FEniCSx and MFEM libraries comparison

January 24, 2025

Abstract

This work provides a partial comparison of two libraries: FEniCSx and MFEM . The evaluation focuses on various aspects such as the problem encoding complexity, available documentation, computational performance, input/output (I/O) capabilities, support for parallelism, dependency constraints imposed by the various software tools used by these libraries, ... By examining these factors, we aim to offer some insights into the strengths and weaknesses of each library, enlightening researchers and engineers in selecting a suitable tool for their specific computational needs. The analysis includes practical coding examples, performance benchmarks, and a detailed discussion on the ease of use and scalability, providing a partial assessment of each library's capabilities and usability. This document can also be used as a light tutorial to start encoding in one library knowing the other.

1 Introduction

FEniCSx and MFEM are both open-source libraries used for solving partial differential equations (PDEs) using finite element methods (FEM). The target audience for this document is diverse: it ranges from newcomers to both codes (but familiar with the principle of the finite element and with the formulation of a physical problem in weak form) who are looking for insights to help them choose one or the other, to users who are tempted to switch from one to the other, to intermediate users who are looking at some details in one library with a view to possibly using the approach of the other library,...

Written primarily in Python with core components in C++, FEniCSx focuses on providing a high-level, user-friendly interface to FEM. It is designed to be accessible to users with minimal programming experience, without compromising on computational performance and scalability. It is part of the FEniCS project, with significant contributions from the scientific computing community.

MFEM is primarily written in C++ with interfaces for Python. It focuses on being a high-performance, scalable finite element discretization library. MFEM requires more familiarity with programming, particularly C++, for advanced use. It is developed by the Center for Applied Scientific Computing at Lawrence Livermore National Laboratory.

As free libraries, they can be downloaded from `https://fenicsproject.org/` for FEniCSx(version 0.8.0 in this work) and from `https://mfem.org` for MFEM (version 4.7.0 in this work). Both offers a wide range of possibilities for solving various physical problems in a fairly general way. To get an idea of how problems are formulated in each library, a series of examples will be used to compare them. In this work only parallel example will be used. These examples are structured so that the following topics, common to many simulations using FEM libraries, can be compared:

1. Initialize the library:
    - Put in place MPI context.
    - Parse command-line options.
2. Mesh construction:
    - Read or create the mesh.
    - Distribute the mesh in parallel MPI context
    - Eventually refine the mesh.
3. Define a finite element space using the mesh and a given polynomial order.
4. Define material properties:
    - Potentially different for some sub-domain.
    - Not forcefully constant during the simulation.

5. Boundary Conditions and Load:
    - Define essential boundary conditions (Dirichlet)
    - Set up the right-hand side or the residual (Neuman,...)
6. Linear or non-linear problem creation:
    - Assemble various matrices and vectors.
    - In non-linear set up initial solution.
7. Solve the system:
    - In general pass a linear system to an external library dedicated to solve this kind of system in parallel.
    - In non-linear loop to update material properties and solve again the new linear system up to convergence (null residual)
8. Output:
    - Save the different field in files that can be read by visualization tools

The following section comments on each of these topics. The implementation of the examples in the section 2 is not fully provided in this document. Only parts of the code are used in support of the main explanations. The full implementation can be found in the following repository: `https://github.com/SalzmanA/fem-libraries.git`

2 Examples

2.1 Mechanic

2.1.1 Asymmetric traction/compression elasto-damaged constitutive law

This test case is designed to evaluate the behavior of a square sample (1mx1m) under traction or compression (Dirichlet boundary condition) and volume loading ($\mathbf{f}$) , considering an elasto-damaged constitutive law that accounts

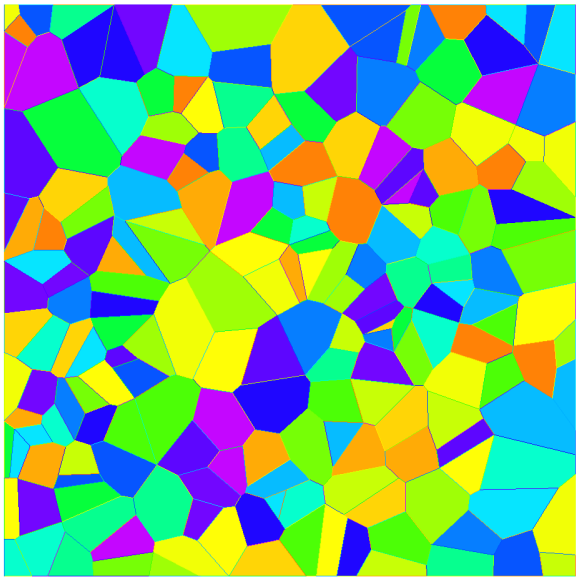


Figure 1: 2D Mesh created by Neper (color represent different physical properties tag).

for asymmetric traction/compression responses. The model employs a polycrystalline structure generated by Neper library[1, 4], introducing variability in material properties to mimic real-world conditions :

- Mesh Generation: The mesh for the square domain Ω is obtained using Neper and saved in gmsh format(Figure 1).
- Damage Field: An arbitrary damage field is created, introducing total stiffness losses at certain grain interfaces (Figure 2).
- Grain Properties: Grains are modeled as isotropic materials, with no material orientation inside a grain.
- Young's Modulus Variation: Each grain has a different Young's modulus, with random values in the range [5e6Pa, 1e8Pa] (Figure 3).

Spatial discretization will use order 1 Lagrange polynomial. The constitutive law in 2D is based on the following potential:

$$\psi(\epsilon, d) = \frac{\lambda}{2}(1 - \alpha d)\, tr(\epsilon)^2 + \mu \sum_{i=1}^{2} (1 - \alpha_i d)\Lambda_i{}^2 \tag{1}$$

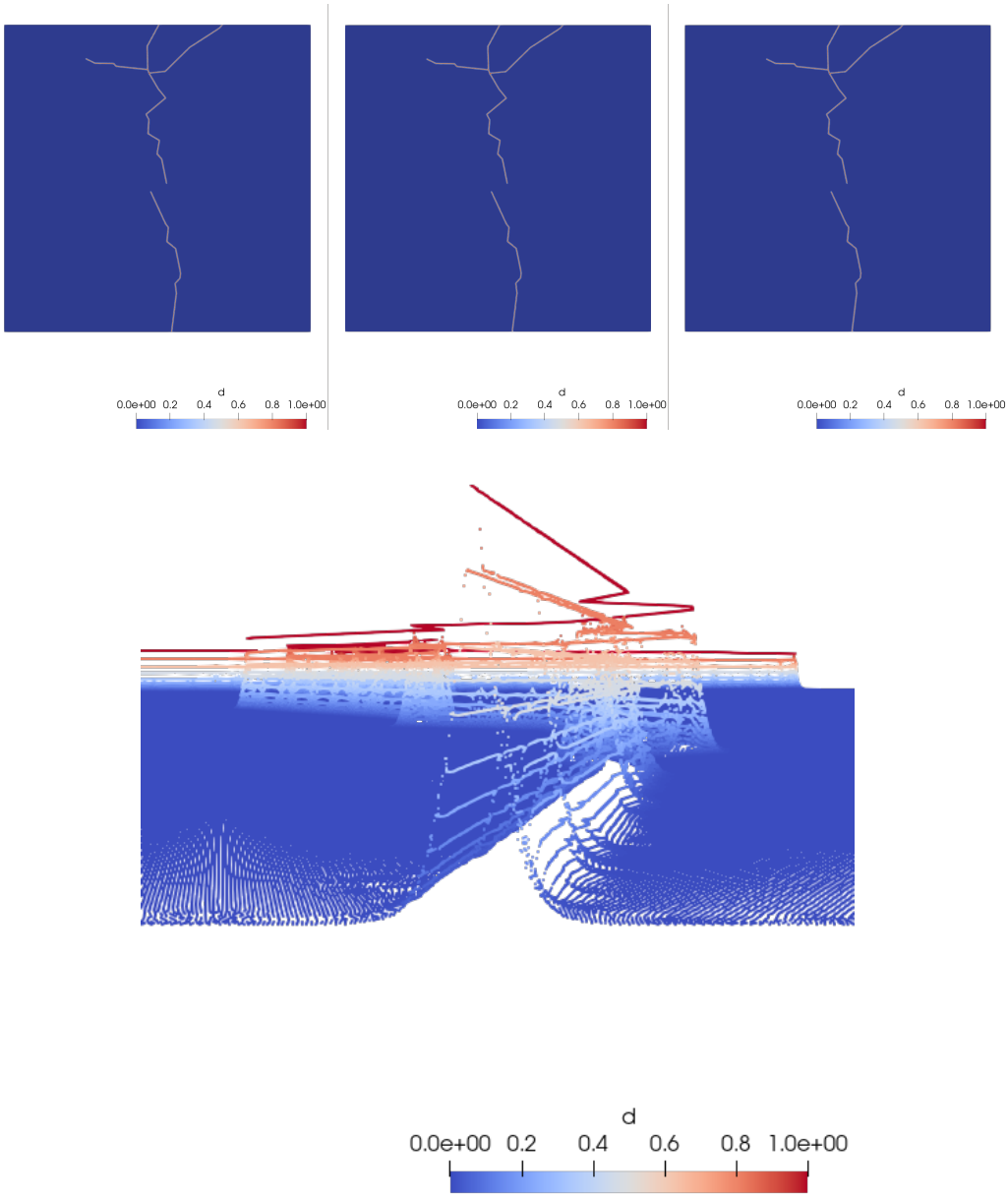


Figure 2: Imposed damage computed for a refined mesh, in MFEM (left top) FEniCSx_C++(center top) and in FEniCSx_py(right top). Bottom image presents an elevation of the damage field (z=d) with value at nodes using a lateral view angle. It aims to shows the chosen arbitrary damage profile imposed around border of grain fixed to 1 .

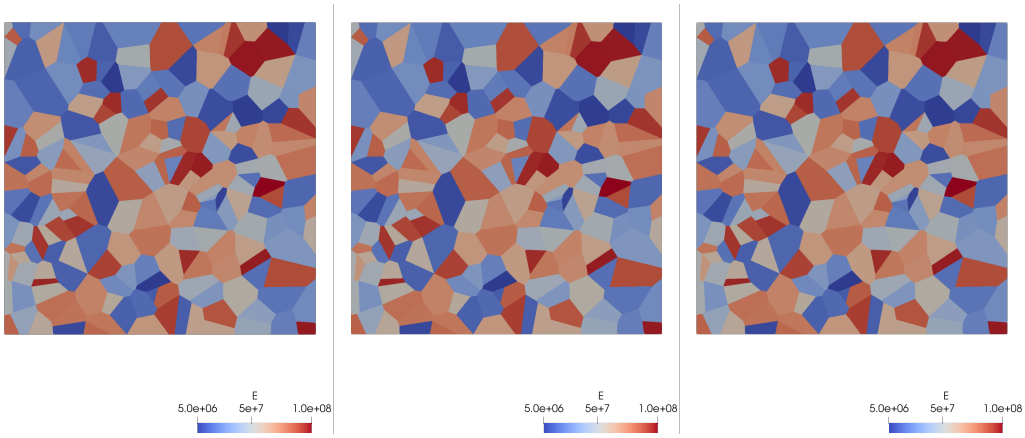


Figure 3: Grain Young's Modulus values in Pa after creation in MFEM (left), FEniCSx_C++(center) and FEniCSx_py(right).

where:

- λ and μ are the Lamé elasticity coefficients.
- Λ_i are the eigenvalues of the strain tensor ϵ.
- α_i and α are coefficients introduced to take into account an asymmetric behavior in traction/compression:
  - $\alpha_i = 0 \; if \; \Lambda_i < 0$
  - $\alpha_i = 1 \; if \; \Lambda_i \geqslant 0$
  - $\alpha = 0 \; if \; tr(\epsilon) < 0$
  - $\alpha = 1 \; if \; tr(\epsilon) \geqslant 0$

The Cauchy stress tensor σ is obtained from the free energy ψ as follows:

$$\sigma = \frac{\partial \psi}{\partial \epsilon} \tag{2}$$

The variationnal formulation to encode in the libraries considering non-linear behavior is thus the residual computation. If $\mathcal{M}$ is the continuous space of the problem defined on Ω and compatible with the Dirichlet boundary conditions ($\mathcal{M}_0$ being $\mathcal{M}$ with null Dirichlet boundary conditions) the residual is given by:

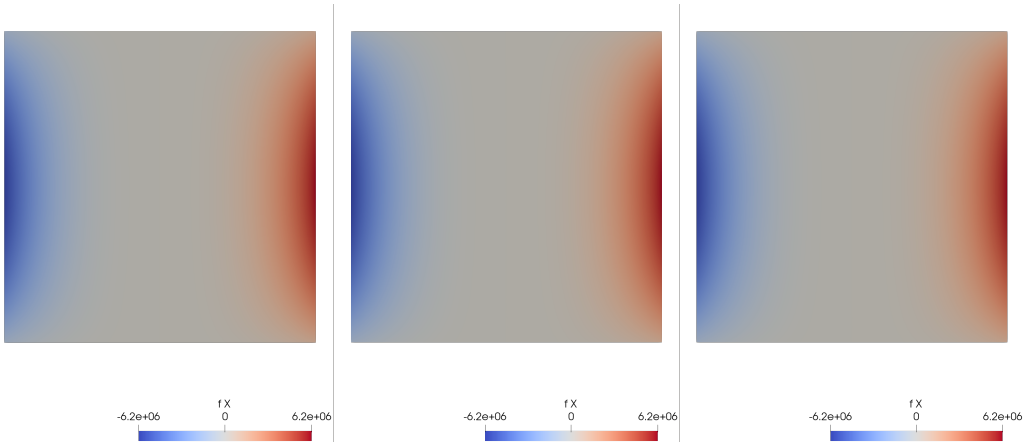


Figure 4: Arbitrary imposed interior loading interpolated on refined mesh by MFEM (left), FEniCSx_C++(center) and by FEniCSx_py(right). It is the X component value of the 2D vectorial forces $\mathbf{f}$ that is shown.

$$F(\mathbf{u},\mathbf{v})_\Omega = \int_\Omega \sigma(\mathbf{u}) : \epsilon(\mathbf{v})\, dV - \int_\Omega \mathbf{f}.\mathbf{v} dV \tag{3}$$

with:

- $\mathbf{u} \in \mathcal{M}$ is the displacement at current non-linear iteration
- $\mathbf{v} \in \mathcal{M}_0$ is the test function
- $\mathbf{f}$ is an arbitrary imposed Neumann load (in Newton), shown in the figure 4 and given by the equation (4).

$$\mathbf{f} = \begin{cases} -100000(1600 - (y-0.5)^2 - 500) * (x-0.5)^3 \\ 0. \end{cases} \tag{4}$$

The Jacobian matrix needed in the non-linear resolution is given by differentiating (3).

Finally, small strains and displacements are assumed which implies the following strain tensor/displacement relation:

$$\epsilon = \frac{1}{2}\left(\nabla\mathbf{u} + (\nabla\mathbf{u})^t\right) \text{ on } \Omega \tag{5}$$

FEniCSx		MFEM	
C++	python	C++	python
C++20 C++23 add-on	3.10	C++11	?

Table 1: Required standard/version to compile/run

The first i_1 and the second i_2 invariant of ϵ are given by the following formulas:

$$\begin{aligned} i_1 &= tr(\epsilon) \\ i_2 &= \tfrac{1}{2}\left(tr(\epsilon^2) - i_1^2\right) \end{aligned} \tag{6}$$

3 API and documentation

As already mentionned the FEniCSx language is primarily Python with core components in C++. In fact, the C++ sources are divided into the following libraries:

- dolfinx: Problem solving environement
- basix: FEniCSx finite element basis evaluation library
- ufl: UFL - Unified Form Language
- ffcx: Form compiler for finite element forms

The Python API follows this division and provides modules for each library. It also gives some extra features not (yet) available in C++ like gmshio.

For MFEM it is the other way round, the language is primarily C++ with interfaces to Python. The C++ sources are packaged in a monolithic library called MFEM, and the Python warping is provided by PyMFEM (has not yet been tested in this study). The C++ API is encapsulated in the unique namespace "mfem".

The table 1 gives the requirements for C++ compilers and python interpreters, and the following setting is used in this document for FEniCSx C++ code:

```
using namespace dolfinx;
typedef PetscScalar scalar;
typedef scalar_value_type_t<scalar> scalar_dolf;
```

Listing 1

and MFEM code:

```
using namespace mfem;
```

Listing 2

In terms of documentation and community, FEniCSx has a strong forum (`https://fenicsproject.discourse.group`) inherited from the FEniCS project. All APIs are documented at `https://docs.fenicsproject.org`. An "official" tutorial is available at `https://jsdokken.com/dolfinx-tutorial/` and some interesting contributions can be found on "unofficial" sites such as J. Bleyer's (`https://bleyerj.github.io/comet-fenicsx/index.html`) or J.S. Doken's (`https://jsdokken.com/tutorials.html`). From an external point of view, the FEniCSx project is (at the time of writing) in a state of intense development, so the material that can be found on the web can sometimes be a bit confusing (examples based on an old APi or related to FEniCS, site related to unofficial addons more or less up to date with the current version, thread in the forum obsolete due to library evolution, ...). But as development moves fast, so does documentation.

For its part, MFEM makes all the interesting material available on its website (`https://mfem.org`), with a discussion forum within the source github repository: `https://github.com/orgs/mfem/discussions`.

For both libraries, many examples provide a nice tutorial to get into the implementation of a problem. The API documentation provides the remaining ingredient to complete the encoding.

4 Initialize (point 1)

In parallel, the minimum to implement is the initiation of the MPI library, and as FEnicsx and MFEM may use it, in addition, the initiation of PETSC. With FEniCSx C++ API, a logging tool can also be set up at start-up, using the following code:

```
// Init logging backend of dolfinx (loguru)
// tuned by -dolfinx_loglevel <level> option
```

```
dolfinx::init_logging(argc, argv);
// init petsc environment
// imply init distributed environement
PetscInitialize(&argc, &argv, nullptr, nullptr);
std::string log_name = "what you want distinquishing PID";
loguru::set_thread_name(log_name.c_str());
// very verbose
loguru::g_stderr_verbosity =loguru::Verbosity_INFO;
// only warning
loguru::g_stderr_verbosity =loguru::Verbosity_WARNING;
```

Listing 3

With the version used in this study (0.8), FEniCSx uses the logger library loguru. But they switched to the spdlog library in the next release. Here using " PetscInitialize " function (and "PetscFinalize" at the end of the program) make Petsc library call the "MPI_init" function. If Petsc is not to be used, "MPI_init" (or "MPI_Init_thread") must be called directly (and "MPI_finalize" at the end of the program).

The python conterpart corespond to the importation of all used module as follow:

```
from scipy import sparse
from mpi4py import MPI
import numpy as np
from pathlib import Path
from ctypes import CDLL
import sympy as sp
import ufl
from ufl.classes import (Mesh,FunctionSpace,Coefficient,
    Constant,
TrialFunction,TestFunction,FacetNormal)
from ufl import ( sqrt,grad,inner,tr,Identity,dot,dx,ds,
conditional,lt,gt,eq)
import basix
from dolfinx.io import XDMFFile
from dolfinx.io import VTXWriter
from dolfinx import fem
from dolfinx import mesh
from dolfinx import la
from dolfinx import nls
from dolfinx import cpp
from dolfinx.fem import petsc
from dolfinx.nls import petsc
import adios4dolfinx
from petsc4py import PETSc
```

Listing 4

MFEM does not seem to rely on a logger library strategy. Information is output to a stream on demand, using a special printing method or setting a log level with some algorithms. Or, at compile time, the MFEM_DEBUG macro, if set, forces messages to be output from any routine that encodes them. Compared to FEniCSx, the "MPI_init" function must be called in MFEM code using its "mfem::Mpi" class, which encapsulates the MPI library, with the "mfem::Mpi::Init" method (and the "mfem::Mpi::Finalize" at the end of the program). MFEM provides a handy options parser, unlike FEniCSx, which allows you to encode options in the source and modify them at runtime with the parameters passed to the program at startup. In the following code, for example, "newton_rel_tol" is set to 1.e-7 in the source and can be changed at runtime using the "–rel" or "––relative–tolerance" option:

```
Mpi::Init(argc, argv);
const char *petscrc_file = "";
OptionsParser args(argc, argv);
args.AddOption(&petscrc_file, "-petscopts", "--petscopts", "PetscOptions
    file to use.");
args.AddOption(&verbose, "-v", "--verbose", "-nv", "--not-verbose", "Output
    extra informations.");
real_t newton_rel_tol = 1e-7;
real_t newton_abs_tol = 5e-8;
args.AddOption(&newton_rel_tol, "-rel", "--relative-tolerance", "Relative
    tolerance for the Newton solve.");
args.AddOption(&newton_abs_tol, "-abs", "--absolute-tolerance", "Absolute
    tolerance for the Newton solve.");
args.Parse();
if (!args.Good()){
    if (mpi_rank == 0)  args.PrintUsage(cout);
    return 1;
   }
if (mpi_rank == 0)  args.PrintOptions(cout);
```

Listing 5

To use Petsc in MFEM, the program must call "MFEMInitializePetsc" (and "MFEMFinalizePetsc" at the end):

```
MFEMInitializePetsc(NULL, NULL, petscrc_file, NULL);
```

Listing 6

Here a " petscrc_file " file can be provided at runtime to tune Petsc.

5 Mesh(point 2)

5.1 Reading

In most example, MFEM use the following strategy that start from a serial mesh defined in a file:

- Read the file in all process (via "mfem::Mesh" constructor) and create a serial instance of the full mesh. This implies that the mesh is relatively coarse to fit in memory on all processes.
- Create a parallel mesh instance ("mfem::ParMesh") in all process using the serial mesh instance and a partition provided by the user or, by default, computed by METIS (from the full serial mesh dual-graph). After construction, on each process, this parallel mesh only contains the part of the serial mesh related to the partition with the rank id of the process.
- Refine the coarse distributed mesh.

The above scheme works with MFEM (conforming, non conforming, nurbs), Netgen, TrueGrid, VTK, GMSH, NETCDF, Cubit file format.

One constructor of "mfem::ParMesh" class accepts also a stream argument that give the ability to read a parallel mesh with each MPI rank using its own file/stream. It has not been tested but from source file investigation it looks like it works only with parallel MFEM mesh format (i.e. with a set of file/stream with each having specific information related to mesh distribution).

With C++ API, FEniCSx can use XDMF reader ("dolfinx :: io :: XDMFFile ::read_mesh") to load mesh into "dolfinx :: mesh::Mesh<T>" instance. Compare to MFEM above strategy this method can, if XDMF file use HDF5 storage format, read on each process only a part of the serial mesh stored in this file. The mesh is then redistributed using the partition created in parallel by the user or by ParMetis (or PtScotch or Kahip). With this method, if the starting mesh is very fine, there can be no memory problem because a process only holds a portion of the mesh during the entire read. The drawback is that the amount of communication is higher compared to the MFEM strategy, where all processes perform the same serial computation without exchanging information. This can be seen in figures 21c and 21d with test case

of section 2.1.1. The MFEM curve related to mesh reading is almost constant as all process do the same amount of work (reading,partitioning,filtering). In comparison, FEniCSx scale up to 8 processes and its reading performance stagnate or decrease with 16 process and above (parmetis is certainly using to much process for the given mesh dual-graph size, hardware i/o may be saturated and redistribution may involve too much communication)

At the time of writing, the FEniCSx C++ API does not provide any other mesh reader for any other mesh format. Only the Python API provides, in addition to XDMF, interaction with Gmsh by directly translating a Python gmsh API mesh model into a Dolfinx mesh via gmshio. By the way, it is this gmshio module that has been used to generate XDMF files from gmsh Neper files for the test case of the section 2.1.1:

```
from mpi4py import MPI
from dolfinx.io import XDMFFile, gmshio
msh, c, e = gmshio.read_from_msh("path/to/gmsh/file", MPI.
    COMM_WORLD, gdim=2)
# arbitrary name
msh.name = "neper_dam"
c.name = f"{msh.name}_cells"
e.name = f"{msh.name}_facets"
with XDMFFile(msh.comm, "path/to/xdmf/file", "w") as file:
  file.write_mesh(msh)
  file.write_meshtags(c, msh.geometry, geometry_xpath=f"/Xdmf/
      Domain/Grid[@Name='{msh.name}']/Geometry")
  file.write_meshtags(e, msh.geometry, geometry_xpath=f"/Xdmf/
      Domain/Grid[@Name='{msh.name}']/Geometry")
```

Listing 7

With the Python API, reading the xdmf file gives the same kind of performance as with the C++ API, as can be seen in figures 21c and 21d (variation are certainly due to I/O hardware).

5.2 Partitioning

In figure 5 the mesh distribution of the test case of section 2.1.1 for both library is shown for 64 processes. The partitioning is naturally[1] not the same as MFEM use Metis on full serial mesh dual-graph, and FEniCSx, Parmetis

[1] See paper on ParMetis/Metis

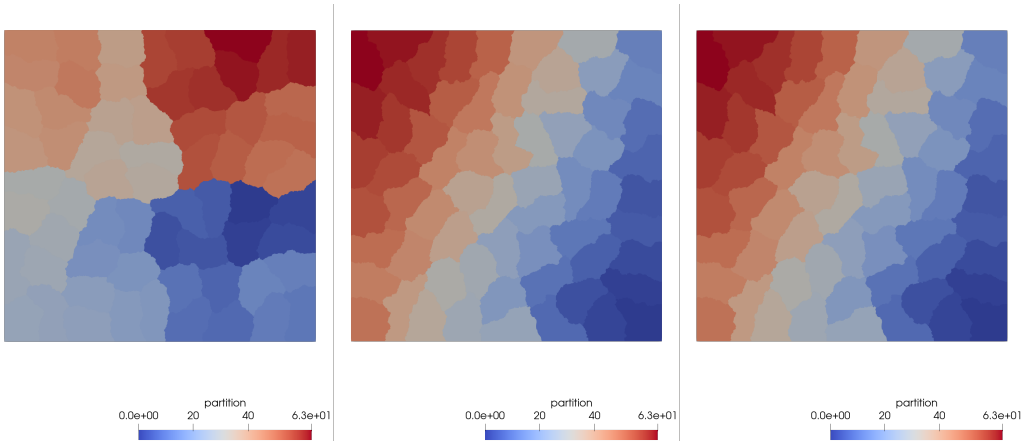


Figure 5: Mesh distribution on 64 processes by MFEM (left) FEniCSx_C++(center) and FEniCSx_py(right) of the test case of section 2.1.1.

on an arbitrary distributed mesh dual-graph. In both case the nested dissection algorithm roughly divided the full square mesh in four blocks themselves divided in four sub blocks and so on. It appears more clearly with MFEM/Metis where pids are following the dissection and no inital mesh distribution interferes in the process of partionning.

The term "ghost" used in MFEM and FEniCSx when using distributed meshes means that some entities are duplicated between processes to achieve certain objectives. At the very least, all entities at the boundary between different processes are duplicated in all those processes (i.e. in 1D nodes, in 2D nodes and edges and in 3D nodes, edges and faces). This ensures that the elements connected to the boundaries of a process have the correct geometric and topological definition.

Otherwise, cells can be duplicated in all connected processes for specific calculations. With FEniCSx it's when reading a mesh that the "dolfinx::mesh::GhostMode" can be selected. It can be:

- none: There is no cell duplication
- shared_facet: For an edge (2D) or face (3D) inside the part but at the boundary of 2 processes, all its related cells are present in both processes.
- shared_vertex: Apparently, at the time of writing, this is equivalent to

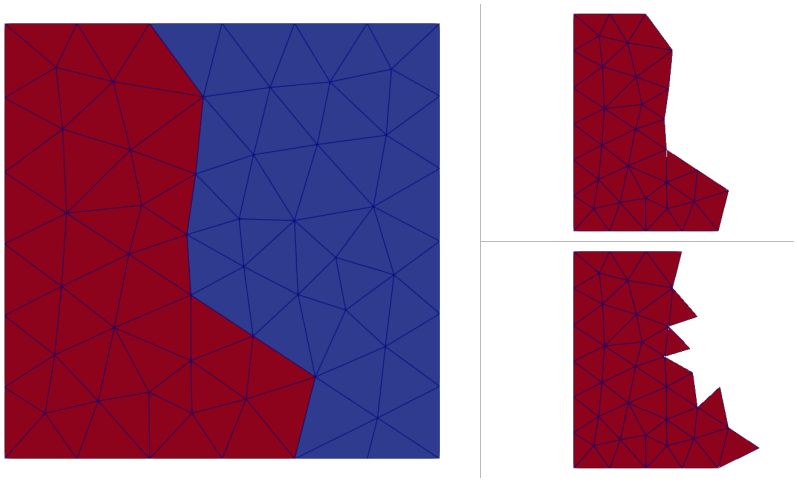


Figure 6: FEniCSx ghosting illustration with a mesh distributed over 2 processes. On the left, the mesh distributed on 2 processes without ghost cell (one colour for each process). On the right, cells present on one process with (bottom) and without (top) ghost cells.

shared_facet for 2D/3D mesh. It may be related to 1D meshes that have not yet been tested.

Figure 6 illustrates the ghost modes of FEniCSx with a simple 2D mesh distributed over 2 processes.

MFEM does not have ghost cells in its mesh structure. Only references to neighbours in other processes are created and communication occurs when data needs to be used for computation.

In any case, to keep the numbering (mesh/dof) consistent, a notion of "owner" is introduced for ghost entities. Among all processes holding an instance of an entity, one is arbitrarily chosen to "own" that entity. Numbering is thus imposed by "owner" processes, and other processes inherit this numbering. For example, in both libraries, vertices are owned by the process with the smallest process ID.

5.3 Refining

Both libraries provide a refinement strategy (local or global). MFEM is more general in this aspect as it can generate non-conforming mesh (with control of hanging entities) and is able to derefine refined zone based on some error

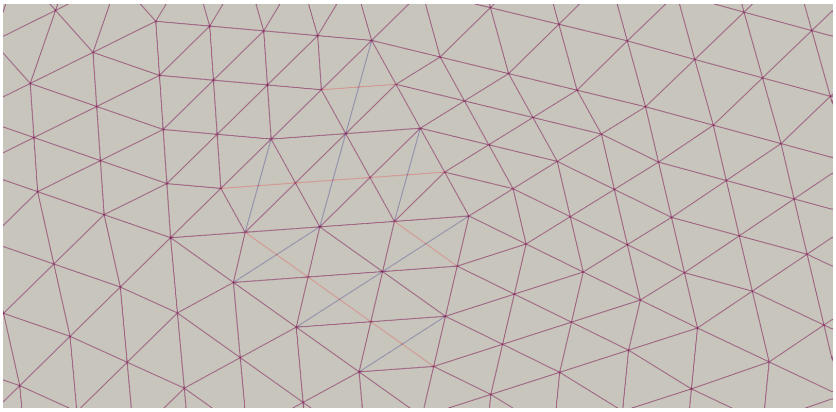


Figure 7: MFEM(red) and FEniCSx(blue) refined mesh of the test case of section 2.1.1. Both have the same number of nodes. MFEM follows always the same refinement pattern. In contrast, FEniCSx has spited some parallelipiped in a different way resulting on edges not parallel to any edges of the original refined element(center left elements in the view)).

criteria. This generality is made possible by a dedicated mesh database that stores the history of the refinement. In contrast, FEniCS's refinement tools generate a new mesh instance (or a set of information describing a mesh) and optionally the parent cell/face indices of child cells/faces.

Only uniform refinement has been tested in this study. The "UniformRefinement " method of the class "mfem::Mesh" of MFEM uses a simple algorithm to split cells:

- generate a new vertex at the center of each edge
- based on the original vertex and the new middle vertices, generate subelement: 2 in 1D, 4 in 2D, 8 in 3D (for pure tetrahedral meshes, an "A" or "B" algorithm with different quality control is proposed).

FEniCSx with "dolfinx::refinement::plaza::refine" function use a more sophisticated algorithm related to the article [3]. In the 2D example of section 2.1.1 they provide almost the same refined mesh except that some element have been flipped (certainly to respect some mesh quality criteria) by FEniCSx(figure 7).

5.4 Creating

Regarding mesh creation on the fly, FEniCSx propose 1D (" create_interval ")/2D (" create_rectangle")/3D ("create_box") interval, rectangle or box creation for quick testing. MFEM proposes "Mesh::MakeCartesian1D", "Mesh:: MakeCartesian2D" and "Mesh::MakeCartesian3D" for the same purpose. And in both libraries, the mesh class API provides a way to code the creation of a mesh from scratch.

6 Space and field(point 3)

With a mesh in hand, the creation of a finite element discretisation of a domain requires the definition of the finite element type.

6.1 Finite element

MFEM provides a large collection of finite elements of arbitrary order. They are grouped by family type represented by class that are all deriving from "mfem::FiniteElementCollection". In 1D, 2D and 3D, families are (not exhaustive):

- "mfem::H1_FECollection": Arbitrary order H^1-conforming (continuous) finite elements.
- "mfem::L2_FECollection": Arbitrary order "L2-conforming" discontinuous finite elements.
- "mfem::ND_FECollection": Arbitrary order H(curl)-conforming Nedelec finite elements.
- "mfem::RT_FECollection": Arbitrary order H(div)-conforming Raviart-Thomas finite elements.
- ...

All these elements use a polynomial bases chosen at construction time. Note that not all elements can use all bases types. Selection is done using enumeration of "mfem::BasisType" class: GaussLegendre, GaussLobatto(Lagrange), Serendipity, Positive (Bernstein), ...

The following code creates a finite element collection (called "fec") representing an H^1-conforming element of dimension 2 and order 1 using (default argument) Gauss-Lobatto polynomial bases:

```
H1_FECollection fec(1, 2);
```

Listing 8

For order 1 elements that are not intended to be extended to a higher degree in the application, it is cheaper to use a handmade simple linear base as encoded in the linear collection:

```
LinearFECollection fec;
```

Listing 9

This is illustrated by the performance analysis in section 8.3.3.

In FEniCSx finite elements are related to the basix library. This library provides the class "basix::FiniteElement" which can hold any type of finite element. This class play the role of "mfem::FiniteElementCollection" but rely on generic structure to define finite elements compared to MFEM which use derivation for genericity. For a full list of basix built-in elements, see for example https://defelement.com/lists/implementations/. The type of element is chosen at instantiation time by calling the "create_element" function with the following arguments:

- family:
  - arbitrary order Lagrange-like: P or "Lagrange","DP","DG","DQ" or "P" string in python
  - arbitrary order H(div)-conforming Raviart-Thomas: RT or "Raviart-Thomas", "N1F", "N1div", "Nedelec 1st kind H(div)", "RTCF", "NCF" or "RT" string in python
  - arbitrary order H(curl)-conforming Nedelec: N1E or "N1curl", "Nedelec 1st kind H(curl)", "RTCE", "NCE" or "N1E" string in python
  - ...
- CellType; type of element (point, interval, triangle, tetrahedron, quadrilateral, hexahedron, prism or pyramid)
- degree: arbitrary order to use

- lagrange_variant: specify the polynomial bases to be used and the DOF placement:
  - unset: Use a default for each family, which depend on many aspects.
  - equispaced: equi distance placement
  - legendre: Legendre instead of Lagrange polynomial
  - bernstein: Bernstein instead of Lagrange polynomial
  - ...

  All variants do not work with all element types.

- discontinuous: If True element is discontinuous. The discontinuous element will have the same DOFs as a continuous element, but the DOFs will all be associated with the interior of the cell.

With C++ API you can also call directly the function that creates an element related to a specific family ("basix::element::create_lagrange" for P family,...).

But in FEniCSx there is no need to instantiate the basix element as in fact all the problem is defined has UFL expression (see 8.1). So in Python the function "basix.ufl.element" can be used instead of "basix.create_element" to create a UFL compatible element. It use the same set of parameter as "basix.create_element". The following code creates a triangular finite element (called "elem") using Lagrange polynomial bases of order 1 to interpolate 2-dimensional vector field on the element:

```
elem = basix.ufl.element("Lagrange","triangle",1,shape=(2, ))
```

Listing 10

With the C++ API, the above code is placed in the Python script interpreted by ffcx (see section 8.1).

6.2 Space

With the mesh and after instantiating an element (or a collection) of a given type, it is possible to create the finite element space on the domain.

In MFEM, with the finite element collection declared in listing 8 or 9 and a "mfem::ParMesh" pmesh (see section 5), the space to interpolate a 2-dimensional vector field is created by the following instruction:

```
ParFiniteElementSpace space(pmesh, &fec, 2, Ordering::byVDIM);
```

Listing 11

The "Ordering::byVDIM" value means that dofs are stored in an element by first looping over the vector dimension (inner loop), then over the nodes (outer loop). Alternatively, the user can select "Ordering::byNODES", which loops first over the nodes (inner loop) and then over the vector dimension (outer loop).

With FEniCSx, using the Python API, the code to create a finite element space on the mesh given by "domain" with finite element "elem" declared in listing 10, is:

```
space = dolfinx.fem.functionspace(domain, elem)
```

Listing 12

With the C++ API, the above code is placed in the Python script interpreted by ffcx and the space is retrieved as presented in section 8.1 listing 16.

6.3 Field

In MFEM, in parallel, a field can be created as an instance of the class "ParGridFunction". An "x" field is created as follows, using the space defined in the previous section (listing 11):

```
  ParGridFunction x(space);
```

Listing 13

With FEniCSx, the same field would be constructed in Python as follows

```
x=dolfinx.fem.Function(space,name='myfield')
```

Listing 14

And using the C++ API, considering that "V" is the space "space" collected from the Python script via ffcx (see for example listing 16), the code is:

```
auto x = std::make_shared<fem::Function<scalar>>(V);
x->name="myfield"
```

Listing 15

where use of shared pointer to encapsulate the field object is related to coefficients argument of "dolfinx::fem::create_form" function (see section 8.1). In this example in Python and C++, the field name is set to a dummy value "myfield". This is the name that will be attached to the field in the output files and will therefore be visible when the data is processed with Paraview or Visite or .....

7 Material property(point 4)

In the example in section 2.1.1, both libraries use cell tags to assign material to mesh elements. These tags come from the original Gmsh mesh files generated by Neper (figure 1).

With MFEM, Gmsh physical property are directly assigned to element attribute while reading mesh file. The element attribute is a single integer that can be accessed using the "GetAttribute" method of the "mfem::Element" class. By default, no other data can be associated with elements.

In FEniCSx, tags are information that can be anything. They are stored outside the mesh/entities object in a dedicated template class ("template< typename T> dolfinx::mesh::MeshTags<T>"). Therefore, when using gmshio, the Gmsh physical property must be transferred to the XDMF file in a separate location in the file tree structure(see section 5.1). And when reading the XDMF file, either from C++ or Python API, extra instructions are mandatory to read the "mesh::MeshTags" objects. For these objects, the tag values are obtained using the "values" method.

These tags (integers) are then used with both libraries to set Young's modulus (and consequently Lamè coefficients λ and μ) using following expression:

$$E(tag) = \frac{1.e^8 - 5.e^6}{199.} \times kr(tag) + 5.e^6 \tag{7}$$

with $kr()$ a semi-random function of tag (based on rand function of stdlib) that return an integer between 0 and 199.

For MFEM, the strategy adopted is to use a piecewise constant coefficient ("mfem::PWConstCoefficient") directly for λ and μ. This "mfem::PWConstCoefficient" class uses constants keyed from the element attribute numbers. The "mfem::Vector" used to construct these objects are filled for each tag with the expressions of the Lamè coefficients as a function of Young's modulus given by (7) and a constant Poisson's ratio value.

For FEniCSx, the choice has been made to start from Young's modulus (E) and Poisson's ratio(ν) in the construction of the formulation as shown in 8.1. In this case, E is a "dolfinx::fem::Function" initialised with (7) and ν is a "dolfinx::fem::Constant". Perhaps starting from Lamè coefficients, as in MFEM, would be more computationally efficient, as the coefficient calculation would be skipped. But as the final formulation is itself written in C code, it's not obvious that we'll get any gain. Furthermore, replacing calls to "dolfinx::fem::Function" and "dolfinx::fem::Constant" with two calls to "dolfinx::fem::Function" may not be better. (to be tested).

7.1 Damage

In the example in section 2.1.1, the material behaviour depends on a damage field. This field is constant throughout the simulation, but is calculated at the start based on an arbitrary setting of 1 at some grain boundary. The smoothing process, which produces the damage profile shown in figure 2, is good for testing in parallel the interaction with the mesh database and field object of both libraries. The smoothing algorithm is very crude. It is presented in algorithm 1. It is mainly a loop to repeat an averaging process on all nodes using their topological neighbourhood. This averaging process is performed twice per loop. Once to enlarge the non-zero damage zones, examining only nodes with damage below a certain arbitrary threshold (0.01). Twice to average all the nodes and thus smooth the damage zone. The averaging process required some communication as the topological neighbourhood of a node can be scattered in different domains/processes.

This algorithm has been used in MFEM and FEniCSx C++ implementations. In both cases, the underlying data array of the "d" field object was used directly (thanks to the order 1 Lagrange simplicity) to speed up the computation. In MFEM this was done by providing the underlying data array at construction time (avoiding costly use of "mfem::GridFunction ::GetNodalValues" for example). And in FEniCSx it was done by calling "d->x()->mutable_array()" method, which directly gives a reference to the underlying data array. The performance shown in figure 21e and 21f shows that both implementations scale relatively well despite the presence of the test that introduces imbalance. Both library provides method to theire field class that easely scatter/gather information at ghost nodes. MFEM is 2 to 6 times slower than FEniCSx C++, but has an additional speed-up rate. This

Algorithm 1 Rough damage smoothing algorithm. $\mathfrak{L}$ is the set of node indices local to a process. $\mathfrak{N}_i$ is the set of nodes whose index is local to a process and connected to node i by an owned edge. Inside the general loop, the first loop at node enlarges the non-zero damage zone and the second loop smooths and enlarges all non-zero damage zones. The general loop is just a repetition (times) of the 2 loops on nodes. At the end, v_dam_cur contains the smoothed field. The ▷ icon indicates a communication step.

Create v_dam_cur and v_dam_new vectors to hold damage at nodes/dofs
Set damage to 1 in v_dam_cur at some grain boundary nodes
for iter_smooth $\in [0, max\_smooth]$ **do**
  v_dam_new$\leftarrow 0$
  for $i \in \mathfrak{L}$ **do**
    if v_dam_cur$[i] < 0.01$ **then**
      v_dam_new$[i] \leftarrow \sum_{k \in \mathfrak{N}_i}$ v_dam_cur$[k]$
      Update ghost nodes v_dam_new values ▷
      v_dam_new$[i] \leftarrow \max\left(\text{v_dam_cur}[i], \frac{\text{v_dam_new}[i]}{card(\mathfrak{N}_i)}\right)$
    end if
  end for
  v_dam_cur$\leftarrow$v_dam_new
  for $i \in \mathfrak{L}$ **do**
    v_dam_new$[i] \leftarrow \sum_{k \in \mathfrak{N}_i}$ v_dam_cur$[k]$
    Update ghost nodes v_dam_new values ▷
    v_dam_new$[i] \leftarrow \max\left(\text{v_dam_cur}[i], \frac{\text{v_dam_new}[i]}{card(\mathfrak{N}_i)}\right)$
  end for
  v_dam_cur$\leftarrow$v_dam_new
end for

is certainly related to the use of roughly $max\_smooth \times \sum_{i \in \mathfrak{L}} card\,(\mathfrak{N}_i)$ operations. Depending on the threshold (0.01), this number can be augmented. In any case, when the domain is split in 2, $card\,(\mathfrak{L})$ is roughly divided by 2, but also $card\,(\mathfrak{N}_i)$, for nodes on the process boundary, are reduced by some proportion. As the number of processes increases, the proportion of nodes at the domain boundary increases and the reduction in operation increases. But surprisingly, with the FEniCSx C++ implementation, this effect does not exist.

Regarding the FEniCSx Python implementation, if algorithm 1 is implemented as loops like in the C++ implementation, the performance is of

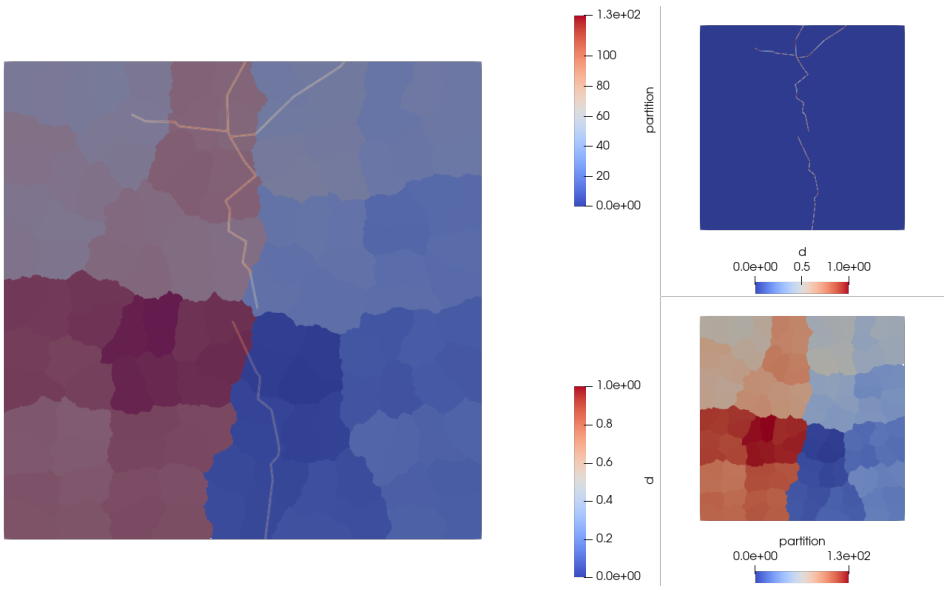


Figure 8: Imbalence illustration with 128 cores (MFEM distribution). The left view is the superposition of the two right views, one of which is partially transparent. Only 20 processes is affected by the damage calculation.

course catastrophic. In this study, after some testing, the two loops on the nodes were replaced by applying an operator to the field via a SciPy sparse matrix. Only loops on the topology remain to construct this operator. This is done once, but still costs a lot. Optimising this part, if possible, would require some additional effort not related to this study. Therefore, the performance of the python implementation shown in figures 21e and 21f is poor (7 to 22 times slower) compared to other implementations in C++. This illustrates the fact that using the Python API requires care, so any algorithm not provided or constructable with FEniCSx must either be very well implemented or moved to a C/C++/Fortran implementation and encapsulated in a Python module.

The introduction of this damage locally, at some grain boundaries, theoretically results in unbalanced parallel domains, as we have not considered the additional cost of damage treatment during partitioning. In figure 8 on the left view, it appears that of the 128 processes, only 20 processes are traversed by a non-zero damaged zone. As the damage zone is rather narrow, only a small proportion of the element in these 20 processes is affected by

	Task	val min	val max	CV	val avg	% total avg
MFEM	All	8.578603	8.633553	0.196100	8.608193	100.0
	Define damage	0.073467	0.074400	0.002707	0.073950	0.9
	Elementary vector	0.130437	0.164723	0.055433	0.134700	1.6
	Elementary matrix	0.078890	0.105820	0.067967	0.083190	1.
FEmiCSx	All	13.518510	13.520770	0.004013	13.519100	100.0
	Define damage	0.042953	0.043807	0.001817	0.043407	0.3
	Elementary vector	0.046653	0.053423	0.006133	0.049870	0.4
	Elementary matrix	0.078733	0.088870	0.011107	0.081627	0.6

Table 2: Elapsed times in seconds of some tasks related to damage for a simulation with 128 processes (intermediate implementation)."val min" and "val max" are respectively the minimum and maximum computing time among all processes for the execution of the task."val avg" is the mean execution time of all processes to complete the task. The "CV" (100 ×"standard deviation"/"val avg"), representing the imbalance, is the coefficient of variation multiplied by 100 to mimic a percentage of variation compare to mean value. "% total avg" represent percentage of "val avg" of a task compare to "All"

	Task	val min	val max	CV	val avg	% total avg
MFEM	All	12.22343	12.26745	0.10736	12.24595	100.0
	Define damage	0.94938	0.95046	0.00204	0.94988	7.8
	Elementary vector	0.17121	0.22073	0.09661	0.19193	1.6
	Elementary matrix	0.10981	0.16517	0.16370	0.14177	1.2

Table 3: Same type of result as in table 2 for a larger damage band thickness simulation not shown in this study, which converges in 2 more non-linear iterations.

the damage calculation. This may explain why no major imbalance effect was observed in relation to the creation or effect of damage (see CV in table 2). Some calculations, not presented here, confirm (see CV in table 3) that imbalance appears when the thickness of the damage band is increased (larger max_smooth in algorithm 1) and thus the proportion of elements with damage for these 20 processes is increased.

In any case, when distributing the mesh, both libraries allow the user to select a domain partition that can take damage costs into account if needed.

8 System creation(point 6)

System creation is related to variational expression of the problem. From finite element discretizations of variational forms, elementary vectors and matrices are computed and assemble into final system. The approaches to compute finite element discretizations of variational forms (like (3)) range from purely formal descriptions, which are close to mathematical expressions, to purely C++ programming implementations.

8.1 FEniCSx variational expression

8.1.1 General overview

FEniCSx uses a formal description with a Unified Form Language (UFL). This description, provided by a Python script, is transformed into a C language code by the ffcx compiler. This code provides functions that transcript in elementary C instructions the different form(s) and space(s) defined in Python script. Form functions, called "kernels", are called by the assembly routine to obtain the elementary vector or matrix for a given finite element. When using full Python API this process is completely hidden to users that do not care to generate and branch kernels. With the C++ API, the user must branch the kernel compiled by ffcx using the following instruction, which illustrates the case of a linear and a bi-linear form defined on the same finite element space, on the full domain Ω, using some coefficients and constants in the formulation:

```
// create function space object from python field "u"
auto V = std::make_shared<fem::FunctionSpace<scalar_dolf>>(fem::
    create_functionspace(SPACE, "u", pmesh));
// create linear form object
auto lin_form = std::make_shared<fem::Form<scalar, scalar_dolf>>(fem::
    create_form<scalar>(*FORM_L, {V}, coefficients,constants, {}));
// create bi-linear form object
auto bilin_form = std::make_shared<fem::Form<scalar, scalar_dolf>>(fem::
    create_form<scalar>(*FORM_BL, {V,V}, coefficients, constants, {}));
```

Listing 16

where:

- "pmesh" pointer (shared) pointing to Mesh object discretizing Ω domain.

- "SPACE" is the name of the helper function generated by ffcx. It is used to create function space using function field name (i.e. name of the Python variable, here "u"). This function name starts with "functionspace_form_" followed by the name of the python script, followed by "_", followed by the Python variable name of the considered form.

- "FORM_L" is the name of the helper function generated by ffcx. It is used to create linear form using name which was given to the form in the python script. This function name starts with "form_" followed by the name of the python script, followed by "_", followed by the python variable name of the considered form.

- "FORM_BL" is the same as "FORM_L" except that it is related to the bi-linear form defined in the python script. The difference is also noticeable with the "const std::vector<std::shared_ptr<const FunctionSpace<scalar_dolf>>>" vector object given as second argument of the "fem::create_form<scalar>" function: with linear form this vector contains only one function space, and with bi-linear form it contains two function spaces (in this example the same one).

- " coefficients " is a "std::map" that contains a set of functions defined on a finite element function space (here "V") indexed by the "std::string" corresponding to their name as defined in the python script. They are the functions used in the form and/or bi-linear form.

- "constants" is a "std::map" that contains a set of constants indexed by the "std::string" corresponding to their name as defined in the python script. They are the constants used in the form and/or bi-linear form.

Lets use the example 2.1.1 to illustrate how formulations are encoded in the python script. First FEniCSx python module as to be include:

```
import ufl
from ufl.classes import (Mesh,FunctionSpace,Coefficient,
                         Constant,TrialFunction,TestFunction,
                         FacetNormal)
from ufl import ( sqrt,grad,inner,tr,Identity,dot,dx,ds,
                  conditional,lt,gt,eq)
import basix
```

Listing 17

The "basix" module provides UFL finite element definition. All other definition commes from "ufl" module itself.

The encoding of (3) imply to define Ω, $\mathrm{d}V$, $\mathbf{u}$, $\mathbf{v}$, $\mathbf{f}$, ϵ and $\sigma(\mathsf{u})$. This last one obtain from (1) by use of (2) imply the definition of λ, μ, d, Λ_i, α_i and α. Encoding in UFL language start by defining Ω and its discretization related to a mesh made of finite elements (triangle here) interpolating function with polynomial bases of specific order (see section 6). In this work $\mathbf{u}$ and $\mathbf{v}$ will be discretized by an order 1 Lagrange vectorial field with 2 components. The damage field d will use an order 1 Lagrange scalar field. As some material properties (Young's modulus E) are constant per element but not the same in between elements, an order 0 discontinuous Galerking field will provide the appropriate space to set λ an μ for all elements. The finite elements, mesh and function spaces encoding is thus the following:

```
elem = basix.ufl.element("Lagrange","triangle",1,shape=(2, ))
element_scal = basix.ufl.element("Lagrange", "triangle", 1)
DGelement = basix.ufl.element("DG", "triangle", 0)
domain = Mesh(elem)
space = FunctionSpace(domain, elem)
DGspace = FunctionSpace(domain, DGelement)
space_scal = FunctionSpace(domain, element_scal)
```

Listing 18

where "space" will be used for $\mathbf{u}$ and $\mathbf{v}$, "DGspace" will be used for E, and "space_scal" will be used for d. Not that "domain" here is a simple UFL mesh based on the basix element description. It is not the real mesh that is read in the C++ implementation.

The damage field d is encoded as a mathematical function defined on "space". In this formulation d is a constant field during the simulation with the damage arbitrary fixed to a specific value. Thus in UFL it must be defined with the UFL Coefficient definition as follows:

```
d=Coefficient(space_scal)
```

Listing 19

In the same way Young's modulus E and $\mathbf{f}$ load are defined by:

```
E=Coefficient(DGspace)
f=Coefficient(space)
```

Listing 20

The Poisson ration ν is constant on Ω thus the UFL Constant is used to define it:

```
nu=Constant(space)
```

Listing 21

Note that "d", "f", "E" and "nu" are just declaring UFL formal variable for the form (3) but there concrete setting are done in the C++ implementation.

The Lamè coefficients λ and μ, are based on Young's modulus E and Poisson ratio ν. They are simply expressed as mathematical expression[2] in python language that lead to UFL expression when parsed by ffcx compiler as they use UFL variables "E" and "nu":

```
mu = E / (2.0 * (1.0 + nu))
lmbda = E * nu / ((1.0 + nu) * (1.0 - 2.0 * nu))
```

Listing 22

In non-linear resolution, the displacement from the previous iteration ($\mathbf{u}$), is seen as a constant function for the form thus it is defined with "Coefficient" definition. The test function $\mathbf{v}$ (named "delta_u" to avoid confusion with c++ "V" object) is created with the special "TestFunction" UFL definition. The incremental displacement (named "du", that play the role of trial function in Jacobian operator construction) use "TrialFunction" for its definition.

```
u = Coefficient(space)
delta_u = TestFunction(space)
du = TrialFunction(space)
```

Listing 23

The strain tensor ϵ is simply defined as a python function ("eps") using formal UFL gradient operator and its transpose according to small strain assertion given by (5):

```
def eps(x):
  return 0.5*(grad(x) + grad(x).T)
```

Listing 24

Regarding stress tensor derived from potential (1) there is different way to treat this part as discussed bellow. In all case the idea is to obtain a python function called "sigma" that takes as argument a displacement field and return a rank-2 tensor. This lead to write form 3 as follow:

[2] As mentioned in section 7, λ and μ may have been encoded directly as a field.

```
F=inner(sigma(u), eps(delta_u))*dx - inner(f, delta_u)*dx
```

Listing 25

where "dx" stand for dV and "inner" define the inner product as a contraction over all axes of its two argument, that is the sum of all component-wise products. Finally, thanks to the automatic functional differentiation provided by ufl module the Jacobian operator corresponding to the derivation of the form (3) against "du" is given in one line by:

```
J=ufl.derivative(F,u,du)
```

Listing 26

This illustrates the power of FEniCSx to simplify implementation of variational formulation as "J" bi-linear form is defined by one line compare to the full C++ implementation given for MFEM in 8.2.

Using the full Python API, these UFL formal representations are used to create kernel over the creation of form object as in the following:

```
lin_form=dolfinx.fem.form(F)
```

Listing 27

It's during this step that the ffcx compiler is called on the fly and the equivalent C++ code of listing 16 is done.

Back to the UFL expression, the numerical and formal complexity in "sigma" comes from strain eigenvalue evaluation and potential derivation. The following subsection 8.1.2, 8.1.3, 8.1.5 and 8.1.4 give some solutions and comments on different version of "sigma". In all cases, to stick to MFEM version, two tests are introduced that check that if the damage is non-zero or if one strain tensor invariant (given by (6)) at least is non-zero, asymmetric potential (1) is used. Otherwise the following symmetric version is used :

$$\psi'(\epsilon, d) = (1-d)\left(\frac{\lambda}{2}tr(\epsilon)^2 + \mu\sum_{i=1}^{2}\sum_{j=1}^{2}{\epsilon_{ij}}^2\right) \tag{8}$$

The first test is related to performance. If d is zero in (1), the α_i and α can be anything as they will vanish by the multiplication by d. Thus it is expansive to pass by eigenvalues computation and using (8) is cheaper. The second test is also related to performance but derivation can be an issue if both invariant are zero as disused below.

8.1.2 Symbolic

Thanks to python large collection of libraries it is possible to use symbolic python library SymPy on top of UFL implementation. The idea is to obtain σ with SymPy as a symbolic expression, translate it in strings that are used to create UFL expression on the fly in "sigma" function. The advantage is that the differentiation of the potential (1) and the eigenvalues symbolic computation is done by SymPy if you are more familiar with this library. And this library may provides nice simplified symbolic expression.

Firstly we define symbolic variables and symbolic strain tensor as a 2x2 matrix T (defined as symmetric):

```
import sympy as sp
l,m,dam,psi,al,al1,al2=sp.symbols('lmbda mu d psi alpha alpha1
    alpha2')
T = sp.Matrix(2, 2, lambda i, j: sp.Symbol('T[%d, %d]' % (i, j),
     symmetric=True, real=True))
```

Listing 28

Note that the identifier of the symbolic variables must be the same as the UFL variable already defined ('lmbda' string for "lmbda", 'mu' string for "mu" and 'd' string for "d").

Before describing the potential (1) the symbolic eigenvalues have to be computed with the help of the related SymPy function:

```
eigv = T.eigenvals(multiple=True)
```

Listing 29

The potential ψ (1), using SymPy symbolic variable defined above, is then:

```
psi=(T.trace()**2)*(1-al*dam)*l/2+m*((1-al1*dam)*eigv[0]**2+(1-
    al2*dam)*eigv[1]**2)
```

Listing 30

The stress tensor is then simply obtained as the following symbolic variable, result of the differentiation of "psi" by "T":

```
sig=sp.simplify(sp.diff(psi,T))
```

Listing 31

As mentioned, first and second invariant of strain tensor (6) are also required in "sigma" function to check if they are both zero or not. Here is their symbolic expression:

```
i1=T.trace()
i2=sp.simplify(((T*T).trace()-i1*i1))/2
```

Listing 32

As already mentioned, all SymPy symbolic expression have to be transformed in strings to be used in the final "sigma" function:

```
eig_expr = list(map(str, eigv))
sig_expr = list(map(str, sig))
i1_expr = str(i1)
i2_expr = str(i2)
```

Listing 33

The final "sigma" function is then the following:

```
1 def sigma(x):
2   T=eps(x);
3   eps_i1=eval(i1_expr)
4   return  conditional(gt(d,0.0),sig_dam(T,eps_i1),sig_sym_dam(T,
        eps_i1))
```

Listing 34

On line 2 "T" UFL variable is computed as the result of applying "eps" function to argument "x". Note here that the name of the tensor is chosen so that it correspond to the variable in the string expression that follows. Than in line 3 the python eval function replace all variable in string expression by there UFL variable to form new UFL expresion coresponding to i_1 invariant. At this stage a test is put in place to avoid any complex computation if the damage "d" is null. This test is provided in UFL language by "conditional" definition which correspond to the formal expression "if test(first argument) is true return second argument otherwise return third argument". This test may have been built into the SymPy symbolic expression "sig", but translating it with "eval" to "conditional" wouldn't have been easy, if possible. The nature of the returned arguments of "conditional" need to be the same, thus use of intermediate python function that return both a rank-2 tensor simplify the script. When the damage is null the function "sig_sym_dam" corresponding to derivation of potential ψ' (8) is called. Its implementation is the following:

```
def sig_sym_dam(strain,trace):
  return (1.-d)*(2.0*mu*strain + lmbda*trace*Identity(2))
```

Listing 35

When the damage is not null the following function "sig_dam" is called

```
1 def sig_dam(T, eps_i1):
2   eps_i2=eval(i2_expr)
3   return  conditional(eq(eps_i1 ,0.0) ,conditional(eq(eps_i2 ,0.0) ,
        sig_sym_dam(T, eps_i1) ,sigma_strain_not_nul(T, eps_i1)) ,
        sigma_strain_not_nul(T, eps_i1))
```

Listing 36

Its only purpose is to compute i_2 (line2) and test if both invariant are null or not. If they are both null it calls the function "sig_sym_dam" for performance purpose and to avoid singularity when computing derivative of "J" even if null strain tensor (one case where invariants are both null) imply null stress tensor.

When one invariant at least is not null the function "sigma_strain_not_nul" is called:

```
1 def sigma_strain_not_nul(T, tre):
2   eig1 ,eig2=map(eval , eig_expr)
3   alpha1 = conditional(lt(eig1 ,0.0) ,0.0 ,1.0)
4   alpha2 = conditional(lt(eig2 ,0.0) ,0.0 ,1.0)
5   alpha = conditional(lt(tre ,0.0) ,0.0 ,1.0)
6   s00 ,s01 ,s10 ,s11=map(eval , sig_expr)
7   v= ([s00 ,s01] ,[s10 ,s11])
8   return ufl.tensors.as_tensor(v)
```

Listing 37

Again string expression are transformed in UFL expression in line 2. Eigenvalues are transformed into "eig1" and "eig2" UFL variable to be able to do test in UFL language in following lines. In line 3, 4 and 5 test on sign of eigenvalues give a way to set α_i and α. Note again that the choice of the name of the UFL variable in these lines have to correspond to the one used in the declaration of the SymPy symbolic variable. In line 6,7 and 8, using again "eval", the function create the stress tensor as an UFL tensor, based on all UFL variables created so far.

8.1.3 Symbolic manually symetrized

This version uses the same tools as 8.1.2 solution except that symmetry of strain tensor (and thus stress tensor) is forced in the implementation. Desfacto the SymPy 2x2 matrix construction was defined as symmetric but during simplification the fact that T(1,0)=T(0,1) was not used. To force this aspect this matrix is created by hand using 3 new symbolic variables:

```
l,m,dam,psi,al,al1,al2,e11,e22,e12=sp.symbols('lmbda mu d psi
    alpha alpha1 alpha2 eps11 eps22 eps12')
T = sp.Matrix([[e11,e12],[e12,e22]], symmetric=True, real=True)
```

Listing 38

The "eigv" and "psi" remain the same but "sig" as to be corrected because we introduce an $eps12^2$ in the expression that provides a spurious 2 in the derivation that is corrected as follow storing only 3 terms out of the 4 for symbolic stress tensor:

```
siga=sp.simplify(sp.diff(psi,T))
sig=[siga[0,x] for x in [0]]
sig.append(siga[0,1]/2.)
sig.append(siga[1,1])
```

Listing 39

Invariants and string expression construction are the same and the final "sigma" function becomes:

```
def sigma(x):
  T=eps(x);
  eps11=T[0,0]
  eps12=T[0,1]
  eps22=T[1,1]
  eps_i1=eval(i1_expr)
  return conditional(gt(d,0.0),sig_dam(T,eps11,eps22,eps12,
      eps_i1),sig_sym_dam(T,eps_i1))
```

Listing 40

with the same implementation for "sig_sym_dam" function as in 8.1.2 and the following slightly modified implementation compare to 8.1.2 for the "sig_dam" function and "sigma_strain_not_nul":

```
def sig_dam(T,eps11,eps22,eps12,eps_i1):
  eps_i2=eval(i2_expr)
```

```
  return  conditional(eq(eps_i1,0.0),conditional(eq(eps_i2,0.0),
      sig_sym_dam(T,eps_i1),sigma_strain_not_nul(eps11,eps22,
      eps12,eps_i1)),sigma_strain_not_nul(eps11,eps22,eps12,
      eps_i1))
def sigma_strain_not_nul(eps11,eps22,eps12,eps_i1):
  eig1,eig2=map(eval, eig_expr)
  alpha1 = conditional(lt(eig1,0.0),0.0,1.0)
  alpha2 = conditional(lt(eig2,0.0),0.0,1.0)
  alpha = conditional(lt(eps_i1,0.0),0.0,1.0)
  s00,s01,s11=map(eval, sig_expr)
  v= ([s00,s01],[s01,s11])
  return ufl.tensors.as_tensor(v)
```

Listing 41

8.1.4 UFL potential derivation

Compare to previous version in 8.1.2 and 8.1.3 this version is not using SymPy. Instead, the two hard aspect of the implementation of "sigma" function, eigenvalues computation and potential (1) derivation, are treated using only UFL capabilities:

- The eigenvalues are coded using the following mathematical solution for a 2x2 matrix $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ with the trace being $T = a + d$ and the determinant $D = ad - bc$:

  $$\Lambda_i = \frac{T + (-1)^{(i-1)} \times \sqrt{T^2 - 4 \times D}}{2}$$

  In the present case A is the strain tensor matrix ϵ, thus using i_1 and i_2 invariant from (6) we can rewrite eigenvalues as follows:

  $$\Lambda_i = \frac{i_1 + (-1)^{(i-1)} \times \sqrt{i_1^2 + 4i_2}}{2} \tag{9}$$

- The potential (1) derivation is done using UFL "diff" that take the derivative of a formal function given as argument with respect to a formal variable given also as argument.

As the potential (1) use Λ_i and as (9) use a square root function which can't be derived on 0, the python function to encode ψ is adding a test to check if all invariants are null or not:

```
def psi(S):
  i1=tr(S)
  i2=0.5*(tr(S*S)-i1*i1)
  return conditional(eq(i1,0.0),conditional(eq(i2,0.0),
      psi_sym_dam(S,i1),psi_non_null_strain(S,i1,i2)),
      psi_non_null_strain(S,i1,i2))
```

Listing 42

When both invariant are null, to avoid derivation problem mentioned above when computing σ and to optimize computation the following "psi_sym_dam" function is called:

```
def psi_sym_dam(T,i1):
 return (1 - d) * (i1 * i1 * lmbda / 2. + mu * (T[0,0] * T[0,0]
    + T[1,1] * T[1,1] + T[1,0] * T[1,0] + T[0,1] * T[0,1]))
```

Listing 43

which correspond to (8). Otherwise the following " psi_non_null_strain " function coresponding to (1) is called:

```
def psi_non_null_strain(T,i1,i2):
  delta=i1*i1+4*i2
  r=sqrt(delta)
  eig1 = (i1 + r) / 2.
  eig2 = (i1 - r) / 2.
  alpha1 = conditional(lt(eig1,0.0),0.0,1.0)
  alpha2 = conditional(lt(eig2,0.0),0.0,1.0)
  alpha = conditional(lt(eig1+eig2,0.0),0.0,1.0)
  return i1*i1*(1. - alpha * d) * lmbda / 2. + mu * ((1 - alpha1
      * d) * eig1 * eig1 + (1. - alpha2 * d) * eig2 * eig2)
```

Listing 44

The implementation of the "sigma" function is then:

```
def sig_sym_dam(T):
  T=ufl.variable(T)
  i1=tr(T)
  v= ufl.diff(psi_sym_dam(T,i1),T)
  return ufl.tensors.as_tensor(v)

def sig_dam(T):
  T=ufl.variable(T)
  v= ufl.diff(psi(T),T)
  return ufl.tensors.as_tensor(v)
```

```
def sigma(x):
  T=eps(x);
  return   conditional(gt(d,0.0),sig_dam(T),sig_sym_dam(T))
```

Listing 45

If damage is not null derivation of (1) is done via "psi" function. Otherwise, it is derivation of (8) which is computed via "psy_sym_dam" function.

8.1.5 Hand made encoding

This last version is close of standard C++ MFEM implementation in the sens that eigenvalues computation and the potential (1) derivation are both coming from mathematical development. Equation (9) will be used for the eigenvalue. To do the derivation of ψ we are going to use directly eignvalues derivation and pass result in the physical bases by using eigenvectors associate to (9):

$$\frac{\partial \psi}{\partial \Lambda_i} = \lambda(1-\alpha d)i_1 + 2\mu(1-\alpha_i d)\Lambda_i \tag{10}$$

where $i_1 = \Lambda_1 + \Lambda_2$ or from (6) $i_1 = tr(\epsilon)$. The normalized eignvectors associated to (9) are given by the following matrix:

$$\phi = \begin{pmatrix} \frac{\Lambda_1 - \epsilon_{22}}{n_1} & \frac{\Lambda_2 - \epsilon_{22}}{n_2} \\ \frac{\epsilon_{12}}{n_1} & \frac{\epsilon_{12}}{n_2} \end{pmatrix} \tag{11}$$

with $n_1 = \sqrt{(\Lambda_1 - \epsilon_{22})^2 + \epsilon_{12}^2}$ and $n_2 = \sqrt{(\Lambda_2 - \epsilon_{22})^2 + \epsilon_{12}^2}$. If $\epsilon_{12} = 0$ then ϕ is chosen to be the identity matrix. The srtess tensor σ is then, using matrix operator:

$$\sigma = \phi^t \cdot \begin{pmatrix} \frac{\partial \psi}{\partial \Lambda_1} & 0 \\ 0 & \frac{\partial \psi}{\partial \Lambda_2} \end{pmatrix} \cdot \phi \tag{12}$$

Note that if $i_1 = i_2 = 0$ then only the symmetric traction/compression potential must be used in the computation as eignvalues are both null. And in the same spirit if damage is null there is no need to pass by eigenvalues to obtain α_i and α as they are multiplied by zero anyway, so symmetric traction/compression potential is enough. The implementation take into account those test in the same way as in 8.1.2, 8.1.3, and 8.1.4 by dispatching the different cases in some python functions. Encoding of those function follows directly (6), (9), (10), (11), (12) and derivation of (8).

8.1.6 Full Python (based on section 8.1.3 approach)

Compared to the general approach presented in section 8.1.1 and the codes presented in sections 8.1.2 and 8.1.3, the differences are as follows:

- Everything is coded in the python script.
- "domain" is now the full mesh read from a file and not a UFL mesh.
- Volumic load, damage and Young's modulus are now directly "dolfinx.fem.Function" and they are computed in the python script.
- Poisson ratio remain a constant but it is now a "dolfinx.fem.Constant" fixed in the script.

The Lamè coefficients λ and μ have the same python expression as in listing 22 of section 8.1.1 and represent a UFL expression despite the new nature of the Poisson ratio and Young's modulus given above. In fact, in Python, "dolfinx.fem.Function" and "dolfinx.fem.Constant" are intrinsically understood as UFL expressions. And it is the same for all other functions and expressions ("eps", "sigma", "F", "J", ...) that are the same as in the 8.1.2 / 8.1.3 section and use SymPy for symbolic differentiation.

8.2 MFEM variational expression

8.2.1 Standard way

Setting up the variational expression in MFEM starts with looking at the large set of linear or bi-linear formulations available. More precisely, it is the set of derived classes of "mfem::LinearFormIntegrator" or "mfem::NonlinearFormIntegrator" that provide object that can be used in generic formulation class like "mfem::ParNonlinearForm","mfem::ParLinearForm" or "mfem::ParBilinearForm". Again, as with FEniCSx, the test case of section 2.1.1 is used to illustrate how to use MFEM when no class satisfies the problem to be coded (here the asymmetric tension/compression elasto-damaged constitutive law). The user is then responsible for creating his own derived class from the base class "mfem::NonlinearFormIntegrator" to encode (3) and the Jacobian operator derived from it. This new class, hereafter called "damIntegrator", implement the following virtual methods:

- "AssembleElementVector" which, for an element with a given displacement filed value, computes the elementary vector corresponding to (3). For a fair comparison with FEniCSx, both integrals of (3) are calculated. However, as the load is constant during the non-linear loop, in MFEM the second integral can be calculated and assembled only once in a vector given as an argument to the Newton solver (see below).

- "AssembleElementGrad" which, for an element with a given displacement filed value, computes the elementary matrix corresponding to Jacobian of (3).

The class "damIntegrator" has the following definition:

```
1  class damIntegrator : public NonlinearFormIntegrator
2  {
3    public:
4        damIntegrator(Coefficient &l, Coefficient &m,
             QuadratureFunctionCoefficient &d, const IntegrationPoint &ip_,
             const IntegrationRule *ir, VectorQuadratureFunctionCoefficient &
             load_)
5      void AssembleElementVector(const FiniteElement &el,
           ElementTransformation &Tr, const Vector &elfun, Vector &elvect);
6      void AssembleElementGrad(const FiniteElement &el, ElementTransformation
         &Tr, const Vector &elfun, DenseMatrix &elmat);
7    private:
8      Coefficient &lambda, &mu;
9      QuadratureFunctionCoefficient &dam;
10     const IntegrationPoint &ip;
11     VectorQuadratureFunctionCoefficient &load;
12     const int nd;
13     const int dim;
14     real_t limit, mlimit;
15     DenseMatrix dshape, gdshape, strain, sig, hook, block, disp, res;
16     DenseMatrix B, C, mgdshapex, mgdshapey;
17     DenseMatrix evec, potentiel_sderivative, dedeps, M;
18     Vector eval, eigns;
19     Vector vl, shape, f;
20   };
```

Listing 46

The implementation of the constructor takes as arguments the Lamè coefficients λ and μ, the damage "d", the unique integration point (and weight) to compute the first integral of (3), the integration rule (set of integration points and weights) to compute the second integral of (3) and the load (f). It sets all small vectors and dense matrices used during the calculation.

```
1  damIntegrator(Coefficient &l, Coefficient &m, QuadratureFunctionCoefficient
       &d, const IntegrationPoint &ip_, const IntegrationRule *ir,
       VectorQuadratureFunctionCoefficient &load_)
```

```
  : NonlinearFormIntegrator(ir),
  lambda(l), mu(m), dam(d), ip(ip_), load(load_),
  nd(3), dim(2),limit(1.e-12), mlimit(-1.e-12),
  dshape(nd, dim), strain(dim), sig(dim), hook(3),
  block(nd), B(nd * dim, 3), C(nd * dim, 3),
  evec(dim), potentiel_sderivative(3), dedeps(3),
 M(3), eval(dim),eigns(dim),
  vl(dim * nd), shape(nd), f(dim)
  {
    B=0.;
    gdshape.UseExternalData(vl.GetData(), nd, dim);
  }
```

Listing 47

Note that "mfem::DenseMatrix" can use its internal memory or a user-defined array("UseExternalData"). This way a monodimensional array can be interpreted by "DenseMatrix" as a bidimensional array, just like "mdspan", which is widely used in FEniCSx.

The implementation of "AssembleElementVector" starts by reshaping the result container, placing warpers (i.e. "mfem::DenseMatrix" for result and displacement vector) and then computes the result vector in two steps (first and second integral of (3)):

```
void AssembleElementVector(const FiniteElement &el, ElementTransformation &
    Tr, const Vector &elfun, Vector &elvect)
{
  // reshape vector result
  elvect.SetSize(nd * dim);

  // set vectors into dense matrix used as warper
  disp.UseExternalData(elfun.GetData(), nd, dim);
  res.UseExternalData(elvect.GetData(), nd, dim);

  // compute the first integral of (3): sig:grad
  elvect = 0.;
  Tr.SetIntPoint(&ip);
  real_t wt = Tr.Weight();
  real_t w = ip.weight * wt;
  asym_stress(el,Tr,ip,w,disp,dam,lambda,mu,dshape,gdshape,strain,limit,
      mlimit,eval,eigns,evec,sig);
  AddMult(gdshape, sig, res);

  // compute the second integral of (3): load contribution
  // integration loop
  const int nip = IntRule->GetNPoints();
  for (int i = 0; i < nip; i++)
  {
    const IntegrationPoint &ipl = IntRule->IntPoint(i);
    Tr.SetIntPoint(&ipl);
    real_t wl = ipl.weight * wt;
    // get volumic load
    load.Eval(f, Tr, ipl);
```

```
28      // get form function in physical space
29      el.CalcPhysShape(Tr, shape);
30      // add -f:v
31      AddMult_a_VWt(-wl, shape, f, res);
32    }
33  }// end of AssembleElementVector method
```

Listing 48

In this implementation it is assumed that the elements use a polynomial interpolation basis of order 1. Thus the Jacobian of the transformation of these elements is a constant over one element. The "Weight" method is then called only once in line 13 and reused in lines 14 and 25 for all integration points. This reduces the computational overhead, but makes the encoding less general. In the same spirit, the order of integration is hard coded for the two integrals of (3). It provides a way to precompute the fixed field d and $\mathbf{f}$ just once before entering the non-linear loop, at the chosen integration points, and store this information in arrays controlled by the "QuadratureFunctionCoefficient" and "VectorQuadratureFunctionCoefficient " classes. A call to the "Eval" method of these classes simply finds the information in an array according to the integration point index. With a field instance ("ParGridFunction"), a call to the "GetValue" method will compute the same information at the integration point, which is more computationally expensive. So the first integral uses an order 1 integration (which uses the given "ip" integration point) and the second integral uses an order 2 integration (which leads to a loop on the given "ir"/"IntRule" integration rule).

The first integral is mainly computed by calling an extra "asym_stress" function that computes σ ("sig") for the "ip"integration point. It uses (6), (9), (10), (11), (12) and the derivation of (8) with the same test for non-zero damage and non-zero invariants as with FEniCSx:

```
1  void asym_stress(const FiniteElement &el, ElementTransformation &Tr, const
       IntegrationPoint &ip, const real_t &w,const DenseMatrix &disp,
       QuadratureFunctionCoefficient &dam, Coefficient &lambda, Coefficient &mu
       , DenseMatrix &dshape,DenseMatrix &gdshape, DenseMatrix &strain, real_t
      &limit, real_t &mlimit, Vector &eval, Vector &eigns,DenseMatrix &evec,
       DenseMatrix &sig)
2  {
3    // get damage, lambda and mu
4    real_t d = dam.Eval(Tr, ip);
5    real_t l = lambda.Eval(Tr, ip);
6    real_t m = mu.Eval(Tr, ip);
7
8    // get gradiant of form function
9    el.CalcDShape(ip, dshape);
```

```
10  // pass it in physical space
11  Mult(dshape, Tr.InverseJacobian(), gdshape);
12  // computes gradiant of displacement
13  MultAtB(disp, gdshape, strain);
14  // set as strain
15  strain.Symmetrize();
16
17  // if non null damage
18  if (d > 0.)
19  {
20    // computing the invariants
21    real_t I1 = strain(0, 0) + strain(1, 1);
22    real_t I2 = strain(0, 1) * strain(0, 1) - strain(0, 0) * strain(1, 1);
23
24    // if non null tensor
25    if (I1 > limit || I2 > limit || I1 < mlimit || I2 < mlimit)
26    {
27      // computing the eigen values
28      real_t delta = I1 * I1 + 4 * I2;
29      MFEM_ASSERT(delta > mlimit, "!!! Strain tensor rank deficient !!!")
30      real_t r = sqrt(max(0., delta));
31      eval[0] = (I1 + r) / 2.;
32      eval[1] = (I1 - r) / 2.;
33
34      // modifying values of alpha's based on the signs of eigenvalues
35      real_t alpha, alpha1, alpha2;
36      if (eval[0] >= 0) alpha1 = 1.; else alpha1 = 0.;
37      if (eval[1] >= 0) alpha2 = 1.; else alpha2 = 0.;
38      if ((eval[0] + eval[1]) >= 0) alpha = 1.; else alpha = 0.;
39
40      // genral case : not pure traction and full damage (i.e. alpha_i=1=
            damage)
41      if (!((d == 1.) && (alpha == 1) && (alpha1 == 1) && (alpha2 == 1)))
42      {
43        // computing the eigenvectors
44        if (fabs(strain(1, 0)) > limit)
45        {
46          evec(0, 0) = eval[0] - strain(1, 1);
47          evec(0, 1) = eval[1] - strain(1, 1);
48          evec(1, 0) = evec(1, 1) = strain(1, 0);
49          real_t norms[2];
50          evec.Norm2(norms);
51          evec(0, 0) /= norms[0];
52          evec(1, 0) /= norms[0];
53          evec(0, 1) /= norms[1];
54          evec(1, 1) /= norms[1];
55        }
56        else
57        {
58          evec(0, 0) = evec(1, 1) = 1.;
59          evec(1, 0) = evec(0, 1) = 0.;
60        }
61        real_t temp = 2. * m * w; // E/(1+nu) multiply here by weight
62        real_t gamma = 0.5 * l / m;  // nu/(1-2*nu)
63
64        real_t c = 1 - alpha * d;
65        real_t c1 = 1 - alpha1 * d;
```

```
66          real_t c2 = 1 - alpha2 * d;
67          real_t D0 = temp * (c1 + gamma * c);
68          real_t D1 = temp * gamma * c;
69          real_t D2 = temp * (c2 + gamma * c);
70          eigns[0] = D0 * eval[0] + D1 * eval[1];
71          eigns[1] = D1 * eval[0] + D2 * eval[1];
72          MultADAt(evec, eigns, sig);
73        }
74        // pure traction and full damage: alpha_i=1=damage
75        else
76        {
77          sig = 0.;  // stress=0
78        }
79      }
80      // null strain tensor
81      else
82      {
83        sig = 0.;  // stress=0
84      }
85    }
86    // if null damage
87    else
88    {
89      // simple linear stress: integration of E:strain
90      // multiply here by weight
91      real_t m2plw = w * (2 * m + l);
92      real_t lw = l * w;
93      sig(0, 0) = m2plw * strain(0, 0) + lw * strain(1, 1);
94      sig(1, 1) = m2plw * strain(1, 1) + lw * strain(0, 0);
95      sig(1, 0) = sig(0, 1) = w * m * (strain(0, 1) + strain(1, 0));
96    }
97  }
```

Listing 49

The "AssembleElementGrad" method uses the same ingredients, but implements the mathematical expression of the derivative of (3). It uses about 200 lines of code.

With this class it is now possible to create the formulation object as follows:

```
ParNonlinearForm F(&space);
auto pdi = new damIntegrator(lambda_func, mu_func, qfcd, ip1, ir2, qfcl);
F.AddDomainIntegrator(pdi);
```

Listing 50

with :

- "lambda_func" and "mu_func" objects of type "mfem::PWConstCoefficient " corresponding to the Lamè coefficients λ and μ (section 7)
- "ip1" the integration point related to order 1 integration

- "ir2" the integration rule related to order 2 integration
- "qfcd" the damage field constructed in section 7.1 and pres computed at the integration point "ip1" in a "QuadratureFunctionCoefficient" object
- "qfcl" the load (f) (see section 9.2 listing 68)

Alternatively, as mentioned above, the load in (3) is constant during the non-linear loop, so its integration can be calculated only once using:

```
Vector load_rhs(space.GetTrueVSize());
ParLinearForm b(&space);
b.AddDomainIntegrator(new VectorDomainLFIntegrator(qfcl));
b.Assemble();
// store in load_rhs
b.ParallelAssemble(load_rhs);
// reset to zero dirichlet so that they don't mess up the computation
for (auto idx : ess_tdof_list) load_rhs[idx] = 0.;
```

Listing 51

where "qfcl" is the load (f) (see section 9.2 listing 68) and " ess_tdof_list " is the list of Dirichlet boundary condition DOFs (see section 9.1 listing 62). The "VectorDomainLFIntegrator" class corresponds exactly to the second integral of (3) and can therefore be used directly with a linear form (here "b") to assemble the corresponding vector "load_rhs", contribution to the residual formulation. This vector is then passed to the Newton solver. However, since Dirichlet dofs rows of the residual vector are nullified during its construction in the non-linear loop before "load_rhs" is subtracted from it, the user must also nullify Dirichlet dofs rows of "load_rhs". In this way, the final residual vector corresponding to (3) got zero terms at Dirichlet dofs, as expected. Using this vector means that the "AssembleElementVector" method of the "damIntegrator" class now only calculates the first integral of (3) and lines 18 to 32 of the listing 48 must be removed in this case.

8.2.2 Automatic Differentiation

The miniapps/autodiff subdirectory of MFEM contains an example that illustrates the automatic differentiation (AD) of arbitrary functions implemented in C++. To get more insite on the AD technique, look at the example on the MFEM site or on the Internet. The AD technique appears to be still in development, as the miniapps/autodiff relies on files not provided by the MFEM 4.7.0 version. By attempting to evaluate the gain in implementation work

provided by this technique compared to standard MFEM usage, this study will be relatively fair compared to FEniCSx, which provides user-friendly implementation of formulations.

In order to use AD, the following headers are required: admfem.hpp, taddensemat.hpp and tadvector.hpp. For AD, the principle is similar to the strategy used in section 8.1.4 with FEniCSx full UFL differentiation. The idea is to start from the potential (1) by declaring the following class:

```
/// Functor used in automatic differentiation that compute the asymmetric
    traction/compression damaged elasticity potential.
/// State vector strain store strain tensor in the following order:
///  0     1      2      3
/// eps11, eps21, eps12, eps22
/// and parameters provided in vector vparam are Lame and damage
template <typename TDataType, typename TParamVector, typename TStateVector,
    int state_size, int param_size>
class Potential
{
  double limit=1.e-12, mlimit=-1.e-12;
  public:
  TDataType operator()(TParamVector &vparam, TStateVector &strain)
  {
    real_t d = vparam[2];  // Damage

    // computing the invariants
    TDataType I1 = strain[0] + strain[3];
    TDataType I2 = strain[1] * strain[2] - strain[0] * strain[3];

    // non null tensor
    if (I1 > limit || I2 > limit || I1 < mlimit || I2 < mlimit)
    {
      // computing the eigen values
      TDataType delta = I1 * I1 + 4. * I2;
      MFEM_ASSERT(delta > mlimit, "Strain tensor is rank deficient !!!")
      TDataType r = sqrt(delta);
      TDataType ev1 = (I1 + r) / 2.;
      TDataType ev2 = (I1 - r) / 2.;

      // alpha's
      real_t alpha, alpha1, alpha2;
      if (ev1 >= 0) alpha1 = 1.; else alpha1 = 0.;
      if (ev2 >= 0) alpha2 = 1.; else alpha2 = 0.;
      if ((ev1 + ev2) >= 0) alpha = 1.; else alpha = 0.;

      // return the asymetric potential
      return I1 * I1 * (1. - alpha * d) * vparam[0] / 2. +
      vparam[1] * ((1 - alpha1 * d) * ev1 * ev1 + (1. - alpha2 * d) * ev2 *
          ev2);
    }
    // null tensor
    else
    {
      // switch to simple linear potential to avoid singular second
          derivative (due to sqrt at zero).
```

```
      return (1 - d) * (I1 * I1 * vparam[0] / 2. + vparam[1] * (strain[0] *
          strain[0] + strain[3] * strain[3] +
      strain[1] * strain[1] + strain[2] * strain[2]));
    }
  }
};
```

Listing 52

Unlike the usual C++ implementation, this function object manipulates variables with their type (here provided has a "TDataType" generic type), which contains their value and derivative. This makes it possible to calculate expressions and their derivatives by overloading mathematics operation for this type. Thus, potential first and second AD derivatives are used directly in the "AssembleElementVector" and "AssembleElementMatrix" methods of the slightly modified "damIntegrator" class. This class definition only change for the private member with:

```
    DenseMatrix hess;
    Vector vparam;
    QFunctionAutoDiff<Potential, 4, 3> ad_potentiel;
```

Listing 53

replacing following lines in listing 46:

```
17    DenseMatrix evec, potentiel_sderivative, dedeps, M;
18    Vector eval, eigns;
```

Listing 54

and by consequence in the constructor following lines :

```
    hess(4, 4),
    vparam(3),
```

Listing 55

replace lines in listing 47:

```
7   evec(dim), potentiel_sderivative(3), dedeps(3),
8     M(3), eval(dim),eigns(dim),
```

Listing 56

The data member "ad_potentiel" is an instance of the template class "mfem::QFunctionAutoDiff", which provides an evaluation of the first derivatives and the Hessian of a templated scalar function provided as a functor. In the present it is the object function "Potential" that is in use.

The "AssembleElementVector" is the same as the standard version presented in listing 48 except that now it calls the following new "asym_stress" function:

```
1  void asym_stress(const FiniteElement &el, ElementTransformation &Tr, const
       IntegrationPoint &ip, const real_t &w, const DenseMatrix &disp,
       QuadratureFunctionCoefficient &dam, Coefficient &lambda, Coefficient &mu
       , DenseMatrix &dshape, DenseMatrix &gdshape, DenseMatrix &strain, Vector
       &vparam, mfem::QFunctionAutoDiff<Potential, 4, 3> &ad_potentiel,
       DenseMatrix &sig)
2  {
3
4    // get damage, lambda and mu
5    real_t d = dam.Eval(Tr, ip);
6    real_t l = lambda.Eval(Tr, ip);
7    real_t m = mu.Eval(Tr, ip);
8
9    // get gradiant of form function
10   el.CalcDShape(ip, dshape);
11   // pass it in physical space
12   Mult(dshape, Tr.InverseJacobian(), gdshape);
13   // computes gradiant of displacement
14   MultAtB(disp, gdshape, strain);
15   // set as strain
16   strain.Symmetrize();
17
18   // if non null damage
19   if (d > 0.)
20   {
21     vparam[0] = l * w;
22     vparam[1] = m * w;
23     vparam[2] = d;
24     Vector vstrain(strain.GetData(),4);
25     Vector vsig(sig.GetData(),4);
26     ad_potentiel.Grad(vparam, vstrain, vsig);
27   }
28   // if null damage
29   else
30   {
31     // simple linear stress: integration of E:strain
32     // multiply here by weight
33     real_t m2plw = w * (2 * m + l);
34     real_t lw = l * w;
35     sig(0, 0) = m2plw * strain(0, 0) + lw * strain(1, 1);
36     sig(1, 1) = m2plw * strain(1, 1) + lw * strain(0, 0);
37     sig(1, 0) = sig(0, 1) = w * m * (strain(0, 1) + strain(1, 0));
38   }
39 }
```

Listing 57

Compared to the standard version, the use of AD greatly simplifies the code. Lines 20-84 of listing 49 are in listing 57, replaced by lines 21-26. By simply calling the "Grad" method of the "mfem::QFunctionAutoDiff" class, all

the manual calculation of (10), (11), (12) and derivation of (8) is done by differentiating the potential (1) expressed in the "Potential" class.

Similarly, the "AssembleElementGrad" method is greatly simplified by using AD. In terms of line count, the new version is only 40% the size of the standard version. In algorithmic terms, using the "Hessian" method of the "mfem::QFunctionAutoDiff" class removes all the complex and error-prone mathematical implementation associated with the second derivative of 1. The code is shown below:

```
void AssembleElementGrad(const FiniteElement &el, ElementTransformation &Tr,
     const Vector &elfun, DenseMatrix &elmat)
{
  // reset result matrix
  elmat.SetSize(nd*dim);
  elmat = 0.0;
  // local matrices and vector setting
  Vector gdshapex(vl.GetData(), nd);
  Vector gdshapey(vl.GetData()+nd, nd);
  disp.UseExternalData(elfun.GetData(), nd, dim);
  mgdshapex.UseExternalData(gdshapex.GetData(), nd, 1);
  mgdshapey.UseExternalData(gdshapey.GetData(), nd, 1);
  Vector vstrain(strain.GetData(),dim*dim);
  // get weight
  Tr.SetIntPoint(&ip);
  real_t w = ip.weight * Tr.Weight();
  // get lambda, mu
  real_t l = lambda.Eval(Tr, ip);
  real_t m= mu.Eval(Tr,ip);
  // get gradiant of form function
  el.CalcDShape(ip, dshape);
  // pass it in physical space
  Mult(dshape, Tr.InverseJacobian(), gdshape);
  // set B
  B.SetSubMatrix(0, 0, mgdshapex);
  B.SetSubMatrix(nd, 1, mgdshapey);
  B.SetSubMatrix(0, 2, mgdshapey);
  B.SetSubMatrix(nd, 2, mgdshapex);
  // get damage
  real_t d = dam.GetValue(Tr);
  // if non null damage
  if (d>0.)
  {
    // to secure matrix non singular use limiter for damage
    d = min(d, 1. - limit);
    // computes strain
    MultAtB(disp, gdshape, strain);
    strain.Symmetrize();
    // compute hessian
    vparam[0] = l; vparam[1] = m; vparam[2] = d;
    ad_potentiel.Hessian(vparam, vstrain, hess);
    // reorder into hook
    for (int i = 0; i < 3; ++i)
       for (int j = 0; j < 3; ++j)
            hook(i, j) = hess(i + 2 * (i % 2), j + 2 * (j % 2));
```

```
      hook(2, 2) = 0.5 * (hook(2, 2) + hess(1, 2));
   }
   // if null damage
   else
   {
      // simple linear hook
      hook=0.;
      hook(0, 0) = hook(1, 1) = 2 * m + l;
      hook(1, 0) = hook(0, 1) = l;
      hook(2, 2) = m;
   }
   // compute elementary matrix
   Mult(B, hook, C);
   AddMult_a_ABt(w, C, B, elmat);
}
```

Listing 58

For the formulation object itself, the same code can be used as in the listing 50. The use of AD is fully transparent at this level in this example.

8.3 Assembly

The task of assembling an elementary matrix or vector provided by the formulation described in the previous sections is launched either explicitly or implicitly in more global library functions. In all cases, the two libraries have a set of classes that provide the distributed vector and matrix concepts to create the algebraic system populated by these assembler routines.

8.3.1 MFEM

For non-linear problems, assembly is done in "mfem::ParNonlinearForm::Mult "/"mfem::ParNonlinearForm:: GetGradient" methods that call the "AssembleElementVector" / "AssembleElementGrad" (e.g. listing 48 or its AD version and 58 or its standard version) for all elements of a domain to get elementary vectors / matrices and assemble them into a residual vector / Jacobian matrix. These methods are called inside the non-linear loop of the "mfem::NewtonSolver::Mult" method (the function that solves the non-linear problem). The user only needs to assign the non-linear form ("F" in the listing 50) to the Newton solver instance and call the "mfem::NewtonSolver ::Mult" method to get the linear system created and updated during the non-linear loop.

For linear problems, the assembly is computed more explicitly by calling the "mfem::ParLinearForm::Assemble" / "mfem::ParBilinearForm::Assemble " methods with a linear / bi-linear formulation object. The user only needs to assign an ad hoc integrator to the formulation object to obtain the elementary vectors / matrices during the assembly loop in these methods.

As an alternative to the assembly in large vectors and matrices, MFEM offers among others the so-called "partial assembly" and the "matrix-free assembly" (especially interesting when using GPUs, see [2]). Both were not tested in this study, but the documentation contains the following comments on them:

- Compared to the "full assembly" strategy, the "partial assembly" strategy, which computes and stores data only at quadrature points, results in significantly faster computations and less memory consumption.
- Compared to the "full assembly" strategy, the "matrix-free assembly", which computes all actions on the fly without significant storage, is also significantly faster, but currently slower than the "partial assembly" strategy due to the increased number of computations. However, in the case of operators that need to be reassembled frequently, this level of assembly could be faster than partial assembly by skipping all reassembly steps.

The general assembly loops are summarised in the algorithm 2.

8.3.2 FEniCSx

With the Python API, in non-linear, everything is encapsulated in a "dolfinx.nls.petsc.NonlinearProblem" object. At construction time, it takes "F" (listing 25) and "J" (listing 26) as arguments. The non-linear problem object is passed to the "dolfinx.nls.petsc.NewtonSolver" object, which is used by the user to solve the problem. As in MFEM, the linear system is created and updated during the non-linear loop by the resolution method.

In the linear case, as in the non-linear case, everything can be encapsulated in a single object of type "dolfinx.fem.petsc.LinearProblem". It takes as argument at least the bi- and linear form describing the problem. The call to its "solve" method causes the system matrix and vector to be assembled on the fly before it is resolved. Otherwise, the following illustrates the implementation of the assembly step of a "a" bi-linear form and a "L" linear form

Algorithm 2 MFEM general assembly loop structure ("full assembly" strategy) with $\mathfrak{F}_i$, $\mathfrak{E}$ respectively the set of integral ids (integrator) for the treated formulation corresponding to current element i and the set of elements. AssembleElementXXX$_{id}$(data$_i$) computes, for an element i (described by data$_i$), the elementary "matrix,vector" with a quadrature loop corresponding to integral id. "XXX" stands for "Vector", "Grad" or "Matrix". Ae_i, be_i and A, b are the elementary "matrix,vector" of element i and the global "matrix,vector" respectively.

for $i \in \mathfrak{E}$ **do**
  for $id \in \mathfrak{F}_i$ **do**
    $Ae_i, be_i \leftarrow Ae_i, be_i +$ AssembleElementXXX$_{id}(data_i)$
  end for
  After possible treatment, assemble Ae_i, be_i into A, b
end for

created from UFL expressions (by the instruction of the listing 27) under Dirichlet boundary conditions:

```
1 A = dolfinx.fem.petsc.assemble_matrix(a, bcs=bdirichlet)
2 A.assemble()
3 b = dolfinx.fem.petsc.assemble_vector(L)
4 dolfinx.fem.petsc.apply_lifting(b, [a], bcs=[bdirichlet])
5 b.ghostUpdate(addv=PETSc.InsertMode.ADD, mode=PETSc.ScatterMode.
    REVERSE)
6 dolfinx.fem.petsc.set_bc(b, bdirichlet)
```

Listing 59

where:

- " bdirichlet " is the Dirichlet boundary condition, defined e.g. by listing 63
- "A" and "b" are respectively the generated matrix and vector of the linear algebra system to be computed to solve the problem.
- lines 4 and 6 are the calls related to the Dirichlet boundary condition treatment as explained in section 9.1 (non-zero contribution and setting).
- lines 2 and 5 are calls to functions that synchronise information between processes.

With the FEniCSx C++ API, in linear and non-linear, the user is responsible for explicitly implementing the assembly step that can be summarized as follows with Dirichlet boundary conditions for a linear formulation (like in listing 59):

```
la::Vector<scalar> b(V->dofmap()->index_map,V->dofmap()->index_map_bs());
auto field_array=x->x()->mutable_array();
b.set(0);
fem::assemble_vector<scalar>(b.mutable_array(), *lin_form);
fem::apply_lifting<scalar, scalar_dolf>(b.mutable_array(), {bilin_form}, {{
    bclamp,bimp}}, {field_array}, -1.);
b.scatter_rev(std::plus<scalar>());
```

Listing 60

with

- the scalar types("scalar","scalar_dolf") described in listing 1
- the linear and bi-linear formulation ("lin_form","bilin_form") and space ("V") described in listing 16
- the "dolfinx::fem::DirichletBC<scalar>" ("bclamp" and "bimp") objects described in listing 64
- the field ("x") described in listing 15 that provides imposed values at Dirichlet DOFs.

In particular, the user must not forget to:

- Add the Dirichlet boundary condition contribution if non-zero values are imposed ("dolfinx::fem::apply_lifting").
- Appropriately sum values at the ghost DOF in parallel ("dolfinx::la::Vector::scatter_rev").

For the bi-linear formulation the assembly process with Dirichlet boundary condition can be as follows:

```
// Matrix creation and initialization
auto A = la::petsc::Matrix(fem::petsc::create_matrix(*bilin_form), false);
Mat A_ = A.mat();
MatZeroEntries(A_);
// Assemble formulation in A Matrix
fem::assemble_matrix(la::petsc::Matrix::set_block_fn(A_, ADD_VALUES),
*bilin_form, {bclamp,bimp});
// Comunicate to treat ghost dofs (PETSc)
MatAssemblyBegin(A_, MAT_FLUSH_ASSEMBLY);
```

```
MatAssemblyEnd(A_, MAT_FLUSH_ASSEMBLY);
// Set Dirichlet DOFs diagonal term to 1
fem::set_diagonal<scalar>(la::petsc::Matrix::set_fn(A_, INSERT_VALUES), *V,
{bclamp, bimp});
// Comunicate to treat ghost dofs (PETSc)
MatAssemblyBegin(A_, MAT_FINAL_ASSEMBLY);
MatAssemblyEnd(A_, MAT_FINAL_ASSEMBLY);
```

Listing 61

In non-linear, with Newton solver, these instructions have to be implemented in the operators provided by the "setF" /"setJ" method of "dolfinx::nls::petsc::NewtonSolver".

The general assembly loops are summarised in the algorithm 3.

Algorithm 3 FEniCSx general assembly loop structure with $\mathfrak{D}$, $\mathfrak{E}_{id}$ respectively the set of integral domain ids in the treated formulation and the set of elements related to the current integral (may be a subset of the local domain). $\text{Kernel}_{id}(\text{data}_i)$ computes, for an element i (described by data_i), the elementary "matrix,vector" with a quadrature loop corresponding to integral id. Ae_i, be_i and A, b are the elementary "matrix,vector" of element i and the global "matrix,vector" respectively.

for $id \in \mathfrak{D}$ **do**
  for $i \in \mathfrak{E}_{id}$ **do**
    $Ae_i, be_i \leftarrow \text{Kernel}_{id}(\text{data}_i)$
    After possible treatment, assemble Ae_i, be_i in A, b
  end for
end for

8.3.3 Performances

Apparently, in FEniCSx, compared to MFEM, there is no specific strategy to partially assemble the system and use operator applications to bypass full matrix exploitation during system resolution. And in FEniCSx, there is no GPU or multithreading used in the assembly task, as there might be with MFEM.

So, in the example in section 2.1.1, the assembly in the two codes was compared using only single-threaded code (although distributed over several cores by the MPI library), without any GPU usage.

To add profiling measures to track the assembly task, one would need to hack MFEM library which is out of scope of this study. For this reason,

only vector and matrix generation, part of the system assembly task, will be analysed in this work. Figure 22 shows the creation of elementary vectors and matrices with the different strategies used to implement the formulation. The MFEM curves show a simple measure of the elapsed time between the entry and exit of the "AssembleElementVector" and "AssembleElementGrad" methods of the "damIntegrator" class for the standard and AD strategies with different computation variants. The FEniCSx curves show the measure of the elapsed time between the entry and exit of the kernel function generated by ffcx (hacked after generation to provide this measure) for the strategy of section 8.1.2, 8.1.3, 8.1.4 and 8.1.5. Both measures are assumed to be equivalent in the context of this test case (simple element and formulation), but some details differ. In particular, in FEniCSx, part of the Dirichlet boundary condition treatment is done by the function " apply_lifting " (see listing 59 or 60), which calls the same kernel function as for matrix construction to obtain the coupling terms. Thus, for the creation of elementary matrices, part of the FEniCSx curves is linked to the treatment of Dirichlet boundary conditions, which is not the case in MFEM. Anyway this is the best that can be done without hacking the FEniCSx or MFEM library to obtain more fine profiling data.

The curves show that both codes scale perfectly thanks to Metis/ParMetis, which provides load-balanced domains and a rather thin damage band that does not induce unbalanced computations (see section 7.1).

For MFEM, the following calculation variants are tested:

- "MFEM h1": Use the generic "mfem::H1_FECollection" for the elements, with the volume load being calculated at each non-linear iteration when calculating the residual (i.e. in the "AssembleElementVector" method). Also, a rather generic coding is used, which calculates the weight of the transformation at each integration point, as it is not constant in high order. In this way, this variant aims to get closer to encoding higher order elements.

- "MFEM": Use the specific "mfem::LinearFECollection" for the elements, a unique weighting (transformation) calculation for all integration points (see listing 48), with the volume load calculated at each non-linear iteration when calculating the residual (i.e. in the "AssembleElementVector" method). This variant is chosen as the reference because it is the closest to FEniCSx.

- "MFEM Fcst": Use the specific "mfem::LinearFECollection" for the elements, with the volume load calculated only once outside the non-linear resolution (see listing 51), so not in "AssembleElementVector". Regarding profiling, the elementary volume load unique calculation is counted in the "Create elementary vectors" curve. However, the subtraction of the calculated vector from the residual vector during non-linear loops is included in the "non-linear" curve (figure 24c and 24d).

These variants show that using the generic "mfem::H1_FECollection" for the elements and sytematic weight of the transformation computation is costly in the context of this test case. "MFEM h1" is 2.7 times slower than "MFEM" for vector computation and up to 1.45 times slower for matrix computation. This is related to shape function (and gradient) computations, which in the general case are obtained by solving a dense system: a factorisation and a backward/forward subtitution at each request for shape function values. With the "mfem::LinearFECollection" these shape functions (and gradiants) are hard coded so they can be computed much faster (but not pre-calculated like ffcx does in FEniCSx). The cost of the "Weight" method, which is calculated once or 4 times (general case) per element, also has an impact. Because behind this method there is the Jacobian computation, which does costly matrix operations. On the other hand, the "MFEM Fcst" variant has no effect on the creation of elementary matrices, but is 1.2 times faster than "MFEM" for elementary vectors. Its logical as load elementary computation is done only once with this variant, and its elementary contribution is postponed at system level by an algebraic operation (which unfortunately is not tracked in an isolated counter). So this 1.2 time gain would certainly be less if all contributions could be collected. Nevertheless, this "MFEM Fcst" variant can be considered in the context of this test case, where the load is applied to the wall domain, as an efficient alternative (not proposed by FEniCSx). It would certainly have more effect with higher order elements (non-constant Jacobian transformation, more integration points, ...).

For "MFEM" reference the AD and standard strategy are equivalent. Only a small overhead (less than 1.06) appears when using AD to create the elementary matrix. In this case, AD replaces the computation of the complex mathematical implementation associated with the second derivative of 1 with 10 ($= \frac{4\times5}{2}$) calls to the potential operator in AD mode differentiation to compute the upper triangular terms of the Hessian. Thanks to the small number of variables to be derived (the 4 strain components), the number of

calls for AD differentiation remains small and almost transparent. The same observation applies to the "MFEM h1" and "MFEM Fcst" variants. In these cases, AD and the standard strategy give ratio curves for elementary vector calculations that overlap. For elementary matrix generation, AD is slightly slower than the standard strategy, as in the reference variant.

FEniCSx is 1.6 to 2.2 times faster than "MFEM" reference for elementary vector calculations. For elementary matrix calculations, FEniCSx is 1.2 times slower or 1.06 times faster than "MFEM" reference, depending on the formulation strategy used. This gain with FEniCSx kernels can be explained by the work of the ffcx compiler:

- It computes shape function values, shape function gradient and weight at integration points. These values are stored by ffcx in static arrays in the kernel code, which no longer needs to compute them during elementary calculations.
- It develops the mathematical expression into a series of elementary operations associated with local variables that are concatenated sequentially. Some small loops are also coded, but there is no explicit matrix-vector or matrix-matrix operation as in MFEM. This helps the C compiler with its aggressive compilation optimisation, and the kernels are immediately floating-point efficient with this test case.

Note that, as already mentioned, FEniCSx profiling takes into account calls to the matrix kernel when dealing with the Dirichlet contribution, which MFEM does not, so the performance of FEniCSx compared to MFEM is certainly even better in the context of this test case.

The complete assembly task of the matrix and vector of the linear system can only be viewed in FEniCSx, by adding measure in lambda function used by"setF" and "setJ" method of "dolfinx::nls::petsc::NewtonSolver". In both methods, the profiling measures the full assembly task (elementary computation and addition of the elementary matrix/vector to the system matrix/vector) and the Dirichlet treatment. Figure 23 shows that the full assembly task (crea+ass curves) scale relatively well. Another full assembly task measure is also shown in this figure, but without kernel profiling (crea no prof+ass curves). This shows that the profiling of kernels has a small effect on the profiling of the functions that call them. By dividing the curves in Figure 22

by the curve in Figure 23, it can be seen that elementary creation represents only 22 to 37% of the full assembly task with Dirichlet treatment. And since these ratios are relatively stable with the number of processes, it can be assumed that the assembly task itself and the Dirichlet treatment also scale well.

As an alternative to profiling using times measured at specific locations, the Valgrind callgrind tool can be used. Running the simulation with this tool is much slower, so it was only used on an unrefined mesh, producing the results (focusing on vector/matrix construction) shown in figures 19 and 20. From these figures, using the callgrind instruction counts, one can obtain

	handmade	UFL	sympy	sympy sym
vector	$\frac{10\ 526\ 934\ 750}{3\ 171\ 489\ 252} \approx 3.32$	$\frac{10\ 526\ 934\ 750}{2\ 873\ 391\ 266} \approx 3.66$	$\frac{10\ 526\ 934\ 750}{2\ 616\ 863\ 101} \approx 4.02$	$\frac{10\ 526\ 934\ 750}{2\ 638\ 397\ 649} \approx 3.99$
matrix	$\frac{9\ 940\ 681\ 357}{8\ 372\ 947\ 591} \approx 1.19$	$\frac{9\ 940\ 681\ 357}{8\ 377\ 993\ 751} \approx 1.19$	$\frac{9\ 940\ 681\ 357}{7\ 036\ 416\ 905} \approx 1.41$	$\frac{9\ 940\ 681\ 357}{6\ 906\ 577\ 463} \approx 1.44$

Table 4: Ratio of "MFEM" reference and FEniCSx (different strategies) callgrind instruction counts.

the same kind of ratio as in figures 22b and 22d, which are given in table 4. The ratios obtained in this way are larger[3] than those obtained by measuring elapsed time, but give the same trends. They show that FEniCSx performs better in this test case, especially for vector creation. These gains are even more significant if we consider the elementary computational task not by simulation but by non-linear iteration. In fact, with FEniCSx we iterate more. With unrefined mesh FEniCSx and MFEM iterates 6 and 4 times respectively. So the ratio of the table 4 must be multiplied by $\frac{6}{4} = 1.5$, which gives a gain per non-linear iteration from 1.79 to 6.03 with FEniCSx compared to MFEM.

Finally, the different FEniCSx strategies of the 8.1.2, 8.1.3, 8.1.4 and 8.1.5 sections can be compared using both profiling (elapsed time and number of callgrind instructions). From slower to faster strategies are handmade (8.1.5), UFL (8.1.4), sympy (8.1.2), and sympy symmetric (8.1.3). First, we can see that the number of lines of C code (given in the captions of the figure 19), which more or less represents the set of instructions generated by ffcx,

[3]Valgrind counting can be abused in C++ when using a function object (which is the case in FEniCSx regarding kernel wrapping), so only tandency needs to be considered.

follows roughly this order (fewer lines, fewer instruction counts). But the number of lines of C code is not the only parameter, as better test locations or cheaper algebraic instructions can lead to more lines but fewer instructions (as with vector generation for handmade and UFL strategies). But clearly sympy strategies (symmetric or not) seem to be a bit more efficient. We can imagine that sympy expressions are then simplified as they pass through the "simplify" method, thus producing smaller UFL expressions compared to the strategy of sections8.1.4 and 8.1.5. These simplifications only apply to the stress expression, but since the Jacobian is derived from the stress, it seems that the maxtrix generation is also affected by these sympy simplifications. This suggests that if sympy can simplify long mathematical expressions, it is worth trying to use it instead of just using UFL expressions. As for the handmade version, it is clearly not implemented optimally. Sure, working on it can lead to better performance, but why do the work of UFL or Sympy?

9 Boundary condition(point 5)

9.1 Dirichlet

The Dirichlet boundary condition can be treated in several ways. Here are 3 classic approaches:

1. Eliminate the Dirichlet DOFs from the system and add the non-zero imposed DOF contribution to the right-hand term of the system.(reduce system size)

2. Add Lagrange multipliers to impose Dirichlet dofs. Imposed values are set directly in the right-hand terms of the system on associated Lagrange multipliers DOF. (increase system size)

3. Change the system matrix row and column associated with the Dirichlet DOFS so that the corresponding sub-block is an identity matrix. Imposed values are set directly in the solution or right-hand terms of the system. (system size is unchanged).

Both libraries use the identity approach (3 above) in most of the cases presented in their documentation. The modification of matrix terms is almost transparent to the user. The user is simply responsible for constructing the

list of DOFs of type Dirichlet and setting the imposed value in the solution or the right-hand terms of the system.

With MFEM, a simple mechanism to generate the list of Dirichlet DOFs is to use the physical properties of the mesh ("pmesh") and the discretisation space("space"):

```
// array to mark prop of interest
Array<int> ess_bdr(pmesh->bdr_attributes.Max());
ess_bdr = 0;
ess_bdr[XX] = 1;
ess_bdr[YY] = 1;
...
// array to store dof of interest
Array<int> ess_tdof_list;
// get list of dof
space.GetEssentialTrueDofs(ess_bdr, ess_tdof_list);
```

Listing 62

The list " ess_tdof_list " is then made available to the non-linear form using the "mfem::ParNonlinearForm::SetEssentialTrueDofs" method or when creating the system using the "mfem::ParBilinearForm::FormLinearSystem" method of the bi-linear form. The solution vector is modified by imposed value before the resolution.

With FEniCSx, the following Python code uses coordinates to generate the list of Dirichlet DOFs using the mesh ("domain") and discretisation space ("space"):

```
clamped_nodes=mesh.locate_entities(domain, 0, lambda x: np.
    isclose(x[0], 0.))
clamped_dofs=fem.locate_dofs_topological(space,0,clamped_nodes
    )
bclamp=fem.dirichletbc(np.array([0.,0.]),clamped_dofs,space)
imp_nodes=mesh.locate_entities(domain, 0, lambda x: np.isclose
    (x[0], 1.))
imp_dofs=fem.locate_dofs_topological(space,0,imp_nodes)
bimp=fem.dirichletbc(np.array([0.01,0.]),imp_dofs,space)
bdirichlet=[bclamp,bimp]
```

Listing 63

In this example the 2D vector value at dof is set to " [0.,0.] " and " [0.01,0.] " for nodes with zero and one x-coordinate respectively. The " bdirichlet"

is then used when constructing the "dolfinx.nls.petsc.NonlinearProblem" or "dolfinx.fem.petsc.LinearProblem" object. It can also be used directly with the "dolfinx.fem.petsc.apply_lifting" function to add the contribution of non-zero Dirichlet values to the right-hand vector of the system (see for example line 4 of the listing 59). This contribution is the product of the coupling block of the matrix (between the imposed dofs and the free dofs) and the vector of imposed dofs. It can also be used to set the Dirichlet rows of the right-hand side vector of the system (e.g. see line 6 of listing 59) to the imposed Dirichlet values with the "dolfinx.fem.petsc.set_bc" function. The C++ API uses the same principles:

```
const std::vector<std::int32_t> clamped_nodes = mesh::
    locate_entities_boundary(*pmesh, 0, [&eps](auto x) {
  std::vector<int8_t> marker(x.extent(1), false);
  for (std::size_t p = 0; p < x.extent(1); ++p)
  {
    double x0 = x(0, p);
    if (std::abs(x0) < eps) marker[p] = true;
  }
  return marker;
});
const std::vector<std::int32_t> clamped_dofs = fem::
    locate_dofs_topological(*V->mesh()->topology_mutable(), *V->
    dofmap(), 0, clamped_nodes);
auto bclamp = std::make_shared<fem::DirichletBC<scalar>>(u0,
    clamped_dofs);
const std::vector<std::int32_t> imp_nodes = mesh::
    locate_entities_boundary(*pmesh, 0, [&eps](auto x) {
  std::vector<int8_t> marker(x.extent(1), false);
  for (std::size_t p = 0; p < x.extent(1); ++p)
  {
    double x0 = x(0, p);
    if (std::abs(x0-1.) < eps) marker[p] = true;
  }
  return marker;
});
const std::vector<std::int32_t> imp_dofs = fem::
    locate_dofs_topological(*V->mesh()->topology_mutable(), *V->
    dofmap(), 0, imp_nodes);
scalar imp[3] = {0.01, 0.};
auto imp_val = std::make_shared<const fem::Constant<scalar>>(fem
    ::Constant<scalar>(std::span<const scalar>(imp, 2)));
auto bimp = std::make_shared<fem::DirichletBC<scalar>>(imp_val,
    imp_dofs, V);
```

Listing 64

The "dolfinx::mesh::locate_entities_boundary" function returns the list of entities (here the list of nodes) that match a given criterion based on their coordinates. This list is then used with the "dolfinx::fem::locate_dofs_topological" function to obtain the list of DOFs associated with these entities by the space "V". Finally, the objects of the class "dolfinx::fem::DirichletBC<scalar>" are instantiated with these lists of DOFs and the imposed values. The resulting objects "bclamp" and "bimp" are then ready to be used by the "fem::apply_lifting" (listing 60) and "fem::set_diagonal" (listing 61) functions. This last function is responsible for transforming the Dirichlet dofs subblock of "A" into an identity matrix.

9.2 Neumann

As in many FEM libraries, the implementation of this type of loading, which corresponds to an additional term in the weak formulation, corresponds to an assembly task of the right-hand side of the linear or non-linear system. So everything shown in the 8 section can be used to impose the Neumann boundary condition.

In particular, for the example in section 2.1.1 Neumann volume loading in FEniCSx is directly integrated into the formulation "F" (listing 25). The load vector **f** is constructed by interpolating (4) onto a field. In C++ :

```
auto f = std::make_shared<fem::Function<scalar>>(V);
f->name="f";
f->interpolate([](auto x) -> std::pair<std::vector<scalar>,
    std::vector<std::size_t>> {
  size_t nb_col = x.extent(0);
  size_t nb_row = x.extent(1);
  size_t nb_col_v = 2;
  std::vector<scalar> vdata(nb_col_v * nb_row);
  namespace stdex = MDSPAN_IMPL_STANDARD_NAMESPACE::
      MDSPAN_IMPL_PROPOSED_NAMESPACE;
  MDSPAN_IMPL_STANDARD_NAMESPACE::mdspan<
  scalar, MDSPAN_IMPL_STANDARD_NAMESPACE::extents<std::size_t,
       2, MDSPAN_IMPL_STANDARD_NAMESPACE::dynamic_extent>>
  v(vdata.data(), nb_col_v, nb_row);
  for (std::size_t p = 0; p < nb_row; ++p)
  {
```

```
      double r=x(0, p)-0.5;
      double xf = -r * r * r;
      double y = x(1, p) - 0.5;
      v(0, p) = 100000.*(1600. * y * y - 500.) * xf;
      v(1, p) = 0.;
    }
    return {vdata, {nb_col_v, nb_row}};
  });
```

Listing 65

This "dolfinx::fem::Function" is then made available to the formulation via the "coefficients" map (listing 16). Note that the FEniCSx developers, in anticipation of the C++ 23 standard, use multidimensional span "mdspan" with MACRO to get the correct compiler implementation, which makes the syntax a bit hard to read. For the full Python FEniCSx implementation, Numpy is of course used for the interpolation calculations:

```
f=fem.Function(space,name="f")
def f_vol_func(x):
nb_row = len(x[1])
assert nb_row == len(x[0])
nb_col_v = 2
v = np.empty((nb_col_v,nb_row))
xf=100000.
v[0,:]=-xf*np.power(x[0]-0.5,3)*(1600.*np.square(x[1]-0.5)
    -500.)
v[1,:]= 0.
return v
f.interpolate(f_vol_func)
```

Listing 66

Compared to C++, the "f" object is directly linked to the formulation via its expression (listing 25).

With MFEM, still for the example of the section 2.1.1, two approaches can be used. The first one follows the FEniCSx implementation, and the Neumann volume loading is calculated at each non-linear iteration, directly in the "AssembleElementVector" method of the "damIntegrator" class (see end of listing 48). However, since $\mathbf{f}$ is constant during all non-linear iterations, a second approach removes the second integral from the residual vector (3) calculation in "mfem::NewtonSolver::Mult" by considering it as a precomputed vector argument that is subtracted from the assembled residual vector. In

both cases, as to conform to FEniCSx interpolation, a projection of the (4) formula onto a field using the space of the listing 11 is used:

```
VectorFunctionCoefficient vl(dim,[](const Vector &x,Vector &f){
  real_t r = x(0) - 0.5;
  real_t xf = -r * r * r;
  real_t y = x(1) - 0.5;
  f(0) = 100000.*(1600. * y * y - 500.) * xf;
  f(1) = 0.;
});
ParGridFunction load(&space);
load.ProjectCoefficient(vl);
```

Listing 67

For efficiency, in both approaches this field is actually embedded in a class that only stores its values at integration points for a given quadrature:

```
QuadratureSpace qspace2(pmesh, 2);
QuadratureFunction qfl(qspace2, 2);
qfl.ProjectGridFunction(load);
VectorQuadratureFunctionCoefficient qfcl(qfl);
```

Listing 68

10 Solving(point 7)

As already introduced in section 8.3 the linear/non-linear algebra system resolution is handled by high level concepts. However, both FEniCSx and MFEM use mainly external libraries to solve the linear system resulting from the integration of the weak formulation of a problem. All these libraries offer a wide range of parallel linear algebra methods (Krylov, factorisation, multigrid, ...) to solve sparse systems of different nature (symmetric or not, defined or not, positive or not, real or complex, ...). All of them are designed to be used in parallel on modern CPUs (with many cores) or GPUs.

10.1 MFEM

HYPRE is of course interfaced with MFEM (some MFEM contributors are also HYPRE developers), but many other libraries are possible if available (i.e. provided at MFEM compilation time). Among them, we can mention PETSc, which itself gives access to many libraries, AmgX (algebraic

multigrid GPU-accelerated solver library from NVIDIA), Ginkgo (numerical linear algebra library targeting many-core architectures: interfaced with DPC++, OpenMP, CUDA (NVIDIA), HIP(AMD/NVIDIA), ...), SUNDIALS, MUMPS, SUPERLU, UMFPACK, PARDISO, ... And for eigenproblem solving, MFEM can use HYPRE's LOBPCG Eigensolver or SLEPc. In addition, many implementations of well-known solvers are available in the library (e.g. Preconditioned Conjugate Gradient, GMRES, MINRES, ...).

In all cases, the classes used to solve a system are all derived from the "mfem::Solver" class (itself derived from "mfem:Operator"). The pure virtual "Mult" method must be implemented in the derived class. This method takes two arguments, "x" and "y". "y" is the result of "applying" the operator to "x" and corresponds to the resolution of the linear system associated with the solver. The following code illustrate the use of the gradient conjugate solver with the HYPRE Boomeramg algebraic multi grid preconditionner to solve a linear system constructed from "a" and "b" respectively instance of "ParBilinearForm" and "ParLinearForm".

```
1  Vector B, X;
2  OperatorPtr A;
3  a.FormLinearSystem(ess_tdof_list, x, b, A, X, B);
4  HypreBoomerAMG prec;
5  prec.SetElasticityOptions(&space, true);
6  CGSolver lin_solv(MPI_COMM_WORLD);
7  lin_solv.SetRelTol(1e-12);
8  lin_solv.SetMaxIter(2000);
9  lin_solv.SetPreconditioner(prec);
10 lin_solv.SetOperator(*A);
11 lin_solv.Mult(B, X);
12 a.RecoverFEMSolution(X, b, x);
```

Listing 69

In this example:

- Algebraic containers are created in lines 1 (vectors) and 2 (matrix).
- Line 3 the system creation is finished:
  - assign the matrix in "a" (assembly of section 8.3.1) to "A"
  - assign the vector in "b" (assembly of section 8.3.1) to "B"
  - assign the vector in "x" field (listing 13) to "X"

  - Use " ess_tdof_list " (listing 62) to check if Dirichlet boundary condition exist and if yes:
    * treat the Dirichlet boundary condition by adding its contribution to "B"
    * treat the Dirichlet boundary condition by replacing the Dirichlet sub-block of A with an identity block
  - ...

- The HYPRE Boomeramg preconditioner is created in line 4.
- The preconditioner is tuned to remove rigid-body modes introduced by mechanical elasticity (in this example it is an elasticity problem) in line 5.
- In line 6 the linear solver instance (conjugate gradient solver) is created.
- As this solver is an iterative solver, lines 7 and 8 impose some rules on the solver to make it converge (stop iteration).
- The preconditioner and the operator (i.e. A matrix in this example) are attached to the solver in lines 9 and 10.
- Finally, the system is solved considering that "A" is "applied" to "B" giving "X" solution ($\mathbf{X} = \mathbf{A}^{-1}.\mathbf{B}$).
- The "X" solution is then used in line 12 to reset "x" field appropriately using the "mfem::ParBilinearForm::RecoverFEMSolution" method. This line must be used when constructing the linear system with "mfem::ParBilinearForm::FormLinearSystem".

In the same way the resolution of a non-linear problem defined by a "mfem::ParNonlinearForm" instance "F" can be coded as follows:

```
1 F.SetEssentialTrueDofs(ess_tdof_list);
2 Vector X(space.GetTrueVSize());
3 X=0.;
4 HypreBoomerAMG prec;
5 prec.SetElasticityOptions(&space, true);
6 CGSolver lin_solv(MPI_COMM_WORLD);
7 lin_solv.SetRelTol(1e-12);
8 lin_solv.SetMaxIter(2000);
9 lin_solv.SetPreconditioner(prec);
```

```
10 NewtonSolver nl_solv(MPI_COMM_WORLD);
11 nl_solv.SetSolver(lin_solv);
12 nl_solv.SetOperator(F);
13 nl_solv.SetMaxIter(10);
14 nl_solv.SetRelTol(1.e-7);
15 nl_solv.SetAbsTol(5.e-8);
16 Vector zero;
17 x.ParallelProject(X);
18 nl_solv.Mult(zero, X);
19 x.SetFromTrueDofs(X);
```

Listing 70

In this example:

- The list (listing 62) of Dirichlet DOFs (if any) must be assigned to the "F" object. This is done in line 1.
- The solution to the non-linear problem will be stored in a vector "X" set to zero (lines 2 and 3).
- Lines 4 to 9 are exactly the same as lines 4 to 9 of the listing 69, which create and configures a linear solver to be used by the Newton solver.
- Line 10 creates a Newton solver instance.
- The linear solver and the non-linear formulation (i.e. "F") are attached to the Newton solver in lines 11 and 12.
- As this Newton solver is an iterative solver, lines 13 to 15 impose some rules on the non-linear loop to make it converge (stop iteration).
- In this example there is no right hand side vector to provide to the Newton solver (i.e. it may be embedded in "F"). So a dummy empty vector is created in line16, which is interpreted as zero r.h.s. by the Newton solver. Alternatively, as mentioned in the 8.2.1 section, if a constant load is applied to the system, it can be calculated once as in the 51 listing and then the computed "b" vector replaces the "zero" vector in line 18 so that it can be subtracted from the residual vector during non-linear loops.
- Lines 17 and 19 transfer the information from/to the "x" field (listing 13) to/from the "X" vector (required in parallel).

- In line 18 the non-linear system is solved by calling the "Mult" method and its solution (if any) is stored in "X".

Note that in non-linear resolution the relative tolerance parameter (1.e-12), which determines the convergence of the linear solver, has a large influence on the number of non-linear iterations and the computation time. A larger value may increase the number of non-linear iterations, but the elapsed time may decrease because linear convergence is faster. This is an aspect that is used in the Newton–Krylov method.

Note that for the test case in section 2.1.1 the following setting was used for the preconditioner of the linear solver:

```
1 HypreBoomerAMG prec;
2 HYPRE_Solver prec_amg(prec);
3 HYPRE_BoomerAMGSetNumFunctions(prec_amg, 2);
4 HYPRE_BoomerAMGSetAggNumLevels(prec_amg, 0);
5 HYPRE_BoomerAMGSetStrongThreshold(prec_amg, 0.25);
6 prec.SetErrorMode(HypreSolver::ErrorMode::IGNORE_HYPRE_ERRORS);
```

Listing 71

This configuration was found after some testing. It gives the best performance in the context of this test case and allows us to converge on the FEniCSx configuration. In this setting, unlike the listings 69 and 70 , the "SetElasticityOptions" method is not used. Instead, the default setting is used and some additional tuning is done by constructing a "HYPRE_Solver" instance (line 2) with a "prec" instance. In this way, the HYPRE's functions can be used directly to set the number of functions (line 3), the aggressive coarsening (line 4) and the imposed strong threshold (line 5). Line 6 bypass HYPRE errors. This was set after analysis of "SetElasticityOptions", which also skips them. Otherwise it don't work.

10.2 FEniCSx

For FEniCSx, only PETSc (for linear/non-linear resolution) and SLEPc (for eigenresolution) are interfaced.

10.2.1 linear

Two approaches to linear problems using linear systems are available with the Python API. The first one uses "petsc4py" to set the solver:

```
1 opts = PETSc.Options()
2 opts["ksp_type"] = "preonly"
3 opts["pc_type"] = "lu"
4 solver = PETSc.KSP().create(domain.comm)
5 solver.setFromOptions()
6 solver.setOperators(A)
7 solver.solve(b, x.x.petsc_vec)
8 x.x.scatter_forward()
```

Listing 72

Line 4 creates a solver (based on the same communicator as the mesh "domain"). In this example, the PETSc option mechanism is used to tune this solver to be a direct solver (lines 1,2,3 and 5). Then the operator (i.e. A matrix in this example e.g. listing 59) is attached to the solver in lines 6. The "solver" object is then ready to resolve the system. Given the right-hand side "b" vector (e.g. listing 59), the solution field "x" (e.g. listing 14) of the system is computed by the "solve" method of the "solver" object in line 7. Line 8 synchronises the solution across processes.

The second approach encapsulates everything (assembly and resolution) in a "dolfinx.fem.petsc.LinearProblem" object, as shown below:

```
1 problem = fem.petsc.LinearProblem(a, L, u=x, bcs=bdirichlet,
      petsc_options={"ksp_type": "preonly", "pc_type": "lu"})
2 problem.solve()
```

Listing 73

In line 1 the creation of the object "problem" requires a bilinear form ("a"), a linear form ("L"), a field to store the solution ("x") and a Dirichlet boundary condition (bdirichlet). The optional "petsc_options" gives a way to configure the embedded PETSc linear solver (here to make it a direct solver). Line 2 does the equivalent of listing 59 and 75, providing the solution of the assembled system in the field "x".

With the C++ API, only the first approach (using the C++ PETSc API directly) exists, as the "dolfinx.fem.petsc.LinearProblem" is only implemented in Python:

```
1 la::petsc::options::set("ksp_type", "preonly");
2 la::petsc::options::set("pc_type", "lu");
3 la::petsc::KrylovSolver solver(MPI_COMM_WORLD);
4 solver.set_from_options();
5 solver.set_operator(A.mat());
```

```
6 la::petsc::Vector x_petsc(la::petsc::create_vector_wrap(*x->x())
      , false);
7 la::petsc::Vector b_petsc(la::petsc::create_vector_wrap(b),
      false);
8 solver.solve(x_petsc.vec(), b_petsc.vec());
9 x->x()->scatter_fwd();
```

Listing 74

It corresponds exactly to the 59 listing, except that "x" and "b" vectors must be encapsulated in the PETSc vector (lines 6 and 7) when passed to the "solve" method of the PETSc KrylovSolver instance.

10.2.2 Non-linear

For the non-linear problem in the example in section 2.1.1, the Newton solver object is fed either by the "dolfinx.fem.petsc.NonlinearProblem" object (Python API) or directly by the "setF"/"setJ" methods of the "NewtonSolver" class itself (C++ API). In both cases the "dolfinx::nls::petsc::NewtonSolver" (or its derived Python equivalent "dolfinx.nls.petsc.NewtonSolver") embeds a PETSc linear solver instance to solve the linear system resulting from linearising the non-linear problem at each non-linear loop. The parameterisation of this linear solver is done after its construction (by the Newton solver constructor). The following are the Python API instructions to solve the non-linear problem of the example in section 2.1.1:

```
1  problem = fem.petsc.NonlinearProblem(F, u, bdirichlet,J)
2  newton_solver = nls.petsc.NewtonSolver(MPI.COMM_WORLD, problem)
3  ksp = newton_solver.krylov_solver
4  opts = PETSc.Options()
5  option_prefix = ksp.getOptionsPrefix()
6  opts[f"{option_prefix}ksp_type"] = "cg"
7  opts[f"{option_prefix}ksp_rtol"] = "1e-12"
8  opts[f"{option_prefix}pc_type"] = "hypre"
9  opts[f"{option_prefix}pc_hypre_type"] = "boomeramg"
10 opts[f"{option_prefix}pc_hypre_type"] = "boomeramg"
11 opts[f"{option_prefix}pc_hypre_boomeramg_coarsen_type"] = "HMIS"
12 opts[f"{option_prefix}pc_hypre_boomeramg_relax_type_up"] = "
       l1scaled-SOR/Jacobi"
13 opts[f"{option_prefix}pc_hypre_boomeramg_relax_type_down"] = "
       l1scaled-SOR/Jacobi"
14 opts[f"{option_prefix}pc_hypre_boomeramg_relax_type_coarse"] = "
       Gaussian-elimination"
```

```
opts[f"{option_prefix}pc_hypre_boomeramg_interp_type"] = "ext+i"
opts[f"{option_prefix}pc_hypre_boomeramg_numfunctions"] = "2"
opts[f"{option_prefix}pc_hypre_boomeramg_agg_nl"] = "0"
opts[f"{option_prefix}pc_hypre_boomeramg_strong_threshold"] =
    "0.25"
opts[f"{option_prefix}pc_hypre_boomeramg_print_statistics"] =
    "1"
ksp.setFromOptions()
newton_solver.rtol = 1.e-7
newton_solver.atol = 5.e-8
r=newton_solver.solve(u)
```

Listing 75

where:

- Line 1 creates the non-linear problem object from "F" (e.g. listing 25), "J" (e.g. listing 26), " bdirichlet" (e.g. listing 63) and "u" (e.g. listing 14).
- The Newton solver instance is created from the non-linear problem in line 2.
- The linear solver is extracted from the Newton solver instance as the variable "ksp" on line 3.
- Lines 4 to 19 set options which are passed to "ksp" in line 20. This setting is the same as the one specified in the 71 listing for MFEM.
- Lines 21 and 22 set the Newton solver itself.
- The non-linear resolution can then be calculated using the solve function in line 23.

With the C++ API instruction it is the same principle:

```
dolfinx::nls::petsc::NewtonSolver newton_solver(MPI_COMM_WORLD);
KSP ksp = newton_solver.get_krylov_solver().ksp();
KSPSetTolerances(ksp, 1.e-12, PETSC_DEFAULT, PETSC_DEFAULT,
    2000);
PetscOptionsSetValue(NULL, "-nls_solve_ksp_type", "cg");
PetscOptionsSetValue(NULL, "-nls_solve_pc_type", "hypre");
PetscOptionsSetValue(NULL, "-nls_solve_pc_hypre_type", "
    boomeramg");
```

```
7  PetscOptionsSetValue(NULL, "-
       nls_solve_pc_hypre_boomeramg_print_statistics", "1");
8  PetscOptionsSetValue(NULL, "-
       nls_solve_pc_hypre_boomeramg_coarsen_type", "HMIS");
9  PetscOptionsSetValue(NULL, "-
       nls_solve_pc_hypre_boomeramg_relax_type_up", "l1scaled-SOR/
       Jacobi");
10 PetscOptionsSetValue(NULL, "-
       nls_solve_pc_hypre_boomeramg_relax_type_down", "l1scaled-SOR/
       Jacobi");
11 PetscOptionsSetValue(NULL, "-
       nls_solve_pc_hypre_boomeramg_relax_type_coarse", "Gaussian-
       elimination");
12 PetscOptionsSetValue(NULL, "-
       nls_solve_pc_hypre_boomeramg_interp_type", "ext+i");
13 PetscOptionsSetValue(NULL, "-
       nls_solve_pc_hypre_boomeramg_numfunctions", "2");
14 PetscOptionsSetValue(NULL, "-nls_solve_pc_hypre_boomeramg_agg_nl
       ", "0");
15 PetscOptionsSetValue(NULL, "-
       nls_solve_pc_hypre_boomeramg_strong_threshold", "0.25");
16 KSPSetFromOptions(ksp);
17 newton_solver.max_it=10;
18 newton_solver.rtol = 1.e-7;
19 newton_solver.atol = 5.e-8;
20 newton_solver.setF(.....
21 newton_solver.setJ(.....
22 ......
23 std::pair<int, bool> r = newton_solver.solve(x_petsc.vec());
```

Listing 76

except that the non-linear problem (formulations, boundary conditions) is passed to the Newton solver via "setF", "setJ", ... (lines 20 to 22).

10.3 Performances

The elapsed time in the solve/Mult method of the non-linear resolution, corresponding to the example in section 2.1.1, is given in figure 24c. As MFEM needs 5 iterations to converge and FEniCSx 7, the elapsed times in this figure are given per non-linear iteration (i.e. elapsed time divided by number of non-linear iterations). In this way, the comparison between MFEM and FEniCSx focuses on the non-linear iteration performance and is

not affected by the convergence criterion (which is slightly different in the two libraries, as discussed below).

In all cases (FEniCSx strategies and MFEM variants) the curves show a fairly good scalability: for MFEM only the first point is strange and for FEniCSx a slight deterioration is observed for a high number of processes. The elapsed time of the non-linear iteration is mainly dominated by the linear system resolution. Thanks to the common setup of the PETSc instance (PCG with Boomeramg preconditioner with specific options), this part is expected to be the same[4] between the two libraries. The remaining tasks (in this test case: residual vector norm, solution update and parallel system assembly) are certainly the ones responsible for the rather small variations shown in figure 24c. The figure 24d shows the ratio of the different strategies and variants against the "MFEM" reference. FEniCSx is 1.05 to 1.35 times faster than MFEM for up to 32 processes, and becomes slower (1.15 to 1.2 times) for 64 and 128 processes. As the non-linear iteration includes the measure of the figures 22b and 22d, it can be assumed that the matrix and vector generation are not involved in this small FEniCSx scalability degradation, as they scale perfectly. So the assembly could be the source of this degradation (but this is not clear in the figure 23) or the parallel vector computation (residual norm) and perhaps some linear solver fluctuation. In any case, all FEniCSx strategies are consistent, as are all MFEM variants.

Regarding the non-linear convergence criterion, MFEM uses an absolute and relative test on the norm r of the residual vector (the first test satisfied (i.e. less than the thresholds) stops the loop). For the relative test, the current r is divided by the norm of the residual vector computed at the beginning of the nonlinear loops: r_0. With FEniCSx the same tests are used, but surprisingly r_0 is the vector norm of the first variation of the solution: $r_0 = |\,(\delta\mathbf{u})_0\,|$. This caused r to be divided by something smaller than in MFEM and forced the non-linear resolution to iterate 2 more times (as the absolute threshold is not reached in either library and only the relative test is involved in convergence).

[4]Nevertheless, the PETSc solver with the Hypre Boomeramg preconditioner seems to exhibit some elapsed time fluctuation has observer in test not presented here.

11 Output (point 8)

11.1 Save to file

Both MFEM and FEniCSx can use different output format to save various information (fields, constant, ...). Those outputs can be operated in sequential or in parallel. In particular, both libraries can use ADIOS2 library for parallel outputs.

With FEniCSx each space imply a different output stream but fields on these space can be grouped. With test case of section 2.1.1 outputting displacement ("u"), Neuman loading ("f"), Young's modulus constant ("E") and mesh partition id ("part") can be done in python with the following instructions:

```
filename = Path(f"XXX_{MPI.COMM_WORLD.Get_size():d}.bp")
with VTXWriter(MPI.COMM_WORLD, filename, [u,f]) as ofile:
  ofile.write(0.)
filename = Path(f"YYY_{MPI.COMM_WORLD.Get_size():d}.bp")
with VTXWriter(MPI.COMM_WORLD, filename, [E,part]) as ofile:
  ofile.write(0.)
```

Listing 77

where XXX and YYY are dummy file names (with path location). The same outputs in C++ is given by the following instructions:

```
{
  std::string n = "XXX_" + std::to_string(nb_proc) + ".bp";
  io::adios2_writer::U<scalar_dolf> fields = {u, f};
  io::VTXWriter<scalar_dolf> vtx(MPLCOMM_WORLD, n, fields, "bp4
      ");
  vtx.write(0);
}
{
  std::string n = "YYY_" + std::to_string(nb_proc) + ".bp";
  io::adios2_writer::U<scalar_dolf> fields = {E, part};
  io::VTXWriter<scalar_dolf> vtx(MPLCOMM_WORLD, n, fields, "bp4
      ");
  vtx.write(0);
}
```

Listing 78

In this case ADIOS2 engine type is set to "bp4". In python example engine is not set explicitely and thus is "BPFile" which in ADIOS2 library may

correspond to "bp4" or "bp5" depending on library version.

With MFEM, all information can be outputed in a single files. The output example above is then given by the following set of C++ instruction :

```
std::string n = "ZZZ_" + std::to_string(nb_proc) + ".bp";
adios2stream os(n, adios2stream::openmode::out, MPI_COMM_WORLD,
    "BP4");
pmesh->Print(os);
x.Save(os, "disp_mfem");
part.Save(os, "part");
Ef.Save(os, "E");
load.Save(os, "load");
```

Listing 79

where "x", "part", "Ef" and "load" corresponds respectively to FEniCSx fields "u", "part", "E" and "f".

11.2 Use streaming channel/interfaced library

In this work it has not been tested (yet?), but results in both libraries can be redirected directly to visualisation tools without the use of an intermediate set of files. In MFEM a "streaming channel" can be opened to connect to the GLVis product. In FEniCSx, by using the Python API, you can use any tool adapted to the visualisation of fields on a mesh, and pyvista is, from the demos, the library to work with.

11.3 Preformances

Regarding the output, the elapsed time for the test case of the section 2.1.1 shown in figures 24e and 24f lead to the following observations:

- In sequential MFEM is from 5 to 10 times slower then FEniCSx. After investigation, it appears that in fact MFEM is outputting information at nodes per element and FEniCSx only at nodes. This induces that MFEM output ~ 6[5] times more information per field compare to FEniCS.

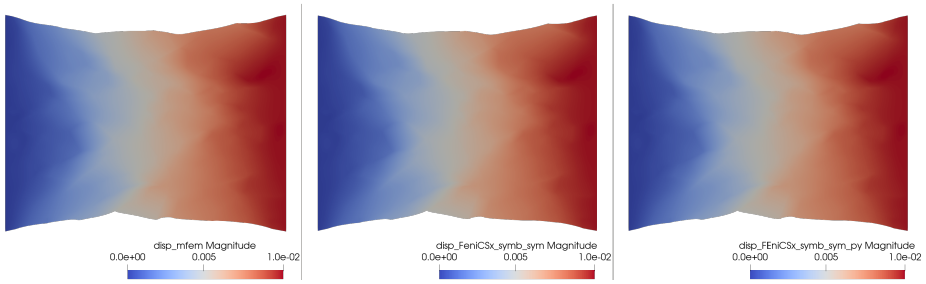
[5] the ratio is obtained on one field where the total number of outputted information (scalar or vector) are compared: $\frac{36916368 \text{ MFEM}}{6168934 \text{ FEniCSx}}$

- MFEM scale well and FEniCSx not. Maybe the fact that FEniCSx outputs are split in 4 different files/stream has an impact but it is quite surprising.
- FEniCSx output with the python version is slightly different (slightly better for more than 16 processes) compared to the C++ version. Except the ADIOS2 engine format which is explicit in the C++ version there is no real reason for this difference (same computer,compiler,...). Perhaps some difference in the hardware disk IO load between the two benchmark campaigns (C++ and python) explains this fact.

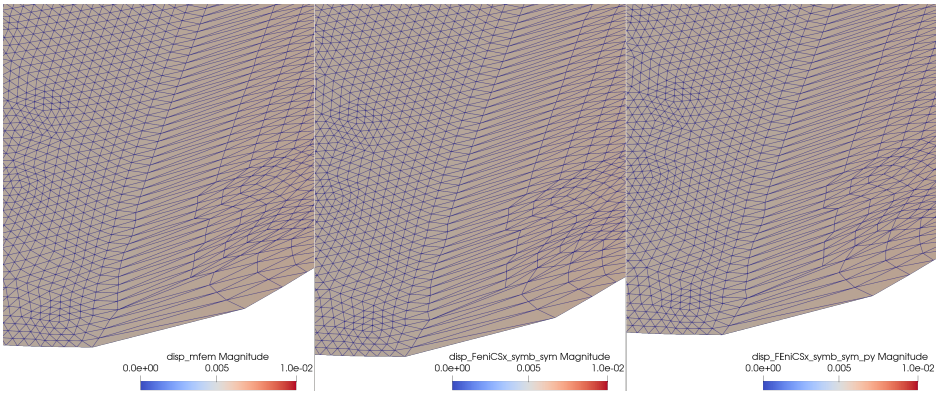
12 Results of test case of section 2.1.1

The displacement field obtained in traction with a double refined mesh is shown in figure 9. In figure 9a the general view of the displacement for MFEM, FEniCSx_C++ (FEniCSx using c++ api) and FEniCSx_py (FEniCSx using python api) looks similar. The Derichlet boundary conditions ((0.,0.) on the left-hand side, (0.1,0.) on the right-hand side) are correctly taken into account in all the simulations. The non-uniformity of Young's modulus is also clearly evident in all simulations, as the top and bottom edges of the part do not have a smooth displacement. Figures 9b and 9c zoom in on two damaged zones. The asymmetric law clearly influence the displacement as damaged zone exhibit large mesh deformation related to displacement jump associated with rigidity loss in traction. In compression (horizontal damaged zone in figure 9c), rigidity is maintained and there is no displacement jump. On the same figure, due to Young's modulus difference in grains and damage orientation, some opening can be observed, has some traction occurs. In all cases all simulations give the same jumps at the same locations.

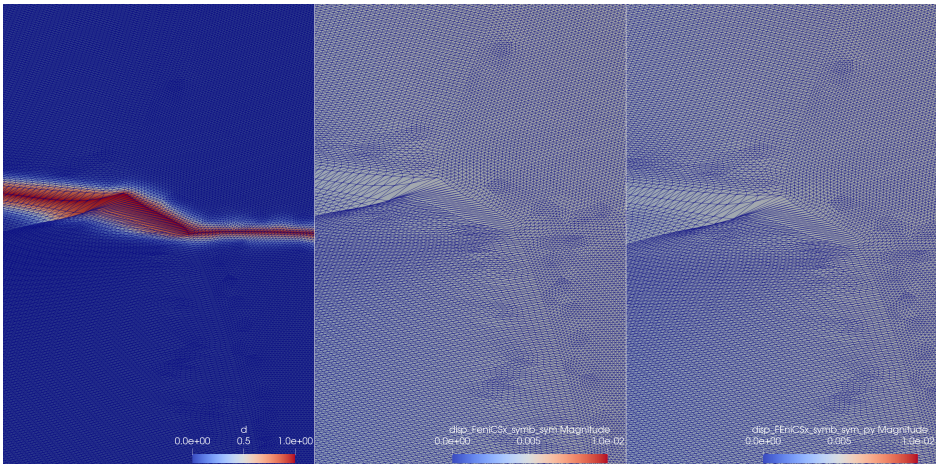
The strain tensor component ϵ_{xx} is shown in figure 10 and 11 with FEniCSx and MFEM simulations. Simulations are consistent. Tensile deformation is clearly related to stiffness weakness, either due to damage or low modulus of elasticity, as shown in figure 11 by comparing these fields.



(a) general view

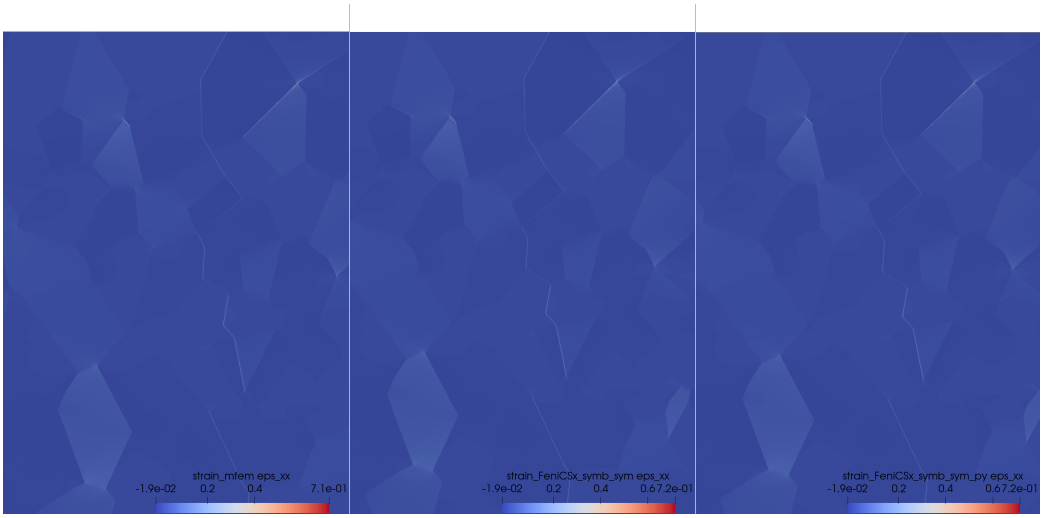


(b) zoom lower part

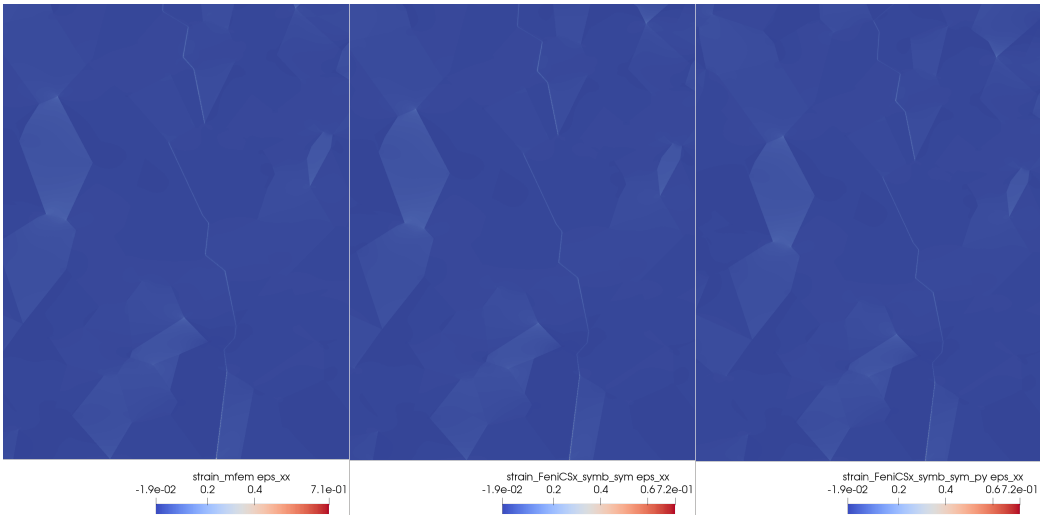


(c) zoom upper part (with damage field coloring on MFEM deformed mesh)

Figure 9: Displacement field given by MFEM (left) FEniCSx_C++(center) and FEniCSx_py(right) for the test case of section 2.1.1 in traction.

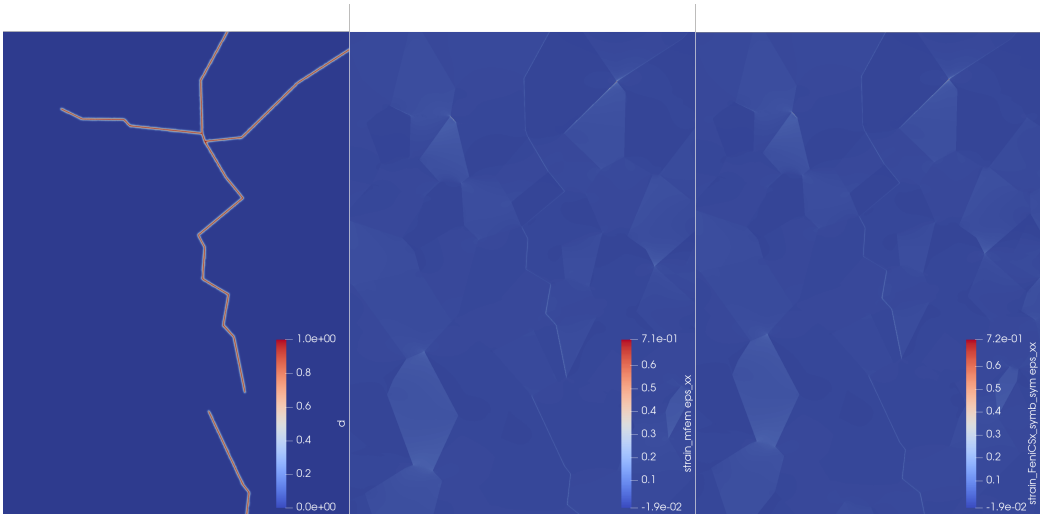


(a) upper view

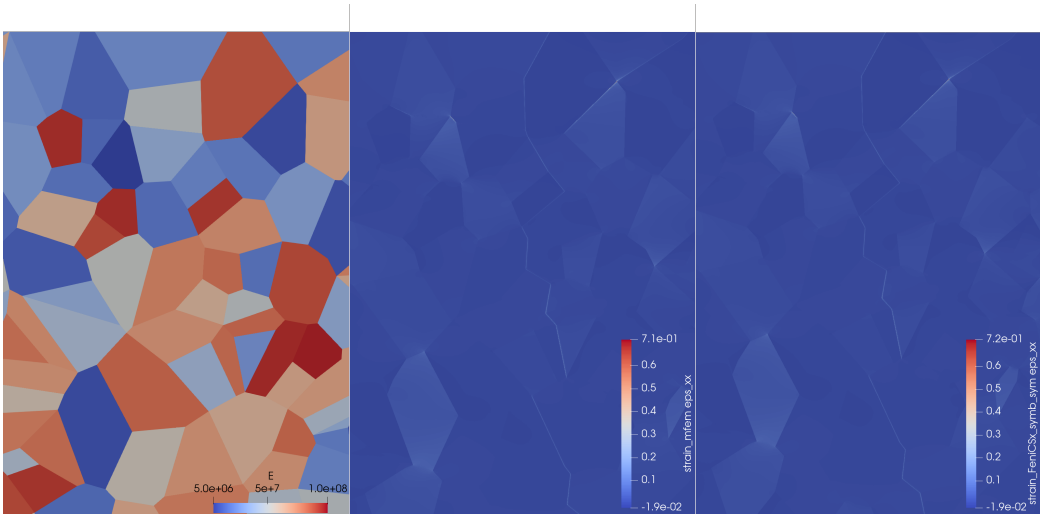


(b) lower view

Figure 10: Strain field component ϵ_{xx} given by MFEM (left) FEniCSx_C++(center) and FEniCSx_py(right) for the test case of section 2.1.1 in traction.



(a) with damage



(b) with Young's modulus

Figure 11: Strain field component ϵ_{xx} given by MFEM (center) and FEniCSx_C++(right) for the test case of section 2.1.1 in traction. Damage and Young Modulus, coefficient (left) give localization of rigidity weakness that involve deformation.

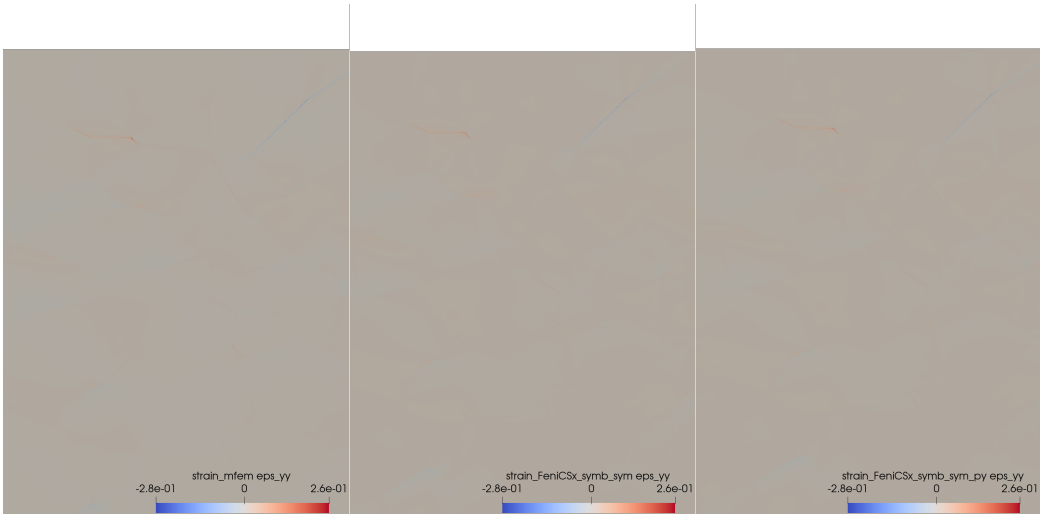


Figure 12: Strain field component ϵ_{yy} given by MFEM (left) FEniCSx_C++(center) and FEniCSx_py(right) for the test case of section 2.1.1 in traction.

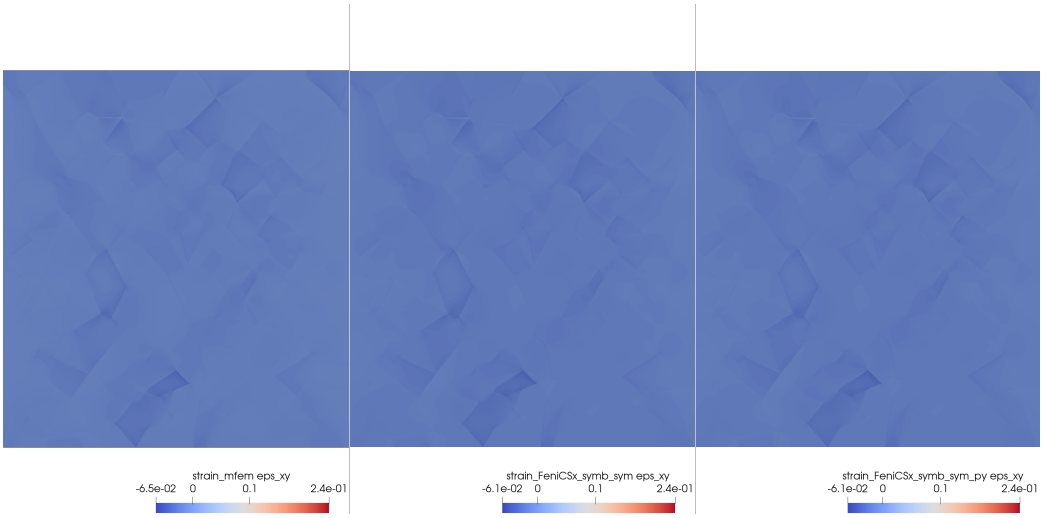


Figure 13: Strain field component ϵ_{xy} given by MFEM (left) FEniCSx_C++(center) and FEniCSx_py(right) for the test case of section 2.1.1 in traction.

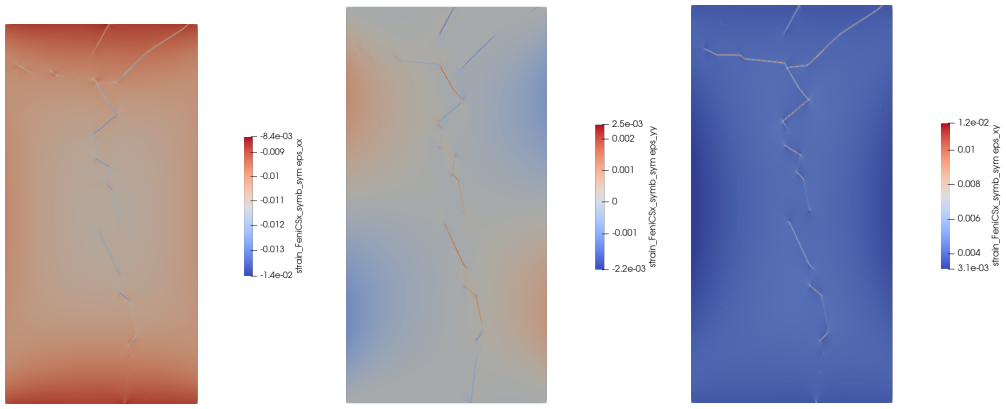
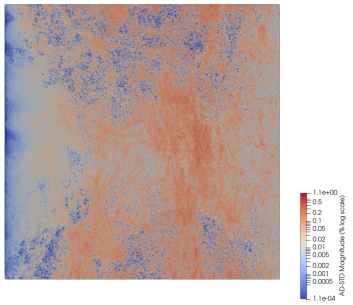


Figure 14: Strain field component ϵ_{xx},ϵ_{yy} and ϵ_{xy} given by FEniCSx_C++ for the test case of section 2.1.1 in compression without volume force.

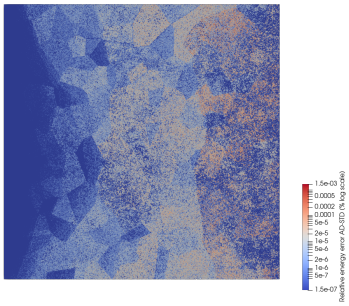
Quantitative comparison of both code for test case of section 2.1.1 in traction is done via extra code that:

- in MFEM, at the end of the simulation, export the mesh and displacement field to a binary file (via handmade code).
- in FEniCSx or MFEM AD (MFEM with Automatic Differentiation of section 8.2.2)
  - import (from binary file via handmade code) MFEM grid coordinates and field ($\mathbf{u}_{MFEM}$) values at the end of simulation.
  - with both field (FEniCSx or MFEM AD and MFEM):
    * Compute at node the displacement vector difference $\delta\mathbf{u}_{sol}$ by using coordinates to associate results. $\delta\mathbf{u}_{sol} = \mathbf{u}_{sol} - \mathbf{u}_{MFEM}$ where $\mathbf{u}_{sol}$ is the displacements of FEniCSx or MFEM AD resolution.
    * At the same time, calculate the L2 norm per component of $\delta\mathbf{u}_{sol}$. The error in displacement for the x component using the L2 norm is: $L2_x(\mathbf{u}_{sol}) = \sqrt{\delta\mathbf{u}_{sol}^t.\mathbf{OP}_x.\delta\mathbf{u}_{sol}}$ with $\mathbf{OP}_x$ a diagonal operator with 1 only for x dofs. The error in displacement for the y component using the L2 norm is: $L2_y(\mathbf{u}_{sol}) = \sqrt{\delta\mathbf{u}_{sol}^t.\mathbf{OP}_y.\delta\mathbf{u}_{sol}}$ with $\mathbf{OP}_y$ a diagonal operator with 1 only for y dofs.
    * Based on $\delta\mathbf{u}_{sol}$, calculate the elementary energy density error at the centre of the elements. For an element k this energy density error is $EE_k(\mathbf{u}_{sol}) = \epsilon\left(\delta\mathbf{u}_{sol}(X_k)\right) : \sigma\left(\delta\mathbf{u}_{sol}(X_k)\right)$ where σ and ϵ come from the implementation of (2) and (5) respectively, and X_k is the center position of element k.
    * At the same time, calculate the sum of these elementary energy density errors: $EE(\mathbf{u}_{sol}) = \sum_{k\in\mathfrak{E}} EE_k(\mathbf{u}_{sol})$ with $\mathfrak{E}$ the set of elements.
    * Calculate relative (percentage) error field in displacement: $RE_{\mathbf{u}}(\mathbf{u}_{sol}) = 100.\left(\frac{1}{L2_x(\mathbf{u}_{sol})}.\mathbf{OP}_x + \frac{1}{L2_y(\mathbf{u}_{sol})}.\mathbf{OP}_y\right).\delta\mathbf{u}_{sol}$ and energy density: $RE_{EE_k}(\mathbf{u}_{sol}) = 100.\frac{\epsilon(\delta\mathbf{u}_{sol}(X_k)):\sigma(\delta\mathbf{u}_{sol}(X_k))}{EE(\mathbf{u}_{sol})}$

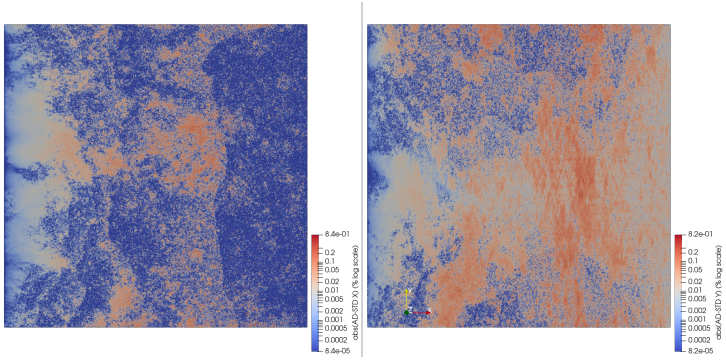
For a mesh refined 2 times, MFEM AD vs. MFEM gives the following errors:



(a) Distribution of the displacement magnitude error in %



(b) Distribution of the energy density error in %

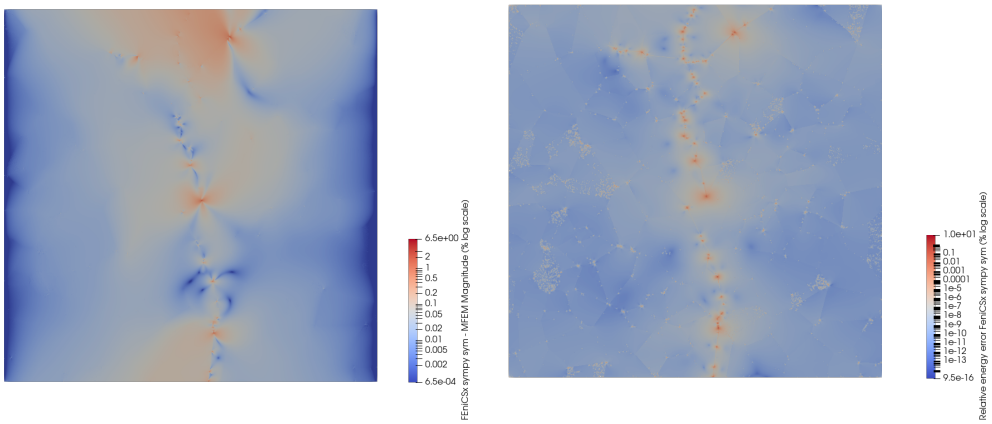


(c) Distribution of the displacement error in % per component

Figure 15: Error in displacement and energy density (distribution in %) of the MFEM AD approach (see section 8.2.2) compare to MFEM. The simulation correspond to the test case of section 2.1.1 in traction and refined 2 times.

- $L2_x(\mathbf{u}_{\text{MFEM AD}}) = 1.34818e^{-15}\text{m}$
- $L2_y(\mathbf{u}_{\text{MFEM AD}}) = 6.34363e^{-16}\text{m}$
- $EE(\mathbf{u}_{\text{MFEM AD}}) = 2.10673e^{-15}J/m^3$

These errors are small compared to the resolution thresholds (see section 10). The distribution of these errors is shown in the figure 15. The percentages are rather small, indicating that the distribution of the error is relatively uniform. All these elements lead to the conclusion that AD and standard solutions are the same. For a mesh refined 2 times, the FEniCSx approach of the 8.1.3 section compared to MFEM gives the following errors:



(a) Distribution of the displacement magnitude error in %

(b) Distribution of the energy density error in %

Figure 16: Error in displacement and energy density (distribution in %) of the FEniCSx Sympy symmetric approach (see section 8.1) compare to MFEM. The simulation correspond to the test case of section 2.1.1 in traction and refined 2 times.

- $L2_x(\mathbf{u}_{\text{FEniCSx sympy symb}}) = 4.103503e^{-04}\text{m}$
- $L2_y(\mathbf{u}_{\text{FEniCSx sympy symb}}) = 1.675378e^{-04}\text{m}$
- $EE(\mathbf{u}_{\text{FEniCSx sympy symb}}) = 4.239257e^{6}J/m^3$

The distribution of these important errors is shown in the figure 16. It shows some localised peaks that represent more than 6% of their respective error. Their positions and intensities are related to the difference in mesh refinement between MFEM and FEniCSx, amplified by the sensitivity of the law to damage. This can be seen in figure 17, where the difference in mesh orientation, exactly where the damage is not zero, leads to these displacement fluctuations. In the same figure, the MFEM mesh quality (aspect ratio) shows that the problem is not related to this aspect. This is mainly due to the sensitivity of the asymmetric model to damage, which is not the same for rotated elements. The full comparison of displacements between FEniCSx and MFEM is therefore only meaningful for the unrefined mesh, which is the same for both codes. The figure 18 shows, for the unrefined mesh, the distribution between the four FEniCSx approaches (see section 8.1) and MFEM.

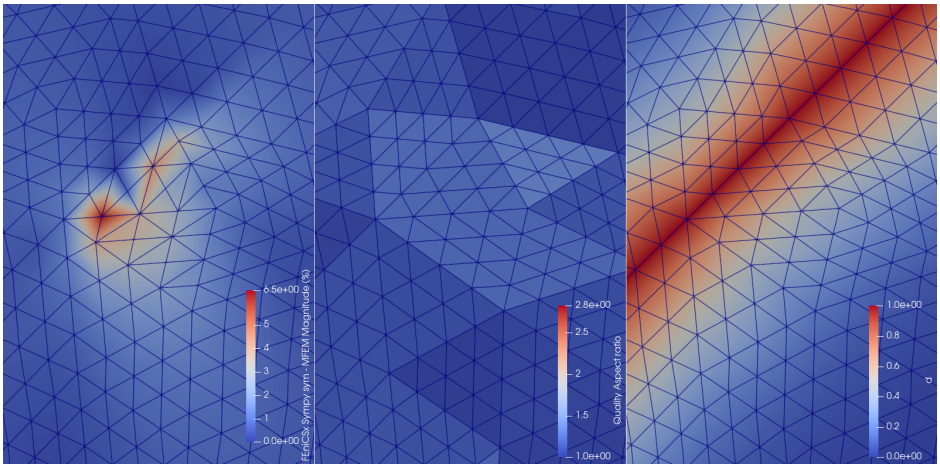


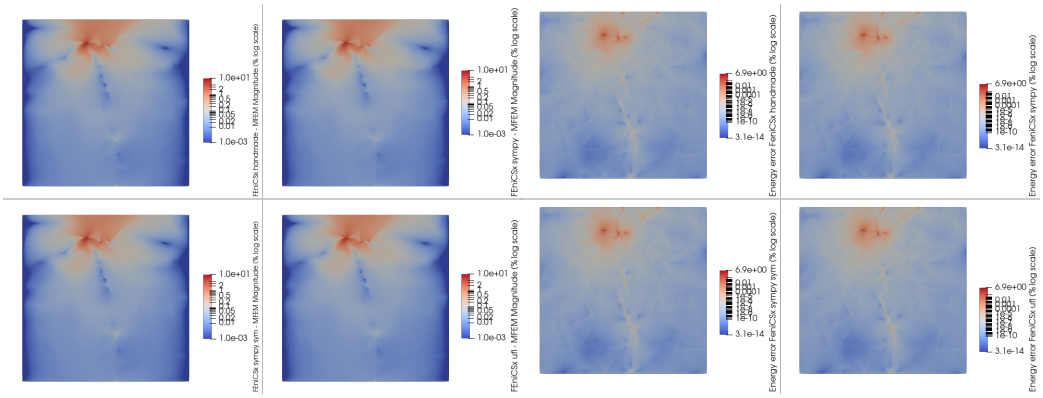
Figure 17: Refinement impact for the test case of section 2.1.1 in traction. On left error in displacement (% error on FEniCSx refined mesh) of FEniCSx Sympy symmetric approach (see section 8.1) compare to MFEM. On center mesh quality (aspect ratio) of MFEM refined mesh. On right damage field on MFEM mesh in this location.

All four approaches lead to the same errors:

- $L2_x(\mathbf{u}_{\text{FEniCSx}}) = 7.514989e^{-08}\text{m}$
- $L2_y(\mathbf{u}_{\text{FEniCSx}}) = 1.738645e^{-07}\text{m}$
- $EE(\mathbf{u}_{\text{FEniCSx}}) = 1.755394e^{-02}J/m^3$

and the same distribution (easily visible in the figure 18).

The displacement errors are of the order of the resolution thresholds (see section 10). It can therefore be assumed that both libraries give the same solution at the specified resolution accuracy. The distribution of the error, either in displacement or energy density, clearly shows that it is concentrated in a few damaged areas. This may be due to small differences in damage values calculated using the same algorithm (section 7.1) but not the same mesh database.



(a) Distribution of the displacement magnitude error in %

(b) Distribution of the energy density error in %

Figure 18: Distribution of error in displacement (a) and energy density (b) of the 4 FEniCSx approaches (see section 8.1) compare to MFEM. The simulation correspond to the test case of section 2.1.1 without refinement in traction with volume loading.

13 Conclusions

Choosing between FEniCSx and MFEM largely depends on your specific needs and expertise.

FEniCSx is excellent for users looking for ease of use. The user-friendly python interface uses UFL (Unified Form Language) to define variational problems, which automatically generates efficient low-level code. The extensive use of python for high-level problem definition is well suited for rapid prototyping (e.g. engineering applications requiring rapid implementation and testing of PDE models) and educational purposes. Yet it offers high performance (scalability), as this study shows. Thanks to a strong community support, it is relatively easy to learn the python API. However, as this library is still in development (the API is still undergoing major changes), it is important to take this into account in any project based on it. Using the C++ API is not recommended at the beginning of a project, which should logically start with prototyping in Python (unless you are more comfortable with C++ vs. Python). However, switching to C++ may be important when moving on to larger problems where computational speed is no longer

entirely dependent on the library (e.g., damage smoothing in this study). Finally, at the time of writing, FEniCSx may not be as efficient for extremely large simulations on heterogeneous computers (CPU/GPU) and has limited advanced customization compared to MFEM.

MFEM is ideal for users who require high performance, scalability and extensive customization (for advanced users) for complex simulations. It integrates to a larger set of external library compared to FEniCSx, which provides many alternative solution to obtain high performance on distributed CPU and/or GPU. The use of the MFEM library involves a steeper learning curve, especially for users who are not familiar with the C++ language, and its implementation and use require more effort for the simplest problems. This last statement, which may change in the future, is made without investigating the python API proposed by the library but not used so much in the provided nice set of demos/examples. Finding the right implementation requires a deep understanding of the library and very good FEM skills. In this respect, FEniCSx is easier for FEM beginners, and in particular ffcx will choose for you the order of integration, what to put in the integration loop and what is constant, ... The use of Automatic Differentiation (AD) for the formulation description makes MFEM relatively competitive to FEniCSx UFL. AD offers the same simplicity from an implementation point of view (but not from a language point of view) to describe e.g. a potential that can be differentiated and used in a problem formulation. In this study, no major drop in performance was observed with AD compared to the standard handmade implementation. It's a shame, though, that AD is only available via a miniapps and is not a basic feature. Compared to FEniCSx, MFEM offers extended support for complex geometries and higher-order elements with out-of-the-box adaptive mesh refinement (AMR) and NURBS mesh.

Both libraries are powerful tools in the realm of FEM and PDE solving, and your choice should be guided by your project's requirements and your familiarity with programming languages.

With regard to this study, many aspects still need to be investigated / compared by introducing new specific test cases.

References

[1] Neper: Polycrystal generation and meshing https://neper.info.

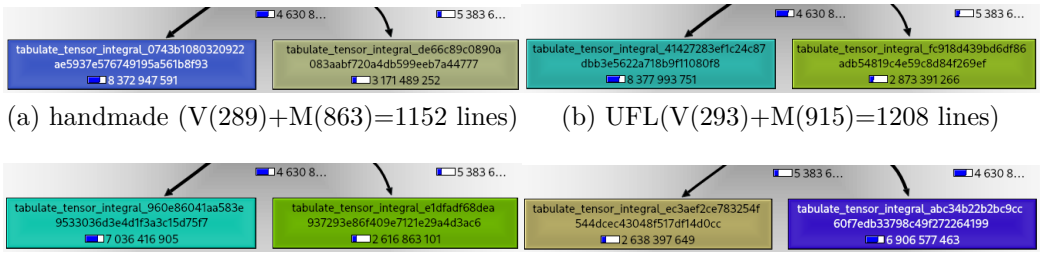
[2] J. Andrej, N. Atallah, J.-P. Bäcker, J. Camier, D. Copeland, V. Dobrev, Y. Dudouit, T. Duswald, B. Keith, D. Kim, T. Kolev, B. Lazarov, K. Mittal, W. Pazner, S. Petrides, S. Shiraiwa, M. Stowell, and V. Tomov. High-performance finite elements with mfem, 2024.

[3] A. Plaza and G. Carey. Local refinement of simplicial grids based on the skeleton. *Applied Numerical Mathematics*, 32(2):195–218, 2000.

[4] R. Quey, P. Dawson, and F. Barbe. Large-scale 3d random polycrystals for the finite element method: Generation, meshing and remeshing. *Computer Methods in Applied Mechanics and Engineering*, 200(17):1729–1745, 2011.

Copyright

A Example section 2.1.1



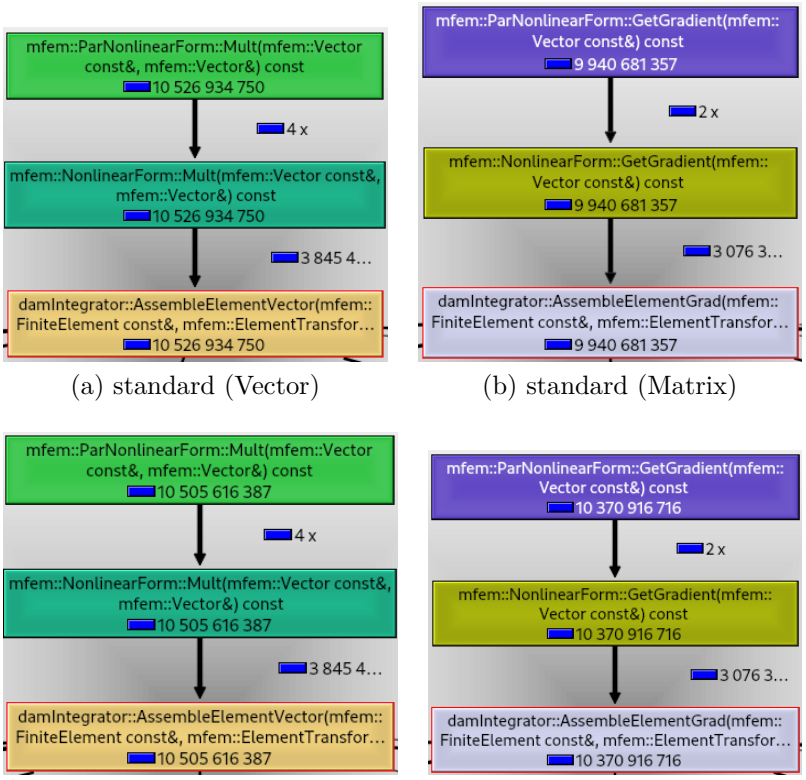
(a) handmade (V(289)+M(863)=1152 lines) (b) UFL(V(293)+M(915)=1208 lines)

(c) sympy (V(262)+M(713)=975 lines) (d) sympy sym.(V(261)+M(969)=708 lines)

Figure 19: Valgrind instructions count for FEniCSx strategies. In each case, the kernel with the higher number of instruction corresponds to matrix generation and the lower to vector generation. The number of C source lines for each kernel (V:vector generation, M:matrix generation) is given in brackets in the captions of the sub-figures. The mesh is not refined.

A.1 Valgrind

With unrefined mesh for both library as Valgrind slow down simulation, the callgrind tool provides instructions count shown in figure 19 and 20. These figures focus on elementary vector and matrix creation.



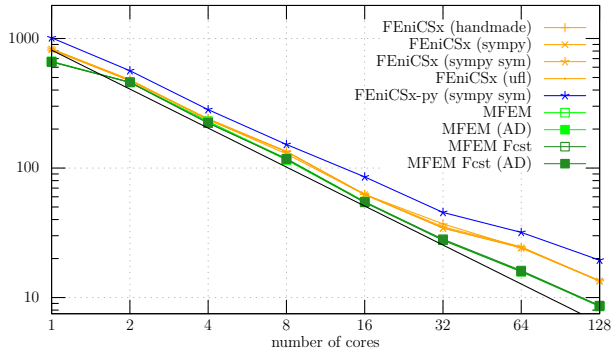
(a) standard (Vector) (b) standard (Matrix)

(c) AD (Vector) (d) AD (Matrix)

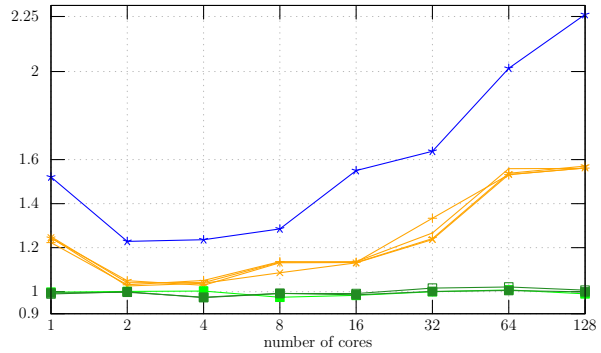
Figure 20: Valgrind instructions count for MFEM standard and AD. Elementary vector and matrix calculations. Mesh is not refined.

A.2 Elapsed time

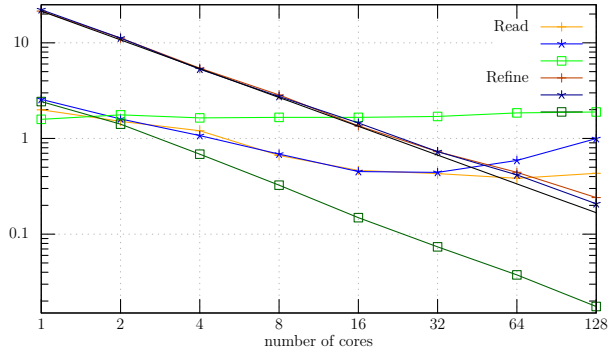
With a double refined mesh, elapsed time collected by specific implementation of the C++ source code for both library are presented in figure 21, 22, 23, 24 and 25. For all of them the left column present



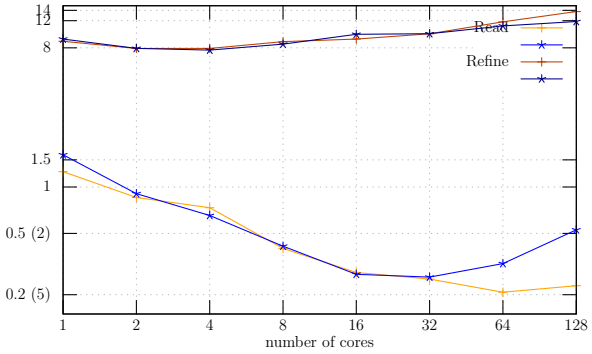
(a) Simulation (in s)



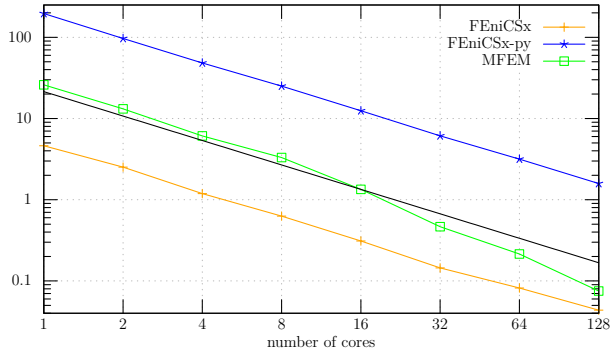
(b) Simulation (Ratio vs MFEM)



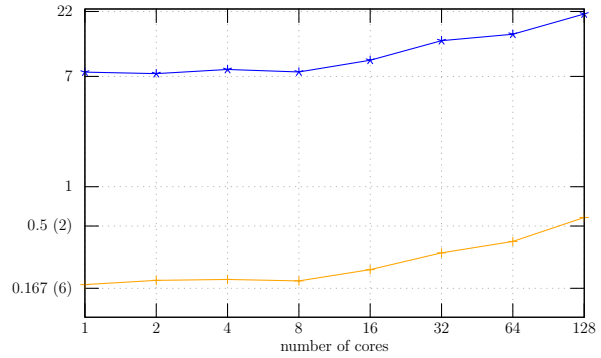
(c) Mesh (in s)



(d) Mesh (Ratio vs MFEM)

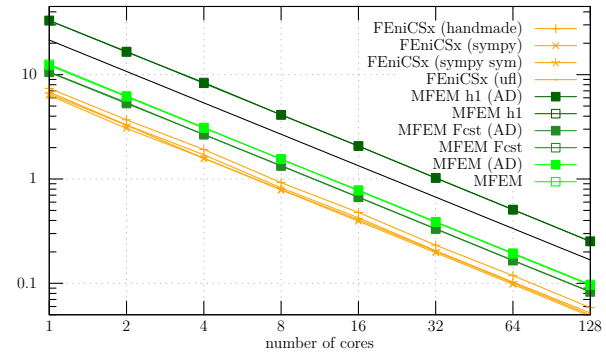


(e) Damage generation (in s)

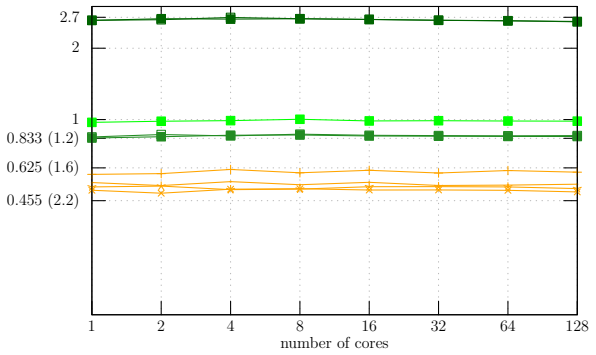


(f) Damage generation (Ratio vs MFEM)

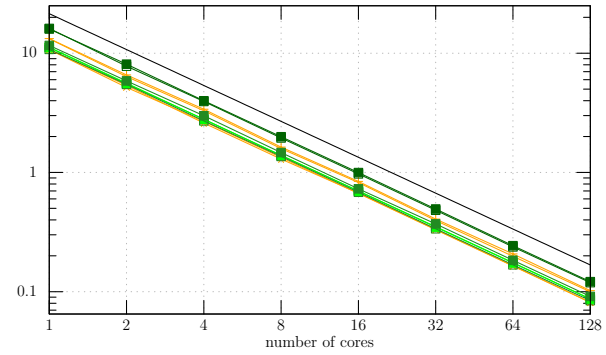
Figure 21: Elapsed time function of the number of processes: Simulation, mesh and damage creation (in log-log scale except for y axes of b)



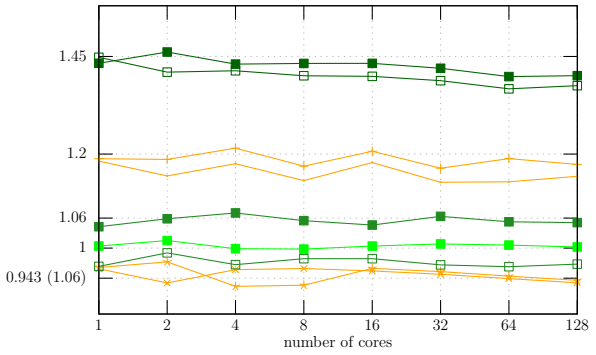
(a) Elementary vector creation (in s)



(b) Elementary vector creation (Ratio vs MFEM)



(c) Elementary matrix creation(in s)



(d) Elementary matrix creation(Ratio vs MFEM)

Figure 22: Elapsed time function of the number of processes: Elementary vector and matrix creation (in log-log scale)

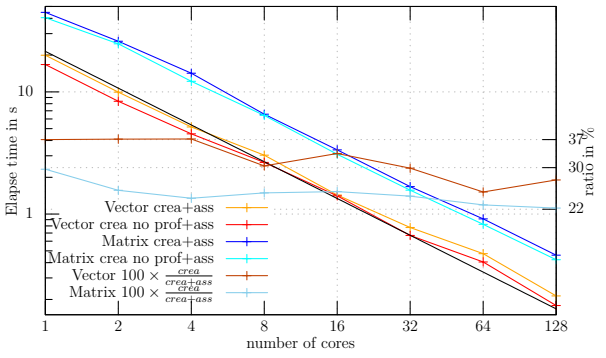
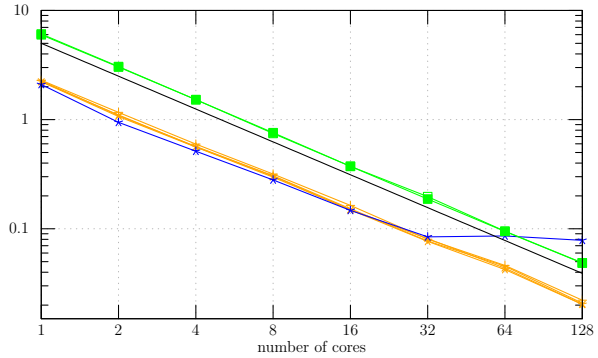
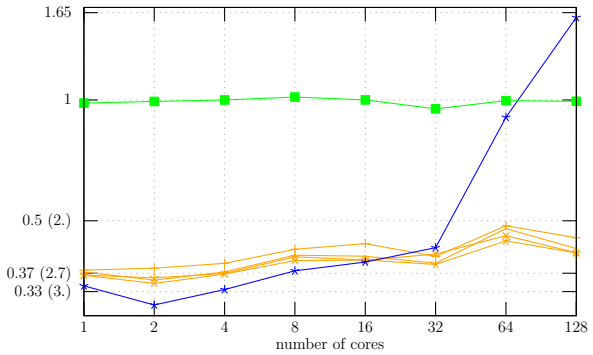


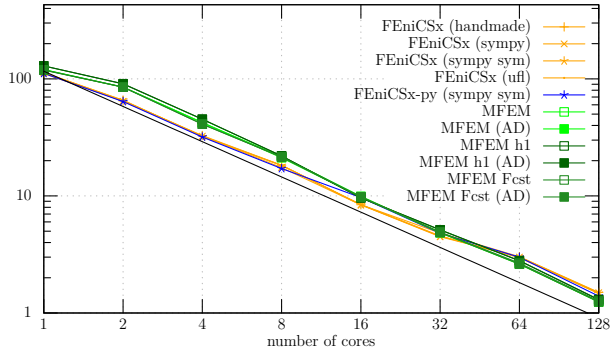
Figure 23: Elementary creation and assembly with FEniCSx (handmade): Elapsed times in s (log-log scale)



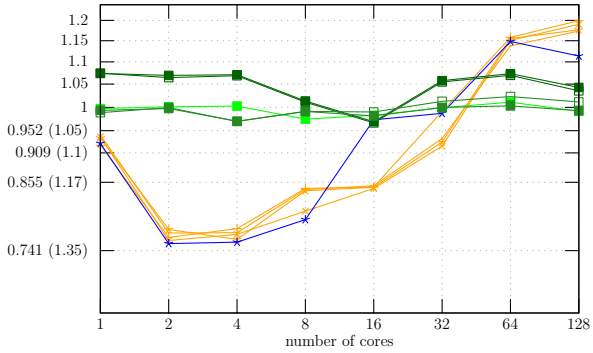
(a) Strain/stress creation (in s)



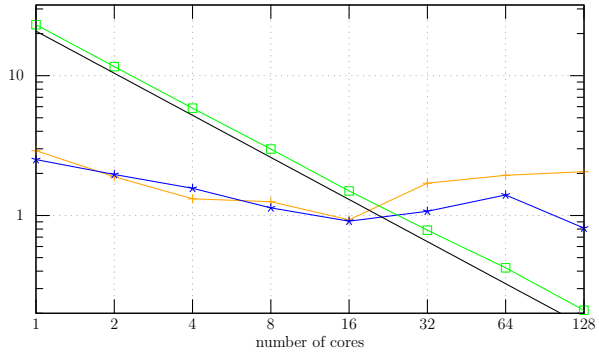
(b) Strain/stress creation (Ratio vs MFEM)



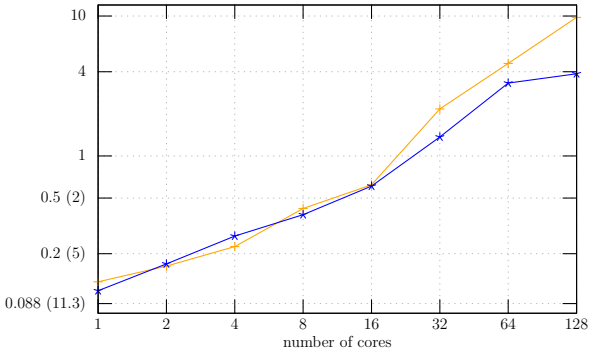
(c) One non-linear loop iteration (in s)



(d) One non-linear loop iteration(Ratio vs MFEM)

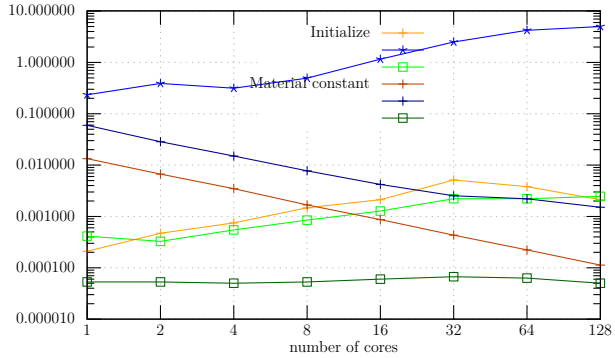


(e) Outputs (in s)

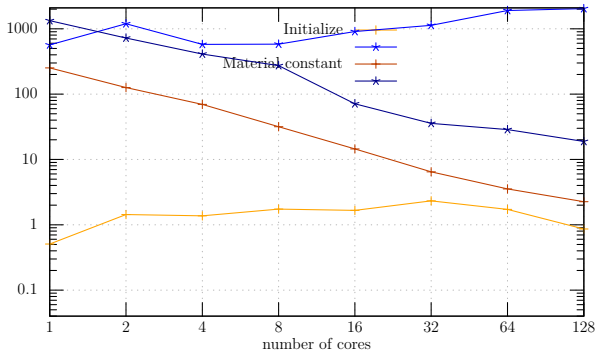


(f) Outputs (Ratio vs MFEM)

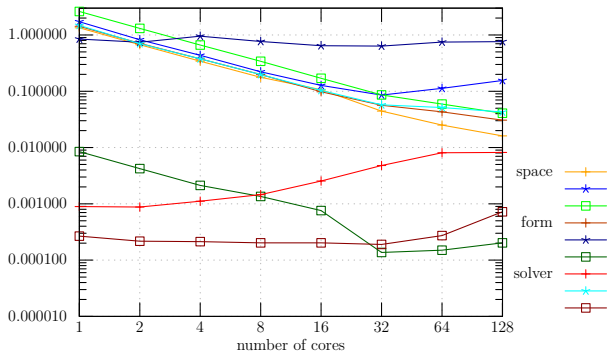
Figure 24: Elapsed time function of the number of processes: Strain/stress creation, non-linear loop and outputs (in log-log scale)



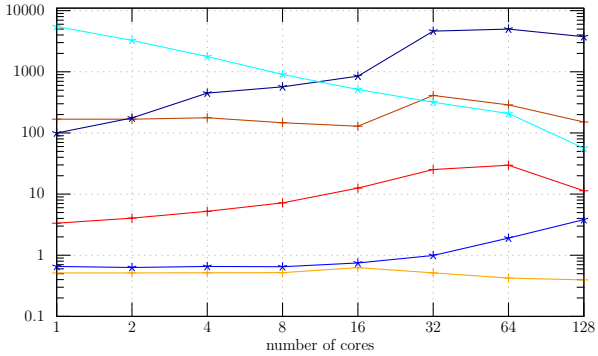
(a) Small (in s)



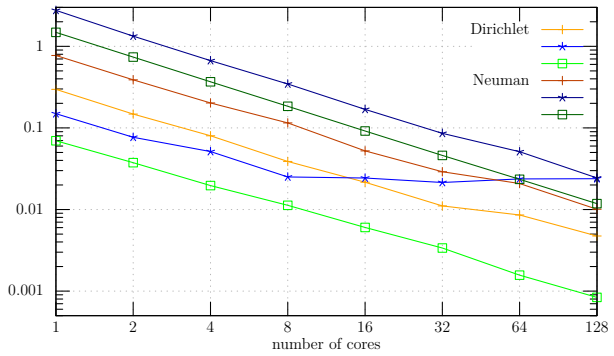
(b) Small (Ratio vs MFEM)



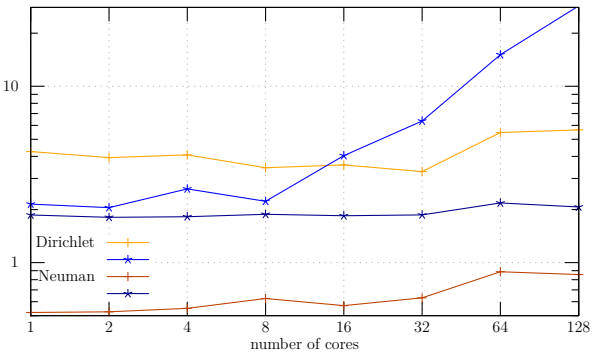
(c) Object creation (in s)



(d) Object creation (Ratio vs MFEM)



(e) Boundary condition (in s)



(f) Boundary condition (Ratio vs MFEM)

Figure 25: Elapsed time function of the number of processes: miscellaneous initiation, major object creation and Boundary conditions (in log-log scale except for y axes of d)