Sofa Documentation

Also See **http://www.sofa-framework.org/documentation**

The Sofa Team

April 13, 2012

Abstract

SOFA (Simulation Open Framework Architecture) is an open-source C++ library primarily targeted at interactive medical simulation. SOFA facilitates collaborations between specialists from various domains, by decomposing complex simulators into components designed independently and organized in a scenegraph data structure. Each component encapsulates one of the aspects of a simulation, such as the degrees of freedom, the forces and constraints, the differential equations, the main loop algorithms, the linear solvers, the collision detection algorithms or the interaction devices. The simulated objects can be represented using several models, each of them optimized for a different task such as the computation of elastic forces, collision detection, haptics or visual display. To ensure a consistent simulation, these models are synchronized during the simulation using a mapping mechanism. CPU and GPU implementations can be transparently combined to exploit the computational power of modern hardware architectures. As a result of this flexible and efficient architecture, SOFA can be used as a test-bed to compare models and algorithms, or as a basis for the development of complex, high-performance simulators.

Contents

Part I

Foundation of SOFA

Chapter 1

Introduction to SOFA

1.1 Why SOFA?

Programming interactive physical simulation of rigid and deformable objects requires multiple skills in geometric modeling, computation mechanics, numerical analysis, collision detection, rendering, user interface and haptics feedback, among others. It is also challenging from a software engineering standpoint, with the need for computationally efficient algorithms, multi-threading, or the deployment of applications on modern hardware architectures such as the GPU. The development of complex medical simulations has thus become an increasingly complex task, involving more domains of expertise than a typical research and development team can provide. The goal of SOFA is to address these issues within a highly modular yet efficient framework, to allow researchers and developers to focus on their own domain of expertise, while re-using other expert's contributions.

1.2 The "philosophy" of SOFA

SOFA introduces the concept of multi-model representation to easily build simulations composed of complex anatomical structures. The pool of simulated objects and algorithms used in a simulation (also called a scene) is described using a hierarchical data structure similar to scene graphs used in graphics libraries. Simulated objects are decomposed into collections of independent components, each of them describing one feature of the model, such as state vectors, mass, forces, constraints, topology, integration scheme, and solving process. As a result, switching from internal forces based on springs to a finite element approach can be done by simply replacing one component with another, all the rest (mass, collision models, time integration, ...) remaining unchanged. Similarly, it is possible to keep the same force model and modify the solver and state vectors in order to compute the model on the GPU instead of the CPU. Moreover, the simulation algorithms, embedded in components, can be customized with the same flexibility as the physical models.

In addition to this first level of modularity, it is possible to go one step further and decompose simulated objects into multiple partial models, each optimized for a given type of computation. Typically, a physical object in SOFA is described using three partial models: a mechanical model with mass and constitutive laws, a collision model with contact geometry, and a visual model with detailed geometry and rendering parameters. Each model can be designed independently of the others, and more complex combinations are possible, for instance for coupling two different physical objects. During run-time, the models are synchronized using a generic mechanism called *mapping* to propagate forces and displacements. The user can interact in real-time with the mechanical models simulated in SOFA, using the mouse but also using other type of input device. Haptic rendering is also supported.

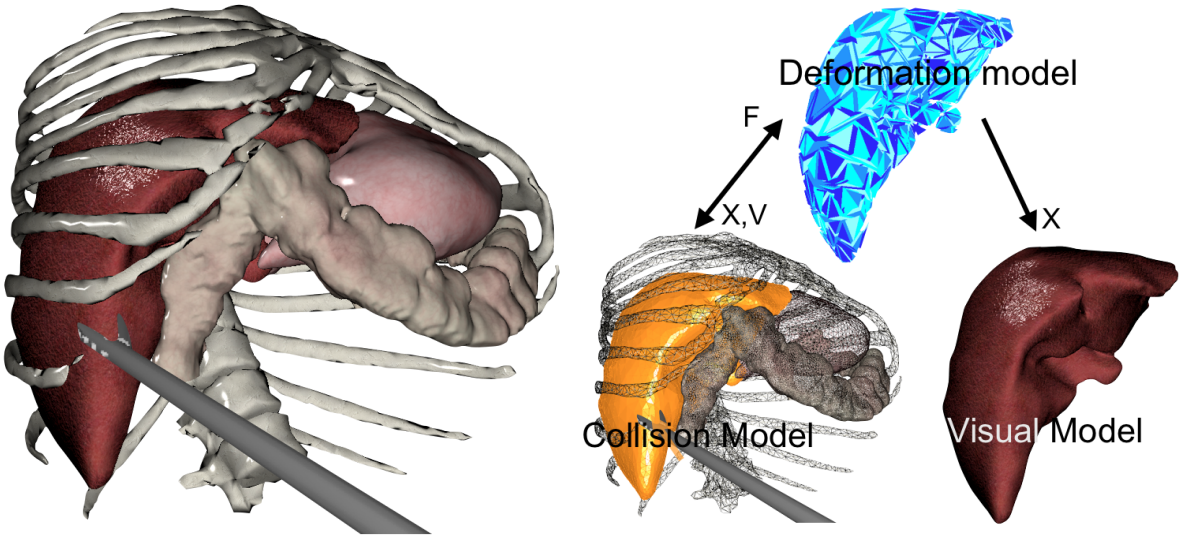


Figure 1.1: A simulated Liver Left: The simulation of the liver (dataset from IRCAD, France). Right: Three representations are used for the liver: one for the internal mechanics, one for the collisions, and one for the visualization. These representations are linked using mappings (black arrows).

1.3 Why should I contribute to SOFA?

SOFA was first released in 2007 [1]. Since then, it has evolved toward a comprehensive, high-performance library used by an increasing number of academics and commercial companies. The code is open-source and the license is LGPL. You can use this code to build your own medical simulations needs or for other applications. You can also include this code in a commercial product. The only requirement is that if you modify the code for a commercial product you need to share this modification with your client.

Moreover, you can build upon SOFA using the plug-in system. Your plug-in can have an other license than LGPL. Consequently there is a considerable freedom for you to use SOFA for your research, your developments or your products !

Finally, SOFAis also intended for the research community to help the development and the sharing of newer algorithms and models. So, do not hesitate to share your experience of SOFA, your code and your results with the SOFAcommunity !!!

Consider the deformable model of a liver shown in the left of Figure 1.1. It is surrounded by different anatomical structures (including the diaphragm, the ribs, the stomach, the intestines...) and is also in contact with a grasper (modeled as an articulated rigid chain). In SOFA, this liver is simulated using three different representations: the first is used to model its internal mechanical behavior, which may be computed using Finite Element Method (FEM) or other models. The geometry of the mechanical model is optimized for the internal force computations, e.g. one will try to use a reduced number of well-shaped tetrahedra for speed and stability. However, we may want to use different geometrical models for visualization or contact computation. The second representation is used for collision detection and response, while the third is dedicated to the visual rendering process. This sections presents these representations and their connections.

1.4 Solid Mechanics

Different models can be employed to discretize a deformable solid continuum as a dynamic or quasi-static system of particles (also called simulation nodes). The node coordinates are the independent degrees of freedom (DOFs) of the object, and they are typically governed by equations of the following type:

$$\mathbf{a} = \mathbf{P}\mathbf{M}^{-1} \sum_i \mathbf{f}_i(\mathbf{x}, \mathbf{v}) \tag{1.1}$$

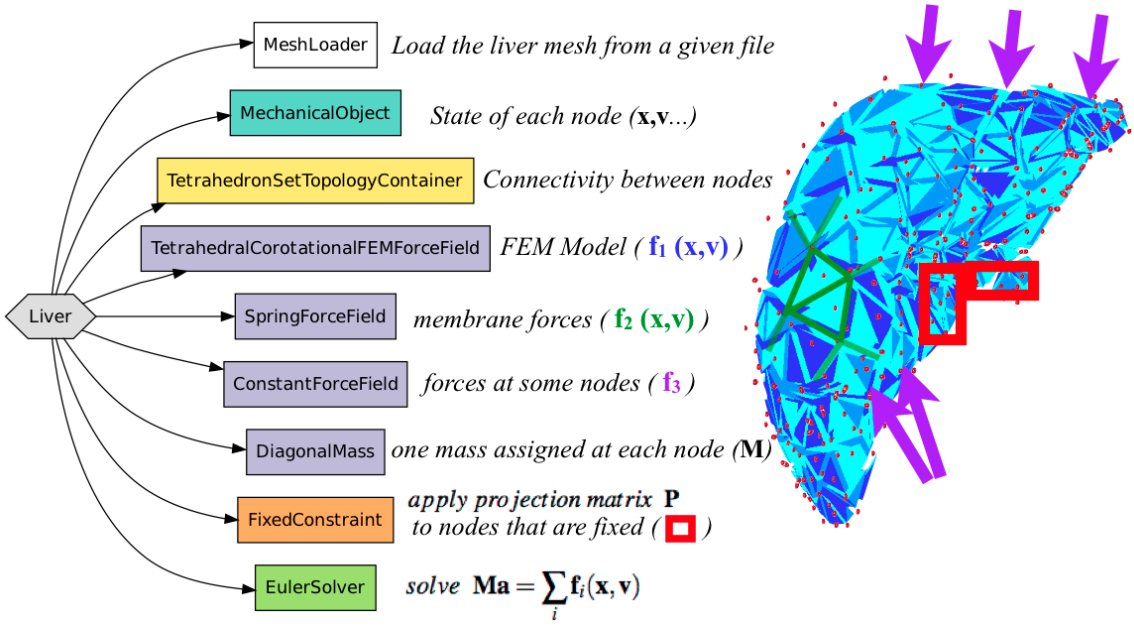


Figure 1.2: Mechanical model of a liver. In order to facilitate the combination of models and algorithms, the liver is described as a composition of specialized components.

where $\mathbf{x}$ and $\mathbf{v}$ are the position and velocity vectors, the $\mathbf{f}_i$ are the different force functions (volume, surface and external forces in this example), $\mathbf{M}$ is the mass matrix and $\mathbf{P}$ is a projection matrix to enforce boundary conditions on displacements. Note that the modeling of rigid body dynamics leads to the same type of equations.

The corresponding model in SOFA is a set of components connected to a common graph node, as shown in the right of Figure 1.2. Each component is responsible for a small number of tasks implemented using virtual functions in an object-oriented approach. Each operator in Equation 1.1 corresponds to a component. *MeshLoader* is used to read the topology and the geometry. The coordinate vector $\mathbf{x}$ of the mesh nodes and all the other state vectors (velocity $\mathbf{v}$, net force $\sum \mathbf{f}$, etc.) are stored in *MechanicalObject*, which is the core component of the mechanical model. A tetrahedral connectivity is stored in *TetrahedronSetTopologyContainer*, and made available to other components such as *TetrahedralCorotationalFEMForceField*, which accumulates one of the terms of the force sum using the Finite Element method. The two other terms come from *SpringForceField*, which accumulates the forces generated by the membrane, and *ConstantForceField*, which accumulates external forces to a given subset of simulation nodes (for instance the pressure exerted by the diaphragm on the liver). *DiagonalMass* is used to implement the product with matrix $\mathbf{M}^{-1}$. *FixedConstraint* implements the product with matrix $\mathbf{P}$ to cancel the displacements of the squared particles. *EulerSolver* implements the logic of time integration.

This approach is highly modular because the components are completely independent of each other and are implemented using C++ classes with a reduced number of abstract functions. For instance, in the example of figure 1.2, if one want to use a FEM for the membrane force instead of the spring based computation, only *SpringForceField* has to be changed for *TriangleFEMForceField*. Similarly, the mass matrix, stored as diagonal matrix in this example, can be stored as a single scalar value (*UniformMass*) if less accuracy but faster computation is sought, in combination with an iterative solver for instance.

For efficiency, each mechanical state vector contains the values of all the simulation nodes, to avoid multiple call of virtual function resolutions. The vector size is basically the number of particles times the number of space dimensions. We use C++ templates to avoid code redundancy between scalar types (float, double), the types of degrees of freedom (particles, frames, generalized coordinates), and the number of space dimensions. All the particles in a vector have the same type known at compile time. Degrees of freedom of different types must be grouped in different objects, possibly connected with interaction forces, as discussed in Section 4.2. This greatly simplifies the design and allows aggressive

compiler optimizations.

More than 30 classes of forces are implemented in SOFA, including springs, FEM for volumetric (tetrahedron or hexahedron) or surface (triangular shell and membrane) deformable objects using corotational or hyperelastic formulations, and for wire or tubular object (beam models meshed with segments), have been implemented. Different types of springs allow for easy and fast modeling of the deformations (bending, compression/traction, volume, interactions between two bodies, joints...). In rigid objects, the main components are the degrees of freedom (a single frame with 3 rotations and 3 translations) and the mass matrix that contains the inertia of the object. Surfaces can be attached to objects using *mappings*, as discussed in Section **??**.

1.5 Collision models

When a lot of primitives comes into contact, collision detection and response can become the bottleneck of a simulation. Several collision detection approaches have been implemented: distances between pairs of geometric primitives (triangles and spheres), points in distance fields, distances between colliding meshes using ray-tracing [12], and intersection volume using images [2] The collision pipeline is described in section **??** with more details.

In order to adapt the models to the data structure of the different collision algorithms, we have defined a *collision model*. This model is similar to a mechanical model, except that its topology and its geometry are of its own and can be stored in a data structure dedicated to collision detection. For instance, the component *TriangleModel* is the interface for the computation of collision detection on a triangular mesh surfaces.

If the collision of a given simulation takes too much time, or to reduce the number of collision points, the meshes used for collision detection can be chosen less detailed than the mechanical ones. In the opposite, if precise collision detection and response is needed with smooth surfaces, it is sometimes suitable to use more detailed mesh for collision detection.

1.6 Visual models

In the context of surgical simulation for training, to reach the state of what is often called *suspension of disbelief* i.e. when the user forgets that he or she is dealing with a simulator, there are other factors than the mechanical behavior. Realistic rendering is one of them. It involves visually recreating the operating field with as much detail as possible, as well as reproducing visual effects such as bleeding, smoke, lens deformation, etc. The main feature of the visual model of SOFA is that the meshes used for the visualization can be disconnected from the models used for the simulation. The mappings described in section **??** maintain the coherency between them. Hence, SOFA simulation results can easily be displayed using models much more detailed than used for internal mechanics, and rendered using external libraries such as OGRE[1] and Open Scene Graph[2].

We have also implemented our own rendering library based on openGL. This library allows for modeling and render the visual effects that occurs during an intervention or the images that the surgeon is watching during the procedure. For instance, in the context of interventional radiology simulator, we have developed a dedicated interactive rendering of X-ray and fluoroscopic images.

[1] www.ogre3d.org
[2] www.openscenegraph.org

Chapter 2

Data Structure

The organization of simulation data is a complex issue. We have identified three relevant levels, and proposed different solutions for each of them. The coarsest structure is the scenegraph, used to hierarchically organize the groups of objects and their multi-models. At a finer scale, the attributes of the components can be linked by relations. Finally, the geometrical models and the topological changes deserve a special attention.

2.1 Scene-Graph

When the simulation involves several objects, we model them as different branches in a scenegraph data structure. In the example shown in Figure 2.1, the scene contains two objects animated using different time integrators, collision detection components (discussed in Section **??**), an interaction force, and a camera to display the objects. The root node does not contain DOFs. It is used to contain the components which are common to its child nodes.

Scenegraphs are popular in Computer Graphics due to their versatility. The data structure is processed using visitors (discussed in Section 2.1) which apply virtual functions to each node they traverse, which in turn apply virtual functions to the components they contain. In basic scenegraph frameworks, the visitors are exclusively fired from an external control structure such as the main loop of the application. In SOFA, the components are allowed to suspend the current traversal to send an arbitrary number of other visitors, then to resume or to prune the suspended visitor. This allows us to implement global algorithms (typically ODE solution or collision detection), such as the explicit Euler velocity update of Equation 1.1, in components. This neatly decouples the physical model from the simulation algorithms, in sharp contrast with dataflow graphs which intricate data and operators in the same graph. Replacing a time integrator requires the replacement of one component in our scenegraph, whereas the corresponding dataflow graph would have to be completely rewritten.

Interactions between objects may be handled using penalty forces or Lagrange multipliers. In all cases, a component connected to the two objects is necessary to geometrically model the contact and compute the interaction forces. This component being shared between the two objects, it is located in their common ancestor node. The coupling created by penalty forces should be seen as soft or stiff, depending on the stiffness and the size of time step [3]. A soft coupling can be modeled by an interaction force constant during each time step. In this case, each object can be animated using its own, possibly different, ODE solver. The assumption of constant interaction force during each time step is compatible with all explicit time integration methods. However, when the interaction forces are stiff, implicit integration is necessary to apply large time steps without instabilities. This requires the solution of an equation system involving the two objects as well as their interaction force together. In this case, the ODE solver is placed in the common ancestor node, at the same level as the interaction component. This is also true for constraint-based interaction which requires the computation of Lagrange multipliers based on interaction Jacobians. Due to the superlinear time complexity of equation solvers, it is generally more efficient to process independent interaction groups using separated solvers rather than a unique solver.

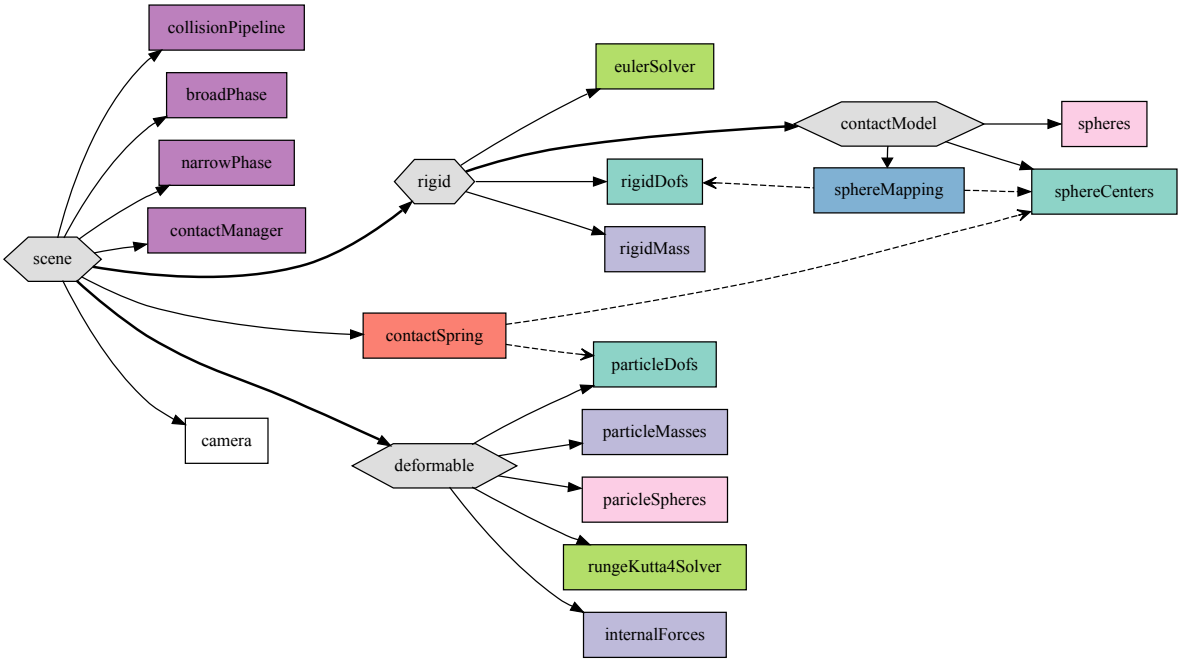


Figure 2.1: A scenegraph with collision detection and two independent objects interacting through a spring.

Visitors

We implement the simulation using visitors which traverse the scene top-down and bottom-up, and call the corresponding virtual functions at each graph node traversal. A possible implementation of the traversal of a tree-like graph is shown in the left of Figure 2.2. Algorithmic operations on the simulated objects are implemented by deriving the Visitor class and overloading its virtual functions *topDown( )* and *bottomUp( )*. This approach hides the scene structure (parent, children) from the components, for more implementation flexibility and a better control of the execution model. The data structure can easily be generalized from strict hierarchies to directed acyclic graphs for more general kinematic dependencies. Moreover, various parallelism strategies can be applied independently of the mechanical computations performed at each node.

Forward time stepping is implemented using the *AnimateVisitor* traversal method, shown in the right of Figure 2.2. Applied to the simple scene in Figure 2.3, it triggers the ODE solver, which in turn applies its algorithm using visitors for mechanical operations such as propagating states through the mappings or accumulating forces. Note that the traversal of the AnimateVisitor is pruned when a ODE solver is encountered. This allows the ODE solver to take control of its subgraph and to overload lower-level solvers, which are not reached by the AnimateVisitor. In the more complex scene shown in Figure 2.1, the solver triggers the collision detection, which may create a contact between the chidren, such as contactSpring. The visitor then triggers the computation of the interaction force, which will be seen by the objects as a constant, external force during the time step. The visitor then continues the traversal and triggers each object ODE solver. The default behavior is to model the contacts prior to applying time integration. To implement other strategies, a *MasterSolver* can be used to prune the visitor and apply time integration and collision detection in a different order, possibly looping until all collisions are solved.

```
void Visitor::traverse(Node n)
bool continue = this.topDown( n )
if continue then
  for all c child of n do
    traverse( c )
  end for
  this.bottomUp( n )
end if
```

```
bool AnimateVisitor::topDown(Node n)
if masterSolver then
  masterSolver.animate(this.dt)
  return false
end if
if collisionPipeline then
  collisionPipeline.modelContacts()
end if
if odeSolver then
  odeSolver.solve(this.dt)
  return false
end if
for all InteractionForce f do
  f.apply()
end for
return true
```

Figure 2.2: Left: a recursive implementation of the visitor traversal. Right: the AnimateVisitor.

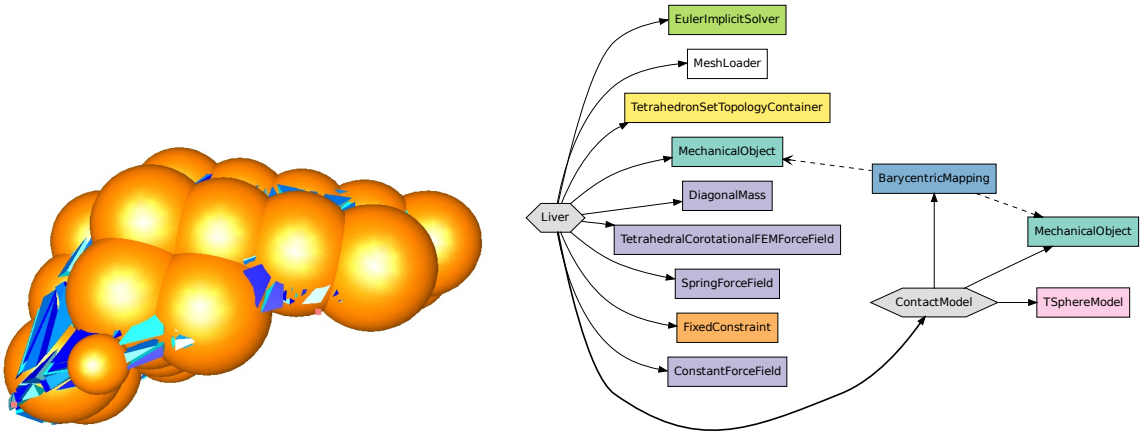


Figure 2.3: Left: simple mechanical (in blue) and collision (in yellow) models of a liver. Right: the corresponding scene graph. The plain arrows denote hierarchy, while the stippled arrows represent connections.

2.2 Data, Engines and Tags

Component parameters are stored in member objects using *Data* containers, templated on the type of attribute they represent. For instance, the list of particle indices constrained by a FixedConstraint is stored in a Data< vector<unsigned> >. These containers provides a reflective API, used for serialization in XML files and the automatic creation of input/output widgets in the user interface, as discussed in Section **??**. We can create connection between Data instances to keep their value synchronized. This is used for instance when a *Loader* component loads several attributes from a file (such as topology, positions, stiffnesses, boundary conditions) which are then connected to one or more components using it as input. In some cases we need to not simply copy an existing value but compute it from one or several Data. This feature is provided by *Engine* components. Engines contain input and output Data, and their update method computes the output based on the input. A mechanism of lazy evaluation is used to recursively flag Data values that are not up-to-date, but they are recomputed only when necessary. For instance, based on a bounding box and a vector of coordinates, a BoxROI engine computes the list of indices of the coordinates inside the box. These indices can then be used as input of a FixedConstraint to define a fixed boundary condition. With this design, the simulation can transparently be setup either from data stored in static files, or generated automatically with engines.

The network of interconnected Data objects defines a data dependency graph, superimposed on the scene graph. This two-graph framework is used in other graphics software such as OpenInventor and Maya, where engines are used to generate the animation, by periodically updating the state vectors using time as input. However, while this approach works well for straightforward computation pipelines, such as keyframe interpolation, it does not easily allow the branching and loops control structures used in sophisticated physical simulation algorithms. It is also a rather low-level representation, essentially encoding every computation steps required to compute a given Data. Consequently, we only use Engines to implement straightforward relations between the parameters of the model, which may remain unchanged during the simulation. In SOFA, the state update algorithms are instead determined by combining several components, communicating through scenegraph visitors, as explained in Section **??**.

Objects and node tagging (Tag and TagSet).

The goal of the introduction of tags is to provide one of the pieces necessary to support non-mechanical states (electrical potentials, constrast agent concentrations) as well as cleaner non-geometrical mechanical states (fluid dynamics, reduced-coordinate articulations). For example, in a simulation involving blood in deformable vessels, we would use two tags to distinguish the different states : mechanical, fluid. These tags will be used to easily work with only a subset of the components, so that the mechanical solver works on positions and forcefields but don't interferes with blood flow and pressure, and inversely for the fluid solver (see [1]). We decided on using there tags instead of extending the class hierarchy as was done before with the State and MechanicalState classes. A hierarchy is fine when we have only one feature that we want to differentiate on (such as base vs mechanical vs electrical), but when we add other criteria (lagrangian geometry vs eulerian vs reduced generalized coordinates, velocity vs vorticity, independent vs mapped DOFs) it is no longer manageable as specialized classes. A secondary use of these tags is to replace existing subsets mechanisms within CollisionModels (r2441) and Constraints (r3121). The design is based on the following elements. Tags are added to BaseObject, as a list of string (internally converted to a list of unique ids for faster processing). All visitors now filter the objects they process based on their list of tags. All solvers by default copy their own list of tags to the visitors they execute, so that they only affect the objects with the same tags as they have (TODO: this is currently broken).

2.3 Topology and Geometry

While mesh geometry describes the location of mesh vertices in space, mesh topology indicates how vertices are connected to each other by edges, triangles or any type of mesh element. Both information are required on a computational mesh to perform mesh visualization, mechanical modeling, collision detection, haptic rendering, scalar or vectorial field description.

[1] http://wiki.sofa-framework.org/tdev/wiki/Notes/ProposalGenericStates

Keeping a modular design implies that mesh related information (such as mechanical or visual properties) is not centralized in a single mesh data structure but is instead spread out in the software components that are using this information. We consider meshes that are cellular complexes made of k-simplices (triangulations, tetrahedralisation) or k-cubes (quad or hexahedron meshes). These meshes are the most commonly used in real-time surgery simulation and can be hierarchically decomposed into k-cells, edges being 1-cells, triangles and quads being 2-cells, tetrahedron and hexahedron being 3-cells. To take advantage of this feature, the different mesh topologies are structured as a family tree (see Fig. 2.4) where children topologies are made of their parent topology. This hierarchy makes the design of simulation components very versatile since a component working on a given mesh topology type will also work on its derived types. For instance a spring-mass mechanical component only requires the knowledge of a list of edges (an *EdgeSetTopology* as described in Fig. 2.4) to be effective. With this design, a spring-mass component can be used at no additional cost on triangulation or hexahedral meshes that derive from an *EdgeSetTopology* mesh.

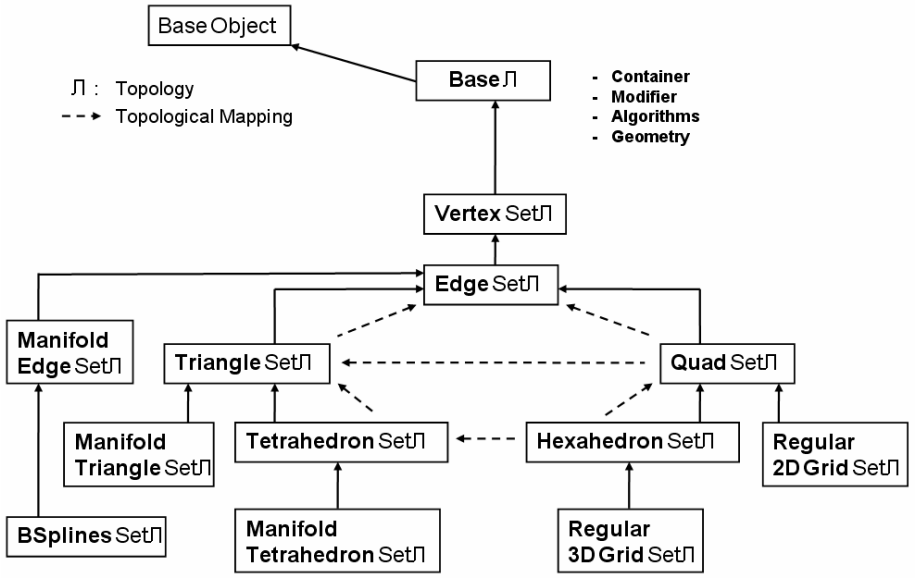


Figure 2.4: Hierarchy of mesh topology. Dashed arrows indicate possible *Topological Mappings* from a topology object to another.

Topology objects are composed of four functional members: *Container*, *Modifier*, *Algorithms* and *Geometry*. The *Container* member creates and updates when needed two complementary arrays. The former describes the l-cells included in a single k-cell, $l < k$, while the latter gives the k-cells adjacent to a single l-cell. From these arrays, generic methods give access to both full topological element description and complete adjacency information. The *Modifier* member provides low-level methods that implement elementary topological changes such as the removal or addition of an element. The *Algorithms* member provides high-level topological modification methods (cutting, refinement) which decompose complex tasks into low-level ones). The *Geometry* member provides geometrical information about the mesh (e.g. length, normal, curvature, ...) and requires the knowledge of the vertex positions stored in the *Degrees of Freedom* component.

Another important concept introduced in SOFA is the notion of *Topological Mapping*. Those mappings define a mesh topology in a from another mesh topology using the same DOFs. For instance, one may

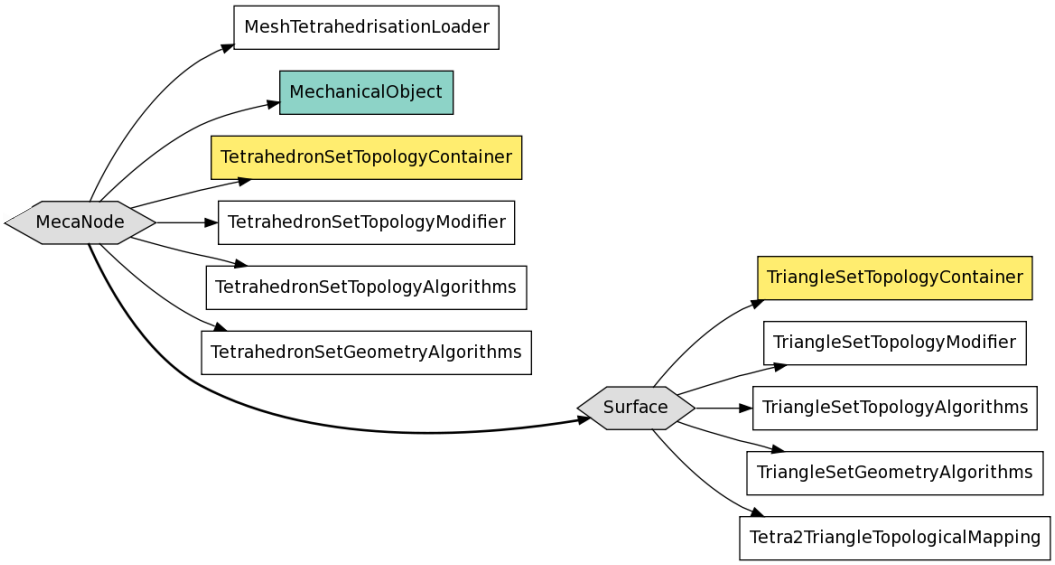


Figure 2.5: Scene Graph of a simple tetrahedral mesh where a topological mapping is defined to have the set of triangles located at the surface of the volumetric mesh.

need to apply specific forces on the surface bounding a volume (for instance to model the Glisson capsule surrounding the liver parenchyma). In this context, a *Tetra2TriangleTopologicalMapping* may be used to generate in a subnode the list of triangles on the border of a tetrahedral surfaces (see Figure 2.5). Similarly, one may obtain the set of edges bordering a triangular mesh or the set of quads at the surface of an hexahedral mesh. Topological mapping may be used to split topological cells into other types of cells. Thus, a quad may be split into 2 triangles and an hexahedron into 5 or 6 tetrahedra. Specific mapping components exist to create an tetrahedral mesh from a set of hexahedra or to create triangular meshes from quads.

Mesh Data Structure

Containers storing mesh information (material stiffness, list of fixed vertices, nodal masses, ...) are stored in *components* and are spread out in the simulation tree. Most containers are simple arrays with contiguous memory storage and a short direct access time. This is important for real-time simulation, but bears some drawbacks when elements of these arrays are being removed since it entails the renumbering of elements. Fortunately, all renumbering tasks that maintain consistent arrays can be automated and hidden to the user when topological changes in the mesh arise. Therefore, efficient access of mesh data structures is granted while the complexity of keeping the container consistent with topological changes is automated.

There are as many containers as topological elements, currently: vertices, edges, triangles, quads, tetras, hexas. These containers are similar to the STL *std::vector* classes and allow one to store any component-related data structure. A typical implementation of spring-mass models would use an edge container that stores for each edge, the spring stiffness and damping value, the i^{th} element of that container being implicitly associated with the i^{th} edge of the topology. Finally, two other types of containers with similar characteristics have been defined. The former stores a data structure for a subset of topological elements (for instance pressure on surface triangles in a tetrahedralisation) while the latter stores only a subset of element indices.

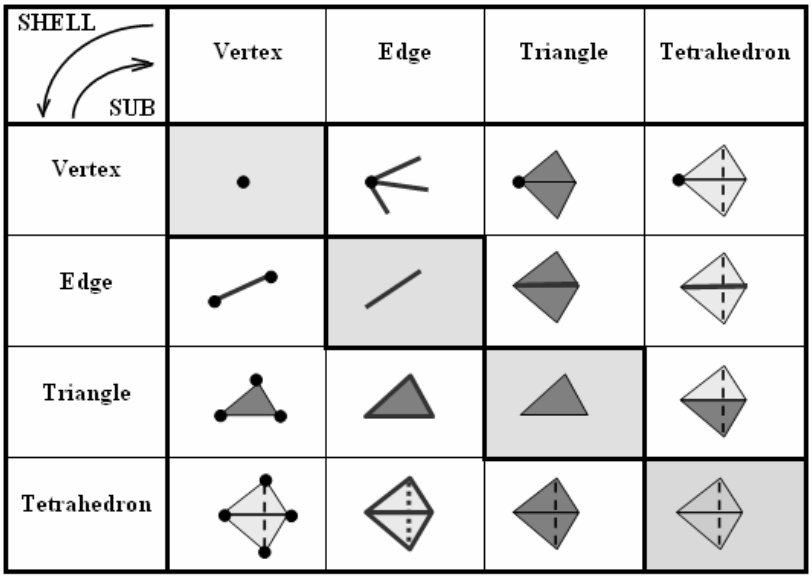


Figure 2.6: The two topological arrays stored in a *Container* correspond to the upper and lower triangular entries of this table. The upper entries provide the k-cells adjacent to a l-cell, $l < k$. The lower entries describe the l-cells included in a k-cell. Similar table exists for quad and hexahedron elements.

Chapter 3

States

3.1 States

The State component is used to store the states vector of a physical object. For a mechanical object, it would be the position, velocity, forces, rest position... Depending on the type of solvers (see chapter 6), additional informations can be stored inside the states, for instance a *mid-step* position if needed in the time integration scheme, or a *free position* that correspond to the position without any constraints if needed during the constraint solving process.

.. to be completed...

3.2 Degrees of Freedom using template

3.3 Data manipulation

Chapter 4

Mechanical forces

to be completed

4.1 ForceField components

4.2 Interaction ForceField

4.3 Mass and inertial forces

Chapter 5

Mappings

5.1 Mapping functions

As seen in section **??**, the mappings propagate positions, velocities, displacements and accelerations top-down, and they propagate forces bottom-up. The top-down propagation methods are:

- `apply (const MechanicalParams*, MultiVecCoordId outPos, ConstMultiVecCoordId inPos )` for positions,
- `applyJ(const MechanicalParams*, MultiVecDerivId outVel, ConstMultiVecDerivId inVel )` for velocities and small displacements,
- `computeAccFromMapping(const MechanicalParams*, MultiVecDerivId outAcc, ConstMultiVecDerivId inVel, ConstMultiVecDerivId inAcc )` for accelerations, taking into account velocity-dependent accelerations in nonlinear mappings.

The bottom-up propagation methods are:

- `applyJT(const MechanicalParams*, MultiVecDerivId inForce, ConstMultiVecDerivId outForce )` for child forces or changes of child forces,
- `applyDJT(const MechanicalParams*, MultiVecDerivId parentForce, ConstMultiVecDerivId childForce )` for changes of parent force due to a change of mapping with constant child force,
- `applyJT(const ConstraintParams*, MultiMatrixDerivId inConst, ConstMultiMatrixDerivId outConst )` for constraint Jacobians,

The name of the methods used to propagate velocities or small displacements top-down contain J, which denotes the kinematic matrix, while the names of the methods used to propagate forces or constraint Jacobians bottom-up contain JT, which denotes the transpose of the same. Method `applyJT(const MechanicalParams*, MultiVecDerivId inForce, ConstMultiVecDerivId outForce )` is used to accumulate forces from a child model to its parent. It performs a cumulative write (+=) since a model may have several children:

$$f_p += J^T f_c \tag{5.1}$$

Some differential equation solvers need compute the change of force df, given a change of position dx. The displacement dx, considered small, is propagated top-down using the linear operator `applyJ(const MechanicalParams*, MultiVecDerivId outVel, ConstMultiVecDerivId inVel )`, then the force changes are accumulated bottom-up. Differentiating eq.5.1, we get:

$$\delta f_p += J^T \delta f_c + \delta J^T f_c \tag{5.2}$$

Once the change of child force δf_c is computed (see the section on force fields), method `applyJT(const MechanicalParams*, MultiVecDerivId inForce, ConstMultiVecDerivId outForce )` is used to accumulate it in the parent, corresponding to the first term in the right of eq.5.2. Method `applyDJT(const`

`MechanicalParams*, MultiVecDerivId parentForce, ConstMultiVecDerivId childForce )` is used to accumulate the second term, which is due to the change of matrix J due to a displacement. It is null in linear mappings, such as BarycentricMapping. This method queries the last displacement propagated and the child force using the MechanicalParams.

Constraints enforced using Lagrange multipliers are represented using linear equations. If a linear constraint on the child DOFS in expressed as $L_c v_c = a$, where L_c is the Jacobian of the constraint, then the equivalent constraint at the parent level is: $L_p v_p = a$, where $L_p = J^T L_c$. Method applyJT(const ConstraintParams*, MultiMatrixDerivId inConst, ConstMultiMatrixDerivId outConst) is used to compute L_p. Since the Jacobians are generally sparse, they are encoded in sparse matrices instead of the dense vectors used for forces.

5.2 Barycentric Mapping

5.3 Rigid Mapping

5.4 Identity Mapping

5.5 Skinning Mapping

Chapter 6

Solvers

6.1 ODE solvers

ODE solvers implement animation algorithms applied at each time step to integrate time and compute positions and velocities one time step forward in time. The solvers do not directly address the physical models. They apply abstract mechanical operations to state vectors represented by IDs, as illustrated in the algorithm shown in Figure 6.1. Each mechanical operation, such as allocating a state vector or accumulating the forces, is implemented using a specialized visitor parameterized on vector IDs or control values such as dt. This allows to implement the solvers completely independenly of the physical model. Each vector used by a solver ID is actually scattered over all the state vector containers in the different nodes in the scope of the solver. Some vector operations such as the dot product apply only to the independent DOFs, stored in the state vectors not attached to a parent by a mapping. Notice that this design avoids the assembly of global state vectors (i.e. copying Vec3 and quaternions to and from vectors of scalars). Moreover, the virtual function calls are resolved at the granularity of the state vectors (i.e. all the particles together, and all the moving frames together) rather than each primitive (i.e. each particle and each frame independently), and allow to optimize each implementation independently. There is thus virtually no loss of efficiency when mixing arbitrary types in the same simulation.

We have identified two families of ODE solvers. The first contains the explicit solvers, which compute the derivative at the beginning of the time step. They are variants of the Euler explicit solver presented in Figure 6.1, and are easily implemented in Sofa using the same operators. The second family contains the implicit solvers, which consider the derivative at the end or somewhere in the middle of the time step. They typically require the solution of equation systems such as:

$$\underbrace{(\alpha\mathbf{M} + \beta\mathbf{B} + \gamma\mathbf{K})}_{\mathbf{A}}\,\delta\mathbf{v} = \mathbf{b} \tag{6.1}$$

where $\mathbf{M}$ is the mass matrix, $\mathbf{K} = \frac{\partial \mathbf{f}}{\partial \mathbf{x}}$ and $\mathbf{B} = \frac{\partial \mathbf{f}}{\partial \mathbf{v}}$ respectively are the *stiffness* and *damping* matrices (the method is explicit if β and γ are null). In order to apply simple displacement constraints, a projection

```
void ExplicitEulerSolver::solve(VecId x, VecId v, double dt)
create auxiliary vectors a,f
resetForce(f)
accumulateForce(f,x,v)
computeAcceleration(a,f)
project(a,a)
v += a * dt
x += v * dt
```

Figure 6.1: Euler's explicit time integration.

```
bool ComputeDfVisitor::topDown():
dof.resetF(this.df)
if mapping then
  mapping.applyJ(this.dx)
end if
return true
```

```
void ComputeDfVisitor::bottomUp():
for all forceField F do
  F.addDF( this.df,this.dx )
end for
mapping.applyJT(this.df)
```

Figure 6.2: Computing *df* given *dx* using a visitor.

matrix $\mathbf{P}$ can be used, and the system becomes $\mathbf{P}^T\mathbf{AP}\delta\mathbf{v} = \mathbf{P}^T\mathbf{b}$ [3]. Implicit integration has the advantage of being more stable for stiff forces or large time steps. However, solving these equation systems requires linear solvers, discussed in the next section. Currently, eight ODE solvers have been implemented, including symplectic Euler and explicit Runge-Kutta4, implicit Euler and statics solution.

6.2 Linear solvers

6.2.1 Conjugate Gradient

An interesting feature of visitor-based mechanical computations is their ability to efficiently and transparently compute matrix products. Thus, we have proposed in SOFA an implementation of the Conjugate Gradient, based on the graph traversal. The visitor shown in Figure 6.2 computes the force change *df* based on a given displacement *dx*, as repeatedly performed in Conjugate Gradient algorithm. An arbitrary number of forces and projections may be present in all the nodes, resulting in a complicated stiffness matrix, as shown in the following equation:

$$\mathbf{df} = \sum_i \left(\prod_{j \in path(i)} \mathbf{J}_j \right)^T \mathbf{K}_i \left(\prod_{j \in path(i)} \mathbf{J}_j \right) \mathbf{dx} \tag{6.2}$$

where $\mathbf{K}_i$ is the stiffness matrix of force i, matrix $\mathbf{J}$ encodes the first-order mapping relation of a node with respect to its parent, and $path(i)$ is the list of nodes from the solver to the node the force applies to. This complex product is computed using only matrix-vector products and with optimal factoring thanks to the recursive implementation. It allows us to efficiently apply implicit time integration to arbitrary scenes using the Conjugate Gradient. This method allows us to trade-off accuracy for speed by limiting the number of steps of the iterative solution.

6.2.2 Direct Solvers

Direct solvers are also available in SOFA. They can be used as preconditionners of the conjugate gradient algorithm [4] or for directly solving equation 6.1. Their implementation are based on external libraries such as Eigen, MKL and Taucs. When dealing with Finite Element Models, the matrices are generally very sparse and efficient implementations based on sparse factorizations allow for fast computations. Moreover, when dealing with specific topologies, like wire-like structures, tri-diagonal band solvers can be used for extremely fast results in $\mathcal{O}(n)$ These different linear solvers address matrices which can be stored in different formats, adapted to the numerical library. The type of matrix is a parameter of the linear solver, and of the visitors the solver uses. Ten linear solvers have been implemented in SOFA. They can be interchanged to compare their efficiency.

6.2.3 From ODE solver to linear solver

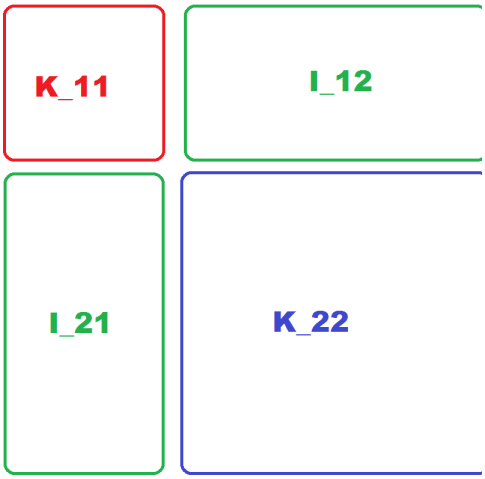
In SOFA, all states of a mechanical object is described by its degree of freedom. The main works for the simulation are filling and inverting a matrix system in order to find the states of mechanical objects by steps of time. This system matrix can be described as below :

$$[\mathbf{MBK}]\,.a = f \text{ , or at time n+1 : } [\mathbf{MBK}]\,.a_{n+1} = f_{n+1}$$

where,

$$\begin{cases} \mathbf{M} : \text{the mass matrix} \\ \mathbf{B} : \text{the damping matrix} \\ \mathbf{K} : \text{the stiffness matrix} \\ [\mathbf{MBK}] : \text{is a linear combination of MBK (not multiplication)} \\ a_{n+1} : \text{accelerator field} \\ f_{n+1} : \text{force field} \end{cases}$$

Usually, **M** is filled by **mass** components, **K** is filled by **forcefield** or **interactionforcefield** components, **B** (often $\alpha\mathbf{M} + \beta\mathbf{K}$) and [**MBK**] are computed by **odesolver** components. The works left to invert the matrix system are done by **linearsolver** components.



On the case for example when there are two mechanical objects, we can see a global stiffness matrix describing the two mechanical states, composed diagonal blocs and non-diagonal blocs. The diagonal blocs are filled by **mass**,**forcefield** components, and the non-diagonal ones are filled by **interactionforcefield** if existed.

Mapping matrix contribution : When existe a mapping on the simulation scene, the states of two mechanical objects are relied by :

$$\begin{array}{rl} \mathbf{x}_2 & = \Im\left(\mathbf{x}_1\right) \text{ , mapping::apply} \\ \mathbf{v}_2 & = [\mathbf{J}]\,\mathbf{v}_1 \text{ , mapping::applyJ} \end{array}$$

The [**J**] matrix is derivative of $\Im$ operator and is defined by the **mapping** components. The dynamic and matrix system of the two objects are relied by :

$$\begin{array}{rl} \mathbf{f} & += \mathbf{f}_1 + [\mathbf{J}]^t\,\mathbf{f}_2 \text{ , mapping::applyJT} \\ [\mathbf{MBK}] & += [\mathbf{MBK}]_1 + [\mathbf{J}]^t\,[\mathbf{MBK}]_2\,[\mathbf{J}] \end{array}$$

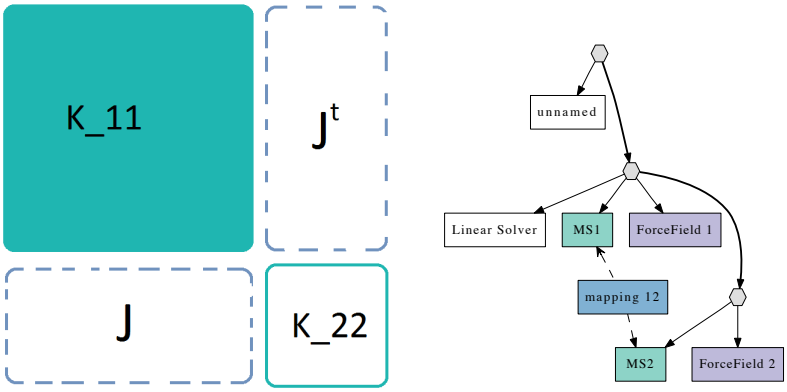
The resolution of the system with the mapping is done in general :

$$\begin{cases} \mathbf{a}^{n+1} & = [\mathbf{MBK}]^{-1}\,\mathbf{f}^{n+1} \\ \mathbf{v}_1^{n+1} & = \mathbf{v}_1^n + dt.\mathbf{a}^{n+1} \\ \mathbf{x}_1^{n+1} & = \mathbf{x}_1^n + dt.\mathbf{v}^{n+1} \\ \mathbf{x}_2^{n+1} & \text{ , mapping::apply} \\ \mathbf{v}_2^{n+1} & \text{ , mapping::applyJ} \end{cases}$$

6.2.4 Particular implementation in SOFA

The direct solver demands to build explicitly the matrix, and invert this matrix after every step of time in order to solve the mechanical response after a solicitation. In SOFA, there are a little more complicated component called mapping, relying geometrical and mechanical properties by master-slave (DOF-mapped object) relation. All changes of geometry or solicitation to one object interfere to other and vice versa. If the mapped object have its own mechanical behavior, it must be counted on the mechanical propagation by the mapping.

self-stiffness propagation



In the simple simulation scene, **MS2** is a mapped object to the **MS1** mechanical object by the mapping. The matrix to be inverted for all mechanical response is the filled colorized one (K_{11}), the matrix K_{22} describing mechanical properties of the second objects must contribute to K_{11} by the formula :

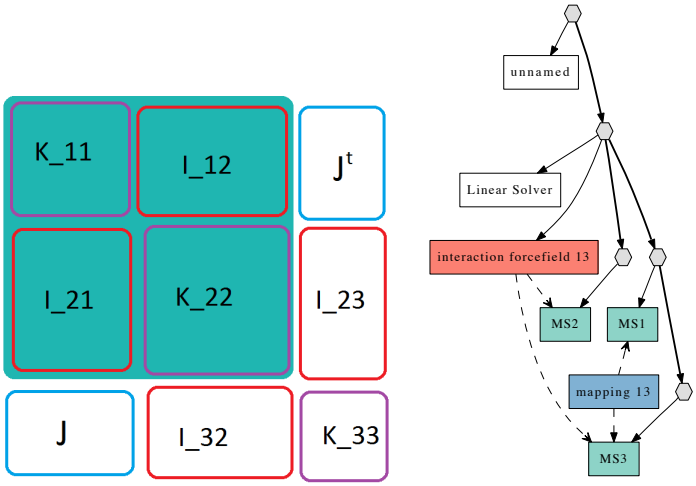
$$\begin{cases} K_{11} & += J^t * K_{22} * J \\ \text{or,} & \\ K_{tempo} & = J^t * K_{22} \\ K_{11} & += K_{tempo} * J \end{cases}$$

By doing this computation, we propagate the stiffness of the mapped mechanical object to its root mechanical object.

interaction-stiffness propagation

In the general case, the may have a simulation scene where there are many level of mapped mechanical states (mapped of mapped state ...) and many interaction forcefield interacting between them. Therefor the stiffness of interaction forcefield and the mapped mechanical state need to be propagated through the mappings. We can imagine for one propagation, there are two simple cases.

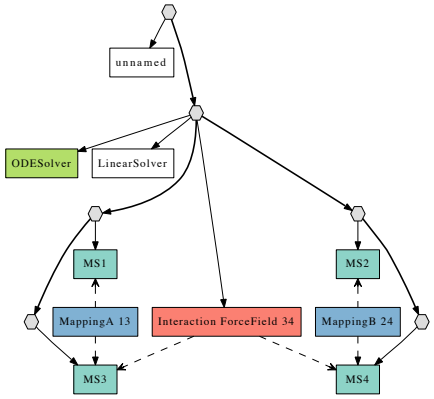
Interaction beweent Real Mechanical Object and Mapped Mechanical Object



In the case where one of the two mechanical states in interaction is non-mapped, the propagation can be computed directly by the formula :

$$\begin{cases} K_{11} & += J^t * K_{33} * J \\ \text{or,} & \\ K_{tempo} & = J^t * K_{33} \\ K_{11} & += K_{tempo} * J \\ \text{and,} & \\ I_{12} & += J^t * I_{32} \\ I_{21} & += I_{23} * J \end{cases}$$

Interaction beweent Mapped Mechanical Object and Mapped Mechanical Object



In the case where the two mechanical states in interaction are mapped, the propagation can be computed by two steps. The first consist to propagate the interaction I_{34} to the interation I_{14} :

$$\begin{cases} K_{11} & += J^t * K_{33} * J \\ \text{or,} & \\ K_{tempo} & = J^t * K_{33} \\ K_{11} & += K_{tempo} * J \\ \text{and,} & \\ I_{14} & += J_A^t * I_{34} \\ I_{41} & += I_{43} * J_A \end{cases}$$

The following step can compute as the one of above paragraph, propagating the interation I_{14} to I_{12}.

6.3 Constraint solvers

To handle different kinds of interactions (contact, friction, joints between particles..) between the simulated objects, SOFA allows the use of Lagrange multipliers [8]. They may be combined with explicit or implicit integration. Each constraint depends on the relative position of the interacting objects, and on optional parameters (such as a friction coefficient, etc.)[1]:

$$\begin{array}{l} \Phi(\mathbf{x}_1, \mathbf{x}_2, ...) = 0 \\ \Psi(\mathbf{x}_1, \mathbf{x}_2, ...) \geq 0 \end{array} \tag{6.3}$$

where Φ represents the bilateral interaction laws (attachments, sliding joints, etc.) whereas Ψ represents unilateral interaction laws (contact, friction, etc.). These functions can be non-linear. The Lagrange multipliers are computed at each simulation step. They add force terms to Equation (6.1):

$$\begin{array}{l} \mathbf{A}_1 \delta \mathbf{v}_1 = \mathbf{b}_1 + \mathbf{H}_1^T \lambda \\ \mathbf{A}_2 \delta \mathbf{v}_2 = \mathbf{b}_2 + \mathbf{H}_2^T \lambda \end{array} \tag{6.4}$$

where

$$\mathbf{H}_1 = [\frac{\delta \Phi}{\delta \mathbf{x}_1} \ ; \ \frac{\delta \Psi}{\delta \mathbf{x}_1}] \qquad \mathbf{H}_2 = [\frac{\delta \Phi}{\delta \mathbf{x}_2} \ ; \ \frac{\delta \Psi}{\delta \mathbf{x}_2}]. \tag{6.5}$$

Matrices $\mathbf{H}_1$ and $\mathbf{H}_2$ are stored in the mechanical state component of each node. Thus, when the constraint applies to a model that is mapped (see section **??**), the constraints are recursively mapped upward like forces to be applied to the independent degrees of freedom [7]. Solving the constraints is done by following these steps:

Step 1, Free Motion: interacting objects are solved independently while setting $\lambda = 0$. We obtain what we call a *free motion* $\delta \mathbf{v}_1^{\mathrm{f}}$ and $\delta \mathbf{v}_2^{\mathrm{f}}$ for each object. After integration, we obtain $\mathbf{x}_1^{\mathrm{f}}$ and $\mathbf{x}_2^{\mathrm{f}}$. During this step, each object solves equation (6.4) with $\lambda = 0$ independently using a dedicated solver.

Step 2, Constraint Solving: The constrained equations can be linearized and linked to the dynamics (see [6] for details).

$$\begin{bmatrix} \Phi(\mathbf{x}_1, \mathbf{x}_2) \\ \Psi(\mathbf{x}_1, \mathbf{x}_2) \end{bmatrix} = \begin{bmatrix} \Phi(\mathbf{x}_1^{\mathrm{f}}, \mathbf{x}_2^{\mathrm{f}}) \\ \Psi(\mathbf{x}_1^{\mathrm{f}}, \mathbf{x}_2^{\mathrm{f}}) \end{bmatrix} + \underbrace{h\mathbf{H}_1 \delta \mathbf{v}_1^{\mathrm{c}} + h\mathbf{H}_2 \delta \mathbf{v}_2^{\mathrm{c}}}_{h\left[\mathbf{H}_1 \mathbf{A}_1^{-1} \mathbf{H}_1^T + \mathbf{H}_2 \mathbf{A}_2^{-1} \mathbf{H}_2^T\right]\lambda} \tag{6.6}$$

With $\delta \mathbf{v}^{\mathrm{c}} = \delta \mathbf{v} - \delta \mathbf{v}^{\mathrm{f}}$. Together with equation (6.3), these equations compose a Mixed Complementarity Problem that can be solved by a variety of solvers. We compute the value of λ using a projected Gauss-Seidel algorithm that iteratively checks and projects the various constraint laws contained in Φ and Ψ [9].

Step 3, Corrective Motion: when the value of λ is available, the corrective motion is computed as follows:

$$\begin{array}{l} \mathbf{x}_1^{t+h} = \mathbf{x}_1^{\mathrm{f}} + h\delta \mathbf{v}_1^{\mathrm{c}} \ \text{ with } \ \delta \mathbf{v}_1^{\mathrm{c}} = \mathbf{A}_1^{-1} \mathbf{H}_1^T \lambda \\ \mathbf{x}_2^{t+h} = \mathbf{x}_2^{\mathrm{f}} + h\delta \mathbf{v}_2^{\mathrm{c}} \ \text{ with } \ \delta \mathbf{v}_2^{\mathrm{c}} = \mathbf{A}_2^{-1} \mathbf{H}_2^T \lambda \end{array} \tag{6.7}$$

A Master Solver, which is generally placed at the top of the graph of SOFA has the role of imposing this new scheduling to the rest of the graph.

Compliance computation : Equations 6.6 and 6.7 involve the inverse of matrix $\mathbf{A}$ (called compliance matrix), which changes at every time step namely in case of a non-linear model. Depending on the simulation case, computing this inverse could be time consuming for real-time simulation. When this is too time-consuming, we propose several strategies to improve the speed of the algorithm such as using the diagonal of $\mathbf{A}$ instead of the hole matrix, or a precomputed inverse [14], or an asynchronous factorization on the GPU [5]. These strategies are implemented in a category of components, called *ConstraintCorrections* that provide different ways of computing $\delta \mathbf{v}^{\mathrm{c}}$ given a value of λ. Given a simulation, it is very easy to make tests and chose the better solution.

[1] For simplicity, we present the equations for two interacting objects (rigid or deformable) 1 and 2, but the solution applies to arbitrary number of interacting bodies.

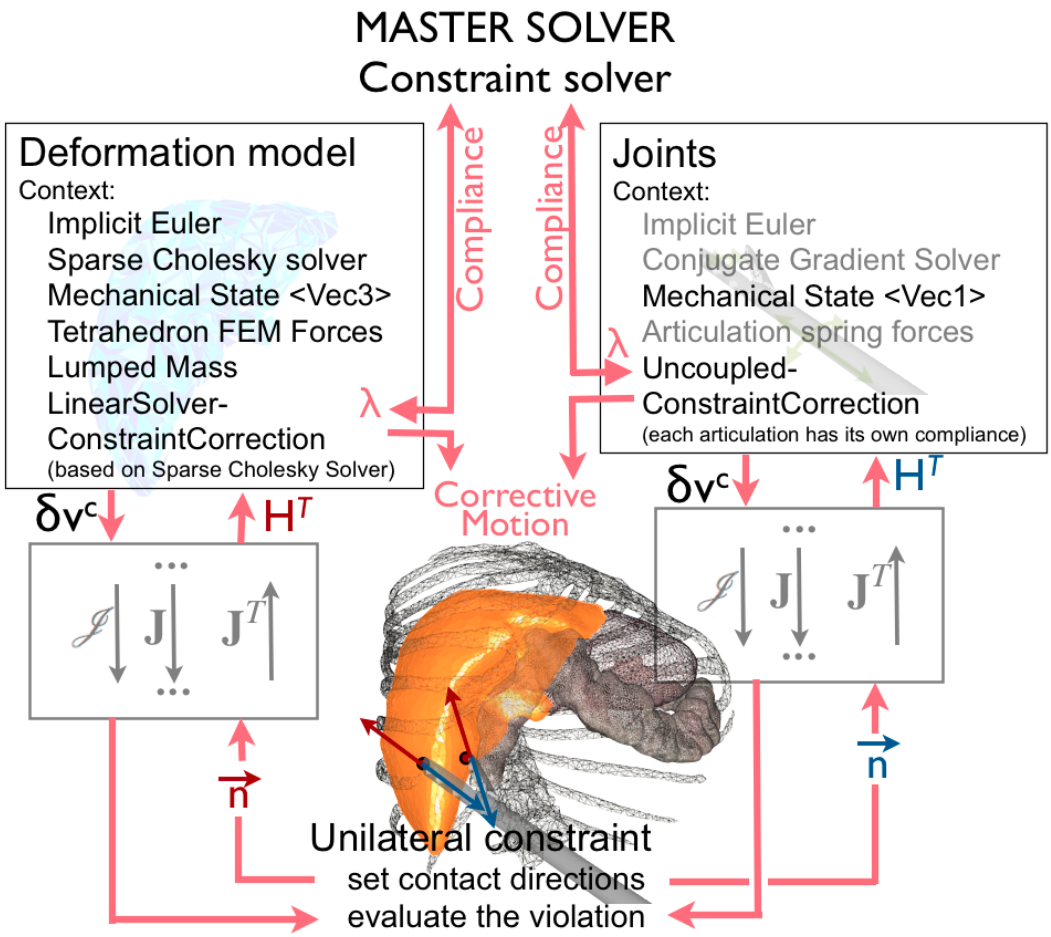


Figure 6.3: Contact process using constraints: A unilateral constraint is placed at the level of the contact points. The constraint direction is mapped to the degrees of freedom of the objects to obtain matrix $\mathbf{H}^T$. The *ConstraintCorrections* components compute the compliance to obtain equation 6.6. The Constraint solver found a new value of λ which is sent to the *ConstraintCorrections* to compute an adequate corrective motion. The Master Solver is placed at the root of the simulation graph to impose the steps of the simulation process.

Chapter 7

Collision detection

Collision detection is split in several phases, each implemented in a different component, and organized using a collision pipeline component. Each potentially colliding object is associated with a collision geometry based on or mapped from the independent DOFs. The broad phase component returns pairs of colliding bounding boxes (currently, axis-aligned bounding boxes). Based on this, the narrow phase component returns pairs of geometric primitives with the corresponding collision points. This is passed to the contact manager, which creates contact interactions of various types based on customizable rules. Repulsion has been implemented based on penalties or on constraints using Lagrange multipliers, and is processed by the solvers together with the other forces at the next time step. This framework has allowed us to efficiently implement popular proximity-based repulsion methods as well as novel approaches based on ray-casting [12] or surface rasterization [11, 2]. Its main limitation is that the contacts can be mechanically processed only after they all have modeled by the collision pipeline. This does not allow to mechanically react to a collision as soon as it is detected, possibly avoiding further collisions between primitives of the same objects.

When stiff contact penalties or contact constraints are created by the contact manager, an additional *GroupManager* component is used to create interaction groups handled by a common solver, as discussed in Section **??**. When contacts disappear, interaction groups can be split to keep them as small as possible. The scenegraph structure thus changes along with the interaction groups.

to be completed

Chapter 8

Visual Rendering

Chapter 9

Haptic Rendering

The main interest of interactive simulation is that the user can modify the course of the computations in real-time. This is essential for surgical simulation : during a training procedure, when a virtual medical instrument comes into contact with some models of a soft-tissue, *instantaneous* deformations must be computed. This visual feedback of the contact can be enhanced by haptic rendering so that the surgeon can really "feel" the contact.

There are two main issues for a platform like SOFA for providing haptic: The first is that haptic forces need to be computed at $1kHz$ whereas real-time visual feedback (without haptic) is obtained at $30Hz$. The second is that haptic feedback can add artificially some energy inside the simulation and can create instabilities, if the control is not *passive*.

Thus two different approaches are currently implemented into SOFA. The first one is the *Virtual Coupling* technique and the other, more advanced, allows for rendering the constraints presented in section 6.3.

9.1 Virtual Coupling

Plugging of a haptic device is bidirectional: the user applies some motions or some forces on the device and this device, in return, applies forces and/or motions to the user. The majority of the haptic devices propose a *Impedance* coupling: the position of the device is provided by the API and this API asks for force values from the application. A very simple scheme of coupling, presented in Fig9.1, could have been used. In this *Direct coupling* case, the simulation would play the role of a controller in an open loop.

Such design is not suitable when stable and robust haptic feedback on a virtual environment is desired. Indeed some combination of the environment impedance and human user reactions can destabilize the system [10]. To avoid this, a virtual mechanical coupling is set. It corresponds to the use of a damped-stiffness between the position measured on the device and the simulated position in the virtual environment (see Fig9.2). If very stiff constraints are being simulated then, the stiffness perceived by the user will not be infinite but will correspond to the stiffness of this virtual coupling. Hence, a compromise between stability and performance must be found by tuning the stiffness value of the coupling.

The damped spring is simulated two times. One time in the haptic loop and one time in the simulation loop. If the two loops are synchronized, then the result is the same. But it can also be used in asynchronous mode: fast update of the haptic loop and low rates in the simulation. In such case, the haptic feedback

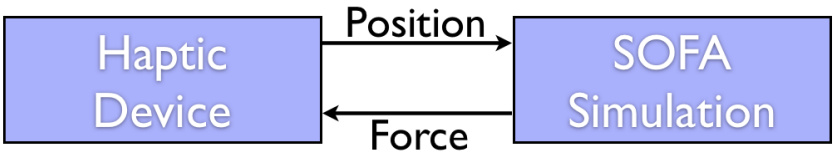


Figure 9.1: Direct coupling

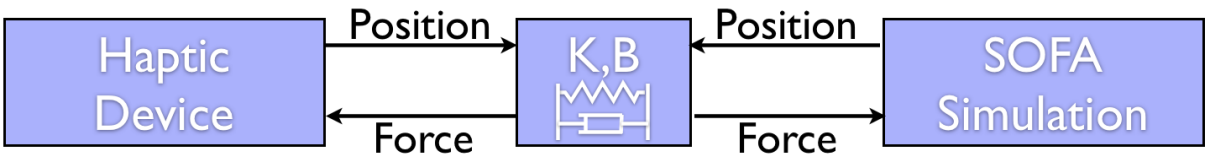


Figure 9.2: Virtual coupling technique. A 6 DoFs Damped spring is placed between the haptic loop and the simulation.

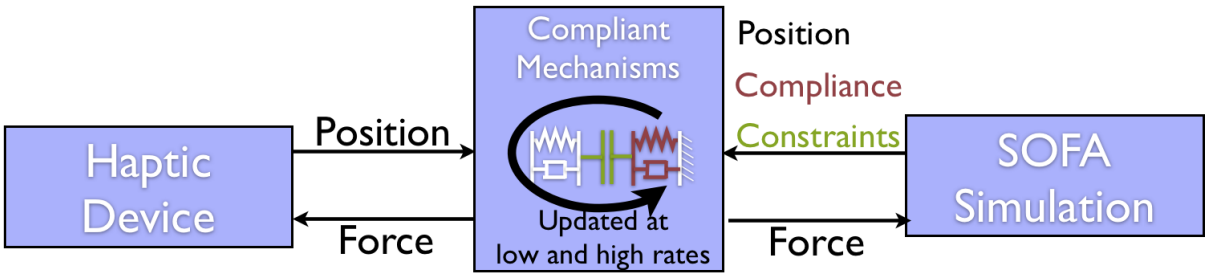


Figure 9.3: Compliant mechanisms technique. The simulation shares the mechanical compliance of the objects and the constraints between them. The constraint response is being computed at low rate within the simulation and at high rates within a separate haptic thread. A 6 DoFs Damped spring is still used to coupled the position of the device to its position in the simulation

remains stable but the delay between the two loop is creating an artificial damping. There is an option to cancel this artificial damping if no contact is detected in the simulation. However, this option can create a sensation of sticking contacts. The main advantage of the virtual coupling technique is that it can be easily employed with every simulation of SOFA. The main drawback is that the haptic rendering is not transparent.

9.2 Constraint-based rendering

An innovative way of dealing with haptic rendering for medical simulation has been proposed in the context of SOFA (see[1] [14] and [13]). The approach deals with the mechanical interactions using appropriate force and/or motion transmission models named *compliant mechanisms* (see Fig9.3). These mechanisms are formulated as a constraint-based problem (like presented in section 6.3) that is solved in two separate threads running at different frequencies. The first thread processes the whole simulation including the soft-tissue deformations, whereas the second one only deals with computer haptics. With this approach, it is possible to describe the specific behavior of various medical devices while relying on a unified method for solving the mechanical interactions between deformable objects and haptic rendering.

9.3 How to use it in SOFA ?

Please see the web page: http://wiki.sofa-framework.org/wiki/Haptic and use the tutorial "dentistry".

[1]The implementation of [14] is available in open-source, for the implementation of [13], please contact: christian.duriez@inria.fr

Part II

Design and Development

Bibliography

[1] Jérémie Allard, Stéphane Cotin, François Faure, Pierre-Jean Bensoussan, François Poyer, Christian Duriez, Hervé Delingette, and Laurent Grisoni. SOFA - an open source framework for medical simulation. In *Medicine Meets Virtual Reality, MMVR 15, February, 2007*, pages 1–6, Long Beach, California, Etats-Unis, 2007.

[2] Jérémie Allard, François Faure, Hadrien Courtecuisse, Florent Falipou, Christian Duriez, and Paul G. Kry. Volume contact constraints at arbitrary resolution. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2010)*, 29(3), August 2010.

[3] David Baraff and Andrew Witkin. Large steps in cloth simulation. In *SIGGRAPH '98*, pages 43–54. ACM Press, 1998.

[4] Hadrien Courtecuisse, Jérémie Allard, Christian Duriez, and Stéphane Cotin. Asynchronous preconditioners for efficient solving of non-linear deformations. In *Proceedings of Virtual Reality Interaction and Physical Simulation (VRIPHYS)*, November 2010.

[5] Hadrien Courtecuisse, Jérémie Allard, Christian Duriez, and Stéphane Cotin. Preconditioner-based contact response and application to cataract surgery. In *MICCAI 2011*. Springer, September 2011.

[6] Hadrien Courtecuisse, Hoeryong Jung, Jérémie Allard, Christian Duriez, Doo Yong Lee, and Stéphane Cotin. Gpu-based real-time soft tissue deformation with cutting and haptic feedback. *Progress in Biophysics and Molecular Biology*, 103(2-3):159–168, December 2010. Special Issue on Soft Tissue Modelling.

[7] Christian Duriez, Hadrien Courtecuisse, Juan-Pablo de la Plata Alcalde, and Pierre-Jean Bensoussan. Contact skinning. In *Eurographics conference (short paper)*, 2008.

[8] Christian Duriez, Frederic Dubois, Claude Andriot, and Abderrahmane Kheddar. Realistic haptic rendering of interacting deformable objects in virtual environments. *IEEE Transactions on Visualization and Computer Graphics*, 12(1):36–47, 2006.

[9] Christian Duriez, Christophe Guébert, Maud Marchal, Stéphane Cotin, and Laurent Grisoni. Interactive simulation of flexible needle insertions based on constraint models. In Guang-Zhong Yang, David Hawkes, Daniel Rueckert, Alison Noble, and Chris Taylor, editors, *Proceedings of MICCAI 2009*, volume 5762, pages 291–299. Springer, 2009.

[10] Richard J. Adams et Blake Hannaford. Stable haptic interaction with virtual environments. *IEEE Transactions on Robotics and Automation*, pages 465–474, 1999.

[11] François Faure, Sébastien Barbier, Jérémie Allard, and Florent Falipou. Image-based collision detection and response between arbitrary volumetric objects. In *ACM Siggraph/Eurographics Symposium on Computer Animation, SCA 2008, July, 2008*, Dublin, Irlande, July 2008.

[12] Everton Hermann, François Faure, and Bruno Raffin. Ray-traced collision detection for deformable bodies. In *3rd International Conference on Computer Graphics Theory and Applications, GRAPP 2008, January, 2008*, Funchal, Madeira, Portugal, January 2008.

[13] Igor Peterlick, Mourad Nouicer, Christian Duriez, Stephane Cotin, and Abderrahmane Kheddar. Constraint-based haptic rendering of multirate compliant mechanisms. *IEEE Transactions on Haptics*, Accepted with minor rev.

[14] Guillaume Saupin, Christian Duriez, and Stephane Cotin. Contact model for haptic medical simulations. In *ISBMS '08: Proceedings of the 4th international symposium on Biomedical Simulation*, pages 157–165, Berlin, Heidelberg, 2008. Springer-Verlag.