Internship

Anita Klein

February 2022 - August 2022

Supervised by Hadrien Courtecuisse and Ziqiu Zeng

Acknowledgments

Before presenting the work I was able to do during my internship, I would first like to thank the people without whom my internship would not have been possible.

I would like to begin by thanking Dr Hadrien Courtecuisse who supervised my internship, for giving me the opportunity to do this latter in his team. Thank you for your support and all your advices and explanations.

A special thank to Ziqiu Zeng for his patience and his availability to answer my questions and concerns and for explaining me the concepts I had a tough time to understand.

I would also like to thank my colleagues with whom I shared the office, for the enjoyable working environment: Long and Claire.

Thank you Long for all the time you took to help me with compilation and all my others issues.

Thank you Claire for always listening to my numerous problems and always looking for a solution.

I would also like to thank (Dr) Paul for taking time to answer my C++ questions and explain me the concepts behind the scope of my question.

Contents

1 Introduction

1.1 Laboratory presentation

ICube [ICu] is a research unit who brings together researchers of the University of Strasbourg, the CNRS (French National Center for Scientific Research), the ENGEES (National School for Water and Environmental Engineering) and the INSA (National Institute of Applied Sciences) of Strasbourg in the fields of engineering science and computer science, with imaging as the unifying theme. Created in 2013, it reunites over 650 members. The laboratory is composed of 4 different departements: the Computer science department, the Solid-state electronics, systems and photonics department, the Imaging, robotics, remote sensing and biological department and the department of mechanics. These departments contain 17 research teams.

Among these teams, one was created in January 2021: the MLMS team (Machine Learning, Modelisation and Simulation) [MLM].
The team is part of the Computer science departement. It is mainly composed of computer scientists, mathematicians, bio-mechanicians, and neuroscientists to develop functional, physical, and geometric models around a transverse axis "Assistance to medical interventions by computer".
The team is interested in data, models and simulations for medical science and human motion.

The main research activities of the team are:

- Developing new bio-inspired models/systems (human, organs, patients)
- Modeling and understanding of certain brain functions
- Optimize our models and numerical simulations so that they run in real time
- Improve or make these models more adapted to each patient and context by exploiting real world data.

My internship took place in the numerical simulations research division.

1.2 Internship subject

One of the topics the MLMS team is working on is numerical simulation.

Numerical simulation can be defined as a representation of complex physical phenomena made possible thanks to a series of calculations and mathematical models.
It represents a major challenge in the health field. It can give assistance to medical interventions. But it can also give the possibility to teach and train clinical skills without risk to actual human patients. Particularly in surgery, the practice of a surgical procedure is essential to reduce the post-operative damages.

Simulation also allows not to perform an intervention for the first time on a patient. In addition, it avoids using cadavers or animals that do not have the exact same biomechanical properties as humans and using these latter raises some ethical issues.

In medical simulation, we distinguish simulation not being computed in real time from those who are. Real time simulation means if the simulation takes a certain time to compute, the same time would have been taken in the real world.

Despite the fact medical simulation has become a powerful tool in learning and medical interventions assistance, simulation of cutting is still a challenge, not only by simulating the topological cut but also by handling the deformations induced by these modifications.

The MLMS team is working on a new method of cutting simulation that leads to a minimal amount of topological changes. However, this method could be optimized in order to gain some computation time (in order to have a real time simulation).

The new method was implemented in a Sofa (Simulation Open Framework Architecture) plugin. However, Sofa had some updates since the plugin implementation and some issues were remaining.

Thus, during my internship, I worked on updating the cutting plugin and on the computation time problematic.

1.3 Plan

In order to be able to modify the code of the cutting plugin and to improve it, I first had to understand its implementation. It included a comprehension of the Sofa software and reading some papers about cutting simulation (the different methods available, the strengths and weaknesses of each method and the remaining issues).
Once this part of the work was done, we were able to focus on the modification of the code and the optimisation of its computation time.

In this report, we are first going to present the state of the art on the cutting simulation methods. Then, we will present some of the techniques we used in our cutting simulation method in order to gain computation time. Next, we will introduce the new cutting method. And finally present the ideas we had to improve the computation time.

2 State of the art

Before working on an algorithm that simulates a cut, we will first look at the different methods that already exist.
Current methods of virtual cutting are divided into three sections: [WM17]

- mesh-based method
- meshless method
- hybrid method.

2.1 Mesh-based method

Mesh-based methods, as indicated in their name, take meshes as a basis for their simulation.

The deformations and the topological changes in the mesh due to the cutting process are simulated using Finite Element Method.

We will describe some ways to simulate a cut with mesh-based methods.
The easiest way to simulate a cut is by deleting elements that cross the cutting path. But this leads to a major issue: volume loss in the mesh (Image A in Figure 1).
Another easy way to simulate a cut is by splitting the mesh along the existing grid boundary (as done in image B in Figure 1).
These two methods are easy to implement but the cutting surface obtained is jagged and the visual effect is unsatisfactory.

Node snapping is another alternative for cutting simulation. It consists of moving points to the cutting surface to fit the cut. It is the process applied on the mesh in image C (Figure 1).

Mesh refinement is another method (image D in Figure 1). It allows to fit the cutting path and avoids the issue encounter in image A (volume loss). But this method can lead to ill-shaped elements that lead to instabilities during the simulation and high computation time due to the number of added points. Besides, this method needs extra memory space for all the new nodes.

Among the mesh-based method, we can cite the element duplication. The elements across the cutting path are duplicated before being separated. It avoids the risk of creating ill-shaped elements, but there is a possibility to modify the volume of the mesh (this procedure is used in the mesh E in Figure 1).

An additional method consists of coupling node snapping and mesh refinement. It gives good result and the number of added points can be easily controlled (it was one of the issue of the remeshing technique). This process is illustrated in the mesh F of Figure 1.

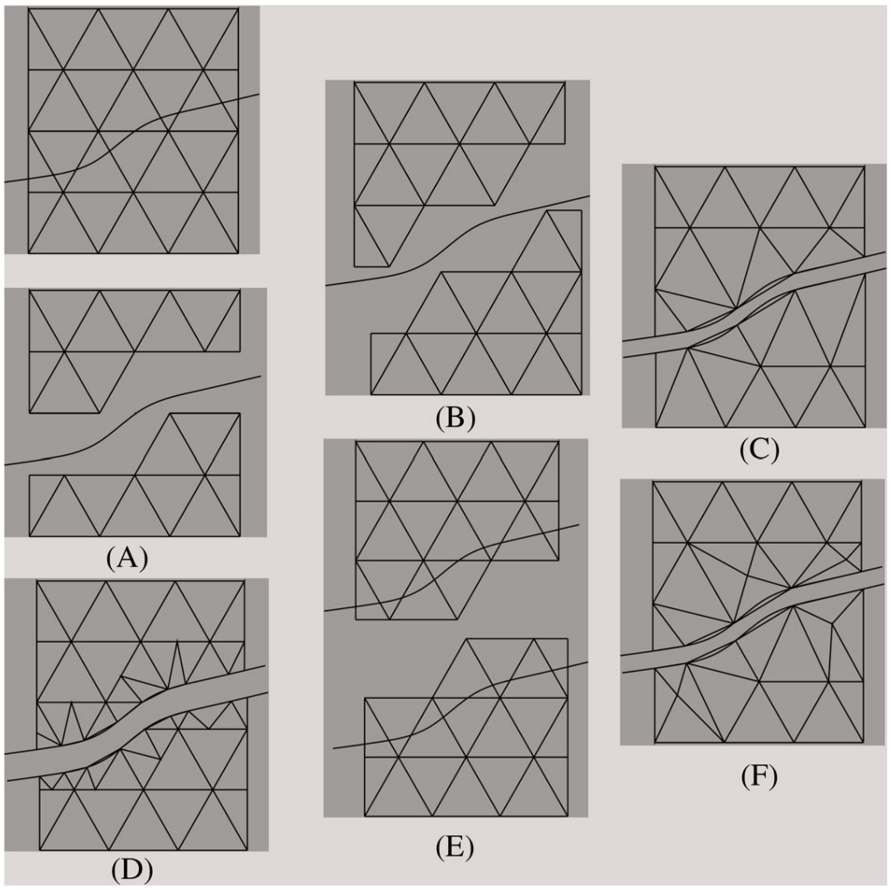


Figure 1: Different cutting simulation techniques. Image from [WM17]

2.2 Meshless method

Meshless methods use particles and their interactions with their neighbors. The cutting process takes place when the forces increase above a threshold. These methods have the advantage of avoiding the instability problem of ill-shaped elements caused by remeshing and model the discontinuity efficiently. But, they require higher computational complexity to solve the equations.

Position Based Dynamics

Position Based Dynamics [IBM17] is an example of meshless methods. The simulation of an object is done thanks to a set of points and constraints. It is a force-based model as forces are applied to the points to move them. The constraints assure the points will not move in a way that violates the simulation.

This meshless method has the specificity of directly manipulating the positions, rather than obtaining it by indirect information (such as the velocity and acceleration of the object). It makes the collision constraints easier.

Also, this method has the advantages of being computationally and memory-wise (there is no need to store a mesh) efficient.

However, if we compare with other methods such as FEM based on force, we find FEM methods are more accurate. PBD methods have lower precision but can offer better visual rendering. It is the reason PBD is mainly used for virtual reality, computer games and special effects in movies and commercials, and not for medical simulation.

2.3 Hybrid method

Hybrid methods are a combinaison of mesh-based and meshless methods. They try to combine the advantages of both methods by using a mesh to handle collision detection and particles to compute deformations.

As seen in the state of the art, the existing cutting simulation methods have weaknesses such as:

- volume changes
- need of extra memory space in the middle of the simulation
- creation of ill-shaped elements
- unsatisfactory visual effect.

All these weaknesses justify the fact we work on new cutting simulation methods.

3 Background

3.1 Some prerequisite on FEM

We are working on simulating a cut in a medical simulation context. This problem involves taking into account some physical phenomenon. Most physical phenomenon are described using partial differential equations (PDEs). No exact solution is known for most PDEs. It is the reason some numerical methods have been developed in order to provide approximate results. One of the most used numerical method to solve PDEs is the finite element method (FEM). We are going to use this method in our simulations.

Since solving PDEs over our random continuous domain Ω is not possible, we divide our domain into sub-domains: the finite elements. We can decide the shape of our subdomain elements from many different elements such as triangles, tetrahedron, hexahedron etc .
The FEM method consists of approximating the integration of a function over a domain by the sum of the integrals over each finite element of the domain.
The division of the domain transforms our continuous problem (over Ω) into a finite one.

Let note E the number of the finite elements, V_e the volume of every element e and f the function we want to integrate. We obtain thanks to the finite element method:

$$\int_{\Omega} f(x)d\Omega = \sum_{e=0}^{E} \int_{V_e} f(x)dV_e \tag{1}$$

The FEM method benefits from the simple shape of the finite elements to compute the integrals. For every point P(x) of an element, we can get the measure of any field such as the displacement field u as a linear combination of interpolation functions ϕ and the values of this field $u(x_i) = u_i$ at each vertex i of the element (N being the total number of vertices in one element).

$$u(x) = \sum_{i=0}^{N} u_i \phi_i \tag{2}$$

The interpolation functions (also known as the shape functions) guarantee the continuity of any field over the element. Thus, shape functions are specific to every element type.

We can take local coordinates (ξ, η, ζ) as a reference configuration of the element and express the shape functions ϕ according to these local coordinates. Every element can always be mapped back to one common reference configuration with local coordinates (ξ, η, ζ), as shown in the figure below, even if the elements get distorted during the simulation. In other words, for any point in space $x = (x, y, z)$ inside the element K a corresponding point $\xi = (\xi, \eta, \zeta)$ can be found in the reference space of the element $\hat{K}$.

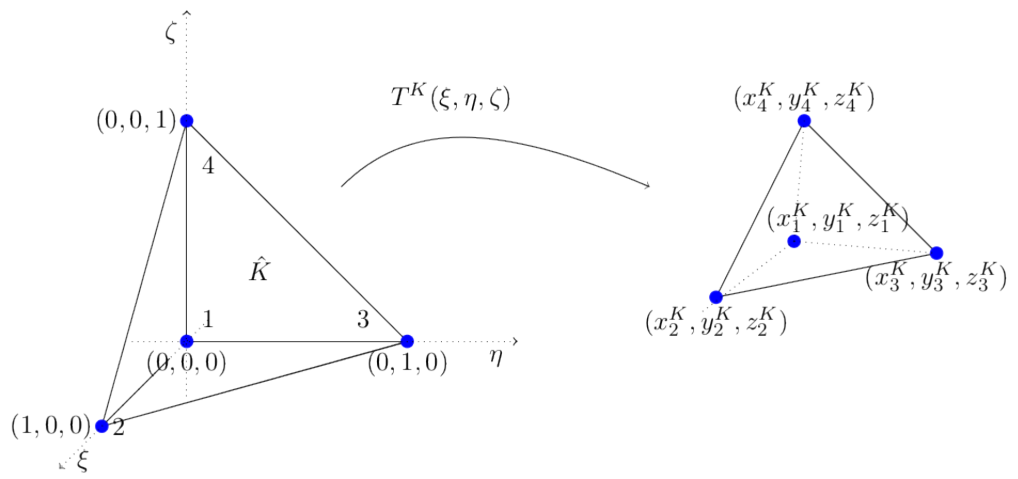


Figure 2: Transformation T^K: mapping from the reference cell $\hat{K}$ to a typical cell K [Sof]

Therefore, it always exists a transformation T^K which can be defined as:

$$T^K : \hat{K} \to K$$

$$\xi = (\xi, \eta, \zeta) \to x = T^K(\xi) = \sum_{i=0}^{N} x_i \phi_i(\xi).$$

We use the FEM method to solve physical models. In our situation, we are going to work with deformable oject. Here are some of the consitutive laws we could use with their main advantages and drawbacks.

Constitutuve law	Advantage	Drawback
Linear model	Simple and fast	Cannot be used for large displacement.
Co-rotational model	Compensate geometrical non linearity	Can only undergo small deformations
Hyperelastic model	Suitable for large deformation	Difficult to solve in real time

We decide to use the co-rotational model as it is a good compromise between being efficient and precise.

3.2 Physical behavior of the mesh

We are working on a topology that is going to undergo a cut. This topology corresponds to a deformable object. Before performing any cut, we first need to describe the physical behavior of the object.

In order to solve some of the equations presented here, we use the FEM method.

In our situation, we use a constraint-based approach (explained in the next section 3.3) on our deformable object to solve the coupled contacts between multiple objects.

Newton's second law is the general way to describe the physical behavior of a deformable object [ZCC22]. Here is Newton's second law equation:

$$M\ddot{q} = p - \mathcal{F}(q, \dot{q}) + c \tag{3}$$

where M is the mass matrix, $\ddot{q}$ the vector of the derivative of the velocity, p the external forces, and $\mathcal{F}(q, \dot{q})$ the function representing the internal forces. c represents the contact and friction forces contribution.

A backward Euler method is used to integrate the time step. Let note h the length of the time interval between two steps. Backward Euler on $[t, t+h]$ gives:

$$q_{t+h} = q_t + h\dot{q}_{t+h} \tag{4}$$

$$\dot{q}_{t+h} = \dot{q}_t + h\ddot{q}_{t+h}. \tag{5}$$

As $\mathcal{F}(q, \dot{q})$ is a non-linear function, we perform a first-order Taylor expansion in order to linearize the problem.
Let $f = f(x, y)$ be a function of two variables. If (x, y) is near (a, b), Taylor's formula goes as follow:

$$f(x, y) \approx f(a, b) + \frac{\partial f}{\partial x}(a, b)(x - a) + \frac{\partial f}{\partial y}(a, b)(y - a).$$

We use this formula on $\mathcal{F}(q_{t+h}, \dot{q}_{t+h})$. (x, y) is replaced by $(q_{t+h}, \dot{q}_{t+h})$. We want (a, b) to be near (x, y). From equation 4, we decide to replace a by q_t and from equation 5, we decide to replace b by $\dot{q}_t$. We obtain:

$$\mathcal{F}(q_{t+h}, \dot{q}_{t+h}) = \mathbf{f}_t + \frac{\partial \mathcal{F}(q_t, \dot{q}_t)}{\partial q_{t+h}} h\dot{q}_{t+h} + \frac{\partial \mathcal{F}(q_t, \dot{q}_t)}{\partial \dot{q}_{t+h}} h\ddot{q}_{t+h} \tag{6}$$

where $\mathbf{f}_t = \mathcal{F}(q_t, \dot{q}_t)$.

The force function is considered constant during a time integration, and the partial derivative terms could be expressed as matrices:

$$\frac{\partial \mathcal{F}(q_t, \dot{q}_t)}{\partial q_{t+h}}$$

is the stiffness matrix K and

$$\frac{\partial \mathcal{F}(q_t, \dot{q}_t)}{\partial \dot{q}_{t+h}}$$

is the damping matrix B.

If we regroup equations 3, 4, 5 and 6, we obtain:

$$\begin{aligned}
M\ddot{q}_{t+h} &= p_{t+h} - \mathcal{F}(q_{t+h}, \dot{q}_{t+h}) + c \\
&= p_{t+h} - \mathbf{f}_t - \frac{\partial \mathcal{F}(q_t, \dot{q}_t)}{\partial q_{t+h}} h\dot{q}_{t+h} - \frac{\partial \mathcal{F}(q_t, \dot{q}_t)}{\partial \dot{q}_{t+h}} h\ddot{q}_{t+h} + c \\
&= p_{t+h} - \mathbf{f}_t - Kh\dot{q}_{t+h} - Bh\ddot{q}_{t+h} + c \\
&= p_{t+h} - \mathbf{f}_t - hK(\dot{q}_t + h\ddot{q}_{t+h}) - hB\ddot{q}_{t+h} + c.
\end{aligned}$$

By rearranging the equation and multiplying it by h, we get:

$$\begin{aligned} M\ddot{q}_{t+h} + hB\ddot{q}_{t+h} + h^2K\ddot{q}_{t+h} &= p_{t+h} - \mathbf{f}_t - hK\dot{q}_t + c \\ (M + hB + h^2K)\, h\ddot{q}_{t+h} &= hp_{t+h} - h\mathbf{f}_t - h^2K\dot{q}_t + hc. \end{aligned}$$

The last equation can be rewritten as follow:

$$Ax = b + hc \tag{7}$$

where

$$A = M + hB + h^2K,$$
$$x = hq_{\ddot{t+h}}$$

and

$$b = hp_{t+h} - h\mathbf{f}_t - h^2K\dot{q}_t.$$

We just obtained a dynamic equation describing the physical behavior of our object.

Simulating a cut on a deformable object requires to compute and solve dynamic equations. Doing so is very costly. Because the deformation of the object is relatively small, the MLMS team had the idea of simulating the cut on a topology using a static model and mapping the result on the initial deformable toplogy.
For this purpose, we need to store the correspondence between a point's actual position and its initial position. We say the object is in its rest shape when no deformation occured. The positions of the rest shape are called the rest positions.
These positions are going to be essential in the implementation of our cutting method.

3.3 Constraint-based technique

The constraint-based methods using Lagrange multipliers ([Jea99], [Ren13]) solve the contact problem in a coupled way, addressing the limitation in the penalty methods. Such methods provide accurate and robust solutions in contact mechanics for large time steps, where interpenetration is entirely eliminated at the end of time steps.

Generally, a Karush-Kuhn-Tucker (KKT) system is assembled to solve the constrained problem:

$$\begin{cases} A_1 x_1 - h H_1^T \lambda = b_1 & \text{(8a)} \\ A_2 x_2 + h H_2^T \lambda = b_2 & \text{(8b)} \\ h H_1 x_1 - h H_2 x_2 + \delta^t = \delta^{t+h} & \text{(8c)} \end{cases}$$

By eliminating the unknowns x_1 and x_2 in Equation (8c), we have:

$$\begin{aligned} \delta^{t+h} &= \delta^t + h[H_1 \underbrace{A_1^{-1} b_1}_{x_1^{\text{free}}} - H_2 \underbrace{A_2^{-1} b_2}_{x_2^{\text{free}}}] \\ &+ h^2 \underbrace{[H_1 A_1^{-1} H_1^T + H_2 A_2^{-1} H_2^T]}_{W} \lambda \end{aligned} \tag{9}$$

which can be simplified as follows:

$$\delta^{t+h} = \delta^{\text{free}} + h^2 W \lambda \tag{10}$$

The unknown λ is solved by a projected Gauss-Seidel algorithm [DDKA06] during the successive iterations.

Once the λ is solved, a *corrective motion* is processed to integrate the final motion x_{t+h}:

$$\begin{aligned} x_1^{t+h} &= x_1^{\text{free}} + h A_1^{-1} H_1^T \lambda \\ x_2^{t+h} &= x_2^{\text{free}} - h A_2^{-1} H_2^T \lambda \end{aligned} \tag{11}$$

3.4 Implementation with Sofa

In order to implement our cutting method, we use the Sofa software.

Figure 3: Sofa logo

SOFA stands for Simulation Open Framework Architecture. Sofa is an open-source library written in C++ distributed under LGPL license, hosted on GitHub. [Sof]
It is primarily targeted at real-time physical simulation, with an emphasis on medical simulation and is mostly intended for the research community to help develop newer algorithms, but can also be used as an efficient prototyping tool or as a physics engine.
Various fields are being investigated such as medical simulation, robotics and control, animation and biology.

Sofa implements mathematical models and algorithms describing the physics around us:

- Soft and rigid body dynamics
- Heat transfer
- Fluid mechanics.

The Sofa framework is very flexible and modular. The user can decide to activate specific plugins, and activate/desactivate features. Some plugins are provided by the Sofa Consortium but every user can create his own and include them into his Sofa simulations. In addition to being flexible, Sofa is also efficient. Multi-threading and GPU computing can be used.

Let's describe some of Sofa's main principles.

3.4.1 Main principles of Sofa

Scene graph

In Sofa, every simulation is described as a graph and more particularly as a directed acyclic graph. There are nodes that contain Components who themselves contain parameters called Data.

In every scene, the parent node is called the "Root" node. All other nodes inherit from the latter, and each node gathers the components associated with the same object.

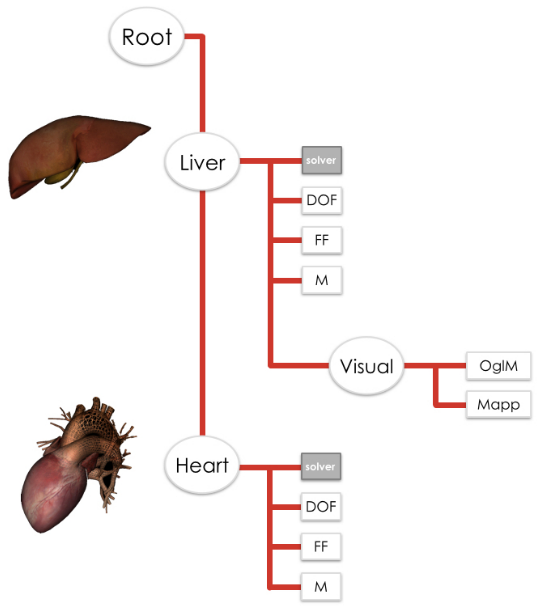


Figure 4: A graph with two different objects

In the scene graph presented in Figure 4, the first subnode can compute the mechanical behavior of a liver whereas the second simulates the electrical behavior of a heart. For each subnode, a different physical phenomenon is taken into account and this is the reason there are separated.

AnimationLoop and visitor

AnimationLoop is a mandatory parameter in every Sofa scene as it rules and orders all simulation steps. If no animation loop is defined, a default one will be automatically added.
The AnimationLoop component activates all the steps of the simulation through a Visitor mechanism. Visitors are part of an implicit mechanism. They allow to recover information from all graph nodes. They traverse the scene and call all the corresponding virtual functions at each graph node traversal. A few AnimationLoops are already available in Sofa, but we will present only two of them:

1. DefaultAnimationLoop
2. FreeMotionAnimationLoop.

DefaultAnimationLoop

This AnimationLoop works as follow:

- detects collisions
- solves the physics and updates the system.

FreeMotionAnimationLoop

This AnimationLoop has more steps:

- computes free physics (the constraints are not yet taken into account)
- detects collisions (it generates constraints)
- solves constraints: physics + constraints/collision
- corrects the motion.

Mapping

Another essential principal of Sofa is the Mapping. In Sofa, every object can be represented by several models, separately.

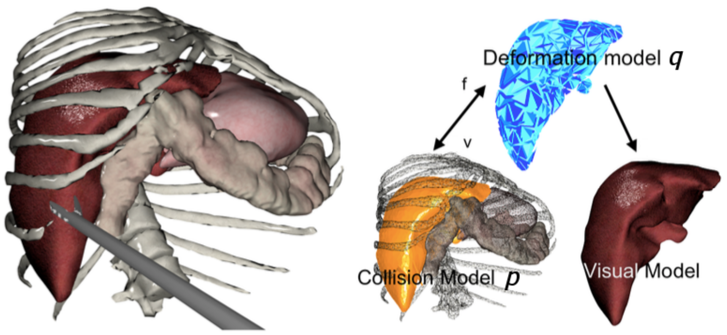


Figure 5: Different representations of a liver

As we can see in Figure 5, every model can rely on a different representation/topology. It is the reason a mapping is needed in order to have the correspondence between models.

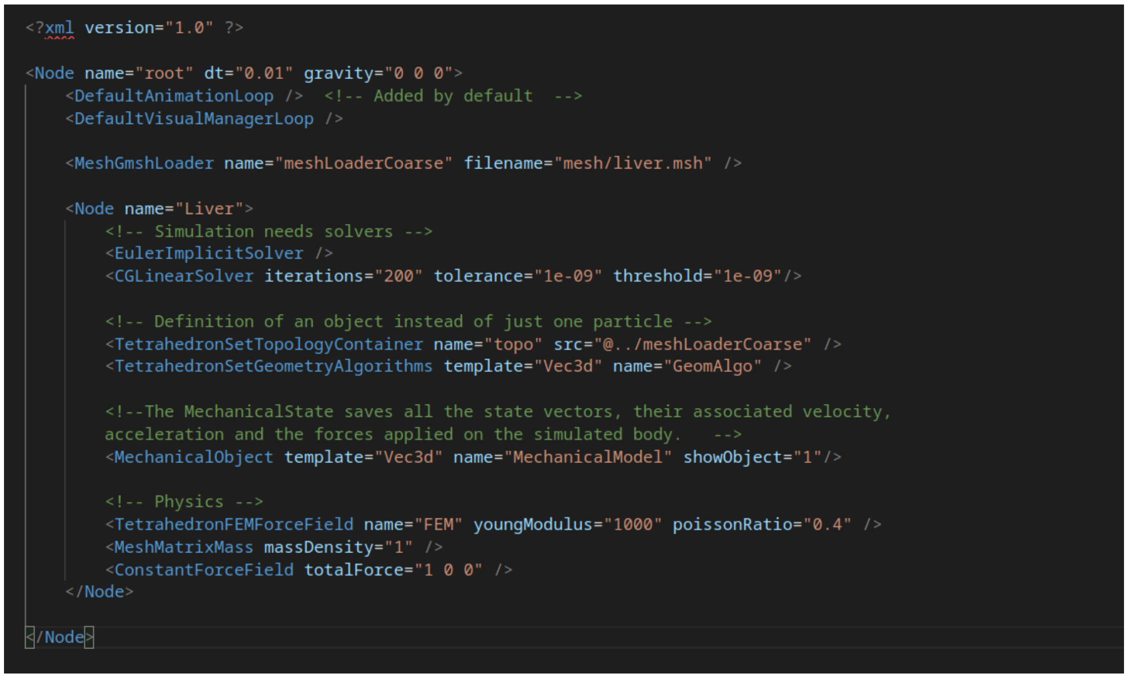
Collision

Collision detection is essential in any simulation and especially in a cutting simulation context. It detects when two objects run into one another and can then call the corresponding functions. Collision detection is done before calling any solver. The detection is divided in two steps: the broad phase and the narrow phase.

The broad phase detects if two objects are colliding by checking if their bounding boxes (rectangle that surrounds the object) intersect. This method is quick and allows to rapidly know when there are no collision.
If the broad phase detects a collision, the narrow phase is then used. It computes the intersections.

3.4.2 Launch a simulation

Now that we have seen the basics of Sofa, let's see how to launch a simulation.
There are two ways to generate a simulation with Sofa: through XML files or through Pyhton scripts. During my internship, I used XML files only.



```
<?xml version="1.0" ?>

<Node name="root" dt="0.01" gravity="0 0 0">
    <DefaultAnimationLoop />  <!-- Added by default  -->
    <DefaultVisualManagerLoop />

    <MeshGmshLoader name="meshLoaderCoarse" filename="mesh/liver.msh" />

    <Node name="Liver">
        <!-- Simulation needs solvers -->
        <EulerImplicitSolver />
        <CGLinearSolver iterations="200" tolerance="1e-09" threshold="1e-09"/>

        <!-- Definition of an object instead of just one particle -->
        <TetrahedronSetTopologyContainer name="topo" src="@../meshLoaderCoarse" />
        <TetrahedronSetGeometryAlgorithms template="Vec3d" name="GeomAlgo" />

        <!--The MechanicalState saves all the state vectors, their associated velocity,
        acceleration and the forces applied on the simulated body.   -->
        <MechanicalObject template="Vec3d" name="MechanicalModel" showObject="1"/>

        <!-- Physics -->
        <TetrahedronFEMForceField name="FEM" youngModulus="1000" poissonRatio="0.4" />
        <MeshMatrixMass massDensity="1" />
        <ConstantForceField totalForce="1 0 0" />
    </Node>

</Node>
```

Figure 6: Example of a XML file to launch a simulation

In our simulation (code in Figure 6), we began by declaring the root node (the parent node). Then, we called the DefaultAnimationLoop (explained above) and the DefaultVisualManagerLoop for visual rendering. Once the simulation is launched, the visual options can be modified directly in Sofa.
Then, we decided to use Gmsh to generate the mesh. We could have used another mesh generator such as VTK for example.
Then, we declared a new node, called Liver. We decided to use an implicit solver (Euler) and the conjugate gradient method.
We also had to define the topology we wanted (in our case, we decided to divide the mesh into tetrahedron).
Once our topology was declared, we defined a MechanicalObject. The MechanicalObject saves all the state vectors, their associated velocity, acceleration and the forces applied on the simulated body.
And we finally added the physics. The force fields component allow to add forces that influence the equilibrium of the system by contributing to its change of state.

This is what we obain:

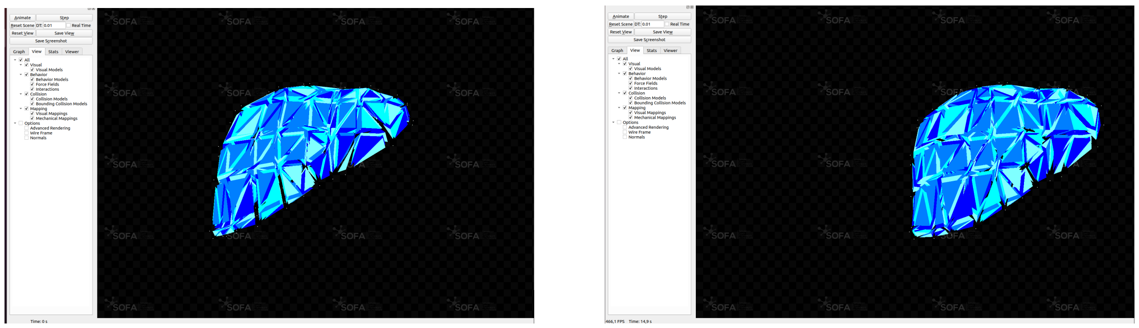


Figure 7: Simulation at 0 sec and at 14.9 sec

As we can see in Figure 7, the liver moved to the right. It is due to the fact we used ConstantForceField with the values ”1 0 0” meaning no forces applied in the y and z direction but only in the x direction. Furthermore, we didn’t put gravity in our simulation. Otherwise, the liver would have fallen.

As we work in real time simulation, we use some results that were already developed, allowing us to gain significant computation time.

3.5 Conjugate Gradient on GPU

The conjugate gradient (CG) is an iterative algorithm used to find the numerical solution of particular systems of linear equations. It is often used for linear systems that are too large to be solved by a direct method. This algorithm is used in our cutting method.

In the following article [ACF11], Allard et al developed a method combining a FEM solver and the conjugate gradient on GPU.

This method is matrix free, meaning we do not have to assembly the matrix, which provides a significant gain in computation time.

The implementation of this method is available in Sofa. We are now able to directly use it and notice faster computation time.

3.6 Asynchronous preconditioner

While working with matrices, an important value is the condition number of the matrix. In fact, the condition number allows us to measure how much our results can change due to a small perturbation in the input argument. It is therefore important to have a good condition number.

With ill-structured meshes, the condition number is often high. Having a high condition number can raise to convergence issues for the conjugate gradient algorithm.

A common solution is to use a preconditioner. It allows to reduce the condition number and consequently ensure a faster convergence of the algorithm.

A preconditioner P of a matrix A is a matrix such as the condition number of $P^{-1}A$ is smaller than the condition number of A.

Instead of solving

$$Ax = b$$

we solve

$$P^{-1}Ax = P^{-1}b.$$

Using a preconditioner is helpful for convergence issues. However it adds computation time in our simulation. In 2013, Courtecuisse et al [HCD13] made the following statement:
if we presume our ill-conditionned matrix A undergoes only small perturbations between two consecutive time steps, we can choose to save the same preconditioner matrix during a few time steps. In fact, A having only small changes, the preconditioner matrix will remain a "good enough" approximation for our simulation.
The preconditioner can then be updated at low frequency on a dedicated CPU thread. The update of the preconditioner can take several time steps, during which we are going to use what became an approximation of our preconditioner.

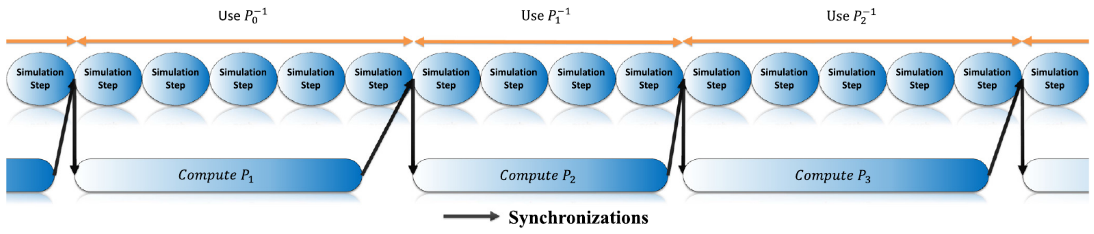


Figure 8: The preconditioner is updated asynchronously. We use the last preconditioner available until a new one is computed. [HCD13]

4 The cutting method

The MLMS team is working on developing a new cutting simulation method. This method has several steps we are going to present.
Please note that in our examples, the topology we work on is composed of tetrahedrons on the inside and triangles on the surface.

The new cutting method includes two distinct Sofa scenes. The two scenes are able to communicate through Ros2. More details on Ros2 can be found in the following section 4.2.1.

The advantage in using two distinct Sofa scenes is the gain in computation time as we can use two computers to perform the cut (or even just different threads).

The first scene follows the physical behavior explained earlier (in section 3.2) while the second one uses a linear physic model.

Ros2 (Robot Operating System 2) is a set of tools for building robot applications. It has the advantage of being open source and used by thousands of developers. In our case, we use this software to transfer data from one scene to the other and vice versa.

We split our two scenes as follow:

- a first scene containing the deformable object
- a second scene containing the same object with a simpler physic model.

Both scenes contain the same topology. The first scene is the one containing all the topology information and sending it to the second one.

In the new cutting method, we use the node snapping technique.
The node snapping strategy [NvdS01] allows us to limit the topological change while avoiding generating new elements that may cause high computation costs. After the node snapping process, the snapped position should be registered to fit the cutting surface. However, the straightforward way that registers the node displacement in the deformed shape will lead to a critical problem that the stress energy of deformation will be lost. To solve the problem, an intelligent strategy is to register the snapping in the rest shape. For this purpose, we transfer the objective cutting surface into the rest shape. Then the node snapping will be processed with a 'virtual' object (a dedicated model) with the rest shape of the object and the transferred cutting surface. The snapped positions will overwrite the rest state of the 'real' object (the actual simulation process). Consequently, the deformation of the 'real' object will follow the displacement in the rest state. Our strategy that snaps the nodes in the rest shape makes it possible to fit the cutting surface without energy losses in the deformed shape.

The method we developed can be separated into several independent steps, as follows:

4.1 The different steps of the method

The first two steps of the cutting method are taking place in the first scene.

4.1.1 Point cloud generation

The point cloud representing the cutting path is generated from the instrument taking as input the result of collision detection between the mesh and the tool. We use the barycentric coordinates for each point to compute the corresponding position in the rest shape. Consequently, a transferred point cloud representing the cutting path in the rest state is generated.

To control the resolution and density of points of the cutting surface, we use a user-specified constant to add points located at a distance d_{min} in the rest shape from each point in the cloud. We integrated a constraint to model the contact between the object and the instrument. This interaction will generate points in the direction of the normal of the contact constraint according to the force computed. In this manner, the force computed on the surface and its direction will have an impact on the depth of the cut.

4.1.2 Cutting plane generation

As it will be challenging to use the generated point cloud directly to present the cutting plane, a triangulated surface mesh will be preferable. For this purpose, we use the Point-Cloud Library (PCL) to generate a 3D surface from the cloud [MRB09]. We do a polynomial fitting using Moving-Least Squares to have a consistent estimation of the normals of the surface points. Due to the computation of the positions inside the rest shape from the deformed shape, the point data can be noisy inside the rest shape. The Moving-Least Squares will smooth the data and guarantee a continuous plane inside the rest shape. For triangulation, we use a method in PCL that will estimate the tangential plane of the current point thanks to its neighborhood and then project the neighborhood on this plane to create connections to form triangles using maximum angle criteria specified by the user. At the end of this step, we obtain a cutting surface formed by triangles in the rest shape of the object.

The cutting scene is now sending the positions of the cutting surface to the virtual scene (thanks to Ros2). This latter will need these positions for the futur steps of the method.

4.1.3 Searching snapping points

This part is taking place in the virtual cut scene.
With the cutting surface, our current objective is to search for the points to snap. As these nodes will be snapped onto the cutting surface to fit it, they will be directly used later in the topological change as the separation boundary of the mesh. In order to avoid the problem caused by the topology connection after cutting (e.g., remaining connection or resolution problems), an essential condition should be satisfied: The connections between these nodes in the original mesh should form a continuous triangle surface. Furthermore, while the cutting process has separated the mesh into two parts, the triangle surface (consisting of snapping nodes) should make both the parts as manifold mesh.

To minimize the snapping displacement, the objective in this section turns out to be searching for a continuous triangle surface in the original mesh, while this triangle surface should be as close as possible to the cutting surface. We propose a method to search the triangle surface: As long as the cutting process continues, we first search for a closed circle (consisting of edges) closest to the cutting surface in each time step. Then we use a propagation algorithm that fills up the circle as a continuous triangle surface. Along the direction of the cutting path, a part of the circle will be used as the starting condition for searching for the circle in the next time step. Consequently, the triangle surface found in each step will be connected and finally form a complete surface separating the mesh as two manifold parts.

4.1.4 Node snapping

We are still working in the virtual scene.
Now we can process the snapping with the nodes on the triangle surface searched in the previous section. Snapping the nodes onto the cutting surface is implemented by the constraint-based technique presented in section 3.3. By projecting each node (in the rest shape) onto the cutting surface, we obtain the projective direction and the distance to the cutting surface. In the 'virtual' model, we apply a bilateral constraint on each node to snap and solve the constrained system using Lagrange multipliers. By fixing the surface nodes of the mesh, the nodes chosen in the last section are snapped onto the cutting surface, pulling as well the other nodes inside the mesh.

Generally, solving a constrained system using Lagrange multipliers can be costly in the case of large-scale problems. In our strategy, we can benefit from a pre-computed factorization since the current step is processed with a virtual model in the rest shape of the mesh where the system matrix is constant. Furthermore, the *isolating mechanical DoFs* method proposed in [ZCC22] makes it highly efficient while handling the constrained system in large-scale problems. Therefore, solving the constrained system is very fast in this step.

We now send back (through Ros2) the new positions of the mesh.

4.1.5 Rest state registration and topological change

The node snapping in the virtual object results in the displacement of the mesh nodes in the rest shape. Such displacement in the rest shape should be applied to the real object so that the deformed shape can follow the cutting surface. Updating the rest position is insufficient, and the initial stiffness matrix for each element should also be updated.

Update FEM information

In our simulation, we use the Finite Element method. Using this method requires to compute at the initialisation a Ke_0 matrix.

The Ke_0 matrix has to be assemblied according to the rest positions. In fact, over the steps of our simulation, as the positions of the nodes evolve, the local stiffness matrices need to be recomputed (at every time step).

The rest positions representing the initial position of the nodes, the Ke_0 usually does not change. But in our cutting simulation scenario, it is different. In fact, when applying the node snapping technique, we move nodes. In addition, the change of topology structure will also require recomputing the Ke_0 since some of the connections will be modified. This implies an update for the Ke_0 matrix in case of node snapping or topological change.

We propose to optimise this update. See more details in the following section 4.2.2.

Once the rest state is updated, the snapping nodes found in 4.1.3 can be used as the separation boundary of the mesh. Each snapping node will be duplicated as two, representing the lower surface of the upper part and the upper surface of the lower part, respectively. For each side, the connection between the snapping nodes and the other side will be broken. Finally, the local mesh is divided into two parts.

4.2 Contributions

During my internship, I was able to bring some contributions to some of the steps presented in the previous section.

4.2.1 Ros2

As explained before, we use Ros2 for data transfer. However, some implementation stages were needed in order for it to work.

Sofa is the software we use for our simulations. Sofa is implemented in C++ and has its own classes. When we want to transfer a data having a specific Sofa type, we need to implement a few functions.

We first need to add the Sofa class in our Ros2 factory. Our Ros2 factory contains all the classes the system can work with.
Once this part is done, we need to implement some functions in order to convert our Sofa class to a more standard class supported by Ros2.

We can give an example. In our situation, we are interested in sending topologies.
A topology contains Points, Edges, Triangles, and Tetrahedrons (in our specific case).

In standard C++ classes, these information can be contain in vectors (std vectors for example). Whereas in Sofa, these information are contained in Sofa types. Let's see the different types we work with and their correspondence.

When working with point positions in C++, we can use a vector of a three dimension vector (assuming we are in 3D and every point has x,y,z values as position). In Sofa, the only difference is we use specific Sofa vector instead of C++ std vector.

For Edges, Triangles and Tetrahedrons, it is more complex. In fact, whereas in C++ these elements would be stored using a vector of vectors of indices, in Sofa their type is defined using a Sofa vector and Sofa specific topology information.

sofa::type::vector<sofa::core::topology::BaseMeshTopology::Edge> is the Sofa type used for the Edges.

Once the conversions were done, we were able to create clients and servers in order to get information from a first scene into a second one. This step had to be done after all the conversion were already functional as the clients and servers only support Ros2 types.

Thanks to these implementations steps, we were able to run our cutting simulation using two Sofa scenes.

4.2.2 Computation of initial matrix

The snapping constraint step is essential as it allows to move some nodes and thereby bring them onto the cutting surface. However, this step raises an issue. We apply constraints in our virtual scene and update the rest shape position of the real object. This update implies the stiffness matrices associated to the initial tetrahedrons are now deprecated and need an update. But recomputing all the stiffness matrices is computationally very costly.

To avoid this costly computation time, we decide to only recompute the matrices of tetrahedrons who have points that moved of a distance bigger than a specific threshold.

We decided to first have an overview of the proportion of matrices that would need to be recomputed according to different thresholds.

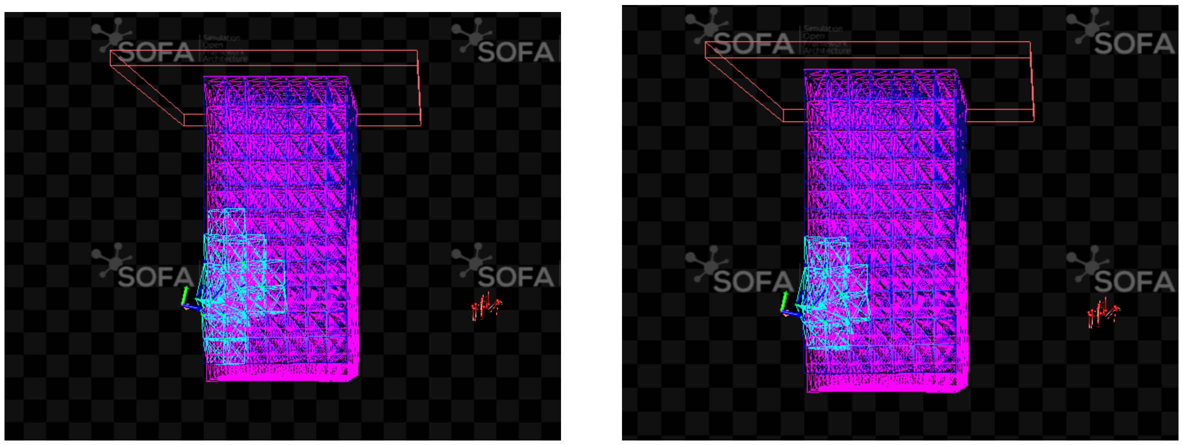


Figure 9: In turquoise, we visualize the tetrahedrons of the scene that would need to be recomputed for a specific threshold. On the left side, we have a threshold of 0.1, and on the right side, we have a threshold of 0.2. Both images were taken at the same time step. The only change between these two simulations is the threshold.

As expected before any visualization, the tetrahedrons that have points that move the most are those near the cutting plane. Furthermore, the smaller the threshold is, the more tetrahedrons to be recomputed we obtain.

Recomputing only some of the stiffness matrices of the mesh let us gain some computation time and the cutting simulation solution was still totally acceptable.

Another point we were able to work on is the following: during some steps of the simulation, tetrahedrons are being added and/or removed. These topological changes force us to recompute all the matrices previously computed according to the previous topology. In fact, the topological changes imply changes in the size and filling of the matrices. To avoid recomputing all our matrices at every time step, we distinguished between the steps where an addition or deletion of tetrahedron.s appeared and the steps with point moves only.
In the previous case, the perturbations in the different matrices are quite small and not all the matrices have to be recomputed.

Given the fact we recomputed a smaller number of matrices, the computation time was accordingly reduced. We give more details on this optimisation is the following section 4.2.3.

So far, we gained some computation time. But we tried to go further in our optimisations.

When making topological changes, some of our elements can be deleted and others can be created. Our elements can be points, edges, triangles or tetrahedrons.
The global stiffness matrix is directly impacted by these changes in the topology. We already explained how not to recompute the whole matrix if only points have moved. But we are now trying to optimise our code in order not to recompute the global stiffness matrix even when topological changes happened. In order to have the best computation time possible, optimising this step is essential.

Sofa has its own way to handle elements addition or suppression in parallel (using Cuda). Let's explain it.

At the beginning of the simulation, Sofa collects all the topology information. With these information, it creates a temporary vector of size the number of elements in the topology multiplied by the size of each element (for a point, the size is 1; 2 for edges; 3 for triangles and 4 for tetrahedrons).
The advantage of this temporary vector is every element has its own space, every node of our elements have a cell in the vector. Thus, there cannot be conflict in writing.

Sofa also creates a pre-computed vector which, for each node of the topology, stores the different addresses of the temporary vector who add a contribution to this node.

We can assure there will be no conflict in writing because even though the operations that need to be done depend on the FEM method, their parallelisation will only depend on the topology. In fact, no matter how our operations were computed, we will have to accumulate the contribution of each element in the pre-computed vector.

The following figure can help us understand the way Sofa works.

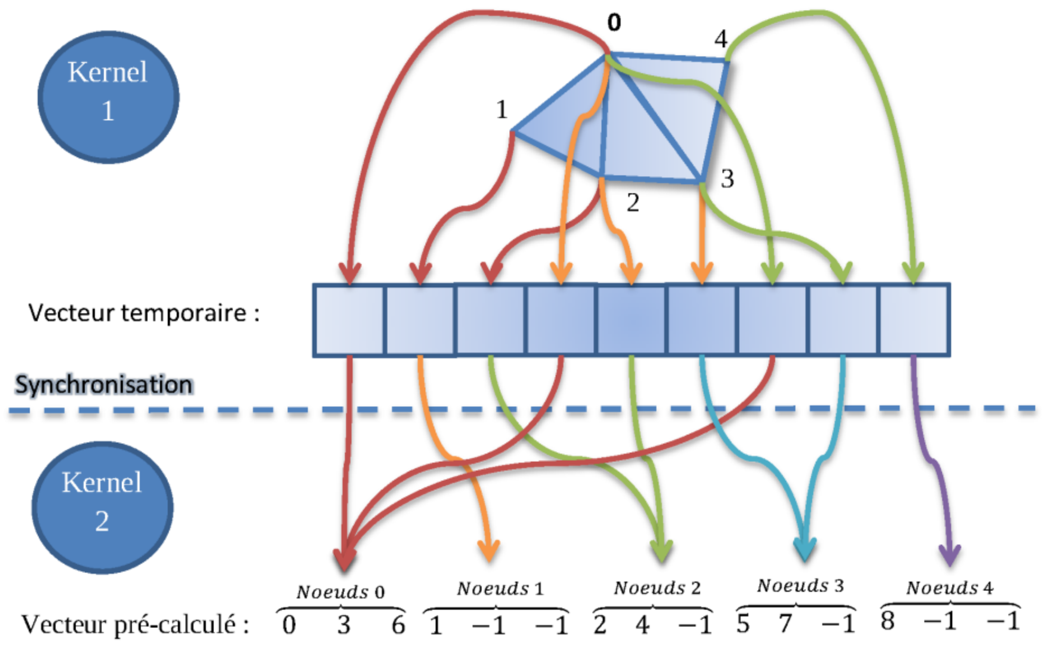


Figure 10: Parallelisation of the force calculation [Cou11]. We distinguish in this figure the first step that creates the temporary vector and the second step that accumulate the contributions we were able to compute in parallel.

Sofa handles tetrahedron removal in the following way: it uses the cells that were attributed in the vector for the removed tetrahedron and uses them for the last tetrahedron. It allows to diminish the size of the vector.

For tetrahedron addition, Sofa places the information of the tetra at the end of the vector and through this step increases the size of the vector.

These actions take time (in particular due to the fact the vector is often resized) and could be optimised for our cutting simulation. In fact, when cutting a mesh, we first delete some elements and recreate them (when we apply our duplication algorithm). Thus, the number of tetrahedron we have in the topology can never go bigger than what was at the initialisation. It is the reason we decided not to change the size of the vector every time a deletion/addition of an element happens but instead, to initially create a bigger vector able to store all the topology information and to handle the elements addition/deletion. This vector will undergo several updates, but its size will remain the same throughout the simulation.

So far, we were only able to implement this optimisation for tetrahedron removal.

We present the time results we obtained in the next section 4.2.3.

4.2.3 Performances results

In this section, we present the performances results we had during our different simulations. The simulation tests were conducted in the open-source SOFA framework with a CPU Intel@ Core i7-6700 at 3.40GHz with 32GB RAM, and a GPU GeForce RTX 3050.

The distinction we made between the steps where an addition or deletion of tetrahedron.s appeared and the steps with point moves only allowed us to gain some time. In order to see to what extent this distinction had an impact, we first calculated the number of steps where addition/deletion of elements happen.

When calculating these number of steps on the topology presented in figure 9, we obtained the following results: our simulation has 51 steps with addition/deletion of elements and 101 steps with point moves only.

In addition, we also calculated the mean time our point moves function takes and the mean time for recomputing all the topology. The mean time was calculated over 100 steps. In our simulation, the point move function takes around 0.22 ms to be computed whereas recomputing the whole topology takes around 1.53 ms.

If we sum up, we have the following approximate gain of time for the entire simulation: given the fact we can use the optimisation on 101 steps (out of 252 steps), we gain approximately 132.31 ms. In fact, we gain $(1.53 - 0.22)$ ms per time step when we can use the optimisation. We can apply the optimisation on 101 time steps which gives:

$$(1.53 - 0.22) * 101 = 132.31.$$

Keeping the same vector along the whole simulation also let us gain some time in our simulation.

The numbers we give in the following table are times in ms. They correspond to the mean time the function we optimised took over 100 steps.

Version	Between step 300 and 400	Between step 400 and 500
Non optimised	0.63	0.46
Optimised	0.08	0.09

As expected, not having to reinitialise the vector saves time.

In fact, as we can notice in the previous table, when we use only one vector for the whole simulation, we replace our previous method by a method that takes 7.875 times less time to compute. The number 7.875 is the factor we had for the simulation steps between 300 and 400. For the steps between 400 and 500, the factor is 5.11.

We made in the previous table a distinction between the steps $300 - 400$ and $400 - 500$. These simulation steps arrive at a different level during the cutting simulation so it is not surprising to find different results.

5 Conclusion

During my internship, we were able to find a few ways that could improve the computation time of a cutting simulation.

Firstly, we can cite the use of Ros2. By being able to split our simulation on different computers, the simulation time can be significantly reduced.

The second major contribution was the optimisation for recomputing all the local stiffness matrices. By the end of my internship, not all Sofa topological changes were functional with the optimisation. However, the removal of tetrahedrons gave good results and we may hope the other topological changes will too.

References

[ACF11] Jérémie Allard, Hadrien Courtecuisse, and François Faure. Implicit FEM Solver on GPU for Interactive Deformation Simulation. In Wen mei W. Hwu, editor, *GPU Computing Gems Jade Edition*, Applications of GPU Computing Series, pages 281–294. Elsevier, November 2011.

[Cou11] Hadrien Courtecuisse. *Nouvelles architectures parallèles pour simulations interactives médicales.* Theses, Université des Sciences et Technologie de Lille - Lille I, December 2011.

[DDKA06] C. Duriez, F. Dubois, A. Kheddar, and C. Andriot. Realistic haptic rendering of interacting deformable objects in virtual environments. *IEEE Transactions on Visualization and Computer Graphics*, 12(1):36–47, apr 2006.

[HCD13] P Kerfriden S. Bordas S. Cotin H. Courtecuisse, J. Allard and C. Duriez. Real-time simulation of contact and cutting of heterogeneous soft-tissues. *Medical image analysis*, 18:394–410, 12 2013.

[IBM17] Rafael Torchelsen Iago Berndt and Anderson Maciel. Efficient surgical cutting with position-based dynamics. *IEEE Computer Society*, 2017.

[ICu] ICube. Icube. https://icube.unistra.fr/en/.

[Jea99] M. Jean. The non-smooth contact dynamics method. *Computer Methods in Applied Mechanics and Engineering*, 177(3-4):235–257, 1999.

[MLM] MLMS. Mlms. https://mlms.icube.unistra.fr/en/index.php/Presentation.

[MRB09] Zoltan Csaba Marton, Radu Bogdan Rusu, and Michael Beetz. On fast surface reconstruction methods for large and noisy point clouds. In *2009 IEEE International Conference on Robotics and Automation*, pages 3218–3223. IEEE, may 2009.

[NvdS01] Han Wen Nienhuys and A. Frank van der Stappe. A surgery simulation supporting cuts and finite element deformation. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 2208, pages 145–152. Springer, Berlin, Heidelberg, 2001.

[Ren13] Yves Renard. Generalized Newton's methods for the approximation and resolution of frictional contact problems in elasticity. *Computer Methods in Applied Mechanics and Engineering*, 256:38–55, 2013.

[Sof] Consortium Sofa. Simulation open framework architecture. https://www.sofa-framework.org/.

[WM17] Monan Wang and Yuzheng Ma. A review of virtual cutting methods and technology in deformable objects. *Wiley*, 2017.

[ZCC22] Z. Zeng, S. Cotin, and H. Courtecuisse. Real-time fe simulation for large-scale problems using precondition-based contact resolution and isolated dofs constraints. *Computer Graphics Forum*, 2022.