Wrinkling instabilities of hyperelastic materials with FEniCS

Carlos. A Lugo.

Department of Plant Sciences, University of Cambridge, Downing Street, Cambridge CB2 3EA, U.K.

June 15, 2021

Abstract

Numerical recipes to study wrinkling instabilities in hyperelastic materials are presented. Specifically, we describe the cases of a stretched layer, the compression of bilayers in two and three dimensions and growth induced wrinkles.

1 Introduction

The finite element method is a popular way to solve partial differential equations in mathematics and physics. It is theoretically well established and it can be implemented in several freely available frameworks as well as in commercial packages. Such maturity implies an overabundance of literature on both fronts, theoretically and numerically. In the case of non-linear mechanics, many of the current research interests involve disciplines such as biology and medicine, which makes the learning curve to students of such disciplines steeper. These notes are aimed at such audiences as well as at audiences with a mathematics and engineering background who wish a hands-on tutorial into solving non-linear mechanical equations.

The worked examples make use of the FEniCS framework [ABH+15] and FEniCS mechanics [RAS19]. All the scripts are coded using the python language. The results here were obtained using dolfin 2019.1.0.

2 Wrinkling of a Neo-Hookean elastomer.

Hyperelastic materials [Ama18] are defined by the existence of a strain density functional W from which we can derive the stress-strain relationship by:

$$\mathbb{S} = \frac{\partial W(\mathbb{F})}{\partial \mathbb{F}}$$

Where $\mathbb{F} = \mathbb{I} + \nabla \mathbf{u}$ is the deformation gradient. For the case of a Neo-Hookean incompressible solid $W = \frac{\mu}{2}(I_1 - 3) - p((J-1)$, where I_1 and $J = I_3^{1/2}$ are the first invariant of the Cauchy stress tensor $\mathbb{C} = \mathbb{F}^T\mathbb{F}$ and the third invariant of $\mathbb{F}$. The parameter p, namely the hydrostatic pressure, is a Lagrange multiplier which is included to ensure the incompressibily constraint $J = 1$.

The first example we consider is illustrated in Fig. 1 and consists of a thin rectangular Neo-Hookean layer clamped at the boundaries and for which the right clamped subdomain is displaced by an amount Δx. This system is thoroughly investigated theoretically and experimentally in [FHS18]. This system is chosen to illustrate several aspects of the python implementation.

The first thing to notice is that the incomprehensibility constraint requires a solution space which is mixed, as it requires the solution space for the displacement $\mathbf{u}$ alongside the scalar field p. This is easily implemented if we use the package FEniCS mechanics which implements this material model. See the documentation here or go to the next url: `https://shaddenlab.gitlab.io/fenicsmechanics/`

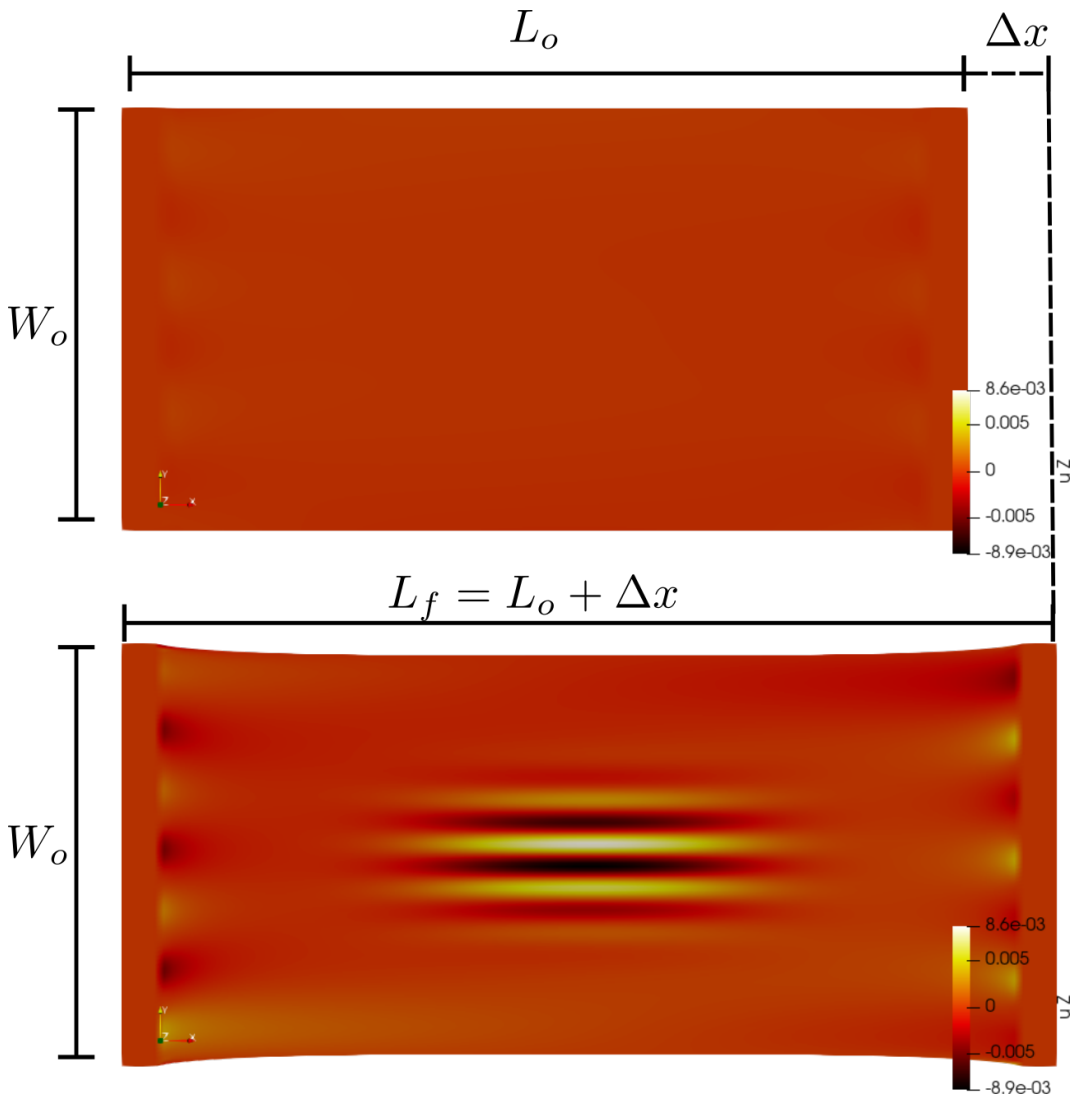

Figure 1: Undeformed initial configuration and solution for the slender Neo-hookean incompressible system.

```
from dolfin import *
import fenicsmechanics as fm
import numpy as np
```

2.1 Mesh preparation and boundaries.

To define the domain, the mesh parameters and the clamped boundary conditions, we need to define a rectangular mesh which fullfills $\Omega_o = [0, L_o] \times [0, W_o] \times [0, h]$.

In this case we use $L_o = 2W_o$. As the domain is a slender layer we only use one element in the depth direction, whereas for the width and length we use 100 cells on each direction.

```

#Number of elements
Nr=100
Nt=100
Nz=1
#Dimensions
Ly=5
Lx=2*Ly
Lz=1.00
#Dolfin Mesh
```

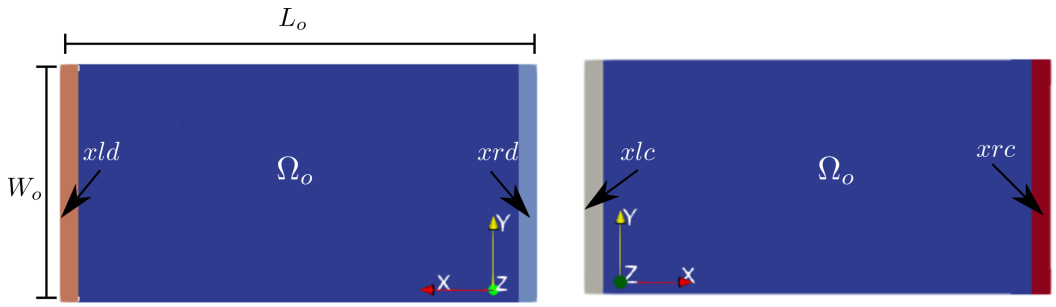

Figure 2: Pre-compiled subdomain of the slender sheet, the figure shows the geometry and both top and bottom surfaces of the system.

```
meshname=BoxMesh(Point(0.0,0.0,0.0),Point(Lx,Ly,Lz),Nr,Nt,Nz)

#Desired depth h
Lzn=0.005

#Transformation of the unit mesh into a slender rectangle

x=meshname.coordinates()[:,0]
y=meshname.coordinates()[:,1]
z=meshname.coordinates()[:,2]
def transmesh(x,y,z,Lzn):#
    return [x,y,Lzn*z]
qn=transmesh(x,y,z,Lzn)
qnx=np.array(qn).transpose()
meshname.coordinates()[:]=qnx
```

To define the clamped boundaries we use a clamp width of $dlx = L_o/20$, then define the four surfaces $(x, y, 0)$, (x, y, h) for which $x < lxa$, and $x > L_o - dlx$.

```
lxa=1.0*Lx/20 #dlx
xlc =  CompiledSubDomain("(near(x[2], side) && on_boundary) && x[0]<lxa", side = 0.0,
    lxa=lxa)
xld =  CompiledSubDomain("(near(x[2], side) && on_boundary) && x[0]<lxa", side = Lzn,
    lxa=lxa)
xrc = CompiledSubDomain("(near(x[2], side) && on_boundary) && x[0]>lxb", side = 0.0,
    lxb=Lx-lxa)
xrd = CompiledSubDomain("(near(x[2], side) && on_boundary) && x[0]>lxb", side = Lzn,
    lxb=Lx-lxa)

boundary_parts = MeshFunction("size_t", meshname, meshname.topology().dim() - 1)
boundary_parts.set_all(0)
xlc.mark(boundary_parts,1)
xld.mark(boundary_parts,2)
xrc.mark(boundary_parts,3)
xrd.mark(boundary_parts,4)
bds=File("bmarks.pvd")
bds << boundary_parts
```

The Meshfunction called boundary parts marks the desired subdomains, which are then exported to a file called *"bmarks.pvd"* so it can be inspected on a visualisation program such as paraview or python's own packages. Fig. 2 shows plots of the boundary regions on both sizes of the mesh as defined above. This completes the mesh preparation and boundary definitions.

2.2 Solutions.

To complete the formulation of the problem we need to define a set of boundary conditions, the material type and its parameters alongside some parameters regarding the solver of choice. For this problem the conditions we need are Dirichlet boundary conditions, for the domains on the left with a zero displacement imposed in the x direction and for the domains on the right the desired Δx displacement which will stretch the rest of the domain. We also stretch the domain by a few iterations in order

to trigger the instability, to do so we apply the rule $y \rightarrow ky$ to the points in all the four boundary domains defined above.

We define the expressions for the conditions as:

```
cr = Expression(("x0","kr*x[1]","z0"),x0 = 0.0, kr = 0.0, z0=0.0,degree=1)
cl = Expression(("x0","kl*x[1]","z0"),x0 = 0.0, kl = 0.0, z0=0.0,degree=1)
```

The parameters of these expressions will then be updated at the moment of solving the problem. The mechanics problem is then written as:

```
material = {
    'type': 'elastic',
    'const_eqn': 'neo_hookean',
    'incompressible': True,
    'kappa': 10e9, # Pa
    'mu': 1.5e6 # Pa
}
mesh = {
    'mesh_file': meshname,
    'boundaries': boundary_parts
}
formulation = {
    'element': 'p2-p1',
    'domain': 'lagrangian',
    'bcs': {
        'dirichlet': {
            'displacement': [cl,cl,cr,cr],
            'regions': [1,2,3,4],
            }
    }
}

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

filen = "displacement.pvd"
problem = fm.SolidMechanicsProblem(config)
solver = fm.SolidMechanicsSolver(problem, fname_disp=filen)
solver.set_parameters(linear_solver="mumps")
solver.set_parameters(newton_maxIters=500)
solver.set_parameters(newton_abstol=1e-7)
solver.set_parameters(newton_reltol=1e-6)
```

As we can see in the table above, once the expressions for the boundary conditions are defined, a dictionary with the material specifications, the mesh, the boundary conditions and some configuration settings is defined in order to define the problem.

The boundary conditions are defined in the same order as the marked domains of the Meshfunction.

A file name is provided in order to store the solutions. We single out the solver object specifications which we use in this particular problem.

Once all the pieces are in place, the static problem is solved by updating the values of the boundary condition expressions. In this case increasing the value of $x0$ and $y0$.

```
d1=0.000
d2=0.000
ky=0.0
kx=0.0
for j in range(400):
    if j<50:
        DX=0.01
    else:
        DX=0.001
    if j<5:
        DY=0.001
    else:
        DY=1e-9

    d2+=DY
```

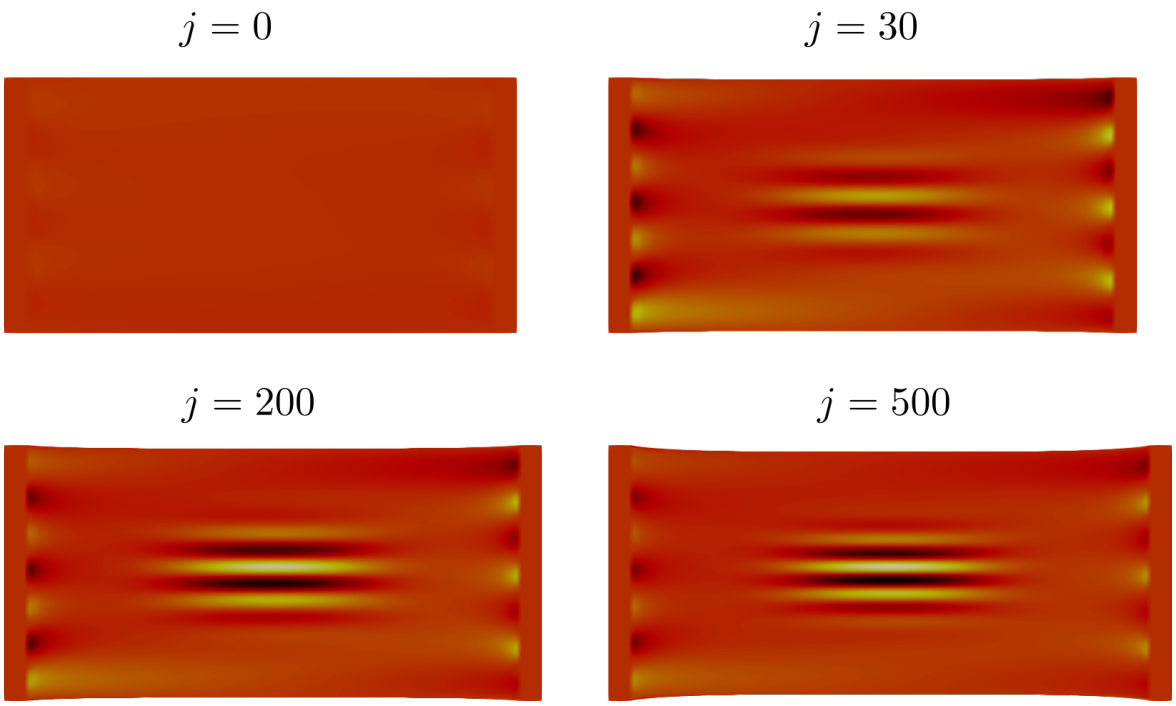

Figure 3: Evolution of the solution to increasing stretch of the initial configuration.

```
    d1+=DX
    cr.x0=d1
    ky=d2/Ly
    cl.kl=ky
    cr.kr=ky
    solver.full_solve()
```

On each iteration of the loop, we increase the value of the variables d_1 and d_2, which are the current value of the displacements by adding increments of value DX and DY which are then used to update the boundary conditions. Figure 3 shows the output of the displacement solutions at several values of the iteration loop.

3 Volumetric compression of a Neo-Hookean bilayer.

Now we present a slightly more complicated system, which is used to model nanostructures in samples which are orders of magnitude larger than the domain under consideration, after being subjected to an external deformation. For this example, we use a pure FEniCS implementation as the system now has two subdomains with different mechanical properties, this system can be solved by solving the Euler-Lagrange equations for the stress or due to the existence of the strain energy density functional, directly solving the stationary action equation for the energy, i.e. finding the configuration which cancels its first variation. We solve the latter, and in the third example we approach the problem solving the Euler-Lagrange system.

The system is now composed by a volume $\Omega_o = [0, L_x] \times [0, L_y] \times [0, H]$ in which all the points $z > H - h$ possess a shear modulus μ_f larger than the points in the rest, which have a shear modulus μ_s. The indices r and s indicate "film" and "substrate". The strain energy function is given by

$$W_j = \frac{\mu_j}{2}(I - 3) - \mu_j \log(J) + \frac{\lambda_j}{2} \log(J)^2$$

.

Where μ_j and λ_j are the Lamé material parameters of the film and substrate. To obtain the solutions like the one shown in Fig. 4 again we need to define a mesh, set of boundary subdomains and the respective solution spaces.

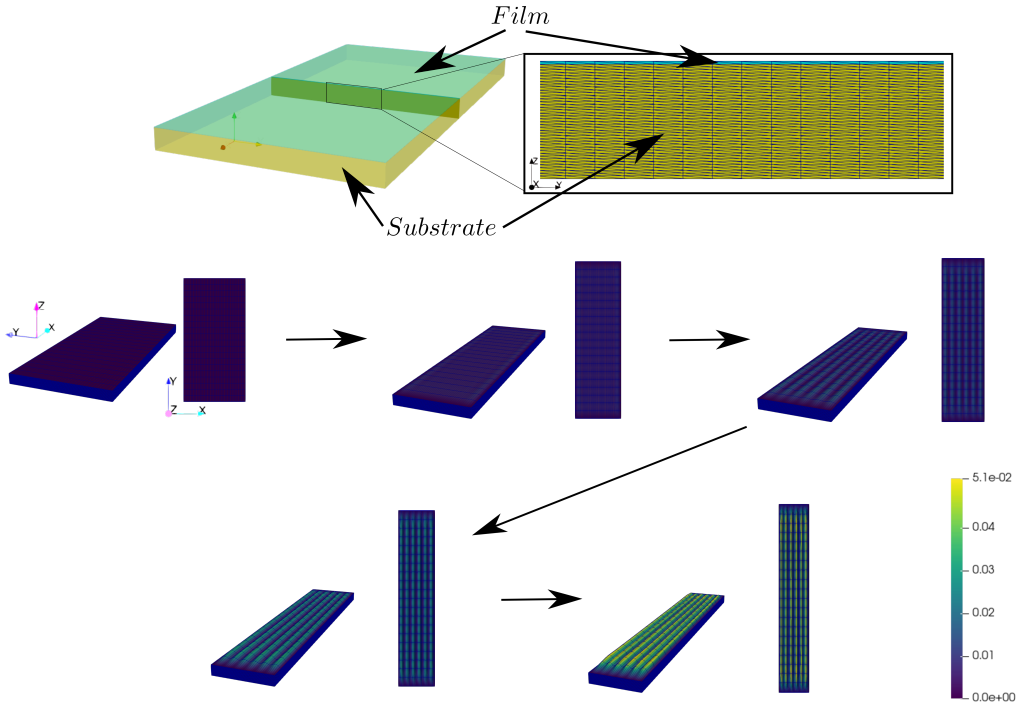

Figure 4: Domain geometry and subdomains for the bilayer system (TOP), and evolution of the deformed configuration for increasing displacement of the lateral and bottom boundary surfaces.

3.1 Mesh preparation and boundaries.

The next lines of code show how to prepare the mesh which we use in the simulation, we use a domain of $20 \times 40 \times 41$ hexahedral elements which we use to define a configuration $\Omega_0 : [0, Lx] \times [0, L_y] \times [0, Lz]$ with $L_x = 2L_y$ and $L_z = 0.1$.

```
from dolfin import *
import numpy as np
Nx=20
Ny=40
Nz=41
Ly=1.0
Lx=2.0#*Ly
Lz=0.1

mesh = UnitCubeMesh.create(Nx, Ny, Nz, CellType.Type.hexahedron)
mesh.coordinates()[:, 0] = mesh.coordinates()[:, 0]*Lx
mesh.coordinates()[:, 1] = mesh.coordinates()[:, 1]*Ly
mesh.coordinates()[:, 2] = mesh.coordinates()[:, 2]*Lz

f = HDF5File(mesh.mpi_comm(), "meshq.hdf5", 'w')
f.write(mesh, "mesh")
```

The previous code generates the mesh and stores it in a file which is called in the solver script. As with the previous example, once the mesh is generated, we define subdomains over it which will represent the film and the boundary surfaces.

The material domains and a file which stores the domains for visualization, as shown in Fig. 4 is also created as shown below.

```
Ho=1.2*(Lz/Nz)
dx1=Lx/10
mesh = Mesh()
f = HDF5File(mesh.mpi_comm(), "meshq.hdf5", 'r')
f.read(mesh, "mesh", False)
materials=MeshFunction("size_t",mesh,mesh.topology().dim())
## Mark boundary subdomians
film=CompiledSubDomain("x[2]>=Rcut",Rcut=Lz-Ho)
```

```
materials.set_all(0)
film.mark(materials,1)
bmats=File("matx.pvd")
bmats << materials
```

Next the boundaries at the bottom, and all the lateral surfaces of the domain are defined, added to a MeshFunction and stored as follows:

```
left =  CompiledSubDomain("near(x[0], side) && on_boundary", side = 0.0)
bott =  CompiledSubDomain("near(x[2], side) && on_boundary", side = 0.0)
right = CompiledSubDomain("near(x[0], side) && on_boundary", side = Lx)
northx= CompiledSubDomain("near(x[1], side) && on_boundary", side = Ly)
southx= CompiledSubDomain("near(x[1], side) && on_boundary", side = 0.0)

boundary_parts = MeshFunction("size_t", mesh, mesh.topology().dim() - 1)
boundary_parts.set_all(0)
bott.mark(boundary_parts, 3)
left.mark(boundary_parts, 1)
right.mark(boundary_parts, 2)
southx.mark(boundary_parts, 4)
northx.mark(boundary_parts, 5)
bmark = File("bmarks_mark.pvd")
bmark << boundary_parts
```

To impose the domain deformation we define the listed expressions below, which are then updated accordingly when solving the problem.

```
cl = Expression(("x0","ky*x[1]","z0"),x0 = 0.0, ky = 0.0, z0=0.00,degree=1)
cr = Expression(("x0","ky*x[1]","z0"),x0 = 0.0, ky = 0.0, z0=0.00,degree=1)
cb = Expression(("kx*x[0]","ky*x[1]","z0"),kx= 0.0, ky = 0.0, z0=0.00,degree=1)
cs = Expression(("kx*x[0]","ky*x[1]","z0"),kx= 0.0, ky = 0.0, z0=0.00,degree=1)
cn = Expression(("kx*x[0]","y0","z0"),kx= 0.0, y0 = 0.0, z0=0.00,degree=1)
bcl = DirichletBC(V, cl, left)
bcr = DirichletBC(V, cr, right)
bcb = DirichletBC(V, cb, bott)
bcs = DirichletBC(V, cs, southx)
bcn = DirichletBC(V, cn, northx)
```

These boundary expressions allow us to move a boundary by an amount D_y and compress the volume by an amount D_x and seamlessly map the point in the surfaces into $k_x x$ and $k_y y$, accordingly. If we wish to keep the in plane area fixed before and after the stretching, then increasing D_y must be coupled with a decrease in the orthogonal direction. A simple calculation leads to the condition $k_x k_y = 1$ or $Dx = \frac{L_x D_y}{L_y - D_y}$ for a given D_y.

3.2 Solutions.

To complete the example we now just need to set the variational problem, this is done as shown below. As it is a simple hyperelastic problem on a compressible material, the solution space is not mixed. It only requires a weak form over the entire volume. This is done as follows:

```
V = VectorFunctionSpace(mesh,"CG",1)
# Define functions
du = TrialFunction(V)            # Incremental displacement
v  = TestFunction(V)             # Test function
u  = Function(V)                 # Displacement from previous iteration
B  = Constant((0.0, -0.0,0.0))  # Body force per unit volume
# Elasticity parameters
E1, nu1 = 1e6, 0.48
E3 = 25*E1
nu3=nu1
# Elasticity parameters 1 BULK-0
mu1, lda1 = Constant(E1/(2*(1 + nu1))), Constant(E1*nu1/((1 + nu1)*(1 - 2*nu1)))
# Elasticity parameters 3 FILM - 2
mu3, lda3 = Constant(E3/(2*(1 + nu3))), Constant(E3*nu3/((1 + nu3)*(1 - 2*nu3)))
d = u.geometric_dimension()
I = Identity(d)             # Identity tensor
F = I + grad(u)             # Deformation gradient
C = F.T * F              # Elastic right Cauchy-Green tensor
I = variable(tr(C))
Je  = variable(det(F))
```

```
psif = (mu3/2)*(I - 3) - mu3*ln(Je) + (lda3/2)*(ln(Je))**2
psis = (mu1/2)*(I - 3) - mu1*ln(Je) + (lda1/2)*(ln(Je))**2
dx = Measure('dx', domain=mesh, subdomain_data=materials)
dx = dx(degree=4)
ds = Measure("ds", subdomain_data=boundary_parts)
ds=ds(degree=4)
Pi=psif*dx(1)+psis*dx(0)
F=derivative(Pi,u,v)
J = derivative(F, u, du)
```

Once the energy Pi and the Jacobian are defined, we can pass some options to the compiler, which can also bet set at runtime.

```
parameters["form_compiler"]["cpp_optimize"]=True
parameters["form_compiler"]["representation"]='uflacs'
parameters['std_out_all_processes'] = False
```

Finally, the problem can be solved by incrementally deforming the boundary domains, namely, displacing the boundary surface at L_y by D_y which compresses the domain by $D_x = \frac{L_x D_y}{Ly - D_y}$

As we can see from the code below, we can store the solutions every certain number of iterations, so we do not end up with solutions written to disk for every displacement. The value of the increments can also be finely tuned depending on the requirements of every system. For instance we can deform the volume by a somewhat abrupt amount until we get close to the wrinkling threshold, where then we can decrease the steps of the increments in order to get convergent solutions at the onset.

```
file = File("displacement.pvd");
file << u;
d1=0.000
d2=0.000
ky=0.0
kx=0.0

DZ=-0.001
DX=0
DY=0
pzo=0.0
mp=1
for j in range(100):
    print(j)
    if j <5:
        DY=0.05
        mp=1
    if j>=5 and j<10:
        DY=0.01
        mp=5
    if j>=10:
        DX=1e-3
        mp=10

    DX=(Lx*DY)/(Ly-DY)
    d2+=(-1.0*DY)
    d1+=(1.0*DX)
    ky=d2/Ly
    kxb=d1/Lx
    cr.x0=d1
    cr.ky=ky
    cl.ky=ky
    cb.kx=kxb
    cb.ky=ky
    cn.y0=d2
    cn.kx=kxb
    cs.kx=kxb
    bcsa =[bcr,bcl,bcb,bcn,bcs]
    solve(F == 0, u , bcsa, J=J, solver_parameters={
                "newton_solver":{"maximum_iterations":100,
                "relative_tolerance": 1.0e-14,
                "absolute_tolerance": 1.0e-6,
                "linear_solver": "mumps"}})
    if j%mp==0:
        file << u;
```

For the case above the variational problem is solved directly by computing the stationary solutions of the energy functional first variation as outlined in: https://fenicsproject.org/olddocs/dolfin/latest/python/demos/hyperelasticity/demo_hyperelasticity.py.html

Now we turn our attention to the final example, which is the growth of the film to trigger the wrinkles, instead of compression.

4 Planar and volumetric growth of Neo-Hookean bilayers.

The addition of growth to a mechanics framework is still a work in progress, however in recent decades the framework of Morpho-elasticity has provided insights into how materials which grow induce stress which then translates into displacements. The theory relies on what is called the morphoelastic decomposition of the deformation gradient.

$$\mathbb{F} = \mathbb{A}\mathbb{G},$$

Where $\mathbb{A}$ and $\mathbb{G}$ represent an elastic deformation part and a growth contribution. On the other hand a system for which a stress tensor $\mathbb{P}$ can be defined obeys an equation of the form:

$$\nabla \cdot \mathbb{P} = \mathbf{0}$$

For its stationary configuration. If we provide the growth tensor, then we can solve the elastic problem posed as the elastic reaction to such growth. To solve bilayer systems using the morphoelastic decomposition, we can re-use much of the code of the compression case by modifying the kinematics.

The weak form of the variational problem is:

$$\int_{\Omega=\Omega_f\cup\Omega_s} \mathbb{P} : \nabla \mathbf{v} dV = \int_{\Omega_f} \mathbb{P}_f : \nabla \mathbf{v} dV + \int_{\Omega_s} \mathbb{P}_s : \nabla \mathbf{v} dV = \mathbf{0} \tag{1}$$

Where $\mathbb{P}$ is obtained using the strain energy densities:

$$\Psi_j = \frac{\mu_j}{2}\left(I_{1,j} - 3\right) - \mu_j \ln J_{e,j} + \frac{\lambda_j}{2} \ln^2 J_{e,j} \tag{2}$$

And $\mu_j = \frac{E_j}{2(1+\nu)}$ and $\lambda_j = \frac{E_j \nu}{(1+\nu)(1-2\nu)}$ are the Lamé material parameters expressed in terms of E_j and ν. Which for materials with the same ν satisfy: $\frac{\lambda_f}{\lambda_s} = \frac{\mu_f}{\mu_s} = \frac{E_f}{E_s} = R$.

The elastic deformation gradients $\mathbb{A}_j = \mathbb{F}\mathbb{G}_j^{-1}$ and the Cauchy-Green tensors $\mathbb{C}_j = \mathbb{A}_j^T \mathbb{A}_j$ allow us to compute, $I_{1,j}$ and $J_{e,j}$ and $J_{g,j}$, the first and third principal invariants of $\mathbb{C}_j$, $\mathbb{A}_j$ and $\mathbb{G}_j$ respectively. Alongside the second Piola-Kirchoff tensors:

$$\mathcal{S}_j = 2\frac{d\Psi_j}{dI_{e,j}}\mathbb{I} + J_{e,j}^2 \frac{d\Psi_{e,j}}{dJ_{e,j}} \mathbb{C}_{e,j}^{-1} \tag{3}$$

and:

$$\mathbb{S}_j = J_{g,j}\mathbb{G}_j^{-1}\mathcal{S}_j\mathbb{G}_j \tag{4}$$

Which gives the desired stress tensors:

$$\mathbb{P}_j = \mathbb{F}\mathbb{S}_j \tag{5}$$

First we define the trial and displacement fields, alongside the material mechanical parameters.

```
##########################################
# Define functions
du = TrialFunction(V)            # Incremental displacement
v  = TestFunction(V)             # Test function
u  = Function(V)                 # Displacement from previous iteration
B  = Constant((0.0, 0.0,0.0))    # Body force per unit volume
###############
# Elasticity parameters
E1, nu1 = 1, 0.48
E2 = 20*E1
```

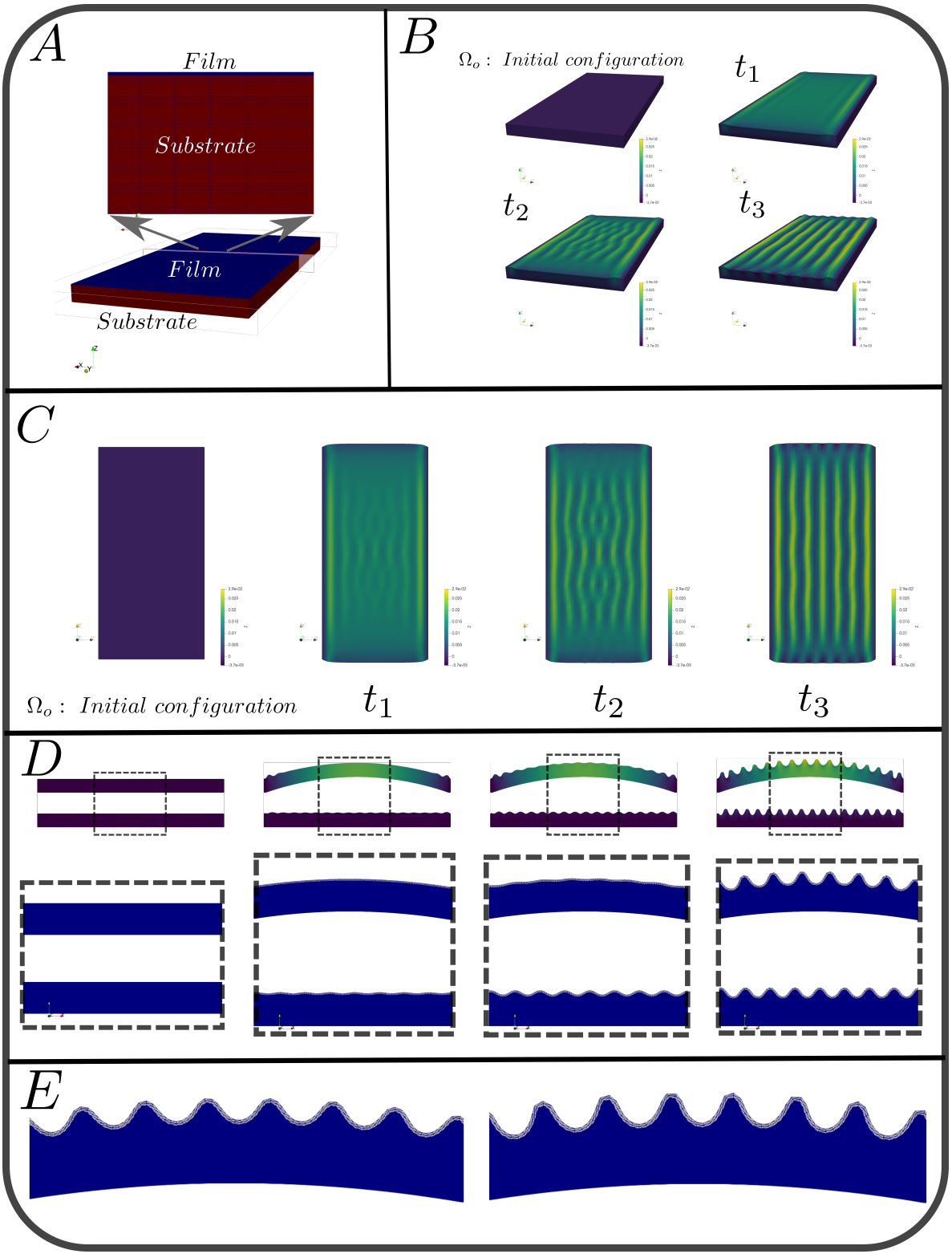

Figure 5: Growth induced instabilities in two and three dimensions. (A)-(C) Show the volumetric case and (D)-(E) the grow of a film in two dimensions using two distinct bottom boundary condition.

```
nu2=nu1
##########################################
# Elasticity parameters
mu1, lda1 = Constant(E1/(2*(1 + nu1))), Constant(E1*nu1/((1 + nu1)*(1 - 2*nu1)))
mu2, lda2 = Constant(E2/(2*(1 + nu2))), Constant(E2*nu2/((1 + nu2)*(1 - 2*nu2)))
##########################################
d = u.geometric_dimension()
I = Identity(d)             # Identity tensor
```

The most important part of the morpho-elastic formalism is the definition of the growth tensors. These can be defined as expressions which then can be updated iteratively thus defining them as sources of stress which are internal degrees of freedom. These can be defined in the following way.

```
#Growth Film
dgnx=1.0
dgny=1.0
dgnz=1.0
Fgf = Expression( (("dgnx",0.0,0.0),(0.0,"dgny",0.0),(0.0,0.0,"dgnz")), dgnx=dgnx,
    dgny=dgny, dgnz=dgnz, degree=1)
Fgfinv = inv(Fgf)
Jgf = det(Fgf)
#Growth Sub
Fgs = Expression ((("dgn", 0.0, 0.0), (0.0, "dgn2", 0.0), (0.0,0.0,"dgn3")),dgn=dgnx,
    dgn2=dgny,dgn3=dgny,degree=1)
Fgsinv = inv(Fgs)
Jgs = det(Fgs)
```

Once the growth tensors are defined the kinematic variables can be defined as follows.

```
# Kinematics
F = I + grad(u)             # Deformation gradient
#FILM
Fef = F * Fgfinv              # Elastic deformation gradient
Cef = Fef.T * Fef              # Elastic right Cauchy-Green tensor
# Invariants of deformation tensors
Icef = variable(tr(Cef))
Jef  = variable(det(Fef))
# Stored strain energy density (compressible neo-Hookean model)
psif = (mu2/2)*(Icef - 3) - mu2*ln(Jef) + (lda2/2)*(ln(Jef))**2
# Elastic second Piola-Kirchhoff stress tensor
Sef = 2*diff(psif, Icef)*I + Jef*Jef*diff(psif, Jef)*inv(Cef)
# Total second Piola-Kirchhoff stress
Sf = Jgf*Fgfinv * Sef * Fgfinv
# First Piola-Kirchhoff stress tensor
Pf = F*Sf
#SUBSTRATE
Fes = F * Fgsinv              # Elastic deformation gradient
Ces = Fes.T * Fes              # Elastic right Cauchy-Green tensor
# Invariants of deformation tensors
Ices = variable(tr(Ces))
Jes  = variable(det(Fes))
# Stored strain energy density (compressible neo-Hookean model)
psis = (mu1/2)*(Ices - 3) - mu1*ln(Jes) + (lda1/2)*(ln(Jes))**2
# Elastic second Piola-Kirchhoff stress tensor
Ses = 2*diff(psis, Ices)*I + Jes*Jes*diff(psis, Jes)*inv(Ces)
# Total second Piola-Kirchhoff stress
Ss = Jgs*Fgsinv * Ses * Fgsinv
# First Piola-Kirchhoff stress tensor
Ps = F*Ss
```

Once all the tensors, growth, elastic deformation gradients and stress are defined then the variational form can be written as:

```
dx = Measure('dx', domain=mesh, subdomain_data=materials)
dx = dx(degree=4)
ds = Measure("ds", subdomain_data=boundary_parts)
ds=ds(degree=4)
F = inner(Ps, grad(v))*dx(0) + inner(Pf, grad(v))*dx(1)-dot(Pz,v)*ds(3)
# Compute Jacobian of F
J = derivative(F, u, du)
```

Armed with all the pertinent definitions of parameters, tensors, and weak variational form, then we can iteratively solve the problem by updating the growth tensor using an internal parameter t. The

next chunk of code shows an example of anisotropic growth used to obtain the solutions in Fig. 5(A). The same code can be used to solve the system in a planar configuration as shown in the figure panel (D). For the planar case, we show results of anisotropic growth in the film's two principal directions, using different boundary conditions for the bottom of the volume. Namely a flat case and a bottom boundary condition obeying a parabolic profile, which is also updated at the same time as the growth tensor.

```
1
2  file = File("displacement.pvd");
3  file << u;
4  mu=0
5  d1=0.000
6  d2=0.000
7  ky=0.0
8  kx=0.0
9  DZ=-0.001
10 DX=0
11 DY=0
12 pzo=0.0
13 mp=1
14 muk=0.0
15 dt=1e-3
16 t=0.0
17 T=0.25
18 dtp=0.01
19 tp=0.01
20 d1=0.0
21 d2=0.0
22 gfx=1.0
23 gfy=1.0
24 gsx=1.0
25 gsy=1.0
26 gsz=1.0
27 dtp=0.01
28 muk=0.0
29 while t<T:
30     if t>0.1500 and t<0.24:
31         dt=1e-4
32     if t>=0.24:
33         dt=1e-7
34         dtp=1e-4
35     t+=dt
36     d1+=0.0*dt
37     d2+=0.0*dt
38     gfx+=1.5*dt
39     gfy+=0.0*dt
40     if t<0.1:
41         gsz+=0.5*dt
42     if t<0.2:
43         muk+=0.0*dt
44     Fgf.dgnx = gfx
45     Fgf.dgny = gsz
46     Fgs.dgn = gsz
47     Fgs.dgn2 = gsz
48     Fgs.dgn3 = gsz
49     ky=d2/Ly
50     kxb=d1/Lx
51     cr.x0=d1
52     cr.ky=ky
53     cl.ky=ky
54     cb.kx=kxb
55     cb.ky=ky
56     cn.y0=d2
57     cn.kx=kxb
58     cs.kx=kxb
59     cb.R=muk
60
61     bcsa =[bcr,bcl,bcb]
62     solve(F == 0, u , bcsa, J=J, solver_parameters={"newton_solver":{"
       maximum_iterations":100,
```

```
                                                    "relative_tolerance": 1.0e-14,
                                                    "absolute_tolerance": 1.0e-6,"
    linear_solver": "mumps"}})
    if t>tp:
        file << u;
        tp+=dtp
```

In the code above the entries of the growth tensors are updated accordingly and as a function of the parameter t, notice that all six entries can be subjected to dynamics operating on a faster time-scale if required. A set of source code and further documentation is available at https://github.com/calugo/wrinkles alongside some animations obtained using several different scenarios.

References

[ABH+15] Martin Alnæs, Jan Blechta, Johan Hake, August Johansson, Benjamin Kehlet, Anders Logg, Chris Richardson, Johannes Ring, Marie E Rognes, and Garth N Wells. The fenics project version 1.5. *Archive of Numerical Software*, Vol 3, 2015.

[Ama18] Marco Amabili. *Nonlinear Mechanics of Shells and Plates in Composite, Soft and Biological Materials*. Cambridge University Press, oct 2018.

[FHS18] Eszter Fehér, Timothy J. Healey, and András A. Sipos. The mullins effect in the wrinkling behavior of highly stretched thin films. *Journal of the Mechanics and Physics of Solids*, 119:417–427, oct 2018.

[RAS19] Miguel A. Rodriguez, Christoph M. Augustin, and Shawn C. Shadden. FEniCS mechanics: A package for continuum mechanics simulations. *SoftwareX*, 9:107–111, jan 2019.