Chrono::Engine Hands-on Exercises

Modeling and simulation of a slider-crank mechanism

Building a Chrono system

Structure of a Chrono C++ program

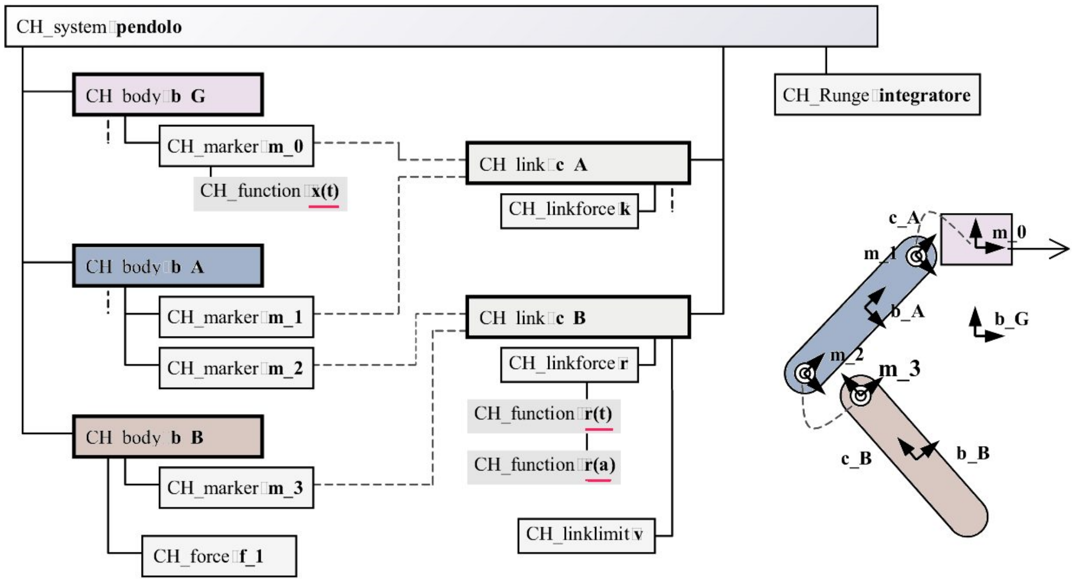
Building a system

ChSystem

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

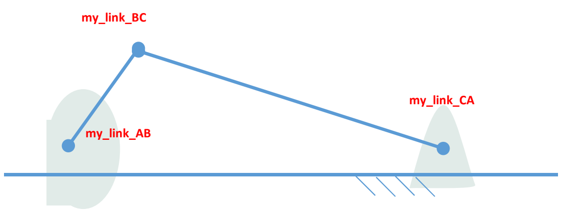
- Use the Add() , Remove() functions to populate it

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



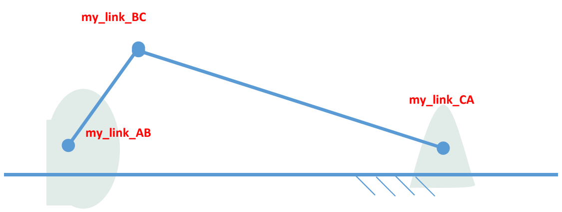
Building a system – example (1/3)

```
// 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
```



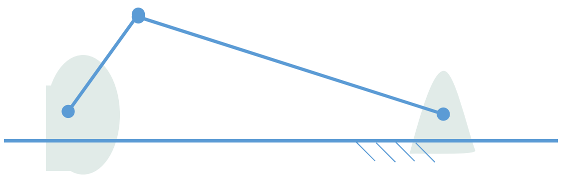
Building a system – example (2/3)

```
// 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);
my_system.SetMaxPenetrationRecoverySpeed(0.2);
my_system.SetMinBounceSpeed(0.1);


// 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).

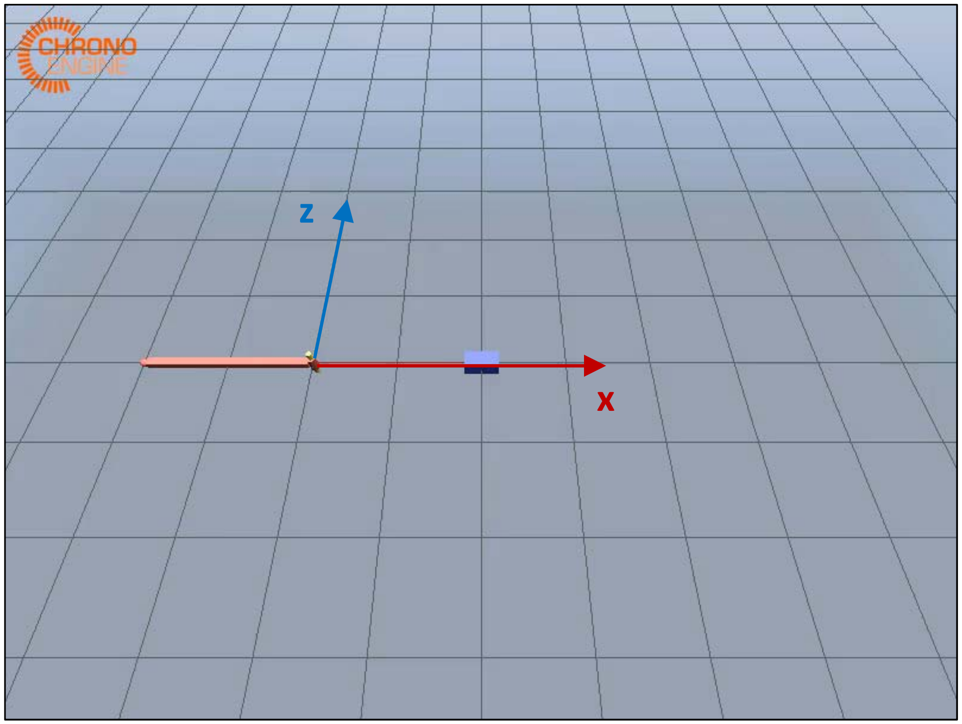
```
my_system.SetMinBounceSpeed(0.1);
```

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.

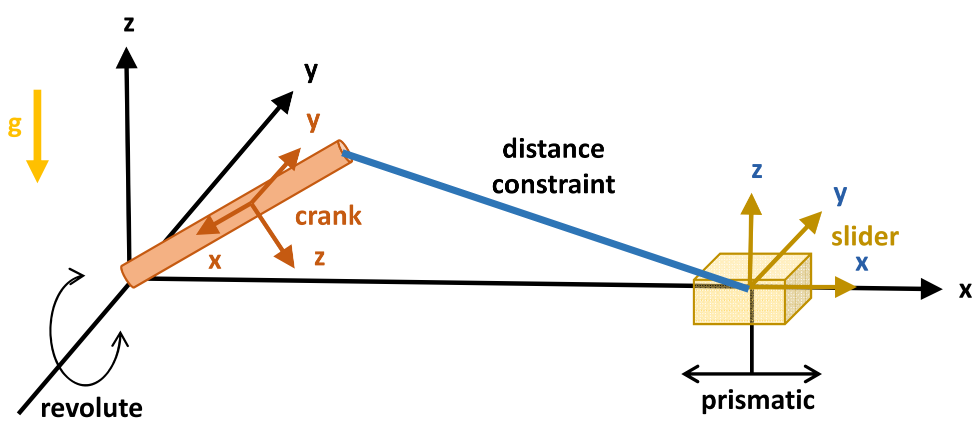
Tutorial exercises

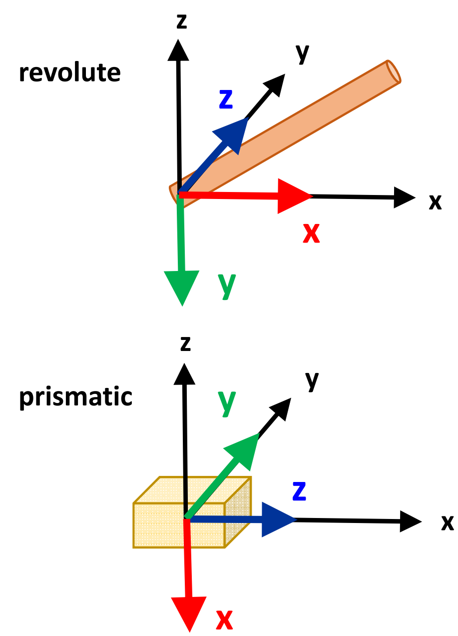
Base model: 2-body slider-crank mechanism

- Crank and slider bodies
- Revolute and prismatic joints
- Distance constraint
- Moving under gravity only

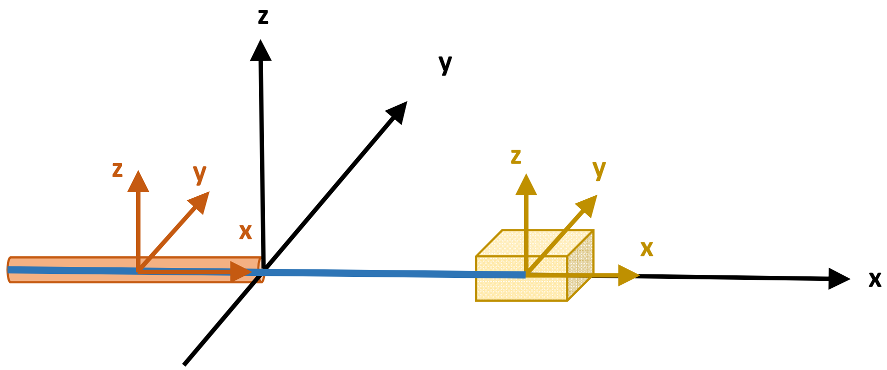


Body and joint frames

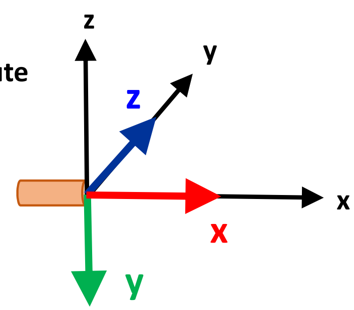




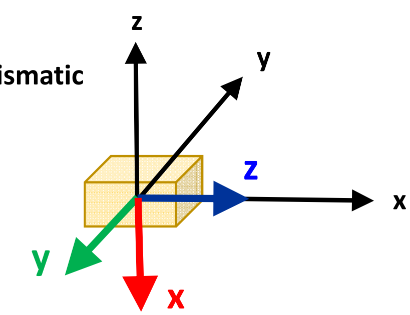
Initial configuration



revolute



prismatic



Defining a body

- Specify mass properties
- Specify initial conditions (relative to global frame)
  - Position and orientation
  - Linear velocity and angular velocity

```
// Crank
auto crank = make_shared<ChBody>();
system.AddBody(crank);
crank->SetIdentifier(1);
crank->SetName("crank");
crank->SetMass(1.0);
crank->SetInertiaXX(ChVector<>(0.005, 0.1, 0.1));
crank->SetPos(ChVector<>(-1, 0, 0));
crank->SetRot(ChQuaternion<>(1, 0, 0, 0));
```

Defining visualization assets

- Specify geometry assets (relative to the body frame)
- Specify color asset

```
auto box_c = make_shared<ChBoxShape>();
box_c->GetBoxGeometry().Size = ChVector<>(0.95, 0.05, 0.05);
crank->AddAsset(box_c);

auto cyl_c = make_shared<ChCylinderShape>();
cyl_c->GetCylinderGeometry().p1 = ChVector<>(-1, 0.1, 0);
cyl_c->GetCylinderGeometry().p2 = ChVector<>(-1, -0.1, 0);
cyl_c->GetCylinderGeometry().rad = 0.05;
crank->AddAsset(cyl_c);

auto sph_c = make_shared<ChSphereShape>();
sph_c->GetSphereGeometry().center = ChVector<>(1, 0, 0);
sph_c->GetSphereGeometry().rad = 0.05;
crank->AddAsset(sph_c);

auto col_c = make_shared<ChColorAsset>();
col_c->SetColor(ChColor(0.6f, 0.2f, 0.2f));
crank->AddAsset(col_c);
```

Defining a joint

- Specify the two connected bodies
- Specify a single global joint frame or two local joint frames

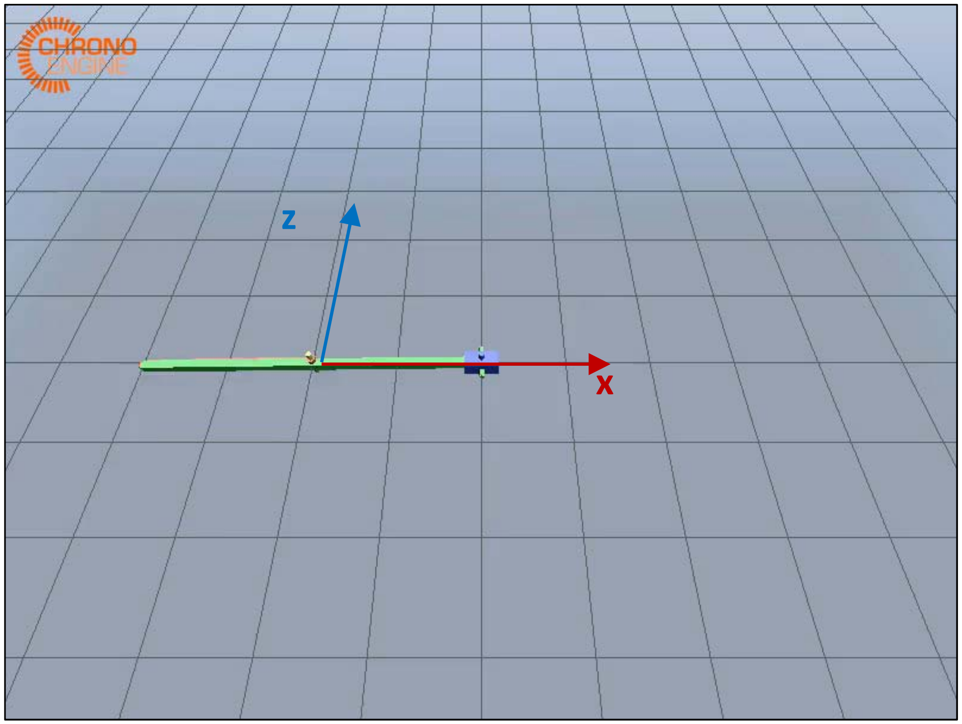
```
// Revolute joint between ground and crank.
// The rotational axis of a revolute joint is along the Z axis of the
// specified joint coordinate frame.  Here, we apply the 'z2y' rotation to
// align it with the Y axis of the global reference frame.
auto revolute_ground_crank = make_shared<ChLinkRevolute>();
revolute_ground_crank->SetName("revolute_ground_crank");
revolute_ground_crank->Initialize(ground, crank, ChFrame<>(ChVector<>(0, 0, 0), z2y));
system.AddLink(revolute_ground_crank);
```

Alternatively

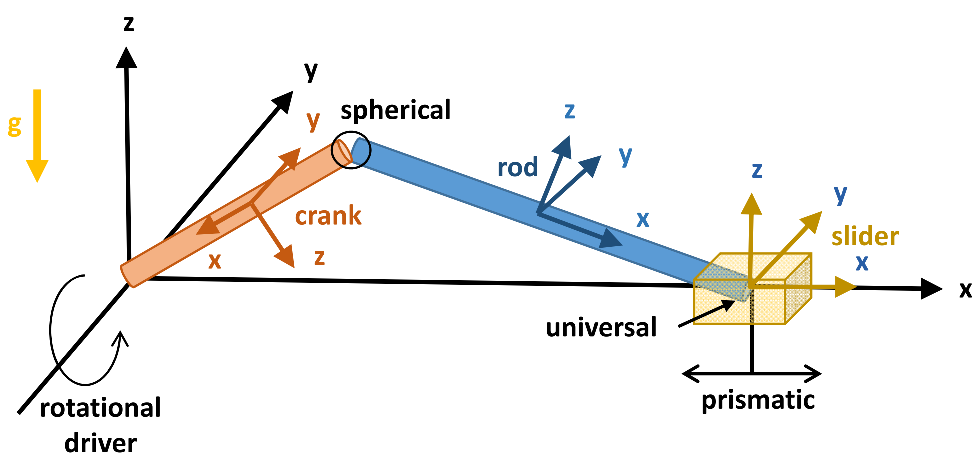
```
revolute_ground_crank->Initialize(ground, crank,
                                  true,
                                  ChFrame<>(ChVector<>(0, 0, 0), z2y),
                                  ChFrame<>(ChVector<>(1, 0, 0), z2y));
```

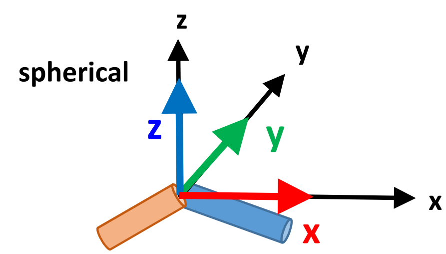
Exercise 1: driven 3-body slider-crank mechanism

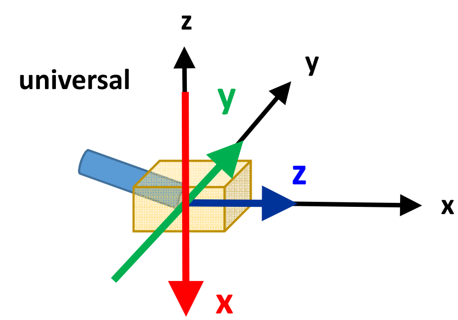
- Add a connecting rod body
- Replace distance constraint with kinematic joints (spherical and universal)
- Replace revolute joint with a rotational driver



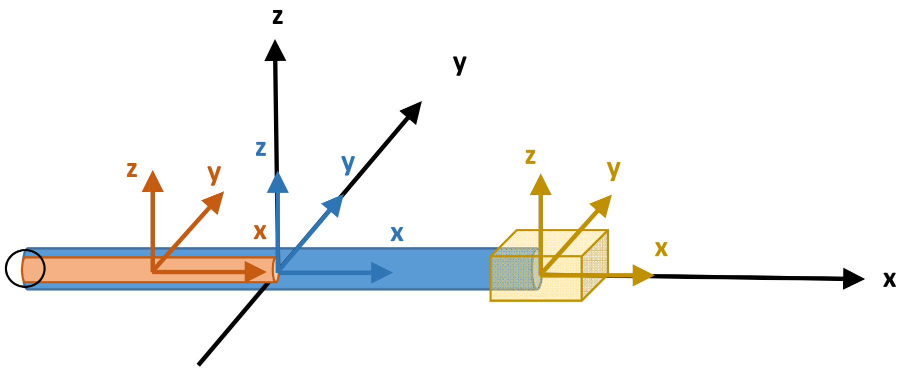
Body and joint frames



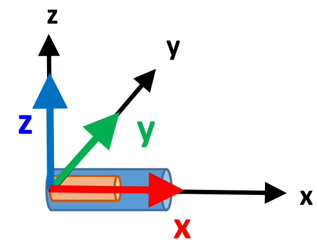




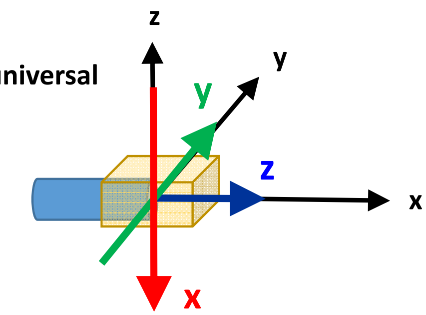
Initial configuration



spherical



universal



Spherical joint: ChLinkLockSpherical

```
/// Use this function after link creation, to initialize the link from
/// two markers to join.
/// Each marker must belong to a rigid body, and both rigid bodies
/// must belong to the same ChSystem.
/// The position of mark2 is used as link's position and main reference.
virtual void Initialize(std::shared_ptr<ChMarker> mark1,  ///< first  marker to join
                        std::shared_ptr<ChMarker> mark2   ///< second marker to join (master)
                        );

/// Use this function after link creation, to initialize the link from
/// two joined rigid bodies.
/// Both rigid bodies must belong to the same ChSystem.
/// Two markers will be created and added to the rigid bodies (later,
/// you can use GetMarker1() and GetMarker2() to access them.
/// To specify the (absolute) position of link and markers, use 'mpos'.
virtual void Initialize(std::shared_ptr<ChBody> mbody1,  ///< first  body to join
                        std::shared_ptr<ChBody> mbody2,  ///< second body to join
                        const ChCoordsys<>& mpos         ///< the current absolute pos.& alignment.
                        );

/// Use this function after link creation, to initialize the link from
/// two joined rigid bodies.
/// Both rigid bodies must belong to the same ChSystem.
/// Two markers will be created and added to the rigid bodies (later,
/// you can use GetMarker1() and GetMarker2() to access them.
/// To specify the (absolute) position of link and markers, use 'mpos'.
virtual void Initialize(
        std::shared_ptr<ChBody> mbody1,  ///< first  body to join
        std::shared_ptr<ChBody> mbody2,  ///< second body to join
        bool pos_are_relative,           ///< if =true, following two positions are relative to bodies. If false, are absolute.
        const ChCoordsys<>& mpos1,       ///< the position & alignment of 1st marker (relative to body1 cords, or absolute)
        const ChCoordsys<>& mpos2        ///< the position & alignment of 2nd marker (relative to body2 cords, or absolute)
        );
```

Universal joint: ChLinkUniversal

```
/// Initialize this joint by specifying the two bodies to be connected and a
/// joint frame specified in the absolute frame. Two local joint frames, one
/// on each body, are constructed so that they coincide with the specified
/// global joint frame at the current configuration. The kinematics of the
/// universal joint are obtained by imposing that the origins of these two
/// frames are the same and that the X axis of the joint frame on body 1 and
/// the Y axis of the joint frame on body 2 are perpendicular.
void Initialize(std::shared_ptr<ChBodyFrame> body1,  ///< first body frame
                std::shared_ptr<ChBodyFrame> body2,  ///< second body frame
                const ChFrame<>& frame               ///< joint frame (in absolute frame)
                );

/// Initialize this joint by specifying the two bodies to be connected and the
/// joint frames on each body. If local = true, it is assumed that these quantities
/// are specified in the local body frames. Otherwise, it is assumed that they are
/// specified in the absolute frame.
void Initialize(std::shared_ptr<ChBodyFrame> body1,  ///< first body frame
                std::shared_ptr<ChBodyFrame> body2,  ///< second body frame
                bool local,                          ///< true if data given in body local frames
                const ChFrame<>& frame1,             ///< joint frame on body 1
                const ChFrame<>& frame2              ///< joint frame on body 2
                );
```

Rotational driver: ChLinkEngine

```
void  Set_eng_mode(int mset);

enum eCh_eng_mode {
        ENG_MODE_ROTATION = 0,
        ENG_MODE_SPEED,
        ENG_MODE_TORQUE,
        ENG_MODE_KEY_ROTATION,
        ENG_MODE_KEY_POLAR,
        ENG_MODE_TO_POWERTRAIN_SHAFT
};
```

```
void Set_rot_funct(std::shared_ptr<ChFunction> mf) { rot_funct = mf; }
void Set_spe_funct(std::shared_ptr<ChFunction> mf) { spe_funct = mf; }
void Set_tor_funct(std::shared_ptr<ChFunction> mf) { tor_funct = mf; }
```

ChFunction

- 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.

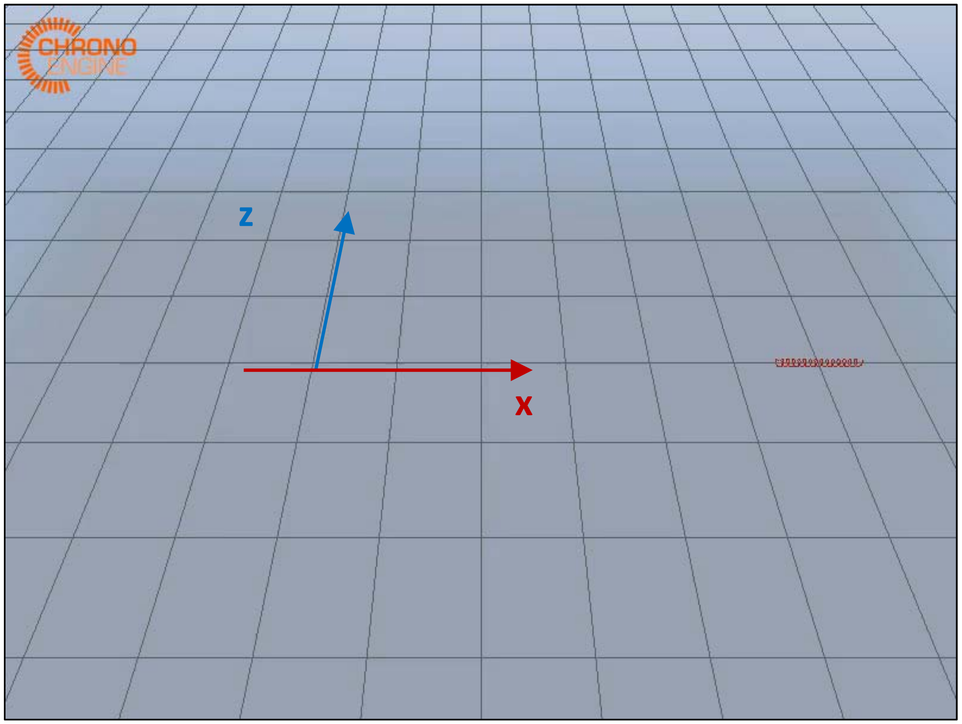
- Inherited classes must override at least the **Get_y()** method, in order to represent more complex functions.

```
#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;}
```

Exercise 2: interaction through contact

- Add a ball connected to ground through a prismatic joint
- Enable contact on slider and ball and add contact geometry
- Add a translational spring-damper



Specifying contact geometry

ChBody functions

```
/// Enable/disable the collision for this rigid body.
void  SetCollide(bool mcoll);
```

```
/// Access the collision model for the collision engine.
/// To get a non-null pointer, remember to SetCollide(true), before.
collision::ChCollisionModel* GetCollisionModel() {return collision_model;}
```

ChCollisionModel functions

```
/// Deletes all inserted geometries.
/// Call this function BEFORE adding the geometric description.
virtual int ClearModel() = 0;

/// Builds the BV hierarchy.
/// Call this function AFTER adding the geometric description.
virtual int BuildModel() = 0;
```

```
/// Add a sphere shape to this model, for collision purposes
virtual bool AddSphere(double            radius,              ///< the radius of the sphere
                       const ChVector<>& pos = ChVector<>()   ///< the position of the sphere in model coordinates
                       ) = 0;
```

```
/// Add a box shape to this model, for collision purposes
virtual bool AddBox(double              hx,                    ///< the halfsize on x axis
                    double              hy,                    ///< the halfsize on y axis
                    double              hz,                    ///< the halfsize on z axis
                    const ChVector<>&   pos = ChVector<>(),    ///< the position of the box COG
                    const ChMatrix33<>& rot = ChMatrix33<>(1)  ///< the rotation of the box
                    ) = 0;
```

Translational spring-damper: ChLinkSpring

```
    /// 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

Tutorial solutions

Rotational driver: ChLinkEngine (solution)

```
// Create a ChFunction object that always returns the constant value PI/2.
auto fun = std::make_shared<ChFunction_Const>();
fun->Set_yconst(CH_C_PI);

// Engine between ground and crank.
// Note that this also acts as a revolute joint (i.e. it enforces the same
// kinematic constraints as a revolute joint).  As before, we apply the 'z2y'
// rotation to align the rotation axis with the Y axis of the global frame.
auto engine_ground_crank = std::make_shared<ChLinkEngine>();
engine_ground_crank->SetName("engine_ground_crank");
engine_ground_crank->Initialize(ground, crank, ChCoordsys<>(ChVector<>(0, 0, 0), z2y));
engine_ground_crank->Set_eng_mode(ChLinkEngine::ENG_MODE_SPEED);
engine_ground_crank->Set_spe_funct(fun);
system.AddLink(engine_ground_crank);
```

Specifying contact geometry (solution)

```
//// Create a new body, with a spherical shape (radius 0.2), used both as
//// visualization asset and contact shape (mu = 0.4). This body should have:
////    mass: 1
////    moments of inertia: I_xx = I_yy = I_zz = 0.02
////    initial location: (5.5, 0, 0)

auto ball = std::make_shared<ChBody>();
system.AddBody(ball);
ball->SetIdentifier(4);
ball->SetName("ball");
ball->SetMass(1);
ball->SetInertiaXX(ChVector<>(0.02, 0.02, 0.02));
ball->SetPos(ChVector<>(5.5, 0, 0));
ball->SetRot(ChQuaternion<>(1, 0, 0, 0));

ball->SetCollide(true);
ball->GetCollisionModel()->ClearModel();
ball->GetCollisionModel()->AddSphere(0.2, ChVector<>(0, 0, 0));
ball->GetCollisionModel()->BuildModel();

auto sphere_b = std::make_shared<ChSphereShape>();
sphere_b->GetSphereGeometry().center = ChVector<>(0, 0, 0);
sphere_b->GetSphereGeometry().rad = 0.2;
ball->AddAsset(sphere_b);

auto col_b = std::make_shared<ChColorAsset>();
col_b->SetColor(ChColor(0.6f, 0.6f, 0.6f));
ball->AddAsset(col_b);
```

Create body
Specify mass properties
Specify initial conditions

Specify contact geometry

Specify visual assets

Translational spring-damper: ChLinkSpring (solution)

```
//// ------------------------------------------------------------------------
//// EXERCISE 2
//// Add a spring-damper (ChLinkSpring) between ground and the ball.
//// This element should connect the center of the ball with the global point
//// (6.5, 0, 0).  Set a spring constant of 50 and a spring free length of 1.
//// Set a damping coefficient of 5.
//// ------------------------------------------------------------------------

auto tsda_ground_ball = std::make_shared<ChLinkSpring>();
tsda_ground_ball->SetName("tsda_ground_ball");
tsda_ground_ball->Initialize(ground, ball, false, ChVector<>(6.5, 0, 0), ChVector<>(5.5, 0, 0));
tsda_ground_ball->Set_SpringK(50.0);
tsda_ground_ball->Set_SpringR(5.0);
tsda_ground_ball->Set_SpringRestLength(1.0);
system.AddLink(tsda_ground_ball);
```