OOFEM Seminar

Contents

- Introduction to new theory manual
- OOFEM python bindings

Author: Borek Patzak

Introduction to New Theory manual

Idea is to provide an extensible system of individual documents, maintained by independent authors, that are put together to establish the manual.

The top-level structure is created in Sphinx [Overview — Sphinx documentation (sphinx-doc.org)].

Sphinx is a tool to generate documentation in different formats, including HTML, LaTeX, etc. It uses reStructuredText as a markup language.

The sources are located in doc/theorymanual2

Introduction to New Theory manual

Top level Sphinx document (index.rst) defines the top-level sections

```
Welcome to OOFEM Theory manual!
==================================================
.. toctree::
   :maxdepth: 2
   :caption: Table of Contents

   introduction
   general
   problems
   elements
   materials

* :download:`How to contribute <docs/Introduction/Adding_documentation/Readme.pdf>`
```

Introduction to New Theory manual

- In individual sections (subsections), the individual documentation entries to be provided.
- The documentation of individual entry is supposed to be a
  - Latex project, consisting at least of a latex file containing the documentation source and Makefile which default target builds pdf version of entry.
  - Sphinx project.
- Requirements
  - The entry source directory should contain Makefile, which default target should generate documentation in PDF.
  - The individual entries should follow the same template, which is provided for LaTex documents in docs/templates.

OOFEM Python3 API

- Python bindings expose to Python OOFEM functions & classes, it also allows to implement new OOFEM classes in Python
- Based on PyBind11 library (exposes C++ functions to Python3)
- Majority of OOFEM clasess & functions exposed now (FloatArray, FloatMatrix, Domain, Problem, Dof, DofManager, Element, BoundaryCondition, ...)
- Provides automatic convertors between C++ and Python types (oofem::FloatArray, std::vector, std::string ↔ list or tuple)

Typical use cases

- Programmatic generation of input
- Scripting the solver to postprocess the results
- Prototypic implementation of custom classes
- Implementing complex, documented simulation workflows (Jupyter notebook integration)

Requires to build oofem with USE_PYBIND_BINDINGS=ON

Python API - build

- Cmake configuration
  - USE_PYBIND_BINDINGS="ON"
  - Suggested USE_PYTHON_EXTENSION="ON" (uses Python.h library for executing Python3 functions from C++)
- Compilation and linking builds a shared library *oofempy.so* (*oofempy.dll*)
  - Can be imported as a module to the Python script

Python API – minimum example

- Vector operations

```
[user@zeus bindings/python]$ python3
```

```
>>> import oofempy

>>> a = oofempy.FloatArray((1.0, 2.0, 3.0))
>>> type(a)
<class 'oofempy.FloatArray'>

>>> a
<oofempy.FloatArray: {1, 2, 3, }>

>>> a.printYourself()
<oofempy.FloatArray: {1, 2, 3, }>

>>> a.computeNorm()
3.7416573

>>> b=2*a #a new vector
>>> a+b
<oofempy.FloatArray: {3, 6, 9, }>
```

Python API – steering the model

- Running a patch test, needs access to module *oofempy.so (oofempy.dll)*

```
o/tests/sm$ python3
Python 3.6.9 (default, Jan 26 2021, 15:33:00)
Type "help", "copyright", "credits" or "license" for more information.
>>> import oofempy
>>> dr=oofempy.OOFEMTXTDataReader("patch010.in")
>>> problem=oofempy.InstanciateProblem(dr, oofempy.problemMode.processor, False, None, False)
>>> problem.init()
>>> problem.solveYourself()
Computing initial guess
StaticStructural :: solveYourselfAt - Solving step 1, metastep 1, (neq = 3)
NRSolver: Iteration ForceError
----------------------------------------------------------------------------
NRSolver: 0      D_u:  0.000e+00
Checking rules...
EngngModel info: user time consumed by solution step 1: 0.00s
>>> problem.terminateAnalysis()
ANALYSIS FINISHED
Real time consumed: 000h:00m:45s
User time consumed: 000h:00m:00s
```

Python API – creating the model programmatically

```
import sys
sys.path.append("..")
import oofempy
import util # some utility functions
problem = oofempy.linearStatic(nSteps=1, outFile="test2.out") # engngModel
domain = oofempy.domain(1, 1, problem, oofempy.domainType._2dBeamMode, tstep_all=True, dofman_all=True,
element_all=True) # domain aka mesh
problem.setDomain(1, domain, True) # associate domain to the problem

# load time function
ltf1 = oofempy.peakFunction(1, domain, t=1, f_t=1)
ltfs = (ltf1, )
# boundary conditions
# loadTimeFunction parameter can be specified as int value or as LoadTimeFunction itself (valid for all objects
with giveNumber() method)
bc1   = oofempy.boundaryCondition(    1, domain, loadTimeFunction=1,    prescribedValue=0.0)
nLoad = oofempy.nodalLoad(            2, domain, loadTimeFunction=1,    components=(-18.,24.,0.))
bcs = (bc1, nLoad)


# continues on the next slide
```

Based on bindings/python/tests/test_2.py

Python API – creating the model programmatically

```
# nodes
# if one value is passed as parameter where oofem expects array of values, it must be passed as tuple or list
(see load in n4)
n1 = oofempy.node(1, domain, coords=(0.,  0., 0. ), bc=(1,1,1))
n2 = oofempy.node(2, domain, coords=(2.4, 0., 0. ), bc=(0,0,0), load = (nLoad,))
nodes = (n1, n2)
# material and cross section
mat = oofempy.isoLE(1, domain, d=1., E=30.e6, n=0.2, tAlpha=1.2e-5)
cs  = oofempy.simpleCS(1, domain, area=0.162, Iy=0.0039366, beamShearCoeff=1.e18, thick=0.54)
# elements
e1 = oofempy.beam2d(1, domain, nodes=(1,n2),  mat=1,   crossSect=1)
elems = (e1, )
# add eveything to domain
util.setupDomain(domain, nodes, elems, (cs,), (mat,),  bcs, (), ltfs, ())
print("\nSolving problem")

problem.checkProblemConsistency()
problem.init()
problem.postInitialize()
problem.setRenumberFlag()
problem.solveYourself()
problem.terminateAnalysis()
```

Based on bindings/python/tests/test_2.py

Python API – post-processing results

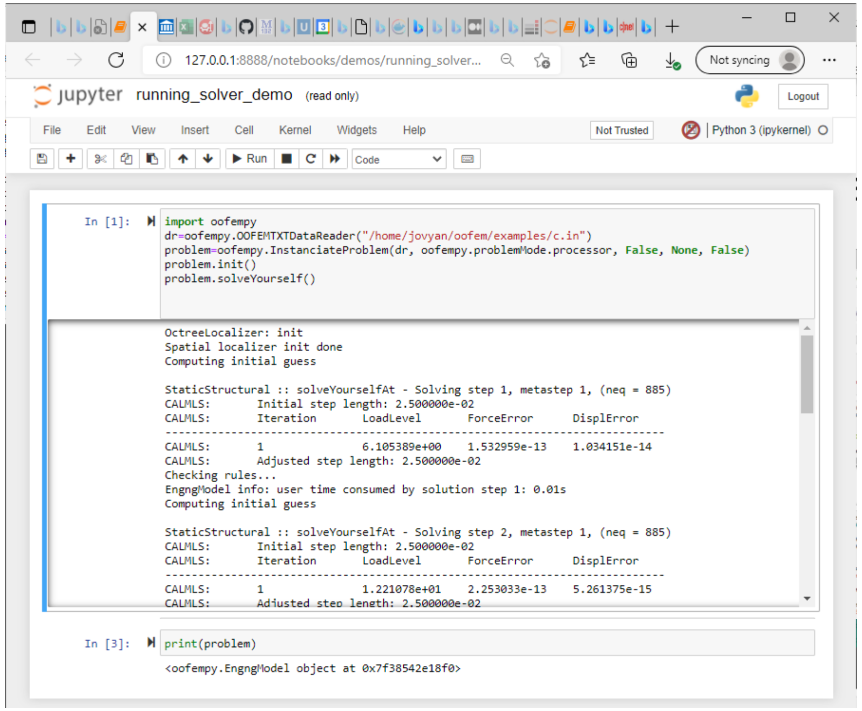
- Convenient access using vtkmemory export module

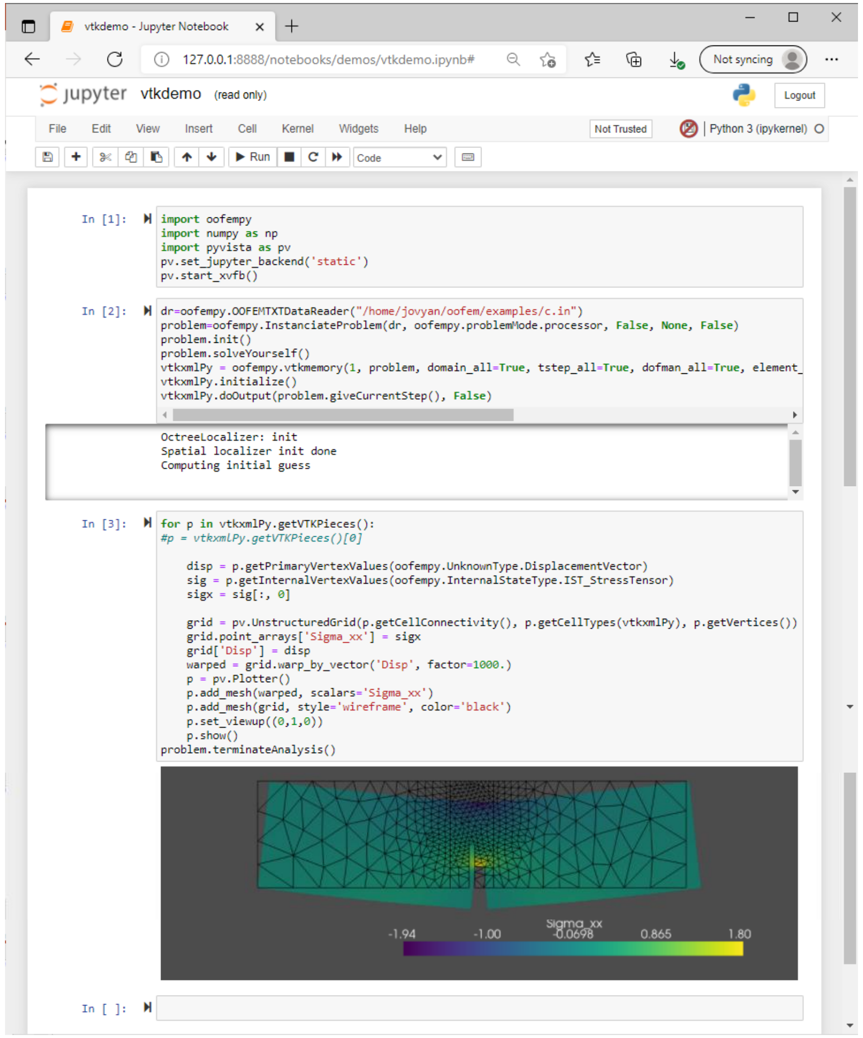
```
[user@zeus bindings/python/tests]$ python3
```

```
import oofempy
...
# create export module to access variables from python
vtkPy = oofempy.vtkmemory(1, problem, domain_all=True, tstep_all=True, dofman_all=True, element_all=True,
           vars=(56,37), primvars=(6,), cellvars = (47,103), stype=1, pythonExport=1)

...

for p in vtkPy.getVTKPieces():
    print ("Piece:", p)
    print("Vertices:", p.getVertices())
    print("Cells:", p.getCellConnectivity())
    print("CellTypes:", p.getCellTypes(vtkPy))
    temperature = p.getPrimaryVertexValues(oofempy.UnknownType.Temperature);
    print ("Temperature:", temperature)

...
```

bindings/python/tests/test_5.py

Jupyter notebook integration for documented workflows



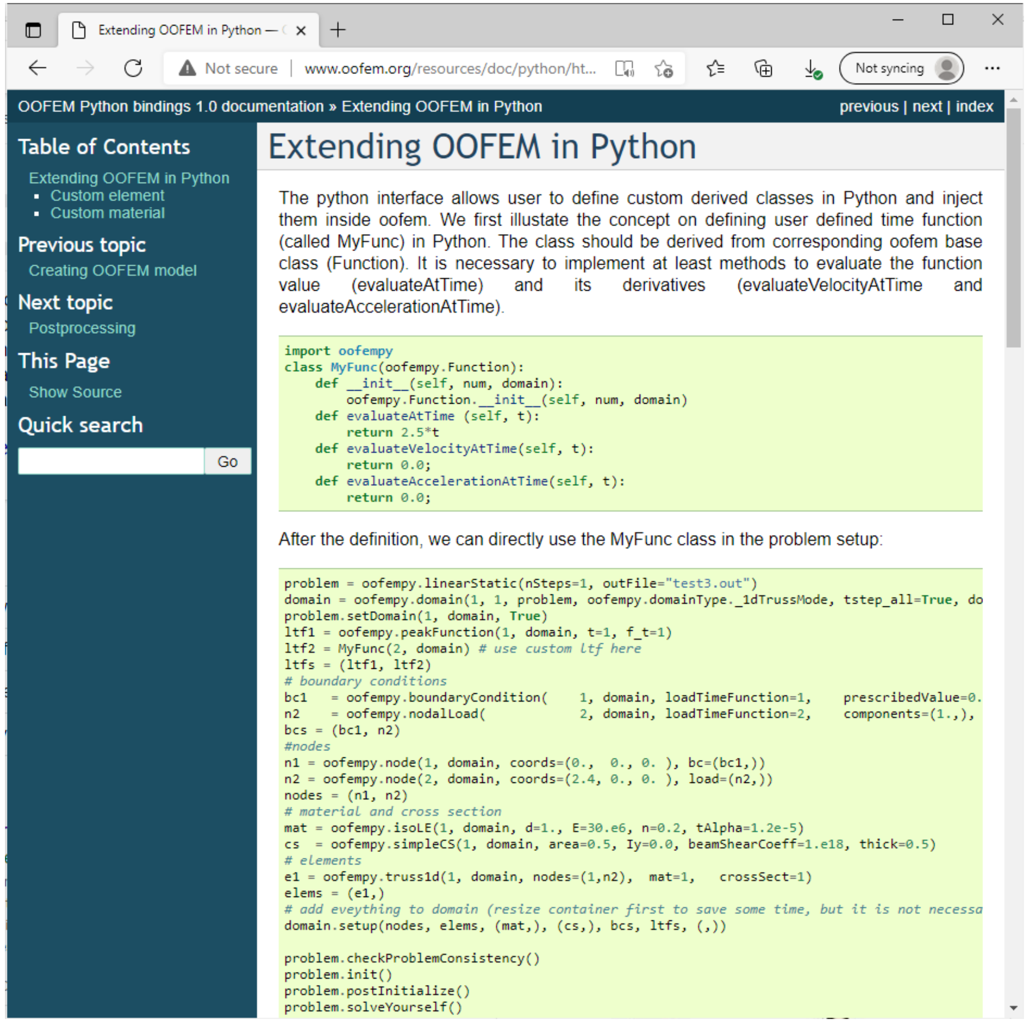


Note: oofem-jupyter-notebook docker image comes with ready to use preinstalled jupyter notebook, python and oofem

See [oofem/docker-stacks: Collection of ready-to-use and build-automation Docker images for OOFEM (github.com)]

Python API – additional features

- Please follow [Python bindings documentation](#) with more examples
  - Implementing custom material model in Python
  - Implementing custom element in Python



OOFEM Python bindings 1.0 documentation » Extending OOFEM in Python

previous | next | index

Table of Contents

Previous topic

Creating OOFEM model

Next topic

Postprocessing

This Page

Show Source

Quick search

Go

Extending OOFEM in Python

The python interface allows user to define custom derived classes in Python and inject them inside oofem. We first illustate the concept on defining user defined time function (called MyFunc) in Python. The class should be derived from corresponding oofem base class (Function). It is necessary to implement at least methods to evaluate the function value (evaluateAtTime) and its derivatives (evaluateVelocityAtTime and evaluateAccelerationAtTime).

```
import oofempy
class MyFunc(oofempy.Function):
    def __init__(self, num, domain):
        oofempy.Function.__init__(self, num, domain)
    def evaluateAtTime (self, t):
        return 2.5*t
    def evaluateVelocityAtTime(self, t):
        return 0.0;
    def evaluateAccelerationAtTime(self, t):
        return 0.0;
```

After the definition, we can directly use the MyFunc class in the problem setup:

```
problem = oofempy.linearStatic(nSteps=1, outFile="test3.out")
domain = oofempy.domain(1, 1, problem, oofempy.domainType._1dTrussMode, tstep_all=True, do
problem.setDomain(1, domain, True)
ltf1 = oofempy.peakFunction(1, domain, t=1, f_t=1)
ltf2 = MyFunc(2, domain) # use custom ltf here
ltfs = (ltf1, ltf2)
# boundary conditions
bc1   = oofempy.boundaryCondition(    1, domain, loadTimeFunction=1,    prescribedValue=0.
n2    = oofempy.nodalLoad(            2, domain, loadTimeFunction=2,    components=(1.,),
bcs = (bc1, n2)
#nodes
n1 = oofempy.node(1, domain, coords=(0.,  0., 0. ), bc=(bc1,))
n2 = oofempy.node(2, domain, coords=(2.4, 0., 0. ), load=(n2,))
nodes = (n1, n2)
# material and cross section
mat = oofempy.isoLE(1, domain, d=1., E=30.e6, n=0.2, tAlpha=1.2e-5)
cs  = oofempy.simpleCS(1, domain, area=0.5, Iy=0.0, beamShearCoeff=1.e18, thick=0.5)
# elements
e1 = oofempy.truss1d(1, domain, nodes=(1,n2),  mat=1,   crossSect=1)
elems = (e1,)
# add eveything to domain (resize container first to save some time, but it is not necessa
domain.setup(nodes, elems, (mat,), (cs,), bcs, ltfs, (,))

problem.checkProblemConsistency()
problem.init()
problem.postInitialize()
problem.solveYourself()
```