Using FEniCS to Determine Spring Rate of Rubber Bellows

Quinn Thorsnes

June 30, 2017

1 Abstract

The purpose of this document is to examine the efficacy of using the python package FEniCS for finite element analysis to determine the spring rate of rubber bellows. The code used for this investigation will be included in an appendix for the reader's convenience. This report will also include a brief outline of the steps taken to produce a functional library.

2 Introduction

When selecting a vibration isolation system it is important to understand the properties of its constituent parts. The properties of this parts depend on both the material properties and the geometry of the materials. The focus here is, as previously mentioned, on the properties of the different candidate rubber bellows. Manufacturers will often list the material and some dimensions they believe to be relevant to incorporation of their part, but there are other properties that they do not think to list, such as spring constant. This is the precise property that we are interested in and, fortunately for us, this is an area that has been thoroughly explored by material scientists.

Determination of spring-rate is easy when confronted with a beam that is an extrusion of some base two-dimentional shape. We can use the Young's modulus, E, of the material along with its length, L, and cross sectional area, A, to find the spring rate of extension and compression for uniform pressures applied to the surface.

$$k = \frac{EA}{L}$$

This formula assumes that the only displacement felt by the material is along the axis of compression/extension. There are other formulae that we would need to use for different situations such as bending. With knowledge of the behaviour of multi-spring systems more complicated geometries can be approximated using these equations.

As the complexity of the shapes grows the certainty in the reliability of these models decreases. It is at this point that we become interested computer models. We specifically become interested in the technique known as finite element analysis (FEA), also known as the finite element method (FEM). Previously the go to program for this was Comsol multiphysics but Comsol is expensive to maintain licensure for, especially when usage will be niche. Thus we look for a free or open source alternative. The alternative I chose to investigate was the FEniCS project as it looked to be cross-platform, have an active community, be under active development, and to be well documented.

3 Installation

The FEniCS website contains instructions for four methods of installation. These methods are using a Docker package (similar to having a virtual machine),

through Anaconda, a native Ubuntu distribution, or the user may build from source. It is important to note that FEniCS is only native to Unix systems. Detailed installation instructions may be found on the FEniCS website [2], [1].

3.1 Docker

Docker is available for the three major platforms and thus allows for FEniCS to be cross-platform regardless of it having been designed for use with Unix systems. This was only partially tested for Windows 10 Home and not tested for other operating systems.

3.2 Anaconda

Installation via Anaconda is available through for all three major operating systems. The direct installation method is only available on iOS and Linux. It is accomplished using `conda create -n fenicsproject -c conda-forge fenics` followed by `source activate fenicsproject` in the terminal after Anaconda has been installed. The first command adds the FEniCS project to anaconda, the second opens the FEniCS environment. As of our testing this second command was required for FEniCS to be a recognized package.

To use Anaconda for installation on Windows one must first have installed Microsoft Azure. Following this one must open a new Jupyter notebook and type: `!conda config --add channels conda-forge` on the first line and `!conda install fenics` on the next. This method has not been tested, but is, instead, copied from the official installation instructions so it is assumed to function similarly to the previous Anaconda.

3.3 Ubuntu

To install FEniCS using Ubuntu's in built package manager one must add the FEniCS Project to their update path by executing the following into their terminal `sudo add-apt-repository ppa:fenics-packages/fenics` followed by `sudo apt-get update` to update the list of available packages and updates. Now the actual installation is available via `sudo apt-get install --no-install-recommends fenics` with `--no-install recommends` being optional. Finally, it may be necessary to execute `sudo apt-get dist-upgrade` to truly update all packages. This was required to gain support for Python 3.x. Our testing was done with a version only compatible with Python 2.7.x. The compatibility with 3.x has given it a great advantage to its previous form as the syntax was already for usage with python 3.x and this may allow for easy access to Jupyter in addition to the packages compatible with pythons both 2.7.x and 3.x.

3.4 Our Choice

Although installing through Anaconda may seem the most convenient, it was determined that using the Ubuntu distribution involved the least amount of additional effort before it became functional. It is important to note that installation and testing was carried out on Ubuntu version 16.04.

The reasoning for this is that installing FEniCS as an Anaconda package resulted in the statement `import fenics` raises a `module not found` error. The initial workaround for this is to specify that you wish to start your Anaconda python distribution in FEniCS mode. This is, however, met with an error almost immediately upon running any program since DOLFIN is then found to fail to load. Specifying desire to work in a FEniCS environment from the start also precludes the usage of the Jupyter notebook meaning that the advantages to using Anaconda are now near negligible.

The preferred method is therefore installing via the dedicated Ubuntu distribution. The only disadvantage to this is that, as of the time of writing, the Ubtuntu distribution is only compatible with Python 2.7.x. This is a disadvantage because active support of Python 2.7.x has been discontinued in Jupyter, but there does exist a long term support version of Jupyter that is compatible. Most other python packages may be installed with relative ease through either Python's pip or Ubuntu's apt-get. This will be required for all packages that are not inherent to the core distribution of python.

4 Implementation

Before proceeding further I would like to remind the reader that code referenced in this section has been made available in full in the appendix. The below information may not be sufficient as a complete tutorial. FEniCS provides their own tutorial in the form of a 153 page PDF on their website.

4.1 Setting up the problem

FEniCS works by applying boundary conditions and user specified functions to meshes that exist within a function space given some material properties of said meshes. Thus these are the first aspects that need definition. The meshes may be defined using the in-built package mshr or through use of an external mesh modelling program. As out geometry was not exceptionally complicated, and in the interest of minimizing external programs, we used the in built mshr. Boundary conditions are more easily implemented. They are generally given as a function space to be applied to, a location in that space, and a boundary to enforce.

Our first step is to define the material properties. We are specifically interested in the spring constant of chloroprene (a.k.a. neoprene) bellows so we must give the elastic properties of chloroprene. FEniCS does not do its calculations with Young's Modulus and shear modulus, but with Lamé's parameters λ and μ, called Lamé's first and second parameters respectively. μ is simply the shear

modulus, G, while λ is not so conveniently renamed. To calculate λ we need to make a conversion using:

$$\lambda = \frac{G(E - 2G)}{3G - E}$$

Next we generated the mesh. Mshr allows for meshes to be generated from basic solids such as ellipsoids, cylinders, and rectangular prisms. The user may then create more complicated meshes by combining basic meshes as one would numbers. To make a hollow cylinder, for example, one would subtract a cylinder from another cylinder of the same height, but greater diameter. To approximate our bellows we made use of cylinders and ellipsoids as both can be made rotationally symmetrical with ease. The cylinders are the main shaft of the bellows, as far as elastic portion is concerned, then we use ellipsoids to make the bulges. We used a cylinder with height and outer diameter of our object, added to this a number of ellipsoids equal to the number of bulges we want our bellows to have. We then subtract a cylinder of equal height to the first and diameter equal to the inner diameter of our bellows followed by ellipsoids smaller than the first set of ellipsoids in each dimension by the thickness of the bellows material. The dimensions of the ellipsoids are determined by the geometry of the bulge(s) present in the bellows. It is important for the short axis of these ellipsoids to lie along the central axis of the cylinder and for the smaller ellipsoids to be concentric with the larger ones else uniform bulges will not be observed. This approximation is more valid for bellows with more bulges as the transitions tend to be more abrupt when many bulges are present.

With our mesh defined we now need to define a function space for it as well as the boundary conditions that constrain it. Definition of the function space is achieved relatively easily with `V = VectorFunctionSpace(mesh, family, degree)` with mesh being our mesh, family being a string that determines the type of function and degree an integer that defines the degree. We want a vector function space as we want to deal with vectors. Definition of a boundary condition is similarly easy with `bc = DirichletBC(function_space, location, condition, method (optional))`. Just like function spaces there are multiple types of boundary condition and we have chosen a Dirichlet boundary condition for ours. For boundary conditions we declare them with the vector function space within our mesh, the location is a place that we know mesh will interact with - the origin, and the condition we specified was that all points in our mesh must be above y = 0 as we use y as our up direction.

Both the force per unit volume within the mesh and the traction force per unit area outside of the mesh can be given as scalars, vectors, or even matrices. Regardless of this they need to be defined as type `Constant` so that it is known to be invariant in This is specified by changing the argument of the constructor to be a scalar, vector, or matrix. e.g. `Constant(7)` for a scalar valued field or `Constant((-4,0,0))` for a field with value -4 in the x direction. Because the traction is given as force per unit area we use the total weight force we want to apply and divide through by the area that it will interact with. This works well for objects that do not vary in width, but may require adjustment when

used with bulged objects as the bulges may be subjected to the traction force as well.

Next we define functions for stress and strain. $\epsilon = 0.5(\bigtriangledown u + (\bigtriangledown u)^\top)$ is the strain tensor and $\sigma = \lambda(\bigtriangledown \cdot u)I + 2G\epsilon$ is the stress tensor. In both cases u is our trial function and is a member of the our vector function space V. Both of these come from the variational formulation of the elastic deformation problem described on page 52 of the FEniCS tutorial volume 1 [3].

4.2 Testing

After constructing our geometry we applied boundary conditions as seen in the cantilevered beam example of the tutorial. This resulted in results dissimilar to those that we desired. There was no apparent deformation of our bellows, for one. This produced the need for returning to basic principles for the sake of determination of errors. We thus used two test cases where the theoretical behaviour was well known. Our two test cases are those of a cube and a hollow cylinder. Still the deformation and stress patterns did not make sense. This was further reinforced by the fact that the situation did not change as the applied force changed. It was discovered that the term labelled force was, in fact, a force per unit volume term only evaluated within the mesh. Hence the load was not being simulated. The correction for this was determined to be the termed that, in the tutorial, was labelled traction. Further investigation revealed that traction represents the force per unit area field external to the mesh.

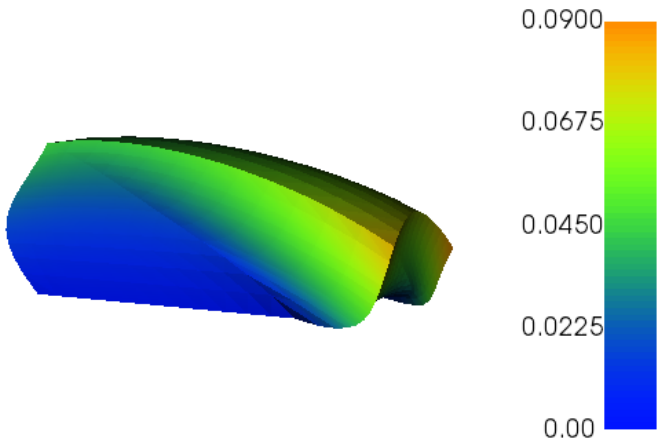

Figure 1: Uniform elastic cube deformed under uniform pressure. Colouring is heat map corresponding to displacement with scale given on the right in units of meters.

The above image shows the results of a uniform cube deformed under a uniformly distributed external pressure force. The heat map corresponds to displacement magnitude of the point in meters. This illustrates a potential issue present in the default mesh generator, mshr. The cube has been compressed with a uniform pressure to the top surface, yet one corner clearly deforms the least and another clearly deforms the most. The reasoning is that the cube is comprised of square segments crossed to become triangles, presumably to prevent torsion forces from being too weak. The issue comes in the distribution of the cross patterning. It was generated such that opposite faces were translations of each other and thus the stiffness became location dependent.

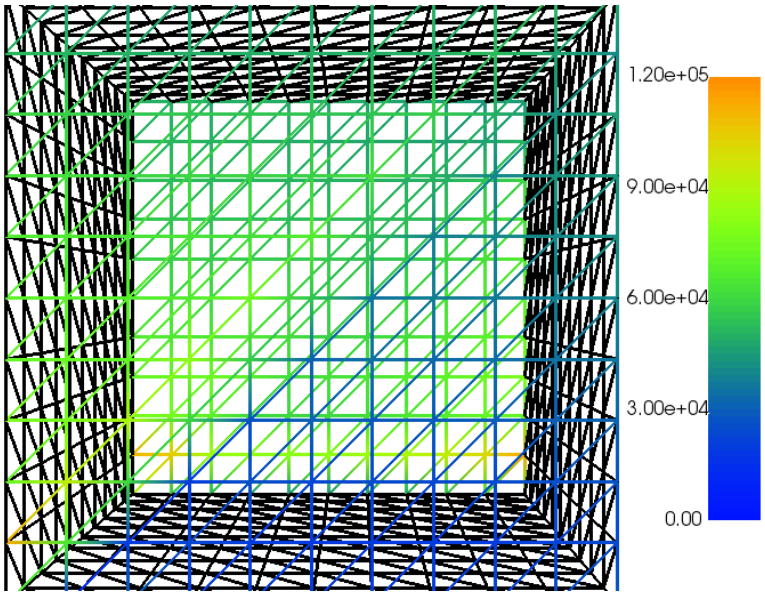

Figure 2: The same cube from earlier undeformed with heat map corresponding to stress and set to wireframe display. This enables the reader to see that opposite faces are translations of each other.

Figure 2 illustrates what has been described above. It is no longer difficult to see why the response to a uniformly distributed force is asymmetric. The alternative where the faces are rotated relative to the centre-point, such that spirals could be traced around the cube using the cross-beams, would likely result in an even more strange behaviour: twisting in response to compression. We do not have a specific solution for this problem other than using another geometry, although it looks likely that using another program to generate the mesh may well solve the problem.

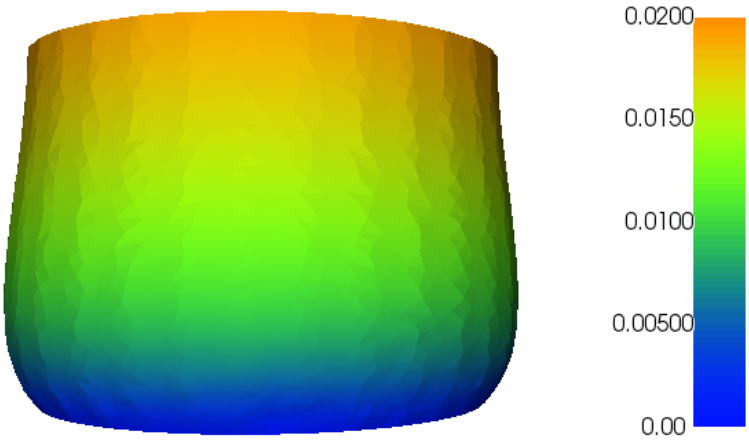

Figure 3: Deformation of a hollow cylinder under extreme pressure conditions. As before the colour coding is a heat-map describing the magnitude of displacement for a given point.

The cylindrical test mesh, for example, did not appear to have this issue. The deformation for this scenario can be seen in Figure 3. When rotated the deformation for this case is visibly rotationally symmetric. This pleased us as it was what we were expecting to be the case. Symmetric loads on symmetric geometries should lead to symmetric responses. Further, the deformation appears to be mostly linear from top to bottom of the cylinder, also as expected. The plot of stress is not quite as pleasing, but is generally in agreement with predictions. Said stress plot is shown in Figure 4.

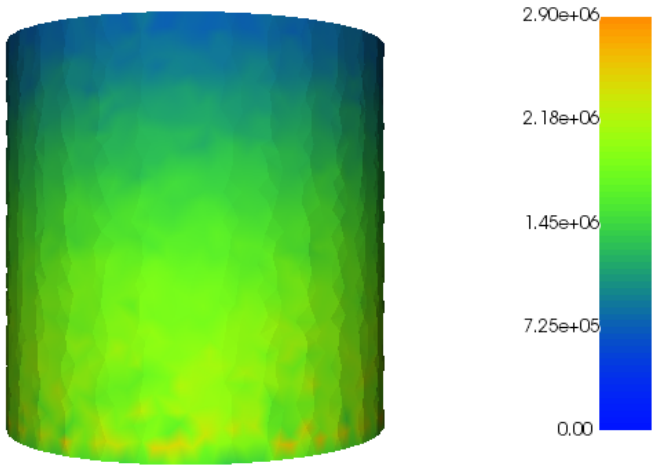

Figure 4: Stress pattern for the same hollow cylinder under the same extreme pressure conditions.

The point of confusion found with the hollow cylinder (and cube as well) was that the determined spring rate was different than the expected rate by a factor of roughly 0.24. We considered that this might be due to a defect in the code where integer division was used on accident, but this did not yield significant improvements. It is also possible that the resolution of our meshes was insufficient to produce the desired results. A third option is that there is some aspect or assumption of the theory that is incorrect in our situation and thus FEniCS, using different theoretical principles, is more correct. This seems unlikely, but is also not impossible.

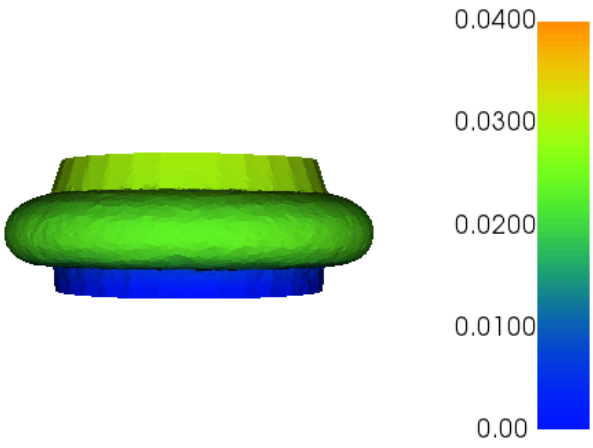

Figure 5: Our bellows under a load. Note that the outer flange is tending downwards. This raised suspicions about if our model was being applied correctly.

Given that our objective was obtaining a rough estimate of the spring rate of the bellows and that the error seemed to be consistent we thought it possible that we could run the simulation for the bellows set up then apply a correction factor. The deformation results may be seen in Figure 5. The fact that, when compressed, the bulge is noticeably downward facing, instead of being in a vertical mirror symmetry, caused us to wonder if the external pressure was being applied to the bulge as well as the rim of the bellows. We did not determine a testing method for whether or not this was the case. Reading of the documentation suggests that it would be possible to define another object, i.e. a load object, on top of the bellows with another mesh and define boundary conditions such that the two objects could not pass through each other. This method would require more work in initial definition of the problem, and thus may require longer computation time, but may also produce a more physically realistic simulation.

5 Conclusion

The result of this investigation is that the FEniCS project could well be a valuable tool for physical simulation, but likely requires additional investigation before use beyond qualitative analysis.

6 Appendix

6.1 Potential Alternatives

The items listed below have not been investigated and, as such, the available information is mostly limited to that found on their respective websites.

FEATool Multiphysics toolbox for Matlab and Octave

- Free for personal use
- Supports general multi-physics couplings with predefined physics for heat-transfer, fluid flow, structural mechanics, chemical reactions, and more.

Elmer

- Open-source multi-physical software developed mainly by CSC-IT Center for Science (CSC)
- Started in 1995 in collaboration with Finish universities, research institutes, and industry
- became international in 2005
- Includes physical models described by PDEs, solved by finite element method (FEM)
  - fluid dynamics
  - structural mechanics
  - electromagnetics
  - heat transfer
  - acoustics
  - more

Kratos

- Open source framework for implementation of numerical methods for solution of engineering problems
- written in C++, designed to allow multiple collaborators
- core and applications approach: basic math is core to build applications

FlexPDE

- claims to be the original unlimited scripted multi-physics finite element solution environment for practical differential equations
  - heat flow
  - stress analysis
  - fluid mechanics

  - chemical reactions
  - electromagnetics
  - diffusion
  - etc.
- from mathematical model to numerical simulation to graphical display
- student edition offers one, two, and three-dimensional analysis for free
- 1795USD for the Academic Pro version with three-dimensional support

Open FOAM

- Free, open source, CFD software released and developed primarily by Open CFD Ltd. since 2004.
- boasts extensive range of features to solve anything from complex fluid flow involving chemical reactions, turbulence, and heat transfer to acoustics, solid mechanics, and electromagnetics

CAELinux

- Open Source
- Linux only
- seems to be a stand-alone package for use with CAD

PyDSTool

- self-described as a sophisticated and integrated simulation and analysis environment for dynamic systems, models of physical systems (ODEs, DAEs, maps, and hybrid systems)
- Platform independent, built mostly on python with some underlying Fortran and legacy code for soft-solving
- extensive use of numpy and scipy
- supports:
  - symbolic math
  - optimisation
  - phase plane analysis
  - continuation and bifurcation analysis
  - data analysis
  - other tools for modelling - particularly for biological applications
- fully open source with a BSD licence
  - Must aknowledge "Clewley RH, Sherwood WE, LaMor MD, Guckeheimer JM (2007), *PyDSTool, a software environment for dynamical systems modelling*, URL http://pydstool.sourceforge.net"

- documentation under development
  - permission explicitly given to email the developers

6.2 Code

The entirety of the code used is listed below. It is worth noting that this code is for use with python 2.7.x as it was the only version compatible with the Ubuntu distribution at the time. Python 3.x is now compatible so statements such as `from __future__ import print_function` may no longer be needed.

```
1  from __future__ import print_function
2
3  from fenics import *
4  from mshr import *
5  import math
6  import numpy
7
8  # Parameters
9  eL = 0.08 # length of a side of our cube in meters
10 E_CR = 0.00614*10**9 # Young's modulus of Chloroprene Rubber (CR) in Pa
11 G_CR = 0.00205*10**9 # Shear modulus of CR in Pa, also Lame's second
       parameter
12 lambda_CR = G_CR*(E_CR - 2*G_CR)/(3*G_CR - E_CR) # Lame's first
       parameter in Pa
13 F_0 = 130 # Total External force in N
14 ID = eL/2
15
16 cyl_top_SA = math.pi*((eL/2.0)**2 - (ID/2.0)**2)
17 cube_top_SA = eL**2
18 # Define meshes
19
20 # real mesh
21 h_large = 130*10**-3 # height of large bellows in m
22 ID_large = 200*10**-3 # inner diameter of large bellows in m
23 thickness_large = 10*10**-3 # thickness of walls of large bellows
24 OD_large = ID_large + thickness_large
25 bulge_large = 25.676*10**-3 # Additional radius due to the bulge
26 r_at_bulge = OD_large/2.0 + bulge_large # works for both x and z because
       they are indistinct
27 top_area = math.pi*((OD_large/2.0)**2 - (ID_large/2.0)**2)
28
29 large_cyl_out = Cylinder(Point(0, 0, 0),Point(0, h_large, 0),
       OD_large/2.0, OD_large/2.0)
30 large_cyl_in = Cylinder(Point(0, 0, 0),Point(0, h_large, 0),
       ID_large/2.0, ID_large/2.0)
31 large_ellip_out = Ellipsoid(Point(0, h_large/2.0, 0), r_at_bulge,
       h_large/2.0, r_at_bulge)
32 large_ellip_in = Ellipsoid(Point(0, h_large/2.0, 0), r_at_bulge -
       thickness_large, h_large/2.0 - thickness_large, r_at_bulge -
```

```
    thickness_large)
domain = large_cyl_out + large_ellip_out - large_cyl_in - large_ellip_in

#the_mesh = generate_mesh(domain, 64)

# Simple test meshes
the_mesh = BoxMesh(Point(0, 0, 0), Point(eL, eL, eL), 10, 10, 10)
#the_mesh = generate_mesh(Cylinder(Point(0,0,0), Point(0,eL,0), eL/2.0,
    eL/2.0) - Cylinder(Point(0,0,0), Point(0,eL,0), ID/2.0, ID/2.0), 45)

# Define function spaces
V1 = VectorFunctionSpace(the_mesh, 'P', 1)

# define boundary condition
tol = 1E-14

#def clamped_boundary_floor(x, on_boundary):
    #return on_boundary and x[1] < tol # y is up/down, x[1] is y => y =
        0 is the floor

bc1 = DirichletBC(V1, Constant((0, 0, 0)), "near(x[1], 0)", method =
    "geometric")

# Stress and strain

def epsilon_CR(u):
    return 0.5*(nabla_grad(u) + nabla_grad(u).T)

def sigma_CR(u):
    return lambda_CR*nabla_div(u)*Identity(d) + 2*G_CR*epsilon_CR(u)

# Define variational problem
u = TrialFunction(V1)
d = u.geometric_dimension() # space dimension
v = TestFunction(V1)
f = Constant((0, 0, 0))
T = Constant((0, -F_0/cube_top_SA, 0)) # Comment out all but the desired
    T
#T = Constant((0, -F_0/cyl_top_SA, 0))
#T = Constant((0, -F_0/top_area, 0))
a = inner(sigma_CR(u), epsilon_CR(v))*dx
L = dot(f, v)*dx + dot(T, v)*ds

# Compute solution
u = Function(V1)
solve(a == L, u, bc1)

# Plot solution
plot(u, title='Displacement', mode='displacement') # good for
    visualising, but I want hard numbers out to compare to theory
```

```
77
78  # Plot stress
79  s = sigma_CR(u) - (1.0/3)*tr(sigma_CR(u))*Identity(d) # deviatoric stress
80  von_Mises = sqrt(3.0/2*inner(s, s))
81  V = FunctionSpace(the_mesh, 'P', 1)
82  von_Mises = project(von_Mises, V)
83  plot(von_Mises, title='Stress intensity')
84
85  # displacement magnitudes
86  u_magnitude = sqrt(dot(u, u))
87  u_magnitude = project(u_magnitude, V)
88  plot(u_magnitude, title = 'Displacement')
89  print('max u/min u:', u_magnitude.vector().array().max(),
        u_magnitude.vector().array().min())
90
91  # Because this is an approximation we want to make sure that we are not
        basing our calculation on an abnormally large value
92  # sort the displacement magnitude array by magnitude.
93  sorted_mag = numpy.sort(u_magnitude.vector().array())
94  # now get the mean of the last 5%
95  mean_max_d = 0
96  i = len(sorted_mag)*0.05
97  i = int(i) # make i an int
98  num = i
99  i *= -1 # make i negative
100 while(i < 0): # add up the top 5% of entries
101     mean_max_d += sorted_mag[i]
102     i += 1
103
104 mean_max_d = mean_max_d/num # divide by the number of entries iterated
        over
105
106 k_imp_1 = F_0/mean_max_d
107
108 print("implied spring rate:", k_imp_1)
109
110 #print("what if the area is the problem?", k_imp_1*cyl_top_SA)
111
112 cube_expected_k = (E_CR*eL**2)/(eL)
113 hollow_cyl_expected_k = E_CR*cyl_top_SA/eL
114 bellows_cyl_expected_k = E_CR*top_area/h_large
115
116 #print("ratio of determined to expected (cylinder)",
        k_imp_1/hollow_cyl_expected_k)
117 print("ratio of determined to expected (cube)", k_imp_1/cube_expected_k)
118
119 print("for a cube of side length L =", eL, "m we expect:",
        cube_expected_k)
120 #print("for a hollow cylinder of height and outer diameter", h_large, "m
        and inner diameter", ID, "m we expect", hollow_cyl_expected_k)
```

```
121 #print("for a hollow cylinder of height", h_large, "m, outer diamter",
        OD_large, "m, and inner diameter", ID_large, "m we expect",
        bellows_cyl_expected_k)
122
123 interactive()
```

References

[1] Martin S. 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*, 3(100), 2015.

[2] FEniCS Project. Fenics.

[3] Hans Petter Langtangen and Anders Logg. *Solving PDEs in Minutes - The FEniCS Tutorial Volume I.* 2017.