Modeling MBS in Chrono

Coordinate transformations

ChVector

$$\mathbf{p} = \{p_x, p_y, p_z\}$$

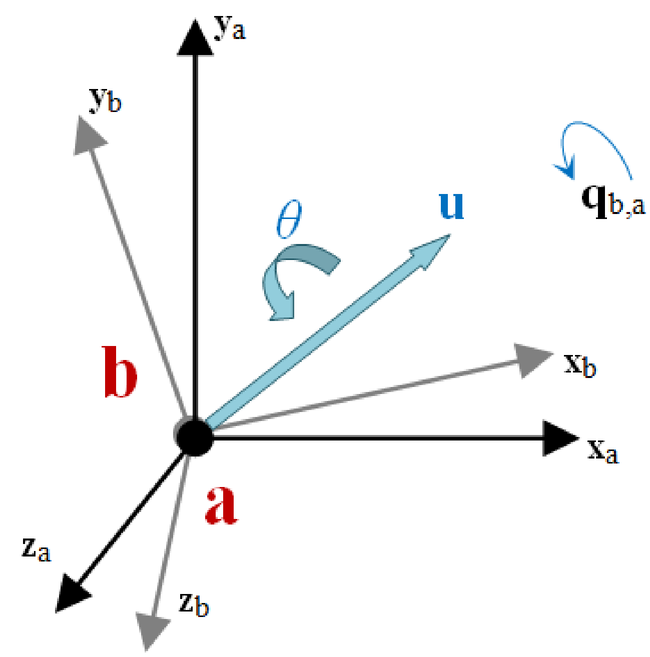
```
ChVector<double> v1(2,3,4);     /// create a vector with given x,y,z ‘double’ components
ChVector<float> v2(4,1,2);      /// create a vector with given x,y,z ‘float’ components
ChVector<>      v3();           /// create a 0,0,0, vector. The <> defaults to ‘double’
ChVector<>      v4(v1 + v2);    /// create a vector by copying another (a result from +)
v3 = v1 + v2;                   /// vector operators: +, -
v3 += v1;                       /// in-place operators
v3 = v2 * 0.003;                /// vector product by scalar
v3.Normalize();                 /// many member functions…
v3 = v1 % v2;                   /// Operator for cross product: A%B means vector cross-product AxB
double val = v1 ^ v2;           /// Operator for inner product (scalar product)
```

ChQuaternion

- Used to represent rotations
- Alternative to 3x3 matrices ChMatrix33<>

```
double theta = 30 * CH_C_DEG_TO_RAD;
ChVector<> u(0.3,0.4,0.1);
u.Normalize()
ChQuaternion<> q;
q = Q_from_AngAxis(theta, u);
```

$$q = \begin{bmatrix} \cos(\theta/2) \\ u_x \sin(\theta/2) \\ u_y \sin(\theta/2) \\ u_z \sin(\theta/2) \end{bmatrix}$$



Coordinate transformations

```
v2 = r + q.Rotate(v1);  /// use Rotate() to rotate a vector

qa = qb * qc;           /// concatenate two rotations, first qc, followed by qb
qa.Rotate(mvect1);

qa = qc >> qb;          /// concatenate two rotations, first qc, followed by qb
qa.Rotate(mvect1);
```

ChCoordsys and ChFrame

`ChCoordsys<>`

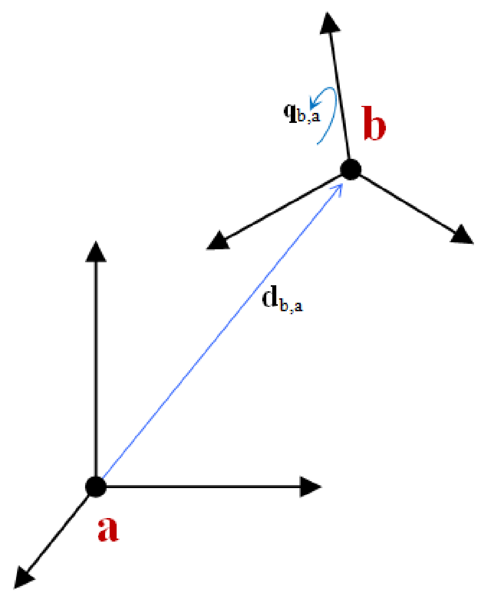
- represents a translation and a rotation
- rotation is a quaternion

$$\{d_{b,a}, q_{b,a}\}$$

`ChFrame<>`

- a more ‘powerful’ version of ChCoordsys
- contains also a ChMatrix33<> to speedup some formulas

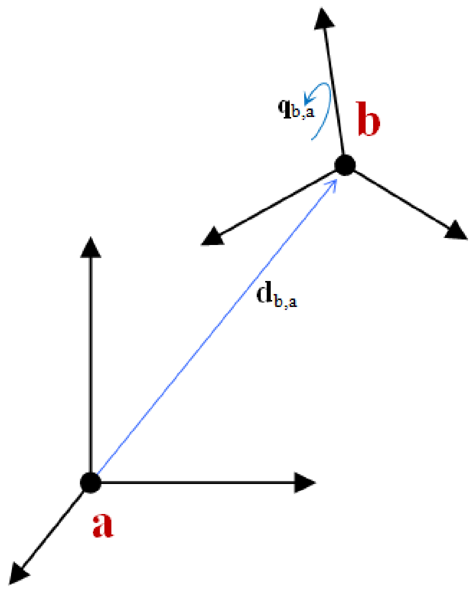
$$\{d_{b,a}, R(q_{b,a})\}$$



ChFrame constructors

Example of initialization:

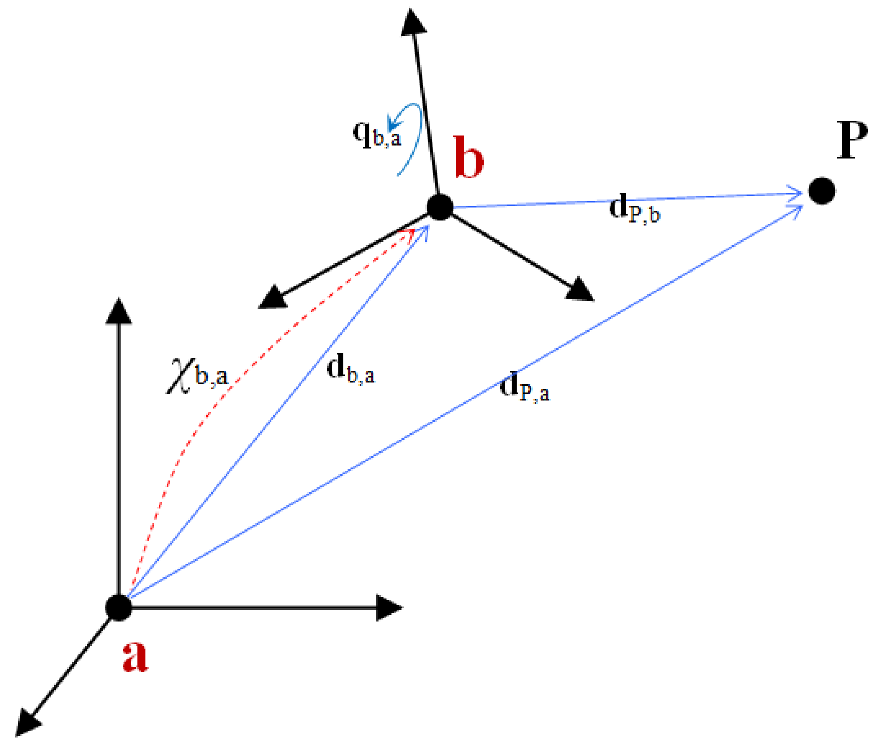
```
ChFrame<> Xa;           // build default frame: zero translation, no rotation

ChFrame<> Xb(va, qa);   // build from given translation va
                        // and rotation quaternion qa

ChFrame<> Xc(csys);     // build from a given ChCoordys<>

ChFrame<> Xd(va, tetha, u); // build from translation va,
                            // rotation theta about axis u
```



ChFrame operations

- ChFrame<> can transform points in space
- Two alternative options for syntax:
  - ***** operator: **RIGHT-TO-LEFT** transformation
  - **>>** operator: **LEFT-TO-RIGHT** transformation

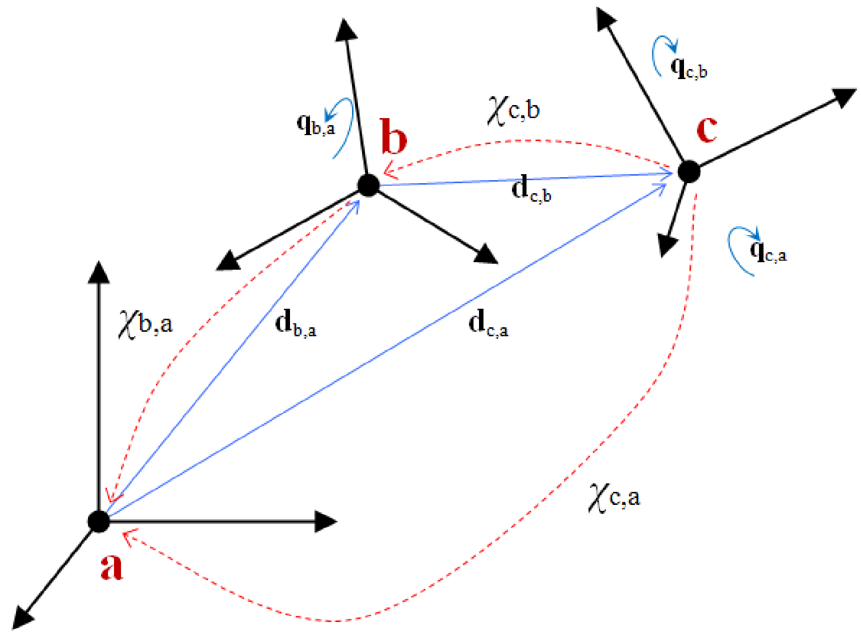
```
ChVector<> d_Paa, d_Pbb;
ChFrame<>  X_ba;
...
d_Paa = X_ba * d_Pbb;

d_Paa = d_Pbb >> X_ba;
```



ChFrame operations

- ChFrame can also be transformed
- Build sequence of transformations

```
ChFrame<> X_ba, X_cb, X_ca;
...
X_ca = X_ba * X_cb;
X_ca = X_cb >> X_ba;
```



ChFrameMoving

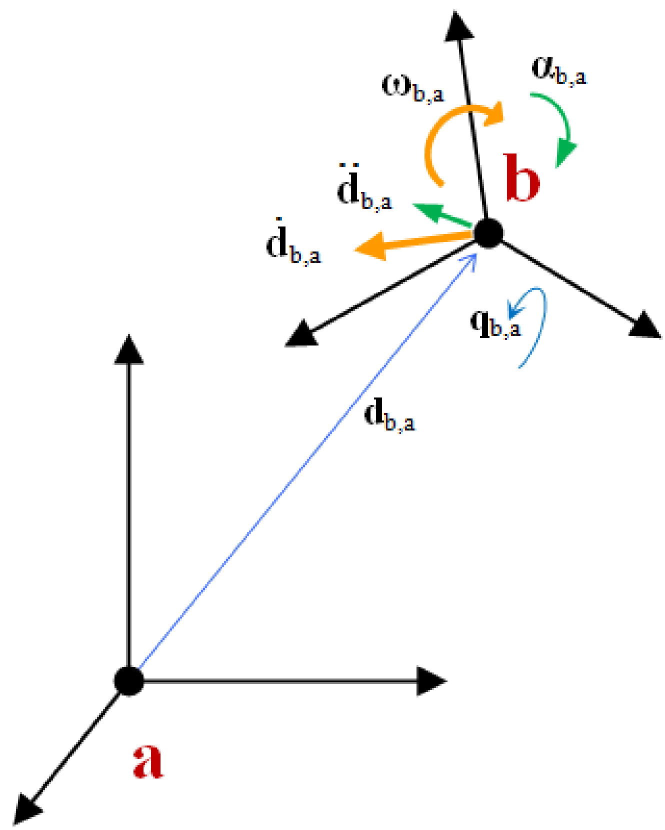
- Inherits ChFrame<> functionality

- Adds information on velocity and acceleration:

$$\mathbf{X} = \{\mathbf{d}, \mathbf{q}, \dot{\mathbf{d}}, \dot{\mathbf{q}}, \ddot{\mathbf{d}}, \ddot{\mathbf{q}}\}$$

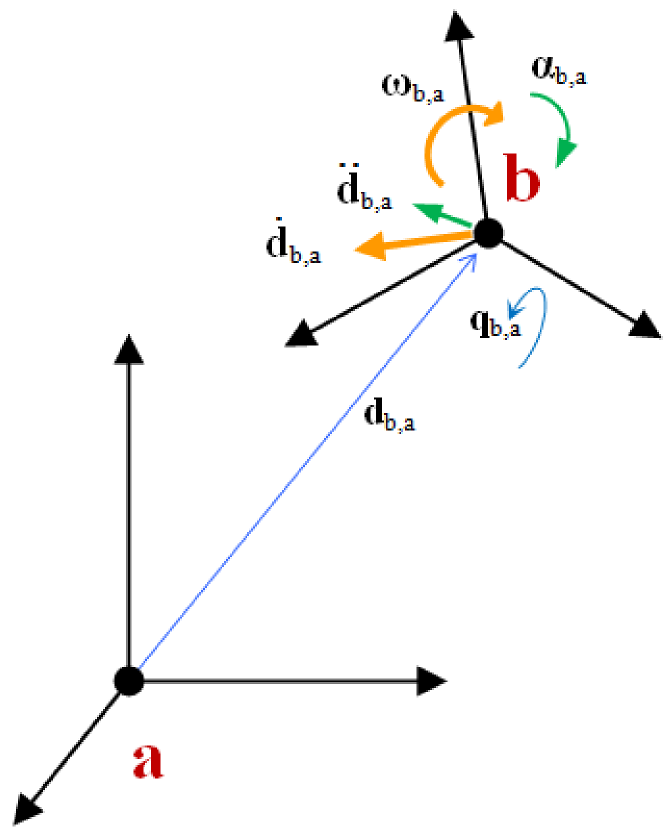
- Alternative: angular velocity and acceleration instead of q derivatives:

$$\mathbf{X} = \{\mathbf{d}, \mathbf{q}, \dot{\mathbf{d}}, \boldsymbol{\omega}, \ddot{\mathbf{d}}, \boldsymbol{\alpha}\}$$



ChFrameMoving operations

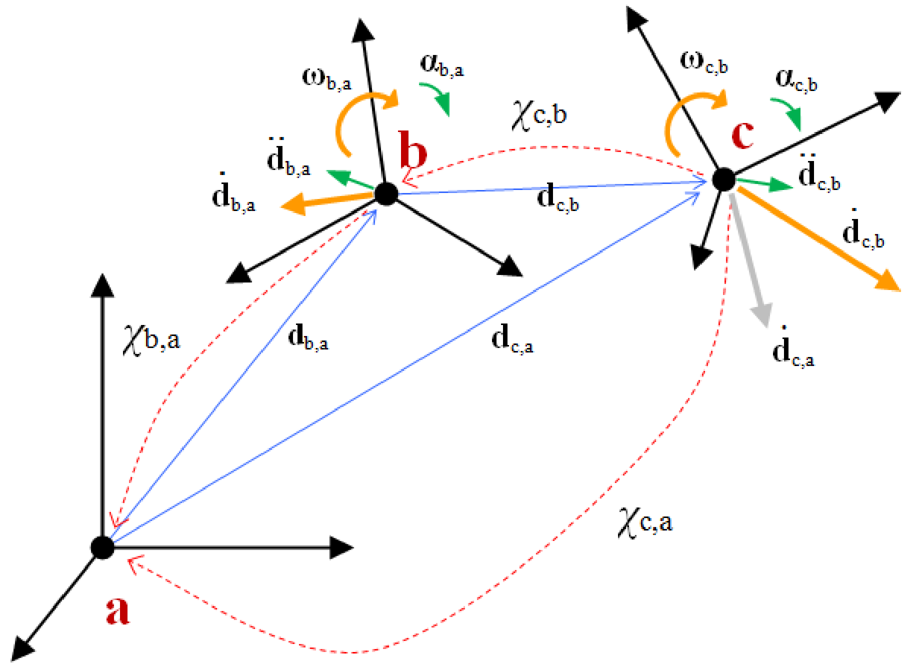
```
ChFrameMoving<> X_ba;

X_ba.SetPos(ChVector<>(2,3,5));

X_ba.SetRot(quaternion);

// set velocity

X_ba.SetPos_dt(ChVector<>(100,20,53));

X_ba.SetWvel_loc(ChVector<>(0,40,0)); // W in local frame, or..
X_ba.SetWvel_par(ChVector<>(0,40,0)); // W in parent frame

// set acceleration

X_ba.SetPos_dtdt(ChVector<>(13,16,22));

X_ba.SetWacc_loc(ChVector<>(80,50,0)); // a in local frame, or..
X_ba.SetWacc_par(ChVector<>(80,50,0)); // a in parent frame
```



ChFrameMoving operations

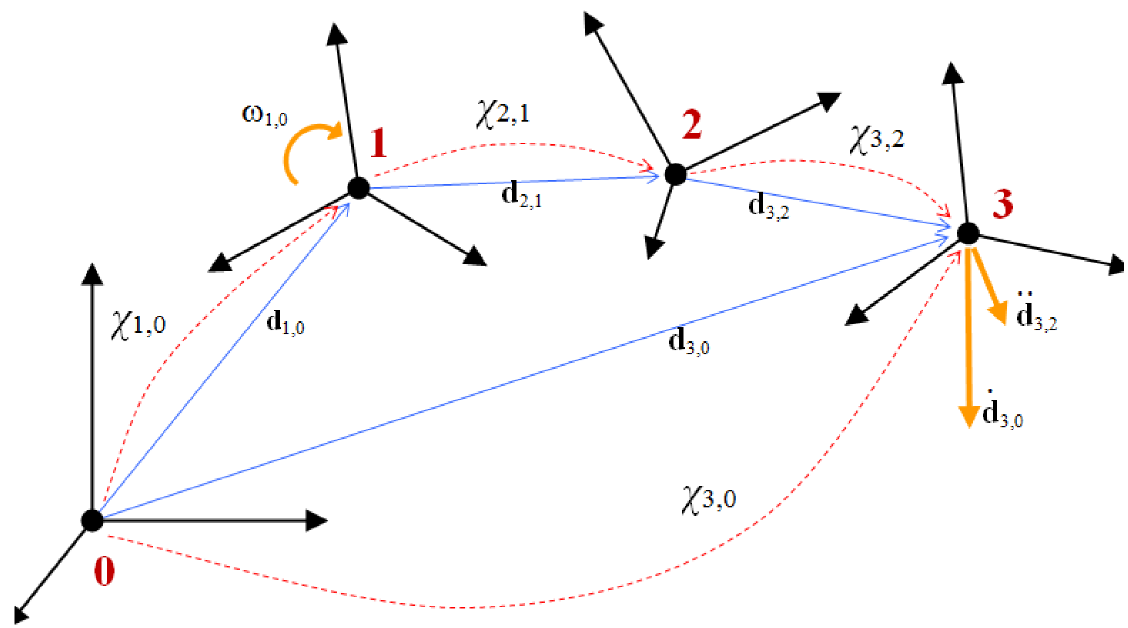
- ChFrameMoving (and ChVector, ChFrame) can be transformed
- Same * or >> operators as for ChFrame<>
- But speeds and accelerations are also automatically transformed!

```
ChFrameMoving<> X_ba, X_cb, X_ca;
...
X_ca = X_ba * X_cb; // otherwise…
X_ca = X_cb >> X_ba;
ChVector<> w_ca = X_ca.GetWvel_rel();
```



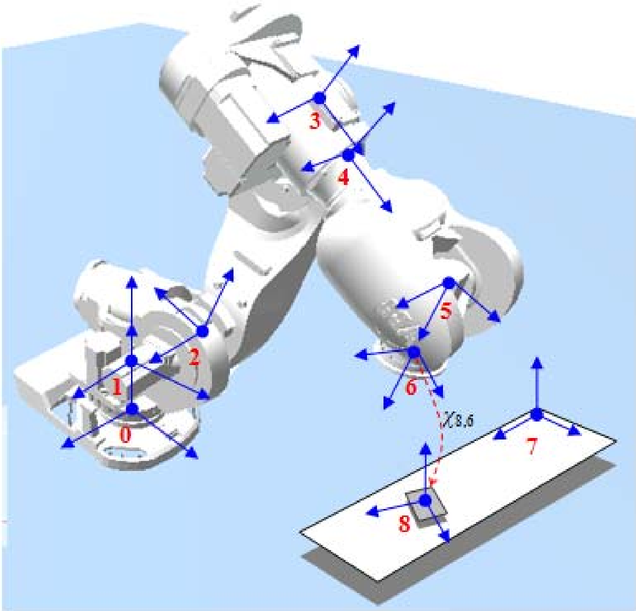
Concatenation of transforms

```
ChFrameMoving<> X_10, X_21, X_32, X_30;
...
X_30 = X_32 >> X_21 >> X_10;

ChVector<> a_03 = X_30.GetPos_dtdt();
```



Inverse transforms

- The GetInverse() and Inverse() functions:

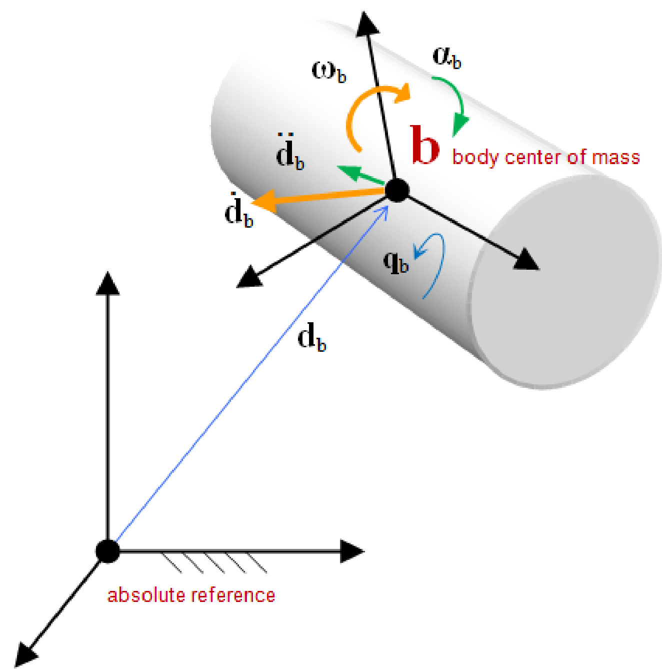



```
ChFrameMoving<> X_10, X_21, X_32, X_43, X_54, X_65, X_70, X_87, X_86;
...
// How to compute X_86 knowing all others?
// Start from two equivalent expressions of X_80:
//   X_86>>X_65>>X_54>>X_43>>X_32>>X_21>>X_10 = X_87>>X_70;
// also:
//   X_86>>(X_65>>X_54>>X_43>>X_32>>X_21>>X_10) = X_87>>X_70;
// Post multiply both sides by inverse of (...) and get:

X_86 = X_87 >> X_70 >> (X_65 >> X_54 >> X_43 >> X_32 >> X_21 >> X_10).GetInverse();
```

Rigid body

ChBody

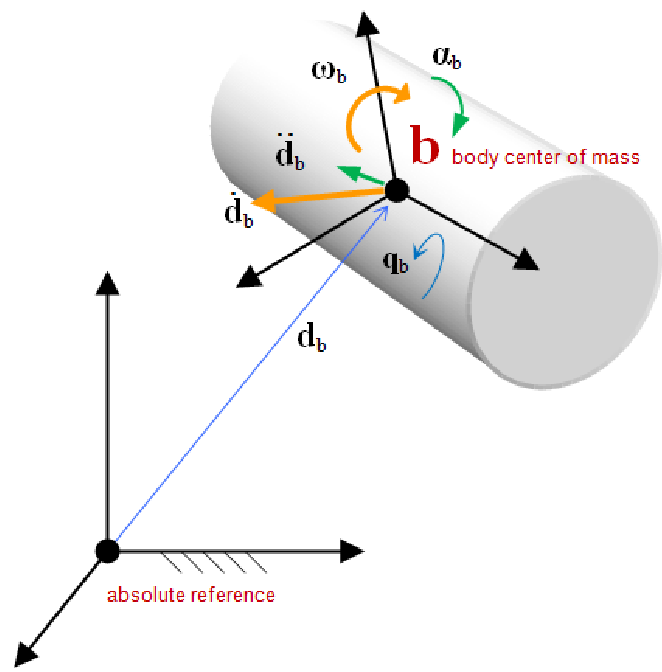
- Rigid bodies inherit ChFrameMoving features (position, rotation, velocity, acceleration, etc.)
- The position, speed, acceleration are those of the center of mass (COG)
- They contain a mass and a tensor of inertia
- They can be connected by ChLink constraints
- They can participate to collisions



ChBody construction

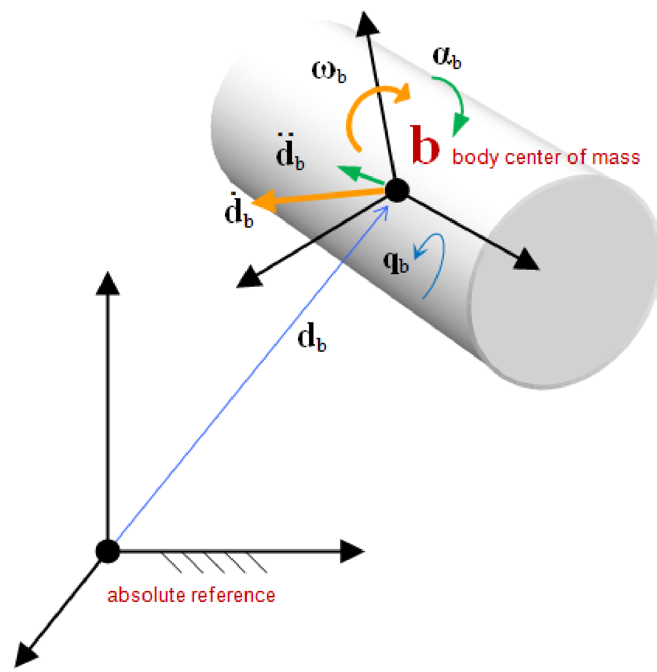
Important steps for each rigid body:

1. Create the ChBody
2. Set position and mass properties
3. Add the body to a ChSystem
4. Optional: add collision shapes
5. Optional: add visualization assets



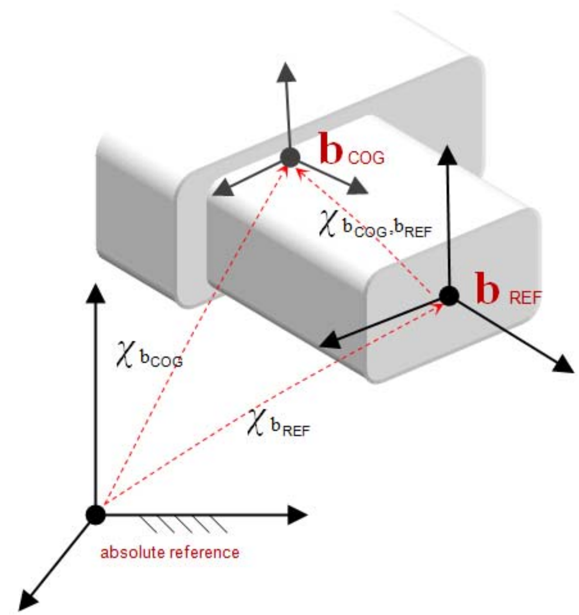
ChBody construction example

```
// Create a body – use shared pointer!

auto body_b = std::make_shared<ChBody>();

// Set initial position & speed of the COG of body,
// using the same syntax used for ChFrameMoving

body_b->SetPos(ChVector<>(0.2, 0.4, 2));
body_b->SetPos_dt(ChVector<>(0.1, 0, 0));

// Set mass and inertia tensor

body_b->SetMass(10);
body_b->SetInertiaXX(ChVector<>(4, 4, 4));

// If body is fixed to ground, use this:

body_b->SetBodyFixed(true);

// Finally do not forget this

my_system.Add(body_b);
```



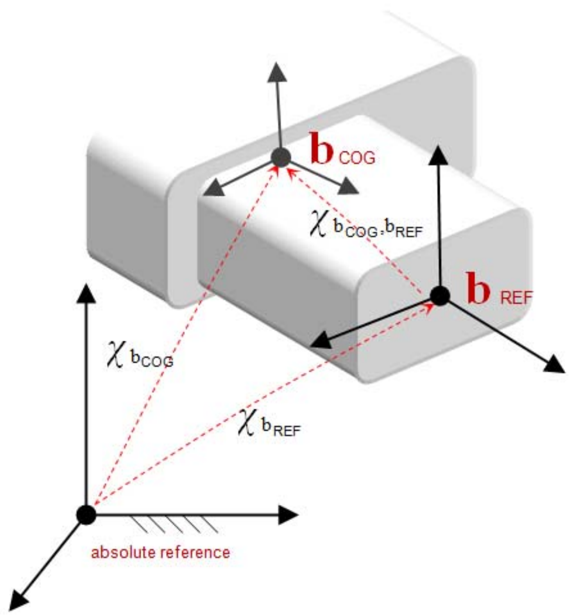
ChBodyAuxRef

- Inherited from ChBody
- Used when the COG is not practical as a main reference for the body, and another reference is preferred, e.g. from a CAD system
  - Use an auxiliary REF frame.
- The REF frame is used for
  - Collision chapes
  - Visualization shapes



ChBodyAuxRef construction example

```
 // Create a body with auxiliary reference frame

auto body_b = std::make_shared<ChBodyAuxRef>();

// Set position of COG respect to reference

body_b->SetFrame_COG_to_REF(X_bcogref);

// Set position of reference in absolute space

body_b->SetFrame_REF_to_abs(X_bref);

// Position of COG in absolute space is simply body_b
// e.g. body_b->GetPos(), body_b->GetRot(), etc.
```



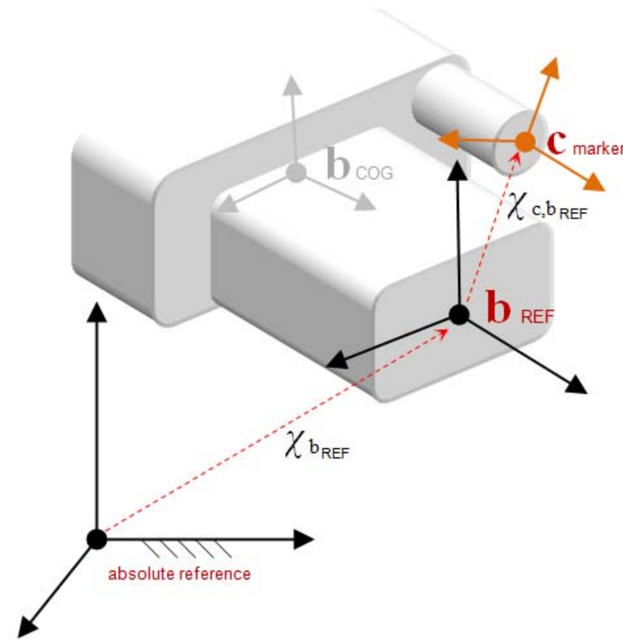
Markers, collision shapes, visualization assets

Markers: ChMarker

- Inherit the features of ChFrameMoving.
- Used to get position/speed/acceleration of a given reference attached to a ChBody
- Used to build many ChLink constraints

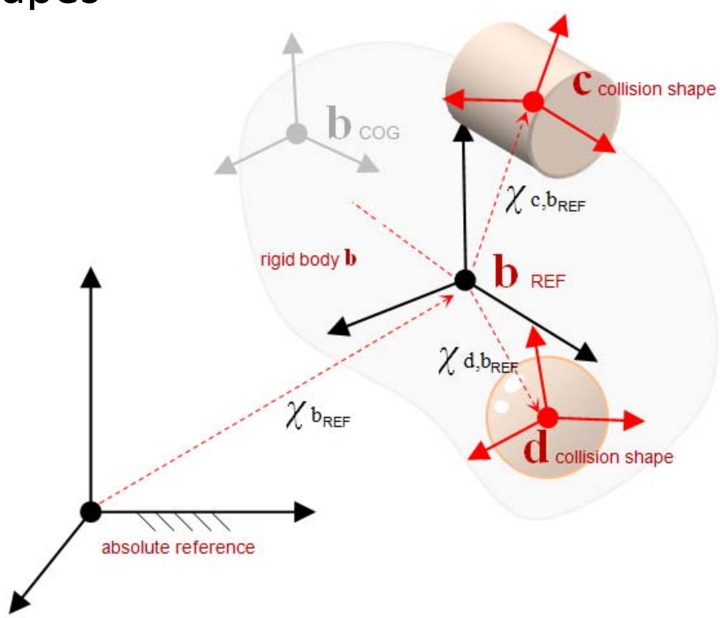
  (pair of ChMarker from two bodies)

```
auto marker_c = std::make_shared<ChMarker>();
marker_c->Impose_Abs_Coord(X_ca); // or..
marker_c->Impose_Rel_Coord(X_cb);
body_b->AddMarker(marker_c);
```



Collision shapes

- Collision shapes are defined respect to the REF frame of the body
- Spheres, boxes, cylinders, convex hulls, ellipsoids, compounds,…
- Concave shapes: decompose in compounds of convex shapes
- For simple ready-to-use bodies with predefined collision shapes, can use:
  - ChBodyEasySphere,
  - ChBodyEasyBox,
  - etc.



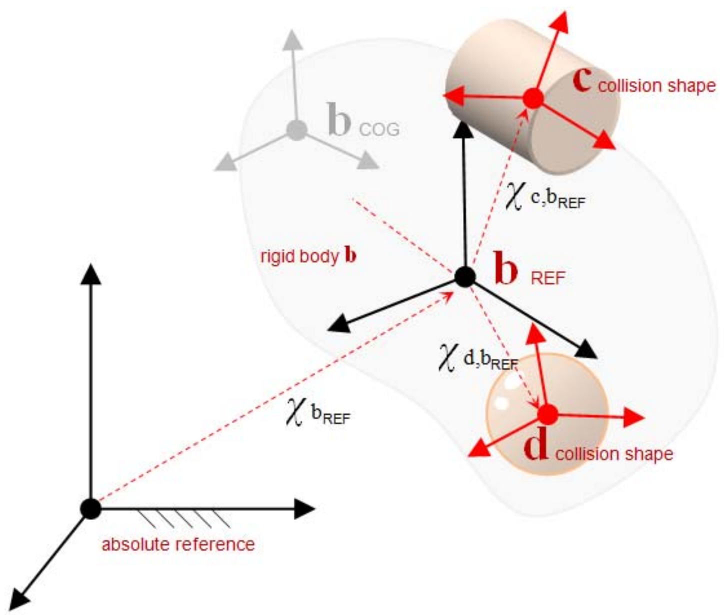
Specifying collision material

- Easy but potentially memory-inefficient:

```
body_b->SetFriction(0.4f);
body_b->SetRollingFriction(0.001f);
```

- Using a shared material:

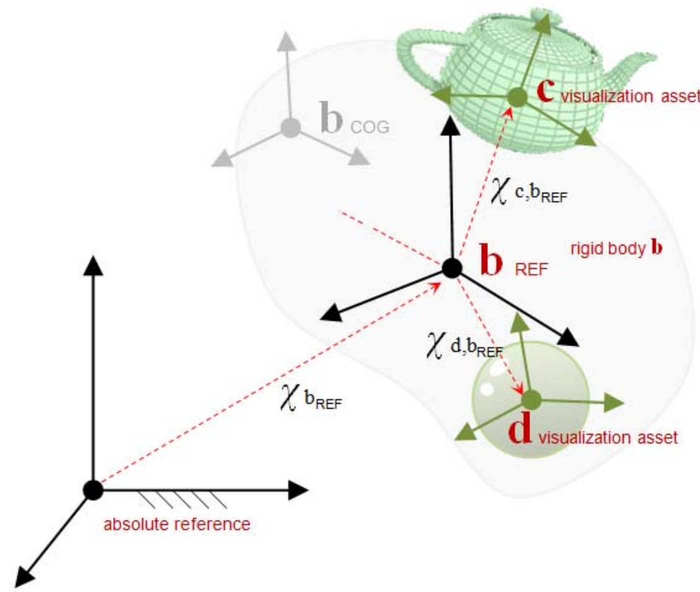
```
// Create a surface material and change properties:
auto mat = std::make_shared<ChMaterialSurface>();

mat->SetFriction(0.4f);
mat->SetRollingFriction(0.001f);

// Assign surface material to body/bodies:

body_b->SetSurfaceMaterial(mat);
body_c->SetSurfaceMaterial(mat);
body_d->SetSurfaceMaterial(mat);
. . .
```



Visualization assets

```
ChAsset
    ChVisualization
        ChSphereShape
        ChCylinderShape
        ChBoxShape
        …
```

- An arbitrary number of visualization assets can be attached to a body
- The position and rotation are defined with respect to REF frame
- Visualization assets are used by postprocessing systems and by the runtime 3D interfaces



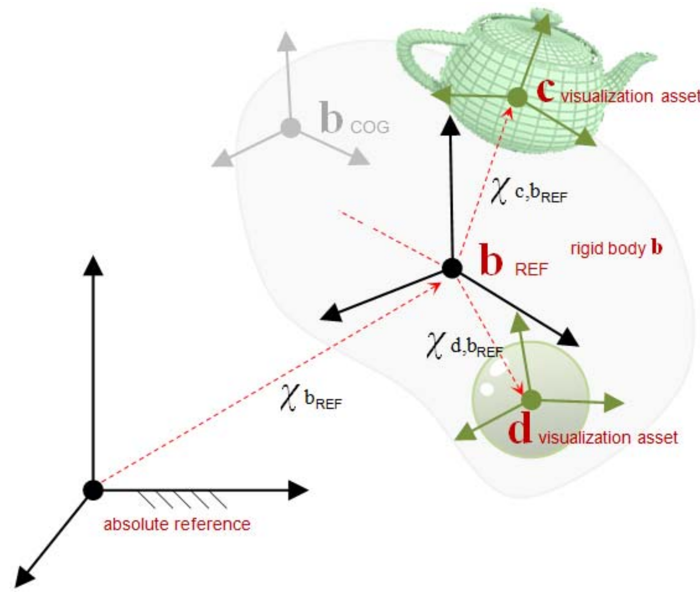
Visualization assets – construction (1/2)

- Example: add a box

```
auto box = std::make_shared<ChBoxShape>();
box->GetBoxGeometry().Pos = ChVector<>(0,-1,0);
box->GetBoxGeometry().Size = ChVector<>(10,0.5,10);
body->AddAsset(box);
```

- Example: add a texture

```
auto texture = std::make_shared<ChTexture>();
texture->SetTextureFilename(GetChronoDataFile("bluwhite.png"));
body->AddAsset(texture);
```



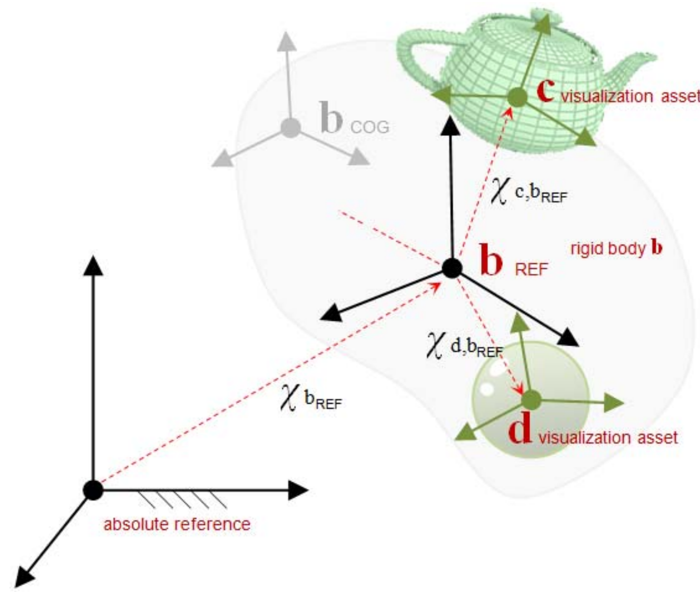
Visualization assets – construction (2/2)

- Example: add a mesh (reference to a Wavefront OBJ file)

```
auto meshfile = std::make_shared<ChObjShapeFile>();
meshfile->SetFilename("forklift_body.obj");
body->AddAsset(meshfile);
```

- Example:

```
auto mesh = std::make_shared<ChTriangleMeshShape>();
mesh->GetMesh()->LoadWavefrontMesh("forklift_body.obj");
body->AddAsset(mesh);
```



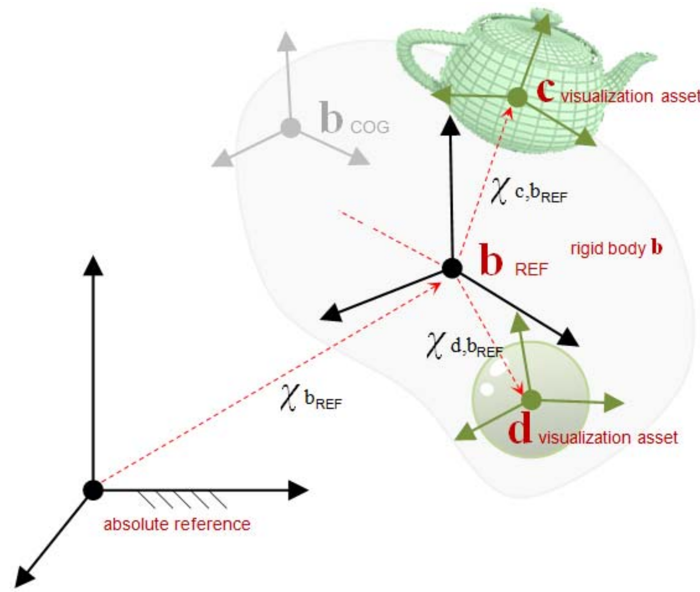
Visualization assets – use with Irrlicht

- After you attached usual visualization assets, do this:

```
auto irr_asset = std::make_shared<ChIrrNodeAsset>();
body->AddAsset(irr_asset);
irr_application->AssetBind(body);
irr_application->AssetUpdate(body);
```

- Otherwise, after all asset creation in all bodies, do:

```
irr_application.AssetBindAll();
irr_application.AssetUpdateAll();
```

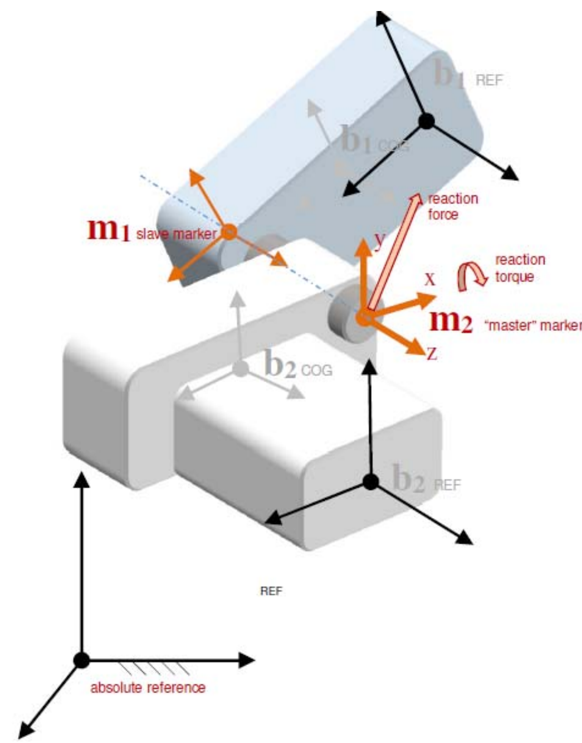


Constraints

ChLink

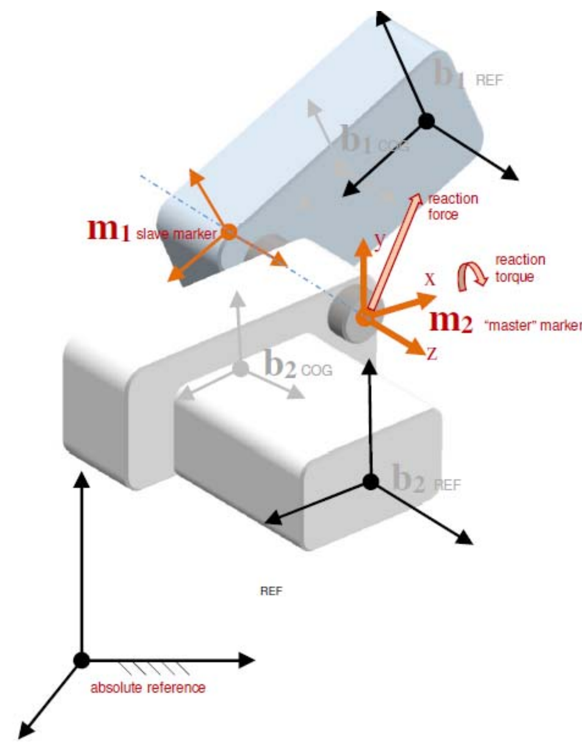
- Links are used to connect two ChBody
- There are many sub-classes of ChLink:
  - ChLinkLockSpherical
  - ChLinkLockRevolute
  - ChLinkLockLock
  - ChLinkLockPrismatic
  - ChLinkGears
  - ChLinkDistance
  - …

- See API documentation



Joints between markers

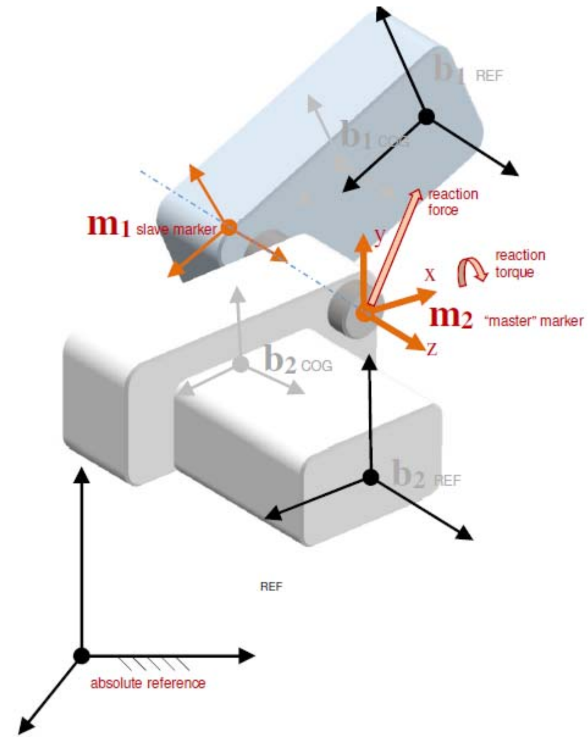
- Most links use two ChMarker as references
- The marker m2 (on body 2) is the **master** marker
- Reactions and joint rotations/speeds etc. are computed respect to the **master** marker
- Motion is constrained respect to the x, y, z axes of the frame of the **master** marker, e.g.:
  - ChLinkLockRevolute: allowed DOF on z axis rotation
  - ChLinkLockPrismatic: allowed DOF on x axis translation
  - etc.



ChLink – construction

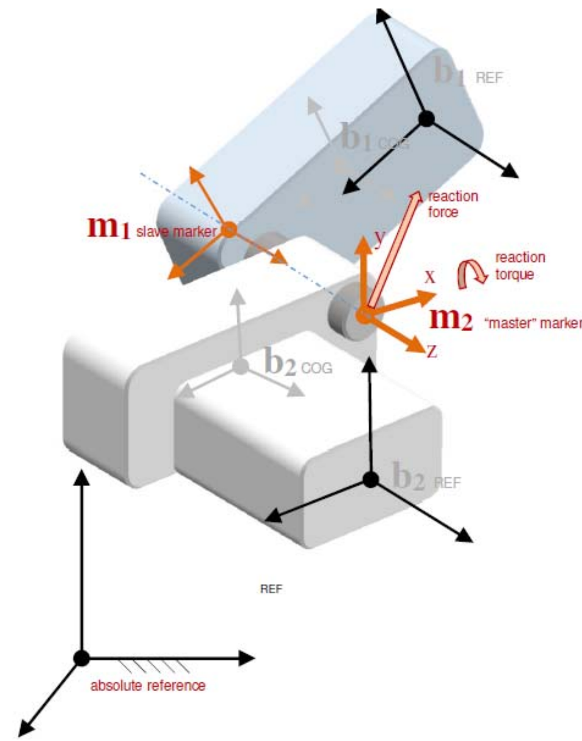
Important steps for each ChLink:

1. Create the link from the desired ChLink*** class
2. Initialize the link to connect two (existing) bodies
3. Add the link to the ChSystem
4. Optional: change default link properties

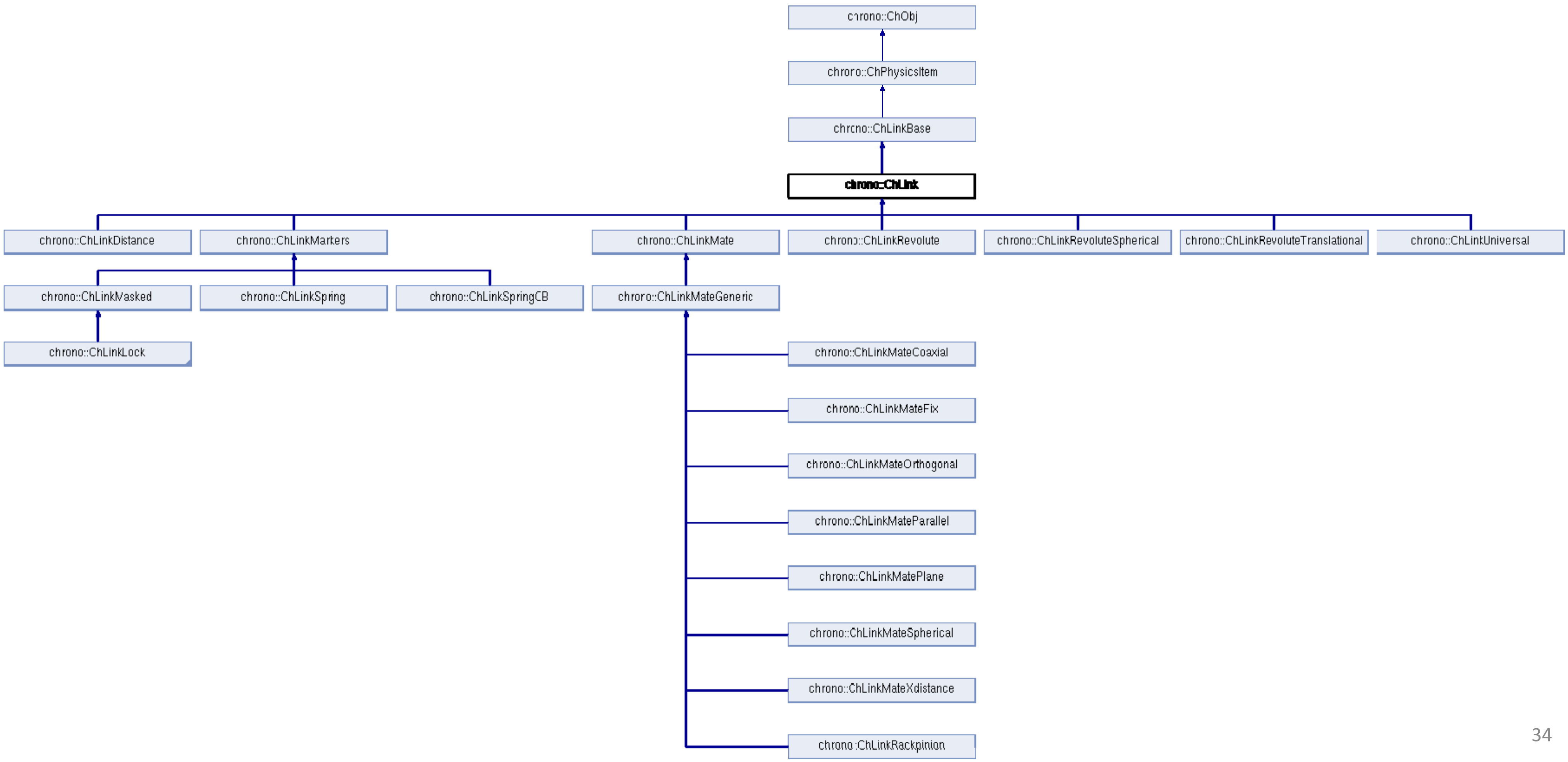


ChLink – construction example

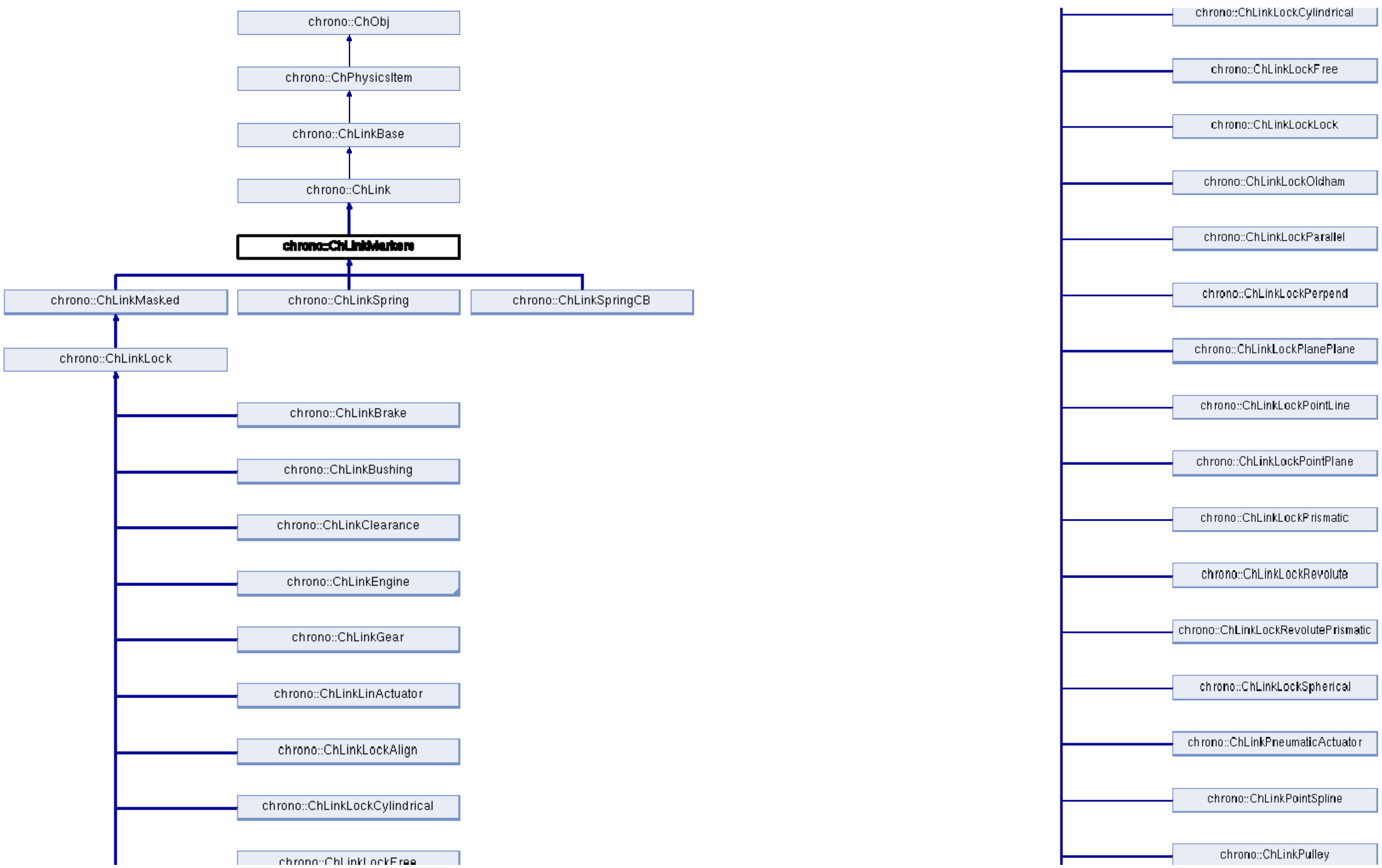
```
// 1- Create a constraint of 'engine' type, that constrains
// all x,y,z,Rx,Ry,Rz relative motions of marker 1 respect
// to 2, and Rz will follow a prescribed rotation.

auto my_motor = std::make_shared<ChLinkEngine>();

// 2- Initialization: define the position of m2 in absolute space:

my_motor->Initialize(rotatingBody,                          // <- body 1
                     floorBody,                             // <- body 2
                     ChCoordsys<>(ChVector<>(2,3,0),        // location
                     Q_from_AngAxis(CH_C_PI_2, VECT_X))     // orientation
                     );

// 3- Add the link to the system!
system.AddLink(my_motor);

// 4- Set some properties:

my_motor->Set_eng_mode(ChLinkEngine::ENG_MODE_SPEED);
if (auto mfun = std::dynamic_pointer_cast<ChFunction_Const>()) {
       // set (angular) speed = 90 deg/s
       mfun->Set_yconst(CH_C_PI/2.0);
}
```



ChLink class hierarchy



ChLinkLock class hierarchy



Force elements and actuators

Linear spring-damper-actuator

- ChLinkSpring - defines a linear spring-damper-actuator between two markers on two bodies

```
/// Specialized initialization for springs, given the two bodies to be connected,
/// the positions of the two anchor endpoints of the spring (each expressed
/// in body or abs. coordinates) and the imposed rest length of the spring.
/// NOTE! As in ChLinkMarkers::Initialize(), the two markers are automatically
/// created and placed inside the two connected bodies.
void Initialize(
     std::shared_ptr<ChBody> mbody1,  ///< first  body to join
     std::shared_ptr<ChBody> mbody2,  ///< second body to join
     bool pos_are_relative,           ///< true: following pos. considered relative to bodies. false: pos.are absolute
     ChVector<> mpos1,                ///< position of spring endpoint, for 1st body (rel. or abs., see flag above)
     ChVector<> mpos2,                ///< position of spring endpoint, for 2nd body (rel. or abs., see flag above)
     bool auto_rest_length = true,    ///< if true, initializes the rest-length as the distance between mpos1 and mpos2
     double mrest_length = 0          ///< imposed rest_length (no need to define, if auto_rest_length=true.)
     );
```

```
void Set_SpringRestLength(double m_r) { spr_restlength = m_r; }
void Set_SpringK(double m_r)          { spr_k = m_r; }
void Set_SpringR(double m_r)          { spr_r = m_r; }
void Set_SpringF(double m_r)          { spr_f = m_r; }
```

Spring coefficient

Damping coefficient

Constant spring force

General spring-damper-actuator

- ChLinkSpringCB – defines a general spring-damper-actuator with the force provided through a callback object

```
/// Base callback function for implementing a general spring-damper force
/// A derived class must implement the virtual operator().
class ChSpringForceCallback {
  public:
    virtual double operator()(double time,         ///< current time
                              double rest_length,  ///< undeformed length
                              double length,       ///< current length
                              double vel           ///< current velocity (positive when extending)
                              ) = 0;
};
```

```
 void Set_SpringCallback(ChSpringForceCallback* force) { m_force_fun = force; }
```

- Callback example

```
class TensionerForce : public ChSpringForceCallback {
  public:
    M113_TensionerForce(double k, double c, double f, double l0) : m_k(k), m_c(c), m_f(f), m_l0(l0) {}

    virtual double operator()(double time, double rest_length, double length, double vel) override {
        return m_f - m_k * (length - m_l0) - m_c * vel;
    }

  private:
    double m_l0, m_k, m_c, m_f;
};
```

Link forces

- ChLinkForce – defines a generic function to be applied to any degree of freedom of a ChLinkMasked

```
/// Class for forces in link joints of type ChLink().

class ChApi ChLinkForce {
  private:
    bool active;                ///< true/false

    double F;                   ///< actuator force
    ChFunction* modF;           ///< time-modulation of imp. force

    double K;                   ///< stiffness of the dof
    ChFunction* modK;           ///< modulation of K along the dof coord

    double R;                   ///< damping of the dof
    ChFunction* modR;           ///< modulation of R along the dof coord
```

$$F = modF(t) \cdot F + [modK(t) \cdot K] \cdot x + [modR(t) \cdot R] \cdot \dot{x}$$

Rotational spring-damper-actuator

- Attach a ChLinkForce to the rotational degree of freedom of a revolute joint

```
ChLinkForce my_link_force;

auto revolute = std::make_shared<ChLinkLockRevolute>();
revolute->Initialize(body1, body2, ChCoordsys<>(ChVector<>(), ChQuaternion<>()));
revolute->SetForce_Rz(&my_link_force);
system->AddLink(revolute);
```

Motion functions

- The ChFunction class defines the base class for all Chrono functions of the type $y = f(x)$
- ChFunction objects are often used to set time-dependent properties, for example to set motion laws in linear actuators, engines, etc.
- Inherited classes must override at least the **Get_y()** method, in order to represent more complex functions.

Motion functions

```
#include "motion_functions/ChFunction_Const.h"
#include "motion_functions/ChFunction_ConstAcc.h"
#include "motion_functions/ChFunction_Derive.h"
#include "motion_functions/ChFunction_Fillet3.h"
#include "motion_functions/ChFunction_Integrate.h"
#include "motion_functions/ChFunction_Matlab.h"
#include "motion_functions/ChFunction_Mirror.h"
#include "motion_functions/ChFunction_Mocap.h"
#include "motion_functions/ChFunction_Noise.h"
#include "motion_functions/ChFunction_Operation.h"
#include "motion_functions/ChFunction_Oscilloscope.h"
#include "motion_functions/ChFunction_Poly345.h"
#include "motion_functions/ChFunction_Poly.h"
#include "motion_functions/ChFunction_Ramp.h"
#include "motion_functions/ChFunction_Recorder.h"
#include "motion_functions/ChFunction_Repeat.h"
#include "motion_functions/ChFunction_Sequence.h"
#include "motion_functions/ChFunction_Sigma.h"
#include "motion_functions/ChFunction_Sine.h"
```

```
/// ChFunction_Const.h

/// Set the constant C for the function, y=C.
void Set_yconst (double y_constant) {C = y_constant;}

/// Get the constant C for the function, y=C.
virtual double Get_yconst () {return C;}
```

General force elements

- ChForce – force object associated with a rigid body:

```
void AddForce(std::shared_ptr<ChForce> force);
```

- Applies either a force (applied at a specified point) or a torque to the associated body.
- Can be specified in absolute or local frame.

$$F = M(t) \cdot \vec{v} + F_x(t) \cdot \vec{\imath} + F_y(t) \cdot \vec{\jmath} + F_z(t) \cdot \vec{k}$$

Force accumulators

- Each rigid body maintains a force and a torque accumulator which hold incremental forces
  - the force accumulator can be incremented by specifying an applied force and an application point (expressed in either absolute or local frame)
  - the torque accumulator can be incremented by specifying an applied torque (expressed in either absolute or local frame)
- Accumulators can be emptied at any time

```
/// Add forces and torques into the "accumulators", as increment.
/// Forces and torques currently in accumulators will affect the body.
/// It's up to the user to remember to empty them and/or set again at each
/// integration step. Useful to apply forces to bodies without needing to
/// add ChForce() objects. If local=true, force,appl.point or torque are considered
/// expressed in body coordinates, otherwise are considered in absolute coordinates.
void Accumulate_force(const ChVector<>& force, const ChVector<>& appl_point, int local);
void Accumulate_torque(const ChVector<>& torque, int local);
void Empty_forces_accumulators() {
    Force_acc = VNULL;
    Torque_acc = VNULL;
}
const ChVector<>& Get_accumulated_force() const { return Force_acc; }
const ChVector<>& Get_accumulated_torque() const { return Torque_acc; }
```

Building a Chrono system

Structure of a Chrono C++ program

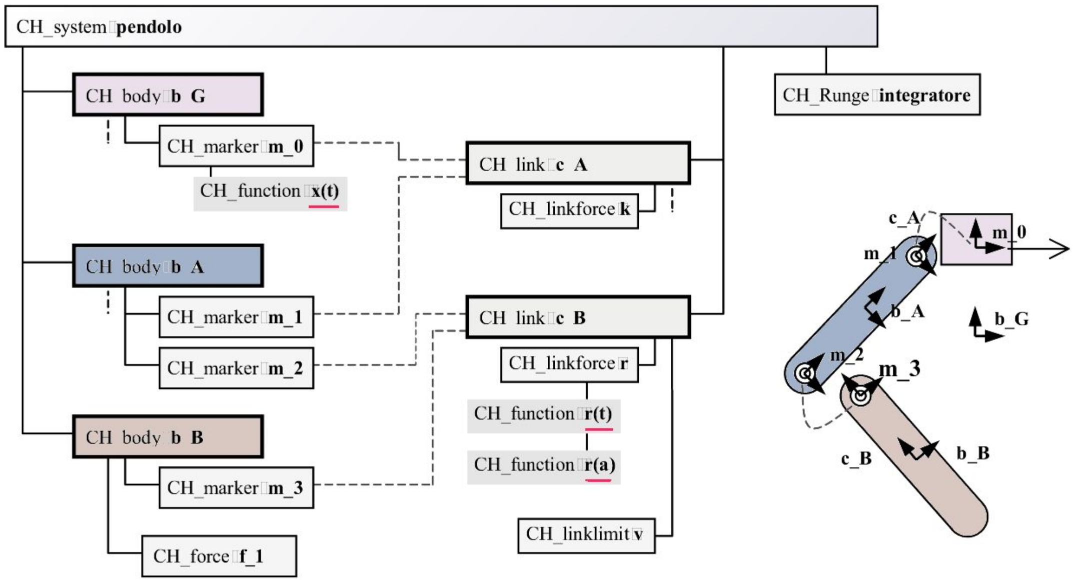
Building a system

ChSystem

- A ChSystem contains all items of the simulation: bodies, constraints, etc.

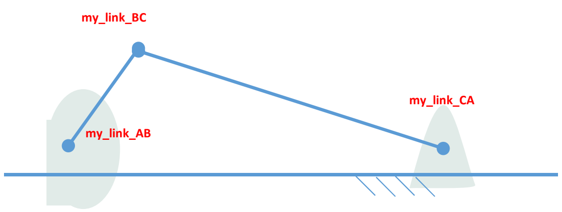
- Use the Add() , Remove() functions to populate it

- Simulation settings are in ChSystem:
  - integrator type
  - tolerances
  - etc.



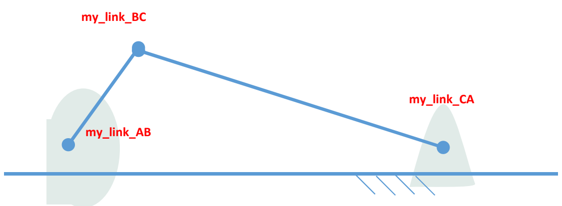
Building a system – example (1/3)

```
// 1- Create a ChronoENGINE physical system: all bodies and constraints
//    will be handled by this ChSystem object.
ChSystem my_system;

// 2- Create the rigid bodies of the slider-crank mechanical system
//   (a crank, a rod, a truss), maybe setting position/mass/inertias of
//   their center of mass (COG) etc.

// ..the truss
auto my_body_A = make_shared<ChBody>();
my_system.AddBody(my_body_A);
my_body_A->SetBodyFixed(true);                     // truss does not move!

// ..the crank
auto my_body_B = make_shared<ChBody>();
my_system.AddBody(my_body_B);
my_body_B->SetPos(ChVector<>(1,0,0));    // position of COG of crank

// ..the rod
auto my_body_C = make_shared<ChBody>();
my_system.AddBody(my_body_C);
my_body_C->SetPos(ChVector<>(4,0,0));    // position of COG of rod
```



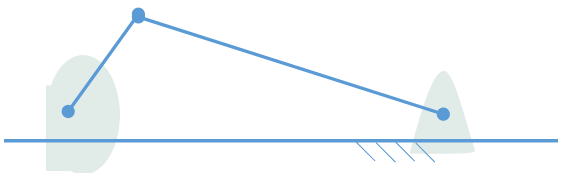
Building a system – example (2/3)

```
// 3- Create constraints: the mechanical joints between the
//    rigid bodies.

// .. a revolute joint between crank and rod
auto my_link_BC = make_shared<ChLinkLockRevolute>();
my_link_BC->Initialize(my_body_B, my_body_C, ChCoordsys<>(ChVector<>(2,0,0)));
my_system.AddLink(my_link_BC);

// .. a slider joint between rod and truss
auto my_link_CA = make_shared<ChLinkLockPointLine>();
my_link_CA->Initialize(my_body_C, my_body_A, ChCoordsys<>(ChVector<>(6,0,0)));
my_system.AddLink(my_link_CA);

// .. an engine between crank and truss
auto my_link_AB = make_shared<ChLinkEngine>();
my_link_AB->Initialize(my_body_A, my_body_B, ChCoordsys<>(ChVector<>(0,0,0)));
my_link_AB->Set_eng_mode(ChLinkEngine::ENG_MODE_SPEED);
my_system.AddLink(my_link_AB);
```



Building a system – example (3/3)

```
// 4- Adjust settings of the integrator (optional):

my_system.SetIntegrationType(ChSystem::INT_HHT)
my_system.SetLcpSolverType(ChSystem::LCP_MINRES);
my_system.SetIterLCPmaxItersSpeed(20);
my_system.SetIterLCPmaxItersStab(20);

// 5- Run the simulation (basic example)

while( my_system.GetChTime() < 10 )
{
        // Here Chrono::Engine time integration is performed:

        my_system.StepDynamics(0.02);

        // Draw items on screen (lines, circles, etc.)
        // or dump data to disk
        [..]
}
```



Some system settings

```
my_system.SetLcpSolverType(ChSystem::LCP_ITERATIVE_SOR);
```

LCP_ITERATIVE_SOR	for maximum speed in real-time applications, low precision, convergence might stall
LCP_ITERATIVE_APGC	slower but better convergence, works also in DVI
LCP_ITERATIVE_MINRES	for precise solution, but only ODE/DAE, no DVI for the moment
(etc.)	

```
my_system.SetIterLCPmaxItersSpeed(20);
```

Most LCP solvers have an upper limit on number of iterations. The higher, the more precise, but slower.

```
my_system.SetMaxPenetrationRecoverySpeed(0.2);
```

Objects that interpenetrate (e.g., due to numerical errors, incoherent initial conditions, etc.) do not ‘separate’ faster than this threshold.

The higher, the faster and more precisely the contact constraints errors (if any) are recovered, but the risk is that objects ‘pop’ out, and stackings might become unstable and noisy.

The lower, the more likely the risk that objects ‘sink’ one into one another when the integrator precision is low (e.g., small number of iterations).

```
my_system.SetMinBounceSpeed(0.1);
```

When objects collide, if their incoming speed is lower than this threshold, a zero restitution coefficient is assumed. This helps to achieve more stable simulations of stacked objects. The higher, the more likely it is to get stable simulations, but the less realistic the physics of the collision.

Validation & Verification

Validation process

- Multiple test cases per joint/constraint/force were created based on simple mechanisms to exercise each components
- MSC ADAMS models were generated for each test case and the simulated translational and rotational positions, velocities, accelerations, and reaction forces and torques were post processed into individual comparison text files
- Equivalent Chrono models were then constructed and setup to generate the corresponding output files for comparing to MSC ADAMS as well as for testing conservation of energy and the constraint violations.
- Since the two programs used different solvers, a set of tolerances were defined for each test to ensure that the results were reasonably close to each other, since in most cases a closed form solution did not exist.
- These tests will be used to validate future changes to the code.

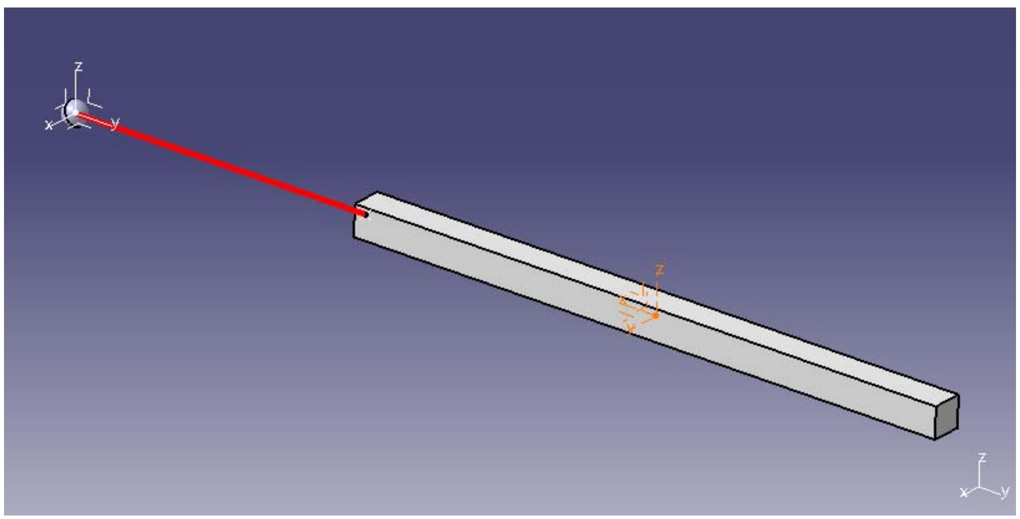
Validated components

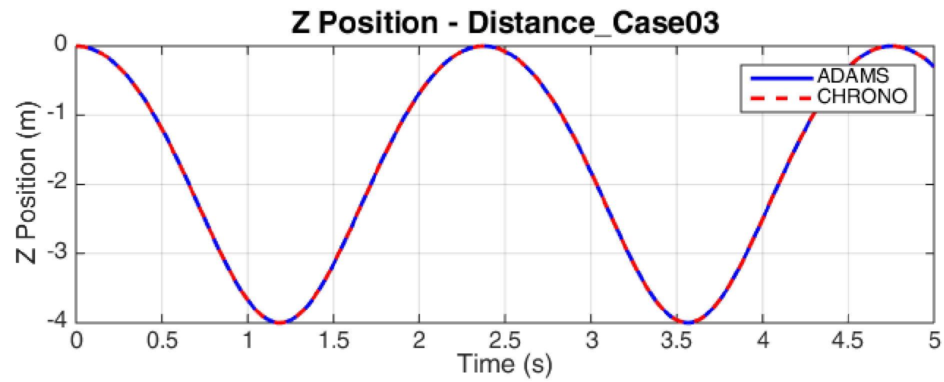
- Joints
  - Revolute (ChLinkLockRevolute)
  - Spherical (ChLinkLockSpherical)
  - Universal (ChLinkUniversal)
  - Prismatic (ChLinkLockPrismatic)
  - Cylindrical (ChLinkLockCylindrical)

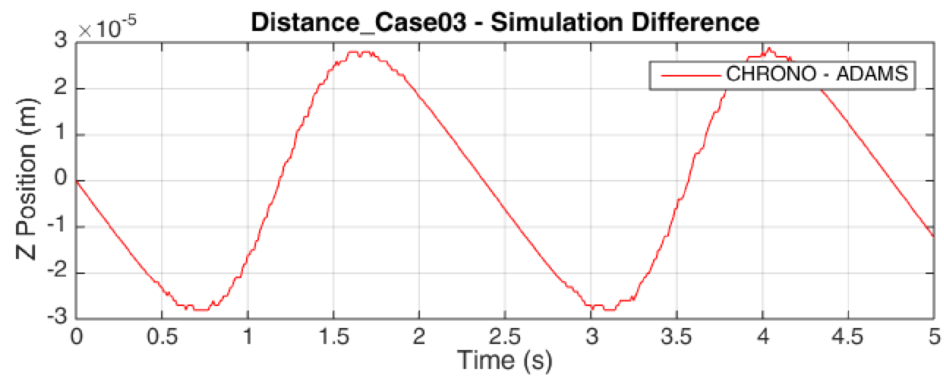
- Constraints
  - Distance (ChLinkDistance)
- Forces
  - Translational Spring/Damper (ChLinkSpring and ChLinkSpringCB)
  - Rotational Spring/Damper (SetForce_Rz applied to ChLinkLockRevolute)

Sample validation: distance constraint

- Distance Constraint Case 03 - Double Pendulum
  - Distance Constraint between ground and the end of the pendulum.
  - Gravity point along –Z
  - Pendulum is initially at rest in the horizontal position







Sample validation: distance constraint