ModelOrderReduction Documentation

Release 1.0

Defrost Team

Mar 28, 2024

CONTENTS

CHAPTER

ONE

INSTALL

1.1 Dependencies

Model order reduction dependencies required and optional and what they are used for.

REQUIRED

SOFA

SOFA itself

This work is a plugin of SOFA which a simulation software. For the moment we haven't got any pre-made SOFA version with our work so the first thing you will need to do is compile SOFA

Sofa Launcher

We use a tool of SOFA named **sofa-launcher** allowing us to gain a lot of calculation time thanks to parallel execution of multiple SOFA scene.

STLIB

Plugin easing the way to write SOFA scene in python. We use some utilities of this plugin to reduce our model, especially the `stlib.scene.Wrapper` feature.

PYTHON

Python 3.X

python3 version

Cheetah

Cheetah is needed in order to use the **sofa-launcher** of SOFA.

yaml

python3 version

OPTIONAL

SoftRobot

Plugin easing the way to write SOFA scene in python. We use some utilities of this plugin to reduce our model, especially the constraints component feature.

PyQt5

We use pyqt5 for our interface

Jupyter

To learn how to reduce your own model we have done a tutorial which will make you learn step by step the process. For this interactive tutorial we use a python notebook.

1.2 Setup & Get Sarted

SOFA setup

You can either build it from sources:

Or download the binaries:

ModelOrderReduction setup

You can either build it from the source as explained here with SOFA. Or take the binaries generated here and link them to your SOFA build/binaries.

Ubuntu

Python install

minimal

```
sudo apt-get install python-cheetah python-yaml
```

all

```
sudo apt-get install python-cheetah python-yaml python-pyqt5 notebook
```

PythonPath

Then don't forget to add into your pythonPath the sofa launcher. To do that in a definitive way add this line at the end of your shell configuration file (usually *.bashrc*)

```
export PYTHONPATH=$PYTHONPATH:/PathToYourSofaSrcFolder/tools/sofa-launcher
```

Windows

Mac

1.2.1 Try some exemples

To confirm all the previous steps and verify that the plugin is working properly you can launch the *test_component.py* SOFA scene situated in:

```
/ModelOrderReduction/tools
```

This example show that after the reduction of a model (here the 2 examples *Diamond Robot*, *Starfish robot*), you can re-use it easily as a python object with different arguments allowing positionning of the model in the SOFA scene.

CHAPTER

TWO

TUTORIALS

2.1 Reduction Process Tutoriel

Note: The following tutorial comes from a python-notebook. If you want to make the tutorial interactively go directly to:

`/ModelOrderReduction/tools`

then, if you have installed jupyter like explained in the requirement, open a terminal there and launch a session:

`jupyter notebook`

It will open in your web-Browser a tab displaying the current files in the directory. Normally you should have one called **modelOrderReduction.ipynb**

You can click on it and follow the tutorial

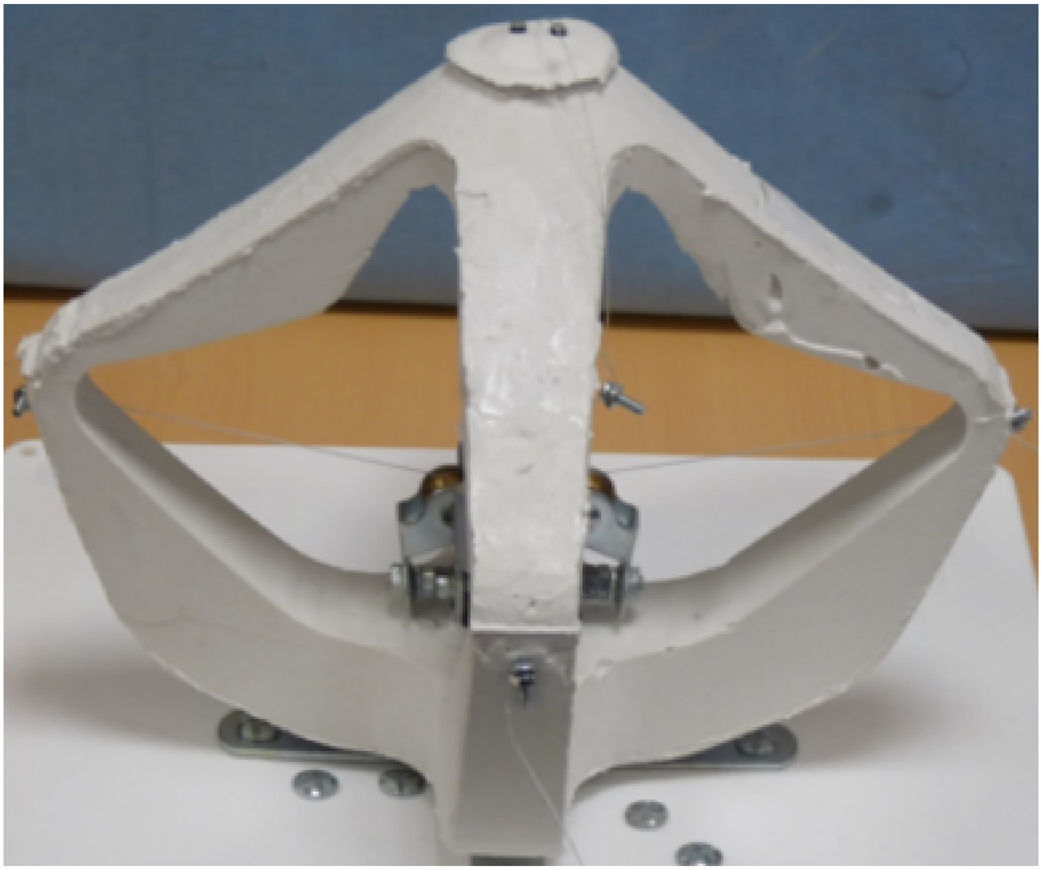
2.1.1 Model Order Reduction Example

Introduction

In this python notebook exemple we will see with 2 real examples how to reduce a model from one of your sofa scene thanks to the **Model Order Reduction** plugin done by the INRIA research team **Defrost**.

the two examples will be :

- **A cable-driven silicone robot** (*paper link : C. Duriez, ICRA, 2013*).



- **A pneumatic Soft Robot** (*paper link : Multigait soft Robot R.F. Shepherd et al, PNAS, 2011*).

After these expample presentation we can now proceed to the reduction. First we have to prepare it by setting a bunch of parameters while explaining there purpose (here the parameters will be set twice, one for the diamond and one for the starfish so you will be able to switch easily between each example)

User Paramters

Before defining the reduction parameters, here are some "import" commands that will be useful for this python notebook:

```
# Import
import os
import sys

sys.path.append(os.getcwd()+'/../python')

# MOR IMPORT
from mor.gui import utility
from mor.reduction import ReduceModel
from mor.reduction.container import ObjToAnimate
```

1. Paths to the SOFA scene, mesh and outputs:

- The scene you want to work on
- The folder containing its mesh
- The folder where you want the results to be put in

```
# Important path
from PyQt4 import QtCore, QtGui
app = QtGui.QApplication(sys.argv)

originalScene = utility.openFileName('Select the SOFA scene you want to reduce')
meshes = utility.openFilesNames('Select the meshes & visual of your scene')
outputDir = utility.openDirName('Select the directory that will contain all the results')

# if you haven't install PyQt the previous function won't work
# As an alternative you can enter the absolute path to the conresponding files directly:
# originalScene = /PathToMy/Original/Scene
```

2. The different reduction parameters

nodesToReduce

- *ie : list containing the SOFA path from the rootnode to the model you want to reduce

```
nodesToReduce_DIAMOND = ['/modelNode']
nodesToReduce_STARFISH =['/model']
```

listObjToAnimate

Contain a list of object from the class `ObjToAnimate`.

A ObjToAnimate will define an object to "animate" during the shaking.

There are 3 main parameter to this object :

- location : Path to obj/node we want to animate
- animFct : the animation function we will use (here we use *defaultShaking*).
- all the argument that will be passed to the animFct we have chose

For example here we want to animate the node named "nord", but we won't specify the animFct so the default animation function will be used and be applied on the first default object it will find. The default function will need 3 additionnal parameters :

- incrPeriod (float): Period between each increment
- incr (float): Value of each increment
- rangeOfAction (float): Until which value the data will increase

nord = ObjToAnimate("nord", incr=5,incrPeriod=10,rangeOfAction=40)

```
# animation parameters

### CABLE-DRIVEN PARALLEL ROBOT PARAMETERS
nodesToReduce = ['/modelNode']
nord = ObjToAnimate("modelNode/nord", incr=5,incrPeriod=10,rangeOfAction=40)
sud = ObjToAnimate("modelNode/sud", incr=5,incrPeriod=10,rangeOfAction=40)
est = ObjToAnimate("modelNode/est", incr=5,incrPeriod=10,rangeOfAction=40)
ouest = ObjToAnimate("modelNode/ouest", incr=5,incrPeriod=10,rangeOfAction=40)
listObjToAnimate_DIAMOND = [nord,ouest,sud,est]

### MULTIGAIT SOFT ROBOT PARAMETERS
centerCavity = ObjToAnimate("model/centerCavity", incr=350,incrPeriod=2,
↪rangeOfAction=3500)
rearLeftCavity = ObjToAnimate("model/rearLeftCavity", incr=200,incrPeriod=2,
↪rangeOfAction=2000)
rearRightCavity = ObjToAnimate("model/rearRightCavity", incr=200,incrPeriod=2,
↪rangeOfAction=2000)
frontLeftCavity = ObjToAnimate("model/frontLeftCavity", incr=200,incrPeriod=2,
↪rangeOfAction=2000)
frontRightCavity = ObjToAnimate("model/frontRightCavity", incr=200,incrPeriod=2,
↪rangeOfAction=2000)
listObjToAnimate_STARFISH = [centerCavity,rearLeftCavity,rearRightCavity,frontLeftCavity,
↪frontRightCavity]
```

Modes parameters

- addRigidBodyModes (Defines if our reduce model will be able to translate along the x, y , z directions)
- tolModes (Defines the level of accuracy we want to select the reduced basis modes)

```
addRigidBodyModes_DIAMOND = [0,0,0]
addRigidBodyModes_STARFISH = [1,1,1]

tolModes = 0.001
```

- tolGIE
  - *tolerance used in the greedy algorithm selecting the reduced integration domain(RID). Values are between 0 and 0.1 . High values will lead to RIDs with very few elements, while values approaching 0 will lead to large RIDs. Typically set to 0.05.*

```
# Tolerance
tolGIE =  0.05
```

3 – Optional parameters

```
# Optionnal
verbose = False
nbrCPU = 4
packageName = 'test'
addToLib = False
```

We can now execute one of the reduction we choose with all these parameters

Execution

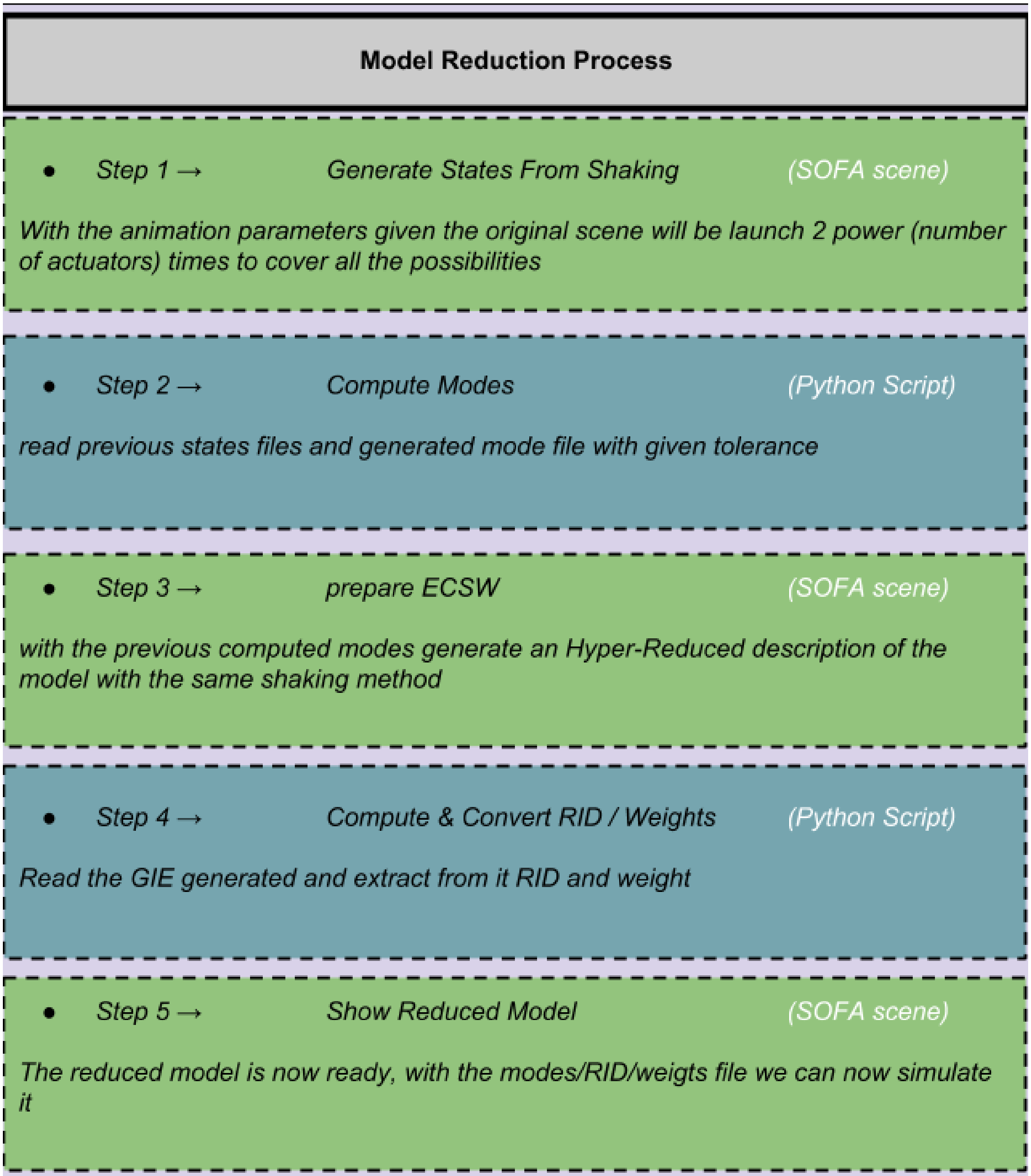
Initialization

The execution is done with an object from the class `ReduceModel`. we initialize it with all the previous argument either for the Diamond or Starfish example

```
# Initialization of our script
nodesToReduce = nodesToReduce_DIAMOND # nodesToReduce_STARFISH
listObjToAnimate = listObjToAnimate_DIAMOND # listObjToAnimate_STARFISH
addRigidBodyModes = addRigidBodyModes_DIAMOND # addRigidBodyModes_STARFISH

reduceMyModel = ReduceModel(    originalScene,
                                nodesToReduce,
                                listObjToAnimate,
                                tolModes,tolGIE,
                                outputDir,
                                packageName = packageName,
                                addToLib = addToLib,
                                verbose = verbose,
                                addRigidBodyModes = addRigidBodyModes)
```

We can finally perform the actual reduction. here is a schematic to resume the differents steps we will perform :



phase1

We modify the original scene to do the first step of MOR :

- We add animation to each actuators we want for our model
- And add a writeState componant to save the shaking resulting states

```
reduceMyModel.phase1()
```

phase2

With the previous result we combine all the generated state files into one to be able to extract from it the different mode

```
reduceMyModel.phase2()
```

```
# Plot result
with open(reduceMyModel.packageBuilder.debugDir+'Sdata.txt') as f:
    content = f.readlines()

content = [x.strip() for x in content]

data = [go.Bar(x=range(1, len(content)+1),
            y=content)]

iplot(data, filename='jupyter/basic_bar')
```

```
print("Maximum number of Modes : ")
reduceMyModel.reductionParam.nbrOfModes
```

phase3

We launch again a set of sofa scene with the sofa launcher with the same previous arguments but with a different scene

This scene take the previous one and add the model order reduction component:

- HyperReducedFEMForceField
- MechanicalMatrixMapperMOR
- ModelOrderReductionMapping and produce an Hyper Reduced description of the model

```
reduceMyModel.phase3()
```

phase4

Final step : we gather again all the results of the previous scenes into one and then compute the RID and Weigts with it. Additionnally we also compute the Active Nodes

```
reducedScene = reduceMyModel.phase4()
```

End of example you can now go test the results in the folder you have designed at the beginning of this tutorial

To go Further

Links to additional information about the plugin:

Publication in IEEE Transactions On Robotics

Plugin website

Plugin doc

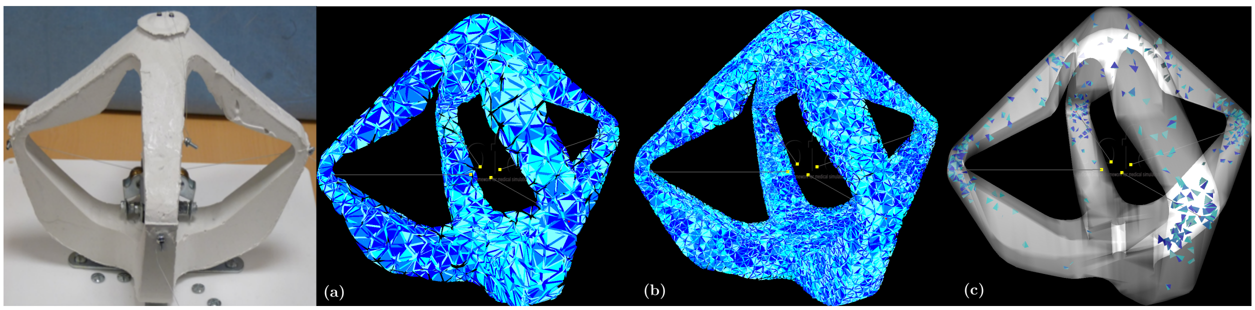
2.1.2 Model Order Reduction GUI

tutorial about the gui

CHAPTER THREE

EXAMPLES

3.1 Cable-driven Soft Robot



3.1.1 Presentation

The Cable-driven Soft Robot is a proof of concept for the DEFROST team showing control of soft robots using SOFA simulation. There are several papers which have been written using it: link. More recently it was reduced using this plugin: link.

Brief description :

The robot is entirely made of soft silicone and is actuated by four cables controlled by step motors located at its center. Pulling on the cables has the effect of lifting the effector located on top of the robot. The "game" with this robot is to control the position of the effector by pulling on the cables.

Little video of presentation showing it in action

Why reduce it :

Previously the robot was controlled through real-time finite element simulation based on a mesh of 1628 nodes and 4147 tetrahedra. That size of mesh was manageable in real-time on a standard desktop computer. The simulation made using this underlying mesh was accurate enough to control the robot, only considering the displacement of the effector point, located on the top of the robot and with a limited range on the pulling of the cable actuators.

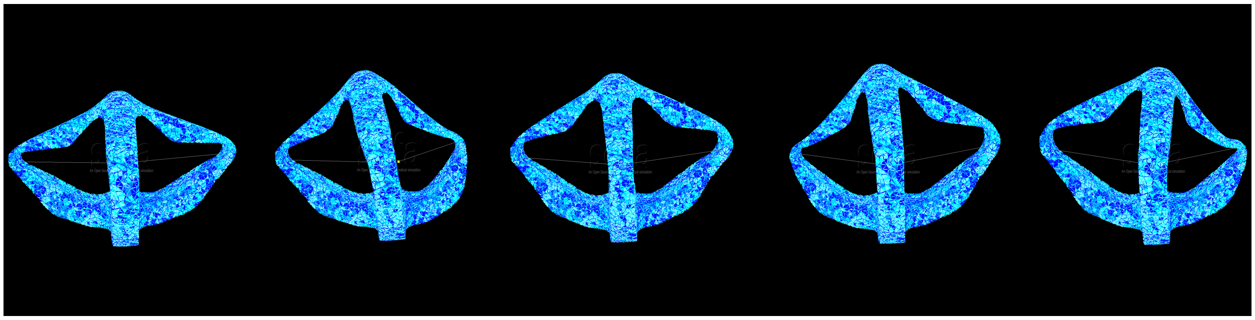
However, this does not show that the actual position of each of the four arms of the robot was accurately predicted for example. When considering an application where the robot arms may enter in contact with the environment, an accurate prediction of their position becomes relevant.

To have this accuracy we need a much more finer mesh which will demand some intensive calculations and in the process we will lose the real-time simulation of it. So here comes our plugin to resolve this issue.

3.1.2 Reduction Parameters

To reduce this robot we will use the defaultShaking(link!) function to shake it because we just need for actuators to perform simple incrementation along there working interval (here *[0 .. 40]* with an increment of *5*)

After that with a raisonnable tolerance (here *0.001*) we will select different modes, here some possible modes selected :

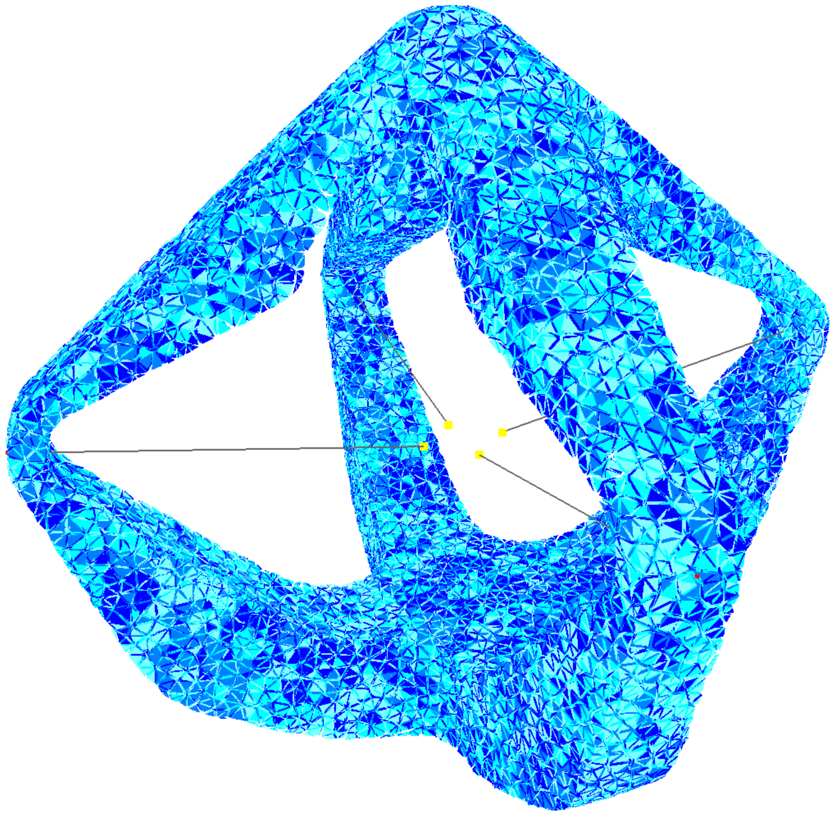


With these different parameters we will after perform the reduction like explained *here*

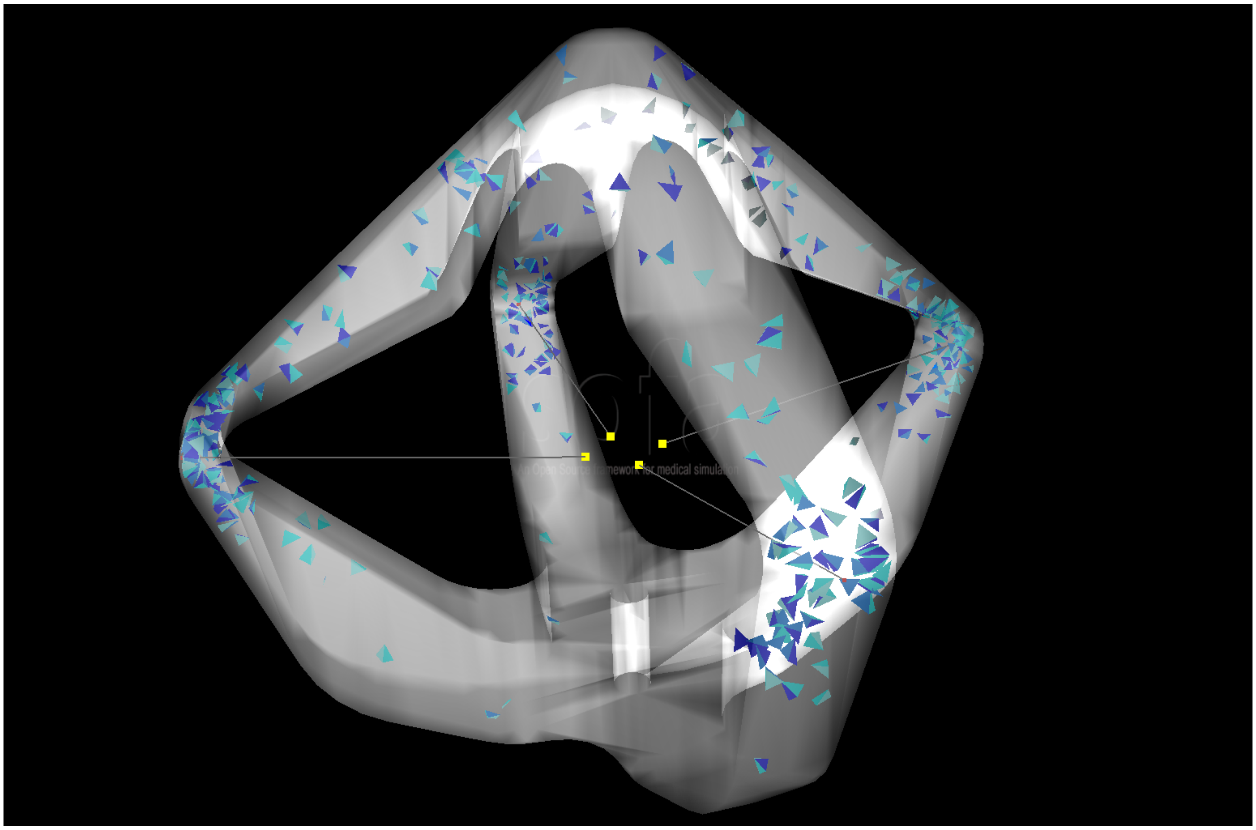
3.1.3 Results

exemple results with a fine mesh:

Before

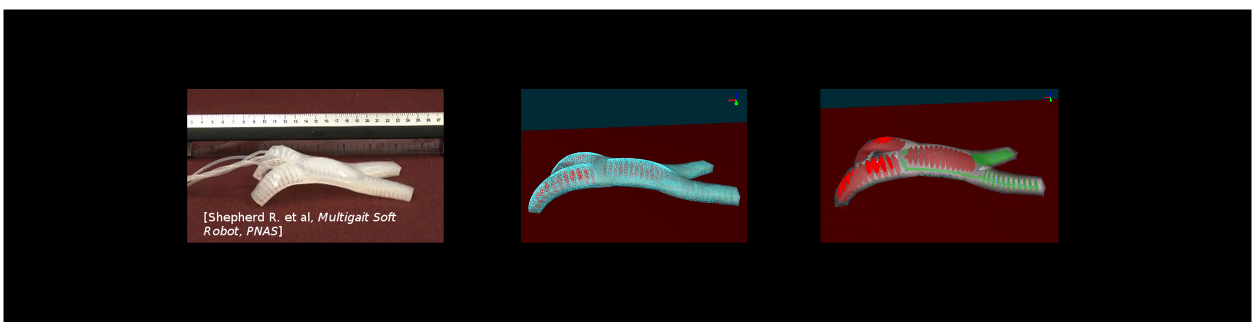


After



For more details about the results, displacmeent error comparison, test with different mesh and other, you can read the paper affiliated with this plugin[1].

3.2 Multigait Soft Robot



[1] Olivier Goury and Christian Duriez. Fast, generic, and reliable control and simulation of soft robots using model order reduction. *IEEE Transactions on Robotics*, 34(6):1565–1576, December 2018. URL: https://doi.org/10.1109/tro.2018.2861900, doi:10.1109/tro.2018.2861900.

3.2.1 Presentation

The multigait soft robot is a pneumatic robot from the work of R. Shepherd et. al[1].

Brief description :

This robot is made of two layers: one thick layer of soft silicone containing the cavities, and one stiffer and thiner layer of Polydimethylsiloxane (PDMS) that can bend easily but does not elongate. The robot is actuated by five air cavities that can be actuated independently. The effect of inflating each cavity is to create a motion of bending. Then, by actuating with various sequences each cavities, the robot can move along the floor.

Why reduce it :

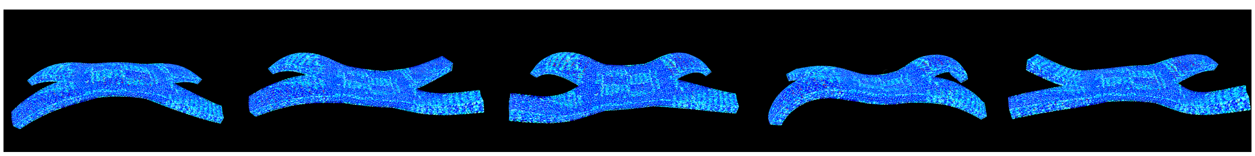
The simulation of this crawling robot has to be really precise in order to simulate properly the differents deformations and the contact with the floor has showned in the previous video.

This needs of precision results with heavy calculations when the simulation is running preventing the fluidity of it, by reducing it we will be able to resolve this issue and also show that we the reduce model can move and handle contact in comparison with the previous example *Diamond Robot* that was fixed.

3.2.2 Reduction Parameters

To reduce this robot we will use the defaultShaking(link!) function to shake it because we just need for actuators to perform simple incrementation along there working interval (here *[0 .. 2000 or 3500]* with an increment of *200 or 350*)

After that with a raisonnable tolerance (here *0.001*) we will select different modes, here some possible modes selected :



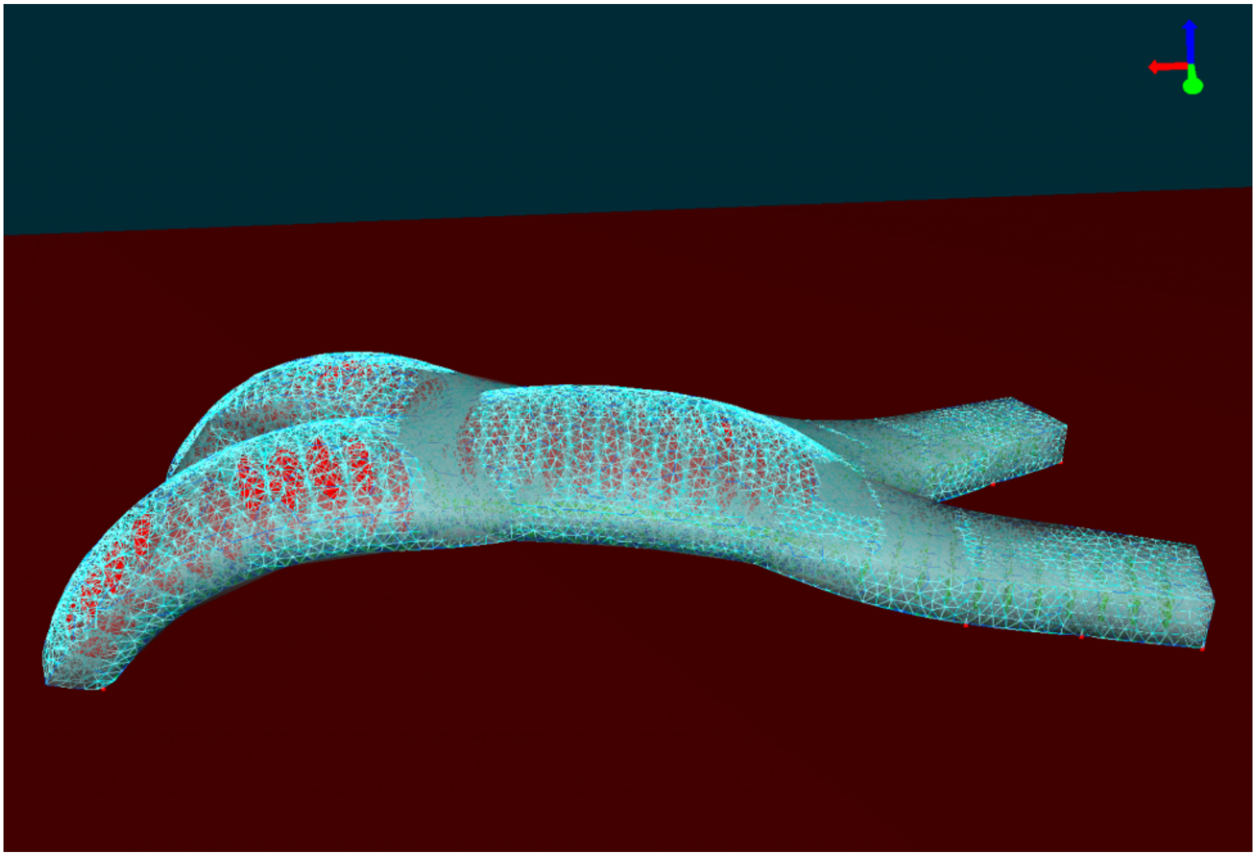
With these different parameters we will after perform the reduction like explained *here*.

3.2.3 Results

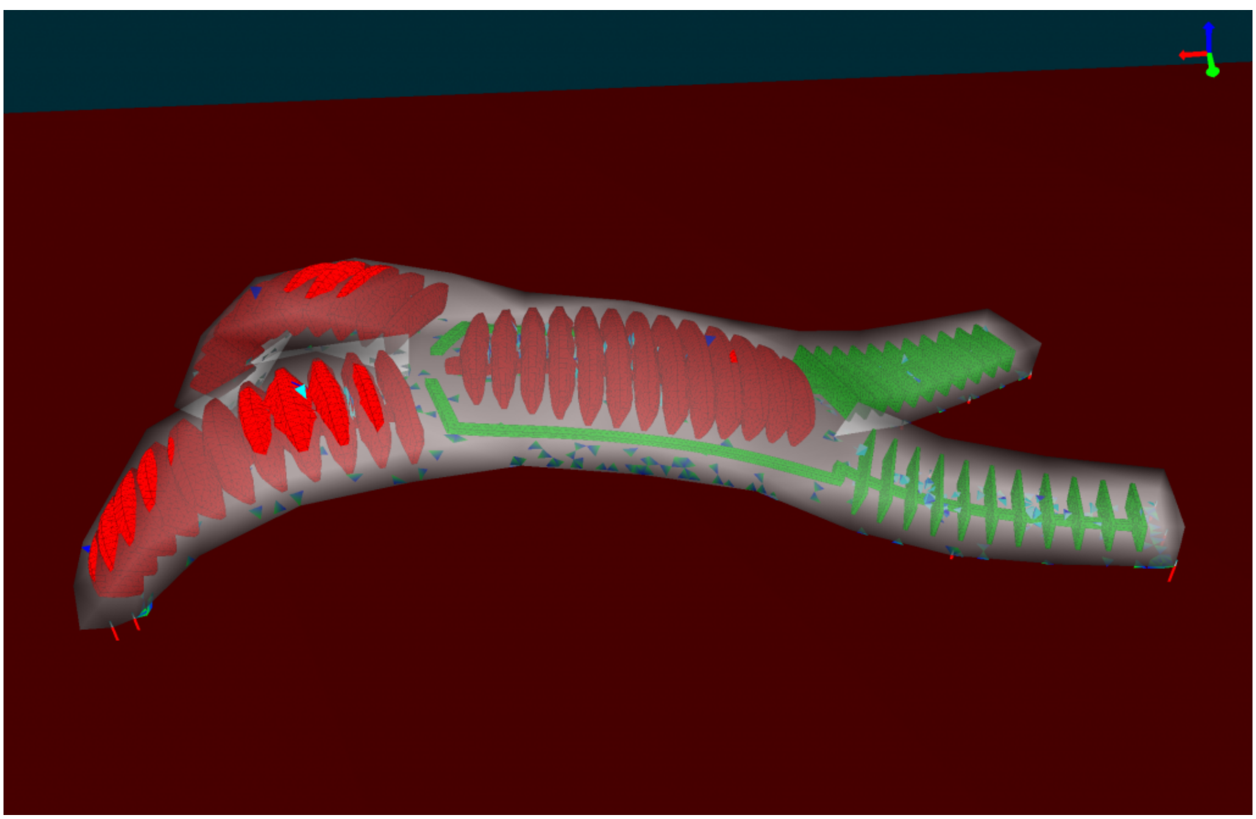
exemple results with a fine mesh:

Before

[1] Robert F. Shepherd, Filip Ilievski, Wonjae Choi, Stephen A. Morin, Adam A. Stokes, Aaron D. Mazzeo, Xin Chen, Michael Wang, and George M. Whitesides. Multigait soft robot. *Proceedings of the National Academy of Sciences*, 108(51):20400–20403, November 2011. URL: https://doi.org/10.1073/pnas.1116564108, doi:10.1073/pnas.1116564108.

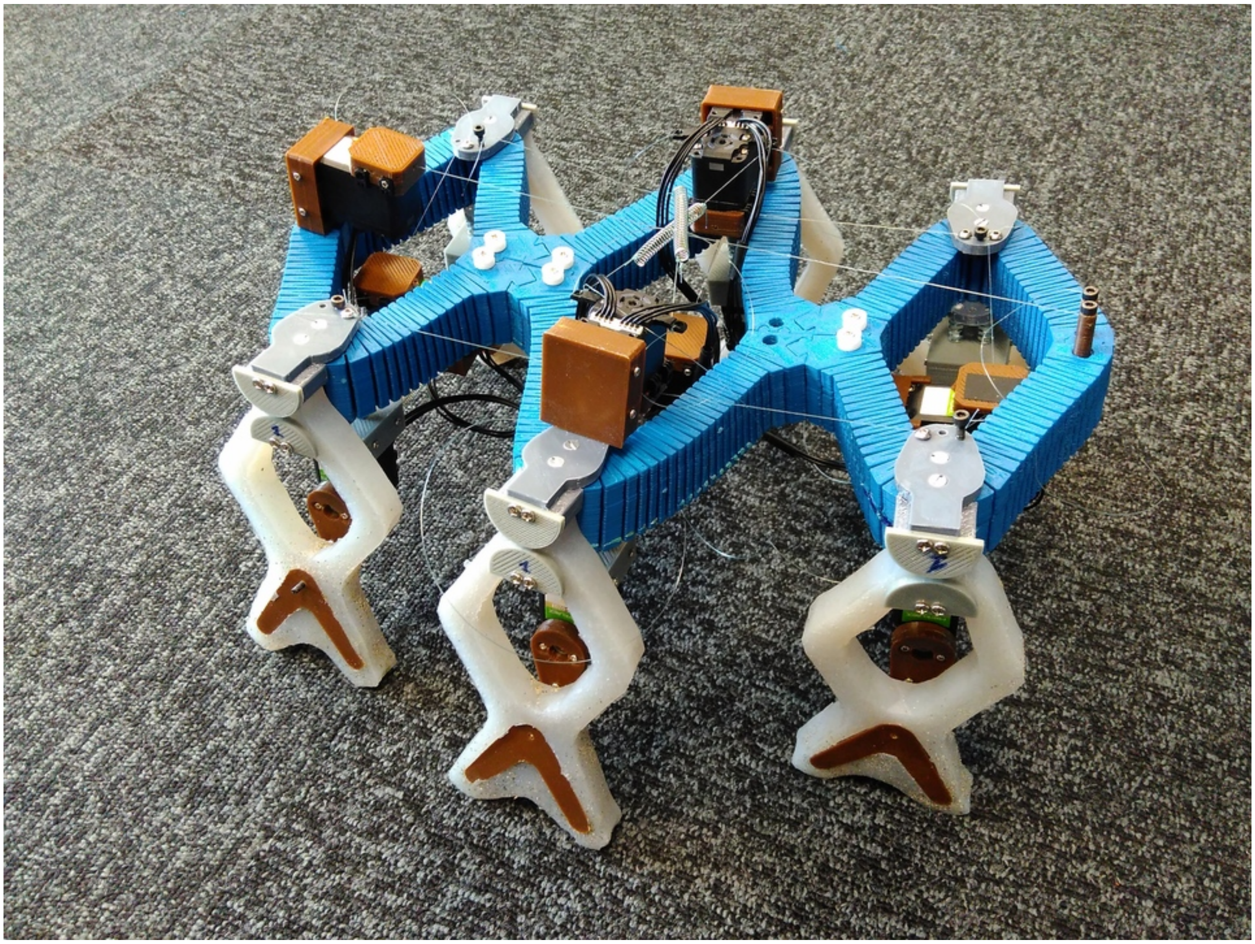


After



For more details about the results, displacement error comparison, test with different mesh and other, you can read the paper affiliated with this plugin[2].

3.3 6-legged Robot



3.3.1 Presentation

Brief description :

This robot has 6 legs actuated independently by 6 motors, which allows it to have various kind of movements.

presentation video of the simulation showing it in action:

video of the realisation based on the previous simulation:

Why reduce it :

To show that we can easily reduce parts of a soft robot and re-use it in the full robot. Here we only reduce the leg of our robot not its core.

[2] Olivier Goury and Christian Duriez. Fast, generic, and reliable control and simulation of soft robots using model order reduction. *IEEE Transactions on Robotics*, 34(6):1565–1576, December 2018. URL: https://doi.org/10.1109/tro.2018.2861900, doi:10.1109/tro.2018.2861900.

3.3.2 Reduction Parameters

To make a reduced model of one leg of this robot, we had to create a new special function to explore its workspace. To create the rotation mouvement we see on the different previous videos we rotate a point that will be followed by the model creating the rotation.

:meth:`mor.animation.defaultShaking` how it was implemented

We have only one actuator here, so our *listObjToAnimate* contains only one object:

```
ObjToAnimate("actuator","shakingSofia",'MechanicalObject',incr=0.05,incrPeriod=3,
↪rangeOfAction=6.4,dataToWorkOn="position",angle=0,rodRadius=0.7)
```

With these different parameters we will after perform the reduction like explained *here*

3.3.3 Results

With coarse mesh

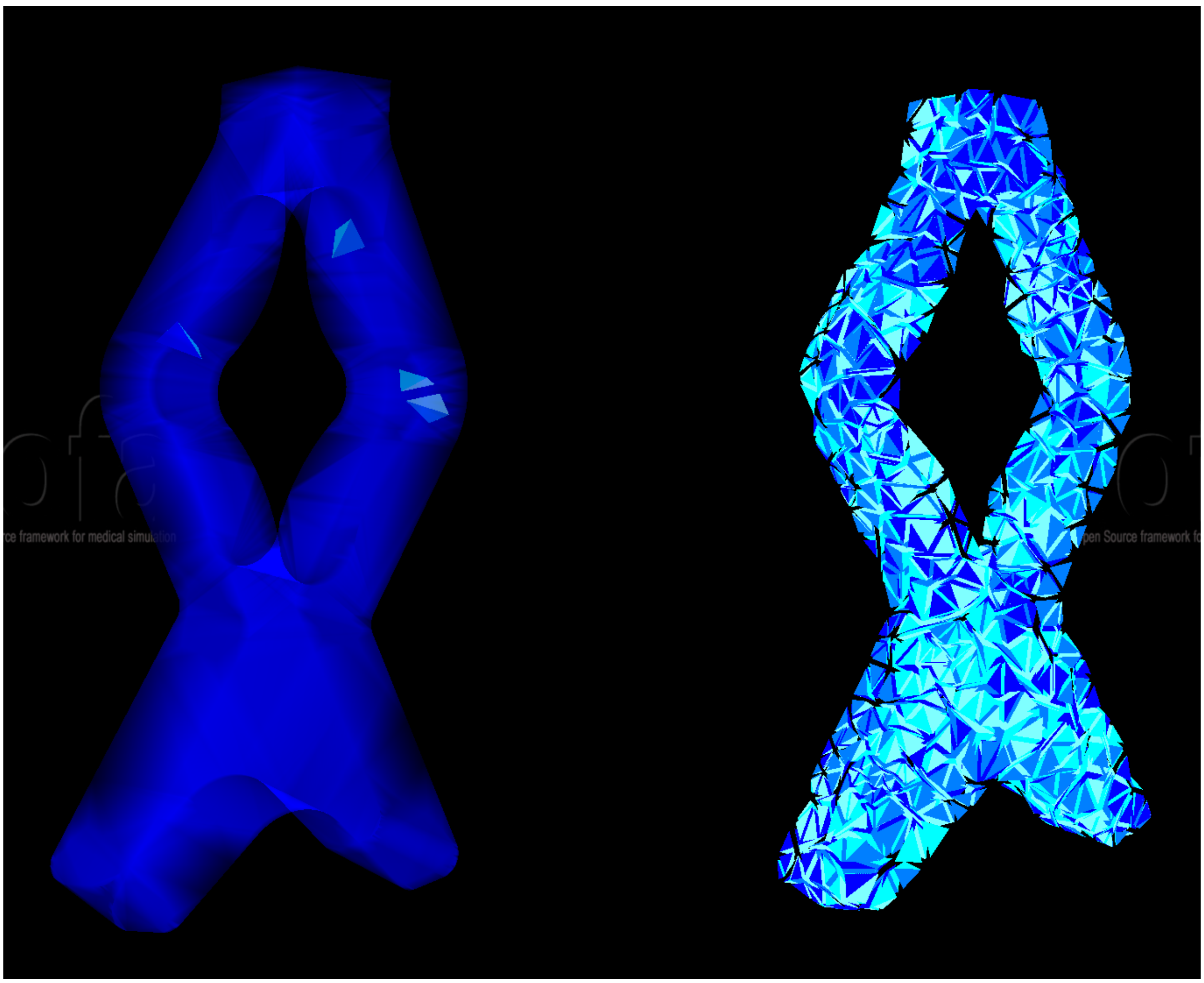


Table 1: FPS before/after reduction

not reduced	reduced
90	300

With fine mesh

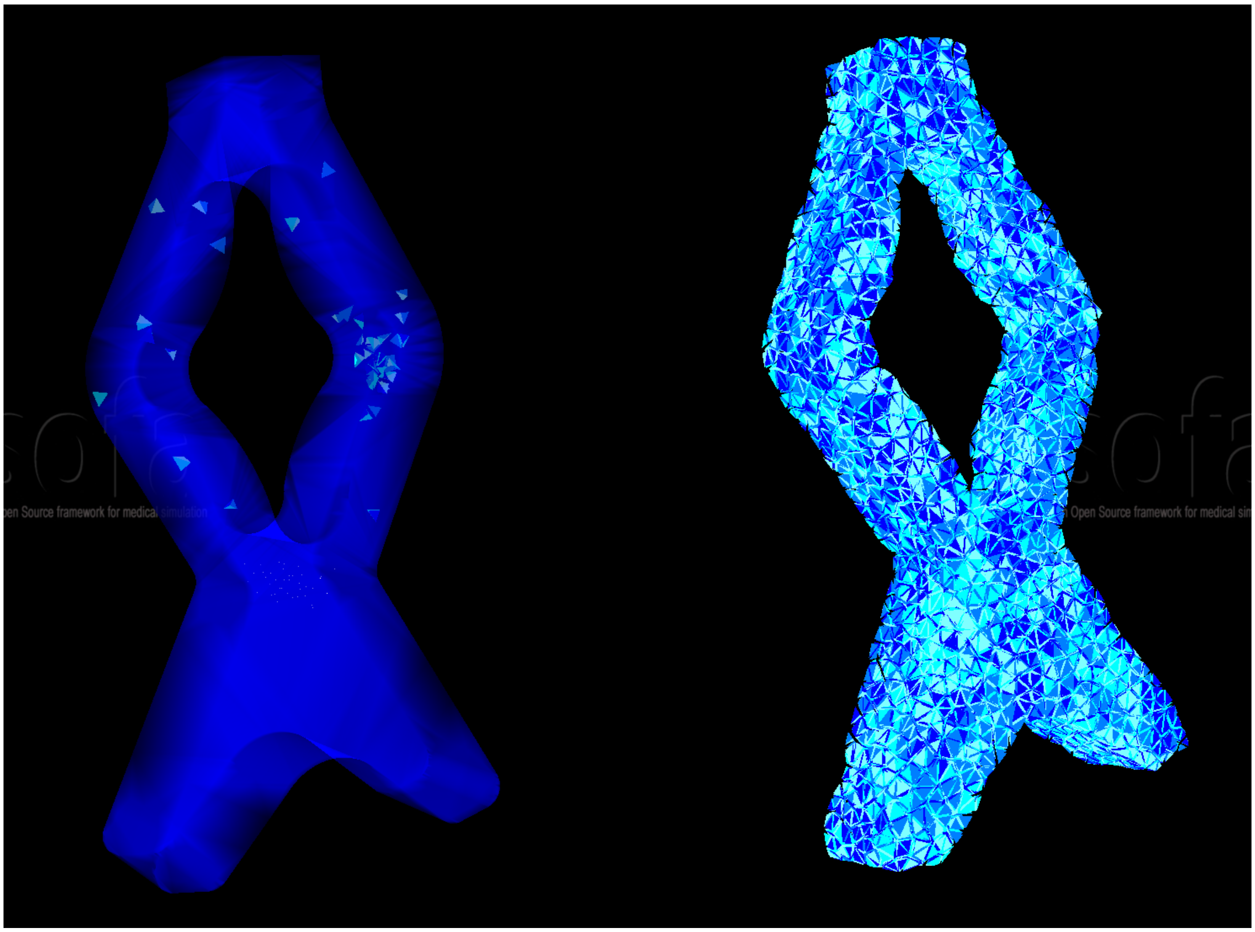


Table 2: FPS before/after reduction

not reduced	reduced
3.8	190

CHAPTER

FOUR

TOOLS

General API to do reduction

animation	**Set of predefined function to shake our model during the reduction**
utility	**Set of utility functions used during the reduction process**
wrapper	**Set of functions to modify the SOFA scene during its construction**

4.1 mor.animation

Set of predefined function to shake our model during the reduction

Each function has to have 3 mandatory arguments:

argument	type	definition
objToAnimate	`ObjToAnimate`	the obj containing all the information/arguments about the animation
dt	seconde (in float)	Time step of the Sofa scene
factor	float	Argument given by the Animation class from STLIB. It indicate where we are in the animation sequence: • 0.0 ——> beginning of sequence. • 1.0 ——> end of sequence. It is calculated as follow: `factor = (currentTime-startTime) / duration`

the animation implemented in *mor.animation* will be added to the templated scene thanks to the `splib.animation.animate`

---

mor.animation.shakingAnimations	Implemented animation functions

4.1.1 mor.animation.shakingAnimations

Implemented animation functions

Functions

defaultShaking	**Default animation function**
rotationPoint	Utility function applying rotation on a given position with some lever arm
shakingInverse	**Animation function to use with iinverse simulation**
shakingLiver	**Animation function made specifically to apply deformation on the liver scene.**
shakingSofia	**Animation function made specifically to shake the leg of the** *6-legged Robot*.
upDateValue	Utility function for default animation.

mor.animation.shakingAnimations.defaultShaking

`defaultShaking`(*objToAnimate*, *dt*, *factor*, ***param*)

Default animation function

The animation consist on *increasing* a value of a Sofa object until it reach its *maximum*

To use it the **params** parameters of `ObjToAnimate` which is a dictionnary will need 4 keys:

Keys:

argument	type	definition
dataToWorkOn	str	Name of the Sofa datafield we will work on by default it will be set to *value*
incrPeriod	float	Period between each increment
incr	float	Value of each increment
rangeOfAction	float	Until which value the data will increase

Returns
None

mor.animation.shakingAnimations.rotationPoint

`rotationPoint`(*Pos0*, *angle*, *brasLevier*)

Utility function applying rotation on a given position with some lever arm

Parameters

- **Pos0** –
- **angle** –
- **brasLevier** –

Returns
New updated position

mor.animation.shakingAnimations.shakingInverse

`shakingInverse`(*objToAnimate*, *dt*, *factor*, ***param*)

Animation function to use with iinverse simulation

mor.animation.shakingAnimations.shakingLiver

`shakingLiver`(*objToAnimate*, *dt*, *factor*, ***param*)

Animation function made specifically to apply deformation on the liver scene.

It's an example of what can be a custom shaking animation. The animation consist on taking a position in entry, rotate it, and then update it in the component.

To use it the **params** parameters of `ObjToAnimate` which is a dictionnary will need 6 keys:

Keys:

argument	type	definition
dataToWorkOn	str	Name of the Sofa datafield we will work on here it will be *position*
incrPeriod	float	Period between each increment
incr	float	Value of each increment
rangeOfAction	float	Until which value the data will increase
angle	float	Initial angle value in radian
rodRadius	float	Radius Lenght of the circle

mor.animation.shakingAnimations.shakingSofia

`shakingSofia`(*objToAnimate*, *dt*, *factor*, ***param*)

Animation function made specifically to shake the leg of the *6-legged Robot*.

It's an example of what can be a custom shaking animation. The animation consist on taking a position in entry, rotate it, and then update it in the component.

To use it the **params** parameters of `ObjToAnimate` which is a dictionnary will need 6 keys:

Keys:

argument	type	definition
dataToWorkOn	str	Name of the Sofa datafield we will work on here it will be *position*
incrPeriod	float	Period between each increment
incr	float	Value of each increment
rangeOfAction	float	Until which value the data will increase
angle	float	Initial angle value in radian
rodRadius	float	Radius Lenght of the circle

mor.animation.shakingAnimations.upDateValue

upDateValue(*actualValue*, *actuatorMaxPull*, *actuatorIncrement*)

Utility function for default animation.

Increment a sofa data value until fixed amount

Parameters

- **actualValue** –
- **actuatorMaxPull** –
- **actuatorIncrement** –

Returns

actualValue :

4.2 mor.utility

Set of utility functions used during the reduction process

mor.utility.graphScene	**Set of functions to extract the graph a scene**
mor.utility.sceneCreation	**Utility to construct and modify a SOFA scene**
mor.utility.writeScene	**Set of functions to create a reusable SOFA component out of a SOFA scene**

4.2.1 mor.utility.graphScene

Set of functions to extract the graph a scene

The extracted results will be put into 2 dictionnary as follow

```
tree:
    node1:
        child1:
    node2:
        child2:

obj:
    node1:
        obj1:
    child1:
        obj2
    node2:
        obj3
```

Functions

dumpGraphScene	**Dump the Graph of the SOFA scene as 2 dictionnaries in a yaml file**
getGraphScene	**This function will iterate over the SOFA graph scene from a node and build from there 2 dictionnaries containing its content**
importScene	**Return the graph of a SOFA scene**

mor.utility.graphScene.dumpGraphScene

`dumpGraphScene`(*node*, *fileName='graphScene.yml'*)

Dump the Graph of the SOFA scene as 2 dictionnaries in a yaml file

argument	type	definition
node	Sofa.node	From which node we want the graph
fileName	str	In which File we will put the result

mor.utility.graphScene.getGraphScene

`getGraphScene`(*node*, *getObj=False*)

This function will iterate over the SOFA graph scene from a node and build from there 2 dictionnaries containing its content

argument	type	definition
node	Sofa.node	From which node we want the graph
getObj	bool	Boolean to choose if we want the node/obj as key or just its name

mor.utility.graphScene.importScene

`importScene`(*filePath*)

Return the graph of a SOFA scene

Thanks to the SOFA Launcher, it will launch a templated scene that will extract from an original scene its content as 2 dictionnaries containing:

- The different Sofa.node of the scene keeping there hierarchy.
- All the SOFA component contained in each node with the node.name as key.

argument	type	definition
filePath	str	Absolute path to the SOFA scene

4.2.2 mor.utility.sceneCreation

Utility to construct and modify a SOFA scene

Functions

addAnimation	**Add/or not animations defined by** `ObjToAnimate` **to the** `splib.animation.AnimationManagerController` **thanks to** `splib.animation.animate`
addPlugin	**Add plugin if not present in Sofa scene**
createDebug	**Will, from our original scene, remove all unnecessary component and add a ReadState component in order to see what happen during** `phase1` or `phase3`
getContainer	Search for **TopologyContainer** and return it
getNodeSolver	Get specific Solver if contained in `Sofa.Core.Node`.
modifyGraphScene	**Modify the current scene to be able to reduce it**
removeNode	From a `Sofa.Core.Node` get its first parent and remove `Sofa.Core.Node.removeChild`
removeNodes	Iterate over list of `Sofa.Core.Node` and remove them with *removeNode*
removeObject	From a `Sofa.Core.Object` get `Sofa.Core.BaseContext` and remove itself `Sofa.Core.Node.removeObject`
removeObjects	Iterate over list of `Sofa.Core.Object` and remove them with *removeObject*
saveElements	**Depending on the forcefield will go search for the right kind of elements (tetrahedron/triangles...) to save**
searchObjectClassInGraphScene	**Search in the Graph scene recursively for all the node with the same className as toFind**
searchPlugin	**Search if a plugin if used in a SOFA scene**

mor.utility.sceneCreation.addAnimation

addAnimation(*node*, *phase*, *timeExe*, *dt*, *listObjToAnimate*)

Add/or not animations defined by `ObjToAnimate` **to the** `splib.animation.AnimationManagerController` **thanks to** `splib.animation.animate`

argument	type	definition
node	Sofa.Core.Node	from which node will search & add animation
phase	list(int)	list of 0/1 that according to its index will activate/desactivate a `ObjToAnimate` contained in *listObjToAnimate*
timeExe	sc	correspond to the total SOFA execution duration the animation will occure, determined with *nbIterations* (of `ReductionAnimations`) multiply by the *dt* of the current scene
dt	sc	time step of our SOFA scene
listObjToAnimate	list(`mor.reduction.container.objToAnimate`)	list conaining all the ObjToAnimate that will be use to shake our model

Thanks to the location parameters of an `ObjToAnimate`, we find the component or Sofa.node it will animate. *If its a Sofa.node we search something to animate by default CableConstraint/SurfacePressureConstraint.*

Returns
None

mor.utility.sceneCreation.addPlugin

addPlugin(*rootNode*, *pluginName*)

Add plugin if not present in Sofa scene

argument	type	definition
rootNode	Sofa.Core.Node	root of scene
pluginName	str	literal name of plugin

Search for it with *searchPlugin* and depending if returned boolean add it or not to current scene

Returns
found boolean

mor.utility.sceneCreation.createDebug

createDebug(*rootNode*, *pathToNode*, *stateFile='stateFile.state'*)

Will, from our original scene, remove all unnecessary component and add a ReadState component in order to see what happen during `phase1` or `phase3`

argument	type	definition
rootNode	`Sofa.Core.Node`	root node of the SOFA scene
pathToNode	str	Path to the only node we will keep to create our debug scene
stateFile	str	file that will be read by default by the ReadState component

Returns
None

mor.utility.sceneCreation.getContainer

getContainer(*node*)

Search for **TopologyContainer** and return it

argument	type	definition
node	`Sofa.Core.Node`	A Node stores other nodes and components

Returns
TopologyContainer object

mor.utility.sceneCreation.getNodeSolver

getNodeSolver(*node*)

Get specific Solver if contained in `Sofa.Core.Node`.

argument	type	definition
node	`Sofa.Core.Node`	A Node stores other nodes and components

searching for ConstraintSolver, LinearSolver and OdeSolver solvers

Returns
list of solvers found

mor.utility.sceneCreation.modifyGraphScene

modifyGraphScene(*node*, *nbrOfModes*, *newParam*)

Modify the current scene to be able to reduce it

argument	type	definition
node	Sofa.Core.Node	from which node will search & modify the graph
nbrOfModes	int	Number of modes choosed in `mor.reduction.reduceModel.ReduceModel.phase3` or `mor.reduction.reduceModel.ReduceModel.phase4` where this function will be called
newParam	dic	Contains numerous argument to modify/replace some component of the SOFA scene. *more details see* `ReductionParam`

For more detailed about the modification & why they are made see here

Returns
None

Raises
Exception: cannot modify scene from path

mor.utility.sceneCreation.removeNode

`removeNode`(*node*)

From a `Sofa.Core.Node` get its first parent and remove `Sofa.Core.Node.removeChild`

argument	type	definition
node	Sofa.Core.Node	A Node stores other nodes and components

Returns
None

mor.utility.sceneCreation.removeNodes

`removeNodes`(*nodes*)

Iterate over list of `Sofa.Core.Node` and remove them with *`removeNode`*

argument	type	definition
nodes	list(`Sofa.Core.Node`)	A Node stores other nodes and components

Returns
None

mor.utility.sceneCreation.removeObject

`removeObject`(*obj*)

From a `Sofa.Core.Object` get `Sofa.Core.BaseContext` and remove itself `Sofa.Core.Node.removeObject`

argument	type	definition
obj	`Sofa.Core.Object`	Base class for components which can be added in a simulation

Returns
None

mor.utility.sceneCreation.removeObjects

`removeObjects`(*objects*)

Iterate over list of `Sofa.Core.Object` and remove them with *`removeObject`*

argument	type	definition
objects	list(`Sofa.Core.Object`)	Base class for components which can be added in a simulation

Returns
None

mor.utility.sceneCreation.saveElements

`saveElements`(*node*, *dt*, *forcefield*)

Depending on the forcefield will go search for the right kind of elements (tetrahedron/triangles…) to save

argument	type	definition
node	Sofa.Core.Node	from which node will search to save elements
dt	sc	time step of our SOFA scene
forcefield	list(str)	list of path to the forcefield working on the elements we want to save see *forcefield*

After determining what to save we will add an animation with a *duration* of 0 that will be executed only once when the scene is launched saving the elements.

To do that we use `splib.animation.animate`

Returns
None

mor.utility.sceneCreation.searchObjectClassInGraphScene

searchObjectClassInGraphScene(*node*, *toFind*)

Search in the Graph scene recursively for all the node with the same className as toFind

argument	type	definition
node	Sofa.Core.Node	Sofa node in wich we are working
toFind	str	className we want to find

Returns
results of search in tab

mor.utility.sceneCreation.searchPlugin

searchPlugin(*rootNode*, *pluginName*)

Search if a plugin if used in a SOFA scene

argument	type	definition
rootNode	Sofa.Core.Node	root of scene
pluginName	str	literal name of plugin

Returns
found boolean

4.2.3 mor.utility.writeScene

Set of functions to create a reusable SOFA component out of a SOFA scene

---

Functions

buildArgStr	**According to the case it will add translation,rotation,scale arguments**
writeFooter	**Write a templated Footer to a file**
writeGraphScene	**Write a SOFA scene from lists**
writeHeader	**Write a templated Header to a file**

mor.utility.writeScene.buildArgStr

buildArgStr(*arg*, *translation=None*)

According to the case it will add translation,rotation,scale arguments

Allowing to move easily in a scene the created component

Args:

argument	type	definition
arg	dic	Contains all argument of a Sofa Component
translation	float	Contanis the initial translation of the model this will allow us to calculate a new position of an object depending of our reduced model by substracting our model relative origin make the TRS in the absolute origin and replace it in our model relative origin

mor.utility.writeScene.writeFooter

writeFooter(*packageName*, *nodeName*, *listplugin*, *dt*, *gravity*)

Write a templated Footer to a file

This footer will finalize the component created by *writeHeader* & *writeGraphScene* allowing the user to test it rapidly while keeping its original root configuration (listplugin/dt/gravity)

Args:

argument	type	definition
packageName	str	Name of the file were we will write (without any extension) |that will also be the name for the new component
nodeName	str	Name of the Sofa.Node we reduce
listplugin	str	Initial scene plugin list
dt	str	Initial scene plugin dt
gravity	str	Initial scene plugin gravity

mor.utility.writeScene.writeGraphScene

writeGraphScene(*packageName*, *nodeName*, *myMORModel*, *myModel*)

Write a SOFA scene from lists

With 2 lists describing the 2 Sofa.Node containing the components for our reduced model, this function will write each component with their initial parameters and clean or add parameters in order to have in the end a reduced model component reusable as a function with arguments as :

```
def MyReducedModel(
                  attachedTo=None,
                  name="MyReducedModel",
                  rotation=[0.0, 0.0, 0.0],
                  translation=[0.0, 0.0, 0.0],
                  scale=[1.0, 1.0, 1.0],
                  surfaceMeshFileName=False,
                  surfaceColor=[1.0, 1.0, 1.0],
                  nbrOfModes=nbrOfModes,
                  hyperReduction=True):
```

Args:

argument	type	definition
packageName	str	Name of the file were we will write (without any extension)
nodeName	str	Name of the Sofa.Node we reduce
myMORModel	list	list of tuple (solver_type , param_solver) *more details see* *myMORModel*
myModel	OrderedDict	Ordered dic containing has key Sofa.Node.name & has var a tuple of (Sofa_componant_type , param) *more details see* myModel

mor.utility.writeScene.writeHeader

writeHeader(*packageName*, *nbrOfModes*)

Write a templated Header to a file

Arg:

argument	type	definition
packageName	str	Name of the file were we will write (without any extension)
nbrOfModes	int	Maximum number of nodes set as a default parameter

4.3 mor.wrapper

Set of functions to modify the SOFA scene during its construction

Content:

mor.wrapper.replaceAndSave	**Functions that will be use during wrapping**

4.3.1 mor.wrapper.replaceAndSave

Functions that will be use during wrapping

Global Variable

forceFieldImplemented

List of ForceField implemented and there associated HyperReduced one This will be use to *swap* forcefield during scene creation with *MORreplace*

myModel

OrderedDict that will contain:

- has key Sofa.node.name
- has items list of tuple (type,argument) each one coresponding to a component

`myMORModel`

list of tuple (type,argument) each one coresponding to a component

`pathToUpdate`

`forcefield`

Methods

Functions

MORreplace	**Will replace classical ForceField by HyperReduced one**
modifyPath	**Correct wrong link induce by the change later done in the scene**

mor.wrapper.replaceAndSave.MORreplace

`MORreplace`(*node*, *type*, *newParam*, *initialParam*)

Will replace classical ForceField by HyperReduced one

argument	type	definition
node	Sofa.node	On which node the current object will be set
type	undefined	Type of the Sofa.object
newParam	dic	Contains numerous argument to modify/replace some component of the SOFA scene. *more details see* `ReductionParam`
initialParam	dic	Contains all the initial argument of the SOFA component being instanciated

This function work thanks to the `stlib.scene.Wrapper` of the STLIB SOFA plugin that will call this function BEFORE creating any SOFA component enabling us to replace/modify the SOFA component before its creation

This function will also, if there is *save* in the *newParam* key, save the initial component type & argument into 2 global variable `myModel` & `myMORModel` that will be used later by `writeGraphScene` to create a reusable component.

We *save* our scene here with all the complications it will produce, wrong links (corrected by `modifyPath`), need to differentiate components from *myModel* that will be moved in *myMORModel*, ect… Because this way the component parameters are not polluted by all unnecessary *dataFields* that are initialized during creation.

mor.wrapper.replaceAndSave.modifyPath

modifyPath(*currentPath*, *type*, *initialParam*, *newParam*)

Correct wrong link induce by the change later done in the scene

This step isn't always needed for execution because all the DataLink are made BEFORE we change the scene with *modifyGraphScene* while the links are all correct (normally). But this way when we will "save" the scene with all the data value the links will be correct.

Also for the links to DATA (@myCoponent.myData) or DataLink poorly implemented if the link is false during initialization this link (string representing the path) will be lost and won't be tried again during bwdInit.

To correct that, we need to update after our scene modification, the changed links. We do that with *pathToUpdate*

User Interface library

gui	**Set of class/functions used to created the MOR GUI**

4.4 mor.gui

Set of class/functions used to created the MOR GUI

Content:

mor.gui.ui_design	**Module describing the visual of MOR GUI**
mor.gui.utility	**Sets of utility Fct used by the GUI**
mor.gui.widget	**Set of custom Widget used to created the MOR GUI**

4.4.1 mor.gui.ui_design

Module describing the visual of MOR GUI

Classes

Ui_MainWindow()

4.4.2 mor.gui.utility

Sets of utility Fct used by the GUI

Functions

`checkExistance`(dir)	
`check_state`(sender)	
`checkedBoxes`(checkBox, items[, checked])	checkedBoxes will with the state of a checkBox change accordingly the state of other checkBoxes
`display`(completer)	
`generatePhaseToExecute`(nbrAnimation)	
`greyOut`(checkBox, items[, checked])	greyOut makes items unavailable for the user by greying them out
`left`(lineEdit)	
`msg_error`(msg, info)	
`msg_info`(msg, info)	
`msg_warning`(msg, info)	
`openDirName`(hdialog[, display])	openDirName will pop up a dialog window allowing the user to choose a directory and potentially display the path to it
`openFileName`(hdialog[, filter, display])	openFileName will pop up a dialog window allowing the user to choose a file and potentially display the path to it
`openFilesNames`(hdialog[, filter, display])	openFilesNames will pop up a dialog window allowing the user to choose multiple files and potentially display there coreponding path
`openLink`(url)	
`removeLine`(tab[, rm])	
`right`(lineEdit)	
`setAnimationParamStr`(cell, items)	
`setBackColor`(widget[, color])	
`setBackground`(obj, color)	
`setCellColor`(tab, dialog, row, column)	
`setShortcut`(listLineEdit)	Fill 2 global variables *shortcut* and *lastVisited* to current path of the scene we are working on.
`update_progress`(progress)	

4.4.3 mor.gui.widget

Set of custom Widget used to created the MOR GUI

Content:

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

m

INDEX