FEniCS implementation of partial mechanics of far fields for nonlinear vibrations problems

Duc-Trung Le (Armines), David Ryckelynck (Mines ParisTech/MAT)

22 juin 2017

1 Problem set-up

1.1 Continuous problem

In this paper, we consider a large displacement, small strain elastodynamics problem. The continuous medium is occupying a domain $\Omega \in \mathbb{R}^d\,(d = 2,3)$. We assume that the reference configuration Ω_0 of the medium is a natural state. The density of the medium in the reference configuration is denoted by ρ, and for a sake of simplicity, we assume it to be constant. Mixed boundary conditions are considered on $\partial\Omega = \partial\Omega_D \cup \partial\Omega_N$. A homogeneous Dirichlet condition is imposed on $\partial\Omega_D$, and a non-homogeneous Neumann-type boundary condition is considered on $\partial\Omega_N$.
We denote $\mathbf{u}(\cdot, t)$ the displacement field at time t with respect to the reference configuration Ω_0. The elastodynamics problem of Saint Venant,ÄìKirchhoff material consist of finding $\mathbf{u} : \Omega \times (0,T) \rightarrow \mathbb{R}^d$ such that :

$$\begin{cases} \rho\ddot{\mathbf{u}} - \mathrm{div}\,\mathbf{P} & = \mathbf{b} \quad \text{in } \Omega \times (0,T) \\ \mathbf{P} & = (\mathbf{I} + \nabla\mathbf{u})\mathbf{S} \\ \mathbf{u} & = 0 \quad \text{on } \partial\Omega_D \times (0,T) \\ \mathbf{P}\,n & = g \quad \text{on } \partial\Omega_N \times (0,T) \\ \mathbf{u}(\cdot,0) = \dot{\mathbf{u}}(\cdot,0) & = 0 \quad \text{in } \Omega \end{cases} \tag{1}$$

Here, $\mathbf{I}$ denotes the identity matrix. $\mathbf{S}$ is the so-called second Piola-Kirchhoff stress tensor, which is calculated by the derivative of the strain energy density function $\mathscr{W}$ with respect to the Green strain tensor $\mathbf{E}$:

$$\mathbf{S} = \frac{\partial \mathscr{W}}{\partial \mathbf{E}} \quad , \quad \mathscr{W} = \frac{\lambda}{2}\left[\mathrm{tr}(\mathbf{E})\right]^2 + \mu\,\mathrm{tr}(\mathbf{E}^2) \quad , \quad \mathbf{E} = \frac{1}{2}\left[(\mathbf{I} + \nabla\mathbf{u})^T(\mathbf{I} + \nabla\mathbf{u}) - \mathbf{I}\right] \tag{2}$$

with λ and μ are the classical Lame coefficients.
Let $V = H^1(\Omega)^d$, $V_0 = \{\mathbf{v} \in V | \mathbf{v} = 0 \text{ on } \partial\Omega_D\}$, by using the principle of virtual work, the weak form of equation of motion consists of finding $\mathbf{u} \in V$ such that :

$$\int_\Omega \mathbf{w} \cdot \rho \ddot{\mathbf{u}}\, d\mathbf{x} + \int_\Omega \nabla \mathbf{w} \cdot \mathbf{P}\, d\mathbf{x} - \int_\Omega \mathbf{w} \cdot \mathbf{b}\, d\mathbf{x} - \int_{\partial\Omega_N} \mathbf{w} \cdot g\, d\mathbf{s} = 0 \qquad \forall \mathbf{w} \in V_0 \tag{3}$$

1.2 Time and space discretization

The time interval $(0, T)$ is divided in to N_t time steps with time increment Δt. By using the classical Newmark time integration algorithm [ref.], which depends on the parameters β and γ , the relations between displacements, velocities and accelerations at t^i and t^{i+1} read :

$$\mathbf{u}^{i+1} = \beta \Delta t^2 \ddot{\mathbf{u}}^{i+1} + \mathbf{u}_p^{i+1}$$
$$\dot{\mathbf{u}}^{i+1} = \gamma \Delta t \ddot{\mathbf{u}}^{i+1} + \dot{\mathbf{u}}_p^{i+1}$$

where $\mathbf{u}_p^{i+1}$ and $\dot{\mathbf{u}}_p^{i+1}$ respectively represent the predictors of the displacement and velocity, calculated from the results of previous step :

$$\mathbf{u}_p^{i+1} = \mathbf{u}^i + \Delta t\, \dot{\mathbf{u}}^i + \frac{1}{2}\Delta t^2 (1 - 2\beta) \ddot{\mathbf{u}}^i$$
$$\dot{\mathbf{u}}_p^{i+1} = \dot{\mathbf{u}}^i + \Delta t (1 - \gamma) \ddot{\mathbf{u}}^i$$

For $\beta > 0$, the motion equation (3) can be posed at time t^{i+1} with only $\mathbf{u}^{i+1}$ as unknown :

$$\int_\Omega \frac{\rho}{\beta \Delta t^2} \mathbf{w} \cdot (\mathbf{u}^{i+1} - \mathbf{u}_p^{i+1})\, d\mathbf{x} + \int_\Omega \nabla \mathbf{w} \cdot \mathbf{P}(\mathbf{u}^{i+1})\, d\mathbf{x} - \int_\Omega \mathbf{w} \cdot \mathbf{b}\, d\mathbf{x} - \int_{\partial\Omega_N} \mathbf{w} \cdot g\, d\mathbf{s} = 0 \tag{4}$$

for all the test functions $\mathbf{w} \in V_0$. Then the velocities and accelerations at t^{i+1} can be corrected with :

$$\ddot{\mathbf{u}}^{i+1} = \frac{1}{\beta \Delta t^2}(\mathbf{u}^{i+1} - \mathbf{u}_p^{i+1}) \quad , \quad \dot{\mathbf{u}}^{i+1} = \dot{\mathbf{u}}_p^{i+1} + \Delta t (1 - \gamma) \ddot{\mathbf{u}}^{i+1}$$

Let,Äôs denote by $\mathbf{q}^n$ the vector of nodal displacements, $\mathbf{q}_p^n$ the vector of the predictor of displacement at time step t_n, the FE discretization of (4) at time t_n leads to the following system of equations :

$$\mathbf{R}^n(\mathbf{q}^n, \mathbf{q}_p^n) = 0 \qquad , n = 1, 2, ..., N_t \tag{5}$$

with $\mathbf{R}^n$ the residual vector of the FE equilibrium equations, for a given degree of freedom (DOF), the residual is defined by :

$$R_i^n = \int_\Omega \frac{\rho}{\beta \Delta t^2} \varphi_i \cdot (\mathbf{q}^n - \mathbf{q}_p^n)\, d\mathbf{x} + \int_\Omega \nabla \varphi_i \cdot \mathbf{P}^n\, d\mathbf{x} - \int_\Omega \varphi_i \cdot \mathbf{b}\, d\mathbf{x} - \int_{\partial\Omega_N} \varphi_i \cdot g\, d\mathbf{s} \tag{6}$$

with φ_i is the corresponding shape function of the DOF. Although $\mathbf{R}^n$ depends on $\mathbf{q}_p^n$, it will not be mentioned in equations in the next sections for simplicity reason.

1.3 Partial mechanics of far fields

For a complete description of of hyper-reduction method and partial mechanics of far fields, reader can refer to

1.4 Numerical example

In this section, a 2D elastodynamics problem of Saint Venant-Kirchhoff material is considered to illustrate the capabilities of the method. A cantilever beam of 1000mm length, 100mm height and 100mm width is made of steel. We assume plane strain transformations. The Youngs modulus is 200 GPa, the Poisson coefficient is 0.3 and the density ρ is $8000\,kg/m^3$.
The goal of this section is to predict the non linear vibration response of the beam when a part of the top right edge is subjected to a vertical, oscillatory force of the form :

$$\begin{cases} g(t) = g_0 \dfrac{t}{T_0} & \forall\, 0 \le t \le T_0 \\ g(t) = g_0 - \Delta_g \sin(\omega\pi \dfrac{t}{T_0}) & \forall\, T_0 \le t \le T_1 \end{cases} \tag{7}$$

while the other edge remains free. The other geometry and loading coefficients are reported in Table 1. The time interval $[0, T_1]$ is split into $N_t = 400$ increments by a $0.0025\,s$ time increment.

Parameter	$T_0(s)$	$T_1(s)$	$g_0(GPa)$	Δ_g (Gpa)	ω
Value	0.25	1	0.01	0.002	4

TABLE 1 – Loading parameters

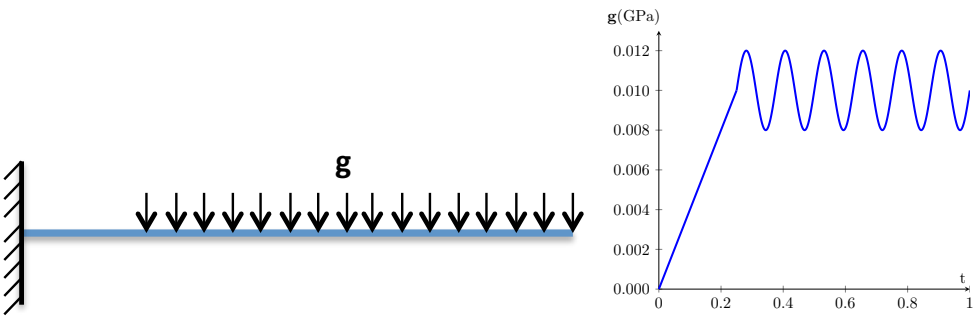

FIGURE 1 – The cantilever beam under cyclic loading

The region of interest (ROI) is defined at the left-end of the beam. The size of ROI is

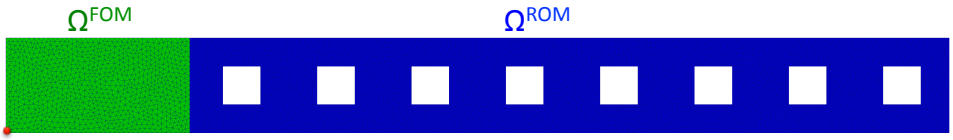

FIGURE 2 – The region of interest in green, the reduced model region in blue

controlled by its length (`lz`), as depicted in figure 2
In order to verify the capability of PMFF, we introduce a circular inclusion inside Ω_F . We now consider the Young modulus of inclusion (E_I) as a parameter and we want to predict the response of the beam when E_I varies from $100\,GPa$ to $200\,GPa$.

FIGURE 3 – Inclusion inside the ROI

2 Finite element solution with FEniCS

In order to use PMFF, we need some FE solutions to build the snapshot matrix and the reduced basis. This section presents the implementation in FEniCS of the problem introduced in previous section. The entire program is saved in `NonlinearDynamicsClass.py`, it contains 2 classes : `Top` and `nonlinear_dynamics`

2.1 `Top` class

This class is derived from `SubDomain` class of FEniCS, it defines the part on the top of the beam which is subjected to the loading.

```
class Top(SubDomain):
  def set_length(self,self_h,self_l,self_lf):
    self.h = self_h
    self.l = self_l
    self.lf = self_lf
  def inside(self, x, on_boundary):
    return (near(x[1], self.h) and between(x[0], (self.lf, self.l)))
```

2.2 `nonlinear_dynamics` class

This class contains the main program to solver the nonlinear dynamic problem presented in section 1.

- **Instantiation of class** :

  `nonlinear_dynamics(meshInput, K_input,f_input)`

  Arguments :
  `meshInput` : the actual mesh of the problem.
  `K_input` : the Young modulus of inclusion.
  `f_input` : loading applied to the part defined by `Top` class.

```
class nonlinear_dynamics(object):
  def __init__(self,meshInput, K_input,f_input):
    # save dir
    self.set_savedir(K_input)
    # Mesh
    self.mesh = meshInput
    # Parameter
    self.set_parameter(K_input)
    #Create function spaces
    self.V = VectorFunctionSpace(self.mesh, 'CG', 1)
    self.VS = TensorFunctionSpace(self.mesh, 'CG', 1)
    self.Vq1 = FunctionSpace(self.mesh,  'CG', 1)
    self.V0 =  FunctionSpace(self.mesh, 'DG', 0)
    #Solution, test and trial functions
    self.du, self.du_t = TrialFunction(self.V), TestFunction(self.V)
    self.uk = Function(self.V)
    self.u_pred,self.v_pred=Function(self.V),Function(self.V)
    self.u_n,self.v_n,self.a_n = Function(self.V),Function(self.V),Function(self.V)
    self.u,self.v,self.a = Function(self.V),Function(self.V),Function(self.V)
    self.dqu= Function(self.V)
    self.d = self.uk.geometric_dimension()
    self.I = Identity(self.d)
    # MeshFunctions and Measures for loading and subdomain
    self.set_mesh_functions()
    self.set_measures()
    # File to stock data
    self.create_data_file(K_input)
    #Loading
    self.set_loading(f_input)

    # Dirichlet BC
    self.bc_u = self.define_bc_u()
    self.K_function()
```

- **`set_savedir(K_re)` method** :
  Arguments :
  `K_re` : the Young modulus of inclusion.
  As its name, this function defines the location to store the FE solutions.

```
def set_savedir(self,K_re):
  self.savedir = "results_EF/K_%01d" %K_re
  if os.path.isdir(self.savedir):
    shutil.rmtree(self.savedir)
```

```
  self.file_offline = File(self.savedir+"/offline/u_offline.pvd")
  self.file_stress = File(self.savedir+"/stress/stress_xx.pvd")
  self.file_mesh=File(self.savedir+"/mesh0.pvd")
```

- **`set_parameter(Kre_input)` method** :
  Arguments :
  `Kre_input` : the Young modulus of inclusion.
  This function defines the constant parameters of problem

```
def set_parameter(self,Kre_input):
  #beam height
  self.h = 0.1
  self.l = 1.
  # loading length
  self.lf=0.2
  # length of interest zone
  self.lz=0.2
  #newmark parameter
  self.theta=0.5
  self.gamma = 0.5
  self.beta = 0.25
  #time step
  self.dt=0.0025
  self.t_max1 = 1.
  # parameters of steel
  self.rho=8000./1000000000.
  self.K, self.nu = 200.0, 0.3
  self.K_re = Kre_input
  self.mu, self.lmbda = Constant(self.K/(2*(1 + self.nu))), Constant(self.K*self.nu/((1
                                                    + self.nu)*(1 - 2*self.nu)))
  self.mu_re, self.lmbda_re = Constant(self.K_re/(2*(1 + self.nu))), Constant(self.K_re
                                                  *self.nu/((1 + self.nu)*(1 - 2*self.nu))
                                                  )
```

- **`create_data_file(Kre_input)` method** :
  Arguments :
  `Kre_input` : the Young modulus of inclusion.
  This function defines the location to save the FE solutions in the form of numpy array

```
def create_data_file(self,Kre_input):
  time1 = pl.arange(0,self.t_max1, self.dt)
  self.Q_dof = np.zeros(shape=(self.u.vector().size(),len(time1)))
  #self.Qstress_tensor = np.zeros(shape=(self.u.vector().size()*2,len(time1)))
  self.Qtress_dof = np.zeros(shape=(self.u.vector().size()/2,len(time1)))
  self.Qtress1_dof = np.zeros(shape=(self.u.vector().size()/2,len(time1)))
  self.Qtress01_dof = np.zeros(shape=(self.u.vector().size()/2,len(time1)))
  self.Qtress10_dof = np.zeros(shape=(self.u.vector().size()/2,len(time1)))
  cwd = os.getcwd()
  directory = "Offline_Data"
  if not os.path.exists(directory):
    os.makedirs(directory)
  self.Q_dof_filepath = cwd+"/"+directory+"/Q_dof_%01d" %Kre_input
  self.Qtress_dof_filepath = cwd+"/"+directory+"/Qstress_%01d" %Kre_input
  self.Qtress1_dof_filepath = cwd+"/"+directory+"/Qstress1_%01d" %Kre_input
  self.Qtress01_dof_filepath = cwd+"/"+directory+"/Qstress01_%01d" %Kre_input
  self.Qtress10_dof_filepath = cwd+"/"+directory+"/Qstress10_%01d" %Kre_input
  print self.Q_dof_filepath
```

- **`set_mesh_functions()` method**:
  This function defines the region in the top part of the beam where the loading is applied and creates a mesh function to designate the inclusion.

```
def set_mesh_functions(self):
  # Initialize sub-domain instances
  self.top = Top()
  self.top.set_length(self.h,self.l,self.lf)
  # Initialize mesh function for loading
  self.boundaries = FacetFunction("size_t", self.mesh)
  self.boundaries.set_all(0)
  self.top.mark(self.boundaries, 2)
  # Load mesh fuction from file for inclusion marker
  self.Ydomains = MeshFunction('size_t', self.mesh, 'mesh_physical_region.xml')
```

- **`set_measures()` method**:
  This is a FEniCS technique to integrate over the sub-domains defined by mesh function.

```
  def set_measures(self):
    self.ds = Measure('ds', domain=self.mesh, subdomain_data=self.boundaries)
    self.dx = Measure('dx', domain=self.mesh, subdomain_data=self.Ydomains)
```

- **`define_bc_u()` method**:
  This function constructs the Dirichlet boundary condition for displacement field `u`

```
  def define_bc_u(self):
    left =  CompiledSubDomain("near(x[0], side) && on_boundary", side = 0.0)
    c0 = Expression(("0.0", "0.0"))
    bc_u0 = DirichletBC(self.V, c0, left)
    bcs = bc_u0
    return bcs
```

- **`K_function()` method**:
  This function defines the elasticity coefficients as the piecewise function over full domain.

```
  def K_function(self):
    self.mu_f = Function(self.V0)
    self.lmbda_f = Function(self.V0)
    self.mu_values = [self.K/(2*(1 + self.nu)), self.K_re/(2*(1 + self.nu))]  # values
                                                  of k in the two subdomains
    self.lmbda_values = [self.K*self.nu/((1 + self.nu)*(1 - 2*self.nu)), self.K_re*self
                                                  .nu/((1 + self.nu)*(1 - 2*self.nu))]
    for cell_no in range(len(self.Ydomains.array())):
      subdomain_no = self.Ydomains.array()[cell_no]
      self.mu_f.vector()[cell_no] = self.mu_values[subdomain_no]
      self.lmbda_f.vector()[cell_no] = self.lmbda_values[subdomain_no]
```

- **`Form(u_f,u_test,u_prediction,force)` method**:
  Arguments:
  `u_f`: the solution of last time-step.
  `u_test`: the test function.
  `u_prediction`: the prediction of `u` at current time-step.

`force` : the loading.
This function return the residual defined in (6)

```
def Form(self,u_f,u_test,u_prediction,force):
  du_f=nabla_grad(u_f)
  du_te=nabla_grad(u_test)
  P_int=(self.I+du_f)*(self.lmbda*tr(0.5*(du_f+du_f.T)+du_f.T*du_f)*self.I+2*self.mu*
                                          (0.5*(du_f+du_f.T)+du_f.T*du_f))
  P_re=(self.I+du_f)*(self.lmbda_re*tr(0.5*(du_f+du_f.T)+du_f.T*du_f)*self.I+2*self.
                                          mu_re*(0.5*(du_f+du_f.T)+du_f.T*du_f))
  F=(self.rho/(self.beta*self.dt*self.dt))*inner(u_f-u_prediction,u_test)*self.dx+(
                                          inner(P_int,du_te)*self.dx(0)+inner(P_re
                                          ,du_te)*self.dx(1)-inner(force,u_test)*
                                          self.ds(2))
  return F;
```

- **`Jacobian(u_f,u_trial,u_test)` method** :
  Arguments :
  `u_f` : the solution of last time-step.
  `u_test` : the test function.
  `u_trial` : the trial function.
  This function return the Jacobian the `Form`, used in Newton Raphson iteration .

```
def Jacobian(self,u_f,u_trial,u_test):
  du_f=nabla_grad(u_f)
  du_tr=nabla_grad(u_trial)
  du_te=nabla_grad(u_test)
  dF0_int=(self.I+du_f)*(self.lmbda*tr(0.5*(du_tr+du_tr.T)+du_tr.T*du_f+du_f.T*du_tr)
                                          *self.I+2*self.mu*(0.5*(du_tr+du_tr.T)+
                                          du_tr.T*du_f+du_f.T*du_tr))
  dF1_int=du_tr*(self.lmbda*tr(0.5*(du_f+du_f.T)+du_f.T*du_f)*self.I+2*self.mu*(0.5*(
                                          du_f+du_f.T)+du_f.T*du_f))
  dF0_re=(self.I+du_f)*(self.lmbda_re*tr(0.5*(du_tr+du_tr.T)+du_tr.T*du_f+du_f.T*
                                          du_tr)*self.I+2*self.mu_re*(0.5*(du_tr+
                                          du_tr.T)+du_tr.T*du_f+du_f.T*du_tr))
  dF1_re=du_tr*(self.lmbda_re*tr(0.5*(du_f+du_f.T)+du_f.T*du_f)*self.I+2*self.mu_re*(
                                          0.5*(du_f+du_f.T)+du_f.T*du_f))
  J=(self.rho/(self.beta*self.dt*self.dt))*inner(u_trial,u_test)*self.dx+(inner(
                                          dF0_int+dF1_int,du_te)*self.dx(0)+inner(
                                          dF0_re+dF1_re,du_te)*self.dx(1))
  return J;
```

- **`Pstress(u_f)` method** :
  Arguments :
  `u_f` : the solution of displacement field.
  This function return the first Piola-Kirchhoff stress.

```
def Pstress(self,u_f):
  du_f=nabla_grad(u_f)
  P_stress=(self.I+du_f)*(self.lmbda_f*tr(0.5*(du_f+du_f.T)+du_f.T*du_f)*self.I+2*
                                          self.mu_f*(0.5*(du_f+du_f.T)+du_f.T*du_f
                                          ))
  return P_stress;
```

- **`variational_formulation_solveur()` method** :
  This is the main part of this program, it assembles the Jacobian and Form matrix,

solves the system of equation and saves the solution on numpy array format

```
def variational_formulation_solveur(self):
    j=0
    M_0=assemble(self.rho*inner(self.du, self.du_t)*self.dx)
    F_0=assemble(-inner(self.Pstress(self.u),nabla_grad(self.du_t))*self.dx)
    def solve_a():
      solve(M_0, self.a_n.vector(), F_0)

    time1 = pl.arange(0,self.t_max1, self.dt)
    for i, t in enumerate(time1):
      self.T.t=t
      tol = 1.0E-5
      iter = 0
      maxiter = 250
      eps = 1.0
      if i==0:
        solve_a()
      else:
        # Predictions for u^n and v^n
        self.u_pred.vector().set_local(self.u_n.vector().array() + self.dt*self.v_n.
                                       vector().array() + 0.5*self.dt**2*(1-2*
                                       self.beta)*self.a_n.vector().array())
        self.v_pred.vector().set_local(self.v_n.vector().array() + self.dt*(1-self.
                                       gamma)*self.a_n.vector().array())
        self.uk.vector().set_local(self.u_n.vector().array()) # use u_n for predicted
                                       solution at n+1 step
        # Solve for displacement at n+1 step
        while eps > tol and iter < maxiter:
          iter += 1
          A=assemble(self.Jacobian(self.uk,self.du,self.du_t))
          b=-assemble(self.Form(self.uk,self.du_t,self.u_pred,self.T))
          self.bc_u.apply(A,b)
          A1_sp=petsc_csr(A)
          b1_sp=b.array()
          q_du=splina.spsolve(A1_sp, b1_sp)
          eps = np.linalg.norm(q_du, ord=np.Inf)
          print 'Norm:', eps
          self.u.vector().set_local(self.uk.vector().array()+q_du)
          self.uk.vector().set_local(self.u.vector().array())

        #plot(self.uk, key = "alpha",mode = "displacement", title = "Damage at loading
                                       %.4f",interactive=False)
        #update displacement, vlocity and acceleration at n+1 step
        self.a_n.vector().set_local((self.u.vector().array()-self.u_pred.vector().array
                                       ()) / (self.beta*self.dt**2))
        self.v_n.vector().set_local(self.v_pred.vector().array() + self.dt*self.gamma*
                                       self.a_n.vector().array())
        self.u_n.vector().set_local(self.u.vector().array())
        #stock solution to numpy narray
        stress=self.Pstress(self.u)
        stress_tensor=project(stress,self.VS)
        stress00=project(stress[0,0],self.Vq1)
        stress11=project(stress[1,1],self.Vq1)
        stress01=project(stress[0,1],self.Vq1)
        stress10=project(stress[1,0],self.Vq1)
        self.Qtress_dof[:,j]=stress00.vector().array()
        self.Qtress1_dof[:,j]=stress11.vector().array()
        self.Qtress01_dof[:,j]=stress01.vector().array()
        self.Qtress10_dof[:,j]=stress10.vector().array()
        #self.Qstress_tensor[:,j]=stress_tensor.vector().array()
        self.Q_dof[:,j] =  self.u.vector().array()
```

```
        j=j+1
        self.u.rename("u_EF_K%01d" %self.K_re , "stress component")
        self.file_offline << self.u
      print t
    np.save(self.Q_dof_filepath, self.Q_dof)
    np.save(self.Qtress_dof_filepath, self.Qtress_dof)
    np.save(self.Qtress1_dof_filepath, self.Qtress1_dof)
    np.save(self.Qtress01_dof_filepath, self.Qtress01_dof)
    np.save(self.Qtress10_dof_filepath, self.Qtress10_dof)
```

3 PMFF implementation

This section presents the implementation in FEniCS of presented model. Python interface is chosen due to the compactness of the code. The program is divided into 5 parts, including 4 classes :

- `data_creator` : this class handles the construction of snapshot matrix and the decomposition of this matrix.
- `pod` : this class constructs the hybrid reduced basis of PMFF method.
- `rid` : this class builds the reduced integration domain from the reduced basis.
- `PMFF` : this class handles the computation of the PMFF solution.

and a collection of useful functions in `tools`.

3.1 The functions of `tools`

- `petsc_csr(A)`
  Arguments :
  `A` : PETSC matrix
  This function convert a PETSC matrix to a CSR matrix

```
def petsc_csr(A):
  A_mat = as_backend_type(A).mat()
  A_sparray = sp.csr_matrix(A_mat.getValuesCSR()[::-1], shape = A_mat.size)
  return A_sparray
```

- `deim(V)`
  Arguments :
  `V` : numpy array
  This function applies DEIM to matrix V to get the relevant components

```
def deim(V):
    r = V[:,0]
    i = sci.argmax(abs(r))
    F=[i]
    P = sci.zeros(sci.shape(V))
    P[i,0] = 1.
    for k in sci.arange(1,sci.shape(V)[1],1):
        gamma =    lina.solve( V[F,:k] ,    V[F,k] )
        r = V[:,k] - V[:,:k].dot(gamma);
```

```
        i = sci.argmax(abs(r))
        F = sci.append(F,i)
        P[i,k] = 1
    return P,F
```

- `iz_dof_struc(meshiz,lz):`
  Arguments :
  `meshiz` : FENICS mesh
  `lz` : the length of region of interest.
  This function creates the set of DoFs located inside region of interest and another set of the remaining DoFs.

```
def iz_dof_struc(meshiz,lz):
  class IZ(SubDomain):
    def inside(self, x, on_boundary):
      return True  if (x[0]<=lz)  else False

  VF = VectorFunctionSpace(meshiz, 'CG', 1)
  duf, du_tf = TrialFunction(VF), TestFunction(VF)
  subdomainsiz = CellFunction('size_t', meshiz, 0)
  subdomainsiz.set_all(0)
  izone = IZ()
  izone.mark(subdomainsiz, 1)
  dx_f =  Measure('dx', domain=meshiz, subdomain_data=subdomainsiz)
  A_test1= assemble(inner(duf, du_tf)*dx_f(1))
  Amatrx_sp = petsc_csr(A_test1)
  Amatrx1 = Amatrx_sp.diagonal()

  dofinside=[]
  dofoutside=[]
  for i in range (np.shape(Amatrx1)[0]):
    if Amatrx1[i] != 0 :
      dofinside.append(i)
  for i in range (np.shape(Amatrx1)[0]):
    if Amatrx1[i] == 0 :
      dofoutside.append(i)

  return dofinside,dofoutside,subdomainsiz
```

3.2 `data_creator` class

This class specifies the locations of folders to load and save data, and it contains the methods to manipulate the snapshot matrix.

- **Instantiation of class** :

  `data_creator(meshInput, lzInput, ssListInput)`

  Arguments :
  `meshInput` : the actual mesh of the problem.
  `lzInput` : the length of region of interest.
  `ssListInput` (python list) : the set of snapshot points used to build reduced basis.

```
  def __init__(self,meshInput, lzInput,ssListInput):
    self.POD_LIST = ssListInput
    #number of snapshot used
    self.NUMBER_OF_POD = len(self.POD_LIST)
    self.cwd = os.getcwd()
    self.podDirectory = "POD_Data"
    self.ssDirectory = "Offline_Data"
    self.ridDirectory = "RID_Data"
    #  The geometrical dimension.
    self.gdim = meshInput.geometry().dim()
    #  The topological dimension.
    self.tdim = meshInput.topology().dim()
    self.rpodDirectory = "RPOD_Data"
    if not os.path.exists(self.podDirectory):
      os.makedirs(self.podDirectory)
    if not os.path.exists(self.ridDirectory):
      os.makedirs(self.ridDirectory)
    if not os.path.exists(self.rpodDirectory):
      os.makedirs(self.rpodDirectory)
    #load snapshot matrix
    self.load_data()
    #create set of DoFs of reduced domain
    self.create_reduced_dof_list(meshInput,lzInput)
    #number of DoFs
    self.Ndof = len(self.IZ_dofinside)+len(self.IZ_dofoutside)
```

- **`load_data()` method** :
  This function loads the separated FE solutions of displacement and stress and combines them in a big matrix named snapshot matrix.

```
def load_data(self):
  self.Q_dof_full = np.load(self.cwd+"/"+self.ssDirectory+"/Q_dof_%01d.npy" %self.
                                                  POD_LIST[0])
  self.S_dof_full = np.load(self.cwd+"/"+self.ssDirectory+"/Qstress_%01d.npy" %self.
                                                  POD_LIST[0])
  self.S1_dof_full = np.load(self.cwd+"/"+self.ssDirectory+"/Qstress1_%01d.npy" %self.
                                                  POD_LIST[0])
  self.S01_dof_full = np.load(self.cwd+"/"+self.ssDirectory+"/Qstress01_%01d.npy" %self
                                                  .POD_LIST[0])
  self.S10_dof_full = np.load(self.cwd+"/"+self.ssDirectory+"/Qstress10_%01d.npy" %self
                                                  .POD_LIST[0])
  for i in range(1,self.NUMBER_OF_POD):
    Q_dof_full_load =np.load(self.cwd+"/"+self.ssDirectory+'/Q_dof_%01d.npy' %self.
                                                  POD_LIST[i]
    self.Q_dof_full=np.concatenate((self.Q_dof_full,Q_dof_full_load), axis=1)
    S_dof_full_load =np.load(self.cwd+"/"+self.ssDirectory+'/Qstress_%01d.npy' %self.
                                                  POD_LIST[i])
    self.S_dof_full=np.concatenate((self.S_dof_full,S_dof_full_load), axis=1)
    S1_dof_full_load =np.load(self.cwd+"/"+self.ssDirectory+'/Qstress1_%01d.npy' %self.
                                                  POD_LIST[i])
    self.S1_dof_full=np.concatenate((self.S1_dof_full,S1_dof_full_load), axis=1)
    S01_dof_full_load =np.load(self.cwd+"/"+self.ssDirectory+'/Qstress01_%01d.npy' %
                                                  self.POD_LIST[i])
    self.S01_dof_full=np.concatenate((self.S01_dof_full,S01_dof_full_load), axis=1)
    S10_dof_full_load =np.load(self.cwd+"/"+self.ssDirectory+'/Qstress10_%01d.npy' %
                                                  self.POD_LIST[i])
    self.S10_dof_full=np.concatenate((self.S10_dof_full,S10_dof_full_load), axis=1)
```

- **`create_reduced_dof_list(mesh,lz)` method** :
  Arguments :
  `mesh` : the actual mesh of the problem.
  `lz` : the length of region of interest.
  This function creates the list of DoFs located inside the region of interest, which is defined by the length of this region.

```
def create_reduced_dof_list(self,mesh,lz):
  self.IZ_dofinside,self.IZ_dofoutside,self.subdomains_iz=iz_dof_struc(mesh,lz)
  self.stress_dofinside,self.stress_dofoutside,self.subdomains_stress=iz_dof_stress(
                                                mesh,lz)
```

- **`do_svd()` method** :
  This function computes the singular value decomposition of snapshot matrix and saves the reduced basis in numpy array format.

```
def do_svd(self):
  time_start = tm.time()
  print 'SVD Q_dof'
  Q_dof_iz_out = np.delete(self.Q_dof_full,self.IZ_dofinside, axis=0)
  self.V_POD_iz_out,self.sigma_POD_iz_out,self.W_POD_iz_out = np.linalg.svd(
                                                Q_dof_iz_out,0)
  np.save(self.cwd+"/"+self.podDirectory+'/V_POD_iz_out', self.V_POD_iz_out)
  np.save(self.cwd+"/"+self.podDirectory+'/sigma_POD_iz_out', self.sigma_POD_iz_out)
  print 'SVD S_dof'
  S_dof_iz_out = np.delete(self.S_dof_full,self.stress_dofinside, axis=0)
  self.S_POD_iz_out,self.sigmaS_POD_iz_out,self.WS_POD_iz_out = np.linalg.svd(
                                                S_dof_iz_out,0)
  np.save(self.cwd+"/"+self.podDirectory+'/S_POD_iz_out', self.S_POD_iz_out)
  np.save(self.cwd+"/"+self.podDirectory+'/sigmaS_POD_iz_out', self.sigmaS_POD_iz_out)
  print 'SVD S1_dof'
  S1_dof_iz_out = np.delete(self.S1_dof_full,self.stress_dofinside, axis=0)
  self.S1_POD_iz_out,self.sigmaS1_POD_iz_out,self.WS1_POD_iz_out = np.linalg.svd(
                                                S1_dof_iz_out,0)
  np.save(self.cwd+"/"+self.podDirectory+'/S1_POD_iz_out', self.S1_POD_iz_out)
  np.save(self.cwd+"/"+self.podDirectory+'/sigmaS1_POD_iz_out', self.sigmaS1_POD_iz_out
                                                )
  print 'SVD S01_dof'
  S01_dof_iz_out = np.delete(self.S01_dof_full,self.stress_dofinside, axis=0)
  self.S01_POD_iz_out,self.sigmaS01_POD_iz_out,self.WS01_POD_iz_out = np.linalg.svd(
                                                S01_dof_iz_out,0)
  np.save(self.cwd+"/"+self.podDirectory+'/S01_POD_iz_out', self.S01_POD_iz_out)
  np.save(self.cwd+"/"+self.podDirectory+'/sigmaS01_POD_iz_out', self.
                                                sigmaS01_POD_iz_out)
  print 'SVD S10_dof'
  S10_dof_iz_out = np.delete(self.S10_dof_full,self.stress_dofinside, axis=0)
  self.S10_POD_iz_out,self.sigmaS10_POD_iz_out,self.WS10_POD_iz_out = np.linalg.svd(
                                                S10_dof_iz_out,0)
  np.save(self.cwd+"/"+self.podDirectory+'/S10_POD_iz_out', self.S10_POD_iz_out)
  np.save(self.cwd+"/"+self.podDirectory+'/sigmaS10_POD_iz_out', self.
                                                sigmaS10_POD_iz_out)
  time_end = tm.time()
  print "svd time : ", time_end-time_start, " s"
```

3.3 `pod` class

This class is derived from `data_creator`, it remaps the restrained reduced basis to the full one and creates the hybrid reduced basis by adding the finite element shape functions to the full reduced basis.

- **Instantiation of class** :

  `pod(meshInput,lzInput,numberModeInput,ssListInput)`

  Arguments :
  `meshInput` : the actual mesh of the problem.
  `lzInput` : the length of region of interest.
  `numberModeInput` : the number of reduced modes used for simulation
  `ssListInput` (python list) : the set of snapshot points used to build reduced basis.

```
def __init__(self,meshInput,lzInput,numberModeInput,ssListInput):
  data_creator.__init__(self,meshInput,lzInput,ssListInput)
  # remap truncated POD matrix to full matrix
  self.create_full_pod(numberModeInput)
  # concatenate with FE shape function
  self.create_full_reduced_basis()
```

- **Overloaded version of `load_data()`** :
  This function override `load_data()` of `data_creator`, it loads the POD basis from disk instead of snapshot data.

```
def load_data(self):
  self.V_POD_iz_out=np.load(self.cwd+"/"+self.podDirectory+'/V_POD_iz_out.npy')
  self.S_POD_iz_out=np.load(self.cwd+"/"+self.podDirectory+'/S_POD_iz_out.npy')
  self.S1_POD_iz_out=np.load(self.cwd+"/"+self.podDirectory+'/S1_POD_iz_out.npy')
  self.S01_POD_iz_out=np.load(self.cwd+"/"+self.podDirectory+'/S01_POD_iz_out.npy')
  self.S10_POD_iz_out=np.load(self.cwd+"/"+self.podDirectory+'/S10_POD_iz_out.npy')
```

- **`create_full_pod(numberModeInput)`** method :
  Arguments :
  `numberModeInput` : the number of reduced modes used for simulation
  Since the singular value decomposition is used with the restrained snapshot matrix, this function creates the full size POD basis and remaps the restrained POD basis onto it.

```
def create_full_pod(self,numberModeInput):
  self.V_POD_iz_out=self.V_POD_iz_out[:,:numberModeInput]
  self.N_iz_out = np.shape(self.V_POD_iz_out)[1]
  self.S_POD_iz_out=self.S_POD_iz_out[:,:self.N_iz_out]
  self.S1_POD_iz_out=self.S1_POD_iz_out[:,:self.N_iz_out]
  self.S01_POD_iz_out=self.S01_POD_iz_out[:,:self.N_iz_out]
  self.S10_POD_iz_out=self.S10_POD_iz_out[:,:self.N_iz_out]

  self.Ndof_iz_out = np.shape(self.V_POD_iz_out)[0]
  print 'Number of POD modes  = ', self.N_iz_out

  self.V_POD_iz_out_full=np.zeros(shape=(self.Ndof,self.N_iz_out))
```

```
  self.S_POD_iz_out_full=np.zeros(shape=(self.Ndof/2,self.N_iz_out))
  self.S1_POD_iz_out_full=np.zeros(shape=(self.Ndof/2,self.N_iz_out))
  self.S01_POD_iz_out_full=np.zeros(shape=(self.Ndof/2,self.N_iz_out))
  self.S10_POD_iz_out_full=np.zeros(shape=(self.Ndof/2,self.N_iz_out))

  for iz in range(len(self.IZ_dofoutside)):
    self.V_POD_iz_out_full[self.IZ_dofoutside[iz],:]=self.V_POD_iz_out[iz,:]

  for iz in range(len(self.stress_dofoutside)):
    self.S_POD_iz_out_full[self.stress_dofoutside[iz],:]=self.S_POD_iz_out[iz,:]

  for iz in range(len(self.stress_dofoutside)):
    self.S1_POD_iz_out_full[self.stress_dofoutside[iz],:]=self.S1_POD_iz_out[iz,:]

  for iz in range(len(self.stress_dofoutside)):
    self.S01_POD_iz_out_full[self.stress_dofoutside[iz],:]=self.S01_POD_iz_out[iz,:]

  for iz in range(len(self.stress_dofoutside)):
    self.S10_POD_iz_out_full[self.stress_dofoutside[iz],:]=self.S10_POD_iz_out[iz,:]
```

- **`create_full_reduced_basis()`** method :
  This function creates the matrix form of finite element shape function corresponding to the DoFs of region of interest and combines it with the full size POD basis to obtain the hybrid reduced basis

```
def create_full_reduced_basis(self):
  V_POD_iz_in_full=np.zeros(shape=(self.Ndof,len(self.IZ_dofinside)))
  for iz in range(len(self.IZ_dofinside)):
    V_POD_iz_in_full[self.IZ_dofinside[iz],iz]=1.
  self.V_POD=np.concatenate((V_POD_iz_in_full, self.V_POD_iz_out_full), axis=1)
```

- **`save_reduced_basis()`** method :
  This function saves the hybrid reduced basis to disk

```
def save_reduced_basis(self):
  np.save(self.cwd+"/"+self.podDirectory+'/V_POD', self.V_POD)
  np.save(self.cwd+"/"+self.podDirectory+'/S_POD_out', self.S_POD_iz_out_full)
  np.save(self.cwd+"/"+self.podDirectory+'/S1_POD_out', self.S1_POD_iz_out_full)
  np.save(self.cwd+"/"+self.podDirectory+'/S01_POD_out', self.S01_POD_iz_out_full)
  np.save(self.cwd+"/"+self.podDirectory+'/S10_POD_out', self.S10_POD_iz_out_full)
  np.save(self.cwd+"/"+self.podDirectory+'/IZ_dofoutside', self.IZ_dofoutside)
```

3.4 `rid` class

This class is derived from `data_creator`, it creates the reduced integration domain (RID) and the restriction of the reduced basis to the RID. In general, the numbering of DoFs in RID is different than that of original mesh, thus a mapping between the DoFs in RID and in original mesh is used to construct the reduced basis of RID.

- **Instantiation of class** :

  `rid(meshInput,lzInput,numberModeInputForRID,ssListInput)`

Arguments :
`meshInput` : the actual mesh of the problem.
`lzInput` : the length of region of interest.
`numberModeInputForRID` : the number of reduced modes used to build the RID
`ssListInput` (python list) : the set of snapshot points used to build reduced basis.

```
def __init__(self,meshInput,lzInput,numberModeInputForRID,ssListInput):
  data_creator.__init__(self,meshInput,lzInput,ssListInput)
  #create dir and files to save data
  self.set_savedir()
  # create reduced basis to build the RID
  self.create_dataRID(meshInput,lzInput,numberModeInputForRID,ssListInput  )
  # create the list of coordinate to build the RID
  self.create_RID_coordinate(meshInput)
  self.create_RID_domain(meshInput,lzInput)
  self.create_RID_DOFs_list(meshInput)
  self.create_VPOD_over_RID(meshInput)
```

- **Overloaded version of `load_data()` :**
  This function override `load_data()` of `data_creator`, it loads the reduced basis created by `pod` instead of snapshot data.

```
def load_data(self):
  self.V_POD_iz_out=np.load(self.cwd+"/"+self.podDirectory+'/V_POD_iz_out.npy')
  self.S_POD_iz_out=np.load(self.cwd+"/"+self.podDirectory+'/S_POD_iz_out.npy')
  self.S1_POD_iz_out=np.load(self.cwd+"/"+self.podDirectory+'/S1_POD_iz_out.npy')
  self.S01_POD_iz_out=np.load(self.cwd+"/"+self.podDirectory+'/S01_POD_iz_out.npy')
  self.S10_POD_iz_out=np.load(self.cwd+"/"+self.podDirectory+'/S10_POD_iz_out.npy')
```

- **`create_dataRID(mesh,lz,numberModeInputForRID,ssListInput)` method :**
  Arguments :
  `mesh` : the actual mesh of the problem.
  `lz` : the length of region of interest.
  `numberModeInputForRID` : the number of reduced modes used to build the RID
  `ssListInput` (python list) : the set of snapshot points used to build reduced basis.
  This function prepares the data used to build the RID, the size of RID is controlled by the parameter `numberModeInputForRID`

```
def create_dataRID(self,mesh,lz,numberModeInputForRID,ssListInput):
  dataRID = pod(mesh,lz,numberModeInputForRID,ssListInput)

  self.V_POD_RID =dataRID.V_POD
  self.V_POD_out_full_RID =dataRID.V_POD_iz_out_full
  self.S_POD_RID=dataRID.S_POD_iz_out_full
  self.S1_POD_RID=dataRID.S1_POD_iz_out_full
  self.S01_POD_RID=dataRID.S01_POD_iz_out_full
  self.S10_POD_RID=dataRID.S10_POD_iz_out_full
  print np.shape(self.V_POD_out_full_RID)
```

- **`create_RID_coordinate(mesh1)` method :**
  Arguments :
  `mesh1` : the actual mesh of the problem.

This function uses DEIM to create the set of relevant DoFs from the reduced basis, and then create the set of coordinates of these DoFs. These sets are the base ingredient to build the RID.

```
def create_RID_coordinate(self,mesh1):
  dofinside0 = np.asarray(self.IZ_dofinside)
  print'deim V_POD'
  P_dof, self.F_dof_out = deim(self.V_POD_out_full_RID)
  #self.F_dof=np.append(dofinside0,self.F_dof_out)
  self.F_dof = self.F_dof_out

  print'deim S_POD'
  PS_dof, FS_dof = deim(self.S_POD_RID)

  print'deim S1_POD'
  PS1_dof, FS1_dof = deim(self.S1_POD_RID)

  print'deim S01_POD'
  PS01_dof, FS01_dof = deim(self.S01_POD_RID)

  print'deim S10_POD'
  PS10_dof, FS10_dof = deim(self.S10_POD_RID)

  FStress_dof = list(set(FS_dof) | set(FS1_dof) | set(FS01_dof) | set(FS10_dof))
  mesh1.init(self.tdim - 1, self.tdim)
  mesh1.init(self.tdim - 2, self.tdim)
  V = VectorFunctionSpace(mesh1, 'CG', 1)
  dofmap = V.dofmap()
  dofs = dofmap.dofs()
  # Get coordinates as len(dofs) x gdim array
  dofs_x = dofmap.tabulate_all_coordinates(mesh1).reshape((-1, self.gdim))
  self.rid_coor=[]
  for dof in self.F_dof:
    self.rid_coor.append(dofs_x[dof])

  for dof1 in FStress_dof:
    self.rid_coor.append(dofs_x[dof1*2])


  print "number of node used to build RID : ", np.shape(self.rid_coor)[0]
```

- **`create_RID_domain(mesh1,lz)` : method :**
  Arguments :
  `mesh1` : the actual mesh of the problem.
  `lz` : the length of region of interest.
  From the set of DoFs obtained from `create_RID_coordinate(mesh1)`, this function creates the RID of far field. The final RID is obtained by combining the RID of far field with the region of interest.

```
def create_RID_domain(self,mesh1,lz):
  print "Loop over all cells to build RID ... "
  self.subdomains = CellFunction('size_t', mesh1, 0)
  for cell in cells(mesh1):
    for i in range(len(self.rid_coor)):
      if (cell.contains(Point(self.rid_coor[i]))):
        self.subdomains[cell] = 1
        if cell.midpoint()[0]>=0.9*lz:
          for vectex_cell in vertices(cell):
```

```
            for cell1 in cells(vectex_cell) :
              self.subdomains[cell1] = 1


  class IZO(SubDomain):
    def inside(self, x, on_boundary):
      return True  if (x[0]<1.05*lz) and (x[0]>=0.) else False

  izon = IZO()
  izon.mark(self.subdomains, 1)
  self.file_subdomain<<self.subdomains
  self.RID<<self.subdomains
```

- **`create_RID_DOFs_list(mesh1)` method :**
  Arguments :
  `mesh1` : the actual mesh of the problem.
  With the RID obtained from `create_RID_domain(mesh1,lz)`, this function generates the list of the DoFs of this sub-domain, and another list of DoFs which are located completely inside the RID.

```
def create_RID_DOFs_list(self,mesh1):
  V1 = VectorFunctionSpace(mesh1, 'CG', 1)
  dx =  Measure('dx', domain=mesh1, subdomain_data=self.subdomains)
  du, du_t = TrialFunction(V1), TestFunction(V1)
  u = Function(V1)
  A_test= assemble(inner(du, du_t)*dx(0))
  A_test1= assemble(inner(du, du_t)*dx(1))

  Amatrx_sp = petsc_csr(A_test)
  Amatrx = Amatrx_sp.diagonal()
  Amatrx1_sp = petsc_csr(A_test1)
  Amatrx1 = Amatrx1_sp.diagonal()

  self.F_dofinside=[]
  self.F_dofoutside=[]
  for i in range (np.shape(Amatrx)[0]):
    if Amatrx[i] == 0 :
      self.F_dofinside.append(i)
  for i in range (np.shape(Amatrx1)[0]):
    if Amatrx1[i] != 0 :
      self.F_dofoutside.append(i)
  print len(self.F_dofinside)
  print len(self.F_dofoutside)
```

- **`create_VPOD_over_RID(mesh1)` method :**
  Arguments :
  `mesh1` : the actual mesh of the problem.
  This function creates a mapping `sub_to_glob_map` of the DoFs numbering between the RID and the original mesh, then the reduced basis of full domain is restrained to the RID and the element of the restrained reduced basis is rearranged with the mapping `sub_to_glob_map` to obtain the reduced basis of hyper reduction problem

```
def create_VPOD_over_RID(self,mesh1):
  self.Nrdof=np.shape(self.F_dofoutside)[0]
```

```
self.Ndof = np.shape(self.V_POD)[0]
self.VR0_POD = np.zeros(shape=(self.Nrdof,np.shape(self.V_POD)[1]))
self.ZtZ0=np.zeros(shape=(self.Nrdof,self.Nrdof))
for i in range(self.Nrdof):
  self.VR0_POD[i,:]=self.V_POD[self.F_dofoutside[i],:]
  if self.F_dofoutside[i] in self.F_dofinside:
    self.ZtZ0[i,i]=1

self.submesh1 = SubMesh(mesh1, self.subdomains, 1)
print 'node', self.submesh1.num_vertices()
print 'cells', self.submesh1.num_cells()
self.file_submesh<<self.submesh1

self.mydomains = CellFunction('size_t', self.submesh1)
self.mydomains.set_all(0)
dx_subdomain = Measure('dx', domain=mesh1, subdomain_data=self.mydomains)
Vt = VectorFunctionSpace(self.submesh1, "Lagrange", 1)
V = VectorFunctionSpace(mesh1, 'CG', 1)
gsub_dim = self.submesh1.geometry().dim()
submesh1_dof_coordinates = Vt.dofmap().tabulate_all_coordinates(self.submesh1).
                                          reshape(-1, gsub_dim)
mesh1_dof_coordinates = V.dofmap().tabulate_all_coordinates(mesh1).reshape(-1,
                                          gsub_dim)

mesh1_dof_index_coordinates0={}
for index,coor in enumerate(mesh1_dof_coordinates):
  mesh1_dof_index_coordinates0.setdefault(coor[0],[]).append(index)

mesh1_dof_index_coordinates1={}
for index,coor in enumerate(mesh1_dof_coordinates):
  mesh1_dof_index_coordinates1.setdefault(coor[1],[]).append(index)

sub_to_glob_map = {}
for bnd_dof_nr, bnd_dof_coords in enumerate(submesh1_dof_coordinates):
  corresponding_dofs = np.intersect1d(mesh1_dof_index_coordinates0[bnd_dof_coords[0]]
                                          , mesh1_dof_index_coordinates1[
                                          bnd_dof_coords[1]])
  if corresponding_dofs[0] not in sub_to_glob_map.values():
    sub_to_glob_map[bnd_dof_nr] = corresponding_dofs[0]
  else:
    sub_to_glob_map[bnd_dof_nr] = corresponding_dofs[1]
#print sub_to_glob_map
glob_to_sub_map = dict((v,k) for k,v in sub_to_glob_map.items())
#print glob_to_sub_map
self.VR_POD=np.zeros(shape=(np.shape(self.VR0_POD)))
self.ZtZ=np.zeros(shape=(self.Nrdof,self.Nrdof))
for i in range(self.Nrdof):
  ai=glob_to_sub_map[self.F_dofoutside[i]]
  self.VR_POD[ai]=self.VR0_POD[i]
  self.ZtZ[ai,ai]=self.ZtZ0[i,i]

np.save(self.cwd+"/"+self.rpodDirectory+'/ZtZ', self.ZtZ)
np.save(self.cwd+"/"+self.rpodDirectory+'/VR_POD', self.VR_POD)
np.save(self.cwd+"/"+self.rpodDirectory+'/F_inside', self.F_dofinside)
np.save(self.cwd+"/"+self.rpodDirectory+'/F_outside', self.F_dofoutside)
```

3.5 `PMFF_solver` class

This class gathers all the data obtained from `pod` and `rid` and performs a reduced model computation over the RID. The `Form` and `Jacobian` matrix will be assembled only over the RID to computed the reduced coordinates. The full solution is then obtained thanks to the full reduced basis.

- **Instantiation of class** :

  `PMFF_solver(meshInput,K_input,f_input)`

  Arguments :
  `meshInput` : the actual mesh of the problem.
  `K_input` : the Young modulus of inclusion.
  `f_input` : loading applied to the part defined by `Top` class.

```
def __init__(self,meshInput,K_input,f_input):
  # save dir
  self.set_savedir(K_input)
  # Mesh
  self.mesh = meshInput
  # Parameter
  self.set_parameter(K_input)
  #Create function spaces
  self.create_fullmesh_functionspace()
  # MeshFunctions and Measures for loading and subdomain
  self.set_fullmesh_measures()
  # File to stock data
  self.create_data_file(K_input)
  #Loading
  self.set_loading(f_input)
  self.load_reduced_basis_RID()
  self.create_submesh(self.mesh)
  self.create_submesh_measures()
  self.create_submesh_functionspace()
  self.define_bc_sub()
  self.K_function()
```

- **`set_savedir(K_re)` method** :
  Arguments :
  `K_re` : the Young modulus of inclusion.
  As its name, this function defines the locations to loads `pod`, `rid` data and to save PMFF solution.

```
def set_savedir(self,K_re):
  self.cwd = os.getcwd()
  self.podDirectory = "POD_Data"
  self.ssDirectory = "Offline_Data"
  self.ridDirectory = "RID_Data"
  self.rpodDirectory = "RPOD_Data"
  savedir = "result_PMFF/K_%01d" %K_re
  if os.path.isdir(savedir):
    shutil.rmtree(savedir)
  self.file_online = File(savedir+"/online/u_online.pvd")
  self.file_stress = File(savedir+"/stress/stress_xx.pvd")
```

- **`set_parameter(Kre_input)` method** :
  Arguments :
  `Kre_input` : the Young modulus of inclusion.
  This function defines the constant parameters of problem

```
def set_parameter(self,Kre_input):
  #beam height
  self.h = 0.1
  self.l = 1.
  # loading length
  self.lf=0.2
  # length of interest zone
  self.lz=0.2
  #newmark parameter
  self.theta=0.5
  self.gamma = 0.5
  self.beta = 0.25
  #time step
  self.dt=0.0025
  self.t_max1 = 1.
  # parameters of steel
  self.rho=8000./1000000000.
  self.K, self.nu = 200.0, 0.3
  self.K_re = Kre_input
  self.mu, self.lmbda = Constant(self.K/(2*(1 + self.nu))), Constant(self.K*self.nu/((1
                                                    + self.nu)*(1 - 2*self.nu)))
  self.mu_re, self.lmbda_re = Constant(self.K_re/(2*(1 + self.nu))), Constant(self.K_re
                                                   *self.nu/((1 + self.nu)*(1 - 2*self.nu))
                                                   )
```

- **`create_fullmesh_functionspace()` method** :
  This function creates the function spaces which are defined over the original mesh. Then it creates the function associated with these spaces.

```
def create_fullmesh_functionspace(self):
  self.V = VectorFunctionSpace(self.mesh, 'CG', 1)
  self.V0 = FunctionSpace(self.mesh, 'DG', 1)
  self.VS = TensorFunctionSpace(self.mesh, 'CG', 1)
  self.Vq1 = FunctionSpace(self.mesh,  'CG', 1)
  #Solution, test and trial functions
  self.du, self.du_t = TrialFunction(self.V), TestFunction(self.V)
  self.uk = Function(self.V)
  self.u_pred,self.v_pred=Function(self.V),Function(self.V)
  self.u_n,self.v_n,self.a_n = Function(self.V),Function(self.V),Function(self.V)
  self.u,self.v,self.a = Function(self.V),Function(self.V),Function(self.V)

  self.d = self.uk.geometric_dimension()
  self.I = Identity(self.d)
```

- **`set_fullmesh_measures()` method** :
  Since we will compute the matrices over the RID, we need to defied the integrations over the full domain and over the RID. This function defines the former, it creates the new measures of the full mesh associated with the sub-domain of inclusion and of the loading boundary part.

```
def set_fullmesh_measures(self):
  # Initialize sub-domain instances
```

```
  self.top = Top()
  self.top.set_length(self.h,self.l,self.lf)
  # Initialize mesh function for loading
  self.boundaries = FacetFunction("size_t", self.mesh)
  self.boundaries.set_all(0)
  self.top.mark(self.boundaries, 2)
  # Load mesh fuction from file for inclusion marker
  self.Ydomains = MeshFunction('size_t', self.mesh, 'mesh_physical_region.xml')
  self.ds = Measure('ds', domain=self.mesh, subdomain_data=self.boundaries)
  self.dx = Measure('dx', domain=self.mesh, subdomain_data=self.Ydomains)
```

- **`create_data_file(Kre_input)` method** :
  This function loads the FE solution saved in numpy array format from disk and create another array stock the PMFF solution

```
  def create_data_file(self,Kre_input):
    time1 = pl.arange(0,self.t_max1, self.dt)
    self.Q_dof_PMFF = np.zeros(shape=(self.u.vector().size(),len(time1)))
    self.Q_dof = np.load(self.cwd+"/"+self.ssDirectory+'/Q_dof_%01d.npy' %Kre_input)
```

- **`load_reduced_basis_RID()` method** :
  As its name, this function loads the full reduced basis defined over the full domain and the restrained reduced basis defined over the RID.

```
def load_reduced_basis_RID(self):
  V_POD_load = np.load(self.cwd+"/"+self.podDirectory+'/V_POD.npy')
  self.V_POD = sp.csr_matrix(V_POD_load)
  VR_POD_load = np.load(self.cwd+"/"+self.rpodDirectory+'/VR_POD.npy')
  self.VR_POD = sp.csr_matrix(VR_POD_load)
  self.ZtZ= np.load(self.cwd+"/"+self.rpodDirectory+'/ZtZ.npy')
  self.N = np.shape(self.V_POD)[1]
  self.Ndof = np.shape(self.V_POD)[0]
  self.Nrdof=np.shape(self.VR_POD)[0]
  self.ZtZ_sp=sp.spdiags(np.diag(self.ZtZ),0,np.shape(self.ZtZ)[0],np.shape(self.ZtZ)[1
                                                    ])
```

- **`create_submesh(mesh)` method** :
  Argument :
  `mesh` : the current mesh of problem.
  This function loads the RID sub-domain created by `rid` and create a new sub-mesh of current mesh.

```
  def create_submesh(self,mesh):
    self.subdomains = MeshFunction('size_t',mesh, self.cwd+"/"+self.ridDirectory+"/
                                              subdomain.xml")
    self.submesh = SubMesh(mesh, self.subdomains, 1)
```

- **`set_submesh_measures()` method** :
  Like the `set_fullmesh_measures()` method, this function creates the new measures of the sub-mesh associated with the sub-domain of inclusion and of the loading boundary part. These measures are used to perform the integration over the RID.

```
def create_submesh_measures(self):
  self.mydomains = CellFunction('size_t', self.submesh)
```

```
  self.mydomains.set_all(0)
  self.subdomain_y = Omega0()
  self.subdomain_y.mark(self.mydomains, 1)
  #plot(self.mydomains,interactive=True)
  self.dx_subdomain =Measure('dx', domain=self.submesh, subdomain_data=self.mydomains)
  self.sub_boundaries = FacetFunction("size_t", self.submesh)
  self.sub_boundaries.set_all(0)
  top = Top()
  top.set_length(self.h,self.l,self.lf)
  top.mark(self.sub_boundaries, 2)
  self.ds_subdomain = Measure('ds', domain=self.submesh, subdomain_data=self.
                                                    sub_boundaries)
```

- **`create_submesh_functionspace()` method** :
  This function creates the function spaces which are defined over the sub-mesh. Then it creates the test and trial function associated with these spaces.

```
def create_submesh_functionspace(self):
  self.Vt = VectorFunctionSpace(self.submesh, "Lagrange", 1)
  self.du_sub, self.du_t_sub = TrialFunction(self.Vt), TestFunction(self.Vt)
```

- **`define_bc_sub()` method** :
  This function constructs the Dirichlet boundary condition for displacement field `u` defined over sub-mesh

```
def define_bc_sub(self):
  left =  CompiledSubDomain("near(x[0], side) && on_boundary", side = 0.0)
  c0 = Expression(("0.0", "0.0"))
  self.bc_sub = DirichletBC(self.Vt, c0, left)
```

- **`FormR(u_f,u_test,u_prediction,force)` method** :
  Arguments :
  `u_f` : the solution of last time-step.
  `u_test` : the test function.
  `u_prediction` : the prediction of `u` at current time-step.
  `force` : the loading.
  This function return the residual defined in (6). It takes the same form as in `Form` of `nonlinear_dynamics` class, but now the integrations are performed over the RID instead of full domain

```
def FormR(self,u_f,u_test,u_prediction,force):
  du_f=nabla_grad(u_f)
  du_te=nabla_grad(u_test)
  P_int=(self.I+du_f)*(self.lmbda*tr(0.5*(du_f+du_f.T)+du_f.T*du_f)*self.I+2*self.mu*(0
                                                    .5*(du_f+du_f.T)+du_f.T*du_f))
  P_re=(self.I+du_f)*(self.lmbda_re*tr(0.5*(du_f+du_f.T)+du_f.T*du_f)*self.I+2*self.
                                                    mu_re*(0.5*(du_f+du_f.T)+du_f.T*du_f))
  Fr=(self.rho/(self.beta*self.dt*self.dt))*inner(u_f-u_prediction,u_test)*self.
                                                    dx_subdomain+inner(P_int,du_te)*self.
                                                    dx_subdomain(0)+inner(P_re,du_te)*self.
                                                    dx_subdomain(1)-inner(force,u_test)*self
                                                    .ds_subdomain(2)
  return Fr;
```

- **`Jacobian(u_f,u_trial,u_test)` method** :
  Arguments :
  `u_f` : the solution of last time-step.
  `u_test` : the test function.
  `u_trial` : the trial function.
  Similar to `Jacobian()` function `nonlinear_dynamics` class, this function return the jacobian the `FormR`, the integrations are also performed over the RID

```
def JacobianR(self,u_f,u_trial,u_test):
  du_f=nabla_grad(u_f)
  du_tr=nabla_grad(u_trial)
  du_te=nabla_grad(u_test)
  dF0_int=(self.I+du_f)*(self.lmbda*tr(0.5*(du_tr+du_tr.T)+du_tr.T*du_f+du_f.T*du_tr)*
                                                self.I+2*self.mu*(0.5*(du_tr+du_tr.T)+
                                                du_tr.T*du_f+du_f.T*du_tr))
  dF1_int=du_tr*(self.lmbda*tr(0.5*(du_f+du_f.T)+du_f.T*du_f)*self.I+2*self.mu*(0.5*(
                                                du_f+du_f.T)+du_f.T*du_f))
  dF0_re=(self.I+du_f)*(self.lmbda_re*tr(0.5*(du_tr+du_tr.T)+du_tr.T*du_f+du_f.T*du_tr)
                                                *self.I+2*self.mu_re*(0.5*(du_tr+du_tr.T
                                                )+du_tr.T*du_f+du_f.T*du_tr))
  dF1_re=du_tr*(self.lmbda_re*tr(0.5*(du_f+du_f.T)+du_f.T*du_f)*self.I+2*self.mu_re*(0.
                                                5*(du_f+du_f.T)+du_f.T*du_f))
  Jr=(self.rho/(self.beta*self.dt*self.dt))*inner(u_trial,u_test)*self.dx_subdomain+(
                                                inner(dF0_int+dF1_int,du_te)*self.
                                                dx_subdomain(0)+inner(dF0_re+dF1_re,
                                                du_te)*self.dx_subdomain(1))
  return Jr;
```

- **`variational_pmff_solveur()` method** :
  This is the main part of this class, it assembles the `JacobianR` and `FormR` matrix over the sub-mesh, apply the boundary condition to these matrices and solves the system of equation. After having the reduced coordinates, the full solution is obtained by using the full reduced basis. The PMFF solution is stocked in a numpy array and then the error between PMFF and FEM is computed.

```
def variational_pmff_solveur(self):
    j=0
    M_0=assemble(self.rho*inner(self.du, self.du_t)*self.dx)
    F_0=assemble(-inner(self.Pstress(self.u),nabla_grad(self.du_t))*self.dx)
    def solve_a():
      solve(M_0, self.a_n.vector(), F_0)

    time2 = pl.arange(0,self.t_max1, self.dt)
    start_online = tm.clock()
    for i, t in enumerate(time2):
      self.T.t=t
      tol = 1.0E-4
      iter = 0
      maxiter = 250
      eps = 1.0
      if i==0:
        solve_a()
      else:
        # Predictions for u^n and v^n
        self.u_pred.vector().set_local(self.u_n.vector().array() + self.dt*self.v_n.
                                                vector().array() + 0.5*self.dt**2*(1-2*
```

```
                                            self.beta)*self.a_n.vector().array())
        self.v_pred.vector().set_local(self.v_n.vector().array() + self.dt*(1-self.
                                            gamma)*self.a_n.vector().array())
        self.uk.vector().set_local(self.u_n.vector().array()) # use u_n for predicted
                                            solution at n+1 step
        self.u_pred_sub=interpolate(self.u_pred, self.Vt)
        # Solve for displacement at n+1 step
        while eps > tol and iter < maxiter:
          iter += 1
          self.uk_sub=interpolate(self.uk, self.Vt)
          A1=assemble(self.JacobianR(self.uk_sub,self.du_sub,self.du_t_sub))
          b1=-assemble(self.FormR(self.uk_sub,self.du_t_sub,self.u_pred_sub,self.T))
          self.bc_sub.apply(A1,b1)
          A1_sp=petsc_csr(A1)
          b1_sp=b1.array()
          A1_spz=self.ZtZ_sp.dot(A1_sp)
          b1_array=self.ZtZ_sp.dot(b1_sp)
          Ar1=self.VR_POD.T.dot(A1_spz.dot(self.VR_POD))
          br1=self.VR_POD.T.dot(b1_array)
          q_du=splina.spsolve(Ar1, br1)
          temps=self.V_POD.dot(q_du)
          eps = lina.norm(q_du, ord=np.Inf)
          print 'Norm:', eps
          self.u.vector().set_local(self.uk.vector().array()+temps)
          self.uk.vector().set_local(self.u.vector().array())

        #update displacement, vlocity and acceleration at n+1 step
        self.a_n.vector().set_local((self.u.vector().array()-self.u_pred.vector().array
                                            ()) / (self.beta*self.dt**2))
        self.v_n.vector().set_local(self.v_pred.vector().array() + self.dt*self.gamma*
                                            self.a_n.vector().array())
        self.u_n.vector().set_local(self.u.vector().array())
        self.Q_dof_PMFF[:,j] =  self.u.vector().array()
        j = j+1
        print t
    end_online = tm.clock()
    self.onlinetime=end_online - start_online
    A_err=Q_dof_PMFF-Q_dof
    Norm_err=np.zeros(np.shape(A_err)[1])
    for i in  range(np.shape(A_err)[1]):
      if np.linalg.norm(Q_dof[:,i], ord=2) !=0:
        Norm_err[i]=(np.linalg.norm(A_err[:,i], ord=2))/(np.linalg.norm(Q_dof[:,i], ord
                                            =2))
    self.data_err= np.sum(Norm_err)/(float(len(Norm_err)-1))
    print self.data_err
```

3.6 Script to run PMFF

This is and example script to use PMFF. First of all, we need to load all class of `PMFF` and `dolfin`:

```
from PMFF import *
from dolfin import *
```

We define a file use to save the error data :

```
savedir = "results_error"
filepod=File(savedir+"/error.pvd")
```

```
myfile_pod=open(savedir+"/error.pvd","a",0)
```

Load the mesh :

```
mesh=Mesh("mesh0.xml")
```

Define the loading function :

```
Pmax=.01
DeltaP=-0.002
omega=16.
tl=0.25
alpha=1.
T = Expression(("0.",'t <= tl ? Pmax*t/tl : alpha*Pmax-DeltaP*sin(omega*pi*t)'), t=0.0, tl=tl
                                                ,Pmax=Pmax,DeltaP=DeltaP,omega=omega,alpha=
                                                alpha)
```

Perform the FE simulation for the Young modulus of inclusion varying from 100 to 200 GPa

```
Kre_list=np.arange(100.,201.,10.)
for K_re in Kre_list:
  problem = nonlinear_dynamics(mesh,K_re,T)
  problem.variational_formulation_solveur()
```

Set the length of ROI (`lz`), number of modes used for `pod` and `rid`, the set of snapshot will be used to build the reduced basis. Then we create an instance of `data_creator` with these parameter. The decomposition of snapshot matrix is performed by calling `do_svd()` function.

```
lz = 0.2
npod, nrid =21, 50
podlist = [100,150,200]
data = data_creator(mesh,lz,podlist)
data.do_svd()
```

The reduced basis and the RID are constructed by instantiating 2 class `pod` and `rid`

```
Vpod = pod(mesh,lz,npod,podlist)
Vpod.save_reduced_basis()
RID_construction = rid(mesh,lz,nrid,podlist)
```

In the end, `PMFF_solver` is instantiated to perform the PMFF simulation.

```
for K_re in Kre_list:
  run_pmff = PMFF_solver(mesh,K_re,T)
  run_pmff.variational_pmff_solveur()
  myfile_pod.write("%s %s \n" % (K_re, run_pmff.data_err))
  myfile_pod.close
```