

Chrono::R3D

Scripting manual



©2007, **Delta Knowledge** . This document cannot be distributed or published without permission of the editor. This document should come together with the software Chrono::R3D, either in printed or as a file.

Dear Reader,

This product, manual and software, are Copyright 2007 DeltaKnowledge Italy. All rights reserved.

Purchasers are entitled to make one copy of this software for archival purposes. All other forms of duplication, whether electronic or physical, are expressly forbidden and will be prosecuted by law.

All products mentioned in this manual and the help files on the disks are trademarks of their respective owners.

DeltaKnowledge Italy assumes no responsibility as to the fitness or suitability of this product for any application. Also, no liability is assumed for the loss or destruction of data and programs resulting from the use of this product, neither for the consequences of the use of Chrono::R3D for the design or analysis of whatever mechanism or structure.

Contents

1	How to use the scripting manual	4
2	Introduction to Javascript	5
3	Chrono::R3D global functions	7
3.1	Introduction	7
3.2	Generic functions	7
4	Mathematics	10
4.1	Vector object	10
4.2	Quaternion object	11
4.3	Coordsys object	13
4.4	Matrix object	13
5	Objects: ChFunctions	20
5.1	ChFunction object	20
5.2	ChFunctionRamp object	21
5.3	ChFunctionSine object	22
5.4	ChFunctionSigma object	22
5.5	ChFunctionPoly object	23
5.6	ChFunctionConstAcc object	23
5.7	ChFunctionPoly345 object	24
5.8	ChFunctionFillet3 object	24
5.9	ChFunctionOperation object	25
5.10	ChFunctionMirror object	25

5.11	ChFunctionRepeat object	26
5.12	ChFunctionDerive object	26
5.13	ChFunctionIntegrate object	27
5.14	ChFunctionRecorder object	27
5.15	ChFunctionSequence object	28
5.16	ChFunctionJscript object	29
6	Objects: multibody items	30
6.1	System object	30
6.2	Body object	33
6.3	Marker object	35
6.4	Force object	37
6.5	Link object	38
6.6	LinkSpring object	40
6.7	LinkLock object	41
6.8	LinkScrew object	42
6.9	LinkGear object	42
6.10	LinkLinActuator object	43
6.11	LinkEngine object	44
6.12	LinkBrake object	45
6.13	LinkPneumaticActuator object	46
6.14	LinkWheel object	48
7	Other objects	49
7.1	Geometry object	49
7.2	Placer object	49
7.3	PID object	50
8	Objects: optimizers	52
8.1	ChOptimizer object	52
8.2	ChGeneticOptimizer object	53
8.3	ChLocalOptimizer object	54
8.4	ChHybridOptimizer object	55
9	Objects: GUI controls	57
9.1	GuiControl object	57
10	Utility functions	59
10.1	Animation control	59
10.2	Other	60
10.3	Marker placement utilities	60
10.4	Jacobian computing	61
10.5	Plotting utilities	61

1 How to use the scripting manual

Many properties and functions of Chrono::R3D can be accessed via scripting language. This scripting language is based on Javascript syntax (ECMA scripting standard), which may be familiar to many web developers and designers. Also, the Javascript syntax is very similar to C and C++ syntax, so it is easy to learn.

Some readers may be already aware of the fact that most common web browsers expand the core Javascript functions with own features, for example utilities for image or html manipulations. Similarly, the Javascript language for Chrono has been expanded with custom Chrono functions, which are described in this chapter.

By using Chrono::R3D scripting, you will be able to change properties of the multi-body system by typing formulas in **Realsoft3D** scripting window or -most important- you can write .JS programs which will be executed for each step of simulation (hence providing, for example, a powerful way to switch on/off forces or engines depending on physical events, such as in the simulation of robots with artificial intelligence, or devices with conditional behavior, or feedback controllers).

The Chrono-Javascript language embeds a standard Javascript interpreter, so we wont describe in detail how to use all basic features of this language.

The following paragraphs will first explain some basic issues of Javascript, for people who hasnt any knowledge about core functionality of this language, then will jump into the deep details of custom Chrono::R3D functions.

2 Introduction to Javascript

Javascript is a scripting language based on ECMA specifications. Unlike compiled languages like C, C++, Java, etc., this language is interpreted on the fly by the host program (Chrono::R3D, in this case), without the need of compilation. Of course this may have a negative impact on execution speed, but the big advantage of all scripting languages is their user friendly attitude.

Among the highlights of the Javascript language: the user can create functions and test them on the fly, variables does not need declaration, object-oriented programming is easy to handle, there are no pointers, syntax is very similar to C++, errors hardly cause program crashes.

In this paragraph you can read just a quick introduction to the core features of Javascript, but interested readers are invited to look at the full documentation, for example on the web site: http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Reference

First of all, to experiment with Chrono-Javascript you should open the interpreter shell, by using the Realsoft3D menu Windows/Scripting window. Then, remember to switch the language to Javascript, since other languages (like RPL, TCL) may be installed in Realsoft3D. This can be done by using the selector on the top of the scripting window.

Then, you can type Javascript commands in the shell, and they will be executed as soon as you press ENTER (carriage return). For example, type:

```
a=2; b=3; print(a+b);
```

and you will get the return value 5 when you press ENTER. Note that you can enter also loop values, which are executed on the fly:

```
for(i=0; i<10; i++) { print("loop index:", i ); }
```

You can define functions in this way:

```
function myfx (ma, mb) { return (ma+mb); }
```

and call them when you need :

```
myresult = myfx (12.4 , 23.4);
```

Note that, in case you need complex functions, you may need to define them in files which can be loaded when needed. This can be done as follows. Prepare an ASCII file with the function -you can use whatever ASCII editor, for example Notepad, VI, Emacs, etc.- and save it with whatever name (we suggest you to use always the suffix .js to remember that it is a Javascript program). For example you can type:

```
// Comment: this file defines a function, and calls
// a print() function at the end.

function myexample (ma, mb, mc)
{
    mtemp = mb*mc;
    return (ma+mtemp);
}

print ("Ok, function myexample() defined!");
```

Then, save this file with the name my-test.js in the directory chrono/javascript/. Well, you can load and execute it by typing in scripting window shell the following command:

```
load("my-test.js");
```

Note that the default path is the directory `chrono/javascript/`, but you can load files elsewhere, for example by typing

```
load("C:/tmp/my-test.js").
```

Some variables in Javascript are objects, and can have sub-properties and sub-functions (these are like member data and member functions in C++ programs), and the “.” operator is used to access them. See the following example:

```
foo = new Object;    // default base class
foo.weight = 23;     // property is automatically created!
foo.height = 123;    // another property...
foo.name = "John";   // another property...

print (foo.name, " is ", foo.height, " tall ");
```

Note that you use `new` to create objects from classes (the Chrono::R3D implementation of JS embeds lot of custom classes, such as `Vector`, and will be explained later).

An interesting tip: the `attrs()` function is very useful if you want to see the properties of an unknown object:

```
attrs(foo); // try this: it will print the list of
            // object properties if you dont remember!
```

Remember that Javascript is case sensitive: for example `Attrs()` is not `attrs()`.

3 Chrono::R3D global functions

3.1 Introduction

The Chrono::R3D implementation of Javascript embeds new functions in the global environment. One of the most important of these features is the `GetSys()` function, which can be used to obtain the reference to the multibody system object -a starting point to get and set properties of all other Chrono::R3D objects as modelled in the 3D editor: forces, rigid bodies, markers, links-.



Some examples about documentation of syntax:

`my_fx (double, Vector)` means that the function `my_fx` needs two arguments, a floating-point number (double datatype) and a Vector object (Vector datatype), and returns nothing.

`double my_fx(int)` means that the function `my_fx` needs an argument (an integer) and returns a number (a double datatype).

`my_fx (double A, [double B])` means that the function needs at least parameter A, while the second (parameter B) is optional.



*In this 'scripting' manual, if not otherwise specified, with the term **object** or **Javascript object** we mean an object of the Javascript language. Otherwise, with the term **3D object** we exclusively mean an object which has been created in 3D editor by **Realsoft3D** or Chrono::R3D tools.*

3.2 Generic functions

print (...)

Prints data in the shell of the scripting window (if scripting window is opened, otherwise nothing is shown).

Parameters can be multiple, if separated by commas, and numbers (double, int) are automatically converted to strings, as in: `print("test", (12+3))`

load (string)

Loads a Javascript file, and executes it. This is very useful because you can write complex functions in ASCII files, then load them by typing this `load()` command in the scripting window. The functions defined in the .js file are also compiled, so their execution is faster, when invoked.

Note that you can use the `load()` function also in .js files, like in C++ you use the `#include<>` statement, but pay attention that you should not create endless loops where, for example, a file A loads B, B loads C, and C loads A..... This causes a deadlock, unless you use global flags to prevent this.

attrs (Object)

Prints the list of sub-properties of the parameter object (all objects in Javascript are inherited from Object class, except strings and numbers). This is useful when doing experiments with objects whose inner structure you don't know well.

objtree (Object)

Similar to `attrs()`, but also prints the sub properties of the sub objects, if any, showing a property tree in form of indented text.

System **GetSys** ()

Returns the reference to the Chrono::R3D multibody system, that is a System object which is documented in a following paragraph. If no system is available, returns false. When getting the System, you can access many properties, for example the numerical tolerance for kinematics, with `GetSys().tol=0.001`; or you can fetch other Chrono::R3D objects with the `Obj()` member function, like in `GetSys().Obj("link_name").react_force.x`

See the documentation of the [System object](#) for more information.

Geometry **Geo** (string)

Fetches a Geometry object whose name matches the parameter string. The Geometry object is a reference to whatever surface, spline, primitive which exists in 3D editor, does not matter if Chrono::R3D objects exist in the scene.

Pay attention that if you have two objects in your 3D project with the same name (ex: "mesh1", "mesh1"), you have no warranty that you get the one you want, so rename the one you want with unique name. Geometry objects are documented later, and can be used to fetch points, vectors, normals, alignments in space.

Object **globals** ()

Returns the current "global" object (the global environment), so if you type for example: `attrs(context())` you get the list of all the variables you have declared.

Object **context** ()

Returns the current "global" object.

By default, this returns the lowest global environment, so if you type `attrs(context())` you get the list of all the variables you have declared.

NB: in some circumstances, this function returns other objects, for example if you create a [probe](#) 3D object into, say, a [spherical link](#) 3D object, when the `context()` function is executed from the "Probe" scripting it returns the reference to that spherical link, i.e. a Link Javascript object.

bool **is_jsowned** (Object)

If the parameter object has been created in 3D scene (for example with mouse and Realsoft or Chrono::R3D tools) this function returns false.

Otherwise the object was created by a new ... function in Javascript or has been created automatically as a return value of some JS function, so it returns true (this means that the object will be automatically deleted by Javascript garbage collection).

set_jsowned (Object, bool)

Forces a change in the state of the 'owned' flag of the Object. [should be used with caution].

4 Mathematics

4.1 Vector object

The Vector object is a custom Chrono::R3D Javascript object which can be used to do mathematical and geometrical operations with three-dimensional vectors of the x,y,z type.

This object supports user construction via new operator, for example:

```
mv = new Vector
```

Inheritance

Object - Vector

Data

x	double
y	double
z	double

Coordinates x,y,z of the three dimensional vector.

Functions

Vset (double, double, double)

Sets all the components x,y,z at once.

double **Vlength** ()

Returns the length of the vector (euclidean norm).

Vnorm (Vector A)

Vector object becomes the vector A normalized (thus it will be forced to unit length, but with same direction of A)

Vmul (double b)

Performs the product by a scalar, the result goes in the object vector, so `mvect.Vmul(b)` means `mvect*=b`

Vcross (Vector A, Vector B)

Performs the cross product between a vector and a vector, the result goes in the object vector, so `mvect.Vcross(A,B)` means `mvect = AxB`

double **Vdot** (Vector A, Vector B)

Returns the dot (scalar) product between vector A and vector B

Vadd

(Vector A, Vector B)

Performs the sum product between vector A and vector B, the result goes in the object vector, so `mvect.Vadd(A,B)` means `mvect = A+B`

Vsub

(Vector A, Vector B)

Performs the difference product between vector A and vector B, the result goes in the object vector, so `mvect.Vsub(A,B)` means `mvect = A-B`

Examples

```
va = new Vector;
va.Vset(1,2,3);
va.x=0; print(" va is:", va);

vb = new Vector;
vb.x= 1;
vb.y=4;
print(" length of vb is: ",vb.Vlength() );

vc = new Vector;
vc.Vadd(va,vb);
print(" sum vc=va+vb is: ", vc );
```

4.2 Quaternion object

The quaternion object is a custom Chrono::R3D Javascript object which can be used to do mathematical and geometrical operations with four-dimensional hypercomplex numbers of the $\bar{q} = e_0, i \cdot e_1, j \cdot e_2, k \cdot e_3$ type, with $\bar{q} \in \{\mathbb{R}^1, \mathbb{S}^3\}$

This object supports user construction via new operator, for example:

```
mv = new Quaternion
```

Inheritance

Object - Quaternion

Data

e0	double
e1	double
e2	double
e3	double

Components of the quaternion (e1 is the real part, other are imaginary).

Functions**Qset**

(double,double, double, double))

Set the four components at once.

double Qlenght ()
 Computes the euclidean norm of the quaternion.

Qnorm (Quaternion A)
 Object quaternion becomes the normalized quaternion A.

Qscale (double A)
 Object quaternion is multiplied by factor A.

Qcross (Quaternion A, Quaternion B)
 Object quaternion becomes the quaternion product of A and B.

Qadd (Quaternion A, Quaternion B)
 Object quaternion becomes the sum of A and B.

Qsub (Quaternion A, Quaternion B)
 Object quaternion becomes the difference of A and B.

Vector QtoEulero ()
 Object quaternion is transformed in a vector of three angles of Eulero which represent the same rotation (this is senseless if quaternion is not normalized).

Vector QtoCardano ()
 Object quaternion is transformed in a vector of three Cardano angles which represent the same rotation (this is senseless if quaternion is not normalized).

Vector QtoRodriguez ()
 Object quaternion is transformed in a vector of three Rodriguez parameters which represent the same rotation (this is senseless if quaternion is not normalized).

QfromAngAxis (double A, Vector B)
 Sets the quaternion so that represents the rotation about an axis B (must be a normalized vector!) with angle of rotation A.

Examples

```
// Type these commands in the R3D script window,
// to view results after each command..

qa = new Quaternion;
qa.el = 2
qa.Qlenght()
```

```

qa
qa.Qnorm(qa)
qa
qa.Qlenght()
qa.QtoCardano()

// From angle-axis of rotation to quaternion, then
// from quaternion to Cardano angles..

v = new Vector;
v.y = 1 ;
v.Vnorm(v);
qa.QfromAngAxis(Math.PI,v);
print( qa );
print( qa.QtoCardano() );

```

4.3 Coordsys object

The Coordsys object is a custom Chrono::R3D Javascript object which can be used to do mathematical and geometrical operations: it represents the coordinate system of a three-dimensional rigid system moving in space (hence defines position and rotation, where rotation is expressed with a quaternion).

This object supports user construction via new operator, for example:

```
mv = new Coordsys
```

Inheritance

Object Coordsys

Data

pos	Vector
position of coordinate system origin	
rot	Quaternion
rotation of coordinate system	

4.4 Matrix object

The matrix object is a custom Chrono::R3D Javascript object which can be used to do linear algebra, mathematical and geometrical operations with NxM matrices of double-precision numbers.

To access the elements, use GetEl() and SetEl() functions.

This object supports user construction via new operator, for example:

```

my_matr = new Matrix % builds a default 3x3 matrix
my_matr = new Matrix(6) % builds a 6x1 column matrix
my_matr = new Matrix(5,7) % builds a 5x7 matrix

```

Inheritance

Object Matrix

Data

```

rows                                int
columns                             int

```

Rows and columns of the matrix.

Functions

```

SetEl                                (int A, int B, double E)

```

Sets the element E into row A and column B (remember that first column and first row have index 0! A must range in [0..row-1] and B in [0..columns-1])

```

double    GetEl                       (int A, int B)

```

Gets the element at row A and column B (remember that first column and first row have index 0! A must range in [0..row-1] and B in [0..columns-1])

```

SafeSetEl                             (int A, int B, double E)

```

Sets the element E into row A and column B (if row A and/or column B are out of ranges [0..row-1] [0..columns-1], no error happens, because the matrix is simply enlarged with padding zeroes, so that the new element can be inserted, like in Matlab(tm).)

```

Copy                                  (Matrix A)

```

Copy from a matrix A (if needed, object matrix also changes number of rows and columns to match A).

Note: this is a complete copy, unlike the Javascript '=' operator, so `matrA.Copy(matrB)` is different from a simple `matrA=matrB` (in the second case, both `matrA` and `matrB` will reference the same matrix)

```

CopyT                                 (Matrix A)

```

Same as `Copy()`, but copies from A transposed.

```

Reset                                (int R, int C)

```

Fills the matrix with zero values, and (if needed) changes it size in order to have R rows and C columns).

```

FillElem                             (double E)

```

Fills the matrix with elements of value E.

```

FillDiag                             (double E)

```

Sets the diagonal elements of a matrix with elements E. Object matrix should be square.

```

FillRandom                           (double A, double B)

```

Fill the matrix with random elements, whose values range between min and max values A and B.

SetIdentity ()

Fills the matrix with 1 (ones) on the diagonal, and 0 otherwise.

Multiply (Matrix A, Matrix B)

Object matrix becomes product between A and B. That is, `mR.Multiply(mA,mB)` means $mR = mA * mB$.

MultiplyT (Matrix A, Matrix B)

Object matrix becomes product between A and B transposed. That is, `mR.MultiplyT(mA,mB)` means $mR = mA^T * mB$.

TMultiply (Matrix A, Matrix B)

Object matrix becomes product between A transposed and B. That is, `mR.TMultiply(mA,mB)` means $mR = mA^T * mB$.

Add (Matrix A, Matrix B)

Object matrix becomes the sum of A and B. That is, `mR.Add(mA,mB)` means $mR = mA + mB$.

Sub (Matrix A, Matrix B)

Object matrix becomes the difference between A and B. That is, `mR.Sub(mA,mB)` means $mR = mA - mB$.

Inc (Matrix A)

Object matrix is incremented by matrix A. That is, `mR.Inc(mA)` means $mR += mA$.

Scale (Matrix A, double B)

Object matrix becomes product between A and scalar B.

Transpose ()

Transpose the matrix (exchange rows with columns).

AtoQ ()

Quaternion

Returns the quaternion which represent the same rotation of the object matrix (note that the matrix must be a 3x3 rotation matrix, orthogonal, Lie-SO).

QtoA (Quaternion A)

Given quaternion A, sets the matrix so that it represents the same rotation. (note that quaternion must be normalized, and the matrix must be 3x3).

QtoG1

(Quaternion A)

Given quaternion A, sets the matrix so that it becomes the G_l matrix for the equation $W_l = [G_l]dq/dt$, where W_l is angular speed in local coordinates. (note that quaternion must be normalized, and the matrix must be 3x4).

QtoGw

(Quaternion A)

Given quaternion A, sets the matrix so that it becomes the G_w matrix for the equation $W_w = [G_w]dq/dt$, where W_w is angular speed in world coordinates. (note that quaternion must be normalized, and the matrix must be 3x4).

VtoX

(Vector A)

Given vector A, sets the matrix so that it becomes the emi-symmetric X_a matrix for the equation $C = [X_a]B$, which is an alternate form for vector cross product $C = A \wedge B$ (note that the matrix must be 3x3).

Vector

A.x.V

(Vector P)

Given vector P, performs matrix-vector product $[A]P$ and returns the transformed vector. The original vector P is not changed. (note that the matrix must be 3x3).

Vector

At.x.V

(Vector P)

Given vector P, performs matrix-vector product $[A]^T P$ by using this matrix, but transposed. Returns the transformed vector. The original vector P is not changed. (note that the matrix must be 3x3).

Vector

G.x.Q

(Quaternion Q)

Given quaternion Q, performs matrix-quaternion product $[A]Q$ by using this matrix. Returns a vector. (Note: of course this function works only if the matrix is rectangular, size 3x4).

double

Det

()

Gets the determinant of the matrix. If null, matrix is singular. Matrix must be square. (Gets the determinant by internally performing a temporary LU decomposition).

double

Cond

()

Gets the condition number of the matrix. Ranges in $[1..\infty]$. If infinite or very high, matrix is singular.

Optimal conditioning is near 1. Tells how precise and well conditioned can be the solution of a linear system based on this matrix. (Internally performs a temporary SVD decomposition: it may take time for very large matrices!).

double

Rcond

()

Same as Cond, but returns the “reciprocal condition number”. Ranges in $[0..1]$, if 0 the matrix is singular, if near 1 the matrix is well conditioned.

SVD

(Matrix U, Matrix W, Matrix V)

Performs a full SVD decomposition of object matrix.

Will set the three parameter matrices U,W,V with the resulting U V W matrices of the SVD decomposition (all three matrices will be reset to correct sizes and will become: U a square or rectangular matrix, W the diagonal matrix of the singular values, V a square orthogonal matrix).

Object matrix can be either rectangular or square. Decomposition may be a cpu-intensive computation.

double **Invert** ()

Inverts the object matrix. Inversion happens in place (original matrix is overwritten).

double **SolveLinSys** (Matrix B, Matrix X, Matrix P)

If object matrix is A, solves the linear system $[A]X = B$, where B is the vector of known terms (a one-columns matrix) and X is the vector of unknowns (a one-column matrix), by calling `A.SolveLinSys(B,X,P)`.

While computing X, also computes P, a matrix of pivot indexes.

Note that A must be square NxN, and matrix B must have size Nx1 (while X and P are automatically resized).

Returns the determinant: if null, solution was not possible (singular matrix).

double **DecomposeLU** (Matrix P)

Performs the in-place $[A] = [L][U]$ decomposition of object matrix (triangular matrices L lower and U upper are both written over original matrix).

Note: object matrix must be square.

Note: a matrix P must be passed, it will be automatically resized and will get the vector of pivots: it is needed because you later may perform `SolveLU()` to solve linear systems.)

Returns the determinant: if null, solution was not possible (singular matrix).

SolveLU

(Matrix B, Matrix X, Matrix P)

Performs the solution of a linear system $[A]X = B$, where B is the vector of known terms (a one-columns matrix) and X is the vector of unknowns (a one-column matrix).

$[A]$ is the object matrix, which has been previously decomposed in place in LU form with `DecomposeLU()`.

double **DecomposeLDL** (Matrix P)

Performs the in-place $[A] = [L][D][L]$ decomposition of object matrix (lower triangular matrix $[L]$, diagonal $[D]$, upper triangular $[L]$, are written over original matrix).

Note: this decomposition needs that object matrix is square and symmetric! It is similar to Cholesky decomposition but does not require matrix to be positive-definite (may also be negative-definite).

Note: a matrix $[P]$ must be passed: it will be automatically resized and will get the vector of diagonal pivots: it is needed because you later may perform `SolveLDL()` to solve linear systems.)

Returns determinant: if null, solution was not possible (singular matrix).

SolveLDL

(Matrix B, Matrix X, Matrix P)

Performs the solution of a linear system $[A]X = B$, where B is the vector of known terms (a one-columns matrix) and X is the vector of unknowns (a one-column matrix). A is the object matrix, which has been previously decomposed in place in LDL form with `DecomposeLDL()`.

Print

()

Prints all elements of matrix, row by row (beware: large matrices will not be printed, to avoid filling the screen).

Examples

```
// ---- how to set and get elements ----

a = new Matrix; // create default 3x3 matrix
a.Reset(5,3);   // resize to 3 rows, 5 columns
a.SetEl(1,2, 12.3+21i);
a.GetEl(1,2);
a.Print();
a.GetEl(5,0);   // ops..out of boundary!

// ---- How to perform operations ----

ma = new Matrix
mb = new Matrix
mc = new Matrix

ma.FillElem(2)
ma.Print()

mb.FillDiag(5)
mb.Print()

mc.Multiply(ma,mb) // mc = ma*mb
mc.Print()

mc.Add(ma,mb)      // mc = ma+mb
mc.Print()

print("determinant is: ", mc.Det() );

mc.Invert();
print("determinant of inverse is:", mc.Det() );

// ---- How to solve linear systems ----

// create a 3x3 matrix, fill with random values
ma = new Matrix
ma.Reset(3,3)
ma.FillRandom(0,1)
ma.Print()

// create a 3x1 matrix, fill with random values
b= new Matrix()
b.Reset(3,1)
b.FillRandom(1,2)
b.Print()
```

```

// prepare a x matrix, the unknown vector
x = new Matrix()

// prepare a pivot vector, will contain pivots
pivots = new Matrix()

// solve for x in [ma]x=b with this instruction!
ma.SolveLinSys(b,x,pivots) x.Print()

// show that the result is correct...
b2 = new Matrix;
b2.Reset(3,1)
b2.Multiply(ma,x)
b2.Print()
b.Print()
print("See the difference? ", (b.GetEl(0,0)-b2.GetEl(0,0)) );

```

5 Objects: ChFunctions

The **ChFunctions** objects are building blocks of complex functions, often used in Chrono::R3D 3D objects to set motion laws, non linear properties, etc.

These **ChFunctions** can also be created and manipulated in the Javascript environment.

5.1 ChFunction object

The ChFunction object is a custom Chrono::R3D Javascript object which can be used as a base class for many functions of the type $y = f(x)$.

Of course you do not need to use ChFunction to make a function (you can just use the javascript syntax), but in some situations it is needed: for example many Chrono::R3D objects contain **ChFunctions** (so you may need to access and modify them).

Also, **ChFunctions** are coded in C++ and their evaluation may be faster than evaluating an equivalent function of type $y=f(x)$ written entirely in Javascript.

This ChFunction object is a base class, just returns a constant value:

$$y = C$$

but many other types of functions are inherited from this object (see later).

This object supports user construction via new operator, for example:

```
mfx = new ChFunction
```

Inheritance

Object - ChFunction

Data

C double
Constant value returned by function: $y=C$

Functions

double **y** (double x)
Returns $y=f(x)$.

double **y.dx** (double x)
Returns dy/dx , with $y=f(x)$

double **y.dxdx** (double x)
Returns $ddy/dxdx$, with $y=f(x)$

double **Min** (double xa, double xb, double dx)
Return the minimum of function in interval $(xa,...,xb)$, with sampling step dx (must be small for oddly and noisy functions).

```
double      Max                                (double xa, double xb, double dx)
    Return the maximum of function in interval (xa,...,xb), with sampling step dx (must be
    small for oddly and noisy functions).
```

```
double      Mean                                (double xa, double xb, double dx)
    Return the mean value of function in interval (xa,...,xb), with sampling step dx (must
    be small for oddly and noisy functions).
```

```
double      SqrMean                            (double xa, double xb, double dx)
    Return the square mean value of function in interval (xa,...,xb), with sampling step dx
    (must be small for oddly and noisy functions).
```

```
double      Int                                (double xa, double xb, double dx)
    Return the definite integral of function in interval (xa,...,xb), by running a simple trape-
    zoidal quadrature with sampling step dx .
```

```
          plot                                ( )
    Opens a window with the plotting of the function. The window can be srolled, zoomed,
    etc. with the mouse. Popup menus of the window can be used to save/load, change plot
    properties. etc.
```

Examples

```
mf = new ChFunction
    // set parameter
mf.C= 10;
    // gets the value at x=0
mf.y(0)
    // gets the derivative at x=2
mf.y_dx(2)
```

5.2 ChFunctionRamp object

Linear function of the type

$$y = C + A \cdot x$$

This object inherits all methods of base ChFunction, like `y()`, `y_dx()` etc.

This object supports user construction via new operator, for example:

```
mfx = new ChFunctionRamp
```

Inheritance

Object - ChFunction - ChFunctionRamp

Data

C	double
costant C	

ang	double
angular coefficient a	

Examples

```
mf = new ChFunctionRamp
mf.ang=2;
mf.C=1;
mf.y(1.5)
mf.y(2.3)
mf.y_dx(0.1)
```

5.3 ChFunctionSine object

Trigonometric function of the type

$$y = A * \sin(x \cdot \omega + \psi)$$

This object inherits all methods of base **ChFunction**, like `y()`, `y_dx()` etc.

This object supports user construction via new operator, for example:

```
mfx = new ChFunctionSine
```

Inheritance

Object - ChFunction - ChFunctionSine

Data

amp	double
amplitude of sine wave	
phase	double
phase	
freq	double
frequency (f=1/period)	
w	double
angular frequency w= freq*2PI	

5.4 ChFunctionSigma object

Sigma function with cubic ramp between two levels.

This object inherits all methods of base **ChFunction**, like `y()`, `y_dx()` etc. This object supports user construction via new operator, for example:

```
mfx = new ChFunctionSigma
```

Inheritance

Object - ChFunction - ChFunctionSigma

Data

start	double
value of x where ramp starts	
end	double
value of x where ramp ends	
amp	double
amplitude of ramp (can be also negative)	

5.5 ChFunctionPoly object

Polynomial function of generic order:

$$y = C_0 + C_1x + C_2x^2 + C_3x^3 + \dots + C_nx^n$$

This object inherits all methods of base **ChFunction**, like `y()`, `y.dx()` etc.

This object supports user construction via `new` operator, for example:

```
mfx = new ChFunctionPoly
```

Inheritance

Object - **ChFunction** - **ChFunctionPoly**

Data

order	int
order of polynomial	

Functions

set_coeff	(double A, int N)
Set the N-th coefficient of the function, with value A.	
double get_coeff	(int N)
Gets the N-th coefficient of the function.	

5.6 ChFunctionConstAcc object

Constant acceleration function, starting with zero speed (constant value) and ending with zero speed (constant value), using constant acceleration and constant deceleration, with desired amplitude. Often used in mechatronics to impose motion in automatic devices.

This object inherits all methods of base **ChFunction**, like `y()`, `y.dx()` etc.

This object supports user construction via `new` operator, for example:

```
mfx = new ChFunctionConstAcc
```

Inheritance

Object - **ChFunction** - **ChFunctionConstAcc**

Data

h	double
amplitude of ramp	
end	double
duration of ramp	
aw	double
duration of acceleration stage, normalized	
av	double
duration of deceleration stage, normalized	

5.7 ChFunctionPoly345 object

“Polynomial 3-4-5” function, a polynomial with 3,4,5 terms, starting with zero speed (constant value) and ending with zero speed (constant value), with desired amplitude. Often used in mechatronics to impose motion in automatic devices.

This object inherits all methods of base **ChFunction**, like `y()`, `y_dx()` etc. This object supports user construction via new operator, for example:
`mfx = new ChFunctionPoly345`

Inheritance

Object - **ChFunction** - **ChFunctionPoly345**

Data

h	double
amplitude of ramp	
end	double
duration of ramp	

5.8 ChFunctionFillet3 object

Polynomial function with imposed starting/ending values, and imposed starting/ending speeds. Mostly used as a segment of a sequence of functions, see **ChFunctionSequence**.

This object inherits all methods of base **ChFunction**, like `y()`, `y_dx()` etc. This object supports user construction via new operator, for example:
`mfx = new ChFunctionFillet3`

Inheritance

Object - **ChFunction** - **ChFunctionFillet3**

Data

y1	double
starting value	
y2	double
ending value	
dy1	double
starting speed	
dy2	double
ending speed	

5.9 ChFunctionOperation object

Operation between two other **ChFunction** objects, in general:

$$y(x) = y(f_a(x), f_b(x))$$

Can be sum of two functions, difference, product, function of function, etc.

This object inherits all methods of base **ChFunction**, like `y()`, `y_dx()` etc.

This object supports user construction via `new` operator, for example:

```
mfx = new ChFunctionOperation
```

Inheritance

Object - **ChFunction** - **ChFunctionOperation**

Data

fa	ChFunction
function A	
fb	ChFunction
function B	
op_type	int
type of operation $f(f_a(x), f_b(x))$, see graphical interface for IDs	

5.10 ChFunctionMirror object

Mirror a **ChFunction** object on a vertical axis, at a specified X value.

This object inherits all methods of base **ChFunction**, like `y()`, `y_dx()` etc.

This object supports user construction via `new` operator, for example:

```
mfx = new ChFunctionMirror
```

Inheritance

Object - **ChFunction** - **ChFunctionMirror**

Data

fa	ChFunction
function to be mirrored, after the value of mirror_axis	
mirror_axis	double
value of X, after which the function will be reversed for mirroring.	

5.11 ChFunctionRepeat object

Repeats a **ChFunction** object, to make periodic functions.

This object inherits all methods of base **ChFunction**, like `y()`, `y_dx()` etc.

This object supports user construction via `new` operator, for example:

```
mfx = new ChFunctionRepeat
```

Inheritance

Object - ChFunction - ChFunctionRepeat

Data

fa	ChFunction
function to be repeated periodically	
window_start	double
value of X for fa function, where the piece to be repeated starts.	
window_length	double
duration of the repeated period	

5.12 ChFunctionDerive object

Derive a **ChFunction** object, using analytic derivation if allowable, otherwise uses numerical differentiation.

This object inherits all methods of base **ChFunction**, like `y()`, `y_dx()` etc.

This object supports user construction via `new` operator, for example:

```
mfx = new ChFunctionDerive
```

Inheritance

Object - ChFunction - ChFunctionDerive

Data

fa	ChFunction
function to be differentiated	

5.13 ChFunctionIntegrate object

Integral of a **ChFunction** object, using numerical method of trapezoidal rule (so this function can work only in a specified finite interval on x).

This object inherits all methods of base **ChFunction**, like $y()$, $y_{.dx}()$ etc.
 This object supports user construction via new operator, for example:
`mfx = new ChFunctionIntegrate`

Inheritance

Object - **ChFunction** - **ChFunctionIntegrate**

Data

fa	ChFunction
function to be integrated	
C_start	double
initial value C_0 for the definite integral	
x_start	double
starting value x_s for computing integral	
x_end	double
ending value x_e for computing integral	
num_samples	int
total number of samples between x_s and x_e for numerical evaluation of trapezoidal rule.	

Functions

ComputeIntegral ($()$)

Updates the value of the integral. (Note: this function MUST be called if the argument function changes, because may be that integral updating is not automatic).

5.14 ChFunctionRecorder object

Function which records points (x,y) , building a polynomial interpolation of points, and can be later evaluated for whatever value, $y=y(x)$.

Linear interpolation between recorded values is used, and custom numerical differentiation is used for speed and acceleration. In intervals where no points are recorded, value is zero.

This object inherits all methods of base **ChFunction**, like $y()$, $y_{.dx}()$ etc.
 This object supports user construction via new operator, for example:
`mfx = new ChFunctionRecorder`

Inheritance

Object - **ChFunction** - **ChFunctionRecorder**

Functions

add_point (double X, double Y)

Set the point (X,Y). If another point at same X exists, the Y value is overwritten.

add_point (double X, double Y, double DX)

Set the point (X,Y). If another point at same X exists, the Y value is overwritten. Also, other points in interval [X...X+DX] are reset to zero.

reset ()

Deletes all recorded points.

5.15 ChFunctionSequence object

Represents a sequence of functions **ChFunction**, each lasting a specified interval.

This is very useful to build complex motion laws, by concatenating many **ChFunction**, like **ChFunctionConstAcc** etc.

This object inherits all methods of base **ChFunction**, like **y()**, **y_dx()** etc.

This object supports user construction via new operator, for example:

```
mfx = new ChFunctionSequence
```

Inheritance

Object - **ChFunction** - **ChFunctionSequence**

Data

start double
x value at which the sequence starts (for x_istart, y=0)

Functions

int **insert** (ChFunction F, double D, double W, bool C0, bool C1, bool C2, int Z)

Inserts a function F in the sequence. The interval of F will have a duration D, and C0, C1, C2 flags can impose position, speed and acceleration continuity with previous function in sequence. The function is inserted at the Z-th position in list of already added functions, or at beginning of sequence if Z=0, or at the end if Z= -1. You can also use the simplified syntax as follows:

int **insert** (ChFunction F, double D, int Z)

Inserts a function F in the sequence. The interval of F will have a duration D. The function is inserted at the Z-th position in list of already added functions, or at beginning of sequence if Z=0, or at the end if Z= -1.

int **append** (ChFunction F1, double D1, ChFunction F2, double D2, ...)

Easy command to insert many children functions at once. Just provide n couples of functions F_i and durations D_i as arguments. If some functions are already inserted before this command, the new functions are appended at the end.

get_fn (int Z)
ChFunction
Gets the function at Z-th position. If Z=0 gets head, if Z=-1 gets tail.

kill_fn (int Z)
ChFunction
Deletes the function at Z-th position. If Z=0 deletes head, if Z=-1 deletes tail.

n_node (int Z)
ChFunction
Gets the function node (object with properties 'fx' and 'duration', at Z-th position. If Z=0 gets head, if Z=-1 gets tail.

setup ()
ChFunction
When changing the duration of some already-inserted function node, you may need to call this command just after, to rearrange the sequence.

5.16 ChFunctionJscript object

Evaluation of a generic Javascript function, or Javascript formulas.

This object inherits all methods of base **ChFunction**, like $y()$, $y_dx()$ etc. This object supports user construction via new operator, for example:
`mfx = new ChFunctionJscript`

Inheritance

Object - ChFunction - ChFunctionJscript

Data

command String
Javascript command to be executed, returning a value for a specific "x" value. For example it could be $\sin(x)$ or $x*2.3$ or $\text{myfunc}(x)$.

6 Objects: multibody items

This chapter contains the description of objects which are used to build physical systems.

Often, these objects are introduced by the user in the 3D editor by using the mouse and the **Realsoft3D** tool, so you do not need to create them in Javascript, but you rather use Javascript to access their data and to modify their properties.

References to already created objects can be obtained, for example, by using the command `GetSys().Obj("name")` which fetches multibody objects by their name.

6.1 System object

Reference to a **System** object of `Chrono::R3D`, which includes all settings of the simulator, state vector, preferences, etc.

It is also a starting point to fetch other `Chrono::R3D` objects, with the `Obj()` member function.

Also provides member functions to perform system analysis such as assembly, dynamic simulation, etc.

Note: this object does not support user construction via new operator: you can access just the global System object via the global `GetSys()` function.

Inheritance

Object - `CHobj` - `System`

Data

t double

Current time of simulation.

tol double

Tolerance for Newton Raphson constraint solver.

Nbodies int

Number of bodies in system (read-only)

Nlinks int

Number of links in system (read-only)

Ndof int

Number of degrees of freedom (read-only)

Ndoc int

Number of scalar constraints, including one norm constraint per quaternion (read-only)

Nsysvars	int
Number of unknown variables, including constraint reactions (read-only)	
Nredundancy	int
Number of redundant constraints (read-only)	
Ncoords_w	int
Number of coordinates, using three angles per body, not quaternions (read-only)	
Ndoc_w	int
Number of scalar constraints, using three angles per body, not quaternions (read-only)	
Nsysvars_w	int
Number of unknown variables, using three angles per body, not quaternions (read-only)	
Gacc	Vector
Gravity acceleration	
step	double
Current integration step	
step_min	double
Limit on minimum integration step (for variable-step integrators)	
step_max	double
Limit on maximum integration step (for variable-step integrators)	
Y	Matrix
State vector (Nx1 matrix). The upper half of vector contains all positions, the lower part contains all speeds. Size depends on number of bodies in the system.	

Functions

Body or **Obj** (String N)
 Marker
 or
 Link..

Returns the Chrono::R3D Javascript object with name N: it can be a **Body** object, or a **Link** object, or a **Marker** object, or false if no Chrono::R3D object exists with that name.

Note that if you have modelled a scene where two or more objects have the same name, you must rename them with unique names.

Update ()

Updates all the system. You may call this after you have applied modifications to object coordinates.

Update_B2Y ()

Updates the systems state with the current position of the bodies in 3d scene, if they does not match.

UpdateGeometry ()

After you have performed coordinate changes to Chrono::R3D data structure of bodies, you may call this in order to update also the 3d representation.

bool **WireRefresh** ()

Call this after you changed coordinates of bodies or markers, in order to see changes in 3d windows, by refreshing the screen (also calls UpdateGeometry())

bool **Assembly** ()

Performs the assembly of the system, satisfying all constraints in terms of position, speed and acceleration.

Return true if assembly was impossible.

bool **AssemblyForces** ()

Performs the assembly of the system, satisfying all constraints in terms of position, speed and acceleration, and also finds the unknown reactions in links, for the current time (it is equivalent to a dynamicssimulation step with infinitesimal length).

Returns true if assembly was impossible.

bool **FrameDynamics** (double T)

Performs dynamical simulation (integration) from current time up to time T. More than one step may take place.

If you intend to perform a dynamic simulation showing results at fixed intervals, you can execute this function with uniform-increasing T, showing and recording values at each repetition. The smaller' time step is automatically arranged to match the frame timestep.

Returns error state (in case of problems during assembly, etc.)

bool **Dynamics** (double T)

Performs dynamical simulation (integration) from current time up to time T, by repeating internally multiple FrameDynamics() operations.

UndoSaveState ()

Save current system state (body positions,speeds and current time) into the state undo buffer. Later you can perform GetSys().UndoState() to restore that state.

UndoState ()

If you previously performed `UndoSaveState()`, with this command you restore the saved state of the system (body positions, speeds, time). Note that you should not delete or add bodies between `UndoSaveState()` and following `UndoState()` calls. This quick type of `UndoState()` is different from the 'full' Undo of the **Re-alsoft3D** user interface, because it just restores the system state vector and time (but changes in geometry, shapes, etc. are not restored).

Examples

```
msys = GetSys()
msys.Assembly();
print("The system has " + msys.Ndoc + " scalar constraints.")
```

6.2 Body object

Reference to a **Body** object of `Chrono::R3D`.

Note: this object does not support user construction via `new` operator: you should model the body in 3D editor with mouse and tools, then you can access the desired body object by name-fetching functions like, for example, `GetSys().Obj("body1")`

Inheritance

Object - `CHobj` - **Body**

Data

mass	double
Mass of rigid body	
inertia	Matrix
Matrix of inertia	
system	System
Reference to the multibody system	
p	Coordsys
Coordinate system (position and rotation) of the body	
p_dt	Coordsys
Time derivative of coordinates, i.e. speed.	
p_dtdt	Coordsys
Double time derivative of coordinates, i.e. acceleration.	

wvel	Vector
Angular speed vector, in local coordinates (read only)	
wacc	Vector
Angular acceleration vector, in local coordinates (read only)	
wvel_abs	Vector
Angular speed vector, in absolute coordinates (read only)	
wacc_abs	Vector
Angular acceleration vector, in absolute coordinates (read only)	
rot_axis	Vector
Current axis of rotation, normalized (read only)	
rot_angle	double
Current angle of rotation about axis (read only)	
A	Matrix
Rotation matrix, 3x3. (read only)	
A.dt	Matrix
Time derivative of rotation matrix, 3x3. (read only)	
A.dtdt	Matrix
Double time derivative of rotation matrix, 3x3. (read only)	
G1	Matrix
$[G]$ matrix for equation $W_t = [G_t]dq/dt$ (read only)	
lock	bool
If true, body is grounded.	
collide	bool
If true, collision detection is active for this body.	
impactC	double
Coefficient of restitution for impacts	

impactCt double
Coefficient of tangential restitution for impacts

Sfriction double
Static friction coefficient for sliding contacts

Kfriction double
Kinetic friction coefficient for sliding contacts

script_force Vector
Set here the force that you want to impose on body (applied in center of mass, and expressed in body coordinates). This is never reset by internal Chrono::R3D code (it just add it to the total internal forces) so its up to your scripts to keep updated the value of this force if you set it different to 0.

script_torque Vector
As script_force, but imposes torque. Torque must be expressed in body coordinates. This is never reset by internal Chrono::R3D code (it just add it to the total internal torques) so its up to your scripts to keep updated the value of this torque if you set it different to 0.

Functions

Update (
)
Updates inner data of body.

Move (Coordsys C)
Move incrementally the rigid body by coordinate system C.

Marker **Obj** (String N)
or Force
Returns a **Marker** or **Force** Javascript object belonging to this rigid body, if the name matches N. If no success, returns false.

Geometry **Geo** (String N)
Returns a **Geometry** object included in the sub level of this rigid body, if the name matches N. If no success, returns false.

6.3 Marker object

Reference to a **Marker** object of Chrono::R3D .

Note: this object does not support user construction via new operator: you should

model the marker in 3D editor with mouse and tools, than you can access the desired marker object by name-fetching functions like, for example, `GetSys () .Obj ("marker1")`

Inheritance

Object - CHobj - Marker

Data

body	Body
Reference to parent rigid body, owning this marker.	
motion_x	ChFunction
Motion of marker respect to coordinate X of parent body.	
motion_y	ChFunction
Motion of marker respect to coordinate Y of parent body.	
motion_z	ChFunction
Motion of marker respect to coordinate Z of parent body.	
motion_axis	Vector
Axis for marker rotation, if desired. Axis is in body coordinates.	
motion_ang	ChFunction
Rotation of marker about motion_axis	
p	Coordsys
Absolute coordinates of marker.	
p_dt	Coordsys
Derivative of absolute coordinates of marker (speed).	
p_dtdt	Coordsys
Double derivative of absolute coordinates of marker (acceleration).	
relp	Coordsys
Relative coordinates of marker.	
relp_dt	Coordsys
Derivative of relative coordinates of marker (speed).	
relp_dtdt	Coordsys
Double derivative of relative coordinates of marker (acceleration).	

restp Coordsys
 resting position' for marker, when using motion functions.

Functions

Update ()
 Updates markers inner data.

6.4 Force object

Reference to a **Force** object of Chrono::R3D .

Note: this object does not support user construction via new operator: you should model the force in 3D editor with mouse and tools, than you can access the desired force object by name-fetching functions.

Inheritance

Object - CHobj - Force

Data

body Body
 Reference to parent rigid body

mode int
 if =0 the force represents a vector force, if =1, represents a torque.

align int
 if= 0 the force rotates together with parent body, if=1 doesnt change dir.

frame int
 if= 0 the force moves together with parent body, if=1 doesnt move.

point Vector
 point of application of force (or torque) in absolute space.

relpoint Vector
 point of application of force (or torque) in body-relative space.

dir Vector
 direction of force (or torque) in absolute space.

reldir Vector
 direction of force (or torque) in body-relative space.

f	double
modulus of force	

Functions

Update	()
Update forces internal data.	

6.5 Link object

Reference to a **Link** object of Chrono::R3D . Many link types are inherited from this base class.

Note: this object does not support user construction via new operator: you should model the link in 3D editor with mouse and tools, than you can access the desired link object by name-fetching functions.

Inheritance

Object - CHobj - Link

Data

system	System
Reference to multibody system.	
marker1	Marker
Reference to marker 1.	
marker2	Marker
Reference to marker 2.	
disabled	bool
If true, link is disabled.	
doc	int
Number of scalar constraints enforced by this link. Read only.	
relM	Coordsys
Coordinate of marker 1 respect to marker 2 (rel.position). Read only.	
relM.dt	Coordsys
Derivative of coordinate of marker 1 respect to marker 2 (rel. speed). Read only.	
relM.dtdt	Coordsys

Double deriv. of coord. of marker 1 resp. to marker 2 (rel.accel.). Read only.

rel_angle Vector

Relative angle of rotation of marker1 respect to marker2. Read only.

rel_axis double

Relative axis of rotation of marker1 respect to marker2. Read only.

rel_Wvel Vector

Relative ang. speed of marker1 resp. to marker2, in marker2 csys. Read only.

rel_Wacc Vector

Ang. acceleration of marker1 respect to marker2, in marker2 csys. Read only.

dist double

Relative polar distance of the two markers. Read only.

dist_dt double

Relative polar speed of the two markers. Read only.

script_force Vector

Set here the force that you want to impose between marker 1 and 2 (applied in origin of marker1, and expressed in marker2 coordinates). This is never reset by internal Chrono::R3D code (it just add it to the total internal forces) so its up to your scripts to keep updated the value of this force if you set it different to 0.

script_torque Vector

As script_force, but imposes torque. Torque must be expressed in marker2 coordinates. This is never reset by internal Chrono::R3D code (it just add it to the total internal torques) so its up to your scripts to keep updated the value of this torque if you set it different to 0.

react_force Vector

Read only: current reactions in link (force part, in marker2 coordinate system)

react_torque Vector

Read only: current reactions in link (torque part, in marker2 coordinate system)

int_force Vector

Read only: current internal force in link, caused by springs etc., but not by reactions (force part, in marker2 coordinate system)

int_torque Vector
 Read only: current internal torque in link, caused by springs etc., but not by reactions
 (torque part, in marker2 coordinate system)

Functions

Update ()
 Update links inner data.

MaskGetN (int N)
 Return state of N-th mask flag.

MaskSetN (int N, int T)
 Set state of N-th mask flag.

6.6 LinkSpring object

Reference to a **Link spring** object of Chrono::R3D which defines a spring-damper-force system, acting along the polar distance of the two markers.

Note: this object does not support user construction via new operator: you should model the link in 3D editor with mouse and tools, than you can access the desired link object by name-fetching functions.

Inheritance

Object - CHobj - Link - LinkSpring

Data

k double
 Stiffness K of spring (force/deformation)

r double
 Damping R of spring (force/deformation speed)

f double
 Force F acting along the direction of spring, if needed.

d_rest double
 Resting length of spring.

mod.f.time ChFunction
 Modulation of force F as time function: $F^*=f(t)$

mod_k_d	ChFunction
Modulation of stiffness K as distance function: $K^*=f(D)$	
mod_r_d	ChFunction
Modulation of damping R as distance function: $R^*=f(D)$	
mod_k_speed	ChFunction
Modulation of stiffness K as speed function: $K^*=f(dD/dt)$	
mod_r_speed	ChFunction
Modulation of damping R as speed function: $R^*=f(dD/dt)$	
spring_react	double
Reaction in spring-damper-force system, in distance direction (read only).	

6.7 LinkLock object

Reference to a **Link** object of Chrono::R3D of subclass LinkLock. Many link types are obtained from this type (for example, revolute joints, spherical joints,...).

Note: this object does not support user construction via new operator: you should model the link in 3D editor with mouse and tools, than you can access the desired link object by name-fetching functions.

Inheritance

Object - CHobj - Link - LinkLock

Data

motion_x	ChFunction
Motion law of marker 1 respect to axis X of marker 2.	
motion_y	ChFunction
Motion law of marker 1 respect to axis Y of marker 2.	
motion_z	ChFunction
Motion law of marker 1 respect to axis Z of marker 2.	
motion_angleset	int
Type of angles (Eulero, ..) for imposing rotation of marker1.	
motion_a1	ChFunction
Rotation law of marker1 respect to marker2, first angle.	

motion_a2	ChFunction
Rotation law of marker1 respect to marker2, second angle.	
motion_a3	ChFunction
Rotation law of marker1 respect to marker2, third angle.	
motion_axis	Vector
With default motion_angleset, rotation is of type angle about axis': the angle is motion_a1 and the axis is this motion_axis (considered in marker2 coord.)	
delta	Coordsys
Imposed coordinate of marker 1 respect to marker 2 (read only).	
delta_dt	Coordsys
Imposed speed of marker 1 respect to marker 2 (read only).	
delta_dtdt	Coordsys
Imposed accelerations of marker 1 respect to marker 2 (read only).	

6.8 LinkScrew object

Reference to a **Link Screw** object of Chrono::R3D defining a screw joint.

Inheritance

Object - CHobj - Link - LinkLock - LinkScrew

Data

tau	double
Tau, the transmission ratio of screw [meters/radian].	
thread	double
Thread of screw [meters]. Note: tau=thread/2PI.	

6.9 LinkGear object

Reference to a **Link Gear** object of Chrono::R3D defining a gear.

Inheritance

Object - CHobj - Link - LinkLock - LinkGear

Data

tau	double
------------	--------

Transmission ratio [rad/rad] between two wheels.

alpha double
Alpha pressure angle depends on teeth geometry, usually 20-

beta double
Beta helix angle -usually 0 .. 20-

inner int
If true, one gear is inside the other (with inner teeth).

6.10 LinkLinActuator object

Reference to a **Link** object of Chrono::R3D defining a **linear actuator** .

Inheritance

Object - CHobj - Link - LinkLock - LinkLinActuator

Data

dist_funct ChFunction
Imposed motion law (or recorded motion, if in learn mode).

offset double
Initial offset (for which motion coordinate is null).

learn int
If true, the constraint is free to move on motion axis, but motion is recorded in dist_funct (it learns the motion, for later use).

tau double
Transmission coefficient of screw and reducer. Optional.

eta double
Transmission efficiency of screw and reducer. Optional.

inertia double
Inertia of motor-reducer, on motor axis. Optional.

re_rot ChFunction
Recorded motor rotation. Optional.

re_torque ChFunction

Recorded motor torque. Optional.

6.11 LinkEngine object

Reference to a **Link** object of Chrono::R3D defining an **engine** .

Inheritance

Object - CHobj - Link - LinkLock - LinkEngine

Data

rot_funct	ChFunction
Imposed rotation time-function (if in impose rotation mode').	
spe_funct	ChFunction
Imposed speed time-function (if in impose speed mode').	
tor_funct	ChFunction
Imposed torque time-function (if in impose torque mode').	
torquew_funct	ChFunction
Torque(w) modulation, for torque depending on ang.speed.	
learn	int
If true, motion is free and it is recorded (in rot_funct or spe_funct, depending on eng_mode).	
apply_reducer	int
If true, rot_funct spe_funct tor_funct are used to define rotation, speed or torque on motor axis before the reducer. Otherwise, they are considered on the output shaft of the reducer.	
eng_mode	int
Can switch between 0= impose rotation, 1= impose speed, 2= impose torque modes.	
shaft_mode	int
Can switch between alternate modes of enforcing additional constraints for the shaft between the two markers.	
tau	double
Reducer transmission ratio.	
eta	double

Reducer transmission efficiency.

inertia	double
Reducer inertia (plus the inertia of the reducer gears, as seen by motor shaft).	
rot	double
Rotation of shaft. Read-only.	
rot_dt	double
Speed of shaft. Read-only.	
rot_dtdt	double
Acceleration of shaft. Read-only.	
torque	double
Torque of shaft. Read-only.	
rerot	double
Rotation of motor shaft (before the reducer). Read-only.	
rerot_dt	double
Speed of motor shaft (before the reducer). Read-only.	
rerot_dtdt	double
Acceleration of shaft (before the reducer). Read-only.	
retorque	double
Torque of motor shaft (before the reducer). Read-only.	

6.12 LinkBrake object

Reference to a **Link** object of Chrono::R3D defining a **brake** .

Inheritance

Object - CHobj - Link - LinkLock - LinkBrake

Data

braking	double
Braking torque, when brake slips (not sticking)	
stick_ratio	double

When sticking (locked, zero speed), the brake can resist a torque less than the braking torque multiplied by this stick ratio (usually 1.01 - 1.20).

6.13 LinkPneumaticActuator object

Reference to a **Link** object of Chrono::R3D defining a **pneumatic actuator**.

Inheritance

Object - CHobj - Link - LinkLock - LinkPneumaticActuator

Data

offset	double
Length of actuator for zero stroke.	
comA	double
Instant value of valve A command. (0..10), with 0=open, 5= close, 10= exhaust.	
comB	double
Instant value of valve B command. (0..10), with 0=open, 5= close, 10= exhaust.	
pA	double
Instant pressure in chamber A. Read only.	
pB	double
Instant pressure in chamber B. Read only.	
pA_dt	double
Instant pressure derivative in chamber A. Read only.	
pB_dt	double
Instant pressure derivative in chamber B. Read only.	
pneu_F	double
Instant force on stroke, caused by pneumatic pressure. Read only.	
pneu_pos	double
Instant stroke position. Read only.	
pneu_pos_dt	double
Instant stroke speed. Read only.	
ci	double

Intake conductance.

Co double

Exhaust conductance.

Bi double

Beta coefficient for intake.

Bo double

Beta coefficient for exhaust.

Ps double

Pressure of external air, to set exhaust pressure.

Pma double

Pressure of intake of valve A.

Pmb double

Pressure of intake of valve B.

L double

Length of stroke.

Wa double

Dead volume of chamber A.

Wb double

Dead volume of chamber B.

area double

Area of cylinder.

alfa double

Area reduction coefficient in chamber B, because of stroke rod.

gamma double

Viscous friction coefficient.

6.14 LinkWheel object

Reference to a **Link** object of Chrono::R3D defining a **wheel-ground** model.

Inheritance

Object - CHobj - Link - LinkLock - LinkWheel

7 Other objects

In this chapter there is the documentation of various objects which can be used for many purposes.

7.1 Geometry object

Reference to a 'Geometry' object, that is whatever object in 3D view window (most often, you will need to reference meshes or Nurbs lines).

Note: this object does not support user construction via new operator: you should model the link in 3D editor with mouse and tools, than you can access the desired link object by name-fetching functions.

Inheritance

Object - Geometry

Data

P	Coordsys
Coordinate of the object	

Functions

Vector	Eval	(double U, double V, double W)
Returns the point whose coordinates are U,V,W. Example: for mesh surfaces, only U and V are useful, so it is $p=p(U,V)$, while for splines the U is the useful parameter, as $p=p(U)$. Note that U and V coordinates on nurb splines correspond to the isoparametric lines. Note: all parameters range in [0..1]		

Vector	Normal	(double U, double V, double W)
Same as Eval(), but returns the direction of the normal of the surface at point U,V. For volumes and splines, results may be unpredictable. Note: all parameters range in [0..1].		

7.2 Placer object

Reference to a **placer** object, that is a Chrono::R3D object which is used to position other objects along a straight direction, with a user specified amount.

Note: this object does not support user construction via new operator: you should model the link in 3D editor with mouse and tools, than you can access the desired link object by name-fetching functions.

Inheritance

Object - Geometry - Placer

Data

dist	double
Distance of positioning (child object will be placed along X axis of placer object).	
angle	double
Angle of rotation (child object will be rotated about an axis directed as the Z axis of placer object).	

7.3 PID object

This is a PID object: a Javascript object which simulates a basic Proportional-Integrative-Derivative digital controller. It can be used to implement simulations of devices with feedback-control.

This has no equivalent 3D object in the environment (it exists only in Javascript).

The total output o of the PID controller, depending on input variable i , is given by the formula which uses three gains (proportional, derivative, integrative):

$$o = K_p i + K_i \int i dt + K_d \frac{\partial i}{\partial t}$$

Usually the output o is some control variable (ex: the strength of an applied force) and the input is something to keep low (ex: control error, trajectory error, etc.).

Note: tuning the three gains may be a complex (try and guess) task, especially for the derivative ("damping") constant, and for the integral ("static") constant. If you use too high gains, the controlled system may show numerical and dynamical instability.

This object supports user construction via new operator, for example:

```
mv = new PID
```

Inheritance

Object - PID

Data

Kp	double
Proportional gain.	
Ki	double
Integrative gain.	
Kd	double
Derivative gain.	
In	double
Current value of input i . Read only (you must use Get_output() to set input and time at once).	
In_dt	double
Current value of input derivative $\partial i / \partial dt$. Read only.	

In_dtdt double
Current value of input double derivative $\partial^2 i / \partial t^2$. Read only.

Out double
Current value of output o (last computed value). Read only.

Pcomp double
Proportional component of PID control output o . Read only.

Icomp double
Integrative component of PID control output o . Read only.

Dcomp double
Derivative component of PID control output o . Read only.

Functions

double **Get_output** (double new_input, double new_time)

During the simulation, call this function to update the input of the controller (the argument `new_input`) and the time of change (the argument `new_time`). Given that new state, the resulting output is returned.

If you repeat this function at successive time intervals, the PID object will automatically compute the time derivative and integral of input, by numerical differentiation. Then it is suggested that you use this function at small time intervals, in order to ensure the accuracy of the integrative and derivative contributions.

Returns the controller output.

Reset ()

Reset the internal integrator (if the simulation must rewind, or repeat, the accumulator of the integral contribution must be reset).

8 Objects: optimizers

Optimization objects contain the methods which allow you to solve complex multivariate problems of optimization.

You can choose between **local** , **global** or **hybrid** methods.

They find the variables which minimize (or maximize) a given objective function.

8.1 ChOptimizer object

This is the base class for all optimizer engines, such as the **Genetic Optimizer** for global optimization, or the **Gradient Optimizer** for local optimization.

This object supports user construction via new operator, for example:

```
my_opt = new ChOptimizer
```

Inheritance

Object - ChOptimizer

Data

objective string

The objective function. It must return a scalar value (the objective function to minimize or maximize).

opt_fx double

Read only. The final value of the objective function after optimization.

minimize int

If =1 minimizes, if =0 maximizes the objective function.

n_eval_fx int

Total number of objective function evaluations which were needed for last optimization. Read only.

n_eval_grad int

Total number of gradient evaluations which were needed for last optimization. Read only.

n_vars int

Total number of variables (these can be added with AddVar() function below. Read only.

Functions

bool Optimize ()
 Performs the optimization.
 This may be a complex and time-consuming process, it may take few seconds or even minutes, in case of complex multivariate genetic optimizations.

AddVar (string Var, double Min, double Max)
 Add a variable to the optimization engine. Variables are referenced through their names (string Var), and have a Min-Max range. For the **Genetic Optimizer**, Min-Max values are also used to set the initial range of random distribution for that variable.
 In case of multiple variables, use many times the command AddVar().

8.2 ChGeneticOptimizer object

This is a genetic optimization engine, which is best suited to find “global” optimum, even without an initial guess (where a local gradient optimization engine would fail). It can also work with non-continuous functions. However the drawback is that the genetic optimizer may be slow.

This object supports user construction via new operator, for example:
`my_opt = new ChGeneticOptimizer`

Inheritance

Object - ChOptimizer - ChGeneticOptimizer

Data

popsize int
 Initial population (number of individuals).

max_generations int
 Max number of generations (the evolutive simulation will be stopped after this generation).

selection int
 Selection method (0= roulette, 1= roulette and best, 2= norm geometric, 3= tournament).

crossover int
 Crossover method (0= arithmetic, 1= blend, 2= blend random, 3= heuristic, 4= disabled).

crossover_prob double
 Probability of crossover (mating probability, usually 0.3.. 0.5)

mutation int

Mutation method (0= uniform, 1= near boundary)

mutation_prob double
Probability of mutation (usually low values, see default).

elite int
Elitism mode (0=off, 1=on, never kill the best).

Examples

```
// define the function to optimize, ex: f=(x-1)^2+(y-2)^2
function pollo() { mx = x-1; my = y-2; return mx*mx+my*my; }
// create an optimizer
op = new ChGeneticOptimizer();
// set some parameters, and set the variables
op.objective="pollo()";
op.minimize = 1;
op.AddVar("x",-10,+10);
op.AddVar("y",-10,+10);
op.population=30;
op.max_generations=100;
x = 5; y= 5;
// perform optimization and see the results
op.Optimize();
print("result x=",x, " y=",y, " optimized funct=", op.opt_fx);
```

8.3 ChLocalOptimizer object

This is a local optimization engine based on the “gradient search” method. This is best suited to find “local” optimum, but needs an initial guess. It should be used on continuous functions.

This object supports user construction via new operator, for example:
my_opt = new ChLocalOptimizer

Inheritance

Object - ChOptimizer - ChLocalOptimizer

Data

max_iters int
Maximum number of iterations allowed.

max_evaluations int
Maximum number of function evaluations allowed.

tol_args double
Tolerance on arguments. Iterations are stopped if changes in arguments are lower than this value.

tol_fx	double
Tolerance on objective function. Iterations are stopped if changes in objective fx are lower than this value.	
initial_step	double
Initial step for gradient search. During iterations, this is automatically adjusted to best value.	
grad_step	double
Delta value for evaluation of gradient with numerical differentiation.	

Examples

```
// define the function to optimize, ex: f=(x-1)^2+(y-2)^2
function pollo() { mx = x-1; my = y-2; return mx*mx+my*my; }
// create an optimizer
op = new ChLocalOptimizer();
// set some parameters, and set the variables
op.objective="pollo()";
op.minimize = 1;
op.AddVar("x",-10,+10);
op.AddVar("y",-10,+10);
// initial (guess) values..
x = 2; y = 3;
// perform optimization and see the results
op.Optimize();
print("result x=",x, " y=",y, " optimized funct=", op.opt_fx);
```

8.4 ChHybridOptimizer object

This is a local optimization engine based on the “gradient search” method. This is best suited to find “local” optimum, but needs an initial guess. It should be used on continuous functions.

This object supports user construction via new operator, for example:
`my_opt = new ChHybridOptimizer`

Inheritance

Object - ChOptimizer - ChHybridOptimizer

Data

genetic_opt	ChGeneticOptimizer
Reference to the internal genetic optimizer , for 1st phase (global search).	
local_opt	ChLocalOptimizer
Reference to the internal local optimizer , for the 2nd phase (local refinement with gradient search).	

Examples

```
// define the function to optimize, ex: f=(x-1)^2+(y-2)^2
function pollo() {mx = x-1; my = y-2; return mx*mx+my*my;}
// create the optimizer
op = new ChHybridOptimizer();
// set some parameters, and set the variables
op.objective="pollo()";
op.minimize = 1;
op.AddVar("x",-10,+10);
op.AddVar("y",-10,+10);
// tune parameters of the genetic phase...
op.genetic_opt.population=30;
op.genetic_opt.max_generations=100;
x = 5; y= 5;
// perform optimization and print the results
op.Optimize();
print("resulting.. x=",x," y=",y, " optimized funct =", op.opt_fx);
```


9 Objects: GUI controls

Chrono::R3D controls can be drag and dropped into the user interface, for example sliders and buttons which allow you to make panels for controlling mechanisms with user interaction. Once you created these panels with drag-and-drop (use Customize menu, Available Objects Window, then pick the Chrono::R3D gadgets in the Misc. tab), such user-interface objects can be updated or modified accessed via Javascript.

You can choose between **local** , **global** or **hybrid** methods.

They find the variables which minimize (or maximize) a given objective function.

9.1 GuiControl object

This is the base class for all GUI (Graphical User Interface) objects which you can drag and drop into panels, namely the Chrono slider, the Chrono button, the Chrono checkmark, etc.

This object does NOT support user construction via new operator, because the creation of the object happens automatically when you drop it into a window, a panel or a toolbar. For example, go to Customize menu, Available Objects Window, then pick the Chrono slider and drop it into a window. Open its setting window (using the small grey button on the left of the slider), and enter MY_GUI_OBJ into the 'Identifier' field. From now on, you can access the object MY_GUI_OBJ with Javascript and you can change its properties or you can update it (until you close the window deleting it).

Inheritance

GuiControl

Data

variable string

A string with the name of a variable which will be automatically set when the value of the gadget changes. For example, if this property is "my_var" in the case of a Chrono-slider, you will get that each time the user moves the slider, the variable "my_var" is changed accordingly. The automatism does not work in the opposite way (you should use the Update() function to refresh the state of the control on the basis of a new value of the associated variable, see later.

callback string

Name of a Javascript function which is executed each time the control is modified by the user. For example, in case of a Chrono-button, if this is property is "print", the function print() will be automatically called each time the user press it.

label string

Label shown near the control.

id string

Read only. The name of this object.

Functions

Update ()

Updates the value of the control, depending on the actual value of its javascript variable. For example, in case of a Chrono slider, the slider knob is moved to the value of the variable whose name can be set either with the graphical interface (open the setting window for the slider), or by modifying the `variableString` property of this Javascript object. In fact you should update the GuiControl objects by yourself, because updating is not automatic when some script changes the value of the associated Javascript variables (but viceversa is automatic).

10 Utility functions

Here we describe global functions and utilities which can be useful for your scripts.

These functions can be accessed in whatever place of your Javascript files or commands.

10.1 Animation control

AnimPlay ()

Play the current animation in 3D editor, until the last frame is reached (starting from current time), just like pressing the "play" button on the screen.

The animation thread is asynchronous: as soon as animation starts, the next Javascript statements are processed while animation is running. There is another way to perform animations with JS commands: the System object (see later) has the `FrameDynamics` member function, which can be repeated in a `for ()` loop.

AnimStop ()

Stops the animation.

AnimGotoTime (double)

Jumps to a specified time (in seconds) of 3D animation. Animation is not played in-between. Jump to 0 to rewind.

double **AnimGetTime** ()

Returns the current time of 3D animation, in seconds.

bool **AnimIsPlaying** ()

Returns true if asynchronous animation has been started, and still running.

Examples

```
//---- Getting objects etc.----

mygeom = Geo("spline1");    // if not found, ->null
msys = GetSys();             // if no chrono sys, ->null

print( "current Chrono system time: ", msys.t );

attrs( msys );
attrs( mygeom );
attrs( globals() );
```

10.2 Other

`bool` **WireRefresh** ()

Performs an asynchronous wire-frame refresh of views.

(In fact most times you modify attributes of Chrono::R3D objects via Javascript, the view windows are not automatically refreshed, for performance reasons, so it may be necessary to call WireRefresh() right after).

10.3 Marker placement utilities

These functions are useful when you want to place a marker along a 3D line (or over a 3D surface) by specifying the U,V coordinates on the surface (or the U coordinate along the line).

This is useful, for example, if you want to set up a parametric model where the coordinates of the markers (hence the position of some links) will depend on variables. Then, you can modify the variables with for() loops to see how the shape of your mechanism changes. Or, even better, you can create an objective function which must be optimized, where the Chrono::R3D optimizer can act on marker (link) positions by changing the variables which place them along some lines or surfaces: you will be able to find which is the best position for a marker in order to get the best objective function (for example, the best position for a revolute joint along a line, in order to have the lowest reaction in another joint, etc.).

The functions are the following (their source code can be seen in `chrono_startup.js`, and is very simple)

marker_at_pos (Marker M, Vector V)

Places a marker M so that it will have its origin in point V, in absolute coordinates.

marker_on_line (Marker M, Geometry G, double U)

Places a marker M on a line G (it may be a 3D spline, but also other generically evaluable 3D primitives can work), with parametric coordinate U. Note that U=0 for beginning of line, and U=1 for end of line.

marker_on_mesh (Marker M, Geometry G, double
U, double V)

Places a marker M on a mesh G (it may be a 3D rectangle, but also other generically u/v evaluable 3D primitives can work, like curved Nurbs surfaces), on the parametric coordinate U,V. Note that parameters U V can range in 0..1.

marker_align_to_geom (Marker M, Geometry G)

Rotates a marker so that it will have its axes aligned to the X,Y,Z axes of the 3D object G (the G object could be whatever geometric shape). Note that position is not affected: only rotation.

marker_rotate (Marker M, Geometry G, double A,
Vector V)

Rotates a marker with A radians of rotation about the V axis. The V axis must be a normalized vector, and must be considered in the coordinate system of geometric object G (for example, a 3d cube, or whatsoever). The position (origin) of marker is not affected: only rotation happens. Thank to the reference G, multiple calls of marker.rotate are not cumulative rotations, then for example you can go back to original marker position by simply using an angle A=0.

10.4 Jacobian computing

Suppose you have a multi-variable vectorial function of the type $\mathbf{X} = \mathbf{Y}(\mathbf{X})$, with vectors $\mathbf{Y} \in \mathbb{R}^n$ and $\mathbf{X} \in \mathbb{R}^m$, you may want to compute the jacobian matrix $[J]$ with $[J] \in \mathbb{R}^{n \times m}$. Consider it to be a sort of table of partial derivatives', as its elements are $[\partial Y_i / \partial X_j]$.

In CHRONO-Javascript you can use the following function to compute generic jacobians, by numerical differentiation:

Matrix **ch_jacobian** (Array X, Array Y, String F)

Array X is an array of M strings, each with the name of the independent variable, for example ["x1", "x2"].

Array Y is an array of N strings, each with the name of the dependent variable, for example ["y1", "y2", "y3"].

String F is the name of a function which computes the dependent variables as functions of the independent ones (such function should be already defined).

Returns the NxM jacobian matrix, by performing M numerical differentiations of the F function.

Note: before using ch_jacobian(), it is wise that you initialize the independent variables with proper values, because may be that your jacobian is non-linear and depends on X where it is evaluated (i.e. it is $[J] = [J(X)]$)

Examples

```
// define a function which sets output=f(input),
function pollo() { y1= x1*x1; y2= x1*x2}

// best remember to initialize inputs!
x1 = 1; x2 = 4;

// now compute Jacobian, telling also the name
// of input and outputs..
mJac = ch_jacobian(["x1","x2"] , ["y1","y2"] ,"pollo()");
mJac.Print();
```

10.5 Plotting utilities

Geometry **PlotXY** (Matrix X, Matrix Y)

Creates a line in 3D space (you will see it in front view window) whose coordinates are given by two parameter matrices X,Y (both are column-vectors of data with same number of rows, and one column). The result will stay on XY plane, that is all points will have Z=0.

Geometry **PlotXYZ** (Matrix X, Matrix Y, Matrix Z)

Creates a line in 3D space (you will see it in 3d view windows) whose coordinates are given by the parameter matrices X,Y,Z (these are column-vectors of data with same number of rows, and one column. The x,y,z coordinate of the i-th point will be fetched from values at the i-th rows of matrices X,Y,Z).

Geometry **PlotPolar** (Matrix R, Matrix A)

Creates a line in 3D space (you will see it in front view window) whose polar coordinates are given by two parameter matrices R,A (both are column-vectors of data with same number of rows, and one column). Note: R is the vector of radii, and A is the vector of corresponding phases (angles measured in radians!).

Geometry **PlotMeshXYZ** (Matrix X, Matrix Y, Matrix Z, int K)

Creates a mesh in 3D space (you will see it in 3d view windows) whose coordinates are given by parameter matrices X,Y,Z (all with same number of rows and columns)
Note: mesh will have NxM points, where N is the number of rows of X,Y,Z matrices, and M is the number of columns of X,Y,Z. Hence, rows of matrices are mapped in U direction of mesh, columns are mapped in V direction of mesh. Note: resulting mesh will be a K-order Nurbs surface (K=2 : linear, K=3: quadratic, K=4: cubic, etc.). Returns the created mesh.

Geometry **PlotMeshOnSurf** (Matrix X, Geometry S, int K)

Creates a mesh in 3D space (you will see it in 3d view windows) which is offset from a surface S by distances in matrix X.
Note: mesh will have NxM points, where N is the number of rows of X, and M is the number of columns of X. Hence, rows of matrix are mapped in U direction of mesh, columns are mapped in V direction of mesh. Note: resulting mesh will be a K-order Nurbs surface (K=2 : linear, K=3: quadratic, K=4: cubic, etc.). Returns the created mesh.

Examples

```
//---- How to plot a mesh in 3d space ----

// Find a 3d mesh named "mesh1" in tree of 3D objects.
// If not existing, please create it! (es. a rectangle)
me = Geo("mesh1");

// matrix to plot
mat = new Matrix; mat.Reset(20,25);
// fill matrix with values v=f(u,v)
for (ir=0; ir <20; ir++)
  for(ic=0; ic <25; ic++)
  {
    mat.SetEl(ir,ic,
      0.02*sin(12*ir/20.0)+
      0.02*sin(20*ic/25.0) );
  }
```

```
// plot the matrix!
PlotMeshOnSurf(mat, me, 2)
```

Plot

(Matrix X, Matrix Y)

Given two column matrices (two vectors of values), opens a plotting window to draw the function $f(x,y)$.

Vectors X and Y must have the same size (nx1 matrices).

Examples

```
vx = new Matrix;
vy = new Matrix;
vx.Reset(50,1);
vy.Reset(50,1);

for (ind = 0; ind < 50; ind++)
{
    // compute x and y
    x = ind*0.05;
    y = sin(x)*exp(x/2);

    // fill the X and Y vectors to plot
    vx.SetEl(ind, 0, x);
    vy.SetEl(ind, 0, y);
}

Plot(vx,vy);
```