Caribou

Release V21.06

Jean-Nicolas Brunet

Mar 08, 2023

CONTENTS

1 Building **3**
- 1.1 Prerequisites - Caribou . . . 3
- 1.2 Prerequisites - SofaCaribou (optional) . . . 3
- 1.3 Prerequisites - SofaPython3 bindings (optional) . . . 4
- 1.4 Compiling . . . 4
- 1.5 Installing python bindings . . . 6

2 <HexahedronElasticForce /> **7**
- 2.1 Quick example . . . 7
- 2.2 Available python bindings . . . 8

3 <TetrahedronElasticForce /> **11**
- 3.1 Quick example . . . 11
- 3.2 Available python bindings . . . 12

4 <HyperelasticForcefield /> **13**
- 4.1 Quick example . . . 13
- 4.2 Available python bindings . . . 14

5 <TractionForcefield /> **15**
- 5.1 Quick example . . . 16
- 5.2 Available python bindings . . . 16

6 <CaribouBarycentricMapping /> **17**
- 6.1 Attributes . . . 17
- 6.2 Quick example . . . 17
- 6.3 Available python bindings . . . 18

7 <CaribouMass /> **19**
- 7.1 Quick example . . . 20
- 7.2 Available python bindings . . . 20

8 <SaintVenantKirchhoffMaterial /> **23**
- 8.1 Quick example . . . 23
- 8.2 Available python bindings . . . 24

9 <NeoHookeanMaterial /> **25**
- 9.1 Quick example . . . 25
- 9.2 Available python bindings . . . 26

10 <BackwardEulerODESolver /> **27**

10.1 Quick example . . . 29
10.2 Available python bindings . . . 29

11 <StaticODESolver /> 31
11.1 Quick example . . . 32
11.2 Available python bindings . . . 33

12 <LegacyStaticODESolver /> 35
12.1 Quick example . . . 36
12.2 Available python bindings . . . 36

13 <ConjugateGradientSolver /> 37
13.1 Quick example . . . 37
13.2 Available python bindings . . . 38

14 <LLTSolver /> 39
14.1 Quick example . . . 39
14.2 Available python bindings . . . 40

15 <LDLTSolver /> 41
15.1 Quick example . . . 41
15.2 Available python bindings . . . 42

16 <LUSolver /> 43
16.1 Quick example . . . 43
16.2 Available python bindings . . . 44

17 <FictitiousGrid /> 45
17.1 Quick examples . . . 47
17.2 Available python bindings . . . 48

18 <CircleIsoSurface /> 51
18.1 Quick example . . . 51
18.2 Available python bindings . . . 52

19 <CylinderIsoSurface /> 53
19.1 Quick example . . . 53
19.2 Available python bindings . . . 54

20 <SphereIsoSurface /> 55
20.1 Quick example . . . 55
20.2 Available python bindings . . . 56

Index 57

The caribou project is aimed at multiphysics computation. It brings a plugin that complements SOFA multiphysics framework. It also provides generic c++ utilities, and SOFA components such as solvers and forcefields.

The project is composed of two modules:

1. The **Caribou library** brings multiple geometric, linear analysis and topological tools that are designed to be as independent as possible from external projects.
2. The **Sofa caribou library** is built on top of the **caribou library**, but brings new components to the SOFA project as a plugin.

CHAPTER

ONE

BUILDING

1.1 Prerequisites - Caribou

At the moment, the only way to install Caribou is by compiling it. The following table states packages dependencies required before starting the compilation.

Package	Type	Ubuntu	Mac OSX	Description
Eigen	**Required**	`sudo apt install libeigen3-dev`	`brew install eigen`	Used everywhere inside Caribou has the main linear algebra library.
Python 3 libs	Optional	`sudo apt install python3-dev`	`brew install python3`	Used for the python bindings of Caribou objects.
pybind11	Optional	`sudo apt install pybind11-dev`	`brew install pybind11`	Used for the python bindings of Caribou objects.
OpenMP	Optional	`sudo apt install libomp-dev`	`brew install libomp`	Used to parallelize the computation of some components.
Intel MKL	Optional	`sudo apt install libmkl-full-dev`	instructions here	Used by Eigen for optimization of some math operations.
Google test suite	Optional	`sudo apt install libgtest-dev`	instructions here	Used for the unit tests of Caribou.
SOFA Framework	Optional	See below for more information		
SofaPython3	Optional	See below for more information		

1.2 Prerequisites - SofaCaribou (optional)

If you want to compile the Caribou's SOFA plugin (alias **SofaCaribou**), you will need to either:

- [**Method 1**] Install the SOFA binaries and headers somewhere on your computer and note its installation directory. Or,
- [**Method 2**] Compile it following the SOFA build documentation. Once it is built, execute the installation by going into the build directory of SOFA (for example, */home/user/sofa/build/master/*), and using the command `cmake --install .`

Once done, export the installation path of SOFA inside the `SOFA_INSTALL` environment variable. For example,

```
$ export SOFA_ROOT="/home/user/sofa/build/master/install"
```

Note: To make sure your SOFA_ROOT is well defined, you can verify that the following file path exists:

```
$SOFA_ROOT/lib/cmake/SofaFramework/SofaFrameworkTargets.cmake
```

1.3 Prerequisites - SofaPython3 bindings (optional)

If you want the **SofaCaribou** python bindings, you will also need to compile the SofaPython3 plugin. This plugin allows you to compile it using two different methods.

- "in-tree" build type, which is, building the plugin along with the compilation of SOFA using the CMAKE_EXTERNAL_DIRECTORIES CMake variable of SOFA. This means that when you compile SOFA, you also compile the SofaPython3 plugin at the same time, using the same build directory. The plugin binaries will be found at the same place as the SOFA ones.
- "out-of-tree" build type, which is, building the plugin in its own build directory, outside of SOFA.

If you used "in-tree" build type, nothing more has to be done. The installation files of the SofaPython3 plugin have been installed alongside the SOFA ones.

If instead you used the "out-of-tree" build type, you will need to install the built files by using the command `cmake --install .` in the build directory of the plugin (similarly to what you have done with SOFA in the last section).

Once done, export the installation path of SofaPython3 inside the SP3_ROOT environment variable. For example, for an out-of-tree build in the */home/user/plugin.SofaPython3* directory:

```
$ export SP3_ROOT="/home/user/plugin.SofaPython3/build/master/install"
```

For an "in-tree" build type, the SP3_ROOT environment variable will be $SOFA_ROOT/plugins/SofaPython3

exactly the same as the SOFA_ROOT environment variable defined earlier.

Note: To make sure your SP3_ROOT is well defined, you can verify that the following file path exists:

```
$SP3_ROOT/lib/cmake/SofaPython3/PluginTargets.cmake
```

1.4 Compiling

All right, at this point you should have everything needed to compile Caribou. If you are also building SofaCaribou and its python bindings, you also have defined the SOFA_ROOT and SP3_ROOT environment variables.

Start by cloning the Caribou source code and create a build directory inside of it.

```
$ git clone https://github.com/jnbrunet/caribou.git
$ cd caribou
$ mkdir build
$ cd build
```

Next, cmake will be use to configure the build option. It is used with the following format: `cmake -DVAR=VALUE ..` where **VAR** is the name of a configuration variable and **VALUE** is the value assigned to the variable. Caribou provides the following configuration variables:

Var	Value	Default	Description
CARIBOU_USE_FLOAT	ON/OFF	OFF	Specify if the floating point type should be float (OFF) or double(ON).
CARIBOU_BUILD_TESTS	ON/OFF	OFF	Whether or not the test suite of Caribou should be build.
CARIBOU_WITH_SOFA	ON/OFF	ON	Compile the Caribou's SOFA plugin (SofaCaribou).
CARIBOU_OPTIMIZE_FOR_NATIVE	ON/OFF	ON	Tell the compiler to optimize Caribou following the architecture of your computer.
CARIBOU_WITH_PYTHON_3	ON/OFF	ON	Compile Caribou's python bindings.
CARIBOU_WITH_MKL	ON/OFF	ON	Compile Caribou with Intel® Math Kernel Library (MKL) support.
CARIBOU_WITH_OPENMP	ON/OFF	ON	Compile Caribou with OpenMP support.
CMAKE_INSTALL_PREFIX	Path	install/	Specify where the built files (following the *make install* command) should be installed.

If you are compiling the Caribou's SOFA plugin, you will also need to tell cmake where it should find it. Caribou will automatically find it by looking inside the `SOFA_ROOT` environment variable. Otherwise, if the `SOFA_ROOT` environment variable cannot be used, setting the cmake variable `CMAKE_PREFIX_PATH` to `$SOFA_ROOT/lib/install/cmake` should also work. The same thing needs to be done with SofaPython3 if you are also compiling Caribou's python bindings *and* if SofaPython3 was compiled out-of-tree. In this case, you can set `CMAKE_PREFIX_PATH` to `$SP3_INSTALL/lib/install/cmake`.

For example, if you want to compile Caribou with MKL support and python bindings:

```
$ cmake -DCARIBOU_WITH_MKL=ON -DCARIBOU_WITH_PYTHON_3=ON ..
```

If you want to compile Caribou with SOFA and python bindings:

```
$ export SOFA_ROOT=/opt/sofa/build/install
$ export SP3_ROOT=/opt/SofaPython3/build/install
$ cmake -DCARIBOU_WITH_PYTHON_3=ON -DCMAKE_PREFIX_PATH="$SP3_ROOT/lib/cmake" ..
```

You can now start the compilation.

```
$ cmake --build . -j4
$ cmake --install .
```

The last command (`cmake --install .`) installed all the built files inside the directory `install` (or the directory specified by the cmake variable `CMAKE_INSTALL_PREFIX` if you changed it). Export this path to the environment variable `CARIBOU_ROOT`:

```
$ export CARIBOU_ROOT="${PWD}/install"
```

Note: To make sure your `CARIBOU_ROOT` is well defined, you can verify that the following file path exists:

```
$CARIBOU_ROOT/lib/cmake/Caribou/CaribouTargets.cmake
```

1.5 Installing python bindings

If you compiled the Caribou's python bindings, and you want them to be found automatically by your python scripts, you can create a symbolic link to the binding directories inside Python's site-package path:

For linux, this can be done with the following command:

```
$ ln -sFfv $(find $CARIBOU_ROOT/lib/python3/site-packages -maxdepth 1 -mindepth 1)
↪$(python3 -m site --user-site)
```

And for Mac OSX:

```
$ ln -sFfv $(find $CARIBOU_ROOT/lib/python3/site-packages -d 1) $(python3 -m site --user-
↪site)
```

You can test that the bindings have been correctly installed by starting a python shell and import Caribou:

```
import Caribou

# Do the following only if you compiled the Caribou's SOFA plugin
import SofaRuntime
import SofaCaribou
```

CHAPTER

TWO

<HEXAHEDRONELASTICFORCE />

Doxygen: `SofaCaribou::forcefield::HexahedronElasticForce`

Implementation of a corotational linear elasticity forcefield for hexahedral topologies.

Requires a mechanical object. Requires a topology container.

Attribute	Format	Default	Description
printLog	bool	false	Output informative messages at the initialization and during the simulation.
youngModulus	float	1000	Young's modulus of the material
poissonRatio	float	0.3	Poisson's ratio of the material
corotated	bool	true	Whether or not to use corotated elements for the strain computation. The rotation is viewed as constant on the element and is extracted at its center point.
topology_container	path		Path to a topology container (or path to a mesh) that contains the hexahedral elements.
integration_method	optional	Regular	Integration method used to integrate the stiffness matrix. • **Regular** Regular 8 points gauss integration (default). • **OnePointGauss** One gauss point integration at the center of the hexahedron

2.1 Quick example

XML

```
<Node>
    <RegularGridTopology name="grid" min="-7.5 -7.5 0" max="7.5 7.5 80" n="9 9 21" />
    <MechanicalObject src="@grid" />
    <HexahedronSetTopologyContainer name="topology" src="@grid" />
    <HexahedronElasticForce topology_container="@topology" youngModulus="3000"␣
```

(continues on next page)

(continued from previous page)

```
↪poissonRatio="0.49" corotated="1" printLog="1" />
</Node>
```

Python

```
node.addObject("RegularGridTopology", name="grid", min=[-7.5, -7.5, 0], max=[7.5, 7.5,␣
↪80], n=[9, 9, 21])
node.addObject("MechanicalObject", src="@grid")
node.addObject("HexahedronSetTopologyContainer", name="topology", src="@grid")
node.addObject("HexahedronElasticForce", topology_container="@topology",␣
↪youngModulus=3000, poissonRatio=0.49, corotated=True, printLog=True)
```

2.2 Available python bindings

class HexahedronElasticForce

class GaussNode

Members

- **weight** : Gauss node's weight. numpy.double
- **jacobian_determinant** : Gauss node's Jacobian determinant. numpy.double
- **dN_dx** : Gauss node's shape derivatives w.r.t. the current position vector. numpy.ndarray
- **F** : Gauss node's strain tensor. numpy.ndarray

gauss_nodes_of(*hexahedron_id*)

Parameters
hexahedron_id (*int*) – Index of the hexahedron in the topology container.

Returns
Reference to the list of Gauss nodes of the element.

Return type
list [*GaussNode*]

Note
No copy involved.

stiffness_matrix_of(*hexahedron_id*)

Parameters
hexahedron_id (*int*) – Index of the hexahedron in the topology container.

Returns
Reference to the elemental 24x24 tangent stiffness matrix

Return type
numpy.ndarray

Note
No copy involved.

Get the elemental 24x24 tangent stiffness matrix of the given hexahedron.

K()

Returns
Reference to the forcefield tangent stiffness matrix

Return type
scipy.sparse.csc_matrix

Note
No copy involved.

Get the tangent stiffness matrix of the force field as a compressed sparse column major matrix.

cond()

Returns
Condition number of the forcefield's tangent stiffness matrix

Return type
numpy.double

eigenvalues()

Returns
Reference to the eigen values of the forcefield's tangent stiffness matrix.

Return type
list[numpy.double]

CHAPTER

THREE

<TETRAHEDRONELASTICFORCE />

Doxygen: `SofaCaribou::forcefield::TetrahedronElasticForce`

Implementation of a corotational linear elasticity forcefield for tetrahedral topologies.

Requires a mechanical object. Requires a topology container.

Attribute	Type	Default	Description
printLog	bool	false	Output informative messages at the initialization and during the simulation.
youngModulus	float	1000	Young's modulus of the material
poissonRatio	float	0.3	Poisson's ratio of the material
corotated	bool	true	Whether or not to use corotated elements for the strain computation. The rotation is viewed as constant on the element and is extracted at its center point.
topology_container	path		Path to a topology container (or path to a mesh) that contains the hexahedral elements.

3.1 Quick example

XML

```
<Node>
    <RegularGridTopology name="grid" min="-7.5 -7.5 0" max="7.5 7.5 80" n="9 9 21" />
    <MechanicalObject src="@grid" />
    <HexahedronSetTopologyContainer name="hexahedral_topology" src="@grid" />
    <TetrahedronSetTopologyContainer name="tetrahedral_topology" />
    <TetrahedronSetTopologyModifier />
    <Hexa2TetraTopologicalMapping input="@hexahedral_topology" output="@tetrahedral_
↪topology" />
    <TetrahedronElasticForce topology_container="@tetrahedral_topology" youngModulus=
↪"3000" poissonRatio="0.49" corotated="1" printLog="1" />
</Node>
```

Python

```
node.addObject("RegularGridTopology", name="grid", min=[-7.5, -7.5, 0], max=[7.5, 7.5,␣
↪80], n=[9, 9, 21])
node.addObject("MechanicalObject", src="@grid")
node.addObject("HexahedronSetTopologyContainer", name="hexahedral_topology", src="@grid")
node.addObject('TetrahedronSetTopologyContainer', name='tetrahedral_topology')
node.addObject('TetrahedronSetTopologyModifier')
node.addObject('Hexa2TetraTopologicalMapping', input='@hexahedral_topology', output=
↪'@tetrahedral_topology')
node.addObject("TetrahedronElasticForce", topology_container="@tetrahedral_topology",␣
↪youngModulus=3000, poissonRatio=0.49, corotated=True, printLog=True)
```

3.2 Available python bindings

None at the moment.

CHAPTER

FOUR

<HYPERELASTICFORCEFIELD />

Doxygen: `SofaCaribou::forcefield::HyperelasticForcefield`

Implementation of an hyperelasticity forcefield for any element type topologies.

Requires a mechanical object. Requires a topology container. Requires a material.

At-tribute	For-mat	De-fault	Description
printLog	bool	false	Output informative messages at the initialization and during the simulation.
enable_multithreading	bool	false	Enable the multithreading computation of the stiffness matrix. Only use this if you have a very large number of threads, otherwise performance might be worse than single threading. When enabled, use the environment variable OMP_NUM_THREADS=N to use N threads.
material	path		Path to a material component.
topology	path		Path to a topology container (or path to a mesh) that contains the elements.
drawScale	float	0.85	Scaling factor for the drawing of elements (between 0 and 1). The factor allows to shrink the element relative to its center point when drawing it.
template	option		The template argument is used to specified the element type on which to compute the hyperelasticity force. By default, the component will try to deduce its element type from the given topology. • **Tetrahedron** - 4 nodes tetrahedral elements • **Tetrahedron10** - 10 nodes tetrahedral elements • **Hexahedron** - 8 nodes hexahedral elements • **Hexahedron20** - 20 nodes hexahedral elements

4.1 Quick example

XML

```
<Node>
    <RegularGridTopology name="grid" min="-7.5 -7.5 0" max="7.5 7.5 80" n="9 9 21" />
    <MechanicalObject src="@grid" />
    <HexahedronSetTopologyContainer name="topology" src="@grid" />
    <SaintVenantKirchhoffMaterial young_modulus="3000" poisson_ratio="0.49" />
    <HyperelasticForcefield topology="@topology" template="Hexahedron" printLog="1" />
</Node>
```

Python

```
node.addObject("RegularGridTopology", name="grid", min=[-7.5, -7.5, 0], max=[7.5, 7.5,␣
↪80], n=[9, 9, 21])
node.addObject("MechanicalObject", src="@grid")
node.addObject("HexahedronSetTopologyContainer", name="topology", src="@grid")
node.addObject("SaintVenantKirchhoffMaterial", young_modulus=3000, poisson_ratio=0.49)
node.addObject("HyperelasticForcefield", topology="@topology", template="Hexahedron",␣
↪printLog=True)
```

4.2 Available python bindings

class HyperelasticForceField

K()

Returns
Reference to the forcefield tangent stiffness matrix

Return type
scipy.sparse.csc_matrix

Note
No copy involved.

Get the tangent stiffness matrix of the force field as a compressed sparse column major matrix.

cond()

Returns
Condition number of the forcefield's tangent stiffness matrix

Return type
numpy.double

eigenvalues()

Returns
Reference to the eigen values of the forcefield's tangent stiffness matrix.

Return type
list[numpy.double]

CHAPTER

FIVE

<TRACTIONFORCEFIELD />

Doxygen: SofaCaribou::forceield::TractionForcefield

Implementation of a traction forcefield for triangle and quad topologies.

Requires a mechanical object. Requires a topology container.

At-tribute	Type	Default	Description
printLog	bool	false	Output informative messages at the initialization and during the simulation.
topology	path		Path to a topology container (or path to a mesh) that contains the elements on which the traction will be applied.
traction	[tx, ty, tz]		Tractive force per unit area (if an incremental load is set by the slope parameter, this is the final load reached after all increments).
slope	float	0	Slope of load increment, the resulting tractive force will be p^t = p^{t-1} + p*slope where p is the traction force passed as a data and p^t is the traction force applied at time step t. If slope = 0, the traction will be constant.
number_of_steps_before_increment	int	1	Number of time steps to wait before adding an increment. This can be used to simulate Newton-Raphson solving steps where increment steps are the Newton iterations.
template	option		The template argument is used to specified the element type on which to compute the traction force. By default, the component will try to deduce its element type from the given topology. • **Triangle** - Linear triangular elements • **Triangle6** - Quadratic triangular elements • **Quad** - Linear quadrangle elements • **Quad8** - Quadratic quadrangle elements
nodalForces	[fx, fy, fz], …]	[0	[OUTPUT] Current nodal forces from the applied traction.
total_load	float	0	[OUTPUT] Accumulated load applied on all the surface area.

5.1 Quick example

XML

```
<Node>
    <RegularGridTopology name="grid" min="-7.5 -7.5 0" max="7.5 7.5 80" n="9 9 21" />
    <MechanicalObject src="@grid" />
    <HexahedronSetTopologyContainer name="topology" src="@grid" />
    <SaintVenantKirchhoffMaterial young_modulus="3000" poisson_ratio="0.49" />
    <HyperelasticForcefield topology="@topology" template="Hexahedron" printLog="1" />

    <BoxROI name="top_roi" box="-7.5 -7.5 79.9 7.5 7.5 80.1" />
    <QuadSetTopologyContainer name="quad_container" quads="@top_roi.quadInROI" />
    <TractionForcefield traction="0 -30 0" slope="0.25" topology="@quad_container"␣
↪printLog="1" />
</Node>
```

Python

```
node.addObject("RegularGridTopology", name="grid", min=[-7.5, -7.5, 0], max=[7.5, 7.5,␣
↪80], n=[9, 9, 21])
node.addObject("MechanicalObject", src="@grid")
node.addObject("HexahedronSetTopologyContainer", name="topology", src="@grid")
node.addObject("SaintVenantKirchhoffMaterial", young_modulus=3000, poisson_ratio=0.49)
node.addObject("HyperelasticForcefield", topology="@topology", template="Hexahedron",␣
↪printLog=True)

node.addObject('BoxROI', name='top_roi', box=[-7.5, -7.5, 79.9, 7.5, 7.5, 80.1])
node.addObject('QuadSetTopologyContainer', name='quad_container', quads='@top_roi.
↪quadInROI')
node.addObject('TractionForcefield', traction=[0, -30, 0], slope=1/increments, topology=
↪'@quad_container', printLog=True)
```

5.2 Available python bindings

None at the moment.

CHAPTER

SIX

<CARIBOUBARYCENTRICMAPPING />

Doxygen: `SofaCaribou::mapping::CaribouBarycentricMapping`

Generic barycentric mapping.

The CaribouBarycentricMapping allows to embed nodes into a containing domain. For each embedded nodes, called mapped nodes, the index of the element (from the container domain) that contains it is stored, paired to the barycentric coordinates of the node within this element. When paired with a mechanical object (mo), each of the mo's node positions will automatically follow the parent element that contains it.

6.1 Attributes

Attribute	Type	Default	Description
topology	path	N/A	Topology that contains the embedding (parent) elements.

6.2 Quick example

Here's an example of a visual model of a cylinder mapped into a rectangular beam. The cylinder is a triangular mesh, while the rectangular beam is a complete Finite Element solution modelled using a quadratic hexahedral mesh.

```
import Sofa, meshio, numpy as np
from pathlib import Path

# FE hexahedral mesh
current_dir = Path(__file__).parent
beam_q2 = meshio.read((current_dir / '..' / 'Validation' / 'meshes' / 'beam_q2.vtu').
↪resolve())

# Mapped surface mesh
cylinder = meshio.read((current_dir / '..' / 'Validation' / 'meshes' / 'cylinder_p1.vtu
↪').resolve())

# Material
young_modulus = 10000
poisson_ratio = 0.49
```

(continues on next page)

(continued from previous page)

```
# Scene creation
def createScene(root):
    root.addObject('RequiredPlugin', pluginName='SofaCaribou SofaBoundaryCondition␣
↪SofaEngine SofaOpenglVisual SofaGeneralVisual')
    root.addObject('VisualStyle', displayFlags='showVisualModels showBehaviorModels')
    root.addObject('StaticODESolver', newton_iterations=10, residual_tolerance_
↪threshold=1e-5, pattern_analysis_strategy="BEGINNING_OF_THE_TIME_STEP")
    root.addObject('LDLTSolver', backend="Pardiso")
    root.addChild('mechanics')

    # Mechanical model of the rectangular beam
    root.mechanics.addObject('MechanicalObject', name='mo', position=(mesh.points + p).
↪tolist(), showObject=True, showObjectScale=5)
    root.mechanics.addObject('CaribouTopology', name='volumetric_topology',␣
↪template=caribou_type, indices=mesh.cells_dict[meshio_type].tolist())
    root.mechanics.addObject('SaintVenantKirchhoffMaterial', young_modulus=young_modulus,
↪ poisson_ratio=poisson_ratio)
    root.mechanics.addObject('HyperelasticForcefield')
    root.mechanics.addObject('BoxROI', name='fixed_roi', box=[p[0]-7.5, p[1]-7.5, p[2]-0.
↪9, p[0]+7.5, p[1]+7.5, p[2]+0.1])
    root.mechanics.addObject('FixedConstraint', indices='@fixed_roi.indices')

    # Visual model of the cylinder mapped inside the parent mechanical beam
    root.mechanics.addChild('visual')
    root.mechanics.visual.addObject('CaribouTopology', name='surface_topology', template=
↪'Triangle', indices=cylinder.cell_dict['triangle'].tolist(), position=cylinder.points.
↪tolist())
    root.mechanics.visual.addObject('OglModel', name='mo', position='@surface_topology.
↪position', triangles='@surface_topology.indices', color='green')
    root.mechanics.visual.addObject('CaribouBarycentricMapping', topology='../volumetric_
↪topology')
```

6.3 Available python bindings

None at the moment.

CHAPTER

SEVEN

<CARIBOUMASS />

Doxygen: SofaCaribou::mass::CaribouMass

Implementation of a consistent Mass matrix. The assembly of this mass matrix takes the form of

$$\boldsymbol{M}_{IK} = \int_{\Omega_e} \rho_0 N_I N_K d\Omega \boldsymbol{I}$$

where I and K are a pair of indices of the element K nodes. Here, ρ_0 is the mass density as the mass per volume unit (ie $\frac{m}{v}$) at the undeformed configuration. Finally, $N_I(\boldsymbol{\Psi})$ is the shape function of the Iboldsymbol{Psi}` relative to the reference (canonical) element.

A diagonal consistent mass matrix is also constructed by scaling down the diagonal terms in a way that the mass is constant within the element. The procedure is the following:

$$\boldsymbol{M}_{II}^{\text{diag}} = s_e M_{II} \boldsymbol{I} \text{ with } M_{II} = \int_e \rho_0 N_I^2 d\Omega$$

With the scaling factor being

$$s_e = \frac{M_e}{\sum_I M_{II}}\ ,\ M_e = \int_e \rho_0 d\Omega$$

See the following book for more information:

Peter Wriggers, Nonlinear finite element methods (2008), DOI: 10.1007/978-3-540-71001-1_2

Requires a mechanical object. Requires a topology container.

Attribute	Type	Default	Description
printLog	bool	false	Output informative messages at the initialization and during the simulation.
lumping	bool	false	Whether or not the mass matrix should be lumped by scaling the diagonal entries such that the mass is constant per element. Note that the lumped matrix is always computed. But this parameter will determine if it (the lumped) matrix should be used to solve the acceleration (a = M^(-1).f).
density	double		Mass density of the material at the undeformed state formulated as the mass per volume unit, ie $\rho_0 = m/v$.
topology	path		Path to a either a SOFA mesh topology container (such as an *HexahedronSetTopologyContainer* or *TetrahedronSetTopologyContainer*) or a CaribouTopology component that contains the elements.
template	option		The template argument is used to specified the element type on which to compute the mass. By default, the component will try to deduce its element type from the given topology. • **Tetrahedron** - 4 nodes tetrahedral elements • **Tetrahedron10** - 10 nodes tetrahedral elements • **Hexahedron** - 8 nodes hexahedral elements • **Hexahedron20** - 20 nodes hexahedral elements

7.1 Quick example

XML

```
<Node>
    <RegularGridTopology name="grid" min="-7.5 -7.5 0" max="7.5 7.5 80" n="9 9 21" />
    <MechanicalObject src="@grid" />
    <HexahedronSetTopologyContainer name="topology" src="@grid" />
    <CaribouMass density="2.5" lumped="true" topology="@topology" />
</Node>
```

Python

```
node.addObject("RegularGridTopology", name="grid", min=[-7.5, -7.5, 0], max=[7.5, 7.5,␣
↪80], n=[9, 9, 21])
node.addObject("MechanicalObject", src="@grid")
node.addObject("HexahedronSetTopologyContainer", name="topology", src="@grid")
node.addObject("CaribouMass", density=2.5, lumped=True, topology="@topology")
```

7.2 Available python bindings

class CaribouMass

M()

Returns
Copy of the consistent mass matrix as a compressed column sparse matrix

Return type
scipy.sparse.csc_matrix

Note
The mass matrix must have been assembled beforehand. See the assemble_mass_matrix() methods to force an assembly.

Get the consistent mass matrix of a topology as a compressed sparse column major matrix.

M_diag()

Returns
Copy of the lumped mass matrix as a compressed column sparse matrix

Return type
scipy.sparse.csc_matrix

Note
The mass matrix must have been assembled beforehand. See the assemble_mass_matrix() methods to force an assembly.

The diagonal lumped mass matrix is constructed by scaling down the diagonal terms in a way that the mass is constant within the element.

assemble(*x*)

Assemble the mass matrix M.

This will force an assembly of the consistent mass matrix. Since the mass matrix is function of the the position vector at rest passed as an nx3 array parameter with n the number of nodes. If x is omitted, it will use the mechanical state vector "restPosition".

A copy of the assembled consistent mass matrix M as a column major sparse matrix can be later obtained using the method M().

CHAPTER

EIGHT

<SAINTVENANTKIRCHHOFFMATERIAL />

Doxygen: `SofaCaribou::material::SaintVenantKirchhoffMaterial`

Implementation of a Saint-Venant-Kirchhoff hyperelastic material.

Attribute	Type	Default	Description
young_modulus	float	1000	Young's modulus of the material.
poisson_ratio	float	0.3	Poisson's ratio of the material.

8.1 Quick example

XML

```
<Node>
    <RegularGridTopology name="grid" min="-7.5 -7.5 0" max="7.5 7.5 80" n="9 9 21" />
    <MechanicalObject src="@grid" />
    <HexahedronSetTopologyContainer name="topology" src="@grid" />
    <SaintVenantKirchhoffMaterial young_modulus="3000" poisson_ratio="0.49" />
    <HyperelasticForcefield topology="@topology" template="Hexahedron" printLog="1" />
</Node>
```

Python

```
node.addObject("RegularGridTopology", name="grid", min=[-7.5, -7.5, 0], max=[7.5, 7.5,␣
↪80], n=[9, 9, 21])
node.addObject("MechanicalObject", src="@grid")
node.addObject("HexahedronSetTopologyContainer", name="topology", src="@grid")
node.addObject("SaintVenantKirchhoffMaterial", young_modulus=3000, poisson_ratio=0.49)
node.addObject("HyperelasticForcefield", topology="@topology", template="Hexahedron",␣
↪printLog=True)
```

8.2 Available python bindings

None at the moment.

CHAPTER

NINE

<NEOHOOKEANMATERIAL />

Doxygen: `SofaCaribou::material::NeoHookeanMaterial`

Implementation of a NeoHookean hyperelastic material.

Attribute	Type	Default	Description
young_modulus	float	1000	Young's modulus of the material.
poisson_ratio	float	0.3	Poisson's ratio of the material.

9.1 Quick example

XML

```
<Node>
    <RegularGridTopology name="grid" min="-7.5 -7.5 0" max="7.5 7.5 80" n="9 9 21" />
    <MechanicalObject src="@grid" />
    <HexahedronSetTopologyContainer name="topology" src="@grid" />
    <NeoHookeanMaterial young_modulus="3000" poisson_ratio="0.49" />
    <HyperelasticForcefield topology="@topology" template="Hexahedron" printLog="1" />
</Node>
```

Python

```
node.addObject("RegularGridTopology", name="grid", min=[-7.5, -7.5, 0], max=[7.5, 7.5,␣
↪80], n=[9, 9, 21])
node.addObject("MechanicalObject", src="@grid")
node.addObject("HexahedronSetTopologyContainer", name="topology", src="@grid")
node.addObject("NeoHookeanMaterial", young_modulus=3000, poisson_ratio=0.49)
node.addObject("HyperelasticForcefield", topology="@topology", template="Hexahedron",␣
↪printLog=True)
```

9.2 Available python bindings

None at the moment.

CHAPTER

TEN

<BACKWARDEULERODESOLVER />

Doxygen: `SofaCaribou::ode::BackwardEulerODESolver`

Implementation of an implicit backward euler solver compatible with non-linear materials.

We are trying to solve to following

$$\boldsymbol{M}\ddot{\boldsymbol{x}} + \boldsymbol{C}\dot{\boldsymbol{x}} + \boldsymbol{R}(\boldsymbol{x}) = \boldsymbol{P}$$

where $\boldsymbol{M}$ is the mass matrix, $\boldsymbol{C}$ is the damping matrix, $\boldsymbol{R}$ is the (possibly non-linear) internal elastic force residual and $\boldsymbol{P}$ is the external force vector (for example, gravitation force or surface traction).

We first transform this second-order differential equation to a first one by introducing two independant variables:

$$\begin{aligned}\boldsymbol{v} &= \dot{\boldsymbol{x}} \\ \boldsymbol{a} &= \ddot{\boldsymbol{x}}\end{aligned}$$

Using the backward Euler scheme, we pose the following approximations:

$$\begin{aligned}\boldsymbol{x}_{n+1} &= \boldsymbol{x}_n + h\boldsymbol{v}_{n+1} \quad (1) \\ \boldsymbol{v}_{n+1} &= \boldsymbol{v}_n + h\boldsymbol{a}_{n+1} \quad (2)\end{aligned}$$

where h is the delta time between the steps n and $n+1$.

Substituting (2) inside (1) gives

$$\begin{aligned}\boldsymbol{x}_{n+1} &= \boldsymbol{x}_n + h\left[\boldsymbol{v}_n + h\boldsymbol{a}_{n+1}\right] \\ &= \boldsymbol{x}_n + h\boldsymbol{v}_n + h^2\boldsymbol{a}_{n+1}\end{aligned}$$

And the problem becomes:

$$\boldsymbol{M}\boldsymbol{a}_{n+1} + \boldsymbol{C}\left[\boldsymbol{v}_n + h\boldsymbol{a}_{n+1}\right] + \boldsymbol{R}(\boldsymbol{x}_n + h\boldsymbol{v}_n + h^2\boldsymbol{a}_{n+1}) = \boldsymbol{P}_n$$

where $\boldsymbol{a}_{n+1}$ is the vector of unknown accelerations.

Finally, assuming $\boldsymbol{R}$ is non-linear in $\boldsymbol{x}_{n+1}$, we iteratively solve for $\boldsymbol{a}_{n+1}$ using the Newton-Raphson method and using the previous approximations to back propagate it inside the velocity and position vectors.

Let i be the Newton iteration number for a given time step n, we pose

$$\begin{aligned}\boldsymbol{F}(\boldsymbol{a}^i_{n+1}) &= \boldsymbol{M}\boldsymbol{a}^i_{n+1} + \boldsymbol{C}\left[\boldsymbol{v}_n + h\boldsymbol{a}^i_{n+1}\right] + \boldsymbol{R}(\boldsymbol{x}_n + h\boldsymbol{v}_n + h^2\boldsymbol{a}^i_{n+1}) - \boldsymbol{P}_n \\ \boldsymbol{J} = \left.\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{a}_{n+1}}\right|_{\boldsymbol{a}^i_{n+1}} &= \boldsymbol{M} + h\boldsymbol{C} + h^2\boldsymbol{K}(\boldsymbol{a}^i_{n+1})\end{aligned}$$

where $\boldsymbol{x}_n$ and $\boldsymbol{x}_n$ are the position and velocity vectors at the beginning of the time step, respectively, and remains constant throughout the Newton iterations.

We then solve for $\boldsymbol{a}_{n+1}^{i+1}$ with

$$\begin{aligned}\boldsymbol{J}\left[\Delta\boldsymbol{a}_{n+1}^{i+1}\right] &= -\boldsymbol{F}(\boldsymbol{a}_{n+1}^{i}) \\ \boldsymbol{a}_{n+1}^{i+1} &= \boldsymbol{a}_{n+1}^{i} + \Delta\boldsymbol{a}_{n+1}^{i+1}\end{aligned}$$

And propagate back the new acceleration using (1) and (2).

In addition, this component implicitly adds a Rayleigh's damping matrix $\boldsymbol{C}_r = r_m\boldsymbol{M} + r_k\boldsymbol{K}(\boldsymbol{x}_{n+1})$. We therefore have

$$\begin{aligned}\boldsymbol{F}(\boldsymbol{a}_{n+1}^{i}) &= \boldsymbol{M}\boldsymbol{a}_{n+1}^{i} + \boldsymbol{C}\left[\boldsymbol{v}_n + h\boldsymbol{a}_{n+1}^{i}\right] + \boldsymbol{R}(\boldsymbol{x}_n + h\boldsymbol{v}_n + h^2\boldsymbol{a}_{n+1}^{i}) - \boldsymbol{P}_n \\ &= \boldsymbol{M}\boldsymbol{a}_{n+1}^{i} + (\boldsymbol{C}_r + \boldsymbol{C})\left[\boldsymbol{v}_n + h\boldsymbol{a}_{n+1}^{i}\right] + \boldsymbol{R}(\boldsymbol{x}_n + h\boldsymbol{v}_n + h^2\boldsymbol{a}_{n+1}^{i}) - \boldsymbol{P}_n \\ &= \boldsymbol{M}\boldsymbol{a}_{n+1}^{i} + (r_m\boldsymbol{M} + r_k\boldsymbol{K})\left[\boldsymbol{v}_n + h\boldsymbol{a}_{n+1}^{i}\right] + \boldsymbol{C}\left[\boldsymbol{v}_n + h\boldsymbol{a}_{n+1}^{i}\right] + \boldsymbol{R}(\boldsymbol{x}_n + h\boldsymbol{v}_n + h^2\boldsymbol{a}_{n+1}^{i}) - \boldsymbol{P}_n \\ &= \left[(1 + hr_m)\boldsymbol{M} + h\boldsymbol{C} + hr_k\boldsymbol{K}\right]\boldsymbol{a}_{n+1}^{i} + \left[r_m\boldsymbol{M} + \boldsymbol{C} + r_k\boldsymbol{K}\right]\boldsymbol{v}_n + \left[\boldsymbol{R}(\boldsymbol{x}_n + h\boldsymbol{v}_n + h^2\boldsymbol{a}_{n+1}^{i}) - \boldsymbol{P}_n\right] \\ \boldsymbol{J} = \left.\frac{\partial\boldsymbol{F}}{\partial\boldsymbol{a}_{n+1}}\right|_{\boldsymbol{a}_{n+1}^{i}} &= (1 + hr_m)\boldsymbol{M} + h\boldsymbol{C} + h(h + r_k)\boldsymbol{K}(\boldsymbol{a}_{n+1}^{i})\end{aligned}$$

Attribute	Type	Default	Description
printLog	bool	false	Output informative messages at the initialization and during the simulation.
rayleigh_stiffness	double	0.0	The stiffness factor r_k used in the Rayleigh's damping matrix $\boldsymbol{D} = r_m\boldsymbol{M} + r_k\boldsymbol{K}$.
rayleigh_mass	double	0.0	The mass factor r_m used in the Rayleigh's damping matrix $\boldsymbol{D} = r_m\boldsymbol{M} + r_k\boldsymbol{K}$.
newton_iterations	int	1	Number of newton iterations between each load increments (normally, one load increment per simulation time-step).
correction_tolerance_threshold	float	1e-5	Relative convergence criterion: The newton iterations will stop when the norm of correction $\frac{\lvert\delta\boldsymbol{u}^k\rvert}{\sum_{i=0}^{k}\lvert\delta\boldsymbol{u}^i\rvert}$ reaches this threshold.
residual_tolerance_threshold	float	1e-5	Relative convergence criterion: The newton iterations will stop when the relative norm of the residual $\frac{\lvert\boldsymbol{R}_k\rvert}{\lvert\boldsymbol{R}_0\rvert}$ at iteration k is lower than this threshold. Use a negative value to disable this criterion.
absolute_residual_tolerance_threshold	float	1e-15	Absolute convergence criterion: The newton iterations will stop when the absolute norm of the residual $\lvert\boldsymbol{R}_k\rvert$ at iteration k is lower than this threshold. This criterion is also used to detect the absence of external forces and skip useless Newton iterations. Use a negative value to disable this criterion.
pattern_analysis_strategy	enum	BEGINNING_OF_THE_TIME_STEP	Define when the pattern of the system matrix should be analyzed to extract a permutation matrix. If the sparsity of the coefficients of the system matrix doesn't change much during the simulation, then this analysis can be computed only one time at the beginning of the simulation. Else, it can be done at the beginning of the time step, or even at each reformation of the system matrix if necessary. The default is to analyze the pattern at each time step. **Options:** • NEVER • BEGINNING_OF_THE_SIMULATION • BEGINNING_OF_THE_TIME_STEP **(default)** • ALWAYS
linearSolver	LinearSolver	None	Linear solver used for the resolution of the system. Will be automatically found in the current context node if none is supplied.
converged	bool	N/A	Whether or not the last call to solve converged.

10.1 Quick example

XML

```
<Node>
    <BackwardEulerODESolver rayleigh_stiffness="0.1" rayleigh_mass="0.1" newton_
↪iterations="10" correction_tolerance_threshold="1e-8" residual_tolerance_threshold="1e-
↪8" printLog="1" />
    <LLTSolver backend="Pardiso" />
</Node>
```

Python

```
node.addObject('BackwardEulerODESolver', rayleigh_stiffness=0.1, rayleigh_mass=0.1,␣
↪newton_iterations=10, correction_tolerance_threshold=1e-8, residual_tolerance_
↪threshold=1e-8, printLog=True)
node.addObject('LLTSolver', backend='Pardiso')
```

10.2 Available python bindings

None at the moment.

CHAPTER

ELEVEN

<STATICODESOLVER />

Doxygen: `SofaCaribou::ode::StaticODESolver`

Implementation of a Newton-Raphson static ODE solver.

The solver does a serie of Newton-Raphson iterations where at each iteration k, the following linear system is solved:

$$\boldsymbol{K}(\boldsymbol{u}^k) \cdot \delta\boldsymbol{u}^{k+1} = -\boldsymbol{R}(\boldsymbol{u}^k)$$
$$\boldsymbol{u}^{k+1} = \boldsymbol{u}^k + \delta\boldsymbol{u}^k$$

where the stiffness matrix $\boldsymbol{K}$ is the derivative of the residual with respect to the displacement, i.e. $\boldsymbol{K} = \frac{\partial \boldsymbol{R}}{\partial \boldsymbol{u}}$ and is typically accumulated by the *addKtoMatrix* method of forcefields. If an iterative linear solver is used, it is possible that the stiffness matrix is never accumulated, instead the operation $\boldsymbol{K}(\boldsymbol{u}^k) \cdot \delta\boldsymbol{u}^{k+1}$ is done through the *addDForce* method of forcefields. The residual vector $\boldsymbol{R}(\boldsymbol{u}^k)$ is accumulated by the *addForce* method of forcefields.

Attribute	Type	Default	Description
printLog	bool	false	Output informative messages at the initialization and during the simulation.
newton_iterations	int	1	Number of newton iterations between each load increments (normally, one load increment per simulation time-step).
correction_tolerance_threshold	float	1e-5	Relative convergence criterion: The newton iterations will stop when the norm of correction $\frac{\|\delta \boldsymbol{u}^k\|}{\sum_{i=0}^{k} \|\delta \boldsymbol{u}^i\|}$ reaches this threshold.
residual_tolerance_threshold	float	1e-5	Relative convergence criterion: The newton iterations will stop when the relative norm of the residual $\frac{\|\boldsymbol{R}_k\|}{\|\boldsymbol{R}_0\|}$ at iteration k is lower than this threshold. Use a negative value to disable this criterion.
absolute_residual_tolerance_threshold	float	1e-5	Absolute convergence criterion: The newton iterations will stop when the absolute norm of the residual $\|\boldsymbol{R}_k\|$ at iteration k is lower than this threshold. This criterion is also used to detect the absence of external forces and skip useless Newton iterations. Use a negative value to disable this criterion.
patternAnalysis	enum	BEGINNING_OF_THE_TIME_STEP	Define when the pattern of the system matrix should be analyzed to extract a permutation matrix. If the sparsity structure of the coefficients of the system matrix doesn't change much during the simulation, then this analysis can be avoided altogether or computed only one time at the beginning of the simulation. Else, it can be done at the beginning of the time step, or even at each reformation of the system matrix if necessary. The default is to analyze the pattern at each time step. **Options:** • NEVER • BEGINNING_OF_THE_SIMULATION • BEGINNING_OF_THE_TIME_STEP **(default)** • ALWAYS
linearSolver	LinearSolver	None	Linear solver used for the resolution of the system. Will be automatically found in the current context node if none is supplied.
converged	bool	N/A	Whether or not the last call to solve converged.

11.1 Quick example

XML

```
<Node>
    <StaticODESolver newton_iterations="10" correction_tolerance_threshold="1e-8"␣
↪residual_tolerance_threshold="1e-8" printLog="1" />
    <LLTSolver backend="Pardiso" />
</Node>
```

Python

```
node.addObject('StaticODESolver', newton_iterations=10, correction_tolerance_
↪threshold=1e-8, residual_tolerance_threshold=1e-8, printLog=True)
node.addObject('LLTSolver', backend='Pardiso')
```

11.2 Available python bindings

`class StaticODESolver`

Variables

- **iteration_times** (*list [int]*) – List of times (in nanoseconds) that each Newton-Raphson iteration took to compute in the last call to Solve().
- **squared_residuals** (list [numpy.double]) – The list of squared residual norms ($|r|^2$) of every newton iterations of the last solve call.
- **squared_initial_residual** (numpy.double) – The initial squared residual ($|r_0|^2$) of the last solve call.

CHAPTER

TWELVE

<LEGACYSTATICODESOLVER />

Doxygen: `SofaCaribou::ode::LegacyStaticODESolver`

Implementation of a Newton-Raphson static ODE solver.

Warning: This component provides a compatibility layer for SOFA's linear solvers. When possible, *<StaticODE-Solver />* should be use with one of Caribou's linear solvers since it provides better performance.

The solver does a series of Newton-Raphson iterations where at each iteration k, the following linear system is solved:

$$\boldsymbol{K}(\boldsymbol{u}^k) \cdot \delta \boldsymbol{u}^{k+1} = -\boldsymbol{R}(\boldsymbol{u}^k)$$
$$\boldsymbol{u}^{k+1} = \boldsymbol{u}^k + \delta \boldsymbol{u}^k$$

where the stiffness matrix $\boldsymbol{K}$ is the derivative of the residual with respect to the displacement, i.e. $\boldsymbol{K} = \frac{\partial \boldsymbol{R}}{\partial \boldsymbol{u}}$ and is typically accumulated by the *addKtoMatrix* method of forcefields. If an iterative linear solver is used, it is possible that the stiffness matrix is never accumulated, instead the operation $\boldsymbol{K}(\boldsymbol{u}^k) \cdot \delta \boldsymbol{u}^{k+1}$ is done through the *addDForce* method of forcefields. The residual vector $\boldsymbol{R}(\boldsymbol{u}^k)$ is accumulated by the *addForce* method of forcefields.

Attribute	Type	Default	Description
printLog	bool	false	Output informative messages at the initialization and during the simulation.
newton_iterations	int	1	Number of newton iterations between each load increments (normally, one load increment per simulation time-step).
correction_tolerance_threshold	float	1e-5	Convergence criterion: The newton iterations will stop when the norm of correction |du| reach this threshold.
residual_tolerance_threshold	float	1e-5	Convergence criterion: The newton iterations will stop when the relative norm of the residual $\frac{\|R_k\|}{\|R_0\|} = \frac{\|f_k - Ku_k\|}{\|f_0 - Ku_0\|}$ is lower than this threshold. Use a negative value to disable this criterion.
should_diverge_when_residual_is_growing	bool	false	Convergence criterion: The newton iterations will stop when the residual is greater than the one from the previous iteration.
warm_start	bool	false	For iterative linear solvers, use the previous solution has a warm start. Note that for the first newton step, the current position is used as the warm start.

12.1 Quick example

XML

```
<Node>
    <LegacyStaticODESolver newton_iterations="10" correction_tolerance_threshold="1e-8"␣
↪residual_tolerance_threshold="1e-8" printLog="1" />
    <ConjugateGradientSolver maximum_number_of_iterations="2500" residual_tolerance_
↪threshold="1e-12" preconditioning_method="Diagonal" printLog="0" />
</Node>
```

Python

```
node.addObject('LegacyStaticODESolver', newton_iterations=10, correction_tolerance_
↪threshold=1e-8, residual_tolerance_threshold=1e-8, printLog=True)
node.addObject('ConjugateGradientSolver', maximum_number_of_iterations=2500, residual_
↪tolerance_threshold=1e-12, preconditioning_method="Diagonal", printLog=False)
```

12.2 Available python bindings

class LegacyStaticODESolver

Variables

- **iteration_times** (*list [int]*) – List of times (in nanoseconds) that each Newton-Raphson iteration took to compute in the last call to Solve().
- **squared_residuals** (list [numpy.double]) – The list of squared residual norms ($|r|^2$) of every newton iterations of the last solve call.
- **squared_initial_residual** (numpy.double) – The initial squared residual ($|r_0|^2$) of the last solve call.
- **iterative_linear_solver_squared_residuals** (list [list [numpy.double]]) – The list of squared residual norms ($|r|^2$) of every iterative linear solver iterations, for each newton iterations of the last solve call.
- **iterative_linear_solver_squared_rhs_norms** (list [numpy.double]) – List of squared right-hand side norms ($|b|^2$) of every newton iterations before the call to the solve method of the iterative linear solver.

CHAPTER

THIRTEEN

<CONJUGATEGRADIENTSOLVER />

Doxygen: `SofaCaribou::solver::ConjugateGradientSolver`

Implementation of a Conjugate Gradient linear solver for selfadjoint (symmetric) positive-definite matrices.

Attribute	Type	Default	Description
printLog	bool	false	Output informative messages at the initialization and during the simulation.
verbose	bool	false	Output convergence status at each iterations of the CG.
maximum_number_of_iterations	int	25	Maximum number of iterations before diverging.
residual_tolerance_threshold	double	1e-5	Convergence criterion: The CG iterations will stop when the residual norm of the residual $\frac{\lvert r_k \rvert}{\lvert r_0 \rvert} = \frac{\lvert r_k - a_k A p_k \rvert}{\lvert b \rvert}$ at iteration k is lower than this threshold (here b is the right-hand side vector).
preconditioning_method	string	None	Preconditioning method used. • **None**: No preconditioning, hence the complete matrix won't be built. **(default)** • **Identity**: A naive preconditioner which approximates any matrix as the identity matrix. This is equivalent as using **None**, except the system matrix is built. • **Diagonal**: Preconditioning using an approximation of A.x = b by ignoring all off-diagonal entries of A. Also called Jacobi preconditioner, work very well on diagonal dominant matrices. This preconditioner is suitable for both selfadjoint and general problems. The diagonal entries are pre-inverted and stored into a dense vector. See here for more details. • **IncompleteCholesky**: Preconditioning based on an modified incomplete Cholesky with dual threshold. See here for more details. • **IncompleteLU**: Preconditioning based on the incomplete LU factorization. See here for more details.

13.1 Quick example

XML

```
<Node>
    <StaticODESolver newton_iterations="10" correction_tolerance_threshold="1e-8"␣
↪residual_tolerance_threshold="1e-8" printLog="1" />
    <ConjugateGradientSolver maximum_number_of_iterations="2500" residual_tolerance_
↪threshold="1e-12" preconditioning_method="Diagonal" printLog="0" />
</Node>
```

Python

```
node.addObject('StaticODESolver', newton_iterations=10, correction_tolerance_
↪threshold=1e-8, residual_tolerance_threshold=1e-8, printLog=True)
node.addObject('ConjugateGradientSolver', maximum_number_of_iterations=2500, residual_
↪tolerance_threshold=1e-12, preconditioning_method="Diagonal", printLog=False)
```

13.2 Available python bindings

class ConjugateGradientSolver

K()

Returns
Reference to the system matrix

Return type
scipy.sparse.csc_matrix

Note
No copy involved.

Get the system matrix A = (mM + bB + kK) as a compressed sparse column major matrix.

assemble(*m*, *b*, *k*)

Parameters

- **m** (*float*) – Factor for the mass (M) matrix.
- **b** (*float*) – Factor for the damping (b) matrix.
- **k** (*float*) – Factor for the stiffness (K) matrix.

Returns
Reference to the system matrix

Return type
scipy.sparse.csc_matrix

Note
No copy involved.

Get the system matrix A = (mM + bB + kK) as a compressed sparse column major matrix.

CHAPTER

FOURTEEN

<LLTSOLVER />

Doxygen: `SofaCaribou::solver::LLTSolver`

Implementation of a sparse Cholesky (LL^T) direct linear solver.

This class provides a LL^T Cholesky factorizations of sparse matrices that are selfadjoint and positive definite. In order to reduce the fill-in, a symmetric permutation P is applied prior to the factorization such that the factorized matrix is PAP^{-1}.

The component uses the Eigen SimplicialLLT class as the solver backend.

Attribute	Format	Default	Description
backend	option	Eigen	**Solver backend to use.** • **Eigen** Eigen LLT solver (SimplicialLLT). **[default]** • **Pardiso** Pardiso LLT solver.

14.1 Quick example

XML

```
<Node>
    <StaticODESolver newton_iterations="10" correction_tolerance_threshold="1e-8"␣
↪residual_tolerance_threshold="1e-8" printLog="1" />
    <LLTSolver backend="Pardiso" />
</Node>
```

Python

```
node.addObject('StaticODESolver', newton_iterations=10, correction_tolerance_
↪threshold=1e-8, residual_tolerance_threshold=1e-8, printLog=True)
node.addObject('LLTSolver', backend="Pardiso")
```

14.2 Available python bindings

None at the moment.

CHAPTER

FIFTEEN

<LDLTSOLVER />

Doxygen: `SofaCaribou::solver::LDLTSolver`

Implementation of a sparse LDL^T linear solver.

Attribute	Format	Default	Description
backend		Eigen	**Solver backend to use.** • **Eigen** Eigen LDLT solver (SimplicialLDLT). **[default]** • **Pardiso** Pardiso LLT solver.

15.1 Quick example

XML

```
<Node>
    <StaticODESolver newton_iterations="10" correction_tolerance_threshold="1e-8"␣
↪residual_tolerance_threshold="1e-8" printLog="1" />
    <LDLTSolver backend="Pardiso" />
</Node>
```

Python

```
node.addObject('StaticODESolver', newton_iterations=10, correction_tolerance_
↪threshold=1e-8, residual_tolerance_threshold=1e-8, printLog=True)
node.addObject('LDLTSolver', backend="Pardiso")
```

15.2 Available python bindings

None at the moment.

CHAPTER

SIXTEEN

<LUSOLVER />

Doxygen: `SofaCaribou::solver::LUSolver`

Implementation of a sparse LU linear solver.

Attribute	Format	Default	Description
backend	option	Eigen	**Solver backend to use.** • **Eigen** Eigen LU solver (SparseLU). **[default]** • **Pardiso** Pardiso LU solver.
symmetric	bool	False	Allows to explicitly state that the system matrix will be symmetric. This will in turn enable various optimizations. This option is only used by the Eigen backend.

16.1 Quick example

XML

```
<Node>
    <StaticODESolver newton_iterations="10" correction_tolerance_threshold="1e-8"␣
↪residual_tolerance_threshold="1e-8" printLog="1" />
    <LUSolver backend="Pardiso" />
</Node>
```

Python

```
node.addObject('StaticODESolver', newton_iterations=10, correction_tolerance_
↪threshold=1e-8, residual_tolerance_threshold=1e-8, printLog=True)
node.addObject('LUSolver', backend="Pardiso")
```

16.2 Available python bindings

None at the moment.

CHAPTER

SEVENTEEN

<FICTITIOUSGRID />

Doxygen: `SofaCaribou::topology::FictitiousGrid`

Implementation of an advanced fictitious (sparse) grid.

An fictitious grid is a regular grid of hexahedral elements that embed an implicit (iso) or explicit (mesh) surface. Elements that lie completely outside the embedded surface are ignored, hence the common name of "sparse" grid. This component allows to retrieve quickly the type (inside, outside or boundary) of a given point location or element. It also provides a recursive subdivision algorithm of the intersected cells that allow an accurate integration of the volume inside the embedded surface.

Requires a topology container or an iso-surface.

Attribute	Format	Default	Description
printLog	bool	false	Output informative messages at the initialization and during the simulation.
template	str	Vec3d	Used to specify the world dimension of the grid. (Vec1D, Vec2D or Vec3D)
n	[nx, ny, nz]		Grid resolution [nx, ny, nz] where nx, ny or nz are the number of nodes in the x,y and z directions.
min	[x, y, z]		First corner node position of the grid's bounding box. If it is not specified, and an explicit embedded surface is given, this will be automatically computed.
max	[x, y, z]		Second corner node position of the grid's bounding box. If it is not specified, and an explicit embedded surface is given, this will be automatically computed.
maximum_number_of_subdivision_levels	int	0	Number of subdivision levels of the boundary cells (one level split the cell in 4 subcells in 2D, and 8 subcells in 3D).
volume_threshold	float	1.0	Ignore (tag as outside) every cells having a volume ratio smaller than this threshold.
isoSurface			Use an implicit surface instead of a tessellated surface. This will be used as a level-set where an iso-value less than zero means the point is inside the boundaries. :note: Cannot be used simultaneously with an explicit surface representation (segments, triangles and quads tesselation)
surface_positions	[[x0, y0, z0], …, [xn, yn, zn]]		Position vector of nodes contained of the explicit embedded surface. :note: Cannot be used simultaneously with an iso_surface.
surface_edges	[[e1p1, e1p2], [e2p1, e2p2], …, [enp1, enp2]]		List of edge's node indices of the explicit embedded surface. :note: Cannot be used simultaneously with an iso_surface. :note: Can only be used in 2D (template="Vec2D")
surface_triangles	[[t1p1, t1p2, t1p3], [t2p1, t2p2, t2p3], …, [tnp1, tnp2, tnp3]]		List of triangle's node indices of the explicit embedded surface. :note: Cannot be used simultaneously with an iso_surface. :note: Can only be used in 3D (template="Vec3D")
surface_quads	[[q1p1, q1p2, q1p3, q1p4], [q2p1, q2p2, q2p3, q2p4],		List of quad's node indices of the explicit embedded surface. :note: Cannot be used simultaneously with an iso_surface. :note: Can only be used in 3D (template="Vec3D")

17.1 Quick examples

17.1.1 Using an implicit surface with level-set

XML

```
<Node>
    <CircleIsoSurface radius="5" center="0 0" />
    <FictitiousGrid name="grid" template="Vec2d" n="4 4" min="-5 -5" max="5 5" maximum_
↪number_of_subdivision_levels="10" draw_boundary_cells="1" draw_outside_cells="1" draw_
↪inside_cells="1" printLog="1" />

    <MechanicalObject template="Vec2d" position="@grid.position" />
    <QuadSetTopologyContainer quads="@grid.quads" />
</Node>
```

Python

```
node.addObject('CircleIsoSurface', radius=5, center=[0, 0])
node.addObject('FictitiousGrid',
               template='Vec2d',
               name='grid',
               n=[4, 4],
               min=[-5, -5],
               max=[+5, +5],
               maximum_number_of_subdivision_levels=10,
               printLog=True,
               draw_boundary_cells=True,
               draw_outside_cells=True,
               draw_inside_cells=True
               )

node.addObject('MechanicalObject', template='Vec2d', position='@grid.position')
node.addObject('QuadSetTopologyContainer', quads='@grid.quads')
```

17.1.2 Using an explicit surface with mesh intersection

XML

```
<Node>
    <MeshVTKLoader name="loader" filename="liver_surface.vtk" />
    <FictitiousGrid name="grid" template="Vec3d" surface_positions="@loader.position"␣
↪surface_triangles="@loader.triangles" n="20 20 20" maximum_number_of_subdivision_
↪levels="4" draw_boundary_cells="1" printLog="1" />

    <MechanicalObject template="Vec3d" position="@grid.position" />
    <HexahedronSetTopologyContainer hexahedrons="@grid.hexahedrons" />
</Node>
```

Python

```
node.addObject('MeshVTKLoader', name='loader', filename='liver_surface.vtk')
node.addObject('FictitiousGrid',
               template='Vec3d',
               surface_positions='@loader.position',
               surface_triangles='@loader.triangles'
               name='grid',
               n=[20, 20, 20],
               maximum_number_of_subdivision_levels=4,
               printLog=True,
               draw_boundary_cells=True
               )

node.addObject('MechanicalObject', template='Vec3d', position='@grid.position')
node.addObject('HexahedronSetTopologyContainer', hexahedrons='@grid.hexahedrons')
```

17.2 Available python bindings

class FictitiousGrid

number_of_cells()

Return type
int

Get the number of **sparse** cells (inside or on the boundary) in the grid.

number_of_nodes()

Return type
int

Get the number of **sparse** nodes (belonging to a **sparse** cell) in the grid.

number_of_subdivisions()

Return type
int

Get the number of subdivisions in the grid.

cell_volume_ratio_distribution(*number_of_decimals=0*)

Parameters
number_of_decimals (*int*) – Round the volume ratio to the given number of decimals. For example, setting this value to 2 will generate a distribution of maximum 100 entries (0.00, 0.01, 0.02, …, 0.99, 1.00). Setting a value at zero deactivate the rounding of volume ratio. Default is 0 which means no rounding.

Returns
A sorted map where the keys are the percentage of volume inside the cell, and the value is a vector containing the ids of all cells having this volume percentage.

Return type
{numpy.float: [int]}

Compute the distribution of volume ratios of the top level cells of the grid.

The volume ratio is the ratio of actual volume of a cell over the total volume of the cell. Hence, the ratio of a cell outside the boundaries is 0, the ratio of a cell inside is 1, and the ratio of boundary cells are between 0 and 1.

CHAPTER

EIGHTEEN

<CIRCLEISOSURFACE />

Doxygen: `SofaCaribou::topology::CircleIsoSurface`

Implicit surface of a circle to be used with level-set compatible components such as the *<FictitiousGrid />*.

Attribute	Format	Default	Description
radius	float	1.0	Radius of the sphere.
center	list	[0, 0, 0]	Coordinates at the center of the sphere.

18.1 Quick example

XML

```
<Node>
    <CircleIsoSurface radius="5" center="0 0" />
    <FictitiousGrid name="grid" template="Vec2d" n="4 4" min="-5 -5" max="5 5" draw_
↪boundary_cells="1" printLog="1" />
</Node>
```

Python

```
node.addObject('CircleIsoSurface', radius=5, center=[0, 0])
node.addObject('FictitiousGrid',
               template='Vec2d',
               name='grid',
               n=[4, 4, 4],
               min=[-5, -5],
               max=[+5, +5],
               printLog=True,
               draw_boundary_cells=True,
               )
```

18.2 Available python bindings

None at the moment.

CHAPTER

NINETEEN

<CYLINDERISOSURFACE />

Doxygen: `SofaCaribou::topology::CylinderIsoSurface`

Implicit surface of a circle to be used with level-set compatible components such as the *<FictitiousGrid />*.

Attribute	Format	Default	Description
radius	float	1.0	Radius of the cylinder.
length	float	1.0	length of the cylinder.
center	list	[0, 0, 0]	Coordinates at the center of cylinder.

19.1 Quick example

XML

```
<Node>
    <CylinderIsoSurface radius="10" length="200" center="0 0 0" />
    <FictitiousGrid name="grid" template="Vec3d" n="9 9 19" min="-5 -5 -100" max="5 5 100
↪" draw_boundary_cells="1" printLog="1" />
</Node>
```

Python

```
node.addObject('CylinderIsoSurface', radius=10, length=200, center=[0, 0, 0])
node.addObject('FictitiousGrid',
               template='Vec3d',
               name='grid',
               n=[9, 9, 19],
               min=[-5, -5, -100],
               max=[+5, +5, +100],
               printLog=True,
               draw_boundary_cells=True,
               )
```

19.2 Available python bindings

None at the moment.

CHAPTER

TWENTY

<SPHEREISOSURFACE />

Doxygen: SofaCaribou::topology::SphereIsoSurface

Implicit surface of a sphere to be used with level-set compatible components such as the *<FictitiousGrid />*.

Attribute	Format	Default	Description
radius	float	1.0	Radius of the sphere.
center	list	[0, 0, 0]	Coordinates at the center of the sphere.

20.1 Quick example

XML

```
<Node>
    <SphereIsoSurface radius="5" center="0 0 0" />
    <FictitiousGrid name="grid" template="Vec3d" n="4 4 4" min="-5 -5 -5" max="5 5 5"␣
↪draw_boundary_cells="1" printLog="1" />
</Node>
```

Python

```
node.addObject('SphereIsoSurface', radius=5, center=[0, 0, 0])
node.addObject('FictitiousGrid',
               template='Vec3d',
               name='grid',
               n=[4, 4, 4],
               min=[-5, -5, -5],
               max=[+5, +5, +5],
               printLog=True,
               draw_boundary_cells=True,
               )
```

20.2 Available python bindings

None at the moment.

INDEX

B

built-in function
    CaribouMass.assemble(), 20
    CaribouMass.M(), 20
    CaribouMass.M_diag(), 20
    ConjugateGradientSolver.assemble(), 38
    ConjugateGradientSolver.K(), 38
    FictitiousGrid.cell_volume_ratio_distribution(), 48
    FictitiousGrid.number_of_cells(), 48
    FictitiousGrid.number_of_nodes(), 48
    FictitiousGrid.number_of_subdivisions(), 48
    HexahedronElasticForce.cond(), 9
    HexahedronElasticForce.eigenvalues(), 9
    HexahedronElasticForce.gauss_nodes_of(), 8
    HexahedronElasticForce.K(), 8
    HexahedronElasticForce.stiffness_matrix_of(), 8
    HyperelasticForceField.cond(), 14
    HyperelasticForceField.eigenvalues(), 14
    HyperelasticForceField.K(), 14

C

CaribouMass (*built-in class*), 20
CaribouMass.assemble()
    built-in function, 20
CaribouMass.M()
    built-in function, 20
CaribouMass.M_diag()
    built-in function, 20
ConjugateGradientSolver (*built-in class*), 38
ConjugateGradientSolver.assemble()
    built-in function, 38
ConjugateGradientSolver.K()
    built-in function, 38

F

FictitiousGrid (*built-in class*), 48
FictitiousGrid.cell_volume_ratio_distribution()
    built-in function, 48
FictitiousGrid.number_of_cells()
    built-in function, 48
FictitiousGrid.number_of_nodes()
    built-in function, 48
FictitiousGrid.number_of_subdivisions()
    built-in function, 48

H

HexahedronElasticForce (*built-in class*), 8
HexahedronElasticForce.cond()
    built-in function, 9
HexahedronElasticForce.eigenvalues()
    built-in function, 9
HexahedronElasticForce.gauss_nodes_of()
    built-in function, 8
HexahedronElasticForce.GaussNode (*built-in class*), 8
HexahedronElasticForce.K()
    built-in function, 8
HexahedronElasticForce.stiffness_matrix_of()
    built-in function, 8
HyperelasticForceField (*built-in class*), 14
HyperelasticForceField.cond()
    built-in function, 14
HyperelasticForceField.eigenvalues()
    built-in function, 14
HyperelasticForceField.K()
    built-in function, 14

L

LegacyStaticODESolver (*built-in class*), 36

S

StaticODESolver (*built-in class*), 33