Usage of MGIS in Manta: a case of study

MFront User Meeting

06/11/2023

T. Helfer O. Jamond

CEA, DES, IRESNE, DEC, SESC, LMCP, CADARACHE, FRANCE

Outline

Introduction

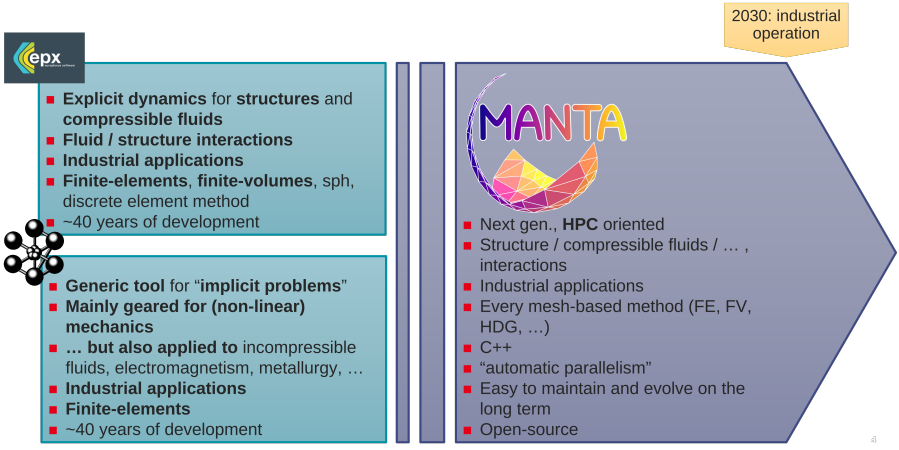
The need for code generation

Conclusions and perspectives

Addendum

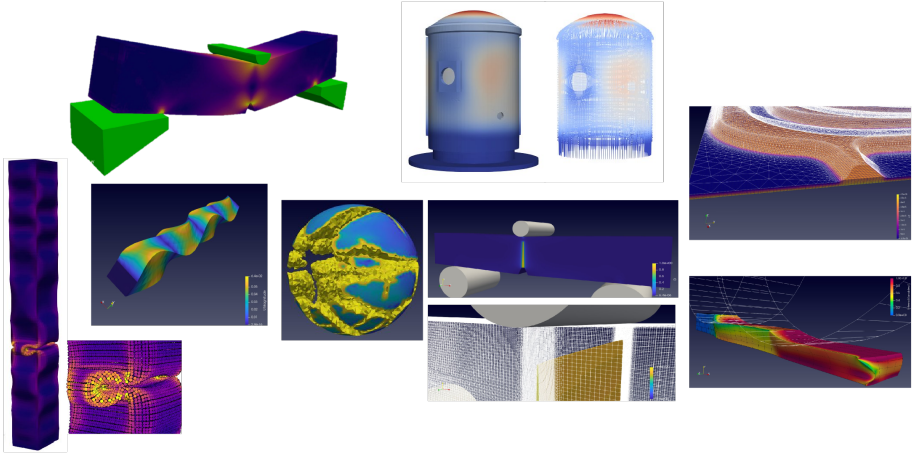
Introduction

About Manta

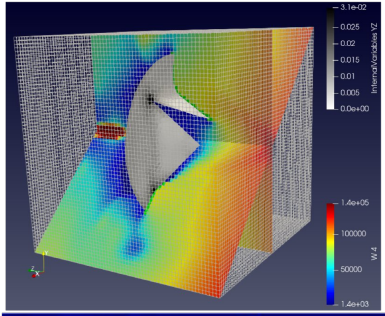


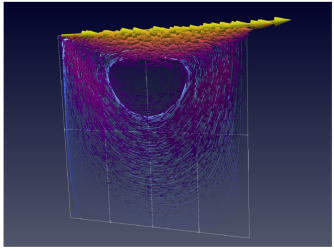
- `Manta` is a new solver developed at CEA.

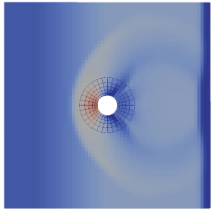
An overview of available functionalities

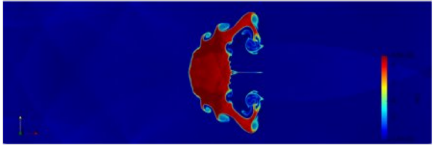


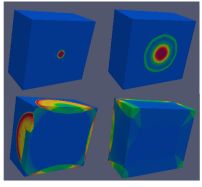
An overview of available functionalities - II

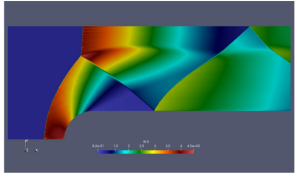












A first example in `python`

```
ps = PhysicalSystem(mesh = {"generator": "Grid",
                            "spaceDimension": 3, "numberOfElements": [1, 2, 4]})
 ...
mechanics_model = Model("ImplicitMechanics", ps)
mechanics_model.add("SolidMaterial", material = "cube",
                    library   = "./libHyperElasticityBehaviours-generic.so",
                    behavior = "SaintVenantKirchhoffElasticity");
mechanics_model.add("ImposedDisplacement", boundary = "right",
                    component = "Ux", value = "ImposedDisplacementLoading")
 ...
ps.setModel(mechanics_model);
 ...
ps.addLoadingEvolution("ImposedDisplacementLoading", {0 : 0, 1 : 1e-2})
 ...
s = Simulation(ps, [0, 0.2, 0.4, 0.6, 0.8, 1]))
s.run()
```

- Strong interaction with the behaviour thanks to `MGIS`.

A first example in `python`

```
ps = PhysicalSystem(mesh = {"generator": "Grid",
                            "spaceDimension": 3, "numberOfElements": [1, 2, 4]})
 ...
mechanics_model = Model("ImplicitMechanics", ps)
mechanics_model.add("SolidMaterial", material = "cube",
                    library   = "./libHyperElasticityBehaviours-generic.so",
                    behavior = "SaintVenantKirchhoffElasticity");
mechanics_model.add("ImposedDisplacement", boundary = "right",
                    component = "Ux", value = "ImposedDisplacementLoading")
 ...
ps.setModel(mechanics_model);
 ...
ps.addLoadingEvolution("ImposedDisplacementLoading", {0 : 0, 1 : 1e-2})
 ...
s = Simulation(ps, [0, 0.2, 0.4, 0.6, 0.8, 1]))
s.run()
```

- Strong interaction with the behaviour thanks to `MGIS`.
- The `dependency resolution algorithm`, adapted from LICOS [2], plays a central role.

A first example in `python`

```
ps = PhysicalSystem(mesh = {"generator": "Grid",
                            "spaceDimension": 3, "numberOfElements": [1, 2, 4]})
 ...
mechanics_model = Model("ImplicitMechanics", ps)
mechanics_model.add("SolidMaterial", material = "cube",
                    library   = "./libHyperElasticityBehaviours-generic.so",
                    behavior = "SaintVenantKirchhoffElasticity");
mechanics_model.add("ImposedDisplacement", boundary = "right",
                    component = "Ux", value = "ImposedDisplacementLoading")
 ...
ps.setModel(mechanics_model);
 ...
ps.addLoadingEvolution("ImposedDisplacementLoading", {0 : 0, 1 : 1e-2})
 ...
s = Simulation(ps, [0, 0.2, 0.4, 0.6, 0.8, 1]))
s.run()
```

- Strong interaction with the behaviour thanks to `MGIS`.
- The `dependency resolution algorithm`, adapted from LICOS [2], plays a central role.
- Internally, information are transfered by a very abstract object called `evaluator`:

A first example in `python`

```
ps = PhysicalSystem(mesh = {"generator": "Grid",
                            "spaceDimension": 3, "numberOfElements": [1, 2, 4]})
 ...
mechanics_model = Model("ImplicitMechanics", ps)
mechanics_model.add("SolidMaterial", material = "cube",
                    library   = "./libHyperElasticityBehaviours-generic.so",
                    behavior = "SaintVenantKirchhoffElasticity");
mechanics_model.add("ImposedDisplacement", boundary = "right",
                    component = "Ux", value = "ImposedDisplacementLoading")
 ...
ps.setModel(mechanics_model);
 ...
ps.addLoadingEvolution("ImposedDisplacementLoading", {0 : 0, 1 : 1e-2})
 ...
s = Simulation(ps, [0, 0.2, 0.4, 0.6, 0.8, 1]))
s.run()
```

- Strong interaction with the behaviour thanks to `MGIS`.
- The `dependency resolution algorithm`, adapted from LICOS [2], plays a central role.
- Internally, information are transfered by a very abstract object called `evaluator`:
  - The imposed displacement can be given by a uniform loading, a loading resoluting from data interpolation by kriging, another model (FSI), another application coupled by `ICoCo`, etc..

A first example in `python`

```
ps = PhysicalSystem(mesh = {"generator": "Grid",
                            "spaceDimension": 3, "numberOfElements": [1, 2, 4]})
 ...
mechanics_model = Model("ImplicitMechanics", ps)
mechanics_model.add("SolidMaterial", material = "cube",
                    library   = "./libHyperElasticityBehaviours-generic.so",
                    behavior = "SaintVenantKirchhoffElasticity");
mechanics_model.add("ImposedDisplacement", boundary = "right",
                    component = "Ux", value = "ImposedDisplacementLoading")
 ...
ps.setModel(mechanics_model);
 ...
ps.addLoadingEvolution("ImposedDisplacementLoading", {0 : 0, 1 : 1e-2})
 ...
s = Simulation(ps, [0, 0.2, 0.4, 0.6, 0.8, 1]))
s.run()
```

- Strong interaction with the behaviour thanks to `MGIS`.
- The `dependency resolution algorithm`, adapted from LICOS [2], plays a central role.
- Internally, information are transfered by a very abstract object called `evaluator`:
  - The imposed displacement can be given by a uniform loading, a loading resoluting from data interpolation by kriging, another model (FSI), another application coupled by `ICoCo`, etc..
  - This abstract feature has a small impact on the performances on CPUs. On GPUs, this may be a completely different story.

Manta's structure

- Manta's is currently divided in three layers: core, the modeling and end-user.

Manta's structure

- Manta's is currently divided in three layers: core, the modeling and end-user.
- The core layer (C++) manages everything related to:

Manta's structure

- Manta's is currently divided in three layers: core, the modeling and end-user.
- The core layer (C++) manages everything related to:
  - Meshes, with currently two backends (MOAB an PUMI), and fields.

Manta's structure

- Manta's is currently divided in three layers: core, the modeling and end-user.
- The core layer (C++) manages everything related to:
  - Meshes, with currently two backends (MOAB an PUMI), and fields.
  - Adaptative mesh refinement (work in progress).

Manta's structure

- Manta's is currently divided in three layers: core, the modeling and end-user.
- The core layer (C++) manages everything related to:
  - Meshes, with currently two backends (MOAB an PUMI), and fields.
  - Adaptative mesh refinement (work in progress).
  - Numerical methods (currently FEM, HHO, FVM).

Manta's structure

- Manta's is currently divided in three layers: core, the modeling and end-user.
- The core layer (C++) manages everything related to:
  - Meshes, with currently two backends (MOAB an PUMI), and fields.
  - Adaptative mesh refinement (work in progress).
  - Numerical methods (currently FEM, HHO, FVM).
  - High performance computing using the PETsC library.

Manta's structure

- Manta's is currently divided in three layers: `core`, the `modeling` and end-user.
- The `core` layer (C++) manages everything related to:
  - Meshes, with currently two backends (MOAB an PUMI), and fields.
  - Adaptative mesh refinement (work in progress).
  - Numerical methods (currently FEM, HHO, FVM).
  - High performance computing using the `PETsC` library.
  - `Dirichlet boundary conditions.`

Manta's structure

- Manta's is currently divided in three layers: core, the modeling and end-user.
- The core layer (C++) manages everything related to:
  - Meshes, with currently two backends (MOAB an PUMI), and fields.
  - Adaptative mesh refinement (work in progress).
  - Numerical methods (currently FEM, HHO, FVM).
  - High performance computing using the PETsC library.
  - Dirichlet boundary conditions.
- The modeling layer aims at implementing formulations, i.e. boundary conditions, structural elements, solid materials, compressible inviscid fluids, etc.

Manta's structure

- Manta's is currently divided in three layers: `core`, the `modeling` and end-user.
- The `core` layer (C++) manages everything related to:
  - Meshes, with currently two backends (MOAB an PUMI), and fields.
  - Adaptative mesh refinement (work in progress).
  - Numerical methods (currently FEM, HHO, FVM).
  - High performance computing using the `PETsC` library.
  - `Dirichlet boundary conditions.`
- The `modeling` layer aims at implementing formulations, i.e. boundary conditions, structural elements, solid materials, compressible inviscid fluids, etc.
  - for the various numerical methods available.

Manta's structure

- Manta's is currently divided in three layers: core, the modeling and end-user.
- The core layer (C++) manages everything related to:
  - Meshes, with currently two backends (MOAB an PUMI), and fields.
  - Adaptative mesh refinement (work in progress).
  - Numerical methods (currently FEM, HHO, FVM).
  - High performance computing using the PETsC library.
  - Dirichlet boundary conditions.
- The modeling layer aims at implementing formulations, i.e. boundary conditions, structural elements, solid materials, compressible inviscid fluids, etc.
  - for the various numerical methods available.
- The end-user layer (C++ and python) mostly:

Manta's structure

- Manta's is currently divided in three layers: `core`, the `modeling` and end-user.
- The `core` layer (C++) manages everything related to:
  - Meshes, with currently two backends (MOAB an PUMI), and fields.
  - Adaptative mesh refinement (work in progress).
  - Numerical methods (currently FEM, HHO, FVM).
  - High performance computing using the `PETsC` library.
  - `Dirichlet boundary conditions.`
- The `modeling` layer aims at implementing formulations, i.e. boundary conditions, structural elements, solid materials, compressible inviscid fluids, etc.
  - for the various numerical methods available.
- The `end-user` layer (C++ and `python`) mostly:
  - targets standard engineers. Most numerical details are hidden by default.

Manta's structure

- Manta's is currently divided in three layers: `core`, the `modeling` and end-user.
- The `core` layer (`C++`) manages everything related to:
  - Meshes, with currently two backends (MOAB an PUMI), and fields.
  - Adaptative mesh refinement (work in progress).
  - Numerical methods (currently FEM, HHO, FVM).
  - High performance computing using the `PETsC` library.
  - `Dirichlet boundary conditions.`
- The `modeling` layer aims at implementing formulations, i.e. boundary conditions, structural elements, solid materials, compressible inviscid fluids, etc.
  - for the various numerical methods available.
- The `end-user` layer (`C++` and `python`) mostly:
  - targets standard engineers. Most numerical details are hidden by default.
  - targets code coupling, thanks to the `ICoCo` standard.

Manta's structure

- Manta's is currently divided in three layers: `core`, the `modeling` and end-user.
- The `core` layer (C++) manages everything related to:
  - Meshes, with currently two backends (MOAB an PUMI), and fields.
  - Adaptative mesh refinement (work in progress).
  - Numerical methods (currently FEM, HHO, FVM).
  - High performance computing using the `PETsC` library.
  - `Dirichlet boundary conditions.`
- The `modeling` layer aims at implementing formulations, i.e. boundary conditions, structural elements, solid materials, compressible inviscid fluids, etc.
  - for the various numerical methods available.
- The `end-user` layer (C++ and `python`) mostly:
  - targets standard engineers. Most numerical details are hidden by default.
  - targets code coupling, thanks to the `ICoCo` standard.
  - aims at being simple to use, focused on the physics, documented, verified and stable in time.

Main concepts of the end-user layer

- The end-user layer simulates the evolution of the state of a physical system du to several physical phenomena which are described by models under the action of external loadings.

Main concepts of the end-user layer

- The end-user layer simulates the evolution of the state of a physical system du to several physical phenomena which are described by models under the action of external loadings.
- The models are generally coupled: the coupling scheme describes the numerical algorithm coupling the models.

Main concepts of the end-user layer

- The end-user layer simulates the evolution of the state of a physical system du to several physical phenomena which are described by models under the action of external loadings.
- The models are generally coupled: the coupling scheme describes the numerical algorithm coupling the models.
- Models are made of **bricks** describing the boundary conditions, structural elements or solid materials.

Main concepts of the end-user layer

- The end-user layer simulates the evolution of the state of a physical system du to several physical phenomena which are described by models under the action of external loadings.
- The models are generally coupled: the coupling scheme describes the numerical algorithm coupling the models.
- Models are made of **bricks** describing the boundary conditions, structural elements or solid materials.
- Bricks may rely on external properties or behaviours to describe the structures or the materials.

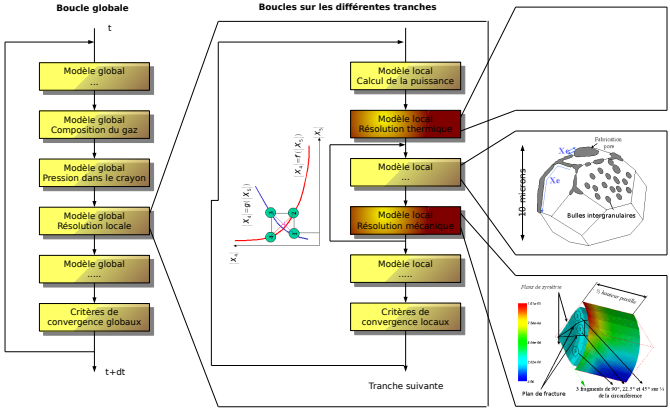
Main concepts of the end-user layer

- The end-user layer simulates the evolution of the state of a physical system du to several physical phenomena which are described by models under the action of external loadings.
- The models are generally coupled: the coupling scheme describes the numerical algorithm coupling the models.
- Models are made of **bricks** describing the boundary conditions, structural elements or solid materials.
- Bricks may rely on external properties or behaviours to describe the structures or the materials.
  - In general, properties and behaviours are generated by `MFront`.

Main concepts of the end-user layer

- The end-user layer simulates the evolution of the state of a physical system du to several physical phenomena which are described by models under the action of external loadings.
- The models are generally coupled: the coupling scheme describes the numerical algorithm coupling the models.
- Models are made of **bricks** describing the boundary conditions, structural elements or solid materials.
- Bricks may rely on external properties or behaviours to describe the structures or the materials.
  - In general, properties and behaviours are generated by `MFront`.
- Internally, bricks must participate to the dependency management algorithm and generate appropriate *formulations*.

Focus on a coupling scheme



- in `Manta`, the user will be able to build arbitrary complex coupling schemes, with accelerators, convergence criteria, etc..

Plugins

- The end-user API is extremly extensible thanks to abstract factory:

Plugins

- The end-user API is extremly extensible thanks to abstract factory:
  - `CurveFactory`, `PostProcessingFactory`, `ModelBrickFactory`, `ModelFactory`, `LoadingFactory`, `PropertyFactory`, `CouplingSchemeFactory`, `CompressibleInviscidFluidSolverFactory`, `ImplicitMechanicsSolidMaterialFormulationsFactory`, etc..

Plugins

- The end-user API is extremly extensible thanks to abstract factory:
  - `CurveFactory`, `PostProcessingFactory`, `ModelBrickFactory`, `ModelFactory`, `LoadingFactory`, `PropertyFactory`, `CouplingSchemeFactory`, `CompressibleInviscidFluidSolverFactory`, `ImplicitMechanicsSolidMaterialFormulationsFactory`, etc..
- Objects generated by plugins are totally equivalent to native objects.

Plugins

- The end-user API is extremly extensible thanks to abstract factory:
  - `CurveFactory`, `PostProcessingFactory`, `ModelBrickFactory`, `ModelFactory`, `LoadingFactory`, `PropertyFactory`, `CouplingSchemeFactory`, `CompressibleInviscidFluidSolverFactory`, `ImplicitMechanicsSolidMaterialFormulationsFactory`, etc..
- Objects generated by plugins are totally equivalent to native objects.
- The `core` layer is used to create the plugins.

Plugins

- The end-user API is extremly extensible thanks to abstract factory:
  - `CurveFactory`, `PostProcessingFactory`, `ModelBrickFactory`, `ModelFactory`, `LoadingFactory`, `PropertyFactory`, `CouplingSchemeFactory`, `CompressibleInviscidFluidSolverFactory`, `ImplicitMechanicsSolidMaterialFormulationsFactory`, etc..
- Objects generated by plugins are totally equivalent to native objects.
- The `core` layer is used to create the plugins.
- Plugins are not covered by the stability guarantee of the end-user API.

The first example revisited

```
ps = PhysicalSystem(mesh = {"generator": "Grid",
                            "spaceDimension": 3, "numberOfElements": [1, 2, 4]})
 ...
mechanics_model = Model("ImplicitMechanics", ps)
mechanics_model.add("SolidMaterial", material = "cube",
                    library   = "./libHyperElasticityBehaviours-generic.so",
                    behavior = "SaintVenantKirchhoffElasticity");
mechanics_model.add("ImposedDisplacement", boundary = "right",
                    component = "Ux", value = "ImposedDisplacementLoading")
 ...
auto c = LoopCoupling(1)
c.add(mechanics_model)
ps.setCouplingScheme(c);
 ...
ps.addLoadingEvolution("ImposedDisplacementLoading", {0 : 0, 1 : 1e-2})
 ...
s = Simulation(ps, [0, 0.2, 0.4, 0.6, 0.8, 1]))
s.run()
```

- Explicit declaration of the coupling scheme.

Other commented examples

- Nonlinear heat transfer.
- Notched bar (ssna303).

The need for code generation

Toward code generation

$$\vec{j} = q\,(T - T_\infty)\,\vec{n}$$

- Let us consider a boundary condition for heat transfer describing a linear convection.

Toward code generation

$$\vec{j} = q\,(T - T_\infty)\,\vec{n}$$

- Let us consider a boundary condition for heat transfer describing a linear convection.
- The physical part of a plugin describing this condition is limited:

Toward code generation

$$\vec{j} = q\,(T - T_\infty)\,\vec{n}$$

- Let us consider a boundary condition for heat transfer describing a linear convection.
- The physical part of a plugin describing this condition is limited:
  - 26 lines of codes for the *integrand*, on which is build a dedicated *formulation*.

Toward code generation

$$\vec{j} = q\,(T - T_\infty)\,\vec{n}$$

- Let us consider a boundary condition for heat transfer describing a linear convection.
- The physical part of a plugin describing this condition is limited:
  - 26 lines of codes for the *integrand*, on which is build a dedicated *formulation*.
  - The plugins is 443 lines long (with comments), mostly for the brick which handles the dependency management algorithms and lot of internal details.

Toward code generation

$$\vec{j} = q\,(T - T_\infty)\,\vec{n}$$

- Let us consider a boundary condition for heat transfer describing a linear convection.
- The physical part of a plugin describing this condition is limited:
  - 26 lines of codes for the *integrand*, on which is build a dedicated *formulation*.
  - The plugins is 443 lines long (with comments), mostly for the brick which handles the dependency management algorithms and lot of internal details.
  - The physically relevant part is only 6% of the total number of lines.

Toward code generation

$$\vec{j} = q\,(T - T_\infty)\,\vec{n}$$

- Let us consider a boundary condition for heat transfer describing a linear convection.
- The physical part of a plugin describing this condition is limited:
  - 26 lines of codes for the *integrand*, on which is build a dedicated *formulation*.
  - The plugins is 443 lines long (with comments), mostly for the brick which handles the dependency management algorithms and lot of internal details.
  - The physically relevant part is only 6% of the total number of lines.
  - The rest is only boilerplate.

A (imaginative)-example of an MFront-like input file

```
@DSL ModelBrick{
  location : "boundaries",
  method: "fem",
  jacobian: true
  configuration : current
};

@Unknown temperature T;
T.setGlossaryName("Temperature");

@MaterialProperty real h;
@ExternalStateVariable temperature T_inf;

@ShapeFunctions N;

@Integrator{
 // the integration point weight is automatically defined
 R = h * (T - T_inf) * N * w;
 dR_dT = h * N * N.transpose() * w;
}
```

- This is the minimal information required to build the plugin in a MFront-like input file.

A (imaginative)-example of an MFront-like input file

```
@DSL ModelBrick{
  location : "boundaries",
  method: "fem",
  jacobian: true
  configuration : current
};

@Unknown temperature T;
T.setGlossaryName("Temperature");

@MaterialProperty real h;
@ExternalStateVariable temperature T_inf;

@ShapeFunctions N;

@Integrator{
 // the integration point weight is automatically defined
 R = h * (T - T_inf) * N * w;
 dR_dT = h * N * N.transpose() * w;
}
```

- This is the minimal information required to build the plugin in a MFront-like input file.
  - That would be limited to FEM, what about HHO, FVM, VEM ?

A (imaginative)-example of an `MFront`-like input file

```
@DSL ModelBrick{
  location : "boundaries",
  method: "fem",
  jacobian: true
  configuration : current
};

@Unknown temperature T;
T.setGlossaryName("Temperature");

@MaterialProperty real h;
@ExternalStateVariable temperature T_inf;

@ShapeFunctions N;

@Integrator{
 // the integration point weight is automatically defined
 R = h * (T - T_inf) * N * w;
 dR_dT = h * N * N.transpose() * w;
}
```

- This is the minimal information required to build the plugin in a `MFront`-like input file.
  - That would be limited to `FEM`, what about `HHO`, `FVM`, `VEM` ?
  - Moreover, `dR_dT` could be computed by automatic differentiation.

A (imaginative)-example of an MFront-like input file

```
@DSL ModelBrick{
  location : "boundaries",
  method: "fem",
  jacobian: true
  configuration : current
};

@Unknown temperature T;
T.setGlossaryName("Temperature");

@MaterialProperty real h;
@ExternalStateVariable temperature T_inf;

@ShapeFunctions N;

@Integrator{
 // the integration point weight is automatically defined
 R = h * (T - T_inf) * N * w;
 dR_dT = h * N * N.transpose() * w;
}
```

- This is the minimal information required to build the plugin in a MFront-like input file.
  - That would be limited to FEM, what about HHO, FVM, VEM ?
  - Moreover, dR_dT could be computed by automatic differentiation.
  - This plugin shall be compiled Just-in-time to exploit the characteristics of the h and T_∞.

A (imaginative)-example of a mixed UFL/MFront-like input file

```
@DSL Formulation {configuration: current};

@Unknown temperature T;
T.setGlossaryName("Temperature");
@TrialFunction temperature Th;

@MaterialProperty real h;
@ExternalStateVariable temperature T_inf;

@Formulation{
  R = h * (T - T_inf) * Th * ds;
}
```

- Ideally, one would like to combine tightly UFL [1, 6] and MFront:

A (imaginative)-example of a mixed UFL/MFront-like input file

```
@DSL Formulation {configuration: current};

@Unknown temperature T;
T.setGlossaryName("Temperature");
@TrialFunction temperature Th;

@MaterialProperty real h;
@ExternalStateVariable temperature T_inf;

@Formulation{
  R = h * (T - T_inf) * Th * ds;
}
```

- Ideally, one would like to combine tightly UFL [1, 6] and MFront:
  - UFL is currently limited to FEM and DG.

A (imaginative)-example of a mixed UFL/MFront-like input file

```
@DSL Formulation {configuration: current};

@Unknown temperature T;
T.setGlossaryName("Temperature");
@TrialFunction temperature Th;

@MaterialProperty real h;
@ExternalStateVariable temperature T_inf;

@Formulation{
  R = h * (T - T_inf) * Th * ds;
}
```

- Ideally, one would like to combine tightly UFL [1, 6] and MFront:
  - UFL is currently limited to FEM and DG.
  - HDG and VEM could theoritically be obtained from the Hu-Washizu variational formulation [4, 5]

A (imaginative)-example of a mixed UFL/MFront-like input file

```
@DSL Formulation {configuration: current};

@Unknown temperature T;
T.setGlossaryName("Temperature");
@TrialFunction temperature Th;

@MaterialProperty real h;
@ExternalStateVariable temperature T_inf;

@Formulation{
  R = h * (T - T_inf) * Th * ds;
}
```

- Ideally, one would like to combine tightly UFL [1, 6] and MFront:
  - UFL is currently limited to FEM and DG.
  - HDG and VEM could theoritically be obtained from the Hu-Washizu variational formulation [4, 5]
- This tool would considerably reduced Manta's maintainance and ease supporting new numerical techniques (matrix-free solvers) or architectures (GPUs).

A (imaginative)-example of a mixed UFL/MFront-like input file

```
@DSL Formulation {configuration: current};

@Unknown temperature T;
T.setGlossaryName("Temperature");
@TrialFunction temperature Th;

@MaterialProperty real h;
@ExternalStateVariable temperature T_inf;

@Formulation{
  R = h * (T - T_inf) * Th * ds;
}
```

- Ideally, one would like to combine tightly UFL [1, 6] and MFront:
  - UFL is currently limited to FEM and DG.
  - HDG and VEM could theoritically be obtained from the Hu-Washizu variational formulation [4, 5]
- This tool would considerably reduced Manta's maintainance and ease supporting new numerical techniques (matrix-free solvers) or architectures (GPUs).
- This tool shall have muliple interfaces (just like MFront) and shall not be limited to Manta nor MFront:

A (imaginative)-example of a mixed UFL/MFront-like input file

```
@DSL Formulation {configuration: current};

@Unknown temperature T;
T.setGlossaryName("Temperature");
@TrialFunction temperature Th;

@MaterialProperty real h;
@ExternalStateVariable temperature T_inf;

@Formulation{
  R = h * (T - T_inf) * Th * ds;
}
```

- Ideally, one would like to combine tightly UFL [1, 6] and MFront:
  - UFL is currently limited to FEM and DG.
  - HDG and VEM could theoritically be obtained from the Hu-Washizu variational formulation [4, 5]
- This tool would considerably reduced Manta's maintainance and ease supporting new numerical techniques (matrix-free solvers) or architectures (GPUs).
- This tool shall have muliple interfaces (just like MFront) and shall not be limited to Manta nor MFront:
  - OpenGeoSys, code_aster, ASet, etc... ?

A (imaginative)-example of a mixed UFL/MFront-like input file

```
@DSL Formulation {configuration: current};

@Unknown temperature T;
T.setGlossaryName("Temperature");
@TrialFunction temperature Th;

@MaterialProperty real h;
@ExternalStateVariable temperature T_inf;

@Formulation{
  R = h * (T - T_inf) * Th * ds;
}
```

- Ideally, one would like to combine tightly UFL [1, 6] and MFront:
  - UFL is currently limited to FEM and DG.
  - HDG and VEM could theoritically be obtained from the Hu-Washizu variational formulation [4, 5]
- This tool would considerably reduced Manta's maintainance and ease supporting new numerical techniques (matrix-free solvers) or architectures (GPUs).
- This tool shall have muliple interfaces (just like MFront) and shall not be limited to Manta nor MFront:
  - OpenGeoSys, code_aster, ASet, etc... ?
- All those ideas are the basis of the subject of a PHD thesis.

A (imaginative)-example of a mixed UFL/MFront-like input file

```
@DSL Formulation {configuration: current};

@Unknown temperature T;
T.setGlossaryName("Temperature");
@TrialFunction temperature Th;

@MaterialProperty real h;
@ExternalStateVariable temperature T_inf;

@Formulation{
  R = h * (T - T_inf) * Th * ds;
}
```

- Ideally, one would like to combine tightly UFL [1, 6] and MFront:
  - UFL is currently limited to FEM and DG.
  - HDG and VEM could theoritically be obtained from the Hu-Washizu variational formulation [4, 5]
- This tool would considerably reduced Manta's maintainance and ease supporting new numerical techniques (matrix-free solvers) or architectures (GPUs).
- This tool shall have muliple interfaces (just like MFront) and shall not be limited to Manta nor MFront:
  - OpenGeoSys, code_aster, ASet, etc... ?
- All those ideas are the basis of the subject of a PHD thesis.
- This may be the end of UFL as we know it

Conclusions and perspectives

Conclusions

- `Manta` is a young project that will be released as an open-source project.

Conclusions

- `Manta` is a young project that will be released as an open-source project.
  - Probably an early α release in the beginning of 2024 to get feed-backs for adventurous users.

Conclusions

- `Manta` is a young project that will be released as an open-source project.
  - Probably an early α release in the beginning of 2024 to get feed-backs for adventurous users.
- The end-user layer is already quite flexible and powerful.

Conclusions

- `Manta` is a young project that will be released as an open-source project.
  - Probably an early α release in the beginning of 2024 to get feed-backs for adventurous users.
- The end-user layer is already quite flexible and powerful.
  - In some aspects, it is already offers more features than `Cast3M` !

Conclusions

- `Manta` is a young project that will be released as an open-source project.
  - Probably an early α release in the beginning of 2024 to get feed-backs for adventurous users.
- The end-user layer is already quite flexible and powerful.
  - In some aspects, it is already offers more features than `Cast3M` !
  - A lot of work still needs to be done to offer decent functionalities to the average user "out of the box".

Conclusions

- `Manta` is a young project that will be released as an open-source project.
  - Probably an early α release in the beginning of 2024 to get feed-backs for adventurous users.
- The end-user layer is already quite flexible and powerful.
  - In some aspects, it is already offers more features than `Cast3M` !
  - A lot of work still needs to be done to offer decent functionalities to the average user "out of the box".
    - A lot of standard boundary conditions, structural elements are not yet implemented or only available in the `modeling` layer.

Conclusions

- `Manta` is a young project that will be released as an open-source project.
  - Probably an early α release in the beginning of 2024 to get feed-backs for adventurous users.
- The end-user layer is already quite flexible and powerful.
  - In some aspects, it is already offers more features than `Cast3M` !
  - A lot of work still needs to be done to offer decent functionalities to the average user "out of the box".
    - A lot of standard boundary conditions, structural elements are not yet implemented or only available in the `modeling` layer.
    - The only nonlinear solver available is the basic Newton-Raphson algorithm. A proper management of linear solvers must be built. There is no prediction step, etc.

Conclusions

- `Manta` is a young project that will be released as an open-source project.
  - Probably an early α release in the beginning of 2024 to get feed-backs for adventurous users.
- The end-user layer is already quite flexible and powerful.
  - In some aspects, it is already offers more features than `Cast3M` !
  - A lot of work still needs to be done to offer decent functionalities to the average user "out of the box".
    - A lot of standard boundary conditions, structural elements are not yet implemented or only available in the `modeling` layer.
    - The only nonlinear solver available is the basic Newton-Raphson algorithm. A proper management of linear solvers must be built. There is no prediction step, etc.
    - The end-user API is currently limited to FEM (no HHO) and implicit mechanics/heat transfer. Implicit damage gradient models and explicit mechanics shall be available by the end of the year. HHO will be introduced in the early 2024.

Conclusions

- `Manta` is a young project that will be released as an open-source project.
  - Probably an early α release in the beginning of 2024 to get feed-backs for adventurous users.
- The end-user layer is already quite flexible and powerful.
  - In some aspects, it is already offers more features than `Cast3M` !
  - A lot of work still needs to be done to offer decent functionalities to the average user "out of the box".
    - A lot of standard boundary conditions, structural elements are not yet implemented or only available in the `modeling` layer.
    - The only nonlinear solver available is the basic Newton-Raphson algorithm. A proper management of linear solvers must be built. There is no prediction step, etc.
    - The end-user API is currently limited to FEM (no HHO) and implicit mechanics/heat transfer. Implicit damage gradient models and explicit mechanics shall be available by the end of the year. HHO will be introduced in the early 2024.
    - Documentation is under construction.

Conclusions

- `Manta` is a young project that will be released as an open-source project.
  - Probably an early α release in the beginning of 2024 to get feed-backs for adventurous users.
- The end-user layer is already quite flexible and powerful.
  - In some aspects, it is already offers more features than `Cast3M` !
  - A lot of work still needs to be done to offer decent functionalities to the average user "out of the box".
    - A lot of standard boundary conditions, structural elements are not yet implemented or only available in the `modeling` layer.
    - The only nonlinear solver available is the basic Newton-Raphson algorithm. A proper management of linear solvers must be built. There is no prediction step, etc.
    - The end-user API is currently limited to FEM (no HHO) and implicit mechanics/heat transfer. Implicit damage gradient models and explicit mechanics shall be available by the end of the year. HHO will be introduced in the early 2024.
    - Documentation is under construction.
    - Test cases and verifications are not yet satisfying.

Conclusions

- `Manta` is a young project that will be released as an open-source project.
  - Probably an early α release in the beginning of 2024 to get feed-backs for adventurous users.
- The end-user layer is already quite flexible and powerful.
  - In some aspects, it is already offers more features than `Cast3M` !
  - A lot of work still needs to be done to offer decent functionalities to the average user "out of the box".
    - A lot of standard boundary conditions, structural elements are not yet implemented or only available in the `modeling` layer.
    - The only nonlinear solver available is the basic Newton-Raphson algorithm. A proper management of linear solvers must be built. There is no prediction step, etc.
    - The end-user API is currently limited to FEM (no HHO) and implicit mechanics/heat transfer. Implicit damage gradient models and explicit mechanics shall be available by the end of the year. HHO will be introduced in the early 2024.
    - Documentation is under construction.
    - Test cases and verifications are not yet satisfying.
- `Manta` will be a sandbox for many ideas related to `MFront` and `MGIS`, notably for JIT compilation and code-generation

Conclusions

- `Manta` is a young project that will be released as an open-source project.
  - Probably an early α release in the beginning of 2024 to get feed-backs for adventurous users.
- The end-user layer is already quite flexible and powerful.
  - In some aspects, it is already offers more features than `Cast3M` !
  - A lot of work still needs to be done to offer decent functionalities to the average user "out of the box".
    - A lot of standard boundary conditions, structural elements are not yet implemented or only available in the `modeling` layer.
    - The only nonlinear solver available is the basic Newton-Raphson algorithm. A proper management of linear solvers must be built. There is no prediction step, etc.
    - The end-user API is currently limited to FEM (no HHO) and implicit mechanics/heat transfer. Implicit damage gradient models and explicit mechanics shall be available by the end of the year. HHO will be introduced in the early 2024.
    - Documentation is under construction.
    - Test cases and verifications are not yet satisfying.
- `Manta` will be a sandbox for many ideas related to `MFront` and `MGIS`, notably for JIT compilation and code-generation
  - The tools developped will be designed to be usable by other solvers, just like `MFront` is.

Avalaible positions

- Two interships:
  - Implementation and verification of damage gradient models in Manta using the finite element method (FEM) and the Hybrid High Order method: application to PWR nuclear fuel element.
  - Development of new boundary conditions and verification of Manta for the simulation of nuclear fuel elements.
- Two PHD positions (october 2024):
  - Damage gradients models discretized by the Hybrid High Order method: adaptative mesh refinement and transition between damage and cracks.
    - with O. Fandeur, O. Jamond, S. Feld-Payet and J. Besson as supervisors.
  - JIT Automated generation of complex kernels for nonlinear multiphysics simulations.
    - with O. Jamond, B. Marchand and P. Kerfriden as supervisors.

Thanks for you attention ! Any questions?

References I

[1] Martin S. Alnæs et al. “Unified Form Language: A Domain-Specific Language for Weak Formulations of Partial Differential Equations”. In: ACM Trans. Math. Softw. 40.2 (Mar. 2014). Place: New York, NY, USA Publisher: Association for Computing Machinery. ISSN: 0098-3500. DOI: 10.1145/2566630. URL: https://doi.org/10.1145/2566630.

[2] Thomas Helfer, Syriac Bejaoui, and Bruno Michel. “Licos, a fuel performance code for innovative fuel elements or experimental devices design”. In: Nuclear Engineering and Design 294 (Dec. 1, 2015), pp. 117–136. ISSN: 0029-5493. DOI: 10.1016/j.nucengdes.2015.07.070. URL: http://www.sciencedirect.com/science/article/pii/S0029549315003842 (visited on 11/30/2015).

References II

[3] Olivier Jamond et al. “MANTA : un code HPC généraliste pour la simulation de problèmes complexes en mécanique”. In: CSMA 2022 15ème Colloque National en Calcul des Structures. Giens, France, May 2022. URL: https://hal.archives-ouvertes.fr/hal-03688160.

[4] Andrea Lamperti et al. “A Hu–Washizu variational approach to self-stabilized virtual elements: 2D linear elastostatics”. In: Computational Mechanics 71.5 (May 1, 2023), pp. 935–955. ISSN: 1432-0924. DOI: 10.1007/s00466-023-02282-2. URL: https://doi.org/10.1007/s00466-023-02282-2 (visited on 11/05/2023).

[5] David Siedel. “Une approche numérique robuste pour la description de la rupture fragile et du comportement viscoplastique des crayons de combustible”. These en préparation. Université Paris sciences et lettres, 2023. URL: https://www.theses.fr/s239728 (visited on 06/23/2023).

References III

[6] UFL — Unified Form Language (UFL) 2021.1.0 documentation. URL: https://fenics.readthedocs.io/projects/ufl/en/latest/ (visited on 11/05/2023).