Chrono::FEA

API

Chrono::FEA

- Chrono::FEA is a C++ module that enables flexible parts in Chrono

- Chrono::FEA contains classes for modeling finite elements of various types:
  - Beams
  - Shells
  - Tetrahedrons, hexahedrons

- Dependencies:
  - Chrono::Engine main module (required)
  - Chrono::Irrlicht and the Irrlicht library and dependencies (optional)
  - Chrono::MKL (optional, but strongly suggested)

Code organization

FOLDER	CONTENT
src/chrono_fea	main Chrono::FEA library implementation
src/demos/fea	Various demo programs (main drivers)

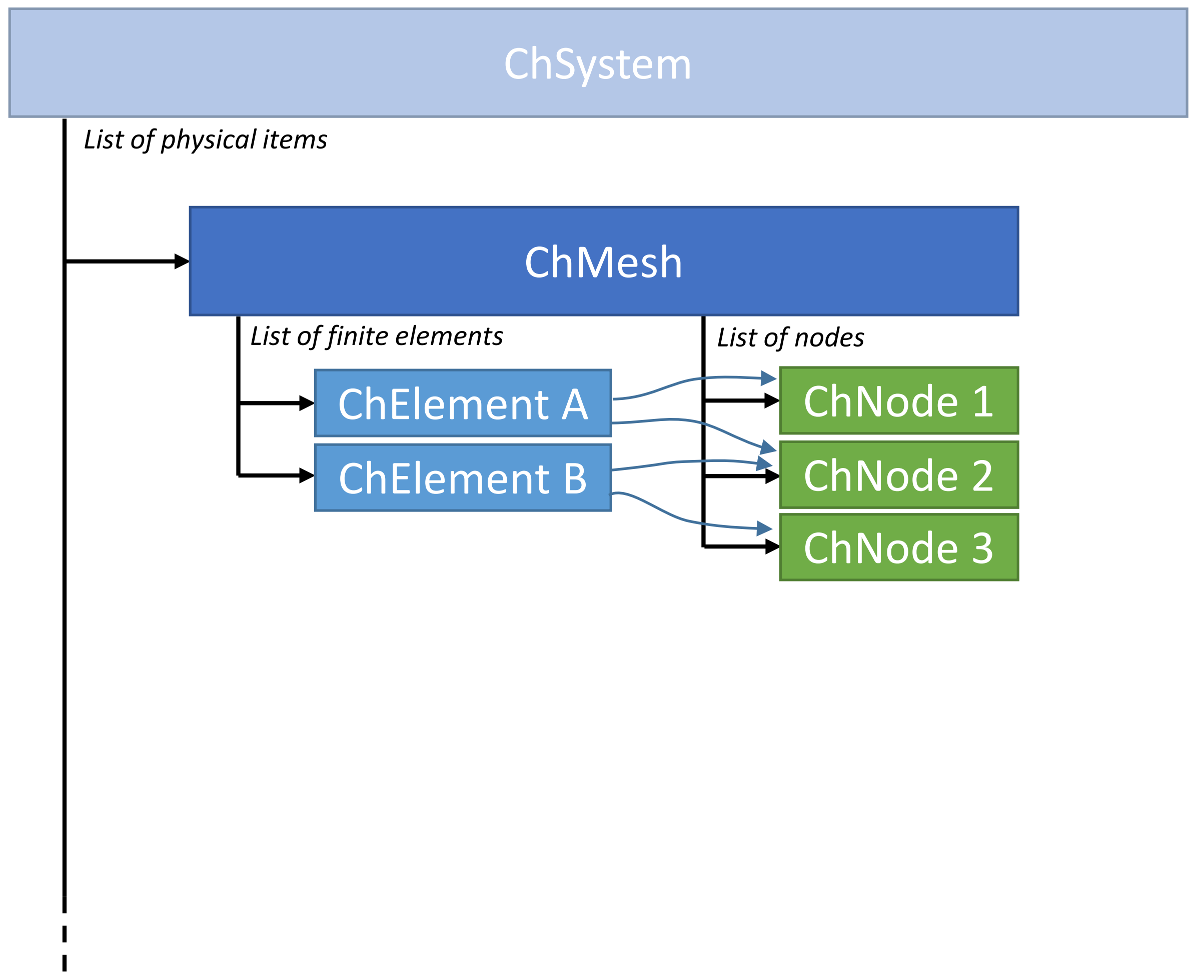
Code organization – demos

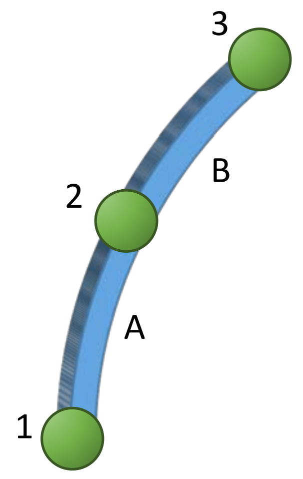
FOLDER	CONTENT
demo_FEA_basic	Simplest example for learning nodes, elements and meshes. No GUI
demo_FEA_beams	Learn how to use Euler-Bernoulli corotational beams
demo_FEA_beams_constr	Learn how to use constraints to connect beams
demo_FEA_brick	Example showing the use of the brick element
demo_FEA_cables	Show how to use the ANCF beam element to model cables, i.e. without twisting resistance
demo_FEA_contacts	Learn how to assign contact surfaces and contact materials to FEA meshes
demo_FEA_cosimulate_granular	Advanced example of cosimulation, FEA on one side, and granular materials on the other
demo_FEA_cosimulate_load	Learn how to transfer loads to a FEA surface in a cosimulation context
demo_FEA_dynamics	Simple examples to learn how to create single elements. No GUI.
demo_FEA_electrostatics	The FEA module can be used also for basic electrostatics analysis using 3d tetrahedrons
demo_FEA_loads	Learn how to apply loads to FEA surfaces/volumes/points, and hot to make custom loads
demo_FEA_shells	Show how to make a mesh made of shells
demo_FEA_thermal	The FEA module can be used also for basic thermal analysis using 3d tetrahedrons
demo_FEA_visualize	A simple demo that shows the functionality of ChVisualizationFEAmesh to plot stresses etc.

Finite element models

How FEA data structures must be organized

Data structures for a FEA model

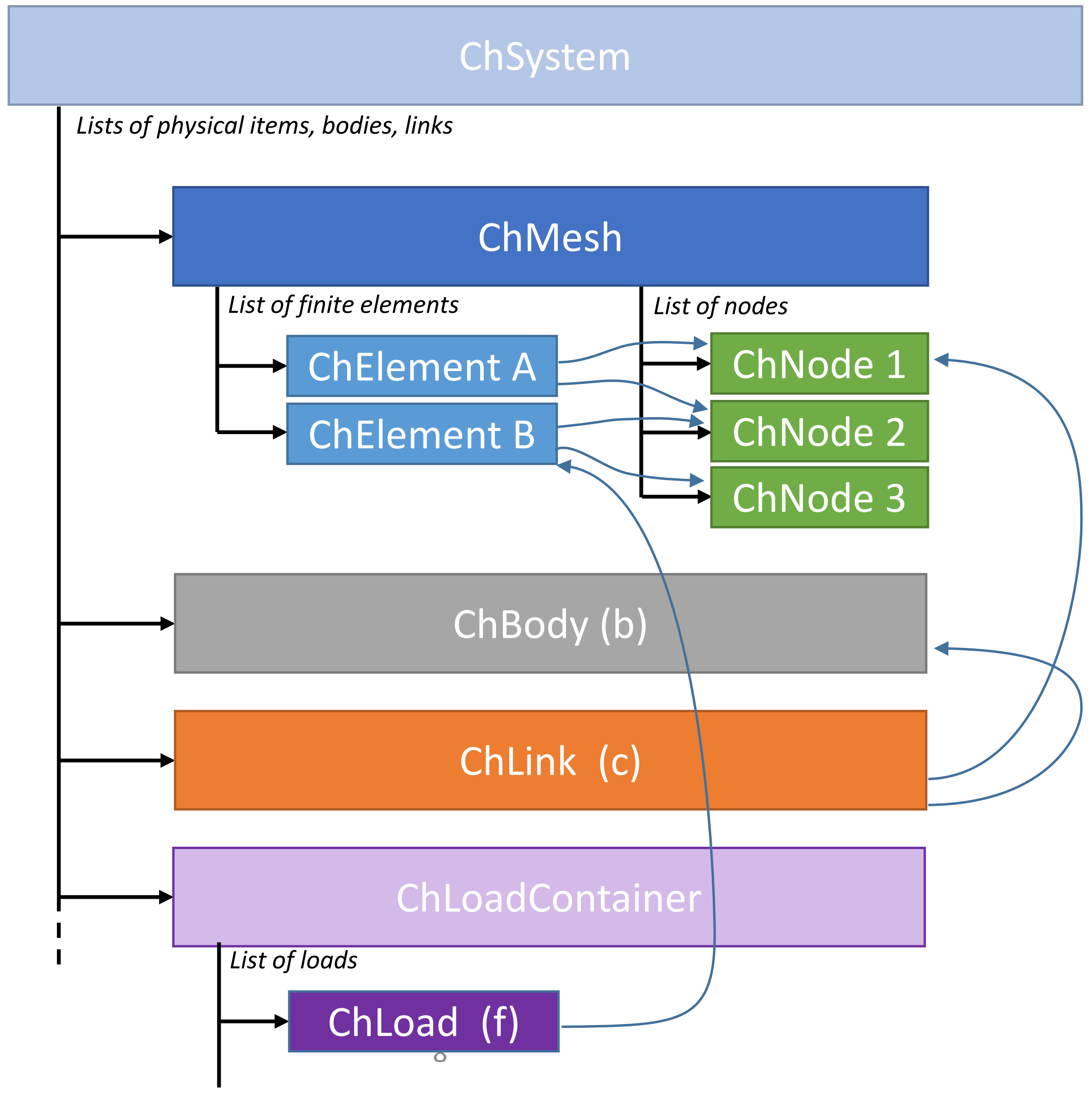


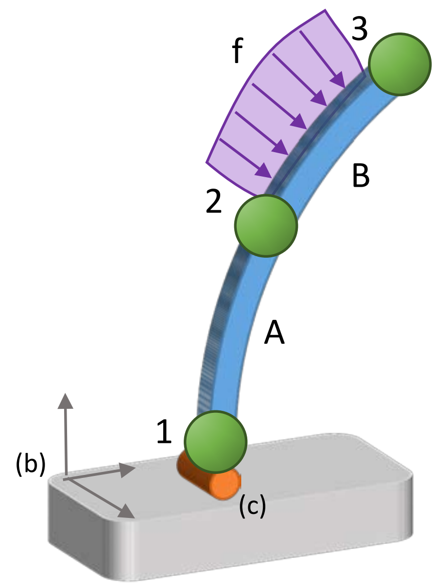


Data structures for a FEA model

- A **mesh** is a container for nodes and elements
  - Add a mesh to the system using ChSystem::Add()
  - Multiple meshes are allowed in a single system
  - Class: ChMesh
- A **node** has degrees of freedom (xyz, rotations, etc.)
  - Add nodes to a mesh using ChMesh::AddNode()
  - Classes: ChNodeFEAxyz, ChNodeFEAxyzrot, ChNodeFEAxyzP, ChNodeFEAxyzD, etc.
- An **element** connects N nodes
  - Add elements to a mesh using ChMesh::AddElement()
  - Initialize the elements by telling which node are connected with SetNodes()
  - Set a material property to the element by using SetMaterial()
  - Classes: ChElementBeamEuler, ChElementBeamANCF, ChElementCableANCF, ChElementTetra_4, ChElementTetra_10, ChElementShellANCF, ChElementShellReissner, etc.

Data structures for a FEA model





Data structures for a FEA model

- A **constraint** acts between bodies, or between nodes, or between nodes and bodies
  - Add a constraint to the system using ChSystem::Add()
  - Multiple meshes are allowed in a single system
  - Classes: ChLink and sub classes (esp. ChLinkMate)
- A **load container** contains loads applied to nodes or elements
  - Add load containers to the system using ChSystem::Add()
  - Class: ChLoadContainer.
- A **load** operates on nodes or elements (ex. pressure, gravity)
  - Add nodes to a load container using ChLoadContainer::Add()
  - Ready-to-use classes for common loads, distributed (ex. pressure) or atomic (ex.point force)
  - Custom loads can be developed by inheriting the ChLoad and ChLoader classes – more later
  - Classes: ChLoad, ChLoader, and subclasses.

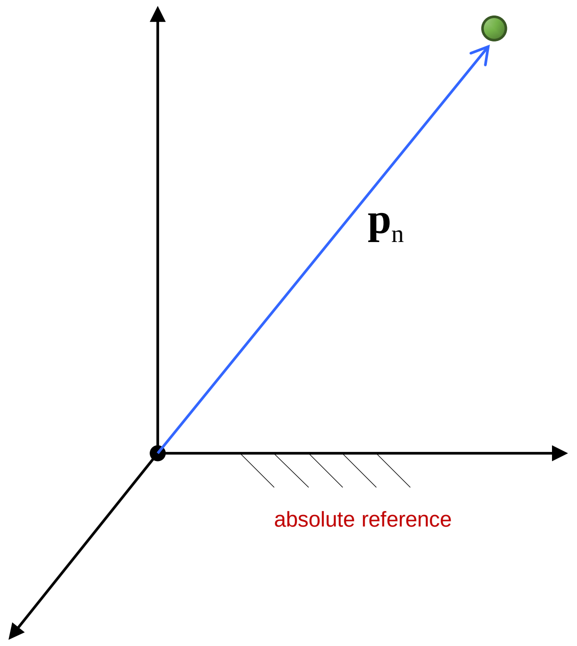
Finite element library

- Nodes
- Elements

Nodes

ChNodeFEAxyz

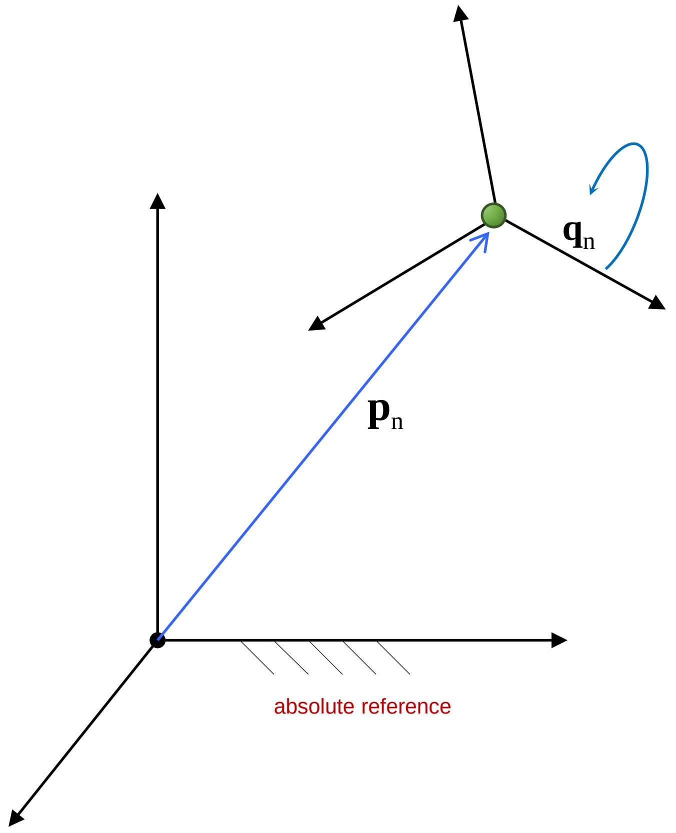
- 3 coordinates ($\mathbf{p}$, ie. x y z translation in 3D)
- E.g. Used by solid elements:
  - ChElementTetra_4
  - ChElementTetra_10
  - ChElementHexa_8
  - ChElementHexa_20
  - ChBrick_9
  - ChBrick



Nodes

ChNodeFEAxyzrot

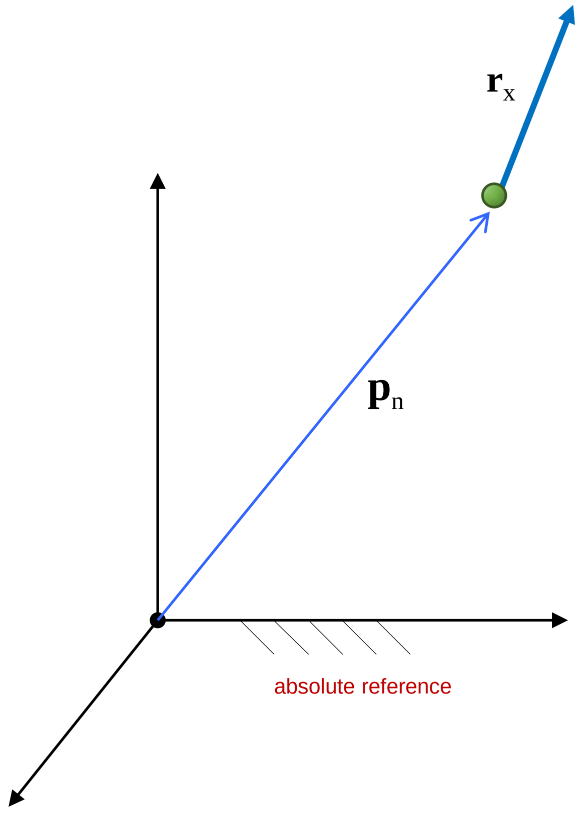
- 6 coordinates (translation $\mathbf{p}$ and rotation in 3D)
- Note: rotation expressed by quaternions $\mathbf{q}$
- E.g used by these elements (corotational formulation):
  - ChElementBeamEuler
  - ChElementShellReissner



Nodes

ChNodeFEAxyzD

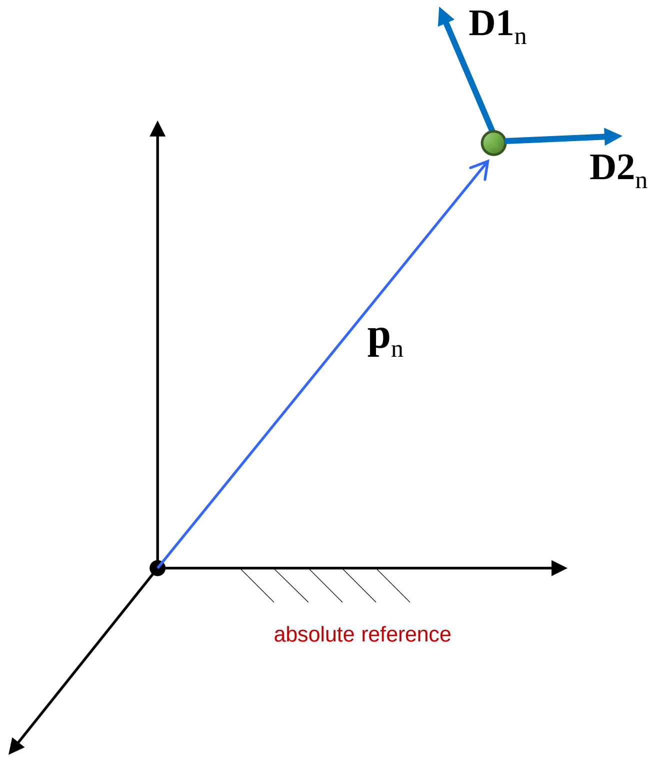
- 6 coordinates (**p** translation and Dx Dy Dz direction)
- Useful for defining simple beams of ‘cable’ type, where information about torsion is not useful
- E.g used by these elements:
  - ChElementCableANCF
  - ChElementShellANCF



Nodes

ChNodeFEAxyzDD

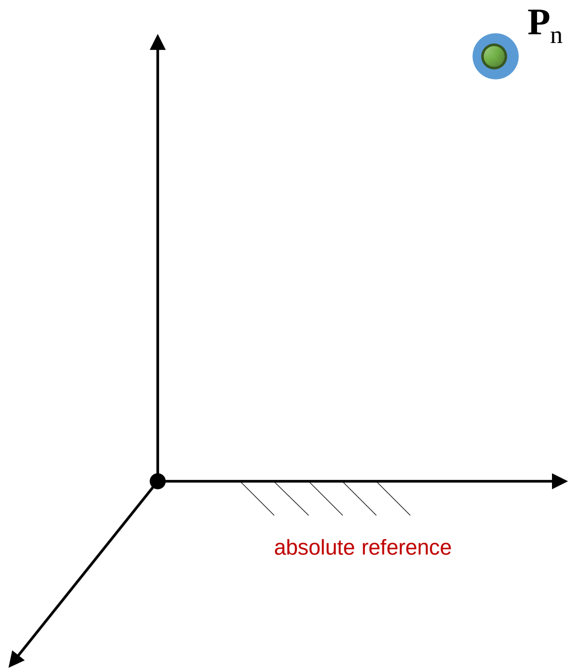
- 9 coordinates (x y z translations and two directions or one direction and a curvature vector)
- E.g used by these elements:
    - ChElementBeamANCF



Nodes

ChNodeFEAxyzP

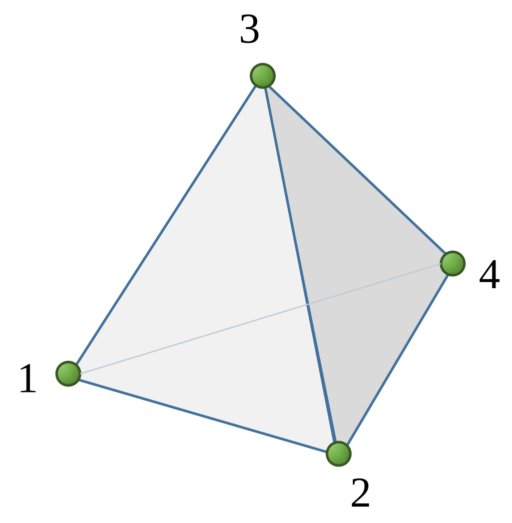
- 1 coordinates (a scalar P, in a 3D space)
- Used for thermal and electrostatic analysis
- E.g used by these elements:
  - ChElementTetra_4_P



Elements

ChElementTetra_4

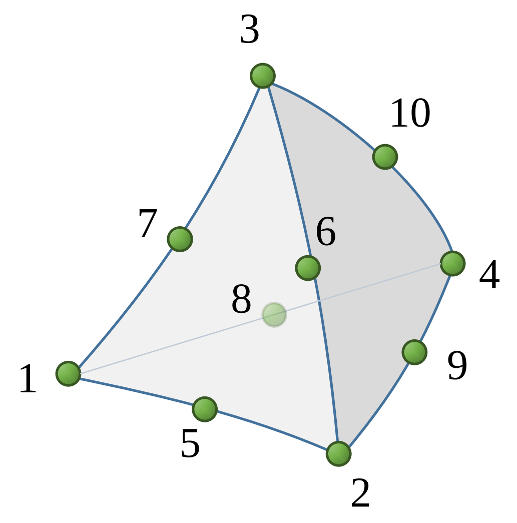
- 4 nodes of ChNodeFEAxyz type
- Linear interpolation, constant stress
- 1 integration point
- Corotational formulation for large displacements
- Use polar decomposition for corotated frame
- Useful for solids
- Fastest element for solids



Elements

ChElementTetra_10

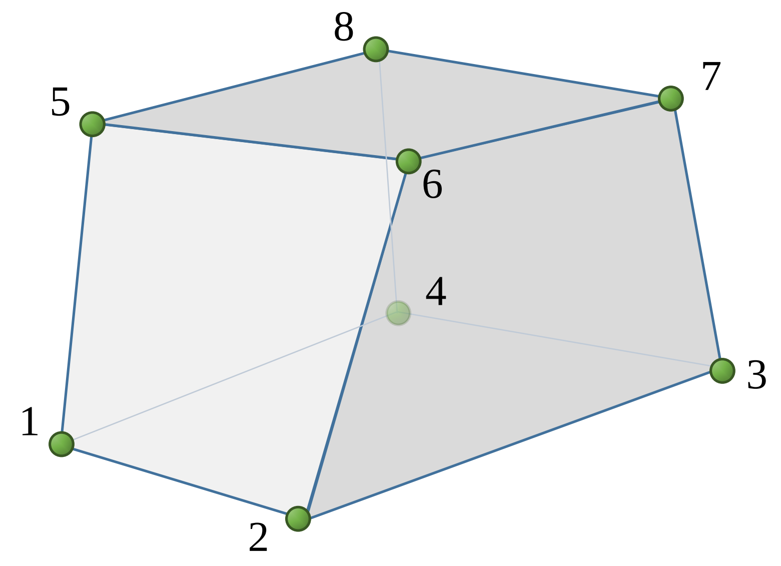
- 10 nodes of ChNodeFEAxyz type
- Quadratic interpolation, linear stress
- 4 integration points
- Corotational formulation for large displacements
- Use polar decomposition for corotated frame
- Note: initial position assuming nodes n>4 exactly at mid-length of edges
- Useful for solids



Elements

ChElementHexa_8

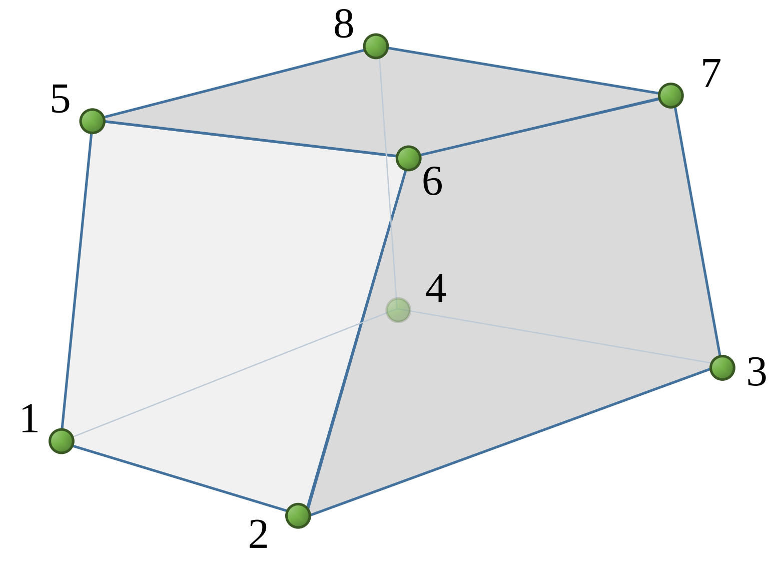
- 8 nodes of ChNodeFEAxyz type
- Linear interpolation
- 8 integration points
- Corotational formulation for large displacements
- Useful for solids, with structured grids



Elements

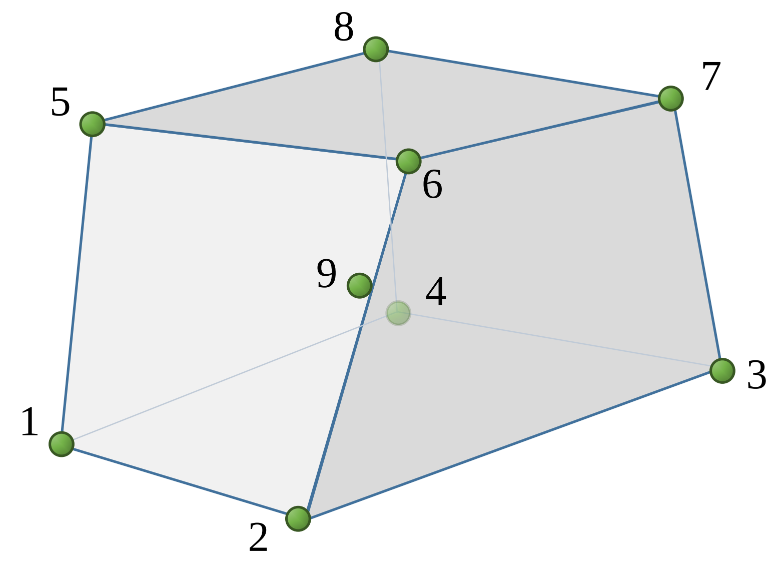
ChElementBrick

- 8 nodes of ChNodeFEAxyz type
- Tri-linear interpolation
- 8 integration points
- Isoparametric formulation: Large deformation
- Useful for solids, with structured grids



Elements

ChElementBrick_9

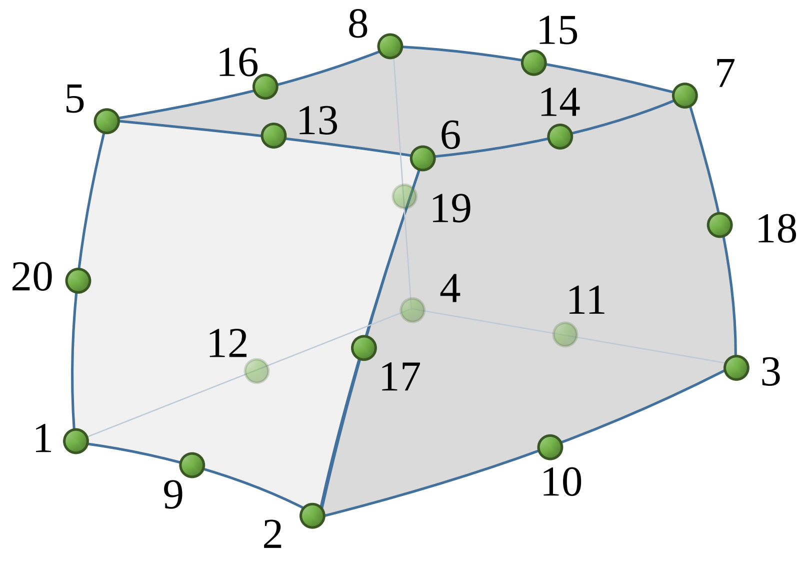


- 8 nodes of ChNodeFEAxyz type
- 1 node of ChNodeFEAcurv type
- Higher-order interpolation
- Does not need numerical techniques to alleviate locking
- Isoparametric formulation: Large deformation
- Useful for solids, with structured grids: Used for soil plasticity

Elements

ChElementHexa_20

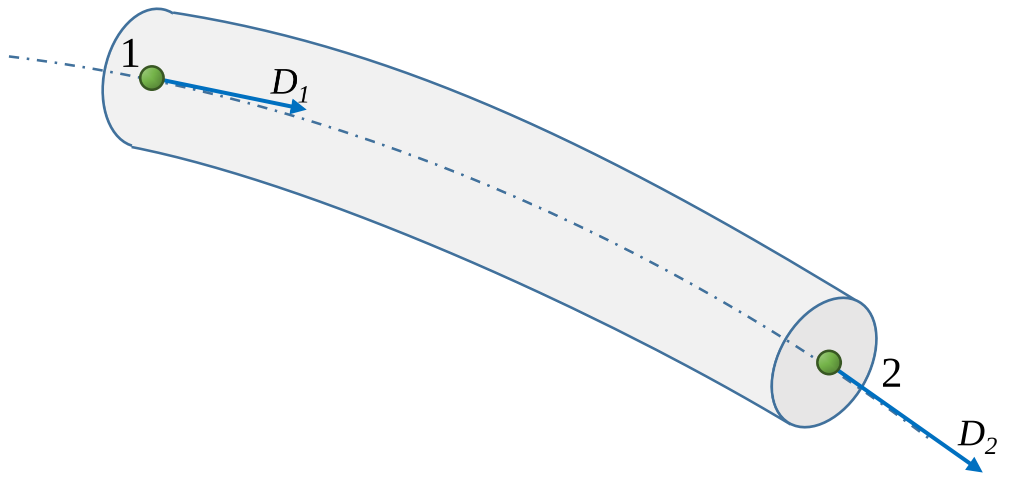
- 20 nodes of ChNodeFEAxyz type
- 8 at vertexes, 12 at edges midpoints
- Quadratic interpolation
- 27 integration points
- Corotational formulation for large displacements
- Useful for solids, with structured grids



Elements

ChElementCableANCF

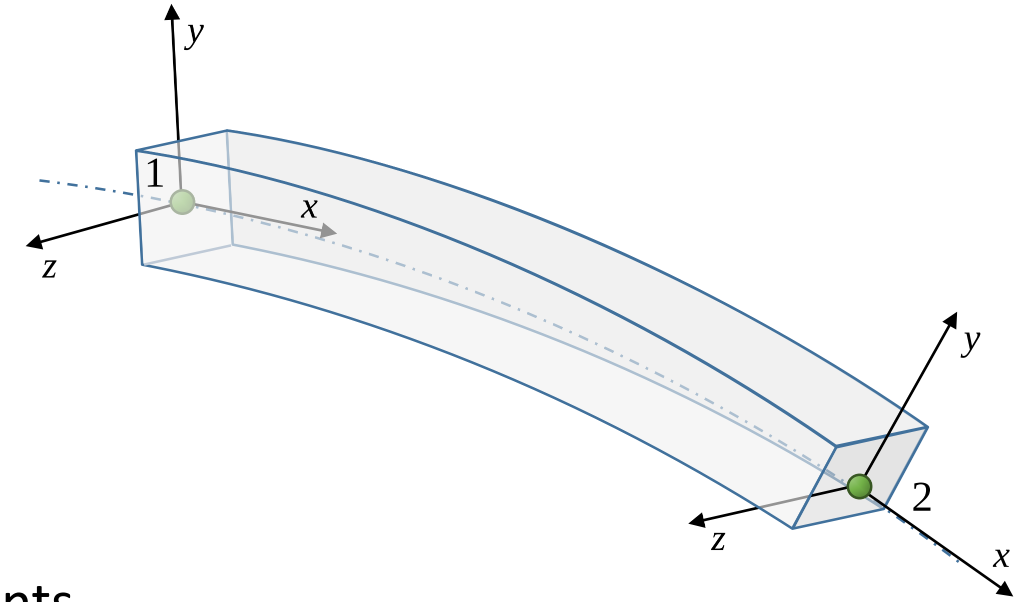
- 2 nodes of ChNodeFEAxyzD type
- 3 integration point (stiffness), 4 (mass)
- ANCF formulation for large displacements
- Thin beam (no shear)
- No torsional stiffness (useful for wires, cables)
- Section property: A, I, E, density, damping

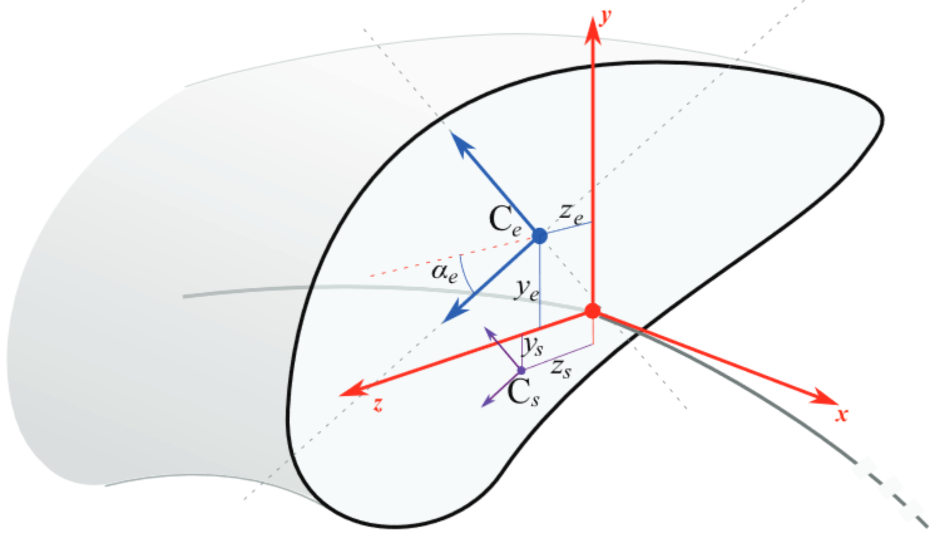


Elements

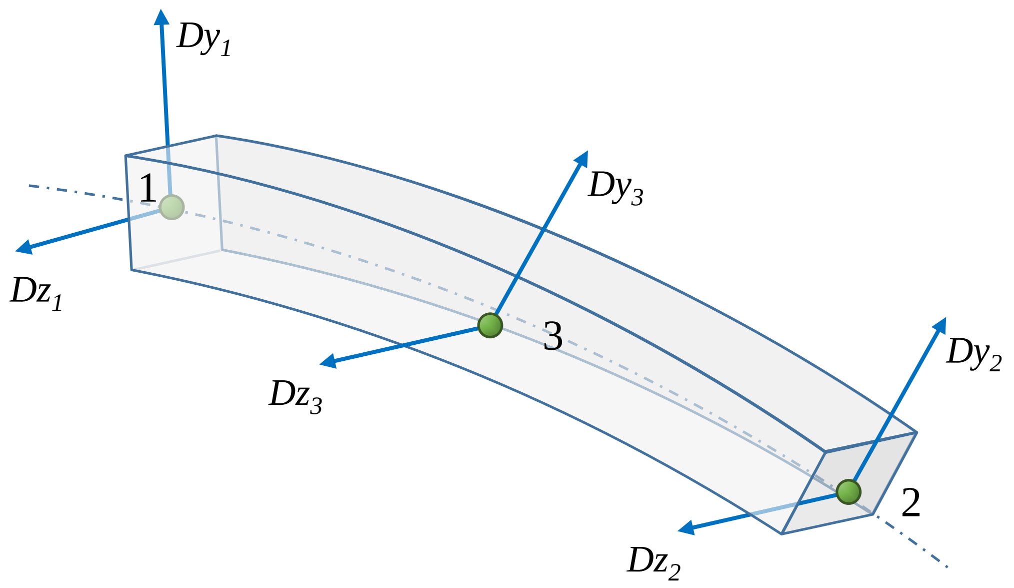
ChElementBeamEuler

- 2 nodes of ChNodeFEAxyzrot type
- Linear interpolation
- 1 integration point (default)
- Corotational formulation for large displacements (small def.)
- Thin beam (no shear), based on the Euler-Bernoulli thin beam theory
- Section property:
  - A, I_{yy}, I_{zz}, E, density, damping
  - G, J for torsional stiffness, plus optional:
  - α_e , z_e , y_e , for offset/rotated section
  - z_s , y_s for offset shear center





Elements

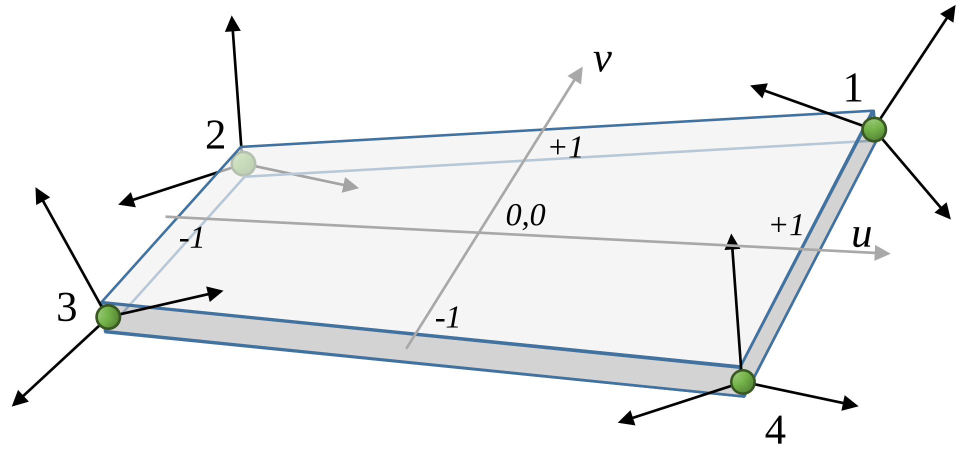


ChElementBeamANCF

- 3 nodes of ChNodeFEAxyzDD type
- ANCF formulation for large deformation
- Includes shear, torsion, two bending curvatures, and stretch
- Validated for small and large deformation
- Continuum-based approach (for now)

Elements

ChElementShellReissner

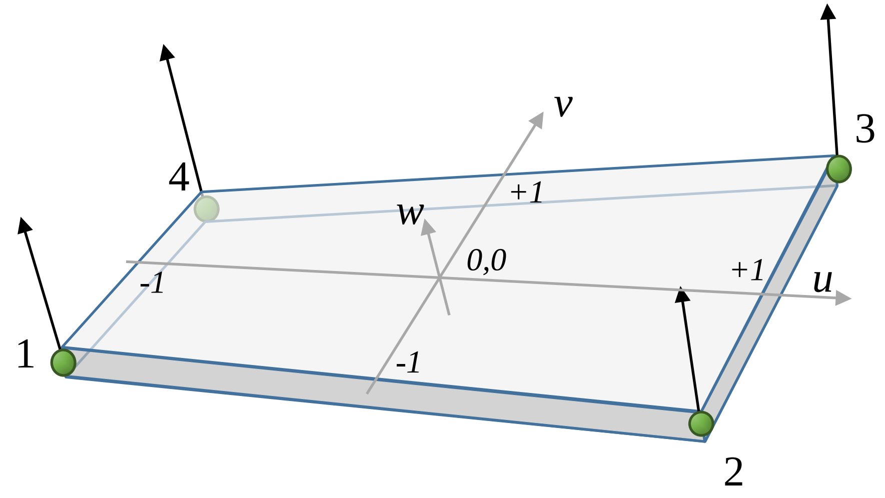


- 4 nodes of ChNodeFEAxyzrot type
- Bi-linear interpolation
- 4 integration points (default)
- Allows large displacements, exponential map used for SO3
- Based on the Reissner 6-field shell theory (w. drilling stiffness)
- Can have multi-layered materials, using Classical Laminate Theory
- ANS, shear-lock free
- Nodes need not to be aligned to shell (rotation offsets auto-computed in initialization)

Elements

ChElementShellANCF

- 4 nodes of ChNodeFEAxyzD type
- Bi-linear interpolation
- 8 integration points (default)
- Allows large deformation
- Can have multi-layered materials
- ANS-EAS, shear-lock free
- Nodes D must be aligned to shell normal at initialization (assuming zero shear)



Contact

- Triangular Mesh
- Node Cloud

Contact for FEA

- Contact of all Chrono::FEA elements can be included in the simulation
- Applications
  - Finite elements can interact though contact with other finite elements –knee articular cartilage surfaces
  - Finite elements can interact with rigid bodies –a tire running on flat terrain
  - Finite elements can interact with discrete particles –shells interacting with DEM particles or rigid bodies
- Two ways for enabling contacts with finite elements:
  - ChContactSurfaceMesh → edge vs edge or node vs face, or edge/nodes vs body shapes
  - ChContactSurfaceNodeCloud → nodes vs rigid body shapes

Contact for FEA

- **Note:** at the time of writing, Non-Smooth-Contacts (**NSC**) are not yet supported in FEA. So, if you add finite elements in Chrono and you also want contacts, the only option is using the Smooth-Contact (**SMC**) approach. That is, you cannot use ChSystemNSC, and you must use for example:

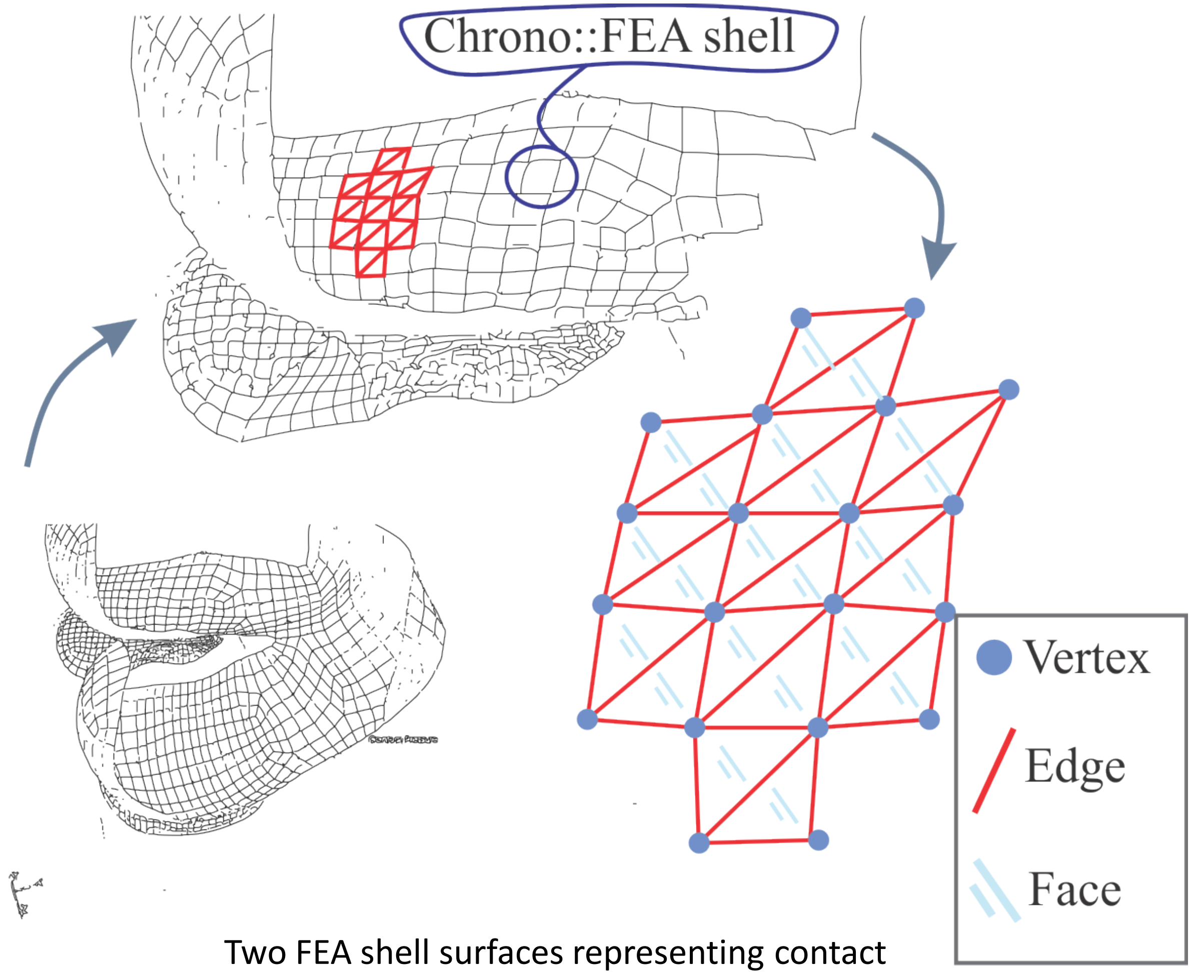
```
ChSystemSMC my_system;
```

  Also, you must use corresponding SMC surface materials, for example:

```
auto mysurfmaterial = std::make_shared<ChMaterialSurfaceSMC>();
mysurfmaterial->SetYoungModulus(6e4);
mysurfmaterial->SetFriction(0.3f);
mysurfmaterial->SetRestitution(0.2f);
```

- The drawback is that SMC uses *penalty* (spring-dashpot systems at each contact) so beware of this:
  - Object may have slight interpenetration
  - Timestep must be small

Triangular mesh: ChContactSurfaceMesh



Two FEA shell surfaces representing contact between knee articular cartilages

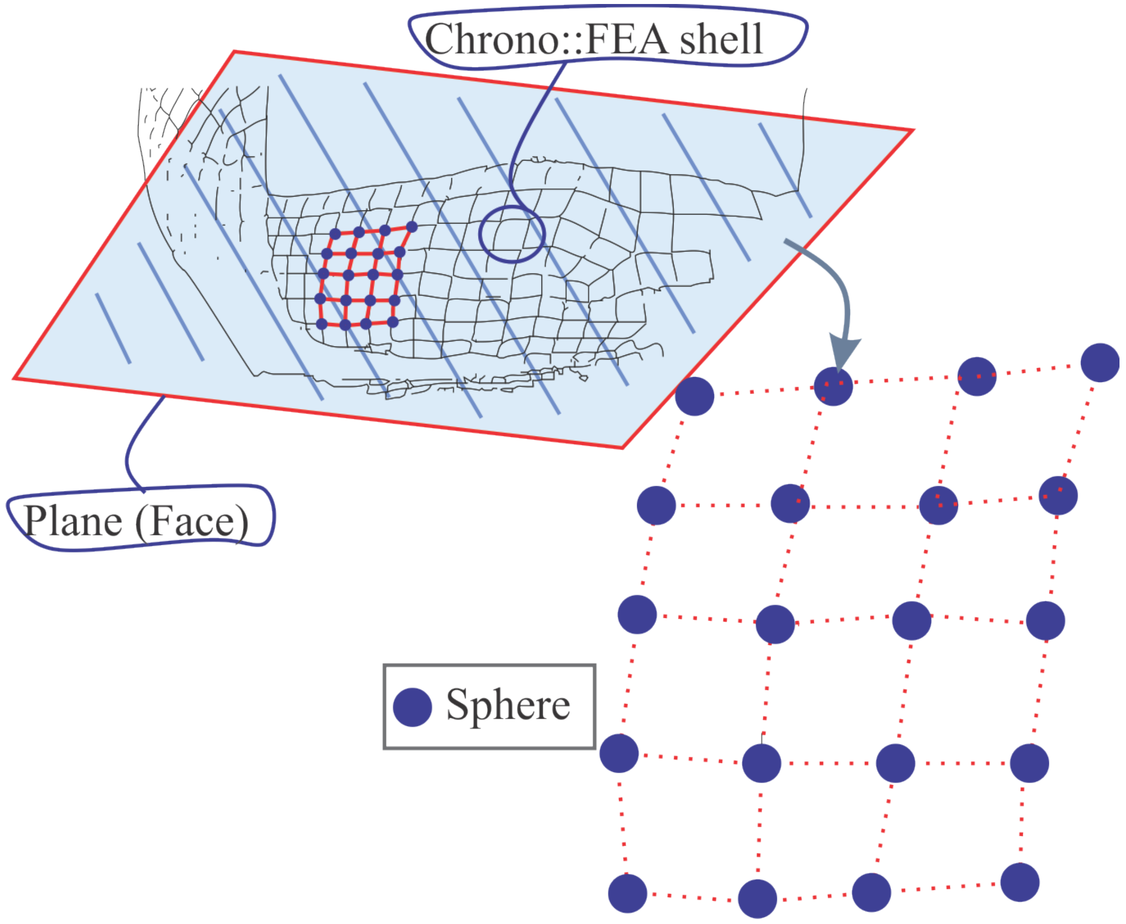
Triangular mesh

- Surface of the finite element mesh is converted into a triangular mesh
- For solid elements, the outer "shell" is converted
- The triangular mesh is composed of vertices, edges, and faces.
- Each vertex is checked for contact with faces
- Each edge is checked for contact with other edges
- Triangular meshes of a finite element model can be used to model contact with other triangular meshes, rigid bodies, and **node clouds** (see next slide)

Syntax:

```
// Create the contact surface and add to the mesh
auto contact_surf = std::make_shared<ChContactSurfaceMesh>();
 m_mesh->AddContactSurface(contact_surf);
 contact_surf->AddFacesFromBoundary(m_contact_face_thickness,
false);
contact_surf->SetMaterialSurface(m_contact_mat);
```

Node cloud: ChContactSurfaceNodeCloud



Node cloud

- Simple way to account for mesh contact
- Each node is represented by a sphere
- All the spheres in the mesh are checked for contact
- It entails a simplification: A continuous surface becomes a discrete representation of a collection of rigid bodies
- Warning: Two node cloud meshes would probably interpenetrate each other without generating contact forces: Node cloud is best indicated when the other contact mesh/es is/are composed of faces

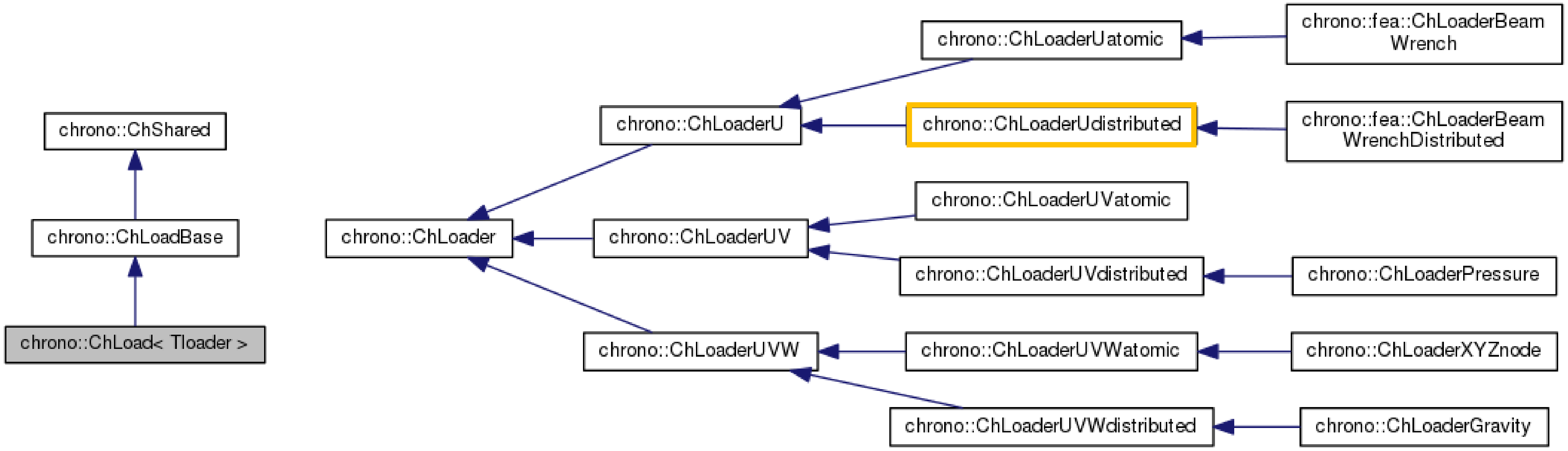
Syntax:

```
// Create the contact surface and add to the mesh
    auto mcontactcloud =
std::make_shared<ChContactSurfaceNodeCloud>();
mesh->AddContactSurface(mcontactcloud);
contact_surf->AddAllNodes(m_contact_node_radius);
contact_surf->SetMaterialSurface(m_contact_mat);
```

Loads and Constraints

- Types of loads
- ChLoader, ChLoadable
- Load inheritance structure

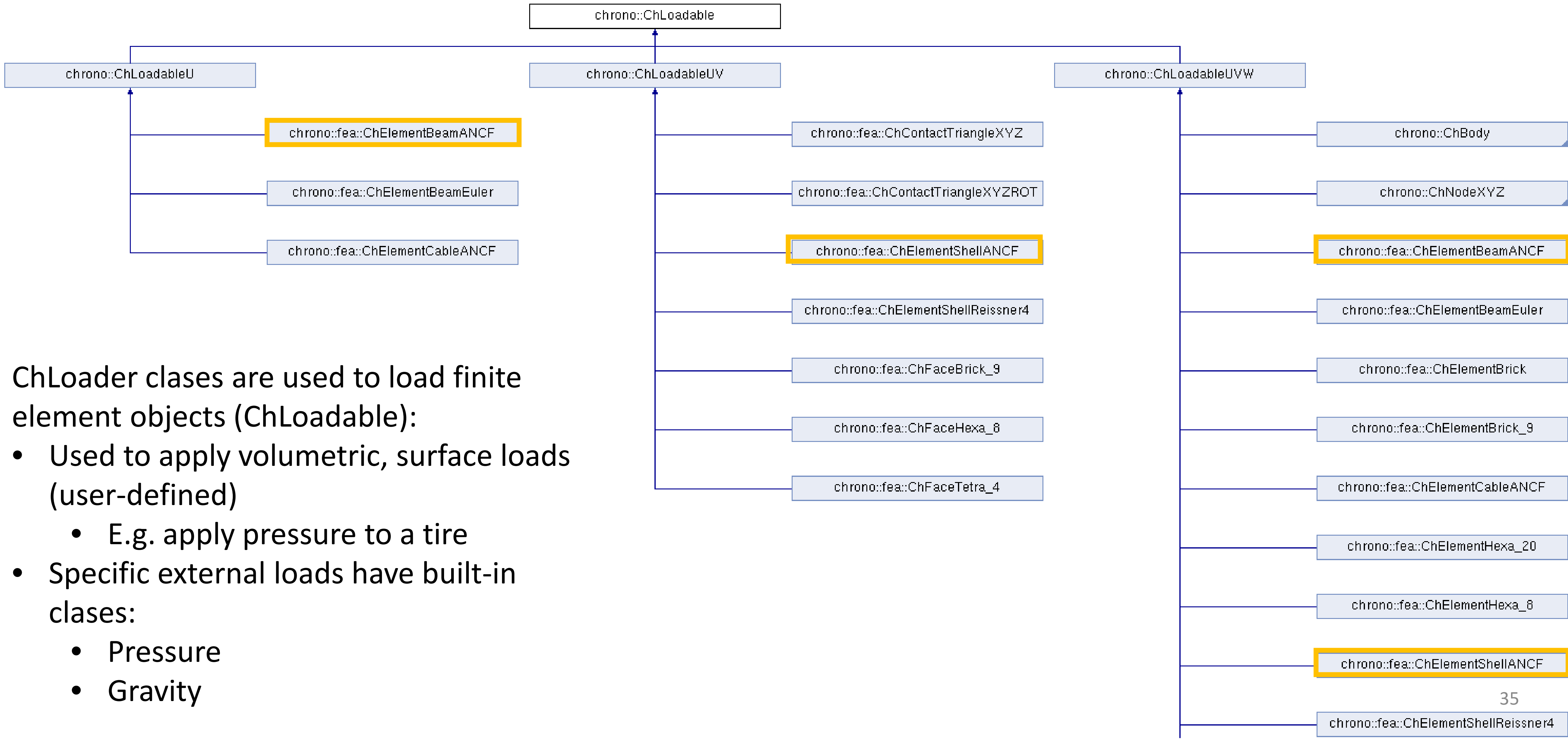
Loads: Inheritance tree



ChLoad inheritance tree: allows the user to apply loads to an FEA mesh in a convenient way

- ChLoader classes perform numerical integration in 1, 2, or 3 dimensions
- ChLoader allows to apply loads on selected nodes or elements or entire mesh
- ChLoad features single (atomic) and distributed loads

Loadables –Finite Elements



chrono::ChLoadable
chrono::ChLoadableU
chrono::ChLoadableUV
chrono::ChLoadableUVW
chrono::fea::ChElementBeamANCF
chrono::fea::ChElementBeamEuler
chrono::fea::ChElementCableANCF
chrono::fea::ChContactTriangleXYZ
chrono::fea::ChContactTriangleXYZROT
chrono::fea::ChElementShellANCF
chrono::fea::ChElementShellReissner4
chrono::fea::ChFaceBrick_9
chrono::fea::ChFaceHexa_8
chrono::fea::ChFaceTetra_4
chrono::ChBody
chrono::ChNodeXYZ
chrono::fea::ChElementBeamANCF
chrono::fea::ChElementBeamEuler
chrono::fea::ChElementBrick
chrono::fea::ChElementBrick_9
chrono::fea::ChElementCableANCF
chrono::fea::ChElementHexa_20
chrono::fea::ChElementHexa_8
chrono::fea::ChElementShellANCF
chrono::fea::ChElementShellReissner4

ChLoader clases are used to load finite element objects (ChLoadable):

- Used to apply volumetric, surface loads (user-defined)
  - E.g. apply pressure to a tire
- Specific external loads have built-in clases:
  - Pressure
  - Gravity

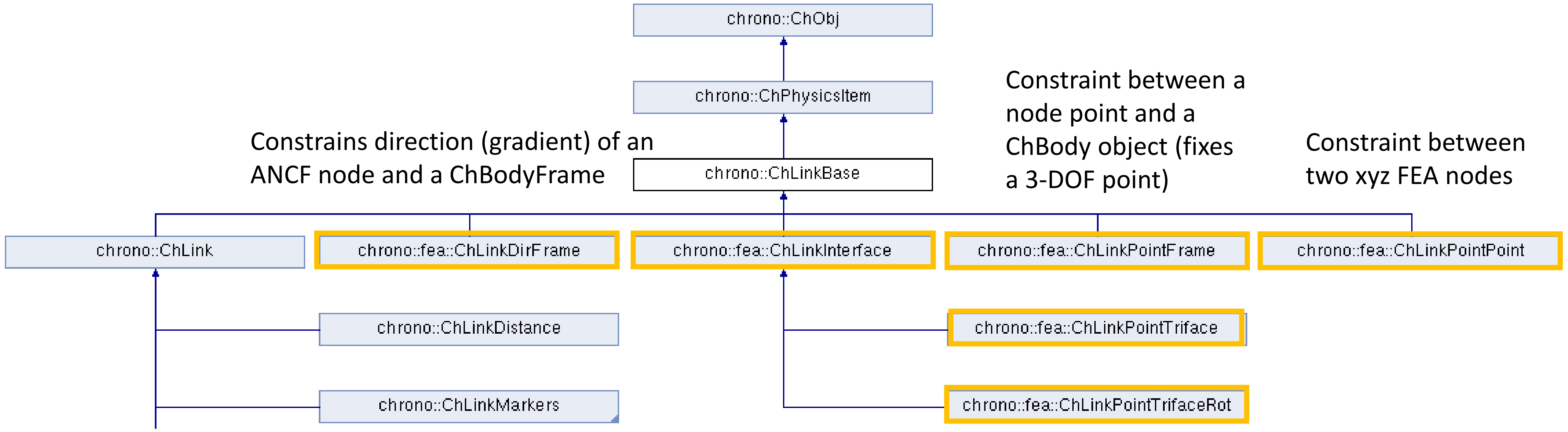
Loads –Example (demo_FEA_loads.cpp)

```
// For example, let's make a distributed triangular load.

class MyLoaderTriangular : public ChLoaderUdistributed {
public:
        // Useful: a constructor that also sets ChLoadable
        MyLoaderTriangular(std::shared_ptr<ChLoadableU> mloadable) :  ChLoaderUdistributed(mloadable) {};

        // Compute F=F(u)
        // This is the function that you have to implement. It should return the
        // load at U. For Euler beams, loads are expected as 6-rows vectors, containing
        // a wrench: forceX, forceY, forceZ, torqueX, torqueY, torqueZ.
        virtual void ComputeF(const double U,     ///< parametric coordinate in line
                     ChVectorDynamic<>& F,       ///< Result F vector here, size must be = n.field coords.of loadable
                     ChVectorDynamic<>* state_x, ///< if != 0, update state (pos. part) to this, then evaluate F
                     ChVectorDynamic<>* state_w  ///< if != 0, update state (speed part) to this, then evaluate F
                     ) {
                      double Fy_max = 0.005;
                      F.PasteVector( ChVector<>(0, (((1+U)/2)*Fy_max),0) ,0,0); // load, force part; hardwired for brevity
                      F.PasteVector( ChVector<>(0,0,0) ,3,0);   // load, torque part; hardwired for brevity
        }

        // Needed because inheriting ChLoaderUdistributed. Use 1 because linear load fx.
        virtual int GetIntegrationPointsU() {return 1;}
};

// Create the load (and handle it with a shared pointer).
std::shared_ptr< ChLoad<MyLoaderTriangular> > mloadtri (new ChLoad<MyLoaderTriangular>(melementA) );
mloadcontainer->Add(mloadtri);  // do not forget to add the load to the load container.
```

FEA constraints

Chrono   features a set of clases that allow

- Imposing position constraints between rigid bodies and meshes
- Imposing direction constraints (on gradients) between rigid bodies and ANCF meshes
- Enforce nodes to get fixed to a point on a triangular face (even with an offset)


chrono::ChObj
chrono::ChPhysicsItem
chrono::ChLinkBase
chrono::ChLink
chrono::fea::ChLinkDirFrame
chrono::fea::ChLinkInterface
chrono::fea::ChLinkPointFrame
chrono::fea::ChLinkPointPoint
chrono::ChLinkDistance
chrono::ChLinkMarkers
chrono::fea::ChLinkPointTriface
chrono::fea::ChLinkPointTrifaceRot

Constrains direction (gradient) of an ANCF node and a ChBodyFrame

Constraint between a node point and a ChBody object (fixes a 3-DOF point)

Constraint between two xyz FEA nodes

Example

Main steps to create a FEA model

1. Make the mesh

- Create a ChMesh mesh and add it to your physical system
  - The mesh will be the container to your FEA nodes and FEA elements
  - You could create more than a single mesh, if needed

```cpp
// The physical system: it contains all physical objects.
ChSystemNSC my_system;

// Create a mesh, that is a container for groups
// of elements and their referenced nodes.
auto my_mesh = std::make_shared<ChMesh>();

// Remember to add the mesh to the system!
my_system.Add(my_mesh);
```

2. Make the nodes

- Create some nodes
  - Usually node positions are set as parameters in their constructors
- Add the nodes to the mesh

```
    // Create some point-like nodes with x,y,z degrees of freedom
    // While creating them, also set X0 undeformed positions.
    auto mnode1 = std::make_shared<ChNodeFEAxyz>(ChVector<>(0, 0, 0));
    auto mnode2 = std::make_shared<ChNodeFEAxyz>(ChVector<>(0, 0, 1));
    auto mnode3 = std::make_shared<ChNodeFEAxyz>(ChVector<>(0, 1, 0));
    auto mnode4 = std::make_shared<ChNodeFEAxyz>(ChVector<>(1, 0, 0));

    // Remember to add nodes and elements to the mesh!
    my_mesh->AddNode(mnode1);
    my_mesh->AddNode(mnode2);
    my_mesh->AddNode(mnode3);
    my_mesh->AddNode(mnode4);
```

2. Make the nodes

- Set node properties, if you need
  - Ex. most node classes provide a way to apply a local force (or even a torque if they have rotational DOFs) by using SetForce() , an easier alternative to using ChLoad classes, if the force is constant.
  - Here you may want also to attach an optional local point-mass using SetMass() for the nodes; otherwise the default mass for FEA nodes is zero, as mass is mostly added by finite elements.

```
    // For example, set some non-zero mass concentrated at the nodes
    mnode1->SetMass(0.01);
    mnode2->SetMass(0.01);

    // For example, set an applied force to a node:
    mnode2->SetForce(ChVector<>(0, 5, 0));
```

3. Make the elements: the material

- Create a material that will be shared between elements
  - Note that not all elements need a material, for instance ChElementSpring has not

- Set material properties

```
    // Create a material, that must be assigned to each element,
    auto mmaterial = std::make_shared<ChContinuumElastic>();

    // …and set its parameters
    mmaterial->Set_E(0.01e9);  // rubber 0.01e9, steel 200e9
    mmaterial->Set_v(0.3);
```

3. Make the elements

- Create FEA elements
- Add the elements to the mesh
- Assign nodes – which you created before
- Assign material(s) to the elements

```
// Create the tetrahedron element,
auto melement1 = std::make_shared<ChElementTetra_4>();

// Remember to add elements to the mesh!
my_mesh->AddElement(melement1);

// assign nodes
melement1->SetNodes(mnode1, mnode2, mnode3, mnode4);

// assign material
melement1->SetMaterial(mmaterial);
```

4. Setup the simulation

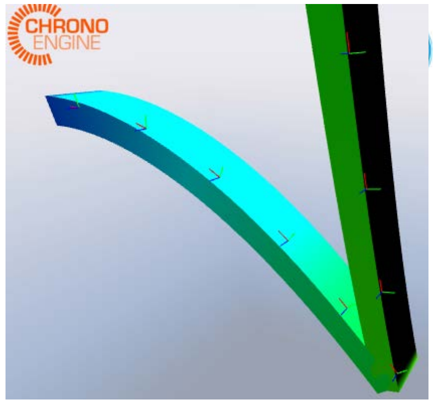
- Remember `my_system.SetupInitial();` before running the simulation
- Change the solver type to MINRES or MKL (or others that can solve FEA)
- Run analysis

```
    // Mark completion of system construction
    my_system.SetupInitial();

    // The FEA problems require MINRES solvers! (or MKL or MUMPS optional solvers)
    my_system.SetSolverType(ChSolver::Type::MINRES);

    // Analysis – Example 1: a static analysis
    my_system.DoStaticLinear();

    // Analysis – Example 1: dynamics
    double timestep = 0.01;
    while (my_system.GetChTime() < 2) {
        my_system.DoStepDynamics(timestep);
    }
```

Visualization

Runtime visualization with IRRLICHT lightning fast realtime 3D engine >>>

Visualization for FEA meshes



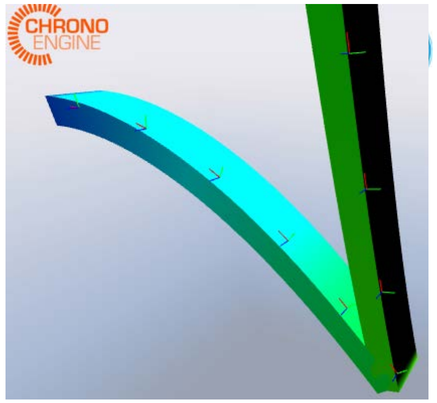
- Create a ChVisualizationFEAmesh
- Add it to the mesh as an asset

The ChVisualizationFEAmesh will automatically update a triangle mesh (a ChTriangleMeshShape asset that is internally managed) by setting proper coordinates and vertex colours as in the FEM elements. Such triangle mesh can be rendered by Irrlicht or POVray or whatever postprocessor that can handle a coloured ChTriangleMeshShape.

The ChVisualizationFEAmesh has many settings. Look at its header file for comments/instructions. In future we will expand its features.

```
auto mvisualizemesh = std::make_shared<ChVisualizationFEAmesh>(*(my_mesh.get()));
mvisualizemesh->SetFEMdataType(ChVisualizationFEAmesh::E_PLOT_NODE_SPEED_NORM);
mvisualizemesh->SetColorscaleMinMax(0.0, 5.50);
mvisualizemesh->SetShrinkElements(true, 0.85);
mvisualizemesh->SetSmoothFaces(true);
my_mesh->AddAsset(mvisualizemesh);
```

Visualization for FEA meshes - Irrlicht



- For the Irrlicht viewer, remember to do the following before running the simulation loop:

```
// ==IMPORTANT!== Use this function for adding a ChIrrNodeAsset to all items
// in the system. These ChIrrNodeAsset assets are 'proxies' to the Irrlicht meshes.
// If you need a finer control on which item really needs a visualization proxy in
// Irrlicht, just use application.AssetBind(myitem); on a per-item basis.

application.AssetBindAll();

// ==IMPORTANT!== Use this function for 'converting' into Irrlicht meshes the assets
// that you added to the bodies into 3D shapes, they can be visualized by Irrlicht!

application.AssetUpdateAll();
```