DynaMo Library User Documentation

last updated April 19th, 2000

Bart Barenbrug

GNU LIBRARY GENERAL PUBLIC LICENSE

Version 2, June 1991

[This is the first released version of the library GPL. It is
numbered 2 because it goes with version 2 of the ordinary GPL.]

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licenses are intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users.

This license, the Library General Public License, applies to some specially designated Free Software Foundation software, and to any other libraries whose authors decide to use it. You can use it for your libraries, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library, or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link a program with the library, you must provide complete object files to the recipients so that they can relink them with the library, after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

Our method of protecting your rights has two steps: (1) copyright the library, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the library.

Also, for each distributor's protection, we want to make certain that everyone understands that there is no warranty for this free library. If the library is modified by someone else and passed on, we want its recipients to know that what they have is not the original version, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that companies distributing free software will individually obtain patent licenses, thus in effect transforming the program into proprietary software. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License, which was designed for utility programs. This license, the GNU Library General Public License, applies to certain designated libraries. This license is quite different from the ordinary one; be sure to read it in full, and don't assume that anything in it is the same as in the ordinary license.

The reason we have a separate public license for some libraries is that they blur the distinction we usually make between modifying or adding to a program and simply using it. Linking a program with a library, without changing the library, is in some sense simply using the library, and is analogous to running a utility program or application program. However, in a textual and legal sense, the linked executable is a combined work, a derivative of the original library, and the ordinary General Public License treats it as such.

Because of this blurred distinction, using the ordinary General Public License for libraries did not effectively promote software sharing, because most developers did not use the libraries. We concluded that weaker conditions might promote sharing better.

However, unrestricted linking of non-free programs would deprive the users of those programs of all benefit from the free status of the libraries themselves. This Library General Public License is intended to permit developers of non-free programs to use free libraries, while preserving your freedom as a user of such programs to change the free libraries that are incorporated in them. (We have not seen how to achieve this as regards changes in header files, but we have achieved it as regards changes in the actual functions of the Library.) The hope is that this will lead to faster development of free libraries.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a “work based on the library” and a “work that uses the library”. The former contains code derived from the library, while the latter only works together with the library.

Note that it is possible for a library to be covered by the ordinary General Public License rather than by this special one.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any software library which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Library General Public License (also called “this License”). Each licensee is addressed as “you”.

   A “library” means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

   The “Library”, below, refers to any such software library or work which has been distributed under these terms. A “work based on the Library” means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term “modification”.)

   “Source code” for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.
1. You may copy and distribute verbatim copies of the Library’s complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a. The modified work must itself be a software library.

   b. You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.

   c. You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.

   d. If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

      (For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

   Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

   In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

   Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU

General Public License applies to all subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

   If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

   However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

   When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

   If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

   Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also compile or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

   You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

a. Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)

b. Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.

c. If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.

d. Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

   a. Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.

   b. Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.
9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.
10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.
11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

    If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

    It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

    This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.
12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so

NO WARRANTY

END OF TERMS AND CONDITIONS

1 Introduction

This is the user documentation for the Dynamo Library: a C++ class library developed at the Eindhoven University of Technology for handling dynamics and inverse dynamics motion behaviour in computer animation and simulation.

This documentation is structured as follows: first, a very brief overview is given of what (inverse) dynamics entails. Then the few things needed to connect the library to a host system are discussed. This involves implementing a few simple callbacks. Once these callbacks have been implemented, the library can be used. That is what the remainder of this documentation focuses on, by first giving some general hints about the usage of the library, and then presenting the API of the library. In this API description all relevant classes in the library are presented along with a description of each of their methods.

More in-depth documentation describing the theory on which the library is based can be found at the author's home page on the Internet at url '`http://www.win.tue.nl/win/cs/tt/bartb/`'. For questions, the author can be reached at e-mail address '`bartb@win.tue.nl`'.

2 Dynamics overview

Dynamics provides a way to specify motion behaviour for an animation or simulation. Unlike kinematics, where motions are specified in terms of translations and rotations, dynamics uses forces and torques to set the animation geometries in motion. The dynamics algorithms use the laws of physics to calculate the motion of objects. In order to be able to calculate the dynamic motions, some physical properties of the animated objects are required, such as the mass of the objects.

Inverse dynamics provides a tool to specify the dynamics forces on a higher level by stating properties of the dynamics objects. Inverse dynamics takes care of calculating and applying the (reaction) forces that maintain those properties. The properties are specified in so-called "constraints". A constraint might for example specify that two objects are to remain connected in a given manner at all times (with a ball joint, or a pin joint for example). The constraint then acts as a hinge between the two objects by calculating and applying the hinge force. The advantage of constraints is that they can be used in a declarative fashion: once a constraint is specified, the inverse dynamics engine will make sure it remains satisfied for the remainder of the animation/simulation (or until the constraint is "deactivated").

3 General notes

The task of the Dynamo Library is calculate the motions of the geometric objects that are under its control. These geometric objects exist outside the Dynamo library, and so the library itself maintains companion objects that make the information required for the dynamics calculation available to the objects in the library.

The calculations themselves are controlled by two system components: one of type `DL_dyna_system` (for the forward dynamics), and one of type `DL_constraint_manager` (for the inverse dynamics). The dyna system component provides the entry-point for the dynamics calculation in the form of the `dynamics()` method: this method is to be called once for each time step in the animation/simulation and will take care that all the dynamically controlled objects ('dynas' for short) will also make a time step according to their inertia, external forces and torques, and the constraints imposed on them. The method is also available as a C-function `dynamics()`.

Sometimes constraints are specified between a dyna and a geometric object that is not under control of the Dynamo library. Companion objects ('geos') for such objects are also maintained by the library. The dyna system object provides methods that enable you to register a geometric object as either a geo or a dyna. When the `dynamics` method make a time step, it assumes that the geos have already made their step: that way it can position the dynas in such a way that at the new time, the constraints are valid again.

The header files for the Dynamo library list more classes, attributes and methods than described in this document. However, only the methods documented here are intended to be used externally to the library. Other methods (which are marked as such) are present in the header definitions, but are only for use by other classes in the library. The classes in the Dynamo library have the prefix `DL_` prepended to their name to distinguish them from other classes.

Points and vectors are often expressed in the local coordinate system of some geo. So often you see as parameters a pair `DL_point *p, DL_geo *g`, expressing that point `p` is given in the local coordinate system of geo `g`. In such cases, `g` can be passed the value `NULL` to indicate that `p` is given in world coordinates.

4 Installation

Before the Dynamo library can be used, several things have to be done.

Since the dynamics calculations are carried out by the dyna system and the constraint manager object (see the General notes section), those two objects need be created during the initialization (if no inverse dynamics is used, the creation of the constraint manager object may be omitted). Moreover, the dyna_system object requires a motion integrator to be present (see the section on forward dynamics), so a motion integrator of choice will have to be created and installed as well

For the information exchange between the actual geometric objects and their companions, a few very simple callbacks have to be implemented. These callbacks are grouped together in a class-definition `DL_dyna_system_callbacks`. Its API is given below, and the class is meant to be used as a superclass for an actual implementation. An object of this class also needs to be created during the initialization phase to serve as a companion for the dyna system object.

An object of this class is used as companion to the dyna_system. Since the dyna_system also needs a reference to a force_drawer, this callbacks class inherits from the `DL_force_drawer` class so its methods can (optionally) be implemented too and are easily accessible.

```
class DL_dyna_system_callbacks : public DL_force_drawer {
    void update_dyna_companion(DL_dyna*);
    void get_new_geo_info(DL_geo*);
    void get_first_geo_info(DL_geo*);
    void check_inertiatensor(DL_dyna*);
    void do_collision_detection();
    void Msg(char*, ...);
}
```

Only the first two methods need to be implemented: the others are optional since they have a default implementation:

`void DL_dyna_system_callbacks::update_dyna_companion(DL_dyna *d)`

This method should copy the newly calculated position and orientation of a dyna to its geometric companion object after its new position is calculated.

`void DL_dyna_system_callbacks::get_new_geo_info(DL_geo *g)`

This method has the same function as the `update_dyna_companion` method, but does its copying in the other direction. It is used to copy new information from a non-dynamically controlled geometric object to its companion in the Dynamo library. Its argument provides a reference to the `DL_geo` who's position and orientation need to be updated. The `DL_geo` class has a method `move` especially for this purpose. A reference to the companion object of this `DL_geo` can be retrieved using its `get_companion()` method. The difference between this callback and the `get_first_geo_info` callback is that the latter is used to initialise a companion (so it has to set the motion state for time `t` as well as for time `t+h`), and this callback is used to increment the time by shifting the next motion state into the current one, and only specify the new motion motion state.

`void DL_dyna_system_callbacks::get_first_geo_info(DL_geo *g)`

This method is used to copy the information from a geometric object to its companion in the Dynamo library when the companion is created. Its argument provides a reference to the `DL_geo` who's positions and orientations need to be set. A reference to the companion object of this `DL_geo` can be retrieved using its `get_companion()` method. The default implementation of this method calls the `get_new_geo_info` method to obtain the position and orientation of the geometry, and copies this into both the new and current motion states, while setting the velocities to zero. For non-zero initial velocities, this method should be overriden.

`void DL_dyna_system_callbacks::check_inertiatensor(DL_dyna*)`

The dynamic calculations for each dyna depend on some of the physical properties of the dyna, such as its mass and its tensor of inertia. This callback gives the geometric companion object of a dyna a chance to update these properties if necessary (for example when the geometric companion object has been deformed).

`void DL_dyna_system_callbacks::do_collision_detection()`

The Dynamo library has a `collision` constraint for collision handling. But it relies on external collision detection to determine when such a constraint should be created. This callback is used to call the collision detector at the right moment. This callback should implement a call to the collision detector to do its work (possibly preceded by updating the positions of the geometries in the collision detector's datastructures), which in turn should create collision constraints upon detection of a collision.

The collision detection should use the estimates for the next motion state to determine if a collision is taking place. To this end, the geometry class has a set of methods (not listed in its interface description below, since they normally are only used by the other components of Dynamo, and only have valid values during the dynamics phase) for retrieving the geometry's properties based on the estimate for next frame's motion state. See the `geo.h` file for their interfaces.

When overlap is detected by a collision detector, it is usually a good idea to only create a collision constraint for the case where the colliding geometries are moving closer to each other: if they are already moving away from each other (maybe because of a collision constraint in the previous frame), the `"mirror the velocity"` approach of Dynamo's collision constraint only has adverse effects.

`void DL_dyna_system_callbacks::Msg(char *fmt, ...)`

This method is used by Dynamo to send messages (warnings, errors, notificications) to the outside world. The default implementation prints the message on stderr, but for windowing systems it is often better to reroute these messages to some window.

5 Example

The following code shows the use of the Dynamo library classes in creating a chain of cubes swinging due to gravity. The render functionality is provided in the `render.cpp` file

```
// A demo for the use of the Dynamo classes.
// It shows a chain of cubes that swings under influence of gravity.

#include "rungekutta2.h"
#include "ptp.h"

// How many cubes make up the chain?:
#define NCUBES 15

// Here is the class for our cubes, these function mainly as an intermediate
// between the dyna companion, the control routine, and the rendering software.
class MyCube {
  public:
    // the attributes of the cube:
    DL_point pos;        // the position of the center of the cube
    DL_matrix orient;    // the orientation of the cube
    DL_dyna* companion; // the companion which does all the dynamics
                        // calculations for us
    DL_ptp* link;       // used to connect this cube to another

    // the functions needed for the DL_dyna_callback to copy position and
    // orientation to/from the companion:
    void get_new_geo_info(DL_geo *g){
      g->move(&pos,&orient);
    }
    void update_dyna_companion(DL_dyna *d){
      pos.assign(d->get_position());
      orient.assign(d->get_orientation());
    }

    // show an example of a constraint: this method uses a point-to-point
    // constraint to connect one of the cube's corners to a corner of the
    // other cube.
    void ConnectTo(MyCube *cube){
      DL_point posm(1,1,1),posp(-1,-1,-1);
      link=new DL_ptp();
      if (cube) link->init(companion,&posm,cube->companion,&posp);
      else { // connect to the current position of the corner in the world.
        DL_point posw;
        companion->to_world(&posm,&posw);
        link->init(companion,&posm,NULL,&posw);
      }
    }
    void Disconnect(){
      if (link) delete link;
      link=NULL;
    }

    MyCube(DL_point& newpos){
    // constructor: its main task is creating and initialising the companion
      pos.assign(&newpos);
      orient.makeone();
      companion=new DL_dyna((void*)this);
      companion->set_mass(1);
      companion->set_inertiatensor(1,1,1);
      companion->set_velodamping(0.995); // introduce a bit of friction
      link=NULL;
    }
    ~MyCube(){
      if (link) delete link;
      delete companion;
    }
};

// we need to implement the callbacks that allow the dyna system to commicate
// its calculated positions and orientations to us:
class My_dyna_system_callbacks : public DL_dyna_system_callbacks {
  public:
    virtual void get_new_geo_info(DL_geo *g){
      ((MyCube*)(g->get_companion()))->get_new_geo_info(g);
    }
    virtual void update_dyna_companion(DL_dyna *d){
      ((MyCube*)(d->get_companion()))->update_dyna_companion(d);
    }
};

// rendering is taken care of elsewhere:
#include "render.cpp"

// here comes the main control function: it initialises the dyna system
// and then creates a chain of cubes
void main(){
  int i;
  MyCube* cube[NCUBES];
  My_dyna_system_callbacks *dsc=new My_dyna_system_callbacks();
  DL_m_integrator* my_int=new DL_rungekutta2();
  DL_dyna_system dsystem(dsc,my_int);
  DL_constraint_manager *constraints=new DL_constraint_manager();
  constraints->max_error=0.0001;

  DL_point pos(0.75*NCUBES,1.75*NCUBES,4*NCUBES);
  DL_vector vec(-2,-2,-2);
```

```
    for (i=0;i<NCUBES;i++){
      cube[i]=new MyCube(pos);
      pos.plusis(&vec);
    }

    cube[0]->ConnectTo(NULL);
    for (i=1;i<NCUBES;i++) cube[i]->ConnectTo(cube[i-1]);

    vec.init(0,-1,0);
    dsystem.set_gravity(&vec);
    my_int->set_stepsize(0.02);

    InitRender(dsystem,cube,NCUBES);

    // everything initialised. Now let the animation run:
    while (RenderCubes(cube)) {
      dsystem.dynamics();
      if (fabs(dsystem.time()-70)<0.01) cube[NCUBES/2]->Disconnect();
    }

    // all done: clean up:
    for (i=0;i<NCUBES;i++) delete cube[i];
    delete constraints; delete my_int; delete dsc;
  }
```

6 General classes

Before presenting the classes that are used for the dynamics, several standard classes for manipulating with points, vectors, matrices and lists have to be presented: objects of these classes are often used as parameters in the other classes. Many objects use the type `DL_Scalar` which resolves to `double` (but which can easily be changed if required).

6.1 Points

Here is the API for the 3-D point class:

```
class DL_point : public DL_ListElem {
    DL_Scalar  x,y,z;

    void       init(DL_Scalar,DL_Scalar,DL_Scalar);
    void       assign(DL_point*);
    boolean    equal(DL_point*);
    void      plus(DL_vector*,DL_point*);
    void      minus(DL_point*,DL_vector*);
    void      times(DL_Scalar,DL_point*);

    void      plusis(DL_vector*);
    void      minusis(DL_vector*);
    void      timesis(DL_Scalar);

    void      tovector(DL_vector*);

             DL_point();
             DL_point(DL_point*);
             DL_point(DL_Scalar,DL_Scalar,DL_Scalar);
             ~DL_point();
  }
```

`DL_Scalar DL_point::x`
`DL_Scalar DL_point::y`
`DL_Scalar DL_point::z`

These attributes hold the position of the point.

`void DL_point::init(DL_Scalar nx, DL_Scalar ny, DL_Scalar nz)`

This method assigns coordinates (`nx,ny,nz`) to the point

`void DL_point::assign(DL_point *p)`

This method assigns the coordinates of point `p` to this point.

`boolean DL_point::equal(DL_point *p)`

This method returns if point `p` and this point have the same coordinates.

`void DL_point::plus(DL_vector *v, DL_point *sum)`

This method assigns to point `sum` the sum of this point and vector `v`.

`void DL_point::minus(DL_point *p, DL_vector *diff)`

This method assigns to vector `diff` the difference between this point and point `p`.

`void DL_point::times(DL_Scalar f, DL_point *p)`

This method assigns to point `p` the product of this point and `f`.

`void DL_point::plusis(DL_vector *v)`

This method adds vector `v` to this point.

`void DL_point::minusis(DL_point *p)`

This method substracts vector `diff` from this point.

`void DL_point::timesis(DL_Scalar f)`

This method multiplies this point by a factor `f`.

`void DL_point::tovector(DL_vector *v)`

This method assigns the coordinates of this point to vector `v`.

6.2 Vectors

Here is the API for the 3-D vector class:

```
class DL_vector {
    DL_Scalar x,y,z;

    void      init(DL_Scalar,DL_Scalar,DL_Scalar);
    void      assign(DL_vector*);
    DL_Scalar norm();
    void      normalize();
    DL_Scalar inprod(DL_vector*);
    void      plusis(DL_vector*);
    void      minusis(DL_vector*);
    void      timesis(DL_Scalar);
    boolean   equal(DL_vector*);

    DL_Scalar get(int);
    void      set(int,DL_Scalar);

    void      crossprod(DL_vector*,DL_vector*);
    void      neg(DL_vector*);
    void      times(DL_Scalar,DL_vector*);
    void      plus(DL_vector*,DL_vector*);
    void      minus(DL_vector*,DL_vector*);
    void      topoint(DL_point*);

            DL_vector();
            DL_vector(DL_vector*);
            DL_vector(DL_Scalar,DL_Scalar,DL_Scalar);
            ~DL_vector();
}
```

`DL_Scalar DL_vector::x`

`DL_Scalar DL_vector::y`

`DL_Scalar DL_vector::z`

These attributes hold the coordinates of the vector

`void DL_vector::init(DL_Scalar nx, DL_Scalar ny, DL_Scalar nz)`

This method assigns coordinates (`nx,ny,nz`) to the vector

`void DL_vector::assign(DL_vector *v)`

This method assigns the coordinates of vector `v` to this vector.

`DL_Scalar DL_vector::norm()`

This method returns the length of the vector `void DL_vector::normalize()`

  This vector normalizes the length of the vector to one.

`DL_Scalar DL_vector::inprod(DL_vector *v)`

  This method returns the inner product of this vector and vector `v`.

`void DL_vector::plusis(DL_vector *v)`

  This method adds vector `v` to this vector.

`void DL_vector::minusis(DL_vector *v)`

  This method subtracts vector `v` from this vector.

`void DL_vector::timesis(DL_Scalar f)`

  This method multiplies this vector by factor `f`.

`boolean DL_vector::equal(DL_vector *v)`

  This method returns if this vector has the same coordinates as vector `v`.

`DL_Scalar DL_vector::get(int i)`

  This method returns `x` if `i` is zero, `y` if `i` is one, and `z` if `i` is two.

`void DL_vector::set(int i, DL_Scalar f)`

  This method assigns `f` to the `i`-th coordinate of this vector (i=0,1,2).

`void DL_vector::crossprod(DL_vector *v, DL_vector *c)`

  This method assigns the cross product of vector `v` and this vector to vector `c`.

`void DL_vector::neg(DL_vector *v)`

  This method assigns the negation of this vector to vector `v`.

`void DL_vector::times(DL_Scalar f, DL_vector *v)`

  This method assigns the product of this vector and `f` to `v`

`void DL_vector::plus(DL_vector *v, DL_vector *sum)`

  This method assigns the sum of this vector and `v` to `sum`.

`void DL_vector::minus(DL_vector *v, DL_vector *diff)`

  This method assigns the difference between this vector and `v` to `diff`.

`void DL_vector::topoint(DL_point *p)`

  This method assigns the coordinates of this vector to point `p`.

6.3 Matrices

Here is the API for the 3 by 3 matrix class:

```
class DL_matrix {
    DL_vector c0;
    DL_vector c1;
    DL_vector c2;

    void  makeone();
    void  makezero();
    void  normalize();

    void  assign(DL_matrix*);
    void  assign(DL_vector*,DL_vector*,DL_vector*);
    void  assign(DL_Scalar,DL_Scalar,DL_Scalar,
                 DL_Scalar,DL_Scalar,DL_Scalar,
                 DL_Scalar,DL_Scalar,DL_Scalar);

    DL_Scalar get(int,int);
    void  set(int,int,DL_Scalar);

    void  plus(DL_matrix*,DL_matrix*);
    void  minus(DL_matrix*,DL_matrix*);

    void  plusis(DL_matrix*);
    void  minusis(DL_matrix*);
    void  timesis(DL_Scalar);

    void  times(DL_matrix*,DL_matrix*);
    void  times(DL_Scalar,DL_matrix*);
    void  times(DL_vector*,DL_vector*);
    void  times(DL_point*,DL_point*);
    void  invert(DL_matrix*);
    void  transpose(DL_matrix*);
    void  timestranspose(DL_matrix*, DL_matrix*);
    void  transposetimes(DL_vector*, DL_vector*);
    void  jacobi(DL_matrix*,DL_vector*);
    void  diag_transpose_vec(DL_vector*,DL_vector*,DL_vector*);
    void  negcrossdiagcross(DL_point*,DL_vector*,DL_point*);
    void  diagcrosstranspose(DL_vector*,DL_point*,DL_matrix*);

    void  tensor(DL_vector*,DL_vector*);

    DL_matrix();
    DL_matrix(DL_vector*,DL_vector*,DL_vector*);
    ~DL_matrix();
}
```

`DL_vector DL_matrix::c0`

`DL_vector DL_matrix::c1`

`DL_vector DL_matrix::c2`

These attributes hold the three column vectors that constitute the matrix.

`void DL_matrix::makeone()`

This method assigns the unit matrix to this matrix

`void DL_matrix::makeone()`

This method assigns the zero matrix to this matrix

`void DL_matrix::normalize()`

This method normalizes each of the three column vectors of the matrix.

`void DL_matrix::assign(DL_matrix *A)`

This method assigns matrix `A` to this matrix.

`void DL_matrix::assign(DL_vector *v0, DL_vector *v1, DL_vector *v2)`

This method assigns the three vectors `v0`, `v1` and `v2` to the three column vectors of this matrix.

```
void DL_matrix::assign(DL_Scalar c0x, DL_Scalar c1x, DL_Scalar c2x,
                       DL_Scalar c0y, DL_Scalar c1y, DL_Scalar c2y,
                       DL_Scalar c0z, DL_Scalar c1z, DL_Scalar c2z)
```

This method assigns the given DL_Scalars to the elements of this matrix

`DL_Scalar DL_matrix::get(int i, int j)`

This method returns the `i`-th coordinate of the `j`-th column of this matrix (i,j=0,1,2).

`void DL_matrix::set(int i, int j,DL_Scalar f)`

This method assigns DL_Scalar `f` to the element at the `i`-th coordinate of the `j`-th column of this matrix (i,j=0,1,2)

`void DL_matrix::plus(DL_matrix *A, DL_matrix *sum)`

This method assigns to `sum` the sum of this matrix and matrix `A`.

`void DL_matrix::minus(DL_matrix *A, DL_matrix *diff)`

This method assigns to `diff` the difference between this matrix and matrix `A`.

`void DL_matrix::times(DL_matrix *A, DL_matrix *prod)`

This method assigns to `prod` the product of this matrix and matrix `A`.

`void DL_matrix::plusis(DL_matrix *A)`

This method adds matrix `A` to this matrix

`void DL_matrix::minusis(DL_matrix *A)`

This method substracts matrix `A` from this matrix.

`void DL_matrix::timesis(DL_Scalar f)`

This method multiplies this matrix by factor `f`.

`void DL_matrix::times(DL_vector *v, DL_vector *prod)`

This method assigns to `prod` the product of this matrix and vector `v`.

`void DL_matrix::times(DL_point *p, DL_point *prod)`

This method assigns to `prod` the product of this matrix and point `p`.

`void DL_matrix::times(DL_Scalar f, DL_matrix *prod)`

This method assigns to `prod` the product of this matrix and DL_Scalar `f`.

`void DL_matrix::invert(DL_matrix *A)`

This method assigns the inverse of this matrix to matrix `A`.

`void DL_matrix::transpose(DL_matrix *AT)`

This method assign the transpose of this matrix to matrix `AT`.

`void DL_matrix::timestranspose(DL_matrix *A, DL_matrix *prod)`

This method assigns the product of this matrix and the transpose of matrix `A` to matrix `prod`.

`void DL_matrix::transposetimes(DL_vector *v, DL_vector *prod)`

This method assigns the product of the transpose of this matrix and vector `v` to vector `prod`

`void DL_matrix::jacobi(DL_matrix *A, DL_vector *v)`

This method performs a jacobi decomposition on this matrix, calculating `A` and `v` such that the product of `A` and `diag(v)` and the transpose of `A` is this matrix. Here `diag(v)` is the diagonal matrix containing the eigenvalues of this matrix.

`void DL_matrix::diag_transpose_vec(DL_vector *d, DL_vector *v, DL_vector *r)`

This method assigns to vector `r` the product of `diag(d)`, the transpose of this matrix and `v`

`void DL_matrix::negcrossdiagcross(DL_point *p, DL_vector *d, DL_point *q);`

This method assigns to this matrix the negation of the crossmatrix induced by `p` times the diagonal matrix induced by `d` and crossmatrix induced by `q`.

`void DL_matrix::diagcrosstranspose(DL_vector *d, DL_point *q, DL_matrix *m);`

This method assigns to this matrix the product of the diagonal matrix induced by `d`, the crossmatrix induced by `q`, and the transpose of `m`.

`void DL_matrix::tensor(DL_vector *v, DL_vector *w)`

This method assigns the tensor matrix of `v` and `w` to this matrix.

6.4 Lists

The next two classes provide support for storing objects in linked lists. Each of the object will have to have DL_ListElem as one of its ancestors, and then the DL_List class can be used to list such items. An item can only occur in one list at a time.

The list element class does not have any public methods (only methods visible to DL_List), so its API is rather simple:

```
class DL_ListElem {
    DL_ListElem();
    ~DL_ListElem();
}
```

The DL_List is the actual list class which can store elements of type `DL_ListElem`. Here is its API:

```
class DL_List {
   void addelem(DL_ListElem*);
   void addbefore(DL_ListElem*,DL_ListElem*);
   void addafter(DL_ListElem*,DL_ListElem*);
   void remelem(DL_ListElem*);
   void delete_all();
   int length();

   DL_ListElem* getfirst();
   DL_ListElem* getnext(DL_ListElem*);
   DL_ListElem* getlast();
   DL_ListElem* getprev(DL_ListElem*);
   DL_ListElem* element(int);

      DL_List();
      ~DL_List();
}
```

`void DL_List::addelem(DL_ListElem *e)`

This method adds element `e` to the end of the list.

`void DL_List::addbefore(DL_ListElem *le, DL_ListElem *el)`

This methods adds element `le` to the list, in the position before element `el` (or at the tail of the list if `el` is `NULL`).

`void DL_List::addafter(DL_ListElem *le, DL_ListElem *el)`

This methods adds element `le` to the list, in the position after element `el` (or at the head of the list if `el` is `NULL`).

`void DL_List::remelem(DL_ListElem *e)`

This method removes element `e` from the list

`void DL_List::delete_all()`

This method deletes all elements in the list (thereby making the list empty). The elements are actually deleted, not just removed from the list.

`int DL_List::length()`

This method returns the number of elements stored in the list.

`DL_ListElem* DL_List::getfirst()`

This method returns a reference to the first element in the list (or `NULL` if the list is empty). Use this method in combination with `getnext` to traverse this list elements

`DL_ListElem* DL_List::getnext(DL_ListElem *e)`

This method returns a reference to the element in the list following element `e`. This method is often used to traverse the list in the following way (`T` is a type inheriting from `DL_ListElem`):

```
T *my_elem=my_list->getfirst();
while (my_elem) {
  // do something with my_elem
  my_elem=(T*)my_list->getnext(my_elem);
}
```

`DL_ListElem* DL_List::getlast()`

This method returns a reference to the last element in the list (or `NULL` if the list is empty). Use this method in combination with `getprev` to traverse this list elements in reverse order.

`DL_ListElem* DL_List::getprev(DL_ListElem *e)`

This method returns a reference to the element in the list preceding element `e`. This method is used in combination with the `getlast()` method to traverse the list in reverse order.

`DL_ListElem* DL_List::element(int i)`

This method returns a reference to the `i`-th element in the list (or `NULL` if such an element does not exist).

7 Forward dynamics classes

The forward dynamics subsystem handles everything that has to do with making geometric objects respond to forces, torques and inertia. The main component is the dyna system, which manages all the dynas and geos, and which will make sure that the motions of the dynas are calculated and communicated to their companions.

The following sections describe the different classes in the forward dynamics subsystem and their API's

7.1 Force Drawables

Since the dynamics system revolves around (the effects) of forces, torques and impulse exchanges, such 'actuators' will often have to be visualised. Visualisation is handled by the host system, but force_drawables provides a uniform interface for controlling if forces, torques and impulse exchanges are visualised.

7.1.1 Force Drawable

The `DL_force_drawable class` is used as base class for for example the controller and constraint classes and any otehr classes that have 'actuators' that might need to be visualised.

```
class DL_force_drawable {
    virtual void show_forces();
    virtual void hide_forces();

    virtual void get_fd_info(int&,int&);
    virtual void get_force_info(int, DL_actuator_type&,
        DL_dyna*&, DL_point*, DL_vector*);

       DL_force_drawable();  // constructor
       ~DL_force_drawable(); // destructor
}
```

`void DL_force_drawable::show_forces()`

This method registers this force_drawable with the force_drawer so that its 'actuators' are visualised.

`void DL_force_drawable::hide_forces()`

This method removes this force_drawable from the force_drawer so that its 'actuators' are not visualised anymore.

`void DL_force_drawable::get_fd_info(int& nrf, int& tf)`

This method is used by the force_drawer to retrieve the number of actuators that should be drawn. A difference is made between forces and reaction forces, torques and reaction torques etc. The actuators are numbered from zero to `nrf`, and the reaction actuators are numbered from `nrf` to `tf`. For each of the actuators between zero and `tf` the `get_force_method` can then be called to retrieve the actual information about the actuator.

By default this method returns two zeros indicating that no actuators are to be shown: it should be overriden in any descendant of `DL_force_drawable` that wants actuators to be visualised.

`void DL_force_drawable::get_force_info(int i, DL_actuator_type& at,`
`DL_dyna*& d, DL_point *p, DL_vector *f)`

This method is used by the force_drawer to retrieve the actual information about the `i`-th actuator of this force_drawable. It returns in `at` the type of actuator using the enumeration type: `enum DL_actuator_type {none, force, torque, impulse}` In `d` and `p` the attachement point of the actuator is specified, and in `f` the actuator itself.

This method should be implemented by each descendant that wants to visualise its actuators: by default it returns actuator type `none` regardless of `i`.

7.1.2 User Force Drawable

For existing classes that do not inherit from force_drawable, this class provides a means to still visualise its 'actuators'. It can create a `usr_force_drawable` object and set its attributes to visualise one 'actuator'.

```
class DL_usr_force_drawable : public DL_force_drawable {
    boolean reaction;
    DL_actuator_type at;
    DL_dyna *d;
    DL_point p;
    DL_vector f;

      DL_usr_force_drawable();  // constructor
      ~DL_usr_force_drawable(); // destructor
}
```

`boolean DL_usr_force_drawable::reaction`

This boolean indicates if the force is a reaction-force or not (the forcedrawer might give reaction forces a different appearance from the corresponding force).

`DL_actuator_type DL_usr_force_drawable::at`

This attribute signifies what type of actuator has to be visualised here: its value is of the following enumeration type:

`enum DL_actuator_type {none, force, torque, impulse}`

`DL_dyna DL_usr_force_drawable::*d`

`DL_point DL_usr_force_drawable::p`

These two attributes specify the attachment point of the 'actuator': it is the point with local coordinates `p` in dyna `d`.

`DL_vector DL_usr_force_drawable::f`

This attribute finally specifies the 'actuator' vector (in world coordinates).

7.2 Force Drawer

The force_drawer class is the manager of force_drawable objects. It provides the methods `register_fd` and `remove_fd` using which DL_force_drawable objects can make sure their forces are visualised or not (these are used in the `show_forces` and `hide_forces` methods of such objects). The implementation of a force_drawer should maintain a list of registered force_drawables, the 'actuators' of which should be visualised each frame. The class also provides a few methods that enable any object to visualise an 'actuator' for one frame (though this is relatively expensive: if such an object has to visualise forces for an extensive period of time, it had better use a `usr_force_drawable` object). The class is ment as a base class for the actual implementation in the host system which can do the actual visualisation using the host system's visualisation primitives.

```
class DL_force_drawer {
  virtual void draw_force(DL_dyna*,DL_point*,DL_vector*);
  virtual void draw_torque(DL_dyna*,DL_vector*);
  virtual void draw_impulse(DL_dyna*,DL_point*,DL_vector*);

  virtual void draw_reaction_force(DL_dyna*,DL_point*,DL_vector*);
  virtual void draw_reaction_torque(DL_dyna*,DL_vector*);
  virtual void draw_reaction_impulse(DL_dyna*,DL_point*,DL_vector*);

  virtual void register_fd(DL_force_drawable*);
  virtual void remove_fd(DL_force_drawable*);

       DL_force_drawer();  // constructor
       ~DL_force_drawer(); // destructor
}
```

`void DL_force_drawer::draw_force(DL_dyna *d, DL_point *p, DL_vector *f)`

This method is used to tell the force_drawer to visualise force `f` applied to point `p` of dyna `d` for the coming frame.

`void DL_force_drawer::draw_torque(DL_dyna *d, DL_vector *t)`

This method is used to tell the force_drawer to visualise torque `t` applied to dyna `d` for the coming frame.

`void DL_force_drawer::draw_impulse(DL_dyna *d, DL_point *p, DL_vector *i)`

This method is used to tell the force_drawer to visualise impulse change `i` applied to point `p` of dyna `d` for the coming frame.

`void DL_force_drawer::draw_reaction_force(DL_dyna *d, DL_point *p, DL_vector *f)`

This method is used to tell the force_drawer to visualise reaction force `f` applied to point `p` of dyna `d` for the coming frame.

`void DL_force_drawer::draw_reaction_torque(DL_dyna *d, DL_vector *t)`

This method is used to tell the force_drawer to visualise reaction torque `t` applied to dyna `d` for the coming frame.

`void DL_force_drawer::draw_reaction_impulse(DL_dyna *d, DL_point *p, DL_vector *i)`

This method is used to tell the force_drawer to visualise reaction impulse change `i` applied to point `p` of dyna `d` for the coming frame.

`void DL_force_drawer::register_fd(DL_force_drawable *fd)`

This is the method that is used in `DL_force_drawable::show_forces` to tell the force_drawer to visualise `fd`'s actuators from now on.

`void DL_force_drawer::remove_fd(DL_force_drawable *fd)`

This is the method that is used in `DL_force_drawable::hide_forces` to tell the force_drawer to stop visualising `fd`'s actuators from now on.

7.3 Dyna System

The dyna system is a one-of-a-kind object that steers the whole dynamics process. It keeps track of all the dynas and the geos in the system, and makes sure that they are activated properly. If there is a constraint manager, the dyna system object will also invoke the inverse dynamics routine, so that constraints are corrected along the way as well. Here is its API:

```
class DL_dyna_system {
    DL_dyna_system_callbacks* get_companion();

    void dynamics();

    DL_geo* register_geo(void*);
    void remove_geo(DL_geo*);

    void set_gravity(DL_vector*);
    void get_gravity(DL_vector*);
    void set_integrator(DL_m_integrator*);
    DL_m_integrator* get_integrator();
    DL_Scalar kinenergy();
    DL_Scalar potenergy();
    DL_Scalar totenergy();

    int frame_number();
    DL_Scalar time();

    void show_controller_forces();
    void hide_controller_forces();
    boolean showing_controller_forces();
    DL_dyna_system(DL_dyna_system_callbacks*,DL_m_integrator*);
    ~DL_dyna_system();
}
```

`DL_dyna_system_callbacks* DL_dyna_system::get_companion()`

As explained in Chapter 4 [Installation], page 12, the dyna system has a companion of type `DL_dyna_system_callbacks` which is used for the callbacks. This method provides easy access to that object (which was provided to the dyna system in its constructor)

`void DL_dyna_system::dynamics()`

This method is the entry point for the whole Dynamo library. Call this method once a frame to have the dyna system update the positions and orientations of all the dynamically controlled geometric objects with respect to their inertia, applied forces and torques, and the constraints imposed on them.

`DL_geo* DL_dyna_system::register_geo(void *g)`

This method is provided for establishing the links between a geo and its companion. It takes a reference to a geometric object as an argument, and will check if there is already a `DL_geo` companion for this object. If so, it will return a reference to this object, if not, it will create a companion for the object and return a reference to it.

The method is typically used when initializing constraints in case the constraint acts on a geometric object that is not dynamically controlled. The constraints require a `DL_geo` as parameter, and this method can be used to make sure that that geo is known to the library, and to obtain the reference to it.

Note that dynas add and remove themselves to and from the dyna system upon their creation and deletion, but for geos (many of which are probably not of interest for the dynamics calculations) this is inefficient, since the `get_new_geo_info` callback is called each frame for each of the registered geos.

`void DL_dyna_system::remove_geo(DL_geo*)`

if a geometric object is no longer required by the Dynamo library (and it was made known to it using the `register_geo()` method, its companion can be removed from the dyna system using this method. This removes some overhead since the dyna systems inquires about the new position and orientation of each geo at the beginning of each time step.

`void DL_dyna_system::set_gravity(DL_vector *g)`

This method sets the global gravity to `g`. This gravity acceleration will be applied to all dynas in each frame of the animation/simulation.

`void DL_dyna_system::get_gravity(DL_vector *g)`

This method assigns a copy of the current gravity vector to `g`.

`void DL_dyna_system::set_integrator(DL_m_integrator *mi)`

This method sets the motion integrator used by all the dyna to `mi`. The step size from the previous motion integrator will be retained. Use this method to switch to a different (faster or more accurate) type of integrator.

`DL_m_integrator* DL_dyna_system::get_integrator()`

This method returns a reference to the current motion integrator.

`DL_Scalar DL_dyna_system::kinenergy()`

This methods returns the total amount of kinetic energy in the system (i.e. of all dynas).

`DL_Scalar DL_dyna_system::potenergy()`

This methods returns the total amount of potential energy in the system (i.e. of all dynas).

`DL_Scalar DL_dyna_system::totenergy()`

This methods returns the total amount of kinetic and potential energy in the system (i.e. of all dynas).

`int DL_dyna_system::frame_number()`

This method returns the number of times the `dynamics` method has been called.

`DL_Scalar DL_dyna_system::time()`

This method returns the current virtual time: the sum of stepsizes of each of the calls to `dynamics`.

`void DL_dyna_system::show_controller_forces()`

This method calls `show_forces` for all controllers that are present in the system, and will make sure that it is called for any controller that is added in the future.

`void DL_dyna_system::hide_controller_forces()`

This method calls `hide_forces` for all controllers that are present in the system, and will make sure that it is called for any controller that is added in the future.

`boolean DL_dyna_system::showing_controller_forces()`

This method returns if controller forces are shown or not.

`DL_dyna_system::DL_dyna_system(DL_dyna_system_callbacks *c, DL_m_integrator *i)`

This is the constructor of the dyna system. The dyna system needs a (reference to) a `DL_dyna_system_callbacks` object, so it can communicate with its environment (see Chapter 4 [Installation], page 12). It also needs a reference to a motion integrator, so it knows in what manner to integrate the motions of the dynas.

7.4 Geo

Objects from the `DL_geo` class are companions to the geometric objects in the host application. The class only functions to administrate the two motion states (containing the position, velocity, orientation and angular velocity at time `t` (the current time) and at time `t+h` (the next time)) of such geometries, and provides an interface to the Dynamo library to access that state, and to transform points and vectors from world-coordinates to coordinates in the local coordinate system (induced by the geo's position and the orientation). Here is its API:

```
class DL_geo {
   void*      get_companion();

   void       set_position(DL_point*);
   DL_point*  get_position();
   void       set_velocity(DL_vector*);
   DL_vector* get_velocity();
   void       set_orientation(DL_matrix*);
   DL_matrix* get_orientation();
   void       set_angvelocity(DL_vector*);
   DL_vector* get_angvelocity();

   void move(DL_point*,DL_matrix*);

   void to_world(DL_point*,DL_point*);
   void to_world(DL_vector*,DL_vector*);
   void to_local(DL_point*,DL_geo*,DL_point*);
   void to_local(DL_vector*,DL_geo*,DL_vector*);
   void get_velocity(DL_point*,DL_vector*);
   void get_velocity(DL_vector*,DL_vector*);

   void assign(DL_geo*,void*);
   boolean is_dyna();

   void set_elasticity(DL_Scalar);
   DL_Scalar get_elasticity();

   DL_geo(void*);
   ~DL_geo();
}
```

`void* DL_geo::get_companion()`

This method returns a reference to the companion object (which was provided in the constructor of the geo).

`void DL_geo::set_position(DL_point *p)`

This methods assigns `p` to the geo's current position

`DL_point* DL_geo::get_position()`

This methods returns a reference to the geo's current position

`void DL_geo::set_velocity(DL_vector *v)`

This method assigns `v` to the geo's current velocity

`DL_vector* DL_geo::get_velocity()`

This method returns a reference to the geo's current velocity

`void DL_geo::set_orientation(DL_matrix *o)`

This method assigns `o` to the current orientation of the geo

`DL_matrix* DL_geo::get_orientation()`

This method returns a reference to the geo's current orientation

`void DL_geo::set_angvelocity(DL_vector *w)`

This method assigns `w` to the current angular velocity of the geo

`DL_vector* DL_geo::get_angvelocity()`

This method returns a reference to the geo's current angular velocity

`void DL_geo::move(DL_point *p, DL_matrix *o)`

This method sets the geo's position to `p` and its orientation to `o`, while updating the velocity and angular velocity according to the previous values of the position and orientation, and the current integration step size

`void DL_geo::to_world(DL_point *pl, DL_point *pw)`

This method converts the coordinates of point `pl` which are given in local coordinates, to world coordinates (using the current motion state), and returns the result in point `pw`

`void DL_geo::to_world(DL_vector *vl, DL_vector *vw)`

This method converts the coordinates of vector `vl` which are given in local coordinates, to world coordinates (using the current motion state), and returns the result in vector `vw`

`void DL_geo::to_local(DL_point *p ,DL_geo *g, DL_point *pl)`

This method converts the coordinates of point `p` (given in the current local coordinate system of geo `g`) to local coordinates, and returns these local coordinates in point `pl`.

`void DL_geo::to_local(DL_vector *v, DL_geo *g, DL_vector *vl)`

This method converts the direction vector `v` (given in the current local coordinate system of geo `g`) to local coordinates, and returns these local coordinates in vector `vl`.

`void DL_geo::get_velocity(DL_point *p, DL_vector *v)`

This method returns in vector `v` the velocity of the point with local coordinates `p` (based on the current translational velocity and angular velocity of the geo).

`void DL_geo::get_velocity(DL_vector *d, DL_vector *v)`

This method returns in vector `v` the velocity of the direction (given in local coordinates) `d` (based on the current angular velocity).

`void DL_geo::assign(DL_geo *g, void *c)`

This method assigns geo `g` to itself, and updates itself to be the companion of geometric object `c` now.

`boolean DL_geo::is_dyna()`

Returns if this geometry is dynamically controlled or not (it returns if the dynamic type of the geo is a general geo, or a dyna specialization). The dynamic type of a geo depends on how it was created: as a dyna, or as a geo by the `register_geo` method of the dyna system.

`void DL_geo set_elasticity(DL_Scalar el)`

Sets this geometries collision elasticity to el (which should have a value between zero and one). A zero value stands for a completely inelastic collision behaviour, and a value of one stands for completely elastic collision behaviour. Initially a geometry has a collision elasticity coefficient of one.

`DL_Scalar DL_geo::get_elasticity()`

This method returns the collision elasticity of the geometry

`DL_geo::DL_geo(void *c)`

The constructor of the geo, which sets the geo up to be a companion of geometric object `c`.

7.5 Dyna

A dyna is a dynamically controlled geo. Next to the motion state, a dyna also administrates the geo's mass and moments of inertia, which it needs to calculate the motion behaviour according to its inertia and the applied forces and torques.

For the correct calculation of the motion of the dyna, it is assumed that the origin of the dyna's local coordinate system is the dyna's center of mass, and that the axis of the local coordinate system are the major inertia axis. This is something to take into account when setting up the local coordinate system of the companion of the dyna.

```
class DL_dyna : public DL_geo {
  void  assign(DL_dyna*,void*);

  void      set_inertiatensor(DL_Scalar,DL_Scalar,DL_Scalar);
```

```
        DL_Scalar get_inertiamoment(int);
        void      set_mass(DL_Scalar);
        DL_Scalar get_mass();

        void      set_velodamping(DL_Scalar);
        DL_Scalar get_velodamping();

        DL_Scalar kinenergy();
        DL_Scalar potenergy();
        DL_Scalar totenergy();

        void  applyforce(DL_point*, DL_geo*, DL_vector*);
        void  applycenterforce(DL_vector*);
        void  applytorque(DL_vector*);
        void  applyimpulse(DL_point*, DL_geo*, DL_vector*);

            DL_dyna(void*);
            ~DL_dyna();
    }
```

`void dyna::assign(DL_dyna *d, void *c)`

This method assigns dyna `d` to itself, and updates itself to be the companion of geometric object `c` now.

`void DL_dyna::set_inertiatensor(DL_Scalar Ixx, DL_Scalar Iyy, DL_Scalar Izz)`

This method allows you to set the moments of inertia for the object.

`DL_Scalar DL_dyna::get_inertiamoment(int i)`

This method returns the i-th moment of inertia (i=0,1,2)

`void DL_dyna::set_mass(DL_Scalar m)`

This method allows you to specify the dyna's mass

`DL_Scalar DL_dyna::get_mass()`

This method returns the dyna's mass.

`void DL_dyna::set_velodamping(DL_Scalar vd)`

This methods sets the velocity damping of the dyna to `vd`. The velocity damping is a coefficient which is applied to the inertia of the dyna: 0 for no inertia, and 1 for full inertia. Its effect can be seen as a very rough approximation of friction since velocity damping dissipates kinetic energy.

`DL_Scalar DL_dyna::get_velodamping()`

This method returns the current velocity damping

`DL_Scalar DL_dyna::kinenergy()`

This method returns the kinetic energy of the dyna

`DL_Scalar DL_dyna::potenergy()`

This method returns the potential energy of the dyna (with respect to the central components of external forces, including the gravity)

`DL_Scalar DL_dyna::totenergy()`

This method returns the total (kinetic plus potential) energy of the dyna (with respect to external forces, including the gravity)

`void DL_dyna::applyforce(DL_point *p, DL_geo *g, DL_vector *f)`

This method is used to apply force `f` (given in world coordinates) to the point of the dyna with coordinates `p` (specified in the local coordinate system of `g`).

`void DL_dyna::applycenterforce(DL_vector *f)`

This method is used to apply a force `f` (given in world coordinates) to center of mass of the dyna.

`void DL_dyna::applytorque(DL_vector *t)`

This method is used to apply a torque `t` to the dyna.

`void DL_dyna::applyimpulse(DL_point *p, DL_geo *g, DL_vector *i)`

This method is used to a change of impulse `i` (given in world coordinates) to the point of the dyna with coordinates `p` (specified in the local coordinate system of `g`).

`DL_dyna::DL_dyna(void *c)`

This is the constructor of the dyna, which sets the dyna up to be a companion of geometric object `c`.

7.6 Motion Integrator

The dynas calculate their motions by integrating Newton's Law '$F = ma$' (and a similar law for orientation). By providing the integrator used in this calculation separately, we can choose the integrator used at runtime. Several integrators are available, each with their own advantages and disadvantages. The `DL_m_integrator` class functions as an interface specification for all these integrators, so that they can be addressed uniformly. Here is its API:

```
class DL_m_integrator {
   DL_Scalar stepsize();
   DL_Scalar halfstepsize();
   void      set_stepsize(DL_Scalar);

   DL_m_integrator();
   ~DL_m_integrator();
}
```

`DL_Scalar DL_m_integrator::stepsize()`

This method returns the current step size of the integration.

`DL_Scalar DL_m_integrator::halfstepsize()`

This method returns the half the current step size of the integration.

`void DL_m_integrator::set_stepsize(DL_Scalar h)`

Using this method one can specify the step size of the integration. Each call to the `dynamics` method of the dyna system advances the time by this step size.

There are currently four kinds of motion integrators: two variants of the first order Euler integrator, and second and fourth order Runge Kutta integrators.

7.6.1 Euler

The Euler motion integrator is a rather fast, but not so accurate motion integrator. Its error is linear in the step size of the integration. Its API is defined by the motion integrator API:

```
class DL_euler : public DL_m_integrator {
    DL_euler();
    ~DL_euler();
}
```

7.6.2 Double Euler

The double euler motion integrator takes two euler integration steps of half the step size for each of its integration steps. It is therefore about twice as expensive as the euler integrator (making it approximately as fast as the Runge Kutta 2 integrator), and still of linear order. Its API is defined by the motion integrator API:

```
class DL_double_euler : public DL_m_integrator {
    DL_double_euler();
    ~DL_double_euler();
}
```

7.6.3 Runge Kutta 2

The Runge Kutta 2 integrator is a second order integrator which (like the double euler integrator) costs about twice as much as the euler integrator. It is more accurate than the double euler integrator, and therefore preferable to it. Its API is defined by the motion integrator API:

```
class DL_rungekutta2 : public DL_m_integrator {
   DL_rungekutta2();
   ~DL_rungekutta2();
}
```

7.6.4 Runge Kutta 4

The Runge Kutta 4 motion integrator is a fourth order motion integrator. It is very accurate, but at about four times the price of the euler integrator. Its API is defined by the motion integrator API:

```
class DL_rungekutta4 : public DL_m_integrator {
   DL_rungekutta4();
   ~DL_rungekutta4();
}
```

8 Inverse dynamics classes

8.1 Constraint Manager

The constraint manager is a one-of-a-kind object that controls the inverse dynamics calculations. Constraints register themselves automatically, and the dyna system will signal the constraint manager each time constraint correction needs to take place. Its API only provides access to the attributes that hold the parameter values that control the constraint correction process. The default values for these parameters should be fine in most cases, but for the more advanced users they are available for fine-tuning. Here is the API:

```
class DL_constraint_manager {
   boolean analytical;
   int       MaxIter;
   DL_Scalar max_error;
   int       NrSkip;
   DL_Scalar error;
   int       nriter;
   int       max_collisionloops;
```

```
    int      get_nr_constraints();
    int      get_dof();

    void     solve_using_lud();
    boolean solving_using_lud();
    void     solve_using_cg();
    boolean solving_using_cg();
    void     solve_using_svd();
    boolean solving_using_svd();

    void     show_constraint_forces();
    void     hide_constraint_forces();
    boolean showing_constraint_forces();

             DL_constraint_manager();
             ~DL_constraint_manager();
}
```

`boolean DL_constraint_manager::analytical`

This boolean determines how the constraint manager will determine how changes in the reaction forces will change the constraint error: analytically or empirically (using test forces). The analytical method is faster, so by default the boolean is `true`

`int DL_constraint_manager::MaxIter`

The constraint manager adjusts the reaction forces in an iterative manner, trying to decrease the constraint error in each step. The `MaxIter` attribute specifies the maximum number of iterations per frame (note that less iteration steps may actually be taken, if the constraint error decreases below the thresh hold set by the `max_error` attribute before the maximum number of iterations is reached).

`DL_Scalar DL_constraint_manager::max_error`

This attribute governs the number of iterations used in the constraint correction, together with the `MaxIter` attribute. It provides an error thresh hold for the constraint error: as soon as the constraint error is less than this thresh hold, constraint correction stops. Note that iteration is also stopped if the maximum number of iterations as determined by the `MaxIter` attribute is reached: the `max_error` attribute therefore only provides a guaranty that the final constraint error is indeed smaller than the thresh hold, if `MaxIter` is sufficiently high.

`int DL_constraint_manager::NrSkip`

This attribute governs the recalculation of the dependencies between the reaction forces and the constraint errors: this dependency is calculated once every `NrSkip+1` frames. These dependencies are used to determine how to change the reaction forces to bring the constraint error to zero: the more accurate the dependencies are, the more faster the constraint error will converge to zero. The default value is zero, so the dependency is recalculated at the start of every frame.

`DL_Scalar DL_constraint_manager::error`

This attribute provides the most recently calculated constraint error. This information can be used in adjusting the `MaxIter` and `max_error` attributes, or to study the constraint correction process itself. It is meant as a read-only attribute: changing it will not have an effect.

`int DL_constraint_manager::nriter`

This attribute provides the actual number of iteration steps taken in the last frame, so it provides an indication of the converge speed of the constraint correction process. It is meant as a read-only attribute.

`int DL_constraint_manager::max_collisionloops`

When collision detection is used, this attribute governs the number of times the collision detection routines are called to determine if there are any secondary collisions. The default value of this attribute is one, so there is no secondary collision detection within one frame by default.

`int DL_constraint_manager::get_nr_constraints();`

This method returns the number of active constraints.

`int DL_constraint_manager::get_dof();`

This method returns the number of restricted degrees of freedom that the constraint manager is currently managing.

`void DL_constraint_manager::solve_using_lud()`

Solve for the reaction forces using LU decomposition and backward substitution. Advantages of this method are that it is fast and the processing time does not depend much on the configuration, so the frame rate will be stable as far as solving is concerned. Disadvantage is that it can only handle configuartaion which have exactly one solution for the constraint forces.

`boolean DL_constraint_manager::solving_using_lud()`

This method returns if the constraints are being solved using LU decomposition and backward substitution.

`void DL_constraint_manager::solve_using_cg()`

Solve for the reaction forces using conjugate gradient. This method can handle configurations which have one or more solutions configurations, but the required processing time depends on the configuration: it might be very fast for "easy" configuration, but rather slow for "hard" situations, thereby giving a large variance in framerate. In general it is significantly slower than using LU decomposition, but more stable.

`boolean DL_constraint_manager::solving_using_cg()`

This method returns if the constraints are being solved using conjugate gradient solving `void DL_constraint_manager::solve_using_svd()`

> Solve for the reaction forces using singular value decomposition. This method can handle configurations with zero or more solutions (although configurations with zero solutions, i.e. configurations with conflicting constraints, can give rise to very large reaction forces), but in general it is rather slow. So this is the most stable, though usually slowest solution finding method.

`boolean DL_constraint_manager::solving_using_svd()`

> This method returns if the constraints are being solved using the singular value decomposition method

`void DL_constraint_manager::show_constraint_forces()`

> This method calls `show_forces` for all constraints that are present in the system, and will make sure that it is called for any constraint that is added in the future.

`void DL_constraint_manager::hide_constraint_forces()`

> This method calls `hide_forces` for all constraints that are present in the system, and will make sure that it is called for any constraint that is added in the future.

`boolean DL_constraint_manager::showing_constraint_forces()`

> This method returns if constraint forces are shown or not.

8.2 Constraint

The `DL-constraint` class is the generic constraint class (which also doubles as the *empty* constraint which does not constrain anything). It provides the common API for all inverse dynamics constraints:

```
class DL_constraint : public DL_force_drawable {
  DL_Scalar stiffness;
  void  init();
  void  activate();
  void  deactivate();
  int   get_dim();
  void  soft();
  void  hard();
  void  autosofthard();
  void  autosofthard(int);

        DL_constraint();
        ~DL_constraint();
}
```

`DL_Scalar DL_constraint::stiffness`

This attribute specifies the stiffness of the constraint. The stiffness of a constraint is a factor which determines how much of the constraint's error the constraint manager will attempt to correct in each iteration step. The default value is one: attempt to correct all error in the constraint. Lower (but non-negative) values will "soften" the constraint with respect to other constraints (at a cost penalty, since it will take the constraint error longer to converge to a value lower than the constraint manager's `max_error` value).

`void DL_constraint::init()`

Initialize the constraint. This method is only for when the constraint is used as an empty constraint (which does not influence anything...). it should not be used to initialize any of the specializations of the constraint class: these have their own initialization methods.

`void DL_constraint::activate()`

Activates the constraint (if it had been de-activated before: constraints activate themselves upon initialization).

`void DL_constraint::deactivate()`

Deactivate the constraint.

`int DL_constraint::get_dim()`

This method returns the dimension of the constraint: the number of degrees of freedom the constraint restricts.

`void DL_constraint::soft()`

This method softens constraints that are based on positional information by also taking velocity into account: positions do not have to match if there is a velocity that will make sure they match in the next frame. This method makes sure that such velocity-terms are added for the constraint correction from now on. Normally, velocity-terms are only added (automatically) when required (velocity terms also help in stabilizing some numerical artifact caused by the time discretization). Softening constraints can be used in situations where constraints are not valid initially, but are used for self assembling purposes. In such cases, the self assembling proces will go smooth, instead of instantanious (with the accompanying very large forces which can destabelize calculations).

`void DL_constraint::hard()`

This method is the counterpart of the `soft()` method: it makes sure that the constraint is "hardened" by making sure that no velocity terms are added from now on.

`void DL_constraint::auto_softhard()`

This method can be used after invocation of soft() or hard() to revert back to the automatic addition of velocity terms.

`void DL_constraint::auto_softhard(int i)`

This method specifies that this constraint is to be made soft once every `i` frames from now on.

8.3 Point-to-point constraint

The point-to-point constraint specifies that the coordinates of two points (from different dynas, or from a dyna and a geometry) should have the same coordinates. This constraint imposes a ball-joint connection between the dyna and the geo. Here is its API:

```
class DL_ptp : public DL_constraint {
  DL_Scalar maxforce;

  void init(DL_dyna*, DL_point*, DL_geo*, DL_point*);
  void reactionforce(DL_vector*);

  DL_dyna* get_dyna();
  void     get_dyna_point(DL_point*);
  void     set_dyna_point(DL_point*);
  DL_geo*  get_geo();
  void     get_geo_point(DL_point*);
  void     set_geo_point(DL_point*);

           DL_ptp();
           ~DL_ptp();
}
```

`DL_Scalar DL_ptp::maxforce`

With this attribute, a maximum reaction force for the constraint can be specified: if this maximum reaction force is exceed the constraint will deactivate itself, so the connection 'breaks'. A `maxforce` value of zero or smaller indicates that there is no limit to the reaction force magnitude. The default value is zero.

`void DL_ptp::init(DL_dyna *d, DL_point *pd, DL_geo *g, DL_point *pg)`

This method initializes the constraint and activates it. It specifies that the point-to-point constraint should keep the point of dyna `d` with local coordinates `pd`, at the same position as the point of geo `g` (which may be a dyna, since dyna inherits from geo) with local coordinates `pg`.

`void DL_ptp::reactionforce(DL_vector *f);`

This method returns a copy of the reaction force (in world coordinates) in vector `f` (for visualization or for stress-evaluation for example).

`DL_dyna* DL_ptp::get_dyna()`

This method returns a reference to the dyna with which the constraint was initialized.

`void DL_ptp::get_dyna_point(DL_point *dp)`

This method returns in `dp` a copy of the current attachment point in the dyna.

`void DL_ptp::set_dyna_point(DL_point *dp)`

This method allows you to change the attachment point in the dyna. Use this method with caution, since too large jumps in the attachment points can lead to very strong reaction forces which may cause instabilities.

`DL_geo* DL_ptp::get_geo()`

This method returns a copy of the geo with which the constraint was initialized.

`void DL_ptp::get_geo_point(DL_point *gp)`

This method returns in `gp` a copy of the current attachment point in the geo.

`void DL_ptp::set_geo_point(DL_point *gp)`

This method allows you to change the attachment point in the geo. Use this method with caution, since too large jumps in the attachment points can lead to very strong reaction forces which may cause instabilities.

8.4 Velocity-to-velocity constraint

The velocity-to-velocity constraint specifies that the velocities of two points (from different dynas, or from a dyna and a geometry) should remain the same. This constraint has a very similar effect as the point-to-point constraint, except for some possible drift due to residual errors after each frame's constraint correction, and except for the behaviour in a situation where the constraint is not valid initially. Since this constraint is not based on positional behaviour, the `soft()` and `hard()` methods have no effect on this constraint. Here is its API:

```
class DL_vtv : public DL_constraint {
  DL_Scalar maxforce;

  void init(DL_dyna*, DL_point*, DL_geo*, DL_point*);
  void reactionforce(DL_vector*);

  DL_dyna* get_dyna();
  void     get_dyna_point(DL_point*);
  void     set_dyna_point(DL_point*);
  DL_geo*  get_geo();
  void     get_geo_point(DL_point*);
  void     set_geo_point(DL_point*);
```

```
            DL_vtv();
            ~DL_vtv();
    }
```

`DL_Scalar DL_vtv::maxforce`

With this attribute, a maximum reaction force for the constraint can be specified: if this maximum reaction force is exceed the constraint will deactivate itself, so the connection 'breaks'. A `maxforce` value of zero or smaller indicates that there is no limit to the reaction force magnitude. The default value is zero.

`void DL_vtv::init(DL_dyna *d, DL_point *pd, DL_geo *g, DL_point *pg)`

This method initializes the constraint and activates it. It specifies that the velocity-to-velocity constraint should keep the velocity of the point of dyna `d` with local coordinates `pd`, the same as the velocity of the point of geo `g` with local coordinates `pg`.

`void DL_vtv::reactionforce(DL_vector *f);`

This method returns a copy of the reaction force (in world coordinates) in vector `f` (for visualization or for stress-evaluation for example).

`DL_dyna* DL_vtv::get_dyna()`

This method returns a reference to the dyna with which the constraint was initialized.

`void DL_vtv::get_dyna_point(DL_point *dp)`

This method returns in `dp` a copy of the current attachment point in the dyna.

`void DL_vtv::set_dyna_point(DL_point *dp)`

This method allows you to change the attachment point in the dyna. Use this method with caution, since too large jumps in the attachment points can lead to very strong reaction forces which may cause instabilities.

`DL_geo* DL_vtv::get_geo()`

This method returns a copy of the geo with which the constraint was initialized.

`void DL_vtv::get_geo_point(DL_point *gp)`

This method returns in `gp` a copy of the current attachment point in the geo.

`void DL_vtv::set_geo_point(DL_point *gp)`

This method allows you to change the attachment point in the geo. Use this method with caution, since too large jumps in the attachment points can lead to very strong reaction forces which may cause instabilities.

8.5 Line-hinge constraint

The line-hinge constraint specifies that a dyna and a geo share a line, thereby imposing a line hinge (also called a pin joint) between the two geometries. Here is its API:

```
class DL_linehinge : public DL_constraint {
  DL_Scalar maxforce;
  void init(DL_dyna*, DL_point*, DL_point*, DL_geo*, DL_point*, DL_point*);

  DL_dyna* get_dyna();
  void get_dyna_point0(DL_point*);
  void set_dyna_point0(DL_point*);
  void get_dyna_point1(DL_point*);
  void set_dyna_point1(DL_point*);
  DL_geo* get_geo();
  void get_geo_point0(DL_point*);
  void set_geo_point0(DL_point*);
  void get_geo_point1(DL_point*);
  void set_geo_point1(DL_point*);

  void reactionforce0(DL_vector*);
  void reactionforce1(DL_vector*);
  void reactionforce2(DL_vector*);

  DL_Scalar reactionforce();

       DL_linehinge();
       ~DL_linehinge();
}
```

`DL_Scalar DL_linehinge::maxforce`

Using this attribute, you can specify a maximum reaction force for the constraint. If the reaction force exceeds the maximum force, the constraint deactivates itself: the connection 'breaks'. A `maxforce` value of zero or smaller indicates that there is no limit to the reaction force magnitude. The default value is zero. The reaction force is taken as the magnitude of the 5-D vector that contains the magnitudes of the force components.

```
void DL_linehinge::init(DL_dyna *d, DL_point *pd0, DL_point *pd1,
                        DL_geo *g, DL_point *pg0, DL_point *pg1)
```

This method initializes and activates the line-hinge constraint. It specifies that the point with local coordinates `pd0` in dyna `d` should coincide with the point with local coordinates `pg0` in geo `g`, and that the point with local coordinates `pd1` in dyna `d` should coincide with the point with local coordinates `pg1` in geo `g`, effectively fixing the whole line between `pd0` and `pd1` to the line between `pg0` and `pg1`. The same effect can not be reached with two point-to-point constraints, since two point-to-point constraints try to restrict six degrees of freedom, whereas a linehinge only restricts five. The two attachment point pairs should be sufficiently far apart to prevent large torques in the hinge.

`DL_dyna* DL_linehinge::get_dyna()`

This method returns a reference to the dyna with which the constraint was initialized.

`void DL_linehinge::get_dyna_point0(DL_point *dp)`

This method returns in `dp` a copy of the local coordinates of the first attachment point in the dyna.

`void DL_linehinge::set_dyna_point0(DL_point *dp)`

This method allows you to change the first attachment point in the dyna. Use this method with caution, since too large jumps in the attachment points can lead to very strong reaction forces which may cause instabilities.

`void DL_linehinge::get_dyna_point1(DL_point *dp)`

This method returns in `dp` a copy of the local coordinates of the second attachment point in the dyna.

`void DL_linehinge::set_dyna_point1(DL_point *dp)`

This method allows you to change the second attachment point in the dyna. Use this method with caution, since too large jumps in the attachment points can lead to very strong reaction forces which may cause instabilities.

`DL_geo* DL_linehinge::get_geo()`

This method returns a reference to the geo with which the constraint was initialized.

`void DL_linehinge::get_geo_point0(DL_point *gp)`

This method returns in `gp` a copy of the local coordinates of the second attachment point in the geo.

`void DL_linehinge::set_geo_point0(DL_point *gp)`

This method allows you to change the first attachment point in the geo. Use this method with caution, since too large jumps in the attachment points can lead to very strong reaction forces which may cause instabilities.

`void DL_linehinge::get_geo_point1(DL_point *gp)`

This method returns in `gp` a copy of the local coordinates of the second attachment point in the geo.

`void DL_linehinge::set_geo_point1(DL_point *gp)`

This method allows you to change the second attachment point in the geo. Use this method with caution, since too large jumps in the attachment points can lead to very strong reaction forces which may cause instabilities.

`void DL_linehinge::reactionforce0(DL_vector *f)`

This method returns in vector `f` a copy of the current reaction force in the first attachment point, perpendicular to the hinge line.

`void DL_linehinge::reactionforce1(DL_vector *f)`

This method returns in vector `f` a copy of the current reaction force in the second attachment point, perpendicular to the hinge line.

`void DL_linehinge::reactionforce2(DL_vector *f)`

This method returns in vector `f` a copy of the current reaction force in the direction of the hinge line.

`DL_Scalar DL_linehinge::reactionforce()`

This method returns the magnitude of the current 5-D restriction vector (which is used to measure the reaction force magnitude)

8.6 Orientation constraint

The orientation constraint class provides a type of constraint that can be used to make sure that the relative orientation of two geometric objects (at least one of which is a dyna) remains constant. The constraint does not impose any restrictions on the relative position. Here is its API:

```
class DL_orientation : public DL_constraint {
  DL_Scalar maxtorque;
  void  init(DL_dyna*, DL_vector*, DL_vector*, DL_geo*, DL_vector*, DL_vector*);
  void  reactiontorque(DL_vector*);

  DL_dyna* get_dyna();
  void     get_dyna_vector0(DL_vector*);
  void     set_dyna_vector0(DL_vector*);
  void     get_dyna_vector1(DL_vector*);
  void     set_dyna_vector1(DL_vector*);
  DL_geo*  get_geo();
  void     get_geo_vector1(DL_vector*);
  void     set_geo_vector1(DL_vector*);
  void     get_geo_vector2(DL_vector*);
  void     set_geo_vector2(DL_vector*);

           DL_orientation();
           ~DL_orientation();
}
```

`DL_Scalar maxtorque`

Using this attribute, you can specify a maximum torque for the constraint. If the reaction torque exceeds the maximum torque, the constraint deactivates itself: the connection ‘breaks’. A value equal to or smaller than zero means that there is no boundary for the reaction torque. The default value is zero.

`void DL_orientation::init(DL_dyna *d, DL_vector *v0, DL_vector *v1,`
`                          DL_geo *g, DL_vector *w1, DL_vector *w2);`

This method initializes the constraint. It provides the orientation with two perpendicular direction vectors `v0` and `v1` (both expressed in local coordinates of `d`) in `d`, and two perpendicular direction vectors `w1` and `w2` (both expressed in local coordinates of `g`) in `g`. The constraint expresses that `v0` remains perpendicular to both `w1` and `w2`, and that `v1` remains perpendicular to `w2`, effectively fixing the relative orientations of `d` and `g`.

`void DL_orientation::reactiontorque(DL_vector *t)`

This method returns in vector `f` a copy of the reaction torque (in world coordinates).

`DL_dyna* DL_orientation::get_dyna()`

This method returns a reference to the dyna with which the constraint was initialized.

`void DL_orientation::get_dyna_vector0(DL_vector *v0)`

This method returns in `v0` a copy of the first direction vector in the dyna (expressed in local coordinates).

`void DL_orientation::set_dyna_vector0(DL_vector *v0)`

This method allows you to change the first direction vector in the dyna. Use this method with caution, since too large jumps in the relative orientation can lead to very strong reaction torques which may cause instabilities.

`void DL_orientation::get_dyna_vector1(DL_vector *v1)`

This method returns in `v1` a copy of the second direction vector in the dyna (expressed in local coordinates).

`void DL_orientation::set_dyna_vector1(DL_vector *v1)`

This method allows you to change the second direction vector in the dyna. Use this method with caution, since too large jumps in the relative orientation can lead to very strong reaction torques which may cause instabilities.

`DL_geo* DL_orientation::get_geo()`

This method returns a reference to the geo with which the constraint was initialized.

`void DL_orientation::get_geo_vector1(DL_vector *w1)`

This method returns in `w1` a copy of the first direction vector in the geo (expressed in local coordinates).

`void DL_orientation::set_geo_vector1(DL_vector *w1)`

This method allows you to change the first direction vector in the geo. Use this method with caution, since too large jumps in the relative orientation can lead to very strong reaction torques which may cause instabilities.

`void DL_orientation::get_geo_vector2(DL_vector *w2)`

This method returns in `w2` a copy of the second direction vector in the geo (expressed in local coordinates).

`void DL_orientation::set_geo_vector2(DL_vector *w2)`

This method allows you to change the second direction vector in the geo. Use this method with caution, since too large jumps in the relative orientation can lead to very strong reaction torques which may cause instabilities.

8.7 Connector constraint

The connector constraint class provides a type of constraint that can be used to completely restrict the relative motion of a dyna and a geo. It uses a point-to-point and an orientation constraint to accomplish its goals. In general it is more efficient to model those two objects as one, except that now the `maxforce` and `maxtorque` attributes can be used to assign a given strength to the connection, allowing it to break under stress. Here is its API:

```
class DL_connector : public DL_constraint {
  DL_ptp *myptp;
  DL_orientation *myorient;
  void init(DL_dyna*, DL_point*, DL_point*, DL_point*,
          DL_geo* , DL_point*, DL_point*, DL_point*);

      DL_connector();
      ~DL_connector();
}
```

`DL_ptp* DL_connector::myptp`

This attribute contains a reference to the point-to-point constraint of the connector constraint (the point-to-point constraint is created and initialized by the `init` method).

`DL_orientation* DL_connector::myorient`

This attribute contains a reference to the orientation constraint of the connector constraint (the orientation constraint is created and initialized by the `init` method).

```
void DL_connector::init(DL_dyna *d, DL_point *pd0, DL_point *pd1, DL_point *pd2,
                        DL_geo *g, DL_point* pg0, DL_point *pg1, DL_point *pg2)
```

This method initializes the connector constraint. The constraint is created such that the points with local coordinates `pd0`, `pd1` and `pd2` in `d` will always coincide with the points in `g` with local coordinates `pg0`, `pg1`, and `pg2`.

8.8 Cylinder constraint

The cylinder constraint class provides a constraint that can be uses to connect a dyna and a geo such that their only relative motion is uncoupled translation and rotation along a given common axis. The connection is similar to a line-hinge, but the two geometries can now also translate along the hinge line (with respect to each other). Here is the API of the constraint:

```
class DL_cyl : public DL_constraint {
  DL_Scalar maxforce;
  void init(DL_dyna*, DL_point*, DL_point*, DL_geo*, DL_point*, DL_point*);

  DL_dyna* get_dyna();
  void get_dyna_point0(DL_point*);
  void set_dyna_point0(DL_point*);
  void get_dyna_point1(DL_point*);
  void set_dyna_point1(DL_point*);
  DL_geo* get_geo();
  void get_geo_point0(DL_point*);
  void set_geo_point0(DL_point*);
  void get_geo_point1(DL_point*);
  void set_geo_point1(DL_point*);

  void reactionforce0(DL_vector*);
  void reactionforce1(DL_vector*);

  DL_Scalar reactionforce();

        DL_cyl();
        ~DL_cyl();
}
```

`DL_Scalar DL_cyl::maxforce`

Using this attribute, you can specify a maximum reaction force for the constraint. If the reaction force exceeds the maximum force, the constraint deactivates itself: the connection 'breaks'. A value equal to or smaller than zero means that there is no boundary for the reaction force. The default value is zero. The reaction force is taken as the magnitude of the 4-D vector that contains the magnitudes of the force components.

`void DL_cyl::init(DL_dyna *d, DL_point *pd0, DL_point *pd1,`
`DL_geo *g, DL_point *pg0, DL_point *pg1)`

This method initializes and activates the cylinder constraint. It specifies that the line in `d` through the points with local coordinates `pd0` and `pd1` should coincide with the line in `g` through the points with local coordinates `pg0` and `pg1` (allowing for relative rotations and translations along this line). The two attachment point pairs should be sufficiently far apart to prevent large torques in the hinge.

`DL_dyna* DL_cyl::get_dyna()`

This method returns a reference to the dyna with which the constraint was initialized.

`void DL_cyl::get_dyna_point0(DL_point *dp)`

This method returns in `dp` a copy of the local coordinates of the first attachment point in the dyna.

`void DL_cyl::set_dyna_point0(DL_point *dp)`

This method allows you to change the first attachment point in the dyna. Use this method with caution, since too large jumps in the attachment points can lead to very strong reaction forces which may cause instabilities.

`void DL_cyl::get_dyna_point1(DL_point *dp)`

This method returns in `dp` a copy of the local coordinates of the second attachment point in the dyna.

`void DL_cyl::set_dyna_point1(DL_point *dp)`

This method allows you to change the second attachment point in the dyna. Use this method with caution, since too large jumps in the attachment points can lead to very strong reaction forces which may cause instabilities.

`DL_geo* DL_cyl::get_geo()`

This method returns a reference to the geo with which the constraint was initialized.

`void DL_cyl::get_geo_point0(DL_point *gp)`

This method returns in `gp` a copy of the local coordinates of the first attachment point in the geo.

`void DL_cyl::set_geo_point0(DL_point *gp)`

This method allows you to change the first attachment point in the geo. Use this method with caution, since too large jumps in the attachment points can lead to very strong reaction forces which may cause instabilities.

`void DL_cyl::get_geo_point1(DL_point *gp)`

This method returns in `gp` a copy of the local coordinates of the second attachment point in the geo.

`void DL_cyl::set_geo_point1(DL_point *gp)`

This method allows you to change the second attachment point in the geo. Use this method with caution, since too large jumps in the attachment points can lead to very strong reaction forces which may cause instabilities.

`void DL_cyl::reactionforce0(DL_vector *f)`

This method returns in vector `f` a copy of the current reaction force in the first attachment point, perpendicular to the hinge line.

`void DL_cyl::reactionforce1(DL_vector *f)`

This method returns in vector `f` a copy of the current reaction force in the second attachment point, perpendicular to the hinge line.

`DL_Scalar DL_cyl::reactionforce()`

This method returns the magnitude of the current 4-D restriction vector (which is used to measure the reaction force magnitude used for comparing to the `maxforce` attribute)

8.9 Plane constraint

The plane constraint class provides a constraint type that can be used to connect a dyna and a geo in such a way that they have a plane in common. This means that their relative motion is restricted to translations within the plane, and rotations around the plane normal. Here is its API:

```
class DL_plc : public DL_constraint {
  DL_Scalar maxforce;
  DL_Scalar maxtorque;
  void init(DL_dyna*, DL_point*, DL_vector*, DL_geo*, DL_point*, DL_vector*);

  DL_dyna* get_dyna();
  DL_geo* get_geo();

  void reactionforce(DL_vector*);
  void reactiontorque(DL_vector*);

       DL_plc();
       ~DL_plc();
}
```

`DL_Scalar DL_plc::maxforce`

With this attribute, a maximum reaction force for the constraint can be specified: if this maximum reaction force is exceed the constraint will deactivate itself, so the connection ‘breaks’. A `maxforce` value of zero or smaller indicates that there is no limit to the reaction force magnitude. The default value is zero.

`DL_Scalar DL_plc::maxtorque`

With this attribute, a maximum reaction torque for the constraint can be specified: if this maximum reaction torque is exceed the constraint will deactivate itself, so the connection ‘breaks’. A `maxtorque` value of zero or smaller indicates that there is no limit to the reaction torque. The default value is zero.

```
void DL_plc::init(DL_dyna *d, DL_point *pd, DL_vector *nd,
                  DL_geo *g, DL_point *pg, DL_vector *ng);
```

This method initializes the plane constraint, specifying that the plane in dyna `d` through the point with local coordinates `pd` and with normal `pn` (also given in local coordinates), should coincide with the plane in geo `g` through the point with local coordinates `pg` and with normal `ng` (also given in local coordinates). This leaves two translation degrees of freedom (only relative translation perpendicular to the plane is prohibited), and one rotational degree of freedom (rotations along the plane's normal are allowed).

`DL_dyna* DL_plc::get_dyna()`

This method returns a reference to the dyna with which the constraint was initialized.

`DL_geo* DL_plc::get_geo()`

This method returns a reference to the geo with which the constraint was initialized.

`void DL_plc::reactionforce(DL_vector *f)`

This method returns in `f`, a copy of the reaction force that is applied to `pd` (see the `init` method) and (inverted) to `pg`.

`void DL_plc::reactiontorque(DL_vector *t)`

This method returns in `t` a copy of the reaction torque that is applied to the dyna and (inverted) to the geo.

8.10 Prism constraint

The prism constraint class provides a connection constraint where a dyna and a geo have as only relative freedom of motion translation along a pre-defined axis.

```
class DL_pris : public DL_constraint {
  DL_orientation *myorient;
  DL_Scalar maxforce;

  void init(DL_dyna*,DL_point*,DL_vector*,DL_geo*,DL_point*,DL_vector*);

  DL_dyna* get_dyna();
  DL_geo* get_geo();

  void reactionforce(DL_vector*);

      DL_pris();
      ~DL_pris();
}
```

`DL_orientation* DL_pris::myorient`

This method returns a reference to the orientation constraint that this constraint uses to maintain the proper orientation between the dyna and the geo.

`DL_Scalar DL_pris::maxforce`

With this attribute, a maximum reaction force for the constraint can be specified: if this maximum reaction force is exceed the constraint will deactivate itself, so the connection 'breaks'. A `maxforce` value of zero or smaller indicates that there is no limit to the reaction force magnitude. The default value is zero.

`void DL_pris::init(DL_dyna *d, DL_point *pd, DL_vector *ld, DL_vector *rd, DL_geo *g, DL_point *pg, DL_vector *lg, DL_vector *rg)`

This method initializes the constraint, specifying that the only relative motion between `d` and `g` is translation along the axis which in the local coordinate system of `d` is defined by point `pd` and direction vector `ld`, and which in the local coordinate system of `g` is defined by point `pg` and direction `lg`. The relative orientation is such that vectors `rd` and `rg` also coincide (so l, r and their cross product form a basis in both the geo and the dyna and these two basis are kept at the same orientation). Vectors `l` and `r` should be perpendicular.

`DL_dyna* DL_pris::get_dyna()`

This method returns a reference to the dyna with which the constraint was initialized.

`DL_geo* DL_pris::get_geo()`

This method returns a reference to the geo with which the constraint was initialized.

`void DL_pris::reactionforce(DL_vector *f)`

This method returns in `f` a copy of the reaction force which is being applied to point `pd` in `d` (see the`init` method), and (inverted) to point `pg` in `g`.

8.11 Point-to-curve constraint

The point-to-curve constraint class provides a connection constraint that specifies that a given point of a given geo should lie on a given curve (for the curve class, see Section 9.1 [Curves], page 63). This means that the constraint will calculate and apply a reaction force that will pull the point towards the curve, but which will allow unrestricted motion along the curve. The constraint will deactivate itself if the connection point reaches outside the domain of the curve. The curve should be continuous and differentiable in all the points of its domain. Here is the API of the class:

```
class DL_ptc : public DL_constraint {
  DL_Scalar maxforce;

  void init(DL_geo*, DL_point*, DL_curve*);
  void reactionforce(DL_vector*);

  DL_geo* get_geo();
  void get_point(DL_point*);
  void set_point(DL_point*);

  DL_curve* get_curve();
```

```
        DL_Scalar get_s();

            DL_ptc();
            ~DL_ptc();
        }
```

`DL_Scalar DL_ptc::maxforce`

With this attribute, a maximum reaction force for the constraint can be specified: if this maximum reaction force is exceed the constraint will deactivate itself, so the connection 'breaks'. A maximum force value of zero or smaller indicates that there is no maximum. The default value is zero.

`void DL_ptc::init(DL_geo *g, DL_point *p, DL_curve *c)`

This method initializes the constraint, specifying that point `p` (given in local coordinates) of geo `g` should lie on curve `c`. For the constraint to be able to apply its reaction force, either the curve should be part of a dyna, or `g` should be a dyna.

`void DL_ptc::reactionforce(DL_vector *f)`

This method returns in `f` a copy of the reaction force used to keep the constraint valid.

`DL_geo* DL_ptc::get_geo()`

This method returns a copy of the geo with which the constraint was initialized.

`void DL_ptc::get_point(DL_point *p)`

This method returns in `p` a copy of the current attachment point in the geo.

`void DL_ptc::set_point(DL_point *p)`

This method allows you to change the attachment point in the geo. Use this method with caution, since too large jumps in the attachment points can lead to very strong reaction forces which may cause instabilities.

`DL_curve* DL_ptc::get_curve()`

This method returns a reference to the curve the constraint was initialized with.

`DL_Scalar DL_ptc::get_s()`

This method returns the current curve-parameter corresponding to the attachment point.

8.12 Point-to-surface constraint

The point-to-surface constraint class provides a connection constraint that specifies that a given point of a given geo should lie on a given surface (for the surface class, see Section 9.2 [Surfaces], page 69). This means that the constraint will calculate and apply a reaction force that will pull the point towards the surface, but which will allow unrestricted motion within the surface. The constraint will deactivate itself if the connection point reaches outside the domain of the surface. The surface should be C-1 in all the points of its domain. Here is the API of the class:

```
class DL_pts : public DL_constraint {
  DL_Scalar maxforce;

  void init(DL_geo*, DL_point*, DL_surface*);
  void reactionforce(DL_vector*);

  DL_geo* get_geo();
  void get_point(DL_point*);
  void set_point(DL_point*);

  DL_surface* get_surface();
  DL_Scalar get_s();
  DL_Scalar get_t();

        DL_pts();
        ~DL_pts();
}
```

`DL_Scalar DL_pts::maxforce`

With this attribute, a maximum reaction force for the constraint can be specified: if this maximum reaction force is exceed the constraint will deactivate itself, so the connection 'breaks'. A `maxforce` value of zero or smaller indicates that there is no limit to the reaction force magnitude. The default value is zero.

`void DL_pts::init(DL_geo *g, DL_point *p, DL_surface *s)`

This method initializes the constraint, specifying that geo `g` should remain connected to surface `s` in the point with local coordinates `p`. For the constraint to be able to apply its reaction force, either the surface should be part of a dyna, or `g` should be a dyna.

`void DL_pts::reactionforce(DL_vector *f)`

This method returns in `f` a copy of the reaction force used to keep the constraint valid.

`DL_geo* DL_pts::get_geo()`

This method returns a copy of the geo with which the constraint was initialized.

`void DL_pts::get_point(DL_point *p)`

This method returns in `p` a copy of the current attachment point in the geo.

`void DL_pts::set_point(DL_point *p)`

This method allows you to change the attachment point in the geo. Use this method with caution, since too large jumps in the attachment points can lead to very strong reaction forces which may cause instabilities.

`DL_surface* DL_pts::get_surface()`

This method returns a reference to the surface the constraint was initialized with.

`DL_Scalar DL_pts::get_s()`
`DL_Scalar DL_pts::get_t()`

These two methods return the surface parameters corresponding to the current attachment point.

8.13 The bar (rope) constraint

The bar constraint class provides objects that can be used to model ropes or bars between two attachment points. In essence, it acts as a length constraint between the two attachment points.

```
class DL_bar : public DL_constraint {
  DL_Scalar maxforce;
  boolean rope;

  void init(DL_dyna*, DL_point*, DL_geo*, DL_point*, DL_Scalar);
  void reactionforce(DL_vector*);

  DL_dyna* get_dyna();
  void get_dyna_point(DL_point*);
  void set_dyna_point(DL_point*);
  DL_geo* get_geo();
  void get_geo_point(DL_point*);
  void set_geo_point(DL_point*);

  void set_length(DL_Scalar);
  DL_Scalar get_length();

            DL_bar();
            ~DL_bar();
}
```

`DL_Scalar DL_bar::maxforce`

With this attribute, a maximum reaction force for the constraint can be specified: if this maximum reaction force is exceed the constraint will deactivate itself, so the connection 'breaks'. A `maxforce` value of zero or smaller indicates that there is no limit to the reaction force magnitude. The default value is zero.

`boolean DL_bar::rope`

This boolean indicates if the constraint should model a rope or a bar. A bar can excert both pulling and pushing forces, and can hence always make sure that the bar retains its length, while a rope can only exert pulling forces: meaning that it cannot prevent the distance between the two attachment points to become less than the length of the rope. The default value for this attribute is `false`

`void DL_bar::init(DL_dyna *d, DL_point *pd, DL_geo *g, DL_point *pg, DL_Scalar l)`

This method initializes the constraint and activates it. It specifies that a bar (or rope) with length `l` is attached on the one side to the point of dyna `d` with local coordinates `pd`, and on the other side to the point of geo `g` with local coordinates `pg`. The bar/rope in itself is weightless, and thus acts as a (partial) length constraint.

`void DL_bar::reactionforce(DL_vector *f)`

This method returns a copy of the reaction force (in world coordinates) in vector `f` (for visualization or for stress-evaluation for example).

`DL_dyna* DL_bar::get_dyna()`

This method returns a reference to the dyna with which the constraint was initialized.

`void DL_bar::get_dyna_point(DL_point *dp)`

This method returns in `dp` a copy of the current attachment point in the dyna.

`void DL_bar::set_dyna_point(DL_point *dp)`

This method allows you to change the attachment point in the dyna. Use this method with caution, since too large jumps in the attachment points can lead to very strong reaction forces which may cause instabilities.

`DL_geo* DL_bar::get_geo()`

This method returns a copy of the geo with which the constraint was initialized.

`void DL_bar::get_geo_point(DL_point *gp)`

This method returns in `gp` a copy of the current attachment point in the geo.

`void DL_bar::set_geo_point(DL_point *gp)`

This method allows you to change the attachment point in the geo. Use this method with caution, since too large jumps in the attachment points can lead to very strong reaction forces which may cause instabilities.

`void DL_bar::set_length(DL_Scalar l)`

This method sets the length of the bar/rope to `l`. Use this method with caution, since too large changes in the length can lead to very strong reaction forces which may cause instabilities.

`DL_Scalar DL_bar::get_length()`

This method returns the length of the rope.

8.14 The multibar (rope) constraint

The multibar constraint class provides objects that can be used to model ropes or bars between two or more attachment points. In essence, it acts as a constraint trying to preserve the total length of the sum of the distances between the attachment points.

```
class DL_multi_bar : public DL_constraint {
  DL_Scalar maxforce;
  boolean rope;

  void init(int);
  void addpair(DL_geo*,DL_point*);

  void reactionforce(int, DL_vector*);

  DL_geo* get_geo(int);
  void get_point(int, DL_point*);
  void set_point(int, DL_point*);

  void set_length(DL_Scalar);
  DL_Scalar rest_length();
  DL_Scalar actual_length();

          DL_multi_bar();
    ~DL_multi_bar();
}
```

`DL_Scalar DL_multibar::maxforce`

With this attribute, a maximum reaction force for the constraint can be specified: if this maximum reaction force is exceed the constraint will deactivate itself, so the connection 'breaks'. A `maxforce` value of zero or smaller indicates that there is no limit to the reaction force magnitude. The default value is zero.

`boolean DL_multibar::rope`

This boolean indicates if the constraint should model a multirope or a multibar. A multibar can excert both pulling and pushing forces, and can hence always make sure that it retains its length, while a multirope can only exert pulling forces: meaning that it cannot prevent the sum of distances between the attachment points to become less than the length of the multirope. The default value for this attribute is `false`

`void DL_multibar::init(int nr)`

This method initializes the multibar constraint, and specifies that there will be at most `nr` attachment points. The multibar maintains a list of attachment points. The multibar/rope is strung between the subsequent attachment points in the list. Before actual use, attachment points can be added using the `addpair` method.

`void DL_multibar::addpair(DL_geo*,DL_point*)`

This method adds an attachment point to the end of the list. The rest length of the rope is set to the current length.

`void DL_multibar::reactionforce(int i, DL_vector *f)`

This method assigns to f, the reaction force in the segment between attachment points `i` and `i+1`.

`DL_geo* DL_multibar::get_geo(int i)`

This method returns a reference to the geometry of the `i`-th attachment point.

`void DL_multibar::get_point(int i, DL_point *dp)`

This method returns in `dp` a copy of the coordinates of the `i`-th attachment point (in local coordinates of `get_geo(i)`).

`void DL_multibar::set_point(int i, DL_point *dp)`

This method sets the coordinates of the `i`-th attachment point (in local coordinates of `get_geo(i)`) to `dp`.

`void DL_multibar::set_length(DL_Scalar)`

This method sets the rest length of the multibar/rope.

`DL_Scalar DL_multibar::rest_length()`

This method returns the rest length of the multibar/rope.

`DL_Scalar DL_multibar::actual_length()`

This method returns the actual length of the multibar. In case of the multibar being a rope, this length can actually be less than the rest length.

8.15 Wheel constraint

The wheel constraint class provides a constraint type that can be used to model the contact of a wheel with the surface it is rolling over. The constraint will calculate a reaction force such that the wheel rolls without slipping (either sideways or in the rolling direction).

```
class DL_wheel : public DL_constraint {
  DL_Scalar maxforce;

  void init(DL_dyna*, DL_vector*, DL_point*, DL_Scalar,
          DL_surface*, DL_Scalar, DL_Scalar);
  void reactionforce(DL_vector*);
  DL_dyna* get_dyna();
  void get_contact_point(DL_point*);

      DL_wheel();
```

```
        ~DL_wheel();
    }
```

`DL_Scalar DL_wheel::maxforce`

With this attribute, a maximum reaction force for the constraint can be specified: if this maximum reaction force is exceed the constraint will deactivate itself, so the connection 'breaks'. A `maxforce` value of zero or smaller indicates that there is no limit to the reaction force magnitude. The default value is zero.

```
void DL_wheel::init(DL_dyna *w, DL_vector *n, DL_point *c, DL_Scalar r,
                    DL_surface *sf, DL_Scalar s, DL_Scalar t)
```

This method initializes the constraint. The first four parameters specify the wheel: `w` is the dyna, point `c` gives (in local coordinates) the center of the wheel (normally this is point `(0,0,0)`), vector `n` gives the normal of the wheel-plane, and `r` is the radius of the wheel. The last three parameters specify the *road* the wheel is rolling on: surface `sf` describes this *road*, and the pair `(s,t)` gives the initial position of the contact point of the wheel in the surface.

`void DL_wheel::reactionforce(DL_vector *f)`

This method returns in `f` a copy of the current reaction force used to keep the constraint valid.

`DL_dyna* DL_wheel::get_dyna();`

This method retruns a reference to the dyna that is the wheel.

`void DL_wheel::get_contact_point(DL_point *p);`

This method returns in p the coordinates of the contact point between the wheel and the surface (in local coordinates of the wheel);.

8.16 Collision constraint

The collision constraint class provides a constraint type that can be used for collision handling of point-point collisions. A collision detector can create collision constraints when it has detected that a collision is taking place, and then the collision constraint will calculate and apply the appropriate collision forces. A collision constraint always deletes itself at the end of the frame, when it has been handled. Here is its (current) API:

```
    class DL_collision : public DL_constraint {
        DL_collision(DL_geo*, DL_point*,
                     DL_geo*, DL_point*,
                     DL_vector*, int mode=1);
        ~DL_collision();
    }
```

```
DL_collision::DL_collision(DL_geo *g0, DL_point *p0,
                           DL_geo *g1, DL_point *p1, DL_vector *n, int mode=0)
```

The constructor of the collision constraint class is also used to initialize the constraint. It specifies that a collision between geo `g0` and `g1` (at least one of which should be a dyna) will be handled. The contact point in the first geo is given by local coordinates `p0`. The contact point in the second geo is given by local coordinates `p1`.

Vector `n` is the collision normal which is used to determine the direction of the collision reaction force. For a point-plane collision this is the normal of the plane for example. It does not matter to which side of the plane the normal points.

The `mode` parameter is optional. It allows control over the way the collision is handled: a value of 1 signifies handling via a velocity constraint only. A value larger than one adds positional constraints. A value smaller than 1 lets the collision constraint decide whether to imploy the extra positional constraint or not (this is the default behaviour).

The elasticity of the collision is determined from the elasticity values given for the colliding geometries.

9 Miscellaneous classes

In this section the remaining classes in the Dynamo library are presented. First, the curve and the surface classes are discussed which can be used to model the curves and surfaces required by the point-to-curve and the point-to-surface constraints. Then the generic controller class and the one controller (for modeling dampers and (damped) springs).

9.1 Curves

The curve classes are used to model explicit curves: functions from one curve parameter to 3-D points. These are the curves that are use by the point-to-curve constraint (see Section 8.11 [ptc], page 55). With each curve, a geo is associated: the curve is expressed in the local coordinate system of that geo (or in world coordinates if the geo-reference happens to be the `NULL` pointer). In case the geo is a dyna, the point-to-curve constraint's reaction forces are also applied to this geo. The `DL_curve` class is the generic explicit curve class which presents the general curve-API, while the other classes presented here are actual implementations of specific types of curves.

9.1.1 Curve

The generic explicit curve class provides the generic API for curves. It should be considered an abstract class, and only the curve specializations provide an initialization method.

```
class DL_curve {
  void assign(DL_curve*);

  boolean pos(DL_Scalar,DL_point*);
  boolean deriv(DL_Scalar,DL_vector*);
  boolean indomain(DL_Scalar);
  DL_Scalar closeto(DL_point*);

  DL_geo* get_geo();
  DL_Scalar get_minparam();
  DL_Scalar get_maxparam();

        DL_curve();
        ~DL_curve();
}
```

`void DL_curve::assign(DL_curve *c)`

This method assigns curve `c` to this curve

`boolean DL_curve::pos(DL_Scalar s, DL_point *p)`

This method assigns the position (in the local coordinate system of the geo associated with the curve) for curve parameter `s` to `p`. It returns if `s` is within the curve domain.

`boolean DL_curve::deriv(DL_Scalar s, DL_vector *d)`

This method assigns the derivative (in the local coordinate system of the geo associated with the curve) for curve parameter `s` to `d`. It returns if `s` is within the curve domain.

`boolean DL_curve::indomain(DL_Scalar s)`

This method returns whether s is within the curve domain or not

`DL_Scalar DL_curve::closeto(DL_point *p)`

This method returns a curve-parameter which is such that the distance between that point on the curve and point `p` (given in world coordinates) is minimal.

`DL_geo* DL_curve::get_geo()`

This method returns the geo associated with this curve.

`DL_Scalar DL_curve::get_minparam()`

`DL_Scalar DL_curve::get_maxparam()`

These two methods return the two curve parameters that are the boundaries for the curve domain. Some curve specializations may also provide methods to change these boundaries, but not all of them will want to allow this.

9.1.2 Line

The line curve class provides a curve specification for straight lines (without restrictions on the domain). Here is its API:

```
class DL_line : public DL_curve {
  void assign(DL_line*);
  void init(DL_geo*, DL_point*, DL_vector*);
  void set_minparam(DL_Scalar);
  void set_maxparam(DL_Scalar);

      DL_line();
      ~DL_line();
}
```

`void DL_line::assign(DL_line *l)`

This method assigns line `l` to this line.

`void DL_line::init(DL_geo *g, DL_point *p, DL_vector *l)`

This method initializes the curve. The geo associated with it, is provided in reference `g`, and the line is the line through point `p`, and with direction vector `l`.

`void DL_line::set_minparam(DL_Scalar m)`
`void DL_line::set_maxparam(DL_Scalar m)`

These two methods allow setting the domain boundaries for the line. These are only used by the *line segment* specialization, but may be for example be useful when adding a method that provides a topology for the curve: only the piece of the line specified by the domain parameters can then be visualized.

9.1.3 Line segment

The line segment curve class is a specialization of the line curve. Its initialization method takes different parameters, and the curve is only defined for a limited domain.

```
class DL_linesegment : public DL_line {
  void assign(DL_linesegment*);
  void init(DL_geo*, DL_point*, DL_point*);

      DL_linesegment();
      ~DL_linesegment();
}
```

`void DL_linesegment::assign(DL_linesegment *l)`

This method assigns line segment `l` to this line segment.

`void DL_linesegment::init(DL_geo *g, DL_point *p0, DL_point *p1)`

This method initializes the line segment, and specifies its two end points as `p0` and `p1`.

9.1.4 Circle

The circle curve class provides a curve definition for a circle. Its domain delimiters are specified at zero and two pi (for ease of for example visualization), but any curve-parameter is considered valid and with the curve domain.

```
class DL_circle : public DL_curve {
  void assign(DL_circle*);
  void init(DL_geo*, DL_point*, DL_Scalar, DL_vector*);

      DL_circle();
      ~DL_circle();
}
```

`void DL_circle::assign(DL_circle *c)`

This method assigns circle `c` to this circle.

`void DL_circle::init(DL_geo *g, DL_point *p, DL_Scalar r, DL_vector *n)`

This method initializes the circle. The circle has point `p` as its center. It has radius `r`, and the plane it lies in had normal `n`.

9.1.5 B-spline segment

The B-spline segment curve class provides cubic B-splines with exactly four control points. It is designed for use by the B-spline class, so it also has parameters `t` and `dt` which model the start and the duration of the interval within the larger B-spline's parameter space.

```
class DL_bsplinesegment : public DL_curve {
  void assign(DL_bsplinesegment*)
  void init(DL_geo*,DL_point*,DL_point*,DL_point*,DL_point*);
  void update_control_point(int,DL_point*);
  boolean deriv2(DL_Scalar,DL_vector*);
  void set_interval(DL_Scalar,DL_Scalar);
  DL_Scalar t();
  DL_Scalar dt();

      DL_bsplinesegment();
      ~DL_bsplinesegment();
}
```

`void DL_bsplinesegment::assign(DL_bsplinesegment *b)`

This method assigns B-spline segment `b` to this B-spline segment.

`void DL_bsplinesegment::init(DL_geo *g, DL_point *p0, DL_point *p1, DL_point *p2, DL_point *p3)`

This method initializes the B-spline as a B-spline with geo `g` associated to it, and control points `p0`, `p1`, `p2` and `p3`.

`void DL_bsplinesegment::update_control_point(int i, DL_point *p)`

This method assigns new coordinates `p` to the `i`-th control point.

`void DL_bsplinesegment::deriv2(DL_Scalar s, DL_vector *v)`

This method returns the second derivative at curve parameter `s` in vector `v`.

`void DL_bsplinesegment::set_interval(DL_Scalar t, DL_Scalar dt)`

This method sets the parameter space interval for which this B-spline segment is valid within any larger B-spline. It also sets the minimal domain parameter to `t` and the maximum domain parameter to `t`+`dt`.

`DL_Scalar DL_bsplinesegment::t()`

This method returns the minimum curve parameter for this spline segment.

`DL_Scalar DL_bsplinesegment::dt()`

This method returns the size of the curve parameter interval for this spline segment.

9.1.6 B-spline

The B-spline curve class allows you to specify B-splines with any number (greater than one) control points. The spline consists of a series of B-spline segments. The spline can be cyclic meaning that there is also a spline segment between the last and the first control points specified.

```
class DL_bspline : public DL_curve {
  void assign(DL_bspline*)
  void init(DL_geo*,DL_List*,boolean);
  void update_control_point(int,DL_point*);
  boolean deriv2(DL_Scalar,DL_vector*);

      DL_bspline();
      ~DL_bspline();
}
```

`void DL_bspline::assign(DL_bspline *b)`

This method assigns B-spline `b` to this B-spline.

`void DL_bspline::init(DL_geo *g, DL_List *p, boolean c)`

This method initializes the B-spline with control points as listed in list `p`, and with boolean `c` indicating if the B-spline has to be cyclic or not.

`void DL_bsplinesegment::update_control_point(int i, DL_point *p)`

This method assigns new coordinates `p` to the `i`-th control point.

`void DL_bspline::deriv2(DL_Scalar s, DL_vector *v)`

This method returns the second derivative at curve parameter `s` in vector `v`.

9.1.7 C-spline segment

The C-spline segment curve class provides cubic C-splines with exactly four control points. It is designed for use by the C-spline class, so it also has parameters `t` and `dt` which model the start and the duration of the interval within the larger C-spline's parameter space.

```
class DL_csplinesegment : public DL_curve {
  void assign(DL_csplinesegment*)
  void init(DL_geo*,DL_point*,DL_point*,DL_point*,DL_point*);
  void update_control_point(int,DL_point*);
  boolean deriv2(DL_Scalar,DL_vector*);
  void set_interval(DL_Scalar,DL_Scalar);
  DL_Scalar t();
  DL_Scalar dt();

        DL_csplinesegment();
        ~DL_csplinesegment();
}
```

`void DL_csplinesegment::assign(DL_csplinesegment *b)`

This method assigns C-spline segment `b` to this C-spline segment.

`void DL_csplinesegment::init(DL_geo *g, DL_point *p0, DL_point *p1, DL_point *p2, DL_point *p3)`

This method initializes the C-spline as a C-spline with geo `g` associated to it, and control points `p0`, `p1`, `p2` and `p3`.

`void DL_csplinesegment::update_control_point(int i, DL_point *p)`

This method assigns new coordinates `p` to the `i`-th control point.

`void DL_csplinesegment::deriv2(DL_Scalar s, DL_vector *v)`

This method returns the second derivative at curve parameter `s` in vector `v`.

`void DL_csplinesegment::set_interval(DL_Scalar t, DL_Scalar dt)`

This method sets the parameter space interval for which this C-spline segment is valid within any larger C-spline. It also sets the minimal domain parameter to `t` and the maximum domain parameter to `t+dt`.

`DL_Scalar DL_csplinesegment::t()`

This method returns the minimum curve parameter for this spline segment.

`DL_Scalar DL_csplinesegment::dt()`

This method returns the size of the curve parameter interval for this spline segment.

9.1.8 C-spline

The C-spline curve class allows you to specify C-splines with any number (greater than one) control points. The spline consists of a series of C-spline segments. The spline can be cyclic meaning that there is also a spline segment between the last and the first control points specified.

```
class DL_cspline : public DL_curve {
  void assign(DL_cspline*)
  void init(DL_geo*,DL_List*,boolean);
  boolean deriv2(DL_Scalar,DL_vector*);

      DL_cspline();
      ~DL_cspline();
}
```

`void DL_cspline::assign(DL_cspline *b)`

This method assigns C-spline `b` to this C-spline.

`void DL_cspline::init(DL_geo *g, DL_List *p, boolean c)`

This method initializes the C-spline with control points as listed in list `p`, and with boolean `c` indicating if the C-spline has to be cyclic or not.

`void DL_cspline::update_control_point(int i, DL_point *p)`

This method assigns new coordinates `p` to the `i`-th control point.

`void DL_cspline::deriv2(DL_Scalar s, DL_vector *v)`

This method returns the second derivative at curve parameter `s` in vector `v`.

9.2 Surfaces

The surface classes are used to model explicit surfaces: functions from one surface parameter pair to 3-D points. These are the surfaces that are use by the point-to-surface constraint (see Section 8.12 [pts], page 56). With each surface, a geo is associated: the surface is expressed in the local coordinate system of that geo (or in world coordinates if the geo-reference happens to be the `NULL` pointer). In case the geo is a dyna, the point-to-surface constraint's reaction force is also applied to this geo. The `DL_surface` class is the generic surface class which presents the general explicit surface-API, while the other classes discussed here are actual implementations of specific types of surfaces.

9.2.1 Surface

The generic explicit surface class provides the generic API for surfaces. It should be considered an abstract class, and only the surface specializations provide an initialization method.

```
class DL_surface {
  void assign(DL_surface*);
  boolean pos(DL_Scalar,DL_Scalar,DL_point*);
  boolean deriv0(DL_Scalar,DL_Scalar,DL_vector*);
  boolean deriv1(DL_Scalar,DL_Scalar,DL_vector*);
  boolean indomain(DL_Scalar,DL_Scalar);
  boolean closeto(DL_point*,DL_Scalar*,DL_Scalar*);

  DL_geo* get_geo();
  DL_Scalar get_minparam0();
  DL_Scalar get_maxparam0();
  DL_Scalar get_minparam1();
  DL_Scalar get_maxparam1();

        DL_surface();
        ~DL_surface();
}
```

`void DL_surface::assign(DL_surface *s)`

This method assigns surface `s` to this surface

`boolean DL_surface::pos(DL_Scalar s, DL_Scalar t, DL_point *p)`

This method assigns the position (in the local coordinate system of the geo associated with the surface) for surface parameter pair `(s,t)` to `p`. It returns if `(s,t)` is within the surface domain.

`boolean DL_surface::deriv0(DL_Scalar s, DL_Scalar t, DL_vector *d)`

This method assigns the derivative in the first surface parameter (given in the local coordinate system of the geo associated with the surface) for surface parameter pair `(s,t)` to `d`. It returns if `(s,t)` is within the surface domain.

`boolean DL_surface::deriv1(DL_Scalar s, DL_Scalar t, DL_vector *d)`

This method assigns the derivative in the second surface parameter (given in the local coordinate system of the geo associated with the surface) for surface parameter pair `(s,t)` to `d`. It returns if `(s,t)` is within the surface domain.

`boolean DL_surface::indomain(DL_Scalar s, DL_Scalar t)`

This method returns whether surface parameter pair `(s,t)` is within the surface domain.

`boolean DL_surface::closeto(DL_point *p, DL_Scalar *s, DL_Scalar *t)`

This method returns in `*s` and `*t` a surface parameter pair which is such that the distance between that point on the surface and point `p` (given in world coordinates) is minimal.

`DL_geo* DL_surface::get_geo()`

This method returns a reference to the geo that is associated with this surface.

`DL_Scalar DL_surface::get_minparam0()`
`DL_Scalar DL_surface::get_maxparam0()`
`DL_Scalar DL_surface::get_minparam1()`
`DL_Scalar DL_surface::get_maxparam1()`

These methods return the minimum and maximum values for the two surface parameters. These values determine the domain of the surface. Some surface specializations may offer methods that allow you to also change these values, but not all surface types may want to allow this.

9.2.2 Flat surface (plane)

The flat surface class provides a surface definition for a plane (without restrictions on the domain). Here is its API:

```
class DL_flatsurface : public DL_surface {
  void assign(DL_flatsurface*);

  void init(DL_geo*,DL_point*,DL_vector*);
  void init(DL_geo*,DL_point*,DL_vector*,DL_vector*);

  void set_minparam0(DL_Scalar);
  void set_maxparam0(DL_Scalar);
  void set_minparam1(DL_Scalar);
  void set_maxparam1(DL_Scalar);

       DL_flatsurface();
       ~DL_flatsurface();
```

```
    }
```

`void DL_flatsurface::assign(DL_flatsurface *f)`

This method assigns flat surface `f` to this surface.

`void DL_flatsurface::init(DL_geo *g, DL_point *p, DL_vector *n)`

This method initializes the plane as the plane in `g`, through point `p` and with normal `n`.

`void DL_flatsurface::init(DL_geo *g, DL_point *p, DL_vector *v, DL_vector *w)`

This method initializes the plane as the plane in `g`, through point `p` and spanned by vectors `v` and `w` (this has as advantage above the other `init` method that there is more control over the directions governed by the surface parameters).

`void DL_flatsurface::set_minparam0(DL_Scalar m)`
`void DL_flatsurface::set_maxparam0(DL_Scalar m)`
`void DL_flatsurface::set_minparam1(DL_Scalar m)`
`void DL_flatsurface::set_maxparam1(DL_Scalar m)`

These methods provide a means of setting the domain boundaries. These are not actually used by the surface itself, but might be used by for example a visualization routine which only visualizes part of the (infinite) surface.

9.2.3 Ellipsoid

The ellipsoid surface class provides an explicit surface definition for an ellipsoid surface. Due to the explicit nature, there are two points on the ellipsoid (at the tips of the second major axis) that have no clearly defined derivative. So any point-to-surface constraint which happens to visit either of these two points will fail.

```
    class DL_ellipsoid : public DL_surface {
      void assign(DL_ellipsoid*);
      void init(DL_geo*,DL_point*,DL_vector*,DL_vector*,DL_vector*);

          DL_ellipsoid();
          ~DL_ellipsoid();
    }
```

`void DL_ellipsoid::assign(DL_ellipsoid *e)`

This method assigns ellipsoid `e` to this ellipsoid.

`void DL_ellipsoid::init(DL_geo *g, DL_point *c,`
`                        DL_vector *x, DL_vector *y, DL_vector *z)`

This method initializes the ellipsoid. It is associates with geo `g`, and has point `c` as its center. The three major axis are vectors `x`, `y` and `z`. The surface is defined as $sin(s)x+cos(s)cos(t)y+sin(t)z$ for surface parameter pairs (s,t).

9.3 Sensors

The sensor class provides a uniform way to take measurements in the simulated system (via the `sense` method). Each sensor provides one scalar value as its sensor reading. Sensors are used by some of the `controller` classes, which try to steer the system (though `actuators`) in such a way that the sensor reading equals a given reference signal.

Here is the sensor interface description:

```
class DL_sensor {
protected:
public:
  virtual DL_Scalar sense();

      DL_sensor();
      ~DL_sensor();
}
```

`DL_Scalar DL_sensor::sense()`

This method lets the sensor take a measurement and returns the sensor reading.

9.3.1 Sensor_dist_v

A sensor measuring distance in a given direction:

```
class DL_sensor_dist_v: public DL_sensor {
  void init(DL_dyna*, DL_point*, DL_vector*, DL_geo*, DL_point*);

      DL_sensor_dist_v();
      ~DL_sensor_dist_v();

}
```

```
void DL_sensor_dist_v::init(DL_dyna *d, DL_point *pd, DL_vector *rd,
                            DL_geo *g, DL_point *pg)
```

This method initialises the sensor to measure the distance between point `pd` in `d`, and point `pg` in `g`, in direction `rd` in `d`.

9.3.2 Sensor_velo_v

A sensor measuring velocity in a given direction:

```
class DL_sensor_velo_v: public DL_sensor {
  void init(DL_dyna*, DL_point*, DL_vector*, DL_geo*, DL_point*);

     DL_sensor_velo_v();
     ~DL_sensor_velo_v();

}
```

```
void DL_sensor_velo_v::init(DL_dyna *d, DL_point *pd, DL_vector *rd,
                            DL_geo *g, DL_point *pg)
```

This method initialises the sensor to measure the relative velocity between point `pd` in `d`, and point `pg` in `g`, in direction `rd` in `d`.

9.3.3 Sensor_angle_v

A sensor measuring the angle between two vectors around an axis with a given direction:

```
class DL_sensor_angle_v: public DL_sensor {
  void init(DL_dyna*, DL_vector*, DL_vector*, DL_geo*, DL_vector*);
  void initw(DL_dyna*, DL_vector*, DL_vector*, DL_geo*, DL_vector*);

     DL_sensor_angle_v();
     ~DL_sensor_angle_v();

}
```

```
void DL_sensor_angle_v::init(DL_dyna *d, DL_vector *dd, DL_vector *rd,
                             DL_geo *g, DL_vector *dg);
```

This method initialises the sensor to measure the relative angle around the axis with direction `rd` in `d`, between vectors `dd` in `d` and `dg` in `g`.

`void DL_sensor_angle_v::initw(DL_dyna *d, DL_vector *dd, DL_vector *rd, DL_geo *g, DL_vector *dg);`

This method does the same as the `init` method above, but now the three vectors are given in world coordinates (the method will convert them to local coordinates).

9.3.4 Sensor_avelo_v

A sensor measuring angular velocity between two vectors around an axis with a given direction:

```
class DL_sensor_avelo_v: public DL_sensor {
  void init(DL_dyna*, DL_vector*, DL_vector*, DL_geo*, DL_vector*);
  void initw(DL_dyna*, DL_vector*, DL_vector*, DL_geo*, DL_vector*);

      DL_sensor_avelo_v();
      ~DL_sensor_avelo_v();

}
```

`void DL_sensor_avelo_v::init(DL_dyna *d, DL_vector *dd, DL_vector *rd, DL_geo *g, DL_vector *dg);`

This method initialises the sensor to measure the relative angular velocity around the axis with direction `rd` in `d`, between vectors `dd` in `d` and `dg` in `g`.

`void DL_sensor_avelo_v::initw(DL_dyna *d, DL_vector *dd, DL_vector *rd, DL_geo *g, DL_vector *dg);`

This method does the same as the `init` method above, but now the three vectors are given in world coordinates (the method will convert them to local coordinates).

9.4 Actuators

The actuator class provides a uniform way to apply different forces, torques and impulses in a simulated system (via the `apply` method). Each actuator transforms one scalar value to an applied force, torque or impulse and applies it. Sensors are used by some of the `controller` classes, which try to steer the system though `actuators` in such a way that certain `sensor` readings equal given reference signals.

In order to make it easy to visualise the calculated forces, torques and impulses, the `actuator` class inherits from the `force_drawable` class.

Here is the actuator interface description:

```
class DL_actuator: public DL_force_drawable {
protected:
public:
  virtual void apply(DL_Scalar);

  void set_max_actuator(DL_Scalar);
  DL_Scalar get_max_actuator();

       DL_actuator();
       ~DL_actuator();
}
```

`void DL_actuator::apply(DL_Scalar a)`

This method lets the actuator apply its forces, torques and/or impulses based on actuator value `a`.

`void DL_actuator::set_max_actuator(DL_Scalar m)`

Using this method, a maximum actuator value can be given (no limit if the value is smaller or equal than zero). If a value larger than this value is provided in the `apply` method, it is replaced with the maximum. The default is no maximum (a value of 0).

`DL_Scalar DL_actuator::get_max_actuator()`

This method returns the current maximum actuator value.

9.4.1 Actuator_fv

An actuator that applies a force in a given direction:

```
class DL_actuator_fv: public DL_actuator {
  void init(DL_dyna*, DL_point*, DL_vector*, DL_geo*, DL_point*);

       DL_actuator_fv();
       ~DL_actuator_fv();
}
```

```
void DL_actuator_fv::init(DL_dyna *d, DL_point *pd, DL_vector *rd,
                          DL_geo *g, DL_point *pg)
```

This method initialises the actuator. The `apply` method will calculate a force in the direction of `rd` in `d` with a magnitude of the scalar given as a parameter to the `apply` method. This force is then applied to point `pd` of `d`, and -inverted- to point `pg` of `g`.

9.4.2 Actuator_tv

An actuator that applies torques in a given direction:

```
class DL_actuator_tv: public DL_actuator {
  void init(DL_dyna*, DL_vector*, DL_geo*);
  void initw(DL_dyna*, DL_vector*, DL_geo*);

      DL_actuator_tv();
      ~DL_actuator_tv();
}
```

`void DL_actuator_tv::init(DL_dyna *d, DL_vector *rd, DL_geo *g)`

This method initialised the actuator. The `apply` method will calculate a torque in the direction of `rd` in `d` with a magnitude of the scalar given as a parameter to the `apply` method. This torque is then applied to `d`, and -inverted- to `g`.

`void DL_actuator_tv::initw(DL_dyna *d, DL_vector *rd, DL_geo *g)`

This method does the same as the `init` method above, but now vector rd is given in world coordinates (the method will convert it to local coordinates).

9.5 Controllers

Controllers are objects that calculate and apply controller forces once each frame, in contrast to constraints which –within a frame– can repeatedly improve on the reaction forces and can also ‘look ahead’ at the results of the application of the forces. As a result of this, controllers are usually of a ‘slower’, less exact nature than constraints, but they also take less computational effort. Often a controller is used in concert with a constraint to regulate the degrees of freedom that the constraint leaves free.

9.5.1 Controller

The controller class provides a base-class for controllers. This base class makes sure that controllers are registered with the dyna system, and that they indeed are prompted once a frame to calculate and apply their controller force. The ‘deactivate’ and ‘activate’ methods can be used to (temporarily) deactivate and reactivate the controller. Only the controller’s descendants should be instantiated.

```
class DL_controller : public DL_force_drawable {
  void activate();
  void deactivate();

      DL_controller();
      ~DL_controller();
}
```

`void DL_controller::activate()`

This method activates the controller (it is usually automatically invoked by the initialization methods of specific controllers, so it is usually only used explicitly to reactivate a constraint after it has been deactivated.

`void DL_controller::deactivate()`

this method deactivates the controller: it will not calculate or apply controller forces until it is activated again.

9.5.2 Spring (/damper)

The spring class is used to model (dampened) springs. As with connection constraints, upon initialization the two connection-points should be supplied, optionally with the rest-length of the spring, the spring constant, and -if required- a damping factor. This class can also be used to model dampers: in that case a spring constant of zero and a non-zero damping constant are used.

```
class DL_spring : public DL_controller {
  DL_Scalar maxforce;
  DL_Scalar l,c,dc;
  boolean el;

  void init(DL_dyna*, DL_point*, DL_geo*, DL_point*);
  void init(DL_dyna*, DL_point*, DL_geo*, DL_point*,DL_Scalar,DL_Scalar);
  void init(DL_dyna*, DL_point*, DL_geo*, DL_point*,DL_Scalar,DL_Scalar,DL_Scalar);

  void springforce(DL_vector*);

      DL_spring();
      ~DL_spring();
}
```

`DL_Scalar DL_spring::maxforce`

With this attribute, a maximum spring force for the controller can be specified: if this maximum force is exceed the controller will deactivate itself, so the connection ‘breaks’. A `maxforce` value of zero indicates that there is no limit to the spring force magnitude.

For a negative `maxforce` value, the spring does not deactivate itself when the maxforce magnitude is exceeded, but the portion of the force exceeding the maxforce value is not applied. The default value for `maxforce` is zero.

`DL_Scalar DL_spring::l`

This attribute specifies the rest length of the spring (initial value: 0)

`DL_Scalar DL_spring::c`

This attribute specifies the spring constant (initial value: 0)

`DL_Scalar DL_spring::dc`

This attribute specifies the damping constant (initial value: 0)

`boolean DL_spring::el`

This attribute specifies if the spring acts like a rubber band: if true the spring will only exert pulling forces (no pushing forces are applied). The initial value for this attribute is false.

`void DL_spring::init(DL_dyna *d, DL_point *pd, DL_geo *g, DL_point *pg)`

This method initializes the spring with attachment points `pd` in `d`, and `pg` in `g` (both points in local coordinates of their respective geometries). This method leaves the rest length, and the spring and damping constants at their current value.

```
void DL_spring::init(DL_dyna *d, DL_point *pd, DL_geo *g, DL_point *pg,
                     DL_Scalar l, DL_Scalar c)
```

This method initializes the spring just like the method above, but also specifies the new rest length to be `l`, and the new spring constant to be `c`.

```
void DL_spring::init(DL_dyna *d, DL_point *pd, DL_geo *g, DL_point *pg,
                     DL_Scalar l, DL_Scalar c, DL_Scalar dc)
```

This method initializes the spring just like the method above, but also specifies the new damping constant to be `dc`.

`void DL_spring::springforce(DL_vector *f)`

This method assigns the most recently calculated spring force to `f`.

9.5.3 Torquespring (/damper)

The torquespring class is used to model (dampened) rotational springs that try to enforce a relative orientation of two vectors around a given axis. Upon initialization the three evctors have to be given, optionally with the rest-angle of the spring, the spring constant, and -if required- a damping factor. This class can also be used to model dampers: in that case a spring constant of zero and a non-zero damping constant are used.

```
class DL_torquespring : public DL_controller {
  DL_Scalar maxtorque;
  DL_Scalar a,c,dc;

  void init(DL_dyna*, DL_vector*, DL_vector*, DL_geo*, DL_vector*);
  void init(DL_dyna*, DL_vector*, DL_vector*, DL_geo*, DL_vector*,
           DL_Scalar,DL_Scalar);
  void init(DL_dyna*, DL_vector*, DL_vector*, DL_geo*, DL_vector*,
           DL_Scalar,DL_Scalar,DL_Scalar);

  void springtorque(DL_vector*);

      DL_torquespring();
      ~DL_torquespring();
}
```

`DL_Scalar DL_torquespring::maxtorque`

With this attribute, a maximum spring torque for the controller can be specified: if this maximum force is exceed the controller will deactivate itself, so the connection 'breaks'. A `maxtorque` value of zero indicates that there is no limit to the spring torque magnitude. For a negative `maxtorque` value, the spring does not deactivate itself when the maxtorque magnitude is exceeded, but the portion of the torque exceeding the maxtorque value is not applied. The default value for `maxtorque` is zero.

`DL_Scalar DL_torquespring::a`

This attribute specifies the rest angle of the spring (initial value: 0)

`DL_Scalar DL_torquespring::c`

This attribute specifies the spring constant (initial value: 0)

`DL_Scalar DL_torquespring::dc`

This attribute specifies the damping constant (initial value: 0)

```
void DL_torquespring::init(DL_dyna *d, DL_vector *dd, DL_vector *dir,
                           DL_geo *g, DL_vector *dg)
```

This method initializes the spring, specifying that the angle is to be determined between vector `dd` in `d` and vector `dg` in `g`, around vector `dir` in `d` (all vectors in local coordinates of their respective geometries). This method leaves the rest length, and the spring and damping constants at their current value.

```
void DL_torquespring::init(DL_dyna *d, DL_vector *dd, DL_vector *dir,
                           DL_geo *g, DL_vector *dg,
                           DL_Scalar a, DL_Scalar c)
```

This method initializes the spring just like the method above, but also specifies the new rest angle to be `a`, and the new spring constant to be `c`.

```
void DL_torquespring::init(DL_dyna *d, DL_vector *dd, DL_vector *dir,
                           DL_geo *g, DL_vector *dg,
                           DL_Scalar a, DL_Scalar c, DL_Scalar dc)
```

This method initializes the spring just like the method above, but also specifies the new damping constant to be `dc`.

```
void DL_torquespring::springtorque(DL_vector *t)
```

This method assigns the most recently calculated spring torque to `t`.

9.5.4 PID Controller

The PID controller class is a class of controllers that works in conjunction with the sensor and actuator classes. It tries to activate the actuator in such a way that the sensor reading from the sensor matches a given target signal (which may be varying over time). It does so by measuring the error between the target value and the sensor value and steering the actuator proportionally to that error, its integral, and/or it's derivative.

The PID class can be used in two different ways:

1. With the sensors and actuators from this class library. This method is used when the controller is provided with a sensor and an actuator upon contruction. The controller is the automatically activated and will start reading its sensor and applying its calculates actuator values.
2. With external means for measuring and applying actuator values. This method is used when the controller is not given a sensor and actuator upon contruction. The controller is not activated then, but a user can still call the `sens2act()` method each frame providing the controller with a sensor reading and getting the appropriate actuator value in return.

For example, the `sensor_dist_v` (or `sensor_velo_v`) sensor and the `actuator_fv` actuator can be combined with the PID controller class to govern the left-over degree of freedom of the prism constraint. In the same manner, the `sensor_angle_v` (or `sensor_avelo_v`) sensor and the `actuator_tv` actuator can be combined to control the remaining degree of freedom of the linehinge constraint.

Here is the PID controller interface:

class **DL_pid** : public **DL_controller {**
  void init_coefs(DL_Scalar, DL_Scalar, DL_Scalar);

  void set_target(DL_Scalar);

```
    DL_Scalar get_target();

    void set_pcoef(DL_Scalar);
    DL_Scalar get_pcoef();
    void set_icoef(DL_Scalar);
    DL_Scalar get_icoef();
    void set_dcoef(DL_Scalar);
    DL_Scalar get_dcoef();

    DL_Scalar sens2act(DL_Scalar);

            DL_pid(DL_sensor*, DL_actuator*);
            DL_pid();
      ~DL_pid();
  }
```

`void DL_pid::init_coefs(DL_Scalar pc, DL_Scalar ic, DL_Scalar dc)`

This method sets the proportional coefficient to `pc`, the coefficient for the integrated error to `ic`, and the coefficient for the derivative of the error to `dc`.

`void DL_pid::set_target(DL_Scalar t)`

This method sets the current controller target to `t`.

`DL_Scalar DL_pid::get_target()`

This method returns the current controller target.

`void DL_pid::set_pcoef(DL_Scalar pc)`

This method sets the proportional coefficient to `pc`.

`DL_Scalar DL_pid::get_pcoef()`

This methods returns the current proportional coefficient.

`void DL_pid::set_icoef(DL_Scalar ic)`

This method sets the coefficient for the integrated error to `ic`.

`DL_Scalar DL_pid::get_icoef()`

This methods returns the current coefficient for the integrated error.

`void DL_pid::set_dcoef(DL_Scalar dc)`

This method sets the coefficient for the derivative of the error to `dc`.

`DL_Scalar DL_pid::get_dcoef()`

This methods returns the current coefficient for the derivative of the error.

`DL_Scalar sens2act(DL_Scalar)`

This method returns the actuator value for the given sensor reading (to be called each frame when no sensor and actuator were provided to the controller at construction time.

Table of Contents