# **Akantu** User's Guide

April 10, 2018 — Version 3.0.0

# Contents

1.	Introduction	5
2.	Getting Started	7
	2.1 Downloading the Code	7
	2.2 Compiling Akantu	7
	2.3 Writing a main Function	7
	2.4 Creating and Loading a Mesh	8
	2.5 Using Arrays	8
	2.5.1 Arrays iterators	9
	2.5.2 Vector & Matrix	10
	2.6 Manipulating group of nodes and/or elements	12
	2.6.1 The NodeGroup object.	12
	2.6.2 The ElementGroup object	13
	2.7 Compiling your simulation	13
3.	FEEngine	15
	2.1 Mathematical Operations	15
	2.2 Elements	15
	2.2.1 Looparametria Elements	10
	3.2.1 Isoparametric Elements	10
	3.2.2 Conesive Elements	10
	5.2.5 Structural Elements	20
4.	Solid Mechanics Model	21
	4.1 Model Setup	22
	4.1.1 Setting Initial Conditions	22
	4.1.2 Setting Boundary Conditions	22
	4.1.3 Material Selector	25
	4.1.4 Insertion of Cohesive Elements	25
	4.2 Static Analysis	26
	4.2.1 Static implicit analysis with dynamic insertion of cohesive elements	27
	4.3 Dynamic Methods	29
	4.3.1 Implicit Time Integration	30
	4.3.2 Explicit Time Integration	31
	4.4 Constitutive Laws	33
	4.4.1 Elasticity	34
	4.4.2 Neo-Hookean (B.4)	36
	4.4.3 Visco-Elasticity (B.5)	37
	4.4.4 Small-Deformation Plasticity (B.6)	38
	4.4.5 Damage	40

	<ul> <li>4.5 Cohesive Constitutive laws</li> <li>4.5.1 Linear Irreversible Law (B.10)</li> <li>4.5.2 Linear Cohesive Law with Friction (B.11)</li> <li>4.5.3 Linear Cohesive Law with Fatigue (B.12)</li> <li>4.5.4 Exponential Cohesive Law (B.13)</li> <li>4.6 Non-Local Constitutive Laws</li> <li>4.6.1 Non-local constitutive laws</li> <li>4.6.2 Non-local weight functions</li> <li>4.7 Adding a New Constitutive Law</li> <li>4.7.1 Adding a New Non-Local Constitutive Law.</li> </ul>	41 42 43 43 44 45 45 45 50
5.	Structural Mechanics Model	53
	<ul> <li>5.1 Model Setup</li></ul>	53 53 54 55
6.	Heat Transfer Model	57
	<ul> <li>6.1 Theory</li> <li>6.2 Using the Heat Transfer Model</li> <li>6.2.1 Explicit Dynamic</li> </ul>	57 57 58
7.	Input/Output	61
	<ul> <li>7.1 Input file</li> <li>7.1.1 Akantu Parser</li> <li>7.1.2 Grammar</li> <li>7.1.3 Material section</li> <li>7.2 Output data</li> <li>7.2.1 Generic data</li> <li>7.2.2 Cohesive elements' data</li> <li>7.2.3 Advanced dumping.</li> </ul>	61 61 62 62 63 64
8.	Parallel Computation	67
	<ul> <li>8.1 Initializing the Parallel Context</li></ul>	67 68 68
9.	Python interface	69
A.	Shape Functions	71
	<ul> <li>A.1 1D-Shape Functions</li> <li>A.1.1 Segment 2.</li> <li>A.1.2 Segment 3.</li> <li>A.2 2D-Shape Functions</li> <li>A.2.1 Triangle 3</li> <li>A.2.2 Triangle 6</li> <li>A.2.3 Quadrangle 4</li> <li>A.2.4 Quadrangle 8</li> <li>A.3 3D-Shape Functions</li> <li>A.3.1 Tetrahedron 4</li> <li>A.3.2 Tetrahedron 10</li> </ul>	71 71 72 72 73 73 74 74 74
	A.3.4 Pentahedron 6	75 75

A.3.5 Hexahedron 20	76
A.3.6 Pentahedron 15	79
B. Material parameters	81
B.1 Linear elastic isotropic	81
B.2 Linear elastic anisotropic	81
B.3 Linear elastic orthotropic	81
B.4 Neohookean (finite strains)	82
B.5 Standard linear solid	82
B.6 Elasto-plastic linear isotropic hardening	82
B.7 Damage: Marigo.	82
B.8 Damage: Mazars	83
B.9 Cohesive linear	83
B.10 Cohesive bilinear	83
B.11 Cohesive linear with friction	83
B.12 Cohesive linear fatigue	84
B.13 Cohesive exponential	84
C. Package dependencies	85
Index	90
References	93

## Authors

- Nicolas Richart
- Guillaume Anciaux
- Jean-François Molinari
- Alejandro M. Aragón
- Alodie Schneuwly
- Aranda Ruiz Josue
- Aurelia Isabel Cuba Ramos
- Benjamin Paccaud
- Clement Roux
- Cyprien Wolff
- Damien Scantamburlo
- Damien Spielmann
- Dana Christen
- Daniel Pino Muñoz
- David Simon Kammer
- Fabian Barras
- Jaehyun Cho
- Jean-François Jerier
- Leonardo Snozzi

- Lucas Frerot
- Marco Vocialta
- Marion Estelle Chambart
- Mathilde Radiguet
- Mauro Corrado
- Nicholas Molyneaux
- Okan Yilmaz
- Pedro Romero
- Peter Spijker
- Ramin Aghababaei
- Rui Wang
- Sacha Laffely
- Seyedeh Mohadeseh Taheri Mousavi
- Srinivasa Babu Ramisetti
- Sébastien Hartmann
- Thomas Menouillard
- Till Junge
- Vladislav Yastrebov

## Changes, New Features, and Fixes

### Version 3.0.0

**c++14** Switch from C++ standard 2003 to 2014 Example of changes implied by this:

```
for (UInt g = _not_ghost; g <= _ghost; ++g) {
  GhostType gt = (GhostType)g;
  Mesh::type_iterator it = this->mesh.firstType(spatial_dimension, gt);
  Mesh::type_iterator end = this->mesh.lastType(spatial_dimension, gt);
  for (; it != end; ++it) {
    ElementType & type = *it;
    ...
  }
}
```

becomes

feature Parallel cohesive elements

feature Models using new interface for solvers

- Same configuration for all models
- Solver can be configured in input file
- PETSc interface temporary inactive
- Periodic boundary condition temporary inactive

feature Element groups created by default for "physical\_names"

feature Named arguments for functions (e.g. model.initFull(\_analysis\_method = \_static))

- api Only one function to solve a step model.solveStep()
- api Simplification of the parallel simulation with the mesh.distribute() function

## Chapter 1

# Introduction

**Akantu** means "little element" in Kinyarwanda, a Bantu language. From now on, it is also an opensource object-oriented *Finite-Element* library with the ambition to be generic and efficient. **Akantu** is developed within the LSMS (Computational Solid Mechanics Laboratory, 1sms.epf1.ch) at the Ecole Polytechnique Federale of Lausanne, Switzerland. The open-source philosophy is important for any scientific software project evolution. The collaboration permitted by shared codes enforces sanity when users (and not only developers) can scrutinize (and possibly criticize) the implementation details.

**Akantu** was born with the vision to associate genericity, robustness and efficiency while benefiting from the open-source visibility. Genericity is necessary to allow the easy exploration of mathematical formulations through algorithmic ideas. Robustness and reliability is naturally expected from any simulation software, even more in the context of parallel computations. In order to achieve these goals, we made noticeable choices in the architecture of **Akantu**. First we decided to use the object-oriented paradigm through C++. Then, in order to prevent extra cost associated to virtual function calls, we designed the library as a hybrid architecture with objects at high level layers and vectorization at low level layers. Thus, **Akantu** benefits from inheritance and polymorphism mechanisms without the counterpart of having virtual calls within critical loops. This coding philosophy, which was demonstrated to be highly efficient, is innovative in the field of *Finite-Element* software.

This document is appropriate for researchers and engineers willing to use **Akantu** in order to perform a finite-element calculation for solid mechanics, structural mechanics, contact mechanics or heat transfer. The solid mechanics solver, which is the most complete and functional part of **Akantu**, is presented in details in the remainder of this document.

## Chapter 2

# **Getting Started**

## 2.1 Downloading the Code

The **Akantu** source code can be requested using the form accessible at the URL http://lsms. epfl.ch/akantu. There, you will be asked to accept the LGPL license terms.

## 2.2 Compiling Akantu

**Akantu** is a cmake project, so to configure it, you can either follow the usual way:

```
> cd akantu
> mkdir build
> cd build
> ccmake ..
[ Set the options that you need ]
> make
> make install
```

Or, use the Makefile we added for your convenience to handle the cmake configuration

```
> cd akantu
> make config
> make
> make install
```

All the Akantu options are documented in Appendix C.

## 2.3 Writing a main Function

First of all, **Akantu** needs to be initialized. The memory management included in the core library handles the correct allocation and de-allocation of vectors, structures and/or objects. Moreover, in parallel computations, the initialization procedure performs the communication setup. This is achieved by a pair of functions (initialize and finalize) that are used as follows:

```
#include "aka_common.hh"
#include "..."
using namespace akantu;
int main(int argc, char *argv[]) {
    initialize("input_file.dat", argc, argv);
    // your code
    ...
```

finalize();

}

The initialize function takes the text inpute file and the program parameters which can be parsed by **Akantu** in due form (see 7.1). Obviously it is necessary to include all files needed in main. In this manual all provided code implies the usage of akantu as namespace.

## 2.4 Creating and Loading a Mesh

In its current state, **Akantu** supports three types of meshes: Gmsh [1], Abaqus [2] and Diana [3]. Once a Mesh object is created with a given spatial dimension, it can be filled by reading a mesh input file. The method read of the class Mesh infers the mesh type from the file extension. If a non-standard file extension is used, the mesh type has to be specified.

```
UInt spatial_dimension = 2;
Mesh mesh(spatial_dimension);
// Reading Gmsh files
mesh.read("my_gmsh_mesh.msh");
mesh.read("my_gmsh_mesh", _miot_gmsh);
// Reading Abaqus files
mesh.read("my_abaqus_mesh.inp");
mesh.read("my_abaqus_mesh", _miot_abaqus);
// Reading Diana files
mesh.read("my_diana_mesh.dat");
mesh.read("my_diana_mesh", _miot_diana);
```

The Gmsh reader adds the geometrical and physical tags as mesh data. The physical values are stored as a UInt data called tag\_0, if a string name is provided it is stored as a std::string data named physical\_names. The geometrical tag is stored as a UInt data named tag\_1.

The Abaqus reader stores the ELSET in ElementGroups and the NSET in NodeGroups. The material assignment can be retrieved from the std::string mesh data named abaqus\_material.

## 2.5 Using Arrays

Data in **Akantu** can be stored in data containers implemented by the **Array** class. In its most basic usage, the **Array** class implemented in **Akantu** is similar to the **vector** class of the Standard Template Library (STL) for C++. A simple **Array** containing a sequence of **nb\_element** values (of a given type) can be generated with:

```
Array<type> example_array(nb_element);
```

where type usually is Real, Int, UInt or bool. Each value is associated to an index, so that data can be accessed by typing:

```
auto & val = example_array(index)
```

**Arrays** can also contain tuples of values for each index. In that case, the number of components per tuple must be specified at the **Array** creation. For example, if we want to create an **Array** to store the coordinates (sequences of three values) of ten nodes, the appropriate code is the following:

```
UInt nb_nodes = 10;
UInt spatial_dimension = 3;
```

Array<Real> position(nb\_nodes, spatial\_dimension);

In this case the *x* position of the eighth node number will be given by position(7, 0) (in C++, numbering starts at 0 and not 1). If the number of components for the sequences is not specified, the default value of 1 is used. Here is a list of some basic operations that can be performed on **Array**:

- resize(size) change the size of the Array.
- clear() set all entries of the Array to zero.
- set(t) set all entries of the Array to t.
- **copy(const Array**<T> & **other)** copy another **Array** into the current one. The two **Array** should have the same number of components.
- push\_back(tuple) append a tuple with the correct number of components at the end of the Array.
- o erase(i) erase the value at the i-th position.
- o find(value) search value in the current Array. Return position index of the first occurence or −1 if not found.
- storage() Return the address of the allocated memory of the Array.

#### 2.5.1 Arrays iterators

It is very common in **Akantu** to loop over arrays to perform a specific treatment. This ranges from geometric calculation on nodal quantities to tensor algebra (in constitutive laws for example). The **Array** object has the possibility to request iterators in order to make the writing of loops easier and enhance readability. For instance, a loop over the nodal coordinates can be performed like:

```
//accessing the nodal coordinates Array (spatial_dimension components)
const auto & nodes = mesh.getNodes();
//creating the iterators
auto it = nodes.begin(spatial_dimension);
auto end = nodes.end(spatial_dimension);
for (; it != end; ++it){
    const auto & coords = (*it);
    //do what you need
    ....
}
```

In that example, each coords is a Vector<Real> containing geometrical array of size spatial\_dimension and the iteration is conveniently performed by the Array iterator. With the switch to c++14 this can be also written as

with the switch to C++14 this can be also written as

```
//accessing the nodal coordinates Array (spatial_dimension components)
const auto & nodes = mesh.getNodes();
for (const auto & coords : make_view(nodes, spatial_dimension) {
    //do what you need
    ....
}
```

The Array object is intensively used to store second order tensor values. In that case, it should be specified that the returned object type is a matrix when constructing the iterator. This is done when calling the begin function. For instance, assuming that we have a Array storing stresses, we can loop over the stored tensors by:

```
//creating the iterators
auto it = stresses.begin(spatial_dimension, spatial_dimension);
auto end = stresses.end(spatial_dimension, spatial_dimension);
for (; it != end; ++it){
   Matrix<Real> & stress = (*it);
   //do what you need
   ....
}
```

In that last example, the Matrix objects are spatial\_dimension × spatial\_dimension matrices. The light objects Matrix and Vector can be used and combined to do most common linear algebra. If the number of component is 1, it is possible to use a scalar\_iterator rather than the vector/matrix one.

In general, a mesh consists of several kinds of elements. Consequently, the amount of data to be stored can differ for each element type. The straightforward example is the connectivity array, namely the sequences of nodes belonging to each element (linear triangular elements have fewer nodes than, say, rectangular quadratic elements etc.). A particular data structure called ElementTypeMapArray is provided to easily manage this kind of data. It consists of a group of Arrays, each associated to an element type. The following code can retrieve the ElementTypeMapArray which stores the connectivity arrays for a mesh:

```
const ElementTypeMapArray<UInt> & connectivities = mesh.getConnectivities();
```

Then, the specific array associated to a given element type can be obtained by

const Array<UInt> & connectivity\_triangle = connectivities(\_triangle\_3);

where the first order 3-node triangular element was used in the presented piece of code.

#### 2.5.2 Vector & Matrix

The Array iterators as presented in the previous section can be shaped as Vector or Matrix. This objects represent  $1^{st}$  and  $2^{nd}$  order tensors. As such they come with some functionalities that we will present a bit more into detail in this here.

#### Vector<T>

- 1. Accessors:
  - v(i) gives the *i*<sup>th</sup> component of the vector v
  - v[i] gives the *i*<sup>th</sup> component of the vector v
  - v.size() gives the number of component
- 2. Level 1: (results are scalars)
  - v.norm() returns the geometrical norm (*L*<sub>2</sub>)
  - **v.norm**<**N**>() returns the  $L_N$  norm defined as  $(\sum_i |\mathbf{v}(\mathbf{i})|^N)^{1/N}$ . N can take any positive integer value. There are also some particular values for the most commonly used norms, **L\_1** for the Manhattan norm, **L\_2** for the geometrical norm and **L\_inf** for the norm infinity.

- v.dot(x) return the dot product of v and x
- v.distance(x) return the geometrical norm of v x
- 3. Level 2: (results are vectors)
  - v += s, v -= s, v \*= s, v /= s those are element-wise operators that sum, substract, multiply or divide all the component of v by the scalar s
  - v += x, v -= x sums or substracts the vector x to/from v
  - v.mul(A, x, alpha) stores the result of  $\alpha Ax$  in v,  $\alpha$  is equal to 1 by default
  - v.solve(A, b) stores the result of the resolution of the system Ax = b in v
  - $\circ$  <code>v.crossProduct(v1, v2)</code> computes the cross product of <code>v1</code> and <code>v2</code> and stores the result in <code>v</code>

#### Matrix<T>

- 1. Accessors:
  - A(i, j) gives the component  $A_{ij}$  of the matrix A
  - A(i) gives the *i*<sup>th</sup> column of the matrix as a Vector
  - **A[k]** gives the  $k^{th}$  component of the matrix, matrices are stored in a column major way, which means that to access  $A_{ij}$ , k = i + jM
  - A.rows() gives the number of rows of A (M)
  - A.cols() gives the number of columns of A (N)
  - A.size() gives the number of component in the matrix  $(M \times N)$
- 2. Level 1: (results are scalars)
  - A.norm() is equivalent to A.norm<L\_2>()
  - **A.norm**<**N**>() returns the  $L_N$  norm defined as  $(\sum_i \sum_j |\mathbf{A}(i,j)|^N)^{1/N}$ . N can take any positive integer value. There are also some particular values for the most commonly used norms, **L\_1** for the Manhattan norm, **L\_2** for the geometrical norm and **L\_inf** for the norm infinity.
  - A.trace() return the trace of A
  - A.det() return the determinant of A
  - A.doubleDot(B) return the double dot product of A and B, A : B
- 3. Level 3: (results are matrices)
  - A.eye(s), Matrix<T>::eye(s) fills/creates a matrix with the *sI* with *I* the identity matrix
  - A.inverse(B) stores  $B^{-1}$  in A
  - A.transpose() returns  $A^t$
  - A.outerProduct(v1, v2) stores  $v_1v_2^t$  in A
  - C.mul<t\_A, t\_B>(A, B, alpha): stores the result of the product of A and codeB time the scalar alpha in C. t\_A and t\_B are boolean defining if A and B should be transposed or not.

t_A	t_B	result
false	false	$C = \alpha AB$
false	true	$C = \alpha A B^t$
true	false	$C = \alpha A^t B$
true	true	$\boldsymbol{C} = \alpha \boldsymbol{A}^t \boldsymbol{B}^t$

• A.eigs(d, V) this method computes the eigenvalues and eigenvectors of A and store the results in d and V such that  $d(i) = \lambda_i$  and  $V(i) = v_i$  with  $Av_i = \lambda_i v_i$  and  $\lambda_1 > ... > \lambda_i > ... > \lambda_N$ 

#### Tensor3<T>

Accessors:

- t(i, j, k) gives the component  $T_{ijk}$  of the tensor t
- t(k) gives the k<sup>th</sup> two-dimensional tensor as a Matrix
- t[k] gives the k<sup>th</sup> two-dimensional tensor as a Matrix

## 2.6 Manipulating group of nodes and/or elements

**Akantu** provides the possibility to manipulate subgroups of elements and nodes. Any **ElementGroup** and/or **NodeGroup** must be managed by a **GroupManager**. Such a manager has the role to associate group objects to names. This is a useful feature, in particular for the application of the boundary conditions, as will be demonstrated in section 4.1.2. To most general group manager is the Mesh class which inheritates from the **GroupManager** class.

For instance, the following code shows how to request an element group to a mesh:

```
// request creation of a group of nodes
NodeGroup & my_node_group = mesh.createNodeGroup("my_node_group");
// request creation of a group of elements
ElementGroup & my_element_group = mesh.createElementGroup("my_element_group"
);
/* fill and use the groups */
```

#### 2.6.1 The NodeGroup object

A group of nodes is stored in **NodeGroup** objects. They are quite simple objects which store the indexes of the selected nodes in a **Array**<**UInt>**. Nodes are selected by adding them when calling **NodeGroup::add**. For instance you can select nodes having a positive X coordinate with the following code:

```
const auto & nodes = mesh.getNodes();
auto & group = mesh.createNodeGroup("XpositiveNode");
for (auto && data : enumerate(make_view(nodes, spatial_dimension))){
  auto node = std::get<0>(data);
  const auto & position = std::get<1>(data);
  if (position(0) > 0) group.add(node);
}
```

#### 2.6.2 The ElementGroup object

A group of elements is stored in ElementGroup objects. Since a group can contain elements of various types the ElementGroup object stores indexes in a ElementTypeMapArray<UInt> object. Then elements can be added to the group by calling addElement.

For instance, selecting the elements for which the barycenter of the nodes has a positive *X* coordinate can be made with:

```
auto & group = mesh.createElementGroup("XpositiveElement");
Vector<Real> barycenter(spatial_dimension);
for(auto type : mesh.elementTypes()){
  UInt nb_element = mesh.getNbElement(type);
  for(UInt e = 0; e < nb_element; ++e) {
    Element element{type, e, _not_ghost};
    mesh.getBarycenter(element, barycenter);
    if (barycenter(_x) > 0.) group.add(element);
    }
}
```

## 2.7 Compiling your simulation

The easiest way it to create a cmake project. The minimum CMakeLists.txt file would look like

```
project(my_simu)
cmake_minimum_required(VERSION 3.0.0)
find_package(Akantu REQUIRED)
```

add\_akantu\_simulation(my\_simu my\_simu.cc)

## **Chapter 3**

# FEEngine

The **FEEngine** interface is dedicated to handle the finite-element approximations and the numerical integration of the weak form. As we will see in Chapter 4, Model creates its own **FEEngine** object so the explicit creation of the object is not required.

## 3.1 Mathematical Operations

Using the **FEEngine** object, one can compute a interpolation, an integration or a gradient. A simple example is given below.

```
// having a FEEngine object
FEEngine *fem = new FEEngineTemplate<IntegratorGauss,ShapeLagrange>(my_mesh,
                                                                     dim.
                                                                     "my_fem");
// instead of this, a FEEngine object can be get using the model:
// model.getFEEngine()
//compute the gradient
Array<Real> u; //append the values you want
Array<Real> nablauq; //gradient array to be computed
// compute the gradient
fem->gradientOnIntegrationPoints(const Array<Real> &u,
         Array<Real> &nablaug.
         const UInt nb_degree_of_freedom,
         const ElementType & type);
// interpolate
Array<Real> uq; //interpolated array to be computed
// compute the interpolation
fem->interpolateOnIntegrationPoints(const Array<Real> &u,
                                    Array<Real> &uq,
                                    UInt nb_degree_of_freedom,
                                    const ElementType & type);
// interpolated function can be integrated over the elements
Array<Real> int_val_on_elem;
// integrate
fem->integrate(const Array<Real> &uq,
               Array<Real> &int_uq,
               UInt nb_degree_of_freedom,
               const ElementType & type);
```

Another example below shows how to integrate stress and strain fields over elements assigned to a particular material.

```
UInt sp_dim = 3; //spatial dimension
UInt m = 1; //material index of interest
const ElementType type = _tetrahedron_4; //element type
// get the stress and strain arrays associated to the material index m
const Array<Real> & strain_vec = model.getMaterial(m).getGradU(type);
const Array<Real> & stress_vec = model.getMaterial(m).getStress(type);
// get the element filter for the material index
const Array<UInt> & elem_filter = model.getMaterial(m).getElementFilter(type);
// initialize the integrated stress and strain arrays
Array<Real> int_strain_vec(elem_filter.getSize(),
                           sp_dim*sp_dim, "int_of_strain");
Array<Real> int_stress_vec(elem_filter.getSize(),
                           sp_dim*sp_dim, "int_of_stress");
// integrate the fields
model.getFEEngine().integrate(strain_vec, int_strain_vec,
                              sp_dim*sp_dim, type, _not_ghost, elem_filter);
model.getFEEngine().integrate(stress_vec, int_stress_vec,
                              sp_dim*sp_dim, type, _not_ghost, elem_filter);
```

## 3.2 Elements

The base for every Finite-Elements computation is its mesh and the elements that are used within that mesh. The element types that can be used depend on the mesh, but also on the dimensionality of the problem (1D, 2D or 3D). In **Akantu**, several isoparametric Lagrangian element types are supported (and one serendipity element). Each of these types is discussed in some detail below, starting with the 1D-elements all the way to the 3D-elements. More detailed information (shape function, location of Gaussian quadrature points, and so on) can be found in Appendix A.

### 3.2.1 Isoparametric Elements

1D

In **Akantu**, there are two types of isoparametric elements defined in 1D. These element types are called <u>\_segment\_2</u> and <u>\_segment\_3</u>, and are depicted schematically in Figure 3.1. Some of the basic properties of these elements are listed in Table 3.1.



Figure 3.1: Schematic overview of the two 1D element types in **Akantu**. In each element, the node numbering as used in **Akantu** is indicated and also the quadrature points are highlighted (gray circles).

Element type	Order	# nodes	# quad. points
_segment_2	linear	2	1
_segment_3	quadratic	3	2

Table 3.1: Some basic properties of the two 1D isoparametric elements in Akantu.

#### 2D

In **Akantu**, there are four types of isoparametric elements defined in 2D. These element types are called \_triangle\_3, \_triangle\_6, \_quadrangle\_4 and \_quadrangle\_8, and all of them are depicted in Figure 3.2. As with the 1D elements, some of the most basic properties of these elements are listed in Table 3.2. It is important to note that the first element is linear, the next two quadratic and the last one cubic. Furthermore, the last element type (\_quadrangle\_8) is not a Lagrangian but a serendipity element.



Figure 3.2: Schematic overview of the four 2D element types in **Akantu**. In each element, the node numbering as used in **Akantu** is indicated and also the quadrature points are highlighted (gray circles).

#### 3D

In **Akantu**, there are three types of isoparametric elements defined in 3D. These element types are called \_tetrahedron\_4, \_tetrahedron\_10 and \_hexahedron\_8, and all of them are depicted schematically in Figure 3.3. As with the 1D and 2D elements some of the most basic properties of these elements are listed in Table 3.3.

Element type	Order	# nodes	# quad. points
_triangle_3	linear	3	1
_triangle_6	quadratic	6	3
_quadrangle_4	quadratic	4	4
_quadrangle_8	cubic	8	9

Table 3.2: Some basic properties of the four 2D isoparametric elements in Akantu.



Figure 3.3: Schematic overview of the three 3D element types in **Akantu**. In each element, the node numbering as used in **Akantu** is indicated and also the quadrature points are highlighted (gray spheres).

Element type	Order	# nodes	# quad. points
_tetrahedron_4	linear	4	1
_tetrahedron_10	quadratic	10	4
_hexahedron_8	cubic	8	8

Table 3.3: Some basic properties of the three 3D isoparametric elements in Akantu.

### 3.2.2 Cohesive Elements

The cohesive elements that have been implemented in **Akantu** are based on the work of Ortiz and Pandolfi [4]. Their main properties are reported in Table 3.4.

Element type	Facet type	Order	# nodes	# quad. points
_cohesive_1d_2	_point_1	linear	2	1
_cohesive_2d_4	_segment_2	linear	4	1
_cohesive_2d_6	_segment_3	quadratic	6	2
_cohesive_3d_6	_triangle_3	linear	6	1
_cohesive_3d_12	_triangle_6	quadratic	12	3

Table 3.4: Some basic properties of the cohesive elements in **Akantu**.

Cohesive element insertion can be either realized at the beginning of the simulation or it can be carried out dynamically during the simulation. The first approach is called *intrinsic*, the second



Figure 3.4: Cohesive element in 2D for quadratic triangular elements T6.



Figure 3.5: Insertion of a cohesive element.

one *extrinsic*. When an element is present from the beginning, a bilinear or exponential cohesive law should be used instead of a linear one. A bilinear law works exactly like a linear one except for an additional parameter  $\delta_0$  separating an initial linear elastic part from the linear irreversible one. For additional details concerning cohesive laws see Section 4.5.

Extrinsic cohesive elements are dynamically inserted between two standard elements when

$$\sigma_{\rm eff} > \sigma_{\rm c} \quad {\rm with} \quad \sigma_{\rm eff} = \sqrt{\sigma_{\rm n}^2 + \frac{\tau^2}{\beta^2}}$$

$$(3.1)$$

in which  $\sigma_n$  is the tensile normal traction and  $\tau$  the resulting tangential one (Figure 3.5).

For the static analysis of the structures containing cohesive elements, the stiffness of the cohesive elements should also be added to the total stiffness of the structure. Considering a 2D quadratic cohesive element as that in Figure 3.4, the opening displacement along the mid-surface can be written as:

$$\Delta(s) = \llbracket u \rrbracket N(s) = \begin{bmatrix} u_3 - u_0 & u_4 - u_1 & u_5 - u_2 \\ v_3 - v_0 & v_4 - v_1 & v_5 - v_2 \end{bmatrix} \begin{bmatrix} N_0(s) \\ N_1(s) \\ N_2(s) \end{bmatrix} = N^k A U = P U$$
(3.2)

The U, A and  $N^k$  are as following:

$$\boldsymbol{U} = \begin{bmatrix} u_0 & v_0 & u_1 & v_1 & u_2 & v_2 & u_3 & v_3 & u_4 & v_4 & u_5 & v_5 \end{bmatrix}$$
(3.3)

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 \end{bmatrix}$$
(3.4)

$$\boldsymbol{N}^{k} = \begin{bmatrix} N_{0}(s) & 0 & N_{1}(s) & 0 & N_{2}(s) & 0\\ 0 & N_{0}(s) & 0 & N_{1}(s) & 0 & N_{2}(s) \end{bmatrix}$$
(3.5)

The consistent stiffness matrix for the element is obtained as

$$K = \int_{S_0} \mathbf{P}^{\mathrm{T}} \frac{\partial T}{\partial \delta} \mathbf{P} \,\mathrm{d}S_0 \tag{3.6}$$

where *T* is the cohesive traction and  $\delta$  the opening displacement (for more details check Section 3.4).

#### 3.2.3 Structural Elements

#### **Bernoulli Beam Elements**

These elements allow to compute the displacements and rotations of structures constituted by Bernoulli beams. **Akantu** defines them for both 2D and 3D problems respectively in the element types \_bernoulli\_beam\_2 and \_bernoulli\_beam\_3. A schematic depiction of a beam element is shown in Figure 3.6 and some of its properties are listed in Table 3.5.

**Note:** Beam elements are of mixed order: the axial displacement is linearly interpolated while transverse displacements and rotations use cubic shape functions.

Figure 3.6: Schematic depiction of a Bernoulli beam element (applied to 2D and 3D) in **Akantu**. The node numbering as used in **Akantu** is indicated, and also the quadrature points are highlighted (gray circles).

Element type	Dimension	# nodes	# quad. points	# d.o.f.
_bernoulli_beam_2	2D	2	3	6
_bernoulli_beam_3	3D	2	3	12

Table 3.5: Some basic properties of the beam elements in Akantu

## **Chapter 4**

# **Solid Mechanics Model**

The solid mechanics model is a specific implementation of the Model interface dedicated to handle the equations of motion or equations of equilibrium. The model is created for a given mesh. It will create its own FEEngine object to compute the interpolation, gradient, integration and assembly operations. A SolidMechanicsModel object can simply be created like this:

SolidMechanicsModel model(mesh);

where **mesh** is the mesh for which the equations are to be solved. A second parameter called **spatial\_dimension** can be added after **mesh** if the spatial dimension of the problem is different than that of the mesh.

This model contains at least the following six Arrays:

- blocked\_dofs contains a Boolean value for each degree of freedom specifying whether that degree is blocked or not. A Dirichlet boundary condition can be prescribed by setting the blocked\_dofs value of a degree of freedom to true. A Neumann boundary condition can be applied by setting the blocked\_dofs value of a degree of freedom to false. The displacement, velocity and acceleration are computed for all degrees of freedom for which the blocked\_dofs value is set to false. For the remaining degrees of freedom, the imposed values (zero by default after initialization) are kept.
- **displacement** contains the displacements of all degrees of freedom. It can be either a computed displacement for free degrees of freedom or an imposed displacement in case of blocked ones (*u* in the following).
- **velocity** contains the velocities of all degrees of freedom. As **displacement**, it contains computed or imposed velocities depending on the nature of the degrees of freedom (*u* in the following).
- acceleration contains the accelerations of all degrees of freedom. As **displacement**, it contains computed or imposed accelerations depending on the nature of the degrees of freedom ( $\ddot{u}$  in the following).

**external\_force** contains the external forces applied on the nodes ( $f_{ext}$  in the following).

**internal\_force** contains the internal forces on the nodes ( $f_{int}$  in the following).

Some examples to help to understand how to use this model will be presented in the next sections.

## 4.1 Model Setup

### 4.1.1 Setting Initial Conditions

For a unique solution of the equations of motion, initial displacements and velocities for all degrees of freedom must be specified:

$$u(t = 0) = u_0$$
(4.1)  
$$\dot{u}(t = 0) = v_0$$
(4.2)

The solid mechanics model can be initialized as follows:

```
model.initFull()
```

This function initializes the internal arrays and sets them to zero. Initial displacements and velocities that are not equal to zero can be prescribed by running a loop over the total number of nodes. Here, the initial displacement in *x*-direction and the initial velocity in *y*-direction for all nodes is set to 0.1 and 1, respectively.

```
auto & disp = model.getDisplacement();
auto & velo = model.getVelocity();
for (UInt node = 0; node < mesh.getNbNodes(); ++node) {
    disp(node, 0) = 0.1;
    velo(node, 1) = 1.;
}
```

## 4.1.2 Setting Boundary Conditions

This section explains how to impose Dirichlet or Neumann boundary conditions. A Dirichlet boundary condition specifies the values that the displacement needs to take for every point *x* at the boundary ( $\Gamma_u$ ) of the problem domain (Fig. 4.1):

$$\boldsymbol{u} = \bar{\boldsymbol{u}} \quad \forall \boldsymbol{x} \in \Gamma_{\boldsymbol{u}} \tag{4.3}$$

A Neumann boundary condition imposes the value of the gradient of the solution at the boundary  $\Gamma_t$  of the problem domain (Fig. 4.1):

$$t = \sigma n = \bar{t} \quad \forall x \in \Gamma_t \tag{4.4}$$

Different ways of imposing these boundary conditions exist. A basic way is to loop over nodes or elements at the boundary and apply local values. A more advanced method consists of using the notion of the boundary of the mesh. In the following both ways are presented.

Starting with the basic approach, as mentioned, the Dirichlet boundary conditions can be applied by looping over the nodes and assigning the required values. Figure 4.2 shows a beam with a fixed support on the left side. On the right end of the beam, a load is applied. At the fixed support, the displacement has a given value. For this example, the displacements in both the *x* and the *y*-direction are set to zero. Implementing this displacement boundary condition is similar to the implementation of initial displacement conditions described above. However, in order to impose a displacement boundary condition for all time steps, the corresponding nodes need to be marked as boundary nodes using the function **blocked**. While, in order to impose a load on the right side, the nodes are not marked. The detail codes are shown as follows:

```
auto & blocked = model.getBlockedDOFs();
const auto & pos = mesh.getNodes();
```

```
UInt nb_nodes = mesh.getNbNodes();
```



Figure 4.1: Problem domain  $\Omega$  with boundary in three dimensions. The Dirchelet and the Neumann regions of the boundary are denoted with  $\Gamma_u$  and  $\Gamma_t$ , respectively.

```
for (UInt node = 0; node < nb_nodes; ++node) {
    if(Math::are_float_equal(pos(node, _x), 0)) {
        blocked(node, _x) = true; // block dof in x-direction
        blocked(node, _y) = true; // block dof in y-direction
        disp(node, _x) = 0.; // fixed displacement in x-direction
        disp(node, _y) = 0.; // fixed displacement in y-direction
    } else if (Math::are_float_equal(pos(node, _y), 0)) {
        blocked(node, _x) = false; // unblock dof in x-direction
        forces(node, _x) = 10.; // force in x-direction
    }
}</pre>
```



Figure 4.2: Beam with fixed support and load.

For the more advanced approach, one needs the notion of a boundary in the mesh. Therefore, the boundary should be created before boundary condition functors can be applied. Generally the boundary can be specified from the mesh file or the geometry. For the first case, the function createGroupsFromMeshData is called. This function can read any types of mesh data which are provided in the mesh file. If the mesh file is created with Gmsh, the function takes one input strings which is either tag\_0, tag\_1 or physical\_names. The first two tags are assigned by Gmsh to each element which shows the physical group that they belong to. In Gmsh, it is also possible to consider strings for different groups of elements. These elements can be separated by giving a string physical\_names to the function createGroupsFromMeshData:

mesh.createGroupsFromMeshData<std::string>("physical\_names").

Boundary conditions support can also be created from the geometry by calling **createBoundaryGroupFromGeometry**. This function gathers all the elements on the boundary of the geometry.

To apply the required boundary conditions, the function applyBC needs to be called on a SolidMechanicsModel. This function gets a Dirichlet or Neumann functor and a string which specifies the desired boundary on which the boundary conditions is to be applied. The functors specify the type of conditions to apply. Three built-in functors for Dirichlet exist: FlagOnly, FixedValue, and IncrementValue. The functor FlagOnly is used if a point is fixed in a given direction. Therefore, the input parameter to this functor is only the fixed direction. The FixedValue functor is used when a displacement value is applied in a fixed direction. The IncrementValue applies an increment to the displacement in a given direction. The following code shows the utilization of three functors for the top, bottom and side surface of the mesh which were already defined in the Gmsh file:

```
model.applyBC(BC::Dirichlet::FixedValue(13.0, _y), "Top");
model.applyBC(BC::Dirichlet::FlagOnly(_x), "Bottom");
model.applyBC(BC::Dirichlet::IncrementValue(13.0, _x), "Side");
```

To apply a Neumann boundary condition, the applied traction or stress should be specified before. In case of specifying the traction on the surface, the functor **FromTraction** of Neumann boundary conditions is called. Otherwise, the functor **FromStress** should be called which gets the stress tensor as an input parameter.

```
Vector<Real> surface_traction = {0., 0., 1.};
auto surface_stress(3, 3) = Matrix<Real>::eye(3);
model.applyBC(BC::Neumann::FromTraction(surface_traction), "Bottom");
model.applyBC(BC::Neumann::FromStress(surface_stress), "Top");
```

If the boundary conditions need to be removed during the simulation, a functor is called from the Neumann boundary condition to free those boundary conditions from the desired boundary.

```
model.applyBC(BC::Neumann::FreeBoundary(), "Side");
```

User specified functors can also be implemented. A full example for setting both initial and boundary conditions can be found in **examples/boundary\_conditions.cc**. The problem solved in this example is shown in Fig. 4.3. It consists of a plate that is fixed with movable supports on the left and bottom side. On the right side, a traction, which increases linearly with the number of time steps, is applied. The initial displacement and velocity in *x*-direction at all free nodes is zero and two respectively.



Figure 4.3: Plate on movable supports.

As it is mentioned in Section 2.6, node and element groups can be used to assign the boundary conditions. A generic example is given below with a Dirichlet boundary condition.

```
// create a node group
NodeGroup & node_group = mesh.createNodeGroup("nodes_fix");
/*
fill the node group with the nodes you want
```

\*/

```
// create an element group using the existing node group
mesh.createElementGroupFromNodeGroup("el_fix", "nodes_fix",
    spatial_dimension-1);
// boundary condition can be applied using the element group name
```

model.applyBC(BC::Dirichlet::FixedValue(0.0, \_x), "el\_fix");

## 4.1.3 Material Selector

If the user wants to assign different materials to different finite elements groups in **Akantu**, a material selector has to be used. By default, **Akantu** assigns the first valid material in the material file to all elements present in the model (regular continuum materials are assigned to the regular elements and cohesive materials are assigned to cohesive elements or element facets).

To assign different materials to specific elements, mesh data information such as tag information or specified physical names can be used. MeshDataMaterialSelector class uses this information to assign different materials. With the proper physical name or tag name and index, different materials can be assigned as demonstrated in the examples below.

```
auto mat_selector = std::make_shared<MeshDataMaterialSelector<std::string>>(
    "physical_names", model);
model.setMaterialSelector(mat_selector);
```

In this example the physical names specified in a GMSH geometry file will by used to match the material names in the input file.

Another example would be to use the first (tag\_0) or the second (tag\_1) tag associated to each elements in the mesh:

```
auto mat_selector = std::make_shared<MeshDataMaterialSelector<UInt>>(
    "tag_1", model, first_index);
model.setMaterialSelector(*mat_selector);
```

where first\_index (default is 1) is the value of tag\_1 that will be associated to the first material in the material input file. The following values of the tag will be associated with the following materials.

There are four different material selectors pre-defined in **Akantu**. MaterialSelector and DefaultMaterialSelector is used to assign a material to regular elements by default. For the regular elements, as in the example above, MeshDataMaterialSelector can be used to assign different materials to different elements.

Apart from the **Akantu**'s default material selectors, users can always develop their own classes in the main code to tackle various multi-material assignment situations.

For cohesive material, **Akantu** has a pre-defined material selector to assign the first cohesive material by default to the cohesive elements which is called DefaultMaterialCohesiveSelector and it inherits its properties from DefaultMaterialSelector. Multiple cohesive materials can be assigned using mesh data information (for more details, see 4.1.4).

### 4.1.4 Insertion of Cohesive Elements

Cohesive elements are currently compatible only with static simulation and dynamic simulation with an explicit time integration scheme (see section 4.3.2). They do not have to be inserted when the mesh is generated (intrinsic) but can be added during the simulation (extrinsic). At any time during the simulation, it is possible to access the following energies with the relative function:

Real Ed = model.getEnergy("dissipated");
Real Er = model.getEnergy("reversible");
Real Ec = model.getEnergy("contact");

A new model have to be call in a very similar way that the solid mechanics model:

#### **Extrinsic approach**

During the simulation, stress has to be checked along each facet in order to insert cohesive elements where the stress criterion is reached. This check is performed by calling the method checkCohesiveStress, as example before each step resolution:

```
model.checkCohesiveStress();
model.solveStep();
```

The area where stresses are checked and cohesive elements inserted can be limited using the method setLimit on the CohesiveInserter during initialization. As example, to limit insertion in the range [-1.5, 1.5] in the *x* direction:

```
auto & inserter = model.getElementInserter();
inserter.setLimit(_x, -1.5, 1.5);
```

Additional restrictions with respect to y and z directions can be added as well.

#### **Intrinsic approach**

Intrinsic cohesive elements are inserted in the mesh with the method initFull. Similarly, the range of insertion can be limited with setLimit before the initFull call.

In both cases extrinsic and intrinsic the insertion can be restricted to group of elements. To do so the list of groups should be specified in the input file.

```
model solid_mechanics_model_cohesive [
   cohesive_inserter [
      cohesive_surfaces = [surface1, surface2, ...]
]
material cohesive_linear [
   name = insertion
   beta = 1
   G_c = 10
   sigma_c = 1e6
]
]
```

### 4.2 Static Analysis

The SolidMechanicsModel class can handle different analysis methods, the first one being presented is the static case. In this case, the equation to solve is

$$Ku = f_{\text{ext}} \tag{4.5}$$

where *K* is the global stiffness matrix, *u* the displacement vector and  $f_{\text{ext}}$  the vector of external forces applied to the system.

To solve such a problem, the static solver of the SolidMechanicsModel object is used. First, a model has to be created and initialized. To create the model, a mesh (which can be read from a file) is needed, as explained in Section 2.4. Once an instance of a SolidMechanicsModel

is obtained, the easiest way to initialize it is to use the initFull method by giving the SolidMechanicsModelOptions. These options specify the type of analysis to be performed and whether the materials should be initialized with initMaterials or not.

```
SolidMechanicsModel model(mesh);
model.initFull(_analysis_method = _static);
```

Here, a static analysis is chosen by passing the argument \_static to the method. By default, the Boolean for no initialization of the materials is set to false, so that they are initialized during the initFull. The method initFull also initializes all appropriate vectors to zero. Once the model is created and initialized, the boundary conditions can be set as explained in Section 4.1.2. Boundary conditions will prescribe the external forces for some free degrees of freedom  $f_{ext}$  and displacements for some others. At this point of the analysis, the function solveStep can be called:

```
model.solveStep<_scm_newton_raphson_tangent_modified, _scc_residual>(1e-4, 1);
```

This function is templated by the solving method and the convergence criterion and takes two arguments: the tolerance and the maximum number of iterations (100 by default), which are  $1 \times 10^{-4}$  and 1 for this example. The modified Newton-Raphson method is chosen to solve the system. In this method, the equilibrium equation (4.5) is modified in order to apply a Newton-Raphson convergence algorithm:

$$\boldsymbol{K}^{i+1}\delta\boldsymbol{u}^{i+1} = \boldsymbol{r} \tag{4.6}$$

$$= f_{\text{ext}} - f_{\text{int}} \tag{4.7}$$

$$= f_{\text{ext}} - K^{i} u^{i}$$

$$u^{i+1} = u^{i} + \delta u^{i+1} ,$$

$$(4.8)$$

where  $\delta u$  is the increment of displacement to be added from one iteration to the other, and *i* is the Newton-Raphson iteration counter. By invoking the solveStep method in the first step, the global stiffness matrix *K* from Equation (4.5) is automatically assembled. A Newton-Raphson iteration is subsequently started, *K* is updated according to the displacement computed at the previous iteration and one loops until the forces are balanced (\_scc\_residual), *i.e.*,  $||r|| < _scc_residual$ . One can also iterate until the increment of displacement is zero (\_scc\_increment) which also means that the equilibrium is found. For a linear elastic problem, the solution is obtained in one iteration and therefore the maximum number of iterations can be set to one. But for a non-linear case, one needs to iterate as long as the norm of the residual exceeds the tolerance threshold and therefore the maximum number of iterations has to be higher, e.g. 100:

model.solveStep<\_scm\_newton\_raphson\_tangent\_modified,\_scc\_residual>(1e-4, 100)

At the end of the analysis, the final solution is stored in the **displacement** vector. A full example of how to solve a static problem is presented in the code **examples/static/static.cc**. This example is composed of a 2D plate of steel, blocked with rollers on the left and bottom sides as shown in Figure 4.4. The nodes from the right side of the sample are displaced by 0.01% of the length of the plate.

The results of this analysis is depicted in Figure 4.5.

### 4.2.1 Static implicit analysis with dynamic insertion of cohesive elements

In order to solve problems with the extrinsic cohesive method in the static implicit solution scheme, the function solveStepCohesive has to be used:

```
model.solveStepCohesive<_scm_newton_raphson_tangent, _scc_increment>(1e-13,
error, 25, false, 1e5, true);
```



Figure 4.4: Numerical setup



Figure 4.5: Solution of the static analysis. Left: the initial condition, right: the solution (deformation magnified 50 times)

in which the arguments are: tolerance, error, max\_iteration, load\_reduction, tol\_increase\_factor, do\_not\_factorize. This function, first applies the Newton-Raphson procedure to solve the problem. Then, it calls the method checkCohesiveStress to check if cohesive elements have to be inserted. Since the approach is implicit, only one element is added, the most stressed one (see Section 4.1.4). After insertion, the Newton-Raphson procedure is applied again to solve the same incremental loading step, with the new inserted cohesive element. The procedure loops in this way since no new cohesive elements have to be inserted. At that point, the solution is saved, and the simulation can advance to the next incremental loading step. In case the convergence is not reached, the obtained solution is not saved and the simulation return to the main file with the error given by the solution saved in the argument of the function *error*. In this way, the user can intervene in the simulation in order to find anyhow convergence. A possibility is, for instance, to reduce the last incremental loading step. The variable load\_reduction can be used to identify if the load has been already reduced or not. At the same time, with the variable tol\_increase\_factor it is possible to increase the tolerance by a factor defined by the user in the main file, in order to accept a solution even with an error bigger than the tolerance set at the beginning. It is possible to increase the tolerance only in the phase of loading reduction, i.e., when load\_reduction = true. A not converged solution is never saved. In case the convergence is not reached even after the loading reduction procedure, the displacement field is not updated and remains the one of the last converged incremental steps. Also, cohesive elements are inserted only if convergence is reached. An example of the extrinsic cohesive method in the static implicit solution scheme is presented in examples/cohesive\_element/cohesive\_extrinsic\_implicit.

## 4.3 Dynamic Methods

Different ways to solve the equations of motion are implemented in the solid mechanics model. The complete equations that should be solved are:

$$M\ddot{u} + C\dot{u} + Ku = f_{\text{ext}} \,, \tag{4.9}$$

where *M*, *C* and *K* are the mass, damping and stiffness matrices, respectively.

In the previous section, it has already been discussed how to solve this equation in the static case, where  $\ddot{u} = \dot{u} = 0$ . Here the method to solve this equation in the general case will be presented. For this purpose, a time discretization has to be specified. The most common discretization method in solid mechanics is the Newmark- $\beta$  method, which is also the default in **Akantu**.

For the Newmark- $\beta$  method, (4.9) becomes a system of three equations (see [5] [6] for more details):

$$M\ddot{u}_{n+1} + C\dot{u}_{n+1} + Ku_{n+1} = f_{\text{ext } n+1}$$
(4.10)

$$\boldsymbol{u}_{n+1} = \boldsymbol{u}_n + (1-\alpha)\,\Delta t \dot{\boldsymbol{u}}_n + \alpha \Delta t \dot{\boldsymbol{u}}_{n+1} + \left(\frac{1}{2} - \alpha\right) \Delta t^2 \ddot{\boldsymbol{u}}_n \tag{4.11}$$

$$\dot{\boldsymbol{u}}_{n+1} = \dot{\boldsymbol{u}}_n + (1-\beta)\,\Delta t \ddot{\boldsymbol{u}}_n + \beta \Delta t \ddot{\boldsymbol{u}}_{n+1} \tag{4.12}$$

In these new equations,  $\ddot{u}_n$ ,  $\dot{u}_n$  and  $u_n$  are the approximations of  $\ddot{u}(t_n)$ ,  $\dot{u}(t_n)$  and  $u(t_n)$ . Equation (4.10) is the equation of motion discretized in space (finite-element discretization), and equations (4.11) and (4.12) are discretized in both space and time (Newmark discretization). The  $\alpha$  and  $\beta$  parameters determine the stability and the accuracy of the algorithm. Classical values for  $\alpha$  and  $\beta$  are usually  $\beta = 1/2$  for no numerical damping and  $0 < \alpha < 1/2$ .

α	Method ( $\beta = 1/2$ )	Туре
0	central difference	explicit
1/6	Fox-Goodwin (royal road)	implicit
1/3	Linear acceleration	implicit
1/2	Average acceleration (trapezoidal rule)	implicit

The solution of this system of equations, (4.10)-(4.12) is split into a predictor and a corrector system of equations. Moreover, in the case of a non-linear equations, an iterative algorithm such as the Newton-Raphson method is applied. The system of equations can be written as:

1. Predictor:

$$u_{n+1}^{0} = u_{n} + \Delta t \dot{u}_{n} + \frac{\Delta t^{2}}{2} \ddot{u}_{n}$$
(4.13)

$$\dot{\boldsymbol{u}}_{n+1}^0 = \dot{\boldsymbol{u}}_n + \Delta t \ddot{\boldsymbol{u}}_n \tag{4.14}$$

$$\ddot{\boldsymbol{u}}_{n+1}^0 = \ddot{\boldsymbol{u}}_n \tag{4.15}$$

2. Solve:

$$\left(cM + dC + eK_{n+1}^{i}\right)w = f_{\text{ext}\,n+1} - f_{\text{int}\,n+1}^{i} - C\dot{u}_{n+1}^{i} - M\ddot{u}_{n+1}^{i} = r_{n+1}^{i}$$
(4.16)

3. Corrector:

$$\ddot{u}_{n+1}^{i+1} = \ddot{u}_{n+1}^{i} + cw \tag{4.17}$$

$$\dot{u}_{n+1}^{i+1} = \dot{u}_{n+1}^{i} + dw \tag{4.18}$$

$$u_{n+1}^{i+1} = u_{n+1}^{i} + ew ag{4.19}$$

	w	е	d	С
in acceleration	δü	$\alpha\beta\Delta t^2$	$\beta \Delta t$	1
in velocity	δü	$\alpha \Delta t$	1	$\frac{1}{\beta \Delta t}$
in displacement	би	1	$\frac{1}{\alpha \Delta t}$	$\frac{1}{\alpha\beta\Delta t^2}$

where *i* is the Newton-Raphson iteration counter and *c*, *d* and *e* are parameters depending on the method used to solve the equations

#### 4.3.1 Implicit Time Integration

To solve a problem with an implicit time integration scheme, first a SolidMechanicsModel object has to be created and initialized. Then the initial and boundary conditions have to be set. Everything is similar to the example in the static case (Section 4.2), however, in this case the implicit dynamic scheme is selected at the initialization of the model.

```
SolidMechanicsModel model(mesh);
model.initFull(_analysis_method = _implicit_dynamic);
/*Boundary conditions see Section ~ 4.1.2 */
```

Because a dynamic simulation is conducted, an integration time step  $\Delta t$  has to be specified. In the case of implicit simulations, **Akantu** implements a trapezoidal rule by default. That is to say  $\alpha = 1/2$  and  $\beta = 1/2$  which is unconditionally stable. Therefore the value of the time step can be chosen arbitrarily within reason.

```
model.setTimeStep(time_step);
```

Since the system has to be solved for a given amount of time steps, the method solveStep(), (which has already been used in the static example in Section 4.2), is called inside a time loop:

```
/// time loop
Real time = 0.;
auto & solver = model.getNonLinearSolver();
solver.set("max_iterations", 100);
solver.set("threshold", 1e-12);
solver.set("convergence_type", _scc_solution);
for (UInt s = 1; time <max_time; ++s, time += time_step) {
    model.solveStep();
}</pre>
```

An example of solid mechanics with an implicit time integration scheme is presented in **examples/implicit/implicit\_dynamic.cc**. This example consists of a 3D beam of  $10 \text{ m} \times 1 \text{ m} \times 1 \text{ m}$  blocked on one side and is on a roller on the other side. A constant force of 5 kN is applied in its middle. Figure 4.6 presents the geometry of this case. The material used is a fictitious linear elastic material with a density of  $1000 \text{ kg m}^{-3}$ , a Young's Modulus of 120 MPa and Poisson's ratio of 0.3. These values were chosen to simplify the analytical solution.

An approximation of the dynamic response of the middle point of the beam is given by:

$$u\left(\frac{L}{2},t\right) = \frac{1}{\pi^4} \left(1 - \cos\left(\pi^2 t\right) + \frac{1}{81} \left(1 - \cos\left(3^2 \pi^2 t\right)\right) + \frac{1}{625} \left(1 - \cos\left(5^2 \pi^2 t\right)\right)\right)$$
(4.20)

Figure 4.7 presents the deformed beam at 3 different times during the simulation: time steps 0, 1000 and 2000.


Figure 4.6: Numerical setup





### 4.3.2 Explicit Time Integration

The explicit dynamic time integration scheme is based on the Newmark- $\beta$  scheme with  $\alpha = 0$  (see equations 4.10-4.12). In **Akantu**,  $\beta$  is defaults to  $\beta = 1/2$ , see section 4.3.

The initialization of the simulation is similar to the static and implicit dynamic version. The model is created from the SolidMechanicsModel class. In the initialization, the explicit scheme is selected using the \_explicit\_lumped\_mass constant.

```
SolidMechanicsModel model(mesh);
model.initFull(_analysis_method = _explicit_lumped_mass);
```

**Note:** Writing model.initFull() or model.initFull(); is equivalent to use the \_explicit\_lumped\_mass keyword, as this is the default case.

The explicit time integration scheme implemented in **Akantu** uses a lumped mass matrix M (reducing the computational cost). This matrix is assembled by distributing the mass of each element onto its nodes. The resulting M is therefore a diagonal matrix stored in the **mass** vector of the model.

The explicit integration scheme is conditionally stable. The time step has to be smaller than the stable time step which is obtained in **Akantu** as follows:

critical\_time\_step = model.getStableTimeStep();

The stable time step corresponds to the time the fastest wave (the compressive wave) needs to travel the characteristic length of the mesh:

$$\Delta t_{\rm crit} = \frac{\Delta x}{c} \tag{4.21}$$

where  $\Delta x$  is a characteristic length (*e.g.*, the inradius in the case of linear triangle element) and *c* is the celerity of the fastest wave in the material. It is generally the compressive wave of celerity  $c = \sqrt{\frac{2\mu + \lambda}{\rho}}$ ,  $\mu$  and  $\lambda$  are the first and second Lame's coefficients and  $\rho$  is the density. However, it is recommended to impose a time step that is smaller than the stable time step, for instance, by multiplying the stable time step by a safety factor smaller than one.

```
const Real safety_time_factor = 0.8;
Real applied_time_step = critical_time_step * safety_time_factor;
model.setTimeStep(applied_time_step);
```

The initial displacement and velocity fields are, by default, equal to zero if not given specifically by the user (see 4.1.1).

Like in implicit dynamics, a time loop is used in which the displacement, velocity and acceleration fields are updated at each time step. The values of these fields are obtained from the Newmark- $\beta$  equations with  $\beta = 1/2$  and  $\alpha = 0$ . In **Akantu** these computations at each time step are invoked by calling the function solveStep:

```
for (UInt s = 1; (s-1)*applied_time_step < total_time; ++s) {
  model.solveStep();
}</pre>
```

The method solveStep wraps the four following functions:

. .

• model.explicitPred() allows to compute the displacement field at t + 1 and a part of the velocity field at t + 1, denoted by  $\dot{u}_{n+1}^{p}$ , which will be used later in the method model.explicitCorr(). The equations are:

$$\boldsymbol{u}_{n+1} = \boldsymbol{u}_n + \Delta t \dot{\boldsymbol{u}}_n + \frac{\Delta t^2}{2} \ddot{\boldsymbol{u}}_n \tag{4.22}$$

$$\dot{\boldsymbol{u}}^{\mathbf{p}}_{n+1} = \dot{\boldsymbol{u}}_n + \Delta t \ddot{\boldsymbol{u}}_n \tag{4.23}$$

 model.updateResidual() and model.updateAcceleration() compute the acceleration increment δ*ü*:

$$\left(M + \frac{1}{2}\Delta tC\right)\delta\ddot{u} = f_{\text{ext}} - f_{\text{int}\,n+1} - C\dot{u}^{p}_{n+1} - M\ddot{u}_{n}$$
(4.24)

**Note:** The internal force  $f_{int n+1}$  is computed from the displacement  $u_{n+1}$  based on the constitutive *law*.

• model.explicitCorr() computes the velocity and acceleration fields at *t* + 1:

$$\dot{\boldsymbol{u}}_{n+1} = \dot{\boldsymbol{u}}^{\mathbf{p}}_{n+1} + \frac{\Delta t}{2} \delta \ddot{\boldsymbol{u}} \tag{4.25}$$

$$\ddot{\boldsymbol{u}}_{n+1} = \ddot{\boldsymbol{u}}_n + \delta \ddot{\boldsymbol{u}} \tag{4.26}$$

The use of an explicit time integration scheme is illustrated by the example: examples/explicit/explicit\_dynamic.cc

This example models the propagation of a wave in a steel beam. The beam and the applied displacement in the *x* direction are shown in Figure 4.8.



Figure 4.8: Numerical setup

The length and height of the beam are L = 10 m and h = 1 m, respectively. The material is linear elastic, homogeneous and isotropic (density: 7800 kg m<sup>-3</sup>, Young's modulus: 210 GPa and Poisson's ratio: 0.3). The imposed displacement follow a Gaussian function with a maximum amplitude of A = 0.01 m. The potential, kinetic and total energies are computed. The safety factor is equal to 0.8.

# 4.4 Constitutive Laws

In order to compute an element's response to deformation, one needs to use an appropriate constitutive relationship. The constitutive law is used to compute the element's stresses from the element's strains.

In the finite-element discretization, the constitutive formulation is applied to every quadrature point of each element. When the implicit formulation is used, the tangent matrix has to be computed.

The chosen materials for the simulation have to be specified in the mesh file or, as an alternative, they can be assigned using the element\_material vector. For every material assigned to the problem one has to specify the material characteristics (constitutive behavior and material properties) using the text input file (see 7.1.3).

In order to conveniently store values at each quadrature in a material point **Akantu** provides a special data structure, the InternalField. The internal fields are inheriting from the ElementTypeMapArray. Furthermore, it provides several functions for initialization, auto-resizing and auto removal of quadrature points.

Sometimes it is also desired to generate random distributions of internal parameters. An example might be the critical stress at which the material fails. To generate such a field, in the text input file, a random quantity needs be added to the base value:

```
sigma_c = base
sigma_c = base uniform [min, max]
sigma_c = base weibull [\lambda, m]
```

All parameters are real numbers. For the uniform distribution, minimum and maximum values have to be specified. Random parameters are defined as a *base* value to which we add a random number that follows the chosen distribution.

The *Uniform* distribution is gives a random values between in [*min, max*). The *Weibull* distribution is characterized by the following cumulative distribution function:

$$F(x) = 1 - e^{-(x/\lambda)^m}$$
(4.27)

which depends on *m* and  $\lambda$ , which are the shape parameter and the scale parameter. These random distributions are different each time the code is executed. In order to obtain always the same one, it possible to manually set the *seed* that is the number from which these pseudo-random distributions are created. This can be done by adding the following line to the input file *outside* the material parameters environments:

seed = 1.0

where the value 1.0 can be substituted with any number. Currently **Akantu** can reproduce always the same distribution when the seed is specified *only* in serial. The value of the *seed* can be also specified directly in the code (for instance in the main file) with the command:

```
RandGenerator<Real>:: seed(1.0)
```

The same command, with empty brackets, can be used to check the value of the *seed* used in the simulation.

The following sections describe the constitutive models implemented in **Akantu**. In Appendix B a summary of the parameters for all materials of **Akantu** is provided.

### 4.4.1 Elasticity

The elastic law is a commonly used constitutive relationship that can be used for a wide range of engineering materials (*e.g.*, metals, concrete, rock, wood, glass, rubber, etc.) provided that the strains remain small (*i.e.*, small deformation and stress lower than yield strength).

The elastic laws are often expressed as  $\sigma = C : \varepsilon$  with where  $\sigma$  is the Cauchy stress tensor,  $\varepsilon$  represents the infinitesimal strain tensor and *C* is the elastic modulus tensor.

#### Linear isotropic (B.1)

The linear isotropic elastic behavior is described by Hooke's law, which states that the stress is linearly proportional to the applied strain (material behaves like an ideal spring), as illustrated in Figure 4.9. The equation that relates the strains to the displacements is: point) from the



Figure 4.9: (a) Stress-strain curve for elastic material and (b) schematic representation of Hooke's law, denoted as a spring.

displacements as follows:

$$\boldsymbol{\varepsilon} = \frac{1}{2} \left[ \nabla_0 \boldsymbol{u} + \nabla_0 \boldsymbol{u}^T \right] \tag{4.28}$$

where  $\varepsilon$  represents the infinitesimal strain tensor,  $\nabla_0 u$  the displacement gradient tensor according to the initial configuration. The constitutive equation for isotropic homogeneous media can be expressed as:

$$\sigma = \lambda \operatorname{tr}(\varepsilon)I + 2\mu\varepsilon \tag{4.29}$$

where  $\sigma$  is the Cauchy stress tensor ( $\lambda$  and  $\mu$  are the first and second Lame's coefficients).

In Voigt notation this correspond to

$$\begin{bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{33} \\ \sigma_{23} \\ \sigma_{13} \\ \sigma_{12} \end{bmatrix} = \frac{E}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1-\nu & \nu & \nu & 0 & 0 & 0 \\ \nu & 1-\nu & \nu & 0 & 0 & 0 \\ \nu & \nu & 1-\nu & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1-2\nu}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1-2\nu}{2} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1-2\nu}{2} \end{bmatrix} \begin{bmatrix} \varepsilon_{11} \\ \varepsilon_{22} \\ \varepsilon_{33} \\ 2\varepsilon_{23} \\ 2\varepsilon_{13} \\ 2\varepsilon_{12} \end{bmatrix}$$
(4.30)

#### Linear anisotropic (B.2)

This formulation is not sufficient to represent all elastic material behavior. Some materials have characteristic orientation that have to be taken into account. To represent this anisotropy a more general stress-strain law has to be used. For this we define the elastic modulus tensor as follow:

$$\begin{bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{33} \\ \sigma_{23} \\ \sigma_{13} \\ \sigma_{12} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} & c_{15} & c_{16} \\ c_{21} & c_{22} & c_{23} & c_{24} & c_{25} & c_{26} \\ c_{31} & c_{32} & c_{33} & c_{34} & c_{35} & c_{36} \\ c_{41} & c_{42} & c_{43} & c_{44} & c_{45} & c_{46} \\ c_{51} & c_{52} & c_{53} & c_{54} & c_{55} & c_{56} \\ c_{61} & c_{62} & c_{63} & c_{64} & c_{65} & c_{66} \end{bmatrix} \begin{bmatrix} \varepsilon_{11} \\ \varepsilon_{22} \\ \varepsilon_{33} \\ 2\varepsilon_{23} \\ 2\varepsilon_{13} \\ 2\varepsilon_{12} \end{bmatrix}$$

$$(4.31)$$

Figure 4.10: Material basis

e1

To simplify the writing of input files the *C* tensor is expressed in the material basis. And this basis as to be given too. This basis  $\Omega_{mat} = \{n_1, n_2, n_3\}$  is used to define the rotation  $R_{ij} = n_j \cdot e_i$ . And *C* can be rotated in the global basis  $\Omega = \{e_1, e_2, e_3\}$  as follow:

$$C_{\Omega} = R_{1}C_{\Omega_{mat}}R_{2}$$

$$(4.32)$$

$$R_{1} = \begin{bmatrix} R_{11}R_{11} & R_{12}R_{12} & R_{13}R_{13} & R_{12}R_{13} & R_{11}R_{13} & R_{11}R_{12} \\ R_{21}R_{21} & R_{22}R_{22} & R_{23}R_{23} & R_{22}R_{23} & R_{21}R_{22} \\ R_{31}R_{31} & R_{32}R_{32} & R_{33}R_{33} & R_{32}R_{33} & R_{31}R_{33} \\ R_{21}R_{31} & R_{22}R_{32} & R_{23}R_{33} & R_{22}R_{33} & R_{21}R_{33} & R_{11}R_{32} \\ R_{11}R_{31} & R_{12}R_{32} & R_{13}R_{33} & R_{12}R_{33} & R_{11}R_{32} \\ R_{11}R_{21} & R_{12}R_{22} & R_{13}R_{23} & R_{12}R_{23} & R_{11}R_{23} \\ R_{11}R_{21} & R_{12}R_{22} & R_{32}R_{32} & R_{22}R_{32} & R_{12}R_{23} & R_{11}R_{22} \end{bmatrix}$$

$$R_{2} = \begin{bmatrix} R_{11}R_{11} & R_{21}R_{21} & R_{31}R_{31} & R_{21}R_{31} & R_{11}R_{31} & R_{11}R_{21} \\ R_{12}R_{12} & R_{22}R_{22} & R_{32}R_{32} & R_{22}R_{32} & R_{12}R_{22} \\ R_{13}R_{13} & R_{23}R_{23} & R_{33}R_{33} & R_{23}R_{33} & R_{13}R_{33} & R_{13}R_{23} \\ R_{12}R_{13} & R_{22}R_{23} & R_{32}R_{33} & R_{12}R_{33} & R_{12}R_{23} \\ R_{11}R_{13} & R_{21}R_{23} & R_{31}R_{33} & R_{21}R_{33} & R_{12}R_{33} \\ R_{11}R_{13} & R_{21}R_{23} & R_{31}R_{33} & R_{21}R_{33} & R_{11}R_{23} \\ R_{11}R_{12} & R_{21}R_{22} & R_{31}R_{32} & R_{21}R_{33} & R_{11}R_{23} \\ R_{11}R_{12} & R_{21}R_{22} & R_{31}R_{33} & R_{21}R_{33} & R_{11}R_{23} \\ R_{11}R_{12} & R_{21}R_{22} & R_{31}R_{32} & R_{21}R_{33} & R_{11}R_{23} \\ R_{11}R_{12} & R_{21}R_{22} & R_{31}R_{33} & R_{21}R_{33} & R_{11}R_{23} \\ R_{11}R_{12} & R_{21}R_{22} & R_{31}R_{32} & R_{11}R_{32} & R_{11}R_{22} \\ R_{13}R_{13} & R_{21}R_{22} & R_{31}R_{32} & R_{21}R_{32} & R_{11}R_{22} \\ R_{13}R_{11}R_{12} & R_{21}R_{22} & R_{31}R_{32} & R_{11}R_{32} \\ R_{11}R_{12} & R_{21}R_{22} & R_{31}R_{32} & R_{21}R_{32} & R_{11}R_{22} \\ R_{13}R_{11}R_{12} & R_{21}R_{22} & R_{31}R_{32} & R_{11}R_{32} \\ R_{13}R_{13} & R_{21}R_{22} & R_{11}R_{22} \\ R_{13}R_{11}R_{12} & R_{11}R_{22} & R_{11}R_{22} \\ R_{13}R_{11}R_{12} & R_{$$

#### Linear orthotropic (B.3)

A particular case of anisotropy is when the material basis is orthogonal in which case the elastic modulus tensor can be simplified and rewritten in terms of 9 independents material parameters.

$$\begin{bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{33} \\ \sigma_{23} \\ \sigma_{13} \\ \sigma_{12} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} & 0 & 0 & 0 \\ & c_{22} & c_{23} & 0 & 0 & 0 \\ & & c_{33} & 0 & 0 & 0 \\ & & & c_{44} & 0 & 0 \\ & & & & c_{55} & 0 \\ & & & & & c_{66} \end{bmatrix} \begin{bmatrix} \varepsilon_{11} \\ \varepsilon_{22} \\ \varepsilon_{33} \\ 2\varepsilon_{23} \\ 2\varepsilon_{13} \\ 2\varepsilon_{12} \end{bmatrix}$$
(4.36)

$$c_{11} = E_1(1 - \nu_{23}\nu_{32})\Gamma \qquad c_{22} = E_2(1 - \nu_{13}\nu_{31})\Gamma \qquad c_{33} = E_3(1 - \nu_{12}\nu_{21})\Gamma$$
(4.37)

$$c_{12} = E_1(\nu_{21} - \nu_{31}\nu_{23})\Gamma = E_2(\nu_{12} - \nu_{32}\nu_{13})\Gamma$$
(4.38)

$$c_{13} = E_1(\nu_{31} - \nu_{21}\nu_{32})\Gamma = E_2(\nu_{13} - \nu_{21}\nu_{23})\Gamma$$
(4.39)

$$c_{23} = E_2(v_{32} - v_{12}v_{31})\Gamma = E_3(v_{23} - v_{21}v_{13})\Gamma$$
(4.40)

$$c_{44} = \mu_{23} \qquad c_{55} = \mu_{13} \qquad c_{66} = \mu_{12} \tag{4.41}$$

$$\Gamma = \frac{1}{1 - \nu_{12}\nu_{21} - \nu_{13}\nu_{31} - \nu_{32}\nu_{23} - 2\nu_{21}\nu_{32}\nu_{13}}$$
(4.42)

The Poisson ratios follow the rule  $v_{ij} = v_{ji}E_i/E_j$ .

# 4.4.2 Neo-Hookean (B.4)

The hyperelastic Neo-Hookean constitutive law results from an extension of the linear elastic relationship (Hooke's Law) for large deformation. Thus, the model predicts nonlinear stress-strain behavior for bodies undergoing large deformations.



Figure 4.11: Neo-hookean Stress-strain curve.

As illustrated in Figure 4.11, the behavior is initially linear and the mechanical behavior is very close to the corresponding linear elastic material. This constitutive relationship, which accounts for compressibility, is a modified version of the one proposed by Ronald Rivlin [7].

The strain energy stored in the material is given by:

$$\Psi(\mathbf{C}) = \frac{1}{2}\lambda_0 \left(\ln J\right)^2 - \mu_0 \ln J + \frac{1}{2}\mu_0 \left(\operatorname{tr}(\mathbf{C}) - 3\right)$$
(4.43)

where  $\lambda_0$  and  $\mu_0$  are, respectively, Lamé's first parameter and the shear modulus at the initial configuration. *J* is the jacobian of the deformation gradient ( $F = \nabla_X x$ ):  $J = \det(F)$ . Finally *C* is the right Cauchy-Green deformation tensor.

Since this kind of material is used for large deformation problems, a finite deformation framework should be used. Therefore, the Cauchy stress ( $\sigma$ ) should be computed through the second Piola-Kirchhoff stress tensor *S*:

$$\boldsymbol{\sigma} = \frac{1}{J} \boldsymbol{F} \boldsymbol{S} \boldsymbol{F}^{T} \tag{4.44}$$

Finally the second Piola-Kirchhoff stress tensor is given by:

$$S = 2\frac{\partial\Psi}{\partial C} = \lambda_0 \ln J C^{-1} + \mu_0 \left(I - C^{-1}\right)$$
(4.45)

The parameters to indicate in the material file are the same as those for the elastic case: E (Young's modulus), nu (Poisson's ratio).

#### 4.4.3 Visco-Elasticity (B.5)

Visco-elasticity is characterized by strain rate dependent behavior. Moreover, when such a material undergoes a deformation it dissipates energy. This dissipation results in a hysteresis loop in the stress-strain curve at every loading cycle (see Figure 4.12a). In principle, it can be applied to many materials, since all materials exhibit a visco-elastic behavior if subjected to particular conditions (such as high temperatures). The standard rheological linear solid model (see Sections 10.2 and



Figure 4.12: (a) Characteristic stress-strain behavior of a visco-elastic material with hysteresis loop and (b) schematic representation of the standard rheological linear solid visco-elastic model.

10.3 of [8]) has been implemented in **Akantu**. This model results from the combination of a spring mounted in parallel with a spring and a dashpot connected in series, as illustrated in Figure 4.12b. The advantage of this model is that it allows to account for creep or stress relaxation. The equation that relates the stress to the strain is (in 1D):

$$\frac{d\varepsilon(t)}{dt} = (E + E_V)^{-1} \cdot \left[\frac{d\sigma(t)}{dt} + \frac{E_V}{\eta}\sigma(t) - \frac{EE_V}{\eta}\varepsilon(t)\right]$$
(4.46)

where  $\eta$  is the viscosity. The equilibrium condition is unique and is attained in the limit, as  $t \to \infty$ . At this stage, the response is elastic and depends on the Young's modulus *E*. The mandatory parameters for the material file are the following: **rho** (density), **E** (Young's modulus), **nu** (Poisson's ratio), **Plane\_Stress** (if set to zero plane strain, otherwise plane stress), **eta** (dashpot viscosity) and **Ev** (stiffness of the viscous element). Note that the current standard linear solid model is applied only on the deviatoric part of the strain tensor. The spheric part of the strain tensor affects the stress tensor like an linear elastic material.

#### 4.4.4 Small-Deformation Plasticity (B.6)

The small-deformation plasticity is a simple plasticity material formulation which accounts for the additive decomposition of strain into elastic and plastic strain components. This formulation is applicable to infinitesimal deformation where the additive decomposition of the strain is a valid approximation. In this formulation, plastic strain is a shearing process where hydrostatic stress has no contribution to plasticity and consequently plasticity does not lead to volume change. Figure 4.13 shows the linear strain hardening elasto-plastic behavior according to the additive decomposition of strain into the elastic and plastic parts in infinitesimal deformation as

$$\varepsilon = \varepsilon^{e} + \varepsilon^{p}$$

$$\sigma = 2G(\varepsilon^{e}) + \lambda \operatorname{tr}(\varepsilon^{e})I$$
(4.47)
(4.48)



Figure 4.13: Stress-strain curve for the small-deformation plasticity with linear isotropic hardening.

In this class, the von Mises yield criterion is used. In the von Mises yield criterion, the yield is independent of the hydrostatic stress. Other yielding criteria such as Tresca and Gurson can be easily implemented in this class as well.

In the von Mises yield criterion, the hydrostatic stresses have no effect on the plasticity and consequently the yielding occurs when a critical elastic shear energy is achieved.

$$f = \sigma_{\rm eff} - \sigma_y = \left(\frac{3}{2}\sigma^{\rm tr} : \sigma^{\rm tr}\right)^{\frac{1}{2}} - \sigma_y(\varepsilon^p) \tag{4.49}$$

f < 0 Elastic deformation, f = 0 Plastic deformation (4.50)

where  $\sigma_y$  is the yield strength of the material which can be function of plastic strain in case of hardening type of materials and  $\sigma^{tr}$  is the deviatoric part of stress given by

$$\sigma^{\rm tr} = \sigma - \frac{1}{3} {\rm tr}(\sigma) I \tag{4.51}$$

After yielding (f = 0), the normality hypothesis of plasticity determines the direction of plastic flow which is normal to the tangent to the yielding surface at the load point. Then, the tensorial form of the plastic constitutive equation using the von Mises yielding criterion (see equation 4.34) may be written as

$$\Delta \varepsilon^{p} = \Delta p \frac{\partial f}{\partial \sigma} = \frac{3}{2} \Delta p \frac{\sigma^{\text{tr}}}{\sigma_{\text{eff}}}$$
(4.52)

In these expressions, the direction of the plastic strain increment (or equivalently, plastic strain rate) is given by  $\frac{\sigma^{tr}}{\sigma_{eff}}$  while the magnitude is defined by the plastic multiplier  $\Delta p$ . This can be obtained using the *consistency condition* which impose the requirement for the load point to remain on the yielding surface in the plastic regime.

Here, we summarize the implementation procedures for the small-deformation plasticity with linear isotropic hardening:

1. Compute the trial stress:

$$\sigma^{\rm tr} = \sigma_t + 2G\Delta\varepsilon + \lambda {\rm tr}(\Delta\varepsilon)I \tag{4.53}$$

2. Check the Yielding criteria:

$$f = \left(\frac{3}{2}\sigma^{\text{tr}} : \sigma^{\text{tr}}\right)^{1/2} - \sigma_y(\varepsilon^p)$$
(4.54)

3. Compute the Plastic multiplier:

$$d\Delta p = \frac{\sigma_{eff}^{tr} - 3G\Delta P^{(k)} - \sigma_y^{(k)}}{3G + h}$$

$$\tag{4.55}$$

$$\Delta p^{(k+1)} = \Delta p^{(k)} + d\Delta p \tag{4.56}$$

$$\sigma_y^{(k+1)} = (\sigma_y)_t + h\Delta p \tag{4.57}$$

4. Compute the plastic strain increment:

$$\Delta \epsilon^{p} = \frac{3}{2} \Delta p \frac{\sigma^{\text{tr}}}{\sigma_{\text{eff}}}$$
(4.58)

5. Compute the stress increment:

$$\Delta \sigma = 2G(\Delta \varepsilon - \Delta \varepsilon^p) + \lambda \operatorname{tr}(\Delta \varepsilon - \Delta \varepsilon^p) I \tag{4.59}$$

6. Update the variables:

$$\varepsilon^p = \varepsilon^p_t + \Delta \varepsilon^p \tag{4.60}$$

$$\sigma = \sigma_t + \Delta \sigma \tag{4.61}$$

We use an implicit integration technique called *the radial return method* to obtain the plastic multiplier. This method has the advantage of being unconditionally stable, however, the accuracy remains dependent on the step size. The plastic parameters to indicate in the material file are:  $\sigma_y$  (Yield stress) and **h** (Hardening modulus). In addition, the elastic parameters need to be defined as previously mentioned: E (Young's modulus), **nu** (Poisson's ratio).

#### 4.4.5 Damage

In the simplified case of a linear elastic and brittle material, isotropic damage can be represented by a scalar variable *d*, which varies from 0 to 1 for no damage to fully broken material respectively. The stress-strain relationship then becomes:

$$\sigma = (1 - d)C:\varepsilon$$

where  $\sigma$ ,  $\varepsilon$  are the Cauchy stress and strain tensors, and C is the elastic stiffness tensor. This formulation relies on the definition of an evolution law for the damage variable. In **Akantu**, many possibilities exist and they are listed below.

#### Marigo (B.7)

This damage evolution law is energy based as defined by Marigo [9,10]. It is an isotropic damage law.

$$Y = \frac{1}{2}\varepsilon : C : \varepsilon \tag{4.62}$$

$$F = Y - Y_d - Sd \tag{4.63}$$

$$d = \begin{cases} \min\left(\frac{Y-Y_d}{S}, 1\right) & \text{if } F > 0\\ \text{unchanged} & \text{otherwise} \end{cases}$$
(4.64)

In this formulation, Y is the strain energy release rate,  $Y_d$  the rupture criterion and S the damage energy. The non-local version of this damage evolution law is constructed by averaging the energy Y.

#### Mazars (B.8)

This law introduced by Mazars [11] is a behavioral model to represent damage evolution in concrete. This model does not rely on the computation of the tangent stiffness, the damage is directly evaluated from the strain.

The governing variable in this damage law is the equivalent strain  $\varepsilon_{eq} = \sqrt{\langle \varepsilon \rangle_+ \langle \varepsilon \rangle_+}$ , with  $\langle \cdot \rangle_+$  the positive part of the tensor. This part is defined in the principal coordinates (I, II, III) as  $\varepsilon_{eq} = \sqrt{\langle \varepsilon_I \rangle_+^2 + \langle \varepsilon_{II} \rangle_+^2 + \langle \varepsilon_{III} \rangle_+^2}$ . The damage is defined as:

$$D = \alpha_t^\beta D_t + (1 - \alpha_t)^\beta D_c \tag{4.65}$$

$$D_t = 1 - \frac{\kappa_0 (1 - A_t)}{\varepsilon_{\text{eq}}} - A_t \exp^{-B_t(\varepsilon_{\text{eq}} - \kappa_0)}$$
(4.66)

$$D_c = 1 - \frac{\kappa_0 (1 - A_c)}{\varepsilon_{\text{eq}}} - A_c \exp^{-B_c(\varepsilon_{\text{eq}} - \kappa_0)}$$
(4.67)

$$\alpha_t = \frac{\sum_{i=1}^3 \langle \varepsilon_i \rangle_+ \langle \varepsilon_{\text{nd } i}}{\varepsilon_{\text{eq}}^2}$$
(4.68)

With  $\kappa_0$  the damage threshold,  $A_t$  and  $B_t$  the damage parameter in traction,  $A_c$  and  $B_c$  the damage parameter in compression,  $\beta$  is the shear parameter.  $\alpha_t$  is the coupling parameter between traction and compression, the  $\varepsilon_i$  are the eigenstrain and the  $\varepsilon_{nd i}$  are the eigenvalues of the strain if the material were undamaged.

The coefficients *A* and *B* are the post-peak asymptotic value and the decay shape parameters.



Figure 4.14: Irreversible cohesive laws for explicit simulations.

# 4.5 Cohesive Constitutive laws

#### 4.5.1 Linear Irreversible Law (B.10)

**Akantu** includes the Snozzi-Molinari [12] linear irreversible cohesive law (see Figure 4.14). It is an extension to the Camacho-Ortiz [13] cohesive law in order to make dissipated fracture energy path-dependent. The concept of free potential energy is dropped and a new independent parameter  $\kappa$  is introduced:

$$\kappa = \frac{G_{\rm c,II}}{G_{\rm c,I}} \tag{4.69}$$

where  $G_{c,I}$  and  $G_{c,II}$  are the necessary works of separation per unit area to open completely a cohesive zone under mode I and mode II, respectively. Their model yields to the following equation for cohesive tractions *T* in case of crack opening  $\delta$ :

$$\boldsymbol{T} = \left(\frac{\beta^2}{\kappa}\Delta_{\rm t}\boldsymbol{t} + \Delta_{\rm n}\boldsymbol{n}\right)\frac{\sigma_{\rm c}}{\delta}\left(1 - \frac{\delta}{\delta_{\rm c}}\right) = \hat{\boldsymbol{T}}\frac{\sigma_{\rm c}}{\delta}\left(1 - \frac{\delta}{\delta_{\rm c}}\right) \tag{4.70}$$

where  $\sigma_c$  is the material strength along the fracture,  $\delta_c$  the critical effective displacement after which cohesive tractions are zero (complete decohesion),  $\Delta_t$  and  $\Delta_n$  are the tangential and normal components of the opening displacement vector  $\Delta$ , respectively. The parameter  $\beta$  is a weight that indicates how big the tangential opening contribution is. The effective opening displacement is:

$$\delta = \sqrt{\frac{\beta^2}{\kappa^2} \Delta_t^2 + \Delta_n^2} \tag{4.71}$$

In case of unloading or reloading  $\delta < \delta_{max}$ , tractions are calculated as:

$$T_{\rm n} = \Delta_{\rm n} \frac{\sigma_{\rm c}}{\delta_{\rm max}} \left( 1 - \frac{\delta_{\rm max}}{\delta_{\rm c}} \right) \tag{4.72}$$

$$T_{\rm t} = \frac{\beta^2}{\kappa} \Delta_{\rm t} \frac{\sigma_{\rm c}}{\delta_{\rm max}} \left( 1 - \frac{\delta_{\rm max}}{\delta_{\rm c}} \right) \tag{4.73}$$

so that they vary linearly between the origin and the maximum attained tractions. As shown in Figure 4.14, in this law, the dissipated and reversible energies are:

$$E_{\rm diss} = \frac{1}{2} \sigma_{\rm c} \,\delta_{\rm max} \tag{4.74}$$

$$E_{\rm rev} = \frac{1}{2}T\,\delta\tag{4.75}$$

Moreover, a damage parameter *D* can be defined as:

$$D = \min\left(\frac{\delta_{\max}}{\delta_{c}}, 1\right) \tag{4.76}$$

which varies from 0 (undamaged condition) and 1 (fully damaged condition). This variable can only increase because damage is an irreversible process. A simple penalty contact model has been incorporated in the cohesive law so that normal tractions can be returned in case of compression:

$$T_{\rm n} = \alpha \Delta_{\rm n} \quad \text{if } \Delta_{\rm n} < 0 \tag{4.77}$$

where  $\alpha$  is a stiffness parameter that defaults to zero. The relative contact energy is equivalent to reversible energy but in compression.

The material name of the linear decreasing cohesive law is material\_cohesive\_linear and its parameters with their respective default values are:

sigma\_c: 0
delta\_c: 0
beta: 0
G\_c: 0
kappa: 1
penalty: 0

where  $G_c$  corresponds to  $G_{c,I}$ . A random number generator can be used to assign a random  $\sigma_c$  to each facet following a given distribution (see Section 4.4). Only one parameter between delta\_c and  $G_c$  has to be specified. For random  $\sigma_c$  distributions, the chosen parameter of these two is kept fixed and the other one is varied.

The bilinear constitutive law works exactly the same way as the linear one, except for the additional parameter delta\_0 that by default is zero. Two examples for the extrinsic and intrinsic cohesive elements and also an example to assign different properties to intergranular and transgranular cohesive elements can be found in the folder examples/cohesive\_element/.

#### 4.5.2 Linear Cohesive Law with Friction (B.11)

This law represents a variation of the linear irreversible cohesive of the previous section, which adds friction. The friction behavior is approximated with an elasto-plastic law, which relates the friction force to the relative sliding between the two faces of the cohesive element. The slope of the elastic branch is called *penalty\_for\_friction*, and is defined by the user, together with the friction coefficient, as a material property. The friction contribution evolves with the damage of the cohesive law: it is null when the damage is zero, and it becomes maximum when the damage is equal to one. This is done by defining a current value of the friction coefficient (mu) that increases linearly with the damage, up to the value of the friction coefficient defined by the user. The yielding plateau of the friction law is given by the product of the current friction coefficient and the local compression stress acting in the cohesive element. Such an approach is equivalent to a node-to-node contact friction. Its accuracy is acceptable only for small displacements.

The material name of the linear cohesive law with friction is material\_cohesive\_linear\_friction. Its additional parameters with respect to those of the linear cohesive law without friction, with the respective default values, are:

```
• mu:0
```

```
o penalty_for_friction: 0
```

#### 4.5.3 Linear Cohesive Law with Fatigue (B.12)

This law represents a variation of the linear irreversible cohesive law of the previous section, that removes the hypothesis of elastic unloading-reloading cycles. With this law, some energy is dissipated also during unloading and reloading with hysteresis. The implementation follows the work of [14]. During the unloading-reloading cycle, the traction increment is computed as

$$\dot{T} = \begin{cases} K^- \dot{\delta} & \text{if } \dot{\delta} < 0\\ K^+ \dot{\delta} & \text{if } \dot{\delta} > 0 \end{cases}$$
(4.78)

where  $\dot{\delta}$  and  $\dot{T}$  are respectively the effective opening displacement and the cohesive traction increments with respect to time, while  $K^-$  and  $K^+$  are respectively the unloading and reloading incremental stiffness. The unloading path is linear and results in an unloading stiffness

$$K^{-} = \frac{T_{\max}}{\delta_{\max}}$$
(4.79)

where  $T_{\text{max}}$  and  $\delta_{\text{max}}$  are the maximum cohesive traction and the effective opening displacement reached during the precedent loading phase. The unloading stiffness remains constant during the unloading phase. On the other hand the reloading stiffness increment  $\dot{K}^+$  is calculated as

$$\dot{K}^{+} = \begin{cases} -K^{+} \dot{\delta} / \delta_{\rm f} & \text{if } \dot{\delta} > 0\\ (K^{+} - K^{-}) \dot{\delta} / \delta_{\rm f} & \text{if } \dot{\delta} < 0 \end{cases}$$
(4.80)

where  $\delta_f$  is a material parameter (refer to [15] for more details). During unloading the stiffness  $K^+$  tends to  $K^-$ , while during reloading  $K^+$  gets decreased at every time step. If the cohesive traction during reloading exceeds the upper limit given by equation (4.70), it is recomputed following the behavior of the linear decreasing cohesive law for crack opening.

#### 4.5.4 Exponential Cohesive Law (B.13)

Ortiz and Pandolfi proposed this cohesive law in 1999 [4]. The traction-opening equation for this law is as follows:

$$T = e\sigma_c \frac{\delta}{\delta_c} e^{-\delta/\delta_c}$$
(4.81)

This equation is plotted in Figure 4.15. The term  $\partial T/\partial \delta$  of equation (3.6) after the necessary derivation can expressed as

$$\frac{\partial T}{\partial \delta} = \hat{T} \otimes \frac{\partial (T/\delta)}{\partial \delta} \frac{\hat{T}}{\delta} + \frac{T}{\delta} \left[ \beta^2 I + \left( 1 - \beta^2 \right) (n \otimes n) \right]$$
(4.82)

where

$$\frac{\partial (T/\delta)}{\partial \delta} = \begin{cases} -e\frac{\sigma_c}{\delta_c^2} e^{-\delta/\delta_c} & if\delta \ge \delta_{max} \\ 0 & if\delta < \delta_{max}, \delta_n > 0 \end{cases}$$
(4.83)

As regards the behavior in compression, two options are available: a contact penalty approach with stiffness following the formulation of the exponential law and a contact penalty approach with constant stiffness. In the second case, the stiffness is defined as a function of the tangent of the exponential law at the origin.



Figure 4.15: Exponential cohesive law

# 4.6 Non-Local Constitutive Laws

Continuum damage modeling of quasi-brittle materials undergo significant softening after the onset of damage. This fast growth of damage causes a loss of ellipticity of partial differential equations of equilibrium. Therefore, the numerical simulation results won't be objective anymore, because the dissipated energy will depend on mesh size used in the simulation. One way to avoid this effect is the use of non-local damage formulations. In this approach a local quantity such as the strain is replaced by its non-local average, where the size of the domain, over which the quantity is averaged, depends on the underlying material microstructure. **Akantu** provides non-local versions of many constitutive laws for damage. Examples are for instance the material Mazar and the material Marigo, that can be used in a non-local context. In order to use the corresponding non-local formulation the user has to define the non-local material he wishes to use in the text input file:

```
material constitutive_law_non_local [
    name = material_namerho = value...]
```

where *constitutive\_law\_non\_local* is the name of the non-local constitutive law, *e.g. marigo\_non\_local*. In addition to the material the non-local neighborhood, that should be used for the averaging process needs to be defined in the material file as well:

```
non_local neighborhood_name weight_function_type [
   radius = value
   ...
   weight_function weight_parameter [
      damage_limit = value
      ...
]
]
```

for the non-local averaging, *e.g. base\_wf*, followed by the properties of the non-local neighborhood, such as the radius, and the weight function parameters. It is important to notice that the non-local neighborhood must have the same name as the material to which the neighborhood belongs! The following two sections list the non-local constitutive laws and different type of weight functions available in **Akantu**.

#### 4.6.1 Non-local constitutive laws

Let us consider a body having a volume *V* and a boundary  $\Gamma$ . The stress-strain relation for a non-local damage model can be described as follows:

$$\sigma = (1 - \bar{d})D:\epsilon \tag{4.84}$$

with D the elastic moduli tensor,  $\sigma$  the stress tensor,  $\epsilon$  the strain tensor and  $\bar{d}$  the non-local damage variable. Note that this stres-strain relationship is similar to the relationship defined in Damage model except  $\bar{d}$ . The non-local damage model can be extended to the damage constitutive laws: Marigo (Section 4.4.5) and Mazars (Section 4.4.5).

The non-local damage variable  $\bar{d}$  is defined as follows:

$$\bar{d}(\mathbf{x}) = \int_{V} W(\mathbf{x}, \mathbf{y}) d(\mathbf{y}) dV(\mathbf{y})$$
(4.85)

with W(x, y) the weight function which averages local damage variables to describe the non-local interactions. A list of available weight functions and its functionalities in **Akantu** are explained in the next section.

#### 4.6.2 Non-local weight functions

The available weight functions in **Akantu** are follows:

- base\_weight\_function: This weight function averages local damage variables by using a bellshape function on spatial dimensions.
- *damaged\_weight\_function*: A linear-shape weight function is applied to average local damage variables. Its slope is determined by damage variables. For example, the damage variables for an element which is highly damaged are averaged over large spatial dimension (linear function including a small slope).
- *remove\_damaged\_weight\_function*: This weight function averages damage values by using a bell-shape function as *base\_weight\_function*, but excludes elements which are fully damaged.
- *remove\_damaged\_with\_damage\_rate\_weight\_function*: A bell-shape function is applied to average local damage variables for elements having small damage rates.
- stress\_based\_weight\_function: Non local integral takes stress states, and use the states to construct weight function: an ellipsoid shape. Detailed explanations of this weight function are given in Giry et al.: Stress-based nonlocal damage model IJSS, 48, 2011.

# 4.7 Adding a New Constitutive Law

There are several constitutive laws in **Akantu** as described in the previous Section 4.4. It is also possible to use a user-defined material for the simulation. These materials are referred to as local materials since they are local to the example of the user and not part of the **Akantu** library. To define a new local material, two files (material\_XXX.hh and material\_XXX.cc) have to be provided where XXX is the name of the new material. The header file material\_XXX.hh defines the interface of your custom material. Its implementation is provided in the material\_XXX.cc. The new law must inherit from the Material class or any other existing material class. It is therefore necessary to include the interface of the parent material in the header file of your local material and indicate the inheritance in the declaration of the class:

```
/* ----- */
#include "material.hh"
/* ----- */
#ifndef __AKANTU_MATERIAL_XXX_HH__
#define __AKANTU_MATERIAL_XXX_HH__
namespace akantu {
class MaterialXXX : public Material {
/// declare here the interface of your material
};
```

In the header file the user also needs to declare all the members of the new material. These include the parameters that a read from the material input file, as well as any other material parameters that will be computed during the simulation and internal variables.

In the following the example of adding a new damage material will be presented. In this case the parameters in the material will consist of the Young's modulus, the Poisson coefficient, the resistance to damage and the damage threshold. The material will then from these values compute its Lamé coefficients and its bulk modulus. Furthermore, the user has to add a new internal variable damage in order to store the amount of damage at each quadrature point in each step of the simulation. For this specific material the member declaration inside the class will look as follows:

```
class LocalMaterialDamage : public Material {
/// declare constructors/destructors here
/// declare methods and accessors here
 /* _____
                                                           */
 /* Class Members
 /* ----- */
 AKANTU_GET_MACRO_BY_ELEMENT_TYPE_CONST(Damage, damage, Real);
private:
 /// the young modulus
 Real E;
 /// Poisson coefficient
 Real nu;
 /// First Lame coefficient
 Real lambda;
 /// Second Lame coefficient (shear modulus)
 Real mu;
 /// resistance to damage
 Real Yd;
 /// damage threshold
 Real Sd;
 /// Bulk modulus
```

```
Real kpa;
/// damage internal variable
InternalField<Real> damage;
};
```

In order to enable to print the material parameters at any point in the user's example file using the standard output stream by typing:

```
for (UInt m = 0; m < model.getNbMaterials(); ++m)
std::cout << model.getMaterial(m) << std::endl;</pre>
```

the standard output stream operator has to be redefined. This should be done at the end of the header file:

However, the user still needs to register the material parameters that should be printed out. The registration is done during the call of the constructor. Like all definitions the implementation of the constructor has to be written in the material\_XXX.cc file. However, the declaration has to be provided in the material\_XXX.hh file:

The user can now define the implementation of the constructor in the material\_XXX.cc file:

```
this->registerParam("E", E, 0., _pat_parsable, "Young's modulus");
this->registerParam("nu", nu, 0.5, _pat_parsable, "Poisson's ratio");
this->registerParam("lambda", lambda, _pat_readable, "First Lame coefficient
");
this->registerParam("mu", mu, _pat_readable, "Second Lame coefficient");
this->registerParam("kapa", kpa, _pat_readable, "Bulk coefficient");
this->registerParam("Yd", Yd, 50., _pat_parsmod);
this->registerParam("Sd", Sd, 5000., _pat_parsmod);
damage.initialize(1);
AKANTU_DEBUG_OUT();
```

During the intializer list the reference to the model and the material id are assigned and the constructor of the internal field is called. Inside the scope of the constructor the internal values have to be initialized and the parameters, that should be printed out, are registered with the function: registerParam:

```
void registerParam(name of the parameter (key in the material file),
    member variable,
    default value (optional parameter),
    access permissions,
    description);
```

The available access permissions are as follows:

- \_pat\_internal : Parameter can only be output when the material is printed.
- \_pat\_writable: User can write into the parameter. The parameter is output when the material is printed.
- \_pat\_readable : User can read the parameter. The parameter is output when the material is printed.
- \_pat\_modifiable: Parameter is writable and readable.
- \_pat\_parsable : Parameter can be parsed, *i.e.* read from the input file.
- \_pat\_parsmod: Parameter is modifiable and parsable.

In order to implement the new constitutive law the user needs to specify how the additional material parameters, that are not defined in the input material file, should be calculated. Furthermore, it has to be defined how stresses and the stable time step should be computed for the new local material. In the case of implicit simulations, in addition, the computation of the tangent stiffness needs to be defined. Therefore, the user needs to redefine the following functions of the parent material:

}

In the following a detailed description of these functions is provided:

- initMaterial: This method is called after the material file is fully read and the elements corresponding to each material are assigned. Some of the frequently used constant parameters are calculated in this method. For example, the Lamé constants of elastic materials can be considered as such parameters.
- **computeStress**: In this method, the stresses are computed based on the constitutive law as a function of the strains of the quadrature points. For example, the stresses for the elastic material are calculated based on the following formula:

$$\sigma = \lambda \operatorname{tr}(\varepsilon) I + 2\mu\varepsilon \tag{4.86}$$

Therefore, this method contains a loop on all quadrature points assigned to the material using the two macros:

MATERIAL\_STRESS\_QUADRATURE\_POINT\_LOOP\_BEGIN MATERIAL\_STRESS\_QUADRATURE\_POINT\_LOOP\_END MATERIAL\_STRESS\_QUADRATURE\_POINT\_LOOP\_BEGIN(element\_type); // sigma <- f(grad\_u)

MATERIAL\_STRESS\_QUADRATURE\_POINT\_LOOP\_END;

**Note:** The strain vector in **Akantu** contains the values of  $\nabla u$ , i.e. it is really the displacement gradient,

• **computeTangentStiffness**: This method is called when the tangent to the stress-strain curve is desired (see Fig 4.16). For example, it is called in the implicit solver when the stiffness matrix for the regular elements is assembled based on the following formula:

$$K = \int B^T D(\varepsilon) B \tag{4.87}$$

Therefore, in this method, the tangent matrix (D) is computed for a given strain. Note: The tangent matrix is a 4<sup>th</sup> order tensor which is stored as a matrix in Voigt notation.



Figure 4.16: Tangent to the stress-strain curve.

• getCelerity: The stability criterion of the explicit integration scheme depend on the fastest wave celerity (4.21). This celerity depend on the material, and therefore the value of this velocity should be defined in this method for each new material. By default, the fastest wave speed is the compressive wave whose celerity can be defined in getPushWaveSpeed.

Once the declaration and implementation of the new material has been completed, this material can be used in the user's example by including the header file:

```
#include "material_XXX.hh"
```

For existing materials, as mentioned in Section 4.4, by default, the materials are initialized inside the method initFull. If a local material should be used instead, the initialization of the material has to be postponed until the local material is registered in the model. Therefore, the model is initialized with the boolean for skipping the material initialization equal to true:

```
/// model initialization
model.initFull(_analysis_method = _explicit_lumped_mass);
```

Once the model has been initialized, the local material needs to be registered in the model:

```
model.registerNewCustomMaterials<XXX>("name_of_local_material");
```

Only at this point the material can be initialized:

```
model.initMaterials();
```

A full example for adding a new damage law can be found in **examples/new\_material**.

## 4.7.1 Adding a New Non-Local Constitutive Law

In order to add a new non-local material we first have to add the local constitutive law in **Akantu** (see above). We can then add the non-local version of the constitutive law by adding the two files (material\_XXX\_non\_local.hh and material\_XXX\_non\_local.cc) where XXX is the name of the corresponding local material. The new law must inherit from the two classes, non-local parent class, such as the MaterialNonLocal class, and from the local version of the constitutive law, *i.e.* MaterialXXX. It is therefore necessary to include the interface of those classes in the header file of your custom material and indicate the inheritance in the declaration of the class:

As members of the class we only need to add the internal fields to store the non-local quantities, which are obtained from the averaging process:

}

The following four functions need to be implemented in the non-local material:

```
/// initialization of the material
void initMaterial();
/// loop over all element and invoke stress computation
virtual void computeNonLocalStresses(GhostType ghost_type);
/// compute stresses after local quantities have been averaged
virtual void computeNonLocalStress(ElementType el_type, GhostType ghost_type
)
/// compute all local quantities
void computeStress(ElementType el_type, GhostType ghost_type);
```

In the intialization of the non-local material we need to register the local quantity for the averaging process. In our example the internal field *grad\_u\_nl* is the non-local counterpart of the gradient of the displacement field (*grad\_u\_nl*):

```
void MaterialXXXNonLocal::initMaterial() {
   MaterialXXX::initMaterial();
   MaterialNonLocal::initMaterial();
   /// register the non-local variable in the manager
   this->model->getNonLocalManager().registerNonLocalVariable(this->grad_u.
   getName(), this->grad_u_nl.getName(), spatial_dimension *
   spatial_dimension);
```

The function to register the non-local variable takes as parameters the name of the local internal field, the name of the non-local counterpart and the number of components of the field we want to average. In the *computeStress* we now need to compute all the quantities we want to average. We can then write a loop for the stress computation in the function *computeNonLocalStresses* and then provide the constitutive law on each integration point in the function *computeNonLocalStress*.

# **Chapter 5**

# **Structural Mechanics Model**

Static structural mechanics problems can be handled using the class StructuralMechanicsModel. So far, **Akantu** provides 2D and 3D Bernoulli beam elements [16]. This model is instantiated for a given Mesh, as for the SolidMechanicsModel. The model will create its own FEEngine object to compute the interpolation, gradient, integration and assembly operations. The StructuralMechanicsModel constructor is called in the following way:

StructuralMechanicsModel model(mesh, spatial\_dimension);

where mesh is a Mesh object defining the structure for which the equations of statics are to be solved, and spatial\_dimension is the dimensionality of the problem. If spatial\_dimension is omitted, the problem is assumed to have the same dimensionality as the one specified by the mesh.

**Note 1:** *Dynamic computations are not supported to date.* 

**Note 2:** Structural meshes are created and loaded as described in Section 2.4 with \_miot\_gmsh\_struct instead of \_miot\_gmsh:

Mesh mesh; mesh.read("structural\_mesh.msh", \_miot\_gmsh\_struct);

This model contains at least the following Arrays:

- blocked\_dofs contains a Boolean value for each degree of freedom specifying whether that degree is blocked or not. A Dirichlet boundary condition can be prescribed by setting the blocked\_dofs value of a degree of freedom to true. The displacement is computed for all degrees of freedom for which the blocked\_dofs value is set to false. For the remaining degrees of freedom, the imposed values (zero by default after initialization) are kept.
- **displacement\_rotation** contains the generalized displacements (*i.e.* displacements and rotations) of all degrees of freedom. It can be either a computed displacement for free degrees of freedom or an imposed displacement in case of blocked ones (*u* in the following).
- **external\_force** contains the generalized external forces (forces and moments) applied to the nodes (*f*<sub>ext</sub> in the following).
- **internal\_force** contains the generalized internal forces (forces and moments) applied to the nodes (*f*<sub>int</sub> in the following).

An example to help understand how to use this model will be presented in the next section.

# 5.1 Model Setup

#### 5.1.1 Initialization

The easiest way to initialize the structural mechanics model is:

```
model.initFull();
```

The method initFull computes the shape functions, initializes the internal vectors mentioned above and allocates the memory for the stiffness matrix, unlike the solid mechanics model, its default argument is \_static.

Material properties are defined using the StructuralMaterial structure described in Table 5.1. Such a definition could, for instance, look like

```
StructuralMaterial mat1;
mat.E=3e10;
mat.I=0.0025;
mat.A=0.01;
```

Field	Description
Ε	Young's modulus
Α	Cross section area
Ι	Second cross sectional moment of inertia (for 2D elements)
Iy	I around beam <i>y</i> -axis (for 3D elements)
Iz	I around beam <i>z</i> -axis (for 3D elements)
GJ	Polar moment of inertia of beam cross section (for 3D elements)

Table 5.1: Material properties for structural elements defined in the class StructuralMaterial.

Materials can be added to the model's element\_material vector using

```
model.addMaterial(mat1);
```

They are successively numbered and then assigned to specific elements.

```
for (UInt i = 0; i < nb_element_mat_1; ++i) {
    model.getElementMaterial(_bernoulli_beam_2)(i,0) = 1;
}</pre>
```

#### 5.1.2 Setting Boundary Conditions

As explained before, the Dirichlet boundary conditions are applied through the array **blocked\_dofs**. Two options exist to define Neumann conditions. If a nodal force is applied, it has to be directly set in the array **force\_momentum**. For loads distributed along the beam length, the method **computeForcesFromFunction** integrates them into nodal forces. The method takes as input a function describing the distribution of loads along the beam and a functor **BoundaryFunctionType** specifing if the function is expressed in the local coordinates (\_bft\_traction\_local) or in the global system of coordinates (\_bft\_traction).

# 5.2 Static Analysis

The StructuralMechanicsModel class can perform static analyses of structures. In this case, the equation to solve is the same as for the SolidMechanicsModel used for static analyses

$$Ku = f_{\text{ext}} , \qquad (5.1)$$

where *K* is the global stiffness matrix, *u* the generalized displacement vector and  $f_{\text{ext}}$  the vector of generalized external forces applied to the system.

To solve such a problem, the static solver of the StructuralMechanicsModel object is used. First a model has to be created and initialized.

```
StructuralMechanicsModel model(mesh);
model.initFull();
```

• model.initFull initializes all internal vectors to zero.

Once the model is created and initialized, the boundary conditions can be set as explained in Section 5.1.2. Boundary conditions will prescribe the external forces or moments for the free degrees of freedom  $f_{ext}$  and displacements or rotations for the others. To completely define the system represented by equation (5.1), the global stiffness matrix K must be assembled.

```
model.assembleStiffnessMatrix();
```

The computation of the static equilibrium is performed using the same Newton-Raphson algorithm as described in Section 4.2.

**Note:** To date, **StructuralMechanicsModel** handles only constitutively and geometrically linear problems, the algorithm is therefore guaranteed to converge in two iterations.

```
model.updateResidual();
model.solve();
```

- model.updateResidual assembles the internal forces and removes them from the external forces.
- model.solve solves the Equation (5.1). The increment vector of the model will contain the new increment of displacements, and the displacement\_rotation vector is also updated to the new displacements.

At the end of the analysis, the final solution is stored in the **displacement\_rotation** vector. A full example of how to solve a structural mechanics problem is presented in the code **examples/structural\_mechanics/bernoulli\_beam\_2\_example.cc**. This example is composed of a 2D beam, clamped at the left end and supported by two rollers as shown in Figure 5.1. The problem is defined by the applied load  $q = 6 \text{ kN m}^{-1}$ , moment  $\overline{M} = 3.6 \text{ kN m}$ , moments of inertia  $I_1 = 250\,000 \text{ cm}^4$  and  $I_2 = 128\,000 \text{ cm}^4$  and lengths  $L_1 = 10 \text{ m}$  and  $L_2 = 8 \text{ m}$ . The resulting rotations at node two and three are  $\varphi_2 = 0.001\,167$  and  $\varphi_3 = -0.000\,771$ .



Figure 5.1: 2D beam example

# **Chapter 6**

# Heat Transfer Model

The heat transfer model is a specific implementation of the Model interface dedicated to handle the dynamic heat equation.

# 6.1 Theory

The strong form of the dynamic heat equation can be expressed as

$$\rho c_v \dot{T} + \nabla \cdot \kappa \nabla T = b \tag{6.1}$$

with *T* the scalar temperature field,  $c_v$  the specific heat capacity,  $\rho$  the mass density,  $\kappa$  the conductivity tensor, and *b* the heat generation per unit of volume. The discretized weak form with a finite number of elements is

$$\forall i \quad \sum_{j} \left( \int_{\Omega} \rho c_{v} N_{j} N_{i} d\Omega \right) \dot{T}_{j} - \sum_{j} \left( \int_{\Omega} \kappa \nabla N_{j} \nabla N_{i} d\Omega \right) T_{j} = - \int_{\Gamma} N_{i} \boldsymbol{q} \cdot \boldsymbol{n} d\Gamma + \int_{\Omega} b N_{i} d\Omega \tag{6.2}$$

with *i* and *j* the node indices, *n* the normal field to the surface  $\Gamma = \partial \Omega$ . To simplify, we can define the capacity and the conductivity matrices as

$$C_{ij} = \int_{\Omega} \rho c_v N_j N_i d\Omega \quad \text{and} \quad K_{ij} = -\int_{\Omega} \kappa \nabla N_j \nabla N_i d\Omega \tag{6.3}$$

and the system to solve can be written

$$C \cdot \dot{T} = Q^{\text{ext}} - K \cdot T , \qquad (6.4)$$

with  $Q^{\text{ext}}$  the consistent heat generated.

# 6.2 Using the Heat Transfer Model

A material file name has to be provided during initialization. Currently, the HeatTransferModel object uses dynamic analysis with an explicit time integration scheme. It can simply be created like this

```
HeatTransferModel model(mesh, spatial_dimension);
```

while an existing mesh has been used (see 2.4). Then the model object can be initialized with:

model.initFull()

This function will load the material properties, and allocate / initialize the nodes and element Array's More precisely, the heat transfer model contains 4 Arrays:

**temperature** contains the nodal temperature *T* (zero by default after the initialization).

**temperature\_rate** contains the variations of temperature  $\dot{T}$  (zero by default after the initialization).

**blocked\_dofs** contains a Boolean value for each degree of freedom specifying whether the degree is blocked or not. A Dirichlet boundary condition  $(T_d)$  can be prescribed by setting the **blocked\_dofs** value of a degree of freedom to true. The **temperature** and the **temperature\_rate** are computed for all degrees of freedom where the **blocked\_dofs** value is set to false. For the remaining degrees of freedom, the imposed values (zero by default after initialization) are kept.

external\_heat\_rate contains the external heat generations. *Q*<sup>*ext*</sup> on the nodes.

**internal\_heat\_rate** contains the internal heat generations.  $Q^{int} = -K \cdot T$  on the nodes.

Only a single material can be specified on the domain. A material text file (*e.g.*, material.dat) provides the material properties as follows:

```
model heat_transfer_model [
  capacity = XXX
  density = XXX
  conductivity = [XXX ... XXX]
]
```

where the capacity and density are scalars, and the conductivity is specified as a 3×3 tensor.

#### 6.2.1 Explicit Dynamic

The explicit time integration scheme in **Akantu** uses a lumped capacity matrix C (reducing the computational cost, see Chapter 4). This matrix is assembled by distributing the capacity of each element onto its nodes. Therefore, the resulting C is a diagonal matrix stored in the capacity **Array** of the model.

```
model.assembleCapacityLumped();
```

**Note:** *Currently, only the explicit time integration with lumped capacity matrix is implemented within Akantu.* The explicit integration scheme is *Forward Euler* [5].

- Predictor:  $T_{n+1} = T_n + \Delta t \dot{T}_n$
- Update residual:  $R_{n+1} = (Q_{n+1}^{ext} KT_{n+1})$
- Corrector :  $\dot{T}_{n+1} = C^{-1}R_{n+1}$

The explicit integration scheme is conditionally stable. The time step has to be smaller than the stable time step, and it can be obtained in **Akantu** as follows:

time\_step = model.getStableTimeStep();

The stable time step is defined as:

$$\Delta t_{\rm crit} = 2\Delta x^2 \frac{\rho c_v}{\|\kappa\|^{\infty}} \tag{6.5}$$

where  $\Delta x$  is the characteristic length (*e.g.*, the inradius in the case of linear triangle element),  $\rho$  is the density,  $\kappa$  is the conductivity tensor, and  $c_v$  is the specific heat capacity. It is necessary to impose a time step which is smaller than the stable time step, for instance, by multiplying the stable time step by a safety factor smaller than one.

```
const Real safety_time_factor = 0.1;
Real applied_time_step = time_step * safety_time_factor;
model.setTimeStep(applied_time_step);
```

The following loop allows, for each time step, to update the temperature, residual and temperature\_rate fields following the previously described integration scheme.

```
for (UInt s = 1; (s-1)*applied_time_step < total_time; ++s) {
  model.solveStep();
}</pre>
```

An example of explicit dynamic heat propagation is presented in examples/heat\_transfer/explicit\_heat\_transfer.cc.

This example consists of a square 2D plate of  $1 \text{ m}^2$  having an initial temperature of 100 K everywhere but a none centered hot point maintained at 300 K. Figure 6.1 presents the geometry of this case. The material used is a linear fictitious elastic material with a density of  $8940 \text{ kg/m}^3$ , a conductivity of  $401 \text{ W} \text{ m}^{-1} \text{ K}^{-1}$  and a specific heat capacity of  $385 \text{ J} \text{ K}^{-1} \text{ kg}^{-1}$ . The time step used is

0.12 s.





Figure 6.1: Initial temperature field (left) and after 15000 time steps = 30 minutes (right). The lines represent iso-surfaces.

# Chapter 7

# Input/Output

# 7.1 Input file

The text input file of a simulation should be precised using the method initialize which will instantiate the static Parser object of **Akantu**. This section explains how to manipulate Parser objects to input data in **Akantu**.

```
int main(int argc, char *argv[]) {
    initialize("input_files.dat", argc, argv);
    ...
```

# 7.1.1 Akantu Parser

**Akantu** file parser has a tree organization.

• Parser, the root of the tree, can be accessed using

```
Parser & parser = getStaticParser();
```

 ParserSection, branch of the tree, contains map a of sub-sections (SectionType, ParserSection) and a ParserSection \* pointing to the parent section. The user section of the input file can directly be accessed by

```
const ParserSection & usersect = getUserParser();
```

• **ParserParameter**, the leaf of the tree, carries data of the input file which can be casted to the correct type with

```
Real mass = usersect.getParameter("mass");
```

or used directly within an expression

Real dead\_weight = 9.81 \* usersect.getParameterValue<Real>("mass");

#### 7.1.2 Grammar

The structure of text input files consists of different sections containing a list of parameters. As example, the file parsed in the previous section will look like

```
user parameters [
   mass = 10.5
]
```

Basically every standard arithmetic operations can be used inside of input files as well as the constant **pi** and **e** and the exponent operator  $\hat{}$ . Operations between ParserParameter are also possible with the convention that only parameters of the current and the parent sections are available. Vector and Matrix can also be read according to the NumPy [17] writing convention (a.e. cauchy\_stress\_tensor =  $[[\sigma_{xx}, \sigma_{xy}], [\sigma_{yx}, \sigma_{yy}]]$ ). An example illustrating how to parse the following input file can be found in example\io\parser\example\_parser.cc.

```
user parameters [
   spatial_dimension = 2
   mesh_file = swiss_cheese.msh
   inner_holes = holes
   outter_crust = crust
   lactostatic_p = 30e3
   stress = [[lactostatic_p,0],[0,lactostatic_p]]
   max_nb_iterations = 100
   precision = 1e-9
]
```

### 7.1.3 Material section

The input file should also be used to specify material characteristics (constitutive behavior and material properties). The dedicated material section is then read by initFull method of SolidMechanicsModel which initializes the different materials specified with the following convention:

```
material constitutive_law <optional flavor> [
   name = value
   rho = value
   ...
]
```

where *constitutive\_law* is the adopted constitutive law, followed by the material properties listed one by line in the bracket (*e.g.*, **name** and density **rho**). Some constitutive laws can also have an *optional flavor*. More information can be found in sections relative to material constitutive laws 4.4 or in Appendix B.

# 7.2 Output data

## 7.2.1 Generic data

In this chapter, we address ways to get the internal data in human-readable formats. The models in **Akantu** handle data associated to the mesh, but this data can be split into several **Arrays**. For example, the data stored per element type in a **ElementTypeMapArray** is composed of as many **Array**s as types in the mesh.

In order to get this data in a visualization software, the models contain a object to dump VTK files. These files can be visualized in software such as ParaView [18], ViSit [19] or Mayavi [20].

The internal dumper of the model can be configured to specify which data fields are to be written. This is done with the addDumpField method. By default all the files are generated in a folder called paraview/

```
model.setBaseName("output"); // prefix for all generated files
model.addDumpField("displacement");
model.addDumpField("stress");
...
```

key	type	support
displacement	Vector <real></real>	nodes
mass	Vector <real></real>	nodes
velocity	Vector <real></real>	nodes
acceleration	Vector <real></real>	nodes
force	Vector <real></real>	nodes
residual	Vector <real></real>	nodes
increment	Vector <real></real>	nodes
blocked_dofs	Vector <bool></bool>	nodes
partitions	Real	elements
material_index	variable	elements
strain	Matrix <real></real>	quadrature points
Green strain	Matrix <real></real>	quadrature points
principal strain	Vector <real></real>	quadrature points
principal Green strain	Vector <real></real>	quadrature points
grad_u	Matrix <real></real>	quadrature points
stress	Matrix <real></real>	quadrature points
Von Mises stress	Real	quadrature points
material_index	variable	quadrature points

Table 7.1: List of dumpable fields for SolidMechanicsModel.

#### model.dump()

The fields are dumped with the number of components of the memory. For example, in 2D, the memory has Vector's of 2 components, or the  $2^{nd}$  order tensors with  $2 \times 2$  components. This memory can be dealt with addDumpFieldVector which always dumps Vector's with 3 components or addDumpFieldTensor which dumps  $2^{nd}$  order tensors with  $3 \times 3$  components respectively. The routines addDumpFieldVector and addDumpFieldTensor were introduced because of ParaView which mostly manipulate 3D data.

Those fields which are stored by quadrature point are modified to be seen in the VTK file as elemental data. To do this, the default is to average the values of all the quadrature points.

The list of fields depends on the models (for SolidMechanicsModel see table 7.1).

#### 7.2.2 Cohesive elements' data

Cohesive elements and their relative data can be easily dumped thanks to a specific dumper contained in SolidMechanicsModelCohesive. In order to use it, one has just to add the string "cohesive elements" when calling each method already illustrated. Here is an example on how to dump displacement and damage:

```
model.setBaseNameToDumper("cohesive elements", "cohesive_elements_output");
model.addDumpFieldVectorToDumper("cohesive elements", "displacement");
model.addDumpFieldToDumper("cohesive elements", "damage");
...
model.dump("cohesive elements");
```

#### **Fragmentation data**

Whenever the SolidMechanicsModelCohesive is used, it is possible to dump additional data about the fragments that get formed in the simulation both in serial and parallel. This task is

carried out by the **FragmentManager** class, that takes care of computing the following quantities for each fragment:

- index;
- mass;
- moments of inertia;
- velocity;
- number of elements.

These computations can be realized at once by calling the function computeAllData, or individually by calling the other public functions of the class. The data can be dumped to be visualized in Paraview, or can be accessed within the simulation. An example of usage is:

```
FragmentManager fragment_manager(model);
fragment_manager.buildAllData();
. . .
model.addDumpField("fragments");
                                         // this field contains the indices
model.addDumpField("fragments mass");
model.addDumpField("moments of inertia");
model.addDumpField("elements per fragment");
. . .
for (UInt step = 1; step <= total_steps; ++step) {</pre>
  . . .
  fragment_manager.buildAllData();
 model.dump();
}
. . .
const Array<Real> & fragment_velocities = fragment_manager.getVelocity();
. . .
```

At the end of this example the velocities of the fragments are accessed with a reference to a **const Array**<**Real**>. The size of this array is the number of fragments, and its number of components is the spatial dimension in this case.

#### 7.2.3 Advanced dumping

#### **Arbitrary fields**

In addition to the predetermined fields from the models and materials, the user can add any data to a dumper as long as the support is the same. That is to say data that have the size of the full mesh on if the dumper is dumping the mesh, or of the size of an element group if it is a filtered dumper.

For this the easiest is to use the "external" fields register functions

The simple case force nodal and elemental data are to pass directly the data container itself if it as the good size.

• For nodal fields :

```
Array<T> nodal_data(nb_nodes, nb_component);
...
model.addDumpFieldExternal("my_field", nodal_data);
```

• For elemental fields :

```
ElementTypeMapArray<T> elem_data;
...
model.addDumpFieldExternal("my_field", elem_data);
```

If some changes have to be applied on the data as for example a padding for ParaView vectors, this can be done by using the field interface.

An example of code presenting this interface is present in the **examples/io/dumper**. This interface is part of the **Dumpable** class from which the **Mesh** inherits.

#### Creating a new dumper

You can also create you own dumpers, **Akantu** uses a third-party library in order to write the output files, IOHelper. **Akantu** supports the ParaView format and a Text format defined by IOHelper.

This two files format are handled by the classes DumperParaview and DumperText.

In order to use them you can instantiate on of this object in your code. This dumper have a simple interface. You can register a mesh registerMesh, registerFilteredMesh or a field, registerField.

An example of code presenting this low level interface is present in the examples/io/dumper. The different types of Field that can be created are present in the source folder src/io/dumper.
# **Chapter 8**

# **Parallel Computation**

This section explains how to launch a parallel computation. The strategy adopted by **Akantu** uses a mesh partitioning where elements are mapped to processors. Mesh partitions are then distributed to available processors by adequate routines as will be described below. The sequence of additional operations to be performed by the user are:

- Initializing the parallel context
- Partitioning the mesh
- Distributing mesh partitions

After these steps, the Model object proceeds with the interprocess communication automatically without the user having to explicitly take care of them. In what follows we show how it works on a SolidMechanics model.

## 8.1 Initializing the Parallel Context

The user must initialize **Akantu** by forwarding the arguments passed to the program by using the function initialize, and close **Akantu** instances at the end of the program by calling the finalize function.

**Note:** This step does not change from the sequential case as it was stated in Section 2.3. It only gives a additional motivation in the parallel/MPI context.

The initialize function builds a StaticCommunicator object responsible for handling the interprocess communications later on. The StaticCommunicator can, for instance, be used to ask the total number of declared processors available for computations as well as the process rank through the functions getNbProc and whoAmI respectively.

An example of the initializing sequence and basic usage of the StaticCommunicator is:

```
int main(int argc, char *argv[]) {
    initialize("material.dat", argc, argv);
    const auto & comm = Communicator::getStaticCommunicator();
    Int psize = comm.getNbProc();
    Int prank = comm.whoAmI();
    ...
    finalize();
}
```

#### 8.2 Partitioning and distributing the Mesh

The mesh is partitioned after the correct initialization of the processes playing a role in the computation. We assume that a Mesh object is constructed as presented in Section 2.4. Then it can be distributed among the different processes. At the moment, the only partitioner available is MeshPartitionScotch which is used by the function distribute using the Scotch [21] program. This is achieved by the following code:

```
Mesh mesh(spatial_dimension);
if(prank == 0) {
  mesh.read("my_mesh.msh");
}
mesh.distribute();
```

The algorithm that partition the mesh needs the generation of a random distribution of values. Therefore, in order to run several time a simulation with the same partition of the mesh, the *seed* has to be set manually. This can be done either by adding the following line to the input file *outside* the material parameters environments:

seed = 1

where the value 1.0 can be substituted with any number, or by setting it directly in the code with the command:

```
RandGenerator:: seed(1)
```

The latter command, with empty brackets, can be used to check the value of the *seed* used in the simulation.

**Note:** Only the processor of rank 0 should load the mesh file to partition it. Nevertheless, the **Mesh** object must by declared for all processors since the mesh distribution will store mesh pieces to that object.

An example of an explicit dynamic 2D bar in compression in a parallel context can be found in examples/parallel\_2d.

#### 8.3 Launching a Parallel Program

Using MPI a parallel run can be launched from a shell using the command

mpirun -np #procs program\_name parameter1 parameter2 ...

# Chapter 9

# **Python interface**

Akantu as a python interface which allows to implement a finite element case entirely in the Python language. The interface is currently in beta version and bugs (including memory leaks) can be expected.

The philosophy of the python wrappers is that it should follow, as much as possible, the C++ interface. This is made possible thanks to the software which can port C++ interfaces to Python. However, the possibilities of the Python language have some restrictions which do not allow the concept of template. For this reason the templated classes and methods are currently hardly supported in the Akantu Python interface (some are, but without any warranty that the interface shall not change in a near future).

In order to use the python interface you want to import the akantu module which is possibly located in directory where you built or where you installed **Akantu**. The easiest way is to source the file akantu\_environement.sh in your terminal environment:

```
source AKANTU_BUILD_DIR/akantu_environement.sh
```

or for an installed akantu version:

source AKANTU\_INSTALL\_DIR/share/akantuVERSION/akantu\_environement.sh

Then, you can create a Python script, like script.py which starts with:

import akantu

Then, you can write the initialization sequence just like for the C++ case:

```
akantu.initialize('materials.dat') spatial_dimension = 2
```

mesh = akantu.Mesh(spatial\_dimension) mesh.read('mesh.msh')
model = akantu.SolidMechanicsModel(mesh)
model.initFull(\_analysis\_method = akantu.\_static)

Then for the dumpers declaration:

```
model.setBaseName("example") model.addDumpFieldVector("displacement")
model.addDumpFieldVector("force") model.addDumpField("boundary")
model.addDumpField("strain") model.addDumpField("stress")
model.addDumpField("blocked_dofs")
```

The **Akantu** arrays can be retrieved at all time. For convenience there is an automatic transtyping operator which allows to use **Akantu** Array, Vectors and Matrix as mere **Numpy**s.

displacement = model.getDisplacement()

There is no copy made and the Numpy vectors are wrapped around Akantu's memory pool.

displacement[:,:] = 1. displacement[0,1] = 10.

Also to assign an entire vector to another the '[]' operator must be used:

```
result = numpy.linspace(0.,1.,100)
# This works
displacement[:] = result[:]
# This does not
displacement = result
```

Then you can solve the problem for instance with:

```
model.solveStaticDisplacement(1e-10,2);
```

And dump to Paraview:

model.dump()

Do not forget to finalize at the end of the script:

akantu.finalize()

Finally the python script can be launched with:

python ./script.py

Warning! : We recall once more that the Python interface is in **beta** state and is provided only for its nice services: it avoids recompilation of akantu when coding the main function. Several preliminary examples are provided in the **examples** directory where it is described how to specify boundary conditions and also constitutive behavior.

# Appendix A

# **Shape Functions**

Schmatic overview of all the element types defined in **Akantu** is described in Section 3.2. In this appendix, more detailed information (shape function, location of Gaussian quadrature points, and so on) of each of these types is listed. For each element type, the coordinates of the nodes are given in the isoparametric frame of reference, together with the shape functions (and their derivatives) on these respective nodes. Also all the Gaussian quadrature points within each element are assigned (together with the weight that is applied on these points). The graphical representations of all the element types can be found in Section 3.2.

#### A.1 1D-Shape Functions

#### A.1.1 Segment 2

#### **Element properties**

Node (i)	Coord. $(\xi)$	Shape function $(N_i)$	Derivative $(\partial N_i / \partial \xi)$
1	-1	$\frac{1}{2}(1-\xi)$	$-\frac{1}{2}$
2	1	$\frac{1}{2}(1+\xi)$	$\frac{1}{2}$

#### Gaussian quadrature points

Coord.  $(\xi)$  0 Weight 2

#### A.1.2 Segment 3

#### **Element properties**

Node (i)	Coord. (ξ)	Shape function $(N_i)$	Derivative $(\partial N_i / \partial \xi)$
1	-1	$\frac{1}{2}\xi(\xi-1)$	$\xi - \frac{1}{2}$
2	1	$\frac{1}{2}\xi(\xi+1)$	$\xi + \frac{1}{2}$
3	0	$1 - \xi^2$	-2ξ

Coord. $(\xi)$	$-1/\sqrt{3}$	$1/\sqrt{3}$
Weight	1	1

# A.2 2D-Shape Functions

## A.2.1 Triangle 3

#### **Element properties**

Node (i)	Coord. $(\xi, \eta)$	Shape function $(N_i)$	Derivative $(\partial N_i/\partial \xi , \partial N_i/\partial \eta)$
1	(0,0)	$1 - \xi - \eta$	(-1,-1)
2	(1,0)	ξ	(1,0)
3	(0,1)	η	(0,1)

#### Gaussian quadrature points

Coord.  $(\xi, \eta)$   $\left(\frac{1}{3}, \frac{1}{3}\right)$ Weight  $\frac{1}{2}$ 

## A.2.2 Triangle 6

#### **Element properties**

Node (i)	Coord. $(\xi, \eta)$	Shape function $(N_i)$	Derivative $(\partial N_i / \partial \xi , \partial N_i / \partial \eta)$
1	(0,0)	$-(1-\xi-\eta)(1-2(1-\xi-\eta))$	$(1-4(1-\xi-\eta), 1-4(1-\xi-\eta))$
2	(1,0)	$-\xi (1-2\xi)$	$(4\xi - 1, 0)$
3	(0,1)	$-\eta \left(1-2\eta\right)$	$(\ 0\ ,\ 4\eta -1\ )$
4	$\left(\frac{1}{2}, 0\right)$	$4\xi\left(1-\xi-\eta\right)$	$(4(1-2\xi-\eta), -4\xi)$
5	$\left(\begin{array}{cc} \frac{1}{2} \\ \frac{1}{2} \end{array}\right)$	$4\xi\eta$	$(\ 4\eta$ , $4\xi$ $)$
6	$\left(\begin{array}{c}0\ ,\ \frac{1}{2}\end{array}\right)$	$4\eta \left(1-\xi-\eta\right)$	$(-4\eta , 4(1-\xi -2\eta))$

Coord. ( $\xi$ , $\eta$ )	$\left(\frac{1}{6}, \frac{1}{6}\right)$	$\left(\frac{2}{3},\frac{1}{6}\right)$	$\left(\frac{1}{6}, \frac{2}{3}\right)$
Weight	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{6}$

## A.2.3 Quadrangle 4

#### **Element properties**

Node (i)	Coord. $(\xi, \eta)$	Shape function $(N_i)$	Derivative $(\partial N_i/\partial \xi , \partial N_i/\partial \eta)$
1	(-1,-1)	$\frac{1}{4}\left(1-\xi\right)\left(1-\eta\right)$	$\left( -\frac{1}{4} \left( 1 - \eta \right) , -\frac{1}{4} \left( 1 - \xi \right) \right)$
2	(1,-1)	$\frac{1}{4}\left(1+\xi\right)\left(1-\eta\right)$	$\left( \frac{1}{4} \left( 1 - \eta \right), - \frac{1}{4} \left( 1 + \xi \right) \right)$
3	(1,1)	$\frac{1}{4}\left(1+\xi\right)\left(1+\eta\right)$	$\left( \frac{1}{4} \left( 1 + \eta \right), \frac{1}{4} \left( 1 + \xi \right) \right)$
4	(-1,1)	$\frac{1}{4}\left(1-\xi\right)\left(1+\eta\right)$	$\left( -\frac{1}{4} \left( 1+\eta \right), \ \frac{1}{4} \left( 1-\xi \right) \right)$

#### Gaussian quadrature points

(ξ,η)	$\left(-\frac{1}{\sqrt{3}},-\frac{1}{\sqrt{3}}\right)$	$\left(\frac{1}{\sqrt{3}}, -\frac{1}{\sqrt{3}}\right)$	$\left(\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}\right)$	$\left(-\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}\right)$
Weight	1	1	1	1

# A.2.4 Quadrangle 8

#### **Element properties**

Node (i)	Coord. $(\xi, \eta)$	Shape function $(N_i)$	Derivative $(\partial N_i/\partial \xi, \partial N_i/\partial \eta)$
1	(-1,-1)	$\frac{1}{4}(1-\xi)(1-\eta)(-1-\xi-\eta)$	$\left(\frac{1}{4}\left(1-\eta\right)\left(2\xi+\eta\right), \frac{1}{4}\left(1-\xi\right)\left(\xi+2\eta\right)\right)$
2	(1,-1)	$\frac{1}{4}(1+\xi)(1-\eta)(-1+\xi-\eta)$	$\left(\frac{1}{4}\left(1-\eta\right)\left(2\xi-\eta\right), -\frac{1}{4}\left(1+\xi\right)\left(\xi-2\eta\right)\right)$
3	(1,1)	$\frac{1}{4}\left(1+\xi\right)\left(1+\eta\right)\left(-1+\xi+\eta\right)$	$\left(\frac{1}{4}\left(1+\eta\right)\left(2\xi+\eta\right), \frac{1}{4}\left(1+\xi\right)\left(\xi+2\eta\right)\right)$
4	(-1,1)	$\frac{1}{4}(1-\xi)(1+\eta)(-1-\xi+\eta)$	$\left( \begin{array}{c} \frac{1}{4} \left( 1+\eta \right) \left( 2\xi -\eta \right)  , \ -\frac{1}{4} \left( 1-\xi \right) \left( \xi -2\eta \right) \right)$
5	(0,-1)	$\frac{1}{2}(1-\xi^2)(1-\eta)$	$\left( \left\xi \left( 1-\eta \right) ,\right. \leftrac{1}{2} \left( 1-\xi^2  ight)  ight)$
6	(1,0)	$\frac{1}{2}(1+\xi)(1-\eta^2)$	$\left( \frac{1}{2} \left( 1 - \eta^2 \right), \ -\eta \left( 1 + \xi \right) \right)$
7	(0,1)	$\frac{1}{2}(1-\xi^2)(1+\eta)$	$\left( \left\xi \left( 1+\eta \right) ,\right. \left. rac{1}{2} \left( 1-\xi^2 \right)  ight)  ight)$
8	(-1,0)	$\frac{1}{2}(1-\xi)(1-\eta^2)$	$\left( \leftrac{1}{2}\left( 1-\eta^{2} ight) , -\eta\left( 1-\xi ight)  ight)  ight)$

Coord. ( $\xi$ , $\eta$ )	(0,0)	$\left(\sqrt{\frac{3}{5}},\sqrt{\frac{3}{5}}\right)$	$\left(-\sqrt{\frac{3}{5}}, \sqrt{\frac{3}{5}}\right)$	$\left(-\sqrt{\frac{3}{5}},-\sqrt{\frac{3}{5}}\right)$	$\left(\sqrt{\frac{3}{5}}, -\sqrt{\frac{3}{5}}\right)$
Weight	64/81	25/81	25/81	25/81	25/81
Coord. ( $\xi$ , $\eta$ )	$\left(0, \sqrt{\frac{3}{5}}\right)$	$\left(-\sqrt{\frac{3}{5}},0\right)$	$\left(0, -\sqrt{\frac{3}{5}}\right)$	$\left(\sqrt{\frac{3}{5}},0\right)$	
Weight	40/81	40/81	40/81	40/81	

# A.3 3D-Shape Functions

#### A.3.1 Tetrahedron 4

#### **Element properties**

Node (i)	Coord. $(\xi, \eta, \zeta)$	Shape function $(N_i)$	Derivative $(\partial N_i/\partial \xi, \partial N_i/\partial \eta, \partial N_i/\partial \zeta)$
1	(0,0,0)	$1 - \xi - \eta - \zeta$	(-1,-1,-1)
2	(1,0,0)	ξ	(1,0,0)
3	(0,1,0)	η	(0,1,0)
4	(0,0,1)	ζ	(0,0,1)

#### Gaussian quadrature points

Coord.  $(\xi, \eta, \zeta)$   $\left(\frac{1}{4}, \frac{1}{4}, \frac{1}{4}\right)$ Weight  $\frac{1}{6}$ 

#### A.3.2 Tetrahedron 10

#### **Element properties**

Node (i)	Coord. $(\xi, \eta, \zeta)$	Shape function $(N_i)$	Derivative $(\partial N_i / \partial \xi, \partial N_i / \partial \eta, \partial N_i / \partial \zeta)$
			$(4\xi + 4\eta + 4\zeta - 3)$
1	(0,0,0)	$(1-\xi-\eta-\zeta)(1-2\xi-2\eta-2\zeta)$	$4\xi+4\eta+4\zeta-3\;,$
			$4\xi+4\eta+4\zeta-3\;)$
2	(1,0,0)	$\xi (2\xi - 1)$	$(4\xi - 1, 0, 0)$
3	(0,1,0)	$\eta \left( 2\eta -1 ight)$	$(\ 0 \ , \ 4\eta - 1 \ , \ 0 \ )$
4	(0,0,1)	$\zeta (2\zeta - 1)$	$(0, 0, 4\zeta - 1)$
5	$\left(\begin{array}{cccccccccccccccccccccccccccccccccccc$	$4\xi\left(1-\xi-\eta-\zeta\right)$	$\left( \ 4-8\xi-4\eta-4\zeta \ , \ -4\xi \ , \ -4\xi \ \right)$
6	$\left( \begin{array}{ccc} rac{1}{2} \ , \ rac{1}{2} \ , \ 0 \end{array} \right)$	$4\xi\eta$	( $4\eta$ , $4\xi$ , $0$ )
7	$\left(\begin{array}{c}0\ ,\ \frac{1}{2}\ ,\ 0\end{array}\right)$	$4\eta\left(1-\xi-\eta-\zeta\right)$	$(-4\eta$ , $4-4\xi-8\eta-4\zeta$ , $-4\eta$ $)$
8	$(0, 0, \frac{1}{2})$	$4\zeta\left(1-\xi-\eta-\zeta\right)$	$(-4\zeta, -4\zeta, 4-4\xi-4\eta-8\zeta)$
9	$\left( \frac{1}{2} , 0 , \frac{1}{2} \right)$	4ξζ	$(4\zeta,0,4\xi)$
10	$\left( \begin{array}{ccc} 0 \end{array}, \begin{array}{c} rac{1}{2} \end{array}, \begin{array}{c} rac{1}{2} \end{array}  ight)$	$4\eta\zeta$	$(\ 0\ ,\ 4\zeta\ ,\ 4\eta\ )$

Coord. (ξ , η , ζ)	$\left(\frac{(5-\sqrt{5})}{20}, \frac{(5-\sqrt{5})}{20}, \frac{(5-\sqrt{5})}{20}\right)$	$\left(\frac{(5+3\sqrt{5})}{20}, \frac{(5-\sqrt{5})}{20}, \frac{(5-\sqrt{5})}{20}\right)$
Weight	1/24	1/24
Coord. ( $\xi$ , $\eta$ , $\zeta$ )	$\left(\frac{(5-\sqrt{5})}{20}, \frac{(5+3\sqrt{5})}{20}, \frac{(5-\sqrt{5})}{20}\right)$	$\left(\frac{(5-\sqrt{5})}{20}, \frac{(5-\sqrt{5})}{20}, \frac{(5+3\sqrt{5})}{20}\right)$
Weight	1/24	1/24

#### A.3.3 Hexahedron 8

#### **Element properties**

Node (i)	Coord. $(\xi, \eta, \zeta)$	Shape function $(N_i)$	Derivative $(\partial N_i/\partial \xi, \partial N_i/\partial \eta, \partial N_i/\partial \zeta)$
1	(-1,-1,-1)	$\frac{1}{8}(1-\xi)(1-\eta)(1-\zeta)$	$\left(-\frac{1}{8}(1-\eta)(1-\zeta), -\frac{1}{8}(1-\xi)(1-\zeta), -\frac{1}{8}(1-\xi)(1-\eta)\right)$
2	(1,-1,-1)	$\frac{1}{8}(1+\xi)(1-\eta)(1-\zeta)$	$\left(\frac{1}{8}(1-\eta)(1-\zeta), -\frac{1}{8}(1+\xi)(1-\zeta), -\frac{1}{8}(1+\xi)(1-\eta)\right)$
3	(1,1,-1)	$\frac{1}{8}\left(1+\xi\right)\left(1+\eta\right)\left(1-\zeta\right)$	$\left(\frac{1}{8}(1+\eta)(1-\zeta),\frac{1}{8}(1+\xi)(1-\zeta),-\frac{1}{8}(1+\xi)(1+\eta)\right)$
4	(-1,1,-1)	$\frac{1}{8}(1-\xi)(1+\eta)(1-\zeta)$	$\left(-\tfrac{1}{8}(1+\eta)(1-\zeta),\tfrac{1}{8}(1-\xi)(1-\zeta),-\tfrac{1}{8}(1-\xi)(1+\eta)\right)$
5	(-1,-1,1)	$\frac{1}{8}(1-\xi)(1-\eta)(1+\zeta)$	$\left(-\tfrac{1}{8}\left(1-\eta\right)\left(1+\zeta\right),-\tfrac{1}{8}\left(1-\xi\right)\left(1+\zeta\right),\tfrac{1}{8}\left(1-\xi\right)\left(1-\eta\right)\right)$
6	(1, -1, 1)	$\frac{1}{8}\left(1+\xi\right)\left(1-\eta\right)\left(1+\zeta\right)$	$\left(\frac{1}{8}(1-\eta)(1+\zeta),-\frac{1}{8}(1+\xi)(1+\zeta),\frac{1}{8}(1+\xi)(1-\eta)\right)$
7	(1,1,1)	$\frac{1}{8}\left(1+\xi\right)\left(1+\eta\right)\left(1+\zeta\right)$	$\left(\frac{1}{8}(1+\eta)(1+\zeta),\frac{1}{8}(1+\xi)(1+\zeta),\frac{1}{8}(1+\xi)(1+\eta)\right)$
8	(-1,1,1)	$\frac{1}{8}(1-\xi)(1+\eta)(1+\zeta)$	$\left(-\frac{1}{8}(1+\eta)(1+\zeta), \frac{1}{8}(1-\xi)(1+\zeta), \frac{1}{8}(1-\xi)(1+\eta)\right)$

#### Gaussian quadrature points

(ξ,η,ζ)	$\left(-\frac{1}{\sqrt{3}}, -\frac{1}{\sqrt{3}}, -\frac{1}{\sqrt{3}}\right)$	$\left(\frac{1}{\sqrt{3}}, -\frac{1}{\sqrt{3}}, -\frac{1}{\sqrt{3}}\right)$	$\left(\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}, -\frac{1}{\sqrt{3}}\right)$	$\left(-\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}, -\frac{1}{\sqrt{3}}\right)$
Weight	1	1	1	1
(ξ,η,ζ)	$\left(-\frac{1}{\sqrt{3}}, -\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}\right)$	$\left(\frac{1}{\sqrt{3}}, -\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}\right)$	$\left(\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}\right)$	$\left(-\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}\right)$
Weight	1	1	1	1

#### A.3.4 Pentahedron 6

#### **Element properties**

Node (i)	Coord. $(\xi, \eta, \zeta)$	Shape function $(N_i)$	Derivative $(\partial N_i / \partial \xi, \partial N_i / \partial \eta, \partial N_i / \partial \zeta)$
1	(-1,1,0)	$\frac{1}{2}(1-\xi)\eta$	$\left( -\frac{1}{2}\eta , \frac{1}{2}(1-\xi) , 0.0 \right)$
2	(-1,0,1)	$\frac{1}{2}\left(1-\xi\right)\zeta$	$\left( -\frac{1}{2}\zeta , \ 0.0 \ , \ \frac{1}{2} \left( 1 - \xi \right) \right)$
3	(-1,0,0)	$\frac{1}{2}\left(1-\xi\right)\left(1-\eta-\zeta\right)$	$\left( -\frac{1}{2} \left( 1 - \eta - \zeta \right) , -\frac{1}{2} \left( 1 - \xi \right) , -\frac{1}{2} \left( 1 - \xi \right) \right)$
4	(1,1,0)	$\frac{1}{2}\left(1+\xi\right)\eta$	$\left(\begin{array}{c}\frac{1}{2}\eta \ , \ \frac{1}{2}\left(1+\xi\right) \ , \ 0.0 \end{array}\right)$
5	(1,0,1)	$\frac{1}{2}\left(1+\xi\right)\zeta$	$\left(\begin{array}{c}\frac{1}{2}\zeta\;,\;0.0\;,\;\frac{1}{2}\left(1+\xi\right)\right)$
6	(1,0,0)	$\frac{1}{2}\left(1+\xi\right)\left(1-\eta-\zeta\right)$	$\left(\frac{1}{2}\left(1-\eta-\zeta\right), -\frac{1}{2}\left(1+\xi\right), -\frac{1}{2}\left(1+\xi\right)\right)$

(ξ,η,ζ)	$\left(-\frac{1}{\sqrt{3}}, 0.5, 0.5\right)$	$\left(-\frac{1}{\sqrt{3}}, 0.0, 0.5\right)$	$\left(-rac{1}{\sqrt{3}}$ , 0.5 , 0.0 $ ight)$	$\left(\frac{1}{\sqrt{3}}, 0.5, 0.5\right)$	$\left(\frac{1}{\sqrt{3}}, 0.0, 0.5\right)$	$\left(\frac{1}{\sqrt{3}}, 0.5, 0.0\right)$
Weight	1/6	1/6	1/6	1/6	1/6	1/6

#### A.3.5 Hexahedron 20

## **Element properties**

Node (i)	Coord. $(\xi, \eta, \zeta)$	Shape function $(N_i)$
1	(-1,-1,-1)	$\frac{1}{8}(1-\xi)(1-\eta)(1-\zeta)(-2-\xi-\eta-\zeta)$
2	(1, -1, -1)	$\frac{1}{8}(1+\xi)(1-\eta)(1-\zeta)(-2+\xi-\eta-\zeta)$
3	(1,1,-1)	$\frac{1}{8}(1+\xi)(1+\eta)(1-\zeta)(-2+\xi+\eta-\zeta)$
4	(-1,1,-1)	$\frac{1}{8}(1-\xi)(1+\eta)(1-\zeta)(-2-\xi+\eta-\zeta)$
5	(-1,-1,1)	$\frac{1}{8}(1-\xi)(1-\eta)(1+\zeta)(-2-\xi-\eta+\zeta)$
6	(1, -1, 1)	$\frac{1}{8} (1 + \xi) (1 - \eta) (1 + \zeta) (-2 + \xi - \eta + \zeta)$
7	(1,1,1)	$\frac{1}{8}\left(1+\xi\right)\left(1+\eta\right)\left(1+\zeta\right)\left(-2+\xi+\eta+\zeta\right)$
8	(-1,1,1)	$\frac{1}{8}\left(1-\xi\right)\left(1+\eta\right)\left(1+\zeta\right)\left(-2-\xi+\eta+\zeta\right)$
9	(0, -1, -1)	$\tfrac{1}{4}\left(1-\xi^2\right)\left(1-\eta\right)\left(1-\zeta\right)$
10	(1,0,-1)	$\frac{1}{4}(1+\xi)(1-\eta^2)(1-\zeta)$
11	(0,1,-1)	$\tfrac{1}{4}\left(1-\xi^2\right)\left(1+\eta\right)\left(1-\zeta\right)$
12	(-1,0,-1)	$\frac{1}{4}(1-\xi)(1-\eta^2)(1-\zeta)$
13	(-1,-1,0)	$\frac{1}{4}(1-\xi)(1-\eta)(1-\zeta^2)$
14	(1, -1, 0)	$\frac{1}{4}(1+\xi)(1-\eta)(1-\zeta^2)$
15	(1,1,0)	$\frac{1}{4}\left(1+\xi\right)\left(1+\eta\right)\left(1-\zeta^{2}\right)$
16	(-1,1,0)	$\frac{1}{4}\left(1-\xi\right)\left(1+\eta\right)\left(1-\zeta^{2}\right)$
17	(0, -1, 1)	$\tfrac{1}{4}\left(1-\xi^2\right)\left(1-\eta\right)\left(1+\zeta\right)$
18	(1,0,1)	$\tfrac{1}{4}\left(1+\xi\right)\left(1-\eta^2\right)\left(1+\zeta\right)$
19	(0,1,1)	$\tfrac{1}{4}\left(1-\xi^2\right)\left(1+\eta\right)\left(1+\zeta\right)$
20	(-1,0,1)	$\frac{1}{4}(1-\xi)(1-\eta^2)(1+\zeta)$

#### A.3. 3D-SHAPE FUNCTIONS

## **Element properties**

Node (i)	Derivative $(\partial N_i/\partial \xi, \partial N_i/\partial \eta, \partial N_i/\partial \zeta)$
1	$\left(\frac{1}{4}\left(\xi + \frac{1}{2}(\eta + \zeta + 1)\right)(\eta - 1)(\zeta - 1), \frac{1}{4}\left(\eta + \frac{1}{2}(\xi + \zeta + 1)\right)(\xi - 1)(\zeta - 1), \frac{1}{4}\left(\zeta + \frac{1}{2}(\xi + \eta + 1)\right)(\xi - 1)(\eta - 1)\right)(\xi - 1)(\eta - 1)(\zeta - 1)(\xi - 1)(\eta - 1)(\xi - $
2	$\left(\frac{1}{4}\left(\xi - \frac{1}{2}\left(\eta + \zeta + 1\right)\right)(\eta - 1)\left(\zeta - 1\right), -\frac{1}{4}\left(\eta - \frac{1}{2}\left(\xi - \zeta - 1\right)\right)\left(\xi + 1\right)\left(\zeta - 1\right), -\frac{1}{4}\left(\zeta - \frac{1}{2}\left(\xi - \eta - 1\right)\right)\left(\xi + 1\right)\left(\eta - 1\right)\right)\left(\xi + 1\right)\left(\eta - 1\right)\right)\left(\xi + 1\right)\left(\zeta - 1\right), -\frac{1}{4}\left(\zeta - \frac{1}{2}\left(\xi - \eta - 1\right)\right)\left(\xi + 1\right)\left(\eta - 1\right)\right)\left(\xi + 1\right)\left(\zeta - 1\right), -\frac{1}{4}\left(\zeta - \frac{1}{2}\left(\xi - \eta - 1\right)\right)\left(\xi + 1\right)\left(\eta - 1\right)\right)\left(\xi + 1\right)\left(\zeta - 1\right), -\frac{1}{4}\left(\zeta - \frac{1}{2}\left(\xi - \eta - 1\right)\right)\left(\xi - 1\right), -\frac{1}{4}\left(\zeta - \frac{1}{2}\left(\xi - \eta - 1\right)\right)\left(\xi - 1\right), -\frac{1}{4}\left(\zeta - \frac{1}{2}\left(\xi - \eta - 1\right)\right)\left(\xi - 1\right), -\frac{1}{4}\left(\zeta - \frac{1}{2}\left(\xi - \eta - 1\right)\right)\left(\xi - 1\right), -\frac{1}{4}\left(\zeta - \frac{1}{2}\left(\xi - \eta - 1\right)\right)\left(\xi - 1\right), -\frac{1}{4}\left(\zeta - \frac{1}{2}\left(\xi - \eta - 1\right)\right)\left(\xi - 1\right), -\frac{1}{4}\left(\zeta - \frac{1}{2}\left(\xi - \eta - 1\right)\right)\left(\xi - 1\right), -\frac{1}{4}\left(\zeta - \frac{1}{2}\left(\xi - \eta - 1\right)\right)\left(\xi - 1\right), -\frac{1}{4}\left(\zeta - \frac{1}{2}\left(\xi - \eta - 1\right)\right)\left(\xi - 1\right), -\frac{1}{4}\left(\zeta - \frac{1}{2}\left(\xi - \eta - 1\right)\right)\left(\xi - 1\right), -\frac{1}{4}\left(\zeta - \frac{1}{2}\left(\xi - \eta - 1\right)\right)\left(\xi - 1\right), -\frac{1}{4}\left(\zeta - \frac{1}{2}\left(\xi - \eta - 1\right)\right)\left(\xi - 1\right), -\frac{1}{4}\left(\xi - \frac{1}{2}\left(\xi - \eta - 1\right)\right)\left(\xi - 1\right), -\frac{1}{4}\left(\xi - \frac{1}{2}\left(\xi - \eta - 1\right)\right)\left(\xi - 1\right), -\frac{1}{4}\left(\xi - \frac{1}{2}\left(\xi - \eta - 1\right)\right)\left(\xi - 1\right), -\frac{1}{4}\left(\xi - \frac{1}{2}\left(\xi - \eta - 1\right)\right)\left(\xi - 1\right), -\frac{1}{4}\left(\xi - \frac{1}{2}\left(\xi - \eta - 1\right)\right)\left(\xi - 1\right), -\frac{1}{4}\left(\xi - \frac{1}{2}\left(\xi - \eta - 1\right)\right)\left(\xi - 1\right), -\frac{1}{4}\left(\xi - \frac{1}{2}\left(\xi - \eta - 1\right)\right)\left(\xi - 1\right), -\frac{1}{4}\left(\xi - \frac{1}{2}\left(\xi - \eta - 1\right)\right)\left(\xi - 1\right), -\frac{1}{4}\left(\xi - \frac{1}{2}\left(\xi - \eta - 1\right)\right)\left(\xi - 1\right), -\frac{1}{4}\left(\xi - \frac{1}{2}\left(\xi - \eta - 1\right)\right)\left(\xi - 1\right), -\frac{1}{4}\left(\xi - \frac{1}{2}\left(\xi - \eta - 1\right)\right)\left(\xi - 1\right), -\frac{1}{4}\left(\xi - \frac{1}{2}\left(\xi - \eta - 1\right)\right)\left(\xi - 1\right), -\frac{1}{4}\left(\xi - \frac{1}{2}\left(\xi - \eta - 1\right)\right)\left(\xi - 1\right), -\frac{1}{4}\left(\xi - \frac{1}{2}\left(\xi - \eta - 1\right)\right)\left(\xi - 1\right), -\frac{1}{4}\left(\xi - \frac{1}{2}\left(\xi - \eta - 1\right)\right)\left(\xi - 1\right), -\frac{1}{4}\left(\xi - \frac{1}{2}\left(\xi - \eta - 1\right)\right)\left(\xi - 1\right), -\frac{1}{4}\left(\xi - \frac{1}{2}\left(\xi - \eta - 1\right)\right)\left(\xi - 1\right), -\frac{1}{4}\left(\xi - \frac{1}{2}\left(\xi - \eta - 1\right)\right)\left(\xi - 1\right), -\frac{1}{4}\left(\xi - \frac{1}{2}\left(\xi - \eta - 1\right)\right)\left(\xi - 1\right), -\frac{1}{4}\left(\xi - \frac{1}{2}\left(\xi - \eta - 1\right)\right)\left(\xi - 1\right), -\frac{1}{4}\left(\xi - \frac{1}{2}\left(\xi - 1\right)\right)\left(\xi - 1\right), -\frac{1}{4}\left(\xi - \frac{1}{2}\left(\xi - 1\right)\right)\left(\xi - 1\right), -\frac{1}{4}\left(\xi - 1\right)\right)\left(\xi - 1\right), -\frac{1}{4}\left(\xi - 1\right)\right)\left(\xi - 1\right), -\frac{1}{4}\left(\xi - 1\right)\right)\left(\xi - 1\right)\right)\left(\xi -$
3	$\left(-\frac{1}{4}\left(\xi+\frac{1}{2}\left(\eta-\zeta-1\right)\right)\left(\eta+1\right)\left(\zeta-1\right), -\frac{1}{4}\left(\eta+\frac{1}{2}\left(\xi-\zeta-1\right)\right)\left(\xi+1\right)\left(\zeta-1\right), \frac{1}{4}\left(\zeta-\frac{1}{2}\left(\xi+\eta-1\right)\right)\left(\xi+1\right)\left(\eta+1\right)\right)\left(\xi+1\right)\left(\eta+1\right)\right)\left(\xi+1\right$
4	$\left(-\frac{1}{4}\left(\xi-\frac{1}{2}\left(\eta-\zeta-1\right)\right)\left(\eta+1\right)\left(\zeta-1\right),\ \frac{1}{4}\left(\eta-\frac{1}{2}\left(\xi+\zeta+1\right)\right)\left(\xi-1\right)\left(\zeta-1\right),\ -\frac{1}{4}\left(\zeta+\frac{1}{2}\left(\xi-\eta+1\right)\right)\left(\xi-1\right)\left(\eta+1\right)\right)\left(\xi-1\right)\left(\eta+1\right)\left(\xi-1\right)\left(\eta+1\right)\left(\xi-1\right)\left(\xi-1\right)\left(\eta+1\right)\left(\xi-1\right$
5	$\left(-\frac{1}{4}\left(\xi+\frac{1}{2}\left(\eta-\zeta+1\right)\right)\left(\eta-1\right)\left(\zeta+1\right), -\frac{1}{4}\left(\eta+\frac{1}{2}\left(\xi-\zeta+1\right)\right)\left(\xi-1\right)\left(\zeta+1\right), \frac{1}{4}\left(\zeta-\frac{1}{2}\left(\xi+\eta+1\right)\right)\left(\xi-1\right)\left(\eta-1\right)\right)\left(\xi-1\right)\left(\eta-1\right)\left(\xi-1\right)\left$
6	$\left(-\frac{1}{4}\left(\xi-\frac{1}{2}\left(\eta-\zeta+1\right)\right)\left(\eta-1\right)\left(\zeta+1\right),\ \frac{1}{4}\left(\eta-\frac{1}{2}\left(\xi+\zeta-1\right)\right)\left(\xi+1\right)\left(\zeta+1\right),\ -\frac{1}{4}\left(\zeta+\frac{1}{2}\left(\xi-\eta-1\right)\right)\left(\xi+1\right)\left(\eta-1\right)\right)\left(\xi+1\right)\left(\eta-1\right)\left(\xi+1\right)\left(\eta-1\right)\left(\xi+1\right)\left(\xi+1\right)\left(\eta-1\right)\left(\xi+1\right$
7	$\left(\frac{1}{4}\left(\xi + \frac{1}{2}\left(\eta + \zeta - 1\right)\right)\left(\eta + 1\right)\left(\zeta + 1\right), \frac{1}{4}\left(\eta + \frac{1}{2}\left(\xi + \zeta - 1\right)\right)\left(\xi + 1\right)\left(\zeta + 1\right), \frac{1}{4}\left(\zeta + \frac{1}{2}\left(\xi + \eta - 1\right)\right)\left(\xi + 1\right)\left(\eta + 1\right)\right)\left(\xi + 1\right)\left(\eta + 1\right)\right)\left(\xi + 1\right)\left(\xi + 1\right)\left(\xi$
8	$\left(\frac{1}{4}\left(\xi - \frac{1}{2}\left(\eta + \zeta - 1\right)\right)\left(\eta + 1\right)\left(\zeta + 1\right), -\frac{1}{4}\left(\eta - \frac{1}{2}\left(\xi - \zeta + 1\right)\right)\left(\xi - 1\right)\left(\zeta + 1\right), -\frac{1}{4}\left(\zeta - \frac{1}{2}\left(\xi - \eta + 1\right)\right)\left(\xi - 1\right)\left(\eta + 1\right)\right)\left(\xi - 1\right)\left(\eta + 1\right)\right)\left(\xi - 1\right)\left(\xi - 1\right)\left$
9	$\left(-\frac{1}{2}\xi(\eta-1)(\zeta-1), -\frac{1}{4}(\xi^2-1)(\zeta-1), -\frac{1}{4}(\xi^2-1)(\eta-1)\right)$
10	$\left( \begin{array}{c} rac{1}{4} \left( \eta^2 - 1  ight) \left( \zeta - 1  ight)  ,  rac{1}{2} \eta \left( \xi + 1  ight) \left( \zeta - 1  ight)  ,  rac{1}{4} \left( \xi + 1  ight) \left( \eta^2 - 1  ight)  ight)$
11	$\left(\frac{1}{2}\xi(\eta+1)(\zeta-1), \frac{1}{4}(\xi^2-1)(\zeta-1), \frac{1}{4}(\xi^2-1)(\eta+1)\right)$
12	$\left( -\frac{1}{4} \left( \eta^2 - 1 \right) \left( \zeta - 1 \right) , -\frac{1}{2} \eta \left( \xi - 1 \right) \left( \zeta - 1 \right) , -\frac{1}{4} \left( \xi - 1 \right) \left( \eta^2 - 1 \right) \right)$
13	$\left( -\frac{1}{4} (\eta - 1) (\zeta^2 - 1) , -\frac{1}{4} (\xi - 1) (\zeta^2 - 1) , -\frac{1}{2} \zeta (\xi - 1) (\eta - 1) \right)$
14	$\left( \frac{1}{4} \left( \eta - 1 \right) \left( \zeta^2 - 1 \right), \frac{1}{4} \left( \xi + 1 \right) \left( \zeta^2 - 1 \right), \frac{1}{2} \zeta \left( \xi + 1 \right) \left( \eta - 1 \right) \right)$
15	$\left( -\frac{1}{4} (\eta + 1) (\zeta^2 - 1) , -\frac{1}{4} (\xi + 1) (\zeta^2 - 1) , -\frac{1}{2} \zeta (\xi + 1) (\eta + 1) \right)$
16	$\left( \frac{1}{4} \left( \eta + 1 \right) \left( \zeta^2 - 1 \right), \frac{1}{4} \left( \xi - 1 \right) \left( \zeta^2 - 1 \right), \frac{1}{2} \zeta \left( \xi - 1 \right) \left( \eta + 1 \right) \right)$
17	$\left(\frac{1}{2}\xi(\eta-1)(\zeta+1), \frac{1}{4}(\xi^2-1)(\zeta+1), \frac{1}{4}(\xi^2-1)(\eta-1)\right)$
18	$\left(-\frac{1}{4}(\eta^2-1)(\zeta+1), -\frac{1}{2}\eta(\xi+1)(\zeta+1), -\frac{1}{4}(\xi+1)(\eta^2-1)\right)$
19	$\left(-\frac{1}{2}\xi(\eta+1)(\zeta+1), -\frac{1}{4}(\xi^2-1)(\zeta+1), -\frac{1}{4}(\xi^2-1)(\eta+1)\right)$
20	$\left( \frac{1}{4} \left( \eta^2 - 1 \right) \left( \zeta + 1 \right), \frac{1}{2} \eta \left( \xi - 1 \right) \left( \zeta + 1 \right), \frac{1}{4} \left( \xi - 1 \right) \left( \eta^2 - 1 \right) \right)$

Coord. ( $\xi$ , $\eta$ , $\zeta$ )	$\left(-\sqrt{\frac{3}{5}}, -\sqrt{\frac{3}{5}}, -\sqrt{\frac{3}{5}}\right)$	$\left(-\sqrt{\frac{3}{5}}, -\sqrt{\frac{3}{5}}, 0\right)$	$\left(-\sqrt{\frac{3}{5}},-\sqrt{\frac{3}{5}},\sqrt{\frac{3}{5}}\right)$	$\left(-\sqrt{\frac{3}{5}}, 0, -\sqrt{\frac{3}{5}}\right)$
Weight	125/729	200/729	125/729	200/729
Coord. ( $\xi$ , $\eta$ , $\zeta$ )	$\left(-\sqrt{\frac{3}{5}}, 0, 0\right)$	$\left(-\sqrt{\frac{3}{5}}, 0, \sqrt{\frac{3}{5}}\right)$	$\left(-\sqrt{\frac{3}{5}},\sqrt{\frac{3}{5}},-\sqrt{\frac{3}{5}}\right)$	$\left(-\sqrt{\frac{3}{5}}, \sqrt{\frac{3}{5}}, 0\right)$
Weight	320/729	200/729	125/729	200/729
Coord. ( $\xi$ , $\eta$ , $\zeta$ )	$\left(-\sqrt{\frac{3}{5}}, \sqrt{\frac{3}{5}}, \sqrt{\frac{3}{5}}\right)$	$\left(0, -\sqrt{\frac{3}{5}}, -\sqrt{\frac{3}{5}}\right)$	$\left(0,-\sqrt{\frac{3}{5}},0 ight)$	$\left(0, -\sqrt{\frac{3}{5}}, \sqrt{\frac{3}{5}}\right)$
Weight	125/729	200/729	320/729	200/729
Coord. ( $\xi$ , $\eta$ , $\zeta$ )	$\left(0, 0, -\sqrt{\frac{3}{5}}\right)$	(0, 0, 0)	$\left(0, 0, \sqrt{\frac{3}{5}}\right)$	$\left(0, \sqrt{\frac{3}{5}}, -\sqrt{\frac{3}{5}}\right)$
Weight	320/729	512/729	320/729	200/729
Coord. ( $\xi$ , $\eta$ , $\zeta$ )	$\left(0, \sqrt{\frac{3}{5}}, 0\right)$	$\left(0, \sqrt{\frac{3}{5}}, \sqrt{\frac{3}{5}}\right)$	$\left(\sqrt{\frac{3}{5}}, -\sqrt{\frac{3}{5}}, -\sqrt{\frac{3}{5}}\right)$	$\left(\sqrt{\frac{3}{5}}, -\sqrt{\frac{3}{5}}, 0\right)$
Weight	320/729	200/729	125/729	200/729
Coord. ( $\xi$ , $\eta$ , $\zeta$ )	$\left( \sqrt{\frac{3}{5}}, -\sqrt{\frac{3}{5}}, \sqrt{\frac{3}{5}} \right)$	$\left(\sqrt{\frac{3}{5}}, 0, -\sqrt{\frac{3}{5}}\right)$	$\left(\sqrt{\frac{3}{5}}, 0, 0\right)$	$\left(\sqrt{\frac{3}{5}}, 0, \sqrt{\frac{3}{5}}\right)$
Weight	125/729	200/729	320/729	200/729
Coord. ( $\xi$ , $\eta$ , $\zeta$ )	$\left(\sqrt{\frac{3}{5}}, \sqrt{\frac{3}{5}}, -\sqrt{\frac{3}{5}}\right)$	$\left(\sqrt{\frac{3}{5}}, \sqrt{\frac{3}{5}}, 0\right)$	$\left(\sqrt{\frac{3}{5}},\sqrt{\frac{3}{5}},\sqrt{\frac{3}{5}}\right)$	
Weight	125/729	200/729	125/729	

#### A.3.6 Pentahedron 15

#### **Element properties**

Node (i)	Coord. $(\xi, \eta, \zeta)$	Shape function $(N_i)$
1	(-1,1,0)	$\frac{1}{2}\eta\left(1-\xi\right)\left(2\eta-2-\xi\right)$
2	(-1,0,1)	$\frac{1}{2}\zeta\left(1-\xi\right)\left(2\zeta-2-\xi\right)$
3	(-1,0,0)	$\frac{1}{2}\left(\xi-1\right)\left(1-\eta-\zeta\right)\left(\xi+2\eta+2\zeta\right)$
4	(1,1,0)	$\frac{1}{2}\eta\left(1+\xi\right)\left(2\eta-2+\xi\right)$
5	(1,0,1)	$\frac{1}{2}\zeta\left(1+\xi\right)\left(2\zeta-2+\xi\right)$
6	(1,0,0)	$\frac{1}{2}\left(-\xi-1\right)\left(1-\eta-\zeta\right)\left(-\xi+2\eta+2\zeta\right)$
7	(-1,0.5,0.5)	$2\eta\zeta\left(1-\xi ight)$
8	(-1,0,0.5)	$2\zeta \left(1-\eta-\zeta\right) (1-\xi)$
9	(-1,0.5,0)	$2\eta \left(1-\xi\right) \left(1-\eta-\zeta\right)$
10	(0,1,0)	$\eta \left(1-\xi^2\right)$
11	(0,0,1)	$\zeta(1-\xi^2)$
12	(0,0,0)	$(1-\xi^2)(1-\eta-\zeta)$
13	(1,0.5,0.5)	$2\eta\zeta\left(1+\xi ight)$
14	(1,0,0.5)	$2\zeta \left(1+\xi\right) \left(1-\eta-\zeta\right)$
15	(1,0.5,0)	$2\eta \left(1+\xi\right) \left(1-\eta-\zeta\right)$

Node (i)	Derivative $(\partial N_i/\partial \xi, \partial N_i/\partial \eta, \partial N_i/\partial \zeta)$
1	$\left(\frac{1}{2}\eta \left(2\xi - 2\eta + 1\right), -\frac{1}{2}(\xi - 1)(4\eta - \xi - 2), 0.0\right)$
2	$\left( \begin{array}{c} rac{1}{2} \zeta \left( 2 \xi - 2 \zeta + 1  ight)  ,   0.0  ,  - rac{1}{2} \left( \xi - 1  ight) \left( 4 \zeta - \xi - 2  ight)  ight)$
3	$\left(-\frac{1}{2}\left(2\xi+2\eta+2\zeta-1\right)\left(\eta+\zeta-1\right),-\frac{1}{2}\left(\xi-1\right)\left(4\eta+\xi+2\left(2\zeta-1\right)\right),-\frac{1}{2}\left(\xi-1\right)\left(4\zeta+\xi+2\left(2\eta-1\right)\right)\right)$
4	$\left( \frac{1}{2}\eta \left( 2\xi + 2\eta - 1 \right), \frac{1}{2} \left( \xi + 1 \right) \left( 4\eta + \xi - 2 \right), \ 0.0 \right)$
5	$\left( \frac{1}{2} \zeta \left( 2\xi + 2\zeta - 1 \right) , \ 0.0 \ , \ \frac{1}{2} \left( \xi + 1 \right) \left( 4\zeta + \xi - 2 \right) \right)$
6	$\left(-\frac{1}{2}\left(\eta+\zeta-1\right)\left(2\xi-2\eta-2\zeta+1\right),\ \frac{1}{2}\left(\xi+1\right)\left(4\eta-\xi+2\left(2\zeta-1\right)\right),\ \frac{1}{2}\left(\xi+1\right)\left(4\zeta-\xi+2\left(2\eta-1\right)\right)\right)$
7	$(-2\eta\zeta, -2(\xi-1)\zeta, -2(\xi-1)\eta)$
8	$\left( 2\zeta \left( \eta + \zeta - 1 \right) \right), 2\zeta - \left( \xi - 1 \right), 2\left( \xi - 1 \right) \left( 2\zeta + \eta - 1 \right) \right)$
9	$(2\eta(\eta + \zeta - 1), 2(2\eta + \zeta - 1)(\xi - 1), 2\eta(\xi - 1))$
10	$(-2\xi\eta,-(\xi^2-1),0.0)$
11	$(-2\xi\zeta , 0.0 , -(\xi^2 - 1))$
12	$\left(2\xi\left(\eta+\zeta-1 ight),\left(\xi^{2}-1 ight),\left(\xi^{2}-1 ight) ight)$
13	$(2\eta\zeta, 2\zeta(\xi+1), 2\eta(\xi+1))$
14	$(-2\zeta(\eta+\zeta-1), -2\zeta(\xi+1), -2(\xi+1)(2\zeta+\eta-1))$
15	$(-2\eta(\eta+\zeta-1), -2(2\eta+\zeta-1)(\xi+1), -2\eta(\xi+1))$

Coord. ( $\xi$ , $\eta$ , $\zeta$ )	$\left(-\frac{1}{\sqrt{3}}, \frac{1}{3}, \frac{1}{3}\right)$	$\left(-\frac{1}{\sqrt{3}}, 0.6, 0.2\right)$	$\left(-\frac{1}{\sqrt{3}}, 0.2, 0.6\right)$	$\left(-rac{1}{\sqrt{3}}$ , 0.2 , 0.2 $ ight)$
Weight	-27/96	25/96	25/96	25/96
Coord. ( $\xi$ , $\eta$ , $\zeta$ )	$\left(\frac{1}{\sqrt{3}}, \frac{1}{3}, \frac{1}{3}\right)$	$\left(\frac{1}{\sqrt{3}}, 0.6, 0.2\right)$	$\left(rac{1}{\sqrt{3}},0.2,0.6 ight)$	$\left(\frac{1}{\sqrt{3}}, 0.2, 0.2\right)$
Weight	-27/96	25/96	25/96	25/96

# Appendix **B**

# **Material parameters**

## **B.1** Linear elastic isotropic

Keyword: elastic Description here: 4.4.1 Parameters:

- rho: (*Real*) Density
- E: (*Real*) Young's modulus
- **nu**: (*Real*) Poisson's ratio
- Plane\_Stress: (bool) Plane stress simplification (only 2D problems)

## **B.2** Linear elastic anisotropic

Keyword: elastic\_anisotropic Description here: 4.4.1 Parameters:

- **rho**: (*Real*) Density
- n1: (*Vector<Real>*) Direction of the main material axis
- n2: (*Vector*<*Real*>) Direction of the second material axis (if applicable)
- n3: (*Vector*<*Real*>) Direction of the third material axis (if applicable)
- **C..**: (*Real*) Coefficient ij of the material tensor C (all the 36 values in Voigt notation can be entered .)

#### **B.3** Linear elastic orthotropic

Keyword: elastic\_orthotropic Description here: 4.4.1 Parameters:

- rho: (Real) Density
- n1: (Vector<Real>) Direction of the main material axis
- n2: (*Vector<Real>*) Direction of the second material axis (if applicable)
- n3: (*Vector*<*Real*>) Direction of the third material axis (if applicable)
- E1: (Real) Young's modulus (n1)
- E2: (*Real*) Young's modulus (n2)
- E3: (*Real*) Young's modulus (n3)
- nu12: (*Real*) Poisson's ratio (12)
- nu13: (*Real*) Poisson's ratio (13)
- nu23: (Real) Poisson's ratio (23)

- G12: (*Real*) Shear modulus (12)
- G13: (*Real*) Shear modulus (13)
- G23: (*Real*) Shear modulus (23)

## **B.4** Neohookean (finite strains)

Keyword: neohookean Description here: 4.4.2 Parameters:

- rho: (*Real*) Density
- E: (Real) Young's modulus
- nu: (Real) Poisson's ratio
- **Plane\_Stress**: (*bool*) Plane stress simplification : false = plane strain, true = plane stress (default: false) (only 2D problems)

# **B.5** Standard linear solid

Keyword: sls\_deviatoric Description here: 4.4.3 Parameters:

- **rho**: (*Real*) Density
- E: (*Real*) Young's modulus
- nu: (Real) Poisson's ratio
- Plane\_Stress: (bool) Plane stress simplification (only 2D problems)
- Eta: (*Real*) Viscosity
- Ev: (Real) Stiffness of the viscous element

# **B.6** Elasto-plastic linear isotropic hardening

Keyword: plastic\_linear\_isotropic\_hardening

Description here: 4.4.4

Parameters:

- rho: (*Real*) Density
- E: (*Real*) Young's modulus
- nu: (Real) Poisson's ratio
- h: (Real) Hardening modulus
- sigma\_y: (*Real*) Yielding stress

# B.7 Damage: Marigo

Keyword: marigo Description here: 4.4.5 Parameters:

- rho: (*Real*) Density
- E: (Real) Young's modulus
- nu: (Real) Poisson's ratio
- Plane\_Stress: (bool) Plane stress simplification (only 2D problems)
- Yd: (Random) Rupture criterion
- Sd: (*Real*) Damage Energy

# B.8 Damage: Mazars

Keyword: mazars Description here: 4.4.5 Parameters:

- **rho**: (*Real*) Density
- E: (*Real*) Young's modulus
- nu: (*Real*) Poisson's ratio
- At : (Real) Traction post-peak asymptotic value
- Bt: (Real) Traction decay shape
- Ac: (*Real*) Compression post-peak asymptotic value
- Bc: (*Real*) Compression decay shape
- KO: (Real) Damage threshold
- beta: (*Real*) Shear parameter

# **B.9** Cohesive linear

Keyword: cohesive\_linear

Description here: 4.5.1

Parameters:

- sigma\_c: (*Real*) Critical stress σ<sub>c</sub>
   Either G\_c and kappa or, G\_cI and G\_cII or delta\_c have to be specified
- G\_c: (*Real*) Mode I fracture energy
- **kappa**: (*Real*)  $\kappa = G_cII/G_cI$  parameter (default 1)
- delta\_c: (*Real*) Critical displacement  $\delta_c$
- beta: (*Real*)  $\beta$  parameter (default 1)
- penalty: (*Real*) penalty coefficient for compression  $\alpha$  (optional; default 0)
- volume\_s & m\_s: (*Reals*) optional; to adapt statistical distribution following [22]
- **contact\_after\_breaking**: (*bool*) Activation of contact when the elements are fully damaged (default false)
- max\_quad\_stress\_insertion: (*bool*) Insertion of cohesive element when stress is high enough just on one quadrature point (default false)

# **B.10** Cohesive bilinear

Keyword: cohesive\_bilinear Description here: 4.5.1

Parameters:

- sigma\_c: (*Real*) Critical stress  $\sigma_c$
- delta\_0: (*Real*) Elastic limit displacement  $\delta_0$ 
  - Either G\_c and kappa or, G\_cI and G\_cII or delta\_c have to be specified
- G\_c: (*Real*) Mode I fracture energy
- **kappa**: (*Real*)  $\kappa = G_cII/G_cI$  parameter (default 1)
- delta\_c: (*Real*) Critical displacement  $\delta_c$
- **beta**: (*Real*)  $\beta$  parameter (default 1)
- penalty: (*Real*) Penalty coefficient for compression  $\alpha$  (optional; default 0)

# **B.11** Cohesive linear with friction

Keyword: cohesive\_linear\_friction Description here: 4.5.2 Parameters:

- sigma\_c: (*Real*) Critical stress  $\sigma_c$ Either G\_c and kappa or, G\_cI and G\_cII or delta\_c have to be specified
- G\_c: (*Real*) Mode I fracture energy
- kappa: (*Real*)  $\kappa = G_cII/G_cI$  parameter (default 1)
- delta\_c: (*Real*) Critical displacement  $\delta_c$
- **beta**: (*Real*)  $\beta$  parameter (default 1)
- penalty: (*Real*) Penalty coefficient for compression  $\alpha$  (optional; default 0)
- volume\_s & m\_s: (Reals) optional; to adapt statistical distribution following [22]
- **contact\_after\_breaking**: (*bool*) Activation of contact when the elements are fully damaged (default false)
- max\_quad\_stress\_insertion: (bool) Insertion of cohesive element when stress is high enough just on one quadrature point (default false)
- mu: (*Real*) Maximum attainable value of the friction coefficient,  $\mu$  (default 0)
- penalty\_for\_friction: (*Real*) Penalty parameter for the elasto-plastic friction law (default 0)

# **B.12** Cohesive linear fatigue

Keyword: cohesive\_linear\_fatigue

Description here: 4.5.3

Parameters:

- sigma\_c: (*Real*) Critical stress  $\sigma_c$
- delta\_c: (*Real*) Critical displacement  $\delta_c$
- beta: (*Real*)  $\beta$  parameter (default 1)
- G\_c: (*Real*) Mode I fracture energy
- kappa: (*Real*)  $\kappa$  parameter (default 1)
- penalty: (*Real*) penalty coefficient  $\alpha$  (optional, default 0)
- delta\_f: (*Real*) Characteristic opening displacement  $\delta_f$  (see [15])

# **B.13** Cohesive exponential

Keyword: cohesive\_exponential Description here: 4.5.4

Parameters:

- sigma\_c: (*Real*) Critical stress  $\sigma_c$
- delta\_c: (*Real*) Displacement at the peak traction  $\delta_c$
- **beta**: (*Real*)  $\beta$  parameter (default 1)
- **exponential\_penalty**: (*Bool*) parameter to activate contact penalty following the exponential law (default true)
- **contact\_tangent** : (*Real*) ratio of the contact tangent over the initial exponential tangent (to be defined if exponential\_penalty is false; default 1.0)

# Appendix C

# Package dependencies

During the configuration, **cmake** offers several **Akantu** options which have dependencies with each other or with external packages and software. Each of these are described in details now.

**AKANTU\_USE\_BOOST** Akantu uses Boost header only for preprocessor and Spirit Under Ubuntu (14.04 LTS) the installation can be performed using the commands:

> sudo apt-get install libboost

**AKANTU\_COHESIVE\_ELEMENT** This package activates the cohesive elements engine within Akantu. It depends on:

- A fortran compiler.
- An implementation of BLAS/LAPACK.

Dependencies: AKANTU\_USE\_LAPACK AKANTU\_SOLID\_MECHANICS AKANTU\_CORE AKANTU\_USE\_BOOST

AKANTU\_CORE This package is the core engine of Akantu. It depends on:

- A C++ compiler (GCC  $\geq$  = 4, or Intel).
- The cross-platform, open-source CMake build system.
- The Boost C++ portable libraries.
- The zlib compression library.

Under Ubuntu (14.04 LTS) the installation can be performed using the commands:

> sudo apt-get install **cmake** libboost-dev zlib1g-dev g++

Under Mac OS X the installation requires the following steps:

- Install Xcode
- Install the command line tools.
- Install the MacPorts project which allows to automatically download and install opensource packages.

Then the following commands should be typed in a terminal:

```
> sudo port install cmake gcc48 boost
```

Dependencies: AKANTU\_USE\_BOOST

**AKANTU\_DAMAGE\_NON\_LOCAL** This package activates the non local damage feature of AKANTU *Dependencies: AKANTU\_USE\_LAPACK* 

**AKANTU\_DOCUMENTATION\_MANUAL** This package alows to compile the user manual in the build folder **build/doc/manual/manual.pdf**. Under Ubuntu (14.04 LTS), the installation of the dependencies can be performed using the following command:

```
> sudo apt-get install install rubber texlive texlive-science texlive-latex-
extra
```

**AKANTU\_DUMPERS** This package activates the IOHelper facilities withing Akantu. This is mandatory if you want to be able to output Paraview files as well as any Dumper within Akantu. *Dependencies: AKANTU\_USE\_IOHELPER* 

**AKANTU\_HEAT\_TRANSFER** This package activates the heat transfer model within Akantu. It has no additional dependencies.

**AKANTU\_IMPLICIT** This package activates the sparse solver necessary to solve implicitely static/dynamic finite element problems. It depends on:

- MUMPS, a parallel sparse direct solver.
- Scotch, a graph partitioner.

Dependencies: AKANTU\_USE\_MUMPS

**AKANTU\_USE\_IOHELPER** TODO: No Documentation in iohelper.cmake looking for the sequence:

```
\package_declare_documentation(
   IOHelper
   "documentation text"
)
```

**AKANTU\_USE\_LAPACK** This package provides access to a LAPACK implementation. Under Ubuntu (14.04 LTS), the installation can be performed using the following command:

> sudo apt-get install libatlas-base-dev

**AKANTU\_USE\_MPI** This is a meta package providing access to MPI. Under Ubuntu (14.04 LTS) the installation can be performed using the commands:

> sudo apt-get install libopenmpi-dev

Under Mac OS X the installation requires the following steps:

```
> sudo port install mpich-devel
```

**AKANTU\_USE\_MUMPS** This package enables the MUMPS parallel direct solver for sparce matrices. This is necessary to solve static or implicit problems. Under Ubuntu (14.04 LTS) the installation can be performed using the commands:

> sudo apt-get install libmumps-seq-dev # for sequential > sudo apt-get install libmumps-dev # for parallel

Under Mac OS X the installation requires the following steps:

> sudo port install mumps

If you activate the advanced option AKANTU\_USE\_THIRD\_PARTY\_MUMPS the make system of akantu can automatically compile MUMPS. For this you will have to download MUMPS from http://mumps.enseeiht.fr/ or http://graal.ens-lyon.fr/MUMPS and place it in <a href="mailto:</a> (Akantu source>/third-party

**AKANTU\_USE\_NUMPY** This package allows to wrap Akantu arrays to numpy arrays Under Ubuntu (14.04 LTS) the installation can be performed using the commands:

> sudo apt-get install python-numpy

**AKANTU\_PARALLEL** This option activates the parallel features of AKANTU. *Dependencies: AKANTU\_USE\_MPI\_AKANTU\_USE\_SCOTCH* 

**AKANTU\_PYTHON\_INTERFACE** This package enables the python interface of Akantu. It relies on swig3.0 to generate the code Under Ubuntu (14.04 LTS) the installation can be performed using the commands:

> sudo apt-get install swig3.0

Dependencies: AKANTU\_USE\_PYTHONLIBS AKANTU\_USE\_NUMPY

**AKANTU\_USE\_PYTHONLIBS** This package is a dependency of the python interface Under Ubuntu (14.04 LTS) the installation can be performed using the commands:

> sudo apt-get install libpython2.7-dev

Dependencies: AKANTU\_USE\_NUMPY

**AKANTU\_USE\_SCOTCH** This package enables the use the Scotch library in order to perform a graph partitioning leading to the domain decomposition used within **Akantu** Under Ubuntu (14.04 LTS) the installation can be performed using the commands:

> sudo apt-get install libscotch-dev

If you activate the advanced option AKANTU\_USE\_THIRD\_PARTY\_SCOTCH the make system of akantu can automatically compile Scotch. If the automated download fails due to a SSL access not supported by your version of CMake please download the file scotch\_5.1.12b\_esmumps.tar.gz and then place it in the directory <a href="https://cakantu.source/third-party">cakantu source/third-party</a>

**AKANTU\_SOLID\_MECHANICS** TODO: No Documentation in solid\_mechanics.cmake looking for the sequence:

```
\package_declare_documentation(
   solid\_mechanics
   "documentation text"
   )
```

Dependencies: AKANTU\_CORE AKANTU\_USE\_BOOST

**AKANTU\_STRUCTURAL\_MECHANICS** This package activates the compilation for the Structural Mechanics engine of Akantu *Dependencies: AKANTU\_IMPLICIT AKANTU\_USE\_MUMPS* 

# Index

Constitutive laws, 62 Elements, 16, 71 1D, 16, 71 Segment 2,71 Segment 3, 71 2D, 17, 72 Quadrangle 4, 73 Quadrangle 8,73 Triangle 3,72 Triangle 6,72 3D, 17, 74 Hexahedron 20, 76 Hexahedron 8,75 Pentahedron 15, 79 Pentahedron 6,75 Tetrahedron 10, 74 Tetrahedron 4, 74 Bernoulli, 20 Isoparametric, 16 Structural, 20 FEEngine, 15 Forward Euler, 58 HeatTransferModel assembleCapacityLumped, 58 solveStep(), 59 StableTimeStep, 58 IO, 61 addDumpField, 62 addDumpFieldExternal, 64 addDumpFieldTensor, 63 addDumpFieldVector, 63 DumperParaview, 65 DumperText, 65 registerField, 65 registerFilteredMesh, 65 registerMesh, 65 Material, 33, 44 create a new material, 45

create a new non-local material, 50

Elastic, 34 Neohookean, 36 registerParam, 48 Small-deformation Plasticity, 38 Packages AKANTU\_CORE, 85, 87 AKANTU\_IMPLICIT, 88 AKANTU\_SOLID\_MECHANICS, 85 AKANTU\_USE\_BOOST, 85, 87 AKANTU\_USE\_IOHELPER, 86 AKANTU\_USE\_LAPACK, 85, 86 AKANTU\_USE\_MPI, 87 AKANTU\_USE\_MUMPS, 86, 88 AKANTU\_USE\_NUMPY, 87 AKANTU\_USE\_PYTHONLIBS, 87 AKANTU\_USE\_SCOTCH, 87 Python, 69 SolidMechanicsModel, 21, 26 initFull, 27, 31 setTimeStep, 30, 32 solveStep, 27, 32 StableTimeStep, 31 StructuralMechanicsModel, 55 assembleStiffnessMatrix, 55 solve, 55 updateResidual, 55

# References

- C. Geuzaine and J.F. Remacle, *Gmsh: A 3-D finite element mesh generator with built-in preand post-processing facilities*, International Journal for Numerical Methods in Engineering, 79(11):1309–1331 (2009).
- [2] Simulia ABAQUS FEA.
- [3] TNO DIANA.
- [4] M. Ortiz and A. Pandolfi, *Finite-deformation irreversible cohesive elements for three-dimensional crack-propagation analysis*, International Journal for Numerical Methods in Engineering (IJNME), 44:1267–1282 (1999).
- [5] A. Curnier, Méthodes numériques en mécanique des solides, EPFL (1992).
- [6] T. Hughes and T. Belytschko, *A precis of developments in computational methods for transient analysis*, Journal. of Applied Mechanics (ASME), **50**:1033–1041 (1983).
- [7] B.M. Ted Belytschko, Wing Kam Liu, Nonlinear Finite Elements for Continua and Structures, Wiley (2000).
- [8] J. Simo and T. Hughes, *Computational Inelasticity*, Springer (1992).
- [9] J.J. Marigo, Formulation d'une loi d'endommagement d'un matériau élastique, C. R. Acad. Sci., Paris, Sér. II, **292**(19):1309–1312 (1981).
- [10] J. Lemaitre, A Course on Damage Mechanics, Springer Berlin Heidelberg (1996).
- [11] J. Mazars, Application de la mécanique de l'endommagement au comportement non linéaire et à la rupture du béton de structure, Ph.D. thesis, Université Paris 6 (1984).
- [12] L. Snozzi and J.F. Molinari, A cohesive element model for mixed mode loading with frictional contact capability, International Journal for Numerical Methods in Engineering, 93(5):510–526 (2013).
- [13] G. Camacho and M. Ortiz, Computational modelling of impact damage in brittle materials, International Journal of Solids and Structures, 33(20–22):2899–2938 (1996).
- [14] O. Nguyen, E.A. Repetto, M. Ortiz, and R.A. Radovitzky, A cohesive model of fatigue crack growth, International Journal of Fracture, 110(4):351–369 (2001).
- [15] M. Vocialta, N. Richart, and J.F. Molinari, 3D dynamic fragmentation with parallel dynamic insertion of cohesive elements (2015), submitted to IJNME.
- [16] F. Frey and J. Jirousek, Méthodes des éléments finis: Analyse des structures et milieux continus, Traité de génie civil de l'Ecole Polytechnique Fédérale de Lausanne, PPUR (2009).
- [17] NumPy Fundamental package for scientific computing with Python.
- [18] ParaView Open Source Scientific Visualization.
- [19] VisIt Visualization Tool.
- [20] The MayaVi Data Visualizer.
- [21] SCOTCH: Static Mapping, Graph, Mesh and Hypergraph Partitioning.
- [22] F. Zhou and J.F. Molinari, *Dynamic crack propagation with cohesive elements: a methodology to address mesh dependency*, International Journal for Numerical Methods in Engineering (2004).