LASER INTERFEROMETER GRAVITATIONAL WAVE OBSERVATORY
- LIGO -
CALIFORNIA INSTITUTE OF TECHNOLOGY
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Document Type **LIGO-T990108-00 - E** Nov. 1999

Mechanical Simulation Engine : Reference Manual

Giancalro Cella

Univ. of Pisa

Distribution of this draft:

xyz

This is an internal working note
of the LIGO Project..

California Institute of Technology
LIGO Project - MS 51-33
Pasadena CA 91125
Phone (626) 395-2129
Fax (626) 304-9834
E-mail: info@ligo.caltech.edu

Massachusetts Institute of Technology
LIGO Project - MS 20B-145
Cambridge, MA 01239
Phone (617) 253-4824
Fax (617) 253-7014
E-mail: info@ligo.mit.edu

WWW: http://www.ligo.caltech.edu/

MSE

— Version 0.1 November 6, 1999 —

A library for the simulation of mechanical systems.

Giancarlo Cella

Contents

1 MSE tutorial. — *Some simple examples of system construction.* 4
1.1 Simple pendulum. 4
1.2 A composite suspended system. 6
1.3 A simple attenuation chain. 7

2 High level library classes — *The reference documentation for the C++ API* 8
2.1 Beam — *Complete model for a beam.* 8
2.2 Clamp — *A constraint between two frames.* 11
2.3 ForceActuator — *A generator of force.* 14
2.4 ForceSensor — *A force sensor.* 15
2.5 Frame — *A frame owned by a mechanical object.* 21
2.6 MObject — *This is the base class for all object that can be composedinto a mechanical system.* 29
2.7 Mse — *This is the base class for all the mechanical classes.* 42
2.8 MSystem — *A mechanical subsystem.* 50
2.9 PositionActuator — *A position actuator.* 73
2.10 PositionSensor — *A position sensor.* 80
2.11 RigidBody — *A rigid body without internal structure.* 84
2.12 Spring — *This is the model for a simple spring.* 87
2.13 TwoNodesElement — *Base class for all the mechanical object with twoframes.* 89
2.14 Wire — *This is the model for a simple wire with torsional stiffness.* 91

3 Low level library — *The reference documentation for the utility library* 94
Class Graph 102

Mechanical Simulation Engine documentation

The mechanical simulation engine is a library of C++ classes which can be used to simulate a mechanical system in the time domain. It provides also additional capabilities of extracting, with some restrictions, frequency domain informations (transfer functions).

The mechanical system is defined by making an instance of the MSystem class. Next different mechanical objects can be added to the system and connected together. Each mechanical object is a derivate class of the MObject, see the relative documentation for the common methods. The more important point is that the configuration of a mechanical object is completely specified by a finite number of Frames. For details on the definition of a frame see the documentation of the Frame class.

The connection of different mechanical object is specified by a rigid constraint between two frames, using the Connect method of the MSystem class. When all the connections are done, the system can be decomposed in clusters of unconnected frames. Each cluster is in a one–to–one correspondence with six degrees of freedom of the system, three for the translations and three for the rotations.

On each frame a force and a torque is defined. For cluster of connected frames we can choose a representative frame (which we call master frame) which parametrize the dynamics. In order to write the motion equations we must evaluate the total force and torque applied to the master frame. This can be done using the relations

$$\vec{f}_i = \vec{f}_j$$

$$\vec{\tau}_i = \vec{\tau}_j + \vec{r}_{ij} \wedge \vec{f}_j$$

which connect the force and the torque between two different geometrical points ($\vec{r}_{ij} = \vec{r}_i - \vec{r}_j$ is the separation between the two points).

The linearized dynamics can be written in the form

$$M\frac{d^2x}{dt^2} + \Lambda\frac{dx}{dt} + Kx = f \tag{1}$$

where M is a mass matrix, Λ a damping matrix and K a stiffness matrix. The symbol x stands for a vector of six components, which are the three small linear variations and the three small angular variations, and f is a vector with three forces and three torques

$$x \equiv (\delta x, \delta y, \delta z, \delta\theta_x, \delta\theta_y, \delta\theta_z) \tag{2}$$

$$f \equiv (f_x, f_y, f_z, \tau_x, \tau_y, \tau_z) \tag{3}$$

Each mechanical object provide on each of its frames the quantities f,M,Λ and K, which must be added together after a transformation to the reference of the master frame. This can be done in the following way. When we write the motion equation in the reference of the master frame we get

$$MQ_X\frac{d^2x_{MF}}{dt^2} + \Lambda Q_X\frac{dx_{MF}}{dt} + KQ_Xx_{MF} = Q_Ff_{MF} \tag{4}$$

or

$$Q_F^{-1}MQ_X\frac{d^2x_{MF}}{dt^2} + Q_F^{-1}\Lambda Q_X\frac{dx_{MF}}{dt} + Q_F^{-1}KQ_Xx_{MF} = f_{MF} \tag{5}$$

and this means that the mass, damping and stiffness array transform as

$$A_{MF} = Q_F^{-1}AQ_X \tag{6}$$

More explicitly this means

$$\begin{pmatrix} A_{11,MF} & A_{12,MF} \\ A_{11,MF}^T & A_{22,MF} \end{pmatrix} = \begin{pmatrix} I & 0 \\ \vec{r}\wedge & I \end{pmatrix} \begin{pmatrix} A_{11} & A_{12} \\ A_{11}^T & A_{22} \end{pmatrix} \begin{pmatrix} I & -\vec{r}\wedge \\ 0 & I \end{pmatrix} . \tag{7}$$

Note that the transformed array preserve its simmetry. Explicitly

1 MSE tutorial.

Some simple examples of system construction.

Names

1.1 **Simple pendulum.** 4
1.2 **A composite suspended system.** 6
1.3 **A simple attenuation chain.** 7

1.1 Simple pendulum.

We want to construct a very simple mechanical system, namely a pendulum. In the mse framework this can be done by connecting a simple spring to a rigid body.

First of all we need to include some header files which contains the definition of the classes we will use. In this case:

```
#include <mse/MSystem.H>
#include <mse/Spring.H>
#include <mse/RigidBody.H>
#include <mse/PositionActuator.H>
#include <mse/PositionSensor.H>
```

Now we start the main program, and we declare an instance of the MSystem class, which will represent the pendulum.

```
int main()
{
  MSystem pendulum;
  pendulum.SetParameter("Name","Simple pendulum");
```

There are several parameters of the MSystem class which can be altered. We have changed the `Name` parameter, for documentation purposes. Next we declare a spring of given rest length,

```
  double wire_length = 1.0;
  Spring wire = Spring();
  wire.SetParameter("Name","Pendulum wire");
  wire.SetParameter("Separation",&wire_length);
```

and a spherical rigid body of given mass

```
double body_mass = 1.0;
double body_inertia = 1.0;
RigidBody mass = RigidBody(Frame::d3(-1.0));
mass.SetParameter("Name","Pendulum mass");
mass.SetParameter("Mass",&body_mass);
mass.SetParameter("Ixx",&body_inertia);
mass.SetParameter("Iyy",&body_inertia);
mass.SetParameter("Izz",&body_inertia);
```

Note the argument of the constructor, which is optional. It provides informations about the initial positioning of the mass, and can be used to speed-up the search for the working point of the system.

In order to apply a force to our system we need an actuator, so we declare

```
PositionActuator piezo = PositionActuator();
piezo.SetParameter("Name","Piezo");
```

and we need also a sensor to get informations about the evolution of the system

```
PositionSensor pos = PositionSensor();
pos.SetParameter("Name","Sensor");
```

Now we have to construct the system, connecting together all the pieces. We want to move the top of the pendulum wire, and to look at the movement of the mass. So we first attach to the system the position actuator

```
pendulum.Connect(piezo.frame(1), pendulum.frame(0), Frame::Coincident);
```

This instruction generate a rigid connection between the frame 1 of the position actuator and the frame 0 (the unique one) of the mechanical system. The third argument describe the nature of this connection. In this case we want the two frames completely overimposed.

The frame 0 of the position actuator must be attached to the top of the wire. As we can set the relative position of the two frames in the position actuator, we will be able to move the top of the wire with respect to the mechanical system fixed reference frame.

```
pendulum.Connect(piezo.frame(0), wire.frame(1), Frame::Coincident);
```

Now we attach the mass at the bottom of the wire

```
pendulum.Connect(mass.frame(0), wire.frame(0), Frame::Coincident);
```

There is a single frame defined in the `RigidBody class`, which is positioned in the center of mass of the object. As the third argument of the `MSystem::Connect` class is `Frame::Coincident` we attached the wire to the center of mass of the body.

The last step of the construction phase is to attach a position sensor. We want to get the position of the center of mass of the body relative to the reference frame, so we write

```
pendulum.Connect(mass.frame(0), pos.frame(0), Frame::Coincident);
pendulum.Connect(pendulum.frame(0), pos.frame(1), Frame::Coincident);
```

In order to use the mechanical system we constructed we must find its equilibrium position. This can be done with the following sequence of instructions

```
  pendulum.FindWorkingPointInit();

  do {


  } while(pendulum.FindWorkingPoint());

  pendulum.FindWorkingPointEnd();
```

Inside the `do { }` loop we can put some code which can be used to monitor the iterative search of the working point. This loop ends when some accuracy requirement, which depends on the search algorithm, is reached. Both the algorithm and the required accuracy can be entered as parameter of the `MSystem` class.

Now it is possible to start the evolution of the system. This can be done with the loop

```
  pendulum.TimeDynamicsInit();
  do {
    piezo.set_x(sin(pendulum.CurrentTime()));
    cout << pendulum.CurrentTime() << " " << pos.get_x() << "\n";
  } while(pendulum.TimeDynamics());
  TimeDynamicsEnd();
}
```

Inside the loops we put an example of an instruction that set the values of the actuator displacement with some externally provided function (a sinusoidal function), and another that read the values of the current positions. In the current example we print the current x displacement of the pendulum's bottom.

We can also get informations about transfer function. In the following example we print a table with the real and the imaginary part of the horizontal transfer function between the top and the bottom of the pendulum, between 10^{-1} and 10^3 Hz.

```
double logstep = pow(10.0,0.1);
pendulum.FrequencyDynamicsInit();
piezo.set_x(1.0,Real);
piezo.set_x(0.0,Imaginary);
for(double f=0.1;f<=1.0e3;f*=logstep) {
  pendulum.FrequencyDynamics(f);
  cout << f << " " << pos.get_x(Real) << " " << pos.get_x(Imaginary) << "\n";
}
pendulum.FrequencyDynamicsEnd();
```

Note that the initialization of the system in the frequency domain set all actuators to the "no actuation" condition, so it is redundant to set the imaginary part of the piezo amplitude to zero. If many actuators are included in the system the amplitudes and the phases of each of them can be initialized in an arbitrary way.

1.2 A composite suspended system.

The system consist in a rigid body suspended to four wires.

1.3 A simple attenuation chain.

This is a toy model for a complete attenuation chain which provides attenuation both for horizontal and vertical sollecitations. In is made of a triple inverted pendulum, with a double stage pendulum on the top. Each stage of the double pendulum provide vertical attenuation capabilities, obtained with three cantilevered blades mounted horizontally.

2

High level library classes

The reference documentation for the C++ API

Names

2.1 class **Beam** : public TwoNodesElement
Complete model for a beam. 8

2.2 class **Clamp** : public Mse *A constraint between two frames.* 11

2.3 class **ForceActuator** : public TwoNodesElement
A generator of force. 14

2.4 class **ForceSensor** : public TwoNodesElement
A force sensor. 15

2.5 class **Frame** : public Mse *A frame owned by a mechanical object.* 21

2.6 class **MObject** : public Mse *This is the base class for all object that can be composedinto a mechanical system.* 29

2.7 class **Mse** *This is the base class for all the mechanical classes.* 42

2.8 class **MSystem** : public MObject
A mechanical subsystem. 50

2.9 class **PositionActuator** : public TwoNodesElement
A position actuator. 73

2.10 class **PositionSensor** : public TwoNodesElement
A position sensor. 80

2.11 class **RigidBody** : public MObject
A rigid body without internal structure. 84

2.12 class **Spring** : public TwoNodesElement
This is the model for a simple spring. 87

2.13 class **TwoNodesElement** : public MObject
Base class for all the mechanical object with twoframes. 89

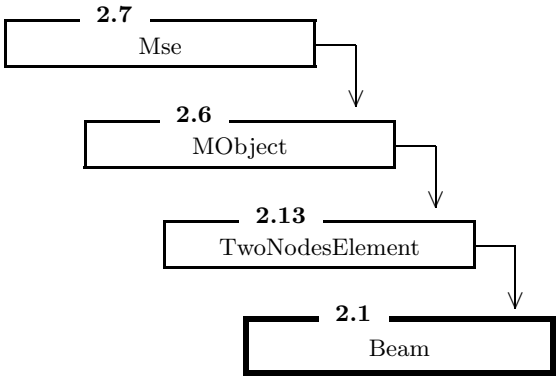
2.14 class **Wire** : public TwoNodesElement
This is the model for a simple wire with torsional stiffness. 91

2.1

class **Beam** : public TwoNodesElement

Complete model for a beam.

Inheritance



Public Members

2.1.1 **Beam** () *Default constructor.* 10

2.1.2 **Beam** (double length, double width0, double height0, double width1, double height1, double young, double poisson, double rho, int elements)
Constructor with parameters. 10

2.1.3 **~Beam** () *Destructor* 10

2.1.4 virtual char* **name** () *Identificative string.* 10

Protected Members

virtual void **UpdateForces** (bool EvalJacobian=true)

virtual void **psdraw** (int i, int j, int lbl)

Private Members

2.1.5 virtual void **FindWorkingPointInit** () *This method must be called before the simulation-phase.* 11

Complete model for a beam.

Parameters:

Parameter name	Parameter type	Description	Default value
`Name`	`TYPE\_STRING64`	The name of the instance of the class	""
`Separation`	`TYPE\_DOUBLE`	Length of the wire at rest	···
`Young Modulus`	`TYPE\_DOUBLE`	Young's modulus of the material	···
`Poisson Ratio`	`TYPE\_DOUBLE`	Poisson's ratio of the material	···
`Initial Width`	`TYPE\_DOUBLE`	Initial width of the beam	···
`Initial Height`	`TYPE\_DOUBLE`	Initial height of the beam	···
`Final Width`	`TYPE\_DOUBLE`	Final width of the beam	···
`Final Height`	`TYPE\_DOUBLE`	Final height of the beam	···
`Density`	`TYPE\_DOUBLE`	Density of the material	···
`Elements`	`TYPE\_INT`	Number of internal elements in the model	···

2.1.1

Beam ()

Default constructor.

Default constructor.

2.1.2

Beam (double length, double width0, double height0, double width1, double height1, double young, double poisson, double rho, int elements)

Constructor with parameters.

Constructor with parameters.

2.1.3

~Beam ()

Destructor

Destructor

2.1.4

virtual char* **name** ()

Identificative string.

Identificative string.

Return Value: the name of the class

2.1.5

virtual void **FindWorkingPointInit** ()

This method must be called before the simulationphase.

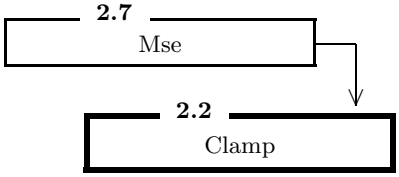
This method must be called before the simulationphase.

2.2

class **Clamp** : public Mse

A constraint between two frames.

Inheritance



Public Members

2.2.1		**Clamp** (Frame *f1, Frame *f2, const mech_frame& t2_1)	*Constructor.*	12
2.2.2	Frame*	**C1** ()	*Actual first frame value.*	12
2.2.3	Frame*	**C2** ()	*Actual second frame value.*	12
2.2.4	mech_frame	**F2_1** ()	*Connection between the frames.*	12
2.2.5	mech_frame	**F1_2** ()	*Connection between the frames.*	13
2.2.6	virtual void	**PutOnStream** (ostream& os)	*Write on stream.*	13
2.2.7	int&	**dof** ()	*Degree of freedom.*	13
2.2.8	virtual char*	**name** ()	*Identificative string.*	13

A constraint between two frames.

Parameters:

Parameter name	Parameter type	Description	Default value
`Name`	`TYPE\_STRING64`	The name of the instance of the class	""

2.2.1

Clamp (Frame *f1, Frame *f2, const mech_frame& t2_1)

Constructor.

Constructor.

Parameters:

`f1`	a pointer to the first frame
`f2`	a pointer to the second frame

2.2.2

Frame* **C1** ()

Actual first frame value.

Actual first frame value.

Return Value: a pointer to the first frame

2.2.3

Frame* **C2** ()

Actual second frame value.

Actual second frame value.

Return Value: a pointer to the second frame.

2.2.4

mech_frame **F2_1** ()

Connection between the frames.

Connection between the frames.

Return Value: the transformation that map the first frame on the second one.

2.2.5

mech_frame **F1_2** ()

Connection between the frames.

Connection between the frames.

Return Value: the transformation that map the second frame on the first one.

2.2.6

virtual void **PutOnStream** (ostream& os)

Write on stream.

Write on stream. This method can be used to write a description of the class instance on a stream.

Parameters: `os` the output stream to use

2.2.7

int& **dof** ()

Degree of freedom.

Degree of freedom.

Return Value: the id of the degree of freedom this clamp belong

2.2.8

virtual char* **name** ()

Identificative string.

Identificative string.

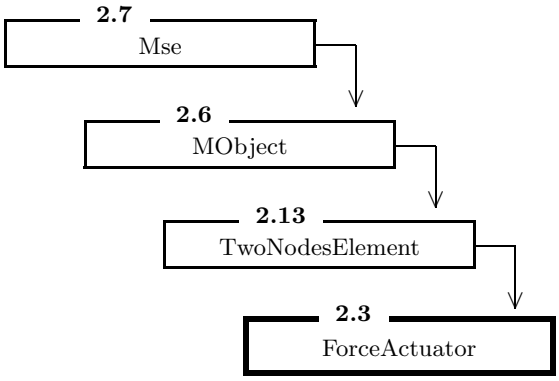
Return Value: the name of the class

2.3

class **ForceActuator** : public TwoNodesElement

A generator of force.

Inheritance



Public Members

2.3.1			**ForceActuator** ()	*Default constructor.*	15
2.3.2	virtual	char*	**name** ()	*Identificative string.*	15
2.3.3	virtual	void	**set_force** (mech_dvec *frc, ComplexType sel=Real)	*Set an internal generalized force to a given value.*	15
	virtual	void	**set_fx** (double f, ComplexType sel=Real)		
	virtual	void	**set_fy** (double f, ComplexType sel=Real)		
	virtual	void	**set_fz** (double f, ComplexType sel=Real)		
	virtual	void	**set_tx** (double t, ComplexType sel=Real)		
	virtual	void	**set_ty** (double t, ComplexType sel=Real)		
	virtual	void	**set_tz** (double t, ComplexType sel=Real)		

Protected Members

virtual	void	**psdraw** (int i, int j, int lbl)
int		**InputMappingArray** (double *m=0)

A generator of force. This object has two frames, which are completely independent during the search for the working point (no forces and torques between them). During the linear evolution the force and the torque that the frame 0 apply on the frame 1 can be specified, giving a way to apply forces on the system.

Parameters:

Parameter name	Parameter type	Description	Default value
`Name`	`TYPE\_STRING64`	The name of the instance of the class	""

2.3.1

ForceActuator ()

Default constructor.

Default constructor.

2.3.2

virtual char* **name** ()

Identificative string.

Identificative string.

Return Value: the name of the class

2.3.3

virtual void **set_force** (mech_dvec *frc, ComplexType sel=Real)

Set an internal generalized force to a given value.

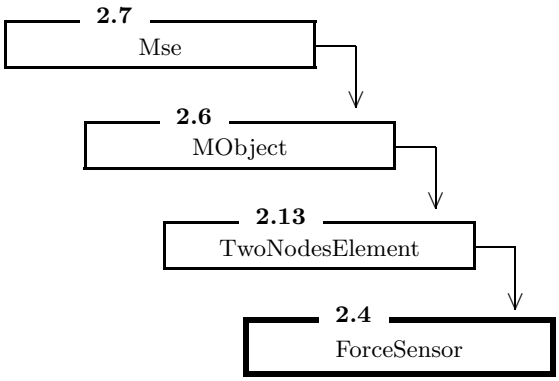
Set an internal generalized force to a given value.

2.4

class **ForceSensor** : public TwoNodesElement

A force sensor.

Inheritance



Public Members

2.4.1 **ForceSensor** () *Default constructor* 17

virtual int **LinearDim** ()

2.4.2 virtual void **get_force** (mech_dvec *frc, ComplexType sel=Real)
Get the forces relative to the working point. ... 17

2.4.3 virtual double **get_fx** (ComplexType sel=Real)
Get the force in the x direction relative to the working point. 17

2.4.4 virtual double **get_fy** (ComplexType sel=Real)
Get the force in the y direction relative to the working point. 18

2.4.5 virtual double **get_fz** (ComplexType sel=Real)
Get the force in the z direction relative to the working point. 18

2.4.6 virtual double **get_tx** (ComplexType sel=Real)
Get the torque in the x direction relative to the working point. 19

2.4.7 virtual double **get_ty** (ComplexType sel=Real)
Get the torque in the y direction relative to the working point. 19

2.4.8 virtual double **get_tz** (ComplexType sel=Real)
Get the torque in the z direction relative to the working point. 19

2.4.9 virtual bool **PositionConstrained** (int f1, int f2)
Redefinition of PositionConstrained. 20

2.4.10 virtual char* **name** () *Identificative string.* 20

Protected Members

void **psdraw** (int i, int j, int lbl)

2.4.11 virtual int **StiffMatrix** (double *m) *This is a redefinion of the Stiff matrix for this system.* 20

virtual int **OutputMappingArray** (double *m=0)

A force sensor. This object has two coincident frames. It is not possible to change the relative position and orientation of them, but the force and the torque that is applied on the frame 1 from the frame 0 can be obtained.

Parameters:

Parameter name	Parameter type	Description	Default value
`Name`	`TYPE\_STRING64`	The name of the instance of the class	""

2.4.1

ForceSensor ()

Default constructor

Default constructor

2.4.2

virtual void **get_force** (mech_dvec *frc, ComplexType sel=Real)

Get the forces relative to the working point.

Get the forces relative to the working point. The values returned by this method depend on the current state of the mechanical system. If we are working in the time domain the current linear variations from the working point forces are returned. These are real numbers, so the case `sel=Imaginary` is not applicable. If we are working in the frequency domain the response at the current frequency is returned, the real part if `sel=Real` and the imaginary part if `sel=Imaginary`.

Parameters:

- `pos` a pointer to the structure which the current displacements are copied to.
- `sel` select in the frequency domain if the real (`sel=Real`) or the imaginary (`sel=Imaginary`) part is returned.

2.4.3

virtual double **get_fx** (ComplexType sel=Real)

Get the force in the x direction relative to the working point.

Get the force in the x direction relative to the working point. The value returned by this method depends on the current state of the mechanical system. If we are working in the time domain the current x force increment relative to the working point condition is returned. This is a real number, so the case `sel=Imaginary` is not applicable.

If we are working in the frequency domain the x response in force at the current frequency is returned, the real part if `sel=Real` and the imaginary part if `sel=Imaginary`.

Return Value: the current x force relative to the working point condition
Parameters: `sel` select in the frequency domain if the real (`sel=Real`) or the imaginary (`sel=Imaginary`) part is returned.

2.4.4

virtual double **get_fy** (ComplexType sel=Real)

Get the force in the y direction relative to the working point.

Get the force in the y direction relative to the working point. The value returned by this method depends on the current state of the mechanical system. If we are working in the time domain the current y force increment relative to the working point condition is returned. This is a real number, so the case `sel=Imaginary` is not applicable. If we are working in the frequency domain the y response in force at the current frequency is returned, the real part if `sel=Real` and the imaginary part if `sel=Imaginary`.

Return Value: the current y force relative to the working point condition
Parameters: `sel` select in the frequency domain if the real (`sel=Real`) or the imaginary (`sel=Imaginary`) part is returned.

2.4.5

virtual double **get_fz** (ComplexType sel=Real)

Get the force in the z direction relative to the working point.

Get the force in the z direction relative to the working point. The value returned by this method depends on the current state of the mechanical system. If we are working in the time domain the current z force increment relative to the working point condition is returned. This is a real number, so the case `sel=Imaginary` is not applicable. If we are working in the frequency domain the z response in force at the current frequency is returned, the real part if `sel=Real` and the imaginary part if `sel=Imaginary`.

Return Value: the current z force relative to the working point condition
Parameters: `sel` select in the frequency domain if the real (`sel=Real`) or the imaginary (`sel=Imaginary`) part is returned.

2.4.6

virtual double **get_tx** (ComplexType sel=Real)

Get the torque in the x direction relative to the working point.

Get the torque in the x direction relative to the working point. The value returned by this method depends on the current state of the mechanical system. If we are working in the time domain the current x torque increment relative to the working point condition is returned. This is a real number, so the case `sel=Imaginary` is not applicable. If we are working in the frequency domain the x response in torque at the current frequency is returned, the real part if `sel=Real` and the imaginary part if `sel=Imaginary`.

Return Value: the current x torque relative to the working point condition
Parameters: `sel` select in the frequency domain if the real (`sel=Real`) or the imaginary (`sel=Imaginary`) part is returned.

2.4.7

virtual double **get_ty** (ComplexType sel=Real)

Get the torque in the y direction relative to the working point.

Get the torque in the y direction relative to the working point. The value returned by this method depends on the current state of the mechanical system. If we are working in the time domain the current y torque increment relative to the working point condition is returned. This is a real number, so the case `sel=Imaginary` is not applicable. If we are working in the frequency domain the y response in torque at the current frequency is returned, the real part if `sel=Real` and the imaginary part if `sel=Imaginary`.

Return Value: the current y torque relative to the working point condition
Parameters: `sel` select in the frequency domain if the real (`sel=Real`) or the imaginary (`sel=Imaginary`) part is returned.

2.4.8

virtual double **get_tz** (ComplexType sel=Real)

Get the torque in the z direction relative to the working point.

Get the torque in the z direction relative to the working point. The value returned by this method depends on the current state of the mechanical system. If we are working in the time domain the current z torque increment relative to the working point condition is returned. This is a real number, so the case `sel=Imaginary` is not applicable. If we are working in the frequency domain the z response in torque at the current frequency is returned, the real part if `sel=Real` and the imaginary part if `sel=Imaginary`.

Return Value: the current z torque relative to the working point condition
Parameters: `sel` select in the frequency domain if the real (`sel=Real`) or the imaginary (`sel=Imaginary`) part is returned.

2.4.9

virtual bool **PositionConstrained** (int f1, int f2)

Redefinition of PositionConstrained.

Redefinition of PositionConstrained. For a force sensor this function must return true.

Return Value: true if the relative position of frames f1,f2 is constrained, 0 otherwhise.
Parameters: `f1` the first frame
`f2` the second frame

2.4.10

virtual char* **name** ()

Identificative string.

Identificative string.

Return Value: the name of the class

2.4.11

virtual int **StiffMatrix** (double *m)

This is a redefinion of the Stiff matrix for this system.

This is a redefinion of the Stiff matrix for this system. There are 18 variables $(\vec{\lambda}, \vec{x}_1, \vec{x}_2)$, where $\vec{x}_i$ correspond to the two clamps and $\vec{\lambda}$ are six internal variables. In block form the stiffness array is

$$K = \begin{pmatrix} 0 & I & -I \\ I & 0 & 0 \\ -I & 0 & 0 \end{pmatrix} \tag{8}$$

where each block is a 6×6 array.

Return Value: the total storage required for the stiffness array (18×18)

Parameters:

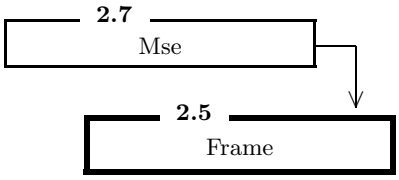
`m` a pointer to the start of the memory location where the stiffness array must be copied. If m is a null pointer no operation is performed, only the total storage required for the stiffness array is returned.

2.5

class **Frame** : public Mse

A frame owned by a mechanical object.

Inheritance



Public Members

2.5.1	class MSystem	**Frame** (MObject *o)	*The constructor.*	23
2.5.2		**~Frame** ()	*The destructor.*	23
2.5.3	virtual void	**PutOnStream** (ostream& os)	*Write on stream.*	23
2.5.4	virtual char*	**name** ()	*Identificative string.*	23
2.5.5	mech_frame*	**frame** ()	*Raw access to frame.*	24
2.5.6	mech_mat*	**ee** ()	*Raw access to orientation.*	24
2.5.7	double&	**ee** (int i, int j)	*Axes orientation.*	24
2.5.8	double&	**o** (int k)	*Origin's coordinate.*	25
2.5.9	double&	**f** (int k)	*Equilibrium position force.*	25
2.5.10	MObject*&	**owner** ()	*The owner of the frame.*	25
2.5.11	static mech_frame	**Coincident**	*A frame coincident with the reference one.*	25
2.5.12	static mech_frame	**dx** (double d)	*A frame aligned with the reference one, but translated in the x direction.*	26
2.5.13	static mech_frame	**dy** (double d)	*A frame aligned with the reference one, but translated in the y direction.*	26
2.5.14	static mech_frame			

		dz (double d)	*A frame aligned with the reference one, but translated in the z direction.*	26
2.5.15	static	mech_frame **rx** (double a)	*A frame with the same origin of the reference one, but rotated aroundthe x axis of a radiants.*	27
2.5.16	static	mech_frame **ry** (double a)	*A frame with the same origin of the reference one, but rotated aroundthe y axis of a radiants.*	27
2.5.17	static	mech_frame **rz** (double a)	*A frame with the same origin of the reference one, but rotated aroundthe z axis of a radiants.*	27
2.5.18	static	mech_frame **dxdydz** (double x, double y, double z)	*A frame aligned with the reference one, but with a general translation.*	28
2.5.19	bool	**positioned** ()	*Positioned flag.*	28
2.5.20	void	**positioned** (bool flag)	*Set the positioned flag to a desired value.*	28

Private Members

2.5.21	mech_dvec*	**frc_link** (mech_dvec *nfrc)	*Link the equilibrium position forces to a memory location.*	29
2.5.22	mech_frame*	**pos_link** (mech_frame *nfrm)	*Link the frame position to a memory location.*	29

A frame owned by a mechanical object. A frame is defined by a vector which defines its origin o and by a reference frame U, which is a set of three orthonormal vectors

$$U \equiv (e_1, e_2, e_3), \qquad e_i \cdot e_j = \delta_{ij} \tag{9}$$

It is convenient also to see U as an orthogonal array, where e_i is the i-th array's column. The rigid connection between two frames can be expressed in the following way:

$$U' = UT \tag{10}$$

$$o' = o + Ud \tag{11}$$

where the orthogonal transformation T and the vector d do not depend on the external reference frame used to specify U,U',o,o'. The couple (T, d) can be interpreted as the frame (U', o') as seen from the reference frame (U, o). We can write

$$T = U^T U' \tag{12}$$

$$d = U^T (o' - o) \tag{13}$$

Parameters:

Parameter name	Parameter type	Description	Default value
`Name`	`TYPE\_STRING64`	The name of the instance of the class	""

2.5.1

class MSystem **Frame** (MObject *o)

The constructor.

The constructor. The orientation and the position of the constructed frame is the default Reference.

Parameters: `o` is the owner of the frame

2.5.2

~Frame ()

The destructor.

The destructor.

2.5.3

virtual void **PutOnStream** (ostream& os)

Write on stream.

Write on stream. This method can be used to write a description of the class instance on a stream.

Parameters: `os` the output stream to use

2.5.4

virtual char* **name** ()

Identificative string.

Identificative string.

Return Value: the name of the class

2.5.5

mech_frame* **frame** ()

Raw access to frame.

Raw access to frame.

Return Value: a pointer

2.5.6

mech_mat* **ee** ()

Raw access to orientation.

Raw access to orientation.

Return Value: a pointer

2.5.7

double& **ee** (int i, int j)

Axes orientation.

Axes orientation. The orientation is described by a ortogonal 3x3 array which transform the fundamental base vectors in the rotated ones. This array can be written in the form

$$R = \left(\begin{array}{c|c|c} & & \\ \vec{e}_1 & \vec{e}_2 & \vec{e}_3 \\ & & \end{array} \right) \tag{14}$$

where $\vec{e}_i$ are the rotated base vectors

Return Value: the value of the (i,j) element of the orthogonal array
Parameters:
`i` the row index of the rotation array
`j` the column index of the rotation array

2.5.8

double& **o** (int k)

Origin's coordinate.

Origin's coordinate.

Return Value: the value of the k-th coordinate of the origin.
Parameters: `k` the coordinate index

2.5.9

double& **f** (int k)

Equilibrium position force.

Equilibrium position force. This is the generalized force which acts on the frame in the equilibrium position.

Return Value: the k-th component of the generalized force
Parameters: `k` is the index of the component of the generalized force.

2.5.10

MObject*& **owner** ()

The owner of the frame.

The owner of the frame.

Return Value: a pointer to the MObject which is the owner of the frame

2.5.11

static mech_frame **Coincident**

A frame coincident with the reference one.

A frame coincident with the reference one. All the components of the o vector are zero, and the U array is the identity.

2.5.12

static mech_frame **dx** (double d)

A frame aligned with the reference one, but translated in the $x$direction.

A frame aligned with the reference one, but translated in the $x$direction. The U array is the identity, and $o = (d, 0, 0)$.

Return Value: the frame (U, o).
Parameters: `d` the displacement

2.5.13

static mech_frame **dy** (double d)

A frame aligned with the reference one, but translated in the $y$direction.

A frame aligned with the reference one, but translated in the $y$direction. The U array is the identity, and $o = (0, d, 0)$.

Return Value: the frame (U, o).
Parameters: `d` the displacement

2.5.14

static mech_frame **dz** (double d)

A frame aligned with the reference one, but translated in the $z$direction.

A frame aligned with the reference one, but translated in the $z$direction. The U array is the identity, and $o = (0, 0, d)$.

Return Value: the frame (U, o).
Parameters: `d` the displacement

2.5.15

static mech_frame **rx** (double a)

A frame with the same origin of the reference one, but rotated aroundthe x axis of a radiants.

A frame with the same origin of the reference one, but rotated aroundthe x axis of a radiants. The components of the o vector are zero, and the array U is

$$U = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos a & \sin a \\ 0 & -\sin a & \cos a \end{pmatrix} \tag{15}$$

Return Value: the frame (U, o).
Parameters: `a` the rotation angle in radiants

2.5.16

static mech_frame **ry** (double a)

A frame with the same origin of the reference one, but rotated aroundthe y axis of a radiants.

A frame with the same origin of the reference one, but rotated aroundthe y axis of a radiants. The components of the o vector are zero, and the array U is

$$U = \begin{pmatrix} \cos a & 0 & \sin a \\ 0 & 1 & 0 \\ -\sin a & 0 & \cos a \end{pmatrix} \tag{16}$$

Return Value: the frame (U, o).
Parameters: `a` the rotation angle in radiants

2.5.17

static mech_frame **rz** (double a)

A frame with the same origin of the reference one, but rotated aroundthe z axis of a radiants.

A frame with the same origin of the reference one, but rotated aroundthe z axis of a radiants. The components of the o vector are zero, and the array U is

$$U = \begin{pmatrix} \cos a & \sin a & 0 \\ -\sin a & \cos a & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{17}$$

Return Value: the frame (U, o).
Parameters: `a` the rotation angle in radiants

2.5.18

static mech_frame **dxdydz** (double x, double y, double z)

A frame aligned with the reference one, but with a general translation.

A frame aligned with the reference one, but with a general translation. The U array is the identity, and the o vector is $o = (dx, dy, dz)$.

Return Value: the frame (U, o).
Parameters: `dx` the x component od the translation
`dy` the y component od the translation
`dz` the z component od the translation

2.5.19

bool **positioned** ()

Positioned flag.

Positioned flag. It is used to indicate that the frame was positioned explicitly.

Return Value: true if the frame was explicitly positioned, false otherwise
See Also: positioned(bool)

2.5.20

void **positioned** (bool flag)

Set the positioned flag to a desired value.

Set the positioned flag to a desired value. The flag is used to indicate that the frame was explicitly positioned.

Parameters: `flag` the desired value of the flag
See Also: positioned()

2.5.21

mech_dvec* **frc_link** (mech_dvec *nfrc)

Link the equilibrium position forces to a memory location.

Link the equilibrium position forces to a memory location.

Return Value: the pointer to the position in the memory where the force values are stored.
Parameters: `nfrc` a pointer to the position in the memory where the force values must be stored.

2.5.22

mech_frame* **pos_link** (mech_frame *nfrm)

Link the frame position to a memory location.

Link the frame position to a memory location.

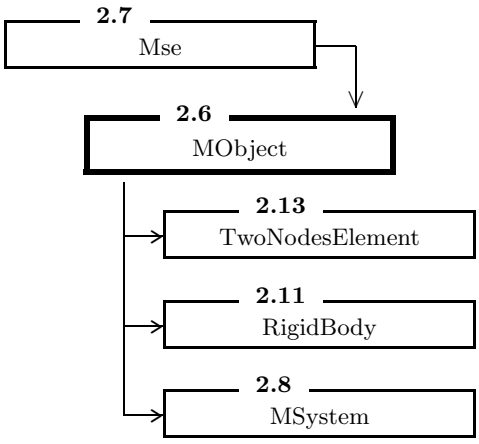
Return Value: the pointer to the position in the memory where the position values are stored.
Parameters: `nfrm` a pointer to the position in the memory where the position values must be stored.

2.6

class **MObject** : public Mse

This is the base class for all object that can be composedinto a mechanical system.

Inheritance



Public Members

2.6.1	class MSystem		**MObject** (int NumberOfFrames) *Default constructor.*	33
2.6.2			**~MObject** () *Destructor.*	34
2.6.3	Frame*		**frame** (int k) *Access to a predefined frame.*	34
2.6.4	int		**frames** () *This method return the number of predefined frames of the object.*	34
2.6.5	virtual	bool	**ValidFrame** (int f) *This method can be used to verify the presence of a given frame.*	34
2.6.6	virtual	bool	**LinearForceConstrained** (int frame) *This method can be used to verify if the linearized generalized force applied to the given frame is constrained or not.*	35
2.6.7	virtual	bool	**LinearPositionConstrained** (int frame1, int frame2) *This method can be used to verify if the linearized relative displacement betweentwo frames is constrained or not.*	35
2.6.8	virtual	bool	**ForceConstrained** (int frame) *This method can be used to verify if the generalized force applied to the given frame is constrained or not.*	35
2.6.9	virtual	bool	**PositionConstrained** (int frame1, int frame2) *This method can be used to verify if the relative displacement betweentwo frames is constrained or not.*	36
2.6.10	virtual	char*	**name** () *Identificative string.*	36
	double		**random_force** ()	
	int		**n_input**	
	int		**n_output**	
	double**		**inputs**	
	double**		**outputs**	

Protected Members

2.6.11	int		**Inputs** () *This method return the number of input variables that can be adjusted in the linear regime.*	36
2.6.12	int		**Outputs** () *This method return the number of output variables that can be readin the linear regime.*	36
	void		**LinkOutput** (int n, int offset, double *base)	
	void		**LinkInput** (int n, int offset, double *base)	
	void		**SetFlag** (unsigned int fl)	
	void		**ResetFlag** (unsigned int fl)	
	bool		**TestFlag** (unsigned int fl)	

2.6.13 virtual void **UpdateForces** (bool EvalJacobian=true)
This method evaluate the forces applied by the mechanicalelement on its nodes in the configuration specified by thecurrent frame coordinates.

37

2.6.14 virtual int **LinearDim** () *This method return the dimension of the mass, damping and stiffnessarray which describe the system in the linearized dynamic regime.* 37

2.6.15 int **InternalDim** () *This method return the number of internal variables added tothe description of this mechanical element.* 38

2.6.16 virtual int **BlockBase** (int f1, int f2) *This method return the base offset of the block in themass, damping or stiffness array which connect two given frames, or a given frame to the internal variables space.* 38

2.6.17 virtual int **BlockRows** (int f1, int f2) *This method return the number of rows in the block in themass, damping or stiffness array which connect two given frames, or a given frame to the internal variables space.* 38

2.6.18 virtual int **BlockCols** (int f1, int f2) *This method return the number of column in the block in themass, damping or stiffness array which connect two given frames, or a given frame to the internal variables space.* 39

2.6.19 virtual int **BlockLeadingDimension** (int f1, int f2)
This method return the leading dimension of the block in themass, damping or stiffness array which connect two given frames, or a given frame to the internal variables space. 39

virtual int **InputMapLeadingDimension** ()

virtual int **InputMapBlockBase** (int f)

virtual int **OutputMapLeadingDimension** ()

virtual int **OutputMapBlockBase** (int f)

2.6.20 virtual int **MassMatrix** (double *m=0)
This method evaluate the current mass matrix for the system. 40

2.6.21 virtual int **DampMatrix** (double *m=0)
This method evaluate the current damping matrix for the system. 40

2.6.22 virtual int **StiffMatrix** (double *m=0)
This method evaluate the current stiffness matrix for the system. 41

virtual int **InputMappingArray** (double *m=0)

virtual int **OutputMappingArray** (double *m=0)

2.6.23 virtual void **ApplyGravity** (double ax, double ay, double az)

This method apply on the element an external gravitationalconstant acceleration field. 41

double& **hess** (int frame1, int frame2, int i, int j)

mech_hess* **hess_link** (mech_hess *nh, int f1, int f2)

virtual void **psdraw** (int i, int j, int lbl)

int **nof**

Frame** **_frames**

2.6.24 mech_hess** **_hessians** *Local stiffness array.* 42

int* **allocated_h**

static double **strength**

static double **rnd**

static int **Internal**

Private Members

2.6.25 virtual void **FindWorkingPointInit** () *This method must be called before the search for theworking point.* 42

2.6.26 virtual void **LinearRegimeInit** () *This method must be called before operations in thelinear regime.* 42

unsigned int **flags**

static int **_count**

This is the base class for all object that can be composedinto a mechanical system. A mechanical object can be seen as a set of N_F frames. Each of them carries six degrees of freedom, which can be parametrized by the three coordinates of the origin and by the three Euler's angles which describe the orientation of the reference frame. A mechanical element must provide:

1. A complete statical description of the system, that is a potential energy W which is function of all the frames.
2. A linearized dynamical description, in term a set of mass, damping and stiffness array in the coordinate space. $6 \times N_F$ coordinates are the linearized version of the general coordinates of the N_F frames. Addictional coordinates can be present, in order to carry informations about internal modes.

A generalized (statical) force is defined as minus the derivative of the potential energy with respect to a coordinate, so on each frame six generalized forces are defined. For the three coordinates of the origin of the frame, the generalized force is the traditional forces f_i. For the variation of the angular coordinates we set a convention. A general rotation array can be written as

$$\Delta(\alpha_x, \alpha_y, \alpha_z) = R_z(\alpha_z) R_y(\alpha_y) R_x(\alpha_x) \tag{18}$$

where $R_i(\alpha)$ is a rotation of angle α around the i-th axis. The potential energy can be written as

$$w(\cdots, \alpha_i^{(k)}, x_i^{(k)}, \cdots) = W(\cdots, \Delta^{(k)} U^{(k)}, o^{(k)} + \delta^{(k)}, \cdots) \tag{19}$$

which we can expand to the first order around the frame (U, o). We obtain

$$f_i^{(k)} = -\frac{\partial w}{\partial \delta_i^{(k)}} = -\frac{\partial W}{\partial o_i^{(k)}} \tag{20}$$

and

$$\tau_i^{(k)} = -\frac{\partial w}{\partial \alpha_i^{(k)}} = -\frac{\partial W}{\partial U_{ab}^{(k)}} \left(\frac{\partial \Delta^{(k)}}{\partial \alpha_i^{(k)}} U^{(k)} \right)_{ab} \tag{21}$$

evaluated for $\alpha_i = \delta_i = 0$.

We are interested also in the second order variation of the energy with respect to the coordinates. This can be used in some algorithm in order to find the equilibrium position, and in the most common case it is simply related to the stiffness array of the system. We get

$$-\frac{\partial^2 w}{\partial \delta_i^{(k)} \partial \delta_j^{(l)}} = -\frac{\partial^2 W}{\partial o_i^{(k)} \partial o_j^{(l)}} = \frac{\partial f_i^{(k)}}{\partial x_j^{(l)}} = \frac{\partial f_j^{(l)}}{\partial x_i^{(k)}} \tag{22}$$

$$-\frac{\partial^2 w}{\partial \delta_i^{(k)} \partial \alpha_j^{(l)}} = -\frac{\partial^2 W}{\partial o_i^{(k)} \partial U_{ab}^{(l)}} \left(\frac{\partial \Delta^{(l)}}{\partial \alpha_j^{(l)}} U^{(l)} \right)_{ab} = \frac{\partial f_i^{(k)}}{\partial \alpha_j^{(l)}} \tag{23}$$

$$-\frac{\partial^2 w}{\partial \alpha_i^{(k)} \partial \delta_j^{(l)}} = -\frac{\partial^2 W}{\partial U_{ab}^{(k)} \partial \delta_j^{(l)}} \left(\frac{\partial \Delta^{(k)}}{\partial \alpha_i^{(k)}} U^{(k)} \right)_{ab} = \frac{\partial \tau_i^{(k)}}{\partial x_j^{(l)}} \tag{24}$$

$$-\frac{\partial^2 w}{\partial \alpha_i^{(k)} \partial \alpha_j^{(l)}} = -\frac{\partial^2 W}{\partial U_{ab}^{(k)} \partial U_{cd}^{(l)}} \left(\frac{\partial \Delta^{(k)}}{\partial \alpha_i^{(k)}} U^{(k)} \right)_{ab} \left(\frac{\partial \Delta^{(l)}}{\partial \alpha_j^{(l)}} U^{(l)} \right)_{cd} - \delta^{kl} \frac{\partial W}{\partial U_{ab}^{(k)}} \left(\frac{\partial^2 \Delta^{(k)}}{\partial \alpha_i^{(k)} \partial \alpha_j^{(k)}} U^{(k)} \right)_{ab} = \frac{\partial \tau_i^{(k)}}{\partial \alpha_j^{(l)}} = \frac{\partial \tau_j^{(l)}}{\partial \alpha_i^{(k)}} \tag{25}$$

The `UpdateForces()` method evaluate these quantities.

The linearized description is of the following form:

$$M \frac{d^2}{dt^2} Y + \Lambda \frac{d}{dt} Y + KY = F \tag{26}$$

and is parametrized by the constant arrays M, Λ, K. The forcing variables F is mapped on the input variables X_{in} by the array A, and the output variables X_{out} are mapped on the internal variables by the array B:

$$F = AX_{in}, \qquad X_{out} = BY \tag{27}$$

It is responsibility of the mechanical object to construct the arrays M,Λ,K,A and B.

Parameters:

Parameter name	Parameter type	Description	Default value
`Name`	`TYPE\_STRING64`	The name of the instance of the class	""

2.6.1

class MSystem **MObject** (int NumberOfFrames)

Default constructor.

Default constructor. There are no user-defined parameters for this class.

Parameters: `NumberOfFrames` is the number of independent frames

2.6.2

~**MObject** ()

Destructor.

Destructor.

2.6.3

Frame* **frame** (int k)

Access to a predefined frame.

Access to a predefined frame. On each mechanical object there is a set of predefined frames, which correspond to independent generalized degrees of freedom. This method can be used to access them.

Return Value: a pointer to the k-th frame
Parameters:
`k` the number of the frame, between 0 and frames()-1 (inclusive)

2.6.4

int **frames** ()

This method return the number of predefined frames of the object.

This method return the number of predefined frames of the object. The frames can be accessed using the frame(int) method.

Return Value: the max number of frames

2.6.5

virtual bool **ValidFrame** (int f)

This method can be used to verify the presence of a given frame.

This method can be used to verify the presence of a given frame.

Return Value: true if the given frame or internal space exists, false otherwhise
Parameters:
`f` is the frame number. If f==MObject::Internal the existence of internal degree of freedom is checked.

2.6.6

virtual bool **LinearForceConstrained** (int frame)

This method can be used to verify if the linearized generalized force applied to the given frame is constrained or not.

This method can be used to verify if the linearized generalized force applied to the given frame is constrained or not. This is the case, for example, for a force actuator.

Return Value: true if the force applied to the frame is constrained, false otherwise.
Parameters: `frame` the frame which must be tested

2.6.7

virtual bool **LinearPositionConstrained** (int frame1, int frame2)

This method can be used to verify if the linearized relative displacement betweentwo frames is constrained or not.

This method can be used to verify if the linearized relative displacement betweentwo frames is constrained or not. This is the case, for example, for a position actuator.

Parameters: `frame1` the first frame which must be tested
`frame2` the second frame which must be tested

2.6.8

virtual bool **ForceConstrained** (int frame)

This method can be used to verify if the generalized force applied to the given frame is constrained or not.

This method can be used to verify if the generalized force applied to the given frame is constrained or not.

Return Value: true if the force applied to the frame is constrained, false otherwise.
Parameters: `frame` the frame which must be tested

2.6.9

virtual bool **PositionConstrained** (int frame1, int frame2)

This method can be used to verify if the relative displacement betweentwo frames is constrained or not.

This method can be used to verify if the relative displacement betweentwo frames is constrained or not.

Parameters:

`frame1`	the first frame which must be tested
`frame2`	the second frame which must be tested

2.6.10

virtual char* **name** ()

Identificative string.

Identificative string.

Return Value: the name of the class

2.6.11

int **Inputs** ()

This method return the number of input variables that can be adjusted in the linear regime.

This method return the number of input variables that can be adjusted in the linear regime.

Return Value: The number of double precision (time domain) or complex (frequency domain) variables that can be adjusted.

2.6.12

int **Outputs** ()

This method return the number of output variables that can be readin the linear regime.

This method return the number of output variables that can be readin the linear regime.

Return Value: The number of double precision (time domain) or complex (frequency domain) variables that can be read.

2.6.13

virtual void **UpdateForces** (bool EvalJacobian=true)

This method evaluate the forces applied by the mechanicalelement on its nodes in the configuration specified by thecurrent frame coordinates.

This method evaluate the forces applied by the mechanicalelement on its nodes in the configuration specified by thecurrent frame coordinates. It must evaluate also, in the same configuration, the first order variation of forces around the current position. This is an array of the following form

$$H = \begin{pmatrix} H_{1,1} & \cdots & H_{1,j} & \cdots & H_{1,N_F} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ H_{i,1} & \cdots & H_{i,j} & \cdots & H_{i,N_F} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ H_{N_F,1} & \cdots & H_{N_F,j} & \cdots & H_{N_F,N_F} \end{pmatrix} \tag{28}$$

where $H_{i,j}$ is a 6×6 block of the following form (N_F=`frames()`).

$$H_{i,j} = \begin{pmatrix} \partial f_x^{(i)}/\partial x^{(j)} & \partial f_x^{(i)}/\partial y^{(j)} & \partial f_x^{(i)}/\partial z^{(j)} & \partial f_x^{(i)}/\partial \alpha_x^{(j)} & \partial f_x^{(i)}/\partial \alpha_y^{(j)} & \partial f_x^{(i)}/\partial \alpha_z^{(j)} \\ \partial f_y^{(i)}/\partial x^{(j)} & \partial f_y^{(i)}/\partial y^{(j)} & \partial f_y^{(i)}/\partial z^{(j)} & \partial f_y^{(i)}/\partial \alpha_x^{(j)} & \partial f_y^{(i)}/\partial \alpha_y^{(j)} & \partial f_y^{(i)}/\partial \alpha_z^{(j)} \\ \partial f_z^{(i)}/\partial x^{(j)} & \partial f_z^{(i)}/\partial y^{(j)} & \partial f_z^{(i)}/\partial z^{(j)} & \partial f_z^{(i)}/\partial \alpha_x^{(j)} & \partial f_z^{(i)}/\partial \alpha_y^{(j)} & \partial f_z^{(i)}/\partial \alpha_z^{(j)} \\ \partial \tau_x^{(i)}/\partial x^{(j)} & \partial \tau_x^{(i)}/\partial y^{(j)} & \partial \tau_x^{(i)}/\partial z^{(j)} & \partial \tau_x^{(i)}/\partial \alpha_x^{(j)} & \partial \tau_x^{(i)}/\partial \alpha_y^{(j)} & \partial \tau_x^{(i)}/\partial \alpha_z^{(j)} \\ \partial \tau_y^{(i)}/\partial x^{(j)} & \partial \tau_y^{(i)}/\partial y^{(j)} & \partial \tau_y^{(i)}/\partial z^{(j)} & \partial \tau_y^{(i)}/\partial \alpha_x^{(j)} & \partial \tau_y^{(i)}/\partial \alpha_y^{(j)} & \partial \tau_y^{(i)}/\partial \alpha_z^{(j)} \\ \partial \tau_z^{(i)}/\partial x^{(j)} & \partial \tau_z^{(i)}/\partial y^{(j)} & \partial \tau_z^{(i)}/\partial z^{(j)} & \partial \tau_z^{(i)}/\partial \alpha_x^{(j)} & \partial \tau_z^{(i)}/\partial \alpha_y^{(j)} & \partial \tau_z^{(i)}/\partial \alpha_z^{(j)} \end{pmatrix} \tag{29}$$

where $f_k^{(i)}$ $\tau_k^{(i)}$ are the components of the force and torque applied on the i-th frame while $x^{(j)}, y^{(j)}, z^{(j)}$ and $\alpha_k^{(j)}$ are the components of the position of the j-th frame. This is a default that set the forces and the Hessian components to zero. If special actions must be done this method has to be redefined in the derived class. The responsibility of evaluating the forces and the Hessian is completely demanded to the MObject. In this way this information can be obtained using analytical expressions, if they are available, or approximate numerical techniques.

Parameters: `EvalJacobian` If true both forces and Jacobian are evaluated (default). If false only forces are evaluated.

2.6.14

virtual int **LinearDim** ()

This method return the dimension of the mass, damping and stiffnessarray which describe the system in the linearized dynamic regime.

This method return the dimension of the mass, damping and stiffnessarray which describe the system in the linearized dynamic regime. This can be greater than 6*frames(), because the array can contain informations abouot internal degree of freedom

Return Value: the dimension of mass, damping and stiffness array

2.6.15

int **InternalDim** ()

This method return the number of internal variables added tothe description of this mechanical element.

This method return the number of internal variables added tothe description of this mechanical element. This is equal to LinearDim()-6*frames()

Return Value: LinearDim-6*frames()

2.6.16

virtual int **BlockBase** (int f1, int f2)

This method return the base offset of the block in themass, damping or stiffness array which connect two given frames,or a given frame to the internal variables space.

This method return the base offset of the block in themass, damping or stiffness array which connect two given frames,or a given frame to the internal variables space.

Return Value: an integer offset which is the start of the required block in the mass, damping or stiffness array, or -1 if the f1,f2 values are incorrect

Parameters:

`f1` is the first frame, or the internal variables if f1=MObject::Internal

`f2` is the second frame, or the internal variables if f2=MObject::Internal

See Also:

BlockRows
BlockCols
BlockLeadingDimension
MassMatrix
DampMatrix
StiffMatrix

2.6.17

virtual int **BlockRows** (int f1, int f2)

This method return the number of rows in the block in themass, damping or stiffness array which connect two given frames,or a given frame to the internal variables space.

This method return the number of rows in the block in themass, damping or stiffness array which connect two given frames,or a given frame to the internal variables space.

Return Value: an integer offset which is the number of rows in the required block in the mass, damping or stiffness array, or -1 if the f1,f2 values are incorrect

Parameters:
- `f1` is the first frame, or the internal variables if f1=MObject::Internal
- `f2` is the second frame, or the internal variables if f2=MObject::Internal

See Also: BlockBase
BlockCols
BlockLeadingDimension
MassMatrix
DampMatrix
StiffMatrix

2.6.18

virtual int **BlockCols** (int f1, int f2)

This method return the number of column in the block in themass, damping or stiffness array which connect two given frames,or a given frame to the internal variables space.

This method return the number of column in the block in themass, damping or stiffness array which connect two given frames,or a given frame to the internal variables space.

Return Value: an integer offset which is the number of columns in the required block in the mass, damping or stiffness array

Parameters:
- `f1` is the first frame, or the internal variables if f1=MObject::Internal
- `f2` is the second frame, or the internal variables if f2=MObject::Internal

See Also: BlockBase
BlockRows
BlockLeadingDimension
MassMatrix
DampMatrix
StiffMatrix

2.6.19

virtual int **BlockLeadingDimension** (int f1, int f2)

This method return the leading dimension of the block in themass, damping or stiffness array which connect two given frames,or a given frame to the internal variables space.

This method return the leading dimension of the block in themass, damping or stiffness array which connect two given frames,or a given frame to the internal variables space.

Return Value: an integer offset which is the leading dimension of the required block in the mass, damping or stiffness array

Parameters:
- `f1` is the first frame, or the internal variables if f1=MObject::Internal
- `f2` is the second frame, or the internal variables if f2=MObject::Internal

See Also: BlockBase, BlockRows, BlockCols, MassMatrix, DampMatrix, StiffMatrix

2.6.20

virtual int **MassMatrix** (double *m=0)

This method evaluate the current mass matrix for the system.

This method evaluate the current mass matrix for the system. The space for the data must be allocated by the calling procedure and the used space is returned. The array has LinearDim() rows and LinearDim() columns. This base definition simply return a matrix filled with zeros. The routine must be redefined for a derivate class.

Return Value: the total storage used in the workspace m

Parameters:
- `m` a pointer to the workspace where the mass matrix must be copied. If m=0 (default) only the total storage is evaluated and returned.

See Also: BlockBase, BlockRows, BlockCols, BlockLeadingDimension, DampMatrix, StiffMatrix

2.6.21

virtual int **DampMatrix** (double *m=0)

This method evaluate the current damping matrix for the system.

This method evaluate the current damping matrix for the system. The space for the data must be allocated by the calling procedure and the used space is returned. The array has LinearDim() rows and LinearDim() columns. This base definition simply return a matrix filled with zeros. The routine must be redefined for a derivate class.

Return Value: the total storage used in the workspace m

Parameters:
- `m` a pointer to the workspace where the mass matrix must be copied. If m=0 (default) only the total storage is evaluated and returned.

See Also:
BlockBase
BlockRows
BlockCols
BlockLeadingDimension
MassMatrix
StiffMatrix

2.6.22

virtual int **StiffMatrix** (double *m=0)

This method evaluate the current stiffness matrix for the system.

This method evaluate the current stiffness matrix for the system. The space for the data must be allocated by the calling procedure and the used space is returned. The array has LinearDim() rows and LinearDim() columns, and must be destroyed by the calling procedure. This base definition simply return a matrix filled with the current hessian. The routine must be redefined for a derivate class.

Return Value: the total storage used in the workspace m
Parameters:
`m` a pointer to the workspace where the mass matrix must be copied. If m=0 (default) only the total storage is evaluated and returned.
See Also:
BlockBase
BlockRows
BlockCols
BlockLeadingDimension
MassMatrix
DampMatrix

2.6.23

virtual void **ApplyGravity** (double ax, double ay, double az)

This method apply on the element an external gravitationalconstant acceleration field.

This method apply on the element an external gravitationalconstant acceleration field. This is a default for a generic mechanical element which does nothing. If special actions must be done this method has to be redefined in the derived class.

Parameters:
`ax` the x component of the gravitational acceleration
`ay` the x component of the gravitational acceleration
`az` the x component of the gravitational acceleration

2.6.24

mech_hess** **_hessians**

Local stiffness array.

Local stiffness array.

2.6.25

virtual void **FindWorkingPointInit** ()

This method must be called before the search for theworking point.

This method must be called before the search for theworking point.

2.6.26

virtual void **LinearRegimeInit** ()

This method must be called before operations in thelinear regime.

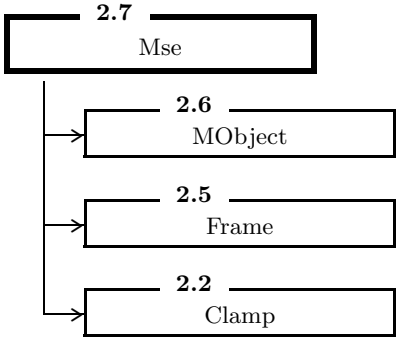
This method must be called before operations in thelinear regime.

2.7

class **Mse**

This is the base class for all the mechanical classes.

Inheritance



Public Members

2.7.1		**Mse** ()	*Default constructor.*	44
2.7.2	virtual	**~Mse** ()	*Destructor.*	44
2.7.3	virtual char*	**name** ()	*Identificative string.*	44
2.7.4	int&	**id** ()	*Unique id.*	45
2.7.5	virtual void	**PutOnStream** (ostream& os)	*Write on stream.*	45
2.7.6	const char*	**uname** ()	*Get the user defined name.*	45
2.7.7	void	**uname** (const char *un)	*Set the user defined name.*	45
2.7.8	int	**Parameters** ()	*Each mechanical element has some parameters thatcan be changed.*	46
2.7.9	char*	**ParametersName** (int k)	*Each settable parameter is indexed with an integerwhich is between 0 and Parameters()-1 inclusive.*	46
2.7.10	int	**ParameterByName** (char *name)	*This method can be used to retrieve the index of a parameter, given itsname.*	46
2.7.11	int	**ParametersType** (int k)	*This call return the type of the parameter indexed by k.*	46
2.7.12	int	**GetParameter** (int k, void *val)	*This method can be used to access the value of a given parameter.*	47
2.7.13	int	**GetParameter** (char *name, void *val)	*This method can be used to access the value of a given parameter.*	47
2.7.14	int	**SetParameter** (int k, void *val)	*This method can be used to set the value of a given parameter.*	48
2.7.15	void	**SetParameter** (char *name, void *val)	*This method can be used to set the value of a given parameter.*	48
2.7.16	void	**operator=** (Mse& el)	*This method set the value of the parameters of this instance ofthe class to the same value of those in the el instance.*	48
2.7.17	static ofstream*	**logfile**	*Log file stream.*	49
2.7.18	static ofstream*	**dumpfile**	*Dump file stream.*	49

Protected Members

2.7.19	int	**AddParameter** (char *name, parameter_type type, void *val)	*This method can be used to add a parameter to the list of settableparameter for the class.*	49

Private Members

2.7.20 void **InitParameters** () *Initialization of the class parameters.* 49

This is the base class for all the mechanical classes.

Parameters:

Parameter name	Parameter type	Description	Default value
`Name`	`TYPE\_STRING64`	The name of the instance of the class	""

2.7.1

Mse ()

Default constructor.

Default constructor.

2.7.2

virtual **~Mse** ()

Destructor.

Destructor.

2.7.3

virtual char* **name** ()

Identificative string.

Identificative string.

Return Value: the name of the class

2.7.4

int& **id** ()

Unique id.

Unique id.

Return Value: an unique integer associated to the class instance

2.7.5

virtual void **PutOnStream** (ostream& os)

Write on stream.

Write on stream. This method can be used to write a description of the class instance on a stream.

Parameters: `os` the output stream to use

2.7.6

const char* **uname** ()

Get the user defined name.

Get the user defined name. This method return a null-terminated string which represent the name a mnemonic identificator for the instance of the object. The identificator can be changed by the user using the uname(char*) method.

Return Value: a pointer to a null-terminated string

2.7.7

void **uname** (const char *un)

Set the user defined name.

Set the user defined name. This method can be used to set the string which identify the instance of the class. The max length of the identification string is 63.

Parameters: `un` the new identification string. If its length is greather than 63 the string is truncated.

2.7.8

int **Parameters** ()

Each mechanical element has some parameters thatcan be changed.

Each mechanical element has some parameters thatcan be changed. This method returns the number of parameters defined for this class.

Return Value: the number of parameters.

2.7.9

char* **ParametersName** (int k)

Each settable parameter is indexed with an integerwhich is between 0 and Parameters()-1 inclusive.

Each settable parameter is indexed with an integerwhich is between 0 and Parameters()-1 inclusive. Each parameter has also a name, which is returned by this method.

Return Value: the name of the parameter or NULL if the parameter is not defined
Parameters: `k` is a parameter index

2.7.10

int **ParameterByName** (char *name)

This method can be used to retrieve the index of a parameter, given itsname.

This method can be used to retrieve the index of a parameter, given itsname.

Return Value: the index of the parameter
Parameters: `name` the name of the parameter

2.7.11

int **ParametersType** (int k)

This call return the type of the parameter indexed by k.

This call return the type of the parameter indexed by k. The available types are:

- `TYPE\_UNDEF`. An undefined parameter.
- `TYPE\_DOUBLE`. A double precision number.
- `TYPE\_INT`. An integer number.
- `TYPE\_STRING64`. An null-terminated string with a maximul length of 63 characters.

Return Value: the type of the parameter
Parameters: `k` is the index of the parameter

2.7.12

int **GetParameter** (int k, void *val)

This method can be used to access the value of a given parameter.

This method can be used to access the value of a given parameter. The value of the parameter is copied in a position specified by a void pointer.

Parameters:

`k`	the index of the parameter
`val`	a pointer to a location where the parameter value must be copied. It is responsibility of the caller that the type of the variable pointed by val is of the correct type.

2.7.13

int **GetParameter** (char *name, void *val)

This method can be used to access the value of a given parameter.

This method can be used to access the value of a given parameter. The value of the parameter is copied in a position specified by a void pointer.

Parameters:

`name`	the name of the parameter.
`val`	a pointer to a location where the parameter value must be copied. It is responsibility of the caller that the type of the variable pointed by val is of the correct type.

2.7.14

int **SetParameter** (int k, void *val)

This method can be used to set the value of a given parameter.

This method can be used to set the value of a given parameter. The value of the parameter is copied from the position specified by a void pointer.

Parameters:

`k`	the parameter index
`val`	a pointer to the location the parameter value must be copied from. It is responsibility of the caller that the type of the variable pointed by val is of the correct type.

2.7.15

void **SetParameter** (char *name, void *val)

This method can be used to set the value of a given parameter.

This method can be used to set the value of a given parameter. The value of the parameter is copied from the position specified by a void pointer.

Parameters:

`name`	the name of the parameter
`val`	a pointer to the location the parameter value must be copied from. It is responsibility of the caller that the type of the variable pointed by val is of the correct type.

2.7.16

void **operator=** (Mse& el)

This method set the value of the parameters of this instance ofthe class to the same value of those in the el instance.

This method set the value of the parameters of this instance ofthe class to the same value of those in the el instance. If a parameter of this instance is not present in the el instance its value is unchanged. Parameters in the el instance which are not in this instance are ignored.

Parameters:

`el`	an instance of the class

2.7.17

static ofstream* **logfile**

Log file stream.

Log file stream.

2.7.18

static ofstream* **dumpfile**

Dump file stream.

Dump file stream.

2.7.19

int **AddParameter** (char *name, parameter_type type, void *val)

This method can be used to add a parameter to the list of settableparameter for the class.

This method can be used to add a parameter to the list of settableparameter for the class. Note that the parameter is added only to the particular instance of the class.

Parameters:

`name`	the name of the new parameter
`type`	the type of the new parameter
`val`	a pointer to an (optional) initial value for the parameter

2.7.20

void **InitParameters** ()

Initialization of the class parameters.

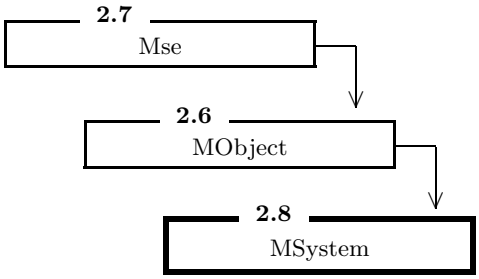
Initialization of the class parameters. This must be called by the constructor.

2.8

class **MSystem** : public MObject

A mechanical subsystem.

Inheritance



Public Members

2.8.1		**MSystem** ()	*Default constructor.*	53
2.8.2		**~MSystem** ()	*Destructor.*	54
2.8.3	void	**Connect** (Frame *f1, Frame *f2, const mech_frame& t12)	*Element connection.*	54
2.8.4	void	**FindWorkingPointInit** ()	*Initialize the search for the working point.*	54
2.8.5	int	**FindWorkingPoint** ()	*Find working point of the system.*	55
2.8.6	void	**FindWorkingPointEnd** ()	*Termination of the search for the working point.*	55
2.8.7	void	**TimeDynamicsInit** ()	*Initialize the linear dynamics.*	55
2.8.8	int	**TimeDynamics** ()	*Linear dynamics evolution.*	56
2.8.9	void	**TimeDynamicsEnd** ()	*Termination of the linear dynamics evolution.*	56
2.8.10	void	**BuildStateSpace** ()	*Construction of the system description in the state space.*	56
2.8.11	double	**CurrentTime** ()	*Return the current value of elapsed time in the linear evolution.*	56
2.8.12	void	**FrequencyDynamicsInit** ()	*Initialization of the evaluation of system response in the frequency domain.*	57
2.8.13	void	**FrequencyDynamics** (double f)	*Evaluate the response of the system in the frequency domain.*	57
2.8.14	void	**FrequencyDynamicsEnd** ()	*Termination of the evaluation of system response in the frequency domain.*	57
2.8.15	virtual char*	**name** ()	*Identificative string.*	57

Private Members

	int	**find_variable** (int dof, int offset)		
	int	**find_variable** (MObject *o, int offset)		
	int	**find_input** (MObject *o, int offset)		
	int	**find_output** (MObject *o, int offset)		
2.8.16	void	**ReduceModel** ()	*This method apply to the system some simplifications.*	58
2.8.17	void	**InitParameters** ()	*Initialize the parameters.*	58
2.8.18	virtual void	**ApplyGravity** (double ax, double ay, double az)	*Apply an external gravity field to the system.*	59
2.8.19	void	**CreateConnectionTable** ()	*Find connections*	59
2.8.20	void	**FindDOF** ()	*Find degrees of freedom.*	59
2.8.21	int	**SetDOF** (int n, list<Frame*>& lst)	*Set a degree of freedom.*	59
2.8.22	int	**DOFs** ()	*Count degrees of freedom.*	60
2.8.23	void	**InitializeDOF** ()	*Initialize the degrees of freedom.*	60
2.8.24	void	**AlignDOF** (int nd)	*Align all the frames in a degree of freedom.*	60
2.8.25	void	**Info** ()	*Print information on the system.*	60
2.8.26	void	**EvalForces** ()	*This method get the description of forces and linear force's variations providedby each mechanical element and compose them in order to evaluate the forces andthe linear force's variation for each degree of freedom which compose the system.*	61
2.8.27	void	**UpdateDOF** (double feps, double teps, double *ferrf, double *terrf)	*Iterative search for the working point.*	62
2.8.28	void	**NewtonRaphson** ()	*Equilibrium point search step.*	62
2.8.29	void	**BuildMassMatrix** ()	*Construct the mass matrix for the system in the current working point.*	62
2.8.30	void	**BuildStiffMatrix** ()	*Construct the stiffness matrix for the system in the current working point.*	63
2.8.31	void	**BuildDampMatrix** ()	*Construct the damping matrix for the system in the current working point.*	64
2.8.32	void	**BuildInputMap** ()	*Construct the input mapping matrix for the system.*	64
2.8.33	void	**BuildOutputMap** ()	*Construct the output mapping matrix for the system.*	64
2.8.34	void	**BuildVarTables** ()	*Construct the frame reference table.*	64
2.8.35	virtual void	**psdraw** (int i, int j, int lbl)		

			Debug.	65
2.8.36	void	**PsDump** (int i, int j, int lbl, char *filen)	*Postscript dump of system.*	65
2.8.37	static int	**_count**	*class counter.*	65
2.8.38	double	**gx**	*x component of gravity.*	65
2.8.39	double	**gy**	*y compponent of gravity.*	65
2.8.40	double	**gz**	*z component of gravity.*	66
2.8.41	list<Clamp*>	**_clamps**	*list of clamps in the system.*	66
2.8.42	list<Clamp*>	**freezing_clamps**	*list of freezing clamps in the system.*	66
2.8.43	list<MObject*>	**objects**	*list of objects in the system.*	66
2.8.44	int	**frames**	*Number of frames in the system.*	66
2.8.45	int	**dofs**	*Number of degrees of freedom in the system.*	67
2.8.46	int	**problem_size**	*Problem size.*	67
2.8.47	mech_frame*	**frames_pos**	*The position of each frame.*	67
2.8.48	mech_dvec*	**frames_frc**	*The force on each frame.*	67
2.8.49	mech_hess*	**frames_hess**	*The Hessian array without constraint (dof = frames).*	67
2.8.50	mech_frame**	**connections**	*The connection array for the frames.*	68
2.8.51	int**	**frame_dof**	*Vector of frame groups (frames which belong to the same dof).*	68
2.8.52	int*	**size_dof**	*A vector which contains the number of frame in each degree of freedom.*	68
2.8.53	int*	**system_frame_dof**	*The vector which contains the frames rigidly connected to the system.*	68
2.8.54	int	**system_size_dof**	*The number of frames which are rigidly connected to the system.*	69
2.8.55	mech_dvec*	**dof_frc**	*Equivalent force on each degree of freedom.*	69
2.8.56	int*	**belong_to_dof**	*Index for fast finding of dof given the frame.*	69
2.8.57	double*	**dof_hess**	*Hessian array for the system.*	69
2.8.58	double*	**delta**	*Evaluated increment of the coordinates of the degrees of freedom.*	69
2.8.59	double*	**wrka**	*Workspace.*	70
2.8.60	double*	**wrkb**	*Workspace.*	70
2.8.61	double*	**linear_wrkspc**	*Workspace.*	70
2.8.62	int	**linear_size**	*Size of the linear problem.*	70
2.8.63	double*	**xvar**	*Dynamical variables 1.*	70
2.8.64	double*	**yvar**	*Dynamical variables 2.*	71
2.8.65	double*	**zvar**	*Dynamical variables 3*	71
2.8.66	double*	**mass_matrix**	*The mass matrix.*	71

2.8.67	double*	**stiff_matrix**	*The stiffness matrix.*	71
2.8.68	double*	**damp_matrix**	*The damping matrix.*	71
2.8.69	int	**linear_input_size**	*Number of inputs in the model **	72
2.8.70	double*	**linear_input**	*The vector of linear input variables **	72
2.8.71	int	**linear_output_size**	*Number of linear output variables **	72
2.8.72	double*	**linear_output**	*Number of outputs in the model **	72
2.8.73	double*	**input_map**	*Input map matrix **	72
2.8.74	double*	**output_map**	*Output map matrix **	73
2.8.75	VarTable*	**var_state**	*Reference table for the system*	73
2.8.76	VarTable*	**var_input**	*Reference table for the inputs*	73
2.8.77	VarTable*	**var_output**	*Reference table for the outputs*	73
	int	**nsv0**		
	int	**nsv1**		
	int	**nsv2**		
	int	**reduced_linear_size**		
	SparseArray	**ss_array**		
	SparseArray	**in_array**		
	SparseArray	**ou_array**		
	double	**current_time**		
	int	**LinearIntegratorType**		

A mechanical subsystem. This class provides a single clamp, which is used specify an inertial reference frame. In this sense it is a MObject, a sort of RigidBody with infinite mass. The main role of this class is to provide a representation for all the system in its different aspects: geometric, static and dynamic. A mechanical system is a set of interconnected mechanical objects.

Parameters:

Parameter name	Parameter type	Description	Default value
`Name`	`TYPE_STRING64`	The name of the instance of the class	""
`Initial strength`	`TYPE_DOUBLE`	Reduction factor for forces (initial value)	
`Time step`	`TYPE_DOUBLE`	Time step for linear evolution	0.001
`gx`	`TYPE_DOUBLE`	The x component of gravitational force	0.0
`gy`	`TYPE_DOUBLE`	The y component of gravitational force	0.0
`gz`	`TYPE_DOUBLE`	The z component of gravitational force	−9.8

2.8.1

MSystem ()

Default constructor.

Default constructor.

2.8.2

~**MSystem** ()

Destructor.

Destructor.

2.8.3

void **Connect** (Frame *f1, Frame *f2, const mech_frame& t12)

Element connection.

Element connection. This method is used to specify a connection between two element of the system. It is designed to completely specify the relative position and orientation of the two elements. This is obtained by enforcing the constraint

$$U_2 = U_1 U_{2,1} \tag{30}$$

and

$$o_2 = o_1 + U_1 o_{2,1} \tag{31}$$

where (U_1, o_1) and (U_2, o_2) are the first and the second frame, and $(U_{2,1}, o_{2,1})$ is a rigid transformation.

Parameters:

`f1`	a pointer to the first frame
`f2`	a pointer to the second frame
`t12`	a pointer to the rigid transformation

2.8.4

void **FindWorkingPointInit** ()

Initialize the search for the working point.

Initialize the search for the working point. There are several actions which must be done:

- The frames are numbered sequentially.
- A vector which contain, in sequential order, all the frames positions is allocated.
- A vector which contain, in sequential order, all the forces which act on the frames is allocated.
- An array which contain the Hessian of the system with rigid constraint removed is allocated.
- The current gravitational force is applied to all the mechanical objects which composes the system
- The degrees of freedom which are irrelevant for the search of working point are freezed, adding additional clamps.

- The degrees of freedom are evaluated and initialized

2.8.5

int **FindWorkingPoint** ()

Find working point of the system.

Find working point of the system. The system is relaxed until the mechanical equilibrium configuration is found. Each call to this method is equivalent to a single relaxation step.

Return Value: 1 if the equilibrium position is found, 0 otherwise.

2.8.6

void **FindWorkingPointEnd** ()

Termination of the search for the working point.

Termination of the search for the working point. If the debugging is activated some informations about the actual configuration are emitted. At the end of the procedure only the information about the equilibrium position of the frames is preserved, in each frame.

2.8.7

void **TimeDynamicsInit** ()

Initialize the linear dynamics.

Initialize the linear dynamics. The working point must be setted correctly. The mass, damping and stiffness array which describe the system are evaluated, and the correct model in the state space is built. Next the model is simplified using the method ReduceModel()

2.8.8

int **TimeDynamics** ()

Linear dynamics evolution.

Linear dynamics evolution. Each call to this method is equivalent to a step in the evolution of the linearized system.

Return Value: 1 if the evolution is finished, 0 otherwise

2.8.9

void **TimeDynamicsEnd** ()

Termination of the linear dynamics evolution.

Termination of the linear dynamics evolution. The memory allocated for the state space description is freed.

2.8.10

void **BuildStateSpace** ()

Construction of the system description in the state space.

Construction of the system description in the state space. We start from a knowledge of the mass, damping and stiffness array and we want to obtain a system of first order linear differential equations. We can write

$$\begin{pmatrix} M(n \times n) & 0(n \times n) \\ 0(n \times n) & I(n \times n) \end{pmatrix} \begin{pmatrix} \frac{dv}{dt} \\ \frac{dx}{dt} \end{pmatrix} \tag{32}$$

2.8.11

double **CurrentTime** ()

Return the current value of elapsed time in the linear evolution.

Return the current value of elapsed time in the linear evolution.

Return Value: the elapsed time

2.8.12

void **FrequencyDynamicsInit** ()

Initialization of the evaluation of system response in the frequency domain.

Initialization of the evaluation of system response in the frequency domain. The working point must be setted correctly.

See Also:
BuildMassMatrix()
BuildStiffMatrix()
BuildDampMatrix()
BuildInputMap()
BuildOutputMap()
ReduceModel()

2.8.13

void **FrequencyDynamics** (double f)

Evaluate the response of the system in the frequency domain.

Evaluate the response of the system in the frequency domain.

Parameters:
`f` the frequency which the response must be evaluated at.

2.8.14

void **FrequencyDynamicsEnd** ()

Termination of the evaluation of system response in the frequency domain.

Termination of the evaluation of system response in the frequency domain.

2.8.15

virtual char* **name** ()

Identificative string.

Identificative string.

Return Value:
the name of the class

2.8.16

void **ReduceModel** ()

This method apply to the system some simplifications.

This method apply to the system some simplifications. The description of the system dynamics is given using a vector $\vec{x}$ of dynamical variables which satisfy the equation

$$(-\omega^2 M - i\omega\Lambda + K)\vec{x} = \vec{f} \tag{33}$$

Here M,Λ and K are the real mass, damping and stiffness arrays of the system. The forcing vector f is connected to the system inputs $\vec{i}$ by the relation

$$\vec{f} = A\vec{i} \tag{34}$$

and the output of the system $\vec{o}$ is a linear function of the internal variables

$$\vec{o} = B\vec{x} \tag{35}$$

We require that there are not vectors v which are at the same time eigenvectors with zero eigenvalues of the M,Λ and K arrays. If these vectors are present (and the input forces, in order to assure consistency) are projected in the orthogonal space. The algorithm is the following:

1. the matrix K is diagonalized with an orthogonal transformation U, and the same transformation is applied on the matrix Λ and M
$$K \to U^T K U, \quad M \to U^T M U, \quad \Lambda \to U^T \Lambda U \tag{36}$$
Also the input and output maps are changed, in order to preserve the external meaning of sensors and actuators. This can be obtained with
$$A \to U^T A, \qquad B \to BU \tag{37}$$
Note that the columns of U are the eigenvectors of K.
2. The kernel of K is identified, and the restriction of M to this subspace is diagonalized with another orthogonal transformation V. The same transformation is applied to Λ, and to the input and output maps.
3. The kernel of the restriction of M is identified, and the restriction of Λ to this subspace is diagonalized with another orthogonal trasformation W.
4. We select the vectors which are zeroed by M,Λ and K among the vectors which are zero eigenvectors of Λ restricted to the last subspace.

2.8.17

void **InitParameters** ()

Initialize the parameters.

Initialize the parameters. Must be called by the constructors.

2.8.18

virtual void **ApplyGravity** (double ax, double ay, double az)

Apply an external gravity field to the system.

Apply an external gravity field to the system.

2.8.19

void **CreateConnectionTable** ()

Find connections

Find connections

2.8.20

void **FindDOF** ()

Find degrees of freedom.

Find degrees of freedom.

2.8.21

int **SetDOF** (int n, list<Frame*>& lst)

Set a degree of freedom.

Set a degree of freedom. The degree of freedom is built from a list of connected frames.

Return Value: the number of degrees of freedom actually defined
Parameters: `n` the degree of freedom to set

2.8.22

int **DOFs** ()

Count degrees of freedom.

Count degrees of freedom.

2.8.23

void **InitializeDOF** ()

Initialize the degrees of freedom.

Initialize the degrees of freedom.

2.8.24

void **AlignDOF** (int nd)

Align all the frames in a degree of freedom.

Align all the frames in a degree of freedom. All the frames which belong to a given degree of freedom are aligned with the master frame.

Parameters: `nd` the degree of freedom that must be aligned.

2.8.25

void **Info** ()

Print information on the system.

Print information on the system.

2.8.26

void **EvalForces** ()

This method get the description of forces and linear force's variations providedby each mechanical element and compose them in order to evaluate the forces andthe linear force's variation for each degree of freedom which compose the system.

This method get the description of forces and linear force's variations providedby each mechanical element and compose them in order to evaluate the forces andthe linear force's variation for each degree of freedom which compose the system. As an example, suppose that there is a two clamp element which is described by the energy W

$$W = W(U_1, o_1, U_2, o_2) \tag{38}$$

where (U_1, o_1) and (U_2, o_2) are the two clamp's position. The element give us, for each configuration, the force and torque which acts on the a-th frame

$$f_i^a = \frac{\partial W}{\partial x_i^a}, \qquad \tau_i^a = \frac{\partial W}{\partial \alpha_i^a} \tag{39}$$

. and the matrix of their linear variations

$$h^{(2)} = \begin{pmatrix} \partial W/(\partial x^1 \partial x^1) & \partial W/(\partial x^1 \partial \alpha^1) & \partial W/(\partial x^1 \partial x^2) & \partial W/(\partial x^1 \partial \alpha^2) \\ \partial W/(\partial \alpha^1 \partial x^1) & \partial W/(\partial \alpha^1 \partial \alpha^1) & \partial W/(\partial x^1 \partial x^2) & \partial W/(\partial \alpha^1 \partial \alpha^2) \\ \partial W/(\partial x^2 \partial x^1) & \partial W/(\partial x^2 \partial \alpha^1) & \partial W/(\partial x^2 \partial x^2) & \partial W/(\partial x^2 \partial \alpha^2) \\ \partial W/(\partial \alpha^2 \partial x^1) & \partial W/(\partial \alpha^2 \partial \alpha^1) & \partial W/(\partial \alpha^2 \partial x^2) & \partial W/(\partial \alpha^2 \partial \alpha^2) \end{pmatrix} \tag{40}$$

In order to add these contribution to the total force and to the total stiffness array we must express them as functions of the coordinates of the relevant degree of freedom, which we call $\hat{x}$,$\hat{\alpha}$. We obtain $(X = (x, \alpha))$

$$\frac{\partial W}{\partial \hat{X}_i^a} = H_{ij}^a \frac{\partial W}{\partial X_i^a} \tag{41}$$

and

$$\frac{\partial W}{\partial \hat{X}_i^a \partial \hat{X}_j^b} = H_{il}^a H_{jm}^b \frac{\partial W}{\partial X_l^a \partial X_m^b} + \delta_{ab} \frac{\partial X_k^a}{\partial \hat{X}_i^a \hat{X}_j^a} \frac{\partial W}{\partial X_k^a} \tag{42}$$

where

$$H_{ij}^a = \frac{\partial X_j^a}{\partial \hat{X}_i^a} \ . \tag{43}$$

For a rigid connection between the frame of the mechanical object and the frame of the degree of freedom we get

$$H_{ij}^a = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & -z^a & y^a & 1 & 0 & 0 \\ z^a & 0 & -x^a & 0 & 1 & 0 \\ -y^a & x^a & 0 & 0 & 0 & 1 \end{pmatrix} \tag{44}$$

where (x^a, y^a, z^a) is the separation between the object's frame and degree of freedom's frame. We have also

$$\frac{\partial X_k^a}{\partial \hat{X}_i^a \hat{X}_j^a} \frac{\partial W}{\partial X_k^a} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -f_y^a y^a - f_z^a z^a & -f_y^a x^a & f_z^a x^a \\ 0 & 0 & 0 & -f_y^a x^a & -f_x^a x^a - f_z^a z^a & -f_z^a y^a \\ 0 & 0 & 0 & f_z^a x^a & -f_z^a y^a & -f_x^a x^a - f_y^a y^a \end{pmatrix} \tag{45}$$

We note that this last term, owing to the non commutativity of the rotations, depend on the ordering in which we choose to apply them. Our convention is to apply in order three rotations around z, y, and x axis:

$$R_x(\alpha_x)R_y(\alpha_y)R_z(\alpha_z) \tag{46}$$

In this way to the first order we get the usual interpretation for τ_i as a torque on the i-th axis. If we choose a different ordering this interpretation does not change, but the expression (45) differ for terms proportional to the total torque applied to the degree of freedom, which are zero at the equilibrium working point. Note that the Equation 41 gives the well known rule for the transformation of forces and torques

$$\hat{f}_i^a = f_i^a, \qquad \hat{\tau}_i^a = \tau_i^a + (r^a \wedge f^a)_i \tag{47}$$

2.8.27

void **UpdateDOF** (double feps, double teps, double *ferrf, double *terrf)

Iterative search for the working point.

Iterative search for the working point. Update the degrees of freedom in order to approach the equilibrium point.

2.8.28

void **NewtonRaphson** ()

Equilibrium point search step.

Equilibrium point search step. This method evaluate the variations that must be applied to the coordinates of the degrees of freedom in order to approach the equilibrium point defined by the equations

$$F_i^P(X) = 0 \quad \forall P, i \tag{48}$$

In order to do that the Newton–Raphson algorithm is used.

2.8.29

void **BuildMassMatrix** ()

Construct the mass matrix for the system in the current working point.

Construct the mass matrix for the system in the current working point. This is an intermediate representation, and we do not worry about the problem of reducing the bandwidth of the array. The 6×6 block which describe the mass submatrix between the i-th and the j-th degree of freedom start with offset `6*(i*linear_size+j)` and

leading dimension `6*linear_size`. The degrees of freedom between `0` and `problem_size-1`, inclusive, correspond to the external degrees of freedom of the system. If `linear_size>problem_size` there are also internal degrees of freedom, numbered between `problem_size` and `linear_size-1`, inclusive.

See Also: BuildStiffMatrix()
BuildDampMatrix()

2.8.30

void **BuildStiffMatrix** ()

Construct the stiffness matrix for the system in the current working point.

Construct the stiffness matrix for the system in the current working point. The addressing scheme is the same of the mass matrix. The following is the algorithm used:

1. The method `StiffMatrix(double*)` is called for every mechanical object which composes the system. The stiffness array returned by this method is of the form

$$K = \left(\begin{array}{c|c|c|c} K_{00} & K_{01} & \cdots & K_{0n} \\ \hline K_{10} & K_{11} & \cdots & K_{1n} \\ \hline \cdots & \cdots & \cdots & \cdots \\ \hline K_{n0} & K_{n1} & \cdots & K_{nn} \end{array}\right) \tag{49}$$

   where K_{ij}, $i,j > 0$ are 6×6 blocks which describe the interaction between frame i and frame j. K_{00} is a block of dimension $n_i \times n_i$ which describe the interaction between internal variables, and K_{0i} (K_{i0}) is a $n_i \times 6$ ($6 \times n_i$) block which describe the interaction bewteen internal variables and the i-th frame.

2. Each frame belongs to a degree of freedom. The variation of i-th frame is connected to the variation of the degree of freedom by an array R_i:

$$\delta\vec{x}_i = R_i \delta\vec{x}_{DOF} \tag{50}$$

   The stiffness array must be transformed in order to describe the interation between degrees of freedom in the following way

$$K = \left(\begin{array}{c|c|c|c} K_{00} & K_{01}R_1 & \cdots & K_{0n}R_n \\ \hline R_1^T K_{10} & R_1^T K_{11} R_1 & \cdots & R_1^T K_{1n} R_n \\ \hline \cdots & \cdots & \cdots & \cdots \\ \hline R_n^T K_{n0} & R_n^T K_{n1} R_1 & \cdots & R_n^T K_{nn} R_n \end{array}\right) \tag{51}$$

3. After the transformation each entry in K is added to the appropriate entry in the general stiffness array of the system.

See Also: BuildMassMatrix()
BuildDampMatrix()

2.8.31

void **BuildDampMatrix** ()

Construct the damping matrix for the system in the current working point.

Construct the damping matrix for the system in the current working point. The addressing scheme is the same of the mass matrix.

See Also: BuildMassMatrix()
BuildStiffMatrix()

2.8.32

void **BuildInputMap** ()

Construct the input mapping matrix for the system.

Construct the input mapping matrix for the system.

2.8.33

void **BuildOutputMap** ()

Construct the output mapping matrix for the system.

Construct the output mapping matrix for the system.

2.8.34

void **BuildVarTables** ()

Construct the frame reference table.

Construct the frame reference table.

2.8.35

virtual void **psdraw** (int i, int j, int lbl)

Debug.

Debug. Write a postscript representation of the system.

2.8.36

void **PsDump** (int i, int j, int lbl, char *filen)

Postscript dump of system.

Postscript dump of system.

2.8.37

static int **_count**

class counter.

class counter.

2.8.38

double **gx**

x component of gravity.

x component of gravity.

2.8.39

double **gy**

y compponent of gravity.

y compponent of gravity.

2.8.40

double **gz**

z component of gravity.

z component of gravity.

2.8.41

list<Clamp*> **_clamps**

list of clamps in the system.

list of clamps in the system.

2.8.42

list<Clamp*> **freezing_clamps**

list of freezing clamps in the system.

list of freezing clamps in the system.

2.8.43

list<MObject*> **objects**

list of objects in the system.

list of objects in the system.

2.8.44

int **frames**

Number of frames in the system.

Number of frames in the system.

2.8.45

int **dofs**

Number of degrees of freedom in the system.

Number of degrees of freedom in the system.

2.8.46

int **problem_size**

Problem size.

Problem size. The number of degrees of freedom without internal ones.

2.8.47

mech_frame* **frames_pos**

The position of each frame.

The position of each frame.

2.8.48

mech_dvec* **frames_frc**

The force on each frame.

The force on each frame.

2.8.49

mech_hess* **frames_hess**

The Hessian array without constraint (dof = frames).

The Hessian array without constraint (dof = frames). This is a square array of mech_hess structures of dimension frames x frames.

2.8.50

mech_frame** **connections**

The connection array for the frames.

The connection array for the frames. This is a square array of pointer to mech_hess structures. If the entry on row i and column j is the null pointer there are no constraint between frame i and frame j. This means that the two frames belong to different degrees of freedom. In the other case the entry point to the transformation that connect frame i and frame j.

2.8.51

int** **frame_dof**

Vector of frame groups (frames which belong to the same dof).

Vector of frame groups (frames which belong to the same dof). This is a vector of vectors of integers. The first integer of each vector is the index of the master frame of the degree of freedom. The other are the indices of the frames rigidly connected to it. To find the connection use the connections array.

2.8.52

int* **size_dof**

A vector which contains the number of frame in each degree of freedom.

A vector which contains the number of frame in each degree of freedom. The value of size_dof[i] is the length of the vector if integers frame_dof[i].

2.8.53

int* **system_frame_dof**

The vector which contains the frames rigidly connected to the system.

The vector which contains the frames rigidly connected to the system. The first element is the index of the frame 0 of the system.

2.8.54

int **system_size_dof**

The number of frames which are rigidly connected to the system.

The number of frames which are rigidly connected to the system. It is the size of the vector system_frame_dof.

2.8.55

mech_dvec* **dof_frc**

Equivalent force on each degree of freedom.

Equivalent force on each degree of freedom. dof_frc[i] contains the total force which acts on the i-th degree of freedom.

2.8.56

int* **belong_to_dof**

Index for fast finding of dof given the frame.

Index for fast finding of dof given the frame. belong_to_dof[i] contains the number of the degree of freedom which contains the frame i.

2.8.57

double* **dof_hess**

Hessian array for the system.

Hessian array for the system.

2.8.58

double* **delta**

Evaluated increment of the coordinates of the degrees of freedom.

Evaluated increment of the coordinates of the degrees of freedom.

2.8.59

double* **wrka**

Workspace.

Workspace.

2.8.60

double* **wrkb**

Workspace.

Workspace.

2.8.61

double* **linear_wrkspc**

Workspace.

Workspace.

2.8.62

int **linear_size**

Size of the linear problem.

Size of the linear problem. In the general case can be different from `problem_size`, because some internal degrees of freedom can be added to the representation.

2.8.63

double* **xvar**

Dynamical variables 1.

Dynamical variables 1.

2.8.64

double* **yvar**

Dynamical variables 2.

Dynamical variables 2.

2.8.65

double* **zvar**

Dynamical variables 3

Dynamical variables 3

2.8.66

double* **mass_matrix**

The mass matrix.

The mass matrix. It is a matrix of linear_size rows and linear_size columns.

2.8.67

double* **stiff_matrix**

The stiffness matrix.

The stiffness matrix. It is a matrix of linear_size rows and linear_size columns.

2.8.68

double* **damp_matrix**

The damping matrix.

The damping matrix. It is a matrix of linear_size rows and linear_size columns.

2.8.69

int **linear_input_size**

*Number of inputs in the model **

Number of inputs in the model *

2.8.70

double* **linear_input**

*The vector of linear input variables **

The vector of linear input variables *

2.8.71

int **linear_output_size**

*Number of linear output variables **

Number of linear output variables *

2.8.72

double* **linear_output**

*Number of outputs in the model **

Number of outputs in the model *

2.8.73

double* **input_map**

*Input map matrix **

Input map matrix *

2.8.74

double* **output_map**

*Output map matrix **

Output map matrix *

2.8.75

VarTable* **var_state**

Reference table for the system

Reference table for the system

2.8.76

VarTable* **var_input**

Reference table for the inputs

Reference table for the inputs

2.8.77

VarTable* **var_output**

Reference table for the outputs

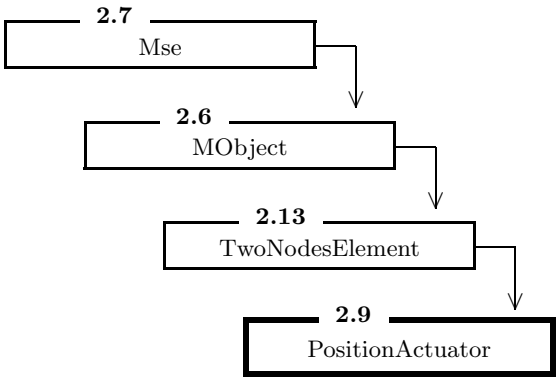
Reference table for the outputs

2.9

class **PositionActuator** : public TwoNodesElement

A position actuator.

Inheritance



Public Members

2.9.1 **PositionActuator** () *Default constructor.* 75

2.9.2 virtual int **LinearDim** () *The number of variables in the object.* 75

2.9.3 void **set_position** (mech_dvec *pos, ComplexType sel=Real)
This method set the six linear displacement of the actuator. 76

2.9.4 void **set_x** (double x, ComplexType sel=Real)
This method set the x displacement of the actuator (the realor the imaginary part, when applicable, accordingly to the second argument's value). 76

2.9.5 void **set_y** (double y, ComplexType sel=Real)
This method set the y displacement of the actuator (the realor the imaginary part, when applicable, accordingly to the second argument's value). 77

2.9.6 void **set_z** (double z, ComplexType sel=Real)
This method set the z displacement of the actuator (the realor the imaginary part, when applicable, accordingly to the second argument's value). 77

2.9.7 void **set_ax** (double ax, ComplexType sel=Real)
This method set the x angular displacement of the actuator (the realor the imaginary part, when applicable, accordingly to the second argument's value). 77

2.9.8 void **set_ay** (double ay, ComplexType sel=Real)
This method set the y angular displacement of the actuator (the realor the imaginary part, when applicable, accordingly to the second argument's value). 78

2.9.9 void **set_az** (double az, ComplexType sel=Real)

				This method set the z angular displacement of the actuator (the realor the imaginary part, when applicable, accordingly to the second argument's value).	78
2.9.10	void		**FindWorkingPointInit** ()	*Initialize internal structures before the working point search.*	78
2.9.11	virtual	void	**LinearRegimeInit** ()	*Initialize internal structures before the time domain linearized evolution or the frequency domain transfer function evaluation.*	79
2.9.12	virtual	char*	**name** ()	*Identificative string.*	79
2.9.13	virtual	bool	**PositionConstrained** (int f1, int f2)	*Redefinition of PositionConstrained.*	79
2.9.14	virtual	int	**StiffMatrix** (double *m)	*This is a redefinion of the Stiff matrix for this system.*	79

Protected Members

virtual	void	**psdraw** (int i, int j, int lbl)
virtual	int	**InputMappingArray** (double *m)

A position actuator. This object has two coincident frames, whose relative position and orientation cannot be changed during the working point search. During the linear dynamics phase the relative position and orientation between frame 0 and frame 1 can be altered. In this way we can apply an action on the system.

Parameters:

Parameter name	Parameter type	Description	Default value
`Name`	`TYPE\_STRING64`	The name of the instance of the class	""

2.9.1

PositionActuator ()

Default constructor.

Default constructor.

2.9.2

virtual int LinearDim ()

The number of variables in the object.

The number of variables in the object. There are two frames and six internal variables (lagrange multipliers), so the total number of variables is 18.

Return Value: the number of variables used to describe the system in the linear regime.

2.9.3

void **set_position** (mech_dvec *pos, ComplexType sel=Real)

This method set the six linear displacement of the actuator.

This method set the six linear displacement of the actuator.

Parameters:

`pos` a pointer to the structure containing the six displacements of the actuator (the real or the imaginary part, when applicable, accordingly to the second argument's value).

`sel` select the real or the imaginary part of the six displacements. If sel=Real (default) the real part of the displacements is setted. If sel=Imaginary the imaginary part of the displacements is setted.

2.9.4

void **set_x** (double x, ComplexType sel=Real)

This method set the x displacement of the actuator (the realor the imaginary part, when applicable, accordingly to the second argument's value).

This method set the x displacement of the actuator (the realor the imaginary part, when applicable, accordingly to the second argument's value).

Parameters:

`x` is the x displacement of the actuator

`sel` select the real or the imaginary part of the displacement. If sel=Real (default) the real part of the displacement is setted. If sel=Imaginary the imaginary part of the displacement is setted.

2.9.5

void **set_y** (double y, ComplexType sel=Real)

This method set the y displacement of the actuator (the realor the imaginary part, when applicable, accordingly to the second argument's value).

This method set the y displacement of the actuator (the realor the imaginary part, when applicable, accordingly to the second argument's value).

Parameters:

`y`	is the y displacement of the actuator
`sel`	select the real or the imaginary part of the displacement. If sel=Real (default) the real part of the displacement is setted. If sel=Imaginary the imaginary part of the displacement is setted.

2.9.6

void **set_z** (double z, ComplexType sel=Real)

This method set the z displacement of the actuator (the realor the imaginary part, when applicable, accordingly to the second argument's value).

This method set the z displacement of the actuator (the realor the imaginary part, when applicable, accordingly to the second argument's value).

Parameters:

`z`	is the z displacement of the actuator
`sel`	select the real or the imaginary part of the displacement. If sel=Real (default) the real part of the displacement is setted. If sel=Imaginary the imaginary part of the displacement is setted.

2.9.7

void **set_ax** (double ax, ComplexType sel=Real)

This method set the x angular displacement of the actuator (the realor the imaginary part, when applicable, accordingly to the second argument's value).

This method set the x angular displacement of the actuator (the realor the imaginary part, when applicable, accordingly to the second argument's value).

Parameters:

`ax`	is the x angular displacement of the actuator
`sel`	select the real or the imaginary part of the displacement. If sel=Real (default) the real part of the displacement is setted. If sel=Imaginary the imaginary part of the displacement is setted.

2.9.8

void **set_ay** (double ay, ComplexType sel=Real)

This method set the y angular displacement of the actuator (the realor the imaginary part, when applicable, accordingly to the second argument's value).

This method set the y angular displacement of the actuator (the realor the imaginary part, when applicable, accordingly to the second argument's value).

Parameters:

`ay`	is the y angular displacement of the actuator
`sel`	select the real or the imaginary part of the displacement. If sel=Real (default) the real part of the displacement is setted. If sel=Imaginary the imaginary part of the displacement is setted.

2.9.9

void **set_az** (double az, ComplexType sel=Real)

This method set the z angular displacement of the actuator (the realor the imaginary part, when applicable, accordingly to the second argument's value).

This method set the z angular displacement of the actuator (the realor the imaginary part, when applicable, accordingly to the second argument's value).

Parameters:

`az`	is the z angular displacement of the actuator
`sel`	select the real or the imaginary part of the displacement. If sel=Real (default) the real part of the displacement is setted. If sel=Imaginary the imaginary part of the displacement is setted.

2.9.10

void **FindWorkingPointInit** ()

Initialize internal structures before the working point search.

Initialize internal structures before the working point search.

2.9.11

virtual void **LinearRegimeInit** ()

Initialize internal structures before the time domain linearized evolution or the frequency domain transfer function evaluation.

Initialize internal structures before the time domain linearized evolution or the frequency domain transfer function evaluation.

2.9.12

virtual char* **name** ()

Identificative string.

Identificative string.

Return Value: the name of the class

2.9.13

virtual bool **PositionConstrained** (int f1, int f2)

Redefinition of PositionConstrained.

Redefinition of PositionConstrained. For a position actuator this function must return true.

Return Value: true if the relative position of frames f1,f2 is constrained, 0 otherwhise.
Parameters:
`f1` the first frame
`f2` the second frame

2.9.14

virtual int **StiffMatrix** (double *m)

This is a redefinion of the Stiff matrix for this system.

This is a redefinion of the Stiff matrix for this system. There are 18 variables $(\vec{lambda}, \vec{x}_1, \vec{x}_2)$, where $\vec{x}_i$ correspond to the two clamps and $\vec{\lambda}$ are six internal variables. In block form the stiffness array is

$$K = \begin{pmatrix} 0 & I & -I \\ I & 0 & 0 \\ -I & 0 & 0 \end{pmatrix} \tag{52}$$

where each block is a 6×6 array.

Return Value: the total storage required for the stiffness array (18×18)

Parameters:

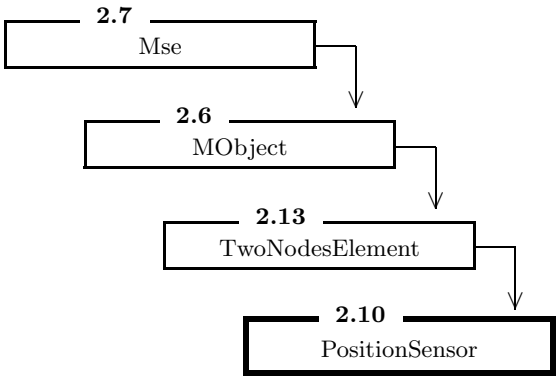
`m` a pointer to the start of the memory location where the stiffness array must be copied. If m is a null pointer no operation is performed, only the total storage required for the stiffness array is returned.

2.10

class **PositionSensor** : public TwoNodesElement

A position sensor.

Inheritance



Public Members

2.10.1 **PositionSensor** () *Default constructor* 81

2.10.2 void **get_position** (mech_dvec *pos, ComplexType sel=Real)
Get the positions of the sensor frame relativeto the working point. 81

2.10.3 double **get_x** (ComplexType sel=Real)
Get the x position of the sensor frame relativeto the working point. 82

2.10.4 double **get_y** (ComplexType sel=Real)
Get the y position of the sensor frame relativeto the working point. 82

2.10.5 double **get_z** (ComplexType sel=Real)
Get the z position of the sensor frame relativeto the working point. 82

2.10.6 double **get_ax** (ComplexType sel=Real)
Get the angular position of the sensor frame relativeto the working point. 83

2.10.7 double **get_ay** (ComplexType sel=Real)

Get the angular position of the sensor frame relativeto the working point. 83

2.10.8 double **get_az** (ComplexType sel=Real)
Get the angular position of the sensor frame relativeto the working point. 84

2.10.9 virtual char* **name** () *Identificative string.* 84

Protected Members

void **psdraw** (int i, int j, int lbl)

virtual int **OutputMappingArray** (double *m=0)

A position sensor. This object has two frames, which are completely independent, in the sense that no forces and torques can be applied between them. The relative position and orientation can be obtained.

Parameters:

Parameter name	Parameter type	Description	Default value
`Name`	`TYPE\_STRING64`	The name of the instance of the class	""

2.10.1

PositionSensor ()

Default constructor

Default constructor

2.10.2

void **get_position** (mech_dvec *pos, ComplexType sel=Real)

Get the positions of the sensor frame relativeto the working point.

Get the positions of the sensor frame relativeto the working point. The values returned by this method depend on the current state of the mechanical system. If we are working in the time domain the current linear displacements from the working point position are returned. These are real numbers, so the case `sel=Imaginary` is not applicable. If we are working in the frequency domain the responses at the current frequency are returned, the real part if `sel=Real` and the imaginary part if `sel=Imaginary`.

Parameters:

`pos` a pointer to the structure which the current displacements are copied to.

`sel` select in the frequency domain if the real (`sel=Real`) or the imaginary (`sel=Imaginary`) part is returned.

2.10.3

double **get_x** (ComplexType sel=Real)

Get the x position of the sensor frame relativeto the working point.

Get the x position of the sensor frame relativeto the working point. The value returned by this method depends on the current state of the mechanical system. If we are working in the time domain the current linear x displacement from the working point position is returned. This is a real numbers, so the case `sel=Imaginary` is not applicable. If we are working in the frequency domain the x response at the current frequency is returned, the real part if `sel=Real` and the imaginary part if `sel=Imaginary`.

Return Value: the current x displacement relative to the working point position
Parameters: `sel` select in the frequency domain if the real (`sel=Real`) or the imaginary (`sel=Imaginary`) part is returned.

2.10.4

double **get_y** (ComplexType sel=Real)

Get the y position of the sensor frame relativeto the working point.

Get the y position of the sensor frame relativeto the working point. The value returned by this method depends on the current state of the mechanical system. If we are working in the time domain the current linear y displacement from the working point position is returned. This is a real numbers, so the case `sel=Imaginary` is not applicable. If we are working in the frequency domain the y response at the current frequency is returned, the real part if `sel=Real` and the imaginary part if `sel=Imaginary`.

Return Value: the current y displacement relative to the working point position
Parameters: `sel` select in the frequency domain if the real (`sel=Real`) or the imaginary (`sel=Imaginary`) part is returned.

2.10.5

double **get_z** (ComplexType sel=Real)

Get the z position of the sensor frame relativeto the working point.

Get the z position of the sensor frame relativeto the working point. The value returned by this method depends on the current state of the mechanical system. If we are working in the time domain the current linear z displacement from the working point position is returned. This is a real numbers, so the case `sel=Imaginary` is not applicable. If we are working in the frequency domain the z response at the current frequency is returned, the real part if `sel=Real` and the imaginary part if `sel=Imaginary`.

Return Value: the current z displacement relative to the working point position
Parameters: `sel` select in the frequency domain if the real (`sel=Real`) or the imaginary (`sel=Imaginary`) part is returned.

2.10.6

double **get_ax** (ComplexType sel=Real)

Get the angular position of the sensor frame relativeto the working point.

Get the angular position of the sensor frame relativeto the working point. The value returned by this method depends on the current state of the mechanical system. If we are working in the time domain the current angular displacement around the x axis, relative to the working point position, is returned. This is a real numbers, so the case `sel=Imaginary` is not applicable. If we are working in the frequency domain the angular response around the x axis at the current frequency is returned, the real part if `sel=Real` and the imaginary part if `sel=Imaginary`.

Return Value: the current angular displacement around the x axis relative to the working point position
Parameters: `sel` select in the frequency domain if the real (`sel=Real`) or the imaginary (`sel=Imaginary`) part is returned.

2.10.7

double **get_ay** (ComplexType sel=Real)

Get the angular position of the sensor frame relativeto the working point.

Get the angular position of the sensor frame relativeto the working point. The value returned by this method depends on the current state of the mechanical system. If we are working in the time domain the current angular displacement around the y axis, relative to the working point position, is returned. This is a real numbers, so the case `sel=Imaginary` is not applicable. If we are working in the frequency domain the angular response around the y axis at the current frequency is returned, the real part if `sel=Real` and the imaginary part if `sel=Imaginary`.

Return Value: the current angular displacement around the y axis relative to the working point position
Parameters: `sel` select in the frequency domain if the real (`sel=Real`) or the imaginary (`sel=Imaginary`) part is returned.

2.10.8

double **get_az** (ComplexType sel=Real)

Get the angular position of the sensor frame relativeto the working point.

Get the angular position of the sensor frame relativeto the working point. The value returned by this method depends on the current state of the mechanical system. If we are working in the time domain the current angular displacement around the z axis, relative to the working point position, is returned. This is a real numbers, so the case `sel=Imaginary` is not applicable. If we are working in the frequency domain the angular response around the z axis at the current frequency is returned, the real part if `sel=Real` and the imaginary part if `sel=Imaginary`.

Return Value: the current angular displacement around the z axis relative to the working point position

Parameters:

`sel` select in the frequency domain if the real (`sel=Real`) or the imaginary (`sel=Imaginary`) part is returned.

2.10.9

virtual char* **name** ()

Identificative string.

Identificative string.

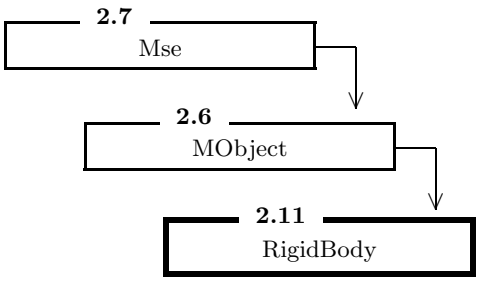
Return Value: the name of the class

2.11

class **RigidBody** : public MObject

A rigid body without internal structure.

Inheritance



Public Members

2.11.1 **RigidBody** () *Default constructor* 85

2.11.2 **RigidBody** (const mech_frame& pos0)
Constructor with initial position 86

2.11.3 **RigidBody** (double mass, double *Iij)
Constructor with parameters 86

2.11.4 **RigidBody** (double mass, double *Iij, const mech_frame& pos0)
Constructor with parameters and initial position
..... 86

2.11.5 virtual char* **name** () *Identificative string.* 86

Protected Members

virtual void **ApplyGravity** (double ax, double ay, double az)

virtual void **UpdateForces** (bool EvalJacobian=true)

2.11.6 virtual int **MassMatrix** (double *m=0)
This method evaluate the current mass matrix for the system. 87

2.11.7 virtual double **cm** (int i) *This method gives the position of the center of mass of the objectrelative to the frame 0.* 87

void **psdraw** (int i, int j, int lbl)

A rigid body without internal structure.

Parameters:

Parameter name	Parameter type	Description	Default value
`Name`	`TYPE\_STRING64`	The name of the instance of the class	""
`Mass`	`TYPE\_DOUBLE`	The total mass of the body	0.0
`Ixx`	`TYPE\_DOUBLE`	The I_{xx} component of the inertia tensor of the body	0.0
`Ixy`	`TYPE\_DOUBLE`	The I_{xy} component of the inertia tensor of the body	0.0
`Ixz`	`TYPE\_DOUBLE`	The I_{xz} component of the inertia tensor of the body	0.0
`Iyy`	`TYPE\_DOUBLE`	The I_{yy} component of the inertia tensor of the body	0.0
`Iyz`	`TYPE\_DOUBLE`	The I_{yz} component of the inertia tensor of the body	0.0
`Izz`	`TYPE\_DOUBLE`	The I_{zz} component of the inertia tensor of the body	0.0

2.11.1

RigidBody ()

Default constructor

Default constructor

2.11.2

RigidBody (const mech_frame& pos0)

Constructor with initial position

Constructor with initial position

2.11.3

RigidBody (double mass, double *Iij)

Constructor with parameters

Constructor with parameters

2.11.4

RigidBody (double mass, double *Iij, const mech_frame& pos0)

Constructor with parameters and initial position

Constructor with parameters and initial position

2.11.5

virtual char* **name** ()

Identificative string.

Identificative string.

Return Value: the name of the class

2.11.6

virtual int **MassMatrix** (double *m=0)

This method evaluate the current mass matrix for the system.

This method evaluate the current mass matrix for the system. The space for the data must be allocated and the used space is returned. The array has 6 rows and 6 columns. The mass matrix is of the following form:

$$M = \begin{pmatrix} m & 0 & 0 & 0 & -mz_{cm} & my_{cm} \\ 0 & m & 0 & mz_{cm} & 0 & -mx_{cm} \\ 0 & 0 & m & -my_{cm} & mx_{cm} & 0 \\ 0 & mz_{cm} & -my_{cm} & I_{xx} & I_{xy} & I_{xz} \\ -mz_{cm} & 0 & mx_{cm} & I_{xy} & I_{zz} & I_{yz} \\ my_{cm} & -mx_{cm} & 0 & I_{xz} & I_{yz} & I_{zz} \end{pmatrix} \tag{53}$$

where m is the total mass, (x_{cm}, y_{cm}, z_{cm}) is the current position of the center of mass relative to the frame, and I_{ij} is the inertia tensor of the body (with the current orientation). Note that in our case the frame is coincident by definition with the center of mass of the object, so the mass matrix does not mix angular and linear variables.

Return Value: the total storage used in the workspace m
Parameters:
`m` a pointer to the workspace where the mass matrix must be copied. If m=0 (default) only the total storage is evaluated and returned

2.11.7

virtual double **cm** (int i)

This method gives the position of the center of mass of the objectrelative to the frame 0.

This method gives the position of the center of mass of the objectrelative to the frame 0.

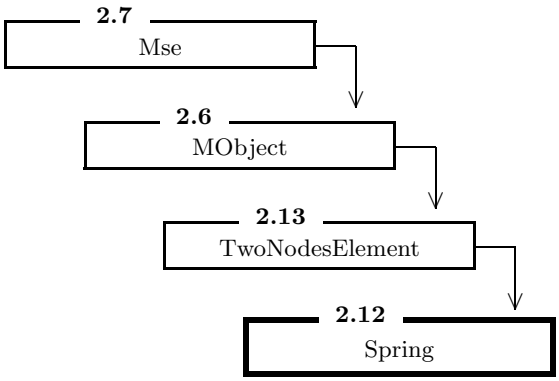
Return Value: the component of the displacement vector
Parameters:
`i` the index of the component of the displacement vector

2.12

class **Spring** : public TwoNodesElement

This is the model for a simple spring.

Inheritance



Public Members

2.12.1			**Spring** ()	*Default constructor.*	88
2.12.2			**Spring** (double length, double k)	*Constructor with parameters.*	89
2.12.3	virtual	char*	**name** ()	*Identificative string.*	89

Protected Members

	virtual	void	**UpdateForces** (bool EvalJacobian=true)		
	virtual	void	**psdraw** (int i, int j, int lbl)		
2.12.4	virtual	void	**FindWorkingPointInit** ()	*This method must be called before the simulation-phase.*	89
	virtual	int	**StiffMatrix** (double *m=0)		

This is the model for a simple spring.

Parameters:

Parameter name	Parameter type	Description	Default value
`Name`	`TYPE\_STRING64`	The name of the instance of the class	""
`Separation`	`TYPE\_DOUBLE`	Length of the spring at rest	1.0
`K`	`TYPE\_DOUBLE`	Stiffness of the spring	1.0

2.12.1

Spring ()

Default constructor.

Default constructor.

2.12.2

Spring (double length, double k)

Constructor with parameters.

Constructor with parameters.

2.12.3

virtual char* **name** ()

Identificative string.

Identificative string.

Return Value: the name of the class

2.12.4

virtual void **FindWorkingPointInit** ()

This method must be called before the simulationphase.

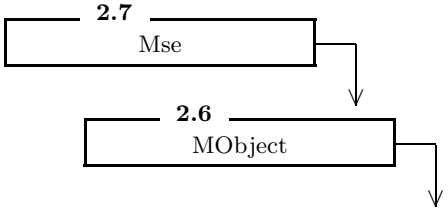
This method must be called before the simulationphase.

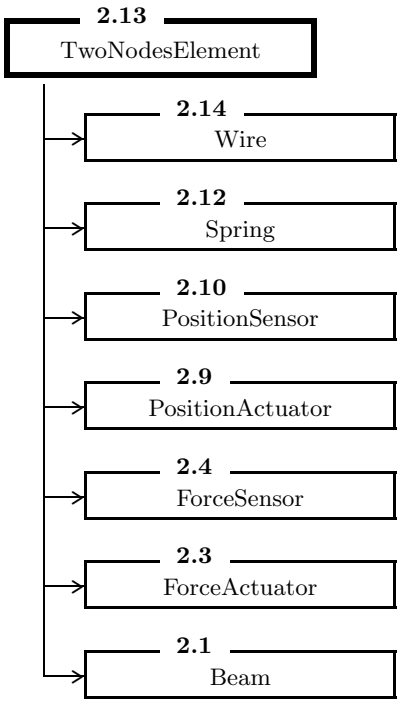
2.13

class **TwoNodesElement** : public MObject

Base class for all the mechanical object with twoframes.

Inheritance





Public Members

2.13.1 **TwoNodesElement** () *Default constructor.* 90
2.13.2 virtual char* **name** () *Identificative string.* 91

Base class for all the mechanical object with twoframes.

Parameters:

Parameter name	Parameter type	Description	Default value
`Name`	`TYPE\_STRING64`	The name of the instance of the class	""

2.13.1

TwoNodesElement ()

Default constructor.

Default constructor.

2.13.2

virtual char* **name** ()

Identificative string.

Identificative string.

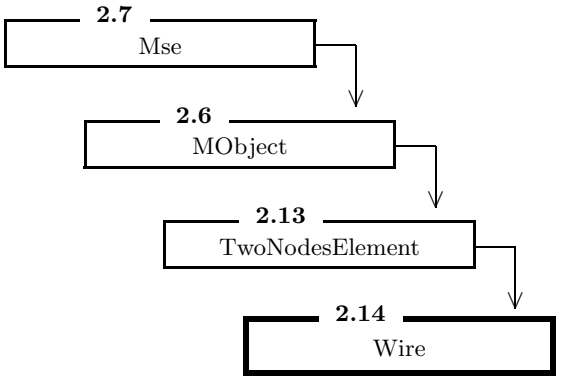
Return Value: the name of the class

2.14

class **Wire** : public TwoNodesElement

This is the model for a simple wire with torsional stiffness.

Inheritance



Public Members

2.14.1 **Wire** () *Default constructor.* 92

2.14.2 **Wire** (double length, double r, double youngm, double poissonr) *Constructor with parameters.* 92

2.14.3 virtual char* **name** () *Identificative string.* 92

Protected Members

virtual void **UpdateForces** (bool EvalJacobian=true)

virtual void **psdraw** (int i, int j, int lbl)

Private Members

2.14.4 virtual void **FindWorkingPointInit** () *This method must be called before the simulation-phase.* 93

This is the model for a simple wire with torsional stiffness.

The model is defined by a potential of the form

$$U = \frac{k_1}{2}\left(|o_1 - o_2| - l_0\right)^2 + \frac{k_2}{2}\left(\omega \cdot n\right)^2 \tag{54}$$

where ω is the rotation vector between the two frames of the wire.

Parameters:

Parameter name	Parameter type	Description	Default value
`Name`	`TYPE\_STRING64`	The name of the instance of the class	""
`Separation`	`TYPE\_DOUBLE`	Length of the wire at rest	1.0
`Young Modulus`	`TYPE\_DOUBLE`	Young's modulus of the material	1.0×10^8
`Poisson Ratio`	`TYPE\_DOUBLE`	Poisson's ratio of the material	0.5
`Radius`	`TYPE\_DOUBLE`	Radius of the wire circular section	1.0

2.14.1

Wire ()

Default constructor.

Default constructor.

2.14.2

Wire (double length, double r, double youngm, double poissonr)

Constructor with parameters.

Constructor with parameters.

2.14.3

virtual char* **name** ()

Identificative string.

Identificative string.

Return Value: the name of the class

2.14.4

virtual void **FindWorkingPointInit** ()

This method must be called before the simulationphase.

This method must be called before the simulationphase.

3

Low level library

The reference documentation for the utility library

Names

struct **_mech_dvec**

struct **_mech_vec**

struct **_mech_mat**

struct **_mech_frame**

struct **_mech_hess**

typedef struct double
l0

extern C void **mech_beam_force** (mech_dvec *f, mech_hess *h, mech_frame *r1, mech_frame *r2, beam_local_parameters *bp)

3.1 extern C void **mech_rotation_mat** (mech_vec *v, mech_mat *e)
Extraction of angular velocity. 96

extern C void **mech_mat_rotation_ordered** (mech_mat *e, double wx, double wy, double wz)

extern C void **mech_mat_rotation** (mech_mat *e, double wx, double wy, double wz)

extern C void **mech_dvec_random** (mech_dvec *dv, double eps)

extern C void **mech_dvec_randomize** (mech_dvec *dv, double eps)

extern C double **mech_dvec_square** (mech_dvec *dv)

extern C void **mech_integrate_velocity** (mech_frame *fm, mech_dvec *v, double feps, double teps)

extern C mech_frame
mech_mech_frame (double o1, double o2, double o3, double a1, double a2, double a3)

3.2 extern C void **mech_sum_force** (mech_frame *res, const mech_frame *op, const mech_frame *src)
This operation evaluate the total force and torque applied on a givenpoint 96

3.3 extern C void **mech_compose_hessian** (mech_hess *hres, mech_hess *h, mech_dvec *f, mech_frame *f1, mech_frame *f1_0, mech_frame *f2, mech_frame *f2_0)
Composition of frame hessians into dof hessians. 97

3.4 extern C void **mech_add_element_block** (double *dst, int dst_base, int dst_ld, double *src, int src_base, int src_ld, mech_frame *f1, mech_frame *f1_master, mech_frame *f2, mech_frame *f2_master, mech_dvec *v)

Add an element to the linear representation. .. 97

extern C void **mech_right_transform** (double *base, int rows, int ld, mech_frame *slave, mech_frame *master)

extern C void **mech_left_transform** (double *base, int cols, int ld, mech_frame *slave, mech_frame *master)

3.5 extern C void **mech_apply** (mech_frame *res, const mech_frame *op, const mech_frame *src)
This operation apply a transformation to a reference frame. 98

3.6 extern C void **mech_connection12** (mech_frame *f2_1, const mech_frame *f1, const mech_frame *f2)
This operation evaluate the frame f2 in the reference frame of f1. 98

3.7 extern C void **mech_compose_connection** (mech_frame *f3_1, const mech_frame *f3_2, const mech_frame *f2_1)
This operation find the frame f3 in the reference frame of f1, given theframe f3 in the reference frame f2 and the frame f2 in the reference frame f1. 99

3.8 extern C void **mech_invert_connection** (mech_frame *f1_2, const mech_frame *f2_1)
This operation evaluate the frame 1 in the reference frame 2, given the frame 2 in the reference frame 1. 99

3.9 extern C void **mech_identity_array** (mech_mat *r)
This operation build a 3x3 identity matrix. ... 100

3.10 extern C void **mech_mat_euler** (mech_mat *r, double a1, double a2, double a3)
This operation build a rotation array parametrized by three Euler'sangles. 100

3.11 extern C void **mech_copy_array** (mech_mat *dest_array, const mech_mat *src_array)
This operation copy a source 3x3 array in a destination 3x3 array. 100

extern C void **mech_mul_mat_mat** (mech_mat *array, const mech_mat *m1, const mech_mat *m2)

extern C void **mech_mul_mat_matT** (mech_mat *array, const mech_mat *m1, const mech_mat *m2)

extern C void **mech_mul_matT_mat** (mech_mat *array, const mech_mat *m1, const mech_mat *m2)

extern C void **mech_mul_matT_matT** (mech_mat *array, const mech_mat *m1, const mech_mat *m2)

extern C void **mech_mul_mat_vec** (mech_vec *vec, const mech_mat *a, const mech_vec *v)

extern C void **mech_mul_matT_vec** (mech_vec *vec, const mech_mat *a, const mech_vec *v)

extern C void **mech_copy_vec** (mech_vec *dst, const mech_vec *src)

extern C void **mech_transpose** (mech_mat *dst, const mech_mat *src)

extern C void **mech_array_copy** (int m, int n, double *a, int lda, double *b, int ldb)

3.12 extern C void **mech_array_copy_lc** (int m, int n, double *a, int lda, double *b, int ldb, double alpha, double beta)
Linear combination. 101

extern C void **mech_array_print** (int m, int n, double *a, int tda)

3.1

extern C void **mech_rotation_mat** (mech_vec *v, mech_mat *e)

Extraction of angular velocity.

Extraction of angular velocity. A rotation array can be represented as a rotation of a finite angle around a given axis. This routine, given a rotation array calculate the fixed axis and the rotation angle around it.

Parameters:

`v` a pointer to the results, which is a vector directed along the rotation axis with length equal to the rotation angle.

`e` a pointer to the rotation array.

3.2

extern C void **mech_sum_force** (mech_frame *res, const mech_frame *op, const mech_frame *src)

This operation evaluate the total force and torque applied on a givenpoint

This operation evaluate the total force and torque applied on a givenpoint

3.3

extern C void **mech_compose_hessian** (mech_hess *hres, mech_hess *h, mech_dvec *f, mech_frame *f1, mech_frame *f1_0, mech_frame *f2, mech_frame *f2_0)

Composition of frame hessians into dof hessians.

Composition of frame hessians into dof hessians. In order to construct the Hessian array of the constrained system, with dof entries, starting from the Hessian of the unconstrained system, with frame entries, we use

$$H^{(D1,D2)} = \sum_{F_1} \sum_{F_2} R^{(F1)} H^{(F1,F2)} T^{(F2)} \tag{55}$$

with $F_1 \in D_1$ and $F_2 \in D_2$. Here D_1 and D_2 are dofs, and F_1,F_2 are frames. The array $T^{(F)}$ is given by

$$T^{(F)} = \begin{pmatrix} I & -\vec{o}\wedge \\ 0 & I \end{pmatrix}$$

and the array $R^{(D)}$ is

$$R^{(F)} = \begin{pmatrix} I & 0 \\ \vec{o}\wedge & I \end{pmatrix} \tag{56}$$

where $\vec{o}$ is the displacement between the frame F and the master frame. This routine evaluate one term of the sum.

Parameters:

`hres`	a pointer to the result
`h`	a pointer to the unconstrined hessian between the frames f1 and f2
`f1`	a pointer to the frame f1
`f1_0`	a pointer to the master frame of frame f1
`f2`	a pointer to the frame f2
`f2_0`	a pointer to the master frame of frame f2

3.4

extern C void **mech_add_element_block** (double *dst, int dst_base, int dst_ld, double *src, int src_base, int src_ld, mech_frame *f1, mech_frame *f1_master, mech_frame *f2, mech_frame *f2_master, mech_dvec *v)

Add an element to the linear representation.

Add an element to the linear representation.

Parameters:

`dst`	is a pointer to the base of the linear array of the system
`dst_base`	is an offset to the position of the block in the linear array of the system
`dst_ld`	is the leading dimension of the linear array of the system
`src`	is a pointer to the array which describe the element
`src_base`	is an offset to the base of the array which describe the element
`src_ld`	is the leading dimension of the array which describe the element
`f1`	the first frame
`f1_master`	the master frame of the degree of freedom the f1 frame belongs to
`f2`	the second frame
`f2_master`	the master frame of the degree of freedom the f2 frame belongs to
`v`	a pointer to first order variation

3.5

extern C void **mech_apply** (mech_frame *res, const mech_frame *op, const mech_frame *src)

This operation apply a transformation to a reference frame.

This operation apply a transformation to a reference frame. If $F_1 = (o_1, R_1)$ is the reference frame and $O = (o, R)$ the operation the transformed reference frame is defined by

$$F_1^\star = (o_1 + o, RR_1) \tag{57}$$

Parameters:

`res`	a pointer to the result (transformed frame)
`op`	a pointer to the transformation
`src`	a pointer to the frame that must be transformed

3.6

extern C void **mech_connection12** (mech_frame *f2_1, const mech_frame *f1, const mech_frame *f2)

This operation evaluate the frame f2 in the reference frame of f1.

This operation evaluate the frame f2 in the reference frame of f1. If $F_1 = (o_1, R_1)$ and $F_2 = (o_2, R_2)$ we have

$$F_{2,1} = (R_1^T(o_2 - o_1), R_1^T R_2) \tag{58}$$

Parameters:

`f2_1`	a pointer to the result (the reference 2 from the reference 1)
`f1`	a pointer to the reference frame f1
`f2`	a pointer to the reference frame f2

3.7

extern C void **mech_compose_connection** (mech_frame *f3_1, const mech_frame *f3_2, const mech_frame *f2_1)

This operation find the frame f3 in the reference frame of f1, given theframe f3 in the reference frame f2 and the frame f2 in the reference frame f1.

This operation find the frame f3 in the reference frame of f1, given theframe f3 in the reference frame f2 and the frame f2 in the reference frame f1. If $F_{2,1} = (b_{2,1}, R_{2,1})$ and $F_{3,2} = (b_{3,2}, R_{3,2})$ we have

$$F_{3,1} = (b_{2,1} + R_{2,1}b_{3,2}, R_{2,1}R_{3,2}) \tag{59}$$

Parameters:

`f3_1`	a pointer to the result (the reference 3 from the reference 1)
`f3_2`	the reference 3 from the reference 2
`f2_1`	the reference 2 from the reference 1

3.8

extern C void **mech_invert_connection** (mech_frame *f1_2, const mech_frame *f2_1)

This operation evaluate the frame 1 in the reference frame 2, given the frame 2 in the reference frame 1.

This operation evaluate the frame 1 in the reference frame 2, given the frame 2 in the reference frame 1. If $F_{2,1} = (b_{2,1}, R_{2,1})$ whe have

$$F_{1,2} = (-R_{2,1}^T b_{2,1}, R_{2,1}^T) \tag{60}$$

Parameters:

`f1_2`	a pointer to the result (the reference 1 from the reference 2)
`f2_1`	a pointer to the reference 2 from the reference 1

3.9

extern C void **mech_identity_array** (mech_mat *r)

This operation build a 3x3 identity matrix.

This operation build a 3x3 identity matrix. This correspond to no rotations.

Parameters:

`r`	a pointer to the result

3.10

extern C void **mech_mat_euler** (mech_mat *r, double a1, double a2, double a3)

This operation build a rotation array parametrized by three Euler'sangles.

This operation build a rotation array parametrized by three Euler'sangles. The array can be written as

$$\begin{pmatrix} \cos\alpha_1\cos\alpha_3 - \cos\alpha_2\sin\alpha_1\sin\alpha_3 & \cos\alpha_3\sin\alpha_1 + \cos\alpha_1\cos\alpha_2\sin\alpha_3 & \sin\alpha_2\sin\alpha_3 \\ -\cos\alpha_2\cos\alpha_3\sin\alpha_1 - \cos\alpha_1\sin\alpha_3 & \cos\alpha_1\cos\alpha_2\cos\alpha_3 - \sin\alpha_1\sin\alpha_3 & \cos\alpha_3\sin\alpha_2 \\ \sin\alpha_1\sin\alpha_2 & -\cos\alpha_1\sin\alpha_2 & \cos\alpha_2 \end{pmatrix} \tag{61}$$

where α_1,α_2 and α_3 are the three Euler's angles.

Parameters:

`r`	a pointer to the result
`a1`	the first Euler's angle
`a2`	the second Euler's angle
`a3`	the third Euler's angle

3.11

extern C void **mech_copy_array** (mech_mat *dest_array, const mech_mat *src_array)

This operation copy a source 3x3 array in a destination 3x3 array.

This operation copy a source 3x3 array in a destination 3x3 array. The memory space for the result must be allocated yet.

Parameters:

`src_array`	a pointer to the array that must be copied
`dest_array`	a pointer to where the array must be copied into

3.12

extern C void **mech_array_copy_lc** (int m, int n, double *a, int lda, double *b, int ldb, double alpha, double beta)

Linear combination.

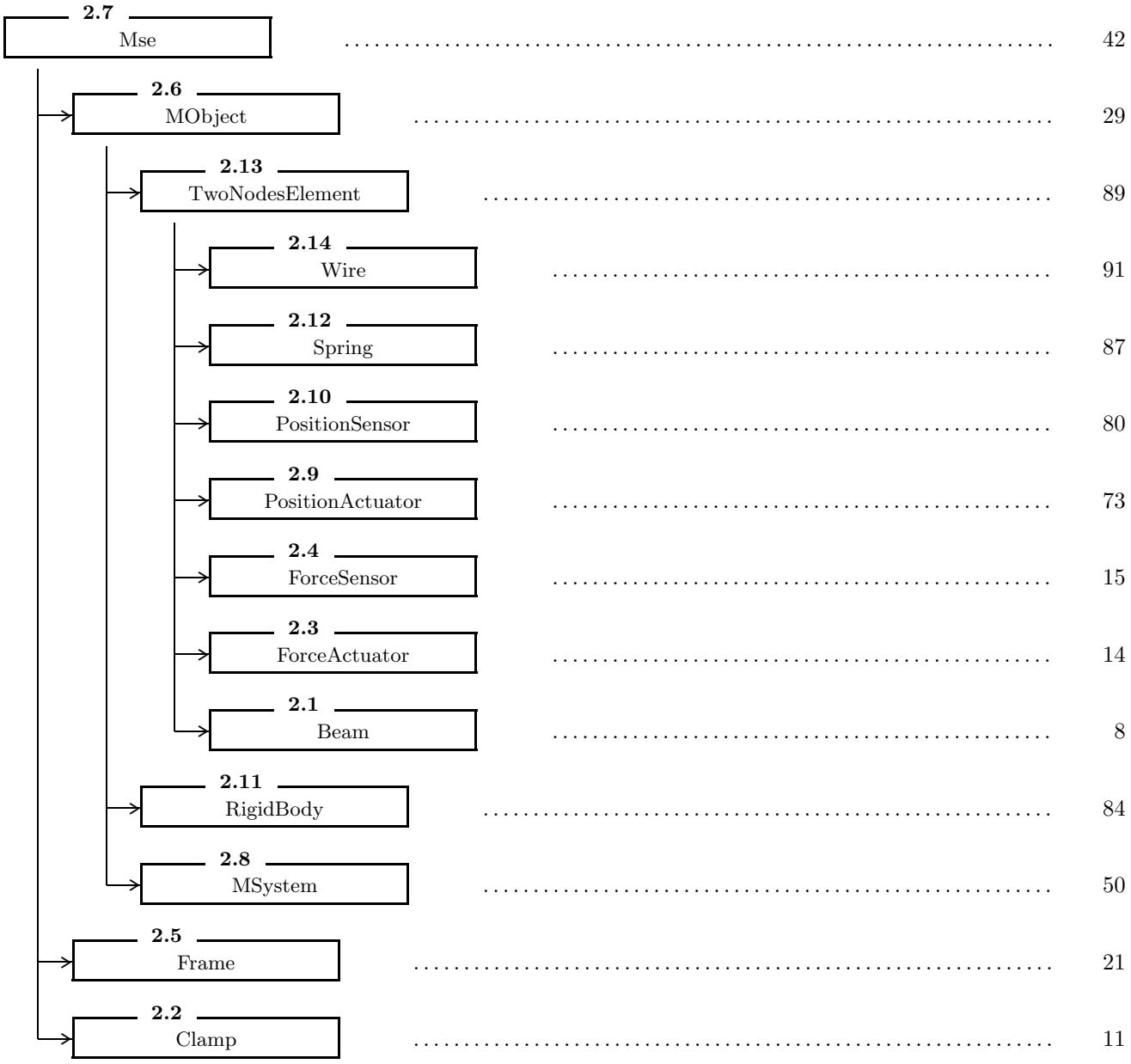
Linear combination. The following operation is executed:

$$A \rightarrow \alpha A + \beta B \tag{62}$$

Parameters:

`m`	is the number of rows of A and B
`n`	is the number of columns of A and B
`a`	the matrix A
`lda`	the leading dimension of A
`b`	the matrix B
`ldb`	the leading dimension of B
`alpha`	the value of α
`beta`	the value of β

Class Graph


2.7
Mse
42
2.6
MObject
29
2.13
TwoNodesElement
89
2.14
Wire
91
2.12
Spring
87
2.10
PositionSensor
80
2.9
PositionActuator
73
2.4
ForceSensor
15
2.3
ForceActuator
14
2.1
Beam
8
2.11
RigidBody
84
2.8
MSystem
50
2.5
Frame
21
2.2
Clamp
11