DynaMechs Simulation Library

— Version 3.0 —

Reference Manual

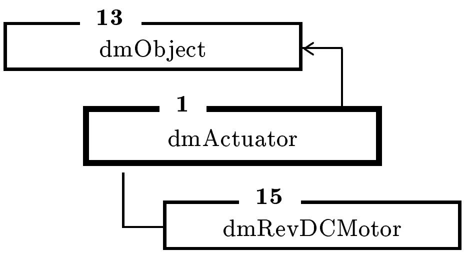
Scott McMillan

Contents

1

class DM_DLL_API **dmActuator** : public dmObject

Inheritance



Public Members

	dmActuator ()
virtual	**~dmActuator** ()
void	**initStiction** (Float qd)
inline bool	**getStictionFlag** ()
virtual void	**updateStiction** (Float *new_joint_vel)
virtual Float	**computeTau** (Float source_voltage, Float external_tau, Float joint_vel)

This is the abstract base class of an experimental (underdeveloped) class hierarchy of actuators. This and the (currently) only derived class (`dmRevDCMotor`) give an example of one way in which this could be done. As it is currently implemented, an actuator is a member of the `dmMDHLink` class and is added to this class in much the same way that a contact model is added to a rigid body.

Part of the motor model provided with this class is stiction. The presence of stiction requires that the velocity must go to zero when ever a sign change in velocity of the simulated joint is detected. This is important because it allows the model to enter the static friction domain. In order to accomplish this though, the joint velocity must be overridden whenever a sign change occurs. First, the stiction simulation must be initialized by calling the `initStiction` function with the initial joint velocity. Then the `updateStiction` function, which is implemented by the derived actuator classes, is called at the beginning of the link's `ABForwardKinematics` function. Finally, the link's `ABForwardAcceleration` function must call the `getStictionFlag` function to determine if the joint acceleration should be overridden to zero as well. There is no gaurantee that this model is correct, but it looks pretty good when it works.

Note that when an actuator is present in a link class the meaning of the parameter to the `setJointInput` function changes from a joint torque/force to the actuator's input (a voltage in the case of `dmRevDCMotor`). The `computeTau` function is called by the link's `ABBackwardDynamics` function and passes the link's joint input value in the first parameter, any external torques about the motor/joint axis that are present in the second parameter, and the current joint velocity in the third parameter. This function is a pure virtual function, which is implemented by the derived classes to simulate the motor dynamics.

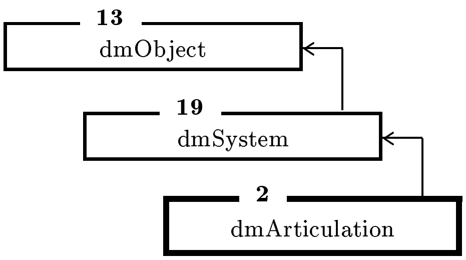
Also note that all of these functions were developed assuming that Euler integration would be used. I do not know what affect a multistep method like Runge-Kutte would have on the stiction flag updating methods.

This page has been automatically generated with DOC++ ©1995 by Roland Wunderling, Malte Zöckler

2

class DM_DLL_API **dmArticulation** : public dmSystem

Inheritance



Public Members

	dmArticulation ()
virtual	**~dmArticulation** ()
bool	**addLink** (dmLink *new_link, dmLink *parent_link)
unsigned int	**getNumLinks** ()
dmLink*	**getLink** (unsigned int index)
int	**getLinkIndex** (dmLink *link)
unsigned int	**getNumDOFs** ()
void	**setJointInput** (Float joint_input[])
void	**setState** (Float q[], Float qd[])
void	**getState** (Float q[], Float qd[])
bool	**forwardKinematics** (unsigned int link_index, dmABForKinStruct *fk)
const dmABForKinStruct*	**getForKinStruct** (unsigned int link_index)
void	**draw** ()

The `dmArticulation` class would be better named the `dmTreeStructure` class. It is a subclass of the `dmSystem` class that implements the functionality need to model links organized in a tree topology. *Soon a subclass of* `dmArticulation` *will allow extension of the package to closed-chain structures.* This class contains a database of links and the predecessor/successor relationships between links that define the tree topology. As such, the first link and any other link added to the articulation using the `addLink` function with the second argument set to NULL are considered that root of the tree. Then, as additional links are added to the articulation, a pointer to a parent link (already added to the articulation) is used to specify its location in the tree. As a result, each link in a tree has only one parent. The only exceptions are the root links that are defined relative to the `dmSystem`'s inertial reference coordinate system.

The default constructor returns an empty articulation (that is, with no links). As links are added to the articulation, internal variables are allocated that will be used to perform the dynamic computations. When the destructor is called these internal variables are freed but any links that have been added to the articulation are not.

The `addLink` function is called to add a link to the tree structure that is maintained by the `dmArticulation` objects. The first parameter is a pointer to a previously instantiated link object that is to be added, and the second parameter is a pointer to a link that has already been added to the articulation and will be the "parent" to the new link and placed in the database accordingly. Note that the first link added has no parent so the `parent` parameter is set to `NULL`. If the operation is successful, this function returns `true`; otherwise it will return `false` for any one of a number of reasons (parent not found, variable allocation failure, etc.)

The articulation can be queried for the number of links in the tree by calling `getNumLinks`. A pointer to a specific link can be obtained by calling `getLink` with the appropriate index. The index corresponds to the order in which the links were added to the tree: 0 for the first link, 1 for the second, etc. If the index is out of range, `getLink` returns `NULL`. The inverse function, `getLinkIndex` takes a `dmLink` pointer and returns its corresponding index, or `-1` if the link is not contained in the tree or the pointer is `NULL`.

Do I need to add tree traversal functions like `getParent`, `getNumChild`*, etc...(either here or in dmLink)?*

Three functions, `setJointInput`, `setState`, and `getState`, are used to set and query joint states and inputs for the links in the entire tree. The first, `setJointInput`, sets the joint inputs (either torques, forces, or motor voltages) for all the links in the articulation. Its `joint_input` argument is a packed (one-dimensional) array with length equal to the total number of DOFs in the articulation. The elements of this array correspond to the DOFs of each link in the order they were added to the articulation. `setState` sets the state of the DOFs in the articulation. The two arguments are both packed arrays (like `joint_input`) containing the joint positions and velocities of the links. The reverse operation, `getState`, fills packed arrays with the joint positions velocities. A convenience function, `getNumDOFs`, returns the total number of degrees of freedom in the articulation and can be used to determine the appropriate size of the above arrays.

The `forwardKinematics` function composes the homogeneous transformation matrix to the link indicated by the first parameter, `link_index`. The second parameter, `fk`, is a pointer to a `dmABForKinStruct` (from `dm.h`):

```
struct dmABForKinStruct
{
   RotationMatrix  R_ICS;     // orientation of links wrt ICS - ^{ICS}R_{i}
   CartesianVector p_ICS;     // position of links wrt ICS - ^{ICS}p_{i}
   SpatialVector   v;         // velocity of link wrt to i.
     ...
};
```

Upon calling this function in normal use, `R_ICS` and `p_ICS` will contain the transformation from the inertial coordinate system to the reference system (defined in `dmSystem`) relative to which this articulation is defined.

Normally this function is called by `dmSystem::forwardKinematics()` after filling the `fk` parameter with the proper values. Upon exit this function will fill `R_ICS` and `p_ICS` with the resulting transformation information and return `true`. If the `link_index` is out of range, this function returns `false`. *Should I make this function private and* `dmSystem`*'s a friend?*

The three functions, `ABForwardKinematics`, `ABBackwardDynamics`, and `ABForwardAccelerations`, comprise the implementation of the Articulated-Body (AB) Simulation alogrithm recursions are hidden from the user and are only be called by `dmSystem::ABDynamics`. However, during the Forward Kinematics traversal, `dmABForKinStruct`s are computed for each link which can be accessed by calling the `getForKinStruct` function with the index of the desired link. This functions returns a pointer to the requested struct, or NULL if the index is out of range.

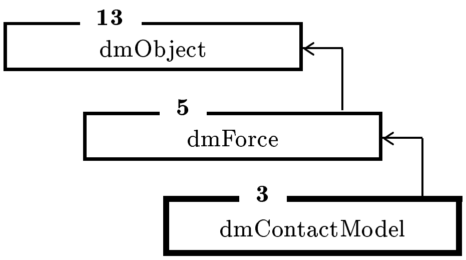
See also `dmSystem, dmLink`.

This page has been automatically generated with DOC++ © 1995 by Roland Wunderling, Malte Zöckler

3

class DM_DLL_API **dmContactModel** : public dmForce

Inheritance



Public Members

	dmContactModel ()
virtual	**~dmContactModel** ()
void	**setContactPoints** (unsigned int num_contact_points, CartesianVector *contact_pos)
unsigned int	**getNumContactPoints** ()
bool	**getContactPoint** (unsigned int index, CartesianVector pos)
void	**computeForce** (dmABForKinStruct *val, SpatialVector force)
inline void	**reset** ()
void	**draw** ()

A `dmContactModel` object is subclassed from the `dmForce` and computes the compliant forces of contact with a prismatic terrain. It is instantiated and "assigned" to a `dmRigidBody` object using the `dmRigidBody::addForce()` function. This class queries the static `dmEnvironment` object about terrain and computes forces of contact between it and the contact points defined in this class for the particular `dmRigidBody` object. *This class is one of those that could use a bit of work. The whole interaction between robot (more specifically, rigid bodies) and environment has not been completely thought through yet, with the dmForce class it has taken one major step forward, though.*

The default constructor instantiates an empty object. A call to the `setContactPoints` function initializes the contact points that will be set. Its first parameter, `num_contact_points`, is the number of points (3D vertices) that are to be allocated, and the second is an array of `num_contact_points CartesianVector`s that specify the constant locations of the contact points relative to the rigid body's coordinate system. The

This page has been automatically generated with DOC++ ©1995 by Roland Wunderling, Malte Zöckler

function `getNumContactPoints` can be used to query the size of the array, and `getContactPoint` can get the values of a particular vertex by index (0 for the first, 1 for the second, etc.). The latter returns TRUE if the index is within range; otherwise, it returns FALSE.

The `computeForce` is the main part of this class and computes the compliant contact force exerted on the rigid body. It is called by the `ABForwardAcceleration` functions of all `dmRefMember` objects and the `dmABBackwardDynamics` functions of the `dmLink` objects that are derived from the `dmRigidBody` class. The `ABForwardKinematics` functions computes the position and orientation of the rigid body with respect to the inertial coordinate system, and the spatial velocity with respect to the body's coordinate system and sets the `p_ICS`, `R_ICS`, and `v` elements of the `dmABForKinStruct` struct (`val`). This struct is used at the beginning of the subsequent pass of the AB algorthm when it calls this class's `computeForce` to compute the resultant spatial force exerted at all of the contact points and returns it in `force` parameter (with respect to the body's coordinate system).

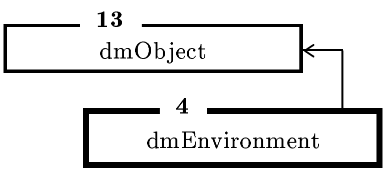
In the current implementation of the `computeForce` function, an initial contact location is set to maintain information from iteration to iteration and compute spring-like forces from the initial contact point. If the state of the system is manually changed while some of these points are in contact with the terrain, large forces can result on the next iteration of the simulation. The `reset` function is used to overcome this potential problem by resetting all contact points. *Also note that this iteration to iteration dependency is really only valid assuming a simple single step integration method like Euler. I do not know how valid this is with multistep methods like Runge-Kutte, but it seems to work so far.*

See also `dmEnvironment`, `dmRigidBody`, `dmLoadFile_dm`

4

class DM_DLL_API **dmEnvironment** : public dmObject

Inheritance



Public Members

	dmEnvironment ()
virtual	**~dmEnvironment** ()
static void	**setEnvironment** (dmEnvironment *env)
static dmEnvironment*	**getEnvironment** ()
void	**loadTerrainData** (char *filename)
const char*	**getTerrainFilename** ()
inline void	**setGravity** (CartesianVector a_g)
inline void	**getGravity** (CartesianVector a_g)
inline void	**setGroundPlanarSpringConstant** (Float k)
inline void	**setGroundNormalSpringConstant** (Float k)
inline void	**setGroundPlanarDamperConstant** (Float k)
inline void	**setGroundNormalDamperConstant** (Float k)
inline void	**setFrictionCoeffs** (Float u_s, Float u_k)
inline Float	**getGroundPlanarSpringConstant** ()
inline Float	**getGroundNormalSpringConstant** ()
inline Float	**getGroundPlanarDamperConstant** ()
inline Float	**getGroundNormalDamperConstant** ()
inline Float	**getGroundStaticFrictionCoeff** ()
inline Float	**getGroundKineticFrictionCoeff** ()
inline Float	**getFluidDensity** ()

This page has been automatically generated with DOC++ ©1995 by Roland Wunderling, Malte Zöckler

inline void	**setFluidDensity** (Float d)
inline void	**getFluidVel** (CartesianVector v)
inline void	**setFluidVel** (CartesianVector v)
inline void	**getFluidAcc** (CartesianVector a)
inline void	**setFluidAcc** (CartesianVector a)
void	**updateFluidMotionEuler** (Float idt, CartesianVector position)
Float	**getGroundDepth** (CartesianVector contact_pos, CartesianVector ground_normal)
Float	**getGroundElevation** (CartesianVector contact_pos, CartesianVector ground_normal)
void	**drawInit** ()
void	**draw** ()

The environment class encapsulates all the information the a dynamic simulation needs for interaction (read forces of contact) between rigid bodies and the surrounding environment (terrain, friction, gravity, etc...). In the case of hydrodynamic simulation (compiled with the -DHYDRODYNAMICS argument), fluid density and motion are also stored in this class. The default constructor initializes an "empty" environment with no terrain, zero gravity, and zero friction coefficients, and when present, zero fluid density and motion.

This class contains a static member variable that will point to the currently "active" environment. It is implemented this way because the dynamic simulation algorithm needs one environment to query during simulation. The current environment is set with a call to the static member function, `dmEnvironment::setEnvironment`, and passing it a pointer to a previously instantiated environment object. Likewise, you can get a pointer to the current environment with a call to the static member function `dmEnvironment::getEnvironment`.

IMPORTANT: The simulation algorithm will fail with a runtime error if an environment has not been set in this fashion because various classes directly reference this object to obtain information about the environment. Gravity is defined by a vector (magnitude *and* direction) and is set by calling the `setGravity` function (and can be queried with the `getGravity`).

Currently, only prismatic terrains are supporte (i.e., gridded elevation data, evenly spaced over a flat plane). Currently the only way to input terrain data is to call the `loadTerrainData` with the name of the file containing the grid spacing and elevation data in the following format:

```
4 5 5.0
0   0    0    0    0
0  -2.0 -2.0 -2.0  0
0  -2.0 -2.0 -2.0  0
0   0    0    0    0
```

This page has been automatically generated with DOC++ ©1995 by Roland Wunderling, Malte Zöckler

where the first two numbers define the dimensions of the grid (in numbers of points along the x and y axes) and the third defines the spacing between the points in both dimensions. The rest define the elevation along the z axis. The `getTerrainFilename` function returns a pointer to the terrain filename that was loaded.

There are many other (reasonably) self-explanatory set and get member functions that assign and return spring and damper parameters. These parameters define the characteristics of the contact surfaces (slipperiness, rigidity, etc.). See the programmer's manual (in process) for more details on the current contact algorithm. *I currently looking for someone to develop the collision algorithms.*

In the case of hydrodynamic simulation, there are also member variables that get and set fluid velocity and acceleration (assumed to be the same over the entire field), and fluid density. There is also an `updateFluidMotionEuler` that I think will update the fluid's velocity based on the current acceleration. *Not much of a fluid simulation....*

The primary user of this class is the `dmContactModel::computeContactForce` function which calls the `getGroundElevation` function (in the case of z-axis up prismatic terrains) to help detect collisions of various contact points with the terrain. *This is a very preliminary class that, like the* `dmContactModel`*, could stand a bit of redesign.* The functions `getGroundElevation` and `getGroundDepth` both take a position vector and returns a distance through the terrain, and passes back the normal vector of the terrain at the point of contact. Currently only the `getGroundElevation` has been tested for use with prismatic terrain defined on a regular grid in the x-y plane with the z-axis pointing up.

Finally two functions, `draw` and `drawInit` are not defined in the library which the user must write to render the system using whatever 3D API the user desires. The framework (no graphics code) for these draw functions for all classes are provided in `draw.C`. Examples of these with OpenGL code is provided with the example that accompanies this distribution (in `gldraw.C`).

A configuration file reader, `dmLoadFile_dm` is being supplied in the dmutils library that can be used to instantiate and intialize contact model objects. The lines in the configuration file are expected to be in the following form:

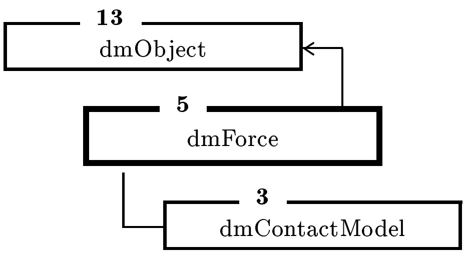
```
Gravity_Vector                      G_x G_y G_z
Terrain_Data_Filename               "filename"
Ground_Planar_Spring_Constant       k_p
Ground_Normal_Spring_Constant       k_n
Ground_Planar_Damper_Constant       b_p
Ground_Normal_Damper_Constant       b_n
Ground_Static_Friction_Coeff        u_s
Ground_Kinetic_Friction_Coeff       u_k
Fluid_Density                       p
```

See also `dmContactModel`, `dmLoadFile_dm`.

This page has been automatically generated with DOC++ ©1995 by Roland Wunderling, Malte Zöckler

5 class DM_DLL_API **dmForce** : public dmObject

Inheritance



Public Members

		dmForce ()
virtual		**~dmForce** ()
virtual	void	**reset** ()
virtual	void	**computeForce** (dmABForKinStruct *val, SpatialVector force)
virtual	void	**draw** ()

This is the abstract base class for all force objects that would be associated with a `dmRigidBody` class. This class defines the interface, as all of its functions are pure virtual ones.

The `reset` function is one required by the `dmContactModel` because certain retained state variables when the state of the rigid body is brutally overridden (which should never be done in a normal simulation). As such this requirement on the interface is most unsatisfying.

The `computeForce` function is the heart of this class which computes the compliant force based on the rigid body's state information provided by the `dmABForKinStruct` parameter and returns the spatial force in the `force` parameter.

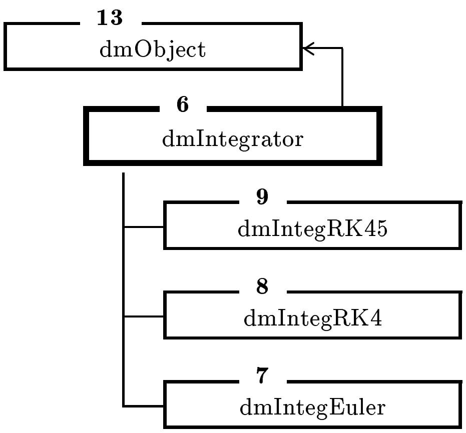
See also `dmContactModel`, `dmRigidBody`.

This page has been automatically generated with DOC++ ©1995 by Roland Wunderling, Malte Zöckler

6

class DM_DLL_API **dmIntegrator** : public dmObject

Inheritance



Public Members

	dmIntegrator ()
virtual	**~dmIntegrator** ()
void	**setSystem** (dmSystem *system)
dmSystem*	**getSystem** ()
int	**isReadyToSim** ()
virtual void	**simulate** (Float &idt)

This class is under major construction.

This is the abstract base class for numerical integrators that will be used in conjunction with a `dmSystem` object to perform the dynamic simulation. After construction of a derived class, a system object must be assigned to the integrator object with a call to the `setSystem` function. A pointer to currently assigned system can be obtained with a call to the `getSystem` function. Currently, a new system cannot be properly assigned to an integrator after the initial assignment.

Immediately after construction of this class, it is not setup to numerically integrate anything and a call to `isReadyToSim` returns FALSE. After a system has been assigned and internal variables have been successfully allocated (based on the number of degrees of freedom in the system), this function will return TRUE.

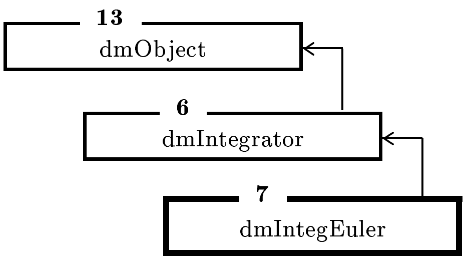
The system is simulated by calling the `simulate` function which is a pure virtual function implemented in the derived integrator classes (see **`dmIntegEuler, dmIntegRK4`**, etc.). The `idt` parameter, indicates how much time should be simulated in seconds, and on return will contain the amount of time that was actually simulated.

This page has been automatically generated with DOC++ ©1995 by Roland Wunderling, Malte Zöckler

7

class DM_DLL_API **dmIntegEuler** : public dmIntegrator

Inheritance



Public Members

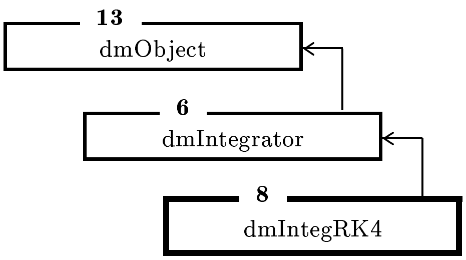
	dmIntegEuler ()
virtual	~**dmIntegEuler** ()
void	**simulate** (Float &idt)

This class is under major construction.

This is a concrete integrator class derived from the `dmIntegrator` class, which implements the first order, fixed stepsize Euler integration algorithm. The `idt` parameter corresponds to the size of the step that is taken with a single iteration of the algorithm. No judgement of success or failure of the step is made and the parameter will be unchanged on return.

8 class DM_DLL_API **dmIntegRK4** : public dmIntegrator

Inheritance



Public Members

	dmIntegRK4 ()
virtual	~**dmIntegRK4** ()
void	**simulate** (Float &delta_t)

This class is under major construction.

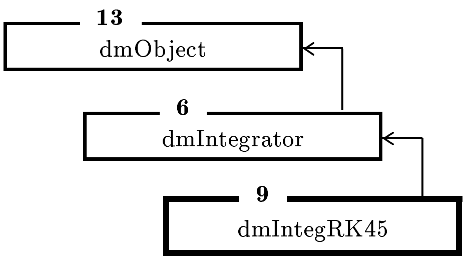
This is a concrete integrator class derived from the `dmIntegrator` class, which implements the fourth order, fixed stepsize Runge-Kutta integration algorithm. Note that in the process, four calls to the associated `dmSystem::ABDynamics` function will be made. The `idt` parameter corresponds to the size of the step that is taken with a single iteration of the algorithm. No judgement of success or failure of the step is made and the parameter will be unchanged on return.

This page has been automatically generated with DOC++ © 1995 by Roland Wunderling, Malte Zöckler

9

class DM_DLL_API **dmIntegRK45** : public dmIntegrator

Inheritance



Public Members

	dmIntegRK45 ()
virtual	**~dmIntegRK45** ()
void	**simulate** (Float &delta_t)

This class is under major construction and is virtually untested.

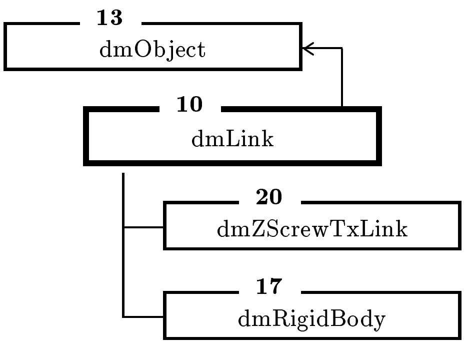
This is a concrete integrator class derived from the `dmIntegrator` class, which implements an embedded fourth/fifth order, adaptive stepsize Runge-Kutta integration algorithm. Note that in the process, six calls to the associated `dmSystem::ABDynamics` function will be made. The `idt` parameter corresponds to the amount of time that should be simulated, but not necessarily the size of the step that is taken with a single iteration of the algorithm.

I have only just started playing with this algorithm. Many of the quality control parameters need to be set-able by the user, but currently no accessor functions have been provided. Initial tests of the algorithm show that it cannot handle steps across collision boundaries very well. This may be overcome with an, as yet, unknown set of control parameters.

10

class DM_DLL_API **dmLink** : public dmObject

Inheritance



Public Members

	dmLink ()
virtual	**~dmLink** ()
virtual int	**getNumDOFs** ()
virtual void	**setState** (Float q[], Float qd[])
virtual void	**getState** (Float q[], Float qd[])
virtual void	**setJointInput** (Float joint_input[])
Float	**getJointFriction** ()
void	**setJointFriction** (Float u_c)
virtual void	**rtxToInboard** (CartesianVector curr, CartesianVector prev)
virtual void	**rtxFromInboard** (CartesianVector prev, CartesianVector curr)
virtual void	**stxToInboard** (SpatialVector curr, SpatialVector prev)
virtual void	**stxFromInboard** (SpatialVector prev, SpatialVector curr)
virtual void	**rcongtxToInboardSym** (CartesianTensor Curr, CartesianTensor Prev)
virtual void	**rcongtxToInboardGen** (CartesianTensor Curr, CartesianTensor Prev)
virtual void	**scongtxToInboardIrefl** (SpatialTensor N_curr, SpatialTensor N_prev)

virtual void **ABForwardKinematics** (Float q[], Float qd[], dmABForKinStruct *link_val_inboard, dmABForKinStruct *link_val_curr)

virtual void **ABBackwardDynamics** (dmABForKinStruct *link_val_curr, SpatialVector f_star_curr, SpatialTensor N_refl_curr, SpatialVector f_star_inboard, SpatialTensor N_refl_inboard)

virtual void **ABBackwardDynamicsN** (dmABForKinStruct *link_val_curr, SpatialVector f_star_inboard, SpatialTensor N_refl_inboard)

virtual void **ABForwardAccelerations** (SpatialVector a_inboard, SpatialVector a_curr, Float qd[], Float qdd[])

void **forwardKinematics** (dmABForKinStruct *link_val_inboard, dmABForKinStruct *link_val_curr)

virtual void **draw** ()

The `dmLink` class is an abstract base class for all derived link classes. Its main purpose is to support polymorphism across all link classes by providing the virtual interfaces for all the functions that the `dmArticulation` class calls in the course of simulating the system. As such, nearly all functions specified in the public section are pure virtual functions that must be defined in the derived link classes (currently, `dmMDHLink`, `dmSphericalLink`, `dmMobileBaseLink`, and `dmZScrewTxLink`).

The link's state variables (generalized position and velocity variables) can be set and retreived using the `setState` and `getState` functions. The `q` and `qd` arguments correspond to position and velocity, respectively. They are arrays containing n elements where n is the number of degrees of freedom for this particular type of link (and determined by the derived classes). This number is returned when the `getNumDOFs` function is called. The joint input vector (containing motor inputs where applicable, but usually generalized joint forces) can be set by calling `setJointInput`. The parameter is also an array of `getNumDOFs` elements containing the desired values.

A coulomb friction parameter can also be specified and retrieved using the `setJointFriction` and `getJointFriction` functions. This scalar parameters multiplies the joint velocity to produce a force/torque opposing the direction of motion as follows:

$$\tau = \mu_c \, \dot{q}.$$

For dynamics simulation of the link using the AB algorithm, a number of transformation functions are required of all link types. Their functions are listed as follows:

rtxToInboard	${}^{i-1}f_i = {}^{i}R_{i-1}^{T}\ f_i$
rtxFromInboard	${}^{i}\omega_{i-1} = {}^{i}R_{i-1}\ \omega_{i-1}$
stxToInboard	${}^{i-1}\mathbf{f}_i = {}^{i}X_{i-1}^{T}\ \mathbf{f}_i$
stxFromInboard	${}^{i}\mathbf{v}_{i-1} = {}^{i}X_{i-1}\ \mathbf{v}_{i-1}$
rcongtxToInboardSym	${}^{i-1}\bar{I}_i = {}^{i}R_{i-1}^{T}\ \bar{I}_i\ {}^{i}R_{i-1}$
rcongtxToInboardGen	${}^{i-1}G_i = {}^{i}R_{i-1}^{T}\ G_i\ {}^{i}R_{i-1}$
scongtxToInboardIrefl	${}^{i-1}\mathbf{N}_i = {}^{i}X_{i-1}^{T}\ \mathbf{N}_i\ {}^{i}X_{i-1}$

The first two rotate Cartesian vectors between this link's and the inboard (parent) link's coordinate systems, while the second two transform (rotate and translate) spatial vectors. The last three functions perform congruence transformations on matrices to the inboard coordinate system. `rcongtxToInboardSym` affects a rotational congruence transform on sysmmetric 3×3 matrices while its counterpart, `rcongtxToInboardGen`, transforms general 3×3 matrices. The last function, `scongtxToInboardIrefl`, is a specialized function for the AB algorithm which efficiently transforms 6×6 reflected articulated-body inertia matrices taking into account special structure of the matrix based on its joint axes. This one function is the centerpiece of the efficiency of this implementation of the AB algorithm.

The next four functions are used to perform the three recursions of the AB simulation algorithm. The first forward kinematics recursion is implemented for each link with the `ABForwardKinematics` function. This function should only be called by `dmArticulation::ABForwardKinematics` *yeah, this should be private and* `dmArticulation` *should be made a friend.* Again, the `q` and `qd` parameters are the joint position and velocity vectors computed by the current interation of the numerical integration algorithm and are used to set the new state of the link. The `link_val_inboard` is the `ABForKinStruct` (see `dmArticulation`) that has been filled with the values for the inboard link's state dependent quantities. Using these values, the `link_val_curr` struct elements are computed for this link.

The second backward recursion of the AB algorithm is implemented with two different functions for reasons of efficiency. If this link has no children, it is a leaf node in the tree structure, and therefore, the `dmArticulation::ABBackwardDynamics` function will call the `ABBackwardDynamicsN` function which will begin the recursion which requires less computation, by computing `f_star_inboard` and `N_refl_inboard`, the AB bias force and inertia that are reflected across this link's joint and transformed to the inboard link's coordinate system. The function `ABBackwardDynamics` performs a full recursive step for a link with children. It takes the accumulated results of the child links in `f_star_curr` and `N_refl_curr` and computes the same quantities as before. The `ABForKinStruct` parameter is the `link_val_curr` computed in the first pass, and is used to call `dmForce::computeForce()` functions if they have been added to the associated dmRigidBody.

The third forward recursion of the AB algorithm is implemented in the `ABForwardAcceleration` function. Its input parameter, `a_inboard` is the spatial acceleration of the inboard link. This function computes and outputs the spatial acceleration of this link in `a_curr` and fills in the derivative of state variables (`qd` and `qdd`). Note that the velocity vector, `qd`, is not necessarily the same as that used in `ABForwardKinematics`. Rather, it contains the time derivatives of `q`. See `dmSphericalLink` for an example where they are not the same.

The only function that is implemented in this class is the `forwardKinematics` function. It is called by `dmArticulation::forwardKinematics` function and computes the current link's position (`link_val_curr->p_ICS`) and orientation (`link_val_curr->R_ICS`) with respect to the inertial coordinate frame given the inboard link's position and orientation (`link_val_inboard->p_ICS` and

This page has been automatically generated with DOC++ ©1995 by Roland Wunderling, Malte Zöckler

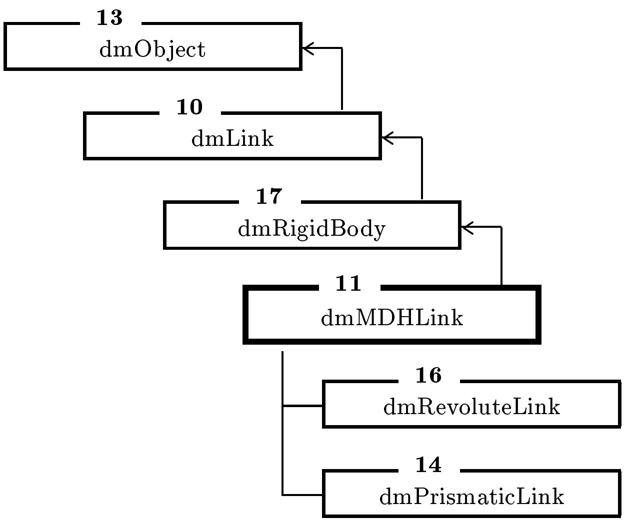
`link_val_inboard->R_ICS`).

See also: `dmArticulation`, `dmMDHLink`, `dmSphericalLink`, `dmZScrewTxLink`, `dmMobileBaseLink`.

11

class DM_DLL_API **dmMDHLink** : public dmRigidBody

Inheritance



Public Members

	dmMDHLink ()
virtual	**~dmMDHLink** ()
void	**setMDHParameters** (Float a, Float alpha, Float d, Float theta)
void	**getMDHParameters** (Float *a, Float *alpha, Float *d, Float *theta)
void	**setJointLimits** (Float min, Float max, Float k_spring, Float b_damper)
void	**getJointLimits** (Float *min, Float *max, Float *k_spring, Float *b_damper)
void	**setActuator** (dmActuator *actuator)
dmActuator*	**getActuator** ()
inline int	**getNumDOFs** ()
void	**setState** (Float q[], Float qd[])

void	**getState** (Float q[], Float qd[])
inline void	**setJointInput** (Float joint_input[])
void	**rtxToInboard** (CartesianVector curr, CartesianVector prev)
void	**rtxFromInboard** (CartesianVector prev, CartesianVector curr)
void	**stxToInboard** (SpatialVector curr, SpatialVector prev)
void	**stxFromInboard** (SpatialVector prev, SpatialVector curr)
void	**rcongtxToInboardSym** (CartesianTensor Curr, CartesianTensor Prev)
void	**rcongtxToInboardGen** (CartesianTensor Curr, CartesianTensor Prev)
virtual void	**scongtxToInboardIrefl** (SpatialTensor N, SpatialTensor I)
void	**ABForwardKinematics** (Float q[], Float qd[], dmABForKinStruct *link_val_inboard, dmABForKinStruct *link_val_curr)
void	**ABBackwardDynamics** (dmABForKinStruct *link_val_curr, SpatialVector f_star_curr, SpatialTensor N_refl_curr, SpatialVector f_star_inboard, SpatialTensor N_refl_inboard)
void	**ABBackwardDynamicsN** (dmABForKinStruct *link_val_curr, SpatialVector f_star_inboard, SpatialTensor N_refl_inboard)
void	**ABForwardAccelerations** (SpatialVector a_inboard, SpatialVector a_curr, Float qd[], Float qdd[])
virtual void	**draw** ()

The `dmMDHLink` class can model the class of links with one degree of freedom (revolute and prismatic) joints. The class is derived from the `dmRigidBody` (for the dynamic parameters) class and, through that, from the `dmLink` base class (required by `dmArticulation` for Articulated Body simulation functions). This, in turn, is an abstract base class for the two kinds of single DOF link classes: `dmRevoluteLink` and `dmPrismaticLink`.

The kinematics (origin and axis of motion) of these links can be defined with four scalar parameters. The algorithm implemented in this library uses the Modified Denavit-Hartenberg (MDH) parameters. This convention assumes that the link's coordinate systems have been placed according to the following rules:

$\hat{z}_i$	lies along the axis of motion of the joint,
$\hat{x}_i$	lies along the common normal between $\hat{z}_i$ and $\hat{z}_{i+1}$, and
$\hat{y}_i$	completes the right-handed coordinate system.

Then the four parameters that define the transformation from the previous body's coordinate system $(i-1)$ to this link's (i) as follows:

a_i	distance along $\hat{x}_{i-1}$ (common normal) from $\hat{z}_{i-1}$ to $\hat{z}_i$,
α_i	angle (in radians) about $\hat{x}_{i-1}$ from $\hat{z}_{i-1}$ to $\hat{z}_i$,
d_i	distance along $\hat{z}_i$ from $\hat{x}_{i-1}$ (the inboard common normal) to $\hat{x}_i$ (the outboard common normal).
θ_i	the angle (in radians) about $\hat{z}_i$ from $\hat{x}_{i-1}$ to $\hat{x}_i$.

For revolute joints, θ_i is the joint position variable, and for prismatic joints, d_i is the variable. These parameters (along with the joint position) is specified with the **setMDHParameters** function.

The **setJointLimits** function is used to specify the range (min and max) of motion for the joint position variable. Should the range be violated, DynaMechs provides a mechanism by which compliant joint limit forces can be simulated with a spring-damper system specified by the **k_spring** and **b_damper** constants. The compliant limit force is computed as follows:

$$\tau \; = \; K_{spring} \, \Delta q - B_{damper} \, \dot{q}$$

where Δq is the amount by which the joint position exceeds the joint limits. This is a force along the joint axis in the case of prismatic joints, and is a torque about the joint axis in the case of revolute joints.

In some cases, actuator dynamics can also be simulated with single DOF **MDHLinks**. Currently only DC motors for revolute joints have been provided with this library (see **dmRevDCMotor**) which can be "assigned" to **dmRevoluteLinks**. This assignment is accomplished with the **setActuator** function. Calling **setActuator** with a NULL argument effectively unsets any existing actuator. A pointer to an assigned actuator can be retrieved with the **getActuator** member function. If no actuator has been assigned it returns NULL.

The remainder of the functions are described in the **dmLink** reference pages and are implemented in this class for the specific case of MDH-type joints. The exceptions is **scongtxToInboardIrefl** which is implemented in the two derived subclasses: **dmRevoluteLink**, and **dmPrismaticLink**. Note that **getNumDOFs** returns 1 for this and the derived classes. As such the joint variables **q**, **qd**, **qdd**, and **joint_input** are arrays of one element.

A configuration file reader, **dmLoadFile_dm** is being supplied with the dmutils library that can be used to instantiate and intialize these link objects. See the dmRevoluteLink and **dmPrismaticLink** classes for the format for each type of link

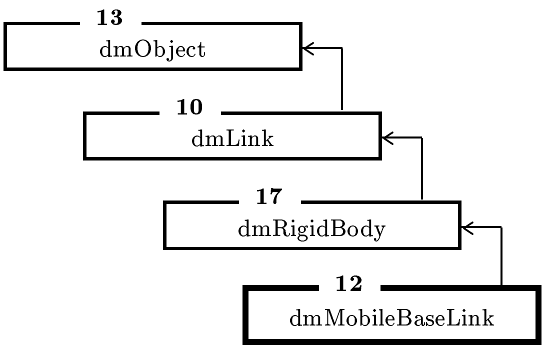
See also **dmRigidBody**, **dmLink**, **dmPrismaticLink**, **dmRevoluteLink**, **dmRevDCMotor**, **dmLoadFile_dm**.

This page has been automatically generated with DOC++ ©1995 by Roland Wunderling, Malte Zöckler

12

class DM_DLL_API **dmMobileBaseLink** : public dmRigidBody

Inheritance



Public Members

	dmMobileBaseLink ()
virtual	**~dmMobileBaseLink** ()
virtual int	**getNumDOFs** ()
virtual void	**setState** (Float q[], Float qd[])
virtual void	**getState** (Float q[], Float qd[])
void	**getDeriv** (Float rdy[13])
void	**rtxToInboard** (CartesianVector curr, CartesianVector prev) *
void	**rtxFromInboard** (CartesianVector prev, CartesianVector curr) *
void	**stxToInboard** (SpatialVector curr, SpatialVector prev) *not implemented*
void	**stxFromInboard** (SpatialVector prev, SpatialVector curr) *not implemented*
void	**rcongtxToInboardSym** (CartesianTensor Curr, CartesianTensor Prev) *not implemented*
void	**rcongtxToInboardGen** (CartesianTensor Curr, CartesianTensor Prev)

		not implemented
void	**scongtxToInboardIrefl**	(SpatialTensor N, SpatialTensor I) *not implemented*
virtual void	**ABForwardKinematics**	(Float q[], Float qd[], dmABForKinStruct *link_val_inboard, dmABForKinStruct *link_val_curr)
virtual void	**ABBackwardDynamics**	(dmABForKinStruct *link_val_curr, SpatialVector f_star_curr, SpatialTensor N_refl_curr, SpatialVector f_star_inboard, SpatialTensor N_refl_inboard)
virtual void	**ABBackwardDynamicsN**	(dmABForKinStruct *link_val_curr, SpatialVector f_star_inboard, SpatialTensor N_refl_inboard)
virtual void	**ABForwardAccelerations**	(SpatialVector a_inboard, SpatialVector a_curr, Float qd[], Float qdd[])
void	**draw**	()

The `dmMobileBaseLinks` objects are links with six degrees of freedom, and hence, the class is derived from the `dmRigidBody` (for the dynamic parameters) and, through that, the `dmLink` base class (required by `dmArticulation` for Articulated-Body simulation functions. The default constructor instantiates a generic mobile base with some valid inertial parameters and zero state variables, but subsequent calls to this and the `dmRigidBody` member functions are needed to set the desired values.

Nearly all of the remainder of the functions are described in the `dmLink` reference pages and are implemented in this class for the specific case of links with a full 6 DOF.

See also: `dmRigidBody`, `dmLink`, `dmLoadFile_dm`

This page has been automatically generated with DOC++ © 1995 by Roland Wunderling, Malte Zöckler

13

class DM_DLL_API **dmObject**

Inheritance

- **13** dmObject
  - **19** dmSystem
  - **10** dmLink
  - **6** dmIntegrator
  - **5** dmForce
  - **4** dmEnvironment
  - **1** dmActuator

Public Members

	dmObject ()
virtual	**~dmObject** ()
void	**setUserData** (void *data)
void*	**getUserData** ()
void	**setName** (const char *name)
const char*	**getName** ()

This is supposed to be a base class for all others, but with the multiple inheritance this gets a little complicated. In the current incarnation, all but the `dmRigidBody` class inherits (either directly or indirectly) from this one. It started out as a class to hold variables needed for the 3D graphics API to render the scene, but after reimplementing it five times (for IRIS GL, Performer, Inventor, Xanimate, and finally OpenGL), I finally got smart and decided I needed a general storage class to store any kind of data with no dependencies on a specific 3D API. The default constructor is the only one available which initializes the storage location to NULL.

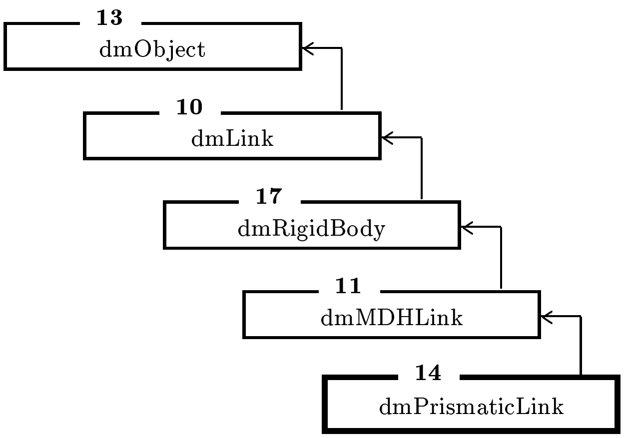
The general storage capability of this class is implemented with the `setUserData` and `getUserData` functions and allows the user to associate application specific data with the various objects created. The former takes a void pointer to some arbitrary data that the user wants to assign to this object which can be retrieved with the latter. Only one pointer to data can be stored with this class so standard use is to allocate a struct containing all of the necessary data and calling `setUserData` with a pointer to the struct (appropriately cast to a `void *`). *A possible bug (potential memory leak) with this class is that when an object derived from this class is destructed no user data pointed to it is deleted. Nor is the old user data deleted when a subsequent call* `setUserData` *is called. This might be solved in the future through a reference counting mechanism, and/or providing a delete callback function.*

As an afterthought, I have also added a capability to name each `dmObject` with a single string. The string can be set using `setName`, in which the string is duplicated inside the class. Calling this function with a new name, or passing a `NULL` parameter will free the space held by any previous names. Use `getName` to get a pointer to this string. Since the string is duplicated (rather than just copying the pointer) no memory leak problems are associated with this mechanism (other than the fact the `getName` returns a pointer to this memory).

14

class DM_DLL_API **dmPrismaticLink** : public dmMDHLink

Inheritance



Public Members

	dmPrismaticLink ()
virtual	**~dmPrismaticLink** ()
void	**scongtxToInboardIrefl** (SpatialTensor N, SpatialTensor I)
void	**draw** ()

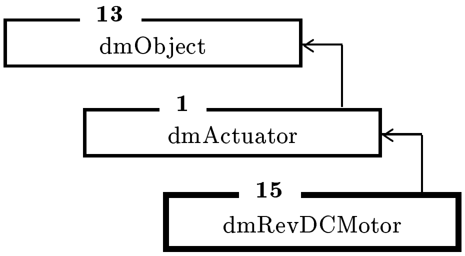
This class is one of the two concrete link classes derived from the `dmMDHLink` class. It implements the dynamics specific to a prismatic (translational) one degree of freedom joint.

Not reference manual material: The primary purpose of this class is to implement the `scongToInboardIrefl` member function, a very efficient congruence transformation of the Articulated-Body inertia matrix that is transformed across the prismatic joint. In this case, the sixth row and column of the matrix are zero and taking this into account results in significant computation savings (for all the gorey details, please read my dissertation or Robotics and Automation Journal article). The `dmMDHLink::ABBackwardDynamics[N]` functions call this function in the course of computing the dynamics. It should probably be a private/protected member function, but since it does not modify any member variables there is no harm in leaving it public.

See also: `dmMDHLink`, `dmLoadFile_dm`.

15 class DM_DLL_API **dmRevDCMotor** : public dmActuator

Inheritance



Public Members

	dmRevDCMotor ()
virtual	**~dmRevDCMotor** ()
void	**setParameters** (Float torque_constant, Float back_EMF_constant, Float armature_resistance, Float rotor_inertia, Float coulomb_friction_constant, Float viscous_friction_constant, Float max_brush_drop, Float half_drop_value)
inline Float	**getRotorInertia** ()
void	**updateStiction** (Float *new_joint_vel)
Float	**computeTau** (Float sourceVoltage, Float extTau, Float joint_vel)

This currently the only concrete example of an actuator. It models a DC motor that drives a revolute joint. The loader function expects the configuration file to containing the following lines after the `Actuator_Type` line:

```
Motor_Torque_Constant              K_t
Motor_BackEMF_Constant             K_b
Motor_Armature_Resistance          R_a
Motor_Inertia                      J_m
Motor_Coulomb_Friction_Constant    u_c
Motor_Viscous_Friction_Constant    u_v
Motor_Max_Brush_Drop               V_max
Motor_Half_Drop_Value              V_half
```

This page has been automatically generated with DOC++ ©1995 by Roland Wunderling, Malte Zöckler

where all of these parameters have been adjusted (where necessary) for any gearing and are the values seen from the outside. Note that the `setParameters` function has been provided to set these parameters under programmer control. *I either need to add a getParameters function or I need to implement individual get/set functions for each parameter.*

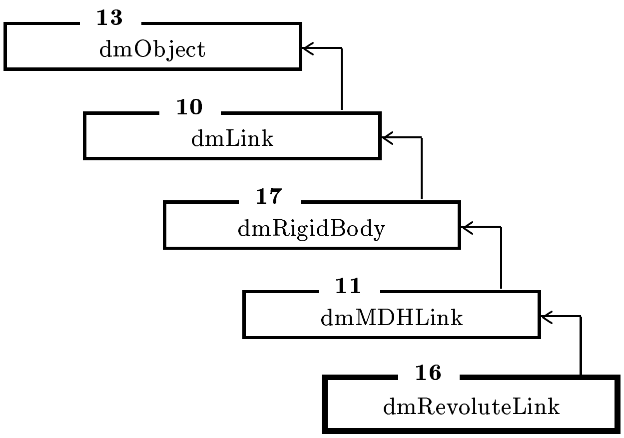
Note that with significant gearing a small rotor inertia (`Motor_Inertia`) can easily become a significant portion of a link's inertia. This value is automatically added to the link's spatial inertia matrix when this actuator is *attached* to the `dmRevoluteLink` object, and the `getRotorInertia` function provides the link access to this value.

The `computeTau` function simulates the motor dynamics. I recommend that you decipher the source code if you want to learn more, or consult Wrus Kristiansen Master's Thesis from the Naval Postgraduate School.

See also `dmActuator`.

16 class DM_DLL_API **dmRevoluteLink** : public dmMDHLink

Inheritance



Public Members

	dmRevoluteLink ()
virtual	**~dmRevoluteLink** ()
void	**scongtxToInboardIrefl** (SpatialTensor N, SpatialTensor I)
void	**draw** ()

This class is one of the two concrete `dmLink` classes derived from the `dmMDHLink` class. It implements the dynamics specific to a revolute (rotational) one degree of freedom joint.

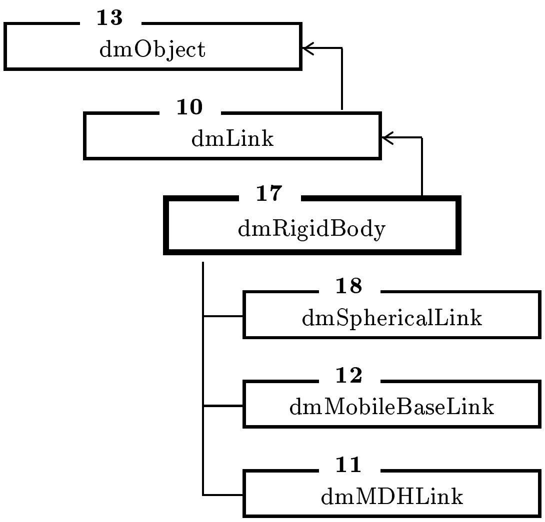
Not reference manual material: The primary purpose of this class is to implement the `scongToInboardIrefl` member function, a very efficient congruence transformation of the Articulated-Body inertia matrix that is transformed across the revolute joint. In this case, the third row and column of the matrix are zero and taking this into account results in significant computation savings (for all the gorey details, please read my dissertation or Robotics and Automation Journal article). The `dmMDHLink::ABBackwardDynamics[N]` functions call this function in the course of computing the dynamics. It should probably be a private/protected member function, but since it does not modify any member variables there is no harm in leaving it public.

See also: `dmMDHLink`, `dmLoadFile_dm`.

17

class DM_DLL_API **dmRigidBody** : public dmLink

Inheritance



Public Members

	dmRigidBody ()
virtual	**~dmRigidBody** ()
bool	**setInertiaParameters** (Float mass, CartesianTensor inertia, CartesianVector cg_pos)
void	**getInertiaParameters** (Float &mass, CartesianTensor inertia, CartesianVector cg_pos)
void	**setHydrodynamicParameters** (Float volume, SpatialTensor I_added_mass, CartesianVector cb_pos, int drag_axis, Float cyl_min, Float cyl_max, Float cyl_radius, Float C_d)
unsigned int	**getNumForces** ()
bool	**addForce** (dmForce *force)

int	**getForceIndex** (dmForce *force)
dmForce*	**getForce** (unsigned int index)
bool	**removeForce** (dmForce *force)
bool	**removeForce** (unsigned int index)
void	**getExternalForce** (SpatialVector force)
void	**setExternalForce** (SpatialVector force)
void	**computeBeta** (CartesianVector omega, SpatialVector beta)
void	**computeDrag** (SpatialVector v_rel, SpatialVector f_D)
void	**computeHydrodynamicBias** (dmABForKinStruct *val, SpatialVector beta_H)

This class is the repository for all the dynamic parameters for rigid bodies (links) that are needed for the AB dynamics algorithm. It is subclassed from `dmLink` and subclassed by all links that have degrees of freedom to be simulated. Remember: `dmLink` is a misleading name as it specifies the API needed by the AB algorithm and could also correspond to non-variable transformations like the `dmZScrewTxLink` class that is only used for kinematic manipulation (transformations), but do not have a physical manifestation.

The `setInertiaParameters` and `getInertiaParameters` functions is used set and retrieve the inertial parameters. The first argument specifies the mass, m, of the body, the second specifies the 3×3 inertia matrix, $\bar{\boldsymbol{I}}$, of the body with respect to the body's coordinate system, and the third specifies the position of the body's center of gravity, $\boldsymbol{p}_{cg}$, with respect to the body's coordinate system origin. Note that from these parameters, the `setInertiaParameters` function constructs the the spatial inertial matrix from the mass, center of gravity and inertia tensor and verifies that it is positive definite:

$$\mathbf{I} \quad = \quad \begin{bmatrix} \bar{\boldsymbol{I}} & m\,\tilde{\boldsymbol{p}}_{cg} \\ m\,\tilde{\boldsymbol{p}}_{cg}^T & m\mathbf{1}_3 \end{bmatrix}.$$

Note that $\bar{\boldsymbol{I}}$ is related to the body's inertia about its principle axes, $\bar{\boldsymbol{I}}_{cg}$, as follows:

$$\bar{\boldsymbol{I}} \quad = \quad \boldsymbol{R}^T \bar{\boldsymbol{I}}_{cg} \boldsymbol{R} + m\tilde{\boldsymbol{p}}_{cg}\tilde{\boldsymbol{p}}_{cg}^T,$$

where $\boldsymbol{R}$ specifies the rotation between the principle axes and the body's coordinate system.

Note that for underwater simulation, an entirely new library is made with the `HYDRODYNAMICS` environment variable defined. In this case, this class requires additional hydrodynamic parameters to compute the additional hydrodynamic effects. These are set with the `setHydrodynamicParameters` function. The `volume` parameter is the volume of fluid that this rigid body displaces. The `I_added_mass` parameter is a 6×6 positive semi-definite matrix corresponding to the added mass. The third parameter, `cb_pos` is the position of the center of bouyancy with the respect to the body's coordinate system. The remaining parameters are used to compute a rough approximation of a drag force. In this algorithm, the body must be approximated with an axis aligned cylinder. The `drag_axis` indicates which axis (0 for x axis, 1 for y, or 2 for z), `cyl_min` and `cyl_max` gives the coordinates along this axis which defines the extent of the approximating cylinder, and `cyl_radius` defines its radius. Finally, `C_d` defines the coefficient of drag.

This discussion requires the obvious note about units. DynaMechs makes no assumptions about length and mass units (only their consistency). It is required that once a convention is chosen (MKS, cgs, etc..), it is adhered to in the specification of ALL the parameters in all of the classes.

Motion of rigid bodies can be under the influence of a variety of external forces. Complaint forces (a function of the rigid body's state) can be implemented by subclassing from `dmForce` and "adding" instantiations of them to this class. The `dmContactModel` is an example of a derived force class. Any number of previously instantiated `dmForce` objects can be added to the `dmRigidBody` class using the `addForce` function. The number of force objects associated with this class is returned when a call to `getNumForces` is made.

Force objects are added to an internal list in order such that the first one is index 0, the second is index 1, and so forth. The current index of a force object in the list of associated objects is returned when a call to `getForceIndex` is made and the pointer to a `dmForce` object, in question, is provided. Alternatively a pointer to a force object is returned when a call to `getForce` is made with a given index. This function returns NULL if the index is out of range (0,..,num_forces-1).

Force objects can be removed with a call to either `removeForce` function, which will return true if the operation is successful. The force to be removed can be referenced either by its index or a pointer. If the index is out of range or the pointer doesn't match any of the current force objects in the list, or there was an internal failure, the function will return false. Note that when a force object is removed all indices for forces added later than the one in question will be renumbered (one lower).

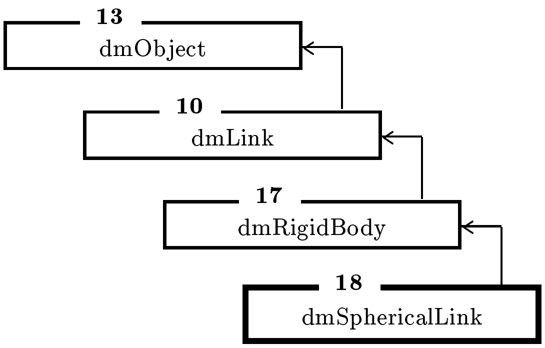
In addition to the force objects, this class also provides the functions, `setExternalForce` and `getExternalForce`, which can be used to directly specify an external force exerted on the rigid body. The parameter, `force`, is a spatial vector consisting of the moment and linear force expressed with respect to the body's coordinate system.

The `computeBeta`, and, in the case of hydrodynamic simulation, `computeDrag` and `computeHydrodynamicBias` are all functions called by the `ABForwardKinematics` functions of the various derived objects to compute state (position and velocity) dependent bias forces experience by the bodies.

See also `dmLink`, `dmForce`, `dmContactModel`, `dmLoadFile_dm`.

18 class DM_DLL_API **dmSphericalLink** : public dmRigidBody

Inheritance



Public Members

	dmSphericalLink ()
virtual	**~dmSphericalLink** ()
inline void	**setJointOffset** (CartesianVector pos)
inline int	**getNumDOFs** ()
void	**setState** (Float q[], Float qd[])
void	**getState** (Float q[], Float qd[])
void	**setJointLimits** (CartesianVector axis_limits, Float K_limit, Float B_limit)
void	**setJointFriction** (Float u_c)
inline void	**setJointInput** (Float joint_input[])
void	**rtxToInboard** (CartesianVector curr, CartesianVector prev)
void	**rtxFromInboard** (CartesianVector prev, CartesianVector curr)
void	**stxToInboard** (SpatialVector curr, SpatialVector prev)
void	**stxFromInboard** (SpatialVector prev, SpatialVector curr)
void	**rcongtxToInboardSym** (CartesianTensor Curr, CartesianTensor Prev)
void	**rcongtxToInboardGen** (CartesianTensor Curr, CartesianTensor Prev)

This page has been automatically generated with DOC++ ©1995 by Roland Wunderling, Malte Zöckler

void	**scongtxToInboardIrefl** (SpatialTensor N, SpatialTensor I)
void	**ABForwardKinematics** (Float q[], Float qd[], dmABForKinStruct *link_val_inboard, dmABForKinStruct *link_val_curr)
void	**ABBackwardDynamics** (dmABForKinStruct *link_val_curr, SpatialVector f_star_curr, SpatialTensor N_refl_curr, SpatialVector f_star_inboard, SpatialTensor N_refl_inboard)
void	**ABBackwardDynamicsN** (dmABForKinStruct *link_val_curr, SpatialVector f_star_inboard, SpatialTensor N_refl_inboard)
void	**ABForwardAccelerations** (SpatialVector a_inboard, SpatialVector a_curr, Float qd[], Float qdd[])
void	**draw** ()

This class is a concrete link class (previously called the BallnSocketLink class) that models a rigid body attached to an articulation with a ball joint (three rotational degrees of freedom). This class is derived from the `dmRigidBody` (for the dynamic parameters) and, through that, the `dmLink` base class (required by `dmSystem` for Articulated-Body simulation functions).

The kinematics (the point of motion) of these links can be defined with a single cartesian vector that specifies the constant offset from the origin of the previous coordinate system (expressed with respect to the previous coordinate system). This can be set by passing the desired offset to the `setJointOffset` member function. Note that currently the default orientation of this link's coordinate system in zero position (all joint angles set to zero) is parallel to the orientation of the previous coordinate system.

The orientation of this link with respect to the inboard coordinate system is specified with three Euler angles. *I am considering converting this to a quaternion just like I did to the* `dmRefMember` *class.* The values of these angles can be set and retrieved with the `setState` and `getState` member functions where the `q` parameter contains the angles ϕ, θ, ψ order. They are applied, however, in the following order from the inboard to this coordinate system:

ψ	angle in radians about the inboard z axis,
θ	angle in radians about the rotated y axis, and
ϕ	angle in radians about the twice rotated x axis (same as this link's x axis.

In dynamic simulation, all Euler angles suffer from a singularity when the middle angle rotates to the point where the inner and outer axes line up. *...which is why I am considering the conversion to a quaternion representation in the near future.*

The **getState** and **setState** functions also set and retrieve the link's angular velocity (in the **qd** parameter). For this link the velocity is the angular velocity of the link relative to the inboard coordinate system and expressed in this link's coordinate system (in radians/sec). Note that it is not the time derivatives of the Euler angles. Note that the size of both the **q** and **qd** arrays are dimension three, which is the value returned by the call to **getNumDOFs**.

Specifying joint limits for this type of joint still needs some further development. Currently, rudimentary joint limits can be specified with a call to **setJointLimits**. The first parameter, **axis_limits**, contains three maximum angles between the inboard and outboard coordinate axes in question (i.e., the first corresponds to the angle between the x-axes, the second y, and the third z). A zero value in any position corresponds to no restriction for the corresponding axis. If the limit is exceeded, a restorative torque equal to:

$$\tau \;=\; K_{limit}\,\Delta q,$$

where Δq is the amount by which the angle limit is exceeded and is applied about an axis normal to the axes in question. In addition the a dampening term is included that opposes the velocity along the other *two* axes as follows:

$$\tau \;=\; -\,B_{limit}\,\dot{q}.$$

where K_limit and B_limit correspond to the last two parameters of the **setJointLimits** member function.

The coulomb friction parameter, μ, is set with the **setJointFriction** function and models a dampening force:

$$\tau \;=\; -\,u\,\omega,$$

which is a 3D torque in a "direction" opposite the rotation (angular velocity). Finally, an external torque to be applied to link can be specified with the **setJointInput** function where the parameter is a 3D torque with respect the link's coordinate system.

The remainder of the functions are described in the **dmLink** reference pages and are implemented in this class for the specific case of ball joints. Note again that **getNumDOFs** returns 3. As such the joint variables **q**, **qd**, **qdd**, and **joint_input** are arrays of three elements.

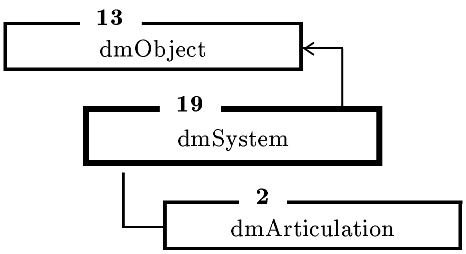
See also: **dmLink**, **dmRigidBody**, **dmLoadFile_dm**.

This page has been automatically generated with DOC++ © 1995 by Roland Wunderling, Malte Zöckler

19

class DM_DLL_API **dmSystem** : public dmObject

Inheritance



Public Members

	dmSystem ()
virtual	**~dmSystem** ()
virtual unsigned int	**getNumDOFs** ()
virtual void	**setState** (Float q[], Float qd[])
virtual void	**getState** (Float q[], Float qd[])
void	**makeJointVariable** (Float **joint_var)
int	**makeStateVariable** (Float **state_var)
void	**initSimVars** (Float *qy, Float *qdy)
void	**setRefSystem** (Quaternion quat, CartesianVector pos)
inline void	**rtxFromICS** (CartesianVector p_ICS, CartesianVector p_ref)
inline void	**rtxToICS** (CartesianVector p_ref, CartesianVector p_ICS)
bool	**forwardKinematics** (unsigned int link_index, HomogeneousTransformationMatrix mat)
virtual bool	**forwardKinematics** (unsigned int link_index, dmABForKinStruct *fk)
virtual void	**ABDynamics** (Float *qy, Float *qdy)

The `dmSystem` is the abstract base class for the top-level in the DynaMechs object hierarchy. Subclasses are derived from it which implement the specifics need for the various types of topologies supported by this library. Currently only tree topologies are implemented in the `dmArticulation` subclass, but soon a class implementing a closed chain algorithm will be added). This class contains all of the functions (pure virtual and otherwise) needed to perform the dynamics computation which is only the derivative computation step of a complete dynamic simulation. To perform dynamic simulation, a complete system is put together by assigning links the subclasses (see the `dmArticulation` class), and is then assigned to one of the `dmIntegrator` objects.

The function, `makeJointVariable`, is a potentially obsolete holdover from previous versions of DynaMechs. That said, this creates a one dimensional vector of `Floats` that is large enough to hold all of a general joint variable (one of position, velocity, acceleration, or joint inputs). The order in which they are packed into this vector is the same as the order in which they were added to the subclass.

A related function, `makeStateVariable`, is now used by the `dmIntegrator` (and derived) classes to allocate a one-dimensional array that is large enough to hold the state (position *and* velocity) variables for all of links of all of the articulations. This is also used to allocate the derivative of state (velocity and acceleration) variables since they are the same size. The `initSimVars` function will initialize the state and derivative of state vectors with the current state of the system. The parameters, `qy` and `qdy`, are the state vectors that have been allocated with calls to `makeStateVariable`.

IMPORTANT: All necessary calls to `makeStateVariable` and `initSimVars` are done automatically by the `dmIntegrator` class when a `dmSystem` object is assigned to it. The `dmIntegrator` class cannot detect subsequent additions of links; therefore, the entire system must be built before it is assigned to the `dmIntegrator` object and no changes can be made after this point.

Specification of an inertial coordinate system other that the origin can be specified by calling `setRefSystem` and passing the position and orientation (relative to the origin) of the new coordinate system. *I am not really sure if this necessary from an efficiency standpoint, but it makes it much more flexible in specifying systems* The functions, `rtxFromICS` and `rtxToICS`, are used to rotate free vectors from one system to the other.

The `forwardKinematics` function computes the homogeneous transformation matrix (4x4) describing the position (last column) and orientation (upper left 3x3 submatrix) of a particular link identified by articulation and link indices with respect to the inertia coordinate system. The function returns `true` if the indices were valid and the result is in `mat`; otherwise, it returns `false` and the result in `mat` is undefined. If the homogeneous transformation matrix of the reference member is desired the link index is to be set to -1.

The `ABDynamics` function is the heart of the DynaMechs library and performs the Articulated Body (AB) dynamics computation for the multibody system. It takes, as input, the state vector `qy` (for all of the links) and computes the derivative of state vector, `qdy`. This is the derivative function that the `dmIntegrator` object uses to perform the numerical integration.

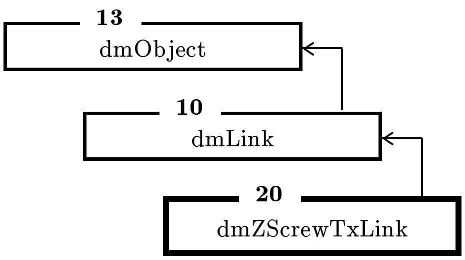
See also: `dmArticulation, dmIntegrator`.

This page has been automatically generated with DOC++ © 1995 by Roland Wunderling, Malte Zöckler

20

class DM_DLL_API **dmZScrewTxLink** : public dmLink

Inheritance



Public Members

	dmZScrewTxLink (Float d, Float theta)
virtual	**~dmZScrewTxLink** ()
inline int	**getNumDOFs** ()
inline void	**setState** (Float [], Float [])
inline void	**getState** (Float [], Float [])
inline void	**setJointInput** (Float [])
void	**rtxToInboard** (CartesianVector p_0, CartesianVector p_inboard)
void	**rtxFromInboard** (CartesianVector p_inboard, CartesianVector p_0)
void	**stxToInboard** (SpatialVector p_0, SpatialVector p_inboard)
void	**stxFromInboard** (SpatialVector p_inboard, SpatialVector p_0)
void	**rcongtxToInboardSym** (CartesianTensor M_0, CartesianTensor M_inboard)
void	**rcongtxToInboardGen** (CartesianTensor M_0, CartesianTensor M_inboard)
void	**scongtxToInboardIrefl** (SpatialTensor M_0, SpatialTensor M_inboard)
void	**ABForwardKinematics** (Float [], Float [], dmABForKinStruct *link_val_inboard, dmABForKinStruct *link_val_curr)

This page has been automatically generated with DOC++ ©1995 by Roland Wunderling, Malte Zöckler

void **ABBackwardDynamics** (dmABForKinStruct *link_val_curr, SpatialVector f_star_curr, SpatialTensor I_refl_curr, SpatialVector f_star_inboard, SpatialTensor I_refl_inboard)

void **ABBackwardDynamicsN** (dmABForKinStruct *link_val_curr, SpatialVector f_star_inboard, SpatialTensor I_refl_inboard)

void **ABForwardAccelerations** (SpatialVector a_inboard, SpatialVector a_curr, Float [], Float [])

virtual void **draw** ()

This is an unusual concrete link class because while it inherits the AB simulation algorithm functionality from the `dmLink` class, it does not represent a rigid body so it does not contain any dynamic parameters (that is, it does not inherit from the `dmRigidBody` class). The purpose of this class is to implement efficient branching within tree stuctures by modelling a constant z axial screw transformation between coordinate systems as recommended in the kinematics literature by Khalil and Kleinfinger.

The z axial screw is defined by two parameters: a translation along and a rotation about the z axis. Since they are constant, these parameters can only be set with arguments to the constructor where `d` is the translation amount and `theta` is the rotation in radians.

The remainder of the functions are described in the `dmLink` reference pages and are implemented in this class for the specific case of z axial screws. Note that `getNumDOFs` returns 0. As such the joint variables `q`, `qd`, `qdd`, and `joint_input` are empty arrays.

See also `dmLink`.

21

(dllimport) DMU_DLL_API dmSystem* **dmuLoadFile_dm** (char *filename)

Load a configuration file

Load a configuration file. A configuration file reader, `dmLoadFile_dm` is being supplied with the dmutils library that can be used to instantiate and intialize all types of objects.

Comments begin with the hash (#) character and continues to the end of the line. Blocks are delimited by braces ({}) and are preceded by a label denoting the type of block.

The configuration files begin with a comment line specifying the format of its contents. Currently three versions are supported (two for backward compatibility). The three valid forms of this comment line are as follows:

```
# DynaMechs V 2.0.3 ascii
# DynaMechs V 2.1 ascii
# DynaMechs V 3.0 ascii
```

The last one corresponds to the most recent version of the library and allows the most flexible specifications of multi-body systems. The other two are provided for backward compatibility with previous version of this library.

Format for Versions 2.0.3 and 2.1

These configuration formats begin with a specification of 3D models to represent various objects indexed in later blocks:

```
Graphics_Models {
        Number_Graphics_Models  2
        "models/obj_cube_center.xan"
        "models/obj_cylinder_x.xan"
}
```

The top-level block is the System block which contains one reference member block and any number of articulation blocks:

```
System {
    Name                             "some_label"    # version 2.1 format only
    (reference member info)

    (articulation 0 info)
         :
```

```
        :
    (articulation n info)
}
```

Reference member blocks can be one of two types:

```
StaticRefMember {
    Name                            "some_label"    # version 2.1 format only
    Graphics_Model_Index            i

    Position                        x   y   z
    Orientation_Quat                a_x a_y a_z  s
}

DynamicRefMember {
    Name                            "some_label"    # version 2.1 format only
    Graphics_Model_Index            i

    (rigid body parameters)

    Position                        x   y   z
    Orientation_Quat                a_x a_y a_z  s
    Velocity                        w_x w_y w_z  v_x v_y v_z
}
```

where specifications for rigid body parameters can be found below and the `Velocity` parameters specify the initial spatial (angular and linear) velocity with respect to the inertial coordinate system.

Articulation blocks are indicated by the following:

```
Articulation {
    Name                            "some_label"    # version 2.1 format only
    (links and branches)
}
```

A simple serial chain of links is organized as follows:

```
Articulation {
    Name                            "some_label"    # version 2.1 format only
    (link 0)
    (link 1)
       :
       :
    (link n)
}
```

This page has been automatically generated with DOC++ ©1995 by Roland Wunderling, Malte Zöckler

A branch after link 1 is organized as follows

```
Articulation {
    Name                              "some_label"    # version 2.1 format only
    (link 0)
    (link 1)
    Branch {
        (link 2)
        (link 3)
    }
    (link 4)
       :
       :
    (links n)
}
```

which has the following topology (note the first link of any branch is connected to the reference member, r):

```
 r - 0 - 1 - 2 - 3
          \
            4 - ... - n
```

Also note that branch can occur anywhere after the first link (a branch before the first link is the same as a different articulation. Branches can also be nested. This structure supports all tree topologies.

See the section on link types for specification possible ones

Format for version 3.0

This format differs from the previous versions in a number of significant ways:

- The Articulation class has been merged with the System class (i.e., a system consists of only one articulation)
- Reference members are no longer used in 3.0. A static reference member specification (basically a constant position and orientation) has been moved to the specification of the System/Articulation. A dynamic reference member is now specified as a MobileBaseLink as the first link in the System.
- The initial list of graphics models has been eliminated and instead of Graphics_Model_Indices within the specifications of the links, the filename is provided in the Graphics_Model field.

The top-level block is the System block which contains all of the links (rigid bodies and transforms) for the multibody system it contains. The label for this block depends on the type of system it describes. Currently only the Articulation type (describing a tree structure) is supported and is specified as follows:

```
Articulation {
    Name                              "some_label"
```

This page has been automatically generated with DOC++ ©1995 by Roland Wunderling, Malte Zöckler

```
    Graphics_Model                  "some_model_file"
    Position                        x   y   z
    Orientation_Quat                a_x a_y a_z  s

    (link and branches)
}
```

The Name is a label given to the dmArticulation object. A graphics model can be associated with the base by specifying a valid file in the Graphics_Model field. An empty field ("") corresponds to no model. The Position and Orientation_Quat specifies the position and orientation of the inertial coordinate system relative to which the system is simulated.

The Articulation block contains any combination of branches an links to describe any tree structure. Some examples of these are given as follows.

A simple serial chain of links is organized as follows:

```
Articulation {
    Name                            "some_label"
    Graphics_Model                  "some_model_file"
    Position                        x   y   z
    Orientation_Quat                a_x a_y a_z  s

    (link 0)
    (link 1)
       :
       :
    (link n)
}
```

A Branch can occur anywhere (even before any links). Branches that occur at the beginning can be used to describe multiple systems:

```
Articulation {
    Name                            "some_label"
    Graphics_Model                  "some_model_file"
    Position                        x   y   z
    Orientation_Quat                a_x a_y a_z  s

    Branch {
        (link 0)
        (link 1)
    }
    Branch {
        (link 2)
        (link 3)
```

This page has been automatically generated with DOC++ © 1995 by Roland Wunderling, Malte Zöckler

```
    }
    (link 4)
       :
       :
    (links n)
}
```

which has the following topology (note the first link of any branch is connected to the System's inertial coordinate system, r) consisting of three distinct serial chains:

```
  r - 0 - 1
  r - 2 - 3
  r - 4 - ... - n
```

A Branch after a link is used to connect two or more links to the same one:

```
Articulation {
    Name                                 "some_label"
    Graphics_Model                       "some_model_file"
    Position                             x   y   z
    Orientation_Quat                     a_x a_y a_z  s

    (link 0)
    (link 1)
    Branch {
        (link 2)
        (link 3)
    }
    (link 4)
       :
       :
    (links n)
}
```

which has the following topology:

```
  r - 0 - 1 - 2 - 3
          \
            4 - ... - n
```

Branches can also be nested. Together this structure is capable of specifying all tree topologies.

Link types for all versions.

The library supports a number of "link" types that could correspond to a constant transformation (with no dynamic properties), or a rigid body connected to the previous link through a "joint" with some number of degrees of freedom. The possible link types supported in the various versions are given below. Tokens specific to a subset of the versions are given in the comments following the pertinent lines.

A constant screw transformation about the z axis is given as follows

```
ZScrewTxLink {
    Name                              "some_label"      # v 2.1 and 3.0
    ZScrew_Parameters                 d  theta
}
```

A full six DOF "link" has been introduced in Version 3.0 which takes the place of the DynamicRefMember (separate reference member specification has been deprecated). Use of this link only makes sense as the first link in a particular tree. Its state is specified only with respect to the System's inertial reference system so placing it anywhere else in the configuration essentially starts the topology over again:

```
MobilBaseLink {
    Name                              "some_label"      # v 2.1 and 3.0
    Graphics_Model                    "model_filename"  # v 3.0
    Graphics_Model_Index              i                 # v 2.0.3 and 2.1

    (rigid body parameters)

    Position                          x   y   z
    Orientation_Quat                  a_x a_y a_z  s
    Velocity                          w_x w_y w_z  v_x v_y v_z
}
```

where specifications for rigid body parameters can be found below and the `Velocity` parameters specify the initial spatial (angular and linear) velocity with respect to the inertial coordinate system.

A three DOF link (three orientation DOFs) is used to simulate a ball-in-socket joint using Euler angles:

```
SphericalLink {
    Name                              "some_label"      # v 2.1 and 3.0
    Graphics_Model                    "model_filename"  # v 3.0
    Graphics_Model_Index              i                 # v 2.0.3 and 2.1

    (rigid body parameters)

    Position_From_Inboard_Link        x y z
    Initial_Joint_Angles              phi theta psi
    Initial_Angular_Velocity          w_x w_y w_z
    Axes_Limits                       phi theta psi
    Joint_Limit_Spring_Constant       K_limit
    Joint_Limit_Damper_Constant       B_limit
    Joint_Friction                    u
}
```

This page has been automatically generated with DOC++ ©1995 by Roland Wunderling, Malte Zöckler

A one DOF translational link is specified as follows:

```
PrismaticLink {
    Name                            "some_label"      # v 2.1 and 3.0
    Graphics_Model                  "model_filename"  # v 3.0
    Graphics_Model_Index            i                 # v 2.0.3 and 2.1

    (rigid body parameters)

    MDH_Parameters                  a  alpha  d  theta
    Initial_Joint_Velocity          qd
    Joint_Limits                    q_min q_max
    Joint_Limit_Spring_Constant     K_limit
    Joint_Limit_Damper_Constant     B_limit

    Joint_Friction                  u
}
```

A one DOF rotational link is specified two different ways (with and without motor model). The version without a motor is specified as follows:

```
RevoluteLink {
    Name                            "some_label"      # v 2.1 and 3.0
    Graphics_Model                  "model_filename"  # v 3.0
    Graphics_Model_Index            i                 # v 2.0.3 and 2.1

    (rigid body parameters)

    MDH_Parameters                  a  alpha  d  theta
    Initial_Joint_Velocity          qd
    Joint_Limits                    q_min q_max
    Joint_Limit_Spring_Constant     K_limit
    Joint_Limit_Damper_Constant     B_limit

    Actuator_Type                   0
    Joint_Friction                  u
}
```

The version with a motor is specified as follows:

```
RevoluteLink {
    Name                            "some_label"      # v 2.1 and 3.0
    Graphics_Model                  "model_filename"  # v 3.0
    Graphics_Model_Index            i                 # v 2.0.3 and 2.1
```

```
    (rigid body parameters)

    MDH_Parameters                    a  alpha  d  theta
    Initial_Joint_Velocity            qd
    Joint_Limits                      q_min q_max
    Joint_Limit_Spring_Constant       K_limit
    Joint_Limit_Damper_Constant       B_limit

    Actuator_Type                     1

    Motor_Torque_Constant             K_t
    Motor_BackEMF_Constant            K_b
    Motor_Armature_Resistance         R_a
    Motor_Inertia                     J_m
    Motor_Coulomb_Friction_Constant   u_c
    Motor_Viscous_Friction_Constant   u_v
    Motor_Max_Brush_Drop              V_max
    Motor_Half_Drop_Value             V_half
}
```

The following is the order of the rigid body parameters:

```
    Mass                              m
    Inertia                           (3x3 matrix - 9 floats)
    Center_of_Gravity                 cg_x cg_y cg_z

    (contact model parameters)
```

In the case of hydrodynamic simulation, additional parameters are also required resulting the following alternate form:

```
    Mass                              m
    Inertia                           (3x3 matrix - 9 floats)
    Center_of_Gravity                 cg_x cg_y cg_z

    Volume                            V
    Added_Mass                        (6x6 matrix - 36 floats)
    Center_of_Bouyancy                cb_x cb_y cb_z
    Drag_Axis                         i (0=x, 1=y, 2=z)
    Cylinder_Bounds                   x0 xf
    Cylinder_Radius                   r
    Drag_Coefficient                  C_d

    (contact model parameters)
```

The contact model parameters consist of the following:

This page has been automatically generated with DOC++ ©1995 by Roland Wunderling, Malte Zöckler

```
Number_of_Contact_Points   n
Contact_Locations          x_1 y_1 z_1
                                :
                           x_n y_n z_n
```

Note that the `Contact_Locations` values are not needed if the `Number_of_Contact_Points` is zero.

This page has been automatically generated with DOC++ ©1995 by Roland Wunderling, Malte Zöckler

22

DMU_DLL_API dmEnvironment* **dmuLoadFile_env** (char *filename)

Load an environment configuration file

23

DMU_DLL_API dmObject* **dmuFindObject** (const char *name, dmArticulation *system)

Find a dmObject with the given name

Class Graph

This page has been automatically generated with DOC++ ©1995 by Roland Wunderling, Malte Zöckler