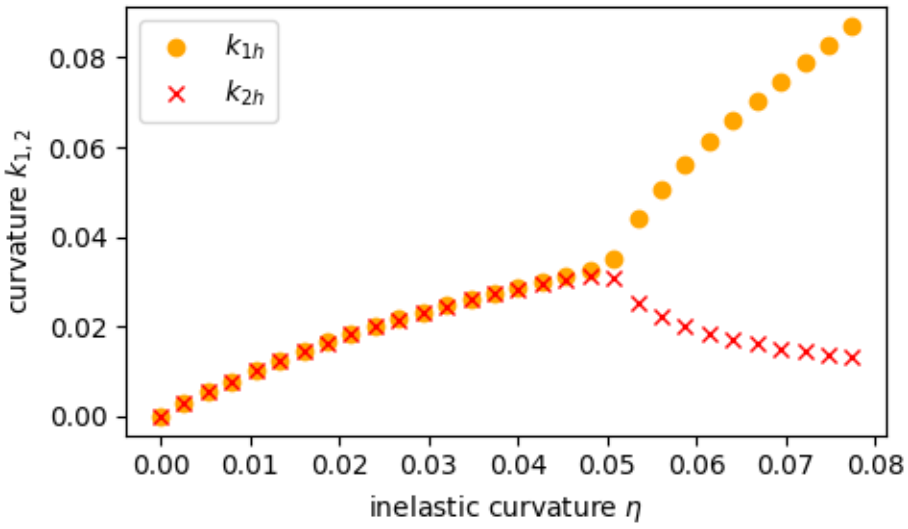


Fig. 3.8: Comparison with the analytical solution.

References

[1] E. H. Mansfield, "Bending, Buckling and Curling of a Heated Thin Plate. Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences. Vol. 268. No. 1334. The Royal Society, 1962.

[2] D. Arnold and F.Brezzi, Mathematics of Computation, 66(217): 1-14, 1997. https://www.ima.umn.edu/~arnold//papers/shellelt.pdf

3.2.5 Bifurcation of a composite laminate plate modeled with von-Kármán theory

This demo is implemented in the single Python file `demo_von-karman-composite.py`.

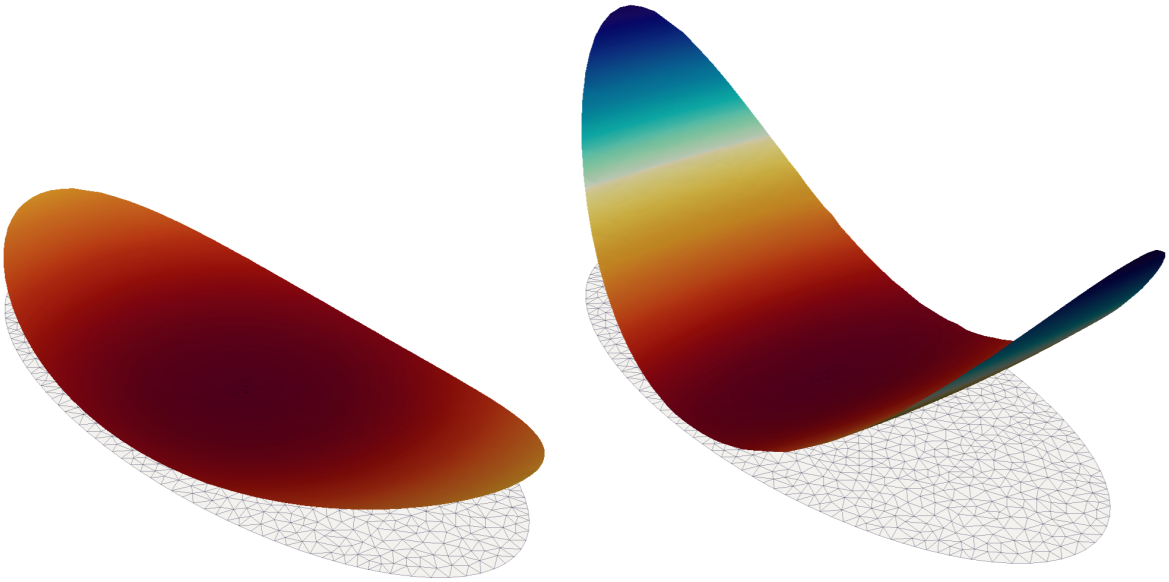
This demo program solves the von-Kármán equations for an elliptic composite plate with a lenticular cross section free on the boundary. The plate is heated, causing it to bifurcate. Bifurcation occurs with a striking shape transition: before the critical threshold the plate assumes a cup-shaped configurations (left); above it tends to a cylindrical shape (right).

An analytical solution can be computed using the procedure outlined in the paper *[Maurini]*.

The standard von-Kármán theory gives rise to a fourth-order PDE which requires the transverse displacement field w to be sought in the space $H^2(\Omega)$. We relax this requirement in the same manner as the Kirchoff-Love plate theory can be relaxed to the Reissner-Mindlin theory. Accordingly, we seek a transverse displacement field w in $H^1(\Omega)$ and a rotation field θ in $[H^2(\Omega)]^2$. To alleviate the resulting shear-locking issue we then apply the Durán-Liberman reduction operator.

To follow this demo you should know how to:

- Define a `MixedElement` and a `FunctionSpace` from it.
- Define the Durán-Liberman (MITC) reduction operator using UFL. This procedure eliminates the shear-locking problem.
- Write variational forms using the Unified Form Language.
- Use the fenics-shell function `laminate()` to compute the stiffness matrices.



- Automatically derive Jacobian and residuals using `derivative()`.
- Apply Dirichlet boundary conditions using `DirichletBC` and `apply()`.
- Assemble forms using `assemble()`.
- Solve linear systems using `LUSolver`.
- Output data to XDMF files with `XDMFFile`.

This demo then illustrates how to:

- Define the Reissner-Mindlin-von-Kármán plate equations using UFL.
- Use the `ProjectedNonlinearSolver` class to drive the solution of a non-linear problem with projected variables.

We begin by setting up our Python environment with the required modules::

```
from dolfin import *
from ufl import RestrictedElement

from fenics_shells import *
import numpy as np

try:
   import matplotlib.pyplot as plt
except ImportError:
   raise ImportError("matplotlib is required to run this demo.")

try:
   import mshr
except ImportError:
   raise ImportError("mshr is required to run this demo.")
```

The mid-plane of the plate is an elliptic domain with semi-axes $a = 1$ and $b = 0.5$. We generate a mesh of the domain using the package `mshr`:

```
a_rad = 1.0
b_rad = 0.5
n_div = 30

centre = Point(0.,0.)

geom = mshr.Ellipse(centre, a_rad, b_rad)
mesh = mshr.generate_mesh(geom, n_div)
```

The lenticular thinning of the plate can be modelled directly through the thickness parameter in the plate model:

```
h = interpolate(Expression('t0*(1.0 - (x[0]*x[0])/(a*a) - (x[1]*x[1])/(b*b))', t0=1E-
↪2, a=a_rad, b=b_rad, degree=2), FunctionSpace(mesh, 'CG', 2))
```

We assume the plate is a composite laminate with 8-layer stacking sequence:

$$[45°, -45°, -45°, 45°, -45°, 45°, 45°, -45°]$$

with elementary layer properties $E_1 = 40.038, E_2 = 1, G_{12} = 0.5, \nu_{12} = 0.25, G_{23} = 0.4$:

```
thetas = [np.pi/4., -np.pi/4., -np.pi/4., np.pi/4., -np.pi/4., np.pi/4., np.pi/4., -
↪np.pi/4.]
E1 = 40.038
E2 = 1.0
G12 = 0.5
nu12 = 0.25
G23 = 0.4
```

We use our function `laminates()` to compute the stiffness matrices according to the Classical Laminate Theory:

```
n_layers= len(thetas)
hs = h*np.ones(n_layers)/n_layers
A, B, D = laminates.ABD(E1, E2, G12, nu12, hs, thetas)
Fs = laminates.F(G12, G23, hs, thetas)
```

We then define our `MixedElement` which will discretise the in-plane displacements $v \in [\mathrm{CG}_1]^2$, rotations $\theta \in [\mathrm{CG}_2]^2$, out-of-plane displacements $w \in \mathrm{CG}_1$. Two further auxilliary fields are also considered, the reduced shear strain γ_R, and a Lagrange multiplier field p which ties together the shear strain calculated from the primal variables $\gamma = \nabla w - \theta$ and the reduced shear strain γ_R. Both p and γ_R are discretised in the space NED_1, the vector-valued Nédélec elements of the first kind. The final element definition is then:

```
element = MixedElement([VectorElement("Lagrange", triangle, 1),
                        VectorElement("Lagrange", triangle, 2),
                        FiniteElement("Lagrange", triangle, 1),
                        FiniteElement("N1curl", triangle, 1),
                        RestrictedElement(FiniteElement("N1curl", triangle, 1), "edge
↪")])
```

We then pass our `element` through to the `ProjectedFunctionSpace` constructor. As we will see later in this example, we can project out both the p and γ_R fields at assembly time. We specify this by passing the argument `num_projected_subspaces=2`:

```
U = ProjectedFunctionSpace(mesh, element, num_projected_subspaces=2)
U_F = U.full_space
U_P = U.projected_space
```

Using only the *full* function space object `U_F` we setup our variational problem by defining the Lagrangian of our problem. We begin by creating a `Function` and splitting it into each individual component function:

```
u, u_t, u_ = TrialFunction(U_F), TestFunction(U_F), Function(U_F)
v_, theta_, w_, R_gamma_, p_ = split(u_)
```

The membrane strain tensor e for the von-Kármán plate takes into account the nonlinear contribution of the transverse displacement in the approximate form:

$$e(v, w) = \mathrm{sym}\nabla v + \frac{\nabla w \otimes \nabla w}{2}$$

which can be expressed in UFL as:

```
e = sym(grad(v_)) + 0.5*outer(grad(w_), grad(w_))
```

The membrane energy density ψ_N is a quadratic function of the membrane strain tensor e. For convenience, we use our function `strain_to_voigt()` to express e in Voigt notation $e_V = \{e_1, e_2, 2e_{12}\}$:

```
ev = strain_to_voigt(e)
Ai = project(A, TensorFunctionSpace(mesh, 'CG', 1, shape=(3,3)))
psi_N = .5*dot(Ai*ev, ev)
```

The bending strain tensor k and shear strain vector γ are identical to the standard Reissner-Mindlin model. The shear energy density ψ_T is a quadratic function of the reduced shear vector:

```
Fi = project(Fs, TensorFunctionSpace(mesh, 'CG', 1, shape=(2,2)))
psi_T = .5*dot(Fi*R_gamma_, R_gamma_)
```

The bending energy density ψ_M is a quadratic function of the bending strain tensor. Here, the temperature profile on the plate is not modelled directly. Instead, it gives rise to an inelastic (initial) bending strain tensor k_T which can be incoporated directly in the Lagrangian:

```
k_T = as_tensor(Expression((("1.0*c","0.0*c"),("0.0*c","1.0*c")), c=1.0, degree=0))
k = sym(grad(theta_)) - k_T
kv = strain_to_voigt(k)
Di = project(D, TensorFunctionSpace(mesh, 'CG', 1, shape=(3,3)))
psi_M = .5*dot(Di*kv, kv)
```

Note: The standard von-Kármán model can be recovered by substituting in the Kirchoff constraint $\theta = \nabla w$.

Finally, we define the membrane-bending coupling energy density ψ_{NM}, even if it vanishes in this case:

```
Bi = project(B, TensorFunctionSpace(mesh, 'CG', 1, shape=(3,3)))
psi_MN = dot(Bi*kv, ev)
```

This problem is a pure Neumann problem. This leads to a nullspace in the solution. To remove this nullspace, we fix the displacements in the central point of the plate:

```
h_max = mesh.hmax()
def center(x,on_boundary):
    return x[0]**2 + x[1]**2 < (0.5*h_max)**2

bc_v = DirichletBC(U.sub(0), Constant((0.0,0.0)), center, method="pointwise")
bc_R = DirichletBC(U.sub(1), Constant((0.0,0.0)), center, method="pointwise")
bc_w = DirichletBC(U.sub(2), Constant(0.0), center, method="pointwise")
bcs = [bc_v, bc_R, bc_w]
```

Finally, we define the Durán-Liberman reduction operator by tying the shear strain calculated with the displacement variables $\gamma = \nabla w - \theta$ to the conforming reduced shear strain γ_R using the Lagrange multiplier field p:

```
gamma = grad(w_) - theta_
L_R = inner_e(gamma - R_gamma_, p_)
```

We can now define our Lagrangian for the complete system:

```
L = (psi_M + psi_T + psi_N + psi_MN)*dx + L_R
F = derivative(L, u_, u_t)
J = derivative(F, u_, u)
```

The solution of a non-linear problem with the *ProjectedFunctionSpace* functionality is a little bit more involved than the linear case. We provide a special class `ProjectedNonlinearProblem` which conforms to the DOLFIN `NonlinearProblem` interface that hides much of the complexity.

Note: Inside `ProjectedNonlinearProblem`, the Jacobian and residual equations on the projected space are assembled using the special assembler in FEniCS Shells. The Newton update is calculated on the space `U_P`. Then, it is necessary to update the variables in the full space `U_F` before performing the next Newton iteration.

In practice, the interface is nearly identical to a standard implementation of `NonlinearProblem`, except the requirement to pass a `Function` on both the full `u_` and projected spaces `u_p_`:

```
u_p_ = Function(U_P)
problem = ProjectedNonlinearProblem(U_P, F, u_, u_p_, bcs=bcs, J=J)
solver = NewtonSolver()
solver.parameters['absolute_tolerance'] = 1E-12
```

We apply the inelastic curvature with 20 continuation steps. The critical loading `c_cr` as well as the solution in terms of curvatures is taken from the analytical solution. Here, R_0 is the scaling radius of curvature and $\beta = A_{2222}/A_{1111} = D_{2222}/D_{1111}$ as in *[Maurini]*:

```
from fenics_shells.analytical.vonkarman_heated import analytical_solution
c_cr, beta, R0, h_before, h_after, ls_Kbefore, ls_K1after, ls_K2after = analytical_
↪solution(Ai, Di, a_rad, b_rad)
cs = np.linspace(0.0, 1.5*c_cr, 20)
```

We solve as usual:

```
domain_area = np.pi*a_rad*b_rad
kx = []
ky = []
kxy = []
ls_load = []

for i, c in enumerate(cs):
    k_T.c = c
    solver.solve(problem, u_p_.vector())
    v_h, theta_h, w_h, R_theta_h, p_h = u_.split()
```

Then, we assemble the bending strain tensor:

```
k_h = sym(grad(theta_h))
K_h = project(k_h, TensorFunctionSpace(mesh, 'DG', 0))
```

we calculate the average bending strain:

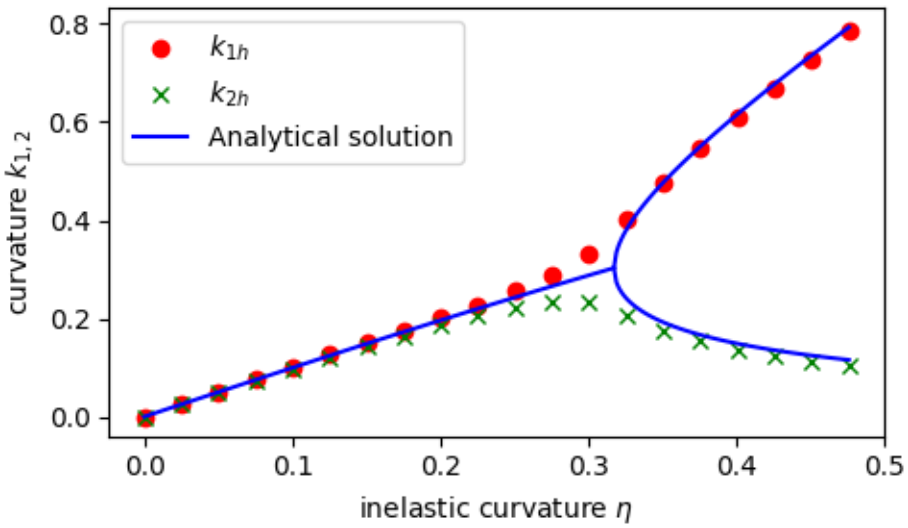
```
Kxx = assemble(K_h[0,0]*dx)/domain_area
Kyy = assemble(K_h[1,1]*dx)/domain_area
Kxy = assemble(K_h[0,1]*dx)/domain_area

ls_load.append(c*R0)
kx.append(Kxx*R0/np.sqrt(beta))
ky.append(Kyy*R0)
kxy.append(Kxy*R0/(beta**(1.0/4.0)))
```

and output the results at each continuation step:

```
save_dir = "output/"
fields = {"theta": theta_h, "v": v_h, "w": w_h, "R_theta": R_theta_h}
for name, field in fields.items():
    field.rename(name, name)
    field_file = XDMFFile("{}/{}_{}.xdmf".format(save_dir, name, str(i).zfill(3)))
    field_file.write(field)
```

Finally, we compare numerical and analytical solutions:

```
fig = plt.figure(figsize=(5.0, 5.0/1.648))
plt.plot(ls_load, kx, "o", color='r', label=r"$k_{1h}$")
plt.plot(ls_load, ky, "x", color='green', label=r"$k_{2h}$")
plt.plot(h_before, ls_Kbefore, "-", color='b', label="Analytical solution")
plt.plot(h_after, ls_K1after, "-", color='b')
plt.plot(h_after, ls_K2after, "-", color = 'b')
plt.xlabel(r"inelastic curvature $\eta$")
plt.ylabel(r"curvature $k_{1,2}$")
plt.legend()
plt.tight_layout()
plt.savefig("curvature_bifurcation.png")
```



Unit testing

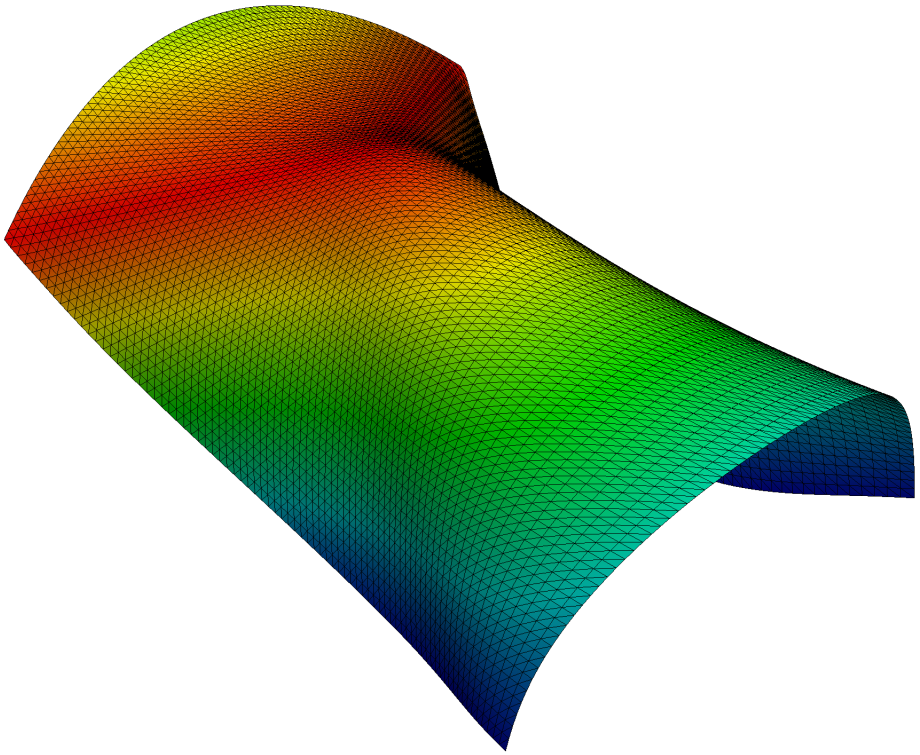
```
import pytest
@pytest.mark.skip
def test_close():
    pass
```

Linear and non-linear Naghdi shell problem solved with MITC and PRSI locking cure

3.2.6 Partly Clamped Hyperbolic Paraboloid

This demo is implemented in the single Python file `demo_naghdi-linear-hypar.py`.

This demo program solves the linear Naghdi shell equations for a partly-clamped hyperbolic paraboloid (hypar) subjected to a uniform vertical loading. This is a well known bending dominated benchmark problem for testing a FE formulation with respect to membrane locking issues, see [1]. Here, locking is cured using enriched finite elements including cubic bubble shape functions and Partial Selective Reduced Integration (PSRI), see [2, 3].



To follow this demo you should know how to:

- Define a MixedElement and EnrichedElement and a FunctionSpace from it.
- Write variational forms using the Unified Form Language.
- Automatically derive Jacobian and residuals using derivative().
- Apply Dirichlet boundary conditions using DirichletBC and apply().

This demo then illustrates how to:

- Define and solve a linear Naghdi shell problem with a curved stress-free configuration given as analytical expression in terms of two curvilinear coordinates.

We start with importing the required modules, setting `matplotlib` as plotting backend, and generically set the integration order to 4 to avoid the automatic setting of FEniCS which would lead to unreasonably high integration orders for complex forms.:

```
import os, sys

import numpy as np
import matplotlib.pyplot as plt

from dolfin import *
from mshr import *
from ufl import Index

from mpl_toolkits.mplot3d import Axes3D

parameters["form_compiler"]["quadrature_degree"] = 4
output_dir = "output/"
if not os.path.exists(output_dir):
    os.makedirs(output_dir)
```

We consider an hyperbolic paraboloid shell made of a linear elastic isotropic homogeneous material with Young modulus `Y` and Poisson ratio `nu`; `mu` and `lb` denote the shear modulus $\mu = Y/(2(1+\nu))$ and the Lamé constant $\lambda = 2\mu\nu/(1-2\nu)$. The (uniform) shell thickness is denoted by `t`.

```
Y, nu = 2.0e8, 0.3
mu = Y/(2.0*(1.0 + nu))
lb = 2.0*mu*nu/(1.0 - 2.0*nu)
t = Constant(1E-2)
```

The midplane of the initial (stress-free) configuration $\mathcal{S}_I$ of the shell is given in the form of an analytical expression

$$y_I : x \in \mathcal{M} \subset R^2 \rightarrow y_I(x) \in \mathcal{S}_I \subset \mathcal{R}^3 \qquad \mathcal{S}_I = \{x_1, x_2, x_1^2 - x_2^2\}$$

in terms of the curvilinear coordinates x. Hence, we mesh the two-dimensional domain $\mathcal{M} \equiv [-L/2, L/2] \times [-L/2, L/2]$.:

```
L = 1.0
P1, P2 = Point(-L/2, -L/2), Point(L/2, L/2)
ndiv = 40
mesh = RectangleMesh(P1, P2, ndiv, ndiv)
```

We provide the analytical expression of the initial shape as an `Expression` that we represent on a suitable `FunctionSpace` (here P_2, but other are choices are possible):

```
initial_shape = Expression(('x[0]','x[1]','x[0]*x[0] - x[1]*x[1]'), degree = 4)
V_y = FunctionSpace(mesh, VectorElement("P", triangle, degree=2, dim=3))
yI = project(initial_shape, V_y)
```

We compute the covariant and contravariant component of the metric tensor, `aI`, `aI_contra`; the covariant base vectors `g0`, `g1`; the contravariant base vectors `g0_c`, `g1_c`.

```
aI = grad(yI).T*grad(yI)
aI_contra, jI = inv(aI), det(aI)
g0, g1 = yI.dx(0), yI.dx(1)
g0_c, g1_c = aI_contra[0,0]*g0 + aI_contra[0,1]*g1, aI_contra[1,0]*g0 + aI_contra[1,
↪1]*g1
```

Given the midplane, we define the corresponding unit normal as below and project on a suitable function space (here P_1 but other choices are possible):

```
def normal(y):
    n = cross(y.dx(0), y.dx(1))
    return n/sqrt(inner(n,n))

V_normal = FunctionSpace(mesh, VectorElement("P", triangle, degree = 1, dim = 3))
nI = project(normal(yI), V_normal)
```

We can visualize the shell shape and its normal with this utility function

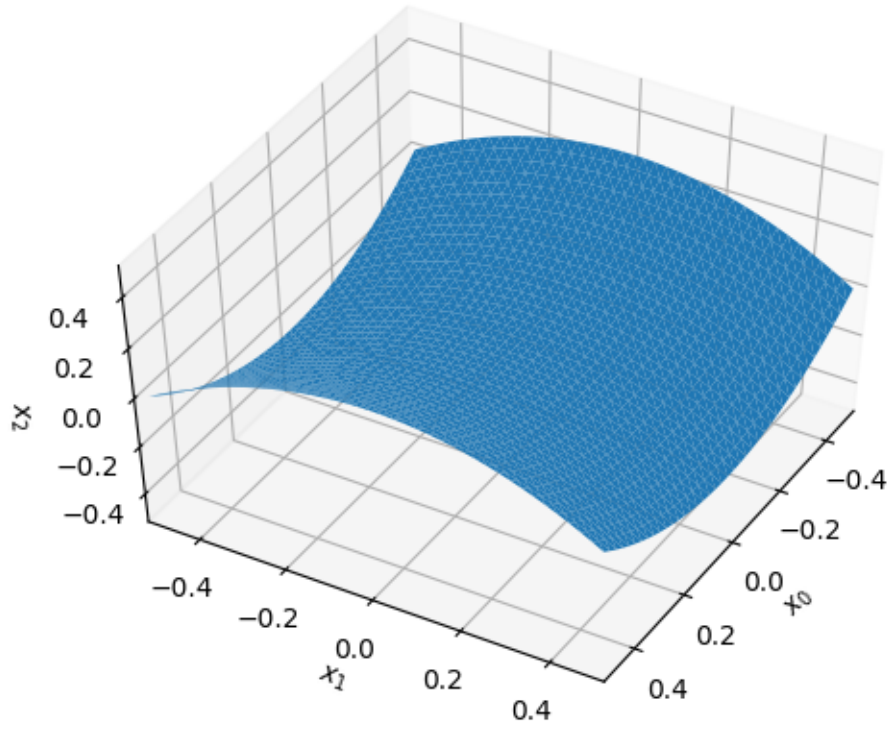
```
def plot_shell(y,n=None):
    y_0, y_1, y_2 = y.split(deepcopy=True)
    fig = plt.figure()
    ax = fig.gca(projection='3d')
    ax.plot_trisurf(y_0.compute_vertex_values(),
                    y_1.compute_vertex_values(),
                    y_2.compute_vertex_values(),
                    triangles=y.function_space().mesh().cells(),
                    linewidth=1, antialiased=True, shade = False)
    if n:
        n_0, n_1, n_2 = n.split(deepcopy=True)
        ax.quiver(y_0.compute_vertex_values(),
              y_1.compute_vertex_values(),
              y_2.compute_vertex_values(),
              n_0.compute_vertex_values(),
              n_1.compute_vertex_values(),
              n_2.compute_vertex_values(),
              length = .2, color = "r")
    ax.view_init(elev=50, azim=30)
    ax.set_xlim(-0.5, 0.5)
    ax.set_ylim(-0.5, 0.5)
    ax.set_zlim(-0.5, 0.5)
    ax.set_xlabel(r"$x_0$")
    ax.set_ylabel(r"$x_1$")
    ax.set_zlabel(r"$x_2$")
    return ax

plot_shell(yI)
plt.savefig("output/initial_configuration.png")
```

In our 5-parameter Naghdi shell model the configuration of the shell is assigned by

- the 3-component vector field `u_` representing the (small) displacement with respect to the initial configuration `yI`
- the 2-component vector field `theta_` representing the (small) rotation of fibers orthogonal to the middle surface.

Following [2, 3], we use a `P2+bubble` element for `y_` and a `P2` element for `theta_`, and collect them in the state vector `z_=[u_, theta_]`. We further define `Function`, `TestFunction`, and `TrialFucntion` and their different split views, which are useful for expressing the variational formulation.

```
P2 = FiniteElement("P", triangle, degree = 2)
bubble = FiniteElement("B", triangle, degree = 3)

Z = FunctionSpace(mesh, MixedElement(3*[P2 + bubble ] + 2*[P2]))
z_ = Function(Z)
```


0.4
0.2
0.0
−0.2
−0.4
x2
−0.4
−0.2
0.0
0.2
0.4
x1
−0.4
−0.2
0.0
0.2
0.4
x0

```
z, zt = TrialFunction(Z), TestFunction(Z)

u0_, u1_, u2_, th0_, th1_ = split(z_)
u0t, u1t, u2t, th0t, th1t = split(zt)
u0, u1, u2, th0, th1 = split(z)
```

We define the displacement vector and the rotation vector, with this latter tangent to the middle surface, $\theta = \theta_\sigma \, g^\sigma$, $\sigma = 0, 1$

```
u_, u, ut = as_vector([u0_, u1_, u2_]), as_vector([u0, u1, u2]), as_vector([u0t, u1t,␣
↪u2t])
theta_, theta, thetat = th0_*g0_c + th1_*g1_c, th0*g0_c + th1*g1_c, th0t*g0_c +␣
↪th1t*g1_c
```

The extensional, `e_naghdi`, bending, `k_naghdi`, and shearing, `g_naghdi`, strains in the linear Naghdi model are defined by

```
e_naghdi = lambda v: 0.5*(grad(yI).T*grad(v) + grad(v).T*grad(yI))
k_naghdi = lambda v, t: -0.5*(grad(yI).T*grad(t) + grad(t).T*grad(yI)) - 0.
↪5*(grad(nI).T*grad(v) + grad(v).T*grad(nI))
g_naghdi = lambda v, t: grad(yI).T*t + grad(v).T*nI
```

Using curvilinear coordinates, the constitutive equations are written in terms of the matrix `A_hooke` below, representing the contravariant components of the constitutive tensor for isotropic elasticity in plane stress, see *e.g.* [4]. We use the index notation offered by `UFL` to express operations between tensors

```
i, j, k, l = Index(), Index(), Index(), Index()
A_hooke = as_tensor((((2.0*lb*mu)/(lb + 2.0*mu))*aI_contra[i,j]*aI_contra[k,l] + 1.
↪0*mu*(aI_contra[i,k]*aI_contra[j,l] + aI_contra[i,l]*aI_contra[j,k])),[i,j,k,l])
```

The membrane stress and bending moment tensors, `N` and `M`, and shear stress vector, `T`, are

```
N = as_tensor((A_hooke[i,j,k,l]*e_naghdi(u_)[k,l]),[i, j])
M = as_tensor(((1./12.0)*A_hooke[i,j,k,l]*k_naghdi(u_, theta_)[k,l]),[i, j])
T = as_tensor((mu*aI_contra[i,j]*g_naghdi(u_, theta_)[j]), [i])
```

Hence, the contributions to the elastic energy densities due to membrane, `psi_m`, bending, `psi_b`, and shear, `psi_s`, are

```
psi_m = .5*inner(N, e_naghdi(u_))
psi_b = .5*inner(M, k_naghdi(u_, theta_))
psi_s = .5*inner(T, g_naghdi(u_, theta_))
```

Shear and membrane locking are treated using the PSRI proposed by Arnold and Brezzi, see [2, 3]. In this approach, shear and membrane energies are splitted as a sum of two weighted contributions, one of which is computed with a reduced integration. Thus, shear and membrane energies have the form

$$(i = m, s) \qquad \alpha \int_{\mathcal{M}} \psi_i \sqrt{j_I} \mathrm{d}x + (\kappa - \alpha) \int_{\mathcal{M}} \psi_i \sqrt{j_I} \mathrm{d}x_h, \qquad \text{where } \kappa \propto t^{-2}$$

While [2, 3] suggest a 1-point reduced integration, we observed that this leads to spurious modes in the present case. We use then 2×2-points Gauss integration for the portion $\kappa - \alpha$ of the energy, whilst the rest is integrated with a 4×4 scheme. As suggested in [3], we adopt an optimized weighting factor $\alpha = 1$

```
dx_h = dx(metadata={'quadrature_degree': 2})
alpha = 1.0
kappa = 1.0/t**2
```

```
shear_energy = alpha*psi_s*sqrt(jI)*dx + (kappa - alpha)*psi_s*sqrt(jI)*dx_h
membrane_energy = alpha*psi_m*sqrt(jI)*dx + (kappa - alpha)*psi_m*sqrt(jI)*dx_h
bending_energy =  psi_b*sqrt(jI)*dx
```

Then, the elastic energy is

```
elastic_energy = (t**3)*(bending_energy + membrane_energy + shear_energy)
```

The shell is subjected to a constant vertical load. Thus, the external work is

```
body_force = 8.*t
f = Constant(body_force)
external_work = f*u2_*sqrt(jI)*dx
```

We now compute the total potential energy with its first and second derivatives

```
Pi_total = elastic_energy - external_work
residual = derivative(Pi_total, z_, zt)
hessian = derivative(residual, z_, z)
```

The boundary conditions prescribe a full clamping on the $x_0 = -L/2$ boundary, while the other sides are left free

```
left_boundary = lambda x, on_boundary: abs(x[0] + L/2) <= DOLFIN_EPS and on_boundary
clamp = DirichletBC(Z, project(Expression(("0.0", "0.0", "0.0", "0.0", "0.0"), degree␣
↪= 1), Z), left_boundary)
bcs = [clamp]
```

We now solve the linear system of equations

```
output_dir = "output/"
A, b = assemble_system(hessian, residual, bcs=bcs)
solver = PETScLUSolver("mumps")
solver.solve(A, z_.vector(), b)
u0_h, u1_h, u2_h, th0_h, th1_h = z_.split(deepcopy=True)
```

Finally, we can plot the final configuration of the shell

```
scale_factor = 1e4
plot_shell(project(scale_factor*u_ + yI, V_y))
plt.savefig("output/finalconfiguration.png")
```

References

[1] K. J. Bathe, A. Iosilevich, and D. Chapelle. An evaluation of the MITC shell elements. Computers & Structures. 2000;75(1):1-30.

[2] D. Arnold and F.Brezzi, Mathematics of Computation, 66(217): 1-14, 1997. https://www.ima.umn.edu/~arnold//papers/shellelt.pdf

[3] D. Arnold and F.Brezzi, The partial selective reduced integration method and applications to shell problems. Computers & Structures. 64.1-4 (1997): 879-880.

[4] P. G. Ciarlet. An introduction to differential geometry with applications to elasticity. Journal of Elasticity, 78-79(1-3):1–215, 2005.

3.2.7 A non-linear Naghdi roll-up cantilever

This demo is implemented in the single Python file `demo_nonlinear-naghdi-cantilever.py`.

This demo program solves the non-linear Naghdi shell equations on a rectangular plate with a constant bending moment applied. The plate rolls up completely on itself. The numerical locking issue is cured using a Durán-Liberman approach.

To follow this demo you should know how to:

- Define a `MixedElement` and a `FunctionSpace` from it.
- Define the Durán-Liberman (MITC) reduction operator using UFL for a linear problem, e.g. Reissner-Mindlin. This procedure extends simply to the non-linear problem we consider here.
- Write variational forms using the Unified Form Language.
- Automatically derive Jabobian and residuals using `derivative()`.
- Apply Dirichlet boundary conditions using `DirichletBC` and `apply()`.
- Apply Neumann boundary conditions by marking a `FacetFunction` and create a new `Measure` object.
- Solve non-linear problems using `ProjectedNonlinearProblem`.
- Output data to XDMF files with `XDMFFile`.

This demo then illustrates how to:

- Define and solve a non-linear Naghdi shell problem with a *flat* reference configuration.

We begin by setting up our Python environment with the required modules:

```
import os
import numpy as np
import matplotlib.pyplot as plt

from dolfin import *
from ufl import RestrictedElement
from fenics_shells import *
```

We set the default quadrature degree. UFL's built in quadrature degree detection often overestimates the required degree for these complicated forms:

```
parameters["form_compiler"]["quadrature_degree"] = 2
```

Our reference middle surface is a rectangle $\omega = [0, 12] \times [-0.5, 0.5]$:

```
length = 12.0
width = 1.0
P1, P2 = Point(0.0, -width/2.0), Point(length, width/2.0)
mesh = RectangleMesh(P1, P2, 48, 4, "crossed")
```

We then define our `MixedElement` which will discretise the in-plane displacements $v \in [\mathrm{CG}_1]^2$, rotations $\beta \in [\mathrm{CG}_2]^2$, out-of-plane displacements $w \in \mathrm{CG}_1$, the shear strains. Two further auxilliary fields are also considered, the reduced shear strain γ_R, and a Lagrange multiplier field p which ties together the Naghdi shear strain calculated from the primal variables and the reduced shear strain γ_R. Both p and γ_R are are discretised in the space NED_1, the vector-valued Nédélec elements of the first kind. The final element definition is then:

```
element = MixedElement([VectorElement("Lagrange", triangle, 1),
                        VectorElement("Lagrange", triangle, 2),
                        FiniteElement("Lagrange", triangle, 1),
```

```
                        FiniteElement("N1curl", triangle, 1),
                        RestrictedElement(FiniteElement("N1curl", triangle, 1), "edge
↪")])
```

We then pass our `element` through to the `ProjectedFunctionSpace` constructor. As in the other documented demos, we can project out p and γ_R fields at assembly time. We specify this by passing the argument `num_projected_subspaces=2`:

```
U = ProjectedFunctionSpace(mesh, element, num_projected_subspaces=2)
U_F = U.full_space
U_P = U.projected_space
```

We assume constant material parameters; Young's modulus E, Poisson's ratio ν, and thickness t:

```
E, nu = Constant(1.2E6), Constant(0.0)
mu = E/(2.0*(1.0 + nu))
lmbda = 2.0*mu*nu/(1.0 - 2.0*nu)
t = Constant(1E-1)
```

Using only the *full* function space object `U_F` we setup our variational problem by defining the Lagrangian of our problem. We begin by creating a `Function` and splitting it into each individual component function:

```
u, u_t, u_ = TrialFunction(U_F), TestFunction(U_F), Function(U_F)
v_, beta_, w_, Rgamma_, p_ = split(u_)
```

For the Naghdi problem it is convienient to recombine the in-plane displacements v and out-of-plane displacements w into a single vector field z:

```
z_ = as_vector([v_[0], v_[1], w_])
```

We can now define our non-linear Naghdi strain measures. Assuming the normal fibres of the shell are unstrechable, we can parameterise the director vector field $d : \omega \to \mathbb{R}^3$ using the two independent rotations β:

```
d = as_vector([sin(beta_[1])*cos(beta_[0]), -sin(beta_[0]), cos(beta_[1])*cos(beta_
↪[0])])
```

The deformation gradient F can be defined as:

```
F = grad(z_) + as_tensor([[1.0, 0.0],
                          [0.0, 1.0],
                          [Constant(0.0), Constant(0.0)]])
```

From which we can define the stretching (membrane) strain e:

```
e = 0.5*(F.T*F - Identity(2))
```

The curvature (bending) strain k:

```
k = 0.5*(F.T*grad(d) + grad(d).T*F)
```

and the shear strain γ:

```
gamma = F.T*d
```

We then define the constitutive law in terms of a general dual strain measure tensor X:

```
S = lambda X: 2.0*mu*X + ((2.0*mu*lmbda)/(2.0*mu + lmbda))*tr(X)*Identity(2)
```

From which we can define the membrane energy density:

```
psi_N = 0.5*t*inner(S(e), e)
```

the bending energy density:

```
psi_K = 0.5*(t**3/12.0)*inner(S(k), k)
```

and the shear energy density:

```
psi_T = 0.5*t*mu*inner(Rgamma_, Rgamma_)
```

and the total energy density from all three contributions:

```
psi = psi_N + psi_K + psi_T
```

We define the Durán-Liberman reduction operator by tying the shear strain calculated with the displacement variables $\gamma = F^T d$ to the reduced shear strain γ_R using the Lagrange multiplier field p:

```
L_R = inner_e(gamma - Rgamma_, p_)
```

We then turn to defining the boundary conditions and external loading. On the left edge of the domain we apply clamped boundary conditions which corresponds to constraining all generalised displacement fields to zero:

```
left = lambda x, on_boundary: x[0] <= DOLFIN_EPS and on_boundary
bc_v = DirichletBC(U.sub(0), Constant((0.0, 0.0)), left)
bc_a = DirichletBC(U.sub(1), Constant((0.0, 0.0)), left)
bc_w = DirichletBC(U.sub(2), Constant(0.0), left)
bcs = [bc_v, bc_a, bc_w]
```

On the right edge of the domain we apply a traction:

```
# Define subdomain for boundary condition on tractions
class Right(SubDomain):
    def inside(self, x, on_boundary):
        return abs(x[0] - 12.0) <= DOLFIN_EPS and on_boundary

right_tractions = Right()
exterior_facet_domains = MeshFunction("size_t", mesh, mesh.geometry().dim() - 1)
exterior_facet_domains.set_all(0)
right_tractions.mark(exterior_facet_domains, 1)
ds = Measure("ds")(subdomain_data=exterior_facet_domains)

M_right = Expression(('M'), M=0.0, degree=0)

W_ext = M_right*beta_[1]*ds(1)
```

We can now define our Lagrangian for the complete system:

```
L = psi*dx + L_R - W_ext
F = derivative(L, u_, u_t)
J = derivative(F, u_, u)
```

Before setting up the non-linear problem with the special *ProjectedFunctionSpace* functionality:

```
u_p_ = Function(U_P)
problem = ProjectedNonlinearProblem(U_P, F, u_, u_p_, bcs=bcs, J=J)
solver = NewtonSolver()
```

and solving:

```
solver.parameters['error_on_nonconvergence'] = False
solver.parameters['maximum_iterations'] = 20
solver.parameters['linear_solver'] = "mumps"
solver.parameters['absolute_tolerance'] = 1E-20
solver.parameters['relative_tolerance'] = 1E-6

output_dir = "output/"
if not os.path.exists(output_dir):
    os.makedirs(output_dir)
```

We apply the moment with 20 continuation steps:

```
M_max = 2.0*np.pi*E.values()[0]*t.values()[0]**3/(12.0*length)
Ms = np.linspace(0.0, M_max, 20)

w_hs = []
v_hs = []

for i, M in enumerate(Ms):
    M_right.M = M
    solver.solve(problem, u_p_.vector())

    v_h, theta_h, w_h, Rgamma_h, p_h = u_.split(deepcopy=True)
    z_h = project(z_, VectorFunctionSpace(mesh, "CG", 1, dim=3))
    z_h.rename('z', 'z')

    XDMFFile(output_dir + "z_{}.xdmf".format(str(i).zfill(3))).write(z_h)

    w_hs.append(w_h(length, 0.0))
    v_hs.append(v_h(length, 0.0)[0])
```

This problem has a simple closed-form analytical solution which we plot against for comparison:

```
w_hs = np.array(w_hs)
v_hs = np.array(v_hs)

Ms_analytical = np.linspace(1E-3, 1.0, 100)
vs = 12.0*(np.sin(2.0*np.pi*Ms_analytical)/(2.0*np.pi*Ms_analytical) - 1.0)
ws = -12.0*(1.0 - np.cos(2.0*np.pi*Ms_analytical))/(2.0*np.pi*Ms_analytical)

fig = plt.figure(figsize=(5.0, 5.0/1.648))
plt.plot(Ms_analytical, vs/length, "-", label="$v/L$")
plt.plot(Ms/M_max, v_hs/length, "x", label="$v_h/L$")
plt.plot(Ms_analytical, ws/length, "--", label="$w/L$")
plt.plot(Ms/M_max, w_hs/length, "o", label="$w_h/L$")
plt.xlabel("$M/M_{\mathrm{max}}$")
plt.ylabel("normalised displacement")
plt.legend()
plt.tight_layout()
plt.savefig("output/cantilever-displacement-plot.pdf")
plt.savefig("output/cantilever-displacement-plot.png")
```

Unit testing

```
def test_close():
    assert(np.isclose(w_h(length, 0.0)/length, 0.0, atol=1E-3, rtol=1E-3))
    assert(np.isclose(v_h(length, 0.0)[0]/length, -1.0, atol=1E-3, rtol=1E-3))
```

CHAPTER 4

FEniCS-Shells

A FEniCS Project-based library for simulating thin structures.

We have recently started development of an experimental version, FEniCSx-Shells, that is compatible with the new FEniCSx components of the FEniCS Project.

4.1 Description

FEniCS-Shells is an open-source library that provides finite element-based numerical methods for solving a wide range of thin structural models (beams, plates and shells) expressed in the Unified Form Language (UFL) of the FEniCS Project.

FEniCS-Shells is compatible with the 2019.1.0 release of the FEniCS Project.

FEniCS-Shells is described fully in the paper:

Simple and extensible plate and shell finite element models through automatic code generation tools, J. S. Hale, M. Brunetti, S. P. A. Bordas, C. Maurini. *Computers & Structures*, 209, 163-181, doi:10.1016/j.compstruc.2018.08.001.

4.2 Getting started

1. Install FEniCS by following the instructions at http://fenicsproject.org/download. We recommend using Docker to install FEniCS. However, you can use any method you want to install FEniCS.
2. Then, clone this repository using the command:

```
git clone https://bitbucket.org/unilucompmech/fenics-shells.git
```

3. If you do not have an appropiate version of FEniCS already installed, use a Docker container (skip the second line if you have already an appropiate version of FEniCS installed):

```
cd fenics-shells
./launch-container.sh
```

4. You should now have a shell inside a container with FEniCS installed. Try out an example:

```
python3 setup.py develop --user
cd demo
./generate_demos.py
cd documented/reissner_mindlin_clamped
python3 demo_reissner-mindlin-clamped.py
```

   The resulting fields are written to the directory `output/` which will be shared with the host machine. These files can be opened using Paraview.

5. Check out the demos at https://fenics-shells.readthedocs.io/.

4.3 Documentation

Documentation can be viewed at http://fenics-shells.readthedocs.org/.

4.4 Automated testing

We use Bitbucket Pipelines to perform automated testing. All documented demos include basic sanity checks on the results. Tests are run in the `quay.io/fenicsproject/stable:current` Docker image.

4.5 Features

FEniCS-Shells currently includes implementations of the following structural models:

- Kirchhoff-Love plates,
- Reissner-Mindlin plates,
- von-Kármán shallow shells,
- Reissner-Mindlin-von-Kármán shallow shells,
- non-linear and linear Naghdi shells with exact geometry.

Additionally, the following models are under active development:

- linear and non-linear Timoshenko beams,

We are using a variety of finite element numerical techniques including:

- MITC reduction operators,
- discontinuous Galerkin methods,
- reduced integration techniques.

4.6 Citing

Please consider citing the FEniCS-Shells paper and code if you find it useful.

```
@article{hale_simple_2018,
    title = {Simple and extensible plate and shell finite element models through␣
↪automatic code generation tools},
    volume = {209},
    issn = {0045-7949},
    url = {http://www.sciencedirect.com/science/article/pii/S0045794918306126},
    doi = {10.1016/j.compstruc.2018.08.001},
    journal = {Computers \& Structures},
    author = {Hale, Jack S. and Brunetti, Matteo and Bordas, Stéphane P. A. and␣
↪Maurini, Corrado},
    month = oct,
    year = {2018},
    keywords = {Domain specific language, FEniCS, Finite element methods, Plates,␣
↪Shells, Thin structures},
    pages = {163--181},
}

@misc{hale_fenics-shells_2016,
    title = {{FEniCS}-{Shells}},
    url = {https://figshare.com/articles/FEniCS-Shells/4291160},
    author = {Hale, Jack S. and Brunetti, Matteo and Bordas, Stéphane P.A. and␣
↪Maurini, Corrado},
    month = dec,
    year = {2016},
    doi = {10.6084/m9.figshare.4291160},
    keywords = {FEniCS, Locking, MITC, PDEs, Python, Shells, thin structures},
}
```

along with the appropriate general FEniCS citations.

4.7 Contributing

We are always looking for contributions and help with fenics-shells. If you have ideas, nice applications or code contributions then we would be happy to help you get them included. We ask you to follow the FEniCS Project git workflow.

4.8 Issues and Support

Please use the bugtracker to report any issues.

For support or questions please email jack.hale@uni.lu.

4.9 Authors (alphabetical)

Matteo Brunetti, Université Pierre et Marie Curie, Paris.
Jack S. Hale, University of Luxembourg, Luxembourg.
Corrado Maurini, Université Pierre et Marie Curie, Paris.

4.10 License

Bibliography

[Maurini] A. Fernandes, C. Maurini, S. Vidoli, “Multiparameter actuation for shape control of bistable composite plates.” International Journal of Solids and Structures. Vol. 47. Pages 1449-145., 2010.

Python Module Index

f

Index

P

R

S

T

Z