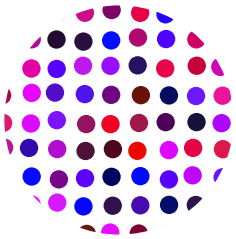

Muesli

a Material UnivErSal LIbrary

version 1.0

IMDEA Materials, Madrid, Spain

muesli.materials@imdea.org

June 2016

Contents

Contents **2**
1 Introduction . . . 2
2 Instaling MUESLI . . . 3
3 Main concepts . . . 4
4 Using MUESLI . . . 7
5 Using MUESLI with a commercial code . . . 13
6 Class hierarchy outline . . . 15
7 Automatic testing . . . 16
8 Extending MUESLI . . . 17
9 Using MUESLI in parallel . . . 17
10 Contributing to MUESLI . . . 18
11 Licence and Developer team . . . 18
A.1 Materials for thermal analysis . . . 19
A.2 Materials for small strain mechanical analysis . . . 19
A.3 Materials for finite strain mechanical analysis . . . 23
A.4 Fluid materials . . . 27
A.5 Materials for coupled thermomechanics at finite strains . . . 30

Bibliography **31**

1 Introduction

At the core of many simulation codes of solids, fluids, and structures lie functions that encapsulate the continuum behavior of all kind of materials. These functions provide, essentially, the same functionality in all codes: elastic, plastic and viscoplastic routines for computational solid mechanics; newtonian and non-newtonian fluids in CFD codes; thermal behavior, diffusion models, etc. Despite the ubiquitousness of such material models, it does not exist, to our knowledge, a single open-source library that provides, at least, the most common material models. Such an omission forces developers of new codes in computational solid and fluid mechanics to start from scratch and implement models that are completely standard. By doing so, lots of time and effort are spent in writing code and debugging it, code which has already been implemented in dozens of other codes but not shared, nor designed to be shared.

A related problem often faced by scientist working in computational mechanics is that material models are developed to be inserted in one existing commercial code. In these cases, the routines can not be used except when linked with this particular code, which in addition poses severe restrictions regarding the interface. For example, Abaqus [1] offers the possibility to enhance the program's material library by adding material routines, or `UMAT`s. Due to

the specific interface specification of these files, none of the `UMAT`s can be used, at least in a straightforward manner, with other commercial or research codes.

Finally, a third limitation for material developers is due to the fact that commercial codes often require that their extensions be written in FORTRAN 77. While this might be an advantage for certain programmers, it is our experience that using such a programming language precludes the use of tools which simplify the coding, for example operator overloading. More importantly, the use of C++ with tensor objects make the code closely resemble the mathematical expressions used to define constitutive models. For complex materials, e.g. finite strain elastoplasticity, access to a powerful and simple programming syntax can make development, debugging, and maintenance much simpler than with traditional programming languages.

MUESLI is a Material Universal LIbrary, a collection of C++ classes and interfaces created with the purpose of simplifying the development and implementation of material models at the continuum scale, and their interface with larger research and commercial computational codes. The library addresses the limitations identified above by:

- Providing well-tested implementations of several standard material models for small and large strain solid mechanics, fluid mechanics, and other.
- Including an interface of the MUESLI material models with commercial codes `LS-DYNA` and `Abaqus` (possibly more in the future).
- Using high-level C++ classes that simplify the tensor algebra involved, especially, in complex material models.

Finally, let us clarify that MUESLI is not designed to solve boundary value problems, neither to obtain strain-stress curves. Either of these goals, like many other, require an external driving program which delegates on MUESLI the tasks regarding the evaluation of the material response.

2 Instaling MUESLI

MUESLI is distributed in a singe zip file named `muesli_xx.zip`, where `xx` denotes the version of the library. The zip file includes the source files, the documentation, and the `makefile`s to build the library in linux and mac os x computers.

could After downloading this file, one needs to uncompress it and possibly edit the `makefile` to tweak the building process if required. The makefile is very simple and no special libraries need to be linked so often it suffices to type `make` in a terminal to build the library, which is called `libmuesli.a`. Installing the library in the standard path of user libraries is accomplished by the command `make install` or `sudo make install`, if root permissions are required. In addition, the command `make test` builds an executable file,

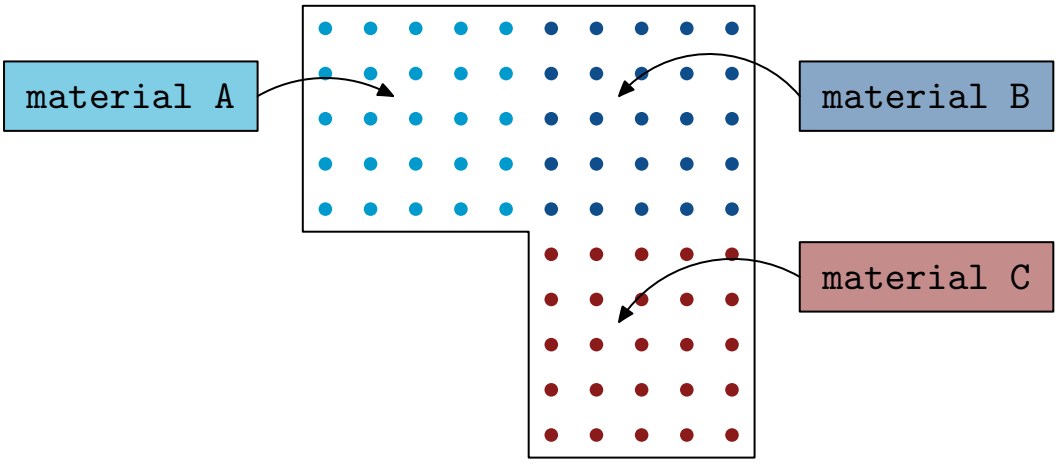

Figure 1: In MUESLI, a `material` is a factory class whose objects are responsible for creating `materialPoints`. In the body of the figure, three materials are responsible for creating, each of them, 25 material points. At creation, all the material points created by the same material are identical.

`testmueli`, that runs all the tests on the material models and writes a log file, `testmuesli.log`, with the output of the tests.

As indicated before, MUESLI is a self-contained library and needs no additional packages. The library makes heavy use of matrix and vector operations, all of which are defined in the files `mtensor.h` and `mtensor.cpp`. Users that rather employ the well-tested library `eigen` [2], can do so by modifying the makefile as indicated therein. Obviously, in this case the `eigen` library must be already installed in the computer.

MUESLI is a thread-safe library and it can be safely used in shared and distributed memory computers. When all the update operations are done in a distributed way, and all of these are completed before a new update, the library can gain some speed by using a compiler library, as explained in the `makefile`. By default, however, all the mechanisms are in-place to avoid any kind of race conditions.

3 Main concepts

MUESLI is designed for the modeling of a wide range of materials at the continuum level, and with diverse applications in mind. While the mechanical behavior is the most developed one, thermal response is also implemented in the current version, and others will be added in the future.

At the foundation of every class in MUESLI lie two abstract concepts namely, the `material` and `materialPoint` parent classes. The first one embodies the

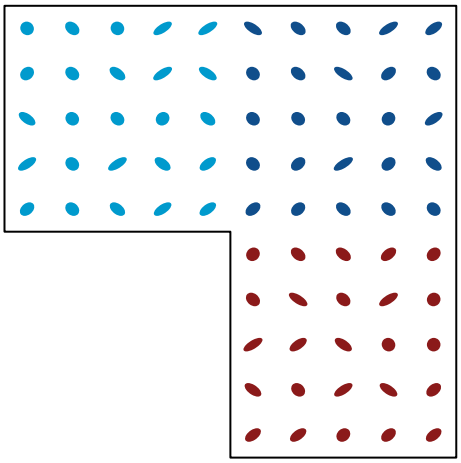

Figure 2: In MUESLI, the `materialPoint`s change in time as their state is modified by their history. Even if `materialPoint`s of the same type belong to the same class, their behaviors might be different with time due to the changed state.

concept of *abstract material*, as entity which can spawn `materialPoints` and hold common data for all the children. The second one is the one that represents individual points in real continuum bodies: each of them having a specific material constitutive behavior and also a current and past state consisting of kinematic and/or thermodynamic variables. The two parent classes are, using object oriented terminology, pure virtual classes. This means that they provide the basic interface requirements for every single `material` and `materialPoint` in MUESLI, but themselves can not be used to construct objects.

Figure 1 illustrates the concepts of `material` and `materialPoint`. When creating a computational model of a continuum, the analyst must decide which materials it is going to be made of. Then, by assigning a subdomain of the analysis model a `material`, it implicitly entrusts the latter with the task of creating `materialPoint`s located on quadrature points, particles, etc., within this subdomain, as required by the specific numerical method. In Figure 1, three different `material`s, create `materialPoint`s of three different kinds. At their inception, all the `materialPoint`s spawned by the same `material` are identical.

As a result of the mechanical, thermal, chemical, etc., evolution in the body, each of the `materialPoint`s in a model changes its state. As schematically illustrated by Figure 2, `materialPoint`s of the same type can become different and behave differently, even though their constitutive constants remain the same, and actually stored in the parent `material`.

This distinction between `material` and `materialPoint` is central to MUESLI. In a typical simulation, only a handful of `material`s exist, while thousands of `materialPoint`s are often allocated. For each of the specific classes of

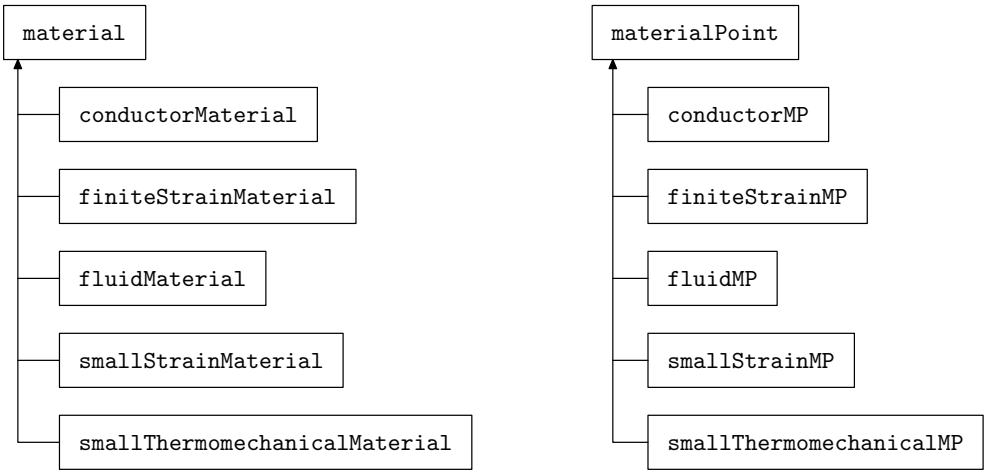

Figure 3: Main material families as sub-classes of the parent class `material`, and the corresponding material points sub-classes of the parent class `materialPoint`.

materials defined in the library, a corresponding class of material point is always defined. See section 6 for more details, as well as the appendices.

From the previous argument it follows that the concept of *state* plays an important role in MUESLI. Specifically, the state of a `materialPoint` is the data that the model requires to yield all possible information that might be requested from it. It follows then that state depends of the constitutive equations. In thermal points, for instance, the current value of the temperature gradient needs to be stored; in finite strain material points, on the other hand, at least the deformation gradient should be included in the state. For other inelastic models, such as viscocplastic materials, more *internal variables* need to be stored at the current instant, and possibly in past evaluation points.

The underlying structure of MUESLI is designed to optimize the performance of iterative methods for nonlinear problems. Such solvers sometimes drift away from the solution path and it is crucial that the model, and thus the material points, have the ability to recall their past equilibrated states and to return to them. In fact, the data structures of MUESLI are so flexible that many previous equilibrated states could be stored in every point, if desired.

The design of MUESLI is such that it allows to work with materials for very different boundary value problems (and also to implement new ones). For this reason, the classes `material` and `materialPoint` are almost empty and are designed only to provide a common inheritance for the library, and some trivial convenience functions.

As illustrated in Figure 3, each of the two parent classes has (currently) five sub-classes which are referred to as *families*. All of them refer to different mechanical problems and although semantically related, have completely dif-

ferent interfaces and behavior. For example, one might need to use copper as a material in an Engineering analysis. However, a `conductorMaterial` and a `smallStrainMaterial` with the properties of copper are completely different entities. The first one only know how heat flows as a response to temperature gradients, while the second is ignorant about this process, while it should be able to calculate the stress tensor for a given strain, among other tasks. As detailed later in Section 6 and in the Appendices, the interfaces for the material families are very different, and developers trying to add new materials to MUESLI should concentrate on the process their analyses require.

4 Using MUESLI

MUESLI is designed to work in two ways, either as a library that can be linked to one's own simulation code or extending a commercial code material simulation capabilities. In both situations, the MUESLI library needs to be built and linked to the main program. However, the specific detail of how MUESLI is used differ in the two cases.

First we indicate how MUESLI could be used when we have access to the full source code of a simulation package written in C++ and we want to include the material modeling capabilities of the library. To access these features, we must link the library `libmuesli.a` to the code and include the header file `muesli.h` everywhere we would like to use MUESLI's functions.

We describe next the typical usage of the library. First, a `material` object must be instantiated. A `material` is an object that encapsulates the data of all the material points with the same constitutive response. According to this idea, if a simulation of a steel structure is considered, and all the solids in it are to be modeled with the same elastoplastic behavior, a single material of the type `splastic` would need to the created. In this case, for example, the code to create the material would be as follows:

```
#include "muesli.h"
using namespace muesli;

// first a material is created with all the nominal data
double E = 210e9, nu = 0.3, Y = 200.0e6, hiso = 100.0, hkine = 50.0;
smallstrainMaterial* theMaterial = new splastic("steel", E, nu, Y, hiso, hkine, "von_mises
```

Once the `material` object exists, it can, among other things create material points of its class. Following the previous example, once the `splastic` material is allocated, it can be use to allocate many `splasticMP`, that is, material points whose behavior is that of an elastoplastic material. This would be accomplished with code like:

```
// the material spawns as many material points as needed
smallstrainMP* theMaterialPoint1 = theMaterial.createPoint();
```

```
smallstrainMP* theMaterialPoint2 = theMaterial.createPoint();
smallstrainMP* theMaterialPoint3 = theMaterial.createPoint();
```

In MUESLI, the difference between a "material" and a "material point" is that the first, once it is created, it never changes; the second one, however, changes its state depending on its history. In the previous example, for instance, each of the `materialMP` can be different, depending on the strain history of each of the points, which itself depends on the local deformation and loading process.

In every material class there are two ways of passing the information the describes the properties of the material it represents. The first, and most straightforward, uses a string — an arbitrary name that will be used when printing the material information — and a sequence of constants with predetermined meaning. In the example above, this sequence must be Young's modulus, Poisson's ratio, yield stress, isotropic hardening, kinematic hardening, and a string with the type of yield function. For the user's convenience, MUESLI defines the class `materialProperties`, a simple dictionary class that can hold material parameters in a more flexible fashion. Objects in this class collect data and their value (as double precision real numbers or strings). For example, the material object in the example above could have been constructed in the alternative way:

```
materialProperties mp;

mp["young"] = 210e9;
mp["poisson"] = 0.3;
mp["isotropich"] = 100.0;
mp["kinematich"] = 50.0;
mp["yieldstress"] = 200.0e6;
mp["model von_mises"] = 0.0

smallstrainMaterial* theMaterial = new splastic("steel", mp);
```

The keywords for the dictionary, which must be lowercase, are material dependent and can be found in the appendices of this manual. Note, that in the case of the plasticity model, its value, which is a string is added to the property name after a space.

Once the material points exist, their state must be set according to the local values of kinematic and thermodynamic variables. In the case of a `smallstrainMP`, a point that has elastic or inelastic small strain behavior, the state depends only of the history of the local value of the infinitesimal strain tensor. The current state of the points is thus set using code such as:

```
// set the state to the material point
istensor epsilon(0.2, -0.1, 0.0, 0.0, 0.0, 0.0);
double theTime = 0.1;
theMaterialPoint1.updateCurrentState(theTime, epsilon);
```

```
epsilon = istensor(0.1, 0.7, 0.0, 0.0, 0.0, 0.0);
theMaterialPoint2.updateCurrentState(theTime, epsilon);

epsilon = istensor(0.0, -0.5, 0.0, 0.0, 0.3, 0.0);
theMaterialPoint3.updateCurrentState(theTime, epsilon);
```

Once the state of the material points is set, the user can request all kind of information from them. For example, in a basic computation the user might be interested in knowing the stored energy density, the stress, and the tensor of elasticities at the points. The code responsible for these requests is:

```
// these are the output quantities
double energy;
istensor sigma;
double tg[3][3][3][3];

// and then request desired data
energy = theMaterialPoint1.storedEnergy();
theMaterialPoint1.stress(sigma);
theMaterialPoint1.tangent(tg);

energy = theMaterialPoint2.storedEnergy();
theMaterialPoint2.stress(sigma);
theMaterialPoint2.tangent(tg);

energy = theMaterialPoint3.storedEnergy();
theMaterialPoint3.stress(sigma);
theMaterialPoint3.tangent(tg);
```

In a typical computation, the state of each point changes while iterating for an equilibrium solution (for example in a Newton-Raphson scheme). In that case, the state needs to be set again, and the energy, stress and tangent, obtained again:

```
// set again state to the material point
epsilon = istensor(0.23, -0.13, 0.0, 0.0, 0.0, 0.0);
theMaterialPoint1.updateCurrentState(theTime, epsilon);

epsilon = istensor(-0.6, 0.0, 1.1, 0.0, 0.0, 0.0);
theMaterialPoint2.updateCurrentState(theTime, epsilon);

epsilon = istensor(1.1, 0.0, 3.1, 4.1, 5.9, 0.0);
theMaterialPoint3.updateCurrentState(theTime, epsilon);

// again request material data
energy = theMaterialPoint1.storedEnergy();
theMaterialPoint1.stress(sigma);
```

```
theMaterialPoint1.tangent(tg);

energy = theMaterialPoint2.storedEnergy();
theMaterialPoint2.stress(sigma);
theMaterialPoint2.tangent(tg);

energy = theMaterialPoint3.storedEnergy();
theMaterialPoint3.stress(sigma);
theMaterialPoint3.tangent(tg);
```

Again, in a typical nonlinear solution, once equilibrium is reached, the state of each of the points needs to be stored for future reference. In MUESLI, each material point calls the function `commitCurrentState` that signals that the last state set in the point corresponds to an equilibrated one.

```
// when the current state is equilibrated (to be determined elsewhere)
// commit the current state and proceed
theMaterialPoint1.commitCurrentState();

// set the state
theTime = 0.2;
epsilon(0,0) = 0.26;
epsilon(1,1) += 0.01;
theMaterialPoint1.updateCurrentState(theTime, epsilon);

// request desired data
theMaterialPoint1.stress(sigma);
energy = theMaterialPoint1.storedEnergy();
theMaterialPoint1.tangent(tg);

// set the state
epsilon(2,2) = 0.22;
epsilon(1,2) = epsilon(2,1) = 0.01;
theMaterialPoint2.updateCurrentState(theTime, epsilon);

// request the data
theMaterialPoint1.stress(sigma);
energy = theMaterialPoint2.storedEnergy();
theMaterialPoint1.tangent(tg);

theMaterialPoint1.commitCurrentState();
```

While using MUESLI in a nonlinear solution, it might happen that the iterative scheme is unable to converge and needs to be restarted, possibly using a smaller time step. In these cases, since the state of the material points is useless, one might recover the last converged step simply by requesting a reset as in

```
// recover last converged states
theMaterialPoint1.resetCurrentState();
```

```
theMaterialPoint2.resetCurrentState();
theMaterialPoint3.resetCurrentState();
```

Finally, when the computations are over, each material point, and the material itself must be deallocated:

```
// clean up at the very end
delete theMaterialPoint3;
delete theMaterialPoint2;
delete theMaterialPoint1;
delete theMaterial;
```

Needless to say, the actual use of MUESLI in a large scale computational code involves the allocation/deallocation of thousands or millions of material points which is routinely done looping through finite elements, or material domains. The previous example just illustrates the general usage of the library.

We illustrate the use of MUESLI with a second example, one that is distributed with the source code of the library. The source coude is as follows:

```
std::ofstream os("cyclictest.data");

const double E = 210e9; // Young's modulus
const double nu = 0.33;   // Poisson's ratio
const double rho = 1.0;   // density
const double Hiso = 1e9;  // isotropic hardening modulus
const double Hkin = 2e9;  // kinematic hardening modulus
const double Y0 = 100e6;  // yield stress
const double alpha = 0.0; // not required
const std::string name = "mises";

muesli::splasticMaterial mat(E, nu, rho, Hiso, Hkin, Y0, alpha, name);
muesli::smallStrainMP* p = mat.createMaterialPoint();

istensor strain;
istensor shear;
shear(0,1) = shear(1,0) = 0.02;

unsigned npercycle = 20;
unsigned ncycles   = 5;
for (unsigned i=0; i<ncycles*npercycle; i++)
  {
    double t = sin(((double)i)/npercycle*2.0*M_PI);
    strain = shear*t;
    p->updateCurrentState(i, strain);

    istensor sigma; p->stress(sigma);
    double   tau = sigma(1,0);
    double   g   = p->plasticSlip();
    p->commitCurrentState();
```

```
      os << "\n" << std::setw(8) << std::scientific
         << t << " " << strain(0,1) << " " << tau << " " << g;
   }

os.close();
delete p;

return 0;
```

The example creates a small strain elastoplastic material, `mat`, which then spawns a point `p`. This point is subjected to a cyclic shear deformation. After each loading step, the strain component $\varepsilon(1,2)$, the shear stress $\sigma(1,2)$ and the plastic slip are dumped to file for postprocessing.

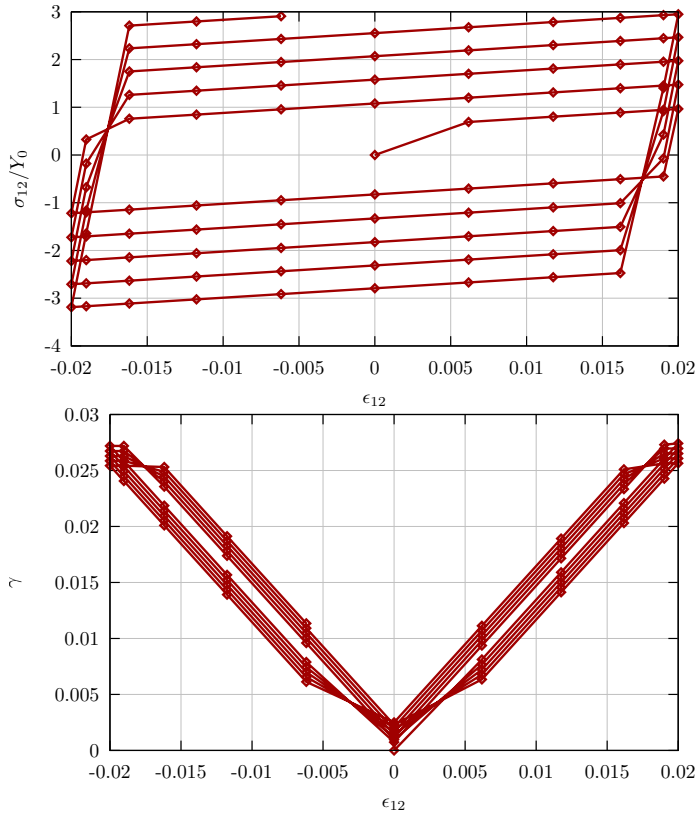

Figure 4: Cyclic test of an elastoplastic point. Top: the shear strain-stress plot; bottom: the plastic slip γ as a function of the shear strain.

5 Using MUESLI with a commercial code

MUESLI has been designed so that all its material models can be, in principle, accessed from existing commercial codes, just by writing the appropriate interface. The advantage of this approach is that one can write the constitutive model functions using high level C++, and use them for both research and commercial codes. While more interfaces could be devised in the future, in the current version only the ones for LS-DYNA and Abaqus are provided.

Interfacing MUESLI and LS-DYNA

LS-DYNA is a simulation code of Livermore Software Technology Corporation (`http://www.lstc.com`). It has many simulation capabilities for quasi-static and transient simulations. In particular, it has a large material library for metals, polymers, soils, etc.

Users are allowed to add new material models to LS-DYNA, and we take advantage of this feature to interface this code with MUESLI. To do so, one must obtain LS-DYNA's package for user defined materials (downloadable from the LSTC web page). In this package, one must edit the file `dyna21.f` and modify material 43 (or any other one) as follows:

```
subroutine umat43(cm,eps,sig,epsp,hsv,dt1,capa,etype,tt,
     1 temper,failel,crv,cma,qmat,elsiz,idele)

      include 'nlqparm'
      include 'bk06.inc'
      include 'iounits.inc'
      dimension cm(*),eps(*),sig(*),hsv(*),crv(lq1,2,*),cma(*),qmat(3,3)
      logical failel
      character*5 etype

      if (ncycle.eq.1) then
        if (cm(16).ne.1234567) then
          call usermsg('mat43')
        endif
      endif

      if (etype.eq.'solid'.or.etype.eq.'shl_t'.or.
     1     etype.eq.'sld2d'.or.etype.eq.'tshel'.or.
     2     etype.eq.'tet13') then
        if (cm(16).eq.1234567) then
          call mitfail3d(cm,eps,sig,epsp,hsv,dt1,capa,failel,tt,crv)
        else
           if (.not.failel) then
c            This is the call to muesli
             call interface_lsdyna(cm,eps,sig,hsv,tt,epsp,temper,failel)
          endif
        end if
```

```
      else
        cerdat(1)=etype
        call lsmsg(3,MSG_SOL+1150,ioall,ierdat,rerdat,cerdat,0)
      endif
      return
      end
```

The LS-DYNA `Makefile` needs to be modified to include MUESLI. For that, the library `libmuesli.a` should be added to the `OBJECTS` definition. If required, the paths to this library should be added. After this, a new LS-DYNA binary can be built, a binary that will now include all the capabilities of MUESLI.

To use MUESLI's material within a computation, we must indicate that the material type (`mt`) is 43 (or another one, see above). In the data provided for this material we must indicate which of MUESLI's material is to be selected, as well as any additional material parameters. For example, the following command line in an LS-DYNA input file uses `XXX`

```
*MAT_USER_DEFINED_MATERIAL_MODELS_TITLE
muesli
$This material is an test for muesli
$#     mid        ro        mt       lmc       nhv    iortho     ibulk        ig
         3 1000.0000        43         7         0         0         3         4
$#   ivect     ifail    itherm    ihyper      ieos      lmca    unused    unused
         0         0         0         1         0         0
$#      p1        p2        p3        p4        p5        p6        p7        p8
2.1000E+11  0.3300002.0588E+117.8940E+10  4.000000    0.0000     0.000     0.000
```

In MUESLI's distribution, the directory `examples` contains several input files for LS-DYNA that use MUESLI's materials for linear and nonlinear, elastic and inelastic materials.

Interfacing MUESLI and Abaqus/standard

Abaqus is a general-purpose finite element code developed by Abaqus Inc., acquired by Dassault Systèmes. Abaqus/standard is the part of Abaqus responsible for implicit analyses, both quasistatic and transient, for linear and nonlinear solids. In addition to the traditional elements of stress analysis, the code now includes a wide range of additional features such as contact, multibody components, coupled thermal effects, etc.

Abaqus/standard allows developers to add material models to the code through user material subroutines (UMAT) that can be linked with the main program. User subroutine UMATs are traditionally written in Fortran and an interface needs to be developed to connect MUESLI and Abaqus. After building `libmuesli.a`, it needs to be linked with Abaqus by modifying the Abaqus environment file `abaqus_v6.env`, where we must add the path and library name as follows:

```
link_sl = (fortCmd +
   " -cxxlib -fPIC -threads -shared " +
   "%E -Wl,-soname,%U -o %U %F %A %L %B -parallel -Wl,-Bdynamic " +
   "-i-dynamic -lifport -lifcoremt  -L/usr/local/lib/ -lmuesli ")
```

Second, we use an UMAT to provide the interface between Abaqus and MUESLI. In this Fortran file, a single line calls MUESLI, transferring all the relevant variables to the library.

```
      SUBROUTINE UMAT(STRESS,STATEV,DDSDDE,SSE,SPD,SCD,
     1           RPL,DDSDDT,DRPLDE,DRPLDT,
     2           STRAN,DSTRAN,TIME,DTIME,TEMP,DTEMP,PREDEF,DPRED,CMNAME,
     3           NDI,NSHR,NTENS,NSTATV,PROPS,NPROPS,COORDS,DROT,PNEWDT,
     4           CELENT,DFGRD0,DFGRD1,NOEL,NPT,LAYER,KSPT,KSTEP,KINC)
c

       INCLUDE 'ABA_PARAM.INC'

      DIMENSiON STRESS(NTENS),STATEV(NSTATV),
     1     DDSDDE(NTENS,NTENS),DDSDDT(NTENS),DRPLDE(NTENS),
     2     STRAN(NTENS),DSTRAN(NTENS),TIME(2),PREDEF(1),DPRED(1),
     3     PROPS(NPROPS),COORDS(3),DROT(3,3),DFGRD0(3,3),DFGRD1(3,3)

      character*80 CMNAME
      integer matlabel

      matlabel = 1
      call interface_abaqus(CMNAME,matlabel,NOEL,NPT,COORDS,PROPS,NPROPS,
     1                      CELENT,TIME,DTIME,PNEWDT,STRAN,DSTRAN,DFGRD0,
     1                      DFGRD1,TEMP,DTEMP,STRESS,NTENS,DDSDDE,STATEV,
     1                      NSTATEV,SSE,SPD,KSTEP,KINC)

      END
```

The interface file `interface_abaqus.cpp` in MUESLI is responsible for receiving the data and variables from the UMAT, converting it to its internal format and calling the appropriate C++ methods of the library.

6 Class hierarchy outline

At the second level, there are specific `material` and `materialPoints` for each of the boundary value problems that can be addressed with MUESLI. Currently, there are

- `smallStrainMaterial`
- `finiteStrainMaterial`

- `fluidMaterial`
- `conductorMaterial`
- `smallThermomechanicalMaterial`

and their corresponding `materialPoints`. See figure 3. All of these are again *pure virtual* classes, and define the interfaces of their child classes. The first class, `smallStrainMaterial`, is designed to provide the interface to all mechanical problems with small strain kinematics; the second, `finiteStrainMaterial`, defines the interface to mechanical problems with finite strain kinematics; `fluidMaterial` is the abstract class for fluid response; `conductorMaterial` defines the interface for heat problems; `smallThermomechanicalMaterial` is the interface for strongly coupled thermomechanical problems.

Each of these base classes has an interface that is different to that of the others. The reason is that, despite all of them being abstractions of material behavior, their intended use is fairly different. A finite element code for small strain elastic and inelastic response only needs to access materials of the class `smallStrainMaterial`. The interface declares, therefore, the minimum set of functions that each specific material class must explicitly implement. This set is completely different to the set of functions required to define the thermal behavior in a heat transfer code, which itself should only access materials of type `conductorMaterial`.

Depending on the field of application, a given simulation code often makes use only of one or two types of materials. However, within this class, all specific materials share the same interface. Going back to the example of a finite element code for small strain analyses, the materials that the code will access are all sub-classes of `smallStrainMaterial`, and they must all implement the same update interface, as well as provide energies, stress, and tangent computations. More details of the specifics will be provided later in the document.

7 Automatic testing

Materials in MUESLI have functions that test the correctness of their formulation, up to a certain extent. A testing binary `testmuesli` can be built as indicated in Section 2. This program executes many tests in each material of the library. Developers of new materials are encouraged to make as much use as possible of this feature, for it provides a certain guarantee of the soundness of the implementation. Of course,

The actual tests run for each material class depends on the family to which it belongs. In general terms, the tests are designed to verify the consistency of the programmed energies, gradients, and hessians. For example, for a small strain elastic material, i.e., one belonging to the class `elasticMaterial`, there

must be a functions returning, for a given state, the stored energy density, the stress, and the tensor of elasticities. In this case, the test will verify that the stress is precisely the derivative of the energy with respect to the strain and that the tensor of elasticities is, in turn, the derivative of the stress with respect to the strain.

8 Extending MUESLI

MUESLI can be extended in three different ways:

- New materials can be added to any of the existing families. To do so it suffices to derive a `material` and a `materialPoint` from the base class of the family, and provide an implementation for the pure virtual functions.
- Whole new families might be added. This can be done when the behavior or the physics of the materials that are to be added do not fit within any of the existing categories. Designing a new family is fairly more involved than adding a single material because the interface has the be decided. Existing classes for parent materials can serve as guidelines.
- New interfaces can be programmed for existing research or commercial codes. This requires the study of the code's requirements, and adapting the latter to the functionality of MUESLI. In this case, new functions should not be added to MUESLI, because this would invalidate the material's usefulness for other pre-existing interfaces.

9 Using MUESLI in parallel

MUESLI has been designed to work in parallel environments. Most of the times a single processor will create the materials and materials points. Then, the more costly updates and computations will be done in parallel, distributing the material points among all available cores or machines. For instance, to perform an update of the states in all of the material points, the driving program will split the points, loop through all the points in each processing unit, and wait until all loops have finished before proceeding, say, to compute stresses or tangents.

If used in this way, MUESLI is thread safe. If for some reason there is no guarantee that the material points are accessed independently, and that one point might be called simultaneously from two processes requesting the update and, for example, the stresses, a compiler option must be set to enforce the thread-safety of the library. This option is activated simply by defining the keyword `STRICT_THREAD_SAFE` in every compilation. The `makefile` included in MUESLI's distribution explains how this should be done for the Gnu compilers.

10 Contributing to MUESLI

One of the motivations for developing MUESLI has been enabling the possibility of sharing code for material modeling. The open distribution of the library contains several material models and interfaces, but researchers are encouraged to contribute to the project by sending their classes, comments, and suggestions to

`muesli.materials@imdea.org`

When a new material is submitted to be included in MUESLI, the author(s) must make sure it passes all the tests, it there are any in the corresponding material family. In addition, the source code should include the basic documentation that enables users to understand where the model comes from (i.e., references), original developers, mathematical expressions, etc. Once it is tested, the new files will be added to the library, with the corresponding credits, noting that it will be subject to the same licence as the original source code.

11 Licence and Developer team

IMDEA Materials is the owner of MUESLI, and the library is distributed under GPL3.0 licence. Details of this licence are included in the software distribution, in the file `licence.txt`. Additional information can be found in the web site `http://www.gnu.org/licenses`.

The developers of MUESLI are: David Portillo, Daniel del Pozo, Daniel Rodríguez, Javier Segurado, and Ignacio Romero.

A.1 Materials for thermal analysis

The class `conductorMaterial` defines the constitutive behaviour of material subjected to heat flux. At the moment, the only thermal behavior considered is given by an isotropic Fourier's law in which the heat flux is proportional to the temperature gradient, and the proportionality is defined by a scalar.

A.2 Materials for small strain mechanical analysis

The class `smallStrainMaterial` is defined for analyses of classical elasticity and other small strain problems such as elastoplasticity, viscoelasticity, and visco-elasto-plasticity. The classes currently implemented are `elasticMaterial`, `plasticMaterial`, `viscoelasticMaterial`, and `viscoplasticMaterial`. The base class, `smallStrainMaterial` is a pure virtual class (in `smallstrain.cpp`) so only objects of the derived classes can be instantiated. The class (see `smallstrain.h`) has the following definition:

```
class smallStrainMaterial : public muesli::material
{
   public:
                            smallStrainMaterial();
                            smallStrainMaterial(const std::string& name,
                                       const materialProperties& cl);

   virtual                  ~smallStrainMaterial(){}

   virtual bool             check() const = 0;
   virtual smallStrainMP*   createMaterialPoint() const = 0;
   virtual double           density() const = 0;
   virtual double           getProperty(const propertyName p) const = 0;
   virtual void             print(std::ostream &of=std::cout) const = 0;
   virtual void             setRandom() = 0;
   virtual bool             test(std::ostream  &of=std::cout) = 0;
   virtual double           waveVelocity() const = 0;
};
```

The first method in a `smallStrainMaterial` is `check()`, which servers to verify if the material parameters are physically viable or consistent.

The main task of a any subclass of `smallStrainMaterial` is to spawn material points of their corresponding type, and is the responsibility of the method `createMaterialPoint()`. As set, for example, in the `elasticMaterial` class, this function creates new material points, allocating their required memory and initializing their state. All the `materialPoint`s created by a `material` share the same factory, and therefore the material parameters.

Small strain materials are subject to a series of tests that verify the consistency of the implementation. These tests, described below, are driven by

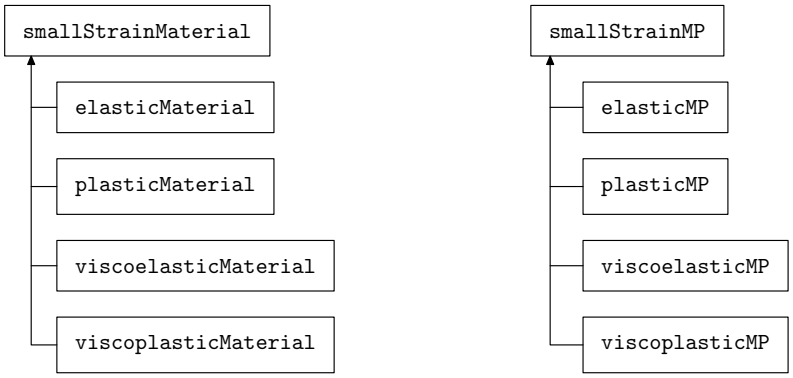

Figure 5: Currently existing classes for small strain analyses.

the method `test()`. After randomly setting all the material parameters with the function `setRandom()`, the `test()` function must create a `materialPoint` in a random state and subject it to all the tests. See the structure of the test function in, e.g., `elasticMaterial`.

The other virtual functions of the class are fairly straightforward and their precise definition is easily understood by looking at their implementation in existing `smallStrainMaterial`s.

Each material point of a class derived from `smallStrainMP` must provide the mechanical behavior its material constitutive law. This includes computing the energy, stresses and tangents for each possible state, and some auxiliary data. The actual functions that have to be programmed are:

```
virtual void            setRandom() = 0;

// energies
virtual double          deviatoricEnergy() const = 0;
virtual double          dissipatedEnergy() const = 0;
virtual double          storedEnergy() const = 0;
virtual double          volumetricEnergy() const = 0;


// stress
virtual void            stress(istensor& sigma) const = 0;
virtual double          plasticSlip() const = 0;


// tangents
virtual void            contractWithDeviatoricTangent(const ivector &v1,
                                                      const ivector &v2,
                                                      itensor &T) const = 0;
virtual void            tangentTensor(double C[3][3][3][3]) const = 0;
virtual double          volumetricStiffness() const = 0;
```

```
// uniaxial response
virtual double          uniaxialStiffness() const = 0;
virtual double          uniaxialStress() const = 0;


// bookkeeping
virtual void            updateCurrentState(const double t,
                                           const double strain11,
                                           const double temp=0.0) = 0;
virtual void            updateCurrentState(const double t,
                                           const istensor& strain,
                                           const double temp=0.0) = 0;
virtual void            commitCurrentState() = 0;
virtual void            resetCurrentState() = 0;
```

Once a `smallStrainMP` is created, its state must be set using one of the `updateCurrentState()` functions; when desired, this state is stored with the `commitCurrentState()` function. Also, the last stored solution can be recovered with the `resetCurrentState()` method. See section 3 for an explanation of the concepts associated with these functions.

After setting its state, a `smallStrainMP` can be queried for quantities of mechanical interest, including energy, stresses, tangents, and internal variables.

The stored energy density is obtained with the function `storedEnergy()`. If the material has a deviatoric/volumetric decoupled stored energy, as in most material, each of these contributions can be calculated with the functions `deviatoricEnergy()` and `volumetricEnergy()`, respectively. The dissipation in the last time step (starting at the last committed state and ending in the current state) is obtained with the function `dissipatedEnergy()`.

The function `stress()` computes the stress tensor at the current state. The construction of the fourth order tensor of tangent elasticities is entrusted to the function `tangentTensor()`.

As in the three dimensional case, one-dimensional versions of the energy/stress/stiffness functions need to be provided.

In addition to the former, mandatory functions, `smallStrainMP` classes might implement additional ones which might become useful in certain calculations. These are

```
virtual double          pressure() const;
virtual void            stressVector(double S[6]) const;
virtual void            deviatoricStress(istensor& s) const;
virtual double          plasticSlip() const = 0;

// tangents
virtual void            contractWithTangent(const ivector &v1,
                                            const ivector &v2,
                                            itensor &T) const;
```

```
virtual void            contractWithDeviatoricTangent(const ivector &v1,
                                                      const ivector &v2,
                                                      itensor &T) const;
virtual void            tangentMatrix(double C[6][6]) const;
virtual double          volumetricStiffness() const;
```

These are not *pure virtual* functions, so the parent class `smallStrainMP` provides implementations for all of them which are based on the mandatory functions. These surrogate implementations might be very slow for specific material models so developers are encourage to implement the ones that are more CPU demanding or more frequently employed.

All these ancillary functions perform tasks that are fairly evident from their names and their arguments, except for the two methods `contractWithTangent()` and `contractWithDeviatoricTangent()`. As explained in [3], when solving linearized equations of equilibrium (whether in a finite element method or in other scheme), the tangent of elasticities $\mathbb{C}$ is an expensive object to compute which is, almost invariably, never used by itself but always pre- and post-multiplied by a vector in operations of the form

$$T_{ik} = \sum_{j,l} \mathbb{C}_{ijkl}\, a_j\, b_l \ . \tag{1}$$

Likewise, in mixed methods, the restriction of the tensor of elasticities to the space of deviatoric tensors $\mathbb{C}_{dev}$, is almost never employed by itself but rather pre- and post-multiplied by vectors as in Eq. 1. The two methods indicated before, namely, `contractWithTangent()` and `contractWithDeviatoricTangent()`, precisely provide these two operations. Although they can always be implemented by calling the `tangentTensor()` function, and then performing the contraction (1), it is far more efficient to program these function for each material developed.

Elastic isotropic materials

The simplest small strain material is the elastic isotropic material. In MUESLI these materials are implemented in the `elasticIsotropicMaterial` and `elasticIsotropicMP` classes. This file should be consulted as the simplest complete implementation provided in MUESLI.

To instantiate an elastic isotropic `material`, MUESLI provides two constructors. The first one creates an `elasticMaterial` from a name and three constants, namely, Young's modulus, Poisson's ratio, and the density, as follows:

```
elasticIsotropicMaterial(const std::string& name,
                         const double E, const double nu,
                         const double rho);
```

Alternatively, a constructor can be employed with a name for the material and a `materialProperties` dictionary. In this case, the name and meaning

Argument name	Explanation
`young`	Young's modulus
`poisson`	Poisson's ratio
`density`	Density

Table 1: Parameters for the constructor of an `elasticMaterial`.

of the labels that can be employed are indicated in Table ??. When choosing this constructor, the elastic constants can be defined either with the pair (`lambda`,`mu`) or with (`young`,`nu`).

Elastic anisotropic materials

The most general anisotropic materials are also implemented in the files `elastic.h` and `elastic.cpp` with the class names `anisotropicElasticMaterial` and `anisotropicElasticMP`, for the material point. They refer to elastic materials with a deformation independent stiffness tensor with only the major and minor symmetries, requiring for its definition 21 constants. All the material points created by a single material will have the same stiffness. If the tensor needs to change its orientation depending, for example, on the coordinates of the material point, the calling program will be responsible for rotating the strains, stresses, and stiffness. With this in mind, the program that calls MUESLI sets the strain in the local coordinate system, and then calls `updateCurrentState`. Then, the stresses and stiffness tensor calculated by MUESLI will all be in the local system. Once computed, the calling program can transform them to the necessary coordinates by the appropriate transformation.

MUESLI offers two constructors for these materials. In the first constructor, a string with a name and an array of 21 constants must be provided. The components of the array have the interpretation explained in ??

A.3 Materials for finite strain mechanical analysis

The class `finiteStrainMaterial` implements the mechanical behavior of materials employed in finite strain analyses. Similarly to their small strain counterpart, these materials create `finiteStrainMP` that can be queried for energies, stresses, tangent elasticities, etc. In contrast with the latter, finite strain classes must provide a richer interface because all tensors (stresses, strains, tangents) can refer to the reference or current configuration (or both!).

The class `finiteStrainMaterial` is, despite its apparent resemblance, completely different to the class `smallStrainMaterial`. The kind of mechanical simulations that can be performed with either material classes are

Component	Elastic stiffness
`c[0]`	C_{11}
`c[1]`	$C_{12} = C_{21}$
`c[2]`	$C_{13} = C_{31}$
`c[3]`	$C_{14} = C_{41}$
`c[4]`	$C_{15} = C_{51}$
`c[5]`	$C_{16} = C_{61}$
`c[6]`	C_{22}
`c[7]`	$C_{23} = C_{32}$

Table 2: Interpretation of the parameters in the constructor for an `elasticAnisotropicMaterial`. Voigt notation is employed with MUESLI's convention $1 \to 11,\ 2 \to 22,\ 3 \to 33,\ 4 \to 12 = 21,\ 5 \to 32 = 23,\ 6 \to 31 = 13$.

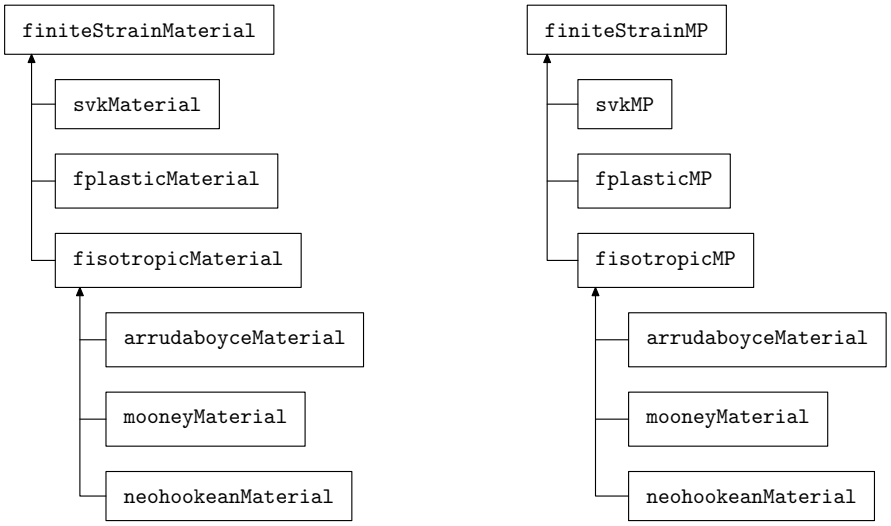

Figure 6: Currently existing classes for finite strain mechanical analyses.

completely different and it would make no sense to combine them or use them in a related fashion.

In this version, MUESLI includes classes for elastic and inelastic, finite strain materials. More specifically, these include `svkMaterial` (for Saint Venant-Kirchhoff models), `fplasticMaterial` (for finite strain, elastoplastic and viscoelastoplastic materials with von Mises criterion), and several isotropic, elastic models collected under the class `fisotropicMaterial`, including `arrudaboyceMaterial` (for compressible and incompressible Arruda-Boyce model), `mooneyMaterial` (Mooney-Rivlin model) and `neohookeanMaterial`

(Neohookean model). MUESLI defines, for each of these models, the corresponding `material` and `materialPoint`. See Figure 6 for an illustration.

The class `finiteStrainMaterial` is a *pure virtual*, parent class of the material family. Its children should be used in boundary value problems of nonlinear elasticity and inelasticity. The class declares the common interface of all children, which is:

```
class finiteStrainMaterial : public muesli::material
{
public:

                            finiteStrainMaterial(const std::string& name,
                                       const materialProperties& cl);

                            finiteStrainMaterial();
   virtual                  ~finiteStrainMaterial(){}


   virtual bool             check() const;
   virtual double           characteristicStiffness() const = 0;
   virtual double           density() const;
   virtual double           getProperty(const propertyName p) const;
   virtual void             print(std::ostream &of=std::cout) const = 0;
   virtual finiteStrainMP*  createMaterialPoint() const = 0;
   virtual void             setRandom() = 0;
   virtual double           waveVelocity() const;
};
```

These functions have the same purpose as their homonymous in the class `smallStrainMaterial`. As in the latter, the main purpose of the child objects of `finiteStrainMaterial` is to call the function `createMaterialPoint` that spawns a material point of the corresponding class.

The actual material behavior is encoded in the children of the class `finiteStrainMP`. As advanced, every material point of these classes should be able to, given its current and past states, compute its energies, stresses, tangent elasticities, as well as some other ancillary quantities. The interface of the class `finiteStrainMP`, listing all the possible functions, is

```
class finiteStrainMP : public muesli::materialPoint
{
public:
                            finiteStrainMP(const finiteStrainMaterial& m);
   virtual                  ~finiteStrainMP(){}

   bool                     testImplementation(std::ostream& of=std::cout,
                                               const bool testDE=true,
                                               const bool testDDE=true) const;
   virtual void             setRandom()=0;
```

```
// energy
virtual double          dissipatedEnergy() const = 0;
virtual double          energyDissipationPotential() const;
virtual double          storedEnergy() const = 0;


// stresses
virtual void            CauchyStress(istensor &sigma) const = 0;
virtual void            CauchyStressVector(double sigma[6]) const;
virtual void            energyMomentumTensor(const itensor& F,
                                             itensor& S) const;
virtual void            firstPiolaKirchhoffStress(itensor &P) const;
virtual void            KirchhoffStress(istensor &tau) const;
virtual void            KirchhoffStressVector(double tau[6]) const;
virtual void            secondPiolaKirchhoffStress(istensor &S) const;
virtual void            secondPiolaKirchhoffStressVector(double sigma[6]) const;


// elasticity tangents
virtual void            convectedTangent(double c[3][3][3][3]) const{}
virtual void            convectedTangentMatrix(double c[6][6]) const;
virtual void            materialTangent(double c[3][3][3][3]) const = 0;
virtual void            materialTangentMatrix(double c[6][6]) const;
virtual void            spatialTangent(double c[3][3][3][3]) const;
virtual void            spatialTangentMatrix(double c[6][6]) const;


// tangent contractions
virtual void            contractWithAllTangents(const ivector &v1,
                                                const ivector& v2,
                                                itensor&  Tdev,
                                                istensor& Tmixed,
                                                double&   Tvol) const;
virtual void            contractWithMaterialTangent(const ivector &v1,
                                                const ivector& v2,
                                                itensor &T) const;
virtual void            contractWithSpatialTangent(const ivector &v1,
                                                const ivector &v2,
                                                itensor &T) const;
virtual void            contractWithDeviatoricTangent(const ivector &v1,
                                                const ivector& v2,
                                                itensor &T) const;
virtual void            contractWithMixedTangent(istensor& CM) const{};
virtual void            materialTangentTimesSymmetricTensor(const istensor& M,
                                                istensor& CM) const;
virtual double          volumetricStiffness() const;
```

```
    // for 1-dimensional problems
    virtual double          stress(const double stretch) const;
    virtual double          stiffness(const double stretch) const;
    virtual double          storedEnergy(const double stretch) const;


    // bookkeeping
    virtual void            updateCurrentState(const double theTime,
                                               const itensor& F);
    virtual void            commitCurrentState();
    virtual void            resetCurrentState();

    virtual void            getCurrentState(double& tc,
                                               double* stc) const = 0;
    virtual void            setCurrentState(const double tc,
                                               const double* stc) = 0;
    virtual void            getLastState(double& tn, const double* stc) const = 0;
    virtual void            setLastState(const double tn, const double* stc) = 0;
    virtual int             getNStateVariables() const = 0;


    // miscellaneous
    double                  density() const;
    virtual double          plasticSlip() const = 0;
    virtual double          waveVelocity() const;
    const finiteStrainMaterial& parentMaterial() const;
}
```

The purpose of most of the functions in the class is self-explanatory, from their names. As in the case of the class `smallStainMP`, the tangent contractions refer to the operation (1).

A.4 Fluid materials

The class `fluidMaterial` is defined for analyses of fluids. The only class that is currently implemented is `newtonianMaterial`. The base class, `fluidMaterial` is a pure virtual class, so only objects of the derived classes can be instantiated. The class (see `fluid.h`) has the following definition:

```
class fluidMaterial : public muesli::material
{
        public:
                                fluidMaterial();
                                fluidMaterial(const std::string& name,
                                              const materialProperties& cl);
        virtual                 ~fluidMaterial() {}
```

```
        virtual bool            check() const = 0;
        virtual fluidMP*        createMaterialPoint() const = 0;
        virtual double          density() const = 0;
        virtual double          getProperty(const propertyName p) const = 0;
        virtual double          kinematicViscosity() const = 0;
        virtual void            print(std::ostream &of=std::cout) const = 0;
        virtual void            setRandom() = 0;
        virtual bool            test(std::ostream  &of=std::cout) = 0;
        virtual double          waveVelocity() const = 0;
};
```

These functions are self-explanatory or have the same purpose as those in the class `smallStrainMaterials`. As in the latter, the main task of any subclass of `fluidMaterial` is to spawn material points of their corresponding type.

The material behavior as well as its equation of state (EOS) are encoded in the children of the class `fluidMP`. Each material point of a class derived from `fluidMP` must provide the mechanical behavior its material constitutive law. This includes computing the energy, stresses and tangents for each possible state, and some auxiliary data. In addition, any `fluidMP` should be provided by an EOS to be completely defined. The actual functions that have to be programmed are:

```
virtual void            setRandom() = 0;

virtual double          density() const = 0;


// eos
virtual double          pressure(eos theEOS,
                                 std::vector<double> eosConst,
                                 double rho,
                                 double e) const;
virtual double          energy  (eos theEOS,
                                 std::vector<double> eosConst,
                                 double rho,
                                 double p) const;


// energies
virtual double          dissipatedEnergy()  const = 0;
virtual double          energyDissipation() const = 0;
virtual double          deviatoricEnergy()  const = 0;
virtual double          volumetricEnergy()  const = 0;


// stress
virtual double          pressure() const = 0;
```

```
virtual void            CauchyStress(istensor &sigma) const = 0;
virtual void            deviatoricStress(istensor &sigma) const = 0;
virtual void            volumetricStress(istensor &sigma) const = 0;


//tangents
virtual void            contractWithTangent(const ivector &v1,
                                            const ivector &v2,
                                            itensor &T) const;
virtual void            contractWithDeviatoricTangent(const ivector &v1,
                                                      const ivector &v2,
                                                      itensor &T) const;
virtual void            tangentTensor(itensor4& C) const{};


// bookkeeping
virtual void            updateCurrentState(const double t,
                                           const itensor& gradu,
                                           const double pressure) = 0;
virtual void            updateCurrentState(const double t,
                                           const itensor& gradu,
                                           const double pressure,
                                           const double rho) = 0;
virtual void            commitCurrentState() = 0;
virtual void            resetCurrentState() = 0;
```

The functions `pressure()` and `energy()` evaluate the direct and inverse EOS respectively. `theEOS` variable stores the type of EOS defined for the fluid and `eosConst` is a vector with the necessary EOS constants.

The function `CauchyStress()` computes the stress tensor at the current state. The construction of the deriviative of the Cauchy stress tensor respect to the strain rate tensor (symmetric part of the gradient of the velocity flow) is entrusted to the function `tangentTensor()`.

The rest of the functions are easily understood from their homonymus in the `smallStrainMP` class. The overload of the `updateCurrentState()` is due to the fact that, depending on the nature of the problem (compressible or incompressible fluid), density must be updated or not.

As in the class `smallStrainMP`, in addition to the former, mandatory functions, `fluidMP` classes might implement additional ones which might become useful in certain calculations. These are

```
// tangents
virtual void            contractWithTangent(const ivector &v1,
                                            const ivector &v2,
                                            itensor &T) const;
virtual void            contractWithDeviatoricTangent(const ivector &v1,
                                            const ivector &v2,
                                            itensor &T) const;
```

A.5 Materials for coupled thermomechanics at finite strains

The class `fthermomechanicalMaterial` models the strongly coupled thermomechanical behaviour of materials at finite strains. These include thermoelastic, thermoplastic, thermoviscoelastic, materials, among others.

Instead of constructing material models from scratch, objects in this class include a `finitestrainMaterial` that is responsible for the purely mechanical part of the response, and enhances this contribution with thermal and coupled terms.

The thermomechanical potentials

Thermomechanical materials are defined in MUESLI with thermomechanical potentials from which all other quantities are derived. Only one of these potentials need to be defined, all the rest obtained by the corresponding Legendre transform. The most convenient potential is often Helmholtz's free energy which can be written in incremental form as

$$\Psi(\boldsymbol{F},\theta;\boldsymbol{F}_n,\boldsymbol{\xi}_n,\theta_n) = W_n(\boldsymbol{F};\boldsymbol{F}_n,\boldsymbol{\xi}_n) + \psi^{tm}(\det\boldsymbol{F},\theta) + \psi^t(\theta)\ . \tag{2}$$

The term W_n is an *incremental* potential for the purely mechanical response, and can include inelastic effects. In MUESLI, the computation of this term is delegated to a `finitestrainMP`. The potential ψ^{tm} refers to the coupled mechanical-thermal energy, and ψ^t to the purely thermal contributions.

If θ_0 is a reference temperature, the contributions for the coupled response and thermal energy are often chosen to be of the form

$$\psi^{tm}(J,\theta) = -3\alpha\kappa(\theta-\theta_0)\log J\ , \qquad \psi^t(\theta) = c_0(\theta-\theta_0-\theta\log\frac{\theta}{\theta_0}). \tag{3}$$

In these relations α is the thermal expansion coefficient, κ is the bulk modulus, c_0 the heat capacity, and $J=\det\boldsymbol{F}$.

In addition to Helmholtz's free energy, the basic quantities that need to be provided by MUESLI are the first and second derivatives of this potential with respect to its arguments.

The first derivatives of the free energy are (up to a sign change):

$$\boldsymbol{P} = \frac{\partial\psi}{\partial\boldsymbol{F}}\ , \qquad s = -\frac{\partial\psi}{\partial\theta}. \tag{4}$$

The first derivative is the first Piola-Kirchhoff stress tensor, which will differ from the stress in a purely mechanical material due to coupling contributions; the second derivative corresponds to the thermodynamic definition of the entropy. For the class of free energies defined above, the entropy evaluates to

$$s = -\frac{\partial\psi}{\partial\theta} = 3\alpha\kappa\log J + c_0\log\frac{\theta}{\theta_0}. \tag{5}$$

MUESLI must also provide the three types of second derivatives of ψ, namely,

$$\mathbb{A} = \frac{\partial \boldsymbol{P}}{\partial \boldsymbol{F}} \ , \qquad \boldsymbol{M} = \frac{\partial \boldsymbol{P}}{\partial \theta} \ , \qquad c = -\theta \frac{\partial^2 \psi}{\partial \theta^2}. \tag{6}$$

The fourth order tensor $\mathbb{A}$ is the material consistent tangent; the tensor $\boldsymbol{M}$ is the tensor of thermomechanical coupling; finally, c is the heat capacity at constant deformation.

Bibliography

[1] *ABAQUS*. Dassault Systèmes. http://www.3ds.com/products-services/simulia/products/abaqus/.

[2] EIGEN: a C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms. http://eigen.tuxfamily.org/.

[3] J Planas, I Romero, and J M Sancho. B free. *Computer Methods in Applied Mechanics and Engineering*, 217-220:226–235, 2012.