SoftRobots Components Documentation

Release 19.01

DEFROST team

Aug 31, 2023

Contents

1 Contents of the library **3**
- 1.1 component.constraint . . . 3
- 1.2 component.controller . . . 9
- 1.3 component.engine . . . 16

2 Indices and tables **19**

Python Module Index **21**

Index **23**

The SoftRobots plugin contains SOFA components dedicated to soft robotics. You can find for example, models for cable and pneumatic actuations, tools to define trajectories for the robot's end effector, or tools to communicate with microcontroller boards.

All these components are discribed in this documentation.

CHAPTER 1

Contents of the library

constraint	In our framework, we handle the actuation and contact by defining specific constraints with Lagrange multipliers on the boundary conditions of the deformable models.
controller	A Controller is a component that can process events from the keyboard or the mouse, launched at the beginning and the end of a time step.
engine	An Engine is a component that computes a set of output Data's from a set of input Data's.

1.1 component.constraint

In our framework, we handle the actuation and contact by defining specific constraints with Lagrange multipliers on the boundary conditions of the deformable models. Different types of actuators are proposed (e.g cable and pneumatic actuators).

1.1.1 Contents

CableConstraint	In this directory you will find multiple examples showing how to use the **CableConstraint** component:
SurfacePressureConstraint	In this directory you will find multiple examples showing how to use the **SurfacePressureConstraint** component:
UnilateralPlaneConstraint	In this directory you will find one example showing how to use the **UnilateralPlaneConstraint** component:

component.constraint.CableConstraint

In this directory you will find multiple examples showing how to use the **CableConstraint** component:

- **Finger.py** : Soft actuated finger
- **FingerWithSTLIB.py** : Soft actuated finger using the STLIB plugin
- **DisplacementVsForceControl.py** : Soft actuated fingers showing different controls

Below is a video of a soft finger actuated with one cable. You can run this simulation by loading the file **Finger.pyscn** with the application runSofa.

Example

```
# This create a new node in the scene. This node is appended to the finger's node.
cable = finger.addChild('cable')

# This create a MechanicalObject, a component holding the degree of freedom of our
# mechanical modelling. In the case of a cable it is a set of positions specifying
# the points where the cable is passing by.
cable.addObject('MechanicalObject',
                position=[
                [-17.5, 12.5, 2.5],
                [-32.5, 12.5, 2.5],
                [-47.5, 12.5, 2.5],
                [-62.5, 12.5, 2.5],
                [-77.5, 12.5, 2.5],

                [-83.5, 12.5, 4.5],
                [-85.5, 12.5, 6.5],
                [-85.5, 12.5, 8.5],
                [-83.5, 12.5, 10.5],

                [-77.5, 12.5, 12.5],
                [-62.5, 12.5, 12.5],
                [-47.5, 12.5, 12.5],
                [-32.5, 12.5, 12.5],
                [-17.5, 12.5, 12.5])

# Create a CableConstraint object with a name.
# The indices are referring to the MechanicalObject's positions.
# The last index is where the pullPoint is connected.
cable.addObject('CableConstraint', name="aCableActuator",
                #indices=range(0,14),
                indices=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13],
                pullPoint=[0.0, 12.5, 2.5])

# This create a BarycentricMapping. A BarycentricMapping is a key element as it will␣
↪create a bi-directional link
# between the cable's DoFs and the finger's ones so that movements of the cable's␣
↪DoFs will be mapped
# to the finger and vice-versa;
cable.addObject('BarycentricMapping')
```

Data fields

Required	Description
indices	List of points connected by the cable (from extremity to actuated point). If no indices are given, default value is 0. In case of multiple indices, one point will be actuated and the others will represent sliding points for the cable.
pullPoint	Fixed point from which the cable is pulled. If unspecified, the default value is {0.0,0.0,0.0}
value	Displacement or force to impose.
valueIndex	Index of the value (in InputValue vector) that we want to impose. If unspecified the default value is {0}.
valueType	Either "displacement", the contstraint will impose the displacement provided in data value[valueIndex], or force, in this case the contstraint will impose the force provided in data value[valueIndex]. If unspecified, the default value is displacement.

Optional	Description
maxForce	Maximum force of the actuator. If unspecified no maximum value will be considered.
minForce	Minimum force of the actuator. If unspecified no minimum value will be considered and the cable will then be seen as a stiff rod able to push.
maxPositiveDisp	Maximum displacement of the actuator in the positive direction. If unspecified no maximum value will be considered.
maxNegativeDisp	Maximum displacement of the actuator in the negative direction. If unspecified no maximum value will be considered.
maxDispVariation	Maximum variation of the displacement allowed. If not set, no max variation will be concidered.
drawPullPoint	If true, will draw the pull point (default true).
drawPoints	If true, will draw the points (default true).
color	Color of the string.
hasPullPoint	If false, the pull point is not considered and the cable is entirely mapped. In that case, needs at least 2 different point in indices

Properties	Description
cableInitialLength	Read only. Gives the initial length of the cable
cableLength	Read only. Gives the current length of the cable. Computation done at the end of the time step.
force	Read only. Output force
displacement	Read only. Output displacement compared to the initial cable length

component.constraint.SurfacePressureConstraint

In this directory you will find multiple examples showing how to use the **SurfacePressureConstraint** component:

- **Springy.pyscn** : Soft actuated accordion
- **PressureVsVolumeGrowthControl.pyscn** : Stanford bunny

Below is a video of a soft Standford bunny actuated with pressure in its inner cavity. You can run this simulation by loading the file **PressureVsVolumeGrowthControl.pyscn** with the application runSofa.

Example

```
#  This create a new node in the scene. This node is appended to the accordion's node.
cavity = accordion.createChild('cavity')

# This create a MechanicalObject, a component holding the degree of freedom of our
# mechanical modelling. In the case of a pneumatic actuation it is a set of positions␣
↪describing the cavity wall.
cavity.createObject('MeshSTLLoader', name='loader', filename=path+'Springy_Cavity.stl
↪')
cavity.createObject('MeshTopology', src='@loader', name='topo')
cavity.createObject('MechanicalObject', name='cavity')

# Create a SurfacePressureConstraint object with a name.
cavity.createObject('SurfacePressureConstraint', template='Vec3', name="pressure",
                    triangles='@topo.triangles',
                    valueType="1",
                    value="8")

# This create a BarycentricMapping. A BarycentricMapping is a key element as it will␣
↪create a bi-directional link
# between the cavity wall (surfacic mesh) and the accordion (volumetric mesh) so that␣
↪movements of the cavity's DoFs will be mapped
# to the accordion and vice-versa;
cavity.createObject('BarycentricMapping', name='mapping',  mapForces=False,␣
↪mapMasses=False)
```

Data fields

Required	Description
triangles	List of triangles on which the surface pressure is applied. If no list is given, the component will fill the two lists with the context topology.
quads	List of quads on which the surface pressure is applied. If no list is given, the component will fill the two lists with the context topology.
value	List of choices for volume growth or pressure to impose.
valueIndex	Index of the value (in InputValue vector) that we want to impose. If unspecified the default value is {0}.
valueType	Either "volumeGrowth", the contstraint will impose the volume growth provided in data value[valueIndex], or "pressure", in this case the contstraint will impose the pressure provided in data value[valueIndex]. If unspecified, the default value is pressure.

Optional	Description
flipNormal	Allows to invert cavity faces orientation. If a positive pressure acts like a depressurization, try to set flipNormal to true.
maxPressure	Maximum pressure allowed for actuation. If no value is set by user, no maximum pressure constraint will be considered.
minPressure	Minimum pressure allowed for actuation. If no value is set by user, no minimum pressure constraint will be considered. A negative pressure will empty/drain the cavity.
maxVolumeGrowth	Maximum volume growth allowed for actuation. If no value is set by user, no maximum will be considered. NB: this value has a dependancy with the time step (volume/dt) in the dynamic case.
minVolumeGrowth	Minimum volume growth allowed for actuation. If no value is set by user, no minimum will be considered. NB: this value has a dependancy with the time step (volume/dt) in the dynamic case.
maxVolumeGrowthVariation	Maximum volume growth variation allowed for actuation. If no value is set by user, no maximum will be considered. NB: this value has a dependancy with the time step (volume/dt) in the dynamic case.
drawPressure	Visualization of the value of pressure. If unspecified, the default value is {false}.
drawScale	Scale for visualization. If unspecified the default value is {0.1}.

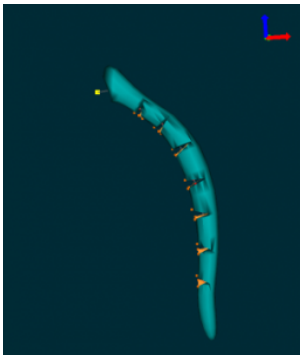
Properties	Description
volumeGrowth	Read only. Output volume growth.
pressure	Read only. Output pressure.
initialCavityVolume	Read only. Output volume of the cavity at init (only relevant in case of closed mesh).
cavityVolume	Read only. Output volume of the cavity (only relevant in case of closed mesh).

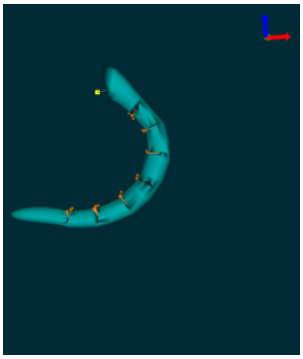
component.constraint.UnilateralPlaneConstraint

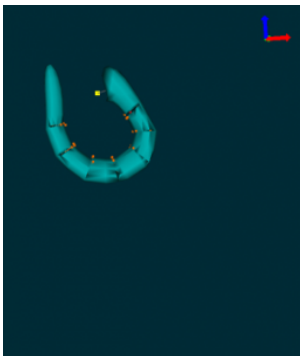
In this directory you will find one example showing how to use the **UnilateralPlaneConstraint** component:

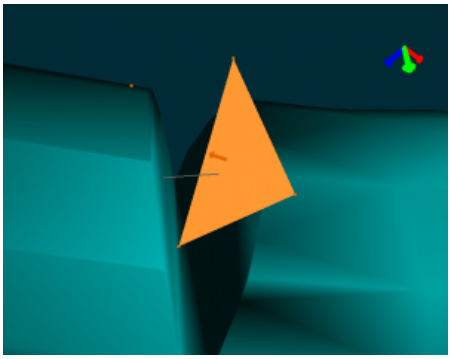
- **ArticulatedTentacle.pyscn** : Soft cable-driven tentacle with self-collision regions

This component is a simple point plane collision model. By providing 4 points to the component, the first point will be constrained to stay in one side of the plane described by the three other points (in the direction of the plane normal). All the four points, the triangle and the normal can be seen by allowing the 'Collision Model' in the 'View' tab. Below are images of the simulation.









Example

```
tentacleContact = tentacle.createChild('contact')
tentacleContact.createObject('MechanicalObject',
        position="64 0 11   69 7 8     69 -7 8     71 0 17 "+
                 "107 0 -23   111 7 -27     111 -7 -27    117 0 -17 "+
                 "93 0 -7.5   97 7 -11     97 -7 -11    102 0 -0.5 "+
                 "138 0 -73   141 7 -77     141 -7 -77     146 0 -72 "+
                 "78 0 3     83 7 0    83 -7 0    86 0 9   "+
                 "118 0 -38   122 6.7 -42     122 -7 -42   129 -0.2 -35  "+
                 "129.5 0 -55.5    132 7 -60    132.5 -7 -59.6   138 0 -53.5")
tentacleContact.createObject('UnilateralPlaneConstraint', indices="0 1 2 3")
tentacleContact.createObject('UnilateralPlaneConstraint', indices="4 5 6 7")
tentacleContact.createObject('UnilateralPlaneConstraint', indices="8 9 10 11")
tentacleContact.createObject('UnilateralPlaneConstraint', indices="12 13 14 15")
tentacleContact.createObject('UnilateralPlaneConstraint', indices="16 17 18 19")
tentacleContact.createObject('UnilateralPlaneConstraint', indices="20 21 22 23")
tentacleContact.createObject('UnilateralPlaneConstraint', indices="24 25 26 27")
```

(continues on next page)

(continued from previous page)

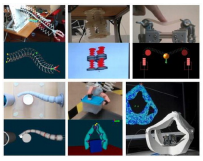
```
tentacleContact.createObject('BarycentricMapping')
```

Data fields

Required	Description
indices	Four indices: First one for the constrained point. The others to describe the plane.

Optional	Description
flipNormal	The normal must be to the direction of the point.

1.1.2 Related paper



"Software toolkit for modeling, simulation and control of soft robots",
E. Coevoet, T. Morales-Bieze, F. Largilliere, Z. Zhang, M. Thieffry, et al.
Advanced Robotics (2017)

1.2 component.controller

A Controller is a component that can process events from the keyboard or the mouse, launched at the beginning and the end of a time step.

1.2.1 Contents

AnimationEditor	The **AnimationEditor** component is used to build an animation from key points motion, or typically to build effector goals trajectories.
CommunicationController	The **CommunicationController** component can be used to send data from a simulation to another using ZMQ library.
DataVariationLimiter	The **DataVariationLimiter** component is used to avoid big variation of an input data.
SerialPortBridgeGeneric	The "SerialPortBridgeGeneric" component is used to send data (force, displacement, pressure...) through the usb port.

component.controller.AnimationEditor

The **AnimationEditor** component is used to build an animation from key points motion, or typically to build effector goals trajectories. In this directory you will find one example showing how to use the component:

- **RigidAnimation.py** : Rigid cube

The **AnimationEditor** component inherites from SOFA controller. In SOFA, a controller has an input data **listening** that has to be set to true if you want the component to be active in your scene, **listening=1**.

Now you can build your animation. To navigate through the timeline, click left on the scene window and use the following keyboard commands:

- ctrl+→ : move the cursor to the right
- ctrl+← : move the cursor to the left
- ctrl+pgUp: move the cursor to the next keyframe
- ctrl+pgDn : move the cursor to the previous keyframe

The cursor is represented by a white triangle. You can now move the key points that you created with **MechanicalObject** and save a keyframe. Between two keyframes an interpolation is computed to create the animation. A keyframe is represented by a yellow triangle upon a line. Here are the keyframes commands:

- ctrl+a : add a key in the cursor location
- ctrl+d : delete the key
- ctrl+c : copy the key
- ctrl+x : cut the key
- ctrl+v : paste the key

You can also save, load or play/pause the animation using the following commands:

- ctrl+w : (write) save the animation in "filename"
- ctrl+m : play/pause the animation

The component is templated with Vec3 and Rigid3. This kind of animation could be used, for instance, for the control of an effector in position and rotation. The **PositionEffector** component is also templated with Rigid3. Thus, you can save a trajectory for a 6DoF target.

Example

```
def createScene(rootNode):
    point = rootNode.addChild('point')
    point.addObject('EulerImplicitSolver', firstOrder=True)
    point.addObject('CGLinearSolver', iterations=100, tolerance=1e-5, threshold=1e-5)
    point.addObject('MechanicalObject', template='Rigid3',
                    position=[0, 0, 0, 0, 0, 0, 1],
                    showObject=True,
                    showObjectScale=0.1,
                    drawMode=1,
                    showColor=[255, 255, 255, 255])
    # The AnimationEditor takes multiple options
    # template : should be the same as the mechanical you want to animate
    # filename : file in which the animation will be saved
```

(continues on next page)

(continued from previous page)

```
    # load : set to true to load the animation at init (default is true)
    # loop : when the animation is playing, set this option to true to loop and start␣
↪again the animation
    # dx : to control the animation in displacement instead of time
    # frameTime (default is 0.01)
    # drawTimeline (default is true)
    # drawTrajectory (default is true)
    # drawSize : coefficient size of displayed elements of trajectory
    point.addObject('AnimationEditor', name='animation',
                    template='Rigid3', filename=path + 'RigidAnimation.txt',
                    load=True,
                    drawTimeline=True, drawTrajectory=True)

    visu = point.addChild('visu')
    visu.addObject('MeshOBJLoader', name='loader', filename='mesh/cube.obj')
    visu.addObject('OglModel', src='@loader', filename='mesh/cube.obj')
    visu.addObject('RigidMapping')

    return rootNode
```

Data fields

Required	Description
maxKeyFrame	Max >= 1, default 150
filename	If no filename given, set default to animation.txt.

Optional	Description
loop	If true, will loop on the animation (only in play mode).
load	If true, will load the animation at init.
dx	Variation of displacement. You can control the animation on displacement instead of time. If dx is set, at each time step, the animation will progress in term of displacement/distance. A positive dx means move forward and a negative dx means backward (on the timeline).
frameTime	Frame time.
drawTimeline	
drawSize	
drawTrajectory	

Properties	Description
cursor	Read only. Current frame of the cursor along the timeline.

component.controller.CommunicationController

The **CommunicationController** component can be used to send data from a simulation to another using ZMQ library. To use this component you need to compile SOFA with the option **SOFTROBOTS_COMMUNICATIONCONTROLLER** enabled in cmake, and install the ZMQ library:

1.1- On Linux

#Debian/Ubuntu

sudo apt-get install libzmq3-dev

#Fedora

sudo dnf install zeromq-devel

1.1- On MacOS (missing)

1.1- On Windows, download the Windows source of libzmq and build using Visual Studio. Put a copy of zmq.hpp from the cppzmq project (github) in the include folder of libzmq.

2- In the cmake gui, enable: SOFTROBOTS_COMMUNICATIONCONTROLLER = true

3- Compile SOFA

In this directory ("SoftRobots/examples/component/controller/CommunicationController") you will find one example showing how to use the component:

- **SimulationDirect_Receiver.py** : Soft actuated accordion, direct problem
- **SimulationInverse_Sender.py** : Soft actuated accordion, inverse problem

Below is a video of the simulations running simultaneously and with a communication between them.

Example

SimulationDirect_Receiver.py:

```
#For local communication
accordion.addObject('CommunicationController', name="sub", listening='1',
                                                job="receiver", port="5558",␣
↪nbDataField="4", pattern="0")
#Between two different computers, specify the ip adress of the sender
#accordion.addObject('CommunicationController', name="sub", listening='1',
                                                job="receiver", port="5558",␣
↪nbDataField="4", ip="...")
```

SimulationInverse_Sender.py:

```
accordion.addObject('CommunicationController', listening='1', job="sender", port="5558
↪",                                          nbDataField="4", pattern="0",
                        data1="@cavity/pressure.volumeGrowth",
                        data2="@cables/cable1.displacement",
                        data3="@cables/cable2.displacement",
                        data4="@cables/cable3.displacement")
```

Data fields

Required	Description
job	If unspecified, the default value is sender.
pattern	Pattern used for communication. publish/subscribe: Messages sent are distributed in a fan out fashion to all connected peers. Never blocks. request/reply: Message sent are waiting for reply. Allows only an alternating sequence of send/reply calls. Default is publish/subscribe. WARNING: the pattern should be the same for both sender and receiver to be effective.
nbDataField	Number of field 'data' the user want to send or receive. Default value is 1.
data	Data to send or receive.
port	Default value 5556.

Optional	Description
HWM	If publisher, you can define the High Water Mark which is a hard limit on the maximum number of outstanding messages shall queue in memory. Default 0 (means no limit).
ip	IP adress of the sender. No given adress will set up a local communication.
atBeginAnimationStep	If true, will send or receive datas at begin of the animation step (if false, at end of the animation step). Default true.
beginAt	Time step value to start the communication at.
timeOut	Set time out (in ms) before killing the communication. Default is 3000ms, 0 means no time out.

component.controller.DataVariationLimiter

The **DataVariationLimiter** component is used to avoid big variation of an input data. It interpolates between two consecutive inputs when a jump is detected. In this directory you will find one example showing how to use the component:

- **DataVariationLimiter.pyscn** : Soft actuated accordion

Below are images of the simulation.









Example

```
goal = rootNode.createChild('goal')
goal.createObject('EulerImplicitSolver')
goal.createObject('CGLinearSolver', iterations='100', tolerance="1e-5", threshold="1e-
↪5")
goal.createObject('MechanicalObject', name='goalMO',
                  position='0 0 5',
                  showObject="1",
                  showObjectScale="1",
                  drawMode="1")
goal.createObject('DataVariationLimiter', name="stabilizer", listening="1", input=
↪"@goalMO.position")
goal.createObject('MechanicalObject', name='goalMOStabilized',
                  position='@stabilizer.output',
                  showObject="1",
                  showObjectScale="1",
                  drawMode="1")
goal.createObject('UncoupledConstraintCorrection')
```

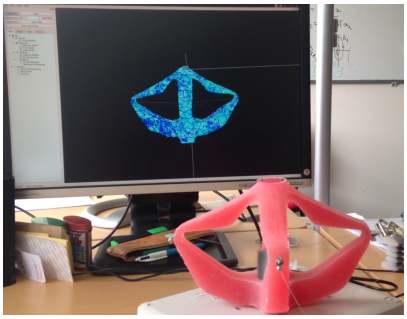
Data fields

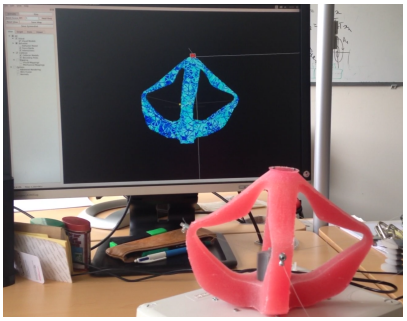
Required	Description
input	Link to the input variables
output	Link to the output
size	Input size.
maxJump	Maximal jump allowed. Default 10% is equivalent to jump = 0.1.
nbStep	Number of interpolation steps. Default is 50.

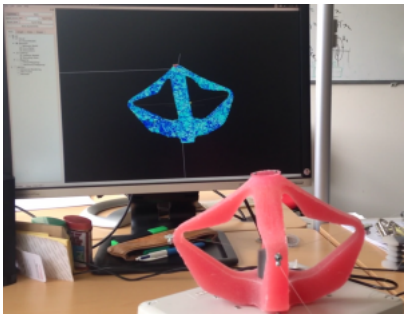
Optional	Description
initOutput	If true, will initialize the output with the input.

component.controller.SerialPortBridgeGeneric

The "SerialPortBridgeGeneric" component is used to send data (force, displacement, pressure...) through the usb port. Usally used to send data to an Arduino card to control the real robot.







Example

```
rootNode.addObject('SerialPortBridgeGeneric', name="serial", port="/dev/ttyACM0",␣
↪baudRate="115200", size="5", listening=True, header=255, packetOut=...)
```

Data fields

Required	Description
port	Serial port name.
baudRate	Transmission speed.
packetOut	Data to send: vector of unsigned char, each entry should be an integer between 0 and (header-1) <= 255. The value of 'header' will be sent at the beginning of the sent data, enabling to implement a header research in the 'receiving' code, for synchronization purposes.
header	Vector of unsigned char. Only one value is espected, two values if splitPacket = 1.
size	Size of the arrow to send. Use to check sentData size. Will return a warning if sentData size does not match this value.
receive	If true, will read from serial port (timeOut = 10ms).

Optional	Description
precise	If true, will send the data in the format [header[0],[MSB,LSB]*2*size].
splitPacket	If true, will split the packet in two for lower error rate (only in precise mode), data will have the format [header[0],[MSB,LSB]*size],[header[1],[MSB,LSB]*size].
redundancy	Each packet will be send that number of times (1=default).

Properties	Description
packetIn	Read only. Data received: vector of unsigned char, each entry should be an integer between 0 and (header-1) <= 255.

1.3 component.engine

An Engine is a component that computes a set of output Data's from a set of input Data's.

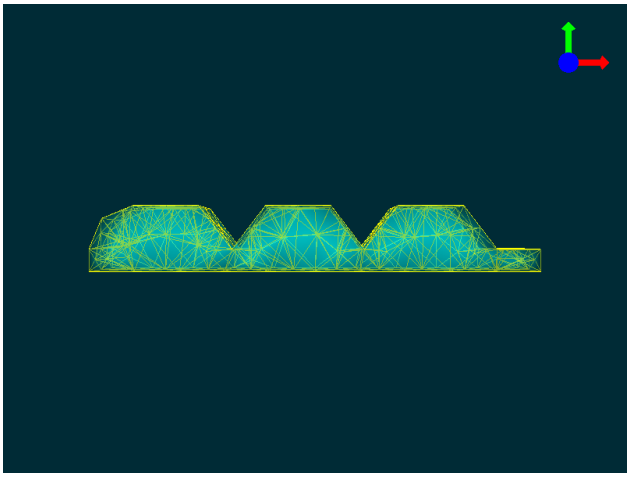
1.3.1 Contents

VolumeFromTetrahedrons	The **VolumeFromTetrahedrons** component computes the volume of a given volumetric mesh (tetrahedrons or/and hexahedrons).
VolumeFromTriangles	The **VolumeFromTriangles** component computes the volume of a given surfacic mesh (triangles or/and quads).

component.engine.VolumeFromTetrahedrons

The **VolumeFromTetrahedrons** component computes the volume of a given volumetric mesh (tetrahedrons or/and hexahedrons). In this directory you will find one example showing how to use the component:

- **Finger.pyscn**



Example

```
finger = rootNode.createChild('finger')

# Create the popology
finger.createObject('MeshVTKLoader', name='loader', filename=path+'finger.vtk')
finger.createObject('TetrahedronSetTopologyContainer', src='@loader', name='container
↪')
finger.createObject('TetrahedronSetTopologyModifier')

finger.createObject('VolumeFromTetrahedrons')
```

Data fields

Required	Description
positions	If not set by user, find the context mechanical.
tetras	If not set by user, find the context topology.
hexas	If not set by user, find the context topology.

Optional	Description
update	If true, will update the volume with the current positions.

Properties	Description
volume	Read only. Output volume.

component.engine.VolumeFromTriangles

The **VolumeFromTriangles** component computes the volume of a given surfacic mesh (triangles or/and quads). In this directory you will find one example showing how to use the component:

- **Finger.pyscn**



Example

```
finger = rootNode.createChild('finger')

# Create the popology
finger.createObject('MeshSTLLoader', name='loader', filename=path+'finger.stl')
finger.createObject('MeshTopology', src='@loader', name="mesh")
finger.createObject('VolumeFromTriangles')
```

Data fields

Required	Description
positions	If not set by user, find the context mechanical.
triangles	If not set by user, find the context topology.
quads	If not set by user, find the context topology.

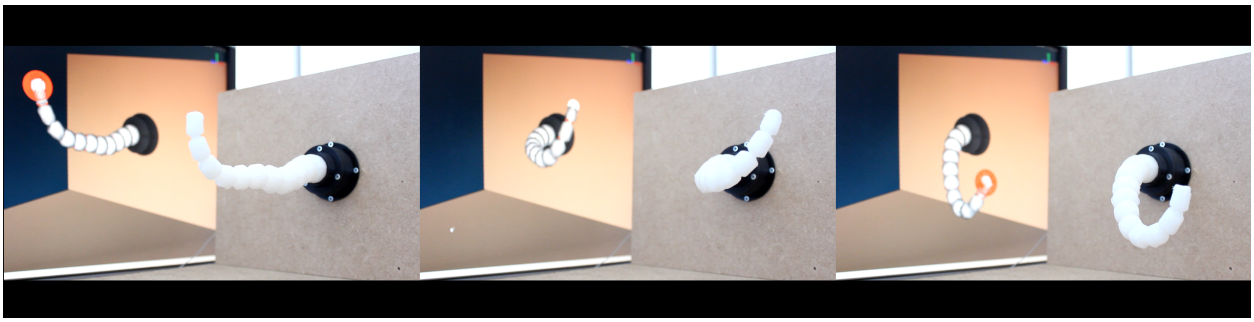
Optional	Description
update	If true, will update the volume with the current positions.

Properties	Description
volume	Read only. Output volume. Only relevant if closed surface.

CHAPTER 2

Indices and tables

- modindex
- search



Python Module Index

C

component, 1
component.constraint, 3
component.constraint.CableConstraint, 4
component.constraint.SurfacePressureConstraint, 5
component.constraint.UnilateralPlaneConstraint, 7
component.controller, 9
component.controller.AnimationEditor, 10
component.controller.CommunicationController, 12
component.controller.DataVariationLimiter, 13
component.controller.SerialPortBridgeGeneric, 15
component.engine, 16
component.engine.VolumeFromTetrahedrons, 16
component.engine.VolumeFromTriangles, 17

Index

C

component (*module*), 1
component.constraint (*module*), 3
component.constraint.CableConstraint (*module*), 4
component.constraint.SurfacePressureConstraint (*module*), 5
component.constraint.UnilateralPlaneConstraint (*module*), 7
component.controller (*module*), 9
component.controller.AnimationEditor (*module*), 10
component.controller.CommunicationController (*module*), 12
component.controller.DataVariationLimiter (*module*), 13
component.controller.SerialPortBridgeGeneric (*module*), 15
component.engine (*module*), 16
component.engine.VolumeFromTetrahedrons (*module*), 16
component.engine.VolumeFromTriangles (*module*), 17