Solfec User Manual

Tomasz Koziara

4th January 2010

Contents

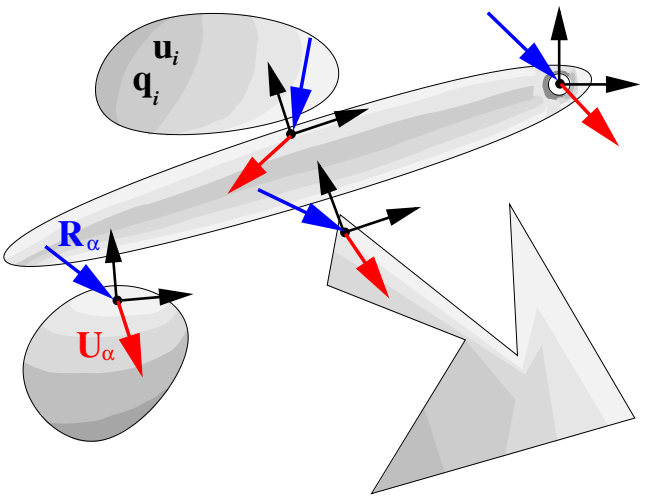
Chapter 1

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 [6] and Jean [3], hence the constraints are solved 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 a figure below



There are four bodies in the figure. Placement of each point of every body is determined by the configuration $\mathbf{q}_i$. Velocity of each point of every body is determined by the 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 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

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

where $\mathbf{M}$ is an inertia operator, $\mathbf{f}$ is an out of balance force, $\mathbf{H}$ is a linear operator, while $\mathbf{R}$ collects some local point forces $\mathbf{R}_\alpha$. While integrating the motion of bodies, one keeps track of a number of local coordinate systems (local frames). There are four of them in the figure. Each local frame is related to a pair of points, 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 in $\mathbf{M}^{-1}\mathbf{H}^T\mathbf{R}$, $\mathbf{R}$ should be interpreted as an *integral* of reactions over $[t, t+h]$. At a start we have

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

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. Finally

$$\mathbf{R} \text{ is such that } \mathbf{C}(\mathbf{R}) = \mathbf{C}(\mathbf{B} + \mathbf{W}\mathbf{R}, \mathbf{R}) = \mathbf{0} \tag{1.12}$$

where $\mathbf{C}$ is a nonlinear and usually nonsmooth operator. In order to obtain $\mathbf{C}$, we first need to compute

$$\mathbf{B} = \mathbf{H}\left(\mathbf{u}^t + \mathbf{M}^{-1} h \mathbf{f}^{t+\frac{h}{2}}\right) \tag{1.13}$$

and

$$\mathbf{W} = \mathbf{H}\mathbf{M}^{-1}\mathbf{H}^T \tag{1.14}$$

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

$$\mathbf{U} = \mathbf{B} + \mathbf{W}\mathbf{R} \tag{1.15}$$

maps constraint reactions $\mathbf{R}$ into local relative velocities $\mathbf{U} = \mathbf{H}\mathbf{u}^{t+h}$ at time $t+h$. Hence, $\mathbf{C}(\mathbf{R}) = \mathbf{C}(\mathbf{U}, \mathbf{R}) = \mathbf{0}$ describes some implicit relations between the velocities and the reactions. 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 $\mathbf{q}^{t+h} = \mathbf{q}^{t+\frac{h}{2}} + \frac{h}{2}\mathbf{u}^{t+h}$.

Let us refer the reader to [4] and references therein 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 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 a directory *solfec* in your current directory. The next thing you need is an ANSI C (and prospectively also FORTRAN) 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 (tested) from http://www.cygwin.com/ or Mingw (not tested) 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*

```
#
# Specify C compiler here
#
CC = cc
#
# Specify FORTRAN compiler here
#
FF = g95
#
# Debug or optimized version switch (yes/no)
#
DEBUG = yes
PROFILE = no
MEMDEBUG = no
GEOMDEBUG = no
PARDEBUG = no
NOTHROW = no
#
# POSIX
#
POSIX = yes
#
```

```
# 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
#
# MPI (yes/no)
#
MPI = yes
MPICC = mpicc
ZOLTANINC = -I/Users/tomek/Devel/lib/zoltan/include
ZOLTANLIB = -L/Users/tomek/Devel/lib/zoltan/lib -lzoltan
```

The above configuration works on Mac OS X. Examples for Linux and Cygwin can be found in *solfec/cfg*. At the current point a FORTRAN compiler is not really needed (there will be a need for it soon though due to planned work). What you need is:

- A **C** compiler
- **BLAS** and **LAPACK** libraries (standard on most systems)
- **Python** together with development files and libraries
- **OpenGL** libraries and developments files
- **MPI** libraries and development files
- **Zoltan** load balancing library

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 ./
```

- 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

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.

3.1 Solfec objects

3.1.1 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**

3.1.2 MESH

An object of type MESH describes an arbitrary volumetric mesh, comprising tetrahedrons, pyramids, wedges, and hexahedrons (Figure 3.1).

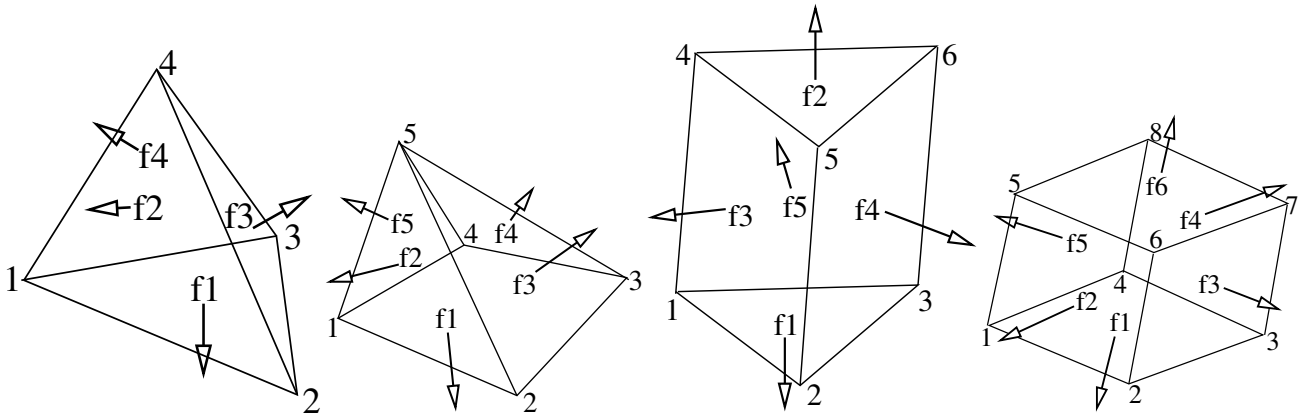


Figure 3.1: Element types in Solfec

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.node (n) *- returns a (x, y, z) tuple storing coordinates of nth node*

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.

3.1.3 SPHERE

An object of type SPHERE is either a sphere, or it is a collections of spheres.

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

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
- **sphere** (emptied) - collection of SPHERE objects appending **obj**

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

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

3.1.4 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 3.1.11)
obj.ncon - number of constraints
obj.bodies - list of bodies (cf. Section 3.1.7)
obj.nbod - number of bodies

Read/write members
obj.step

3.1.5 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.

Model name	Employs variables
'SIGNORINI_COULOMB'	friction, cohesion, restitution
'SPRING_DASHPOT'	spring, dashpot, friction, cohesion

Table 3.1: Surface material models.

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

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 3.1 and Chapter 6
- **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)

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

3.1.6 BULK_MATERIAL

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

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

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 3.2 and Chapter 6

Model name	Employs variables
'KIRCHHOFF'	young, poisson, density

Table 3.2: Bulk material models.

- **label** - label string (default: 'BULK_MATERIAL_i', where i is incremented for each call)
- **young** - Young's modulus (default: 1E6)
- **poisson** - Poisson's coefficient (default: 0.25)
- **density** - material density (default: 1E3)

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

3.1.7 BODY

An object of type BODY represents a solid body.

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

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
- **shape** (emptied) - this is can be a CONVEX/MESH/SPHERE object, or a list *[obj1, obj2, ...]*, where each object is of type CONVEX/MESH/SPHERE. 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 3.4)
- **label** - a label string (no label is assigned by default)
- **formulation** - valid when **kind** equals 'FINITE_ELEMENT', ignored otherwise (default: 'FEM_O1'). This argument specifies a formulation of the finite element method. See Table 3.3.

Formulation	Remarks
'FEM_O1'	Use first order elements
'FEM_O2'	Use second order elements **TODO: implement**

Table 3.3: Bulk material models.

- **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.

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

Read-only members
obj.kind, obj.label
obj.conf - tuple *(q1, q2, ..., qN)* storing configuration of the body. See Table 3.4.
obj.velo - tuple *(u1, u2, ..., uN)* storing velocity of the body. See Table 3.5.

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 3.6.
obj.damping - mass proportional damping coefficient (default: 0.0) for the dynamic case.

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'	Python *None* object.

Table 3.4: Types of configurations.

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'	Python *None* object.

Table 3.5: Types of velocities.

3.1.8 TIME_SERIES

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

Scheme	Kinematics	Remarks
'DEFAULT'	all	Use a default time integrator regardless of underlying kinematics.
'RIG_POS'	rigid	NEW1 in [5]: explicit, positive energy drift, no momentum conservation
'RIG_NEG'	rigid	NEW2 in [5]: explicit, negative energy drift, exact momentum conservation; **default** for rigid kinematics
'RIG_IMP'	rigid	NEW3 in [5]: semi-explicit, no energy drift and exact momentum conservation
'DEF_EXP'	pseudo-rigid, finite element	Explicit scheme described in Chapter 5 of [4]; **default** for deformable kinematics
'DEF_LIM'	pseudo-rigid, finite element	Linearly implicit scheme similar to [1]
'DEF_IMP'	pseudo-rigid, finite element	Implicit scheme similar to [7]

Table 3.6: Time integration schema.

obj = TIME_SERIES (points)

This routine creates a TIME_SERIES object.

- **obj** - created TIME_SERIES object
- **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

3.1.9 GAUSS_SEIDEL_SOLVER

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

obj = GAUSS_SEIDEL_SOLVER (epsilon, maxiter | 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
- **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 3.7.
- **diagepsilon** - diagonal block solver relative accuracy of constraint reactions (default: *epsilon / 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_SAXE_AND_FENG', cf. Chapter 7.

- **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 3.7
obj.iters - number of iterations during a last run of solver
obj.rerhist - if history recording is on, this is a list of relative error values for each iteration of the last run. Otherwise a *None* object is returned.

Read/write members
obj.epsilon, obj.maxiter, obj.diagepsilon, obj.diagmaxiter, obj.diagsolver
obj.history - 'ON' or 'OFF' flag switching history recording (default is 'OFF')
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 3.8. 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 3.7: 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.
'SIMPLIFIED'	A single sweep over contacts is done with previous values of off-processor data. This is followed by local Gauss-Seidel iterations for all non-contact constraints. This is the fastest and least consistent approach, of use for deformable kinematics dominated models.

Table 3.8: Variants of parallel Gauss-Seidel update.

3.1.10 EXPLICIT_SOLVER

An object of type EXPLICIT_SOLVER represents a penalty based constraint solver. 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 = EXPLICIT_SOLVER ()

- **obj** - created EXPLICIT_SOLVER object

3.1.11 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 (solfec, body, point)

This routine creates a fixed point constraint.

- **obj** - created CONSTRAINT object
- **solfec** - **obj** is created for this SOLFEC object
- **body** - BODY object whose motion is constrained
- **point** - *(x, y, z)* tuple with referential point coordinates

obj = FIX_DIRECTION (solfec, body, point, direction)

This routine fixes the motion of a referential point along a specified spatial direction.

- **obj** - created CONSTRAINT object
- **solfec** - **obj** is created for this SOLFEC 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

obj = FIX_SURFACE (solfec, body, surfid, direction) TODO: implement

This routine fixes the motion of a referential surface along a specified spatial direction.

- **obj** - list of created point CONSTRAINT objects
- **solfec** - **obj** is created for this SOLFEC object
- **body** - BODY object whose motion is constrained
- **surfid** - surface identifier
- **direction** - *(vx, vy, vz)* tuple defining the direction of load, or 'NORMAL' if load is normal to the surface, or 'TANGENT1' if load acts along the first tangent direction, or 'TANGENT2' if it acts along the second tangent direction. The normal direction ('NORMAL') is outward. The first tangent direction ('TANGENT1') is the one of the steepest descent, or a global x direction if the surface is horizontal. The second tangent direction ('TANGENT2') is such that the local ('TANGENT1', 'TANGENT2', 'NORMAL') coordinate system is right-handed.

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

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

- **obj** - created CONSTRAINT object
- **solfec** - **obj** is created for this SOLFEC 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 (solfec, body, point, direction, value)

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

- **obj** - created CONSTRAINT object
- **solfec** - **obj** is created for this SOLFEC 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 (solfec, body, point, direction, tms)

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

- **obj** - created CONSTRAINT object
- **solfec** - **obj** is created for this SOLFEC 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 (solfec, body1, body2, point1, point2)

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

- **obj** - created CONSTRAINT object
- **solfec** - **obj** is created for this SOLFEC 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

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 constraint reaction in a form of a tuple: *(RT1, RT2, RN)* 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.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.

3.2 Applying loads

Routines listed in this section apply loads.

GRAVITY (solfec, direction, value)

This routine sets up the gravitational acceleration.

- **solfec** - SOLFEC object for which the acceleration is set up
- **direction** - *(vx, vy, vz)* tuple defining the direction
- **value** - a number or a TIME_SERIES object defining the value of the acceleration

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 returnes a **force** tuple. For rigid body the force reads *(spatial force, spatial torque, referential torque)*, while for other hinds of bodies this is a generalised 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

LOAD (body, kind, surfid, direction, value) TODO: implement

This routine applies a surface load.

- **body** - BODY object to which the load is applied
- **kind** - either 'SPATIAL' or 'CONVECTED'; the *spatial* direction remains fixed, while the *convected* one follows deformation
- **surfid** - the integer surface identifier
- **direction** - *(vx, vy, vz)* tuple defining the direction of load, or 'NORMAL' if load is normal to the surface, or 'TANGENT1' if load acts along the first tangent direction, or 'TANGENT2' if it acts along the second tangent direction. The normal direction ('NORMAL') is outward. The first tangent direction ('TANGENT1') is the one of the steepest descent, or a global x direction if the surface is horizontal. The second tangent direction ('TANGENT2') is such that the local ('TANGENT1', 'TANGENT2', 'NORMAL') coordinate system is right-handed.
- **value** - a number or a TIME_SERIES object defining the value of the applied load

3.3 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, EXPLICIT_SOLVER)
- **duration** - duration of analysis

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 'FASTLZ'. 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: ∞)

3.4 Utilities

Various utility routines are listed below.

IMBALANCE_TOLERANCE (solfec, tolerance)

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)

num = NCPU ()

This routine returns the number CPUs used in the analysis.

- **num** - the number of CPUs

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. 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. 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. 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)

This routine splits a geometrical object (or a collection of objects) by a plane passing by a point.

- **one** - objects placed below the splitting plane (*None* if no objects were placed below)
- **two** - objects placed above the splitting plane (*None* if no objects were placed above)
- **shape** (emptied) - object, collection of objects, or a list *[a, b, c, ...]* of objects of type CONVEX or SPHERE
- **point** - *(px, py, pz)* tuple defining a point passed by the splitting plane
- **normal** - *(nx, ny, nz)* tuple defining the splitting plane normal

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

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'
- **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. Alternately this can be a single BODY tuple.

CONTACT_EXCLUDE_BODIES (solfec, 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).*

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

CONTACT_EXCLUDE_OBJECTS (solfec, body1, point1, body2, point2)

This routine disables contact detection for a specific pair of geometric objects (e.g. elements, convices, sheres). By default, between different bodies, contact detection is enabled for all possible object pairs. **NOTE:** *must be invoked on all processors during a parallel run (do not use from within a callback).*

- **solfec** - SOLFEC object
- **body1** - first BODY object
- **point1** - referential point properly contained in the 1st geometric object
- **bod2** - second BODY object
- **point2** - referential point properly contained in the 2nd geometric object

CONTACT_SPARSIFY (solfec, threshold)

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.

- **solfec** - SOLFEC object
- **threshold** - sparsification threshold (default: 0.01) from within the interval [0, 1]. Zero corresponds to the lack of sparsification.

3.5 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
- **solfec** - SOLFEC object
- **kind** - this is one of: 'TIMINT' (time integration), '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)

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: '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', 'FRICTION' and it corresponds to the energy kind
  - a string 'TIMINT', 'CONDET', 'LOCDYN', 'CONSOL', 'PARBAL' for timing histories
- **t0** - time interval start
- **t1** - time interval end
- **skip** - number of steps to skip between two time instants

Chapter 4

Tutorials

4.1 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.

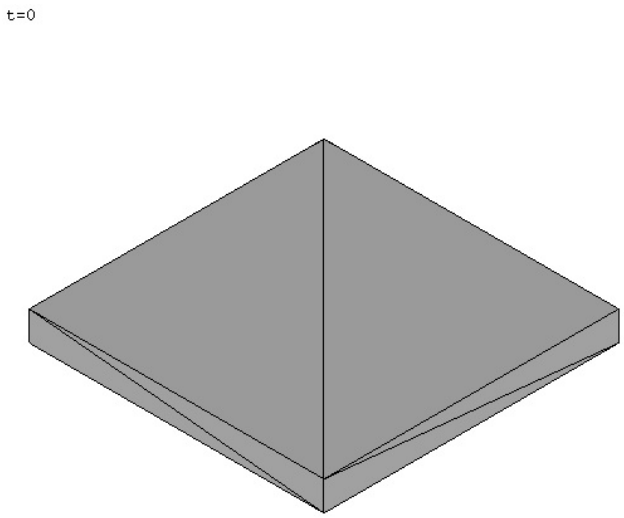
```
step = 1E-3
solfec = SOLFEC ('DYNAMIC', step, 'out/tutorail/three-basic-geometric-objects')
bulk = BULK_MATERIAL (solfec,
                      model = 'KIRCHHOFF',
                      young = 15E9,s
                      poisson = 0.3,
                      density = 2E3)
BODY (solfec, 'OBSTACLE',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

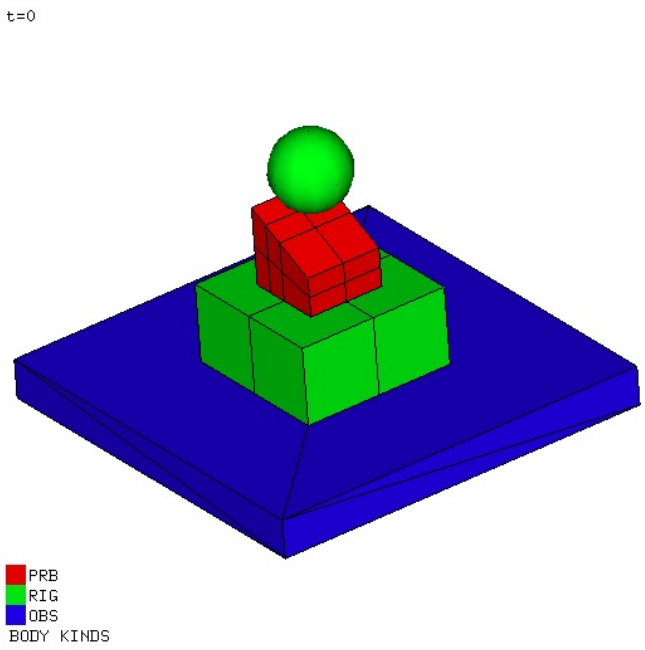
Note, that a base *brick* was created first. Then this brick was copied and manipulated into four different objects: *b1,b2, ..., b4*. The list of those objects was passed as a shape when creating the first rigid body. The second body will be pseudo-rigid and will have its shape defined by a mesh.

```
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, -1), 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)
```

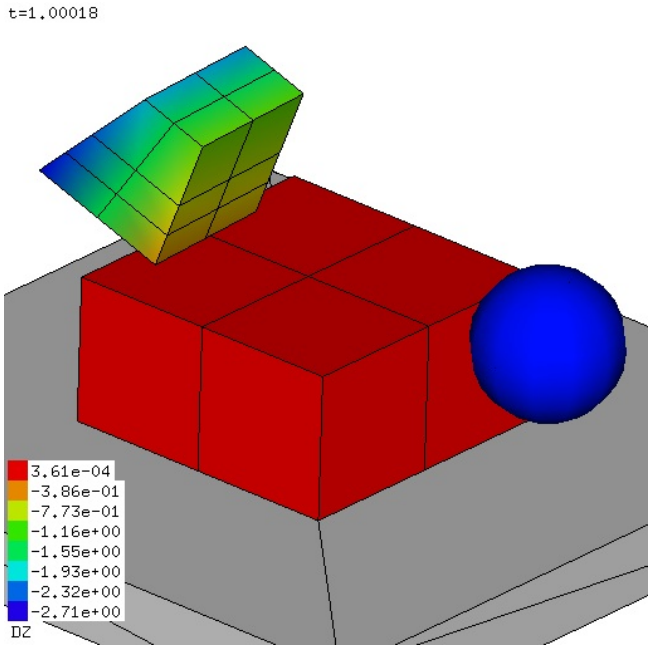
The relative constraint reaction accuracy was set to 1E-3, while the maximal the maximal number of iterations is 1000. It remains to run the simulation.

```
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.



4.2 Ball impact

This example illustrates using multiple SOLFEC objects, application of the EXPLICIT_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, -1), 10)

  xs = EXPLICIT_SOLVER ()

  RUN (solfec, xs, stop)

  return solfec
```

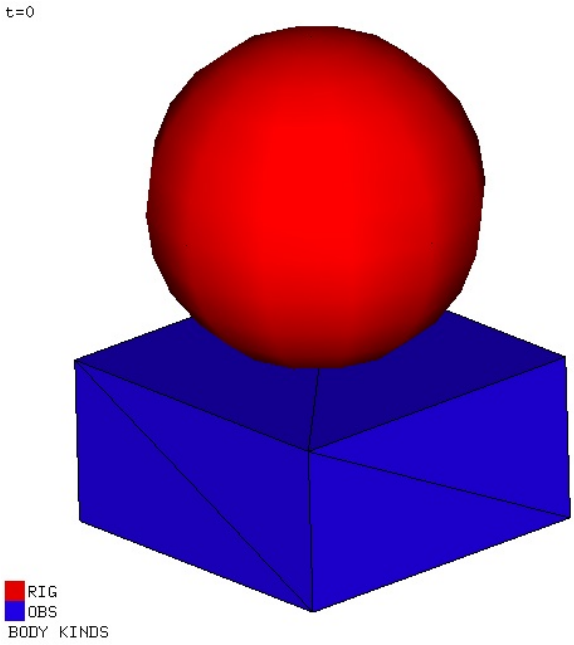
The above code is similar to the previous example. Here though the complete model is created inside of a Python function called *sphere_impact*. By itself this will not run any simulation - this function needs to be called from the main module of the input file (we remind that Python uses indentation 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 *sphere_impact* function. It should be noted, that the SOLFEC object is returned from the routine. Now, here is the main module

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

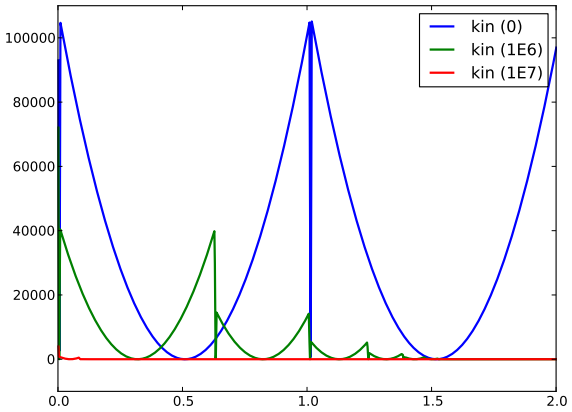
First, we check whether Solfec is not run with the *-v* option and whether it is in 'READ' mode. This is the case, when after running the complete analysis

```
./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/sphere-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 5

Contact points

Shapes are defined as juxtapositions of convex objects (Sections 3.1.1, 3.1.2, 3.1.3). Let us call them *elements*. The *surface elements* are adjacent to the surface of bodies. Contact points are characterised by a location and a normal direction - they are *oriented*. In Solfec *a single oriented contact point results from an overlap of two surface elements*. This is motivated by two factors:

1. The point and the normal direction derived from an overlap of two elements are well defined for nonsmooth geometry.
2. We wish to use as few contact points as possible, but still be able to control the accuracy of contact resolution by mesh refinement.

Two elements are in contact if their intersection is not empty. The intersection is d-dimensional, where $d \in \{0, 1, 2, 3\}$. Only the 3-dimensional, volumetric intersection is considered, while the remaining cases are cast into the volumetric one through a small expansion of the elements. Let elements e_1 and e_2 overlap like in Figure 5.1. Their intersection $o = e_1 \cap e_2$ is a convex polyhedron, with the surface containing two parts $\partial o_1 \cup \partial o_2 \subset \partial o$, where $\partial o_k \subset \partial e_k$ and $\partial o_k = \partial o \cap \partial \mathcal{B}_k$. For each part, one can compute the resultant normal

$$\bar{\mathbf{n}}_k = \int_{\partial o_k} \mathbf{n} da \tag{5.1}$$

and the variation of normal

$$\tilde{\mathbf{n}}_k = \int_{\partial o_k} (\mathbf{n} - \bar{\mathbf{n}}_k)^2 \, da \tag{5.2}$$

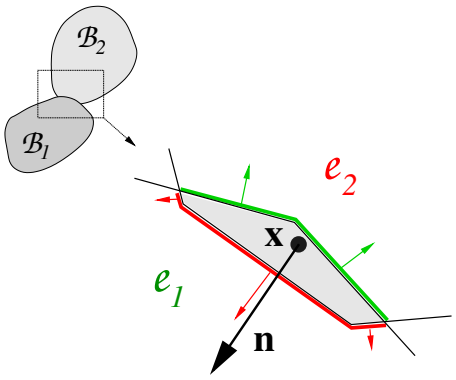


Figure 5.1: A contact point $\mathbf{x}$ and a normal $\mathbf{n}$ extracted from the intersection of two convex elements.

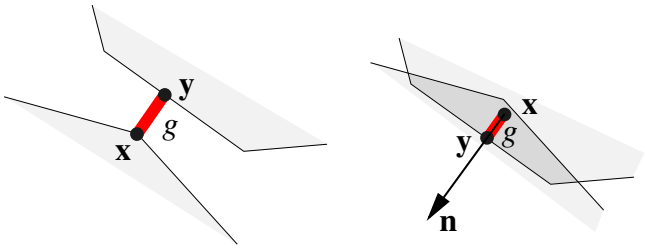


Figure 5.2: Gap according to definition (5.5).

The final outward normal is the one with a smaller variation

$$\mathbf{n}\left(e_1 \cap e_2\right) = \frac{\bar{\mathbf{n}}_k}{\|\bar{\mathbf{n}}_k\|}, \quad k = \arg\min_i \left(\|\tilde{\mathbf{n}}_i\|\right) \tag{5.3}$$

The contact point can be computed as the mass centre of the surface part with the larger variation of normal (hence, it is a deeper submerged point of the two possibly computed this way)

$$\mathbf{x}\left(e_1 \cap e_2\right) = \frac{\int_{\partial o_k} \mathbf{x} da}{\int_{\partial o_k} da}, \quad k = \arg\max_i \left(\|\tilde{\mathbf{n}}_i\|\right) \tag{5.4}$$

The gap function between a pair of elements e_1 and e_2 is defined in the following way

$$gap\left(t\right) = \begin{cases} \min_{\mathbf{x},\mathbf{y}} \|\mathbf{x}-\mathbf{y}\| : \mathbf{x} \in \bar{e}_1, \mathbf{y} \in \bar{e}_2 & \text{when } \bar{e}_1 \cap \bar{e}_2 = \emptyset \\ \min_{\mathbf{y}} \langle \mathbf{n}, \mathbf{x}-\mathbf{y} \rangle : \mathbf{y} \in \partial e_k, \mathbf{x} \in \bar{e}_1 \cap \bar{e}_2 & \text{otherwise} \end{cases} \tag{5.5}$$

where in the second line, $\mathbf{x}$ and $\mathbf{n}$ are given by (5.4) and (5.3). The latter formula defines also the k-index. The first line describes the proximity of the two elements. The second one defines a negative distance along $\mathbf{n}$, from $\mathbf{x}$ towards the surface of the intersection $\bar{e}_1 \cap \bar{e}_2$ (Figure 5.2).

Chapter 6

Materials

6.1 Surface materials

A surface material is assigned to a pairing of surfaces. See Section 3.1.5 for the input syntax.

6.1.1 Signorini-Coulomb

The velocity-level formulation of the Signorini condition reads

$$\bar{U}_N \geq 0 \;\; R_N \geq 0 \;\; \bar{U}_N R_N = 0 \tag{6.1}$$

where

$$\bar{U}_N = U_N^{t+h} + \eta \min\left(0, U_N^t\right) \tag{6.2}$$

U_N is the the normal relative velocity, R_N is the normal reaction and η is the velocity *restitution* coefficient. The normal direction is consistent with the positive gap velocity. Thus (6.1) states that any violation of the non-penetration results in a reactive force or velocity driving at the penetration-free configuration. Using the dashed velocity $\bar{U}_N$ allows to include the simple Newton impact law. ***One must be aware***, that only the cases *restitution* $\in \{0, 1\}$ ($\eta = 0$ or $\eta = 1$) are energy consistent (cannot cause energy growth, cf. Section 10.5 in [4]). **TODO: improve**

The Coulomb's friction law can be expressed in the form of the maximal dissipation principle

$$\mathbf{R}_T \in D\left(\mu R_N\right) \;\; \forall \mathbf{S} \in D\left(\mu R_N\right) \;\; \langle \mathbf{S} - \mathbf{R}_T, \mathbf{U}_T \rangle \geq 0 \tag{6.3}$$

where $D\left(\mu R_N\right)$ is the R_N-level section of the friction cone, $\mathbf{R}_T$ is the two-component tangential force, $\mathbf{S}$ is any force in $D\left(\mu R_N\right)$ and $\mathbf{U}_T$ is the tangential relative velocity. In the above $\langle \cdot, \cdot \rangle$ stands for a scalar product in an Euclidean 2-space. Thus 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 Signorini-Coulomb law can be expressed in the form of projections

$$\begin{cases} R_N = \mathrm{proj}_{R_+}\left(R_N - \rho \bar{U}_N\right) \\ \mathbf{R}_T = \mathrm{proj}_{D(\mu R_N)}\left(\mathbf{R}_T - \rho \mathbf{U}_T\right) \end{cases} \tag{6.4}$$

where $\rho > 0$. Writing

$$d_N\left(U_N, R_N\right) = R_N - \rho \bar{U}_N \tag{6.5}$$

$$\mathbf{d}_T\left(\mathbf{U}_T, \mathbf{R}_T\right) = \mathbf{R}_T - \rho \mathbf{U}_T \tag{6.6}$$

allows to rewrite the projection formulae as

$$\mathbf{C}\left(\mathbf{U},\mathbf{R}\right)=\left[\begin{array}{c}\max\left(\mu d_N,\left\|\mathbf{d}_T\right\|\right)\mathbf{R}_T-\mu\max\left(0,d_N\right)\mathbf{d}_T\\ R_N-\max\left(0,d_N\right)\end{array}\right] \tag{6.7}$$

which is a nonsmooth constraint equation of the frictional contact law [2]. We refer the reader to Chapter 10 of [4] for more details.

6.1.2 Spring-dashpot

The normal reaction is computed as follows

$$R_N=-spring\cdot gap-dashpot\cdot\frac{U_N^{t+h}+U_N^t}{2} \tag{6.8}$$

where U_N is the normal relative velocity. We use the block diagonal local dynamics equation

$$\mathbf{U}^{t+h}=\mathbf{B}+\mathbf{W}\mathbf{R} \tag{6.9}$$

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

$$U_N^{t+h}=B_N+\mathbf{W}_{NT}\mathbf{R}_T+W_{NN}R_N \tag{6.10}$$

where previous tangential reaction $\mathbf{R}_T$ is employed. Inserting this it into (6.8) results in

$$R_N=\left[-spring\cdot gap-dashpot\cdot\frac{B_N+\mathbf{W}_{NT}\mathbf{R}_T+U_N^t}{2}\right]/\left[1+dashpot\frac{W_{NN}}{2}\right] \tag{6.11}$$

We then utilise the tangential part of local dynamics and assuming $\mathbf{U}_T^{t+h}=0$ estimate the tangential *stick* reaction

$$\mathbf{R}_T=-\mathbf{W}_{TT}^{-1}\left(\mathbf{B}_T+\mathbf{W}_{TN}R_N\right) \tag{6.12}$$

Let μ stand for the coefficient of friction. The complete interface law reads

`if not` *cohesive* `and` $gap\geq 0$ `then` $\mathbf{R}=\mathbf{0}$
`else`
  $R_N=\left[-spring\cdot gap-0.5\cdot dashpot\cdot\left(B_N+\mathbf{W}_{NT}\mathbf{R}_T+U_N^t\right)\right]/\left[1+0.5\cdot dashpot\cdot W_{NN}\right]$
  `if not` *cohesive* `and` $R_N<0$ `then` $R_N=0$
  $\mathbf{R}_T=-\mathbf{W}_{TT}^{-1}\left(\mathbf{B}_T+\mathbf{W}_{TN}R_N\right)$
  `if` *cohesive* `and` $\left|R_N\right|>cohesion$ `then` $cohesive=false$
  `if not` *cohesive* `and` $\left\|\mathbf{R}_T\right\|>\mu R_N$ `then` $\mathbf{R}_T=\mu R_N\mathbf{R}_T/\left\|\mathbf{R}_T\right\|$

6.2 Bulk materials

A bulk material is assigned to a volume. See Section 3.1.6 for the input syntax.

6.2.1 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

$$\Psi=\frac{1}{4}\left[\mathbf{F}^T\mathbf{F}-\mathbf{I}\right]:\mathbf{C}:\left[\mathbf{F}^T\mathbf{F}-\mathbf{I}\right] \tag{6.13}$$

where

$$C_{ijkl} = \lambda \delta_{ij} \delta_{kl} + \mu \left[\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk} \right] \tag{6.14}$$

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

$$\lambda = \frac{E\nu}{(1+\nu)(1-2\nu)} \tag{6.15}$$

$$\mu = \frac{E}{2+2\nu} \tag{6.16}$$

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

$$\mathbf{P} = \partial_{\mathbf{F}} \Psi (\mathbf{F}) \tag{6.17}$$

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

Chapter 7

Solvers

7.1 Gauss-Seidel solver

7.2 Explicit solver

Bibliography

[1] Mihai Anitescu Bogdan Gavrea Jeff Trinkle Florian A. Potra. A linearly implicit trapezoidal method for integrating stiff multibody dynamics with contact, joints, and friction. *International Journal for Numerical Methods in Engineering*, 66:1079–1124, 2006.

[2] S. Hüeber, G. Stadler, and B. I. Wohlmuth. A primal-dual active set algorithm for three-dimensional contact problems with Coulomb friction. *SIAM Journal on Scientific Computing*, 30(2):572–596, 2007.

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

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

[5] Tomasz Koziara and Nenad Bićanić. Simple and efficient integration of rigid rotations suitable for constraint solvers. *To appear in the International Journal for Numerical Methods in Engineering*, 2009.

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

[7] T. A. Laursen V. Chawla. Energy consistent algorithms for frictional contact problems. *International Journal for Numerical Methods in Engineering*, 42:799–827, 1998.