Solfec User Manual

Tomasz Koziara

25th February 2015

Contents

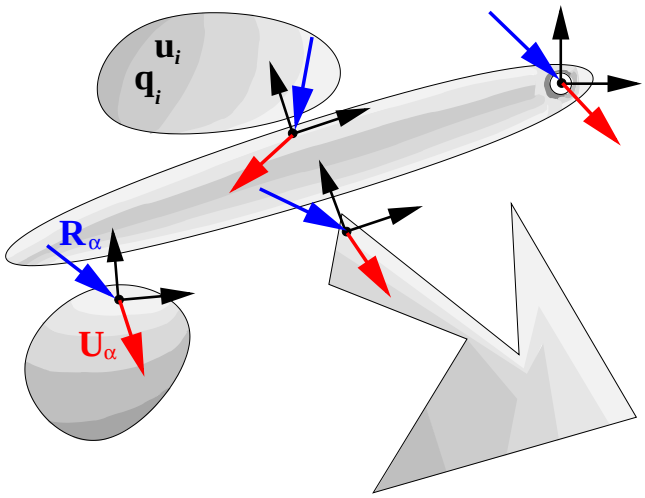
Chapter 1

Introduction

Solfec is a computational code aimed at simulation of multi-body systems with constraints. It implements an instance of the Contact Dynamics (CD) method by Moreau [12] and Jean [7], hence the constraints are handled implicitly. One of the main goals of the software is to provide a user friendly platform for testing formulations and solution methods for the (dynamic) frictional contact problem. It is also meant to serve as a development platform for other aspects of time-stepping methods (e.g. contact detection, time integration). The code implements several kinematic models (rigid, pseudo-rigid, finite element), few contact detection algorithms, several time integrators and a couple of constraint solvers (e.g. penalty, Gauss-Seidel). A distributed memory and a serial versions of the code are available. Solfec is an open-source software and can be downloaded from http://code.google.com/p/solfec.

Basics

It will be useful to introduce some basic notions here. Let us have a look at a figure below



There are four bodies in the figure. Placement of each point of every body is determined by a configuration $\mathbf{q}_i$. Velocity of each point of every body is determined by a velocity $\mathbf{u}_i$. Let $\mathbf{q}$ and $\mathbf{u}$ collect configurations and velocities of all bodies. If the time history of velocity is known, the configuration time history can be computed as

$$\mathbf{q}(t) = \mathbf{q}(0) + \int_0^t \mathbf{u} dt \tag{1.1}$$

The velocity is determined by integrating Newton's law

$$\mathbf{u}(t) = \mathbf{u}(0) + \mathbf{M}^{-1} \int_0^t \left(\mathbf{f} + \mathbf{H}^T \mathbf{R}\right) dt \tag{1.2}$$

where $\mathbf{M}$ is an inertia operator (assumed constant here), $\mathbf{f}$ is an out of balance force, $\mathbf{H}$ is a linear operator, and $\mathbf{R}$ collects some point forces $\mathbf{R}_\alpha$. While integrating the motion of bodies, we keep track of a number of local coordinate systems (local frames). There are four of them in the above figure. Each local frame is related to a pair of points, usually belonging to two distinct bodies. An observer embedded at a local frame calculates the local relative velocity $\mathbf{U}_\alpha$ of one of the points, viewed from the perspective of the other point. Let $\mathbf{U}$ collect all local velocities. Then, we can find a linear transformation $\mathbf{H}$, such that

$$\mathbf{U} = \mathbf{H}\mathbf{u} \tag{1.3}$$

In our case local frames correspond to *constraints*. We influence the local relative velocities by applying local forces $\mathbf{R}_\alpha$. This can be collectively described by an implicit relation

$$\mathbf{C}(\mathbf{U}, \mathbf{R}) = \mathbf{0} \tag{1.4}$$

Hence, in order to integrate equations (1.1) and (1.2), at every instant of time we need to solve the implicit relation (1.4). Here is an example of a numerical approximation of such procedure

$$\mathbf{q}^{t+\frac{h}{2}} = \mathbf{q}^t + \frac{h}{2}\mathbf{u}^t \tag{1.5}$$

$$\mathbf{u}^{t+h} = \mathbf{u}^t + \mathbf{M}^{-1} h \mathbf{f}^{t+\frac{h}{2}} + \mathbf{M}^{-1}\mathbf{H}^T\mathbf{R} \tag{1.6}$$

$$\mathbf{q}^{t+h} = \mathbf{q}^{t+\frac{h}{2}} + \frac{h}{2}\mathbf{u}^{t+h} \tag{1.7}$$

where h is a discrete time step. As the time step h does not appear by $\mathbf{M}^{-1}\mathbf{H}^T\mathbf{R}$, $\mathbf{R}$ should now be interpreted as an impulse (an *integral* of reactions over $[t, t+h]$). At a start we have

$$\mathbf{q}^0 \text{ and } \mathbf{u}^0 \text{ as prescribed initial conditions.} \tag{1.8}$$

The out of balance force

$$\mathbf{f}^{t+\frac{h}{2}} = \mathbf{f}\left(\mathbf{q}^{t+\frac{h}{2}}, t + \frac{h}{2}\right) \tag{1.9}$$

incorporates both internal and external forces. The symmetric and positive-definite inertia operator

$$\mathbf{M} = \mathbf{M}\left(\mathbf{q}^0\right) \tag{1.10}$$

is computed once. The linear operator

$$\mathbf{H} = \mathbf{H}\left(\mathbf{q}^{t+\frac{h}{2}}\right) \tag{1.11}$$

is computed at every time step. The number of rows of $\mathbf{H}$ depends on the number of constraints, while its rank is related to their linear independence. We then compute

$$\mathbf{B} = \mathbf{H}\left(\mathbf{u}^t + \mathbf{M}^{-1} h \mathbf{f}^{t+\frac{h}{2}}\right) \tag{1.12}$$

and

$$\mathbf{W} = \mathbf{H}\mathbf{M}^{-1}\mathbf{H}^T \tag{1.13}$$

which is symmetric and semi-positive definite. The linear transformation

$$\mathbf{U} = \mathbf{B} + \mathbf{W}\mathbf{R} \tag{1.14}$$

maps constraint reactions $\mathbf{R}$ into local relative velocities $\mathbf{U} = \mathbf{H}\mathbf{u}^{t+h}$ at time $t+h$. Relation (1.14) will be here referred to as the *local dynamics*. Finally

$$\mathbf{R} \text{ is such that } \mathbf{C}(\mathbf{U}, \mathbf{R}) = \mathbf{C}(\mathbf{B} + \mathbf{W}\mathbf{R}, \mathbf{R}) = \mathbf{C}(\mathbf{R}) = \mathbf{0} \tag{1.15}$$

where $\mathbf{C}$ is a nonlinear and usually nonsmooth operator. A basic Contact Dynamics algorithm can be summarised as follows:

1. Perform first half-step $\mathbf{q}^{t+\frac{h}{2}} = \mathbf{q}^t + \frac{h}{2}\mathbf{u}^t$.
2. Update existing constraints and detect new contact points.
3. Compute $\mathbf{W}$, $\mathbf{B}$.
4. Solve $\mathbf{C}(\mathbf{R}) = \mathbf{0}$.
5. Update velocity $\mathbf{u}^{t+h} = \mathbf{u}^t + \mathbf{M}^{-1}h\mathbf{f}^{t+\frac{h}{2}} + \mathbf{M}^{-1}\mathbf{H}^T\mathbf{R}$.
6. Perform second half-step $\mathbf{q}^{t+h} = \mathbf{q}^{t+\frac{h}{2}} + \frac{h}{2}\mathbf{u}^{t+h}$.

It should be stressed that the above presentation exemplifies only a particular instance of Contact Dynamics. Let us refer the reader to [11, 13, 3, 12, 7, 6, 1, 10, 14, 8] for more details.

Chapter 2

Installation

Although there will be perpetual releases of Solfec with some fixed version numbers, it is best to use the most recent development version of the code. This is because Solfec is in an active "beta" stage of development for the moment. In order to download the most recent sources, you first need to install *Mercurial*. Have a look at http://mercurial.selenic.com/ for instructions. Once the *hg* command is available on your command line, type

```
hg clone https://solfec.googlecode.com/hg/ solfec
```

This will create the directory *solfec* in your current directory. The next thing you need is an ANSI C development environment at your command line. Users of UNIX-like systems (Linux, FreeBSD, Mac OS X, etc.) are in privileged position here. Windows users can cope by installing Cygwin from http://www.cygwin.com/ or Mingw from http://www.mingw.org/.

Solfec is written in C and it uses a simple makefile to get compiled. The file *solfec/Config.mak* needs to be modified on a new machine so that to set up library paths and compilation flags. Let us have a look at the file *solfec/Config.mak*

```
#
# Operating System (WIN32, SOLARIS, LINUX, AIX, IRIX, OSX)
#
OS = OSX
#
# Specify C compiler here
#
CC = cc
#
# Specify C++ compiler
#
CXX = g++
#
# Specify FORTRAN95 compiler and FORTRAN runtime library
#
FC = g95
FCLIB = -L/opt/local/lib/g95/i386-apple-darwin9/4.0.4/ -lf95
#
# Debug or optimized version switch (yes/no)
#
DEBUG = yes
PROFILE = no
```

```
MEMDEBUG = no
GEOMDEBUG = no
PARDEBUG = no
NOTHROW = no
#
# TIMERS (enable/disable detailed solver timings)
#
TIMERS = yes
#
# POSIX
#
POSIX = yes
#
# XDR
#
XDR = no
XDRINC =
XDRLIB =
#
# BLAS
#
BLAS = -L/usr/lib -lblas
#
# LAPACK
#
LAPACK = -L/usr/lib -llapack
#
# Python
#
PYTHON = -I/usr/include/python2.5
PYTHONLIB = -L/usr/lib -lpython2.5
#
# OpenGL (yes/no)
#
OPENGL = yes
GLINC =
GLLIB = -framework GLUT -framework OpenGL
#
# VBO (OPENGL == yes)
#
VBO = yes
#
# MPI (yes/no)
#
MPI = yes
MPICC = mpicc
#
# Zoltan (MPI == yes)
#
ZOLTANINC = -I/Users/tomek/Devel/lib/zoltan/include
ZOLTANLIB = -L/Users/tomek/Devel/lib/zoltan/lib -lzoltan
```

```
#
# Siconos (yes/no)
#
SICONOS = yes
SICONOSINC = -I/usr/local/include/Siconos/Numerics
SICONOSLIB = -L/usr/local/lib -l SiconosNumerics
```

The above configuration works on Mac OS X. Examples for Linux and Cygwin can be found in *solfec/cfg*. What you need is:

- **C, C++, FORTRAN 95** compilers
- **XDR** (standard part of RPC on all Unix-like systems; on MinGW you will need PortableXDR version 4.9.1 with this patch)
- **BLAS** and **LAPACK** libraries (standard on most systems; available from http://www.netlib.org/lapack/)
- **Python** together with development files and libraries
- **OpenGL** libraries and developments files
- **VBO** (Vertex Buffer Object extension of OpenGL for faster rendering; optional)
- **MPI** libraries and development files
- **Zoltan** load balancing library
- **Siconos**[1] contact solvers library (optional)

All of them, but Zoltan, should be already installed on your system or are quite easy to install otherwise. Zoltan on the other hand can be obtained from http://www.cs.sandia.gov/Zoltan/. In case of troubles - use the Solfec mailing list at http://groups.google.com/group/solfec.

Use "DEBUG = yes" most of the time (this is slower but outputs more information in case you would encounter a bug), but for "proper computations" compile optimized code by selecting "DEBUG = no". After you edit the *Config.mak* file, the first compilation should look like

```
cd solfec
make all
```

This will create files *solfec/solfec* and *solec/solfec-mpi*, that is the serial and the parallel versions of the code. For every subsequent code update and compilation you will like to do the following

- Back up your *Config.mak* file. For example

  ```
  cd solfec
  cp Config.mak ..
  ```

- Now update the sources

  ```
  hg pull
  hg update -C
  ```

- Recover your *Config.mak* file

  ```
  cp ../Congig.mak ./
  ```

[1] http://siconos.gforge.inria.fr/HomePage/index.html

- And finally compile again

```
make clean
make all
```

The *solfec/inp* directory contains example input files. If you haven't used the "POSIX = yes" flag, you will need to create all output directories yourself. Normally though this should be done automatically. If you wish to move *solfec*, *solfec-mpi* and the input files to some other directory - you need to do it by hand. I recommend setting the PATH variable so that the *solfec* directory is included. This way some computations not related to development can be done "outside".

Chapter 3

Running Solfec

Solfec is a command line program. It can be run sequentially (command: *solfec*) or in parallel using the MPI runtime environment (command: *solfec-mpi*). Running it without parameters

```
./solfec
```

results in the hint

```
SYNOPSIS: solfec [-v] [-w] [-f] [-g WIDTHxHEIGHT] [-s sub-directory] path
```

The *-v* switch opens the interactive graphical viewer (cf. Figure 3.1). In this mode the user can view the geometrical mode, run or step through analysis, and view results. The right mouse click on the viewer window expands the menu. The *-w* switch forces the computation (write) mode: if some results are present, they will be overwritten. The *-f* switch, together with *-v*, opens the viewer in the wire-frame mode, which requires substantially less memory and can be used to visualize large models. The *-g* switch allows to specify the initial width and height of the viewer window (512 by default). The *-s* switch allows to output or read the results from a sub-directory. This option is useful when one wishes to output results of similar analyses to different sub-directories of a common root directory specified when creating a SOLFEC object (cf. Section 4.2.5). For example, the bellow commands would run a parallel example and output the results into different sub-directories denoted by the number of processors involved in the analysis

```
mpirun -np 4 solfec-mpi -s 4 inp/cubes.py
mpirun -np 16 solfec-mpi -s 16 inp/cubes.py
```

Because the directory *out/cubes* is specified when creating the SOLFEC obejct in the *inp/cubes.py* input file, the above commands result in creation of two output directories: *out/cubes/4* and *out/cubes/16*. One can then view a specific set of results by running

```
./solfec -v -s 16 inp/cubes.py
```

During a parallel run Solfec updates a file named STATE, placed in the output directory of a simulation. It contains statistics relevant to the run, including an estimated time until the end of the simulation. The output directory contains as well a copy of the input file, which makes reading results more self-contained (it is harder to mismatch input and output files this way). An analysis (both serial and parallel) can be stopped at any time by placing a file named STOP in the output directory of a simulation.

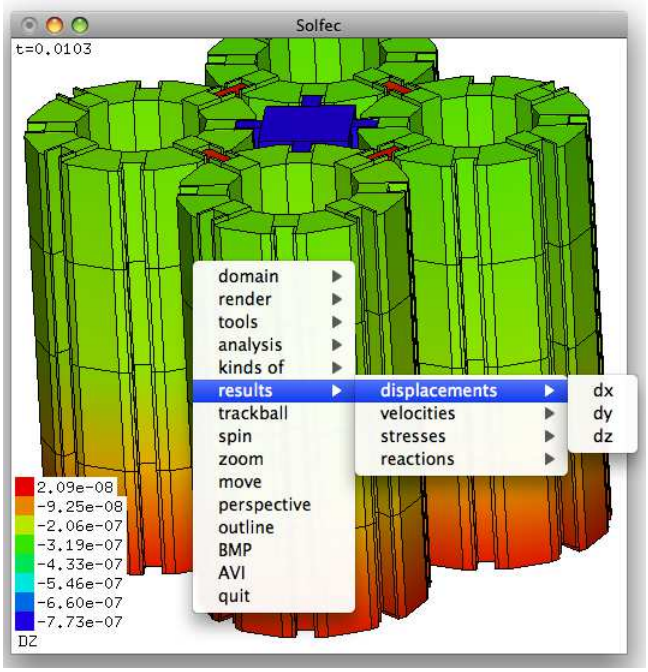


Figure 3.1: Solfec viewer window.

3.1 Read / write mode

Let's say that you have just created an input file 'test.py' in the 'inp' directory. In this input file you have created a SOLFEC object as follows

```
argv = NON_SOLFEC_ARGV()
solfec = SOLFEC ('DYNAMIC', 0.001, 'out/test' + '/' +  argv[0])
```

Hence, the results will be placed in the directory 'out/test/dir1/' + argv[0]. An analysis has not yet been run for this input file and there are no results. If you run

```
./solfec inp/test.py -s dir1 arg1 arg2
```

or

```
mpirun -np 4 ./solfec-mpi inp/test.py -s dir1 arg1 arg2
```

then some results will be written into 'out/test/dir1/arg1'. You can see that command line input parameters can influence the location of the output directory. Since for this first run Solfec did not manage to find any results located at 'out/test/dir1/arg1' the analysis will be pursued and testing

```
if solfec.mode == 'WRITE': print 'WRITE MODE!'
elif solfec.mode == 'READ': print 'READ MODE!'
```

will output 'WRITE MODE!'. The first analysis is done in the write mode. After it has finished you will be able to access the results by again calling

```
./solfec inp/test.py -s dir1 arg1 arg2
```

or

```
./solfec out/test/dir1/arg1.py -s dir1 arg1 arg2
```

since 'out/test/dir1/arg1.py' is an exact copy of 'inp/test.py'. **But crucially note**, that the same parameters '-s dir1 arg1, arg2' need to be passed to Solfec so that the correct output path could be recognized! Of course, on this second run the above if test will result in the output 'READ MODE!'.

Chapter 4

Input language

Solfec input file is essentially a Python source code. Python interpreter is embedded in Solfec. At the same time Solfec extends Python by adding a number of objects and routines. There are few general principles to remember:

- Zero based indexing is observed in routine arguments.
- Parameters after the bar | are optional. For example *FUNCTION (a, b| c, d)* has two optional parameters *c, d*.
- Passing Solfec objects to some routines *empties* them. This means that a variable, that was passed as an argument, no longer stores data. For example: let x = *CREATE1 ()* create an object x, and let y = *CREATE2 (x)* create an object y, using x. If *CREATE2 (x)* empties x, then after the call x becomes an empty placeholder. One can use it to assign value, x = *CREATE1 ()*, but using it as an argument, z = *CREATE2 (x)*, will cause an abnormal termination. One can create a copy of an object by calling z = *COPY (x)*, hence using y = *CREATE2 (COPY (x))* leaves x intact.

Sections below document Solfec objects and routines used for their manipulation.

4.1 Feature maturity matrix

The below table gives indicative assessment of the maturity of various Solfec commands.

	low	average	high
CONVEX		x	
HULL		x	
MESH2CONVEX		x	
MESH		x	
HEX			x
ROUGH_HEX	x		
PIPE			x
TETRAHEDRALIZE	x		
SPHERE			x
ELLIP		x	
SOLFEC			x
SURFACE_MATERIAL		x	
BULK_MATERIAL	x		

BODY		x	
TIME_SERIES		x	
GAUSS_SEIDEL_SOLVER			x
PENALTY_SOLVER		x	
NEWTON_SOLVER		x	
SICONOS_SOLVER	x		
FIX_POINT			x
FIX_DIRECTION			x
SET_DISPLACEMENT	x		
SET_VELOCITY			x
SET_ACCELERATION	x		
PUT_RIGID_LINK			x
GRAVITY			x
FORCE		x	
TORQUE		x	
PRESSURE		x	
SIMPLIFIED_CRACK	x		
RUN			x
OUTPUT			x
EXTENTS			x
CALLBCK		x	
UNPHYSICAL_PENETRARION		x	
GEOMETRIC_EPSILON		x	
WARNINGS			x
INITIALIZE_STATE	x		
IMBALANCE_TOLERANCE		x	
RANK			x
BARRIER			x
NCPU			x
HERE		x	
VIEWER			x
BODY_CHARS		x	
INITIAL_VELOCITY			x
MATERIAL		x	
DELETE		x	
SCALE			x
TRANSLATE			x
ROTATE			x
SPLIT	x		
COPY			x
BEND		x	
BYLABEL			x
MASS_CENTER			x
CONTACT_EXCLUDE_BODIES			x
CONTACT_EXCLUDE_SURFACES		x	
CONTACT_EXCLUDE_OBJECTS	x		
CONTACT_SPARSIFY		x	
LOCDYN_DUMP	x		

OVERLAPPING		x	
MBFC_EXPORT	x		
NON_SOLFEC_ARGV			x
MODAL_ANALYSIS	x		
BODY_MM_EXPORT			x
DISPLAY_POINT		x	
DURATION			x
FORWARD			x
BACKWARD			x
SEEK			x
DISPLACEMENT			x
VELOCITY			x
STRESS			x
ENERGY		x	
TIMING		x	
HISTORY		x	

4.2 Solfec objects

4.2.1 CONVEX

An object of type CONVEX is either an arbitrary convex polyhedron, or it is a collection of such polyhedrons.

obj = CONVEX (vertices, faces, volid | convex)

This routine creates a CONVEX object from a detailed input data.

- **obj** - created CONVEX object
- **vertices** - list of vertices: [*x0, y0, z0, x1, y1, z1, ...*]
- **faces** - list of faces: [*n1, v1, v2, ..., vn1, s1, n2, v1, v2, ..., vn2, s2, ...*], where *n1* is the number of vertices of the first face, *v1, v2, ..., vn1* enumerate the vertices in the CCW order when looking from the outside, and *s1* is the surface identifier of the face. Similarly for the second face and so on.
- **volid** - volume identifier
- **convex** (emptied) - collection of CONVEX objects appending **obj**

Some parameters can also be accessed as members and methods of a CONVEX object. These are

Read-only members and methods
obj.nver - *number of convex vertices*
obj.vertex (n) - *returns a (x, y, z) tuple storing coordinates of nth vertex*

obj = HULL (points, volid, surfid | convex)

This routine creates a CONVEX object as a convex hull of a point set.

- **obj** - created CONVEX object

- **points** - list of points: [*x0, y0, z0, x1, y1, z1, ...*]
- **volid** - volume identifier
- **surfid** - surface identifier common to all faces
- **convex** (emptied) - collection of CONVEX objects appending **obj**

obj = MESH2CONVEX (mesh)

This routine converts a MESH object into a list of CONVEX objects. It can be useful for containing one MESH defining a body shape inside of another background mesh defining deformability (see the background mesh parameters of the BODY object).

- **obj** - created CONVEX object
- **mesh** - input mesh

4.2.2 MESH

An object of type MESH describes an arbitrary volumetric mesh, comprising tetrahedrons, pyramids, wedges, and hexahedrons (Figure 4.1). First order elements are currently supported.

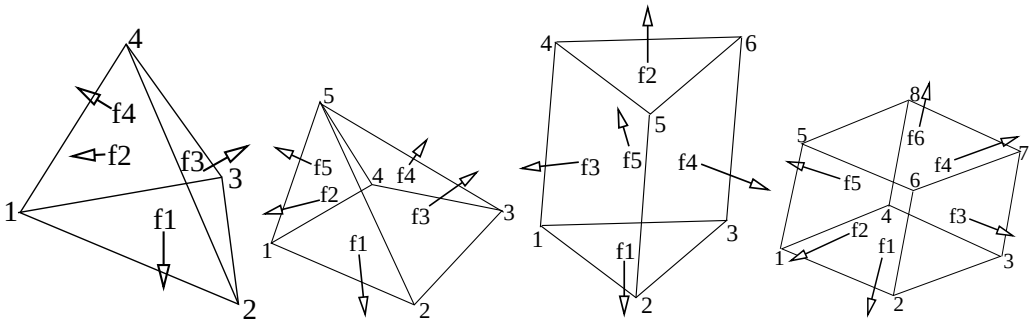


Figure 4.1: Element types in Solfec.

obj = MESH (nodes, elements, surfids)

This routine creates a MESH object from a detailed input data.

- **obj** - created MESH object
- **nodes** - list of nodes: [*x0, y0, z0, x1, y1, z1, ...*]
- **elements** - list of elements: [*e1, n1, n2, ..., ne1, v1, e2, n1, n2, ..., ne2, v2, ...*], where *e1* is the number of nodes of the first element, *n1, n2, ..., ne1* enumerate the element nodes, and *v1* is the volume identifier of the element. Similarly for the second element and so on.
- **surfids** - list of surface identifiers: [*gid, f1, n1, n2, ..., nf1, s1, f2, n1, n2, ..., nf2, s2, ...*], where *gid* is the global surface identifier for all not specified faces, *f1* is the number of nodes in the first specified face, *n1, n2, ..., nf1* enumerate the face nodes, and *s1* is the surface identifier of the face. Similarly for other specified faces. If only the *gid* is given, this can be done either as [*gid*] or as *gid* alone.

Some parameters can also be accessed as members and methods of a MESH object. These are

Read-only members and methods
obj.nnod *- number of mesh nodes*
obj.get_ data() *- return a tuple (*nodes, elements, surfids), *in the same format as for MESH(). Note these are read-only - changing the returned lists does not affect the mesh.*

Read-write members and methods
obj.node (n | x, y, z) *- returns a (x, y, z) tuple storing coordinates of nth node; if x, y or z are given the current coordinates are overwritten*
obj.nodes_ on_ surface (surfid) - returns a list of node numbers belonging to the given surface; *None* object is returned if the list is empty.
obj.set_ volid(volid) - set all elements to have the given volume ID. Returns the volid set. This is mostly useful for being able to distinguish bodies in the Viewer, using menu Kinds of -> Volumes. Note that materials are also assigned by volume ID.

obj = HEX (nodes, i, j, k, volid, surfids | dx, dy, dz)

This routine creates a MESH object corresponding to a hexahedral shape (hexahedral elements are used).

- **obj** - created MESH object
- **nodes** - list of 8 nodes: [*x0, y0, z0, x1, y1, z1, ...*, x7, y7, z7]. The hexahedral shape will be stretched between those nodes using a linear interpolation.
- **i, j, k** - numbers of subdivisions along the local *x, y, z* directions.
- **volid** - volume identifier
- **surfids** - list of six surface identifiers: [*s1, s2, ..., s6*], corresponding to the faces of the hexahedral shape
- **dx, dy, dz** - lists of subdivision schemes along local *x, y, z* directions. By default a subdivision is uniform. When *dx* = [*1, 1, 5, 5, 1, 1*] is present, then this scheme will be normalised (actual numbers do not matter, but their ratios) and applied to the local x direction of the generated shape.

obj = ROUGH_HEX (shape, i, j, k | dx, dy, dz)

This routine creates a hexahedral MESH object corresponding to a given shape. The resultant mesh properly contains the input shape and with its orientation (which is based on the inertia properties of the shape).

- **obj** - created MESH object
- **shape** - an input shape defined by a collection of CONVEX objects; a list of CONVEX objects (or their collections) *[cvx1, cvx2, cvx3, ...]* is as well accepted.
- **i, j, k** - numbers of subdivisions along the local *x, y, z* directions of the principal inertia axes
- **dx, dy, dz** - lists of subdivision schemes along local *x, y, z* directions. By default a subdivision is uniform. When *dx* = [*1, 1, 5, 5, 1, 1*] is present, then this scheme will be normalised (actual numbers do not matter, but their ratios) and applied to the local x direction of the generated shape.

obj = PIPE (pnt, dir, rin, thi, ndir, nrad, nthi, volid, surfids)

This routine creates a MESH object corresponding to a pipe (hexahedral elements are used).

- **obj** - created MESH object
- **pnt** - base point tuple *(x, y, z)*

- **dir** - direction tuple *(dx, dy, dz)*; length of the pipe equals to the length of the direction
- **rin** - inner radius
- **thi** - thickness
- **ndir, nrad, nthi** - number of subdivisions along the direction, radius and thickness
- **volid** - volume identifier
- **surfids** - list of four surface identifiers [*s1, s2, s3, s*4] corresponding to the faces of the pape

obj = TETRAHEDRALIZE (shape, path | volume, quality, volid, surfid)

This routine creates a tetrahedral mesh. Tetgen is invoked internally, http://tetgen.berlios.de/.

- **obj** - created MESH object
- **shape** - an input shape can be:
  - another MESH object
  - a path (e.g. 'path/to/file.stl') to an input file supported by Tetgen (e.g. http://tetgen.berlios.de/fformats.html)
- **path** - path to the output file that will store the mesh; when called again and this file is found the mesh will be red from the file rather than generated
- **volume** - maximum element volume (default: not enforced)
- **quality** - value > 1.0 indicating element quality (default: not enforced); values close to 1.0 result in better mesh quality (mesh generation may fail for small values)
- **volid** - volume identifier (default: 0); if only possible it is inherited from the input
- **surfid** - surface identifier (default: 0); if only possible it is inherited from the input

4.2.3 SPHERE

An object of type SPHERE represents a single sphere.

obj = SPHERE (center, radius, volid, surfid)

This routine creates a SPHERE object.

- **obj** - created SPHERE object
- **center** - tuple *(x, y, z)* defining the center
- **radius** - sphere radius
- **volid** - volume identifier
- **surfid** - surface identifier

Some parameters can also be accessed as members of a SPHERE object. These are

Read-only members and methods
obj.center, obj.radius

4.2.4 ELLIP

An object of type ELLIP represents a single ellipsoid.

obj = ELLIP (center, radii, volid, surfid)

This routine creates an ELLIP object.

- **obj** - created ELLIP object
- **center** - tuple *(x, y, z)* defining the center
- **radii** - tuple *(rx, ry, rz)* of ellipsoid radii
- **volid** - volume identifier
- **surfid** - surface identifier

Some parameters can also be accessed as members of a ELLIP object. These are

Read-only members and methods
obj.center, obj.radii
obj.rot - tuple $(v_{1x}, v_{1y}, v_{1z}, v_{2x}, v_{2y}, v_{2z}, v_{3x}, v_{3y}, v_{3z})$ representing a rotation operator from the ellipsoid natural coordinaet (aligned with principal axes) system to the global cooridinate system

4.2.5 SOLFEC

An object of type SOLFEC represents the Solfec algorithm. One can use several SOLFEC objects to run several analyses from a single input file.

obj = SOLFEC (analysis, step, output)

This routine creates a SOLFEC object.

- **obj** - created SOLFEC object
- **analysis** - 'DYNAMIC' or 'QUASI_STATIC' analysis kind
- **step** - initially assumed time step, regarded as an upper bound
- **output** - defines the output **directory** path (**Important note:** if this directory exists and contains valid output data SOLFEC is created in 'READ' mode, otherwise SOLFEC is created in 'WRITE' mode)

Some parameters can also be accessed as members of a SOLFEC object. These are

Read-only members
obj.analysis
obj.time - current time
obj.mode - either 'READ' or 'WRITE' as described above
obj.constraints - list of constraints (cf. Section 4.2.15)
obj.ncon - number of constraints
obj.bodies - list of bodies (cf. Section 4.2.9)
obj.nbod - number of bodies
obj.outpath - output path, including the sub-directory if the *"-s"* command line argument has been passed

Read/write members
obj.step
obj.verbose - either 'ON' or 'OFF' enabling or disabling writing to standard output (default: 'ON')

4.2.6 FIELD

An object of type FIELD represents a three-dimensional, scalar, time dependent field.

obj = FIELD (solfec, field_callback | label, data)

This routine creates a FIELD object.

- **obj** - created FIELD object
- **solfec** - **obj** is created for this SOLFEC object
- **field_callback** - the Python function defining the scalar field:

  value = field_callback (data, x, y, z, t)

  where **data** is the optional user data passed to **FIELD** routine (if **data** is a tuple it will expand the list of parameters to the callback), **x, y, z** are the point coordinates, and **t** is time. The function should return a numeric value of the scalar field at given point and instant of time.
- **label** - label string (default: 'FIELD_*i*', where *i* is incremented for each call)
- **data** - callback routine user data

Some parameters can also be accessed as members of a FIELD object. These are

Read-only members
obj.label

4.2.7 SURFACE_MATERIAL

An object of type SURFACE_MATERIAL represents material properties on the interface between two surfaces. Surfaces identifiers were included in definitions of all geometric objects.

obj = SURFACE_MATERIAL (solfec | surf1, surf2, model, label, friction, cohesion, restitution, spring, dashpot, hpow)

This routine creates a SURFACE_MATERIAL object.

- **obj** - created SURFACE_MATERIAL object
- **solfec** - **obj** is created for this SOLFEC object
- **surf1** - first surface identifier (default: 0)
- **surf2** - second surface identifier (default: 0). If **surf1** or **surf2** (or both) are not specified, a *default* surface material is being defined (one used when a specific surface pairing cannot be found for a new contact point).
- **model** - material model name (default: 'SIGNORINI_COULOMB'), see Table 4.2 and Chapter 8

Model name	Employs variables
'SIGNORINI_COULOMB'	friction, cohesion, restitution (cf. Section 8.1.1)
'SPRING_DASHPOT'	spring, dashpot, friction, cohesion, hpow (cf. Section 8.1.2)

Table 4.2: Surface material models.

- **label** - label string (default: 'SURFACE_MATERIAL_i', where i is incremented for each call)
- **friction** - friction coefficient (default: 0.0)
- **cohesion** - cohesion per unit area (default: 0.0)
- **restitution** - velocity restitution (default: 0.0)
- **spring** - spring stiffness (default: 0.0)
- **dashpot** - dashpot stiffness (default: 0.0); any negative value indicates critical damping
- **hpow** - Hertz's law power (default: 1.0)

Some parameters can also be accessed as members of a SURFACE_MATERIAL object. These are

Read-only members
obj.surf1, obj.surf2, obj.label

Read/write members
obj.model,obj.friction, obj.cohesion, obj.restitution, obj.spring, obj.dashpot

4.2.8 BULK_MATERIAL

An object of type BULK_MATERIAL represents material properties of a volume.

obj = BULK_MATERIAL (solfec| model, label, young, poisson, density, tensile, fields, fracene)

This routine creates a BULK_MATERIAL object.

- **obj** - created BULK_MATERIAL object
- **solfec** - **obj** is created for this SOLFEC object
- **model** - material model name (default: 'KIRCHHOFF'), see Table 4.3 and Chapter 8
- **label** - label string (default: 'BULK_MATERIAL_i', where i is incremented for each call)
- **young** - Young's modulus (default: 1E9)
- **poisson** - Poisson's coefficient (default: 0.25)
- **density** - material density (default: 1E3)
- **tensile** - tensile strength for fracture check (default: ∞)
- **fields** - list *[field1, field1, ..., fieldN]* of FIELD objects (or FIELD object labels) needed by the material model.

Model name	Employs variables
'KIRCHHOFF'	young, poisson, density (cf. Section 8.2.1)

Table 4.3: Bulk material models.

- **fracene** - fracture energy threshold (default: ∞)

Some parameters can also be accessed as members of a BULK_MATERIAL object. These are

Read-only members
obj.model, obj.label

Read/write members
obj.young, obj.poisson, obj.density

4.2.9 BODY

An object of type BODY represents a solid body.

obj = BODY (solfec, kind, shape, material | label, form, mesh, modal)

This routine creates a body.

- **obj** - created BODY object
- **solfec** - **obj** is created for this SOLFEC object
- **kind** - a string: 'RIGID', 'PSEUDO_RIGID', 'FINITE_ELEMENT' or 'OBSTACLE' describing the kinematic model. See Table 4.4.
- **shape** (emptied) - this is can be a CONVEX/MESH/SPHERE/ELLIP object, or a list *[obj1, obj2, ...]*, where each object is of type CONVEX/MESH/SPHERE/ELLIP. If the **kind** is 'FINITE_ELEMENT', then two cases are possible:
  - **shape** is a single MESH object: the mesh describes both the shape and the discretisation of the motion of a body
  - **shape** is solely composed of CONVEX objects: here a separate **mesh** must be given to discretise motion of a body (see the **mesh** argument below)
- **material** - a BULK_MATERIAL object or a label of a bulk material (specifies an initial body-wise material, see also the **MATERIAL (...)** routine in Section 4.6)
- **label** - a label string (no label is assigned by default)
- **form** - valid when **kind** equals 'FINITE_ELEMENT', ignored otherwise (default: 'TL'). This argument specifies a formulation of the finite element method. See Table 4.5.
- **mesh** - optional when **kind** equals 'FINITE_ELEMENT', ignored otherwise. This variable must be a MESH object describing a finite element mesh properly containing the **shape** composed solely of CONVEX objects. This way the 'FINITE_ELEMENT' model allows to handle complicated shapes with less finite elements, e.g. an arbitrary shape could be contained in just one hexahedron.
- **modal** - the modal analysis results outputed by the **MODAL_ANALYSIS** command (or user results in the same format). This argument must be passed if **form** = 'RO', see Table 4.5.

Body kind	Remarks
'OBSTACLE'	A rigid body ignoring external loads and not contributing to contact constraints. Motion of an obstacle can be controlled through single-body constraints. An obstacle-to-obstacle contact is ignored. **Moving obstacles will not correctly work in the quasi-static case (use rigid bodies instead).** Obstacle bodies do generate contact constraints with other non-obstacle bodies.
'RIGID'	A classical rigid body.
'PSEUDO_RIGID'	A simple body with global linear deformation state.
'FINITE_ELEMENT'	A body discretised with finite elements. Only first order elements are supported at present.

Table 4.4: Body kinds.

Some parameters can also be accessed as members of a BODY object. These are

Read-only members
obj.kind, obj.label, obj.material
obj.conf - tuple *(q1, q2, ..., qN)* storing configuration of the body. See Table 4.6.
obj.velo - tuple *(u1, u2, ..., uN)* storing velocity of the body. See Table 4.7.
obj.mass - referential mass of the body
obj.volume - referential volume of the body
obj.center - referential mass center of the body
obj.tensor - referential Euler (pseudo-rigid, finite element kinematics) or inertia tensor (rigid kinematics) of the body
obj.constraints - list of constraints attached to the body (cf. Section 4.2.15)
obj.ncon - number of constraints attached to the body
obj.id - unique identifier

Read/write members
obj.selfcontact - self-contact detection flag (default: 'OFF") taking values 'ON' or 'OFF'.
obj.scheme - time integration scheme (default: 'DEFAULT') used to integrate motion. See Table 4.8.
obj.damping - stiffness proportional damping coefficient (default: 0.0) for the dynamic case (ignored for rigid bodies).
obj.fracturecheck - check fracture criterion for FEM bodies ('ON' or default: 'OFF'). **Under development.**

4.2.10 TIME_SERIES

An object of type TIME_SERIES is a linear spline based on a series of 2-points.

obj = TIME_SERIES (points)

This routine creates a TIME_SERIES object.

- **obj** - created TIME_SERIES object

Formulation	Remarks
'TL'	Total Lagrangian (default)
'BC'	Body co-rotational (one co-rotated frame per body, suitable for stiff bodies)
'RO'	Reduced order, modal, co-rotational approach. The 'DEF_LIM' integration scheme is always used for this formulation (there would be no computational advantage in using 'DEF_EXP' since the system matrix is diagonal anyway).

Table 4.5: Finite element formulations

Body kind	Configuration description
'RIGID'	Column-wise rotation matrix followed by the current mass center.
'PSEUDO_RIGID'	Column-wise deformation gradient followed by the current mass center.
'FINITE_ELEMENT'	Current coordinates x, y, z of mesh nodes.
'OBSTACLE'	Column-wise rotation matrix followed by the current mass center.

Table 4.6: Types of configurations.

- **points** - either a list *[t0, v0, t1, v1, ...]* of points (where $ti < tj$, when $i < j$), or a path to a file storing times and values pairs in format:

```
t0 v0
t1 v1
...
```

4.2.11 GAUSS_SEIDEL_SOLVER

An object of type GAUSS_SEIDEL_SOLVER represents a nonlinear block Gauss-Seidel solver, employed for the calculation of constraint reactions (cf. Section 10.1).

obj = GAUSS_SEIDEL_SOLVER (epsilon, maxiter | meritval, failure, diagepsilon, diagmaxiter, diagsolver, data, callback)

This routine creates a GAUSS_SEIDEL_SOLVER object.

- **obj** - created GAUSS_SEIDEL_SOLVER object
- **epsilon** - relative accuracy of constraint reactions sufficient for termination
- **maxiter** - maximal number of iterations before termination
- **meritval** - constraints satisfaction merit function value sufficient for termination (default: 1, unused), cf. Chapter 9 for more details.
- **failure** - failure (lack of convergence) action (default: 'CONTINUE'). Available failure actions are: 'CONTINUE' (simulation is continued), 'EXIT' (simulation is stopped and Solfec exits), 'CALLBACK' (a callback function is called if it was set or otherwise the 'EXIT' scenario is executed). In all cases ***obj.error*** variable is set up, cf. Table 4.9.

Body kind	Velocity description
'RIGID'	Referential angular velocity followed by the spatial velocity of mass center.
'PSEUDO_RIGID'	Deformation gradient velocity followed by the spatial velocity of mass center.
'FINITE_ELEMENT'	Components x, y, z of spatial velocities of mesh nodes.
'OBSTACLE'	Column-wise rotation matrix followed by the current mass center.

Table 4.7: Types of velocities.

Scheme	Kinematics	Remarks
'DEFAULT'	all	Use a default time integrator regardless of underlying kinematics.
'RIG_POS'	rigid	NEW1 in [9]: explicit, positive energy drift, no momentum conservation
'RIG_NEG'	rigid	NEW2 in [9]: explicit, negative energy drift, exact momentum conservation; **default** for rigid kinematics
'RIG_IMP'	rigid	NEW3 in [9]: semi-explicit, no energy drift and exact momentum conservation
'DEF_EXP'	pseudo-rigid, finite element	Explicit scheme described in Chapter 5 of [8]; **default** for deformable kinematics, energy and momentum conserving, conditionally stable
'DEF_LIM'	pseudo-rigid, finite element	Linearly implicit scheme similar to [15]; energy and momentum conserving, stable for moderate to large steps; **NOTE:** if the time step is too large, artificial negative internal energy increments may be produced in the event of impacts

Table 4.8: Time integration schema.

- **diagepsilon** - diagonal block solver relative accuracy of constraint reactions (default: *min (epsilon, meritval, 1E-4) / 100*)
- **diagmaxiter** - diagonal block solver maximal number of iterations (default: *max (100, maxiter / 100)*)
- **diagsolver** - diagonal block solver kind (default: 'SEMISMOOTH_NEWTON'). Available diagonal solvers are 'SEMISMOOTH_NEWTON', 'PROJECTED_GRADIENT', 'DE_SAXCE_FENG', 'PROJECTED_NEWTON', cf. Chapter 10.
- **data** - data passed to the failure callback function (if this is a tuple it will accordingly expand the parameter list of the callback routine)
- **callback** - failure callback function of form: *value = callback (obj, data)*, where for the returned value equal zero Solfec run is stopped

Some parameters can also be accessed as members of a GAUSS_SEIDEL_SOLVER object. These are

Read-only members
obj.failure
obj.error - current error code, cf. Table 4.9
obj.iters - number of iterations during a last run of solver
obj.rerhist - a list of relative error values for each iteration of the last run
obj.merhist - a list of merit function values for each iteration of the last run

Read/write members
obj.epsilon, obj.maxiter, obj.meritval, obj.diagepsilon, obj.diagmaxiter, obj.diagsolver
obj.reverse - 'ON' or 'OFF' flag switching iteration reversion modes (whether to alternate backward and forward or not, default is 'OFF')
obj.variant - variant of parallel Gauss-Seidel update (default: 'FULL'), cf. Table 4.10. Ignored in sequential mode.
obj.innerloops - number of inner Gauss-Seidel loops per one global step during a parallel run (default: 1). Ignored in sequential mode.

'OK'	No error has occurred
'DIVERGED'	Global iteration loop divergence
'DIAGONAL_DIVERGED'	Diagonal solver iteration loop divergence
'DIAGONAL_FAILED'	Failure of a diagonal solver (e.g. singularity)

Table 4.9: Error codes of GAUSS_SEIDEL_SOLVER object.

'FULL'	Full Gauss-Seidel update as in sequential case. Although the slowest, it works in all cases. It should be noted, that all of the below variants will usually fail for all-rigid-body models.
'MIDDLE_JACOBI'	Jacobi update for off-processor data of **W** matrix blocks communicating with processors of higher and lower colors. Of use for deformable kinematics, where off-diagonal interactions are weaker. The Gauss-Seidel run-time should be halved for large numbers of processors.
'BOUNDARY_JACOBI'	Use Jacobi update for all off-processor data. This approach will fail in most cases. It servers as illustration.

Table 4.10: Variants of parallel Gauss-Seidel update.

4.2.12 PENALTY_SOLVER

An object of type PENALTY_SOLVER represents a penalty based constraint solver (cf. Section 10.2). When in use, all 'SIGNORONI_COULOMB' type contact interfaces are regarded as 'SPRING_DASHPOT' ones. One should then remember about specifying the *spring* value for those.

obj = PENALTY_SOLVER (| variant)

- **obj** - created PENALTY_SOLVER object
- **variant** - 'IMPLICIT' or 'EXPLICIT' normal force computation variant (default: 'IMPLICIT')

4.2.13 NEWTON_SOLVER

Object of type NEWTON_SOLVER represents a projected quasi-Newton constraints solver (cd. Section 10.3). If local dynamics is enabled (locdyn = 'ON') and iterations fail to converge, the Gauss-Seidel solver will be invoked, starting from the previous time step solution. **WARNING:** for the moment NEWTON_SOLVER may not work well for friction > 1.0.

obj = NEWTON_SOLVER (| meritval, maxiter, locdyn, linver, linmaxiter, maxmatvec, epsilon, delta, theta, omega, gsflag)

- **obj** - created NEWTON_SOLVER object
- **meritval** - value of merit function sufficient for termination (default: 1E-8), cf. Chapter 9 for more details
- **maxiter** - iterations bound (default: 1000)
- **locdyn** - 'ON' or 'OFF' deciding whether to fully assemble local dynamics (default: 'ON'); using the 'OFF' value may be more efficient for implicitly integrated FEM bodies with large meshes
- **linver** - 'GMRES' or 'DIAG' being the linear solver kind (default: 'GMRES')
- **limaxiter** - GMRES iterations bound (ignored for **linver** = 'DIAG', default: 10)
- **maxmatvec** - GMRES matrix-vector products bound (default: linmaxiter * maxiter)
- **epsilon** - relative GMRES accuracy (default: 0.25)
- **delta** - non-negative amount of diagonal regularization (used only for **linver** = 'GMRES', default: 0.0); this parameter has a decisive influence on global convergence; for well-conditioned problems it can be very small or zero; for ill-conditioned problems one should pick a value that delivers an overall best convergence behavior; large values will slow down convergence, but stabilize it; small values may destabilize convergence for ill-conditioned problems; **delta** (typically $\ll 1$) should be tuned together with **epsilon** and **linmaxiter**, so that the linear sub-problems are solved only roughly; *since rigorous analysis is still missing for these parameters, please experiment before settling on specific values for a specific problem;*
- **theta** - relaxation parameter greater than 0 and not greater than 1 (used only for **linver** = 'DIAG', default: 0.25); smaller initial theta may improve overall convergence behavior
- **omega** - positive equation smoothing omega (default: **meritval** $\cdot$ 0.01)
- **gsflag** - 'ON' or 'OFF' deciding whether to us Gauss-Seidel iterations in case of failure (default: 'ON')

Some parameters can also be accessed as members of a NEWTON_SOLVER object. These are

Read-only members
obj.iters - number of iterations during a last run of solver
obj.merhist - a list of merit function values for each iteration of the last run
obj.mvhist - a list of matrix-vector products for each iteration of the last run

Read/write members
obj.meritval, obj.maxiter, obj.locdyn, obj.linver, obj.linmaxiter, obj.maxmatvec, obj.epsilon, obj.delta, obj.theta, obj.omega, obj.gsflag

4.2.14 SICONOS_SOLVER

Object of type SICONOS_SOLVER represents the frictional contact solvers available from Siconos[1]. Currently only the the nonlinear Gauss-Seidel solver is enabled, making the SICONOS_SOLVER equivalent to the GAUSS_SEIDEL_SOLVER. **WARNING1:** only contact constraints are supported at this stage. **WARNING2:** velocity restitution is ignored at the moment. **WARNING3:** only the serial version is available. **WARNING4:** Solfec needs to be compiled with Sicons support for this solver to work.

[1] http://siconos.gforge.inria.fr/HomePage/index.html

obj = SICONOS_SOLVER (| epsilon, maxiter, verbose)

- **obj** - created SICONOS_SOLVER object
- **epsilon** - relative accuracy of constraint reactions sufficient for termination (default: 1E-4)
- **maxiter** - iterations bound (default: 1000)
- **verbose** - verbosity flag: 'ON' or 'OFF' (default: 'OFF')

Some parameters can also be accessed as members of a SICONOS_SOLVER object. These are

Read/write members
obj.epsilon, obj.maxiter

4.2.15 CONSTRAINT

An object of type CONSTRAINT represents a constraint and some of its associated data (e.g. constraint reaction). Both user prescribed constraints and contact constraints are represented by an object of the same type.

obj = FIX_POINT (body, point | strength)

This routine creates a fixed point constraint.

- **obj** - created CONSTRAINT object
- **body** - BODY object whose motion is constrained
- **point** - *(x, y, z)* tuple with referential point coordinates
- **strength** - optionally an ultimate magnitude of the reaction force, beyond which the constraint will be deleted (default: infinity)

obj = FIX_DIRECTION (body, point, direction | body2, point2)

This routine fixes the motion of a referential point along a specified spatial direction. If **body2** is given the motion of **point2** along the **direction** convected with the first **body** is fixed.

- **obj** - created CONSTRAINT object
- **body** - BODY object whose motion is constrained
- **point** - *(x, y, z)* tuple with referential point coordinates
- **direction** - *(vx, vy, vz)* tuple with spatial direction components
- **body2** - BODY object whose motion is constrained with respect to the motion of the first **body** (in this case **body** and **body2** can only be either *rigid* or *pseudo-rigid*)
- **point2** - *(x, y, z)* tuple with referential point on **body2**

obj = SET_DISPLACEMENT (body, point, direction, tms)

This routine prescribes a displacement history of a referential point along a specified spacial direction.

- **obj** - created CONSTRAINT object
- **body** - BODY object whose motion is constrained
- **point** - *(x, y, z)* tuple with referential point coordinates
- **direction** - *(vx, vy, vz)* tuple with spatial direction components
- **tms** - TIME_SERIES object with the displacement history

obj = SET_VELOCITY (body, point, direction, value)

This routine prescribes a velocity history of a referential point along a specified spacial direction.

- **obj** - created CONSTRAINT object
- **body** - BODY object whose motion is constrained
- **point** - *(x, y, z)* tuple with referential point coordinates
- **direction** - *(vx, vy, vz)* tuple with spatial direction components
- **value** - a constant value or a TIME_SERIES object with the velocity history

obj = SET_ACCELERATION (body, point, direction, tms)

This routine prescribes an acceleration history of a referential point along a specified spacial direction.

- **obj** - created CONSTRAINT object
- **body** - BODY object whose motion is constrained
- **point** - *(x, y, z)* tuple with referential point coordinates
- **direction** - *(vx, vy, vz)* tuple with spatial direction components
- **tms** - TIME_SERIES object with the acceleration history

obj = PUT_RIGID_LINK (body1, body2, point1, point2 | strength)

This routine creates a rigid link constraints between two referential points of two distinct bodies.

- **obj** - created CONSTRAINT object
- **body1** - BODY object one whose motion is constrained (could be *None* when **body2** is not *None* - then one of the points is fixed "in the air")
- **body2** - BODY object two whose motion is constrained (could be *None* when **body1** is not *None*)
- **point1** - *(x1, y1, z1)* tuple with the first referential point coordinates
- **point2** - *(x2, y2, z2)* tuple with the second referential point coordinates
- **strength** - optionally an ultimate tensile strength if **point1** != **point2**, beyond which the link will be deleted (default: infinity); or ultimate reaction magnitude (**point1** == **point2**)

obj = PUT_SPRING (body1, point1, body2, point2, function, limits)

This routine creates an arbitrary spring between two referential points of two distinct bodies.

- **obj** - created CONSTRAINT object
- **body1** - BODY object one whose motion is constrained
- **point1** - *(x1, y1, z1)* tuple with the first referential point coordinates
- **body2** - BODY object two whose motion is constrained
- **point2** - *(x2, y2, z2)* tuple with the second referential point coordinates
- **function** - Python function callback returning the value of force as the function of stroke: *force = function (stroke, velocity)*, where *stroke = current (|point2 - point1|) - initial (|point2 - point1|)* and *velocity* is the current relative velocity along the current direction of *point2 - point1* (positive if distance increases).
- **limits** - *(smin, smax)* tuple defining stroke limits ($smin \leq 0$ and $smax \geq 0$)

Some parameters can also be accessed as members of a CONSTRAINT object. These are

Read-only members
obj.kind - kind of constraint: 'CONTACT', 'FIXPNT' (fixed point), 'FIXDIR' (fixed direction), 'VELODIR' (prescribed velocity; note that prescribed displacement and acceleration are converted into this case), 'RIGLNK' (rigid link)
obj.R - current average (over time step $[t, t+h]$) constraint reaction in a form of a tuple: *(RT1, RT2, RN)* given with respect to a local base stored at ***obj.base***
obj.U - constraint output relative velocity tuple: *(UT1, UT2, UN)* given with respect to a local base stored at ***obj.base***
obj.V - contact input relative velocity tuple: *(VT1, VT2, VN)* given with respect to a local base stored at ***obj.base***
obj.base - current spatial coordinate system in a form of a tuple: *(eT1x, eT2x, eNx, eT1y, eT2y, eNy, eT1z, eT2z, eNz)* where *x, y, z* components are global
obj.point - current spatial point where the constraint force acts. This is a *(x, y, z)* tuple for all constraint types, but 'RIGLNK' for which this is a *(x1, y1, z1, x2, y2, z2)* tuple.
obj.area - current area for contact constraints or zero otherwise
obj.gap - current gap for contact constraints or zero otherwise
obj.merit - current value of the per-constraint merit function
obj.adjbod - adjacent bodies. This is a tuple *(body1, body2)* of BODY objects for 'CONTACT' and 'RIGLNK' or a single BODY object otherwise.
obj.matlab - surface material label for constraints of kind 'CONTACT', or a *None* object otherwise.
obj.spair - pairing of surfaces *(surf1, surf2)* for contact constraints or *None* object otherwise. The tuple *(surf1, surf2)* corresponds to the surface identifiers for the *(body1, body2)* body pairing returned by ***obj.adjbod***

4.3 Applying loads

Routines listed in this section apply loads.

GRAVITY (solfec, vector)

This routine sets up the gravitational acceleration.

- **solfec** - SOLFEC object for which the acceleration is set up
- **vector** - *(vx, vy, vz)* tuple defining the gravity acceleration. Each entry is a number or a TIME_SERIES object defining the value of the acceleration component.

FORCE (body, kind, point, direction, value | data)

This routine applies a point force to a body.

- **body** - BODY object to which the force is applied
- **kind** - either 'SPATIAL' or 'CONVECTED'; the *spatial* direction remains fixed, while the *convected* one follows deformation
- **point** - *(x, y, z)* tuple with the referential point where the force is applied
- **direction** - *(vx, vy, vz)* tuple defining the direction of force
- **value** - a number, a TIME_SERIES object or a callback routine defining the value of the applied force. In case of a callback routine, the following format is assumed:

  force = value_ callback (data, q, u, time, step)

  where: **data** is the optional user data passed to **FORCE** routine (if **data** is a tuple it will expand the list of parameters to the callback), **q** is the configuration of the body passed to the callback, **u** is the velocity of the body passed to the callback, **time** is the current time passed to the callback and **step** is the current time step passed to the callback. The callback returns a **force** tuple. For rigid body the force reads *(spatial force, spatial torque, referential torque)*, while for other hinds of bodies this is a generalized force of the same dimension as the velocity **u** (power conjugate to it).
- **data** - callback routine user data

TORQUE (body, kind, direction, value)

This routine applies a torque to a *rigid* body.

- **body** - BODY object of kind 'RIGID' to which the torque is applied
- **kind** - either 'SPATIAL' or 'CONVECTED'; the *spatial* direction remains fixed, while the *convected* one follows deformation
- **direction** - *(vx, vy, vz)* tuple defining the direction of torque
- **value** - a number or a TIME_SERIES object defining the value of the applied torque

PRESSURE (body, surfid, value)

This routine applies a constant surface pressure to MESH based bodies.

- **body** - BODY object to which the pressure is applied (the shape has to be composed of a single MESH)
- **surfid** - the integer surface identifier
- **value** - a number or a TIME_SERIES object defining the value of the applied load

Criterion	Parameters	Description
'TENSILE'	ft	Tensile stress at any point of the the crack plane is larger than the tensile strength.

Table 4.11: Cracking criteria.

4.4 Fragmentation (Under development)

Routines listed in this section control fragmentation of bodies.

SIMPLIFIED_CRACK (body, point, normal, surfid, criterion | topoadj, ft, Gf) (Under development)

This routine prescribes a potential body-wise planar crack together with a cracking criterion. Depending on the topological properties of the body shape, creation of a body-wise crack may or may not result in splitting of the body in two parts.

- **body** - a pseudo-rigid or FEM based BODY object
- **point** - crack plane referential point (x, y, z); if **topoadj** = 'ON' then the **point** is meant to belong to the crack tip
- **normal** - crack plane referential normal (n_x, n_y, n_z)
- **surfid** - tuple *(surf1, surf2)* of the two new crack surfaces identifiers; Surface *surf1* has normal (n_x, n_y, n_z).
- **criterion** - cracking criterion, cf. Table 4.11
- **topoadj** - 'ON' or 'OFF' (default: 'OFF'); when 'OFF' the crack will always propagate across the whole body and result in two body fragments; when 'ON' the crack will propagate from the crack tip through the topologically adjacent elements, which may not produce fragmentation;
- **ft** - tensile strength (default: 0.0)
- **Gf** - fracture energy (default: 0.0); currently unused

4.5 Running simulations

Routines listed in this section control the solution process.

RUN (solfec, solver, duration)

This routine runs a simulation.

- **solfec** - SOLFEC object
- **solver** - constraint solver object (e.g. GAUSS_SEIDEL_SOLVER, PENALTY_SOLVER)
- **duration** - duration of analysis. **Note:** this parameter is ignored when an analysis is run in the viewer mode (with -v switch).

OUTPUT (solfec, interval | compression)

This routine specifies the frequency of writing to the output file.

- **solfec** - SOLFEC object
- **interval** - length of the time interval elapsing before consecutive output file writes
- **compression** - output compression mode: 'OFF' (default) or 'ON'. Compressed output files are smaller, although they might not be portable between hardware platforms.

EXTENTS (solfec, extents)

This routine bounds the simulation space. Bodies falling outside of the extents are deleted from the simulation.

- **solfec** - SOLFEC object
- **extents** - *(xmin, ymin, zmin, xmax, ymax, zmax)* tuple

CALLBACK (solfec, interval, data, callback)

This routine defines a callback function, invoked during a run of Solfec every interval of time. A callback routine can interrupt the course of **RUN** command by returning 0.

- **solfec** - SOLFEC object
- **interval** - length of the time interval elapsing before consecutive callback calls
- **data** - data passed to the callback function
- **callback** - callback function of form: *value = callback (data)*, where for the returned value equal zero Solfec run is stopped

UNPHYSICAL_PENETRATION (solfec, depth)

This routine sets a depth of an unphysical interpenetration. Once it is exceeded, the simulation is stopped and a suitable error message printed out.

- **solfec** - SOLFEC object
- **depth** - interpenetration depth bound (default: ∞)

GEOMETRIC_EPSILON (epsilon)

This routine sets a numerical tolerance for geometric tests performed within Solfec. The tolerance is a characteristic distance between two distinct points below which they can be regarded as one.

- **epsilon** - geometrical tolerance (default: 1E-6)

WARNINGS (state)

This routine disables or enables Solfec warnings. It is a good practice to have the warnings enabled and only switch them off after making sure, that they can be ignored.

- **state** - 'ON' or 'OFF' (default: 'ON')

INITIALIZE_STATE (solfec, path, time)

This routine initializes the state of a Solfec object with the state red from an output directory at a given time. It is ignored in the 'READ' mode.

- **solfec** - Solfec object in the 'WRITE' mode
- **path** - path to the output directory containing matching analysis results (**note:** this cannot be the same output directory as for the **solfec** object)
- **time** - time at which the state should be red from the output files

4.6 Utilities

Various utility routines are listed below.

IMBALANCE_TOLERANCE (solfec, tolerance | weightfactor, updatefreq)

This routine sets the imbalance tolerance for parallel balancing of Solfec data. A ratio of maximal to minimal per processor count of objects used. Hence, 1.0 indicates perfect balance, while any ratio > 1.0 indicates an imbalance. Initially imbalance tolerances are all set to 1.3. This routine is ignored during sequential runs.

- **solfec** - SOLFEC object
- **tolerance** - data imbalance tolerance (default: 1.3)
- **weightfactor** - a local dynamics weight factor between 0.0 and 1.0 (default: 1.0). Computational load of local dynamics assembling is best balanced when weightfactor equals 1.0. This however can sometimes result in a poor load balance for contact detection or time integration. Making it smaller than 1.0 can improve the overall balance in such cases.
- **updatefreq** - geometrical domain partitioning is updated every **updatefreq** time steps (default: 10)

num = RANK ()

This routine returns the rank of the CPU that runs the current copy of Solfec.

- **num** - the CPU rank

BARRIER ()

This routine sets up a parallel barrier in the MPI mode (all processes need to meet at it before they can continue). It is ignored in the serial mode.

num = NCPU (solfec)

This routine returns the number CPUs used in the analysis.

- **num** - the number of CPUs
- **solfec** - SOLFEC object

ret = HERE (solfec, object)

This routine tests whether an object is located on the current processor. During parallel runs objects migrate between processors. When calling a function (or a member) for an object not present on the current processor, the call will usually return None or be ignored. Hence, it is convenient to check whether an object resides on the current processor.

- **ret** - *True* or *False*
- **solfec** - SOLFEC object
- **object** - BODY or CONSTRAINT object

obj = VIEWER ()

This routine tests whether the viewer is enabled.

- **obj** - *True* or *False* depending on whether the viewer (*-v* command line option) was enabled

BODY_CHARS (body, mass, volume, center, tensor)

This routine overwrites referential characteristics of a body.

- **body** - BODY object
- **mass** - body mass
- **volume** - body volume
- **center** - *(x, y, z)* mass center
- **tensor** - $(t_{11}, t_{21}, \ldots, t_{33})$ column-wise inertia tensor for a rigid body or Euler tensor otherwise

INITIAL_VELOCITY (body, linear, angular)

This routine applies initial (at time zero) linear and angular (in the sense of rigid motion) velocity to a body.

- **body** - BODY object
- **linear** - linear velocity (v_x, v_y, v_z)
- **angular** - angular velocity $(\omega_x, \omega_y, \omega_z)$

MATERIAL (solfec, body, volid, material)

This routine applies material to a subset of geometric objects with the given volume identifier.

- **solfec** - SOLFEC object
- **body** - BODY object
- **volid** - volume identifier
- **material** - MATERIAL object or material label

DELETE (solfec, object)

This routine deletes a BODY object or a CONSTRAINT object from a SOLFEC object.

- **solfec** - SOLFEC object
- **object** (emptied) - BODY or CONSTRAINT object

obj = SCALE (shape, coefs)

This routine scales a geometrical object or a collection of such objects.

- **obj** - when **shape** is not *(x, y, z)* tuple: same as **shape**, returned for convenience. Otherwise the $(x \cdot coefs[0], y \cdot coefs[1], z \cdot coefs[2])$ tuple.
- **shape** - object, collection of objects, or a list *[a, b, c, ...]* of objects of type CONVEX, MESH, SPHERE, ELLIP. Alternately this can be a single *(x, y, z)* tuple, but then one must use **point = SCALE (point, coefs)** in order to modify the **point** (Python tuples are immutable - they cannot be modified "in place" after creation).
- **coefs** - *(cx, cy, cz)* tuple of scaling factors along each axis

obj = TRANSLATE (shape, vector)

This routine translates a geometrical object or a collection of such objects.

- **obj** - when **shape** is not *(x, y, z)* tuple: same as **shape**, returned for convenience. Otherwise the $(x + vector[0], y + vector[1], z + vector[2])$ tuple.
- **shape** - object, collection of objects, or a list *[a, b, c, ...]* of objects of type CONVEX, MESH, SPHERE, ELLIP. Alternately this can be a single *(x, y, z)* tuple, but then one must use **point = TRANSLATE (point, vector)** in order to modify the **point** (Python tuples are immutable - they cannot be modified "in place" after creation).
- **vector** - *(vx, vy, vz)* tuple defining the translation

obj = ROTATE (shape, point, vector, angle)

This routine rotates a geometrical object or a collection of such objects.

- **obj** - when **shape** is not *(x, y, z)* tuple: same as **shape**, returned for convenience. Otherwise the rotated *(x1, y1, z1)* image of *(x, y, z)*.
- **shape** - object, collection of objects, or a list *[a, b, c, ...]* of objects of type CONVEX, MESH, SPHERE, ELLIP. Alternately this can be a single *(x, y, z)* tuple, but then one must use **point1 = ROTATE (point1, point2, vector, angle)** in order to modify **point1** (Python tuples are immutable - they cannot be modified "in place" after creation).
- **point** - *(px, py, pz)* tuple defining a point passed by the rotation axis
- **vector** - *(vx, vy, vz)* tuple defining a direction of the rotation axis
- **angle** - rotation angle in degrees

(one, two) = SPLIT (shape, point, normal | surfid, topoadj, remesh)

This routine splits a geometrical object (or a collection of objects) by a plane passing by a point. Depending on the topological properties of the body shape and plane position this may or may not result in splitting of the body in two parts.

- **one** - objects placed below the splitting plane (*None* if no objects were placed below the plane)
- **two** - objects placed above the splitting plane (*None* if no objects were placed above the plane, or if the initial shape has not been fragmented in two parts)
- **shape** (emptied) - object, collection of objects, or a list *[a, b, c, ...]* of objects of type CONVEX, SPHERE, ELLIP or MESH
- **point** - *(px, py, pz)* tuple defining a point passed by the splitting plane
- **normal** - *(nx, ny, nz)* tuple defining the splitting plane normal
- **surfid** - *(surf1, surf2)* tuple defining a pair of surface identifier for the two newly created surfaces (default: 0,0). Surface *surf1* has the outward normal *(nx, ny, nz)*.
- **topoadj** - 'ON' or 'OFF' (default: 'OFF'); when 'OFF' the splitting will always propagate across the whole body and result in two body fragments; when 'ON' the splitting will propagate from the input point through the topologically adjacent elements, which may not produce fragmentation;
- **remesh** - 'ON' or 'OFF' (default: 'ON') flag used only for MESH based shapes; when 'ON' mesh splitting away from inter-element boundaries will lead to tetrahedral re-meshing; when 'OFF' it will raise an error.

WARNING: Mesh splitting generates tetrahedral mesh in place of the input one if the splitting plane is not aligned with element boundaries. The meshing is randomized and it may generate different results for the same input. Use TETRAHEDRALIZE in order to refine and save the generated mesh parts. Otherwise you may encounter input/output errors.

[out1, out2, ...] = MESH_SPLIT (mesh, nodeset | surfid)

This routine splits a mesh object along the internal element boundaries whose nodes belong to the given node set. Depending on the topological properties of the mesh this may or may not result in splitting of the mesh in multiple parts.

- **[out1, out2, ...]** - a list of output meshes (*None* if no internal element boundaries in the input mesh were split)
- **mesh** - input MESH object (the input mesh is not modified by this routine)
- **nodeset** - a list of nodes *[n0, n1, n2, ...]* defining the splitting surface (zero based indexing)
- **surfid** - surface identifier for the newly created surfaces (default: 0).

obj = COPY (shape)

This routine makes a copy of input objects.

- **obj** - created collection of copied objects
- **shape** - object, collection of objects, or a list *[a, b, c, ...]* of objects of type CONVEX, MESH, SPHERE, ELLIP

obj = BEND (shape, point, direction, angle)

This routine bends a shape around an axis. The bending is performed from the section of the shape closest to the axis onward. The orientation of the axis direction determines the orientation of the bending according to the right hand rule. Let $\mathbf{q}$ be the closest to the axis mesh node. Then $\mathbf{v} = \mathbf{d} \times (\mathbf{q} - \mathrm{proj}\,(\mathbf{q}))$, where $\mathbf{d}$ is the axis direction and proj[·] projects a point onto the axis. Bending starts from the section containing $\mathbf{q}$ and proceeds in the direction of $\mathbf{v}$.

- **obj** - same as **shape**
- **shape** - object of type MESH
- **point** - axis point
- **direction** - axis direction
- **angle** - positive bending angle in degrees

obj = BYLABEL (solfec, kind, label)

This routine finds a labelled object inside of a SOLFEC object.

- **obj** - returned object (*None* if a labelled object was not found)
- **solfec** - SOLFEC object
- **kind** - labelled object: 'SURFACE_MATERIAL', 'BULK_MATERIAL', 'BODY', 'FIELD'
- **label** - the label string

obj = MASS_CENTER (shape)

This routine calculates the mass center of a geometrical object or a collection of such objects.

- **obj** - *(x, y, z)* tuple storing the mass center
- **shape** - object, collection of objects, or a list *[a, b, c, ...]* of objects of type CONVEX, MESH, SPHERE, ELLIP. Alternately this can be a single BODY object.

CONTACT_EXCLUDE_BODIES (body1, body2)

This routine disables contact detection for a specific pair of bodies. By default contact detection is enabled for all possible body pairs. **NOTE:** *must be invoked on all processors during a parallel run (do not use from within a callback).*

- **body1** - first BODY object
- **body2** - second BODY object

CONTACT_EXCLUDE_SURFACES (solfec, surf1, surf2)

This routine disables contact detection for a specific pair of surfaces. By default contact detection is enabled for all possible surface pairs. **NOTE:** *must be invoked on all processors during a parallel run (do not use from within a callback).*

- **solfec** - SOLFEC object
- **surf1** - first BODY object
- **surf2** - second BODY object

CONTACT_SPARSIFY (solfec, threshold | minarea, mindist)

This routine modifies contact filtering (sparsification) behaviour. Generally speaking, some contact points are filtered out in order to avoid unnecessary dense contact point clusters. If a pair of bodies is connected by two or more contact points, one of the points generated by topologically adjacent entities (elements, convices) will be removed (sparsified) if the ratio of contact areas of is smaller than the prescribed threshold (cf. Section 7.2).

- **solfec** - SOLFEC object
- **threshold** - sparsification threshold (default: 0.01) from within the interval [0, 1]. Zero corresponds to the lack of sparsification.
- **minarea** - minimal contact area (default: 0.0). Contact points with area smaller then **minarea** are dropped.
- **mindist** - minimal distance between distinct contact points (default: GEOMETRIC_EPSILON).

LOCDYN_DUMP (solfec, path)

This routine dumps into a file the most recent state of local dynamics. It is meant for debugging and test purposes, e.g. comparing local dynamics between runs on various processor counts.

- **solfec** - SOLFEC object
- **path** - file path

obj = OVERLAPPING (obstacles, shapes | not, gap)

This routine looks for shapes (not) overlapping the obstacles.

- **obj** - list of shapes (not) ovrelapping the obstacles.
- **obstacles** - object, collection of objects, or a list *[a, b, c, ...]* of objects of type CONVEX, MESH, SPHERE, ELLIP.
- **shapes** (emptied) - object, collection of objects, or a list *[a, b, c, ...]* of objects of type CONVEX, MESH, SPHERE, ELLIP.
- **not** - 'NOT' string.
- **gap** - maximal negative gap.

MBFCP_EXPORT (solfec, path)

This routine exports Solfec model into the MBFCP problem definition format. See http://code.google.com/p/mbfcp/ for details.

- **solfec** - SOLFEC object
- **path** - output path

NON_SOLFEC_ARGV ()

This routine returns all command line arguments (in the form of a list of strings) that have been passed to 'solfec' or 'solfec-mpi' application and has not been identified as valid Solfec arguments. This way the user can pass some arguments to the input scripts.

obj = MODAL_ANALYSIS (body, num, path | abstol, maxiter, verbose)

This routine performs modal analysis of FEM bodies. The modal analysis results are stored with bodies and can be viewed.

- **obj = (val, vec)** - the returned tuple of: **val** = **obj[0]** eigenvalues and **vec** = **obj[1]** eigen vectors (stored contiguously one after another)
- **body** - input FEM body; the model analysis results are stored with this body
- **num** - number of lowest modes to extract
- **path** - path to file where the results will be stored (to avoid recomputing if possible). Note, that if previous modal analysis results are found they are used rather then recomputed if the number of modes and **num** are the same. If **num** is different from the previous modes count, then new **num** modes is computed from scratch.
- **abstol** - residual tolerance for the eigenvalue solver (default: 1E-11)
- **maxiter** - iterations bound for the eigenvalue solver (default: 100)
- **verbose** - 'ON' or 'OFF' verbosity flag for the eigenvalue solver (default: 'OFF')

BODY_MM_EXPORT (body, pathM, pathK | spdM, spdK)

Export body matrices in the MatrixMarket sparse format.

- **body** - BODY object of 'FINITE_ELEMENT' kind
- **pathM** - output path for mass matrix **M**
- **pathK** - output path for stiffness matrix **K**
- **spdM** - symmetric positive definite flag **M**; 'ON' or 'OFF' (default: 'ON'); only lower triangle is exported when 'ON'
- **spdK** - symmetric positive definite flag K; 'ON' or 'OFF' (default: 'ON'); only lower triangle is exported when 'ON'

DISPLAY_POINT (body, point | label)

Attach a display point to a body. Display points are defined in reference configuration and convected with bodies. Display points can be visualised by selecting 'display points on/off' in the 'tools' viewer menu. They serve purely auxiliary purpose, for example allowing to make sure that the results are read from correct locations.

- **body** - BODY object
- **point** - referential *(x, y, z)* point
- **label** - optional label

ret = FRACTURE_EXPORT_YAFFEMS (body, path | volume, quality) (under development)

Export an input file for fracture analysis in Yaffems for this body. Note, that must be a FEM body for which *fracturecheck* flag was enabled during the analysis.

- **ret** - number of fracture analysis instances exported to Yaffems
- **body** - BODY object
- **path** - export file path
- **volume** - maximum volume of the tetrahedral input Yaffems mesh (default: ∞)
- **quality** - mesh quality indicator > 1.0 for the tetrahedral input Yaffems mesh (default: 1.3)

RENDER(solfec, object) (under development)

Render selected bodies in the Viewer.

- **solfec** - SOLFEC object
- **object** - BODY object or a list of BODY objects

NB: This *CANNOT* be used from within a normal analysis script, but only from a Viewer script.

4.7 Results access

Results can be accessed either in the 'READ' mode of a SOLFEC object, or in the 'WRITE' mode once some analysis has been run.

value = DURATION (solfec)

This routine returns the duration of a simulation in SOLFEC's 'READ' mode, or ***solfec.time*** in the 'WRITE' mode.

- **value** - *(t0, t1)* duration limits of the simulation in 'READ' mode or current *time* in 'WRITE' mode
- **solfec** - SOLFEC object

FORWARD (solfec, steps)

This routine steps forward within the simulation output file. Ignored in SOLFEC's 'WRITE' mode.

- **solfec** - SOLFEC object
- **steps** - numbers of steps forward

BACKWARD (solfec, steps)

This routine steps backward within the simulation output file. Ignored in SOLFEC's 'WRITE' mode.

- **solfec** - SOLFEC object
- **steps** - number of steps backward

SEEK (solfec, time)

This routine to a specific time within the simulation output file. Ignored in SOLFEC's 'WRITE' mode.

- **solfec** - SOLFEC object
- **time** - time to start reading at

disp = DISPLACEMENT (body, point)

This routine outputs the displacement of a referential point.

- **disp** - *(dx, dy, dz)* tuple storing the displacement
- **body** - BODY object
- **point** - *(x, y, z)* tuple storing the referential point

velo = VELOCITY (body, point)

This routine outputs the velocity of a referential point.

- **velo** - *(vx, vy, vz)* tuple storing the velocity
- **body** - BODY object
- **point** - *(x, y, z)* tuple storing the referential point

stre = STRESS (body, point)

This routine outputs the Cauchy stress of a referential point.

- **stre** - *(sx, sy, sz, sxy, sxz, syz, mises)* tuple storing the Cauchy stress and the von Mises norm of it
- **body** - BODY object
- **point** - *(x, y, z)* tuple storing the referential point

ene = ENERGY (solfec | object)

The routine outputs the value of energy of a specific object.

- **ene** - *(kinetic, internal, external, contact, friction)* tuple of energy values; *internal* energy corresponds to the work of internal forces, *external* energy corresponds to the work of external forces (including constraint reactions), *contact* energy corresponds to the work of normal contact reactions, *friction* energy corresponds to the work of tangential contact reactions
- **solfec** - SOLFEC object
- **object** - SOLFEC object, BODY object or a list of BODY objects

tim = TIMING (solfec, kind)

The routine outputs the value of a specific action timing per time step.

- **tim** - value of timing (or Python None object if **solfec** was in the 'WRITE' mode)
- **solfec** - SOLFEC object in 'READ' mode
- **kind** - this is one of: 'TIMINT' (time integration), 'CONUPD' (constraints update), 'CONDET' (contact detection), 'LOCDYN' (local dynamics setup), 'CONSOL' (constraints solution), 'PARBAL' (parallel load balancing). The load balancing timing is non-zero only for parallel runs.

hist = HISTORY (solfec, list, t0, t1 | skip, progress)

This routine outputs time histories of entities.

- **hist** - a tuple of list objects storing the histories: *(times, values1, values2, ..., valuesN)*
- **solfec** - SOLFEC object
- **list** - list of objects *[object1, object2, ..., objectN]* indicating requested values. The valid objects are:
  - a tuple *(body, point, entity)* where *body* is a BODY object, *point* is a *(x, y, z)* tuple storing the referential point, and *entity* is one of: 'CX', 'CY', 'CZ' (current coordinate), 'DX', 'DY', 'DZ' (displacement), 'VX', 'VY', 'VZ' (velocity), 'SX', 'SY', 'SZ', 'SXY', 'SXZ', 'SYZ' (stress), 'MISES' (von Mises norm of stress)
  - a tuple *(object, kind)* where *object* is a SOLFEC object, a BODY object or a list of BODY objects, and *kind* is a string 'KINETIC', 'INTERNAL', 'EXTERNAL', 'CONTACT' (included in external), 'FRICTION' (included in external) and it corresponds to the energy kind; if the list of BODY objects is used, their energies are summed up
  - a string 'TIMINT', 'CONUPD', 'CONDET', 'LOCDYN', 'CONSOL', 'PARBAL' for timing histories
  - a string 'STEP' for time step history
  - a string 'CONS', 'BODS' for constraint and body number histories
  - a string 'DELBODS', 'NEWBODS' for deleted and inserted (after time 0) body number histories (nonzero only for uncompressed outputs)
  - a string 'GSITERS' (Gauss-Seidel iterations count), 'GSCOLORS' (Gauss-Seidel processor colors count), 'GSBOT', 'GSMID', 'GSTOP', 'GSINN' (Gauss-Seidel bottom, middle, top and inner set sizes), 'GSINIT' (Gauss-Seidel setup time), 'GSRUN' (Gauss-Seidel computations time), 'GSCOM' (Gauss-Seidel communication time, except the middle set), 'GSMCOM' (Gauss-Seidel middle set communication time); values other than 'GSITERS' are non-zero only for parallel runs
  - a string 'MERIT' for the time history of the constraints satisfaction merit function
  - a string 'NTITERS' for the NEWTON_SOLVER iterations count
  - a tuple *(object, entity)* or *(object, direction, pair, entity)* where *object* is a SOLFEC object, a BODY object or a list of BODY objects, *direction* is a tuple (d_x, d_y, d_z) storing a direction (use ***None*** if the *normal* direction is preferred), *pair* is a tuple *(surf1, surf2)* defining a surface pair (use ***None*** if no surface pair is preferred), and *entity* is:
    - 'GAP' for the time history of the minimal contact gap among constraints attached to given bodies (negative gap corresponds to the penetration depth)
    - 'R' for the time history of the resultant (and average over time step $[t, t + h]$) constraint reactions along the directions: normal or given by the *direction*
    - 'U' for the time history of the average constraint velocities along the directions: normal or given by the *direction*
    - 'CR' for time histories like in the 'R' case, but for contact constraints only
    - 'CU' for time histories like in the 'U' case, but for contact constraints only
- **t0** - time interval start
- **t1** - time interval end
- **skip** - number of steps to skip between two time instants
- **progress** - 'ON' or 'OFF'; print out a percentage based progress bar (default: 'OFF'); useful for large output files and slow hard disks

Chapter 5

Viewer Scripts

Arbitrary scripts can be run from within the Viewer, using Tools -> Run Python Script or the "P" key. This brings up a dialog where a string should be entered to specify the script to run and any arguments. The string is split into sub-strings on white-space and the parts interpreted as follows:

- All parts are loaded into sys.argv
- The first part is assumed to be the path to the script, from the present working directory. A trailing ".py" is added if necessary.
- Python's execfile() is called to execute the script at this path. Note that code in this Viewer script runs as it if was actually defined in the analysis script, i.e. all names defined by the analysis script are available. It is *not* imported as a module - see the Python documentation for further details. If no script is found at the specified path an error occurs.

At present the only commands which make much sense from within a Viewer script are either printing information about the analysis (e.g. len(solfec.bodies)) or using the RENDER() command to control what is displayed.

Chapter 6

Tutorials

6.1 Three basic geometric objects

This example illustrates the three basic geometric objects: CONVEX, MESH and SPHERE. We are going to construct a simple structure and hit it with a ball. Let us first create a horizontal floor.

```
w = 10
l = 10
h = 1
floor_vid = 1
floor_sid = 1
floor = HULL ([-w/2, -l/2, -h,
                w/2, -l/2, -h,
                w/2,  l/2, -h,
               -w/2,  l/2, -h,
               -w/2, -l/2,  0,
                w/2, -l/2,  0,
                w/2,  l/2,  0,
               -w/2,  l/2,  0], floor_vid, floor_sid)
```

We simply created a convex hull about eight points. This is only a geometric object for the moment. It exists only in Python interpreter, but not yet inside of a Solfec model. In order to insert it into a model, we need first to create a SOLFEC object, a BULK_MATERIAL object, and finally a BODY object having the shape described by the *floor*. Here we go.

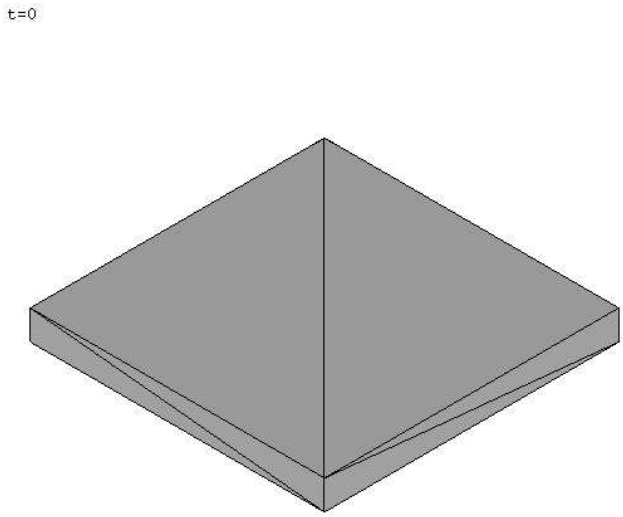
```
step = 1E-3
solfec = SOLFEC ('DYNAMIC', step, 'out/tutorail/three-basic-geometric-objects')
bulk = BULK_MATERIAL (solfec,
                      model = 'KIRCHHOFF',
                      young = 15E9,s
                      poisson = 0.3,
                      density = 2E3)
BODY (solfec, 'OBSTACLE', floor, bulk)
```

Note, that the floor is simply an obstacle - it does not move. Before creating the floor body we had to create a SOLFEC object. This object gather all necessary data for an individual simulation. In our case this will be a dynamic simulation, pursued with the time step at least as small as the specified one. The time step can be automatically decreased during a simulation due to stability requirements. The result files of this tutorial will

be written to the specified path - relative to from there the 'solfec' command was involved. Once a body has been inserted into a model, we can view the effect as follows

```
> ./solfec -v inp/tutorial/three-basic-geometric-objects.py
```

This will create a Solfec viewer, which should result in the following picture



Le us now construct a stack made of two bodies. The first one comprises four juxtaposed convex objects.

```
a = 2
b = 2
c = 2
brick_vid = 2
brick_sid = 2
brick = CONVEX ([0, 0, 0,
                 a/2, 0, 0,
                 a/2, b/2, 0,
                 0, b/2, 0,
                 0, 0, c/2,
                 a/2, 0, c/2,
                 a/2, b/2, c/2,
                 0, b/2, c/2],
                [4, 0, 3, 2, 1, brick_sid,
                 4, 1, 2, 6, 5, brick_sid,
                 4, 2, 3, 7, 6, brick_sid,
                 4, 3, 0, 4, 7, brick_sid,
                 4, 0, 1, 5, 4, brick_sid,
                 4, 4, 5, 6, 7, brick_sid], brick_vid)
b1 = COPY (brick)
b2 = TRANSLATE (COPY (brick), (a, 0, 0))
b3 = TRANSLATE (COPY (brick), (0, b, 0))
```

```
b4 = TRANSLATE (COPY (brick), (a, b, 0))
shape = [b1, b2, b3, b4]
TRANSLATE (shape, (-a, -b, 0))
BODY (solfec, 'RIGID', shape, bulk)
```

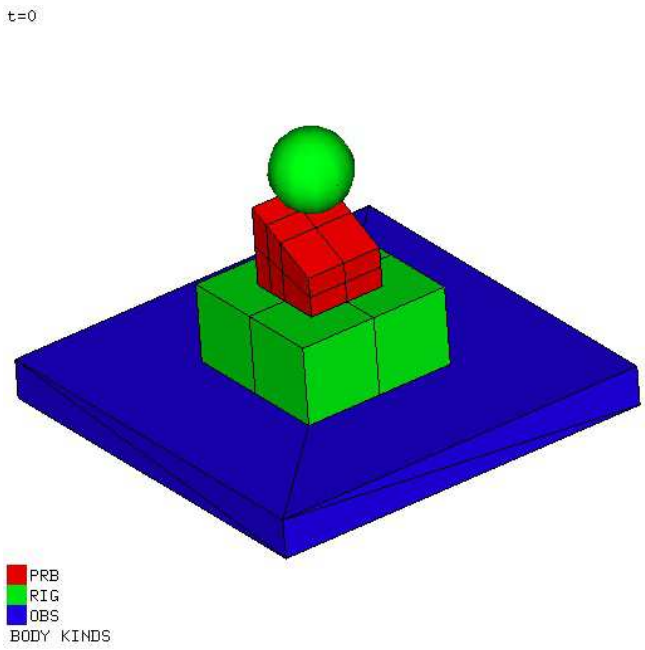
Note, that a base *brick* was created first. Then this brick was copied and manipulated into four different objects: *b1,b2, ..., b4*. The list of those objects was passed as a shape when creating the first rigid body. The second body will be pseudo-rigid and will have its shape defined by a mesh.

```
nodes = [-1.0, -1.0, 0.0,
          1.0, -1.0, 0.0,
          1.0,  1.0, 0.0,
         -1.0,  1.0, 0.0,
         -1.0, -1.0, 2.0,
          1.0, -1.0, 2.0,
          1.0,  1.0, 1.0,
         -1.0,  1.0, 1.0]
mesh = HEX (nodes, 2, 3, 2, 3, [3, 3, 3, 3, 3, 3], dy = [1, 1, 2])
TRANSLATE (mesh, (0, 0, c))
BODY (solfec, 'PSEUDO_RIGID', mesh, bulk)
```

The hexahedral mesh is spanned on eight nodes and has an inclined shape due to z level slope. Note, that we have specified the "dy" argument of the HEX command, so to illustrate non-uniform meshing along one of the directions. The mesh is translated to rest on top of the previous body. Then a pseudo-rigid body is created. Now, let us create the sphere.

```
sphere = SPHERE ((0, 0, 5), 1, 1, 1)
body = BODY (solfec, 'RIGID', sphere, bulk)
INITIAL_VELOCITY (body, (0, 0, -10), (0, 0, 0))
```

When creating the rigid body corresponding to the sphere, we have now retrieved the *body* object. It is needed in order to prescribe the initial velocity. The sphere has the initial linear velocity $v_z = -10$ m/s. Let us have a look at the model so far.



Now, in order to be able to control contact behaviour, we need to define a surface material. This will be a default material, hence we shall not specify a surface pairing (int this example surface identifiers are not used). Whenever a contact is detected, the following Signorini-Coulomb model is employed

```
SURFACE_MATERIAL (solfec, model = 'SIGNORINI_COULOMB',
                  friction = 0.5, restitution = 0.0)
```

It will be also of use to apply some gravity loading.

```
GRAVITY (solfec, (0, 0, -10))
```

Before running the actual simulation, it remains to create a solver object. We use the Gauss-Seidel solver here.

```
gs = GAUSS_SEIDEL_SOLVER (1E-3, 1000)
```

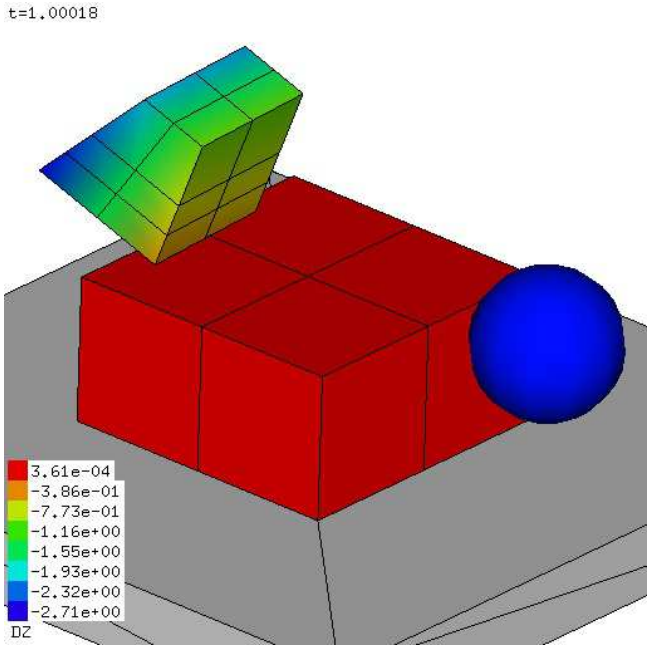
The relative constraint reaction accuracy was set to 1E-3, while the maximal the maximal number of iterations is 1000. It remains to run the simulation.

```
RUN (solfec, gs, 1.0)
```

And then actually run *solfec* from the command line

```
./solfec inp/tutorial/three-basic-geometric-objects.py
```

One second of the simulation was computed. Let us have a look at the displacement along z at the end of this time.



6.2 Ball impact

This example illustrates using multiple SOLFEC objects, application of the PENALTY_SOLVER, and using HISTORY to retrieve and then plot time histories. First we define a Python function that will create a model of ball impacting a plate for a specific set of parameters.

```
def ball_impact (step, stop, spring_value, dashpot_value, output):
  w = 2
  l = 2
  h = 1
  floor_vid = 1
  floor_sid = 1
  floor = HULL ([-w/2, -l/2, -h,
                  w/2, -l/2, -h,
                  w/2, l/2, -h,
                 -w/2, l/2, -h,
                 -w/2, -l/2, 0,
                  w/2, -l/2, 0,
                  w/2, l/2, 0,
                 -w/2, l/2, 0], floor_vid, floor_sid)

  solfec = SOLFEC ('DYNAMIC', step, output)

  bulk = BULK_MATERIAL (solfec, model = 'KIRCHHOFF',
                        young = 15E9, poisson = 0.3, density = 2E3)
  BODY (solfec, 'OBSTACLE', floor, bulk)

  sphere = SPHERE ((0, 0, 1.0), 1, 1, 1)
  body = BODY (solfec, 'RIGID', sphere, bulk)
  INITIAL_VELOCITY (body, (0, 0, -5), (0, 0, 0))

  SURFACE_MATERIAL (solfec, model = 'SPRING_DASHPOT', friction = 0.0,
                    spring = spring_value, dashpot = dashpot_value)

  GRAVITY (solfec, (0, 0, -10))

  xs = PENALTY_SOLVER ()

  RUN (solfec, xs, stop)

  return solfec
```

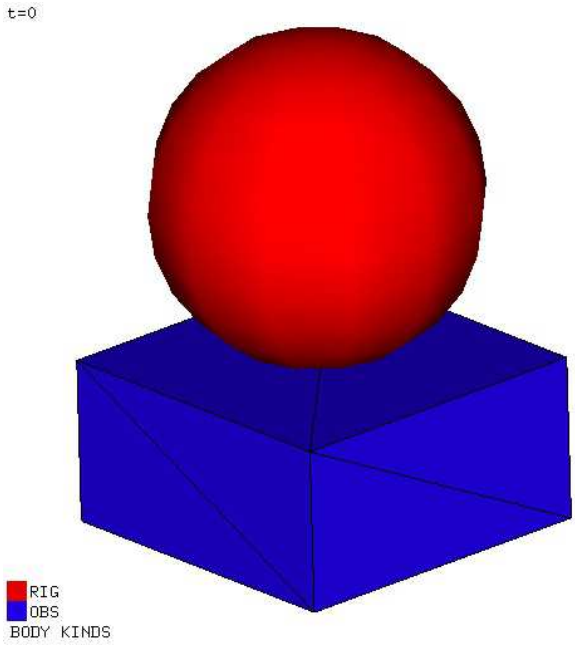
The above code is similar to the previous example. Here though the complete model is created inside of a Python function called *ball_ impact*. By itself this will not run any simulation - this function needs to be called from the main module of the input file (we remind that Python uses indentation to decide upon code blocking - in our case no indentation indicates the main module). Above the 'SPRING_DASHPOT' material model is used for the contact interface. The parameters of the spring and damper are passed as arguments of the *ball_ impact* function. It should be noted, that the SOLFEC object is returned from the routine. Now, here is the main module

```
step = 1E-3
stop = 2.0
spring = 1E+9
sol1 = ball_impact (step, stop, spring, 0E0, 'out/tutorial/ball-impact-1')
sol2 = ball_impact (step, stop, spring, 1E6, 'out/tutorial/ball-impact-2')
sol3 = ball_impact (step, stop, spring, 1E7, 'out/tutorial/ball-impact-3')
```

We simply run three different simulations for three values of the *dashpot* parameter. We can now view the three models by typing

```
./solfec -v ./inp/tutorial/ball-impact.py
```

The viewer allows change the current model by using '<' and '>' keyboard shortcuts, or using the right-mouse click for the drop-down menu and then selecting *menu: domain: previous* or *menu: domain: next*. In this example the three models do not visibly differ.



We could now run each model in the viewer mode (*menu: analysis: run*), but it will be more useful to plot kinetic energy and compare it for all the three values of the *dashpot* parameter. The code below does the job

```
if not VIEWER() and sol1.mode == 'READ':
  import matplotlib.pyplot as plt
  th = HISTORY (sol1, (sol1, 'KINETIC'), 0, stop)
  plt.plot (th [0], th [1], lw = 2, label='kin (0)')
  th = HISTORY (sol2, (sol2, 'KINETIC'), 0, stop)
  plt.plot (th [0], th [1], lw = 2, label='kin (1E6)')
  th = HISTORY (sol3, (sol3, 'KINETIC'), 0, stop)
  plt.plot (th [0], th [1], lw = 2, label='kin (1E7)')
  plt.axis (xmin = 0, xmax = 2, ymin=-10000, ymax=110000)
  plt.legend(loc = 'upper right')
  plt.savefig ('doc/figures/ball-impact.eps')
```

First, we check whether Solfec is not run with the *-v* option and whether it is in 'READ' mode. This is the case, when after running the complete analysis

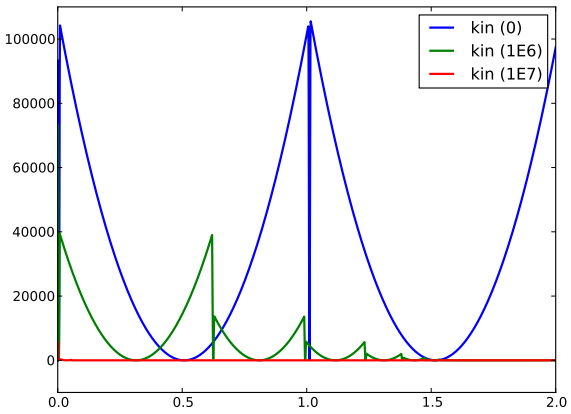
```
./solfec ./inp/tutorial/ball-impact.py
```

we again run

```
./solfec ./inp/tutorial/ball-impact.py
```

Now, Solfec will find out that the output files in *out/tutorial/ball-impact-(1,2,3)* are present. It will open in 'READ' mode. In order to create the plot we are going to use the *matplotlib* Python package - please refer to

http://matplotlib.sourceforge.net/index.html in order to learn how to install it. We next use the HISTORY command in order to retrieve the time histories of the kinetic energy for all three created SOLFEC objects. The result in plotted into an EPS file, visible below.



We can see that the $dashpot = 0$ results in a fully elastic impact (energy conserving behaviour), $dahspot = 1E6$ introduces some fractional energy restitution after impacts, while $dashpot = 1E7$ results in a nearly plastic impact.

Chapter 7

Contact points

7.1 Contacts from overlaps

Body shapes are juxtapositions of convex objects (Sections 4.2.1, 4.2.2, 4.2.3). A contact point and normal direction result from an overlap of two convex objects (Figure 7.1). This is motivated by two factors. Firstly, the point and the normal direction derived from an overlap are well defined for nonsmooth geometry. Secondly, we wish to use as few contact points as possible, but still be able to control the accuracy of contact resolution by mesh refinement. A non-positive *gap* function is suitably derived from such overlap. We refer the reader to Chapter 9 of [8] for more details.

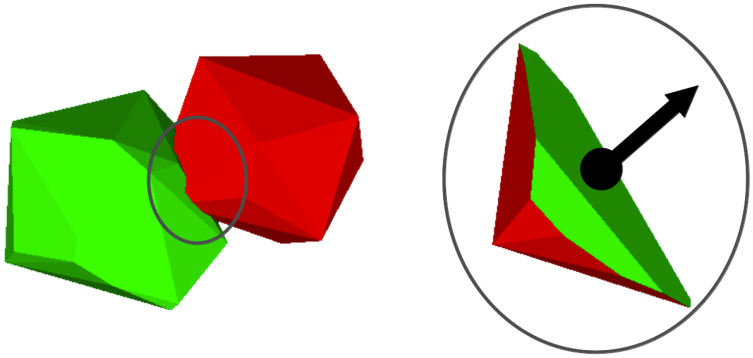


Figure 7.1: A contact point and normal direction extracted from an intersection of two convex objects.

7.2 Contact sparsification

Let us have a look at an arch in Figure 7.2. The detail of two bricks shows narrow meshing near the inner and outer boundaries. This will allow to better reproduce the hinging mechanism of arch collapse. A single brick is composed of six elements: the two large middle elements and the four narrow inner and outer elements. If we now apply contact detection algorithm, all possible volumetric overlaps will be detected. Because identically meshed bodies are perfectly adjacent to each other we shall end up with a clutter of contact points, generated by all of the adjacent element volumes. This is visible on the left in Figure 7.3. If one would have to generate contact points by hand, they would probably look like those on the right in Figure 7.3. A heuristic sparsification algorithm filters out redundant contact points (cf. Algorithm 7.1).

Algorithm 7.1 Contact sparsification algorithm.

```
SPARSIFY ()
  for each contact1 do
    for each body adjacent to contact1 do
      for each contact2 adjacent to body do
        if contact1 = contact2 skip
        if area (contact1) < threshold · area (contact2) and
           topologically_adjacent (contact1, contact2) then remove contact1
        else if point (contact1) = point (contact2) then remove contact1
```

We walk over all contact points and compare each of them with other contacts adjacent through common bodies. If the area of a contact is a smaller than an area of a topologically adjacent neighbouring contact, then we remove it. The same happens if the two contact points coincide. Topological adjacency of contact points indicates that they have been created between the mesh elements that are topologically adjacent. We know then, that next to a contact point with a small supporting area there is another one with a suitably larger contact area. The Solfec command CONTACT_SPARSIFY sets the *threshold* value (cf. Section 4.6).

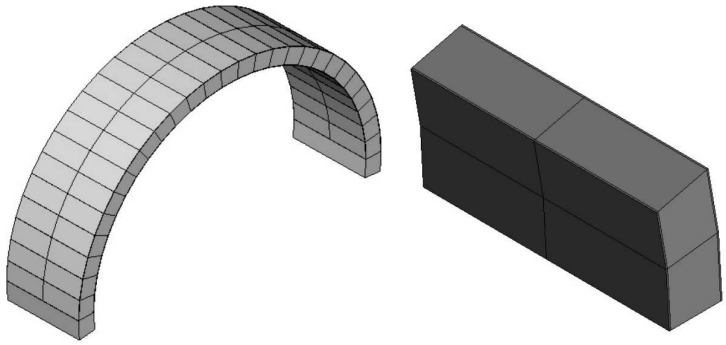


Figure 7.2: An arch and a detail of two bricks. Note the narrow meshing near the edge of inner and outer boundaries.

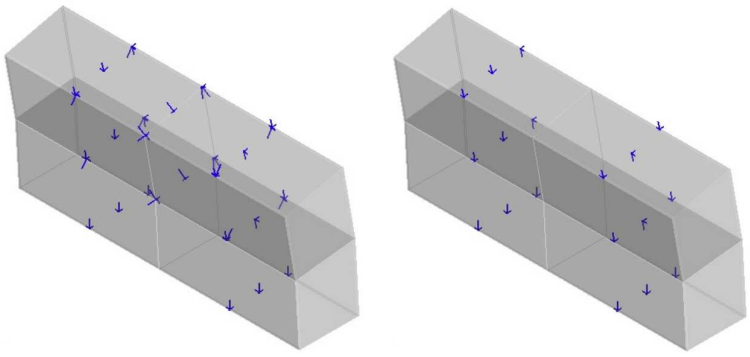


Figure 7.3: A detail of the arch from Figure 7.2. Contact points on the left are not sparsified. Contact points on the right are sparsified.

Chapter 8

Materials

8.1 Surface materials

A surface material is assigned to a pairing of surfaces. See Section 4.2.7 for the input syntax.

8.1.1 Signorini-Coulomb

The velocity Signorini condition reads

$$\bar{U}_N \geq 0 \;\; R_N \geq 0 \;\; \bar{U}_N R_N = 0 \tag{8.1}$$

where $\bar{U}_N = U_N^{t+h} + \eta \min(0, U_N^t)$, η is the velocity *restitution* coefficient, U_N is the the normal relative velocity, and R_N is the normal reaction. The normal direction is consistent with the positive gap velocity so that (8.1) states, that any violation of the non-penetration results in a reactive force or velocity driving at the penetration-free configuration. Using $\bar{U}_N$ allows to account for the Newton impact law. **Only *restitution = 0* or *1* is energy consistent TODO** (cf. Section 10.5 of [8]). The Coulomb's friction law reads

$$\begin{cases} \|\mathbf{R}_T\| \leq \mu R_N & \\ \|\mathbf{R}_T\| < \mu R_N & \Rightarrow \mathbf{U}_T = \mathbf{0} \\ \|\mathbf{R}_T\| = \mu R_N & \Rightarrow \exists_{\lambda \geq 0} \mathbf{U}_T = -\lambda \mathbf{R}_T \end{cases} \tag{8.2}$$

A friction force smaller than μR_N implies sticking, while sliding occurs with the force of value μR_N and direction opposite to the slip velocity. The two laws can expressed in a compact from $\mathbf{C}(\mathbf{U}, \mathbf{R}) = \mathbf{0}$. An examples is

$$\mathbf{C}(\mathbf{U}, \mathbf{R}) = \begin{bmatrix} \max(\mu d_N, \|\mathbf{d}_T\|)\, \mathbf{R}_T - \mu \max(0, d_N)\, \mathbf{d}_T \\ R_N - \max(0, d_N) \end{bmatrix} \tag{8.3}$$

where

$$d_N = R_N - \rho \bar{U}_N \tag{8.4}$$

$$\mathbf{d}_T = \mathbf{R}_T - \rho \mathbf{U}_T \tag{8.5}$$

and $\rho > 0$. We refer the reader to Chapter 10 of [8] for more details.

8.1.2 Spring-dashpot

Let

$$s = spring \text{ and } d = dashpot \text{ and } g = gap \text{ and } m = hpow \tag{8.6}$$

The normal reaction is computed as follows

$$R_N = -s \cdot \frac{g^{t+h} + g^t}{2} - d \cdot \frac{U_N^{t+h} + U_N^t}{2} \tag{8.7}$$

where U_N is the normal relative velocity. Recall, that the gap function is computed for the configuration $\mathbf{q}^t + \frac{h}{2}\mathbf{u}^t$, so that the gap function value computed during geometrical contact detection reads

$$g = g^t + \frac{h}{2} U_N^t \tag{8.8}$$

We then have

$$g^{t+h} = g^t + \frac{h}{2}\left(U_N^{t+h} + U_N^t\right) = g + \frac{h}{2} U_N^{t+h} \tag{8.9}$$

and since $g^t = g - \frac{h}{2}U_N^t$ we can estimate

$$R_N = -s \cdot \left(g + \frac{h}{4}\left(U_N^{t+h} - U_N^t\right)\right) - \frac{d}{2} \cdot \left(U_N^{t+h} + U_N^t\right) \tag{8.10}$$

We then use the diagonal block of local dynamics

$$\mathbf{U}^{t+h} = \mathbf{B} + \mathbf{W}\mathbf{R} \tag{8.11}$$

in order to estimate U_N^{t+h} as follows

$$U_N^{t+h} = B_N + \mathbf{W}_{NT}\mathbf{R}_T + W_{NN}R_N \tag{8.12}$$

where a previous tangential reaction $\mathbf{R}_T$ is employed. Inserting this it into (8.10) results in

$$\bar{B}_N = B_N + \mathbf{W}_{NT}\mathbf{R}_T \tag{8.13}$$

$$R_N = \left[-s \cdot \left(g + \frac{h}{4}\left(\bar{B}_N - U_N^t\right)\right) - \frac{d}{2} \cdot \left(\bar{B}_N + U_N^t\right)\right] / \left[1 + \left(s \cdot \frac{h}{4} + \frac{d}{2}\right) \cdot W_{NN}\right] \tag{8.14}$$

The reason for using the above, rather than the classical $R_N = -s \cdot g - d \cdot U_N^t$ is in an increased stability of the current approach. Since we aim at simplicity and want to avoid any nonlinear solve only at this stage we include the Hertz law power

$$g_1 = \min\left(g + \frac{h}{4}\left(\bar{B}_N - U_N^t\right), 0\right)$$

$$s_1 = sm\left(-g_1\right)^{m-1}$$

$$R_N = \left[s \cdot \left(-g_1\right)^m - \frac{d}{2} \cdot \left(\bar{B}_N + U_N^t\right)\right] / \left[1 + \left(s_1 \cdot \frac{h}{4} + \frac{d}{2}\right) \cdot W_{NN}\right]$$

Again aiming at maximum simplicity and assuming $\mathbf{U}_T^{t+h} = 0$ we then estimate the tangential stick reaction

$$\mathbf{R}_T = -\mathbf{W}_{TT}^{-1}\left(\mathbf{B}_T + \mathbf{W}_{TN}R_N\right) \tag{8.15}$$

The complete interface law is expressed in Algorithm 8.1 (μ refers there to the coefficient of friction). We refer the reader to Chapter 7 of [8] for more details on local dynamics.

8.2 Bulk materials

A bulk material is assigned to a volume. See Section 4.2.8 for the input syntax.

8.2.1 Kirchhoff - Saint Venant

This is a simple extension of the linearly elastic material to the large deformation regime. Suitable for large rotation, small strain problems. The strain energy function Ψ of the Kirchhoff - Saint Venant materials reads

$$\Psi = \frac{1}{4}\left[\mathbf{F}^T\mathbf{F} - \mathbf{I}\right] : \mathbf{C} : \left[\mathbf{F}^T\mathbf{F} - \mathbf{I}\right] \tag{8.16}$$

where

$$C_{ijkl} = \lambda\delta_{ij}\delta_{kl} + \mu\left[\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}\right] \tag{8.17}$$

In the above λ and μ are Lamé constants, while δ_{ij} is the Kronecker delta. The Lamé constants can be expressed in terms of the Young modulus E and the Poisson ratio ν as

$$\lambda = \frac{E\nu}{(1+\nu)(1-2\nu)} \tag{8.18}$$

$$\mu = \frac{E}{2+2\nu} \tag{8.19}$$

The first Piola stress tensor is computed as a gradient of the hyperelastic potential Ψ

$$\mathbf{P} = \partial_{\mathbf{F}}\Psi(\mathbf{F}) \tag{8.20}$$

where $\mathbf{F}$ is the deformation gradient.

Algorithm 8.1 Spring-dashpot reaction calculation.

SPRING_DASHPOT $(h, g, s, d, \mu, cohesion, cohesive)$
  $\bar{B}_N = B_N + \mathbf{W}_{NT}\mathbf{R}_T$
  if implicit then
    $g_1 = \min\left(g + \frac{h}{4}\left(\bar{B}_N - U_N^t\right), 0\right)$
    $s_1 = sm\left(-g_1\right)^{m-1}$
    $R_N = \left[s\cdot\left(-g_1\right)^m - \frac{d}{2}\cdot\left(\bar{B}_N + U_N^t\right)\right] / \left[1 + \left(s_1\cdot\frac{h}{4} + \frac{d}{2}\right)\cdot W_{NN}\right]$
  else $R_N = s\cdot\left(-\min\left(g, 0\right)\right)^m - d\cdot U_N^t$
  if not $cohesive$ and $R_N < 0$ then $\mathbf{R} = 0$ return
  $\mathbf{R}_T = -\mathbf{W}_{TT}^{-1}\left(\mathbf{B}_T + \mathbf{W}_{TN}R_N\right)$
  if $cohesive$ and $R_N < -cohesion$ then $cohesive = false$ and $R_N = -cohesion$
  if $\|\mathbf{R}_T\| > \mu\left|R_N\right|$ then
    $\mathbf{R}_T = \mu R_N \mathbf{R}_T / \|\mathbf{R}_T\|$
    if $cohesive$ then $cohesive = false$

Chapter 9

Constraints accuracy

As explained in Chapter 1, at every time step an implicit equation $\mathbf{C}(\mathbf{R}) = \mathbf{0}$ is solved. Ideally, when for some $\mathbf{R}$ there holds $\mathbf{C}(\mathbf{R}) = \mathbf{0}$ we have an exact solution. Of course, in numerical terms this is not possible. For very large problems, and especially for problems where the amount of constraints exceeds the amount of kinematic freedom, obtaining very accurate solutions is hard and often impractical. In any case, it is useful to have an accuracy measure that has some physical interpretation. In order to compute constraints accuracy, we formulate $\mathbf{C}(\mathbf{R})$ in terms of velocity (cf. Section 9.1 for the Signorini-Coulomb law) and use

$$g(\mathbf{R}) = \sum_{\alpha} \left\langle \mathbf{W}_{\alpha\alpha}^{-1} \mathbf{C}_{\alpha}(\mathbf{R}), \mathbf{C}_{\alpha}(\mathbf{R}) \right\rangle / \sum_{\alpha} \left\langle \mathbf{W}_{\alpha\alpha}^{-1} \mathbf{B}_{\alpha}, \mathbf{B}_{\alpha} \right\rangle \tag{9.1}$$

in order to approximately measure the relative amount of spurious energy, due to an inexact satisfaction of constraints. The denominator corresponds to the kinetic energy of the relative free motion, hence $g(\mathbf{R})$ is the ratio of the spurious energy over the nominal amount of the energy available at the constraints. Since inverting $\mathbf{W}$ would be unpractical or impossible due to singularity, we only use the diagonal blocks, which are always positive definite. To recapitulate, in short

$$g(\mathbf{R}) \simeq \frac{\text{spurious energy due to inaccurate solution}}{\text{free energy available at the constraints}} \tag{9.2}$$

9.1 Signorini-Coulomb revisited

We express the Signorini-Coulomb law defined in Section 8.1.1 in the form suitable for (9.1). The friction cone K_{α} is defined as

$$K_{\alpha} = \{\mathbf{R}_{\alpha} : \|\mathbf{R}_{\alpha T}\| \leq \mu_{\alpha} R_{\alpha N}, R_{\alpha N} \geq 0\} \tag{9.3}$$

where μ_{α} is the coefficient of friction. It has been shown by De Saxcé and Feng [4], that the Signorini-Coulomb law can be expressed in a compact form

$$-\begin{bmatrix} \mathbf{U}_{\alpha T} \\ \bar{U}_{\alpha N} + \mu_{\alpha} \|\mathbf{U}_{\alpha T}\| \end{bmatrix} \in N_{K_{\alpha}}(\mathbf{R}_{\alpha}) \tag{9.4}$$

where $N_{K_{\alpha}}$ stands for the normal cone of the set K_{α}. For a convex set A the normal cone $N_A(\mathbf{R})$ at point $\mathbf{R} \in A$ is defined as the set of all vectors $\mathbf{V}$ such that $\langle \mathbf{V}, \mathbf{S} - \mathbf{R} \rangle \leq 0$ for all $\mathbf{S} \in A$. Let

$$\mathbf{F}(\mathbf{R}) = \begin{bmatrix} \dots \\ \mathbf{U}_{\alpha T}(\mathbf{R}) \\ \bar{U}_{\alpha N}(\mathbf{R}) + \mu_{\alpha} \|\mathbf{U}_{\alpha T}(\mathbf{R})\| \\ \dots \end{bmatrix} \tag{9.5}$$

and

$$K = \bigcup_{\alpha} K_{\alpha} \tag{9.6}$$

where the dependence $\mathbf{U}_{\alpha}(\mathbf{R})$ is defined in (1.14). Formula (9.4) states, that the frictional contact constraints are satisfied if $-\mathbf{F}(\mathbf{R})$ belongs to the normal cone of the friction cone at $\mathbf{R}$. Hence

$$-\mathbf{F}(\mathbf{R}) = \mathbf{R} - \mathbf{F}(\mathbf{R}) - \mathrm{proj}_{K}(\mathbf{R} - \mathbf{F}(\mathbf{R})) \tag{9.7}$$

which can be reduced to the usual projection formula $\mathbf{R} = \mathrm{proj}_{K}(\mathbf{R} - \mathbf{F}(\mathbf{R}))$. Let us not do it though, but rather define a vector field

$$\mathbf{m}(\mathbf{S}) = \mathbf{S} - \mathrm{proj}_{K}(\mathbf{S}) = \mathbf{n}(\mathbf{S}) \langle \mathbf{n}(\mathbf{S}), \mathbf{S} \rangle \tag{9.8}$$

where

$$\mathbf{n}_{\alpha}(\mathbf{S}_{\alpha}) = \begin{cases} \mathbf{0} & \text{if} \quad \|\mathbf{S}_{\alpha T}\| - \mu_{\alpha} S_{\alpha N} \leq 0 \\ \mathbf{S}_{\alpha} / \|\mathbf{S}_{\alpha}\| & \text{if} \quad \mu_{\alpha} \|\mathbf{S}_{\alpha T}\| + S_{\alpha N} < 0 \\ \frac{1}{\sqrt{1+\mu_{\alpha}^2}} \begin{bmatrix} \mathbf{S}_{\alpha T} / \|\mathbf{S}_{\alpha T}\| \\ -\mu_{\alpha} \end{bmatrix} & \text{otherwise} \end{cases} \tag{9.9}$$

We can rewrite (9.4) as

$$\mathbf{C}(\mathbf{R}) = \mathbf{F}(\mathbf{R}) + \mathbf{m}(\mathbf{R} - \mathbf{F}(\mathbf{R})) = \mathbf{0} \text{ and } \mathbf{R} \in K \tag{9.10}$$

Note, that $\mathbf{F}(\mathbf{R})$ is expressed in terms of velocity, and so is $\mathbf{C}(\mathbf{R})$.

Chapter 10

Solvers

Development of solvers for unilateral dynamics is one of the main driving forces behind Solfec. The two solvers described in Sections 10.1 and 10.2 are the classical Gauss-Seidel approach of Contact Dynamics and a somewhat modified penalty solver of the Discrete Element Method. The projected quasi-Newton solver from Section 10.3 has been designed specifically for Solfec.

10.1 Gauss-Seidel solver

The equations of local dynamics (1.14) read

$$\mathbf{U}_\alpha = \mathbf{B}_\alpha + \sum_\beta \mathbf{W}_{\alpha\beta}\mathbf{R}_\beta \tag{10.1}$$

where $\mathbf{U}_\alpha$ are relative velocities and $\mathbf{R}_\alpha$ are reactions at constraint points. $\mathbf{U}_\alpha, \mathbf{R}_\alpha, \mathbf{B}_\alpha$ are 3-vectors, while $\mathbf{W}_{\alpha\beta}$ are 3×3 matrix blocks. Each constraint equation can be formulated as

$$\mathbf{C}_\alpha \left(\mathbf{U}_\alpha, \mathbf{R}_\alpha \right) = \mathbf{0} \tag{10.2}$$

or in other words

$$\mathbf{C}_\alpha \left(\mathbf{B}_\alpha + \sum_\beta \mathbf{W}_{\alpha\beta}\mathbf{R}_\beta, \mathbf{R}_\alpha \right) = \mathbf{0} \tag{10.3}$$

Algorithm 10.1 is quite simple: diagonal block problems are solved until reaction change is small enough. The Gauss-Seidel paradigm corresponds to the fact, that the most recent off-diagonal reactions are used when solving the diagonal problem. Of course, because of that, a perfectly parallel implementation is not possible. After all, reactions are updated in a sequence. We can nevertheless relax the need for sequential processing.

Algorithm 10.1 Serial Gauss-Seidel algorithm

```
SERIAL_GAUSS_SEIDEL (Constraints, ε, γ)
1  do
2    for each α in Constraints do
3      S_α = R_α
4      find R_α such that C_α (B_α + Σ_β W_αβ R_β, R_α) = 0
5                          assuming R_β = constant for β ≠ α
6  while ‖S − R‖ / ‖R‖ > ε and g(R) > γ
```

Algorithm 10.2 Simple processor coloring.

```
COLOR ()
1  for i= 1, ..., n do color [i] = 0
2  for i= 1, ..., n do
3    do
4      color [i] = color [i] + 1
5    while for any j ∈ adj (i) there holds color [i] = color [j]
```

Perhaps the most scalable Gauss-Seidel approach to date was devised by Adams [2]. Although originally it was used as a multi-grid smoother, the core idea can be as well applied in our context. Each processor owes a subset of (internal) constraints Q_i, where $i = 1, 2, ..., n$ are the processors indices. Therefore the local velocity update can be rewritten as

$$\mathbf{U}_\alpha = \mathbf{B}_\alpha + \sum_{\beta \in Q_i} \mathbf{W}_{\alpha\beta}\mathbf{R}_\beta + \sum_{\beta \notin Q_i} \mathbf{W}_{\alpha\beta}\mathbf{R}_\beta \tag{10.4}$$

Some of the $\mathbf{W}_{\alpha\beta}$ blocks and reactions $\mathbf{R}_\beta$ correspond to the (external) constraints stored on other processors ($\beta \notin Q_i$). Let us denote the set of corresponding reaction indices by P_i. That is

$$P_i = \{\beta : \exists \mathbf{W}_{\alpha\beta} \neq \mathbf{0} \text{ and } \alpha \in Q_i \text{ and } \beta \notin Q_i\} \tag{10.5}$$

For each $\beta \in P_i$ we know an index of processor $cpu\,(\beta)$ storing the constraint with index β. For processor i we can then define a set of adjacent processors as follows

$$adj\,(i) = \{cpu\,(\beta) : \beta \in P_i\} \tag{10.6}$$

When updating reactions, a processor needs to communicate only with other adjacent processors. We are going to optimise a pattern of this communication by *coloring* the processors. We shall then assign to each processor a color, such that no two adjacent processors have the same color. A simple coloring method is summarised in Algorithm 10.2. We try to assign as few colors as possible. We then split the index sets Q_i as follows

$$Top_i = \{\alpha : \forall \mathbf{W}_{\alpha\beta} : \beta \in P_i \wedge color\,[cpu\,(\beta)] < color\,[i]\} \tag{10.7}$$

$$Bottom_i = \{\alpha : \forall \mathbf{W}_{\alpha\beta} : \beta \in P_i \wedge color\,[cpu\,(\beta)] > color\,[i]\} \tag{10.8}$$

$$Middle_i = \{\alpha : \forall \mathbf{W}_{\alpha\beta} : \beta \in P_i \wedge \alpha \notin Top_i \cup Bottom_i\} \tag{10.9}$$

$$Inner_i = Q_i \setminus \{Top_i \cup Bottom_i \cup Middle_i\} \tag{10.10}$$

The top constraints require communication only with processors of lower colors. The bottom constraints require communication only with processors of higher colors. The middle constraints require communication with either. The inner constraints require no communication. The inner reactions are further split in two sets

$$Inner_i = Inner1_i \cup Inner2_i \tag{10.11}$$

so that

$$|Bottom_i| + |Inner2_i| = |Top_i| + |Inner1_i| \tag{10.12}$$

The parallel Gauss-Seidel scheme is summarised in Algorithm 10.3. The presented version is simplified in the respect, that alternate forward and backward runs are not accounted for (in terms of constraints ordering).

Algorithm 10.3 Parallel Gauss-Seidel algorithm.

```
SWEEP (Set)
1  for each α ∈ Set do
2    find R_α such that C_α(B_α + Σ_β W_αβ R_β, R_α) = 0
3                       assuming R_β = constant for β ≠ α

LOOP (Set)
1  descending sort of α ∈ Set based on max(color[cpu(β)]) where ∃W_αβ
2  for each ordered α in Set do
3    for each β such that ∃W_αβ and color[cpu(α)] < color[cpu(β)] do
4      if not received (R_β) then receive (R_β)
5    find R_α such that C_α(B_α + Σ_β W_αβ R_β, R_α) = 0
6                       assuming R_β = constant for β ≠ α
7    send (R_α)
8  receive all remaining R_β

PARALLEL_GAUSS_SEIDEL (ε, γ)
1   COLOR ()
2   do
3     S = R
4     SWEEP (Top_i)
5     send (Top_i)
6     SWEEP (Inner2_i)
7     receive (Top_i)
8     LOOP (Middle_i)
9     SWEEP (Bottom_i)
10    send (Bottom_i)
11    SWEEP (Inner1_i)
12    receive (Bottom_i)
13  while ||S − R|| / ||R|| > ε and g(R) > γ
```

We first process the Top_i set: a single sweep over the corresponding diagonal block problems is performed in line 3. Then we send the computed top reactions to the processors with lower colors. We try to overlap communication and computation, hence we sweep over the $Inner2_i$ set (line 5) while sending. We then receive the top reactions. It should be noted that all communication is asynchronous - we only wait to receive reactions immediately necessary for computations. In line 7 we enter the loop processing the $Middle_i$ set. This is the location of the computational bottleneck. Middle nodes communicate with processors of higher and lower colors and hence, they need to be processed in a sequence. The sequential processing is still relaxed by using processor coloring. In the LOOP algorithm we first sort the constraints according to the descending order of maximal colors of their adjacent processors (line 1). We then maintain this ordering while processing constraints. As the top reactions were already sent, some of the constraints from the middle set will have their external reactions from higher colors fully updated. These will be processed first in line 5 of LOOP and then sent to lower and higher (by color) processors in line 7. This way some processors with lower colors will have their higher color off-diagonal reactions of middle set constraints fully updated and they will proceed next. And so on. At the end (line 8), we need to receive all remaining reactions that have been sent in line 7 of LOOP. Coming back to PARALLEL_GAUSS_SEIDEL, after the bottleneck of the LOOP, in lines 8-11 we process the $Bottom_i$ and $Inner1_i$ sets in the same way as we did with the Top_i and $Inner2_i$ sets. The condition (10.12) attempts to balance the amount of computations needed to hide the communication (e.g. the larger the Top_i set is, the larger the $Inner2_i$ set becomes). It should be noted that the convergence criterion in line 12 is global across all processors.

In Section 4.2.11 several variants of the parallel algorithm are listed. Algorithm 10.3 corresponds to the FULL variant. We might like to relax the bottleneck of LOOP in line 7 of Algorithm 10.3 by replacing it with

```
7.1   SWEEP (Middle_i)
7.2   send (Middle_i)
7.3   receive (Middle_i)
```

so that the middle nodes are processed in an inconsistent manner: the off-processor information corresponds to the previous iteration (just like in the Jacobi method). Usually the $Middle_i$ sets are small and hence this inconsistency does not have to lead to divergence (especially for deformable kinematics, where constraint interactions are weak, while $\mathbf{W}$ is diagonally dominant). This is the MIDDLE_JACOBI variant of the algorithm. The last variant corresponds to a rather gross inconsistency: something usually called "a processor Gauss-Seidel method". Let us define the set

$$All_i = Top_i \cup Bottom_i \cup Middle_i \cup Inner_i \tag{10.13}$$

In this case, lines 3-11 of PARALLEL_GAUSS_SEIDEL from Algorithm 10.3 need to be replaced with

```
3   SWEEP (All_i)
4   send (All_i)
5   receive (All_i)
```

Although this kind of approach does work as a multi-grid smoother, it has little use in our context. Nevertheless, we use it for illustration sake and name the BOUNDARY_JACOBI.

10.2 Penalty solver

The penalty solver is quite straightforward. On each processor we split the constraints into $Contacts_i$ and $Others_i$, hence we separate contact constraints from bilateral ones. We then update the contacts using the spring-dashpot model and calculate reactions of bilateral constraints using the Gauss-Seidel solver (fixed accuracy epsilon=1E-4, maxiter = 1000 is used). We use the Gauss-Seidel approach for non-contacts because in this case it is quite fast, while it avoids issues related to penalisation of bilateral constraints. Algorithm 10.4 summarises the method.

Algorithm 10.4 Parallel penalty solver.

```
PARALLEL_PENALTY_SOLVER ()
1  for all α in Contacts_i do
2    SPRING_DASHPOT (h, gap_α, spring_α, dashpot_α, friction_α, cohesion_α, cohesive_α)
3  send (Contacts_i)
4  receive (Contacts_i)
5  GAUSS_SEIDEL (Others_i)
```

10.3 Projected quasi-Newton solver

Let us rewrite the frictional contact problem (9.10) once again

$$\mathbf{C}(\mathbf{R}) = \mathbf{F}(\mathbf{R}) + \mathbf{m}(\mathbf{R} - \mathbf{F}(\mathbf{R})) = \mathbf{0} \text{ and } \mathbf{R} \in K \tag{10.14}$$

where K is the direct sum of friction cones at all contact points. Since $\mathbf{C}(\mathbf{R})$ is not smooth, to compute $\nabla\mathbf{C}$ we generalize the approach from [5], where only the self-dual case (friction coefficient equal to 1) was considered. Our idea is to employ the following projected quasi-Newton step

$$\mathbf{R}^{k+1} = \mathrm{proj}_K\left[\mathbf{R}^k - \mathbf{A}^{-1}\mathbf{C}(\mathbf{R})\right] \tag{10.15}$$

so that, as required, the iterates remain within the friction cone and where

$$\mathbf{A} \simeq \nabla\mathbf{C} \tag{10.16}$$

is an easy to invert approximation of $\nabla\mathbf{C}$. Since in many practical situations $\nabla\mathbf{C}$ is singular, we cannot hope to employ $\nabla\mathbf{C}$. We then employ two variants of $\mathbf{A} \simeq \nabla\mathbf{C}$. The first one reads

$$\mathbf{A}_1 = \nabla\mathbf{C} + \delta\mathbf{I}, \text{ combined with GMRES.} \tag{10.17}$$

where $\delta \geq 0$. This is related to numerical integration of an artificial ODE

$$\frac{d\mathbf{R}}{dt} = \mathbf{C}(\mathbf{R}) \tag{10.18}$$

to a steady state (take one step of implicit Euler, in the literature this is called *pseudo-transient continuation*). The second variant reads

$$\mathbf{A}_2 = \mathrm{diag}_{3\times 3}[\nabla\mathbf{C}], \text{ combined with direct inversion.} \tag{10.19}$$

and it is combined with a heuristic stabilization technique

$$\triangle\mathbf{R}^{k+1} = (1-\theta)\triangle\mathbf{R}^k - \theta\left(\mathbf{A}^k\right)^{-1}\mathbf{C}^k \tag{10.20}$$

where

$$\theta \in [0,1]. \tag{10.21}$$

We then have two variants of the projected quasi-Newton step:

1. PQN1 (cf. Algorithm 10.5):

$$\mathbf{R}^{k+1} = \mathrm{proj}_K\left[\mathbf{R}^k - \left(\nabla\mathbf{C}^k + \delta\mathbf{I}\right)^{-1}_{\mathrm{GMRES}_{(\epsilon\|\mathbf{C}^k\|,m)}}\mathbf{C}^k\right]$$

   where GMRES is preconditioned with $\left[\mathrm{diag}_{3\times 3}\left(\nabla\mathbf{C}^k_{\alpha\alpha} + \delta\mathbf{I}\right)\right]^{-1}$ and δ, ϵ and m need to be suitably selected. The linear problem should be solved only roughly, usually $\epsilon = 0.25$ and $m = 10$ (iterations bound)

Algorithm 10.5 The projected quasi-Newton method with GMRES. Use linver = 'GMRES' in NEWTON_SOLVER to enable this variant (this is the default).

PQN1($\mathbf{R}, \gamma, n, \omega, \delta, m, \epsilon$):

1. $\triangle \mathbf{R}^0 = \mathbf{0}$, $k = 0$.
2. Do
   (a) $\mathbf{U}^k = \mathbf{W}\mathbf{R}^k + \mathbf{B}$.
   (b) Compute $\mathbf{C}^k$ and $\mathbf{A}^k = \nabla \mathbf{C}^k_{\alpha\alpha} + \delta \mathbf{I}$ using smoothing ω.
   (c) $\triangle \mathbf{R}^{k+1} = -\left(\mathbf{A}^k\right)^{-1}_{\mathrm{GMRES}_{(\epsilon \| \mathbf{C}^k \|, m)}} \mathbf{C}^k$.
   (d) $\mathbf{R}^{k+1} = \mathrm{proj}_K \left[\mathbf{R}^k + \triangle \mathbf{R}^{k+1}\right]$.
   (e) $k = k + 1$.

   while $g\left(\mathbf{R}^k\right) \geq \gamma$ and $k < n$.

work well. For ill-conditioned problems a too accurate solution of the linear sub-problem results in a poor convergence rate. The diagonal regularization δ needs to be adjusted "by hand". The automatic update formulas that can be found in literature work only for well-conditioned cases and hence they are not very useful for us. For ill-conditioned problems one should pick δ that delivers an overall best convergence behavior. Large values will slow down convergence, but stabilize it; small values may destabilize convergence for ill-conditioned problems; δ (typically $\ll 1$) should be tuned together with ϵ and m (e.g. find a suitably small δ first, then tweak ϵ). *Since rigorous analysis is still missing for these parameters, please experiment before settling on specific values for a specific problem.* Use linver = 'GMRES' in NEWTON_SOLVER to enable this variant (this is the default).

2. PQN2 (cf. Algorithm 10.6):

$$\mathbf{R}^{k+1} = \mathrm{proj}_K \left[\mathbf{R}^k + (1-\theta) \triangle \mathbf{R}^k - \theta \left(\mathrm{diag}_{3\times 3}\left[\nabla \mathbf{C}^k\right]\right)^{-1} \mathbf{C}^k\right]$$

where $\theta \in [0, 1]$ and the diagonal 3×3 blocks of $\nabla \mathbf{C}^k$ are directly inverted. This simple scheme is interesting because it converges for a sufficiently small θ, while it is essentially a nonlinear Jacobi-type method. Use linver = 'DIAG' in NEWTON_SOLVER to enable this variant.

Algorithm 10.6 The projected quasi-Newton method with diagonal direct solve and averaging. Use linver = 'DIAG' in NEWTON_SOLVER to enable this variant.

PQN2($\mathbf{R}, \theta, \gamma, n, \omega$):

1. $\triangle \mathbf{R}^0 = \mathbf{0}$, $k = 0$.
2. Do
   (a) $\mathbf{U}^k = \mathbf{W}\mathbf{R}^k + \mathbf{B}$.
   (b) Compute $\mathbf{C}^k$ and $\mathbf{A}^k = \text{diag}_{3\times 3}\left[\nabla \mathbf{C}^k_{\alpha\alpha}\right]$ using smoothing ω.
   (c) $\triangle \mathbf{R}^{k+1} = (1-\theta)\triangle \mathbf{R}^k - \theta\left(\mathbf{A}^k\right)^{-1}\mathbf{C}^k$.
   (d) $\mathbf{R}^{k+1} = \text{proj}_K\left[\mathbf{R}^k + \triangle \mathbf{R}^{k+1}\right]$.
   (e) $k = k + 1$.

   while $g\left(\mathbf{R}^k\right) \geq \gamma$ and $k < n$.

Bibliography

[1] V. Acary and B. Brogliato. *Numerical Methods for Nonsmooth Dynamical Systems*, volume 35 of *Lecture Notes in Applied Lecture Notes in Applied and Computational Mechanics*. Springer Verlang, 2008.

[2] Mark F. Adams. A distributed memory unstructured Gauss-Seidel algorithm for multigrid smoothers. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, pages 4–4, New York, USA, 2001. ACM Press.

[3] Bernard Brogliato. *Nonsmooth Mechanics*. Communications and Control Engineering. Springer Verlang, 1999.

[4] G. De Saxcé and Z. Q. Feng. The bipotential method: a constructive approach to design the complete contact law with friction and improved numerical algorithms. *Mathematical and Computer Modelling*, 28:225–245, 1998.

[5] Masao Fukushima, Zhi-Quan Luo, and Paul Tseng. Smoothing functions for second-order-cone complementarity problems. *SIAM Journal on Optimization*, 12(2):436–460, 2002.

[6] Ch. Glocker. *Set-Valued Force Laws*, volume 1 of *Lecture Notes in Applied and Computational Mechanics*. Springer Verlang, 2001.

[7] M. Jean. The non-smooth contact dynamics method. *Computer Methods in Applied Mechanics and Engineering*, 177(3-4):235–257, 1999.

[8] Tomasz Koziara. *Aspects of computational contact dynamics*. PhD thesis, University of Glasgow, http://theses.gla.ac.uk/429/, 2008.

[9] Tomasz Koziara and Nenad Bićanić. Simple and efficient integration of rigid rotations suitable for constraint solvers. *Journal for Numerical Methods in Engineering*, 81(9):1073 – 1092, 2009.

[10] R.I. Leine and N. van de Wouw. *Stability and Convergence of Mechanical Systems with Unilateral Constraints*, volume 36 of *Lecture Notes in Applied and Computational Mechanics*. Springer Verlang, 2008.

[11] J. J. Moreau. *Unilateral Contact and Dry Friction in Finite Freedom Dynamics*, volume 302 of *Non-smooth Mechanics and Applications, CISM Courses and Lectures*. Springer, Wien, 1988.

[12] J. J. Moreau. Numerical aspects of the sweeping process. *Computer Methods in Applied Mechanics and Engineering*, 177(3-4):329–349, 1999.

[13] J. J. Moreau. Some basics of unilateral dynamics. In F. Pfeiffer and C. Glocker, editors, *Unilateral Multibody Contacts*. Kluwer, Dordrecht, 1999.

[14] Christian Studer. *Numerics of Unilateral Contacts and Friction*, volume 47 of *Lecture Notes in Applied and Computational Mechanics*. Springer Verlang, 2009.

[15] M. Zhang and R.D. Skeel. Cheap implicit symplectic integrators. *Applied Numerical Mathematics*, 25:297–302(6), 1997.