CHRONO TUTORIAL
Agenda

- Chrono overview
  - What is Chrono and what is it not?
  - Capabilities, unique features (what differentiates Chrono from other physics packages)
  - Software packages under the ‘Chrono’ umbrella (parallel, fluid, flex) and related packages (spike::GPU)
  - Overall architecture and design philosophy

- Chrono::Vehicle overview
  - Template-based wheeled vehicle modeling
  - Supported topologies, future developments
  - Overall architecture and design philosophy

- Chrono validation
  - Validation of modeling building blocks (against ADAMS and analytical solutions)
  - Validations at subsystem and system level (Chrono::Vehicle)
  - Validation against experimental data (large-scale contact problems)
Agenda

• Chrono software, documentation, support
  • Code availability (Chrono and Chrono::Vehicle)
  • Library and tool dependencies
  • Configuration and build
  • Doxygen documentation (Chrono and Chrono::Vehicle)
  • List of all existing Chrono demos (what they are and what they emphasize)

• Using Chrono for MBD – Basic features
  • Introduction to the API
  • Working with shared pointers
  • Coordinate transformations
  • Bodies, markers, and joints
  • Specifying geometry (for contact and for visualization). Using visualization assets.
  • Putting it all together: building a system. Main system parameters
Agenda

- Hands-on demo
  - Given: working model of a 2-body slider-crank model
    - Define system, bodies, constraints
    - Define visualization assets for rendering
    - Mechanism moving under gravity only
  - Exercise 1: driven 3-body slider-crank mechanism
    - Add new body (connecting rod) and joints (spherical and universal)
    - Use rotational driver (constant angular velocity)
  - Exercise 2: interaction through contact
    - Add a ball on a prismatic joint
    - Add a translational spring-damper
    - Enable contact and specify contact geometry
CHRONO OVERVIEW
Let us go on and win glory for ourselves, or yield it to others

Homer, Iliad
Project Chrono

- Project Chrono is a growing ecosystem of software tools
- Based on the Chrono::Engine simulation library

http://www.projectchrono.org
Chrono::Engine
What is Chrono::Engine?

- **Middleware**: can be embedded in third parties software
- **Open source** with BSD license
- **Library developed in** C++
- **Cross-platform**: compiles on GNU GCC, MSVC, etc

A software library for multi-body mechanical simulations
What is Chrono::Engine?

- Modular: based on optional linking of units
- Expandable via C++ class inheritance
- Efficient, fast, robust algorithms
- Real-time performance when possible

A software library for multi-body mechanical simulations
What is Chrono::Engine?

- Large scale problems with millions of parts
- Support of parallel computing via GPU, MPI, HPC
- DVI formulation for non-smooth dynamics
- State-of-the-art collision-detection

A software library for multi-body mechanical simulations
Chrono::Engine modeling features

- Rigid bodies, markers, forces, torques
- Springs and dampers, with user-defined non-linear features
- Wide set of joints, ex. spherical, revolute joint, prismatic, universal joint, glyph, etc.
- Impose trajectories to parts and markers
- Constraint motion on splines, surfaces, etc.
- Constraints can have limits (ex. elbow)
Chrono::Engine modeling features

- Custom constraint for motors, reducers etc.
- Custom constraint for linear motors.
- 1-DOF elements for powertrains, drivelines, etc.
- Brakes and clutches, with stick-slip effect
- FEM: beams, tetrahedrons,.. (under construction)
- SPH fluids
Chrono::Engine modeling features

• Fast collision detection algorithms
• Collision families and groups
• Coloumb friction model, with stick-slip
• Rolling and spinning friction
• Restitution coefficients for rebouncing
• Collision detection between compound shapes
• Bodies activation/deactivation and sleeping
• Conveyor belts
Chrono::Engine architecture

- Each unit is a C++ library
- Units can be linked when necessary
Example of available C++ objects

Some joint types in our Chrono::Engine software
Example of available C++ objects: classes

Example: class hierarchy for some joint types in our Chrono::Engine software
C++ transient database

- Run-time object database: bodies, links, etc.
- Smart shared pointers are used.
Example of workflow
Chrono::SolidWorks

- Chrono::SolidWorks is an *add-in* for 3D CAD software:
  - Expands SolidWorks with new buttons, tools
  - Export a mechanism into a .PY file
  - Load the system in a C++ simulator
Chrono::SolidWorks
Example
Units

The following slides show an overview of the main units, with ongoing work
Cosimulation unit
Cosimulation unit
Python unit

Python modules for using Chrono::Engine from Python

a Python parser to use .py files in C++ programs
Python unit

- C++ parsing utilities to call Python from C::E
- Python stand-alone module to call C::E from Python!

Example:

```python
my_quat = chrono.ChQuaternionD(1, 2, 3, 4)
my_qconjugate = ~my_quat
print ('quat. conjugate =', my_qconjugate)
print ('quat. dot product=', my_qconjugate ^ my_quat)
print ('quat. product=', my_qconjugate % my_quat)
ma = chrono.ChMatrixDynamicD(4, 4)
ma.FillDiag(-2)
mb = chrono.ChMatrixDynamicD(4, 4)
mb.FillElem(10)
mc = (ma-mb)*0.1;  # operator overloading of +,-,* is supported
print (mc);
mr = chrono.ChMatrix33D()
mr.FillDiag(20)
print (mr*my_vect1);
...```
Postprocessing unit

• Based on ChAsset classes (interface agnostic)
• For batch processing in:
  • POVray
  • planned: VTK
  • ...

Chrono::Engine core library
UNIT_PostProcessing

.pov PovRay 3D rendering scripts

.bmp animation frames
FEM unit

- Will be available in next release of Chrono::Engine
- For dynamics, statics, non-linear statics, etc.
- Compatible with existing constraints, rigid bodies, etc.
- Corotational approach for beams, shells, etc.
FEM unit

- Finite element types
  - Tetrahedrons 4 nodes
  - Tetrahedrons 10 nodes
  - Hexahedrons 8 nodes
  - Hexahedrons 20 nodes
  - Springs
  - Bars
  - 3D beams

- planned:
  - Shells
  - ANCF shells/beams
  - ...
FEM unit

- 3D corotational tetrahedrons and hexahedrons
FEM unit

• Other types of analysis

• Electrostatics

\[ \nabla^2 \varphi = -\frac{\rho_f}{\varepsilon} \]
\[ \mathbf{E} = -\nabla \varphi \]
FEM unit

- Other types of analysis

- Thermal
  - steady state
  - transient
    \[
    \frac{\partial u}{\partial t} = \alpha \left( \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right) + \frac{1}{c_p \rho} q
    \]
CFD unit

- SPH:
  (under development)
Other units...

- Unit_CASCADE
- Unit_MATLAB
- Unit_MPI
- ...

Wisconsin Applied Computing Center
Embedding C::E in third party software

The following slides show examples of Chrono::Engine embedding in third party software
Embedding C::E in third party software

- SimLab Composer 2015
- **Company:** SimLab Soft - Jordan
- **Contact:** Ashraf Sultan asultan@simlab-soft.com
Embedding C::E in third party software

- **Virtual Universe PRO**
  - **Company:** IRAI - France
  - **Contact:** stephane.massart.irai@gmail.com

- Simulation of machines in the field of industrial automation
- Simulate PLC programs with a numerical model of the real machine
- Interfaces with SolidWorks, the most used 3D CAD by designers of automation machines
- Easy creation of interactive simulations of automated plants
Embedding C::E in third party software

- **Virtual Universe PRO**
  - **Company:** IRAI - France
  - **Contact:** stephane.massart.irai@gmail.com
Embedding C::E in third party software

- **Virtual Universe PRO**
- **Company**: IRAI - France
- **Contact**: stephane.massart.irai@gmail.com
Other tools in Project Chrono
Chrono::Parallel

https://github.com/projectchrono/chrono-parallel
Chrono-Parallel Overview

- Uses the Chrono API with minor modifications
  - Supports contact, friction, cohesion, compliance
  - Supports all bilateral constraint types
- Uses OpenMP for parallelism
- Both Matrix and Matrix-Free approaches
- Can scale to millions of bodies
Chrono-Parallel Features

- Custom parallel collision detection
  - Spatial subdivision broadphase
  - Narrowphase dispatch
    - Minkowski portal refinement
    - Separating Axis Theorem

- Supports DVI and DEM

- OpenGL GUI interface
  - Scales to millions of objects
Chrono::FLOW

https://github.com/armanpazouki/chrono-fluid
The Lagrangian-Lagrangian angle

- Fluid dynamics: Smoothed Particle Hydrodynamics (SPH)

- Solid bodies:
  - 3D rigid body dynamics
  - Absolute Nodal Coordinate Formulation (ANCF) for flexible bodies

- Lagrangian-Lagrangian approach attractive since:
  - Consistent with Lagrangian tracking of discrete solid component:
  - Hassle-free simulation of free surface flows prevalent in target applications
  - Maps well to the GPU parallel computing model
Interacting rigid and flexible objects in channel flow

Fluid:
\[
\begin{align*}
\rho &= 1000 \text{ kg/m}^3 \\
\mu &= 1 \text{ N s/m}^2 \\
(l_x, l_y, l_z) &= (1.4, 1, 1) \text{ m} \\
Re &= 45
\end{align*}
\]

Ellipsoids:
\[
\begin{align*}
\rho_s &= 1000 \text{ kg/m}^3 \\
(a_1, a_2, a_3) &= (2.25, 2.25, 3) \text{ cm} \\
N_r &= 2000 \\
Re_p &= 2
\end{align*}
\]

Beams:
\[
\begin{align*}
\rho_s &= 1000 \text{ kg/m}^3 \\
E &= 0.2 \text{ MPa} \\
a &= 1.5 \text{ cm} \\
l &= 64 \text{ cm} \\
N_f &= 40 \\
n_e &= 4
\end{align*}
\]
Chrono::FLEX

https://github.com/uwsbel/implicit-beams-gpu
Chrono::FLEX

- Flexible multibody dynamics: Absolute Nodal Coordinate Formulation (ANCF)
- Used for the dynamics of flexible bodies that undergo large deformation
- ANCF approach attractive since:
  - Consistent with the nonlinear theory of continuum mechanics
  - Zero Coriolis and centrifugal effects
  - Maps well to the GPU parallel computing model
Chrono::FLEX Examples

Beam elements in contact

Plate elements in contact

Beam elements w/ constraints

Plate elements w/ constraints
Chrono::Render
Post-processing

• 3 Levels of visual fidelity
  • Run-time: Synchronized with simulation loop
  • Debug: Debug quality rendering to images/video
  • Presentation: High quality rendering to images/video

• C++ API for each level
Run-time animation

- Irrlicht engine
  - Open Source
  - Cross-Platform
    - OpenGL
    - DirectX
  - Interactive GUI
  - Primitives
  - Meshes
Rendering pipeline

- Persistence Of Vision Ray-tracer (POVRay)
- Open source ray-tracing solution
- Scriptable interface
- Support for global lighting techniques
- Easily scales to millions of objects
  - Analytical representation of simple convex shapes
  - CPU Parallel
- Provide scripts tailored to specific types of simulations
  - Granular, Fluid, Vehicles, Tire Terrain interaction
- Render at different levels of quality
  - Debugging, presentation quality, etc
- Different data formats depending on needs
  - POVRay requires ASCII
Debug animation

- Low quality render of simulation
- Utilizes POV-Ray
- Example 1 Thread:
  - 48s Preprocess
  - 4s Render
Presentation animation

- High quality render of simulation
- Utilizes POV-Ray
- Example 1 Thread:
  - 48s Preprocess
  - 602s Render
CHRONO::VEHICLE OVERVIEW
What is Chrono::Vehicle?

• **Chrono vertical app** (unit) for the modeling, simulation, and visualization of wheeled ground vehicles

• **Middleware**: can be embedded in third parties software

• **Open source** with BSD license

• Library developed in C++

• **Cross-platform**: compiles on GNU GCC, MSVC, etc

• **Dependencies**: Chrono::Engine and (optionally) the Chrono UNIT_Irrlicht
What is Chrono::Vehicle?

• Auxiliary systems for testing vehicle systems in a co-simulation framework:
  • Tire system (rigid, LuGre, Pacejka)
  • Terrain (height-map)
  • Powertrain (engine + TC + transmission)
  • Driver model (interactive, data-based)
What is Chrono::Vehicle?

- **Modular**: vehicle are modeled from instances of subsystems (suspension, steering, driveline, etc.)
- **Flexible**: use parameterized templates
- **Expandable**, via C++ inheritance
  - New subsystems
  - New templates for existing subsystems
  - New vehicle types (e.g. tracked)
Code design – templates

- Template-based modeling (not in the C++ sense)

- In Chrono::Vehicle, templates are parameterized models that define a particular implementation of a subsystem type:
  
  - Define the basic Chrono modeling elements (bodies, joints, force elements, etc.)
  - Impose the subsystem topology (connectivity)
  - Define the template parameters
  - Implement common functionality for the type of subsystem (e.g. ‘suspension’), particularized to the specific template (e.g. ‘double-wishbone’)

Code design – class hierarchy

• Chrono::Vehicle encapsulates templates for systems and subsystems in polymorphic C++ classes:
  • A base abstract class for the system/subsystem type (e.g. ChSuspension)
  • A derived, still abstract class for the system/subsystem template (e.g. ChDoubleWishbone)
  • Concrete class that particularize a given system/subsystem template (e.g. HMMWV_DoubleWishboneFront)

• Concrete classes:
  • User-defined – a derived class that satisfies all virtual functions imposed by the inherited template class
    • not part of the Chrono::Vehicle library
    • several example concrete classes and demo programs are provided
  • Generic – a derived class that satisfies all required virtual functions using parameter data from a specification file
    • part of the Chrono::Vehicle library
    • specification files use the JSON format
Chrono::Vehicle subsystem templates

- Suspension subsystem
  - Double wishbone
  - Multi-link
  - Solid-axle

- Steering subsystem
  - Pitman arm
  - Rack and pinion

- Driveline
  - 4WD
  - 2WD

- Brake

- Wheel
Run-time visualization with Irrlicht
Post-processing visualization with PovRay
Debug and Presentation Animations
CHRONO VALIDATION

- Validation of modeling building blocks (against ADAMS and analytical solutions)
  - Validations at subsystem and system level (Chrono::Vehicle)
  - Validation against experimental data (large-scale contact problems)
Validation: Modeling building blocks

• Multiple test cases per joint/constraint/force were created based on simple mechanisms to exercise each component.

• 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 To Date

- **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)
Example Validation Comparisons – 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
Example Validation Comparisons – Distance Constraint

Z Acceleration - Distance_Case03

Z Reaction Force - Distance_Case03

Distance_Case03 - Simulation Difference

Distance_Case03 - Simulation Difference
Validation: Subsystem and system level

- Quasi-static simulations of vehicle suspensions and other subsystems are performed to validate the solver and tune model parameters.
- Example: How does the double wishbone settle due to gravity?
Purpose/Motivation? A full vehicle’s handling is very sensitive to small changes in combined suspension/steering configurations.

Goal? Analyze combined front suspension/steering subsystem behavior without involving complex full vehicle dynamics.

Examples for a statically loaded vehicle:
1. Shock spring prelength/preload
2. Steering wheel input limits
Suspension Test Rig Overview

• Similar to a full vehicle model
• Front half, chassis fixed to ground
• Shaker posts added to the system  
  • (Left=Green, Right=Red)
• Constrained to move only vertically (global)
• Linear Actuator specifies post displacement
• Wheel spindle body CM point constrained in plane  
  • Plane vertically offset from shaker post surface
• Logs important measurements to console or file
Validation: Experimental data

- Several models involving large-scale contact are validated against experimental data
Granular Flow Experimental Validation

Simulation results – flow measurement

- Total mass of granular material: 6.38 g
- Width of opening: 9.398 mm
- Opening speed: 1 mm/s
- Maximum opening gap: 2 mm
Experimental Validation: Cratering

- Experiment:
  - Measured penetration of spherical projectiles into loose non-cohesive granular media
  - Parameters: $\rho_b$, $D_b$, $h$, $\rho_g$, $\mu$

- Simulations:
  - $\rho_g = 1.51 \text{ g/cm}^3$, $D_b = 2.54 \text{ cm}$, $\mu = 0.3$, and 1 mm grains
  - $h = \{5, 10, 20\} \text{ cm}$, $\rho_b = \{0.28, 0.7, 2.2\} \text{ g/cm}^3$


**Cratering Validation**

**Complementarity**
- 500,400 spheres (1 mm diam.)
- step-size: $2 \cdot 10^{-5} \text{s}$
- $\zeta = 0.1221$

**Penalty**
- 379,290 spheres (1 mm diam.)
- step-size: $10^{-5} \text{s}$
- $\zeta = 0.1290$

\[
d = \frac{\zeta}{\mu} \left( \frac{\rho_b}{\rho_g} \right)^{1/2} D_b^{2/3} H^{1/3}
\]
Direct Shear Test

- **Direct Shear Test**: measures the shear properties of the soil
  - Soil sample is contained between two rigid plates that are held in place
  - The shear box is aligned under a load cell that applies a desired normal load
  - The load cell is attached to a translational joint that uses a linear variable differential transformer to measure shear displacement of the soil
  - The force required to displace the soil horizontally is measured by a dynamometer in the horizontal direction

Experimental device for performing the direct shear test.

The direct shear process.
Direct Shear Test – DEM Validation

• The higher normal loading scenarios correlate well with the experimental data

DEM Parameters:
- Shape: Dodecahedron
- Radius: 2 mm
- Density: 2.6 g/cm³
- Friction Coef: 0.5

Experimental and simulation data for the direct shear test.
CHRONO SOFTWARE, DOCUMENTATION, AND SUPPORT
Code availability

- Project Chrono is hosted on GitHub, a Git repository web-based hosting service
  - Offers distributed revision control and source code management
  - Provides a web-based graphical interface, as well as desktop/mobile integration

- Chrono: https://github.com/projectchrono/chrono.git
- Chrono-T: https://github.com/uwsbel/chrono-T.git
Library and tool dependencies

• Chrono is installed using the free **CMake** utility
  • Manages the build process in an operating system and compiler-independent manner

• Depending on the features (called “units”) that are enabled, Chrono requires additional libraries

• Units are additional libraries that can be *optionally* used to expand the features of Chrono:
  • Irrlicht 3D visualization: Requires the Irrlicht library
  • Matlab interoperation: Requires the Matlab library
  • Automated postprocessing: Requires the POV-Ray Rendering Tool
  • Python scripting: Requires the Python Software Development Kit
Configuration and build

• In order to develop C++ applications based on the Chrono::Engine SDK, the following must be performed:

1. Check/Install a C++ compiler
2. Install CMake
3. Install Git
4. Download the project by cloning the Git repository
5. Download the Irrlicht library
6. Run CMake
7. Compile the project
8. Play with the demos

NOTE: Detailed instructions can be found at [http://projectchrono.org/](http://projectchrono.org/)
Doxygen Documentation

• Project Chrono is documented via Doxygen, a tool for writing software reference documentation
  • Documentation is written within the code and automatically published to a webpage
  • Provides an easy way to keep the documentation up to date and can cross reference documentation and code

• Chrono: http://api.chrono.projectchrono.org/
The Project Chrono webpage (http://projectchrono.org/) provides thorough documentation, including:

- Installation instructions
- Tutorials and examples
- API documentation (via Doxygen)
- Scientific papers on Chrono::Engine

Additionally, a mailing list and forum have been set up using Google Groups for questions and support.
Chrono demos – basic features

- demo_math
  Tutorial on using Chrono mathematical objects and functions (vector math, matrices, linear algebra, etc.)

- demo_cords
  Tutorial on how to perform 3D coordinate manipulation (rotation and translations of points, frames, etc.)

- demo_stream
  Tutorial on files, streams, serialization, etc.

- demo_sharedptr
  Tutorial on using smart and shared pointers.
Chrono demos – basic features

- **demo_buildsystem**
  Tutorial on the basic approach to building and simulating mechanical systems (creating a physical system, adding/removing bodies, creating joints, performing a simulation)

- **demo_powertrain**
  Tutorial on the basic approach to using 1-DOF items (rotating shafts)

- **demo_chfunctions**
  Tutorial on using ChFunction to create and use $y=f(x)$ objects

- **demo_postprocess**
  Tutorial on UNIT_POSTPROCESS – creating animations with PovRay
Chrono demos with the Irrlicht 3D interface

- **demo_crank**
  Create constraints and ‘engine’ objects; create a real-time application

- **demo_fourbar**
  Extract and plot data from simulation; using a direct solver

- **demo_collision**
  Collision, contact, and friction; specifying contact geometry

- **demo_bricks**
  Adjusting time-stepper settings; creating a motor between two parts

- **demo_pendulum**
  Using custom forces; specifying joint limits

- **demo_gears**
  Using kinematic gears and pulleys
Chrono demos with the Irrlicht 3D interface

- **demo_mecanum**
  Create a complex model; using the keyboard for interactive simulation

- **demo_friction**
  Using spinning and rolling friction

- **demoSuspension**
  Create a simplified wheeled vehicle; using distance constraints

- **demo_tracks**
  Create a simplified track vehicle; using collision object families

- **demo_irr_assets**
  Using visualization assets in conjunction with Irrlicht

- **demo_import_solidworks**
  Loading a mechanism exported from SolidWorks
Chrono demos with the Irrlicht 3D interface

- **demo_ballDEM**
  Simple demonstration of penalty method for frictional contact

- **demo_collisionDEM**
  Using penalty method for frictional contact

- **demo_aux_ref**
  Demonstration of using ChBodyAuxRef

- **demo_rev_sph**
  Using the revolute-spherical composite joint

- **demo_univ**
  Using the universal joint

- **demo_spring**
  Using linear spring-dampers with ChLinkSpring and ChLinkSpringCB
USING CHRONO FOR MBD – BASIC FEATURES
Introduction to API

- C++ based API
- Examples in src/demos/...
- Build toolchain based on CMAKE
- Most interactive demos use unit_IRRLICHT.
- Infos on the WIKI in www.projectchrono.org
CMAKE build toolchain (essentials)

Example:

**Chrono SDK** (chrono/src/..)

- **Includes:**
  - ChPhysicsItem.h
  - ChBody.h
  - ChSystem.h
  - ...

**Chrono build directory**

- **Libraries:** (.../lib/..)
  - ChronoEngine.lib
  - ChronoEngine_IRRLICHT.lib
  - ChronoEngine_POSTPROCESS.lib

- **DLLs:** (.../bin/..)
  - ChronoEngine.dll
  - ChronoEngine_IRRLICHT.dll
  - ChronoEngine_POSTPROCESS.dll

**Your project**

- **sources:**
  - my_program.cpp
  - ...

**Your project build directory**

- **Libraries:** (.../lib/..)
  - ChronoEngine.dll
  - ChronoEngine_IRRLICHT.dll
  - ChronoEngine_POSTPROCESS.dll

- **Compile:** my_program.exe

**Build Chrono**

- **Find and include:** ChPhysicsItem.h, ChBody.h, ChSystem.h

**Build your project**

- **Compile:** my_program.cpp

**Move into same directory**

- Move ChronoEngine.dll, ChronoEngine_IRRLICHT.dll, ChronoEngine_POSTPROCESS.dll into same directory as your executable.
CMAKE build toolchain (essentials)

PROJECT(CoronaElectrostaticSeparator)

set(CMAKE_MODULE_PATH ${CMAKE_MODULE_PATH} "${CMAKE_SOURCE_DIR}/cmake/")

# Use the find_package(ChronoEngine…) to define useful macros, paths, and to select C::E units:
find_package(ChronoEngine COMPONENTS unit_IRRLICHT unit_PYPARSER unit_POSTPROCESS)

# After the ChronoEngine package has been found, you can add its include directories with the headers.
INCLUDE_DIRECTORIES( ${CHRONOENGINE_INCLUDES} )

# Set c++ sources for building the exe, as usual in CMake
ADD_EXECUTABLE(conveyor source/conveyor_main.cpp source/conveyor_main.h […] source/conveyor_utils.h )

# This will cause the linker to link Chrono::Engine main lib and units to your projects
TARGET_LINK_LIBRARIES(conveyor ${CHRONOENGINE_LIBRARIES} )
CMAKE build toolchain (essentials)

CMakeLists.txt

Xxxyyyyyzzz.sln

my_program.exe
Coding with shared pointers

- Most «complex» objects in C::E are managed via shared pointers
- Shared pointers frees you from the need of delete()
- ChSharedPtr<> is for objects inherited from ChShared
- ChSmartPtr<> is for whatever object
- Syntax:

  ```
  ChSharedPointer<MyClass> my_body(new MyClass);
  ```
Coding with shared pointers

• Example:

```cpp
ChSharedPointer<ChBody> my_body(new ChBody);
my_body->SetMass(100);
...
```

instead of typical:

```cpp
ChBody* my_body = new ChBody;
my_body->SetMass(100);
...
delete my_body;
```
Coding with shared pointers

Casting between shared pointers:

ChSharedPtr<cTestA> pA(new cTestA);
ChSharedPtr<cTestB> pB(new cTestB);
ChSharedPtr<cTestC> pC(new cTestC);

// Test : convert a shared pointer from a CHILD class to a PARENT class (UPCASTING):
ChSharedPtr<cTestA> pAn(pB);    // OK! because cTestA is base class of cTestB, upcasting is automatic
ChSharedPtr<cTestA> qAn = pB;   // OK! another way of doing the same...

// Test : convert a shared pointer from a PARENT class to a CHILD class (DOWNCASTING):
// ChSharedPtr<cTestB> qBn = pA;  // NO! compile-time error! "Cast from base to derived requires dynamic_cast.."

// Downcasting is possible anyway via "dynamic casting", as in c++ pointers.
// Here pAn is a cTestA pointer to a cTestB obj, so it works:
ChSharedPtr<cTestB> pBn (pAn.DynamicCastTo<cTestB>());
if (pBn)
    GetLog() << "Test: DynamicCastTo pAn->pBn was successful \n";

ChSharedPtr<cTestB> pBs = pAn.StaticCastTo<cTestB>();  // OK downcasting. Correctness is up to you.
ChSharedPtr<cTestA> pAs = pB.StaticCastTo<cTestA>();    // OK upcasting. But superfluous. Doing ... = pB; was enough.
Coordinate transformations

ChVector<> \( p = \{p_x, p_y, p_z\} \)

```cpp
ChVector<double> mvect1(2,3,4);       // create a vector with given x,y,z ‘double’ components
ChVector<float>  mvect2(4,1,2);       // create a vector with given x,y,z ‘float’ components
ChVector<>        mvect3();           // create a 0,0,0, vector. The <> defaults to ‘double’
ChVector<>        mvect4(mvect1 + mvect2); // create a vector by copying another (a result from +)

mvect3 = mvect1 + mvect2;             // vector operators: +, -
mvect3 += mvect1;                     // in-place operators
mvect3 = mvect2 * 0.003;             // vector product by scalar
mvect3.Normalize();                  // many member functions...
mvect3 = mvect1 % mvect2;            // Operator for cross product: A%B means vector cross-product AxB
double val = mvect1 ^ mvect2;        // Operator for inner product (scalar product)
```
Coordinate transformations

ChQuaternion<>

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

```cpp
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);
```
Coordinate transformations

ChQuaternion<>

mvect2 = vtraslA + qrotA.Rotate(mvect1);     /// use Rotate() to rotate a vector

/// quaternion product via operator *
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  (same as before!)
qa.Rotate(mvect1);
Coordinate transformations

ChCoordsys<>
• represents a translation and a rotation \( c = \{ d, q \} \)
• rotation is a quaternion

ChFrame<>  
• a more ‘powerful’ version of ChCoordsys
• contains also a ChMatrix33<> to speedup some formulas
Coordinate transformations

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
Coordinate transformations

ChFrame<>

- ChFrame<> can transform points in space
- Two alternative specular 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; // otherwise...
d_Paa = d_Pbb >> X_ba;
Coordinate transformations

ChFrame<>

- Also ChFrame can be transformed
- Build sequence of transformations

ChFrame<> X_{ba}, X_{cb}, X_{ca};

...  

X_{ca} = X_{ba} * X_{cb}; // otherwise...

X_{ca} = X_{cb} >> X_{ba};
Coordinate transformations

ChFrameMoving<>

- Inherits ChFrame<> functionality
- Adds information on velocity and acceleration:
  \[ c = \{ p, q, \dot{p}, \dot{q}, \ddot{p}, \ddot{q} \} \]
- Alternative: angular velocity and acceleration instead of q derivatives:
  \[ c = \{ p, q, \dot{p}, \omega, \dot{\omega} \} \]
Coordinate transformations

ChFrameMoving<>

- Example:

```plaintext
ChFrameMoving<> X_Ba;
X_Ba.SetPos(ChVector<>(2,3,5));
X_Ba.SetRot(myQuaternion);

// 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
```

\[ \mathbf{c} = \{ p, q, \dot{q}, \omega, \ddot{p}, \alpha \} \]
Coordinate transformations

**ChFrameMoving<>**

- ChFrameMoving (and ChVector, ChFrame) can be transformed
- Same * or >> operators as in ChFrame...
- ... but also speeds and accelerations are automatically transformed!

\[ X_{ca} = X_{ba} \times X_{cb}; \] // otherwise...

\[ X_{ca} = X_{cb} >> X_{ba}; \]

**ChVector<>** \( w_{ca} = X_{ca}.GetWvel\_rel(); \) // example...
Coordinate transformations

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

ChVector<> a_03 = X_30.GetPos_dtdt();
Coordinate transformations

ChFrameMoving<>

- The GetInverse() and Inverse() functions:

\[
\text{ChFrameMoving<> X}_{10}, \text{X}_{21}, \text{X}_{32}, \text{X}_{43}, \text{X}_{54}, \text{X}_{65}, \text{X}_{70}, \text{X}_{87}, \text{X}_{86};
\]

...  

// 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:

\[
X_{86} = X_{87} >> X_{70} >> (X_{65} >> X_{54} >> X_{43} >> X_{32} >> X_{21} >> X_{10}).\text{GetInverse}();
\]
Rigid bodies

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 in collisions
Rigid bodies

ChBody

Important steps for each rigid body:

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

ChBody

// Create a body - use shared pointer!
ChSharedPtr<ChBody> body_b(new 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);
Rigid bodies

ChBodyAuxRef

• Inherited from ChBody

• Used when the COG is not practical as a main reference for the body, and another reference is preferred, ex. from a CAD, so it adds an auxiliary REF frame.

• The REF frame is used for
  • Collision shapes
  • Visualization shapes
Rigid bodies

ChBodyAuxRef

// Create a body with aux.reference
ChSharedPtr<ChBodyAuxRef> body_b(new 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
// ex. body_b->GetPos()  body_b->GetRot()  etc.
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 (couple of ChMarker from two bodies)

```cpp
ChSharedPtr<ChMarker> marker_c(new 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
- Hint: for simple ready-to-use bodies that already contain collision shapes, use ChBodyEasySphere, ChBodyEasyBox, etc.
Collision shapes

• Typical steps to setup collision:

```cpp
body_b->GetCollisionModel()->ClearModel();
body_b->GetCollisionModel()->AddSphere(myradius);
...
body_b->GetCollisionModel()->BuildModel();
body_b->SetCollide(true);
```

• Collision ‘families’ for selective collisions:

```cpp
// default collision family is 0. Change it:
body_b->GetCollisionModel()->SetFamily(2);
body_b->SetFamilyMaskNoCollisionWithFamily(4);
```
Collision material

• Easy but memory-consuming approach:

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

• Advanced approach with a shared material:

```cpp
// Create a surface material and change properties:
ChSharedPtr<ChMaterialSurface> mat(new 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);
...```

Collision tolerances

- Set these tolerances before creating collision shapes:

  ChCollisionModel::SetDefaultSuggestedEnvelope(0.001);
  ChCollisionModel::SetDefaultSuggestedMargin (0.0005);
  ChCollisionSystemBullet::SetContactBreakingThreshold(0.001);

**Too large collision envelope**: too many potential contacts, high CPU time, high waste of RAM

**Too small collision envelope**: risk of tunnelling effects, unstable simulation of stacked objects

**Too large collision margin**: shapes are ‘rounded’ too much

**Too small collision margin**: when interpenetration occurs beyond this value, an inefficient algorithm is used
Visualization assets

ChAsset
ChVisualization
ChSphereShape
ChCylinderShape
ChBoxShape
...

• Unlimited visualization assets can be attached to a body

• The position and rotation are defined respect to REF frame

• Visualization assets are used by postprocessing systems and by realtime 3D interface
Visualization assets

• Example: add a box

ChSharedPtr<ChBoxShape> mbox (new ChBoxShape);
mbox->GetBoxGeometry().Pos = ChVector<>(0,-1,0);
mbox->GetBoxGeometry().Size = ChVector<>(10,0.5,10); body_b->AddAsset(mbox);

• Example: add a texture

ChSharedPtr<ChTexture> mtexture (new ChTexture);
mtexture->SetTextureFilename(GetChronoDataFile("bluwhite.png")); body_b->AddAsset(mtexture);
Visualization assets

• Example add a mesh (just referencing an .obj file):

```cpp
ChSharedPtr<ChObjShapeFile> mobjmeshfile(new ChObjShapeFile);
mobjmeshfile->SetFilename("forklift_body.obj");
body_b->AddAsset(mobjmeshfile);
```

• Example add a mesh with triangle data:

```cpp
ChSharedPtr<ChTriangleMeshShape> mobjmesh(new ChTriangleMeshShape);
mobjmesh->GetMesh()->LoadWavefrontMesh("forklift_body.obj");
body_b->AddAsset(mobjmesh);
```
Visualization assets and unit_IRRLICHT

• After you attached usual visualization assets, do this:

```cpp
ChSharedPtr<ChIrrNodeAsset> irr_asset(new ChIrrNodeAsset);
body_b->AddAsset(irr_asset);
irr_application->AssetBind(body_b);
irr_application->AssetUpdate(body_b);
```

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

```cpp
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)
Constraints

ChLink

• Most links use two ChMarker as references
• The marker m2 (in body n.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, ex:
  • ChLinkLockRevolute: allowed DOF on z axis rotation
  • ChLinkLockPrismatic: allowed DOF on x axis translation
  • etc.
Constraints

ChLink

Important steps for each ChLink:

1. Create the link from the desired ChLinkXxxyyy class
2. Use mylink->Initialize(...) to connect two bodies
3. Add the link to a ChSystem
4. Optional: set link properties
Constraints

ChLink

// 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.
ChSharedPtr<ChLinkEngine> my_motor(new 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),
Q_from_AngAxis(CH_C_PI_2, VECT_X))
);

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

// 4- Set some properties:
my_motor->Set_eng_mode(ChLinkEngine::ENG_MODE_SPEED);
if (ChSharedPtr<ChFunction_Const> mfun = my_motor->Get_spe_funct().DynamicCastTo<ChFunction_Const>())
  mfun->Set_yconst(CH_C_PI/2.0); // speed w=90°/s
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.
Build 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
ChSharedPtr<ChBody> my_body_A(new ChBody);
my_system.AddBody(my_body_A);
my_body_A->SetBodyFixed(true); // truss does not move!

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

// ..the rod
ChSharedPtr<ChBody> my_body_C(new ChBody);
my_system.AddBody(my_body_C);
my_body_C->SetPos(ChVector<>(4,0,0)); // position of COG of rod
Build a system - example (2/3)

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

// .. a revolute joint between crank and rod
ChSharedPtr<ChLinkLockRevolute> my_link_BC(new 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
ChSharedPtr<ChLinkLockPointLine> my_link_CA(new 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
ChSharedPtr<ChLinkEngine> my_link_AB(new 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);
Build a system - example (3/3)

// 4- Adjust settings of the integrator (optional):
my_system.SetIntegrationType(ChSystem::INT_ANITESCU)
my_system.SetLcpSolverType(ChSystem::LCP_ITERATIVE_SOR);
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
    [..]
}
Main system parameters

```cpp
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 (ex for numerical errors, incoherent initial conditions, etc.) do not ‘escape’ one from the other faster than this threshold.
The higher, the faster and more precisely are recovered the contact constraints errors (if any), 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 the other when the integrator precision is low (ex 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 is to get stable simulations, but the less realistic the physics of the collision.
HANDS-ON DEMO

MODELING, SIMULATION, AND VISUALIZATION
OF A SLIDER-CRANK MECHANISM
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

crank

distance constraint

revolute

prismatic

slider

revolute

prismatic
Initial configuration
Defining a body

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

```cpp
// Crank
ChSharedPtr<ChBody> crank(new 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

```cpp
ChSharedPtr<ChBoxShape> box_c(new ChBoxShape);
box_c->GetBoxGeometry().Size = ChVector<>(0.95, 0.05, 0.05);
crank->AddAsset(box_c);

ChSharedPtr<ChCylinderShape> cyl_c(new 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);

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

ChSharedPtr<ChColorAsset> col_c(new 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

```cpp
// 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.
ChSharedPtr<ChLinkRevolute> revolute_ground_crank(new 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

```cpp
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

- **Crank and Rod**: A set of interconnected elements where the crank rotates within a fixed plane, and the rod moves linearly along a fixed axis.

- **Slider**: A mechanism where the linear movement of a rod is constrained to a fixed plane, allowing for back-and-forth motion.

- **Universal Joint**: Allows motion in all three dimensions (xyz), providing rotational and translational freedom.

- **Spherical Joint**: Enables 360-degree rotation in all three dimensions, allowing for complex articulation.

- **Prismatic Joint**: Allows linear movement along a single axis, often used in linear actuators.

- **Rotational Driver**: A component designed to rotate in a specific plane, commonly used in motorized systems for rotation.

These joints and frames are fundamental in mechanical engineering, robotics, and manufacturing to achieve the necessary motion and control in devices.
Initial configuration
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(ChSharedPtr<ChMarker> mark1,///< first marker to join
                       ChSharedPtr<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(ChSharedPtr<ChBody> mb1,///< first body to join
                       ChSharedPtr<ChBody> mb2, ///< 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(ChSharedPtr<ChBody> mb1,///< first body to join
                       ChSharedPtr<ChBody> mb2, ///< 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

void Initialize(
    ChSharedPtr<ChBodyFrame> body1, ///< first body frame
    ChSharedPtr<ChBodyFrame> body2, ///< second body frame
    const ChFrame<> & frame,      ///< joint frame (in absolute frame)
);

void Initialize(
    ChSharedPtr<ChBodyFrame> body1, ///< first body frame
    ChSharedPtr<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

```c
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
};
```

```c
void Set_rot_funct(ChSharedPtr<ChFunction> mf) { rot_funct = mf; }
void Set_spe_funct(ChSharedPtr<ChFunction> mf) { spe_funct = mf; }
void Set_tor_funct(ChSharedPtr<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.

```cpp
#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"
```

```cpp
/// 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;}
```
Rotational driver: ChLinkEngine (solution)

// Create a ChFunction object that always returns the constant value PI/2.
ChSharedPtr<ChFunction_Const> fun(new 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.
ChSharedPtr<ChLinkEngine> engine_ground_crank(new 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);
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

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

```cpp
/// Acess 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

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

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

```cpp
/// Add a sphere shape to this model, for collision purposes
virtual bool AddSphere (double radius,
const ChVector<> & pos = ChVector<>()) {return 0;}
```

```cpp
/// Add a box shape to this model, for collision purposes
virtual bool AddBox (double hx,
double hy,
double hz,
const ChVector<> & pos = ChVector<>(),
const ChMatrix33<> & rot = ChMatrix33<>(1)) {return 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.

```cpp
void Initialize(ChSharedPtr<ChBody> mbody1, /// first body to link
                ChSharedPtr<ChBody> mbody2, /// second body to link
                bool pos_are_relative, /// true: following pos. are 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.)
);```

```cpp
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; }
```
Create body
Specify mass properties
Specify initial conditions
Specify contact geometry
Specify visual assets
Translational spring-damper: ChLinkSpring (solution)

```cpp
//// 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.
////

ChSharedPtr<ChLinkSpring> tsda_ground_ball(new 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);
```