Solfec User Manual

Tomasz Koziara

October 19, 2009

1 Introduction

2 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.

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**

2.2 MESH

An object of type MESH describes an arbitrary volumetric mesh, comprising tetrahedrons, pyramids, wedges, and hexahedrons (Figure 2.1).

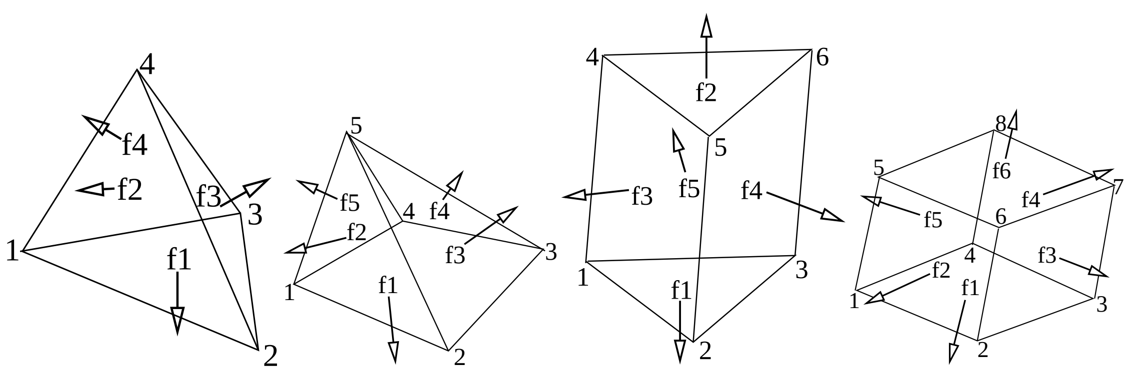

Figure 2.1: Element types in Solfec

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 inerpolation.
- **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 normalized (actual numbers do not matter, but their ratios) and applied to the local x direction of the generated shape.

2.3 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

2.4 SOLFEC

An object of type SOLFEC represents the Solfec algorithm. One can use several SOLFEC objects to run several analyzes 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 2.11)
obj.ncon - number of constraints
obj.bodies - list of bodies (cf. Section 2.7)
obj.nbod - number of bodies

Read/write members
obj.step

2.5 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.

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_COULMB'), see Table 2.1 and Chapter 4
- **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)

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

Table 2.1: Surface material models.

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

2.6 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 2.2 and Chapter 4
- **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

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

Table 2.2: Bulk material models.

2.7 BODY

An object of type BODY represents a solid body.

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

This routine creates a body.

- **obj** - created BODY object
- **solfec - obj** is created for this SOLFEC object
- **kind** - a string: 'RIGID', 'PSEUDO_RIGID', 'EXTENDED_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', only a single MESH object can be given. If the **kind** is 'EXTENDED_PSEUDO_RIGID' then **shape** can be a list of lists *[list1, obj2, list3, obj4, ...]*, where for example *list1 = [objA, objB, ...]* and corresponds to a first extended pseudo-rigid element, while *obj2* by itself becomes a second extended pseudo-rigid element; hence, passing just one object as a shape reduces the 'EXTENDED_PSEUDO_RIGID' model to the 'PSEUDO_RIGID' one.
- **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 2.14)
- **label** - a label string (no label is assigned by default)
- **formulation** - valid when **kind** equals 'FINITE_ELEMENT', ignored otherwise (default: 'HEX_TET_O1'). This argument specifies a formulation of the finite element method. See Table 2.3.

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 2.4.
obj.velo - tuple *(u1, u2, ..., uN)* storing velocity of the body. See Table 2.5.

Formulation	Remarks
'TET_O1'	Use first order tetrahedrons only
'TET_O2'	Use second order tetrahedrons only
'HEX_TET_O1'	Use first order hexahedrons whenever possible or tetrahedrons otherwise

Table 2.3: Bulk material models.

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 2.6.

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.
'EXTENDED_PSEUDO_RIGID'	Column-wise deformation gradients from all elements followed by the current mass center.
'FINITE_ELEMENT'	Current coordinates x, y, z of consecutive mesh nodes.
'OBSTACLE'	Python *None* object.

Table 2.4: Types of configurations.

2.8 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
- **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

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.
'EXTENDED_PSEUDO_RIGID'	Deformation gradient velocities followed by the spatial velocity of mass center.
'FINITE_ELEMENT'	Components x, y, z of spatial velocities consecutive mesh nodes.
'OBSTACLE'	Python *None* object.

Table 2.5: Types of velocities.

Scheme	Body kind	Remarks
'DEFAULT'	all	Use a default time integrator regardless of underlying kinematics.
'RIG_POS'	'RIGID'	NEW1 in [1]: explicit, positive energy drift, no momentum conservation
'RIG_NEG'	'RIGID'	NEW2 in [1]: explicit, negative energy drift, exact momentum conservation; **default** for rigid kinematics
'RIG_IMP'	'RIGID'	NEW3 in [1]: semi-explicit, no energy drift and exact momentum conservation

Table 2.6: Time integration schema.

2.9 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 othewise the 'EXIT' scenario is executed). In all cases ***obj.error*** variable is set up, cf. Table 2.7.

- **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 5.
- **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 2.7
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 - a variant of parallel algorithm (ignored during sequential runs, dafult: 'MID_LOOP'), cf. Table 2.8

2.10 EXPLICIT_SOLVER

An object of type EXPLICIT_SOLVER represents a penlaty based constraint solver. When in use, all 'SIGNORONI_COULOMB' type contact interfaces are regarded as 'SPRING_DASHPOT' ones. One should then remember about specyfying the *spring* value for those.

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

Table 2.7: Error codes of GAUSS_SEIDEL_SOLVER object

'MID_LOOP'	Process mid-nodes in a semi-sequential loop with processor coloring
'MID_THREAD'	Process the loop in a separate thread
'MID_TO_ALL'	Gather mid nodes at all processors
'MID_TO_ONE'	Gather mid nodes at one processor (with smallest load)
'NOB_*'	e.g. 'NOB_MID_LOOP', etc. Use non-blocking communication when possible

Table 2.8: Error codes of GAUSS_SEIDEL_SOLVER object

obj = EXPLICIT_SOLVER ()

- **obj** - created EXPLICIT_SOLVER object

2.11 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 constrint.

- **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

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, tms)

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
- **tms** - 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.

2.12 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 *spactial* 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:

$$\textit{\textbf{force = value\_callback (data, q, u, time, step)}}$$

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 *spactial* 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

This routine applies a surface load.

- **body** - BODY object to which the load is applied
- **kind** - either 'SPATIAL' or 'CONVECTED'; the *spactial* 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

2.13 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: ∞)

2.14 Utilities

Various utility routines are listed below.

IMBALANCE_TOLERANCES (solfec, timint, condet, locdyn)

This routine sets the imbalance tolerances 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
- **timint** - time integration imbalance tolerance (default: 1.3)
- **condet** - contact detection imbalance tolerance (default: 1.3)
- **locdyn** - local dynamics imbalance tolerance (default: 1.3)

LOCDYN_BALANCING (solfec, mode)

This routine sets the mode of local dynamics balancing for a parallel run. It is ignored in sequential mode.

- **solfec** - SOLFEC object
- **mode** - 'OFF' disables balancing (in this case geometrical balancing related to contact detection is inherited by local dynamics), 'GEOM' enables geometrical balancing based on spatial constraint points, 'GRAPG' enables topological balancing based on the adjacency structure of the $\mathbf{W}$ operator.

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 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 sometimes 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** - $(t_{11}, t_{21}, \dots, t_{33})$ 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.
- **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.
- **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.
- **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 coppied 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 labeled object inside of a SOLFEC object.

- **obj** - returned object (*None* if a labeled object was not found)
- **solfec** - SOLFEC object
- **kind** - labeled 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.

- **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.

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

2.15 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 returnes 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

hist = HISTORY (body, point, entity, t0, t1)

This routine outputs the history of an entity at a referential point.

- **hist** - a tuple of list objects storing the history of the entity: *(times, values)*
- **body** - BODY object
- **point** - *(x, y, z)* tuple storing the referential point
- **entity** - this is one of: 'DX', 'DY', 'DZ' (displacement), 'VX', 'VY', 'VZ' (velocity), 'SX', 'SY', 'SZ', 'SXY', 'SXZ', 'SYZ' (stress), 'MISE' (von Mises norm of stress)
- **t0** - time interval start
- **t1** - time interval end

ene = ENERGY (obj, kind) TODO

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

- **ene** - value of the energy
- **obj** - this can be: a SOLFEC object, a BODY object, or a list of BODY objects
- **kind** - this is one of: 'KINETIC', 'POTENTIAL', 'EXTWORK' (work of external forces, including friction), 'FRICWORK' (work of friction forces)

hist = ENERGY_HISTORY (obj, kind, t0, t1) TODO

This routine outputs the history of energy for a collection of objects.

- **hist** - a tuple of list objects storing the history of the entity: *(times, values)*
- **obj** - this can be: a SOLFEC object, a BODY object, or a list of BODY objects
- **kind** - same as **kind** in **ENERGY** routine
- **t0** - time interval start
- **t1** - time interval end

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), 'TIMBAL' (time integration balancing), 'CONBAL' (contact detection balancing), 'LOCBAL' (local dynamics balancing). The balancing timings are non-zero only for parallel runs.

hist = TIMING_HISTORY (solfec, kind, t0, t1)

This routine outputs the history of timing.

- **hist** - a tuple of list objects storing the history of timing: *(times, values)*
- **solfec** - SOLFEC object
- **kind** - same as **kind** in **TIMING** routine
- **t0** - time interval start
- **t1** - time interval end

3 Parallelism

4 Material models

5 Solvers

6 Examples

Bibliography

[1] 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.