A brief introduction to the MGIS C++ library for mechanical behaviours

Technical Report · May 2019

DOI: 10.13140/RG.2.2.22079.76968

1 author:

Thomas Helfer
Atomic Energy and Alternative Energies Commission
32 PUBLICATIONS **82** CITATIONS

SEE PROFILE

A brief introduction to the MGIS C++ library for mechanical behaviours

Thomas Helfer

14/05/2019

Contents

1 Introduction

This introduction is meant to enlight how the use the MGIS library in C++ to integrate a mechanical behaviour. This document targets solver developers who wish to integrate MGIS.

Note

This introduction will not describe formally the `MGIS` API (Application Programming Interface) which would be tedious. The reader may refer to the `doxygen` documentation of the project.

2 Creating a `MFront` behaviour

The first step to use `MGIS` is to generate via `MFront` a shared library which will be loaded by the `MGIS` library.

About the MFront/MGIS versions compatibility

This means that a working implementation of `MFront` is available. The user is responsible to use a version of `MFront` compatible with the version of `MGIS` he plans to use. This is explicitly discussed in the `README.md` file at the root the `MGIS` sources. Failing to use compatible versions may lead to various errors.

In this document, we will consider two mechanical behaviours:

- a simple small strain viscoplastic behaviour based on the Norton-Hoff Law. This file is available here.
- a simple finite strain plastic behaviour based on J_2 plasticity with linear hardening in the logarithmic space (see Miehe, Apel, and Lambrecht (2002)). This file is available here

Those files can be compiled as follows from the command line[1]:

```
# generate the C++ sources associated
# with the Norton behaviour
$ mfront --interface=generic Norton.mfront
# generate the C++ sources associated
# with the plastic behaviour
$ mfront --interface=generic Plasticity.mfront
# generate the library
$ mfront --obuild
Treating target : all
The following library has been built :
- libBehaviour.so :  Norton_AxisymmetricalGeneralisedPlaneStrain
Norton_Axisymmetrical
Norton_PlaneStrain Norton_GeneralisedPlaneStrain
Norton_Tridimensional Plasticity_AxisymmetricalGeneralisedPlaneStrain
Plasticity_Axisymmetrical Plasticity_PlaneStrain
Plasticity_GeneralisedPlaneStrain Plasticity_Tridimensional
```

The output shows:

- That the compilation process led to the creation of the `libBehaviour.so` library.
- That `MFront` has generated one function per behaviour and per supported modelling hypothesis.

For the following, it is important to note the name of the behaviour generated by `MFront`.

About the `TFEL` libraries

The generated library `libBehaviour.so` is linked to various libraries provided by the `TFEL` project. Those libraries must be available in the current environment for the library to be usable in `MGIS`.

3 Loading the behaviour, the `Behaviour` object

The `load` function is used to load the behaviour. This function returns a `Behaviour` object which contains all the relevant information about the behaviour and a function pointer.

[1] For the example, we used the Bourne-again shell (`bash`) syntax here.

3.1 Small strain behaviours

The first overload of the `load` function takes three parameters:

- the name of the library.
- the name of the function.
- the modelling hypothesis to be used.

```
const auto h = Hypothesis::TRIDIMENSIONAL;
const auto b = load('src/libBehaviour.so', 'Norton', h);
```

The `mgis::behaviour` namespace

The example given in this document assumes that the functions and objects defined in the `mgis::behaviour` namespace are available in the current scope.

In pratice, this means that the following statement was made earlier:

```
using namespace mgis::behaviour;
```

3.2 Finite strain behaviours

The second overload of the `load` function specifically handles finite strain behaviours and takes four parameters:

- options related the stress measure used on input/output and the consistenant tangent operator expected.
- the name of the library.
- the name of the function.
- the modelling hypothesis to be used.

The following stress measures are available:

- `CAUCHY`, the Cauchy stress
- `PK1`, the first Piola-Kirchoff stress
- `PK2`, the second Piola-Kirchoff stress

The following tangent operators are available:

- `DSIG_DF`, the derivative of the Cauchy stress with respect to the deformation gradient.
- `DS_DEGL`, the derivative of the second Piola-Kirchoff stress with respect to the Green-Lagrange strain.
- `DPK1_DF`, the derivative of the first Piola-Kirchoff stress with respect to the deformation gradient.

```
const auto h = Hypothesis::TRIDIMENSIONAL;
auto o = FiniteStrainBehaviourOptions{};
o.stress_measure = FiniteStrainBehaviourOptions::PK1;
o.tangent_operator = FiniteStrainBehaviourOptions::DPK1_DF;
const auto b = load(o, 'src/libBehaviour.so', 'Norton', h);
```

3.3 The `Behaviour` object

The `Behaviour` object contains all the metadata required to interact with the behaviour. A complete description of this object is exceed the scope of this document. Thus, only the analysis of the material properties required by the behaviour is discussed here. Handling external state variables, parameters is made in a very similar manner.

3.3.1 Material properties

The list of material properties required by the user is defined in the `mps` data member as a vector of object of the `Variable` type.

The `Variable` data structure is quite simple:

```
struct Variable {
  //! name of the variable
  std::string name;
  //! type of the variable
  enum Type { SCALAR = 0, VECTOR = 1, STENSOR = 2, TENSOR = 3} type;
};  // end of struct Description
```

It contains two data members:

- the name
- the type

For material properties (and external state variables), only scalar variables are allowed.

With those informations, a solver developer is already able to check the validity of an input file.

4 State of an integration point, memory allocation

With the `Behaviour` data structure, one are able to describe a behaviour. The next step is to use this information to create one or more data structures containing the state of an integration point or a set of integrations points.

Three cases can be met in practice:

1. The developer may choose to associate to each integration point an object of the type `BehaviourData` provided by `MGIS`.
2. The developer may choose to use the `MaterialDataManager` provided by `MGIS` to handle the data associated with a set of integration points.
3. The developer may choose to store the state of the material in its own data structures.

Whatever the choice made, at the end, it must be emphasized that the behaviour integration is based on an object of the type `BehaviourDataView` which only contains pointers to a previously allocated memory. A careful study of this data structure may help to properly design the integration of `MGIS` in the solver and eventually minimize the data transfer between the code and the `MGIS` (and even, hopefully, eliminate all data transfer).

Thus, the `BehaviourDataView` structure, and the underlying `StateView` structure, are now discussed. Then more details about the three cases described previously are given.

4.1 The `BehaviourDataView` and `StateView` structures

The `BehaviourDataView` data structure contains the following data members:

- `dt`, the time increment. This is left unmodified by the behaviour integration.
- `K`, a pointer to an array which can hold the consistent tangent operator computed by the behaviour integration.
- `rdt`: proposed time step increment increase factor
- `s0`: the state of the integration point at the beginning of the time step. This is left unmodified by the behaviour integration.
- `s1`: the state of the integration point at the end of the time step.

`s0` and `s1` and instances of the `StateView` structure.

4.1.1 Description of the initial values stored in `K`

On input, the first element of K must contain the type of type of stiffness matrix expected. If this value is negative, only the prediction operator is computed. This value has the following meaning:

- if K[0] is lower than -2.5, the tangent operator must be computed.
- if K[0] is in [-2.5:-1.5]: the secant operator must be computed.
- if K[0] is in [-1.5:-0.5]: the elastic operator must be computed.
- if K[0] is in [-0.5:0.5]: the behaviour integration is performed, but no stiffness matrix.
- if K[0] is in [0.5:1.5]: the elastic operator must be computed.
- if K[0] is in [1.5:2.5]: the secant operator must be computed.
- if K[0] is in [2.5:3.5]: the secant operator must be computed.
- if K[0] is greater than 3.5, the consistent tangent operator must be computed.

Other values of K are meant to store behaviour's option. This is currently only meaningful for finite strain behaviours.

For finite strain behaviours, K[1] holds the choice of the stress measure used by the behaviour on input/output:

- if $K[1] < 0.5$, the Cauchy stress is used
- if $0.5 < K[1] < 1.5$, the second Piola-Kirchoff stress is used
- if $1.5 < K[1] < 2.5$, the first Piola-Kirchoff stress is used

For finite strain behaviours, K[2] holds the choice of the consistent tangent operator returned by the behaviour:

- if $K[2]<0.5$, the derivative of the Cauchy stress with respect to the deformation gradient is returned
- if $0.5<K[2]<1.5$, the derivative of the second Piola-Kirchoff stress with respect to the Green-Lagrange strain is returned
- if $1.5<K[2]<2.5$, the derivative of the first Piola-Kirchoff stress with respect to the deformation gradient is returned

It is important to note that, even if no stiffness matrix is requested, the `K` array must a least be big enough to store the kind of integration to be performed and the behaviour options.

The `Behaviour` structure contains a static constant variable called `nopts` containing the maximum number of options that a `Behaviour` may required. Thus, if no stiffness matrix is requested, `K` must have a size which is at least `Behaviour::nopts+1`.

It is also important to note that behaviour options are automatically set by the `integrate` function describe hereafter.

4.1.2 The `StateView` structure

This `StateView` structure contains the following members, describing the state of the material at one integration points at a given time:

- `gradients`: pointer to an array containing the values of the gradients. This is left unmodified by the behaviour integration.
- `thermodynamic_forces`: pointer to an array containing the values of the thermodynamic forces.
- `material_properties`: pointer to an array containing the values of the material properties. This is left unmodified by the behaviour integration.
- `internal_state_variables`: pointer to an array containing the values of the internal state variables.
- `stored_enery`: pointer to the value of the stored energy (computed by `@InternalEnergy` in `MFront` files) This output is optional.
- `dissipated_enery`: pointer to the value of the dissipated energy (computed by `@DissipatedEnergy` in `MFront` files) This output is optional.
- `external_state_variables`: pointer to an array containing the values of the external state variables. This is left unmodified by the behaviour integration.

4.1.3 Memory management

Most data member are thus pointers to a array of values (i.e. a continous memory block). For efficiency, the size of those arrays are not stored in the `StateView` structure, which means that the developer is responsible of allocating the memory properly.

If one uses the data structure `BehaviourData` or the `MaterialDataManager` data structure provided by `MGIS`, which are described below, this is automatically ensured.

4.2 First case: use of the `BehaviourData` data structure

Starting from a behaviour, one can allocate all the memory required to treat one integration point:

```
auto d = BehaviourData{b};
```

The data members of the `BehaviourData` structure are very close to the one of the `BehaviourDataView` class. In particular, it contains two data members called `s0` and `s1` containing the state of integration point.

> **Note**
>
> The `BehaviourData` structure holds a reference to the behaviour. The use must be sure that the behaviour outlives the behaviour data.

4.2.1 Creating a `BehaviourDataView` from a `BehaviourData`

The `make_view` function creates a `BehaviourDataView` from a `BehaviourData`:

```
auto v = make_view(d);
```

4.2.2 Retrieving the values of a specific variable

All the internal state variables are stored in a continuous array. If one wish to retrieve the value of a specific state variable, one first have to retrieve the offset of this variable as follows:

```
const auto o = getVariableOffset(b.isvs, "EquivalentViscoplasticStrain",
                                 b.hypothesis);
```

The same applies to material properties, external state variables, etc...

4.2.3 Updating the date from one time step to the other

The `update` function overwrites the `s0` member, i.e. the state at the beginning of the time step, with `s1`, i.e. the state at the end of the time step.

4.2.4 Restarting a time step

The `revert` function overwrites the `s1` member, i.e. the state at the end of the time step, with `s0`, i.e. the state at the beginning of the time step.

4.3 Second case: use of the `MaterialDataManager` structure

The `MaterialDataManager` data structure handle the data of a set of integration points (generally representing a material, hence the name), as follows:

```
MaterialDataManager m{b, n};
```

where `n` is the number of integration points.

> **Note**
>
> Note that to avoid copying, the move constructor, the copy constructor, the standard assignment and the move assignment of the `MaterialDataManager` have been disabled.

The `MaterialDataManager` structure has data members that have the same names and meanings than those of the `BehaviourData` structure, except that they holds the values of a set of integration points. In particular, the `MaterialDataManager` structure contains two data members called `s0` and `s1` containing the state of integration point.

4.3.1 External memory management and the `MaterialDataManagerInitializer` class

By default, the `MaterialDataManager` automatically allocates the values used to store all the data required.

However, one may want to use memory allocated by the solver. In this case, one can have a look at the `MaterialDataManagerInitializer` class.

4.3.2 Updating the date from one time step to the other

The `update` function overwrites the `s0` member, i.e. the state at the beginning of the time step, with `s1`, i.e. the state at the end of the time step.

4.3.3 Restarting a time step

The `revert` function overwrites the `s1` member, i.e. the state at the end of the time step, with `s0`, i.e. the state at the beginning of the time step.

4.4 Third case: manual allocation

The `MGIS` library provides a set of functions which allows the developer to properly allocate the memory associated with the data required at each integration point.

For example, if one wants to known the size of an array able to store all the internal state variables of one integration point, one may use the `getArraySize` function:

```
const auto n = getArraySize(b.isvs, b.hypothesis);
```

In the case of the plastic behaviour, there are two state variables: the elastic strain, which is a symmetric tensor and the equivalent plastic strain, which is a scalar. Thus, the value of `n` will be equal to `7`.

5 Behaviour integration

Three overloads of the `integrate` function are available:

The first one has the following interface:

```
int integrate(BehaviourDataView&, const Behaviour&);
```

The user may refer to the `IntegrateTest.cxx` file to have an example on how this use this version.

The second one takes a `MaterialDataManager` and performs the behaviour integration on a range of integration points. The user may refer to the `IntegrateTest2.cxx` file to have an example on how this use this second overload.

The third overload is similar to the second one but also takes an argument of the `ThreadPool` type allowing a parallelization of the behaviour integration. The user may refer to the `IntegrateTest2b.cxx` file to have an example on how this use this second overload.

References

Miehe, C., N. Apel, and M. Lambrecht. 2002. "Anisotropic Additive Plasticity in the Logarithmic Strain Space: Modular Kinematic Formulation and Implementation Based on Incremental Minimization Principles for Standard Materials." *Computer Methods in Applied Mechanics and Engineering* 191 (47–48): 5383–5425. https://doi.org/10.1016/S0045-7825(02)00438-3.