Manual on cell_computation.py

April 1, 2016

1 Table of Contents

2 Overview

This is the main part of the module consisting of class `MicroComputation` and several assisting functions which cover setup of the solver parameters and relation between extended strain and displacement.

3 Example on 2D uni field unit cell computation

- **Module import**

```
In [1]: from dolfin import *

        import numpy as np

        import sys
        sys.path.append('../')

        import cell_geom as geom
        import cell_material as mat
        import cell_computation as comp
```

- **Choose linear backend**

The embedded backends can be viewed through `linear_algebra_backend()`, namely Eigen, PETSc (default), and STL. Notice that the backend should be specified at the beginning of all the calculation and initialization, as the corresponding matrix and vector objects are casted in a style appropriate for the chose backend.

```
In [2]: ## Linear Backend
        parameters['linear_algebra_backend'] = 'PETSc'
```

- **Define unit cell geometry (mesh, inclusions)**

The geometrical properties are set according to the Manual on cellgeom.py_. Inclusions such as circle and rectangular in 2D and 3D are available. Extensions are possible in this context. Mesh can be imported the same as in FEniCS.

```
In [3]: ## Define Geometry
        mesh = Mesh(r'../m_fine.xml')
        cell = geom.UnitCell(mesh)

        # Add inclusion
        inc = geom.InclusionCircle(2, (0.5, 0.5), 0.25)
        inc_di = {'circle_inc': inc}
        cell.set_append_inclusion(inc_di)
```

- **Define materials in composites**

The material properties are set according to the manual on cell_material.py. Three types of materials are provided in the material libraries. Users can also specify their own materials. The definition of a new material follows the steps, 1. free energy function `psi`, 2. invariant relations with physical variables, 3. initialize an instance of `Material` with `psi` and required parameters, 4. append invariant relations to the `Material` instance, 5. call this instance with the physical field variables and complete instantiation

Details and restrictions are referred in manual on cell_material.py.

```
In [4]: ## Define Material
        E_m, nu_m, E_i, nu_i = 10.0, 0.3, 1000.0, 0.3
        mat_m = mat.st_venant_kirchhoff(E_m, nu_m)
        mat_i = mat.st_venant_kirchhoff(E_i, nu_i)

        mat_li = [mat_m, mat_i]
```

- **Create MicroComputation instance (pre-processing)**

Here a general relation between the physical field variable and strain is given. This is realized in the function `deform_grad_with_macro`. `w` stands for the fluctuation that needs to be solved, while `F` is a general strain measure, which are considered in the material total energy. `F_bar` is the general strain from macro computation serving as input for `MicroComputation`.

The instantiation steps are as follows 1. Function spaces for physical field variables and generate Functions for field variables. These field variables are often regarded as general displacements, and in `MicroComputation` they are fluctuation to solve. 2. Define relation between general strains and general displacements. 3. Give function space for general strain, these are for the post processing. 4. Initialize a `MicroComputation` instance using geometry (`cell`), materials (`mat_li`), general strain-displacement relation (`deform_grad_with_macro`), and the strain function space for post-processing (`strain_space`). 5. Specify general strain from macro field and input them into instance of `MicroComputation`. It completes the creation a `MicroComputation`

```
In [5]: ## Define Computation
        # Step 1: Field variables
        VFS = VectorFunctionSpace(cell.mesh, "CG", 1,
                                  constrained_domain=geom.PeriodicBoundary_no_corner(2))
        w = Function(VFS)

        # Step 2: Strain and field variable relations
        def deform_grad_with_macro(F_bar, w_component):
            return F_bar + grad(w_component)

        # Step 3: Strain space for post processing
        strain_space = TensorFunctionSpace(mesh, 'DG', 0)

        # Step 4: Initialization
        compute = comp.MicroComputation(cell, mat_li,
```

```
                                    [deform_grad_with_macro],
                                    [strain_space])
        # Step 5: Complete instantiation
        F_bar = [0.9, 0., 0., 1.]
        compute.input([F_bar], [w])
```

- **Solve fluctuation**

It begins with setting of solver parameters. The solving step is just call the member function of this instance `comp_fluctuation()`

There are two classes of methods, direct (lu solver) and iterative (krylov solver). To list the solvers in direct methods, command `lu_solver_methods()`, while for listing iterative method solvers command is `krylov_solver_methods()`. For iterative solvers a preconditioner is needed, which can be viewed using command `krylov_solver_preconditioners()`. A complete summary of solvers can be referred in the website of PETSc.

```
In [7]: # Parameters for solving of fluctuation
        comp.set_solver_parameters('non_lin_newton', lin_method='direct', linear_solver='mumps')
```

```
.--------------------.
| Solver Parameters |
.--------------------.
direct method is used
```

```
In [8]: compute.comp_fluctuation(print_progress=True, print_solver_info=False)
```

```
fluctuation computation finished
```

- **Post Processing and view the results**

Calculation of homogenized quantity is the central part of homogenization method. Hence various post processing of the `MicroComputation` result are implemented in this class. These are `comp_strain()`, `comp_stress()` in terms of calculating general strain and stress in the unit cell, `avg_merge_strain()`, `avg_merge_stress()`, `avg_merge_moduli()` in terms of calculating the averaged quantities in a unit cell, and `effective_moduli_2()` for calculating the homogenized tangent moduli.

`effective_moduli_2()` is the most consuming part. Specifying a good solver for it can speed up this process. This is involved with using function `set_post_solver_parameters()`.

Visualizing the results are also realized in the current module. Wrapping the visualization tools in FEniCS is included. The member functions are `view_fluctuation()`, `view_displacement()`, and `view_post_processing()`. When multiple fields are considered, specifying the component of visualization is needed.

```
In [10]: compute.comp_strain()
         compute.comp_stress()
         compute.avg_merge_strain()
         compute.avg_merge_stress()
         compute.avg_merge_moduli()
```

```
strain computation finished
strain computation finished
stress computation finished
average merge strain computation finished
average merge stress computation finished
average merge moduli computation finished
```

```
Out[10]: array([[  2.68263384e+02,  -1.47306216e-02,   1.45226354e-02,
                   1.16307090e+02],
                [ -1.47306216e-02,   7.64588022e+01,   7.75364924e+01,
                  -1.46396985e-02],
                [  1.45226354e-02,   7.75364924e+01,   7.64302939e+01,
                   1.44729548e-02],
                [  1.16307090e+02,  -1.46396985e-02,   1.44729548e-02,
                   2.72444929e+02]])

In [12]: # Post processing solver parameters
         comp.set_post_solver_parameters(lin_method='iterative',)

         # Homogenized tangent moduli
         compute.effective_moduli_2()

+----------------------------+
| Post Processing Parameters |
+----------------------------+
iterative method is used
a valid preconditioner should be provided
average merge moduli computation finished

Out[12]: array([[  1.05215604e+01,   6.81100019e-04,   8.10922941e-04,
                   6.09319740e+00],
                [  6.81100019e-04,   3.03957695e+00,   4.12022593e+00,
                  -2.43801203e-04],
                [  8.10922941e-04,   4.12022593e+00,   2.98311033e+00,
                  -3.19622254e-04],
                [  6.09319740e+00,  -2.43801203e-04,  -3.19622254e-04,
                   1.74423266e+01]])

In [13]: # View results
         compute.view_fluctuation()
         compute.view_displacement()
         compute.view_post_processing('strain', (0,1))
         compute.view_post_processing('stress', (0,1))
```

- **Call help for each functions**

Note that in-code docstrings are given to explain functions arguments and outputs. Calling docstrings in IPython is simply add a question mark behind a function, and two question marks will show the detailed implementation the method. In Python context, `help()` is used to list the docstring.

```
In [19]: help(compute.comp_fluctuation)

Help on method comp_fluctuation in module cell_computation:

comp_fluctuation(self, print_progress=False, print_solver_info=False) method of cell_computation.MicroCom
    Solve fluctuation, solver parameters are set before solving

    :param print_progress: (bool) print detailed solving progress
    :param print_solver_info: (bool) print detailed solver info

    :return: updated self.w_merge
```

This example can work as a simulation template.

4 Concepts in implementation

The design concepts of this module are elaborated in this part. The main idea of the `MicroComputation` is to establish a unit cell. When the deformation or field variables from macro state are input, micro state computation can be realized. This computation will fall into several parts such as solving fluctuation, and in the end calculating effective tangent moduli.

If the geometry and materials of this unit cell does not change. This instance of `MicroComputation` can be reused for another computation. This will make `MicroComputation` act like a factory that produce the result of computation in the micro scale.

Various functions and member methods are targeted to achieve this computation.

- **Initializtion**

When an instance is initialized, several methods are called behind scene. Once `input()` is called, the instance is complete. Arguments for `input()` are `F_bar_li` and `w_li` representing the macro field input and to be solved fluctuation respectively. In `input()` `F_bar_li` is transformed to a list of Functions suitable for the later formulation done by `_F_bar_init()`. `set_field()` is called to merge and split fields for the variational formulaiton later on. Then general strain measures are created through `extend_strain()` also for the later formulation, where the input is the splitted FEniCS Functions.

- **Pre-processing**

Pre-processing is the problem formulation stage. `_total_energy()` is to insert general strains into materials to construct its dependency within the physical fields. The summation of all the free enerygies of every material components in composite make the total energy complete.

Then boundary condition is provided with `_bc_fixed_corner()`, which fix all the corners of the unit cell.

`_fem_formulation_composite` follows with derivation of the nonlinear problem using powerful functions defined in FEniCS, `derivative()`

- **Solving**

Solving step is accomplished by `comp_fluctuation()` with previously specified solver parameters.

- **Post-processing**

Post-processing plays the key role in homogenization problem. It starts with `comp_strain`, where convenient FEniCS function `project()` is used. Then material energy is updated with the calculated general strains in `_energy_update()`. It is for the purpose that, general stresses are conjugated part in the total energy. Formulating the total energy in general strain will lead to direct calculation of general stresses with the help of `derivative()`. It is done by `comp_stress()`.

`avg_merge_strain()`, `avg_merge_stress()`, and `avg_merge_moduli()` are implemented using the trick of multiplying trial and test functions defining on a constant Function Space. Detailed elaboration is given in the report.

`effective_moduli_2()` is based on `avg_merge_moduli()`, a LTKL term is subtracted from the averaged merged moduli. The calculation of this term is realized in `sensitivity()`. L matrix is assembled with boundary conditions. Some techniques are taken in imposing the boundary conditions. Using the FEniCS `solve()` on `K_a` and `L[:, i]` gives a intermediate result. Left multiplying with the transpose of L gives the required term.

- **Things to notice**

In the module only the relation between mechanical displacement and deformation gradient is given in the function `deform_grad_with_macro`, other kinds of relation between general strain and displacement should be specified by the user.

Another thing to notice is that a good solver needs to be chosen in complicated cases such as multi field or 3D. Direct solvers are rather slow in this circumstances, while iterative solvers will not always converge and requires a lot of try-outs with the right preconditioners.