Permeability Multiplier Reservoir Simulator *PMRS* project

M. Sanei and O. Duran and P. Devloo

September 8, 2018

Abstract

This document contains details about the implementation and formulations for a novel upscaling approach to consider permeability variations in conventional reservoir simulations.

1 Introduction

2 Fundamentals

2.1 sec:fundamentals

3 Upscaling process for permeability considering geomechanic effects

3.1 sec:upscaling

4 Programming and neopz implementation

PMRS uses plastic materials already implemented inside *neopz* library. Souza Neto, Peri, and Owen 2008.

4.1 About the custom memory materials

This subsection is dedicated for the approach used when is required a template instantiation of a class. This is the case of the materials that use a memory for store some quantities related to the physics being studied. For instance the multiphysics simulation for a nonlinear poromechanics process that considered plasticity and flow through porous media.

In terms of template instantiation there are two categories:

- Implicit
- Explicit

The implicit instantiation consider the definition for all of types that can be instantiated, and the implementation is given defining inside the header file the

class's methods with the advantage of providing custom templates of different types as the user requires.

The explicit instantiation consider the definition for the a small set of concrete types that are ever instantiated. The implementation is given by moving out of from header file the class's methods with the advantage of better compile times.

Thus, for the implementation of the explicit instantiation a two-file approach is adopted as a traditional way. In C++ normally the use of **.h* files are reserved for the definitions and **.cpp* are reserved for the method implementations and the definitions of the instantiations at the end of the file. The two-file approach is well-know and straightforward to use, so is recommended when the small set of types to be instantiated is well know and predefined.

In the case of the iterative approximations performed by *PMRS* the two-file approach is not suitable, and it is required the implicit instantiation, mainly because the implementation of the plastic materials inside *neopz* is given by the two-file approach there is not a mechanism to instantiated another kind of memory types. To over come this difficulty a three-file approach is adopted and implemented inside *neopz* .

The three-file approach is given by following a pattern with third file with the suffix **_ impl.h*. The word *impl* here is an abbreviation for implementation. The files **impl.h* are reserved for the implementation of the class's methods and the **.cpp* is reserved just for the explicit instantiation for the well-known set of types. Thus the difficulty with the instantiation of custom memory materials is solved by including the **impl.h* instead **.h*. For the use of the standard types of templates the implementation should include just the **.h* maintaining the advantage of the explicit instantiation.

As a final remark the three-file approach is an alternative solution that incorporates the explicit and the implicit instantiation.

An example of the thee-file approach is give as follows:

Example of a **.h* the class *TPZMyMatElastoPlastic2D.h* :

```
#include "Tensor.h''
template <class T, class TMEM = TPZElastoPlasticMem>
class  TMatElastoPlastic2D : public TPZMatElastoPlastic<T,TMEM>
{
 public:
  T compute_sigma(Tensor epsilon) const;
};
```

Example of a **_ impl.h* :

```
#include "TMatElastoPlastic2D.h''
template <class T, class TMEM>
T TMatElastoPlastic2D<T,TMEM>::compute_sigma(Tensor epsilon){
  T sigma =  ConsitutiveLaw(epsilon);
  return sigma;
}
```

Example of a **.cpp* :

```
#include "TMatElastoPlastic2D_impl.h''
template class TMatElastoPlastic2D<LinearLaw, ElasticMem>;
template class TMatElastoPlastic2D<PlasticLaw, ElastoPlasticMem>;
```

For a stantandar use of *TMatElastoPlastic2D.h* class, the main should looks like this:

```
#include "TMatElastoPlastic2D.h"
#include "LinearLaw.h"
#include "ElasticMem.h"
int main() {
Tensor epsilon;
 TMatElastoPlastic2D<LinearLaw,ElasticMem> material;
 LinearLaw sigma material.compute_sigma(epsilon);
 std::cout << "sigma '' << sigma << std::endl;
}
```

For a non-stantandar use of *TMatElastoPlastic2D.h* class with another memory type *MultiphysicsMem*, the main should looks like this:

```
#include "TMatElastoPlastic2D_impl.h"
#include "LinearLaw.h"
#include "MultiphysicsMem.h"
int main() {
Tensor epsilon;
 TMatElastoPlastic2D<LinearLaw,MultiphysicsMem> material;
 LinearLaw sigma material.compute_sigma(epsilon);
 std::cout << "sigma '' << sigma << std::endl;
}
```

5 Verifications

5.1 2D simulations

5.2 3D simulations

6 Results

7 References

References

Souza Neto, E. A. de, D. Peri, and D. R. J. Owen (2008). *Computational Methods for Plasticity*. John Wiley & Sons, Ltd. DOI: 10.1002/9780470694626. URL: https://doi.org/10.1002/9780470694626.