Nonlinear analysis of thin-walled structures based on tangential differential calculus with FEniCSx

Andreas Zilian, Michal Habera

Department of Engineering | University of Luxembourg

23 August 2022 | FEniCS'22 conference

UNIVERSITÉ DU LUXEMBOURG

Contents

Challenges (past)

- higher-order geometry of initial configuration (curvature/director)
- quadrilateral cells
- *easy* formulation in UFL

Motivation | thin-walled structures & FEniCS

Challenges (past)

- higher-order geometry of initial configuration (curvature/director)
- quadrilateral cells
- *easy* formulation in UFL

Existing approaches (FEniCS)

- PDEs on manifolds (Rognes et al. 2013, [1], no structures)
- `fenics-shells` (Hale et al. 2018, [2], requires explicit geometry map)
- `comet-fenics` (Bleyer 2018, [3], tangent triad by projection)

Motivation | thin-walled structures & FEniCS

Challenges (past)

- higher-order geometry of initial configuration (curvature/director)
- quadrilateral cells
- *easy* formulation in UFL

Existing approaches (FEniCS)

- PDEs on manifolds (Rognes et al. 2013, [1], no structures)
- `fenics-shells` (Hale et al. 2018, [2], requires explicit geometry map)
- `comet-fenics` (Bleyer 2018, [3], tangent triad by projection)

Straightforward formulation of thin-walled structures: demos in `dolfiny`

Geometry

- manifold Γ of codimension 1 (or 2) embedded in physical space $\mathbb{R}^3$
- manifold representation: *explicitly* (surface atlas) or *implicitly*
- normal vector $\mathbf{n}_\Gamma(\mathbf{x}) \in \mathbb{R}^3$; tangent space projector $\mathbf{P}(\mathbf{x}) = \mathbf{I} - \mathbf{n}_\Gamma(\mathbf{x}) \otimes \mathbf{n}_\Gamma(\mathbf{x})$

Tangential differential calculus (TDC) | brief introduction

Geometry

- manifold Γ of codimension 1 (or 2) embedded in physical space $\mathbb{R}^3$
- manifold representation: *explicitly* (surface atlas) or *implicitly*
- normal vector $\mathbf{n}_\Gamma(\mathbf{x}) \in \mathbb{R}^3$; tangent space projector $\mathbf{P}(\mathbf{x}) = \mathbf{I} - \mathbf{n}_\Gamma(\mathbf{x}) \otimes \mathbf{n}_\Gamma(\mathbf{x})$

Surface gradients

- surface gradient of *scalar* function in UFL

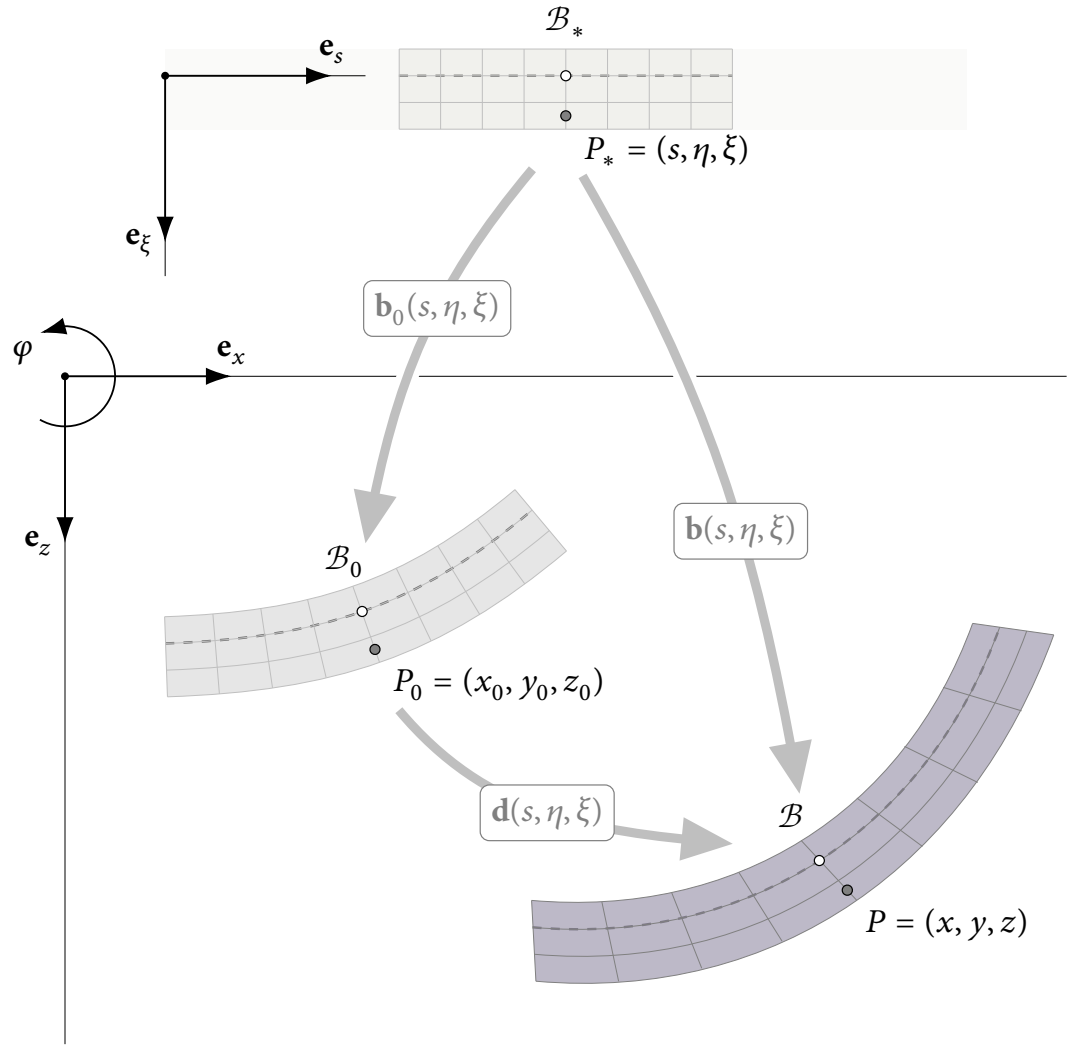
```
U = dolfinx.FunctionSpace(mesh, ...)  # mesh with gdim=3, tdim=1 or 2
u = dolfinx.Function(U)
du_dx = ufl.grad(u)  # (3x1), directional gradient
du_dt = P * du_dx    # (3x1), tangential gradient
```

- surface gradient of *vector* function in UFL

```
U = dolfinx.VectorFunctionSpace(mesh, ...)  # mesh with gdim=3, tdim=1 or 2
u = dolfinx.Function(U)
du_dx = ufl.grad(u)  # (3x3), directional gradient
du_dt = P * du_dx    # (3x3), tangential gradient / covariant gradient
```

- restriction to tangent space: $\mathbf{u}_{\text{tang}} = \mathbf{Pu}$, $\mathbf{A}_{\text{tang}} = \mathbf{PAP}$

Tangential differential calculus (TDC) | configurations



TDC for structures:
beams [4], membranes [5, 6], plates and shells (Kirchhoff [7, 8], Reissner [9])

TDC in FEniCSx/UFL | truss/cable structures

```
mesh = ...  # mesh of gdim=3, tdim=1, q = geometry order
Uf = dolfinx.fem.FunctionSpace(mesh, ("CG", p))  # p = physics order
u, δu = dolfinx.fem.Function(Uf, name='u'), ufl.TestFunction(Uf)  # displacement in R^{gdim}
```

```
# Geometry
x0 = ufl.SpatialCoordinate(mesh)  # x0 = coordinates in undeformed configuration (mesh)
t0 = ufl.geometry.Jacobian(mesh)[:, 0]  # t0 = tangent in undeformed configuration (mesh)
P = ufl.outer(t0, t0) / ufl.dot(t0, t0)
```

```
# Kinematics
b0 = x0  # placement in undeformed configuration
b = x0 + u  # placement in the deformed configuration

J0 = ufl.grad(b0)  # configuration gradient, undeformed configuration
J = ufl.grad(b)  # configuration gradient, deformed configuration

# Green-Lagrange strain tensor (gdim x gdim)
E = (J.T * J - J0.T * J0) / 2  # directional
Em = P * E * P  # tangential (in-plane = membrane)
δEm = ufl.derivative(Em, u, δu)  # 1st variation of Em

# PK2 stress tensor (gdim x gdim)
S = 2 * μ * E + λ * ufl.tr(E) * I  # directional (SVK with Lamé constants)
Sm = P * S * P  # tangential (in-plane = membrane)
Sm *= A0  # tangential stress resultant, A0 = section area

# Weak form
f = - ufl.inner(δEm, Sm) * dx  # + external virtual work
```

```
# Define and solve the nonlinear problem
F = dolfiny.function.extract_blocks(f, [δu])  # list of forms
problem = dolfiny.snesblockproblem.SNESBlockProblem(F, [u], bcs)  # nonlinear problem
problem.solve()
```

TDC in FEniCSx/UFL | membrane structures

```
mesh = ...  # mesh of gdim=3, tdim=2, q = geometry order
Uf = dolfinx.fem.FunctionSpace(mesh, ("CG", p))  # p = physics order
u, δu = dolfinx.fem.Function(Uf, name='u'), ufl.TestFunction(Uf)  # displacement in R^{gdim}
```

```
# Geometry
x0 = ufl.SpatialCoordinate(mesh)  # x0 = coordinates in undeformed configuration (mesh)
n0 = ufl.CellNormal(mesh)  # n0 = unit normal in undeformed configuration (mesh)
P = I - ufl.outer(n0, n0)  # I = ufl.Identity(gdim)
```

```
# Kinematics
b0 = x0  # placement in undeformed configuration
b = x0 + u  # placement in the deformed configuration

J0 = ufl.grad(b0)  # configuration gradient, undeformed configuration
J = ufl.grad(b)  # configuration gradient, deformed configuration

# Green-Lagrange strain tensor (gdim x gdim)
E = (J.T * J - J0.T * J0) / 2  # directional
Em = P * E * P  # tangential (in-plane = membrane)
δEm = ufl.derivative(Em, u, δu)  # 1st variation of Em

# PK2 stress tensor (gdim x gdim)
S = 2 * μ * E + λ * ufl.tr(E) * I  # directional (SVK with Lamé constants, plane stress)
Sm = P * S * P  # tangential (in-plane = membrane)
Sm *= h0  # tangential stress resultant, h0 = membrane thickness

# Weak form
f = - ufl.inner(δEm, Sm) * dx  # + external virtual work
```

```
# Define and solve the nonlinear problem
F = dolfiny.function.extract_blocks(f, [δu])  # list of forms
problem = dolfiny.snesblockproblem.SNESBlockProblem(F, [u], bcs)  # nonlinear problem
problem.solve()
```

TDC in FEniCSx/UFL | shell structures (Reissner/Naghdi)

```
mesh = ...  # mesh of gdim=3, tdim=2, q = geometry order
Uf = dolfinx.fem.FunctionSpace(mesh, ("CG", p))  # p = physics order
Rf = dolfinx.fem.FunctionSpace(mesh, ("CG", p))  # p = physics order
u, δu = dolfinx.fem.Function(Uf, name='u'), ufl.TestFunction(Uf)  # displacement in R^{gdim}
r, δr = dolfinx.fem.Function(Rf, name='r'), ufl.TestFunction(Rf)  # rotation in R^{gdim}
```

```
# Geometry
x0 = ufl.SpatialCoordinate(mesh)  # x0 = coordinates in undeformed configuration (mesh)
n0 = ufl.CellNormal(mesh)  # n0 = unit normal in undeformed configuration (mesh)
P = I - ufl.outer(n0, n0)  # I = ufl.Identity(gdim)
```

```
# Through-tickness variable
Ξ = dolfinx.fem.FunctionSpace(mesh, ("DG", q))
ξ = dolfinx.fem.Function(Ξ, name='ξ')

# Kinematics
b0 = x0 + ξ * n0  # placement in undeformed configuration
b = (x0 + u) + ξ * (R * n0) # placement in the deformed configuration

# Configuration gradient, un-deformed configuration
J0 = ufl.grad(b0) - ufl.outer(n0, n0)  # = P * ufl.grad(x0) + ufl.grad(ξ * n0)
J0 = ufl.algorithms.apply_algebra_lowering.apply_algebra_lowering(J0)
J0 = ufl.algorithms.apply_derivatives.apply_derivatives(J0)
J0 = ufl.replace(J0, {ufl.grad(ξ): n0})

# Configuration gradient, deformed configuration
J = ufl.grad(b) - ufl.outer(n0, n0)  # = P * ufl.grad(x0) + ufl.grad(u + ξ * (R * n0))
J = ufl.algorithms.apply_algebra_lowering.apply_algebra_lowering(J)
J = ufl.algorithms.apply_derivatives.apply_derivatives(J)
J = ufl.replace(J, {ufl.grad(ξ): n0})
```

TDC in FEniCSx/UFL | shell structures (Reissner/Naghdi) II

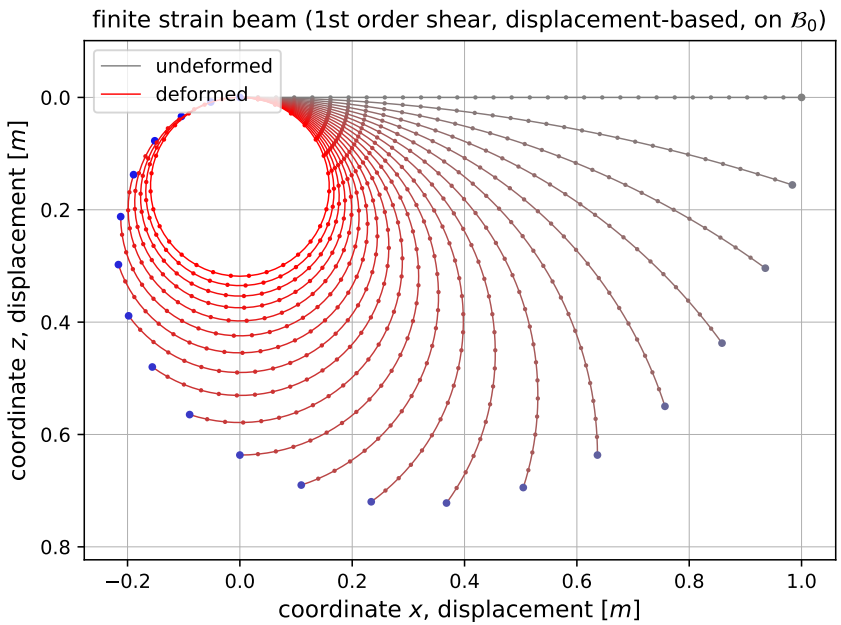
```
# Green-Lagrange strains (total): determined by deformation kinematics
E = (J.T * J - J0.T * J0) / 2  # directional GL strain
# Membrane strain
Em = P * ufl.replace(E, {ξ: 0.0}) * P
# Bending strain
Eb = ufl.diff(E, ξ)
Eb = ufl.algorithms.apply_algebra_lowering.apply_algebra_lowering(Eb)
Eb = ufl.algorithms.apply_derivatives.apply_derivatives(Eb)
Eb = P * ufl.replace(Eb, {ξ: 0.0}) * P
# Shear strain
Es = ufl.replace(E, {ξ: 0.0}) - P * ufl.replace(E, {ξ: 0.0}) * P

# Variation of elastic Green-Lagrange strains
δEm = ufl.derivative(Em, u, δu) + ufl.derivative(Em, r, δr)
δEs = ufl.derivative(Es, u, δu) + ufl.derivative(Es, r, δr)
δEb = ufl.derivative(Eb, u, δu) + ufl.derivative(Eb, r, δr)
```

```
# PK2 stress
S = 2 * μ * E + λ * ufl.tr(E) * I  # directional (SVK with Lamé constants, plane stress)
# Membrane stress
Sm = P * ufl.replace(S, {ξ: 0.0}) * P
# Bending stress
Sb = ufl.diff(S, ξ)
Sb = ufl.algorithms.apply_algebra_lowering.apply_algebra_lowering(Sb)
Sb = ufl.algorithms.apply_derivatives.apply_derivatives(Sb)
Sb = P * ufl.replace(Sb, {ξ: 0.0}) * P
# Shear strain
Ss = ufl.replace(S, {ξ: 0.0}) - P * ufl.replace(S, {ξ: 0.0}) * P

# Stress resultant tensors
Sm *= h0; Ss *= h0; Sb *= h0**3 / 12
```

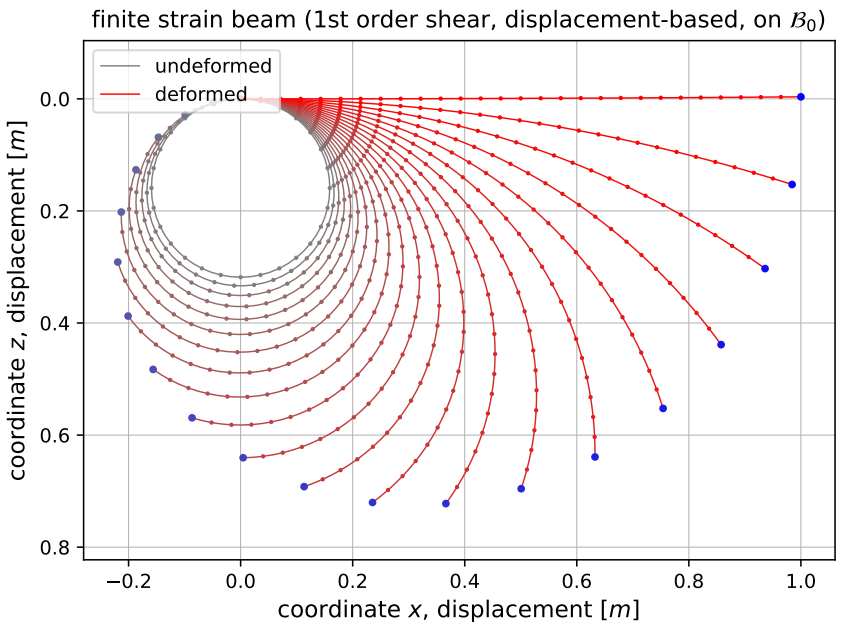
```
# Weak form (shown without reduced integration, drill removal)
f = - ufl.inner(δEm, Sm) * dx - ufl.inner(δEb, Sb) * dx - ufl.inner(δEs, Ss) * dx  # + ext. virtual work
```

TDC Reissner beam | nonlinear analysis (load stepping)

- straight/curved beam subject to tip moment
- planar beam embedded in 3d space (x-z plane)

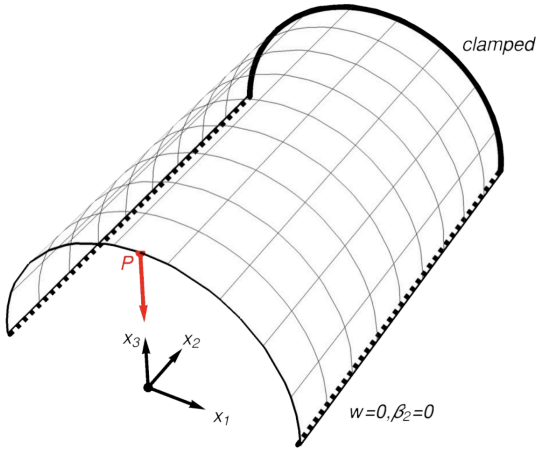


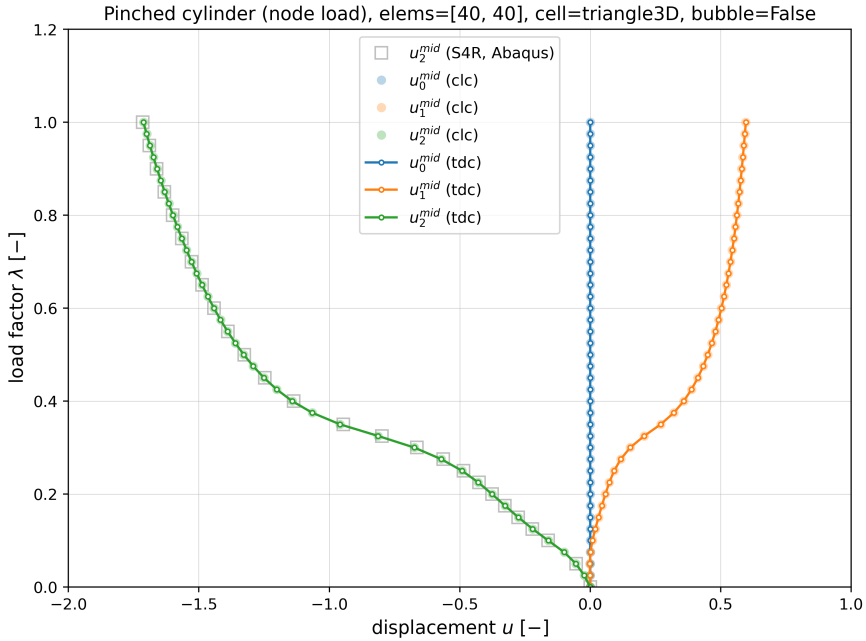
roll-up from straight configuration



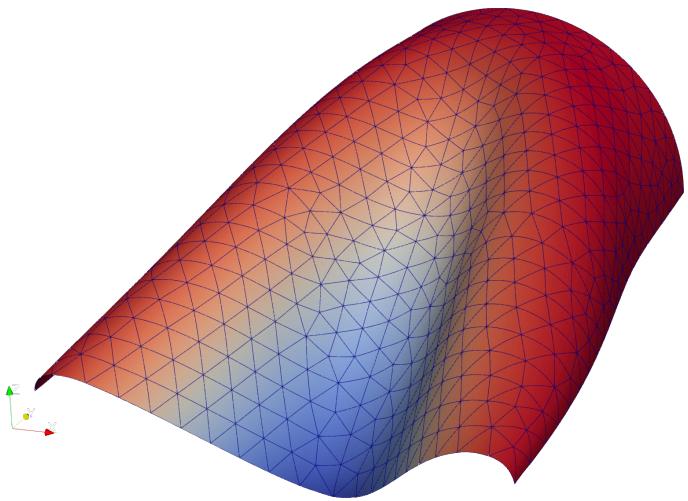
roll-out from curved configuration

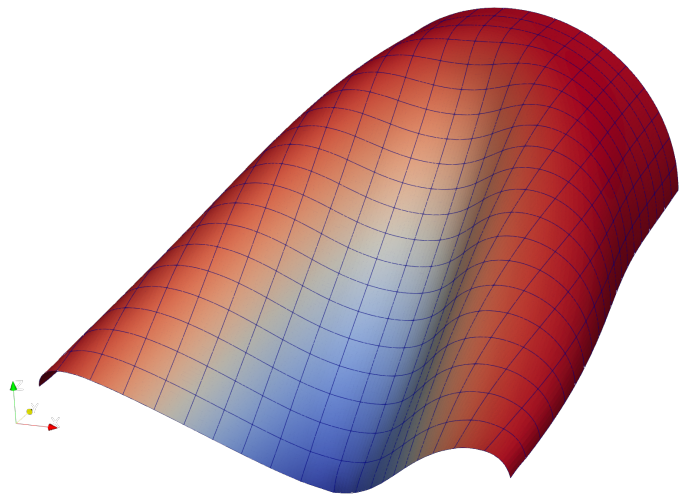
TDC Naghdi shell | nonlinear analysis (load stepping)



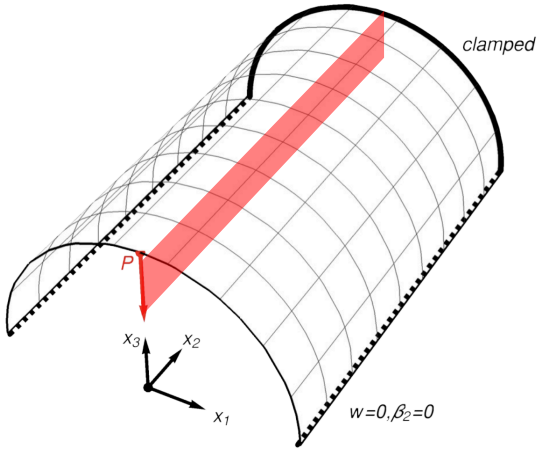


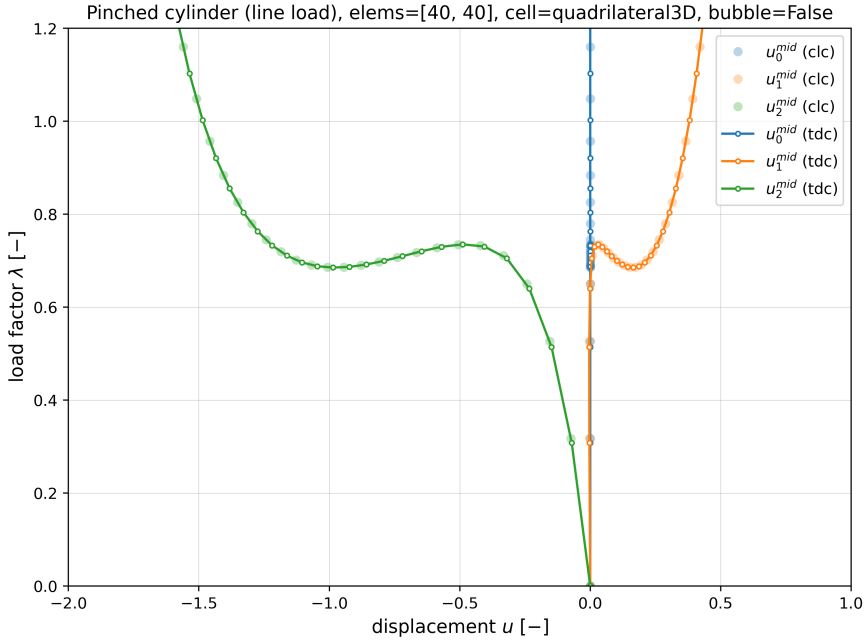
- cylinder subject to *nodal* load, [10]
- fenics-shells, [2]



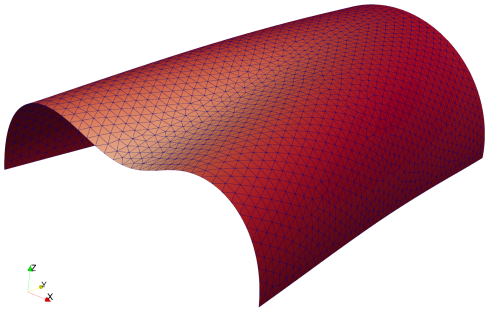


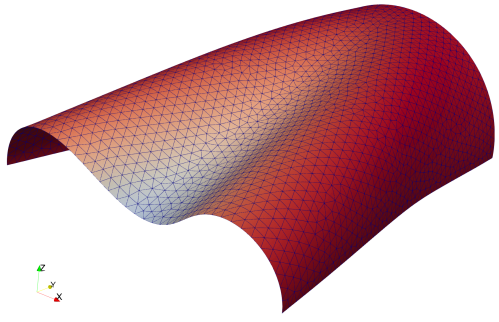
TDC Naghdi shell | nonlinear analysis (continuation)

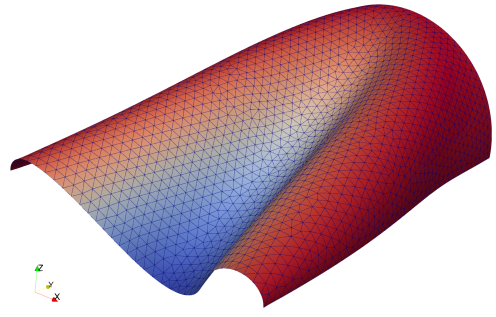




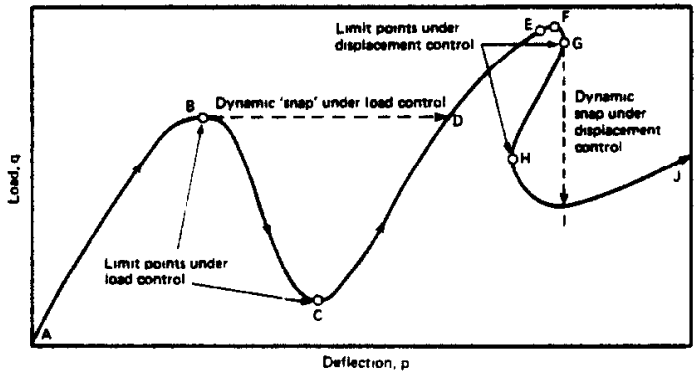
- cylinder subject to *line load*
- fenics-shells, [2]





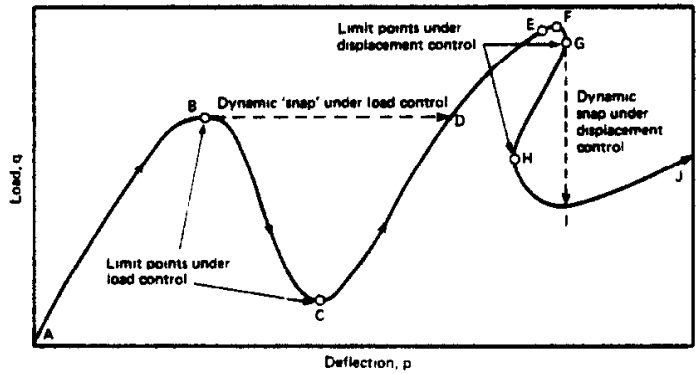


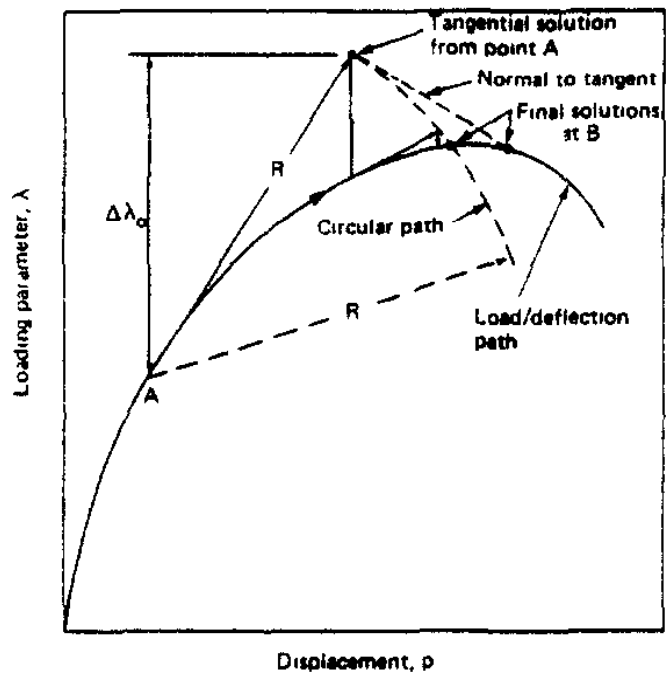
Continuation | follow the equilibrium path



$$\mathbf{r}(\mathbf{u}, \lambda) = -\mathbf{f}_i(\mathbf{u}) + \lambda \mathbf{f}_e(\mathbf{u}) \stackrel{!}{=} \mathbf{0}$$

Continuation | follow the equilibrium path





$$\mathbf{r}(\mathbf{u}, \lambda) = -\mathbf{f}_i(\mathbf{u}) + \lambda \mathbf{f}_e(\mathbf{u}) \stackrel{!}{=} \mathbf{0}$$

Riks arc-length method (1979) [11]

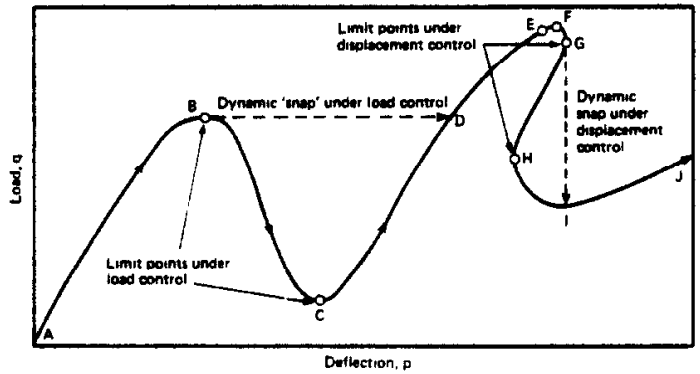
Solve for $\mathbf{u}_{k+1} = \mathbf{u}_k + \Delta\mathbf{u}$, $\lambda_{k+1} = \lambda_k + \Delta\lambda$ in an incremental procedure such that

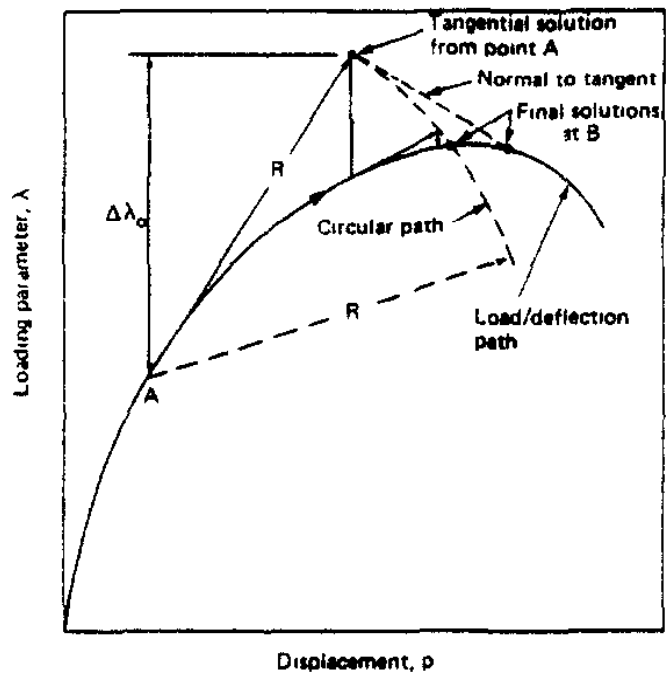
$$\mathbf{0} = -\mathbf{f}_i(\mathbf{u}_{k+1}) + \lambda_{k+1}\mathbf{f}_e(\mathbf{u}_{k+1}) \quad (1)$$

$$0 = \Delta\mathbf{u}^\mathsf{T}\Delta\mathbf{u} + \psi(\Delta\lambda\mathbf{f}_e)^\mathsf{T}(\Delta\lambda\mathbf{f}_e) - \Delta s^2 \quad (2)$$

$\rightarrow$ augmented algebraic system (ndof + 1)

Continuation | follow the equilibrium path





$$\mathbf{r}(\mathbf{u}, \lambda) = -\mathbf{f}_i(\mathbf{u}) + \lambda \mathbf{f}_e(\mathbf{u}) \stackrel{!}{=} \mathbf{0}$$

Riks arc-length method (1979) [11]

Solve for $\mathbf{u}_{k+1} = \mathbf{u}_k + \Delta\mathbf{u}$, $\lambda_{k+1} = \lambda_k + \Delta\lambda$ in an incremental procedure such that

$$\mathbf{0} = -\mathbf{f}_i(\mathbf{u}_{k+1}) + \lambda_{k+1}\mathbf{f}_e(\mathbf{u}_{k+1}) \qquad (1)$$

$$0 = \Delta\mathbf{u}^\mathsf{T}\Delta\mathbf{u} + \psi(\Delta\lambda\mathbf{f}_e)^\mathsf{T}(\Delta\lambda\mathbf{f}_e) - \Delta s^2 \qquad (2)$$

$\rightarrow$ augmented algebraic system (ndof + 1)

Crisfield's approach (1981) [12]

- linearise (1) and isolate NR corrections $\mathrm{d}\mathbf{u}_i$ and $\mathrm{d}\mathbf{u}_e$ through *two* linear solves
- solve (2) for correction $\mathrm{d}\lambda$

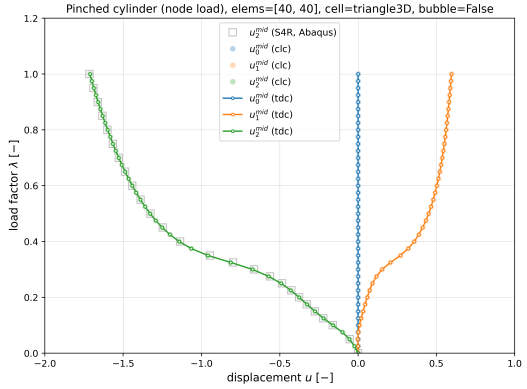
Continuation | arc-length à la Crisfield in `dolfiny`

- `SNESBlockProblem` interfaces to PETSc SNES

Continuation | arc-length à la Crisfield in `dolfiny`

- `SNESBlockProblem` interfaces to PETSc SNES

```python
# Define state as (ordered) list of functions
m, δm = [u, r], [δu, δr]

# Load factor
λ = dolfinx.fem.Constant(mesh, 1.0)

# Define form involving the load factor
f = -f_i + λ * f_e

# Overall form (as list of forms)
F = dolfiny.function.extract_blocks(f, δm)

import dolfiny.snesblockproblem

# Create nonlinear problem: SNES
problem = dolfiny.snesblockproblem.SNESBlockProblem(F, m, bcs)

# Load increment procedure
for factor in np.linspace(0.0, 1.0, 20):

        λ.value = factor  # set/update load factor
        problem.solve()  # solve nonlinear problem
```

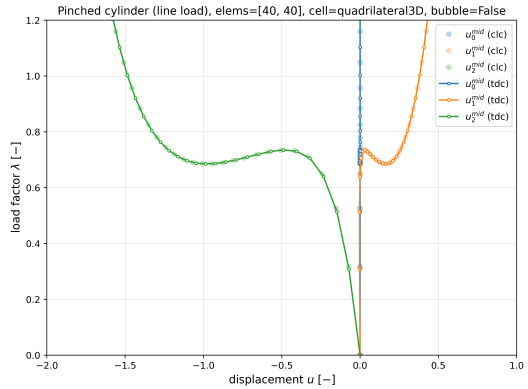



```
+++ Processing load factor λ = 0.4750

### SNES iteration 0
# sub  0 |x|=2.730e+01 |dx|=3.689e-08 |r|=2.500e+01 (u)
# sub  1 |x|=5.101e+01 |dx|=8.504e-08 |r|=1.123e-11 (r)
# all    |x|=5.785e+01 |dx|=9.270e-08 |r|=2.500e+01

### SNES iteration 1
# sub  0 |x|=2.889e+01 |dx|=1.777e+00 |r|=7.078e+02 (u)
# sub  1 |x|=5.341e+01 |dx|=3.531e+00 |r|=1.493e+01 (r)
# all    |x|=6.072e+01 |dx|=3.952e+00 |r|=7.080e+02

...

### SNES iteration 6
# sub  0 |x|=2.879e+01 |dx|=8.100e-06 |r|=2.980e-08 (u)
# sub  1 |x|=5.319e+01 |dx|=1.887e-05 |r|=5.139e-10 (r)
# all    |x|=6.048e+01 |dx|=2.053e-05 |r|=2.980e-08
```

Continuation | arc-length à la Crisfield in `dolfiny`

- `SNESBlockProblem` interfaces to PETSc SNES
- `Crisfield` as thin wrapper around `SNESBlockProblem`
- custom update method to SNES solver realises Crisfield's approach

```python
# Define state as (ordered) list of functions
m, δm = [u, r], [δu, δr]

# Load factor
λ = dolfinx.fem.Constant(mesh, 1.0)

# Define form involving the load factor
f = -f_i + λ * f_e

# Overall form (as list of forms)
F = dolfiny.function.extract_blocks(f, δm)

import dolfiny.snesblockproblem

# Create nonlinear problem: SNES
problem = dolfiny.snesblockproblem.SNESBlockProblem(F, m, bcs)

import dolfiny.continuation

# Create continuation problem context
continuation = dolfiny.continuation.Crisfield(problem, λ)

# Initialise continuation problem
continuation.initialise(ds)

# Arc-length procedure with Crisfield update
for k in range(20):

        continuation.solve_step(ds)
```




```
*** Continuation step 6

### SNES iteration 0
# sub  0 |x|=1.056e+01 |dx|=2.465e-07 |r|=9.655e+02 (u)
# sub  1 |x|=2.140e+01 |dx|=4.804e-07 |r|=1.517e+01 (r)
# all    |x|=2.386e+01 |dx|=5.399e-07 |r|=9.656e+02
# arc    |x|=7.540e-01 |dx|=0.000e+00 |r|=2.602e-18 (λ)

...

### SNES iteration 4
# sub  0 |x|=1.059e+01 |dx|=5.244e-03 |r|=3.021e-03 (u)
# sub  1 |x|=2.130e+01 |dx|=9.689e-03 |r|=6.003e-05 (r)
# all    |x|=2.378e+01 |dx|=1.102e-02 |r|=3.022e-03
# arc    |x|=7.347e-01 |dx|=9.343e-06 |r|=0.000e+00 (λ)

### SNES iteration 5
# sub  0 |x|=1.058e+01 |dx|=4.939e-07 |r|=1.010e-09 (u)
# sub  1 |x|=2.129e+01 |dx|=9.135e-07 |r|=6.241e-12 (r)
# all    |x|=2.377e+01 |dx|=1.039e-06 |r|=1.010e-09
# arc    |x|=7.347e-01 |dx|=8.853e-10 |r|=1.952e-17 (λ)
```

Summary and outlook

`https://github.com/michalhabera/dolfiny`

Now

- TDC as re-formulation of existing models for thin-walled structures
- TDC demo for truss, membrane, beam and shell (nonlinear)
- supports higher-order triangle and quadrilateral meshes
- continuation method (Crisfield's approach to arc-length procedure)
- equilibrium paths associated with *snap-through* and *snap-back*

Future

- explore robustness of various (mixed) formulations in TDC setting
- coupling domains of different topological dimension
- further improvement of continuation interface

References

[1] Marie E Rognes et al. "Automating the solution of PDEs on the sphere and other manifolds in FEniCS 1.2". English. In: *Geoscientific Model Development* 6.6 (Dec. 2013), pp. 2099–2119. ISSN: 1991-959X. DOI: 10.5194/gmd-6-2099-2013.

[2] Jack S. Hale et al. "Simple and extensible plate and shell finite element models through automatic code generation tools". In: *Computers & Structures* 209 (2018), pp. 163–181. ISSN: 0045-7949. DOI: https://doi.org/10.1016/j.compstruc.2018.08.001. URL: https://www.sciencedirect.com/science/article/pii/S0045794918306126.

[3] Jeremy Bleyer. *Numerical Tours of Computational Mechanics with FEniCS*. Zenodo, 2018. DOI: 10.5281/zenodo.1287832.

[4] P. Hansbo, M.G. Larson and K. Larsson. "Variational formulation of curved beams in global coordinates". In: *Computational Mechanics* 53 (2014), pp. 611–623. DOI: https://doi.org/10.1007/s00466-013-0921-0.

[5] P. Hansbo, M.G. Larson and F. Larsson. "Tangential differential calculus and the finite element modeling of a large deformation elastic membrane problem". In: *Computational Mechanics* 56 (2015), pp. 87–95. DOI: https://doi.org/10.1007/s00466-015-1158-x.

[6] T.P. Fries and D. Schöllhammer. "A unified finite strain theory for membranes and ropes". In: *Computer Methods in Applied Mechanics and Engineering* 365 (2020), p. 113031. ISSN: 0045-7825. DOI: https://doi.org/10.1016/j.cma.2020.113031. URL: https://www.sciencedirect.com/science/article/pii/S0045782520302152.

[7] P. Hansbo and M.G. Larson. "Continuous/discontinuous finite element modelling of Kirchhoff plate structures in R3 using tangential differential calculus". In: *Computational Mechanics* 60 (2017), pp. 693–702. DOI: https://doi.org/10.1007/s00466-017-1431-2.

[8] D. Schöllhammer and T.P. Fries. "Kirchhoff–Love shell theory based on tangential differential calculus". In: *Computational Mechanics* 64 (2019), pp. 113–18831. DOI: https://doi.org/10.1007/s00466-018-1659-5.

[9] D. Schöllhammer and T.P. Fries. "Reissner–Mindlin shell theory based on tangential differential calculus". In: *Computer Methods in Applied Mechanics and Engineering* 352 (2019), pp. 172–188. ISSN: 0045-7825. DOI: https://doi.org/10.1016/j.cma.2019.04.018. URL: https://www.sciencedirect.com/science/article/pii/S004578251930221X.

[10] K. Y. Sze, X. H. Liu and S. H. Lo. "Popular Benchmark Problems for Geometric Nonlinear Analysis of Shells". In: *Finite Elem. Anal. Des.* 40.11 (July 2004), pp. 1551–1569. ISSN: 0168-874X. DOI: 10.1016/j.finel.2003.11.001. URL: https://doi.org/10.1016/j.finel.2003.11.001.

[11] E. Riks. "An incremental approach to the solution of snapping and buckling problems". In: *International Journal of Solids and Structures* 15.7 (1979), pp. 529–551. ISSN: 0020-7683. DOI: https://doi.org/10.1016/0020-7683(79)90081-7. URL: https://www.sciencedirect.com/science/article/pii/0020768379900817.

[12] M.A. Crisfield. "A fast incremental/iterative solution procedure that handles "snap-through"". In: *Computers & Structures* 13.1 (1981), pp. 55–62. ISSN: 0045-7949. DOI: https://doi.org/10.1016/0045-7949(81)90108-5. URL: https://www.sciencedirect.com/science/article/pii/0045794981901085.