# Tutorial

## Modeling, Simulation and Control of Deformable Robots on SOFA Framework

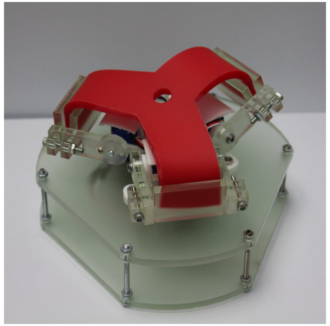
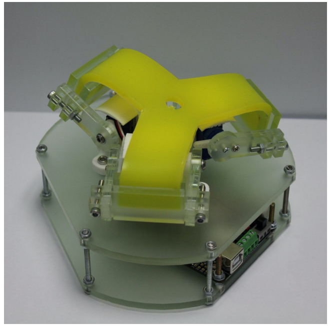
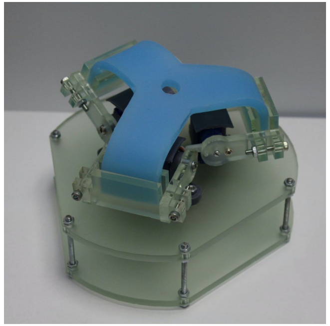
DEFROST team

# General overview of the Tutorial

https://team.inria.fr/defrost/

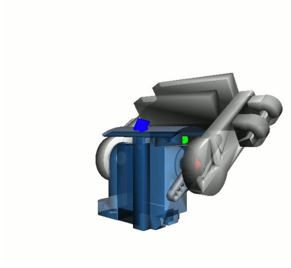
# 9:00 - 11:00: Session 1:

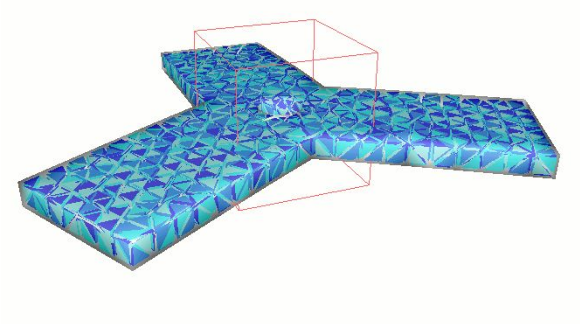
- 9:00 am: Starting
  - 9:00 am: Introduction (round table) and short presentation of the tutorial (in particular the Hybrid mode)
  - 9:10 am: Installation of SOFA on your Machine and first tests
  - 9:30 am: Notions of mechanics useful for the Tutorial (Christian)

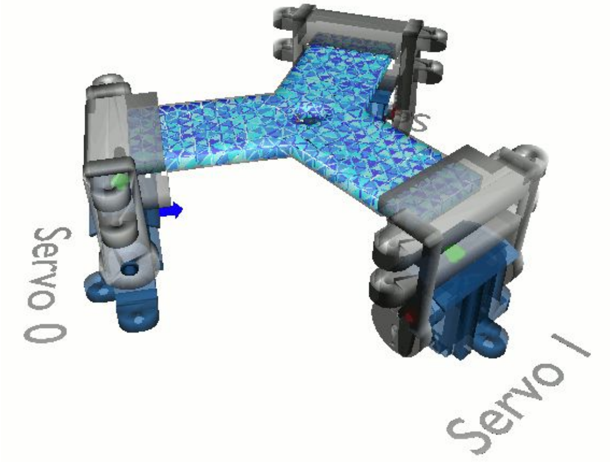




# 9:00 - 11:00: Session 1

- 10:00am: Tripod Tutorial (part 1)
  - Main steps for direct modeling
    - Finite Element Model
    - Articulated system for servo motor
    - Coupling

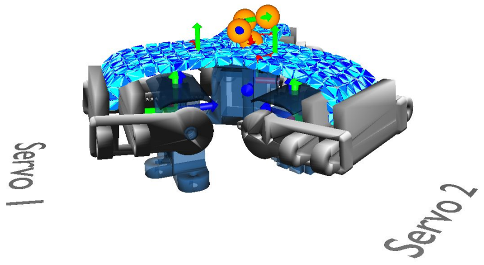






# 11:00 - 12:30: Session 2

- 11:00am: Presentation of the SOFA community and consortium
- 11:15am: Tripod Tutorial (part 2) :
  - Inverse modeling
  - Maze motion planning
  - Test on the digital twin
  - Test on the robot (for people on site)
- 12:15am: Conclusion and ongoing work




Todo : show what we will have at the end of the tutorial ?

# Session 1

**9:10 am to 9:30 am**:
Installation of SOFA on your Machine and first tests

# Practical informations for installation

Follow instructions at

github.com/SofaDefrost/RoboSoft2022

# This is not a commercial product !

- Strong efforts to make it work on all platform Our goal is to disseminate SOFA for Soft-Robotics, and find new usages & contributors
- Any issue using SOFA? your feedback is valuable for us
    - Robosoft  2022: we are here to help you !
    - Later: we stay by your side
- We already would like to thank all the member of the DEFROST team for their contribution as well as the SOFA consortium for helping us to set up this tutorial

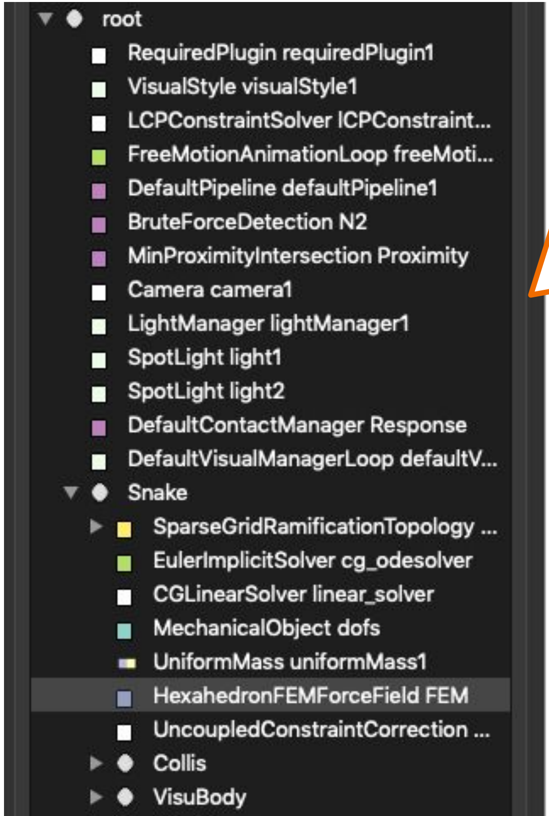
# Installation test

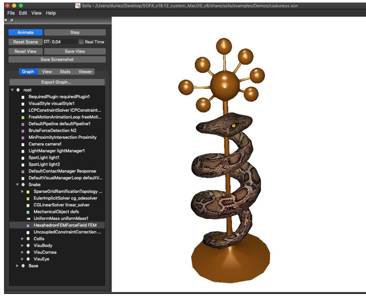
- Let's enter the world of simulation
  - Read instructions for your OS → github.com/SofaDefrost/RoboSoft2022
  - Make sure to install pre-requisites
  - Download the SoftRobots zip
  - try runSofa with the file workshop.pyscn (on the SOFA repository)

- Report us any issue
  - Let's fix this together

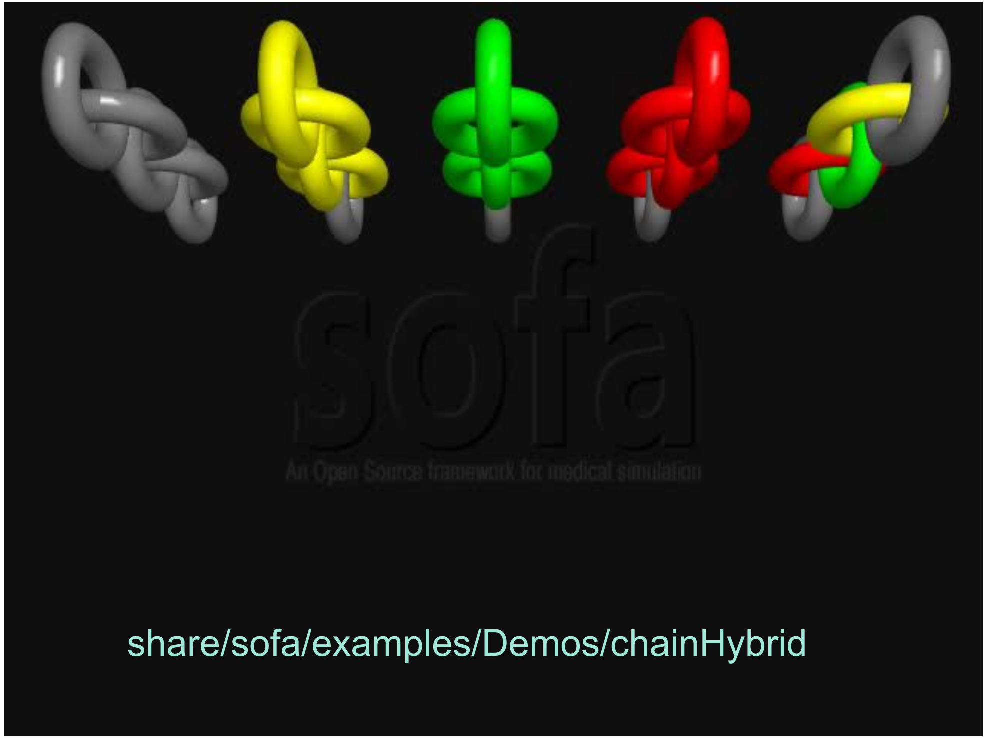
# Main principles of SOFA :: the graph

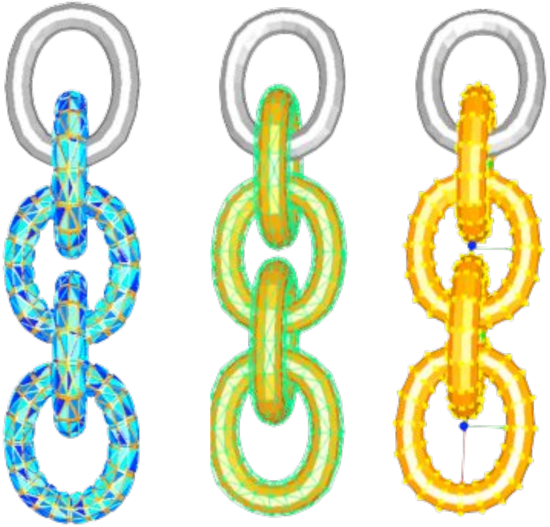
- Scene Graph
  - Nodes
  - Components
  - Data in components



# Multi-models



FEM

Spring Mass

Rigid

# Tutorials ...

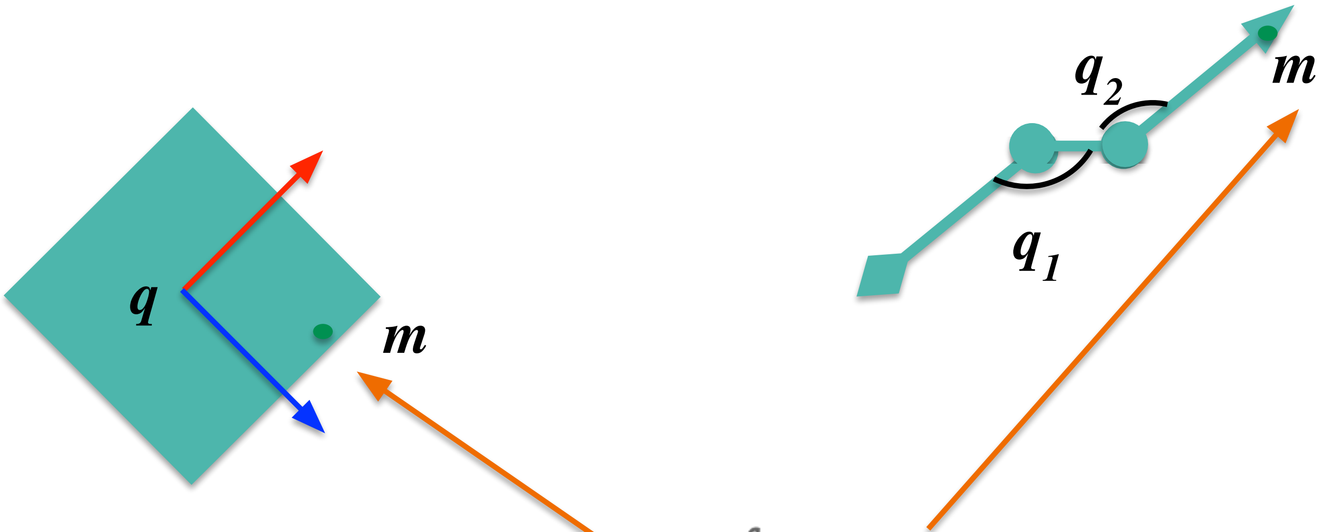
# Session 1

**9:30 am to 10:00 am**:
Notions of mechanics useful for the Tutorial

# Multi-Models Mechanics

- (Articulated) rigid body dynamics

$$J^T(q) \mathrm{M} J(q)\, \ddot{q} + \mathrm{C}(q, \dot{q}) = \tau(q)$$

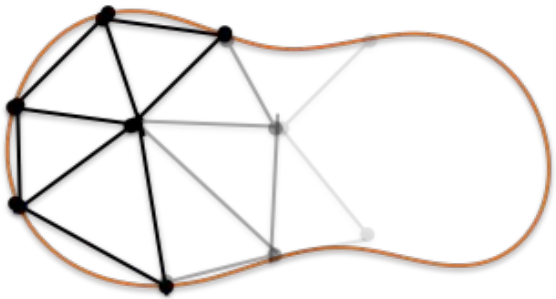



$$\mathrm{M} = \int m \, \partial\Omega$$

# Multi-Models Mechanics

- Deformable body with FEM

$$M\ddot{q} + f(q, \dot{q}) = f_{ext}$$



$q$ are nodes position in global coordinates
$M$ close to diagonal, diagonal if mass lumping
$f(q, \dot{q})$ internal forces from FEM

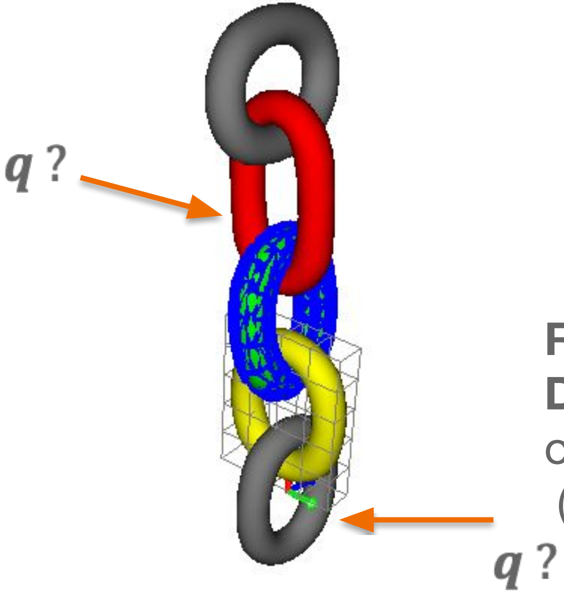
$$f(q + \partial q, \dot{q} + \partial \dot{q}) \approx f(q, \dot{q}) + K(q)\,\partial q + B(q)\partial \dot{q}$$

Updated linearization (at each simulation step)

# Multi-Models Mechanics

- Interaction between models

**FEM Model**
**DOFs:** positions of nodes
(Vec3 types in SOFA)




**Rigid Model**
**DOFs:** positions and orientation of gravity center
(Rigid types in SOFA)

# Configuration space / kinematic links

- Lagrangian Mechanics:
  - State variables: (q, q’) [Generalized coordinates] + t [effort same space]
  - Kinematic relation:  x = g(q)
  - Kinetic relation: x’ = dg/dq q’ => J q’
  - **Virtual work principle =>   t = Jt f (to develop)**
- **In SOFA,**
  - **Mappings= [Kinematic / Kinetic / Force transfer]**
  - **q,q’ = parent models**
  - **x, x’ =  child models**
  - position and velocity imposed by the mapping of a parent MechanicalObject
  - force can be applied on slave models and transmitted to the parent

# Main principles of SOFA :: main components

- Mapped Mechanical objects: **slave models**

**FEM Model**
DOFs: positions of nodes (Vec3)

**Collision Model**
**Mapped DOFs:** positions of points (Vec3)
BarycentricMapping

**Rigid Model**
DOFs: positions and orientation of gravity center (Rigid)

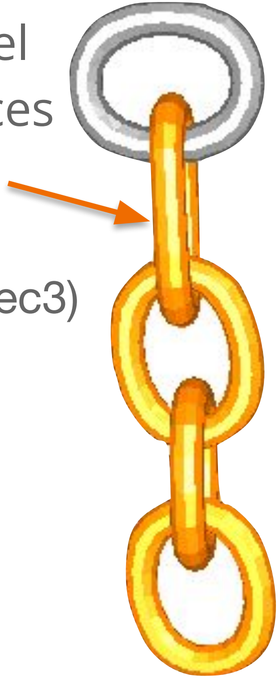
**Collision Model**
**Mapped DOFs:** positions of points (Vec3)
RigidMapping

# Main principles of SOFA :: main components

- Mapping
    - Allow to transfer the motion (pos, vel) to a « slave » model
    - Allow to transfer back to the « parent » model some Forces

# Main principles of SOFA :: main components

- Mapping
  - Allow to transfer the motion (pos, vel) to a « slave » model
  - Allow to transfer back to the « parent » model some Forces

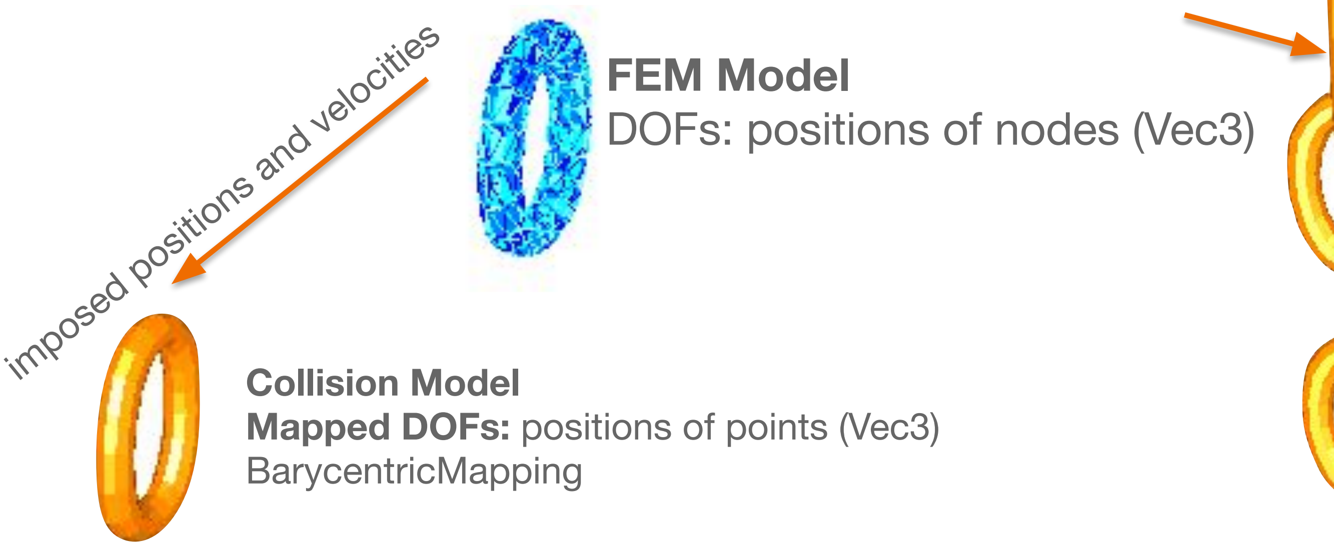


**FEM Model**
DOFs: positions of nodes (Vec3)

**Collision Model**
**Mapped DOFs:** positions of points (Vec3)
BarycentricMapping

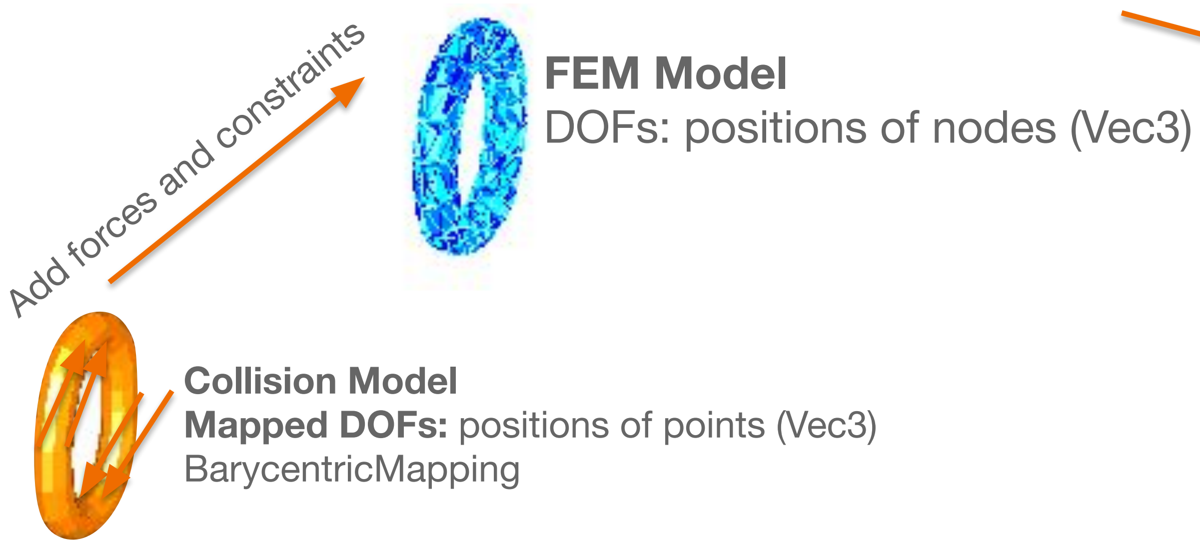
# Main principles of SOFA :: main components

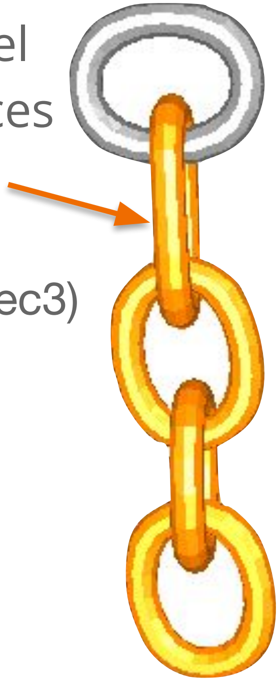
- Mapping
  - Allow to transfer the motion (pos, vel) to a « slave » model
  - Allow to transfer back to the « parent » model some Forces



# Main principles of SOFA :: main components

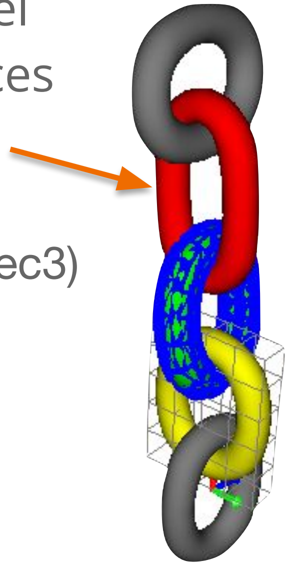
- Mapping
  - Allow to transfer the motion (pos, vel) to a « slave » model
  - Allow to transfer back to the « parent » model some Forces



# Main principles of SOFA :: main components

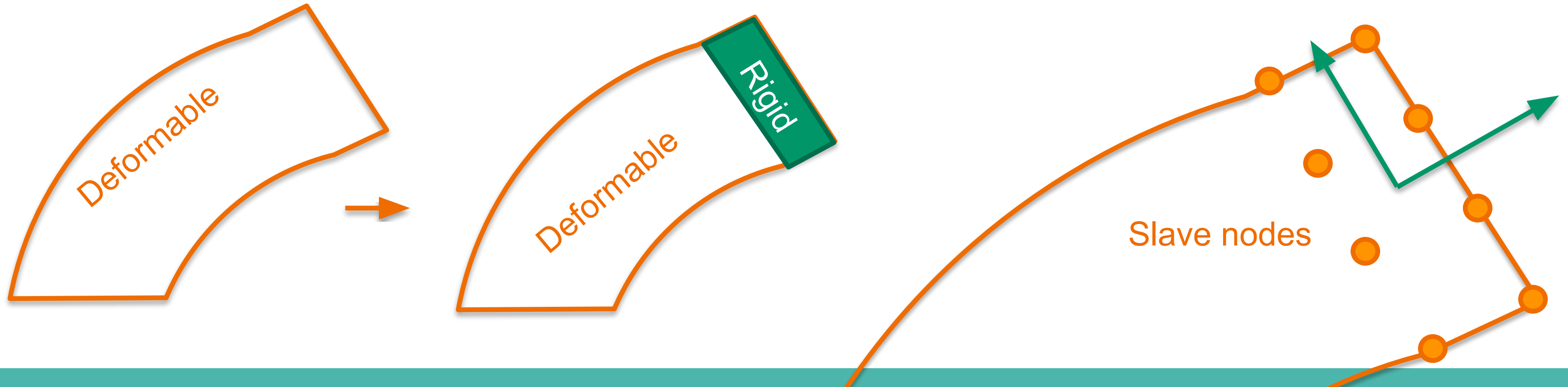
- Mapping
  - Allow to transfer the motion (pos, vel) to a « slave » model
  - Allow to transfer back to the « parent » model some Forces

# Deformable-rigid coupling

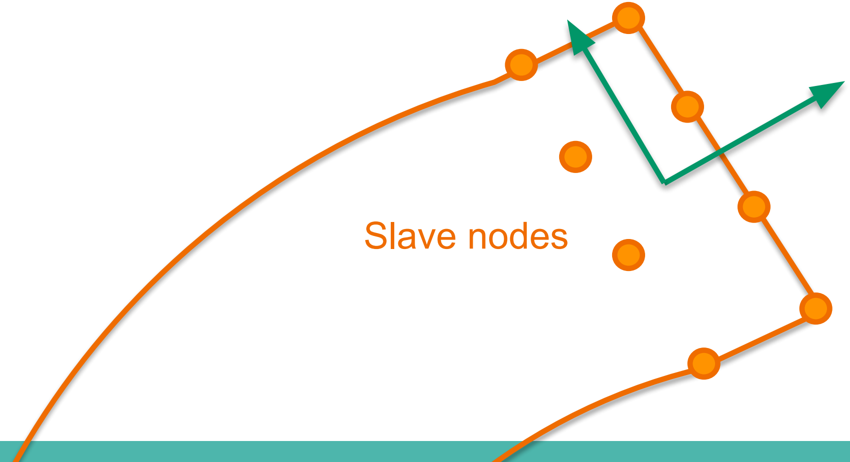
## Why composite mechanics ?

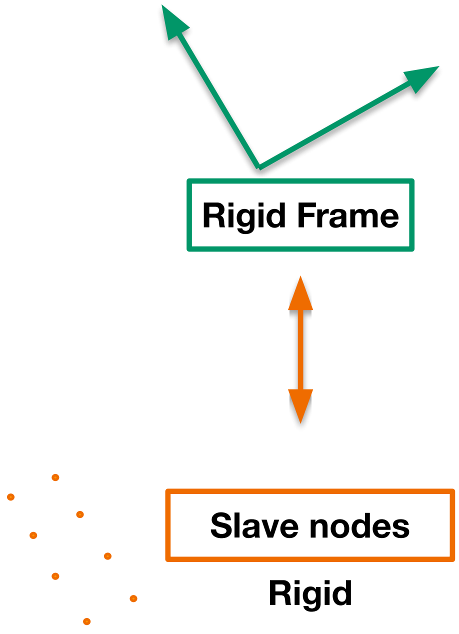
- Soft robot can be composed of rigid sections (backbones)
- Importance of computing the coupling between rigid parts and deformable parts.



# Deformable-rigid coupling

Hierarchical representation

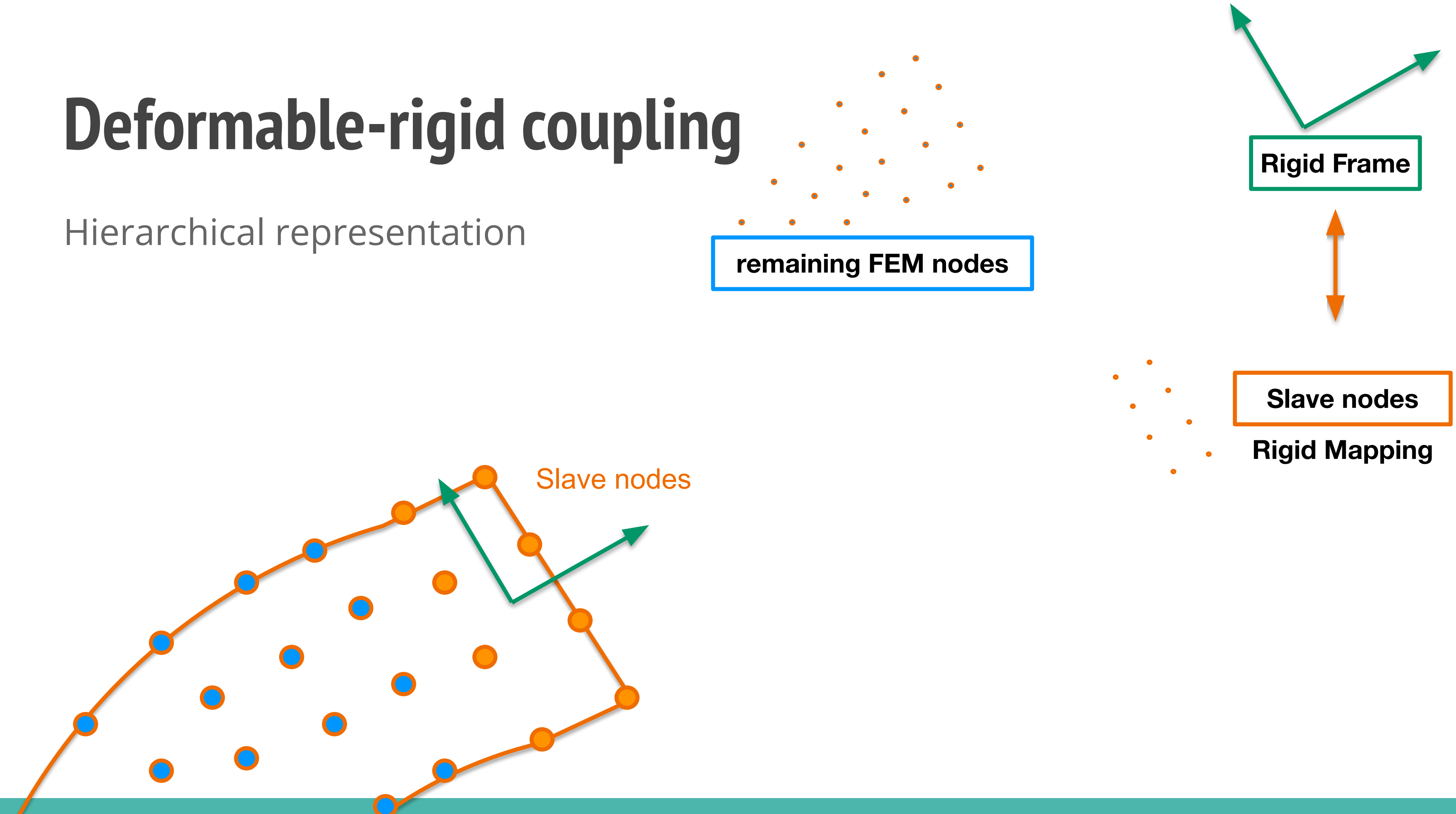



in sofa => (Mapping)

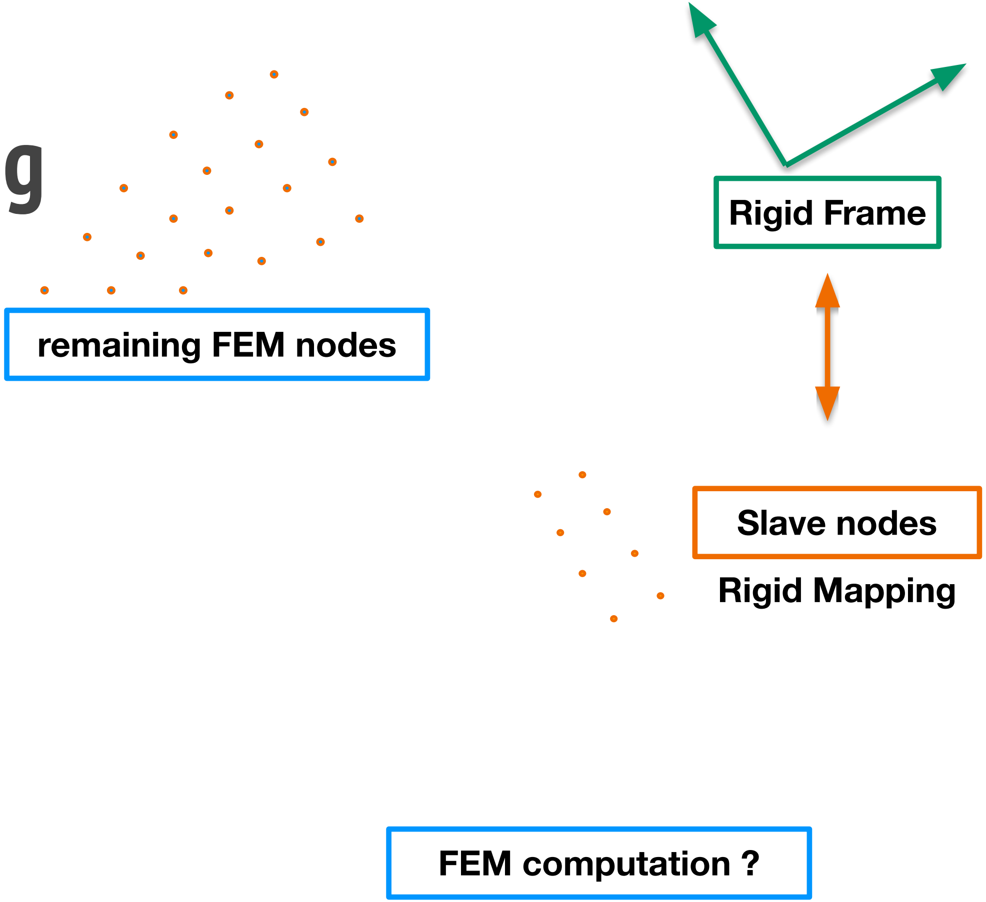
# Deformable-rigid coupling

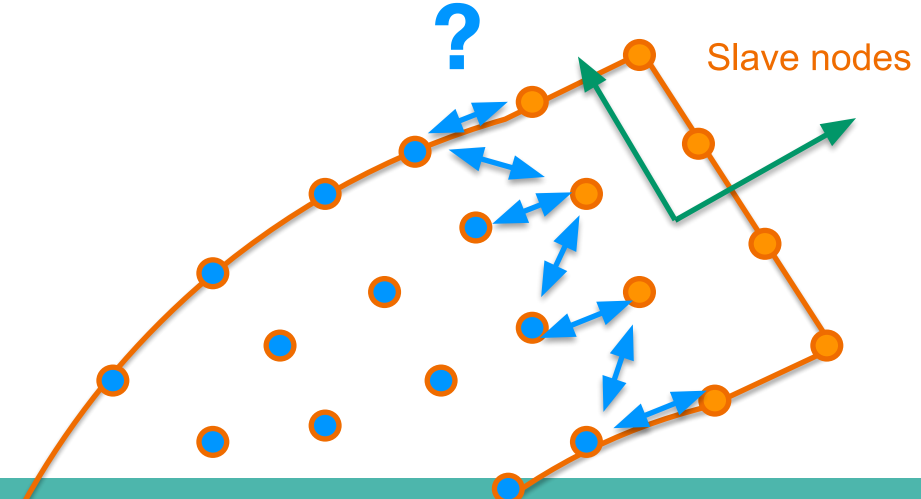
Hierarchical representation



# Deformable-rigid coupling

Hierarchical representation

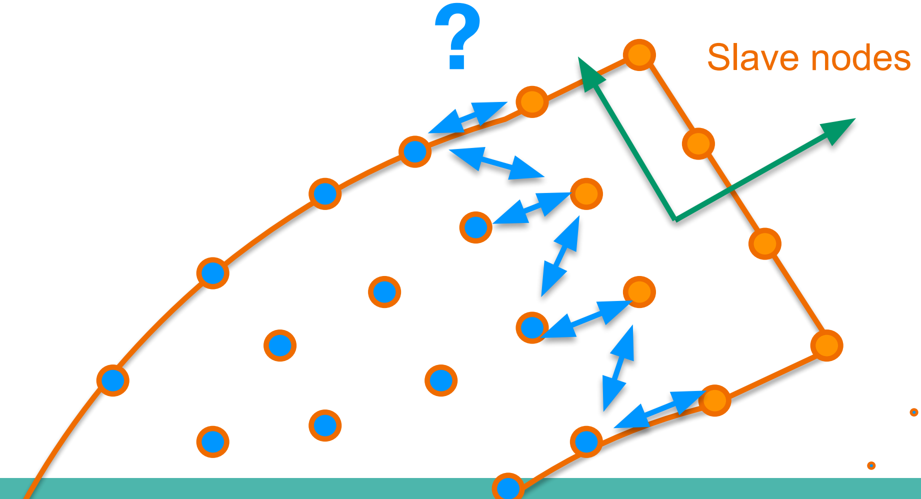


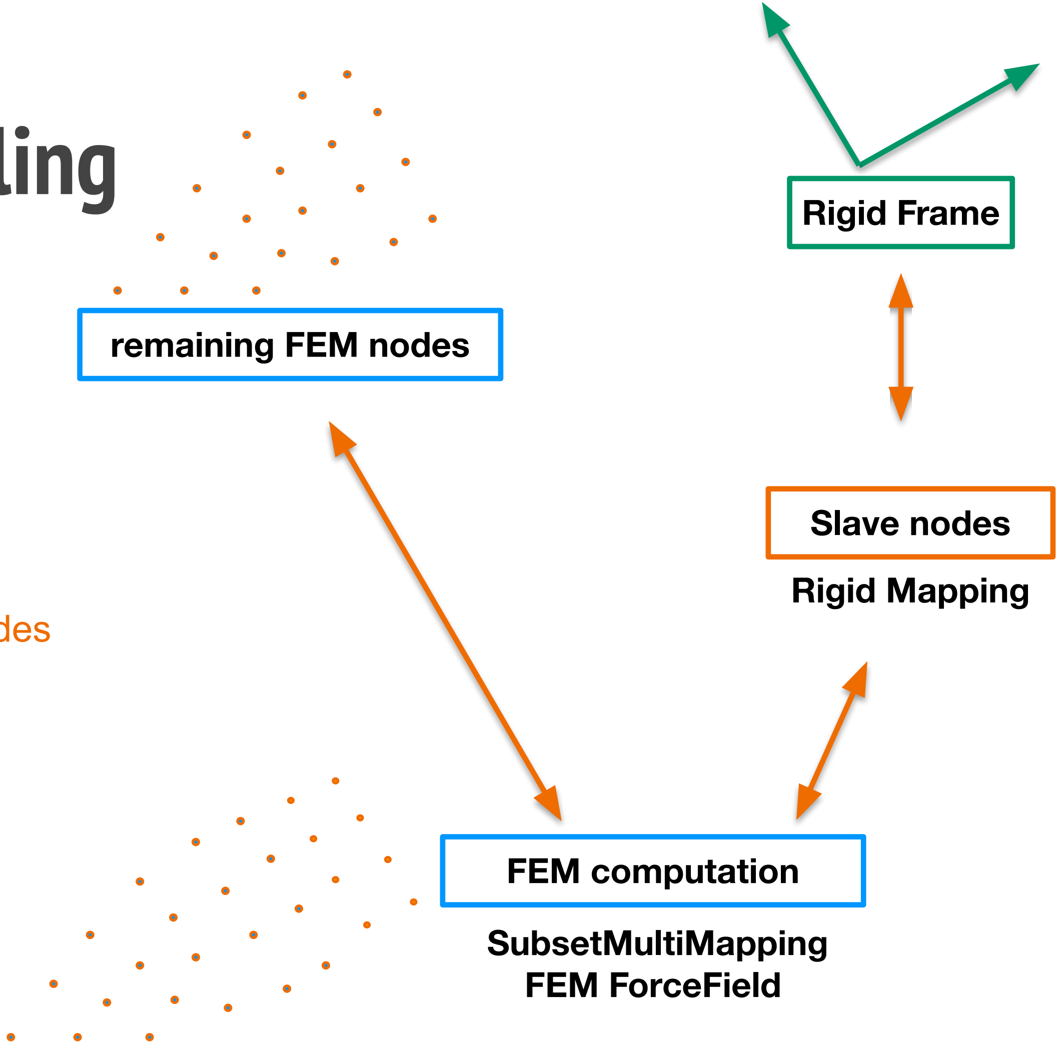



# Deformable-rigid coupling

Hierarchical representation

Multi-Mapping Concept

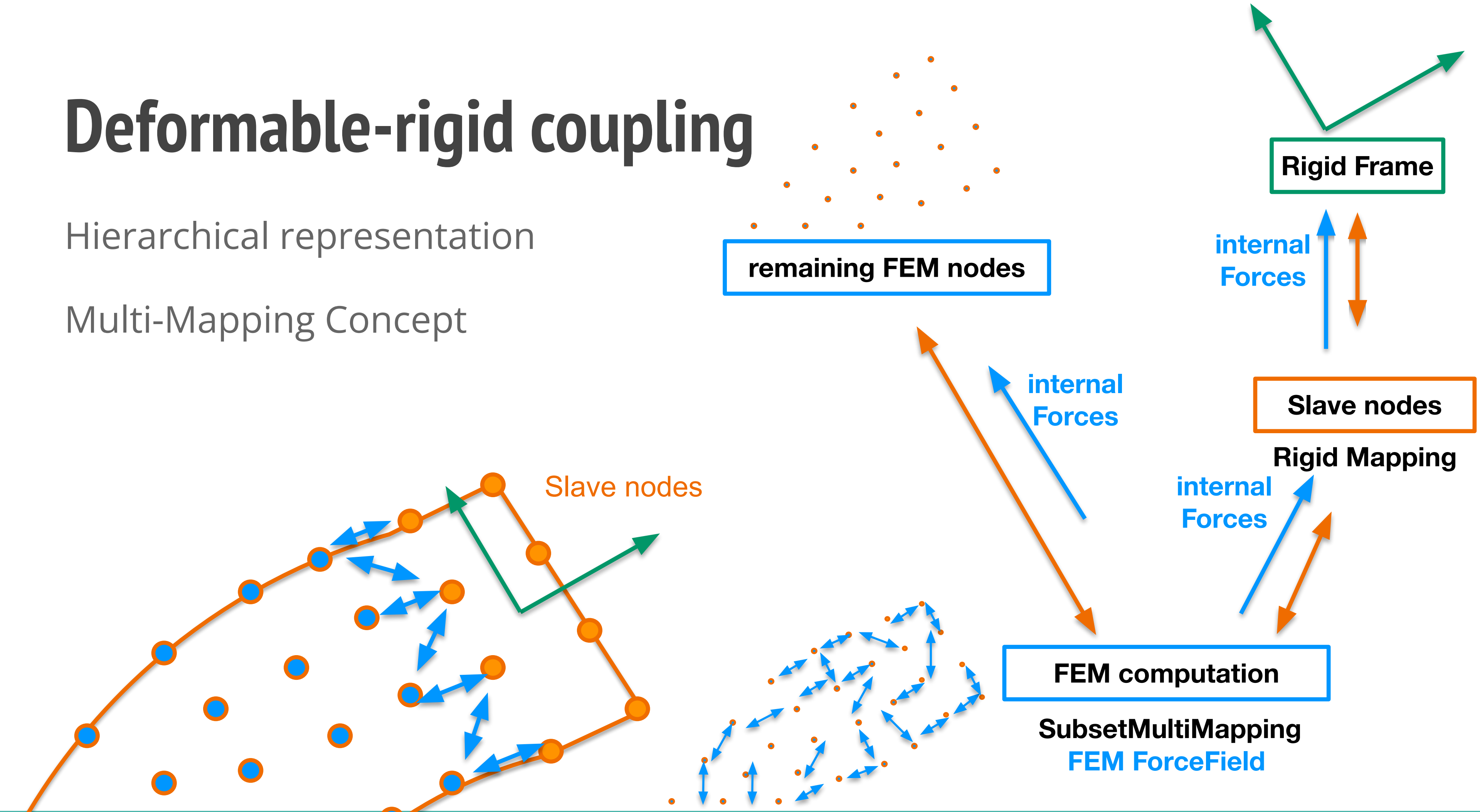


# Deformable-rigid coupling

Hierarchical representation

Multi-Mapping Concept

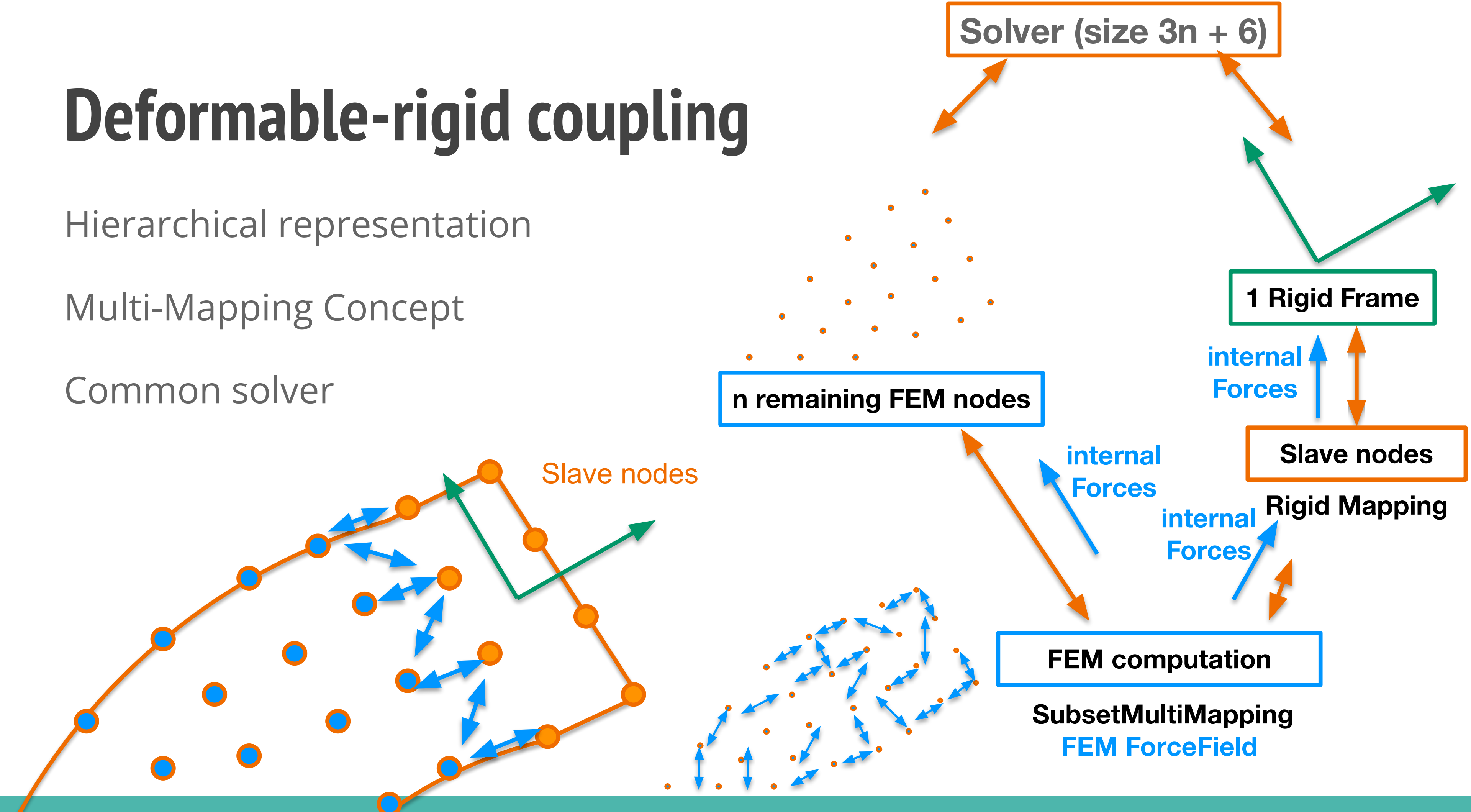


# Deformable-rigid coupling

Hierarchical representation

Multi-Mapping Concept

Common solver

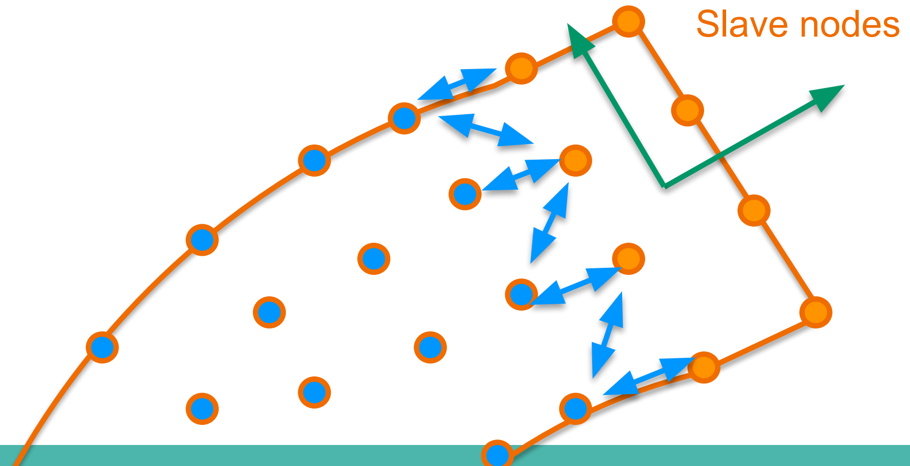


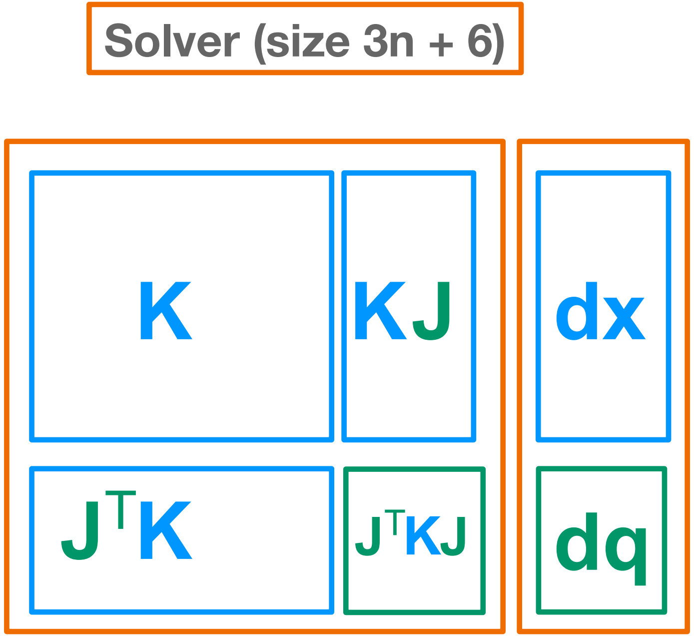
# Deformable-rigid coupling

Hierarchical representation

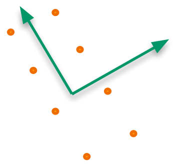
Multi-Mapping Concept

Common solver




with J jacobian of the Rigid Mapping



# Session 1

**10:00 am to 11:00 am**: Tripod Tutorial (part1)

# Step1: Mesh loader, visual model, and DOFs

We are introducing:

- Basic mechanical modeling
- Time integration and a mechanical object to the scene
- Visual model

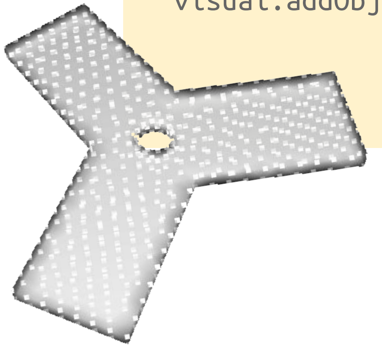
# Step1: Mesh loader, visual model, and DOFs

We are introducing:

- Basic mechanical modeling
- Time integration and a mechanical object to the scene
- Visual model

```
def createScene(rootNode):
  # Tool to load the mesh file of the silicone piece.
  It will be used for both the mechanical and the
  visual models.

  # Visual object
  visual = rootNode.addChild("Visual")
  visual.addObject("MeshSTLLoader", name="loader2",
                   filename="data/mesh/tripod_mid.stl")
  visual.addObject("OglModel", name="renderer",
                          src='@../loader2',
                          color=[1.0, 1.0, 1.0, 0.5])
```

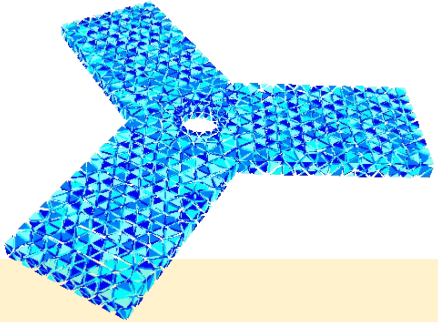


# Step2: Mechanical model

Introducing elastic material modelling:

- Volumetric mesh
- Solver
- Force field

# Step2: Mechanical model



Introducing elastic material modelling:

- Volumetric mesh
- Solver
- Force field

What's new in the scene:

```
# Tetrahedric mesh
body.addObject('GIDMeshLoader', name='loader',
                 filename="data/mesh/tripod_high.gidmsh")
body.addObject('TetrahedronSetTopologyContainer',
                 src='@loader', name='container')
body.addObject("MechanicalObject", name="dofs",
                 position=elasticbody.loader.position)
body.addObject("UniformMass", totalMass=0.032)
# Solver components
body.addObject("EulerImplicitSolver")
body.addObject("SparseLDLSolver")
# ForceField components
body.addObject("TetrahedronFEMForceField",
                youngModulus=800, poissonRatio=0.45)
```

# Step3: Fixed constraint

In this step:

- Add a box to select points
- Fix the select points with a constraint

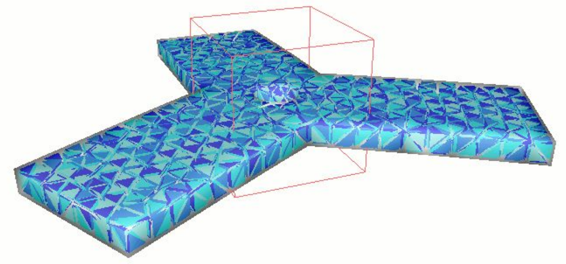
# Step3: Fixed constraint

In this step:

- Add a box to select points
- Fix the select points with a constraint

What's new in the scene:

```
# Instanciating the FixingBox prefab into the graph,
constraining the mechanical object of the ElasticBody.
fix = FixingBox(rootNode, body.ElasticMaterialObject,
                translation=[0.0, 0.0, 0.0],
                scale=[30., 30., 30.])
# Changing the property of the Box ROI so that the
constraint area appears on screen.
fix.boxroi.drawBoxes = True
```



# Prefabs: ServoMotor

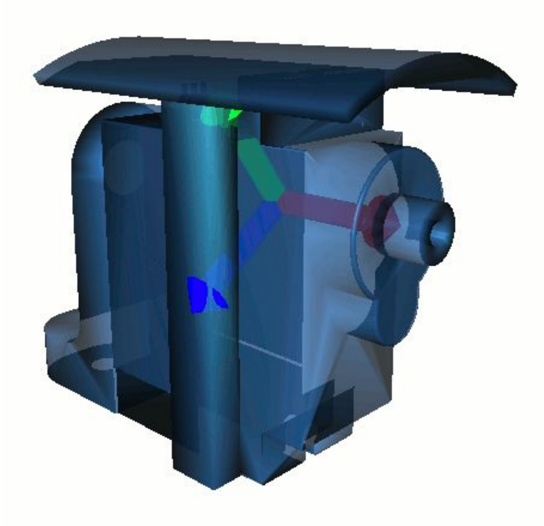
This prefab is implementing a S90 servo motor.

Call to prefab:

```
from s90servo import ServoMotor
def createScene(rootNode):
      ServoMotor(rootNode)
```

Run result:

```
runSofa details/s90servo.py
```



# Prefabs: ActuatedArm

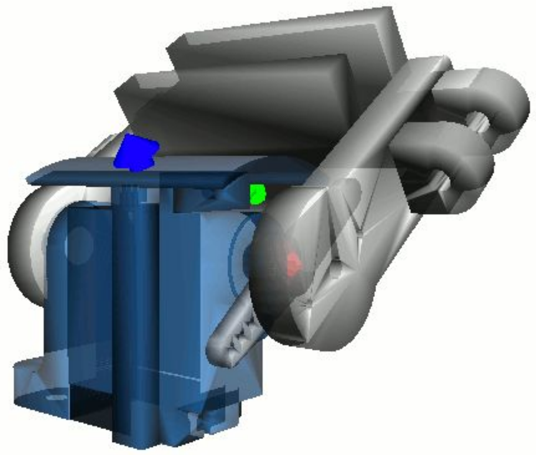
This prefab is implementing a S90 servo motor with the tripod actuation arm.

Call to prefab:

```
from actuatedarm import ActuatedArm
def createScene(rootNode):
      ActuatedArm(rootNode)
```

Run result:

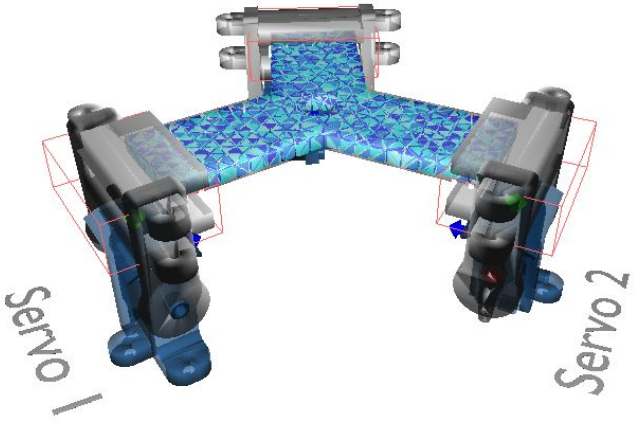
```
runSofa details/actuatedarm.py
```



# Step4: Tripod assembly

Define the tripod prefab in three steps:

1. Add the ActuatedArm prefab
2. Rigidify part to attach to the arms
3. Constraint the deformable object to follow the arms



# Step4-1: Add actuated arms

First step is to:

- Add the three actuated arms
- Correctly place them

# Step4-2: Rigidification

Second step is:

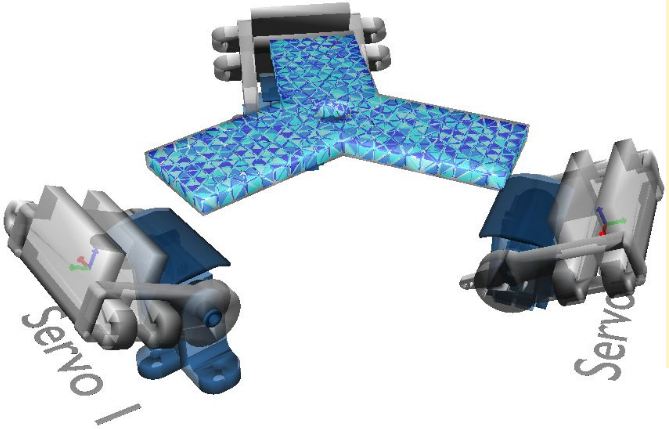
- Deformable part should be attached at each extremity
- So each extremity is rigidified

# Step4-1: Add actuated arms

First step is to:

- Add the three actuated arms
- Correctly place them

Arms not attached to the deformable part yet




What's new in the scene:

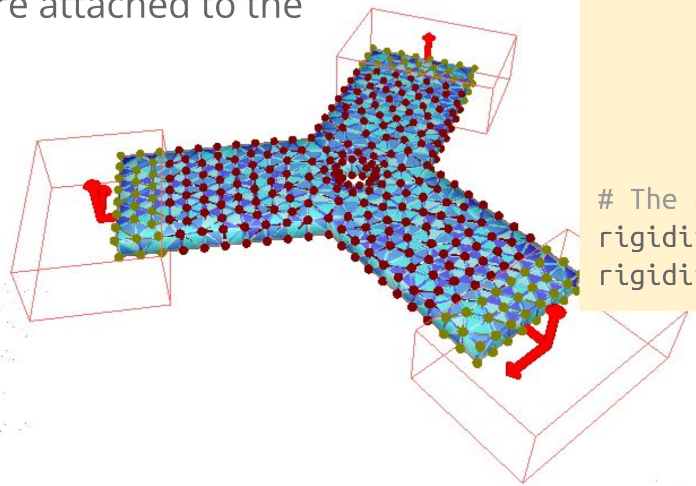
```
from actuatedarm import ActuatedArm
…
for i in range(0, nummotors):
    name = "ActuatedArm"+str(i)
    … compute correct translation and rotation …
    ActuatedArm(self.node, name=name,
                translation=translation,
                eulerRotation=eulerRotation)
    # Add limits to angle that correspond to
    limits on real robot
    arm.ServoMotor.minAngle = -2.0225
    arm.ServoMotor.maxAngle = -0.0255
```

# Step4-2: Rigidification

Second step is:

- Deformable part should be attached at each extremity
- So each extremity is rigidified

Now three frames are attached to the deformable part



What's new in the scene:

```
from stlib.physics.mixedmaterial import Rigidify
…

# Rigidify the deformable part in each extremity
rigidified = Rigidify(self.node,
                      deformableObject,
                      groupIndices=groupIndices,
                      frames=frames,
                      name="RigidifiedStructure")

# The prefab gives access to two nodes
rigidifiedstruct.DeformableParts…
rigidifiedstruct.RigidParts…
```

# Step4-3: Attach parts

Last step of assembly:

- Link rigidified parts with actuated arms
- Use springs to attached the frames

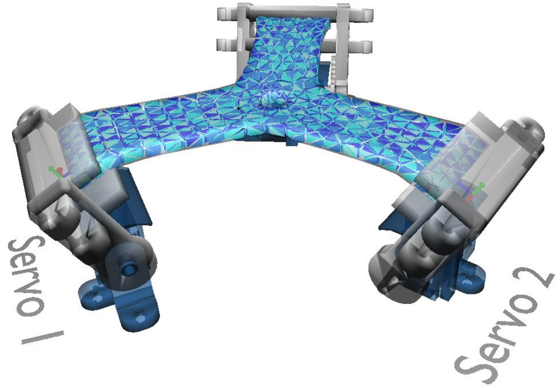
# Step4-3: Attach parts

Last step of assembly:

- Link rigidified parts with actuated arms
- Use springs to attached the frames

What's new in the scene:

```
# Attach arms
rigidParts.addObject('SubsetMultiMapping',
                     input=[self.actuatedarms[0].ServoMotor…,
                            self.actuatedarms[1].ServoMotor…,
                            self.actuatedarms[2].ServoMotor…]
                     output="@./", indexPairs=[[0,1,1,1,2,1,3,0])
```



# Prefabs: Tripod

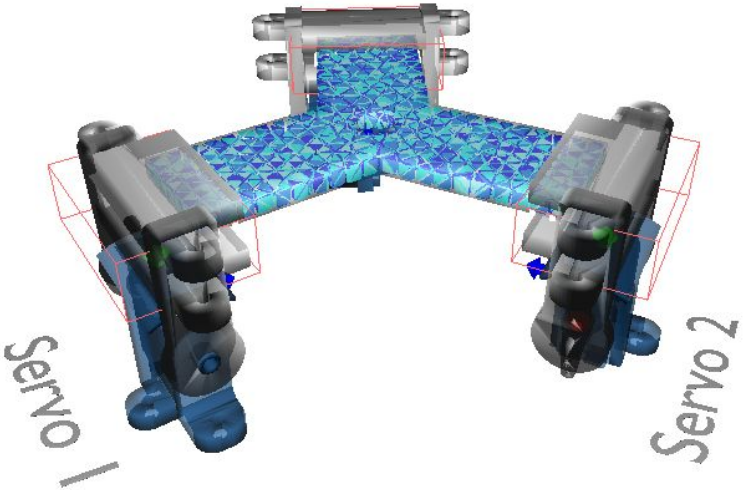
This prefab is implementing the tripod, with three S90 servo motors and actuation arm.

Call to prefab:

```
from tripod import Tripod
def createScene(rootNode):
      Tripod(rootNode)
```

Run result:

```
runSofa details/tripod.py
```



# Step5: Controller

Here you will learn how to:

- Add a controller
- The controller will connect user actions to the simulated behaviour
- We will animate the tripod to put it in the right position

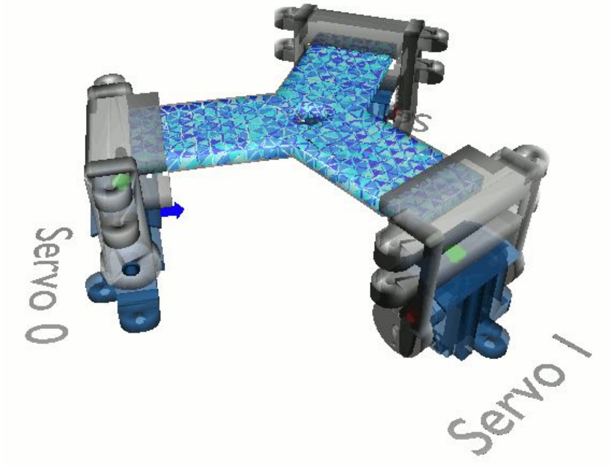
# Step5: Controller

Here you will learn how to:

- Add a controller
- The controller will connect user actions to the simulated behaviour
- We will animate the tripod to put it in the right position

What's new in the scene:

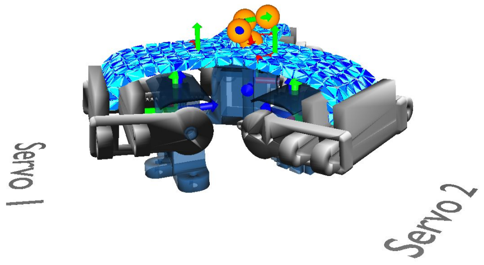
```
from tripodcontroller import TripodController
…
tripod = Tripod(model)
TripodController(rootNode, tripod.actuatedarms)
```



# Plug the robot

# 11:00 - 12:30: Session 2

- 11:00am: Presentation of the SOFA community and consortium
- 11:15am: Tripod Tutorial (part 2) :
    - Inverse modeling
    - Maze motion planning
    - Test on the digital twin
    - Test on the robot (for people on site)
- 12:15am: Conclusion and ongoing work




Todo : show what we will have at the end of the tutorial ?

# Session 2

Presentation of the SOFA community

# Session 2

Tripod Tutorial (part 2)

# Inverse Kinematics

**Quick Reminder**

Time-stepping:

$$Ma_{i+1} = f(x_{i+1}, v_{i+1}) + f_{ext}$$
$$v_{i+1} = v_i + ha_{i+1}$$
$$x_{i+1} = x_i + hv_{i+1}$$

Internal forces linearization :
(at each time step)

$$f(x_{i+1}, v_{i+1}) = f(x_i, v_i) + Kdx + Ddv$$

Matrix system to solve:

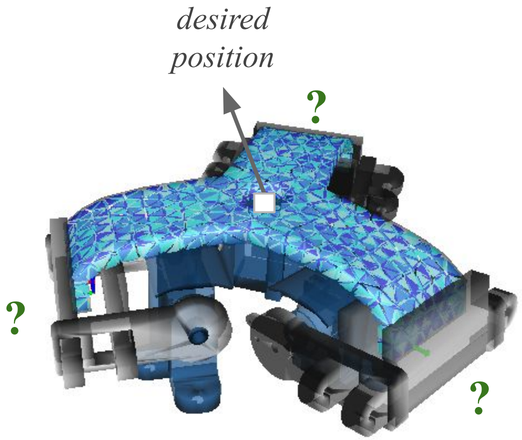
$$\underbrace{(M - hD - h^2K)}_{A}\, dv = \underbrace{hf_{ext} + hf(x_t, v_t) + h^2Kv_t}_{b}$$

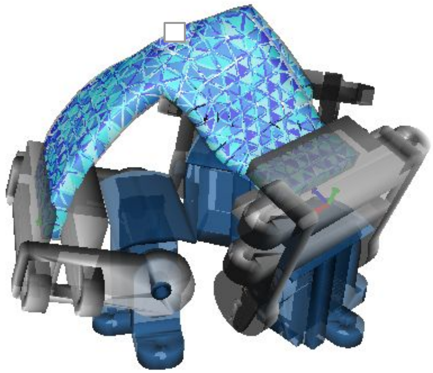
$$\underbrace{-K}_{A}\, dx = \underbrace{f(x_{i-1}) + f_{ext}}_{b} \quad \text{(quasi static)}$$

# Inverse Kinematics

Problem statement:

- Control the end effector position and orientation
- By finding the right angle for each actuated arm



# Inverse Kinematics

**Quick Reminder**

For actuator and contact we use **Lagrange multipliers**:

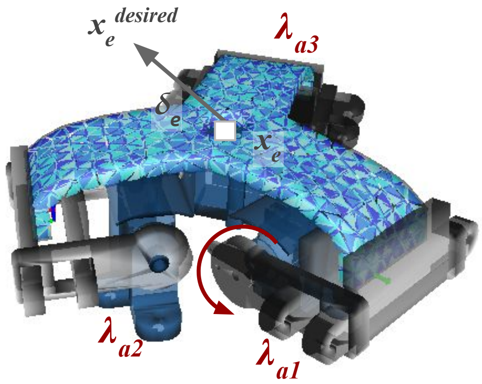
$$\begin{pmatrix} A & H^T \\ H & 0 \end{pmatrix} \begin{pmatrix} dx \\ -\lambda \end{pmatrix} = \begin{pmatrix} b \\ \delta \end{pmatrix}$$

Constraint Jacobian: direction of the constraint forces

Lagrange multiplier: constraint effort

Shift, volume growth...

# Inverse Kinematics




$$\begin{pmatrix} A & H_e^T & H_a^T \\ H_e & 0 & 0 \\ H_a & 0 & 0 \end{pmatrix} \begin{pmatrix} dx \\ -\lambda_e \\ -\lambda_a \end{pmatrix} = \begin{pmatrix} b \\ \delta_e \\ \delta_a \end{pmatrix}$$

Optimization in motion space: computationally expensive

➜ Projection in space of actuation variables using **Schur complement**: $W_{jk} = H_j A^{-1} H_k^T$, with $j, k \in \{e,a\}$

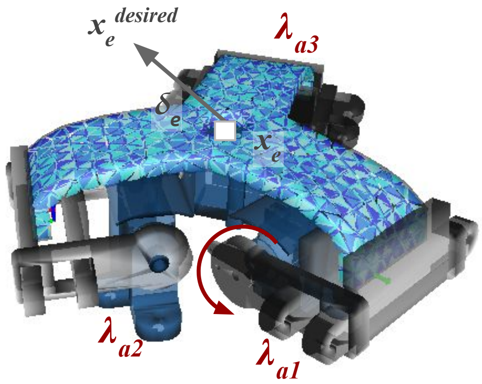
➜ $W_{jk}$ : mechanical coupling between effector points and actuators.

$$\delta_e = W_{ea} \lambda_a + \delta_e^{free}$$
$$\delta_a = W_{aa} \lambda_a + \delta_a^{free}$$

with $\delta^{free} = H_e dx^{free} + \delta(x_i)$

$dx^{free} = A^{-1} b$

# Inverse Kinematics




$$\begin{pmatrix} A & H_e^T & H_a^T \\ H_e & 0 & 0 \\ H_a & 0 & 0 \end{pmatrix} \begin{pmatrix} dx \\ -\lambda_e \\ -\lambda_a \end{pmatrix} = \begin{pmatrix} b \\ \delta_e \\ \delta_a \end{pmatrix}$$

Optimization in motion space: computationally expensive

➜ Projection in space of actuation variables using Schur complement: $W_{jk} = H_j A^{-1} H_k^T$, with $j, k \in \{e,a\}$

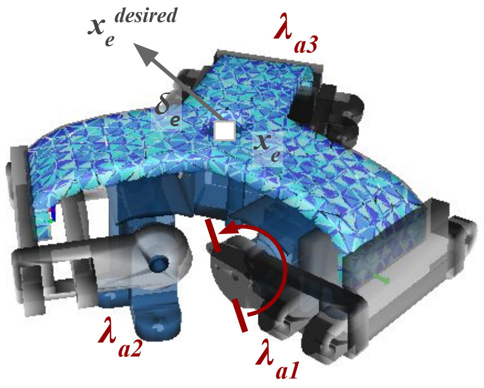
➜ $W_{jk}$ : mechanical coupling between effector points and actuators.

$$\delta_e = W_{ea} \lambda_a + \delta_e^{free}$$
$$\delta_a = W_{aa} \lambda_a + \delta_a^{free}$$

with $\delta^{free} = H_e dx^{free} + \delta(x_i)$

$dx^{free} = A^{-1} b$

# Inverse Kinematics




Formulation of Quadratic Program (QP) with linear constraints:

$$\min_{\lambda_a} \ ||\delta_e = W_{ea}\lambda_a + \delta_e^{free}||^2$$

$$\text{s.t :} \ \ (1) \ \ \delta_{max} \geq \delta_a = W_{aa}\lambda_a + \delta_a^{free} \geq \delta_{min}$$

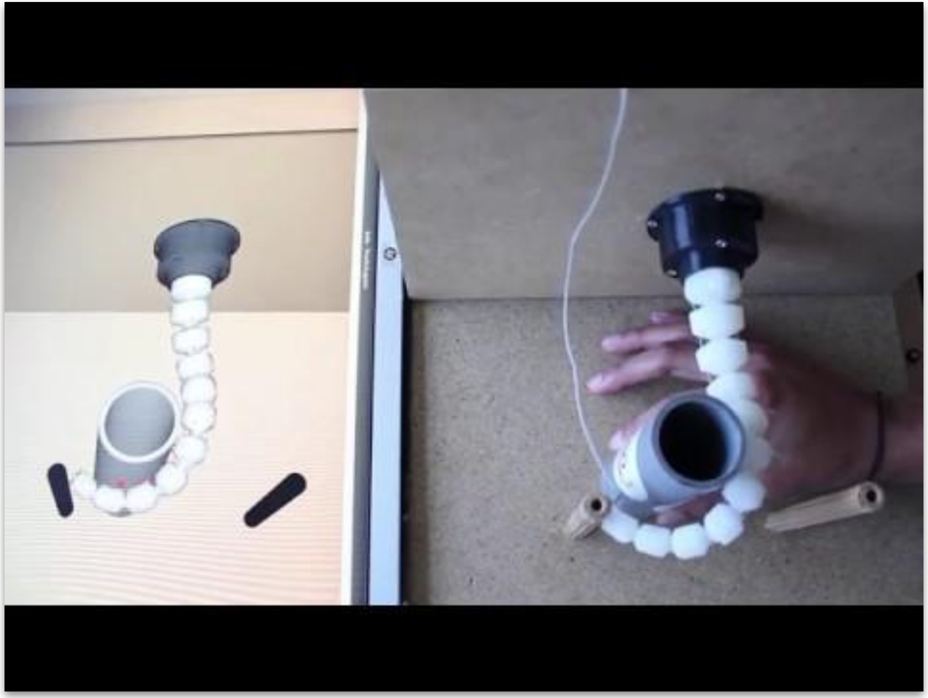
(1) Constraints on actuators (e.g limit on cable displacement)

$$dx = A^{-1}H_a^T\lambda_a + dx^{free}$$
$$x_{i+1} = x_i + dx$$

# Inverse Kinematics with Contacts

- Signorini's condition for contact
- QP with linear complementarity constraints
- Specific solver
  E. Coevoet - RA-Letter 2017

New actuation that moves the trunk forward and backward



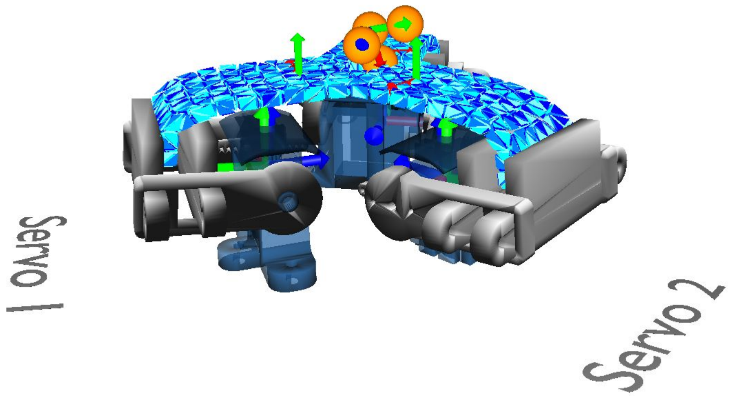
# Step8: Inverse model

In this step we solve the inverse kinematics:

- add effector position
- add effector target
- add joint actuator (to optimize angle)
- add inverse solver

Run examples: Tripod

2 possibilities :
- Control the 3 absolute positions of the effector (x, y, z)
- Control angle x and z and position y



# Maze orientation planning

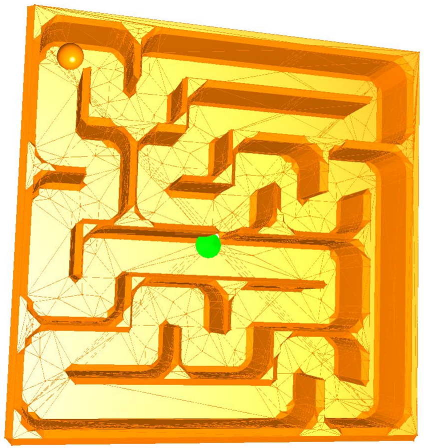
run Maze.py

Create trajectory using control points over time

open mazeplanning.json

Add new points…. And to ctrl+r (reload)
Verify in simulation that it is working

Tips: To make the trajectory work well on the robot, try to emphasize the movements. Sometimes the ball rolls better in the simulation than in reality



# Control with a digital twin

1. run step8-maze.py
2. press Ctrl+a and then Ctrl+i

- Is the desired orientation applied ?
- Can we control the translations of the maze ? which one ? why ?
- In MazeController.py, change: working_y  = 40 (this is the working height of the maze in the planning). Redo step 1 and step 2. What do you observe ? How would you explain ?

# Control the real robot

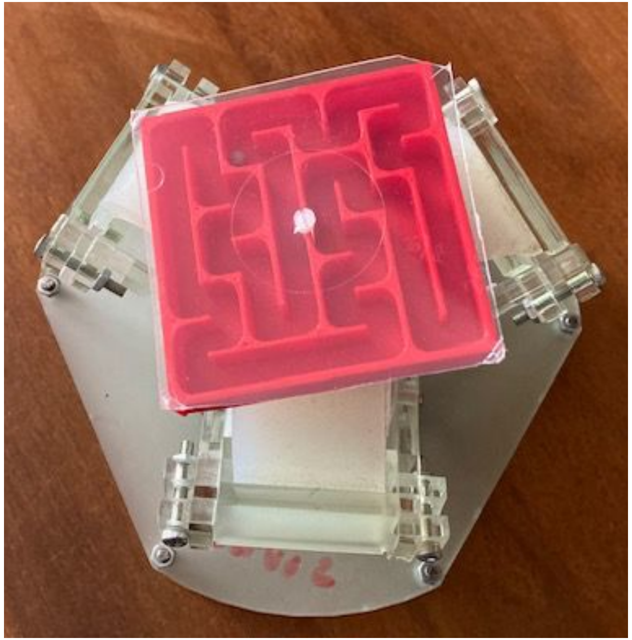
Plug the robot and place the maze

Run step8-maze.py

press Ctrl+a then Ctrl+b then Ctrl+i

What difference do you observe between simulation and reality ? Why ?

What do you propose to correct the error and better control the small ball inside the maze ?



# Thanks!