# Contact Mechanics Problem: Hyperelastic Circle Constrained by Box

Ida Ang (Edited July 16, 2019)

## 1 Problem Definition

This is a document on a contact mechanics problem (currently undocumented demonstration) using the Scalable Nonlinear Equations Solvers (SNES) and Portable, Extensible Toolkit for Scientific Computation (PETSc)'s Toolkit for Advance Optimization (TAO) solvers coded by Corrado Maurini and updated by Tianyi Li (2014).

This example considers a hyperelastic circle under body forces in a box of the same size. Contact occurs as the circle drops in the box and is in contact with the bottom (x = 0) and sides of the cube (y = 0, y =1).
This problem is very similar to documented demonstration #7 hyperelasticity which uses potential energy minimization, an alternative approach to solving static problems

### 1.1 Potential Energy Minimization

Minimization of energy,

$$\min_{u \in V} \Pi$$

where V is a suitable function space that satisfies the boundary conditions on u.

The total potential energy is given by the sum of the internal and external energy:

$$\begin{aligned} \Pi &= \Pi_{int} + \Pi_{ext} \\ &= \left( \int_\Omega \psi(u) dx \right) + \left( - \int_\Omega B \cdot u dx - \int_{\partial\Omega} T \cdot u ds \right) \end{aligned} \tag{1}$$

where $\psi$ is the elastic stored energy density, B is body force (per unit reference volume) and T is a traction force (per unit reference area).

In this problem, there is no traction giving the following potential energy:

$$\Pi = \int_\Omega \psi(u) dx - \int_\Omega B \cdot u dx \tag{2}$$

Minimization of the potential energy corresponds to the directional derivative of $\Pi$ being zero for all possible variations of u. (Note, minimizing $\Pi$ is equivalent to solving the balance of momentum problem.)

$$L(u; v) = D_v \Pi = \frac{d\Pi(u + \epsilon v)}{d\epsilon}\Big|_{\epsilon=0} = 0 \qquad \forall v \in V \tag{3}$$

If we use Newton's method, we also want to find the Jacobian of Eq. 3

$$a(u; du, v) = D_{du}L = \frac{dL(u + \epsilon du; v)}{d\epsilon}\Big|_{\epsilon=0} \tag{4}$$

Note: in the final `solve` function in FEniCS we equate Eq. 4 to J = det(F)

# 2 FEniCS Implementation

Import module

```
from dolfin import *
```

Define SNES solver parameters. `maximum_iterations` sets the maximum newton-raphson iterations the solver will try before exiting. `report` allows for a report of the functional for each iteration as well as a report if the solver fails. `error_on_noncovergence` set to False suppresses the default error message which includes fenics contact information.

```
snes_solver_parameters = {"nonlinear_solver": "snes",
                          "snes_solver": {"linear_solver": "lu",
                                          "maximum_iterations": 20,
                                          "report": True,
                                          "error_on_nonconvergence": False}}
```

The domain is a unit square in the xy plane enclosing a unit circle, where x is directed to the right and y upwards.

$$\Omega = [0, 1] \times [0, 1]$$

Import and mesh the pre-made circle in the xy plane. NOTE that the origin of the xy-plane is in the center of the circle

```
mesh = Mesh("circle_xyplane.xml")
```

Setting a vector function space of order 1 works for the xy-plane defined, because each vector will be (x, y)

```
V = VectorFunctionSpace(mesh, "Lagrange", 1)
```

Define the incremental displacement as the trial function, v as the test function, and u is the displacement unknown

```
du = TrialFunction(V)
v = TestFunction(V)
u = Function(V)
```

**Kinematics**

The kinematics of the problem involve defining the deformation gradient and right Cauchy-Green tensor. First define the spatial dimension of the problem (d) and the identity tensor

```
d = len(u)
I = Identity(d)
```

Use the identity tensor to define the deformation gradient and right Cauchy-Green tensor

$$F = I + \nabla u$$
$$C = F^T F$$

```
F = I + grad(u)
C = F.T*F
```

Define the first and third invariants using F and C

$$I_c = \mathrm{tr}\, C \quad J = \det F$$

```
Ic = tr(C)
J = det(F)
```

**Constants**
Define the body force such that the circle experiences body forces in the downwards direction.
```
B = Constant((0.0, -0.1))
```

Define the elasticity parameters (Note that `lambda` is a keyword in FEniCS so we define `lmbda`).
```
E, nu = 10.0, 0.3
mu = Constant(E/(2*(1 + nu)))
lmbda = Constant(E*nu/((1 + nu)*(1 - 2*nu)))
```

**Strain energy density**

$$\Psi = \frac{\mu}{2}(I_c - 2 - 2\ln J) + \frac{\lambda}{2}(\ln J)^2$$

```
psi = (mu/2)*(Ic - 2 - 2*ln(J)) + (lmbda/2)*(ln(J))**2
```

Minimize the potential energy of the problem according to Eq. 2

$$\Pi = \int_\Omega \psi(u) dx - \int_\Omega B \cdot u dx$$

```
Pi = psi*dx - dot(B, u)*dx
```

Compute the first variation of the potential energy, Π by taking the directional derivative about u in the direction of v according to Eq. 3.
```
F = derivative(Pi, u, v)
```

Compute the Jacobian, by taking the directional derivative about u in the direction of du (incremental displacement) according to Eq. 4
```
J = derivative(F, u, du)
```

**Symmetry and Boundary Conditions**
The purpose of this symmetry condition is to block rigid body rotations. This sets x = 0
```
def symmetry_line(x):
```

```
    return abs(x[0]) < DOLFIN_EPS
```

By subclassing the function space, V, we can constrain displacement in the x direction. The method `pointwise` ensures the displacement constraint is set on the nodes.

```
bc = DirichletBC(V.sub(0), 0.0, symmetry_line, method="pointwise")
```

Recall how displacement is defined

$$u = X(\text{Current configuration}) - x(\text{Reference Configuration})$$
$$x + u = X \quad \text{Must remain within the box}$$

The current configuration x+u must remain within the box [xmin,xmax] x [ymin,ymax], This constraint can be done through the following expressions:

$$\text{con}_l = (-1.0 - x, -1.0 - y)$$
$$\text{con}_u = (1.0 - x, 1.0 - y)$$

`constraint_l` identifies the left side and bottom of the square. `constraint_u` identifies the right side and top of the square

```
constraint_l = Expression(("xmin - x[0]","ymin - x[1]"),
                          xmin=-1.0-DOLFIN_EPS, ymin=-1.0, degree=1)
constraint_u = Expression(("xmax - x[0]", "ymax - x[1]"),
                          xmax=1.0+DOLFIN_EPS, ymax=1.0, degree=1)
```

Interpolate these expressions into the function space

```
umin = interpolate(constraint_l, V)
umax = interpolate(constraint_u, V)
```

## 2.1 Solver Strategy

**SNES Solver**

Note that it appears that the PETSc's TAO solver appears to be the best optimization strategy, because higher body forces can be used for larger deformations.

```
problem = NonlinearVariationalProblem(F, u, bc, J=J)
```

Set the boundaries that ensures the circle stays within the box.

```
problem.set_bounds(umin, umax)
```

Call the solver parameters set above in the update statement.

```
solver = NonlinearVariationalSolver(problem)
solver.parameters.update(snes_solver_parameters)
```

If `info` is set to True, information on all solver parameters will be printed

```
info(solver.parameters, True)
```

Stops at this line when the solution diverges

```
(iter, converged) = solver.solve()
```

Warning because modification of the body force results in divergence of the problem. (in the y direction to > -1.0)

```
if not converged:
   warning("This demo is a complex nonlinear problem.  Convergence is not
guaranteed when modifying some parameters or using PETSC 3.2.")
```

Save to an .xdmf file for multiple fields (time step = 0).

```
file = XDMFFile("displacement_snes.xdmf")
file.write(u, 0.)
```

**TAO solver**

This file had been edited by Tianyi Li in 2014 and has some edits not only to the solver but to the boundary constraints:

Instead of returning an expression for which x[0] must satisfy, the function `near()` returns where x = 0. Functionally, these are equivalent versions of the same definition `symmetry_line`

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

Enforce boundary condition similarly without extra `"method=pointwise"`

```
bc = DirichletBC(V.sub(0), Constant(0.0), symmetry_line)
```

Removes usage of machine precision value `DOLFIN_EPS`

```
constraint_l = Expression(("xmin-x[0]", "ymin-x[1]"),
                          xmin=-1.0, ymin=-1.0, degree=1)
constraint_u = Expression(("xmax-x[0]", "ymax-x[1]"),
                          xmax=1.0, ymax=1.0, degree=1)
```

Interpolation into function space remains the same:

```
u_min = interpolate(constraint_l, V)
u_max = interpolate(constraint_u, V)
```

Constraining the hyperelastic circle with upper and lower limits requires the `apply()` function. Boundary conditions are applied differently in the TAO solver.

```
bc.apply(u_min.vector())
bc.apply(u_max.vector())
```

*Class definition*

Define the minimization problem by using the OptimisationProblem class

```
class ContactProblem(OptimisationProblem):
     def __init__(self):
          OptimisationProblem.__init__(self)
```

Objective function

```
     def f(self, x):
          u.vector()[:]  = x
          return assemble(Pi)
```

Define deformation gradient of the objective function

```
def F(self, b, x):
     u.vector()[:]  = x
     assemble(F, tensor=b)
```

Hessian of the objective function.

```
def J(self, A, x):
     u.vector()[:]  = x
     assemble(J, tensor=A)
```

The Hessian is related to the Jacobian by the following equation:

$$\mathbf{H}(f(x)) = \mathbf{J}(\nabla f(x))^T$$

Solve the problem by specifying the boundary conditions with u_min and u_max

```
solver.solve(ContactProblem(), u.vector(), u_min.vector(), u_max.vector())
```