Chrono::FEA Hands-on Exercises

Modeling and simulation of colliding beams

Tutorial 1

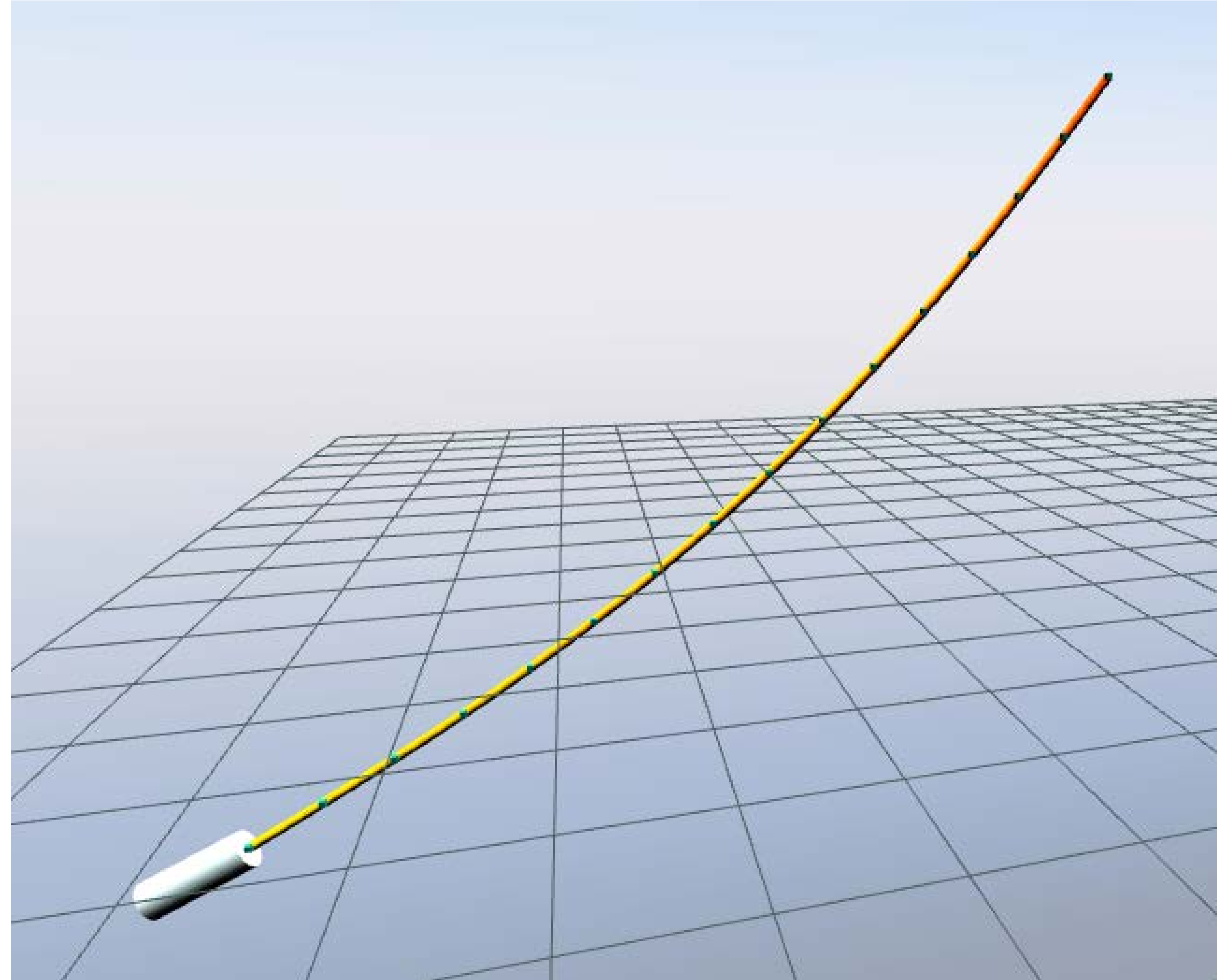
Swinging cable with a weight at the end

The exercise - FEA_cable_collide_1.cpp

- Use ChElementCableANCF to model a flexible cable

- Conect the cable to the ground using a constraint

- TO DO: connect a ChBody (a cylinder) to the free end of the wire



1. Create the system and 2. create the mesh

- Create the system and the mesh
- Add the mesh to the system

```
// 1. Create the system
ChSystemSMC system;

//    Optional: specify the gravitational acceleration vector, default
//    is Y up
system.Set_G_acc(ChVector<>(0, -9.81, 0));


// 2. Create the mesh that will contain finite elements, and add it to the system

auto mesh = std::make_shared<ChMesh>();

system.Add(mesh);
```

3. Create a material

- Create the material
- Set its properties

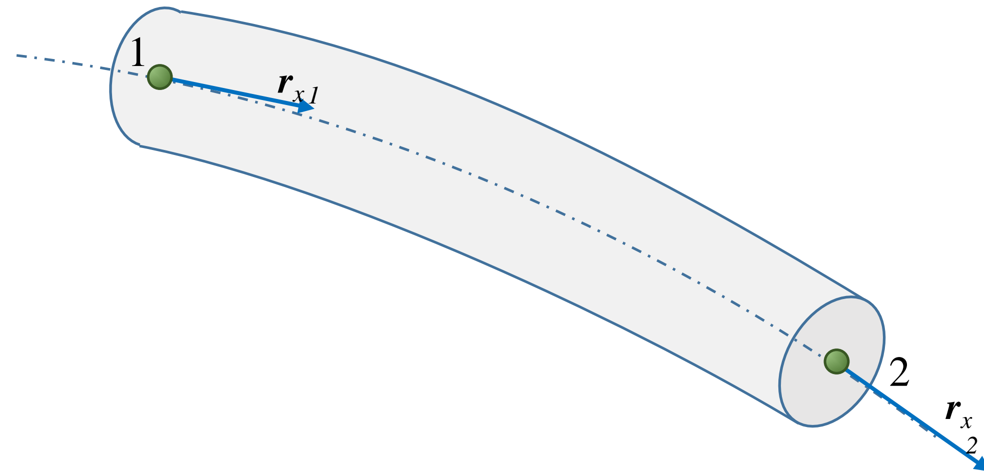
```
auto beam_material = std::make_shared<ChBeamSectionCable>();
beam_material->SetDiameter(0.01);
beam_material->SetYoungModulus(0.01e9);
```

Note that each FEA element type requires some corresponding type of material. Here we will use ChElementCableANCF elements: they use a material of type ChBeamSectionCable

The material will be shared by all elements – in fact, we handle it via a std:: shared pointer

4. Create the nodes

- Nodes for ChElementCableANCF must be of ChNodeFEAxyzD class; i.e. each node has 6 coordinates: {position, longitudinal gradient}, where gradient vector *D* is the tangent to the cable:



- To make things easier in the following, we store node pointers into an optional 'beam_nodes' array, i.e. a std::vector<> that contains shared pointers to our nodes, later we can use such array for easy creation of elements between the nodes:

```
std::vector< std::shared_ptr<ChNodeFEAxyzD> > beam_nodes;
```

4. Create the nodes

- We use a for() loop to create the nodes, all aligned horizontally to X axis:

```
std::vector<std::shared_ptr<ChNodeFEAxyzD> > beam_nodes;

double length = 1.2;  // beam length, in meters;
int N_nodes = 16;
for (int in = 0; in < N_nodes; ++in) {
    // i-th node position
    ChVector<> position(length * (in / double(N_nodes - 1)),  // node position, x
                        0.5,                                   // node position, y
                        0);                                    // node position, z
    // i-th node direction
    ChVector<> direction(1.0, 0, 0);

    // create the node
    auto node = std::make_shared<ChNodeFEAxyzD>(position, direction);

    // add it to mesh
    mesh->AddNode(node);

    // add it to the auxiliary beam_nodes
    beam_nodes.push_back(node);
}
```

5. Create the elements

- Each element must be set with the ChBeamSectionCable material created before
- Do not forget to use SetNodes(), and to add the elements to the mesh

```
for (int ie = 0; ie < N_nodes - 1; ++ie) {
    // create the element
    auto element = std::make_shared<ChElementCableANCF>();

    // set the connected nodes (pick two consecutive nodes in beam_nodes container)
    element->SetNodes(beam_nodes[ie], beam_nodes[ie + 1]);

    // set the material
    element->SetSection(beam_material);

    // add it to mesh
    mesh->AddElement(element);
}
```

6. Add a hinge constraint at one end of the wire

- For the ChNodeFEAxyzD there are specific constraints that can be used to connect them to a ChBody, namely ChLinkPointFrame and ChLinkDirFrame.
- To attach one end of the beam to the ground, we must create and fix a ground ChBody.

```
// create an invisible body to use as an anchoring body, and fix it:
auto truss = std::make_shared<ChBody>();
truss->SetBodyFixed(true);
system.Add(truss);

// lock an end of the wire to the truss
auto constraint_pos = std::make_shared<ChLinkPointFrame>();
constraint_pos->Initialize(beam_nodes[0], truss);
system.Add(constraint_pos);
```

Note. There is a shorter alternative to using a constraint as above. One could simply type:

```
beam_nodes[0]->SetFixed(true);
```

But this works only when the node must be fixed to an absolute reference. And it fixes also direction.

Methods

ChBodyEasyCylinder

```
/// Easy-to-use class for quick creation of rigid bodies with a cylindrical shape.
/// Compared to the base ChBody class, this class also does
/// automatically, at object creation, the following tasks that
/// you would normally do by hand if using ChBody:
/// - a visualization shape is created and added, if visualization asset is desired
/// - a collision shape is created and added, if collision is desired,
/// - mass and moment of inertia is automatically set, according to the geometry.
class ChBodyEasyCylinder : public ChBody {
  public:
    /// Creates a ChBody plus adds an optional visualization shape and, optionally,
    /// a collision shape. Mass and inertia are set automatically depending
    /// on density.
    /// Cylinder is assumed with body Y axis as vertical, and reference is at half height.
    ChBodyEasyCylinder(double radius, double height, double mdensity, bool collide = false, bool
visual_asset = true,
  ChMaterialSurface::ContactMethod contact_method = ChMaterialSurface::NSC) : ChBody(contact_method) {
…
```

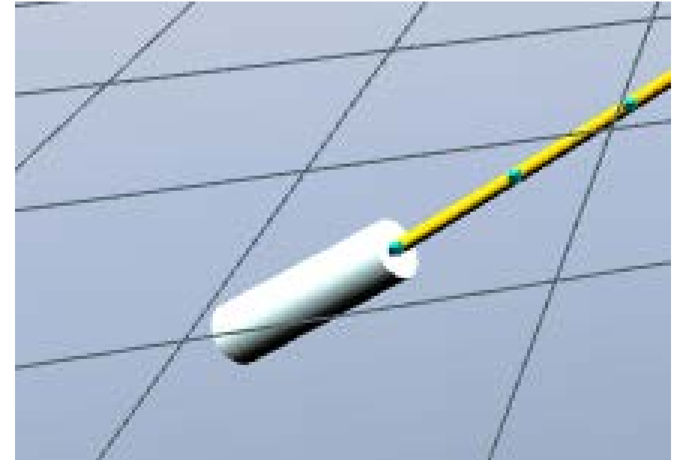
Methods

ChLinkPointFrame

```
/// Class for creating a constraint between an FEA node of ChNodeFEAxyz type
/// and a ChBodyFrame (frame) object.
/// The node position is enforced to coincide to a given position associated with the ChBodyFrame.
class ChApiFea ChLinkPointFrame : public ChLinkBase {
    // Chrono simulation of RTTI, needed for serialization
    CH_RTTI(ChLinkPointFrame, ChLinkBase);
  private:
…

  public:
    ChLinkPointFrame(); /// Creation of constraint
    ChLinkPointFrame(const ChLinkPointFrame& other);

/// Initialize this constraint, given the node and body frame to join.
/// The attachment position is the actual position of the node (unless
/// otherwise defined, using the optional 'pos' parameter).
/// Note: the node and body must belong to the same ChSystem.
virtual int Initialize(std::shared_ptr<ChNodeFEAxyz> node,  ///< xyz node (point) to join
                           std::shared_ptr<ChBodyFrame> body,    ///< body (frame) to join
                           ChVector<>* pos = 0                   ///< attachment position in absolute
coordinates
                           );
```

Exercise

ADD ALSO A CYLINDER ATTACHED
TO THE FREE END OF THE CABLE



Hints:

- Suggested size: radius: 0.02, height: 0.1, density: 1000.
- Use the ChBodyEasyCylinder to make the cylinder, pass size as parameters in construction.
- Use the ChLinkPointFrame to connect the cylinder and the end node.
- Do not forget to add the body and the constraint to your ChSystem

Tutorial 1 solution

Solution

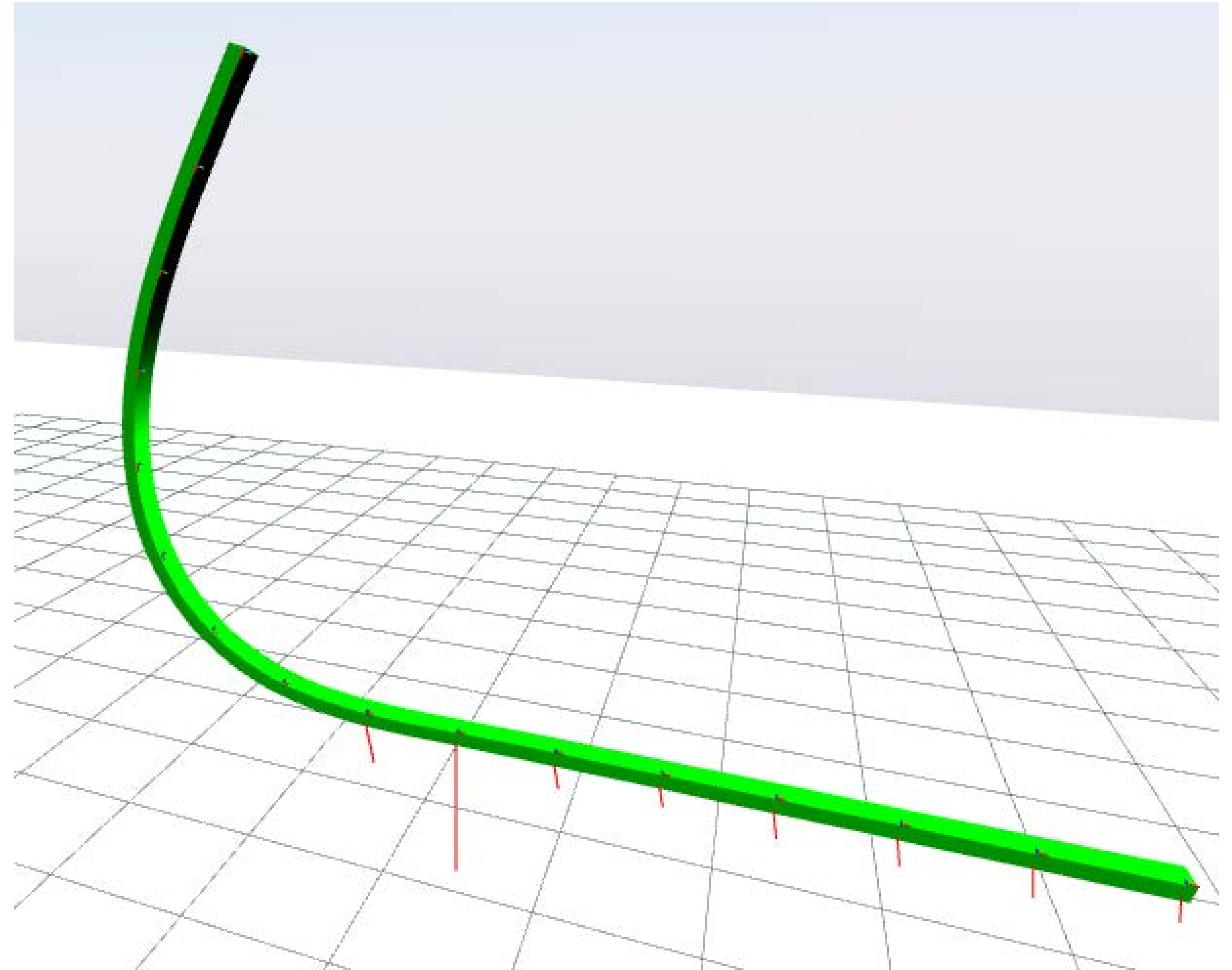
```
// create the cylinder
auto cylinder = std::make_shared<ChBodyEasyCylinder>(
            0.02,  // radius
            0.1,   // height
            1000,  // density (used to auto-set inertia, mass)
            true,  // do collide
            true); // do visualize

// move cylinder to end of beam
cylinder->SetPos( beam_nodes.back()->GetPos() + ChVector<>(0, -0.05, 0) );

// add it to the system
system.Add(cylinder);

// lock an end of the wire to the cylinder
auto constraint_cyl = std::make_shared<ChLinkPointFrame>();
constraint_cyl->Initialize(beam_nodes.back(), cylinder);
system.Add(constraint_cyl);
```

Tutorial 2

Swinging cable colliding with the ground

The tutorial - FEA_cable_collide_2.cpp

- Make a hanging cable, this time using the Bernoulli beams.
- TO DO: create the cable using Bernoulli beams, and constraint one end with a hinge
- Make the cable collide with a flat ground

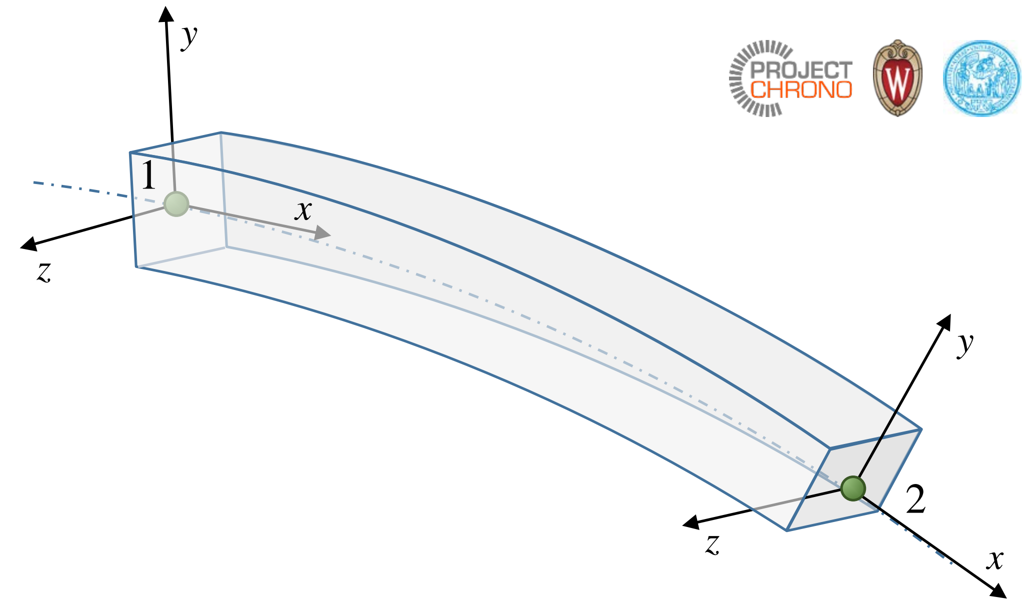


1. Create the system and 2. create the mesh

***NOTE** that we need contact in FEA, so we use the ChSystemSMC, that uses a penalty contact model. The class ChSystemNSC uses DVI hard contacts that cannot yet be used with FEA)*

```
// 1. Create the physical system
//    NOTE that we need contact in FEA, so we use the ChSystemSMC
ChSystemSMC system;

// 2. Create the mesh and add it to the system
auto mesh = std::make_shared<ChMesh>();

system.Add(mesh);
```

Exercise



CREATE THE CABLE USING BERNOULLI BEAMS, AND CONSTRAINT ONE END WITH A HINGE

Hints:

- See FEA_cable_collide_0.cpp, but this time with ChElementBeamEuler elements instead of ChElementCableANCF.
- When creating the section material, in 3., remember that ChElementBeamEuler needs a ChBeamSectionAdvanced material.
- When creating the nodes, in 4., the nodes for ChElementBeamEuler must be of ChNodeFEAxyzrot class; i.e. each node has coordinates of type: {position, rotation}, where the X axis of rotated system represents the direction of the beam, see figure
- When creating the elements, in 5., use ChElementBeamEuler
- When creating the truss-node constraint, in 6., use ChLinkMateSpherical

Make the cable collide (1)

- Create a ChMaterialSurfaceDEM , it must be assigned to FEA meshes and rigid bodies. The ChSystemDEM requires it!

```
// Create a surface material to be shared with some objects
auto mysurfmaterial = std::make_shared<ChMaterialSurfaceSMC>();
mysurfmaterial->SetYoungModulus(2e4);
mysurfmaterial->SetFriction(0.3f);
mysurfmaterial->SetRestitution(0.2f);
mysurfmaterial->SetAdhesion(0);
```

Make the cable collide (2)

- Create a ChContactSurfaceNodeCloud and add to the FEA mesh. This is the easiest representation of an FEA contact surface: it simply creates contact spheres per each node. So, no edge-edge cases can be detected between elements though, but it is enough for dense finite elements meshes that collide with large objects.

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

// Must use this to 'populate' the contact surface.
// Use larger point size to match beam section radius
mcontactcloud->AddAllNodes(0.01);

// Use our DEM surface material properties
mcontactcloud->SetMaterialSurface(mysurfmaterial);
```

Make the cable collide (3)

- Make a huge box to represent the flat ground, to collide with:

```
auto floor = std::make_shared<ChBodyEasyBox>(
      4, 0.2, 4,  // x,y,z size
      1000,       // density
      true,       // visible
      true        // collide
    );

system.Add(floor);

floor->SetBodyFixed(true);
floor->SetPos( ChVector<>(0,-0.1,0) );

// Use our DEM surface material properties
floor->SetMaterialSurface(mysurfmaterial);
```

Tutorial 2 solution

Solution (1 of 4)

```
// 3. Create a material for the beam finite elements.

//    Note that each FEA element type requires some corresponding
//    type of material. ChElemetBeamEuler require a ChBeamSectionAdvanced material.

auto beam_material = std::make_shared<ChBeamSectionAdvanced>();
beam_material->SetAsRectangularSection(0.012, 0.025);
beam_material->SetYoungModulus (0.01e9);
beam_material->SetGshearModulus(0.01e9 * 0.3);
beam_material->SetBeamRaleyghDamping(0.01);
```

Solution (2 of 4)

```
// 4. Create the nodes

std::vector<std::shared_ptr<ChNodeFEAxyzrot> > beam_nodes;

double length = 1.2;  // beam length, in meters;
int N_nodes = 16;
for (int in = 0; in < N_nodes; ++in) {
    // i-th node position
    ChVector<> position(length * (in / double(N_nodes - 1)),  // node position, x
                        0.5,                                  // node position, y
                        0);                                   // node position, z
    // create the node
    auto node = std::make_shared<ChNodeFEAxyzrot>( ChFrame<>(position) );

    // add it to mesh
    mesh->AddNode(node);

    // add it to the auxiliary beam_nodes
    beam_nodes.push_back(node);
}
```

Solution (3 of 4)

```
// 5. Create the elements

for (int ie = 0; ie < N_nodes - 1; ++ie) {
    // create the element
    auto element = std::make_shared<ChElementBeamEuler>();

    // set the connected nodes (pick two consecutive nodes in our beam_nodes container)
    element->SetNodes(beam_nodes[ie], beam_nodes[ie + 1]);

    // set the material
    element->SetSection(beam_material);

    // add it to mesh
    mesh->AddElement(element);
}
```

Solution (4 of 4)

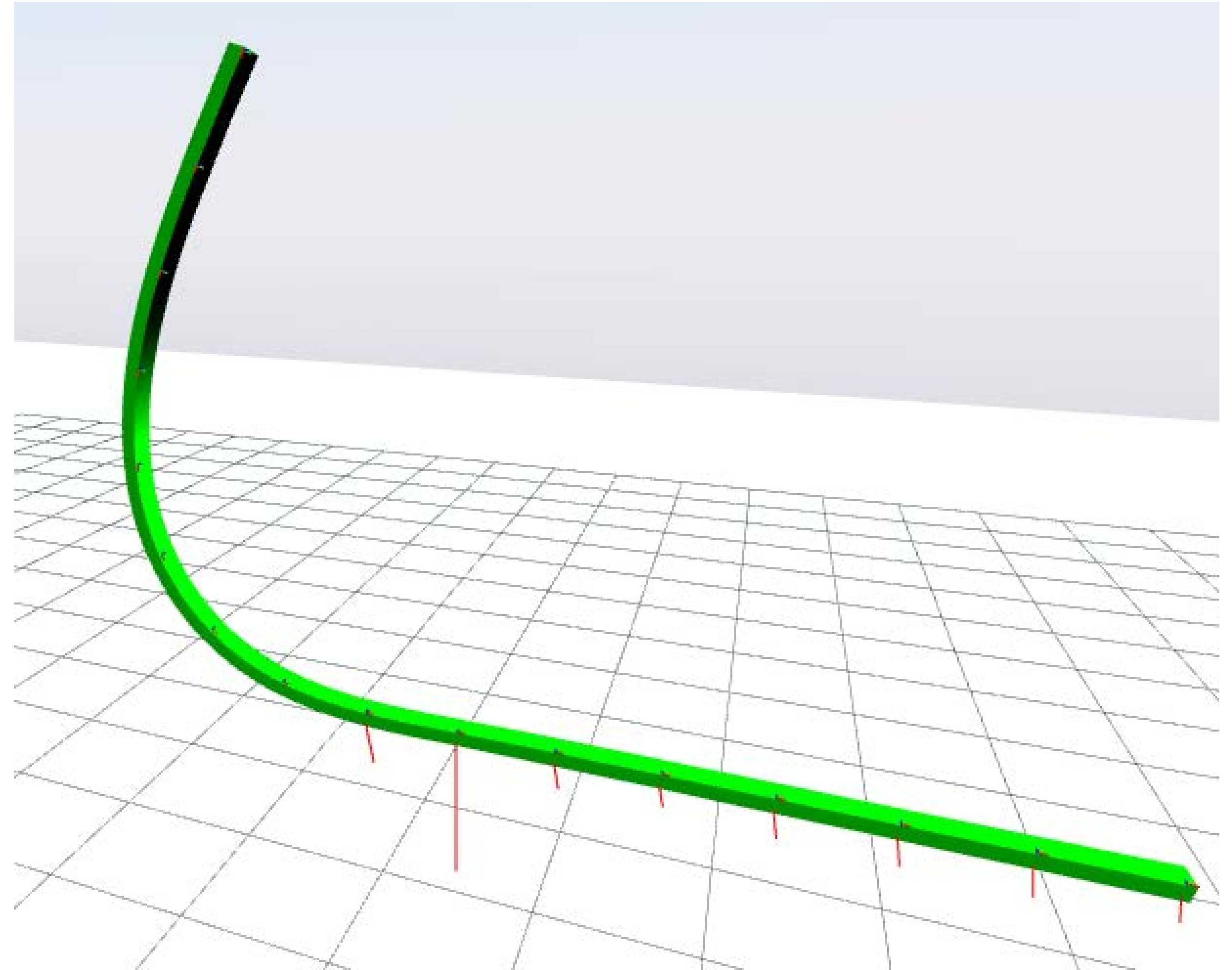
```
// 6. Add constraints

auto truss = std::make_shared<ChBody>();
truss->SetBodyFixed(true);
system.Add(truss);

// lock an end of the wire to the truss
auto constraint_pos = std::make_shared<ChLinkMateSpherical>();
constraint_pos->Initialize(
    beam_nodes[0],  // node to constraint
    truss,          // body to constraint
    false,          // false: next 2 pos are in abs. coords, true: in rel. coords
    beam_nodes[0]->GetPos(), // sphere ball pos
    beam_nodes[0]->GetPos()  // sphere cavity pos
    );
system.Add(constraint_pos);
```

Tutorial 3

Adding loads to the cable

The tutorial - FEA_cable_collide_3.cpp

- Same swinging and colliding cable, but this time we apply some loads.
- TO DO: apply point and distributed loads
- TO DO: develop a custom load (a time-varying load)



1. - 8. Create the FEA model

We omit these steps because they are the same of the previous tutorial

```
// 1. Create the physical system
//    NOTE that we need contact in FEA, so we use the ChSystemSMC
ChSystemSMC system;

// 2. Create the mesh and add it to the system
auto mesh = std::make_shared<ChMesh>();

system.Add(mesh);


ETC . . .
```

9. Apply loads

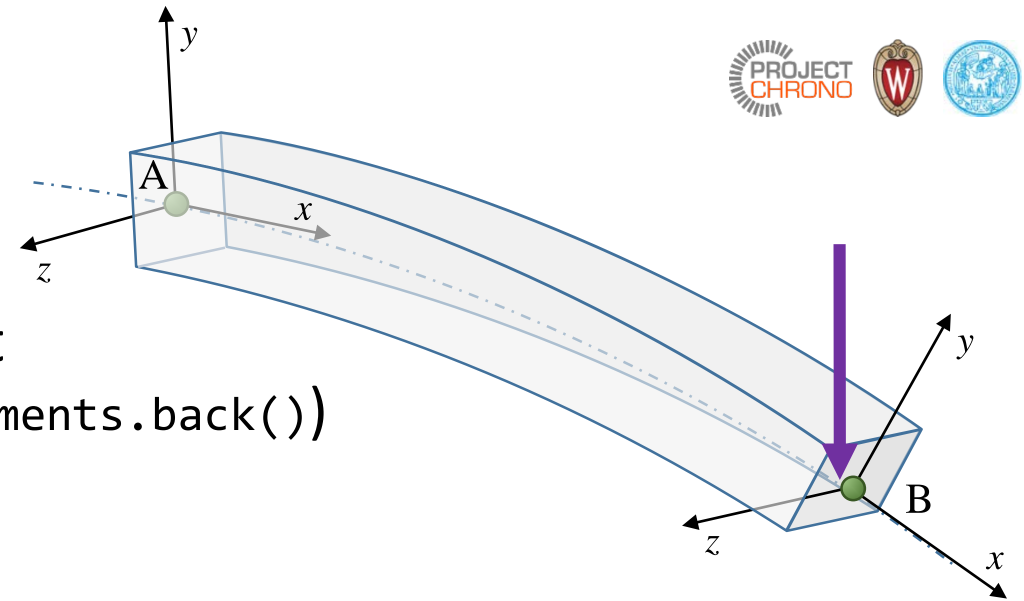
A «load» (a force, a torque, a pressure, etc.) is an object from the ChLoad class, or from inherited subclasses.

Loads can be applied to FEA elements, to FEA nodes, and to ChLoadable-inherited objects in general.

- Loads must be added to a ChLoadContainer.
- The ChLoadContainer must be added to the physical system:

```
//// Create the load container and add it to your ChSystem
auto loadcontainer = std::make_shared<ChLoadContainer>();
system.Add(loadcontainer);
```

9. Apply loads

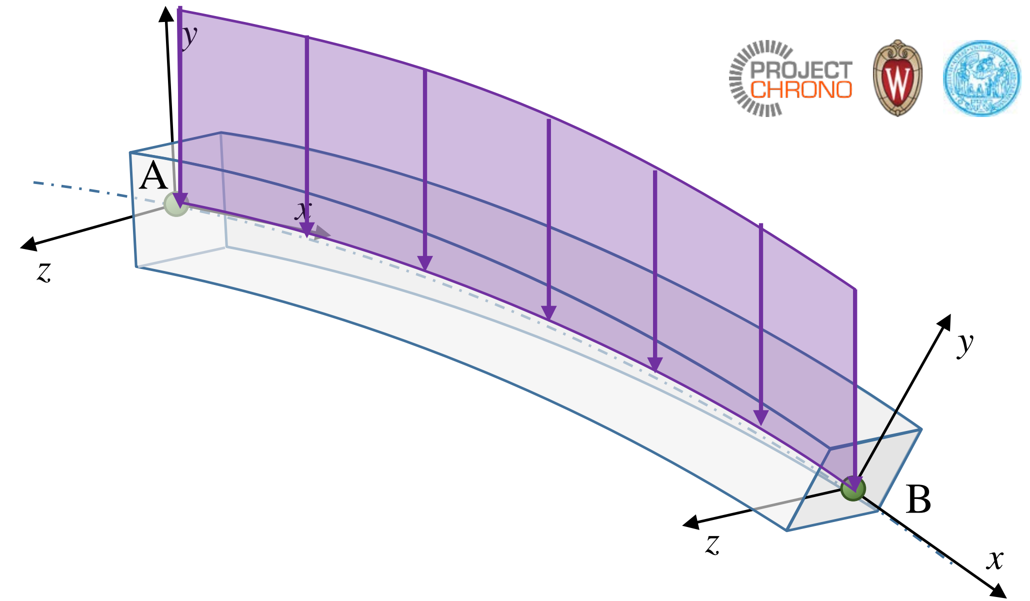


Example: add a vertical force load to the end point of the last beam element (last element is: `beam_elements.back()`)

We will use a ready-to-use load class, available in src/chrono_fea/ChLoadsBeam.h

```
// Create a "wrench" atomic load.
// A wrench is a {force, torque}. We will use only the force part.
auto mwrench = std::make_shared<ChLoadBeamWrench>(beam_elements.back());

// Set the point where the load is applied along the abscissa of the
// element, in -1..+1 range, -1: end A, 0: mid, +1: end B
mwrench->loader.SetApplication(1.0);
mwrench->loader.SetForce(ChVector<>(0,-0.2,0));

// That's all. Do not forget to add the load to the load container.
loadcontainer->Add(mwrench);
```

Exercise 1

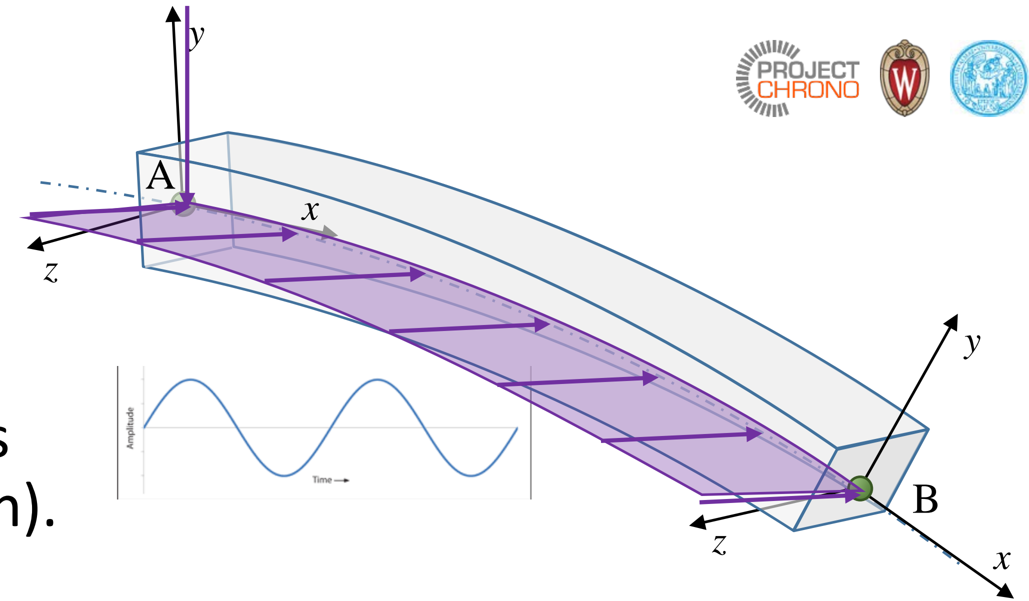


USE A *DISTRIBUTED* VERTICAL LOAD, INSTEAD OF A POINT LOAD, APPLIED AS A CONSTANT VALUE ALONG THE LAST ELEMENT.

Hints:

- It is not much different than the previous load example.
- Use the ChLoadBeamWrenchDistributed class

9. Apply loads



Example: **create a custom load**.

For instance, create a distributed load that changes intensity as a function of time (using a sine function). Make it aligned to absolute Z.

One possibility is to implement a ChLoader-inherited class, and use it in a ChLoad.

A ChLoader is an object that is used by a ChLoad to apply loads to a ChLoadable object: this is useful especially when the load is distributed, because the ChLoad will automatically use ChLoader many times along the integration points.

There are some stubs for loaders in the ChLoaderU.h ChLoaderUV.h ChLoaderUVW.h , inherit from them. For example, inherit your custom loader from ChLoaderUV.h if you apply a per-unit-surface load, inherit from ChLoaderU.h if you apply a per-unit-length load, etc.

9. Apply loads

In this example, let's make a distributed time-dependant load for the last beam element. A load on the beam is a *wrench*, i.e. {force,load} per unit lenght aplied at a certain abscyssa U, that is a six-dimensional load.

- We inherit our class from ChLoaderUdistributed:

```
class MyLoaderHorizontalSine : public ChLoaderUdistributed {
public:
        // Constructor also sets ChLoadable
        MyLoaderHorizontalSine(std::shared_ptr<ChLoadableU> mloadable)
            :  ChLoaderUdistributed(mloadable) {}

        ...
```

9. Apply loads

- We must implement the **ComputeF()** function. Return load in F vector {force,torque}

```
            ...
            // Compute F=F(u)
virtual void ComputeF(const double U,     ///< parametric coordinate in line
             ChVectorDynamic<>& F,        ///< Result F vector goes here
             ChVectorDynamic<>* state_x,  ///< if != 0, update state (pos. part)
             ChVectorDynamic<>* state_w   ///< if != 0, update state (speed part)
                         ) override {
              assert(auxsystem);
              double T = auxsystem->GetChTime();
              double freq = 4;
              double Fmax = 20; // btw, Fmax is in Newton per unit length, this is a distributed load
              double Fz = Fmax*sin(T*freq); // compute the time-dependant load
              // Return load in F: {forceX,forceY,forceZ, torqueX,torqueY,torqueZ}.
              F.PasteVector( ChVector<>(0, 0, Fz), 0,0); // load, force part;
              F.PasteVector( ChVector<>(0,0,0)    ,3,0); // load, torque part;
            }
```

9. Apply loads

- Thanks to the ChLoad-ChLoader concept, Chrono will do the load integration automatically. Just remember to provide the number of integration points (1 suffices for U-constant loads like this)
- We also add a pointer to the ChSystem, it can be used to retrieve the absolute time in ComputeF()

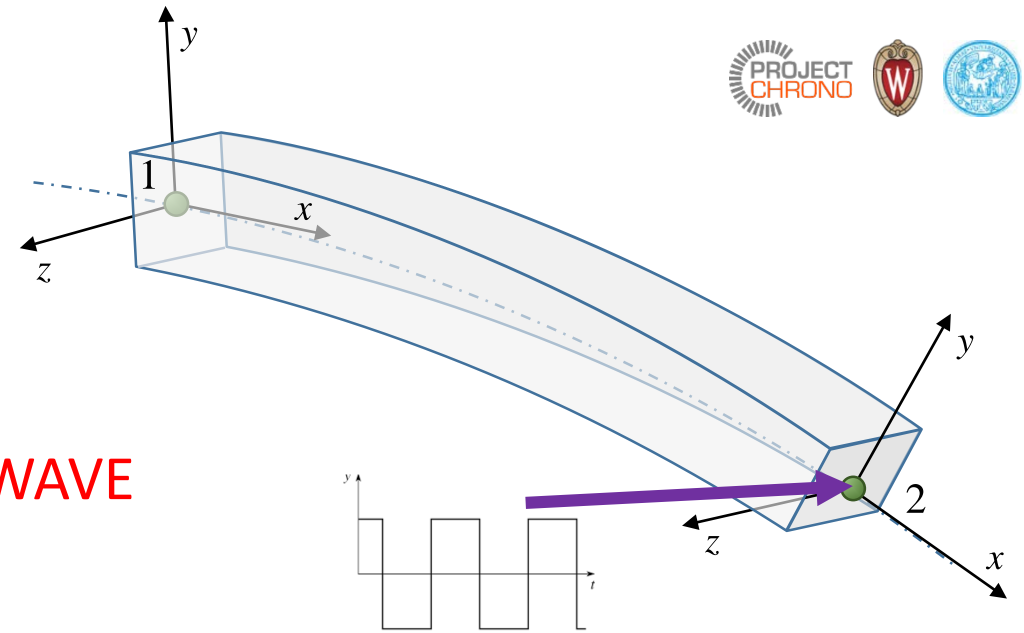
```
        ...
        // Needed because inheriting ChLoaderUdistributed.
        virtual int GetIntegrationPointsU() override {return 1;}

    public:
        // Add aux. data to the class, if you need to access it during ComputeF().
        ChSystem* auxsystem;
    };
```

9. Apply loads

- Create a custom load that uses our custom MyLoaderHorizontalSine loader. The ChLoad is a 'manager' for the ChLoader
- This is achieved using templates, that is instancing a ChLoad<my_loader_class>()

```
// Create a custom load that uses the custom loader above.
std::shared_ptr< ChLoad<MyLoaderHorizontalSine> > mloadsine (
                           new ChLoad<MyLoaderHorizontalSine>(beam_elements.back())
                                                              );

// Initialize auxiliary data of the loader, if needed
mloadsine->loader.auxsystem = &system;

// That's all. So not forget to add the load to the load container.
loadcontainer->Add(mloadsine);
```

Exercise 2



CREATE A CUSTOM LOAD CLASS AS BEFORE,
BUT THIS TIME:
- INSTEAD OF A SINE WAVE, TRY USING A SQUARE WAVE
- INSTEAD OF A CUSTOM DISTRIBUTED LOAD,
  USE A CUSTOM POINT LOAD (ATOMIC LOAD)

Hints:

- It is not much different than the previous custom load example.
- Instead of inheriting from ChLoaderUdistributed, inherit from ChLoaderUatomic class.
- To compute a square wave, look at the sign of a sine wave.
- Use a small value for the amplitude of the square wave, ex 0.8 Newton, otherwise the beam might wobble too much

Tutorial 3 solution

Solution (exercise 1)

```
/// Add a distributed load along the beam element:
auto mwrenchdis = std::make_shared<ChLoadBeamWrenchDistributed>(beam_elements.back());
mwrenchdis->loader.SetForcePerUnit(ChVector<>(0,-0.1,0)); // load per unit length
loadcontainer->Add(mwrenchdis);
```

Solution (exercise 2, part 1)

```
    // Create a custom atomic (point) load: a horizontal load changing in time as a square wave.
    // One has to implement the ComputeF() function to return the load in F. Remember the load on the beam is a
    // wrench, i.e. force+load per unit length applied at a certain abscissa U, that is a six-dimensional load.

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

            // Compute F=F(u)
            // This is the function that YOU MUST implement. It should return the
            // load at U. For Euler beams, loads are expected as 6-rows vectors, i.e.
            // a wrench: forceX, forceY, forceZ, torqueX, torqueY, torqueZ.
            virtual void ComputeF(const double U,      ///< parametric coordinate in line
                           ChVectorDynamic<>& F,       ///< Result F vector here
                           ChVectorDynamic<>* state_x, ///< if != 0, update state (pos. part)
                           ChVectorDynamic<>* state_w  ///< if != 0, update state (speed part)
                           ) override {
                assert(auxsystem);
                double T = auxsystem->GetChTime();
                double freq = 4;
                ...
```

Solution (exercise 2, part 2)

```
            ...
            double freq = 4;
            double Fmax = 0.8; // btw, Fmax is in Newtons, as a regular force: this is a point (atomic) loader
            double Fz;
            // compute the time-dependant load as a square wave
            if (cos(T*freq) >0)
                Fz =  Fmax;
            else
                Fz = -Fmax;
            // Return load in the F wrench: {forceX, forceY, forceZ, torqueX, torqueY, torqueZ}.
            F.PasteVector( ChVector<>(0, 0, Fz), 0,0); // load, force part;
            F.PasteVector( ChVector<>(0,0,0)    ,3,0); // load, torque part;
        }
    public:
        // add auxiliary data to the class, if you need to access it during ComputeF().
        ChSystem* auxsystem;
    };

    // Create a custom load that uses the custom loader above.
    std::shared_ptr< ChLoad<MyLoaderHorizontalSquarewave> > mloadsine (new
                                        ChLoad<MyLoaderHorizontalSquarewave>(beam_elements.back()) );
    mloadsine->loader.auxsystem = &system; // initialize auxiliary data of the loader, if needed
    loadcontainer->Add(mloadsine);  // do not forget to add the load to the load container.
```