Horizon 2020

ROMSOC

Reduced Order Modelling, Simulation and Optimization of Coupled systems

Coupled parameterized reduced order modelling of thermo-mechanical phenomena arising in blast furnace

Deliverable number: D5.3

Version 0.1

Funded by the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie Grant Agreement No. 765374

Project Acronym: ROMSOC
Project Full Title: Reduced Order Modelling, Simulation and Optimization of Coupled systems
Call: H2020-MSCA-ITN-2017
Topic: Innovative Training Network
Type of Action: European Industrial Doctorates
Grant Number: 765374

Editor:	Andrés Prieto, ITMATI
Deliverable nature:	Report (R)
Dissemination level:	Public (PU)
Contractual Delivery Date:	01/07/2021
Actual Delivery Date	01/07/2021
Number of pages:	24
Keywords:	Blast furnace hearth, Thermo-mechanical axisymmetric model, Finite element method, Benchmark verification, Moder order reduction, Proper orthogonal decomposition.
Authors:	Nirav Vasant Shah, ESR10, Mathematical Analysis, Modelling, and Applications, mathLab, SISSA Dr. Michele Girfoglio, Post doctoral researcher, Mathematical Analysis, Modelling, and Applications, mathLab, SISSA Dr. Patricia Barral, Associate Professor, Applied Mathematics, University of Santiago de Compostela, ITMATI Prof. Peregrina Quintela, Full Professor, Applied Mathematics, University of Santiago de Compostela, ITMATI Prof. Gianluigi Rozza, Full professor, Mathematical Analysis, Modelling, and Applications, mathLab, SISSA Dr. Alejandro lengomin, R&D Engineer, Global Research and Development Center, AMIII
Peer review:	

Abstract

The benchmark cases related to coupled thermomechanical phenomena arising in blast furnace have been implemented using open-source libraries. This document provides details of numerical implementation of the benchmark tests to verify, validate and reproduce the results of numerical experiments. This documentation is aimed towards smooth transition for next developers and students interested in detailed investigation of out work. We first provide reference to theoretical discussion to benchmark cases. Next, we provide instructions related to required libraries and installation. Next, we provide reference to theoretical discussion of the benchmark cases. Next, we explain the details of experiment. Finally, we specify licensing and usage terms.

Disclaimer & acknowledgment
This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie Grant Agreement No. 765374.

Contents

1 Introduction 1

2 Prerequisites 1

3 Installation 1

4 Running the benchmark cases 1

5 Benchmark cases 1

5.1 Reading the mesh 1

5.2 Thermal model 4

5.3 Mechanical model 5

5.4 Coupled model 7

5.5 Real case 10

5.6 Reduced basis method 12

6 License 23

7 Disclaimer 23

List of Acronyms

ESR	Early Stage Researcher
ITMATI	Technological Institute of Industrial Mathematics
SISSA	Scuola Internazionale Superiore di Studi Avanzati
AMIII	ArcelorMittal Innovación Investigación e Inversion S.L.

1 Introduction

In our previous work [1], we focused on the physical problem, mathematical formulation, statement of the benchmark problems and design of numerical experiments. We now provide the code for numerical experiments, using open source libraries, with the objective of validating or reproducing the results and smooth handover of numerical software to future developers as per best practices [2].

2 Prerequisites

We use python 3.6.9 as the programming language. In this project we use the libraries :

- FEniCS 2019.1.0 ([3],[4],[5],[6], www.fenicsproject.org)
- RBniCS 0.1.dev1 ([7],www.rbnicsproject.org)
- Matplolib 3.1.2 ([8],www.matplotlib.org)
- numpy 1.17.4 ([9],www.numpy.org)

```
from dolfin import * #FEniCS library
from mshr import * #mshr - mesh generation component of FEniCS
from rbnics import * #RBniCS library
import matplotlib.pyplot as plt #Matplotlib library
import numpy as np #Numpy library
```

The solutions are stored in .pvd format, which can later be viewed with Paraview (www.paraview.org).

3 Installation

Simply clone the public repository:

```
git clone https://github.com/ROMSOC/benchmark_thermomechanical_model
```

4 Running the benchmark cases

Source codes for input data are provided in the folder *source_files*. Source codes for running the benchmark are provided in folder *source*. After running the benchmark, results are stored in folder *result_files*.

Run required .py file e.g. *file_name.py* as,

```
python3 file_name.py
```

5 Benchmark cases

5.1 Reading the mesh

We first construct the polygonal domain using mshr.

```
# Define domain
domain = Polygon([Point(0.,0.),Point(7.05,0.),
        Point(7.05,7.265),Point(5.3,7.265),
        Point(5.3,4.065),Point(4.95,4.065),
        Point(4.95,3.565),Point(4.6,3.565),
        Point(4.6,2.965),Point(4.25,2.965),
        Point(4.25,2.365),Point(0.,2.365)])
```

The domain is divided into triangular subdomains. We define the mapping from the reference domain to the parametrized domain. As an example, we take the subdomain 1 whose coordinates on reference domain are $(0,0),(4.25,0),(0,2.365)$. It is deformed to the parametrized subdomain with coordinates

$(0,0), (\mu_6, 2), (0, \mu_0)$. The μ_6 and μ_0 are the 6th and 0th parameter of zero-indexed tuple Ξ. This mapping can be defined as,

```
  {
    ("0", "0"): ("0", "0"),
    ("4.25", "0"): ("mu[6]/2", "0"),
    ("0", "2.365"): ("0", "mu[0]")
  }, # subdomain 1
```

The RBniCS computes the mapping for each subdomain and uses it during affine transformation of operators. Similarly, the mappings and subdomains are created for other 29 subdomains.

Next, we set the subdomain marker:

```
# Loop over all mappings and set subdomain markers
for i, vertices_mapping in enumerate(vertices_mappings):
  print(i,vertices_mapping.keys())
  subdomain_i = Polygon([Point(*[float(coord) for coord in vertex]) for vertex in
                                             counterclockwise(vertices_mapping.keys())])
  domain.set_subdomain(i + 1, subdomain_i)
```

The mesh and subdomains are created based on subdomain markers.

```
# Create mesh
mesh = generate_mesh(domain, 30) #30 specifies the mesh size.

# Create subdomains
subdomains = MeshFunction("size_t", mesh, 2, mesh.domains())
```

We now set the boundary markers. Since our boundaries are divided across 20 subdomains and these boundaries are later reclassified into 5 different boundaries, we define 20 classes and set markers for each of these boundaries. For example, for the boundary γ_s, we define the class

```
class Gamma_s(SubDomain):
  def inside(self, x, on_boundary):
    return x[0] < DOLFIN_EPS and on_boundary
```

and create an instant of this class and set boundary marker as 1.

```
gamma_s = Gamma_s()
gamma_s.mark(boundaries, 1)
```

Unlike γ_s, the bottom boundary γ_- is shared by 5 subdomains, we define 5 different classes and correspondingly 5 different markers.

```
class Gamma_minus1(SubDomain):
  def inside(self, x, on_boundary):
    return x[1] < DOLFIN_EPS and x[0] < (4.255 + DOLFIN_EPS) and on_boundary

class Gamma_minus2(SubDomain):
  def inside(self, x, on_boundary):
    return x[1] < DOLFIN_EPS and x[0] > (4.2 - DOLFIN_EPS) and x[0] < (4.65 + DOLFIN_EPS)
                                              and on_boundary

class Gamma_minus3(SubDomain):
  def inside(self, x, on_boundary):
    return x[1] < DOLFIN_EPS and x[0] > (4.55 - DOLFIN_EPS) and x[0] < (5.05 + DOLFIN_EPS
                                             ) and on_boundary

class Gamma_minus4(SubDomain):
  def inside(self, x, on_boundary):
    return x[1] < DOLFIN_EPS and x[0] > (4.9 - DOLFIN_EPS) and x[0] < (5.4 + DOLFIN_EPS)
                                              and on_boundary

class Gamma_minus5(SubDomain):
  def inside(self, x, on_boundary):
    return x[1] < DOLFIN_EPS and x[0] > (5.25 - DOLFIN_EPS) and x[0] < (7.15 + DOLFIN_EPS
                                              ) and on_boundary
```

```
gamma_minus1 = Gamma_minus1()
gamma_minus1.mark(boundaries, 2)
gamma_minus2 = Gamma_minus2()
gamma_minus2.mark(boundaries, 3)
gamma_minus3 = Gamma_minus3()
gamma_minus3.mark(boundaries, 4)
gamma_minus4 = Gamma_minus4()
gamma_minus4.mark(boundaries, 5)
gamma_minus5 = Gamma_minus5()
gamma_minus5.mark(boundaries, 6)
```

In a similar manner, markers are set for other boundaries and the mesh, boundary markers, subdomain markers and affine maps are stored.

```
# Save mesh data
os.system("mkdir ../input_data/mesh_data")
VerticesMappingIO.save_file(vertices_mappings, ".", "../data_files/mesh_data/
                                              hearth_vertices_mapping.vmp")
File("../data_files/mesh_data/hearth.xml") << mesh
File("../data_files/mesh_data/hearth_physical_region.xml") << subdomains
File("../data_files/mesh_data/hearth_facet_region.xml") << boundaries
XDMFFile("../data_files/mesh_data/hearth.xdmf").write(mesh)
XDMFFile("../data_files/mesh_data/hearth_physical_region.xdmf").write(subdomains)
XDMFFile("../data_files/mesh_data/hearth_facet_region.xdmf").write(boundaries)
File("../data_files/mesh_data/hearth.pvd") << mesh
File("../data_files/mesh_data/hearth_physical_region.pvd") << subdomains
File("../data_files/mesh_data/hearth_facet_region.pvd") << boundaries
```

At the beginning of any benchmark test, we first read the mesh, which is divided into subdomains and also read the boundary markers. The volume of each subdomains and length of each boundary are measured. Notice that all boundary markers with same boundary condition are combined.

```
# Read the mesh file from specified path.
mesh = Mesh("../../benchmarks/data_files/mesh_data/hearth.xml")
domains = MeshFunction('size_t',mesh,mesh.topology().dim()) # Read domain marker
subdomains = MeshFunction("size_t", mesh, "../../benchmarks/data_files/mesh_data/
                                              hearth_physical_region.xml") # Read
                                              subdomain markers
boundaries = MeshFunction("size_t", mesh, "../../benchmarks/data_files/mesh_data/
                                              hearth_facet_region.xml") # Read boundary
                                              markers
dx = Measure('dx', domain = mesh, subdomain_data = domains) # Volume measure
ds = Measure('ds', domain = mesh, subdomain_data = boundaries) # Boundary measure
n = as_vector(FacetNormal(mesh)) # Edge unit normal vector

d_bottom = ds(2) + ds(3) + ds(4) + ds(5) + ds(6) # Markers of bottom boundary \gamma_{-}
d_out = ds(7) + ds(8) + ds(9) + ds(10) + ds(11) # Markers of outer boundary \gamma_{out}
d_sf = ds(13) + ds(14) + ds(15) + ds(16) + ds(17) + ds(18) + ds(19) + ds(20) # Markers of
                                              inner boundary \gamma_{sf}
```

5.2 Thermal model

We define the norm for $\psi \in H^1_r(\omega)$,

```
#Computation of H^1_r(\omega) norm
def compute_h1r_norm(psi,mesh):
  r = SpatialCoordinate(mesh)[0]
  dx = Measure('dx', domain = mesh)
  a = inner(psi,psi)*r*dx + inner(grad(psi),grad(psi))*r*dx
  A = assemble(a)
  return sqrt(A)
```

Next, we define the range of polynomials degree for measuring $p-$convergence and an object to store the relative error.

```
error_T_vector = [] #List to store error in temperature w.r.t. polynomial degree
p = range(1,4)  # List of polynomial degrees
```

Next, we specify the relevant physical parameters, which are than later used in variational form.

```
k = 10. # Thermal conductivity
h_fluid = 200.  # Convection coefficient on \gamma_{sf}
h_right = 2000.  # Convection coefficient on \gamma_{out}
h_bottom = 2000.  # Convection coefficient on \gamma_{-}
```

For every polynomial degree, we define the relevant ”Lagrange” function space of a particular degree. We also define the solution field and test function in this space.

```
# Define function space
VT = FunctionSpace(mesh,"CG",i)  # Function space for temperature
psi, T_ = TestFunction(VT), TrialFunction(VT)  # Evaluate trial and test function
T = Function(VT, name = "temperature increase")
x = list()
x.append(Expression("x[0]", element=VT.ufl_element())) #r coordinate
x.append(Expression("x[1]", element=VT.ufl_element())) #y coordinate
```

We then define and solve the weak formulation.

```
# solving weak form of energy equation
a_T = k * inner(grad(psi),grad(T_)) * x[0] * dx + \
  h_fluid * psi * T_ * x[0] * d_sf + h_right * psi * T_ * x[0] * d_out + \
  h_bottom * psi * T_ * x[0] * d_bottom # Bilinear side
l_T = h_fluid * psi * (x[0] * x[0] * x[1] + k/h_fluid * ( 2 * x[0] * x[1] * n[0] + x[0] *
                                             x[0] * n[1] ) ) * x[0] * d_sf + \
  h_right * psi * (x[0]*x[0]*x[1]+2*x[0]*x[1]*k/h_right) * x[0] * d_out + \
  h_bottom * psi * (x[0] * x[0] * x[1] - x[0] * x[0] * k / h_bottom) * x[0] * d_bottom +
                                             \
  -4 * k * x[1] * psi * x[0] * dx + psi * k * x[0] * x[0] * x[0] * ds(12) # Linear side
solve(a_T == l_T, T) # Solve the variational form
```

After performing the computations, we store the data in format compatible with paraview for further visualization. Also, we plot the p-convergence.

```
# Plotting and visualization
File("../../benchmarks/result_files/thermal_model/temperature_computed.pvd") << T
File("../../benchmarks/result_files/thermal_model/temperature_analytical.pvd") <<
                                             T_analytical
error_temperature = Function(VT) #Function for Spatial distribution of temperature
                                             absolute error
error_temperature.vector()[:] = abs(T_analytical.vector().get_local() - T.vector().
                                             get_local())
```

```
File("../../benchmarks/result_files/thermal_model/temperature_absolute_error.pvd") <<
                                                  error_temperature

# Plotting and printing convergence tests
plt.figure(figsize=[10,8])
a = plt.semilogy([1,2,3],error_T_vector,marker='o',linewidth=4)
plt.xticks([1,2,3],fontsize=18)
plt.yticks(fontsize=18)
plt.xlabel('Polynomial degree',fontsize=24)
plt.ylabel('Relative error',fontsize=24)
plt.axis('tight')
plt.savefig("../../benchmarks/result_files/thermal_model/convergence_test")
plt.show()
```

5.3 Mechanical model

We define the norm for $\vec{\phi} \in \mathbb{U}$.

```
#Computation of \mathbb{U} norm
def compute_U_norm(phi,mesh):
  x = SpatialCoordinate(mesh)
  dx = Measure('dx', domain = mesh)
  a = inner(phi,phi)*x[0]*dx + inner(grad(phi),grad(phi))*x[0]*dx + (phi[0]**2)/x[0]*dx
  A = assemble(a)
  return sqrt(A)
```

Next, we define the axisymmetric stress and strain tensor.

```
# Axisymmetric strain tensor definition. Alternative could be to express strain as vector
                                                  using Voigt notation.
def eps(u):
  return \
    sym(as_tensor([[u[0].dx(0), u[0].dx(1), 0. ],\
    [u[1].dx(0), u[1].dx(1), 0.],\
    [0., 0., u[0]/x[0]]]))

# Axisymmetric stress tensor definition. Alternative could be to express stress as vector
                                                  using Voigt notation.
def sigma(u):
  return lmbda * tr(eps(u)) * Identity(3) + 2.0 * mu * eps(u)
```

Since, our aim is to assess $p-$convergence, we define the range of polynomial degrees and measure corresponding relative errors.

```
error_u_vector = [] #List for absolute error in displacement
p = range(1,4) #Polynomial degrees
```

We introduce the physical parameters : Young's modulus, Poisson ratio and Lamé parameters.

```
E = Constant(5e9) # Young's modulus
nu = Constant(0.2) # Poisson's ratio
mu = E/2/(1+nu) # Lam\'e parameter
lmbda = E*nu/(1+nu)/(1-2*nu) # Lam\'e parameter
```

We now define the function space for displacement and stress. We also initialize variables for the test function and the solution field.

```
# Define function space
VM = VectorFunctionSpace(mesh,"CG",i) # Function space for displacement
x = Expression(("x[0]","x[1]"), element=VM.ufl_element())
```

```
VS = FunctionSpace(mesh,"CG",max(i-1,1))
# Function space for Von Mises stress NOTE: when i=1, the VS is of degree 1 and not 0.
phi, u_ = TestFunction(VM), TrialFunction(VM)
u = Function(VM, name = "Displacement") # u[0] = u_r and u[1] = u_y
```

Before defining weak formulation, we define the variables related to source term and the boundary data.

```
# Dirichlet boundary data
bcs_M = [DirichletBC( VM.sub(0), Constant(0.), 'x[0] < DOLFIN_EPS and on_boundary'), \
  DirichletBC( VM.sub(1), Constant(0.), 'near(x[1],0) and on_boundary')]
# Set Dirichlet boundary. Note that only functions which satisfy zero normal displacement
                                              on \gamma_s \cup \gamma_- are admissible.

# Source term and relevant boundary data
f0_r = - (2*E*nu*1e-4*x[0]/(1-2*nu)/(1+nu)+2*E*1e-4*x[0]/(1+nu))
f0_y = - (4*E*1e-4*x[1]/(1+nu)+4*E*1e-4*x[1]*nu/(1-2*nu)/(1+nu))

g_plus_r = 2*E*1e-4*x[0]*x[1]/(1+nu)
g_plus_y = E / (1-2*nu) / (1+nu) * (2*nu*1e-4*x[1]*x[1]+(1-nu)*1e-4*x[0]*x[0])

g_minus_r = -g_plus_r

g_sf_r = E / (1-2*nu) / (1+nu) * (1e-4 * x[1] * x[1] + nu * 1e-4 * x[0] * x[0]) * n[0] +
                                             2 * E * 1e-4 * x[0] * x[1] / (1 + nu) * n[1]
g_sf_y = 2 * E * 1e-4 * x[0] * x[1] / (1 + nu) * n[0] + E / (1-2*nu) / (1+nu) * (2 * nu *
                                              1e-4 * x[1] * x[1] + (1 - nu) * 1e-4 * x[0]
                                              * x[0]) * n[1]

g_out_r = E / (1-2*nu) / (1+nu) * (1e-4 * x[1] * x[1] + nu * 1e-4 * x[0] * x[0])
g_out_y = 2*E*1e-4*x[0]*x[1]/(1+nu)
```

The weak form is then defined and solved for computing displacement. Based on this displacement, the Von Mises stress is computed.

```
# solving weak form of momentum equation
# Bilinear form
a_M = inner(sigma(u_),eps(phi)) * x[0] * dx
#linear form
l_M = (phi[0] * f0_r + phi[1] * f0_y) * x[0] * dx + (phi[0] * g_plus_r + phi[1] *
                                             g_plus_y) * x[0] * ds(12) + \
(phi[0] * g_minus_r) * x[0] * d_bottom + (phi[0] * g_sf_r + phi[1] * g_sf_y) * x[0] *
                                             d_sf + \
(phi[0] * g_out_r + phi[1] * g_out_y) * x[0] * d_out
solve(a_M == l_M, u, bcs_M)
```

```
# Von Mises stress computed displacement
sigma_dev = sigma(u) - tr(sigma(u)) / 3 * Identity(3)
sigma_vm = sqrt(3 * inner( sigma_dev, sigma_dev) / 2) # Von mises stress
# Von Mises stress analytical displacement
sigma_dev_analytical = sigma(u_analytical) - tr(sigma(u_analytical)) / 3 * Identity(3)
sigma_vm_analytical = sqrt(3 * inner( sigma_dev_analytical, sigma_dev_analytical) / 2) #
                                             Von mises stress

# Compute H^1_r norm of error
error_u = compute_U_norm(u_analytical-u,mesh)/compute_U_norm(u_analytical,mesh)
error_u_vector.append(error_u)
print("Relative error in U-norm : ",str(error_u))
```

The data is stored in format compatible to paraview. We also plot the polynomial degree vs the relative error.

```
# Post-processing and visualization
```

```
File("../../benchmarks/result_files/mechanical_model/displacement_computed.pvd") << u
File("../../benchmarks/result_files/mechanical_model/displacement_analytical.pvd") <<
                                             u_analytical
File("../../benchmarks/result_files/mechanical_model/displacement_error.pvd") << project(
                                             u_analytical-u,VM)

error_stress = Function(VS) #Function for absolute error in stress tensor
error_stress.vector()[:] = abs(project(sigma_vm,VS).vector().get_local() - project(
                                             sigma_vm_analytical,VS).vector().get_local()
                                             )
File("../../benchmarks/result_files/mechanical_model/von_mises_stress_computed.pvd") <<
                                             project(sigma_vm,VS)
File("../../benchmarks/result_files/mechanical_model/von_mises_stress_analytical.pvd") <<
                                              project(sigma_vm_analytical,VS)
File("../../benchmarks/result_files/mechanical_model/von_mises_stress_error.pvd") <<
                                             project(error_stress,VS)

#Convergence tests
plt.figure(figsize=[10,8])
a = plt.semilogy([1,2,3],error_u_vector,marker='o',linewidth=4)
plt.xticks([1,2,3],fontsize=18)
plt.yticks(fontsize=18)
plt.xlabel('Polynomial degree',fontsize=24)
plt.ylabel('Relative error',fontsize=24)
plt.axis('tight')
plt.savefig("../../benchmarks/result_files/mechanical_model/convergence_test")
plt.show() # To show the plots

print("Relative error in U norm: "+ str(error_u_vector))
```

5.4 Coupled model

Similar to the Thermal model (Section 5.2), we first solve the weak form of energy equation.

```
# Define function space
VT = FunctionSpace(mesh,"CG",3) # Function space for temperature
psi, T_ = TestFunction(VT), TrialFunction(VT) # Evaluate trial and test function
T = Function(VT, name = "temperature increase")

# Known analytical solution, Thermal material properties and Boundary data
T_analytical = Expression('x[0]*x[0]*x[1]',degree = 3)
VT_analytical = FunctionSpace(mesh,"CG",3) #Space for analytical solution
T_analytical = project(T_analytical,VT_analytical)
k = 10. # Thermal conductivity
h_fluid = 200. # Convection coefficient on \gamma_{sf}
h_right = 2000. # Convection coefficient on \gamma_{out}
h_bottom = 2000. # Convection coefficient on \gamma_{-}
x = list()
x.append(Expression("x[0]", element=VT.ufl_element())) #r coordinate
x.append(Expression("x[1]", element=VT.ufl_element())) #y coordinate

# solving weak form of energy equation
a_T = k * inner(grad(psi),grad(T_)) * x[0] * dx + \
  h_fluid * psi * T_ * x[0] * d_sf + h_right * psi * T_ * x[0] * d_out + \
  h_bottom * psi * T_ * x[0] * d_bottom # Bilinear side
l_T = h_fluid * psi * (x[0] * x[0] * x[1] + k/h_fluid * ( 2 * x[0] * x[1] * n[0] + x[0] *
                                              x[0] * n[1] ) ) * x[0] * d_sf + \
  h_right * psi * (x[0]*x[0]*x[1]+2*x[0]*x[1]*k/h_right) * x[0] * d_out + \
  h_bottom * psi * (x[0] * x[0] * x[1] - x[0] * x[0] * k / h_bottom) * x[0] * d_bottom +
                                             \
  -4 * k * x[1] * psi * x[0] * dx + psi * k * x[0] * x[0] * x[0] * ds(12) # Linear side
```

```
solve(a_T == l_T, T) # Solve the variational form
```

Also, similar to mechanical model (section 5.3) , we define the $\mathbb{U}$ norm, stress and strain tensor. Additionally, we define the thermomechanical stress tensor.

```
# Define \mathbb{U} norm
def compute_U_norm(phi,mesh):
  x = SpatialCoordinate(mesh)
  a = inner(phi,phi)*x[0]*dx + inner(grad(phi),grad(phi))*x[0]*dx + (phi[0]**2/x[0])*dx
  A = assemble(a)
  return sqrt(A)

# Axisymmetric strain tensor definition. Alternative could be to express strain as vector
                                             using Voigt notation.
def eps(u):
  return \
    sym(as_tensor([[u[0].dx(0), u[0].dx(1), 0. ],\
    [u[1].dx(0), u[1].dx(1), 0.],\
    [0., 0., u[0]/x[0]]]))

# Axisymmetric thermo-mechanical stress tensor definition. Alternative could be to
                                            express as vector using Voigt notation.
def sigma(u,T):
  return lmbda * tr(eps(u)) * Identity(3) + 2.0 * mu * eps(u) - (2 * mu + 3 * lmbda) *
                                            alpha * (T - T_0) * Identity(3)

# Axisymmetric mechanical stress tensor definition. Alternative could be to express as
                                            vector using Voigt notation.
def sigma2(u):
  return lmbda * tr(eps(u)) * Identity(3) + 2.0 * mu * eps(u)
```

Next, the physical data is specified.

```
T_0 = 298 # Reference temperature for zero thermal stress
E = Constant(5e9) # Young's modulus
nu = Constant(0.2) # Poisson's ratio
mu = E/2/(1+nu) # Lame\'e parameter
lmbda = E*nu/(1+nu)/(1-2*nu) # Lame\'e parameter
alpha = Constant(1e-6) # Thermal expansion coefficient
```

Similar to the mechanical model (section 5.3), we solve the weak form.

```
# Define function space for displacement
VM = VectorFunctionSpace(mesh,"CG",i) # Function space for displacement
x = Expression(("x[0]","x[1]"), element=VM.ufl_element())
phi, u_ = TestFunction(VM), TrialFunction(VM)
u = Function(VM, name = "Displacement") # u[0] = u_r and u[1] = u_y
VS = FunctionSpace(mesh,"CG",max(i-1,1)) # Function space for shear component of stress

# Dirichlet boundary data
bcs_M = [DirichletBC( VM.sub(0), Constant(0.), 'x[0] < DOLFIN_EPS and on_boundary'),
                                            DirichletBC( VM.sub(1), Constant(0.), 'near(
                                            x[1],0) and on_boundary')]

#Boundary and source terms
f0_r = - (2*E*nu*1e-4*x[0]/(1-2*nu)/(1+nu)+2*E*1e-4*x[0]/(1+nu)-2*E*x[0]*x[1]*alpha/(1-2*
                                            nu))
f0_y = - (4*E*1e-4*x[1]/(1+nu)+4*E*1e-4*x[1]*nu/(1-2*nu)/(1+nu)-E*x[0]*x[0]*alpha/(1-2*nu
                                            ))

g_plus_r = 2*E*1e-4*x[0]*x[1]/(1+nu)
```

```
g_plus_y = E / (1-2*nu) / (1+nu) * (2*nu*1e-4*x[1]*x[1]+(1-nu)*1e-4*x[0]*x[0]) - E*alpha/
                                            (1-2*nu)*(x[0]*x[0]*x[1] - T_0)

g_minus_r = -g_plus_r

g_sf_r = (E / (1-2*nu) / (1+nu) * (1e-4 * x[1] * x[1] + nu * 1e-4 * x[0] * x[0]) - E*
                                            alpha/(1-2*nu)*(x[0]*x[0]*x[1] - T_0)) * n[0
                                            ] + 2 * E * 1e-4 * x[0] * x[1] / (1 + nu) *
                                            n[1]
g_sf_y = 2 * E * 1e-4 * x[0] * x[1] / (1 + nu) * n[0] + (E / (1-2*nu) / (1+nu) * (2 * nu
                                            * 1e-4 * x[1] * x[1] + (1 - nu) * 1e-4 * x[0
                                            ] * x[0])- E*alpha/(1-2*nu)*(x[0]*x[0]*x[1]
                                            - T_0)) * n[1]

g_out_r = E / (1-2*nu) / (1+nu) * (1e-4 * x[1] * x[1] + nu * 1e-4 * x[0] * x[0]) - E*
                                            alpha/(1-2*nu)*(x[0]*x[0]*x[1] - T_0)
g_out_y = 2*E*1e-4*x[0]*x[1]/(1+nu)

# solving weak form of momentum equation
# This is not bilinear side as terms related to thermal stress are included.
a_M1 = inner(sigma(u_,T),eps(phi)) * x[0] * dx
# This is not linear side as terms related to thermal stress are not included.
l_M1 = (phi[0] * f0_r + phi[1] * f0_y) * x[0] * dx + (phi[0] * g_plus_r + phi[1] *
                                            g_plus_y) * x[0] * ds(12) + \
(phi[0] * g_minus_r) * x[0] * d_bottom + (phi[0] * g_sf_r + phi[1] * g_sf_y) * x[0] *
                                            d_sf + \
(phi[0] * g_out_r + phi[1] * g_out_y) * x[0] * d_out
F = a_M1 - l_M1
a_M = lhs(F) # Now a_M is bilinear form
l_M = rhs(F) # Now l_M is linear form
solve(a_M == l_M, u, bcs_M) # Solve equation

# Compute \mathbb{U} norm of error
error_u = compute_U_norm(u_analytical-u,mesh)/compute_U_norm(u_analytical,mesh)
error_u_vector.append(error_u)
print("Relative error in U-norm : ",str(error_u))
```

We compute the relevant stress fields.

```
# Von Mises stress for computed displacement
sigma_dev = sigma(u,T) - tr(sigma(u,T)) / 3 * Identity(3)
sigma_vm = sqrt(3 * inner( sigma_dev, sigma_dev) / 2) # Von mises stress
# Von Mises stress for analytical displacement
sigma_dev_analytical = sigma(u_analytical,T) - tr(sigma(u_analytical,T)) / 3 * Identity(3
                                            )
sigma_vm_analytical = sqrt(3 * inner( sigma_dev_analytical, sigma_dev_analytical) / 2) #
                                            Von mises stress
# Spherical stress for computed displacement
sigma_spherical = tr(sigma(u,T)) / 3
# Spherical stress for analytical displacement
sigma_spherical_analytical = tr(sigma(u_analytical,T)) / 3
# Spherical mechanical stress for computed displacement
sigma_spherical_non_thermal = tr(sigma2(u)) / 3
```

Finally, we store the solution field for further visualization.

```
# Post-processing and visualization
File("../../benchmarks/result_files/coupled_model/Temperature_computed.pvd") << T
File("../../benchmarks/result_files/coupled_model/Temperature_analytical.pvd") <<
                                            T_analytical
File("../../benchmarks/result_files/coupled_model/Teperature_error.pvd") << project(T-
                                            T_analytical,VT)
```

```
File("../../benchmarks/result_files/coupled_model/displacement_computed.pvd") << u
File("../../benchmarks/result_files/coupled_model/displacement_analytical.pvd") <<
                                             u_analytical
File("../../benchmarks/result_files/coupled_model/displacement_absolute_error.pvd") <<
                                             project(u_analytical - u,VM)

File("../../benchmarks/result_files/coupled_model/von_mises_computed_coupling.pvd") <<
                                             project(sigma_vm,VS)
File("../../benchmarks/result_files/coupled_model/von_mises_analytical_coupling.pvd") <<
                                             project(sigma_vm_analytical,VS)
error_stress_von_mises = Function(VS)
error_stress_von_mises.vector()[:] = abs(project(sigma_vm,VS).vector().get_local() -
                                             project(sigma_vm_analytical,VS).vector().
                                             get_local())
File("../../benchmarks/result_files/coupled_model/von_mises_stress_error_coupling.pvd") <
                                             < project(error_stress_von_mises,VS)

File("../../benchmarks/result_files/coupled_model/difference_in_spherical_stress.pvd") <<
                                              project(sigma_spherical -
                                             sigma_spherical_non_thermal,VS)
File("../../benchmarks/result_files/coupled_model/thermal_part_of_stress.pvd") << project
                                             (-(2 * mu + 3 * lmbda) * alpha * (T - T_0),
                                             VS)
error_stress_spherical = Function(VS)
error_stress_spherical.vector()[:] = abs(project(sigma_spherical,VS).vector().get_local()
                                              - project(sigma_spherical_non_thermal - (2
                                             * mu + 3 * lmbda) * alpha * (T - T_0),VS).
                                             vector().get_local())
File("../../benchmarks/result_files/coupled_model/absolute_error_spherical_stress.pvd") <
                                             < error_stress_spherical

#Convergence tests
plt.figure(figsize=[10,8])
a = plt.semilogy([1,2,3],error_u_vector,marker='o',linewidth=4)
plt.xticks([1,2,3],fontsize=18)
plt.yticks(fontsize=18)
plt.xlabel('Polynomial degree',fontsize=24)
plt.ylabel('Relative error',fontsize=24)
plt.axis('tight')
plt.savefig('../../benchmarks/result_files/coupled_model/
                                             Convergence_coupling_displacement_benchmark_comparison
                                             .png')
plt.show()
```

5.5 Real case

We first compute the temperature field and insert it into the weak form of momentum equation. The function space for temperature, test function and temperature solution field are initiated.

```
# Function space for temperature
VT = FunctionSpace(mesh,"CG",1) # Function space for temperature
x = list()
x.append(Expression("x[0]", element=VT.ufl_element())) # Read mesh coordinates where x[0]
                                              = r and x[1] = y
psi, T_ = TestFunction(VT), TrialFunction(VT)
T = Function(VT, name = "temperature")
```

We also specify the physical parameters before solving the weak form.

```
# Thermal material properties and boundary data
k = 10 # Thermal conductivity
h_fluid = 200 # Convection coefficient h_{c,f}
h_right = 2000 # Convection coefficient h_{c,out}
h_bottom = 2000 # Convection coefficient h_{c,-}
T_right = 313 # Environmental temperature T_{out}
T_fluid = 1773 # Enviromental temperature T_{sf}
T_bottom = 313 # Temperature on Bottom boundary
```

The weak form for energy equation is specified and solved for computing the temperature field.

```
# solving weak form of energy equation
a_T = k * inner(grad(psi),grad(T_)) * x[0] * dx + h_fluid * psi * T_ * x[0] * d_sf +
                                             h_right * psi * T_ * x[0] * d_out + h_bottom
                                              * psi * T_ * x[0] * d_bottom # bilinear
                                             form
l_T = h_fluid * psi * T_fluid * x[0] * d_sf + h_right * psi * T_right * x[0] * d_out +
                                             h_bottom * psi * T_bottom * x[0] * d_bottom
                                             # linear form
solve(a_T == l_T, T) # solve the equation
```

Similarly, for the momentum equation, we initiate the function space for displacement, test function and displacement solution field, and specify the physical parameters.

```
# # Function space for displacement
VM = VectorFunctionSpace(mesh,"CG",1) # Function space for displacement
x = Expression(("x[0]","x[1]"), element=VM.ufl_element()) # Read mesh coordinates where x
                                             [0] = r and x[1] = y
phi, u_ = TestFunction(VM), TrialFunction(VM)
u = Function(VM, name = "Displacement")
```

```
# Mechanical material parameters and imposition of Dirichlet boundary value
T_0 = 298 # Reference temperature
E = Constant(5e9) # Young's modulus
nu = Constant(0.2) # Poission's ratio
mu = E/2/(1+nu) # Lam\' parameter
lmbda = E*nu/(1+nu)/(1-2*nu) # Lam\' parameter
alpha = Constant(1e-6) # Thermal expansion coefficient
W = 0 # weight at top boundary
rho = 7460 # Molten metal density
g = 10. # Gravitation accelaration
p = rho * g * (7.265-x[1]) # Fluid pressure
```

Additionally, we define the axisymmetric stress and strain tensor.

```
# Axisymmetric strain tensor definition. Alternative could be to express stress as vector
                                              using Voigt notation.
def eps(u):
  return \
    sym(as_tensor([[u[0].dx(0), u[0].dx(1), 0. ],\
    [u[1].dx(0), u[1].dx(1), 0.],\
    [0., 0., u[0]/x[0]]]))

# Axisymmetric stress tensor definition. Alternative could be to express stress as vector
                                              using Voigt notation.
def sigma(u):
  return lmbda * tr(eps(u)) * Identity(3) + 2.0 * mu * eps(u)
```

We specify than the Dirichlet boundary conditions, weak form for momentum equation and solve the system of equations.

```
# Dirichlet boundary data (Displacement).
bcs_M = [DirichletBC( VM.sub(0), Constant(0.), 'x[0] < DOLFIN_EPS and on_boundary'),
                                            DirichletBC( VM.sub(1), Constant(0.), 'near(
                                            x[1],0) and on_boundary')]

#solving weak form of momentum equation
a_M = inner(sigma(u_),eps(phi)) * x[0] * dx # bilinear form
l_M = (2 * mu + 3 * lmbda) * alpha * inner((T - T_0) * Identity(3), eps(phi)) * x[0] * dx
                                             - dot( phi, W * n) * x[0] * ds(12) - dot(
                                            phi, p * n) * x[0] * d_sf # linear form
solve(a_M == l_M, u, bcs_M) # solve variation form
```

We store the displacement field and temperature field for visualization with paraview.

```
# Plotting and visualization
File("../../benchmarks/result_files/actual_problem/Temperature_computed.pvd") << T
File("../../benchmarks/result_files/actual_problem/Displacement.pvd") << u
```

5.6 Reduced basis method

For the affine geometric parametrization, we use 2 decorators : one for the affine Shape Parametrization and the other for transfer of operators between reference domain and parametrized domain.

```
@PullBackFormsToReferenceDomain() #Decorator for operator transformation between
                                            parameterized domain to reference domain
@AffineShapeParametrization("../../benchmarks/data_files/mesh_data/
                                            hearth_vertices_mapping.vmp") #Decorator for
                                             shape parametrization with mapping defined
                                            in specified file
```

To compute the temperature field required to compute displacement for coupling model, we use another decorator,

```
@ExactParametrizedFunctions() #Decorator for computing temperature field required for
                                            linear side
```

5.6.1 Thermal system

We first define the class *HearthThermal*, inherited from *EllipticCoerciveProblem*, for the thermal system.

```
class HearthThermal(EllipticCoerciveProblem):
```

The default initialization involves necessary parameters of the problem.

```
  # Default initialization of members
  def __init__(self, V, **kwargs):
    # Call the standard initialization
    EllipticCoerciveProblem.__init__(self, V, **kwargs)
    # ... and also store FEniCS data structures for assembly
    assert "subdomains" in kwargs
    assert "boundaries" in kwargs
    assert "mesh" in kwargs
    assert "h_cf" in kwargs
    assert "h_out" in kwargs
    assert "h_bottom" in kwargs
    self.subdomains, self.boundaries = kwargs["subdomains"], kwargs["boundaries"]
    self.u = TrialFunction(V)
    self.v = TestFunction(V)
    self.dx = Measure("dx")(subdomain_data=subdomains)
    self.ds = Measure("ds")(subdomain_data=boundaries)
    self.subdomains = subdomains
    self.boundaries = boundaries
    self.reference_mesh = kwargs["mesh"]
    self.h_cf = kwargs["h_cf"]
    self.h_out = kwargs["h_out"]
    self.h_bottom = kwargs["h_bottom"]
    self.x0 = Expression("x[0]", element=V.ufl_element())
```

Next, the affine mulitplicative terms and weak formulation are defined.

```
  # Return theta multiplicative terms of the affine expansion of the problem.
  def compute_theta(self, term):
    mu = self.mu
    if term == "a":
      theta_a0 = mu[10]
      theta_a1 = 1.0
      return (theta_a0, theta_a1)
    elif term == "f":
      theta_f0 = 1.0
      return (theta_f0, )
    else:
      raise ValueError("Invalid term for compute_theta().")

  # Return forms resulting from the discretization of the affine expansion of the problem
                                                    operators.
  def assemble_operator(self, term):
    u = self.u
    v = self.v
    reference_mesh = self.reference_mesh
    dx = self.dx
    ds = self.ds
    h_cf = self.h_cf
    h_out = self.h_out
    h_bottom = self.h_bottom
    r = self.x0
    d_bottom = ds(2) + ds(3) + ds(4) + ds(5) + ds(6)
    d_out = ds(7) + ds(8) + ds(9) + ds(10) + ds(11)
    d_sf = ds(13) + ds(14) + ds(15) + ds(16) + ds(17) + ds(18) + ds(19) + ds(20)
    if term == "a":
      a0 = inner(grad(u), grad(v))*r*dx
      a1 = h_bottom*u*v*r*d_bottom + h_out*u*v*r*d_out + h_cf*u*v*r*d_sf
      return (a0, a1)
    elif term == "f":
      f0 = h_bottom*313*v*r*d_bottom + h_out*313*v*r*d_out + h_cf*1773*v*r*d_sf
      return (f0, )
    elif term == "inner_product":
      x0 = u*v*r*dx + inner(grad(u), grad(v))*r*dx
      return (x0,)
    else:
      raise ValueError("Invalid term for assemble_operator().")
```

Using the *HearthThermal* class we now perform POD-Galerkin approximation of the thermal problem. The function space is defined first. Next, an instant of *HearthThermal* class, *hearth_problem_thermal* is created and reduction with POD-Galerkin method is performed.

```
# 2A. Create Finite Element space (Lagrange P1)
VT = FunctionSpace(mesh, "Lagrange", 1) # For temperature

# 3A. Allocate an object of the Hearth class
```

```
hearth_problem_thermal = HearthThermal(VT, subdomains=subdomains, boundaries=boundaries,
                                             mesh=mesh, h_cf=200., h_out=2000., h_bottom=
                                             2000.)
#specify and set range of each parameter
mu_range = [(2.3,2.4), (0.5,0.7), (0.5,0.7), (0.4,0.6), (3.05,3.35), (13.5,14.5), (8.3,8.
                                             7), (8.8,9.2), (9.8,10.2), (10.4,10.8), (9.8
                                             ,10.2), (2.08e9,2.08e9), (1.39e9,1.39e9), (
                                             1e-6,1e-6)]
hearth_problem_thermal.set_mu_range(mu_range)

# 4A. Prepare reduction with a POD-Galerkin method
#NOTE : truth_problem attribute is FEM problem and reduced_problem is RB problem
pod_galerkin_method_thermal = PODGalerkin(hearth_problem_thermal)
pod_galerkin_method_thermal.set_Nmax(100) #Maximum size of reduced basis space
pod_galerkin_method_thermal.set_tolerance(1e-4) #Maximum eigenvalue tolerance
```

After computing the full order model solution for Coupling thermomechanical model, we perform the offline phase of thermal system.

```
# 5A. Perform the offline phase
pod_galerkin_method_thermal.initialize_training_set(1000) #Initialize training set with
                                             specified number of training parameters
reduced_hearth_problem_thermal = pod_galerkin_method_thermal.offline() #Perform offline
                                             phase
```

Next, we perform the error analysis, compute the time taken for truth solution and reduced solution.

```
# 7A. Perform an error analysis
pod_galerkin_method_thermal.initialize_testing_set(50) #Initialize error analysis with
                                             specified number of parameters
pod_galerkin_method_thermal.error_analysis() #Perform error analysis

# 8A1. Perform a speedup analysis - Compute time for truth solutions
pod_galerkin_method_thermal.initialize_testing_set(50) #Initialize truth time computation
                                              with specified number of parameters
testing_set_speedup_analysis = pod_galerkin_method_thermal.testing_set

pod_galerkin_method_thermal._patch_truth_solve(True) #To enable cahce reading

truth_timer = Timer("parallel") #Timer for computation of FEM solution
time_thermal_truth = np.empty(len(testing_set_speedup_analysis)) #Storage of time taken
                                             for solving FEM equation. It is a vector of
                                             size of number of speedup analysis
                                             parameters

# Iteration over speedup analysis parameters for measuring time taken for FEM solution
for (mu_index, mu_test) in enumerate(testing_set_speedup_analysis):
  print(TextLine(str(mu_index), fill="#"))
  pod_galerkin_method_thermal.truth_problem.set_mu(mu_test) #Set the parameter
  truth_timer.start()
  pod_galerkin_method_thermal.truth_problem.solve() #Solve the FEM problem
  truth_time_thermal = truth_timer.stop()
  print("Truth time thermal : ",truth_time_thermal)
  time_thermal_truth[mu_index] = truth_time_thermal #Save time taken for truth solve

np.save("time_thermal_truth",time_thermal_truth) #Save time taken for computation of FEM
                                             solution

pod_galerkin_method_thermal._undo_patch_truth_solve(True) #To enable cache reading

# 8A2. Perform a speedup analysis - Compute time for reduced solutions
pod_galerkin_method_thermal._patch_truth_solve(True) #To disable cache reading
```

```
reduced_timer = Timer("serial") #Timer for computation of RB solution
max_basis_function = reduced_hearth_problem_thermal.N #Size of reduced basis space
time_thermal_reduced = np.empty([max_basis_function,len(testing_set_speedup_analysis)]) #
                                             Storage of time taken for solving RB
                                             equation. It is a matrix of size size of
                                             reduced basis space \times number of speedup
                                              analysis parameters

# Iteration over speedup analysis parameters for measuring time for RB solution
for basis_size in range(1,max_basis_function+1):
  for (mu_index, mu_test) in enumerate(testing_set_speedup_analysis):
    print(TextLine(str(mu_index), fill="#"))
    pod_galerkin_method_thermal.reduced_problem.set_mu(mu_test) #Set parameter
    reduced_timer.start()
    pod_galerkin_method_thermal.reduced_problem.solve(basis_size) #Solve the RB problem
    rb_time_thermal = reduced_timer.stop()
    print("Reduced time thermal : ",rb_time_thermal)
    time_thermal_reduced[basis_size-1,mu_index] = rb_time_thermal #Save time taken for RB
                                              solve

pod_galerkin_method_thermal._undo_patch_truth_solve(True) #To disable cache reading

np.save("time_thermal_reduced",time_thermal_reduced) #Save time taken for computation for
                                              RB solution
```

For any new parameter *online_mu*, the full order model solution and reduced basis solution can be performed using these classes.

```
pod_galerkin_method_thermal.reduced_problem.set_mu(online_mu) #Set parameter
T_rb = pod_galerkin_method_thermal.reduced_problem.solve() #Reduced problem solve
pod_galerkin_method_thermal.reduced_problem.export_solution(filename="
                                             reference_domain_thermal_rb") #Save solution
                                              for visualization with paraview
T_rb = pod_galerkin_method_thermal.reduced_problem.basis_functions * T_rb #RB solution
                                             projected back to FEM space
pod_galerkin_method_thermal.truth_problem.set_mu(online_mu) #Set parameter
T = pod_galerkin_method_thermal.truth_problem.solve() #FEM problem solve
pod_galerkin_method_thermal.truth_problem.export_solution(filename="reference_domain_fem"
                                             ) #Save solution for visualization with
                                             paraview
pod_galerkin_method_thermal.truth_problem.mesh_motion.move_mesh() #Deform mesh as per
                                             geometric parameters
File("HearthThermal/reference_domain_thermal_spatial_error.pvd") << project(T-T_rb,VT) #
                                             Spatial error
pod_galerkin_method_thermal.truth_problem.mesh_motion.reset_reference() #Restore mesh to
                                             reference configuration
```

5.6.2 Mechanical system

We first define the class *HearthMechanical*, inherited from *EllipticCoerciveProblem*, for the mechanical system.

```
class HearthMechanical(EllipticCoerciveProblem):
```

We specify the default initialization for this class.

```
  # Default initialization of members
  def __init__(self, V, **kwargs):
    # Call the standard initialization
    EllipticCoerciveProblem.__init__(self, V, **kwargs)
    # ... and also store FEniCS data structures for assembly
    assert "subdomains" in kwargs
    assert "boundaries" in kwargs
    assert "mesh" in kwargs
    self.normal = as_vector(FacetNormal(kwargs["mesh"]))
    self.subdomains, self.boundaries = kwargs["subdomains"], kwargs["boundaries"]
    self.u = TrialFunction(V)
    self.v = TestFunction(V)
    self.dx = Measure("dx")(subdomain_data=subdomains)
    self.ds = Measure("ds")(subdomain_data=boundaries)
    self.subdomains = subdomains
    self.boundaries = boundaries
    self.x0 = Expression("x[0]", element=V.sub(0).ufl_element())
    self.x1 = Expression("x[1]", element=V.sub(1).ufl_element())
```

Next, the affine multiplicative terms, weak forms and strain tensors are defined.

```
  # Return theta multiplicative terms of the affine expansion of the problem.
  def compute_theta(self, term):
    mu = self.mu
    if term == "a":
      theta_a0 = mu[11]
      theta_a1 = 2*mu[12]
      return (theta_a0, theta_a1, )
    elif term == "f":
      theta_f0 = 1.0
      return (theta_f0, )
    else:
      raise ValueError("Invalid term for compute_theta().")

  # Return strain tensor
  def strain(self,u):
    r = self.x0
    return sym(as_tensor([[u[0].dx(0), u[0].dx(1), 0. ], [u[1].dx(0), u[1].dx(1), 0.], [0
          ., 0., u[0]/r]]))

  # Return forms resulting from the discretization of the affine expansion of the problem
          operators.
  def assemble_operator(self, term):
    u = self.u
    v = self.v
    dx = self.dx
    ds = self.ds
    r = self.x0
    x1 = self.x1
    n = self.normal
    d_bottom = ds(2) + ds(3) + ds(4) + ds(5) + ds(6)
    d_out = ds(7) + ds(8) + ds(9) + ds(10) + ds(11)
    d_sf = ds(13) + ds(14) + ds(15) + ds(16) + ds(17) + ds(18) + ds(19) + ds(20)
    if term == "a":
      a0 = (u[0].dx(0)+u[1].dx(1)+u[0]/r)*(v[0].dx(0)+v[1].dx(1)+v[0]/r)*r*dx
      a1 = (u[0].dx(0)*v[0].dx(0) + u[1].dx(1)*v[1].dx(1) + (u[0]*v[0])/(r)**2 + 0.5*(u[0
          ].dx(1)+u[1].dx(0))*(v[0].dx(1)+v[1].dx(0)))
           * r * dx
      return (a0, a1,)
    elif term == "f":
      f0 = - dot( v, 7460*9.81*(7.265-x1)*n) * r * d_sf
      return (f0,)
    elif term == "inner_product":
      x0 = inner(u,v) * r * dx + inner(self.strain(u),self.strain(v)) * r * dx
      return (x0,)
    elif term == "dirichlet_bc":
      bc0 = [DirichletBC(self.V.sub(0), Constant(0.), self.boundaries, 1),
        DirichletBC(self.V.sub(1), Constant(0.), self.boundaries, 2),
        DirichletBC(self.V.sub(1), Constant(0.), self.boundaries, 3),
        DirichletBC(self.V.sub(1), Constant(0.), self.boundaries, 4),
        DirichletBC(self.V.sub(1), Constant(0.), self.boundaries, 5),
        DirichletBC(self.V.sub(1), Constant(0.), self.boundaries, 6),]
      return (bc0,)
    else:
      raise ValueError("Invalid term for assemble_operator().")
```

The function space and an instant of *HearthMechanical* class that is *hearth_problem_mechanical* are defined.

```
# 2B. Create Finite Element space (Lagrange P1)
VM = VectorFunctionSpace(mesh,"Lagrange",1) # For mechanical

# 3B. Allocate an object of the HearthThermoMechanical class
hearth_problem_mechanical = HearthMechanical(VM, subdomains=subdomains, boundaries=
                                             boundaries, mesh=mesh)
#specify and set range of each parameter
mu_range = [(2.3,2.4), (0.5,0.7), (0.5,0.7), (0.4,0.6), (3.05,3.35), (13.5,14.5), (8.3,8.
                                             7), (8.8,9.2), (9.8,10.2), (10.4,10.8), (10
                                             .,10.), (1.9e9,2.5e9), (1.2e9,1.8e9), (1e-6,
                                             1e-6)]
hearth_problem_mechanical.set_mu_range(mu_range)
```

Next, reduction with POD-Galerkin method is performed and offline phase is performed.

```
# 4B. Prepare reduction with a POD-Galerkin method
#NOTE : truth_problem attribute is FEM problem and reduced_problem is RB problem
pod_galerkin_method_mechanical = PODGalerkin(hearth_problem_mechanical)
pod_galerkin_method_mechanical.set_Nmax(100) #Maximum size of reduced basis space
pod_galerkin_method_mechanical.set_tolerance(1e-4) #Maximum eigenvalue tolerance

# 5B. Perform the offline phase
pod_galerkin_method_mechanical.initialize_training_set(1000) #Initialize training set
                                             with specified number of training parameters
reduced_hearth_problem_mechanical = pod_galerkin_method_mechanical.offline() #Perform
                                             offline phase
```

Error analysis is performed and time taken for computation of truth solution and reduced basis solution are measured.

```
# 7B. Perform an error analysis
pod_galerkin_method_mechanical.initialize_testing_set(50) #Initialize error analysis set
                                             with specified number of parameters
pod_galerkin_method_mechanical.error_analysis() #Perform error analysis

# 8B1. Perform a speedup analysis - Compute time for truth solutions
pod_galerkin_method_mechanical.initialize_testing_set(50) #Initialize speedup analysis
                                             set with specified number of parameters
testing_set_speedup_analysis = pod_galerkin_method_mechanical.testing_set

pod_galerkin_method_mechanical._patch_truth_solve(True) # To disable cache reading

truth_timer = Timer("parallel") #Timer for computation of FEM solution
time_mechanical_truth = np.empty(len(testing_set_speedup_analysis)) #Storage of time
                                             taken for solving FEM equation. It is a
                                             vector of size of number of speedup analysis
                                              parameters

# Iteration over speedup analysis parameters for measuring time taken for FEM solution
for (mu_index, mu_test) in enumerate(testing_set_speedup_analysis):
  print(TextLine(str(mu_index), fill="#"))
  pod_galerkin_method_mechanical.truth_problem.set_mu(mu_test) #Set the parameter
  truth_timer.start()
  pod_galerkin_method_mechanical.truth_problem.solve() #Solve the FEM problem
  truth_time_mechanical = truth_timer.stop()
  print("Truth time mechanical : ",truth_time_mechanical)
  time_mechanical_truth[mu_index] = truth_time_mechanical #Save time taken for truth
                                         solve

pod_galerkin_method_mechanical._undo_patch_truth_solve(True) #To enable cache reading

np.save("time_mechanical_truth",time_mechanical_truth) #Save numpy array of time taken
                                         for FEM solution

# 8B2. Perform a speedup analysis - Compute time for reduced solutions
pod_galerkin_method_mechanical._patch_truth_solve(True) #To disable cache reading

reduced_timer = Timer("serial") #Timer for computation of reduced solution
max_basis_function = reduced_hearth_problem_mechanical.N # Size of reduced basis space
time_mechanical_reduced = np.empty([max_basis_function,len(testing_set_speedup_analysis)]
                                         ) #Storage of time taken for solving RB
                                         equation. It is a matrix of size size of
                                         reduced basis space \times number of speedup
                                          analysis parameters

# Iteration over speedup analysis parameters for measuring time taken for RB solution
for basis_size in range(1,max_basis_function+1):
  for (mu_index, mu_test) in enumerate(testing_set_speedup_analysis):
    print(TextLine(str(mu_index), fill="#"))
    pod_galerkin_method_mechanical.reduced_problem.set_mu(mu_test) #Set the parameter
    reduced_timer.start()
    pod_galerkin_method_mechanical.reduced_problem.solve(basis_size) #Solve the RB
                                         problem
    rb_time_mechanical = reduced_timer.stop()
    print("Reduced time mechanical : ",rb_time_mechanical)
    time_mechanical_reduced[basis_size-1,mu_index] = rb_time_mechanical #Save time taken
                                         for reduced basis solution

pod_galerkin_method_mechanical._undo_patch_truth_solve(True) #To enable cache reading

np.save("time_mechanical_reduced",time_mechanical_reduced) #Save numpy array of time
                                         taken for RB solution
```

For any new parameter *online_mu*, the FEM solution and reduced basis solution are computed as:

```
# 6B. Perform an online solve
pod_galerkin_method_mechanical.reduced_problem.set_mu(online_mu) #Set parameter
u_rb = pod_galerkin_method_mechanical.reduced_problem.solve() #Reduced problem solve
pod_galerkin_method_mechanical.reduced_problem.export_solution(filename="
                                         reference_domain_mechanical_rb") #Save
                                         solution for visualization with paraview
u_rb = pod_galerkin_method_mechanical.reduced_problem.basis_functions * u_rb #RB solution
                                          projected back to FEM space
pod_galerkin_method_mechanical.truth_problem.set_mu(online_mu) #Set parameter
u = pod_galerkin_method_mechanical.truth_problem.solve() #FEM problem solve
pod_galerkin_method_mechanical.truth_problem.export_solution(filename="
                                         reference_domain_fem") #Save solution for
                                         visualization with paraview
pod_galerkin_method_mechanical.truth_problem.mesh_motion.move_mesh() #Deform mesh as per
                                         geometric parameters
File("HearthMechanical/reference_domain_mechanical_spatial_error.pvd") << project(u-u_rb,
                                         VM) #Spatial error
```

```
pod_galerkin_method_mechanical.truth_problem.mesh_motion.reset_reference() #Restore mesh
                                                    to reference configuration
```

5.6.3 Coupling system

For the coupling system, we define the class *HearthThermoMechanical*.

```
class HearthThermoMechanical(EllipticCoerciveProblem):
```

The default initialization is specified,

```
  # Default initialization of members
  def __init__(self, V, **kwargs):
    # Call the standard initialization
    EllipticCoerciveProblem.__init__(self, V, **kwargs)
    # ... and also store FEniCS data structures for assembly
    assert "subdomains" in kwargs
    assert "boundaries" in kwargs
    assert "mesh" in kwargs
    assert "hearth_problem_thermal" in kwargs
    assert "ref_temperature" in kwargs
    self.subdomains, self.boundaries = kwargs["subdomains"], kwargs["boundaries"]
    self.u = TrialFunction(V)
    self.v = TestFunction(V)
    self.dx = Measure("dx")(subdomain_data=subdomains)
    self.ds = Measure("ds")(subdomain_data=boundaries)
    self.subdomains = subdomains
    self.boundaries = boundaries
    self.hearth_problem_thermal = kwargs["hearth_problem_thermal"]
    self.T_0 = kwargs["ref_temperature"]
    self.x0 = Expression("x[0]", element=V.sub(0).ufl_element())
```

Similar to mechanical system, we define the affine mutiplicative terms, weak formulation and strain tensor.

```
  # Return theta multiplicative terms of the affine expansion of the problem.
  def compute_theta(self, term):
    mu = self.mu
    if term == "a":
      theta_a0 = mu[11]
      theta_a1 = 2*mu[12]
      return (theta_a0, theta_a1,)
    elif term == "f":
      theta_f0 = (2 * mu[11] + 3 * mu[12]) * mu[13]
      return (theta_f0,)
    else:
      raise ValueError("Invalid term for compute_theta().")

  # Return strain tensor
  def strain(self,u):
    r = self.x0
    return sym(as_tensor([[u[0].dx(0), u[0].dx(1), 0. ], [u[1].dx(0), u[1].dx(1), 0.], [0
                                                    ., 0., u[0]/r]]))

  # Return forms resulting from the discretization of the affine expansion of the problem
                                                    operators.
  def assemble_operator(self, term):
    u = self.u
    v = self.v
    dx = self.dx
    ds = self.ds
    T_0 = self.T_0
    T = self.hearth_problem_thermal._solution
    r = self.x0
    d_bottom = ds(2) + ds(3) + ds(4) + ds(5) + ds(6)
    d_out = ds(7) + ds(8) + ds(9) + ds(10) + ds(11)
    d_sf = ds(13) + ds(14) + ds(15) + ds(16) + ds(17) + ds(18) + ds(19) + ds(20)
    if term == "a":
      a0 = (u[0].dx(0)+u[1].dx(1)+u[0]/r)*(v[0].dx(0)+v[1].dx(1)+v[0]/r)*r*dx
      a1 = (u[0].dx(0)*v[0].dx(0) + u[1].dx(1)*v[1].dx(1) + (u[0]*v[0])/(r)**2 + 0.5*(u[0
                                                  ].dx(1)+u[1].dx(0))*(v[0].dx(1)+v[1].dx(0)))
                                                   * r * dx
      return (a0, a1,)
    elif term == "f":
      f0 = (T-T_0) * (v[0].dx(0) + v[1].dx(1) + v[0]/r) * r * dx
      return (f0,)
    elif term == "inner_product":
      x0 = inner(u,v) * r * dx + inner(self.strain(u),self.strain(v)) * r * dx
      return (x0,)
    elif term == "dirichlet_bc":
      bc0 = [DirichletBC(self.V.sub(0), Constant(0.), self.boundaries, 1),
        DirichletBC(self.V.sub(1), Constant(0.), self.boundaries, 2),
        DirichletBC(self.V.sub(1), Constant(0.), self.boundaries, 3),
        DirichletBC(self.V.sub(1), Constant(0.), self.boundaries, 4),
        DirichletBC(self.V.sub(1), Constant(0.), self.boundaries, 5),
        DirichletBC(self.V.sub(1), Constant(0.), self.boundaries, 6),]
      return (bc0,)
    else:
      raise ValueError("Invalid term for assemble_operator().")
```

We define an instant of *HearthThermoMechanical* class and perform reduction with POD-Galerkin method.

```
# 3C. Allocate an object of the HearthThermoMechanical class
hearth_problem_thermo_mechanical = HearthThermoMechanical(VM, subdomains=subdomains,
                                             boundaries=boundaries, mesh=mesh,
                                             hearth_problem_thermal=
                                             hearth_problem_thermal, ref_temperature=T_0)
#specify and set range of each parameter
mu_range = [(2.3,2.4), (0.5,0.7), (0.5,0.7), (0.4,0.6), (3.05,3.35), (13.5,14.5), (8.3,8.
                                             7), (8.8,9.2), (9.8,10.2), (10.4,10.8), (9.8
                                             ,10.2), (1.9e9,2.5e9), (1.2e9,1.8e9), (0.8e-
                                             6,1.2e-6)]
hearth_problem_thermo_mechanical.set_mu_range(mu_range)

# 4C. Prepare reduction with a POD-Galerkin method
#NOTE : truth_problem attribute is FEM problem and reduced_problem is RB problem
pod_galerkin_method_thermo_mechanical = PODGalerkin(hearth_problem_thermo_mechanical)
pod_galerkin_method_thermo_mechanical.set_Nmax(100) #Maximum size of reduced basis space
pod_galerkin_method_thermo_mechanical.set_tolerance(1e-4) #Maximum eigenvalue tolerance
```

Next, we perform the offline phase.

```
# 5C. Perform the offline phase
pod_galerkin_method_thermo_mechanical.initialize_training_set(1000) #Initialize training
                                             set with specified number of training
                                             parameters
reduced_hearth_problem_thermo_mechanical = pod_galerkin_method_thermo_mechanical.offline
                                             () #Perform offline phase
```

Before performing the reduction of thermal problem, we perform the truth solution computation related operations. This is due to the reason that, we want to use temperature computed by FEM solution for the truth solution of displacement. We compute the truth solution at few parameter, truth solution for error analysis and measure time taken for computing the truth solution.

```
# 6C. Perform a truth solve : Reference domain
online_mu = ( 2.365, 0.6, 0.6, 0.5, 3.2, 14.10, 8.50, 9.2, 9.9, 10.6, 10., lame1, lame2,
                                             1e-6)
pod_galerkin_method_thermo_mechanical.truth_problem.set_mu(online_mu)
u_ref = pod_galerkin_method_thermo_mechanical.truth_problem.solve()
pod_galerkin_method_thermo_mechanical.truth_problem.export_solution(filename="
                                             reference_domain_fem")

# 6C. Perform a truth solve : Parametrized domain
online_mu = ( 2.365, 0.6, 0.6, 0.45, 3.2, 14.10, 8.30, 9.2, 9.9, 10.6, 10., lame1, lame2,
                                              1e-6)
pod_galerkin_method_thermo_mechanical.truth_problem.set_mu(online_mu)
u_par = pod_galerkin_method_thermo_mechanical.truth_problem.solve()
pod_galerkin_method_thermo_mechanical.truth_problem.export_solution(filename="
                                             parametric_domain_fem")

# 7C1. Perform an error analysis - Compute truth solutions
pod_galerkin_method_thermo_mechanical.initialize_testing_set(50) #Initialize error
                                             analysis set with specified number of
                                             parameters
testing_set_error_analysis = pod_galerkin_method_thermo_mechanical.testing_set

truth_solution_thermo_mechanical = list()

# Iteration over error analysis parameters for measuring time taken for FEM solution
for (mu_index, mu_test) in enumerate(testing_set_error_analysis):
  print(TextLine(str(mu_index), fill="#"))
  pod_galerkin_method_thermo_mechanical.truth_problem.set_mu(mu_test) #Set parameter
  truth_solution_thermo_mechanical.append(pod_galerkin_method_thermo_mechanical.
                                             truth_problem.solve()) #Solve and store FEM
                                             solution

# 8C1. Perform a speedup analysis - Compute time for truth solutions
pod_galerkin_method_thermo_mechanical.initialize_testing_set(50) #Initialize truth
                                             solution with specified number of parameters
testing_set_speedup_analysis = pod_galerkin_method_thermo_mechanical.testing_set

pod_galerkin_method_thermo_mechanical._patch_truth_solve(True) #To disable cache reading

truth_timer = Timer("parallel") #Timer for computation of FEM solution
time_thermo_mechanical_truth = np.empty(len(testing_set_speedup_analysis)) #Storage of
                                             time taken for solving FEM equation. It is a
                                              vector of size of number of speedup
                                             analysis parameters

# Iteration over speedup analysis parameters for measuring time taken for RB solution
for (mu_index, mu_test) in enumerate(testing_set_speedup_analysis):
  print(TextLine(str(mu_index), fill="#"))
  pod_galerkin_method_thermo_mechanical.truth_problem.set_mu(mu_test) #Set the parameter
  truth_timer.start()
  pod_galerkin_method_thermo_mechanical.truth_problem.solve() #Solve the RB problem
  truth_time_thermo_mechanical = truth_timer.stop()
  print("Truth time thermomechanical : ",truth_time_thermo_mechanical)
  time_thermo_mechanical_truth[mu_index] = truth_time_thermo_mechanical #Save time taken
                                             for reduced basis solution

pod_galerkin_method_thermo_mechanical._undo_patch_truth_solve(True) #To disable cache
                                             reading

np.save("time_thermo_mechanical_truth",time_thermo_mechanical_truth) #Save numpy array of
                                              time taken for RB solution
```

Now, since the operations related to truth solution are performed, we can reduce the thermal system. The reduced solution of temperature field is used for computing reduced solution at few paraneters. Also we compute reduced solution for error analysis and time taken for computing reduced solution.

```
#6C. Perform an online solve : Reference domain
online_mu_reference = ( 2.365, 0.6, 0.6, 0.5, 3.2, 14.10, 8.50, 9.2, 9.9, 10.6, 10.,
                                                  lame1, lame2, 1e-6)
online_mu = online_mu_reference
pod_galerkin_method_thermo_mechanical.reduced_problem.set_mu(online_mu)
u_rb = pod_galerkin_method_thermo_mechanical.reduced_problem.solve()
pod_galerkin_method_thermo_mechanical.reduced_problem.export_solution(filename="
                                                  reference_domain_thermomechanical_rb")
u_rb = pod_galerkin_method_thermo_mechanical.reduced_problem.basis_functions * u_rb
pod_galerkin_method_thermo_mechanical.truth_problem.mesh_motion.move_mesh()
File("HearthThermoMechanical/reference_domain_thermomechanical_spatial_error.pvd") <<
                                                  project(u_ref-u_rb,VM)
pod_galerkin_method_thermo_mechanical.truth_problem.mesh_motion.reset_reference()

# 6C. Perform an online solve : Parametrized domain
online_mu_parametrized = ( 2.365, 0.6, 0.6, 0.45, 3.2, 14.10, 8.30, 9.2, 9.9, 10.6, 10.,
                                                  lame1, lame2, 1e-6)
online_mu = online_mu_parametrized
pod_galerkin_method_thermo_mechanical.reduced_problem.set_mu(online_mu)
u_rb = pod_galerkin_method_thermo_mechanical.reduced_problem.solve()
pod_galerkin_method_thermo_mechanical.reduced_problem.export_solution(filename="
                                                  parametric_domain_thermomechanical_rb")
u_rb = pod_galerkin_method_thermo_mechanical.reduced_problem.basis_functions * u_rb
pod_galerkin_method_thermo_mechanical.truth_problem.mesh_motion.move_mesh()
File("HearthThermoMechanical/parametric_domain_thermomechanical_spatial_error.pvd") <<
                                                  project(u_par-u_rb,VM)
pod_galerkin_method_thermo_mechanical.truth_problem.mesh_motion.reset_reference()

# 7C2. Perform an error analysis - Compute reduced basis solution
dx = Measure("dx")(subdomain_data=subdomains) #Volume measure
r = Expression("x[0]", element=VM.sub(0).ufl_element()) #

max_basis_function = reduced_hearth_problem_thermo_mechanical.N # Size of reduced basis
                                                  space
error_thermo_mechanical = np.empty([max_basis_function,len(testing_set_error_analysis)])
                                                  # Numpy array of size of reduced basis space
                                                   \times number of error analysis parameters
                                                  for storing error
# Iteration over error analysis parameters for measuring time taken for RB solution
for basis_size in range(1,max_basis_function+1):
  for (mu_index, mu_test) in enumerate(testing_set_error_analysis):
    print(TextLine(str(mu_index), fill="#"))
    pod_galerkin_method_thermo_mechanical.reduced_problem.set_mu(mu_test) #Set parameter
    rb_dofs = pod_galerkin_method_thermo_mechanical.reduced_problem.solve(basis_size) #
                                                  Compute reduced basis degrees of freddom
    rb_solution = reduced_hearth_problem_thermo_mechanical.basis_functions[:basis_size] *
                                                   rb_dofs #RB solution projected back to FEM
                                                  space
    # Absolute and relative error measurement
    absolute_error = assemble(inner(truth_solution_thermo_mechanical[mu_index] -
                                                  rb_solution,truth_solution_thermo_mechanical
                                                  [mu_index] - rb_solution) * r * dx + inner(
                                                  hearth_problem_thermo_mechanical.strain(
                                                  truth_solution_thermo_mechanical[mu_index] -
                                                   rb_solution),
                                                  hearth_problem_thermo_mechanical.strain(
                                                  truth_solution_thermo_mechanical[mu_index] -
                                                   rb_solution)) * r * dx)
    error_thermo_mechanical[basis_size-1,mu_index] = np.sqrt(absolute_error / assemble(
                                                  inner(truth_solution_thermo_mechanical[
                                                  mu_index],truth_solution_thermo_mechanical[
                                                  mu_index]) * r * dx + inner(
                                                  hearth_problem_thermo_mechanical.strain(
                                                  truth_solution_thermo_mechanical[mu_index]),
                                                   hearth_problem_thermo_mechanical.strain(
                                                  truth_solution_thermo_mechanical[mu_index]))
                                                   * r * dx))

np.save("HearthThermoMechanical/error_analysis/error_thermo_mechanical",
                                                  error_thermo_mechanical)

# 8C2. Perform a speedup analysis - Compute time for reduced solutions
pod_galerkin_method_thermo_mechanical._patch_truth_solve(True) #To disable cache reading

reduced_timer = Timer("serial") #Timer for computation of RB solution
time_thermo_mechanical_reduced = np.empty([max_basis_function,len(
                                                  testing_set_speedup_analysis)]) #Storage of
                                                  time taken for solving RB equation. It is a
                                                  matrix of size size of reduced basis space \
                                                  times number of speedup analysis parameters

# Iteration over speedup analysis parameters for measuring time taken for RB solution
for basis_size in range(1,max_basis_function+1):
  for (mu_index, mu_test) in enumerate(testing_set_speedup_analysis):
    print(TextLine(str(mu_index), fill="#"))
    pod_galerkin_method_thermo_mechanical.reduced_problem.set_mu(mu_test) #Set parameter
    reduced_timer.start()
    pod_galerkin_method_thermo_mechanical.reduced_problem.solve(basis_size) #Solve the RB
                                                   problem
    rb_time_thermo_mechanical = reduced_timer.stop()
    print("Reduced time thermomechanical : ",rb_time_thermo_mechanical)
    time_thermo_mechanical_reduced[basis_size-1,mu_index] = rb_time_thermo_mechanical #
                                                  Save time taken for reduced basis solution

pod_galerkin_method_thermo_mechanical._undo_patch_truth_solve(True) # To disable cache
                                                  reading

np.save("time_thermo_mechanical_reduced",time_thermo_mechanical_reduced) #Save numpy
                                                  array of time taken for RB solution
```

6 License

- FEniCS and RBniCS are freely available under the GNU LGPL, version 3.
- Matplotlib only uses BSD compatible code, and its license is based on the PSF license. Non-BSD compatible licenses (e.g., LGPL) are acceptable in matplotlib toolkits.

Accordingly, this code is freely available under the GNU LGPL, version 3.

7 Disclaimer

In downloading this SOFTWARE you are deemed to have read and agreed to the following terms: This SOFTWARE has been designed with an exclusive focus on civil applications. It is not to be used for any illegal, deceptive, misleading or unethical purpose or in any military applications. This includes ANY APPLICATION WHERE THE USE OF THE SOFTWARE MAY RESULT IN DEATH, PERSONAL INJURY OR SE-

References

[1] P. Barral, M. Girfoglio, A. Lengomin, P. Quintela, and N. Shah, "Coupled parameterized reduced order modelling of thermo-mechanical phenomena arising in blast furnace," Jun. 2020, project Deliverable (D5.2), Version 2.0. [Online]. Available: https://doi.org/10.5281/zenodo.3888145

[2] J. Fehr, C. Himpe, S. Rave, and J. Saak, "Sustainable research software hand-over," *Journal of Open Research Software*, vol. 9, 2021. [Online]. Available: http://dx.doi.org/10.5334/jors.307

[3] M. S. Alnæs, J. Blechta, J. Hake, A. Johansson, B. Kehlet, A. Logg, C. Richardson, J. Ring, M. E. Rognes, and G. N. Wells, "The fenics project version 1.5," *Archive of Numerical Software*, vol. 3, no. 100, 2015.

[4] A. Logg, K.-A. Mardal, G. N. Wells *et al.*, *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012.

[5] A. Logg, G. N. Wells, and J. Hake, *DOLFIN: a C++/Python Finite Element Library*. Springer, 2012, ch. 10.

[6] A. Logg and G. N. Wells, "Dolfin: Automated finite element computing," *ACM Transactions on Mathematical Software*, vol. 37, no. 2, 2010.

[7] J. S. Hesthaven, G. Rozza, and B. Stamm, *Certified Reduced Basis Methods for Parametrized Partial Differential Equations*, ser. SpringerBriefs in Mathematics. Springer International Publishing, 2015.

[8] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[9] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2

ROMSOC

The ROMSOC project

June 23, 2021

ROMSOC-D5.3-0.1

Horizon 2020