
BriefFiniteElement.NET Documentation

Release stable

Aug 11, 2023

Contents

1	Under Construction	1
2	Elements Available	3
2.1	Finite Elements	3
2.2	MPC Elements	18
3	Loads Available	21
3.1	Elemental Loads	21
3.2	Nodal Loads	26
4	Materials Available	27
4.1	UniformIsotropicMaterial	27
4.2	UniformAnisotropicMaterial	29
5	Getting Started	31
5.1	Download source code of BriefFiniteElement.NET library	31
5.2	Create a project and compile BRiefFiniteElement from source code	34
5.3	Install BFE.NET Nuget Library	42
6	Examples	45
6.1	Small 3D Truss Example	45
6.2	LoadCase and LoadCombination Example	52
6.3	Iso Parametric Coordination System Of Elements Example	55
6.4	Inclined Frame Example	56
6.5	Element Load Coordination System Example	58
6.6	Cantilever Beam (Console Beam) Example	64
6.7	Sections for BarElement	65
7	Code Desgin Documentation and History	67
8	Miscellaneous Topics	69
8.1	Solving Procedure	69
8.2	Install Debugger Visualizers	74
9	Common Objects	79
9.1	Force	79
9.2	Displacement	80
9.3	LoadCase	81

9.4	LoadCombination	81
9.5	Point	81
9.6	Vector	82

CHAPTER 1

Under Construction

This documentation is under construction. And some of documented features (like telepathy link or ...) is not implemented in main library.

2.1 Finite Elements

2.1.1 BarElement

Overview

A bar element is referred to an 1D element, which only have dimension in one direction. It's features in an quick overview:

1. It can act as frame, beam, truss or shaft - see *Behaviours* section.

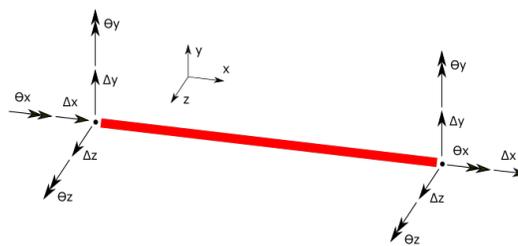


Fig. 1: DoFs of BarElement acting as a Frame

2. It can have a cross section - see *Cross Section* section.
3. It can have a material - see *Material* section.
4. Several types of loads are possible to be apply on them - see *Applicable Loads* section.
5. It Does have a local coordination system, apart from global coordination system - see *Coordination Systems* section.
6. It is possible to find internal force of it - see *Internal Force And Displacement* section.

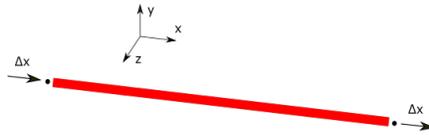


Fig. 2: DoFs of BarElement acting as a Truss

7. It can connect to nodes regarding partial fixity conditions - see BarElement-PartialEndRelease section.

Behaviours

BarElement.Behaviour property is an enum flag (enum flag means an enum that can have several values at same time). It can be set to frame, beam, truss, shaft etc. The possible behaviours for the BarElement is:

- BarElementBehaviour.EulerBernoulyBeamY : Beam in Y direction based on Euler-Bernouly theory. DoFs are shown in below image:

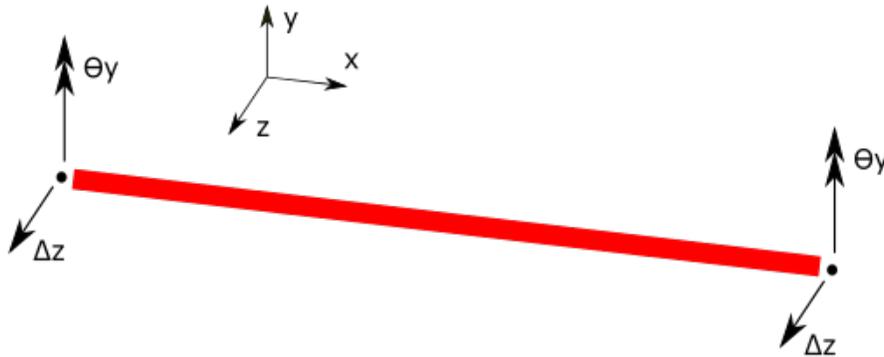


Fig. 3: DoFs of BarElementBehaviour.EulerBernoulyBeamY

- BarElementBehaviour.EulerBernoulyBeamZ : Beam in Z direction based on Euler-Bernouly theory. DoFs are shown in below image:
- BarElementBehaviour.TimoshenkoBeamY : Beam in Y direction based on Timoshenko's theory (shear deformation). DoFs are shown in below image:
- BarElementBehaviour.TimoshenkoBeamZ : Beam in Z direction based on Timoshenko's theory (shear deformation). DoFs are shown in below image:
- BarElementBehaviour.Truss : Only axial load carrying. DoFs are shown in below image:

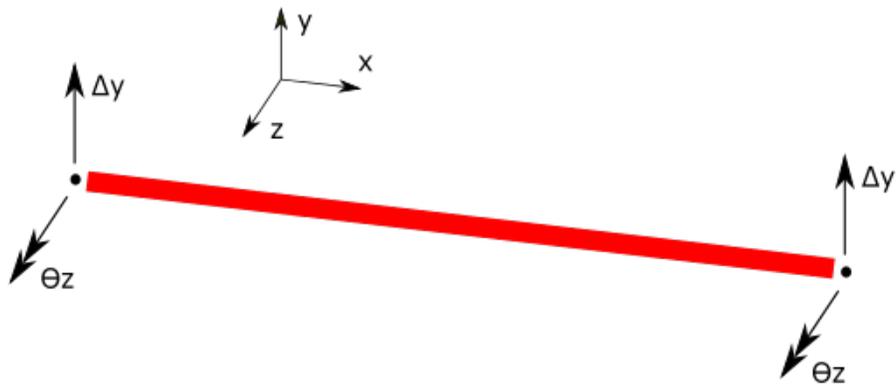


Fig. 4: DoFs of `BarElementBehaviour.EulerBernoulyBeamZ`

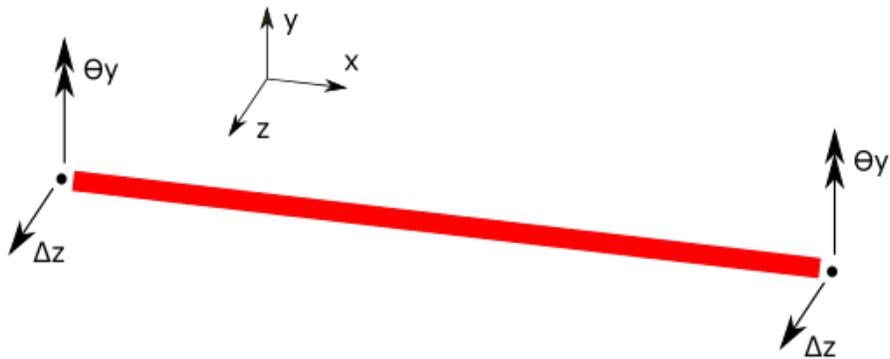


Fig. 5: DoFs of `BarElementBehaviour.TimoshenkoBeamY`

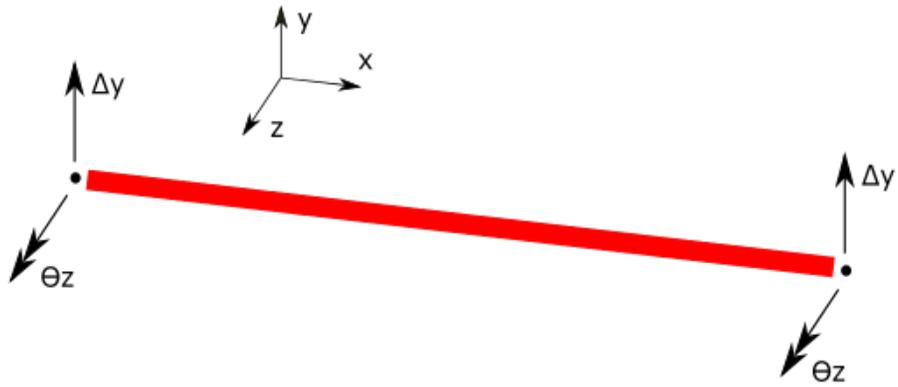


Fig. 6: DoFs of BarElementBehaviour.TimoshenkoBeamZ

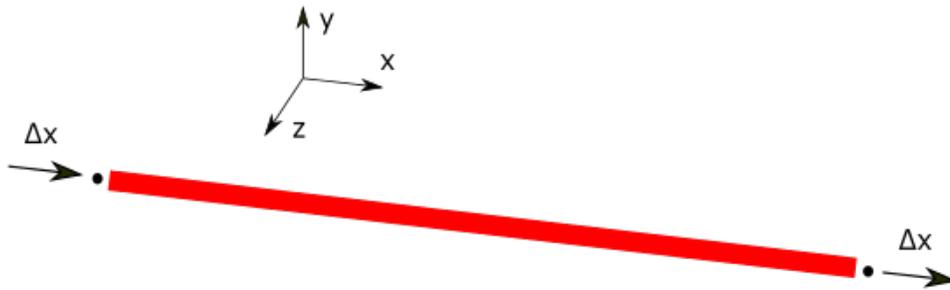


Fig. 7: DoFs of BarElementBehaviour.Truss

- `BarElementBehaviour.Shaft` : Only torsional moment carrying. DoFs are shown in below image:

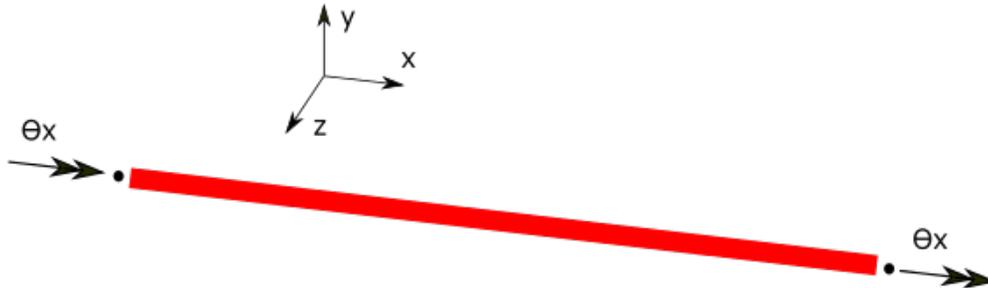


Fig. 8: DoFs of `BarElementBehaviour.Shaft`

These behaviours can be combined, for example a truss member should only have a Truss behaviour, but a 3d frame member does have two beam behaviour in Y and Z directions, a truss behaviour and a shaft behaviour, (all these behaviours at the same time).

This is an example which makes a `BarElement` with truss behaviour which in real acts as a truss member that only can carry axial load:

```
var bar = new BarElement();
bar.Behaviour = BarElementBehaviour.Truss;
```

There is another utility static class named `BarElementBehaviours` which contains predefined combination behaviours for `BarElement` which is more user (developer) friendly than original enum flag. This is example usage of `BarElementBehaviours` class:

```
var bar = new BarElement();
bar.Behaviour = BarElementBehaviours.FullFrame;
```

Which is exactly equal to:

```
var bar = new BarElement();
bar.Behaviour = BarElementBehaviour.Truss | BarElementBehaviour.BeamYEulerBernoulli |
↳BarElementBehaviour.BeamZEulerBernoulli | BarElementBehaviour.Shaft;
```

So better to use `BarElementBehaviours` unless needed manually define combination of behaviours.

- `BarElementBehaviours.FullBeam` and `BarElementBehaviours.FullBeamWithShearDefomation:`
- `BarElementBehaviours.FullFrame` and `BarElementBehaviours.FullFrameWithShearDeformation:`

Cross Section

`BarElement` is modelled as a 1D element, and it needs to have geometrical values of its cross section (like `A`, `Iy`, `Iz`, etc.). `BarElement.CrossSection` does define a cross section for `BarElement`. The type `Base1DSection` is base class that is used for defining a cross section for bar element. This class is a general class which can give every information of section's geometric properties at specific location of length of element. All other cross sections of bar element are inherited from `Base1DSection` class.

UniformParametric1DSection

Inherited from `Base1DSection`, defines a uniform section for the `BarElement`. Uniform section means that section does not change along the length of bar. Parametric means that properties are parametrically defined one by one. for example if we have a circular section, with $Iy = Iz = J/2 = 1e-6 \text{ m}^4$, $A = 2e-6 \text{ m}^2$ then:

```
var section = new UniformParametric1DSection();
section.A = 1e-4;
section.Iy = section.Iz = 1e-6;
section.J = 2e-6;

var bar = new BarElement();
bar.CrossSection = section;
```

Hint: Note that two properties `Ay` and `Az` of `UniformParametric1DSection`, are about shear areas of section and their value will not be used unless `BarElement` have one of `TimoshenkoBeam` behaviours.

UniformGeometric1DSection

Inherited from `Base1DSection`, defines a uniform section for the `BarElement`. Uniform section means that section does not change along the length of bar. Geometric means that section properties are defined as polygon.

Important Note: In `UniformGeometric1DSection` it is only possible to define one polygon. if polygon contains nested holes etc., then it should convert to one polygon. See [Sections for BarElement](#)

Important Note: The way that geometric properties are calculated for section is defined here: (https://en.wikipedia.org/wiki/Second_moment_of_area#Any_polygon). Maybe this method be not applicable to thin walled section.

Important Note: there is no analytical solution for finding torsional constant `J` for noncircular sections, in those cases user must set `UniformGeometric1DSection.JOverriden` property to correct value, otherwise polar area moment will be used (<https://github.com/BriefFiniteElementNet/BriefFiniteElement.Net/issues/38>)

Important Note: The geometric section is defined in Y-Z plane of local coordination system of element. the X axis in local coordination system is the beam length direction.

Material

`BarElement.Material` property defines a material for this element. the type `BaseMaterial` is base class that is used for defining a material for a finite element. This class is a general class which can give every information of section's materials at specific location of length of element.

UniformIsotropicMaterial

This class is inherited from `BaseMaterial` and defines a uniform material for the finite elements. Uniform material means that material does not change along the length of bar. Parametric means that properties are parametrically defined. for example if we have a steel material, with Elastic module = 210 GPa = 210e9 Pa, and Poisson's ratio = 0.3 then:

```
using BriefFiniteElement.Elements
using BriefFiniteElement.Materials

var material = UniformIsotropicMaterial.CreateFromYoungPoisson(210e9, 0.3);
var bar = new BarElement();

bar.Material = material;
```

Applicable Loads

There are several loads currently applicable to `BarElement``. *UniformLoad*, *ConcentratedLoad* and *NonUniformLoad* are applicable loads.

Coordination Systems

Local Coordination System

Local coordination system for `BarElement` has three axis that we name x' , y' and z' .

TODO with images

Relation of global and local system

“The global axes are brought to coincide with the local member axes by sequence of rotation about y, z and x axes respectively. This is referred to an y-z-x transformation.” ref[0].

Imagine a bar element with start node N1 located at (x_1, y_1, z_1) and end node N2 located at (x_2, y_2, z_2) . Four steps are needed to find the directions of the local axis x' - y' - z' :

- Step 1:

Move the element in a way that N1 be placed at origins of global system. TODO: Image

- Step 2:

Rotate global system about global Y axis rotated X axis goes under element length (shown as β in image below). Note that if element is vertical (e.g. $x_1 = x_2$ and $y_1 = y_2$ and $z_1 = z_2$) no need to do this step. TODO: Image

- Step 3:

Rotate the system from previous step about its Z axis in a way that X axis go exactly through same direction of element's length (shown as γ in image below). TODO: Image

- Step 4:

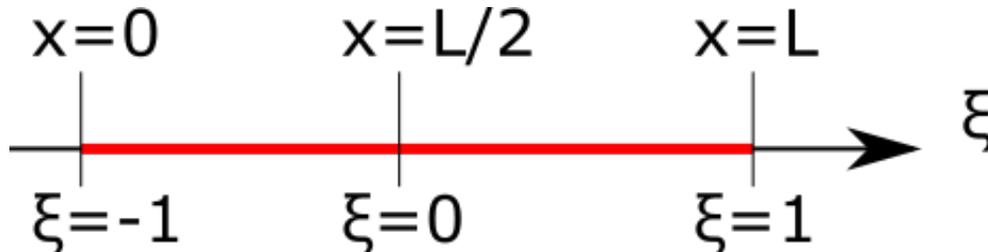
If element have any custom web rotation α , do rotate system about its X axis by α : TODO: Image

the result system is local system of bar element.

Iso Parametric Coordination System

Apart from local and global coordination systems for elements, there is another system based on isoparametric formulation/representation, which is used extensively in finite element method. In BFE also in many places instead of local coordinate system, the iso parametric coordination is used.

Iso Parametric Coordination system for BarElement with two nodes



Based on

- At the beginning point of the element, where $x=0$ the iso parametric coordinate is $\xi=-1$
- At the central point of the element, where $x=L/2$, and L is length of elements, the iso parametric coordinate is $\xi=0$
- At the end point of the element, where $x=L$, and L is length of elements, the iso parametric coordinate is $\xi=1$

In bar element with two nodes the relation between isoparametric ξ coordinate and local x coordinate is:

$$x = (\xi + 1) * L / 2 \text{ and subsequently}$$

$$\xi = (2 * x - L) / L$$

Iso Parametric Coordination system for BarElement with more than two nodes

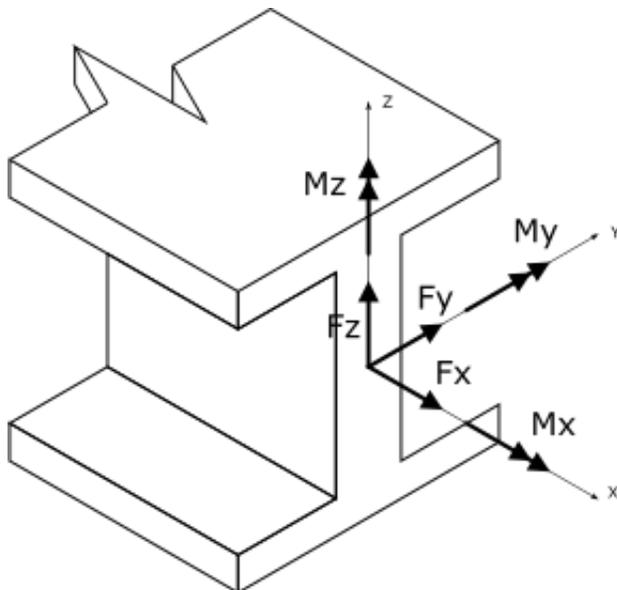
There is no simple formula to show relation of ξ and x in elements with more than two nodes, but there is a method for conversion between local coordinate system and isoparametric coordination system `BarElement.LocalCoordsToIsoCoords` and `BarElement.IsoCoordsToLocalCoords` which works right with any number of node. As this method is defined in base `Element` class, input and output of these is double array, but as `BarElement` is one dimensional element, then only first member of array have value and that is X or ξ .

ref[1]: Finite Element Analysis: Theory and Programming by by C Krishnamoorthy p.243

Internal Force And Displacement

After solving the Model, `BarElement` will have some internal forces. Internal force at each location of element can be different and it can be retrieved with method `BarElement.GetInternalForceAt` and `BarElement.GetExactInternalForceAt`. both methods gives the internal force at specified iso parametric coordinate. The difference between `BarElement.GetInternalForce` and `BarElement.GetExactInternalForce` is that `BarElement.GetInternalForceAt` only consider nodal displacement for internal force but Exact one (`GetExactInternalForceAt`) also considers effect of elemental loads (like distributed loads) in element in addition to nodal displacements. Internal force means 3 forces and 3 moments: axial load (F_x), two shear loads (F_y, F_z), torque moment (M_x) and two biaxial moments (M_y, M_z) which are shown in picture:

Note that value returned from this method is in element's local coordination system.



For example the beam below with both ends fixed, after solve does not have any nodal displacement, so the standard FEM formula $D*B*u$ will return 0, so `BarElement.GetInternalForceAt` for this example returns 0, but `BarElement.GetExactInternalForceAt` at middle will not return zero...

```
var model = new Model();

Node n1, n2;

model.Nodes.Add(n1 = new Node(0, 0, 0) { Constraints = Constraints.Fixed });
model.Nodes.Add(n2 = new Node(1, 0, 0) { Constraints = Constraints.Fixed });

var elm = new BarElement(n1, n2);

elm.Section = new BriefFiniteElementNet.Sections.UniformParametric1DSection(a: 0.
↪01, iy: 0.01, iz: 0.01, j: 0.01);
elm.Material = BriefFiniteElementNet.Materials.UniformIsotropicMaterial.
↪CreateFromYoungPoisson(210e9, 0.3);

var load = new Loads.UniformLoad();

load.Case = LoadCase.DefaultLoadCase;
load.CoordinationSystem = CoordinationSystem.Global;
load.Direction = Vector.K;
load.Magnitude = 10;

elm.Loads.Add(load);
model.Elements.Add(elm);

model.Solve_MPC();

var f1 = elm.GetInternalForceAt(0);
var f2 = elm.GetExactInternalForceAt(0);
```

TODO: internal displacement

Partial End Release

By default connection of BarElement into end nodes are rigid, e.g. all DoFs of BarElement are connected to end node, but there are some situation that there is need for partial connections.

BarElement.NodalReleaseConditions defines the partial end release of BarElement on each of it's nodes. Also BarElement.StartReleaseCondition and BarElement.EndReleaseCondition uses this property to get/set release conditions for start and end nodes:

```

/// <summary>
/// Gets or sets the connection constraints od element to the start node
/// </summary>
public Constraint StartReleaseCondition
{
    get { return _nodalReleaseConditions[0]; }
    set { _nodalReleaseConditions[0] = value; }
}

/// <summary>
/// Gets or sets the connection constraints od element to the end node
/// </summary>
public Constraint EndReleaseCondition
{
    get { return _nodalReleaseConditions[_nodalReleaseConditions.Length - 1]; }
    set { _nodalReleaseConditions[_nodalReleaseConditions.Length - 1] = value; }
}

```

There are 3 properties for BarElement for taking end releases into consideration:

```

“ public Constraint StartReleaseCondition{get;set;} public Constraint EndReleaseCondition{get;set;} public Con-
straint[] NodalReleaseConditions{get;set;} “

```

StartReleaseCondition Gets or sets the release condition to first node, EndReleaseConditionGets or sets the release condition to last node and NodalReleaseConditions Gets or sets the release condition to all nodes (it is an array).

2.1.2 TriangleElement

Overview

A triangle element is referred to a 2D element, which only have dimension in two direction. It's features in an quick overview:

1. It can act as thin shell, thick shell, plate bending or membrane - see *Behaviours* section.

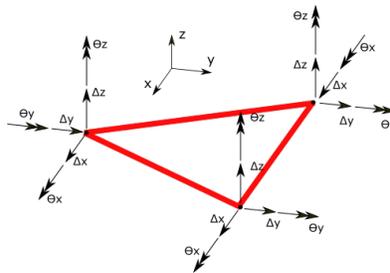


Fig. 9: DoFs of TriangleElement acting as a Shell

2. It can have a cross section - see *Cross Section* section.

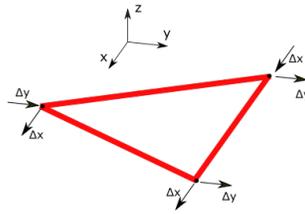


Fig. 10: DoFs of `TriangleElement` acting as a Membrane

3. It can modeled as [PlaneStress](#) or [PlainStrain](#) - see [TriangleElement-MembraneFormulation](#) section.
4. It can have a material - see [Material](#) section.
5. Several types of loads are possible to be apply on them - see [Applicable Loads](#) section.
6. It Does have a local coordination system, apart from global coordination system - see [Coordination Systems](#) section.
7. It is possible to find internal force of it - see [Internal Force](#) section.

Behaviours

`TriangleElement.Behaviour` property is an enum flag (enum flag means an enum that can have several values at same time). It can be set to `ThinShell`, `TODO` etc. The possible behaviours for the `TriangleElement` is:

- `TriangleElementBehaviour.Membrane` : Membrane behaviour for in-plane displacement. DoFs are shown in below image:

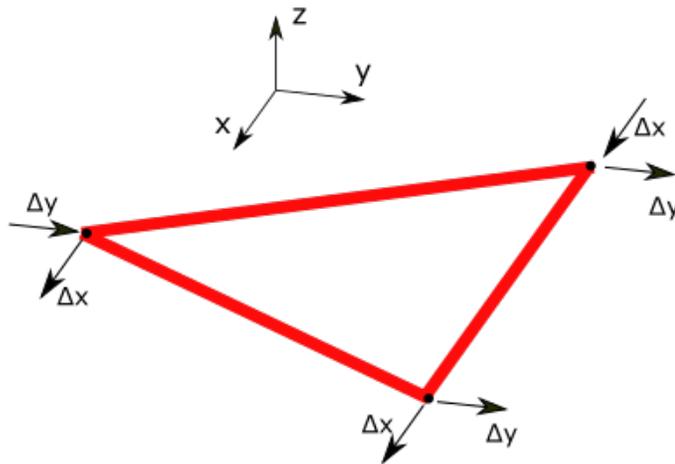


Fig. 11: DoFs of `TriangleElementBehaviour.PlateBending`

The mathematic formulation of this behaviour is based on standard CST ([Constant Stress/Strain Triangle](#)) element.

- `TriangleElementBehaviour.PlateBending`: `PlateBending` behaviour for in-plane rotations and out of plane displacements. DoFs are shown in below image:

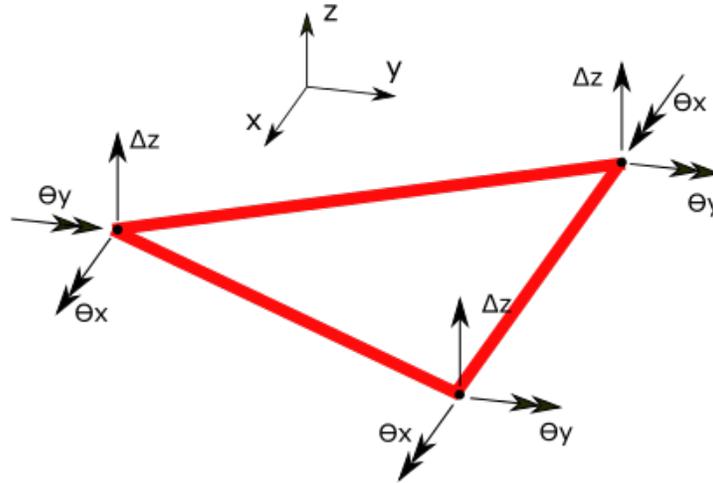


Fig. 12: DoFs of `TriangleElementBehaviour.PlateBending`

- `TriangleElementBehaviour.DrillingDof`: behaviour for out of plane rotations. DoFs are shown in below image:

The mathematic formulation of this behaviour is based on DKT (Discrete Kirchhoff Triangle) element.

These behaviours can be combined, for example a Membrane member should only have a Membrane behaviour, but a thin shell member does have behaviour of platebending and a membrane behaviour (both at the same time).

This is an example which makes a `TriangleElement` with plate bending behaviour which in real acts as a plate bending member that only can carry normal loads and in plate bendings:

```
var tri = new TriangleElement();
tri.Behaviour = TriangleElementBehaviour.ThinPlate;
```

There is another utility static class named `TriangleElementBehaviours` which contains predefined combination behaviours for `TriangleElement` which is more user (developer) friendly than original enum flag. This is example usage of `TriangleElementBehaviours` class:

```
var tri = new TriangleElement();
tri.Behaviour = TriangleElementBehaviours.ThinShell;
```

Which is exactly equal to:

```
var tri = new TriangleElement();
tri.Behaviour = TriangleElementBehaviour.ThinPlate | TriangleElementBehaviour.
    Membrane | TriangleElementBehaviour.DrillingDof;
```

So better to use `TriangleElementBehaviours` unless needed manually define combination of behaviours.

- `TriangleElementBehaviours.ThinShell` and `TriangleElementBehaviours.ThickShell`:

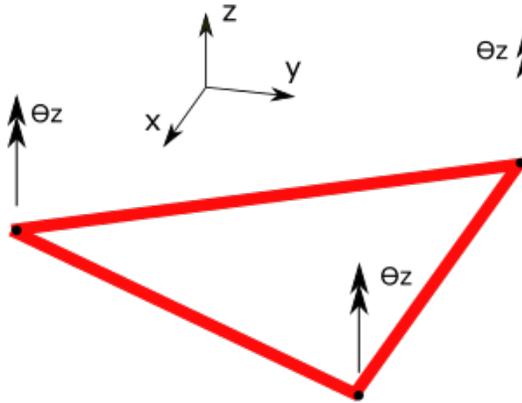


Fig. 13: DoFs of TriangleElementBehaviour.DrillingDof

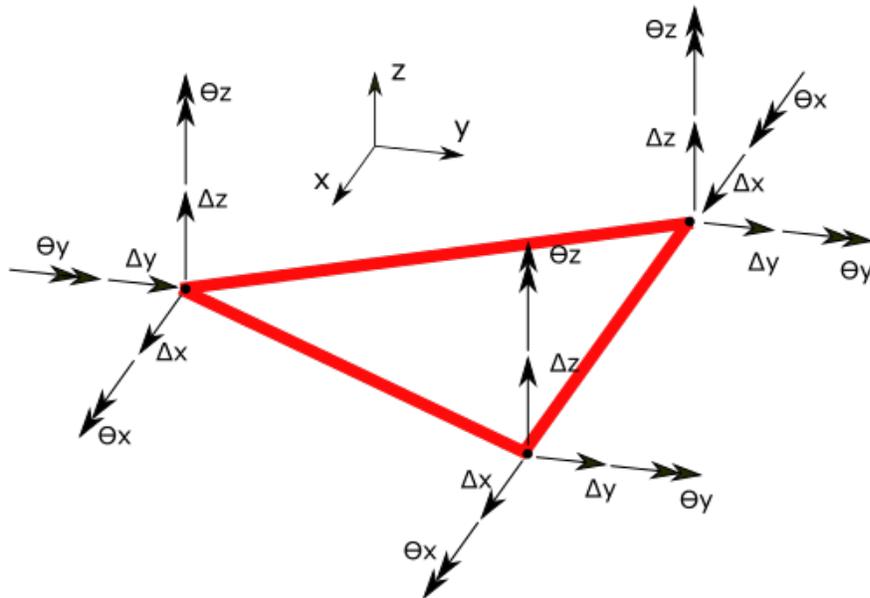


Fig. 14: DoFs of TriangleElementBehaviours.ThinShell and TriangleElementBehaviours.ThickShell

Cross Section

`TriangleElement` is modelled as a 2D element, and it needs to have thickness values of its cross section. `TriangleElement.Section` does define a cross section for `TriangleElement`. The type `Base2DSection` is base class that is used for defining a cross section for triangle element. This class is a general class which can give every information of section's geometric properties at specific location of surface of element. In this case `Base2DSection` gets the isometric location of any arbitrary point and returns the thickness of section at that point. All other cross sections of triangle element are inherited from `Base2DSection` class.

UniformParametric2DSection

Inherited from `Base2DSection`, defines a uniform section for the `TriangleElement`. Uniform section means that section thickness or probably other geometric properties does not change along the surface of triangle. Parametric means that properties are parametrically defined one by one. for example if we have a section, with thickness = 1 cm then:

```
var section = new Base2DSection();
section.Thickness = 0.01;

var tri = new TriangleElement();
tri.CrossSection = section;
```

Material

`TriangleElement.Material` property defines a material for this element. the type `BaseTriangleMaterial` is base class that is used for defining a material for bar element. This class is a general class which can give every information of section's materials at specific location of surface of element.

All other materials of triangle section are inherited from `BaseTriangleMaterial` class.

UniformParametricTriangleMaterial

This class is inherited from `BaseTriangleMaterial` and defines a uniform material for the triangle element. Uniform material means that material does not change along the surface of triangle. Parametric means that properties are parametrically defined (like ``UniformParametricTriangleMaterial.E`` and ``UniformParametricTriangleMaterial.G``). for example if we have a steel material, with $E = 210 \text{ GPa}$, $G = 80 \text{ GPa}$ then:

```
var steelMaterial = new UniformParametricTriangleMaterial();
steelMaterial.E = 210e9; //210 * 10^9 Pas
steelMaterial.G = 80e9; //80 * 10^9 Pas

var tri = new TriangleElement();
tri.Material = steelMaterial;
```

Applicable Loads

There are several loads currently applicable to ``TriangleElement``.

Uniform Load

Uniform load is a uniform, per length load in $[N/m^2]$ dimension, which is applied on the bar element.

[image]

The uniform load have three components, U_x , U_y , U_z which is per length force component in X, Y and Z directions. Please note that if coordination system of load is set to global, U_x and U_y and U_z will be in global directions, else will be in element's local coordination system. TODO: uniform load changed

Example:

Concentrated Load

Concentrated load is a single concentrated load which is applying in a point which exists on the `BarElement`'s length.

Example:

Trapezoidal Load

Trapezoidal load is a linearly varying load, with specific start and end, which is applied on the bar element. This is more general than `UniformLoad`

Coordination Systems

Local Coordination System

Local coordination system for `TriangleElement` has tree axis that we name x , y and z .

Relation of global and local system

local axis x is parallel with the point that connects `node[0]` to `node[1]` of element.

local axis z is normal to triangle's surface.

local axis y is normal to both x and z .

This formulation (and image above) taken from Development of Membrane, Plate and Flat Shell Elements in Java from author Kansara, Kaushalkumar available from <https://vtechworks.lib.vt.edu/handle/10919/9936>

Internal Force

After solving the Model, `TriangleElements` will have some internal forces. Internal force at each location of element can be different and it can be caught with method `TriangleElement.GetInternalForce`. with get the location that you need the internal force as input. Internal force means membrane and bending tensors which are shown in picture: TODO show with returned axis directions

Note that value returned from this method is in element's local coordination system. to convert to global system: TODO

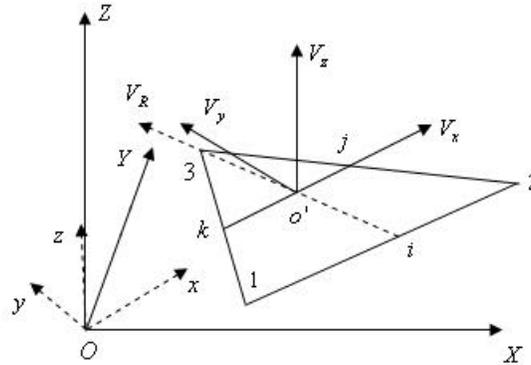


Fig. 15: local coordination system of TriangleElement

2.1.3 TetrahedronElement

Tetrahedron element is only volume element in BFE.

DoFs Tetrahedron element in BFE does have four nodes, each one for one corner of tetrahedron. The tetrahedron element only support 3 translational DoF for each node.

There are several finite elements available in library. Each finite element does provide stiffness, mass and damp matrices. Their difference with special elements is that normal elements does provide stiffness, mass and damp matrices and usually have material, geometrical properties (like section on thickness). Finite elements are inherited from `BriefFiniteElement.Elements.Element` class.

Overview of Finite Elements available:

- BarElement: A 1D, 2 noded element
- TriangleElement: A 2D, 3 noded element
- TetrahedronElement: A 3D, 4 noded element

2.2 MPC Elements

2.2.1 RigidElement

RigidElement is an MpcElement, with virtually infinite stiffness. Nodes contained in a RigidElement have no relative displacement regarding eachother. Best usecase is rigid diaphragm in structures. Some other use cases of rigid element can be found at: https://mashayekhi.iut.ac.ir/sites/mashayekhi.iut.ac.ir/files//files_course/lesson_16.pdf

This does represents a nondeformable element or element with infinite stiffness which does not deform. For more info see <http://www.codeproject.com/Articles/850733/RigidElements-in-BriefFiniteElement-NET>

ref[1]: <http://www.edwilson.org/book-wilson/07-cons~1.doc>

TODO: an appropriated image

2.2.2 VirtualSupport

VirtualSupport is an MpcElement, that can fix any free dofs of model. Technically there is no difference between using *Node.Constraint* and *VirtualSupport* element in these two version of code, both will have exactly same result after solve:

```
for(var i = 0;i < 10;i ++)
{
    model.Nodes[i].Constraint = Constraints.FixedDx;
    model.Nodes[i].Settlements = new Displacement(0.1,0,0,0,0,0);
}
```

```
var elm = new VirtualConstraint();
elm.Constraint = Constraints.FixedDx;
elm.Settlement = new Displacement(0.1,0,0,0,0,0);

for(var i = 0;i < 10;i ++)
    elm.Nodes.Add(model.Nodes[i]);

model.MpcElements.Add(elm);
```

but the second one will let user to define settlements for specific LoadCases. Or set some constraints for only specific load cases. for example it is possible to have set constraint in loadcase *A* when rigid elements are applied only in analysing with load case *B*.

2.2.3 HingLink

HingLink is an MpcElement which connects only translational DoFs of nodes together. There is a restriction where node's location must be same (exactly same) otherwise it throws exception. example usage are connecting a slab into beam with simple connection. where slab for example have 4 nodes, and 4 column in corners each one have 2 nodes, total nodes are 12 nodes and top nodes of columns are connected to slab, each with a separate hing link.

hing link is some sort of link that connects two nodes to each other, but only connect their Dx, Dy, Dz together not rotations of them. Limitation is that both nodes should have same location. Using this sort of link, it is possible to model end release in Shells, etc.

MPC elements or Multi-Point Constraint elements, are kind of virtual elements that binds several DoFs of a model together and reduces the overall number of independent DoFs using technique MPC (Multi-Point Constraints) and Master/Slave model. All MPC elements are inherited from "BriefFiniteElement.Elements.MpcElement".

Overview of special elements available:

- TelepathyLink: Partially binds DoFs of several nodes together
- RigidElement: An non-deformable element with virtually infinite (∞) stiffness
- VirtualConstraint: An element that virtually binds its nodes into ground and make them support nodes.

more info: https://mashayekhi.iut.ac.ir/sites/mashayekhi.iut.ac.ir/files//files_course/lesson_16.pdf

MpcElement have a feature than can be taken into account in particular loads. For example when analyzing a Model against Eq. loads, a rigid diaphragm (with infinite stiffness) can be considered. This rigid diaphragm should not be applied when model is solving against other loads like Dead or Live loads. There are three properties for MpcElement class regarding this feature:

- MpcElement.UseForAllLoads:

It is false by default, if set to true then `MpcElement` will be applied in every situation and all loads. Set this to true, when `MpcElement` should be considered against all loads.

- `MpcElement.AppliedLoadCases`

By defaults is empty, `MpcElement` will be applied when structure is analysing with `LoadCases` inside this.

- `MpcElement.AppliedLoadTypes`

By defaults is empty, `MpcElement` will be applied when structure is analysing with a `LoadCase` which have a `LoadCase.LoadType` which is present inside this.

Example 1: An `MpcElement` (rigid element) which connect several nodes and only taken into account with loads with type of `Eq`.

In this model there are `TODO` number of roofs, that are only considered when `Eq`. loads are applied.

TODO image

3.1 Elemental Loads

3.1.1 ConcentratedLoad

`ConcentratedLoad` in namespace `BriefFiniteElementNet.Loads`, is a concentrated load which can apply on 1D (like `BarElement`), 2D (like `TriangleElement`) or 3D (like `TetrahedronElement`) elements.

Force

`ConcentratedLoad.Force` which is a `Force` property, defines the amount of force that is applied on the element body.

IsoPoint

The iso-parametric location of force inside element's body

CoordinationSystem

`ConcentratedLoad.CoordinationSystem` which is a enum typed property, defines the coordination system of load. It can only have two different values of `CoordinationSystem.Global` or `CoordinationSystem.Local`:

- `CoordinationSystem.Global`: The load is assumed in global coordination system
- `CoordinationSystem.Local`: The load is assumed in local coordination system of element that load is applied to (each element type have different local coordination system which is stated in appropriated section).

Look at *Element Load Coordination System Example* for more information on how to use.

3.1.2 UniformLoad

`UniformLoad` in namespace `BriefFiniteElementNet.Loads`, is a constant distributed load which can apply on 1D (like `BarElement`), 2D (like `TriangleElement`) or 3D (like `TetrahedronElement`) elements. Self weight loads are good examples that can be modeled with this type of load.

Here are examples illustrated in image (note that many of these loads are not available in this library!)

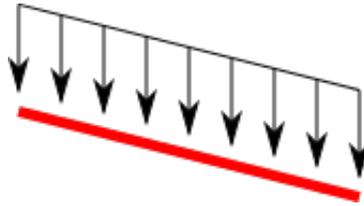


Fig. 1: `UniformLoad` applying on a `BarElement`'s body

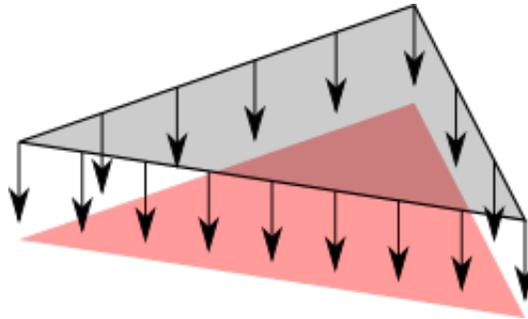


Fig. 2: `UniformLoad` applying on a `TriangleElement`'s body

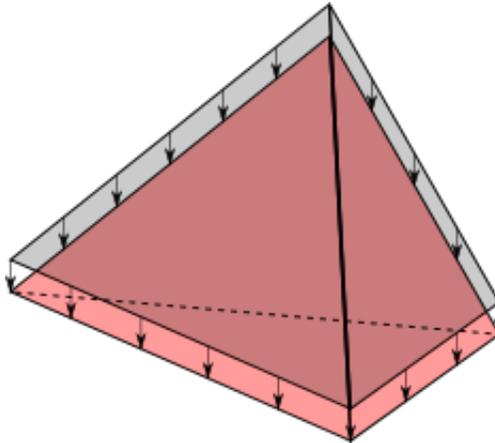


Fig. 3: `UniformLoad` applying on a `TetrahedronElement`'s body

Magnitude

`UniformLoad.Magnitude` which is a double property of `UniformLoad`, defines the Magnitude of uniform load. Based on `UniformLoad` is applied on what element, the dimension is different:

- If it is applied on a 1D element like `BarElement`, then the dimension is $[N/m]$

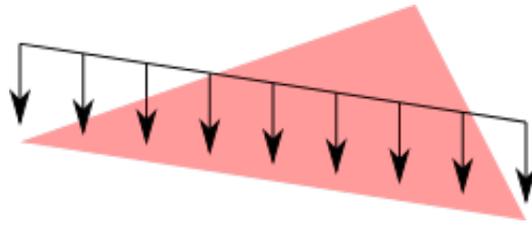


Fig. 4: UniformLoad applying on one of a TriangleElement's edges

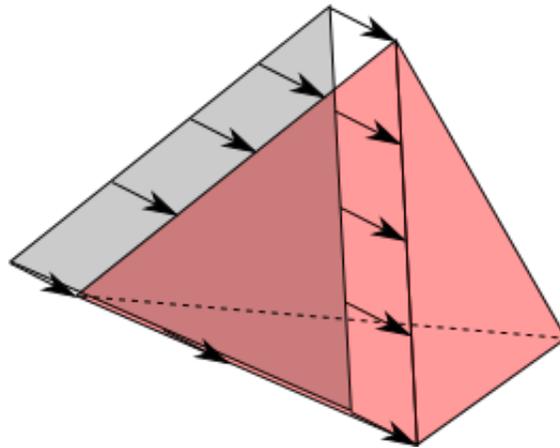


Fig. 5: UniformLoad applying on one of a TetrahedronElement's faces

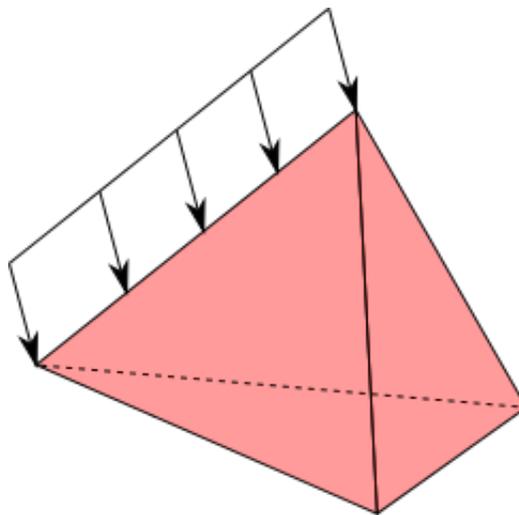


Fig. 6: UniformLoad applying on one of a TetrahedronElement's edges

- If it is applied on a 2D element like `TriangleElement`, then the dimension is $[N/m^2]$
- If it is applied on a 3D element like `TetrahedronElement`, then the dimension is $[N/m^3]$

Coordination System

`UniformLoad.CoordinationSystem` which is a enum typed property of `UniformLoad`, defines the coordination system of uniform load. It can only have two different values of `CoordinationSystem.Global` or `CoordinationSystem.Local`:

- `CoordinationSystem.Global`: The load is assumed in global coordination system
- `CoordinationSystem.Local`: The load is assumed in local coordination system of element that load is applied to (each element type have different local coordination system which is stated in appropriated section).

Look at *Element Load Coordination System Example* for more information on how to use.

LoadDirection (Obsolete: see Direction)

`UniformLoad.LoadDirection` which is a enum typed property of `UniformLoad`, defines the direction of uniform load. It can only have three different values of `LoadDirection.X` or `LoadDirection.Y` or `LoadDirection.Z`.

Look at examples section for more information on how to use.

TODO: obsolete the enum `LoadDirection` and use a vector for more enhanced usage.

Direction

`UniformLoad.Direction` which is a property of `UniformLoad` with type of `Vector`, defines the direction of uniform load. An instance of `Vector` class defines a vector in 3d space with three components of X, Y and Z. Note that length of vector is not taken into account, only its direction is used.

Look at examples section and definition of local `CoordinationSystem` in `BarElement`, `TriangleElement`, etc. for more information on how to use.

Examples

Related Examples:

- `_element-load-coordination-system`

3.1.3 PartialNonUniformLoad

`PartialNonUniformLoad` in namespace “`BriefFiniteElementNet.Loads`“, is a varying distributed load which can apply on 1D (like `BarElement`), 2D (like `TriangleElement`) or 3D (like `TetrahedronElement`) elements.

Here are examples illustrated in image:

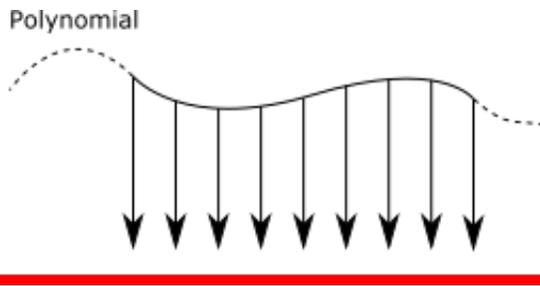


Fig. 7: PartialNonUniformLoad applying on a BarElement

SeverityFunction

Severity function which defines how much is the amount of load in each iso location. For example a linear varying load from $x_i = -0.5$ to $x_i = +0.5$ and start magnitude = 100 N/m and end magnitude 50 N/m , severity function in -0.5 equals 100 and severity function $+0.5$ equals 50

StartLocation

Isoparametric coordination of start location of load on the element. all members of *TrapezoidalLoad.StartIsoLocations* must be in range $[-1, +1]$.

EndLocations

Isoparametric coordination of end location of load on the element. all members of *TrapezoidalLoad.EndIsoLocations* must be in range $[-1, +1]$.

Coordination System

PartialNonUniformLoad.CoordinationSystem which is a enum typed property of *PartialNonUniformLoad*, defines the coordination system of trapezoidal load. It can only have two different values of *CoordinationSystem.Global* or *CoordinationSystem.Local*:

- *CoordinationSystem.Global*: The load is assumed in global coordination system
- *CoordinationSystem.Local*: The load is assumed in local coordination system of element that load is applied to (each element type have different local coordination system which is stated in appropriated section).

Example 1: Linear Varying load

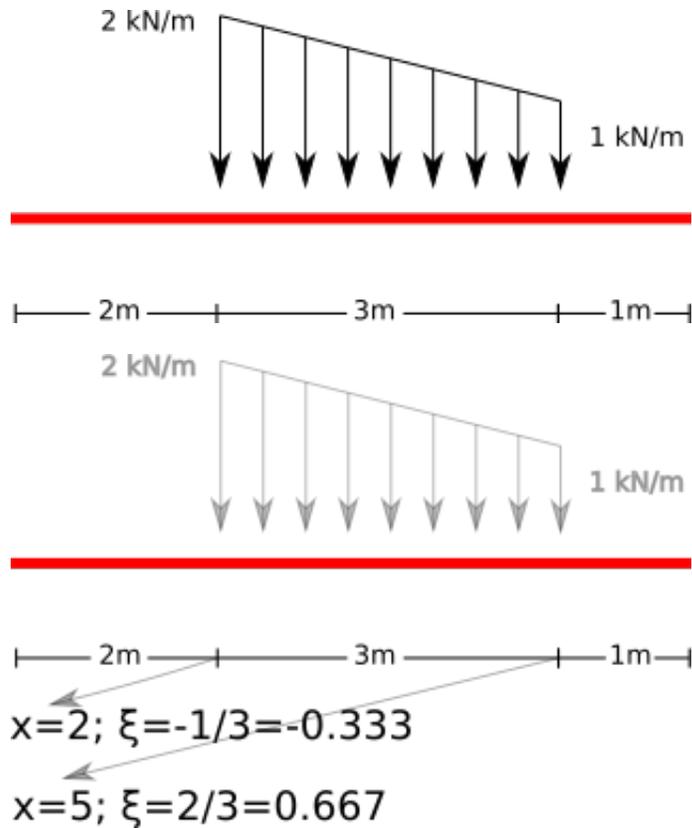
For load shown in the image:

first need to calculate isoparametric coords of locations (see see *Iso Parametric Coordination System Of Elements Example* section for more info)

```
“ var load = new PartialNonUniformLoad(); //creating new instance of load
load.SeverityFunction = Mathh.SingleVariablePolynomial.FromPoints(-1/3, 2000, 2/3, 1000); //this is a totally linear load, defined from two points. since it is partial, need to tell bfe the span this load is applied
```

```
load.StartLocation = new IsoPoint(-1/3); //set locations of trapezoidal load
load.EndLocation = new IsoPoint(2/3); //set locations of trapezoidal load “
```

ElementLoad is a base class that can only apply on the *Element*. There are several “*ElementLoad*“s:



- `UniformLoad`: A uniform load that can apply on a `Element` or one of its faces or edges.
- `PartialNonuniformLoad`: A Partial varying load.
- `ConcentratedLoad`: A concentrated load that applies on a single point in `Element`'s body

3.2 Nodal Loads

TODO: overview of what are nodal loads.

There are two types of load in general: `NodalLoad` and `ElementLoad`. `NodalLoad` does apply on nodes and only have concentrated load, but `ElementLoad` is abstract base class and does apply on elements instead of nodes. Image can shows the difference better:

All loads (including nodal and elemental) have `LoadCase`. See `LoadCase` and combinations for more info.

4.1 UniformIsotropicMaterial

4.1.1 Overview

inherited from `BaseMaterial`, this represents a **uniform** and **isotropic** material:

- **uniform** means material properties are not varying through the element's body, or in every location of element material properties are identical.
- **isotropic** means having identical values of a property in all directions

4.1.2 YoungModulus

`UniformIsotropicMaterial.YoungModulus` represents a value defining the **Young's Modulus** (aka. elastic modulus). The dimension is standard SI unit [Pas].

4.1.3 PoissonRatio

`UniformIsotropicMaterial.PoissonRatio` represents a value defining the **Poisson's ratio**. Poisson's ratio is Dimensionless and has no SI unit.

4.1.4 Mass Density

`UniformIsotropicMaterial.Rho` represents a value defining the **Mass density**. The dimension is standard SI unit [kg/m³].

4.1.5 Damp Density

`UniformIsotropicMaterial.Mu` represents a value defining the **Damp density**. The dimension is standard SI unit [TODO].

4.1.6 static `CreateFromYoungPoisson()`

Creates a new instance of `UniformIsotropicMaterial` using Young's Modulus and Poisson's Ratio.

Example

Create steel material with:

- Young's Modulus = 210 [GPa]
- Poisson's Ratio = 0.3

```
var e = 210e9;//210 gpa
var nu = 0.3;

var steelMat = UniformIsotropicMaterial.CreateFromYoungPoisson(e, nu);
```

4.1.7 static `CreateFromYoungShear()`

Creates a new instance of `UniformIsotropicMaterial` using Young's Modulus and Shear Modulus. Poisson's ratio is calculated based on this formula: $G = E / (2 * (1 - \nu))$ then: $\nu = e / (2 * G) - 1$

Example

Create steel material with:

- Young's Modulus = 210 [GPa]
- Shear Modulus = 79 [GPa]

```
var e = 210e9;//210 gpa
var g = 79e9;//79 gpa

var steelMat = UniformIsotropicMaterial.CreateFromYoungShear(e, g);
```

4.1.8 static `CreateFromShearPoisson()`

Creates a new instance of `UniformIsotropicMaterial` using Shear Modulus and Poisson's Ratio. Elastic modulus is calculated based on this formula: $G = E / (2 * (1 - \nu))$ then: $E = G * (2 * (1 - \nu))$

Example

Create steel material with:

- Shear Modulus = 79 [GPa]
- Poisson's Ratio = 0.3

```
var g = 79e9; // 79 gpa
var nu = 0.3;

var steelMat = UniformIsotropicMaterial.CreateFromShearPoisson(g, nu);
```

4.2 UniformAnisotropicMaterial

4.2.1 Overview

inherited from `BaseMaterial`, this represents a **uniform** and **anisotropic** material:

- **uniform** means material properties are not varying through the element's body, or in every location of element material properties are identical.
- **anisotropic** means having different mechanical properties in different directions

4.2.2 Properties

There are 9 properties with this class:

- **Ex**: Young's Modulus in element's local X direction
- **Ey**: Young's Modulus in element's local Y direction
- **Ez**: Young's Modulus in element's local Z direction
- **NuXy, NuYx**: Poisson's Ratio
- **NuYz, NuZy**: Poisson's Ratio
- **NuZx, NuXz**: Poisson's Ratio

Material defines the mechanical properties of elements. Materials all are inherited from `BaseMaterial`.

5.1 Download source code of BriefFiniteElement.NET library

To download the source code there are three ways:

1. **Use git client**
2. **Direct download source code**

First way (using git client) is suggested as you have more control over source code and can keep source code sync with the latest source code on [github.com](https://www.github.com).

5.1.1 Using Git Client to Download the Source Code

We will use git for windows client over download the code. download and install client from [this link](https://gitforwindows.org/). After installation open the git-gui from either installed location or start menu:

Then click the “Clone Existing Repository” on the main window:

in next window, on the source location insert the git location of source code, this address is available at project main page on github.com:

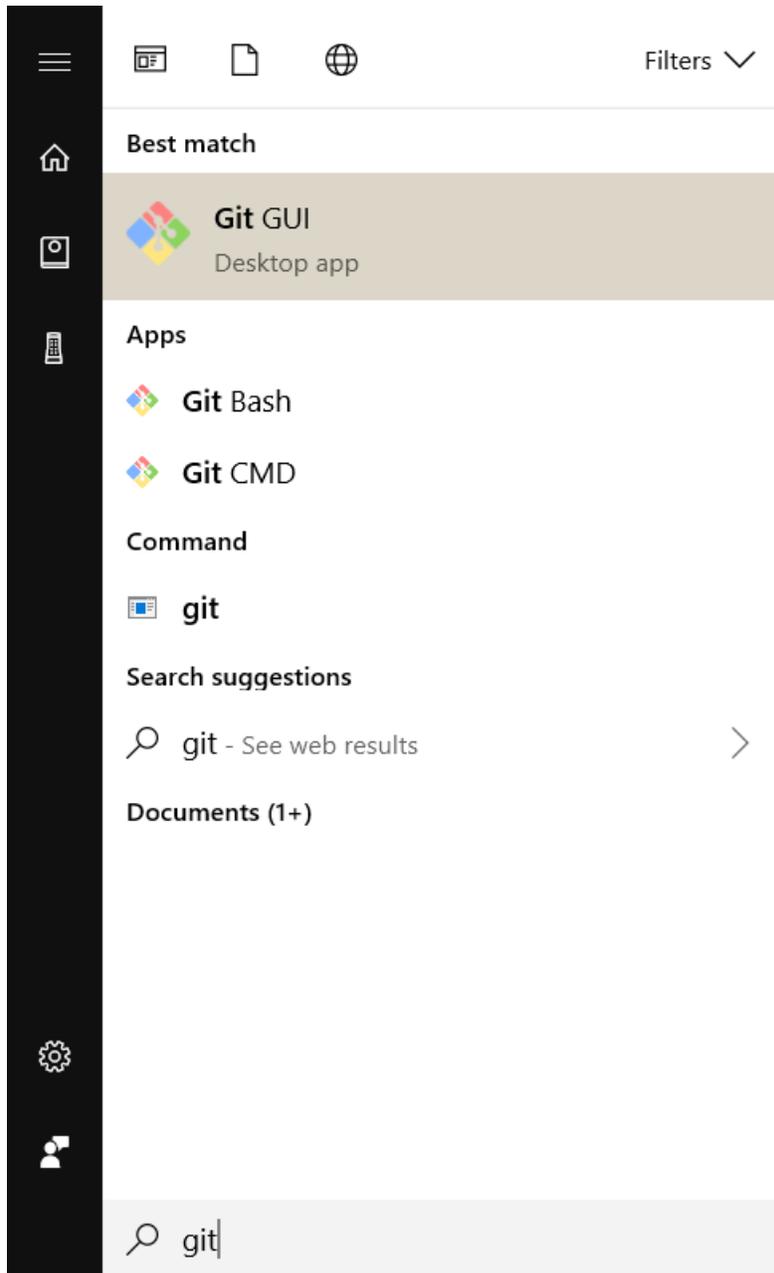
by the way currently it is: `https://github.com/BriefFiniteElementNet/BFE.Net.git` paste it into source location.

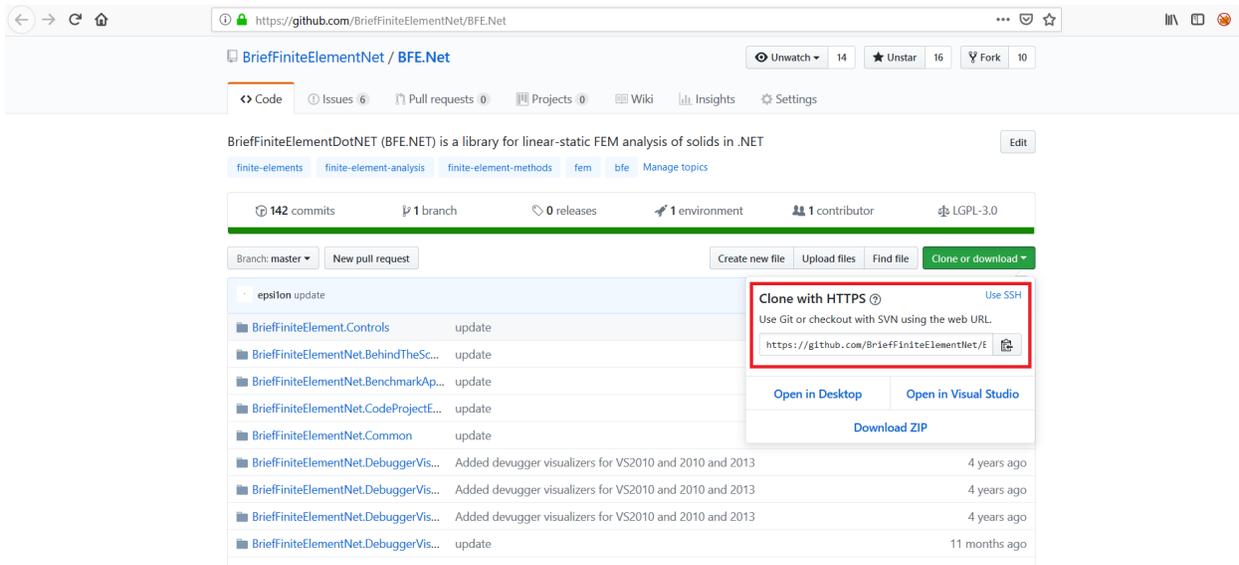
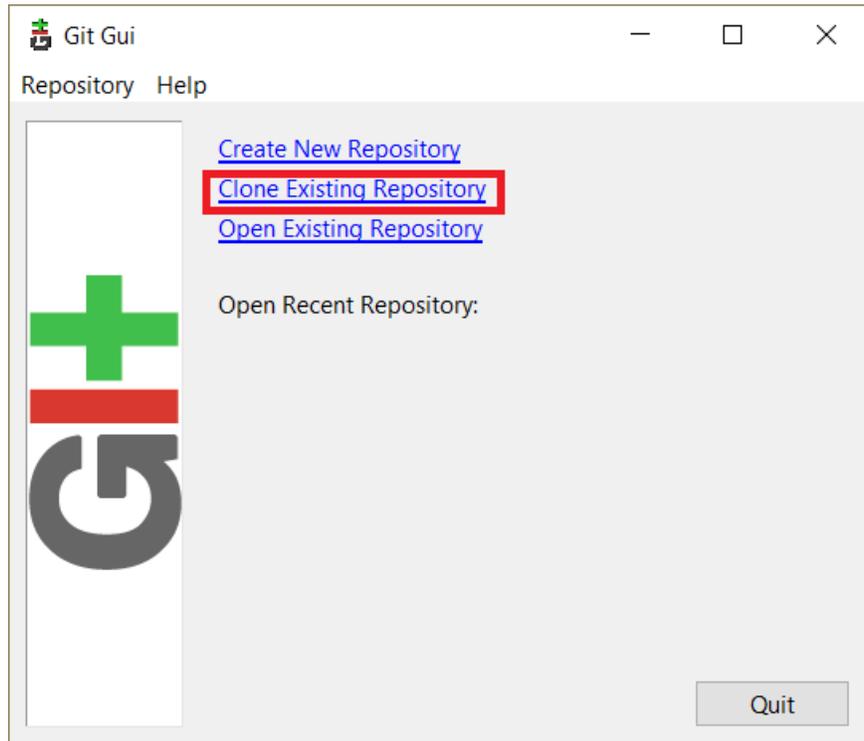
in destination location type the folder you want the source code be downloaded into, note that this folder will be created with git client and should not exists, and finally click clone:

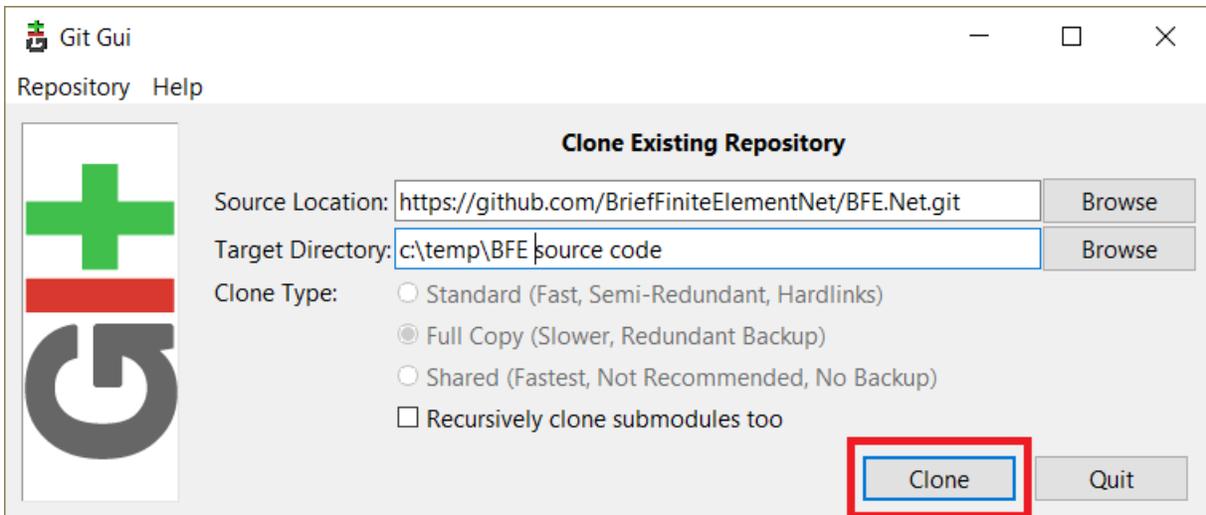
then wait until download finishes. Only note that use latest version of git client. after download finished, the git GUI will show up. close it and check the destination folder, there should be plenty of files there:

5.1.2 Direct download source code

From project main page in github.com click `clone` or `download` button then click `download zip`:







5.2 Create a project and compile BReiefFiniteElement from source code

After downloading the source code you should reference the main projects of bfe in your C# project. we will use Visual Studio 2015 in next. you can use any version of Visual Studio IDE including free express or community version. by the way it is downloadable from <https://visualstudio.microsoft.com/>. In next we will use visual Studio 2015...

Some users reported project does not build with Visual Studio 2017, there are some workarounds in ([issue #42 on project on github](https://github.com/BriefFiniteElementNet/BriefFiniteElement.Net/issues/42)<<https://github.com/BriefFiniteElementNet/BriefFiniteElement.Net/issues/42>>‘_).

5.2.1 Creating new project

After installation, Create a new C# application with name BfeTestApplication and type Console Application ([More Info](#)).

5.2.2 Add BFE codes (projects) into solution

You should add the actual BFE core codes into your solution. ([More Info](#)).

To add BFE code to your solution, do the following:

- 1- In Solution Explorer, select the solution.
- 2- On the File menu, point to Add, and click Existing Project.
- 3- In the Add Existing Project dialog box, locate the project you want to add, select the project file, and then click Open.

you should add two projects to your solution:

- 1- BriefFiniteElementNet located at <root folder>\BriefFiniteElementNet\BriefFiniteElementNet.csproj
- 2- BriefFiniteElementNet.Common located at <root folder>\BriefFiniteElementNet.Common\BriefFiniteElementNet.Common.csproj

adding two projects,

code

View

PC > Local Disk (C:) > temp > BFE source code >

Name	Date modified	Type	Size
.	11/9/2018 11:44 PM	File folder	
ament.Controls	11/9/2018 11:44 PM	File folder	
ament.Net	11/9/2018 11:44 PM	File folder	

[Home](#) | [Share](#) | [View](#)

[This PC](#) > [Local Disk \(C:\)](#) > [temp](#) > [BFE source code](#) >

Name	Date modified	Type	Size
.git	11/9/2018 11:44 PM	File folder	
BriefFiniteElement.Controls	11/9/2018 11:44 PM	File folder	
BriefFiniteElementNet	11/9/2018 11:44 PM	File folder	
BriefFiniteElementNet.BehindTheScene	11/9/2018 11:44 PM	File folder	
BriefFiniteElementNet.BenchmarkApplica...	11/9/2018 11:44 PM	File folder	
BriefFiniteElementNet.CodeProjectExam...	11/9/2018 11:44 PM	File folder	
BriefFiniteElementNet.Common	11/9/2018 11:44 PM	File folder	
BriefFiniteElementNet.DebuggerVisualize...	11/9/2018 11:44 PM	File folder	
BriefFiniteElementNet.DebuggerVisualize...	11/9/2018 11:44 PM	File folder	
BriefFiniteElementNet.DebuggerVisualize...	11/9/2018 11:44 PM	File folder	
BriefFiniteElementNet.DebuggerVisualize...	11/9/2018 11:44 PM	File folder	
BriefFiniteElementNet.TestConsole	11/9/2018 11:44 PM	File folder	
BriefFiniteElementNet.Validation	11/9/2018 11:44 PM	File folder	
BriefFiniteElementNet.Validation.Ui	11/9/2018 11:44 PM	File folder	
docs	11/9/2018 11:44 PM	File folder	
ExternallImages	11/9/2018 11:44 PM	File folder	
packages	11/9/2018 11:44 PM	File folder	
recyclebin	11/9/2018 11:44 PM	File folder	
SharedLibs	11/9/2018 11:44 PM	File folder	
.gitattributes	11/9/2018 11:44 PM	Text Document	3 KB
.gitignore	11/9/2018 11:44 PM	Text Document	3 KB
BriefFiniteElementNet.VS2010.sln	11/9/2018 11:44 PM	Microsoft Visual St...	4 KB
BriefFiniteElementNet.VS2012.sln	11/9/2018 11:44 PM	Microsoft Visual St...	4 KB
BriefFiniteElementNet.VS2013.sln	11/9/2018 11:44 PM	Microsoft Visual St...	6 KB
BriefFiniteElementNet.VS2015.sln	11/9/2018 11:44 PM	Microsoft Visual St...	6 KB

BriefFiniteElementNet / BFE.Net

142 commits 1 branch 0 releases 1 environment 1 contributor LGPL-3.0

Branch: master New pull request

Create new file Upload files Find file Clone or download

File/Folder	Last Update	Time Ago
epsilon	update	
BriefFiniteElement.Controls	update	
BriefFiniteElementNet.BehindTheSc...	update	
BriefFiniteElementNet.BenchmarkAp...	update	
BriefFiniteElementNet.CodeProjectE...	update	
BriefFiniteElementNet.Common	update	
BriefFiniteElementNet.DebuggerVis...	Added devugger visualizers for VS2010 and 2010 and 2013	4 years ago
BriefFiniteElementNet.DebuggerVis...	Added devugger visualizers for VS2010 and 2010 and 2013	4 years ago
BriefFiniteElementNet.DebuggerVis...	Added devugger visualizers for VS2010 and 2010 and 2013	4 years ago
BriefFiniteElementNet.DebuggerVis...	update	11 months ago
BriefFiniteElementNet.TestConsole	update	2 months ago
BriefFiniteElementNet.Validation.Ui	update	4 months ago
BriefFiniteElementNet.Validation	update	2 months ago

Clone with HTTPS Use SSH

Use Git or checkout with SVN using the web URL.

<https://github.com/BriefFiniteElementNet/BFE.Net>

Open in Desktop Open in Visual Studio

Download ZIP

New Project

.NET Framework 4 Sort by: Default

Search Installed Templates (Ctrl+E)

Type: Visual C#

A project for creating a command-line application

Blank App (Universal Windows 8.1) Visual C#

Windows Forms Application Visual C#

WPF Application Visual C#

Console Application Visual C#

Hub App (Universal Windows 8.1) Visual C#

Pivot App (Windows Phone) Visual C#

Shared Project Visual C#

Class Library (Portable for iOS, Android and Windows) Visual C#

Class Library Visual C#

Class Library (Portable) Visual C#

Name: BfeTestApplication

Location: c:\temp

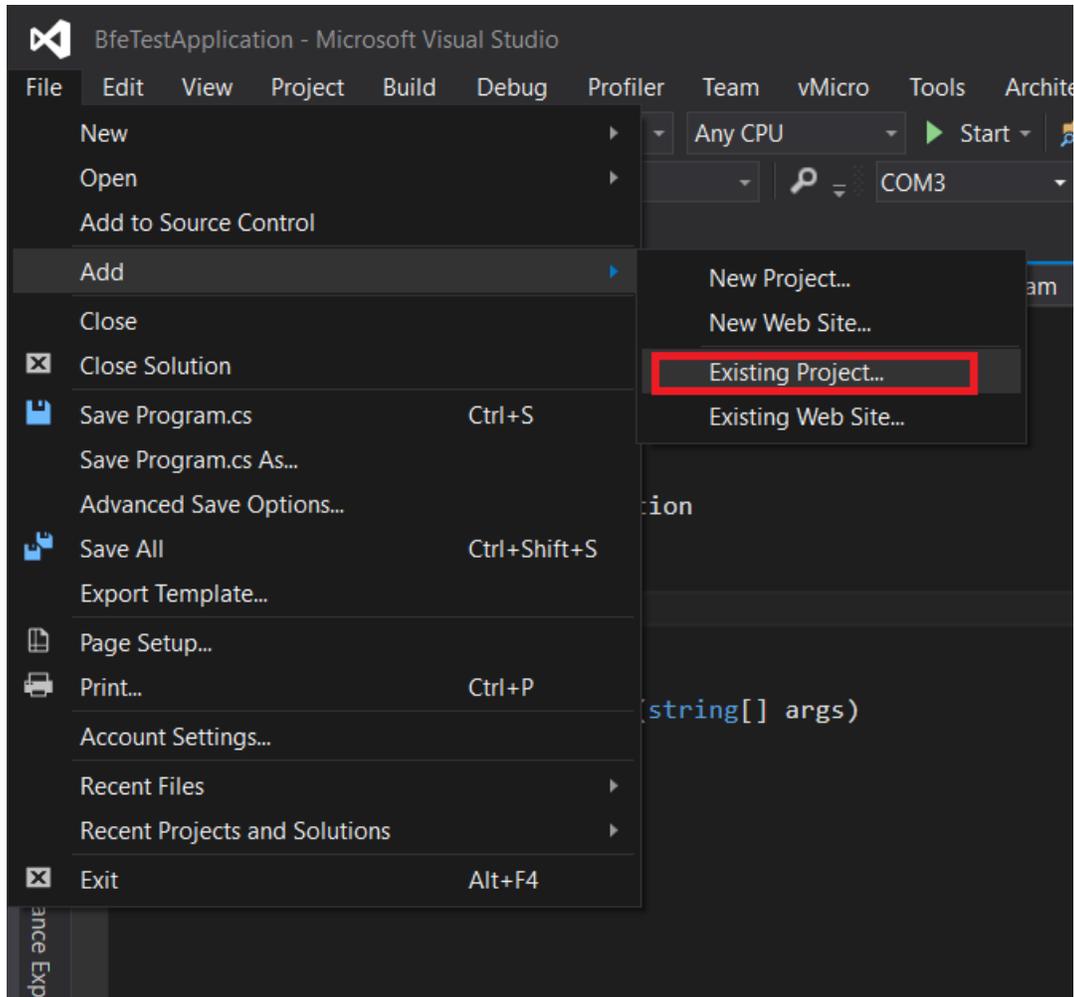
Solution: Create new solution

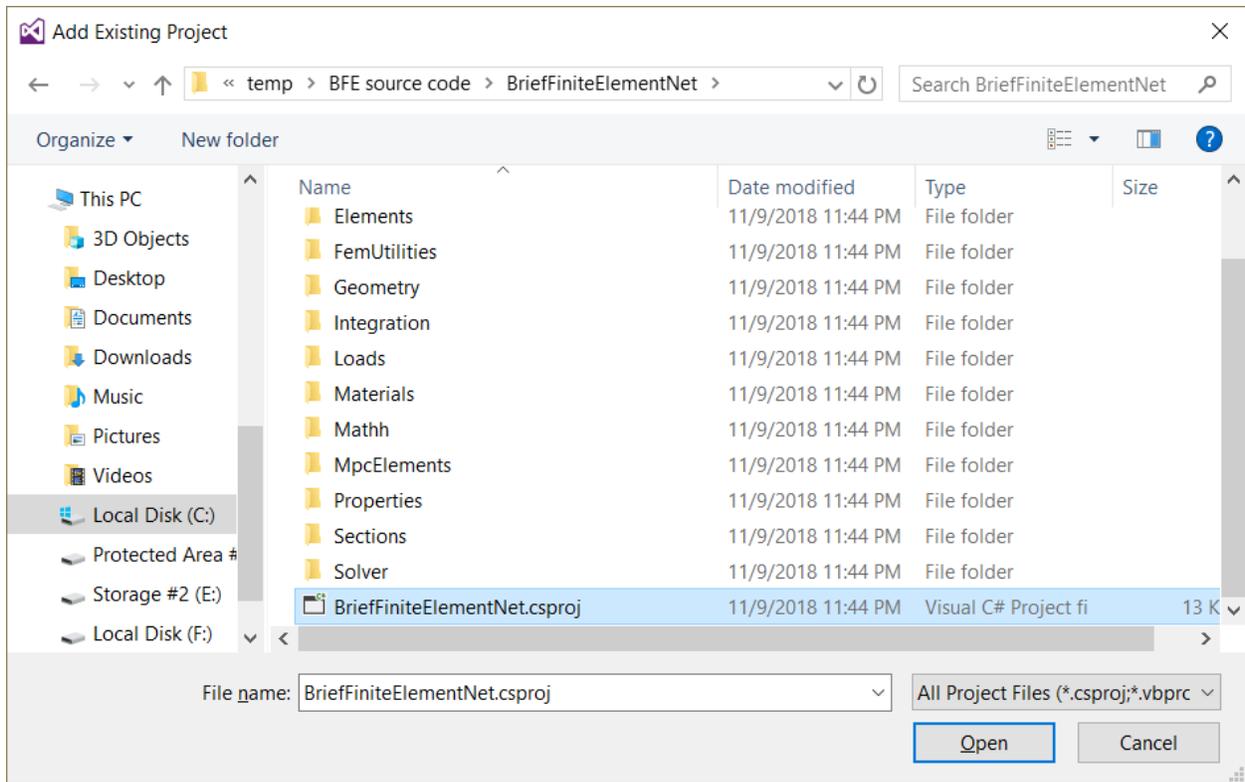
Solution name: BfeTestApplication

Create directory for solution

Create new Git repository

OK Cancel





after adding two projects, solution explorer tab in visual studio should look like this:

5.2.3 Add reference of BFE projects into first project

Now two C# projects `BriefFiniteElementNet` and `BriefFiniteElementNet.Common` are added to our solution. next we should add a reference of each one into first project named `BfeTestApplication` ([More Info](#)).

5.2.4 Start Coding with BFE

Now things are ready to start coding. Open the `Program.cs` file in project `BfeTestApplication` inside Visual Studio, it should be something like this:

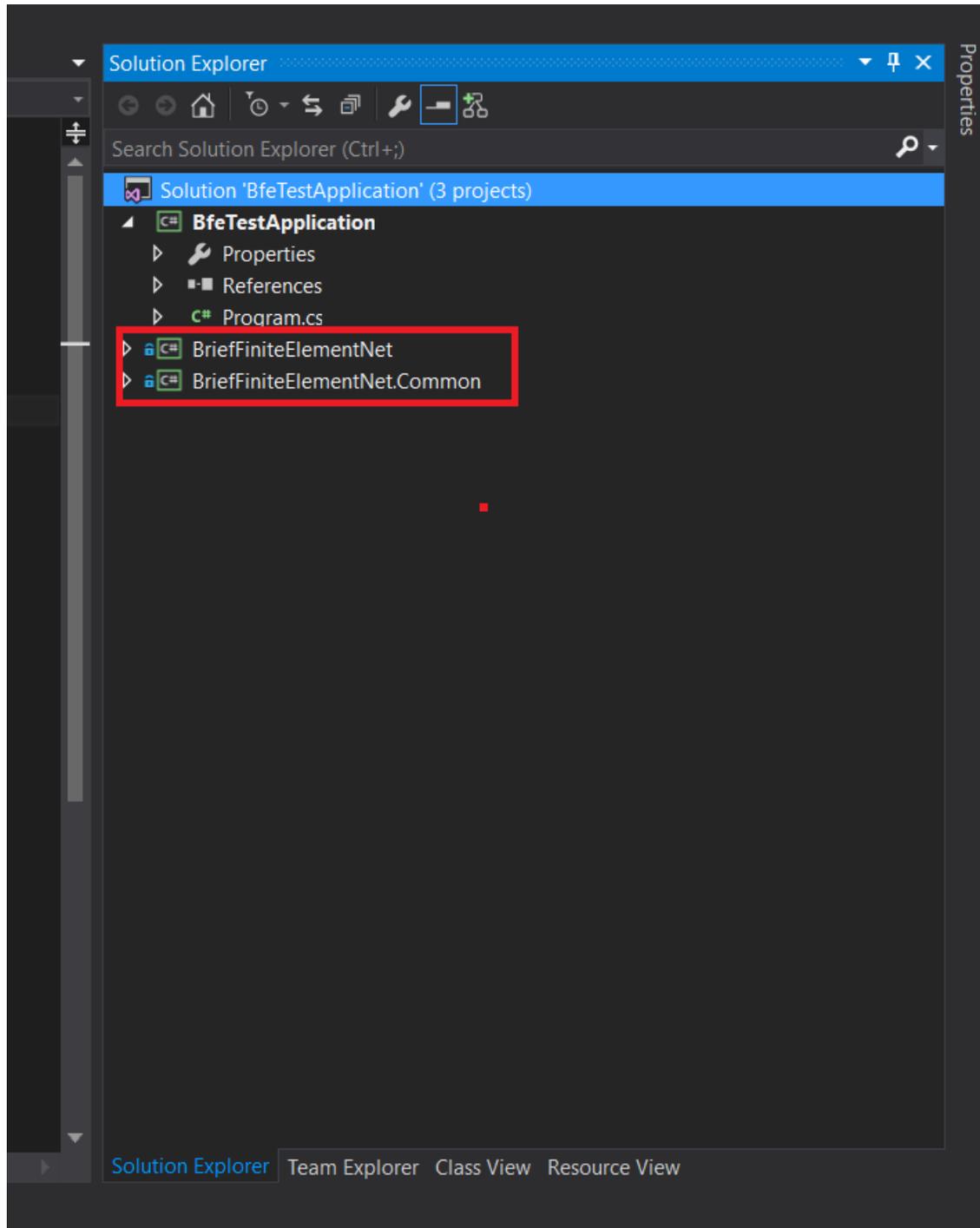
The static `void Main(string[] args)` method will execute once we start to execute the project. so we should add our code inside it:

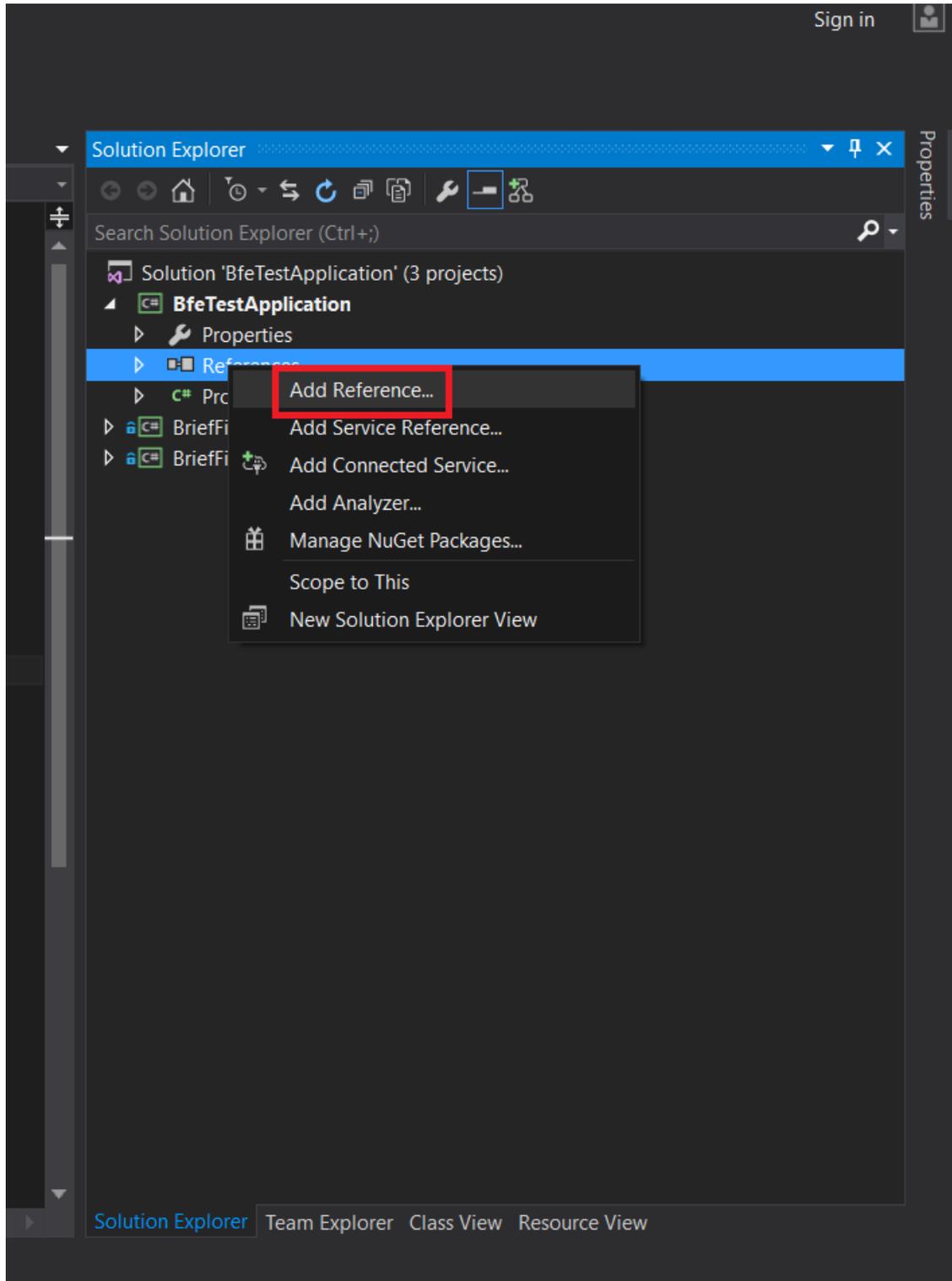
```
//a console beam, totally fixed in start n1, totally free in end n2
// a load of 1000 N
var model = new BriefFiniteElementNet.Model();

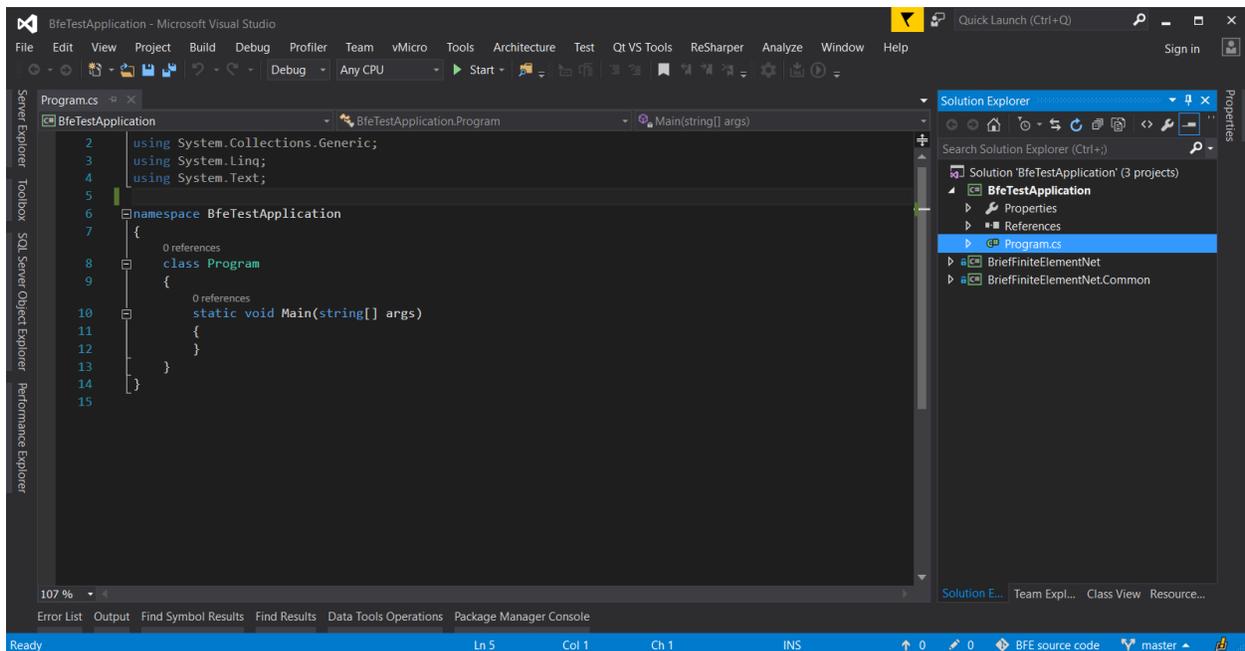
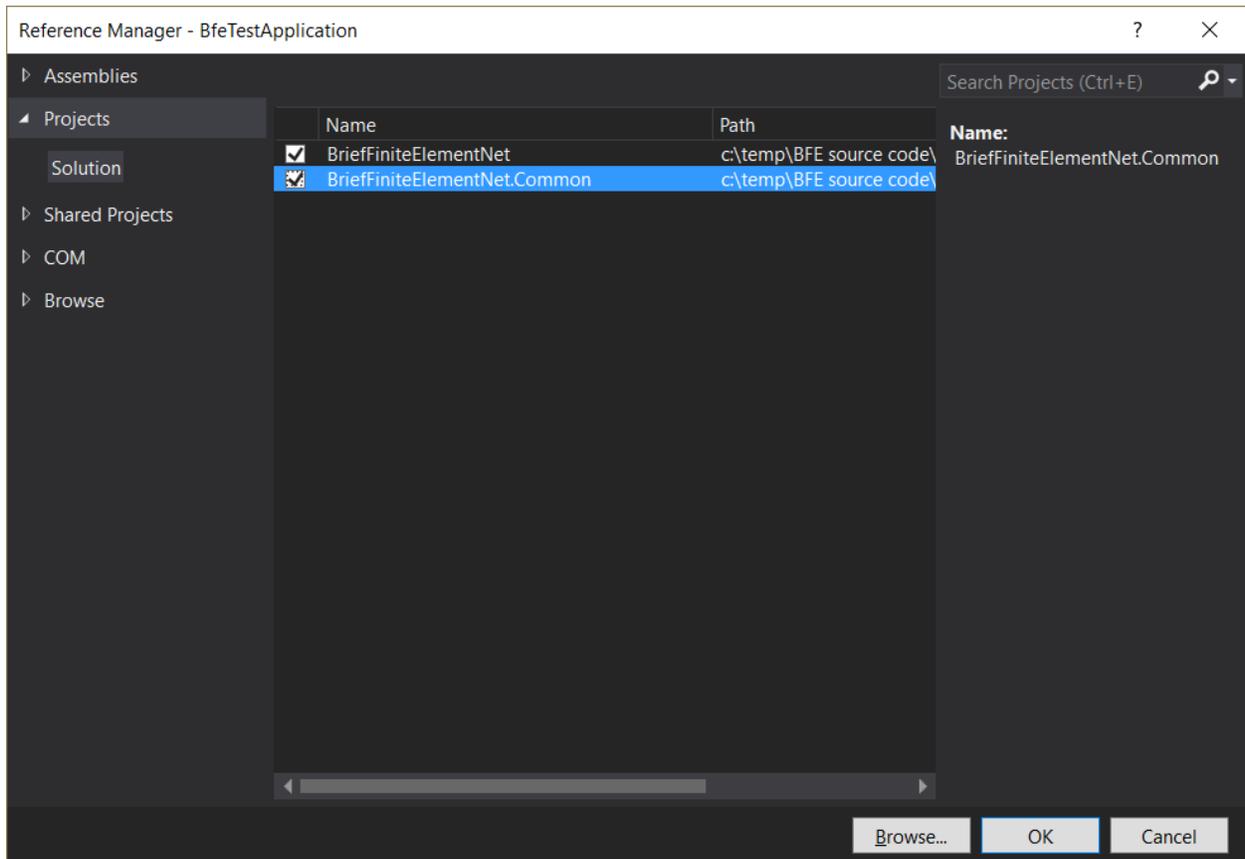
Node n1, n2;

model.Nodes.Add(n1 = new Node(x:0.0, y:0.0, z:0.0) { Constraints = Constraints.Fixed }
↪);
model.Nodes.Add(n2 = new Node(x:1.0, y:0.0, z:0.0) { Constraints = Constraints.
↪Released });
```

(continues on next page)







(continued from previous page)

```
var elm = new BarElement(n1, n2);

model.Elements.Add(elm);

elm.Section = new BriefFiniteElementNet.Sections.UniformParametric1DSection(a: 0.01,
↳iy: 8.3e-6, iz: 8.3e-6, j: 16.6e-6); //section's second area moments Iy and Iz = 8.
↳3*10^-6, area = 0.01
elm.Material = BriefFiniteElementNet.Materials.UniformIsotropicMaterial.
↳CreateFromYoungPoisson(210e9, 0.3); //Elastic mudule is 210e9 and poisson ratio is 0.
↳3

var load = new BriefFiniteElementNet.NodalLoad();
var frc = new Force();
frc.Fz = 1000; // 1kN force in Z direction
load.Force = frc;

n2.Loads.Add(load);

model.Solve_MPC(); //or model.Solve();

var d2 = n2.GetNodalDisplacement();

Console.WriteLine("Nodal displacement in Z direction is {0} meters (thus {1} mm)", d2.
↳DZ, d2.DZ * 1000); //print the Dz of n2 into console
Console.WriteLine("Nodal rotation in Y direction is {0} radians (thus {1} degrees)",
↳d2.RY, d2.RY * 180.0 / Math.PI); //print the Rz of n2 into console

Console.WriteLine("Press any key to continue");
Console.ReadKey();
```

also add two using directives on top of file:

```
using BriefFiniteElementNet;
using BriefFiniteElementNet.Elements;
```

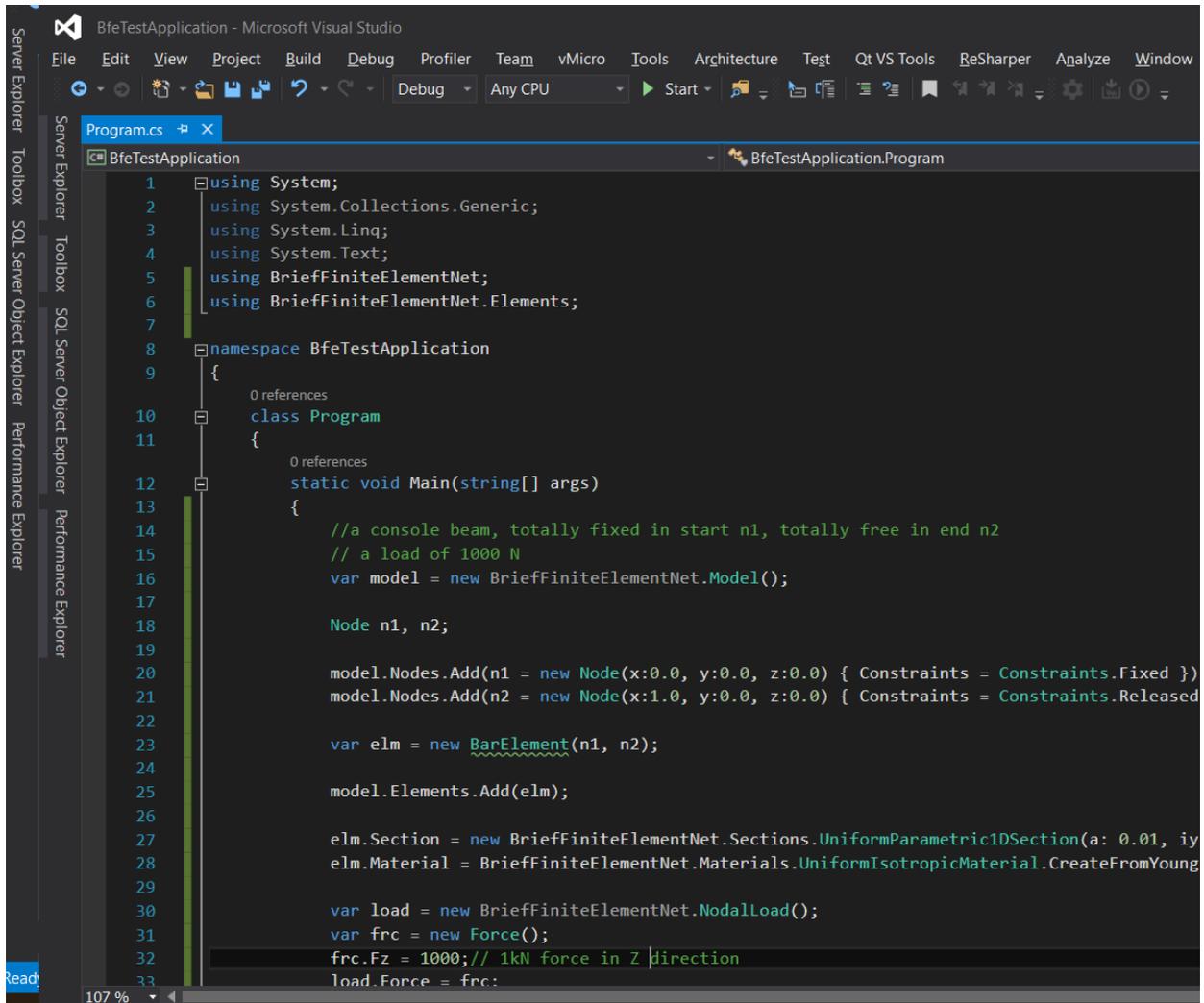
finally it should look like:

Then we start debug by pressign F5 key or “Debug” menu, then “Start Debugging”. console window should show up like this:

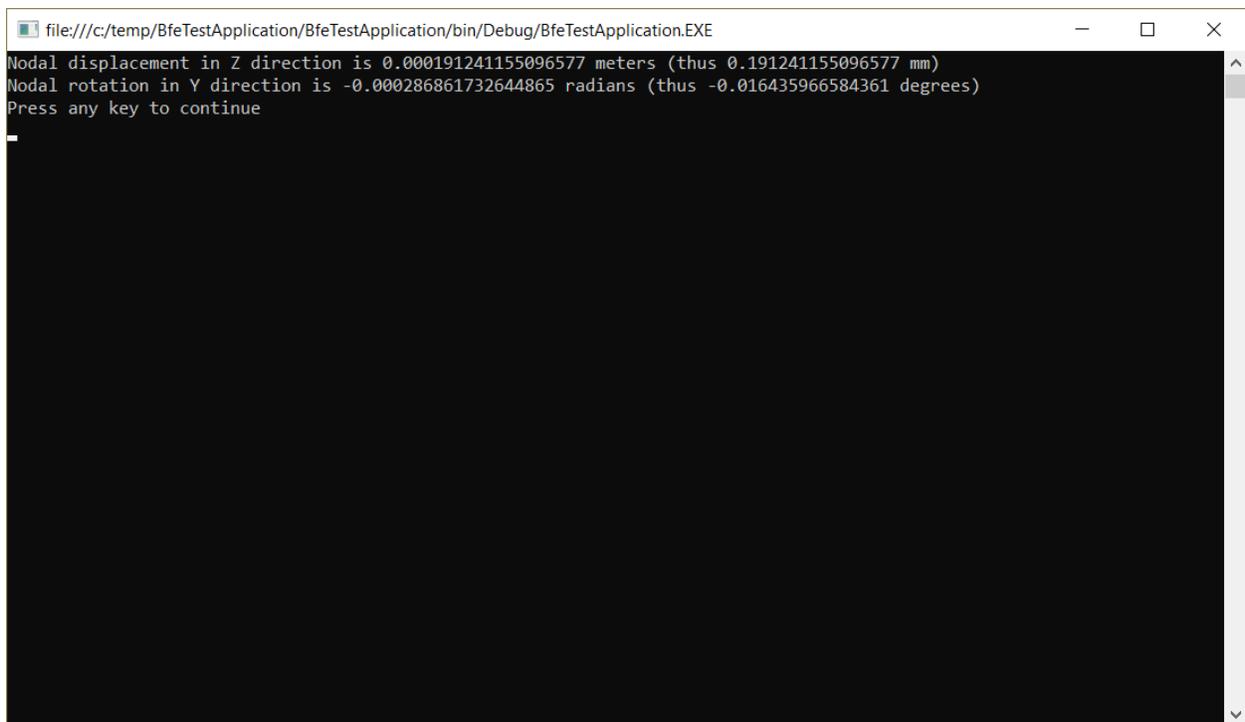
5.3 Install BFE.NET Nuget Library

Install-Package BriefFiniteElement.NET

This section is about getting started to use BFE.NET.



```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using BriefFiniteElementNet;
6 using BriefFiniteElementNet.Elements;
7
8 namespace BfeTestApplication
9 {
10     0 references
11     class Program
12     {
13         0 references
14         static void Main(string[] args)
15         {
16             //a console beam, totally fixed in start n1, totally free in end n2
17             // a load of 1000 N
18             var model = new BriefFiniteElementNet.Model();
19
20             Node n1, n2;
21
22             model.Nodes.Add(n1 = new Node(x:0.0, y:0.0, z:0.0) { Constraints = Constraints.Fixed })
23             model.Nodes.Add(n2 = new Node(x:1.0, y:0.0, z:0.0) { Constraints = Constraints.Released
24
25             var elm = new BarElement(n1, n2);
26
27             model.Elements.Add(elm);
28
29             elm.Section = new BriefFiniteElementNet.Sections.UniformParametric1DSection(a: 0.01, iy
30             elm.Material = BriefFiniteElementNet.Materials.UniformIsotropicMaterial.CreateFromYoung
31
32             var load = new BriefFiniteElementNet.NodalLoad();
33             var frc = new Force();
34             frc.Fz = 1000; // 1kN force in Z direction
35             load.Force = frc;
```



```
file:///c:/temp/BfeTestApplication/BfeTestApplication/bin/Debug/BfeTestApplication.EXE
Nodal displacement in Z direction is 0.000191241155096577 meters (thus 0.191241155096577 mm)
Nodal rotation in Y direction is -0.000286861732644865 radians (thus -0.016435966584361 degrees)
Press any key to continue
```

6.1 Small 3D Truss Example

In this example, I want to analyse a simple truss with 4 members as shown in figure.

All members sections are the same, a square steel section with dimension of 3 cm. So the properties of members will be:

The area of SECTION:

$$A = 0.03\text{m} * 0.03\text{m} = 9 * 10^{-4} \text{ m}^2$$

The Elastic or Young Modulus of MATERIAL:

$$E = 210 \text{ GPa} = 210 * 10^9 \text{ Pa} = 210 * 10^9 \text{ N/M}^2$$

The Poison Ratio of MATERIAL:

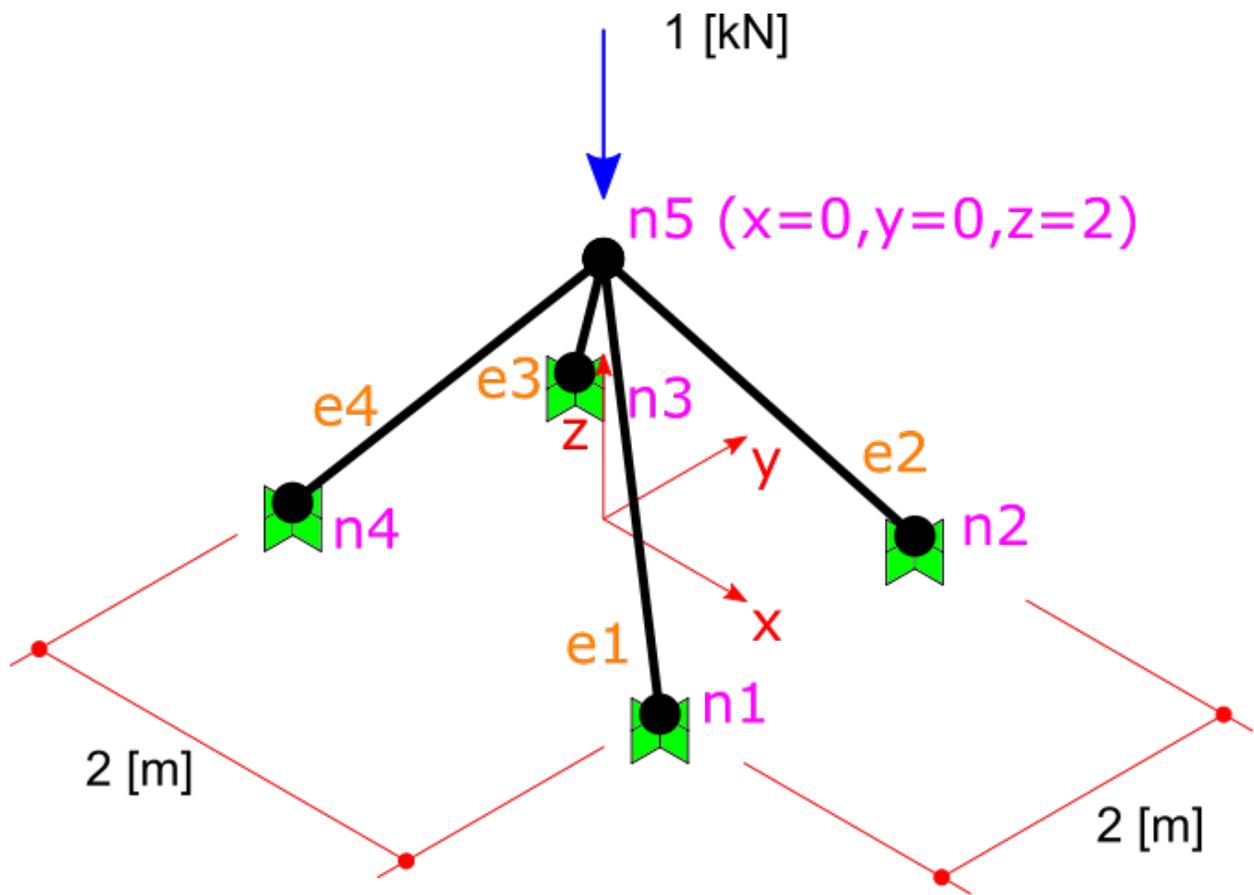
$$\nu = 0.3$$

Please note that for truss member usually Poison ratio is not taken into account anywhere in calculation, so settings it to any value in (0, 0.5) range will not change any part of the result, but setting to zero maybe cause some problem! so better to assume a usual value of 0.3 for it.

We should do these steps before we solve the model:

- Step1: Create Model, Members and Nodes.
- Step2: Add the Nodes and Elements to Model.
- Step3: Assign geometrical and mechanical properties to Elements.
- Step4: Assign Constraints to Nodes (fix the DoF s).
- Step5: Assign Load to Node.

And finally solve model with Model.Solve() method and then extract analysis results like support reactions or member internal forces or nodal deflections.



6.1.1 Step1: Create Model, Members and Nodes

We should create a Finite Element model first and then add members and nodes to it:

```
// Initiating Model, Nodes and Members
var model = new Model();
```

Creating Nodes

We should create nodes like this. In BriefFiniteElement.NET, every node and element have a property of type string named Label and another one named Tag both of which are inherited from BriefFiniteElementNet.StructurePart. In every Model, Label of every member should be unique among all members (both Nodes and Elements) unless the Label be equal to null which is by default. In the below code, we are creating 5 nodes of truss and assigning a unique Label to each one.

```
var n1 = new Node(1, 1, 0);
n1.Label = "n1";//Set a unique label for node
var n2 = new Node(-1, 1, 0) {Label = "n2"};//using object initializer for assigning_
↪Label
var n3 = new Node(1, -1, 0) {Label = "n3"};
var n4 = new Node(-1, -1, 0) {Label = "n4"};
var n5 = new Node(0, 0, 1) {Label = "n5"};
```

Creating Elements

Next we have to create the elements. In BriefFiniteElement.NET, the TrussElement and BarElement classes do represents a truss element in 3D. As TrussElement is old and obsolete, we use BarElement:

```
var e1 = new BarElement(n1, n5) { Label = "e1", Behavior = BarElementBehaviours.Truss_
↪};
var e2 = new BarElement(n2, n5) {Label = "e2", Behavior = BarElementBehaviours.Truss }
↪;
var e3 = new BarElement(n3, n5) {Label = "e3", Behavior = BarElementBehaviours.Truss }
↪;
var e4 = new BarElement(n4, n5) { Label = "e4", Behavior = BarElementBehaviours.Truss_
↪};
```

note that BarElement can be used as a frame too, so you should set the BarElement.Behavior to BarElementBehaviours.Truss in order to make it a truss member, else you will have a frame member instead of truss!

6.1.2 Step2: Add the Nodes and Elements to Model.

You can simply add the elements and nodes we created into the Model. Model has two members Model.Elements and Model.Nodes which both represents an IList<T> of nodes and members, plus an Add() method that accept several items:

```
model.Nodes.Add(n1, n2, n3, n4, n5);
model.Elements.Add(e1, e2, e3, e4);
```

Please note that if Node or Element's Label property is something else than null, then it should be unique among all nodes and elements, else you will receive an error when adding member with duplicated label into model.

6.1.3 Step3: Assign geometrical and mechanical properties to Elements

As elastic module for all members equals to 210 GPa and area of all members equals to 0.0009 m² we can set the element properties like this:

```
e1.Section = new Sections.UniformParametric1DSection() { A = 9e-4 };
e2.Section = new Sections.UniformParametric1DSection() { A = 9e-4 };
e3.Section = new Sections.UniformParametric1DSection() { A = 9e-4 };
e4.Section = new Sections.UniformParametric1DSection() { A = 9e-4 };

e1.Material = Materials.UniformIsotropicMaterial.CreateFromYoungPoisson(210e9, 0.3);
e2.Material = Materials.UniformIsotropicMaterial.CreateFromYoungPoisson(210e9, 0.3);
e3.Material = Materials.UniformIsotropicMaterial.CreateFromYoungPoisson(210e9, 0.3);
e4.Material = Materials.UniformIsotropicMaterial.CreateFromYoungPoisson(210e9, 0.3);
```

6.1.4 Step4: Assign Constraints to Nodes (fix the DoF s)

Now, we should make some DoFs of structure fix in order to make analysis logically possible.

In BriefFiniteElement.NET, every node has 6 degree of freedom: X, Y, and Z rotations and X, Y, and Z translations. For a every truss model, we have to fix rotational DoFs for each Node (X,Y and Z rotation). Also the nodes 1 to 4 are also movement fixed, then nodes 1 to 4 should be totally fixed and node 5 should be rotation fixed. In BriefFiniteElement.NET, a struct named Constraint represents a constraint that is applicable to a 6 DoF node, it have Dx, Dy, Dz, Rx, Ry and Rz properties of type DofConstraint which is an enum and have two possible values 0 (Released) and 1 (Fixed). For making work easier, the Constraint struct has some predefined Constraints in its static properties for example Constraint.Fixed or Constraint.Free. Here is more detailed information:

Property Name	Description
Constraints.Fixed	All 6 DoFs are fixed
Constraints.Released	All 6 DoFs are released
Constraints.MovementFixed	3 translation DoFs are fixed and 3 rotation DoFs are released
Constraints.RotationFixed	3 translation DoFs are released and 3 rotation DoFs are fixed

We can fix DoFs of nodes 1 to 4 like this:

```
n1.Constraints = n2.Constraints = n3.Constraints = n4.Constraints = new
↳ Constraint(dx:DofConstraint.Fixed, dy:DofConstraint.Fixed, dz:DofConstraint.Fixed,
↳ rx:DofConstraint.Fixed, ry:DofConstraint.Fixed, rz:DofConstraint.Fixed);
```

or:

```
n1.Constraints = n2.Constraints = n3.Constraints = n4.Constraints = Constraints.Fixed
```

and should fix the rotational DoFs of node 5:

6.1.5 Step5: Assign Load to Node

In BriefFiniteElement.NET, there is a struct named Force which represent a concentrated force in 3D space which contains of 3 force components in X, Y and Z directions and three moment components in X, Y and Z directions. It have 6 double properties named Fx, Fy, Fz, Mx, My and Mz that are representing the load components. There are also two properties of type Vector for this struct named Forces and Moments. On setting or getting, they will use the Fx, Fy, Fz, Mx, My and Mz to perform operations:

```

/// <summary>
/// Gets or sets the forces.
/// </summary>
/// <value>
/// The forces as a <see cref="Vector"/>.
/// </value>
public Vector Forces
{
    get
    {
        return new Vector(fx, fy, fz);
    }

    set
    {
        this.fx = value.X;
        this.fy = value.Y;
        this.fz = value.Z;
    }
}

```

Same is with Moments property. The Forces and Moments property do not actually store values in something other than 6 obvious properties.

As LoadCase and LoadCombination concepts are supported in BriefFiniteElement.NET, every Load should have a LoadCase. A LoadCase is simply a struct that has two properties: CaseName with string type and LoadType with LoadType type which is an enum and has some possible values:

```

public enum LoadType
{
    Default = 0,
    Dead,
    Live,
    Snow,
    Wind,
    Quake,
    Crane,
    Other
}

```

The LoadType.Default is a load type that is created for built in usage in library and it do not meant to have meaning like Dead, Live, etc. The LoadCase struct has a static property named LoadCase.DefaultLoadCase:

```

/// <summary>
/// Gets the default load case.
/// </summary>
/// <value>
/// The default load case.
/// </value>
/// <remarks>
/// Gets a LoadCase with <see cref="LoadType"/> of <see cref="BriefFiniteElementNet.
↳LoadType.Default"/> and empty <see cref="CaseName"/></remarks>
public static LoadCase DefaultLoadCase
{
    get { return new LoadCase(); }
}

```

Which represents a LoadCase with LoadType of Default and CaseName of null. We will call such a LoadCase as

DefaultLoadCase. For simplicity of usage in BriefFiniteElement.NET everywhere that you'll prompt for a LoadCase, if you do not provide a LoadCase then the LoadCase is assumed DefaultLoadCase by the library. For example, when you want to assign a load to a node, you should provide a LoadCase for it, like this:

```
var load = new NodalLoad(new Force(0, 0, -1000, 0, 0, 0), new LoadCase("Case1",  
↪LoadType.Dead));
```

but if you do not provide the LoadCase in the above code like this:

```
var load = new NodalLoad(new Force(0, 0, -1000, 0, 0, 0));
```

then the load case will be assumed DefaultLoadCase by the library.

Ok, next we have to add 1KN load to node 5 like this, will do it with DefaultLoadCase:

```
var force = new Force(0, 0, -1000, 0, 0, 0);  
n5.Loads.Add(new NodalLoad(force)); //adds a load with LoadCase of DefaultLoadCase to  
↪node loads
```

And finally solve the model with model.Solve() method. Actually solving the model is done in two stages:

- First stage is creating stiffness matrix and factorizing stiffness matrix which will take majority of time for analysing
- Second phase is analysing structure against each load case which takes much less time against first stage (say for example 13 sec for first stage and 0.5 sec for second stage).

First stage is done in model.Solve() method and second stage will done if they'll be need to.

There are loads with different LoadCases that are applied to the Nodes and Elements. So the Node.GetSupportReaction() method have an overload which gets a LoadCombination and returns the support reactions based on the load combination. LoadCombination has a static property named LoadCombination.DefaultLoadCombination which has only one LoadCase in it (the DefaultLoadCase) with factor of 1.0. also everywhere that you should provide a LoadCombination, if you do not provide any, then DefaultLoadCombination will be considered by library. I've used DefaultLoadCase and DefaultLoadCombination in library to make working with library easier for people who are not familiar with load case and load combination stuff.

For getting the support reaction for the truss, we can simply call Node.GetSupportReaction() to get support reaction for every node:

```
Force r1 = n1.GetSupportReaction();  
Force r2 = n2.GetSupportReaction();  
Force r3 = n3.GetSupportReaction();  
Force r4 = n4.GetSupportReaction();
```

The plus operator is overloaded for Force struct, so we can check the sum of support reactions:

```
Force rt = r1 + r2 + r3 + r4; //shows the Fz=1000 and Fx=Fy=Mx=My=Mz=0.0
```

The forces (Fx, Fy and Fz) amount should be equal to sum of external loads and direction should be opposite to external loads to satisfy the structure static equilibrium equations.

6.1.6 All Codes Together

This is all codes above for truss example.

Please note that these codes are available in BriefFiniteElementNet.CodeProjectExamples project in library solution.

```

private static void Example1()
{
    Console.WriteLine("Example 1: Simple 3D truss with four members");

    // Initiating Model, Nodes and Members
    var model = new Model();

    var n1 = new Node(1, 1, 0);
    n1.Label = "n1";//Set a unique label for node
    var n2 = new Node(-1, 1, 0) {Label = "n2"};//using object initializer for_
↪assigning Label
    var n3 = new Node(1, -1, 0) {Label = "n3"};
    var n4 = new Node(-1, -1, 0) {Label = "n4"};
    var n5 = new Node(0, 0, 1) {Label = "n5"};

    var e1 = new TrussElement2Node(n1, n5) {Label = "e1"};
    var e2 = new TrussElement2Node(n2, n5) {Label = "e2"};
    var e3 = new TrussElement2Node(n3, n5) {Label = "e3"};
    var e4 = new TrussElement2Node(n4, n5) {Label = "e4"};
    //Note: labels for all members should be unique, else you will receive_
↪InvalidLabelException when adding it to model

    e1.A = e2.A = e3.A = e4.A = 9e-4;
    e1.E = e2.E = e3.E = e4.E = 210e9;

    model.Nodes.Add(n1, n2, n3, n4, n5);
    model.Elements.Add(e1, e2, e3, e4);

    //Applying restrains

    n1.Constraints = n2.Constraints = n3.Constraints = n4.Constraints = _
↪Constraint.Fixed;
    n5.Constraints = Constraint.RotationFixed;

    //Applying load
    var force = new Force(0, 1000, -1000, 0, 0, 0);
    n5.Loads.Add(new NodalLoad(force));//adds a load with LoadCase of_
↪DefaultLoadCase to node loads

    //Adds a NodalLoad with Default LoadCase

    model.Solve();

    var r1 = n1.GetSupportReaction();
    var r2 = n2.GetSupportReaction();
    var r3 = n3.GetSupportReaction();
    var r4 = n4.GetSupportReaction();

    var rt = r1 + r2 + r3 + r4;//shows the Fz=1000 and Fx=Fy=Mx=My=Mz=0.0

    Console.WriteLine("Total reactions SUM :" + rt.ToString());
}

```

console result after executing:

Console Output

Example 1: Simple 3D truss with four members

Total reactions SUM :F: 0, 0, 1000, M: 0, 0, 0

6.2 LoadCase and LoadCombination Example

In Finite Element, there is a thing named Force or Load. Also there are

There are two concepts named LoadCase and LoadCombination in this library and many other softwares. A `LoadCase` defines the group of loads. For example, in structure below there is a “dead” load and a “live” load, and two “earthquake” loads, in X and Y direction on *n4* node:

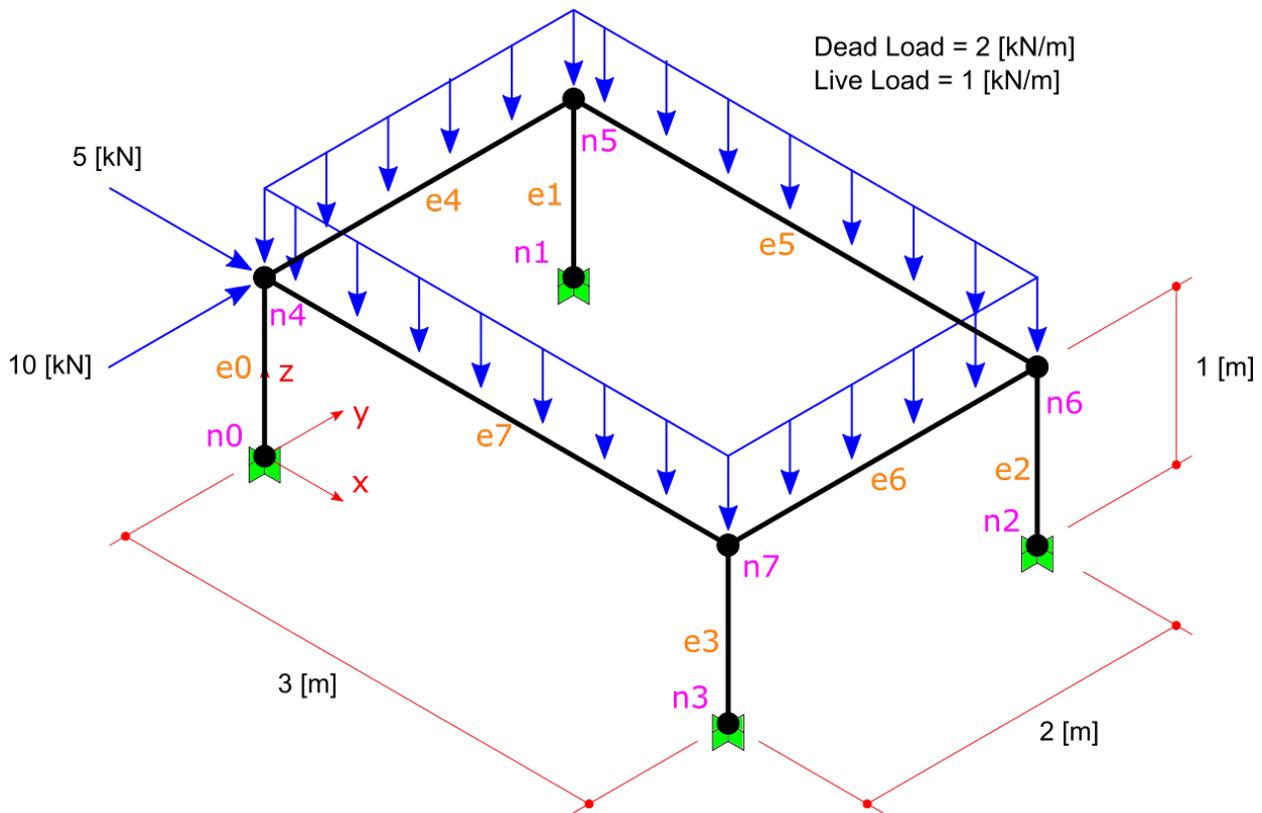


Fig. 1: Model with 4 type of load

The LoadCase struct have a nature property (an enum type) and a title property (with string type). LoadNature can be: Default, Dead, Live, Snow, Wind, Quake, Crane and Other.

So there can be 4 LoadCases for this example:

- case 1: Nature = Dead, Title = “D1”
- case 2: Nature = Live, Title = “L1”
- case 3: Nature = Quake, Title = “Qx”
- case 4: Nature = Quake, Title = “Qy”

We will do these steps before solving model:

- Step1: Create Model, prepare and add Elements and Nodes
- Step2: Assign Constraints to Nodes (fix the DoF s).
- Step3: Assign Load to Node.

6.2.1 Step1: Create Model, prepare and add Elements and Nodes

To make the model, elements and loads:

```
//source code file: LoadCombExample.cs, project: BriefFiniteElementNet.
↪CodeProjectExamples

var model = new Model();

model.Nodes.Add(new Node(0, 0, 0) { Label = "n0" });
model.Nodes.Add(new Node(0, 2, 0) { Label = "n1" });
model.Nodes.Add(new Node(4, 2, 0) { Label = "n2" });
model.Nodes.Add(new Node(4, 0, 0) { Label = "n3" });

model.Nodes.Add(new Node(0, 0, 1) { Label = "n4" });
model.Nodes.Add(new Node(0, 2, 1) { Label = "n5" });
model.Nodes.Add(new Node(4, 2, 1) { Label = "n6" });
model.Nodes.Add(new Node(4, 0, 1) { Label = "n7" });

var a = 0.1 * 0.1;//area, assume sections are 10cm*10cm rectangular
var iy = 0.1 * 0.1 * 0.1 * 0.1 / 12.0;//Iy
var iz = 0.1 * 0.1 * 0.1 * 0.1 / 12.0;//Iz
var j = 0.1 * 0.1 * 0.1 * 0.1 / 12.0;//Polar
var e = 20e9;//young modulus, 20 [GPa]
var nu = 0.2;//poissons ratio

var sec = new Sections.UniformParametric1DSection(a, iy, iz, j);
var mat = Materials.UniformIsotropicMaterial.CreateFromYoungPoisson(e, nu);

model.Elements.Add(new BarElement(model.Nodes["n0"], model.Nodes["n4"]) { Label = "e0
↪", Section = sec, Material = mat});
model.Elements.Add(new BarElement(model.Nodes["n1"], model.Nodes["n5"]) { Label = "e1
↪", Section = sec, Material = mat });
model.Elements.Add(new BarElement(model.Nodes["n2"], model.Nodes["n6"]) { Label = "e2
↪", Section = sec, Material = mat });
model.Elements.Add(new BarElement(model.Nodes["n3"], model.Nodes["n7"]) { Label = "e3
↪", Section = sec, Material = mat });

model.Elements.Add(new BarElement(model.Nodes["n4"], model.Nodes["n5"]) { Label = "e4
↪", Section = sec, Material = mat });
model.Elements.Add(new BarElement(model.Nodes["n5"], model.Nodes["n6"]) { Label = "e5
↪", Section = sec, Material = mat });
model.Elements.Add(new BarElement(model.Nodes["n6"], model.Nodes["n7"]) { Label = "e6
↪", Section = sec, Material = mat });
model.Elements.Add(new BarElement(model.Nodes["n7"], model.Nodes["n4"]) { Label = "e7
↪", Section = sec, Material = mat });
```

6.2.2 Step2: Assign Constraints to Nodes (fix the DoF s)

```
model.Nodes["n0"].Constraints =
    model.Nodes["n1"].Constraints =
        model.Nodes["n2"].Constraints =
            model.Nodes["n3"].Constraints =
                Constraints.Fixed;
```

6.2.3 Step3: Assign load to nodes

This is main purpose of this example, the LoadCase and LoadCombination types. In framework every Load does have a property named LoadCase. this LoadCase property will help us to distribute all Loads into groups. We want to do this because we should solve the model for each LoadCase separately. In this example we will create 4 load cases:

1. a load case with name d1 and load type of dead for dead loads on top horizontal elements
2. a load case with name l1 and load type of live for live loads on top horizontal elements
3. a load case with name qx and load type of quake for 5kN concentrated force applied to n4 node
4. a load case with name qy and load type of quake for 10kN concentrated force applied to n4 node

```
var d_case = new LoadCase("d1", LoadType.Dead);
var l_case = new LoadCase("l1", LoadType.Dead);
var qx_case = new LoadCase("qx", LoadType.Quake);
var qy_case = new LoadCase("qy", LoadType.Quake);
```

Then we should create two distributed loads for top beams:

```
var d1 = new Loads.UniformLoad(d_case, -1 * Vector.K, 2e3, CoordinationSystem.Global);
var l1 = new Loads.UniformLoad(l_case, -1 * Vector.K, 1e3, CoordinationSystem.Global);

var qx_f = new Force(5000 * Vector.I, Vector.Zero);
var qy_f = new Force(10000 * Vector.J, Vector.Zero);
```

note that we've set the load case of these two loads by passing d_case and l_case into constructor of Loads.UniformLoad class.

Next we will add d1 and l1 and two other nodal lo loads to all top elements. you should note that adding same load to more that one element is possible and will work like creating identical loads for each element.

```
model.Elements["e4"].Loads.Add(d1);
model.Elements["e5"].Loads.Add(d1);
model.Elements["e6"].Loads.Add(d1);
model.Elements["e7"].Loads.Add(d1);

model.Elements["e4"].Loads.Add(l1);
model.Elements["e5"].Loads.Add(l1);
model.Elements["e6"].Loads.Add(l1);
model.Elements["e7"].Loads.Add(l1);

model.Nodes["n4"].Loads.Add(new NodalLoad(qx_f, qx_case));
model.Nodes["n4"].Loads.Add(new NodalLoad(qy_f, qy_case));

model.Solve_MPC();//no different with Model.Solve()
```

as said before, all loads in BFE should inherit from NodalLoad or ElementLoad. Both of these loads have a property named LoadCase property of type `LoadCase`. So every load in BFE will have the LoadCase property. In other

hand to get analysis result of model - like internal force on elements, or nodal displacements or support reactions - a parameter of type `LoadCombination` should pass to the appropriated method. For example to get internal force of bar element, this method should be called:

```
BarElement.GetInternalForceAt(double x, LoadCombination combination);
```

Or to get support reaction of a node, this method should be used:

```
Node.GetSupportReaction(LoadCombination combination);
```

A `LoadCombination` in a list of `LoadCases` with a multiplier for each one. Internally it does uses `Dictionary<LoadCase, double>` to keep the list. For example if want to find support reaction for node `n3` with loadCombination `D + 0.8 L`:

```
var combination1 = new LoadCombination();// for D + 0.8 L
combination1[d_case] = 1.0;
combination1[l_case] = 0.8;

var n3Force = model.Nodes["N3"].GetSupportReaction(combination1);
Console.WriteLine(n3Force);
```

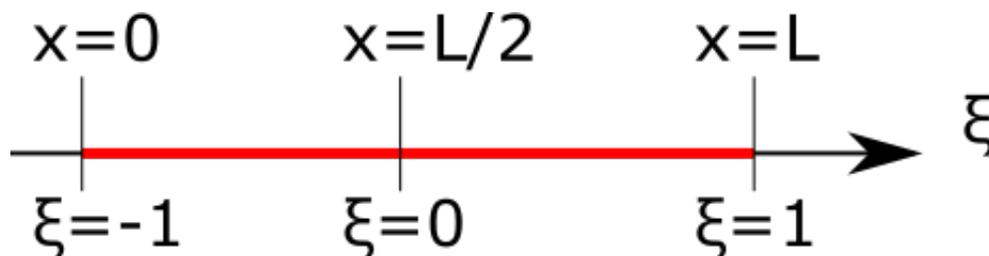
or for finding internal force of `e4` element with combination `D + 0.8 L` at it's centre:

```
var e4Force = (model.Elements["e4"] as BarElement).GetInternalForceAt(0,
↳combination1);
Console.WriteLine(e4Force);or ds
```

6.3 Iso Parametric Coordination System Of Elements Example

Apart from local and global coordination systems for elements, there is another system based on isoparametric formulation/representation, which is used extensively in finite element method. In BFE also in many places instead of local coordinate system, the iso parametric coordination is used.

6.3.1 Iso Parametric Coordination system for BarElement with two nodes



Based on

At the beginning point of the element, where $x=0$ the iso parametric coordinate is $\xi=-1$

At the central point of the element, where $x=L/2$, and L is length of elements, the iso parametric coordinate is $\xi=0$

At the end point of the element, where $x=L$, and L is length of elements, the iso parametric coordinate is $\xi=1$

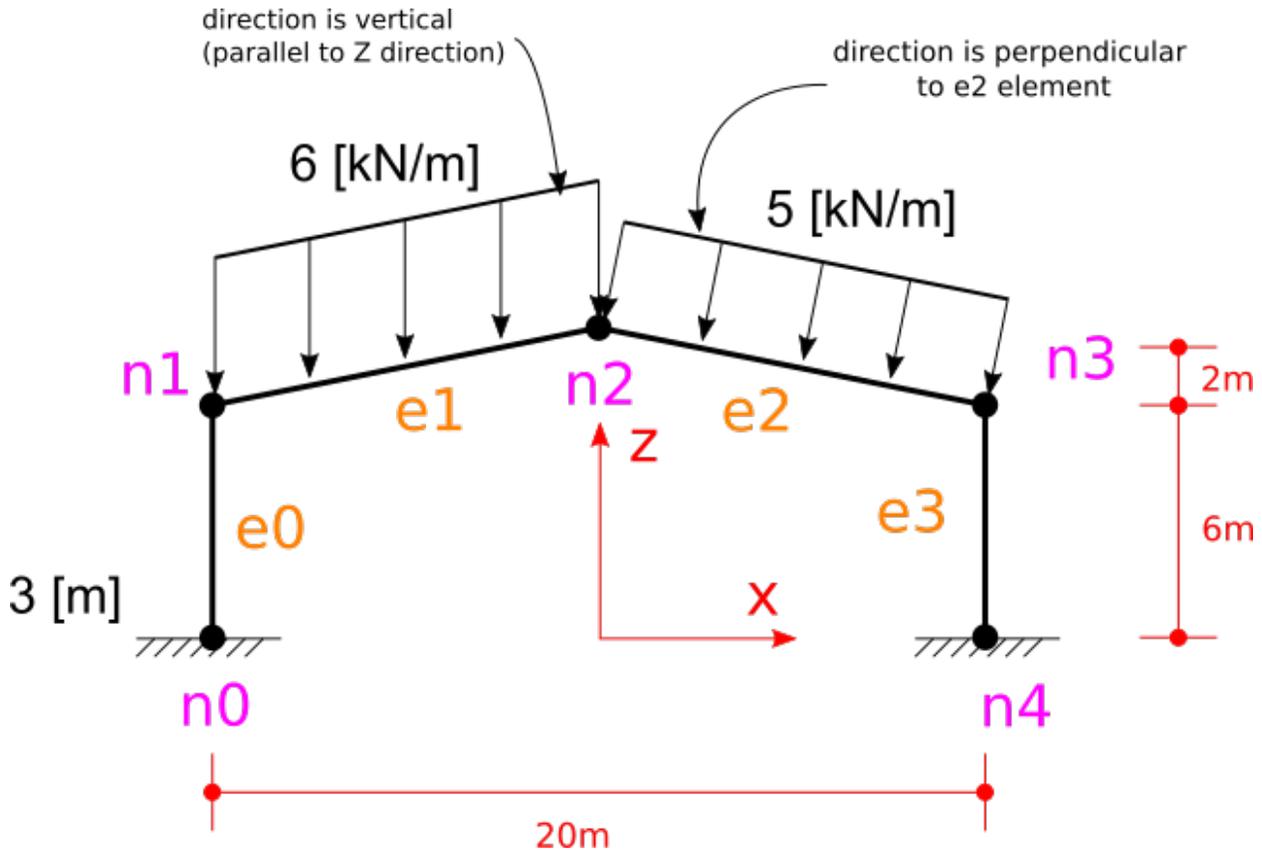
In bar element with two nodes the relation between isoparamtric ξ coordinate and local x coordinate is:

$x = (\xi + 1) * L/2$ and subsequently

$$\xi = (2*x-L) / L$$

6.4 Inclined Frame Example

Consider the inclined frame shown in fig below.



There are two loads on top elements. One has a 6 kn/m magnitude and its direction is vertical, another one has 5kn/m magnitude and it is perpendicular to the $e2$ element.

step 1: create model, nodes and elements:

```
var model = new Model();

model.Nodes.Add(new Node(-10, 0, 0) { Label = "n0" });
model.Nodes.Add(new Node(-10, 0, 6) { Label = "n1" });
model.Nodes.Add(new Node(0, 0, 8) { Label = "n2" });
model.Nodes.Add(new Node(10, 0, 6) { Label = "n3" });
model.Nodes.Add(new Node(10, 0, 0) { Label = "n4" });

model.Elements.Add(new BarElement(model.Nodes["n0"], model.Nodes["n1"]) { Label = "e0
↪" });
model.Elements.Add(new BarElement(model.Nodes["n1"], model.Nodes["n2"]) { Label = "e1
↪" });
model.Elements.Add(new BarElement(model.Nodes["n2"], model.Nodes["n3"]) { Label = "e2
↪" });
model.Elements.Add(new BarElement(model.Nodes["n3"], model.Nodes["n4"]) { Label = "e3
↪" });
```

step 2: define support nodes (nodal constraints)

```
model.Nodes["n0"].Constraints = model.Nodes["n4"].Constraints = Constraints.Fixed;
```

step 3: assign material and section to the elements

```
var secAA = new Sections.UniformGeometric1DSection(SectionGenerator.GetISetion(0.24, ↵
↵0.67, 0.01, 0.006));
var secBB = new Sections.UniformGeometric1DSection(SectionGenerator.GetISetion(0.24, ↵
↵0.52, 0.01, 0.006));
var mat = Materials.UniformIsotropicMaterial.CreateFromYoungPoisson(210e9, 0.3);

(model.Elements["e0"] as BarElement).Material = mat;
(model.Elements["e1"] as BarElement).Material = mat;
(model.Elements["e2"] as BarElement).Material = mat;
(model.Elements["e3"] as BarElement).Material = mat;

(model.Elements["e0"] as BarElement).Section = secAA;
(model.Elements["e1"] as BarElement).Section = secBB;
(model.Elements["e2"] as BarElement).Section = secBB;
(model.Elements["e3"] as BarElement).Section = secAA;
```

step 4: assign loads to elements

```
var u1 = new Loads.UniformLoad(LoadCase.DefaultLoadCase, new Vector(0,0,1), -6000, ↵
↵CoordinationSystem.Global);
var u2 = new Loads.UniformLoad(LoadCase.DefaultLoadCase, new Vector(0,0,1), -5000, ↵
↵CoordinationSystem.Local);

model.Elements["e1"].Loads.Add(u1);
model.Elements["e2"].Loads.Add(u2);
```

step 5: analyse the model

```
model.Solve_MPC();
```

step 6: get analysis results

Usually aim of analysis is to find some quantities like internal force and nodal displacements. After solving the model we can find nodal displacements with *Node.GetNodalDisplacement*, and *BarElement*'s internal force with *BarElement.GetInternalForceAt* and *BarElement.GetExactInternalForceAt* methods. There is a difference between the two methods. Details are available in *Internal Force And Displacement* section in documentation of *BarElement*.

for example the support reaction of node *N3* can be found and printed to application Console like this:

```
var n3Force = model.Nodes["N3"].GetSupportReaction();
Console.WriteLine("Support reaction of n4: {0}", n3Force);
```

This is the result of print on console:

```
Support reaction of n4: F: -37514.9891729259, 0, 51261.532772234, M: 0, -97714.6039503916, 0
```

Element's internal force can be found like this: For example need to find internal force of element in a point with distance of 1m (one meter) of start node. We can use *BarElement.GetInternalForceAt()* method to simply get the internal force of element at desired location of length of element, but there is an important thing here: and that is the input of *BarElement.GetInternalForceAt()* method is not in meter dimension not any other standard units of measuring length. The input is in another coordination system named iso-parametric crs. The isoparametric crs is widely used in FEM. More details about *BarElement* does have a method for converting

whole source code exists in the *BarInclinedFrameExample.cs* file.

6.5 Element Load Coordination System Example

The *Loads.UniformLoad* have a property named *CoordinationSystem* of enum type *BriefFiniteElementNet.CoordinationSystem* which defines the coordination system of load (for more info see *Coordination System*).

Using the combination of *UniformLoad.CoordinationSystem* and *UniformLoad.Direction* property, some specific distributed loads can be applied to elements.

Here are two examples:

6.5.1 Example 1

Consider the inclined frame shown in fig below, under dead load.

There is an inclined element of length = 5 [m], and an *UniformLoad* of magnitude 1000 [N/m].

- Magnitude of **1000 [N/m]**
- Direction of **Z**
- Coordination System of **Global**

step 1: create model, nodes and elements:

```
var m1 = new Model();

var e11 = new BarElement();

e11.Nodes[0] = new Node(0, 0, 0) { Constraints = Constraints.MovementFixed &
↳ Constraints.FixedRX, Label = "n0" };
e11.Nodes[1] = new Node(3, 0, 4) { Constraints = Constraints.MovementFixed, Label =
↳ "n1" };

e11.Section = new Sections.UniformGeometric1DSection(SectionGenerator.GetISection(0.24,
↳ 0.67, 0.01, 0.006));
e11.Material = UniformIsotropicMaterial.CreateFromYoungPoisson(210e9, 0.3);

var l1 = new Loads.UniformLoad();

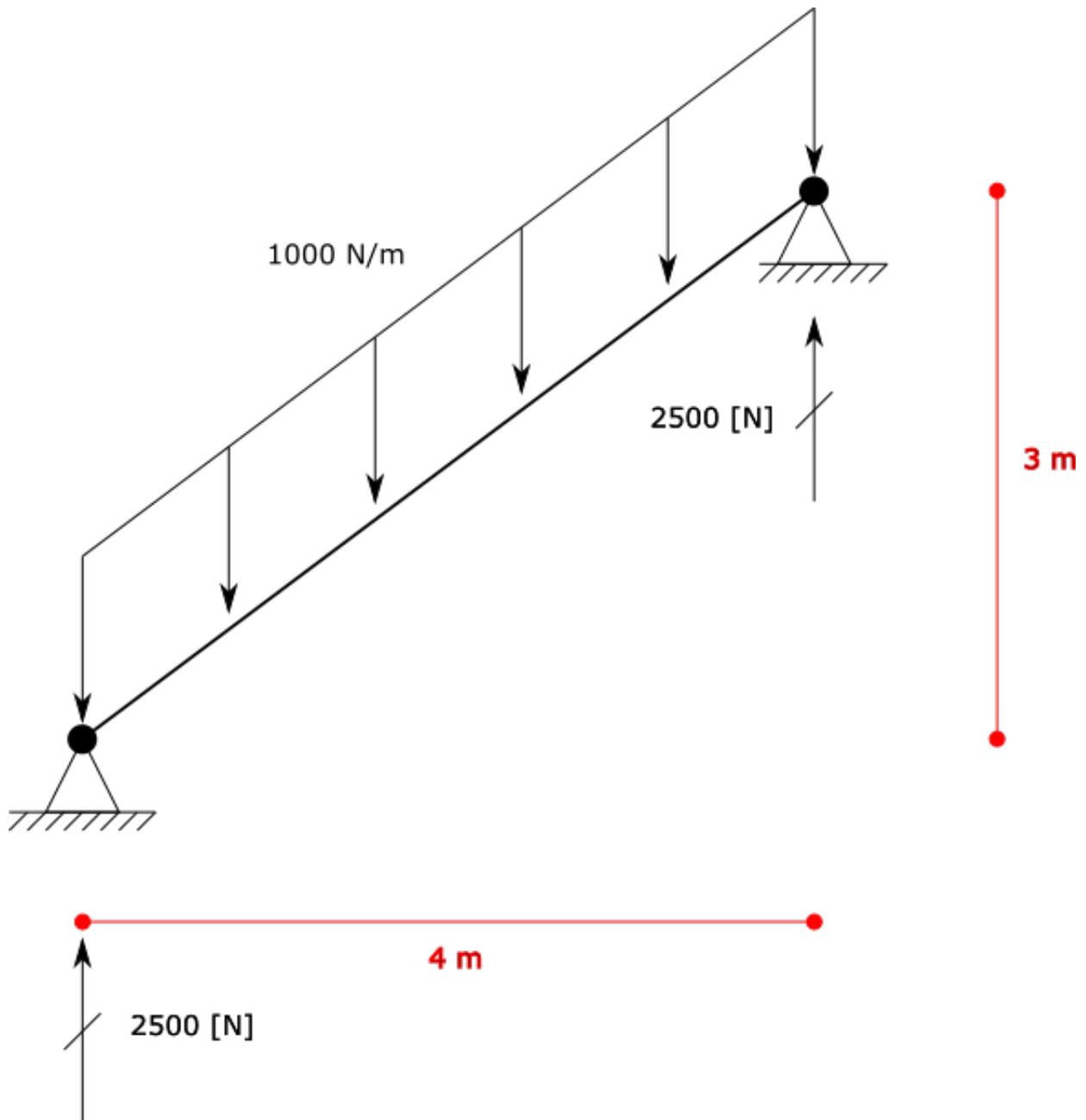
l1.Direction = Vector.K;
l1.CoordinationSystem = CoordinationSystem.Global;
l1.Magnitude = 1e3;

e11.Loads.Add(l1);

m1.Elements.Add(e11);
m1.Nodes.Add(e11.Nodes);

m1.Solve_MPC();

Console.WriteLine("n0 reaction: {0}", m1.Nodes[0].GetSupportReaction());
Console.WriteLine("n1 reaction: {0}", m1.Nodes[0].GetSupportReaction());
```

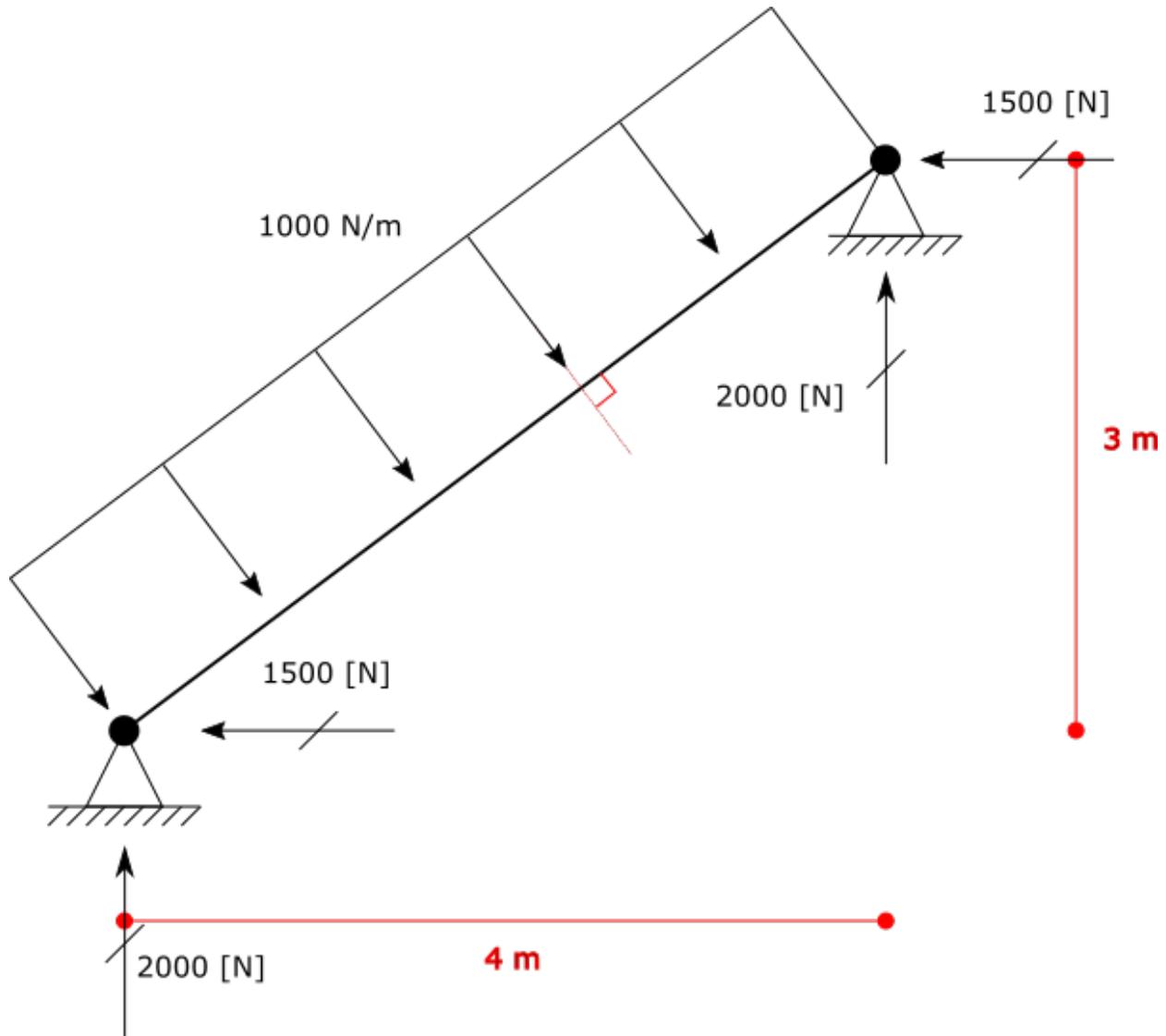


result

n0 reaction: F: 0, 0, -2500, M: 0, 0, 0 n1 reaction: F: 0, 0, -2500, M: 0, 0, 0

6.5.2 Example 2

Consider the inclined frame shown in fig below, under wind load.



There is an inclined element of length = 5 [m], and an `UniformLoad` of magnitude 1000 [N/m].

- Magnitude of **1000 [N/m]**
- Direction of **Z**
- Coordination System of **Local**

```
var m1 = new Model();
var e11 = new BarElement();
```

(continues on next page)

(continued from previous page)

```

e11.Nodes[0] = new Node(0, 0, 0) { Constraints = Constraints.MovementFixed &
↳ Constraints.FixedRX, Label = "n0" };
e11.Nodes[1] = new Node(3, 0, 4) { Constraints = Constraints.MovementFixed, Label =
↳ "n1" };

e11.Section = new Sections.UniformGeometric1DSection(SectionGenerator.GetISetion(0.24,
↳ 0.67, 0.01, 0.006));
e11.Material = UniformIsotropicMaterial.CreateFromYoungPoisson(210e9, 0.3);

var l1 = new Loads.UniformLoad();

l1.Direction = Vector.K;
l1.CoordinationSystem = CoordinationSystem.Local;
l1.Magnitude = 1e3;

e11.Loads.Add(l1);

m1.Elements.Add(e11);
m1.Nodes.Add(e11.Nodes);

m1.Solve_MPC();

Console.WriteLine("n0 reaction: {0}", m1.Nodes[0].GetSupportReaction());
Console.WriteLine("n1 reaction: {0}", m1.Nodes[0].GetSupportReaction());

```

result

n0 reaction: F: 2000, 0, -1500, M: 0, 0, 0

n1 reaction: F: 2000, 0, -1500, M: 0, 0, 0

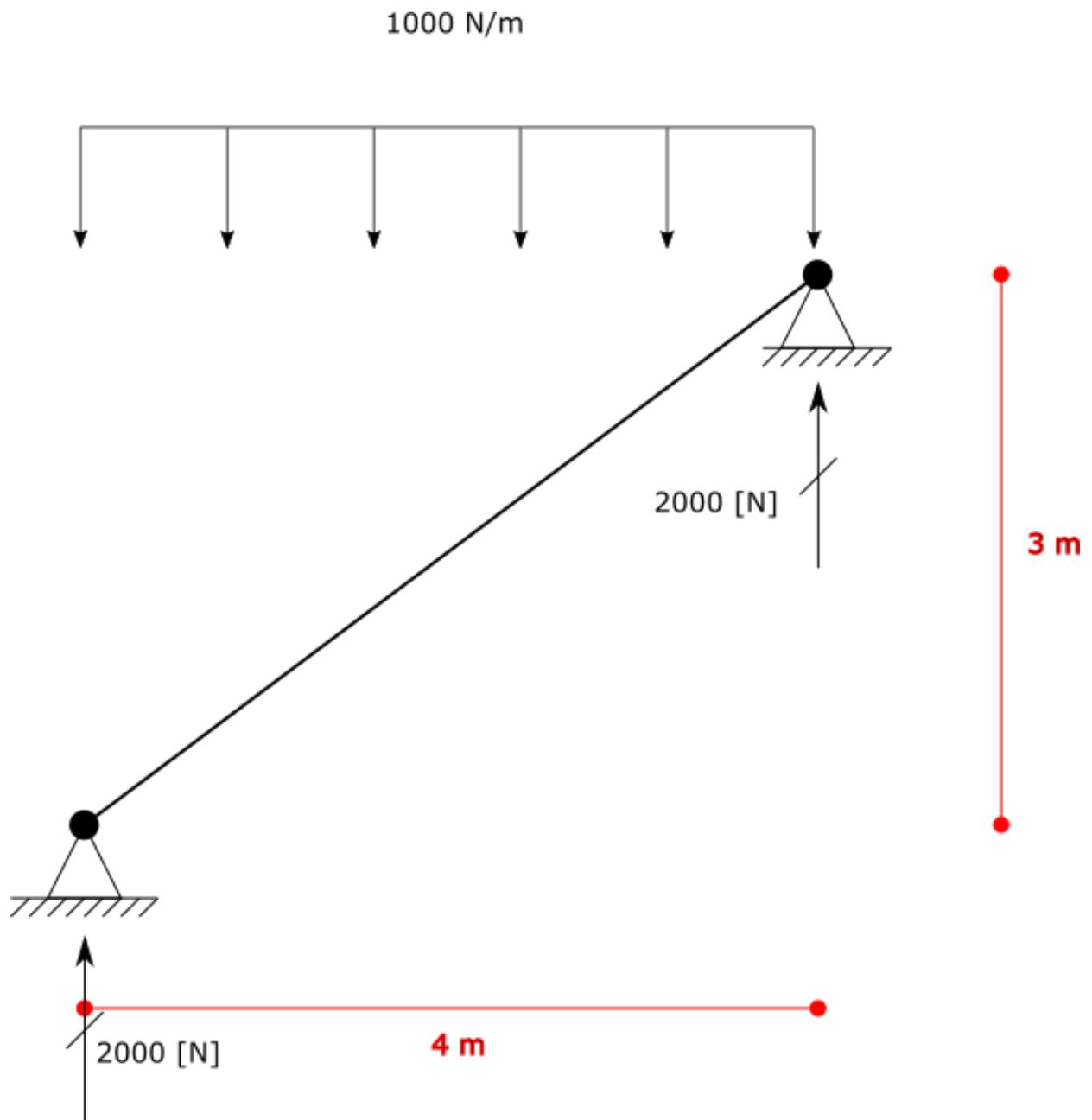
6.5.3 Example 3

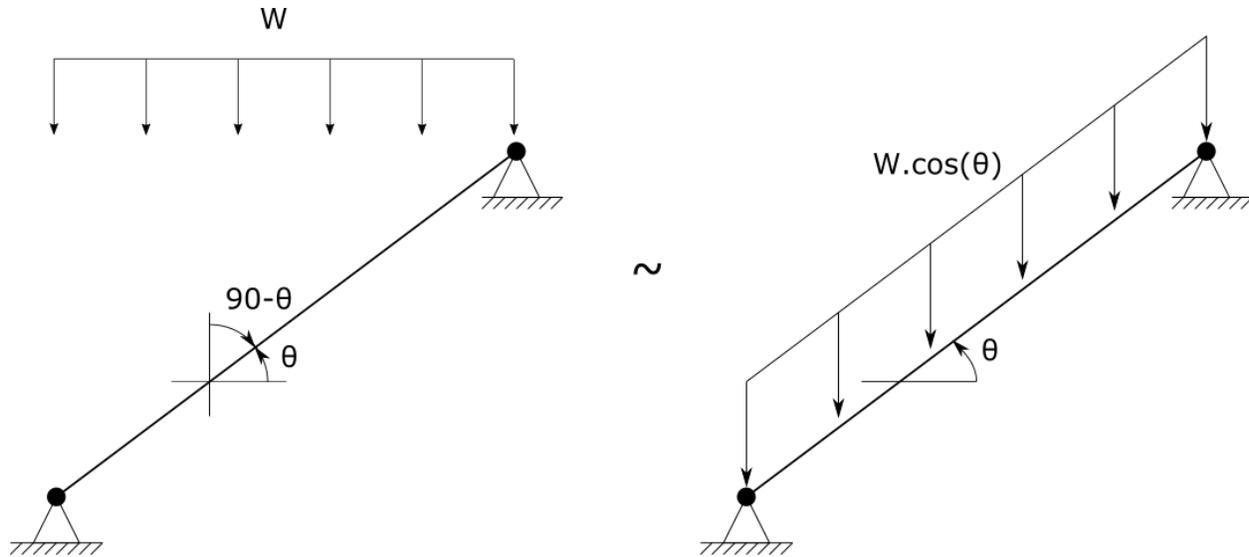
Consider the inclined frame shown in fig below, under snow load.

- Projected Magnitude of **1000 [N/m]**
- Direction of **Z**
- Coordination System of **Global**

There is an inclined element of length = 5 [m], and an UniformLoad of magnitude 1000 [N/m] which is projected. There is a difference about this type of load with two other examples above. For applying such projected load, first we have to convert it to example 1. Based on [tutorial in www.learnaboutstructures.com](http://www.learnaboutstructures.com) this is the way to convert:

So in this example we do not need theta value itself, but we need $Cos(\theta)$ or more precise absolute value of it $|Cos(\theta)|$. Due to elementary trigonometry relations $Cos(\theta) = Sin(90^\circ - \theta)$. So instead of $|Cos(\theta)|$ we can calculate $|Sin(\alpha)|$ where $\alpha = 90^\circ - \theta$ and α equals to angle between load direction and element direction. For finding $|Sin(\alpha)|$ we can use length of cross product of two unit vectors of element direction and load direction. This coefficient is always a non negative value and less than or equal to 1.0. If element is horizontal then $|Cos(\theta)| = 1.0$ if element is vertical then $|Cos(\theta)| = 0.0$.





Hint: Note that actually two vectors have to angles between them, a bigger one and a smaller one, but absolute value of cosine of them both are same i.e $|\cos(\theta)| = |\cos(180^\circ-\theta)|$

```
var m1 = new Model();
var e11 = new BarElement();

e11.Nodes[0] = new Node(0, 0, 0) { Constraints = Constraints.MovementFixed &
↳ Constraints.FixedRX, Label = "n0" };
e11.Nodes[1] = new Node(4, 0, 3) { Constraints = Constraints.MovementFixed, Label =
↳ "n1" };

e11.Section = new Sections.UniformGeometric1DSection(SectionGenerator.GetISetion(0.24,
↳ 0.67, 0.01, 0.006));
e11.Material = UniformIsotropicMaterial.CreateFromYoungPoisson(210e9, 0.3);

var loadMagnitude = -1e3;
var loadDirection = Vector.K;

var l1 = new Loads.UniformLoad();

var elementDir = e11.Nodes[1].Location - e11.Nodes[0].Location; //or n0 - n1, does not
↳ matter

var absCosTeta = Vector.Cross(elementDir.GetUnit(), loadDirection.GetUnit()).Length;

l1.Direction = loadDirection;
l1.CoordinationSystem = CoordinationSystem.Global;
l1.Magnitude = loadMagnitude * absCosTeta; //magnitude should multiple by reduction
↳ coefficient absCosTeta

e11.Loads.Add(l1);

m1.Elements.Add(e11);
m1.Nodes.Add(e11.Nodes);
```

(continues on next page)

(continued from previous page)

```
m1.Solve_MPC();

Console.WriteLine("n0 reaction: {0}", m1.Nodes[0].GetSupportReaction());
Console.WriteLine("n1 reaction: {0}", m1.Nodes[0].GetSupportReaction());
```

result

n0 reaction: F: 0, 0, 2000, M: 0, 0, 0 n1 reaction: F: 0, 0, 2000, M: 0, 0, 0

whole source code exists in the *UniformLoadCoordSystem.cs* file.

6.6 Cantilever Beam (Console Beam) Example

Consider the beam shown in fig below.

TO BE DONE

We need to find out the support reaction on node $n0$ and internal force of beam at any length x .

Step 1: making model

```
// Initiating Model, Nodes and Members
var model = new Model();

    var n1 = new Node(0, 0, 0);
    n1.Label = "n1";//Set a unique label for node
    var n2 = new Node(2, 0, 0) {Label = "n2"};//using object initializer for
↪assigning Label

    var e0 = new BarElement(n1, n2) { Label = "e1", Behavior = BarElementBehaviours.
↪FullFrame };

    model.Nodes.Add(n1, n2);
    model.Elements.Add(e1);

    e1.Section = new Sections.UniformParametric1DSection() { A = 9e-4 };

    e1.Material = Materials.UniformIsotropicMaterial.CreateFromYoungPoisson(210e9, 0.
↪3);

    n1.Constraints = Constraints.Fixed;

    var load = new UniformLoad();

    e1.Loads.Add(load);

    model.Solve_Mpc();

    var r1 = n1.GetSupportReaction();

    var fnc = new Func<double,double>(x=>e1.GetExactInternalForceAt(e1.LocalToIso(x)).
↪Fz);

    FuncVisualizer.VisualizeInNewWindow(fnc);

//note: code not complete
```

6.7 Sections for BarElement

6.7.1 Example 1

Consider example below, a cantilever beam with fixed start node and free end node, under defined loads.

image todo

The section is I section with :

- Width of $w = 10 \text{ cm}$ and
- Height of $h = 15 \text{ cm}$
- Flange thickness of $tf = 5 \text{ mm}$
- Web thickness of $tw = 5 \text{ mm}$
- Material is steel with elastic module of $E = 210 \text{ GPa}$ and Poisson's ratio of $\nu = 0.3$.

For defining the element itself we should do:

TODO

Till here the section for element is not defined. We can define the section in two ways:

- `UniformParametric1DSection`
- `UniformGeometric1DSection`

**** UniformParametric1DSection: **** If we want to define section with *UniformParametric1DSection*, as defined in section *Cross Section*, the parametric means that properties are parametrically defined one by one. for this example assuming we know that:

- Area of section is $A = 0 \text{ mm}^2$
- Second area moment around y axis is $I_y = 0 \text{ mm}^4$
- Second area moment around z axis is $I_z = 0 \text{ mm}^4$
- Torsion constant is $J = 0 \text{ mm}^4$

```
var section = new UniformParametric1DSection();
section.A = 1e-4;
section.Iy = section.Iz = 1e-6;
section.J = 2e-6;

var bar = new BarElement();
bar.CrossSection = section;
```

**** UniformGeometric1DSection: **** If we want to define section with *UniformGeometric1DSection*, as defined in section *Cross Section*, the geometric means that properties are parametrically defined one by one. For this example there is no need to know the properties of section, BFE will calculate them. We just need to define the section geometry:

```
var section = SectionHelper.GenerateISection(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0);
bar.CrossSection = section;
```

This section contains example items.

Code Design Documentation and History

There are two general types of elements available in BFE:

- Normal Finite Elements: Physical elements that provide stiffness, mass and damp matrices.
- MPC (Multy Point Constraint) Elements: are kind of virtual elements that binds several DoFs of a model together

8.1 Solving Procedure

By solving writer means converting model and loads and other things into mathematics form, [e.g. stiffness matrix, force and displacement vector], then solve the result linear equation system and finally convert back the results into .NET objects.

In this library solving a linear finite element model in structural topic have these steps:

1. Forming the full load vector and full stiffness matrix and full displacement vector. note that only unconstrained part of force and constrained part of displacement vector is filled with non-zeros in this step.
2. Applying boundary conditions due to support constrains and MPC (Multi Point Constrain) elements.
3. Solving the equation system as $Kx = F$ where x is unknown displacement
4. Finding the unknown forces (support reactions) with using unknown displacements.
5. Inserting full displacement and force values into full displacement and force vector.
6. Store full displacement and load vectors in *Model* for later usages.

8.1.1 Forming Full Stiffness, Load and Displacement Matrices

a *Model* can have unlimited *Nodes*, where each node have a fixed number of 6 DoF (Degree of Freedom). So the full stiffness matrix will be a square matrix of $6*n$ by $6*n$. Also force and displacement vectors will be vectors with length of $6*n$ that we use single column matrices to store them. A *double[]* array also can be used but we will use single column matrices next.

To form full stiffness matrix, we should first create a zero matrix with size of $6*n$ by $6*n$, then assemble the stiffness matrices, element by element into full stiffness matrix. This happens in *MatrixAssemblerUtil.AssembleFullStiffnessMatrix(Model)* method and returns the full stiffness matrix as a sparse matrix. For example imagine we have a model with 10k free DoFs, then the stiffness matrix would be a matrix by $10000*10000=100'000'000$ members. if each member need a 8 byte RAM as double precision value, then we'll need minimum 800 MB free ram to start analysing such a model. maybe 800 MB is found on all computers but about

a model with 100k free DoFs it would be around 800 GB of RAM, which is almost always is not present. Usually most of members of stiffness matrix are zeros, and member at row i and column j is usually zero if there is no element connecting the corresponding Nodes and DoFs. These type of matrices are named Sparse Matrices which have a few non-zero members. A structure with 4 roofs and 25 column in each level, will have a total number of 125 node, thus $125*6=750$ DoFs, but only 8250 non-zero members on stiffness matrix which is $\sim 1.5\%$ of full stiffness matrix. For a 10 roof structure with 100 column in each level, total nodes are 1000, total DoF are 6000 and non-zero count is 72k members which non-zero ratio is about 0.2%, and for a 50x50x50 3d grid non-zero ratio is 0.0017%. Several techniques are created for only storing the non-zero members of matrices in memory. After forming the stiffness matrix we should apply boundary conditions and then solve a linear system of equations to find unknown displacements. These procedures should all be done on sparse matrices. This library uses another library named CSparse.Net for handling sparse matrices, and that library uses Compressed Column Storage (CSR) format for keeping non-zero members of sparse matrices.

Also another zero matrix with length of $6*n$ by 1 is assumed as total displacement matrix and another one with same dimension for total force matrix. If we have any settlement on nodes, we'll fill them into appropriated members of total displacement vector, also we should convert loads applied on elements (like distributed loads) into equivalent nodal loads, and then add with nodal loads and then finally set members of total force vector. Note that in this stage all members corresponding to free DoFs in total displacement matrix are zero (which are unknown nodal displacements), also all members corresponding to fixed DoFs in force matrix are zero too (which are unknown support reactions).

8.1.2 Applying Boundary Conditions and MPC elements

After forming the total stiffness matrix and total force and displacement vectors, then we should apply the boundary conditions. There are at least a usual way of converting the stiffness matrix into four parts, also displacement and force vectors into two parts each like this:

$$K = \begin{bmatrix} K_{ff} & K_{fs} \\ K_{sf} & K_{ss} \end{bmatrix}$$

$$U = \begin{bmatrix} U_f \\ U_s \end{bmatrix}$$

$$F = \begin{bmatrix} F_f \\ F_s \end{bmatrix}$$

Where the f and s subscript tails in above names are related to Fixed and Not Fixed (or Released) DoFs. Then the main equation of $K * U = F$ will convert to this form:

$$\begin{bmatrix} K_{ff} & K_{fs} \\ K_{sf} & K_{ss} \end{bmatrix} * \begin{bmatrix} U_f \\ U_s \end{bmatrix} = \begin{bmatrix} F_f \\ F_s \end{bmatrix}$$

Expanding the matrix multiplies:

$$K_{ff} * U_f + K_{fs} * U_s = F_f$$

$$K_{fs} * U_f + K_{ss} * U_s = F_s$$

We have all terms known, expect U_f and F_s . We are searching for the U_f and F_s which are displacement of free DoFs and external force on fixed DoFs (e.g support reactions). So we can rewrite these into:

$$U_f = K_{ff}^{-1} * (F_f - K_{fs} * U_s)$$

And then we can find F_s (support reactions) from this:

$$F_s = K_{fs} * U_f + K_{ss} * U_s$$

finally both unknown term will be found.

Also for applying effect of MPC elements, Since the models are linear, the master slave method is used for considering the rigid elements. You can find a detail about how it is with this paper. In linear analysis displacement vector, force

vector and stiffness matrix should be determined and after some calculation, we should solve a linear equation system and displacements will be found.

Because of reducing the count of degree of freedoms (DoF) of structure, rigid element will reduce the stiffness, mass and damp matrix dimensions. For example, consider a problem with no settlement which can be shown as:

$$F = K.U \text{ then } U = K^{-1} * F$$

Let's say we have n DoFs (in previous section we used n for total number nodes, but here as total number of DoFs, because this section is copied from another article). If rigid elements do connect two nodes with constrained supports, we can define a matrix Pf ($m * n$) which when multiplied to the F will get a Fr vector which is force vector for Reduced structure (after applying the rigid elements to reduce DoFs). Also can define a Pd ($n * m$) matrix, which when is multiplied to Dr (Dr is displacement vector for reduced structure) will give the D which is displacement vector for original structure as below:

$$Fr = Pf * F$$

$$D = Pd * Dr$$

Can combine these two equations with first one as:

$$F = K . Pd . Dr \text{ (pre multiply both sides with Pf)} \Rightarrow Pf . F = Fr = Pf . K . Pd . Dr$$

$$\text{Taking } Kr = Pf . K . Pd$$

$$Fr = Kr * Dr$$

This way, we can convert the problem into a reduced problem. Same can be applied for dynamic analysis:

$$M . \ddot{X} + C . \dot{X} + K . X = F(t)$$

Taking:

$$X = Pd * Xr \Rightarrow \ddot{X} = Pd * \ddot{r} \Rightarrow \dot{X} = Pd * \dot{r}$$

Then:

$$M . Pd * \ddot{r} + C . Pd * \dot{r} + K . Pd * Xr = F(t)$$

Premultiply which Pf:

$$Pf . M . Pd * \ddot{r} + Pf . C . Pd * \dot{r} + Pf . K . Pd * Xr = Pf . F = Fr$$

$$Fr = Mr . \ddot{r} + Cr . \dot{r} + Kr . Xr$$

where:

$$Mr = Pf . M . Pd$$

$$Cr = Pf . C . Pd$$

$$Kr = Pf . K . Pd$$

These two ways of solving system for unknowns and also applying effect of MPC elements, are simple ways but not used in latest version of BFE, but in earlier version, because combining these two procedures will probably result in a complex code and also error prone as i remember when i was dealing permutation thing and that resulted into a class named *DofMappingManager* (probably still presented in the source code) with very hard usage. Instead another method is used to apply both boundary conditions and extra equation of MPC element in same time. This method is described in next.

8.1.3 Applying Boundary Conditions and MPC elements - new method

- Step 1: Extract equations related to boundary conditions and constraints
- Step 2: None

- Step 3: Create Reduced Row Echelon Form (RREF) of Step 1
- Step 4: Make pioneer members to -1 by multiplying whole row (for definition of pioneer members please continue reading)
- Step 5: Insert each row into appropriated row of a $n \times n$ matrix where n is total number of DoFs
- Step 6: Remove extra columns

Step 1: Extract equations related to boundary conditions and constraints

A Finite Element model does have boundary conditions (e.g support DoFs and settlements), also MPC (Multi Point Constraint) elements like rigid diaphragm, and SPC (Multi Point Constraint) elements like virtual constraints. All of these can be represented as equations. For example:

- $U_{11} = 0$: U_{11} DoF is connected to ground without settlement
- $U_{12} = 0.1$: U_{12} DoF is connected to ground with settlement amount of 0.1 in that direction
- $U_{12} = U_{13}$: U_{12} is equal to U_{13}
- $U_{16} = 2 * U_{12} + 3 * U_{13}$: U_{16} is connected to U_{12} and U_{13} with a MPC element (like rigid diaphragm or ...).

Every boundary condition and and MPC/SPC element will give a set of these extra equations, and every equation can be represented as a row of a matrix with column count equal to total number of DoFs in that model, plus a right hand side vector. Above equations can turn into matrix rows plus a right hand side like below table:

Table 1: Title

Eq. Number	Equation	U_{11} 's coeff.	U_{12} 's coeff.	U_{13} 's coeff.	U_{14} 's coeff.	U_{15} 's coeff.	U_{16} 's coeff.	Right Side
1	$U_{11} = 0$	1	0	0	0	0	0	0
2	$U_{12} = 0.1$	0	1	0	0	0	0	0.1
3	$U_{13} = U_{12}$	0	-1	1	0	0	0	0
4	$U_{16} = 2 * U_{12} + 3 * U_{13}$	0	-2	-3	0	1	0	0

Finally there will be a system with m rows and n columns and a right side vector:

TODO

$$P_1 * U = R_1$$

Step 3: Create Reduced Row Echelon Form (RREF) of Step 1

Next step is to compute RREF form of P_1 matrix calculated in step 1 with gauss elimination. We should start from column 0, choose a row with non-zero member at column 0, then eliminate members of all other rows that have a non-zero element at column 0. Do same thing for all columns from 1 to n, where n is total number of DoFs. The operation will stop when in every i 'th column of matrix. all members are zero or at most one non-zero element. In other words elimination will stop when in each column there is at most one non-zero member. After elimination done for each i 'th row there is three possible cases:

1. There are one or several non-zeros on row i .
2. All members of row i are zero, also right side at row i is zero. This means the equation corresponding to that row was not useful, but also is not a problem. For example $U_1 = U_2$, $U_2 = U_3$, $U_3 = U_1$ can be result of three SPC elements, but only two of them are useful and third one is result of first and second.

3. All members of row i are zero, but right side at row i is non-zero (we should also consider floating point operation stuff, so check with small epsilon number instead of zero 0.0). This means an error. Like two inconsistent settlements on two DoFs or nodes that are connected with a SPC or MPC element. $U_1 = 0.1, U_2 = 0.2, U_1 = U_2$.

Rows with all members zero and right side zero will be removed from result, and rows with all members zero but right side non zero will cause solving procedure failure, because of invalid user input.

Finally there will be a matrix P_3 with o rows and n columns, that $o \leq m$ (m is total extra equation count) due to removing useless rows. Also as this matrix is RREF form, then there are o columns with only one non-zero element. If a member be the only non-zero member in the column, we call that member “pioneer” or “leading” member. Finally we have $P_3 * U = R_3$ where P_3 is in RREF form.

Step 4: Make pioneer members equal to -1 by multiplying whole row with a coefficient

we should take output of last step, P_3 and R_3 . Then multiply each row and corresponding right side member with a coefficient in a way that pioneer member turn into -1.0 . result is P_4 and R_4 .

Step 5: Insert each row into appropriated row of a $n \times n$ matrix where n is total number of DoFs

Create an empty P_5 matrix with size n by n , also a vector R_5 with size n by 1 . Then for each i 'th row of P_4 with pioneer member at column j , insert it into i 'th row of P_5 and R_5 . Next we should replace the zeros on main diagonal of P_5 with 1.0 and no change in R_5 .

Step 6: Remove extra columns

Remove columns that have pioneer member equal to -1.0 and no change to right side. Final result is P_6 with size n by o and R_6 with size n by 1 .

P_6 is displacement expander and o is total number of master DoF count.

Notes

There is an interface named *IDisplacementPermutationCalculator* in the namespace *BriefFiniteElementNet.Mathh* which should do all 6 steps above or an output equivalent to output of step 6.

What we want to do is to solve $F_t = K_t * U_t$ where there are some extra equations. Maybe there are other ways to handle this, for example maybe QR decomposition. But this is a way also. . .

Example

Step 1:

$$[0 \ 0 \ 1 \ 1 \ 3 \ 0 \ 2] [x_0] = [3] \ [0 \ 0 \ 2 \ 6 \ 1 \ 0 \ 5] \times [x_1] = [1] \ [0 \ 0 \ 3 \ 7 \ 4 \ 0 \ 7] [x_2] = [4]$$

$$[x_3] [x_4] [x_5]$$

$$[x_6]$$

Step 2: None

$$\text{Step 3: } [+0.00 \ +0.00 \ +1.00 \ +0.00 \ +4.25 \ +0.00 \ +1.75 \ | \ +4.25] \ [+0.00 \ +0.00 \ +0.00 \ +1.00 \ -1.25 \ +0.00 \ +0.25 \ | \ -1.25]$$

$$\text{Step 4: } 2 \text{ by } 7 \ [+0.00 \ +0.00 \ -1.00 \ +0.00 \ -4.25 \ +0.00 \ -1.75 \ | \ -4.25] \ [+0.00 \ +0.00 \ +0.00 \ -1.00 \ +1.25 \ +0.00 \ -0.25 \ | \ +1.25]$$

Step 5: 7 by 7 [+1.00 +0.00 +0.00 +0.00 +0.00 +0.00 +0.00 | +0.00] [+0.00 +1.00 +0.00 +0.00 +0.00 +0.00 +0.00 | +0.00] [+0.00 +0.00 -1.00 +0.00 -4.25 +0.00 -1.75 | -4.25] [+0.00 +0.00 +0.00 -1.00 +1.25 +0.00 -0.25 | +1.25] [+0.00 +0.00 +0.00 +1.00 +0.00 +0.00 | +0.00] [+0.00 +0.00 +0.00 +0.00 +0.00 +1.00 +0.00 | +0.00] [+0.00 +0.00 +0.00 +0.00 +1.00 | +0.00] [+0.00 +0.00 +0.00 +0.00 +0.00 +0.00 | +1.00]

step 6: result is 7 by 5 [+1.00 +0.00 +0.00 +0.00 +0.00 | +0.00] [+0.00 +1.00 +0.00 +0.00 +0.00 | +0.00] [+0.00 +0.00 -4.25 +0.00 -1.75 | -4.25] [+0.00 +0.00 +1.25 +0.00 -0.25 | +1.25] [+0.00 +0.00 +1.00 +0.00 +0.00 | +0.00] [+0.00 +0.00 +1.00 +0.00 | +0.00] [+0.00 +0.00 +0.00 +0.00 +1.00 | +0.00]

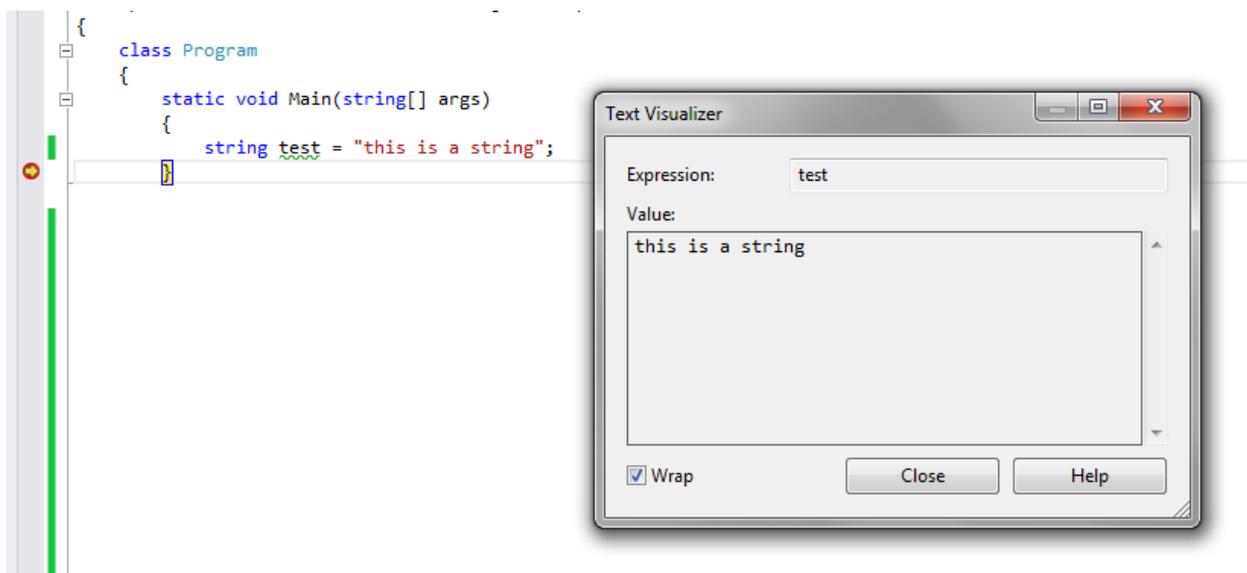
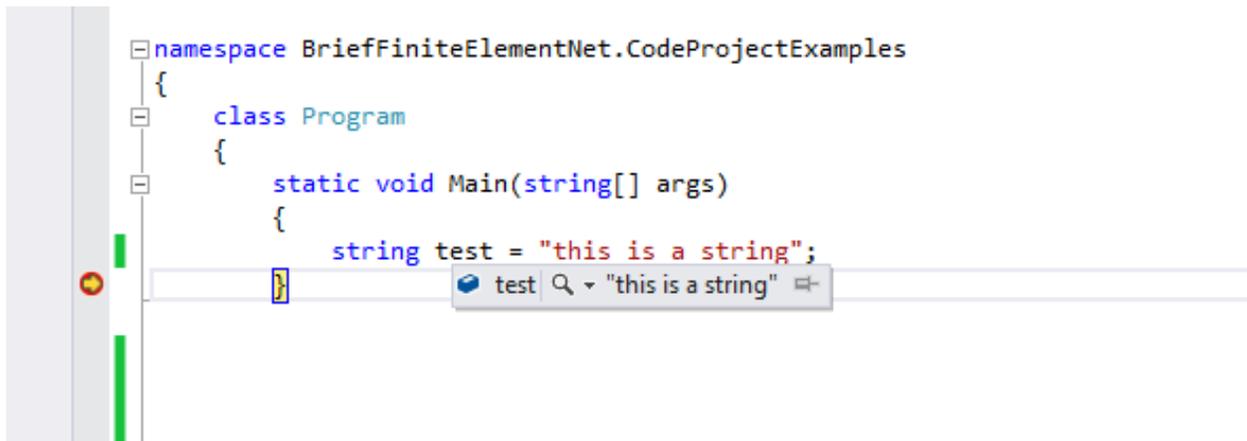
P_U is left side (this 7 by 5 matrix) and R is right side vector (1 by 7 matrix) $U_t = P_U * U_r$, where t is Total, r is Reduced.

m

8.2 Install Debugger Visualizers

8.2.1 What is debugger visualizer

In visual studio when you are debugging an application, you can view the live values of variables like image below:



In image you can see there is a magnifier icon (debug2.png) next to test variable, this shows that there is a debugger visualizer for string type and you can click on the icon to see the details:



BriefFiniteElement.NET library also contains debugger visualizer for visualizing the `BriefFiniteElementNet.Model` instances but you have to install the visualizer first and then when in debug mode you move the cursor on any variable of type `Model`, then the magnifier icon will appear and you can click on that to see your model.

Usually installing a debugger visualizer in Visual Studio is as easy as copying a bunch of dll files in this address of your hard drive: `C:\Users\{YOUR_USER_NAME}\Documents\Visual Studio 20XX\Visualizers` where `{YOUR_USER_NAME}` is pointing to current user profile address and '20XX' is related to installed visual studio (2013 or 2012 or 2010 or ...).

- note: if Visualizers folder not exists, it should be created.

8.2.2 Installing debugger visualizers inside Visual Studio

You should first Download latest source code of project. There are several solution files, for example *BriefFiniteElementNet.VS2019.sln* corresponds to Visual Studio 2019. There is several solution files in the package. Based on version of your Visual Studio you should open one of these solution files:

BriefFiniteElementNet.VS2010.sln (for Visual Studio 2010) BriefFiniteElementNet.VS2012.sln (for Visual Studio 2012) BriefFiniteElementNet.VS2013.sln (for Visual Studio 2013)

BriefFiniteElementNet.VS2015.sln (for Visual Studio 2015) BriefFiniteElementNet.VS2019.sln (for Visual Studio 2019)

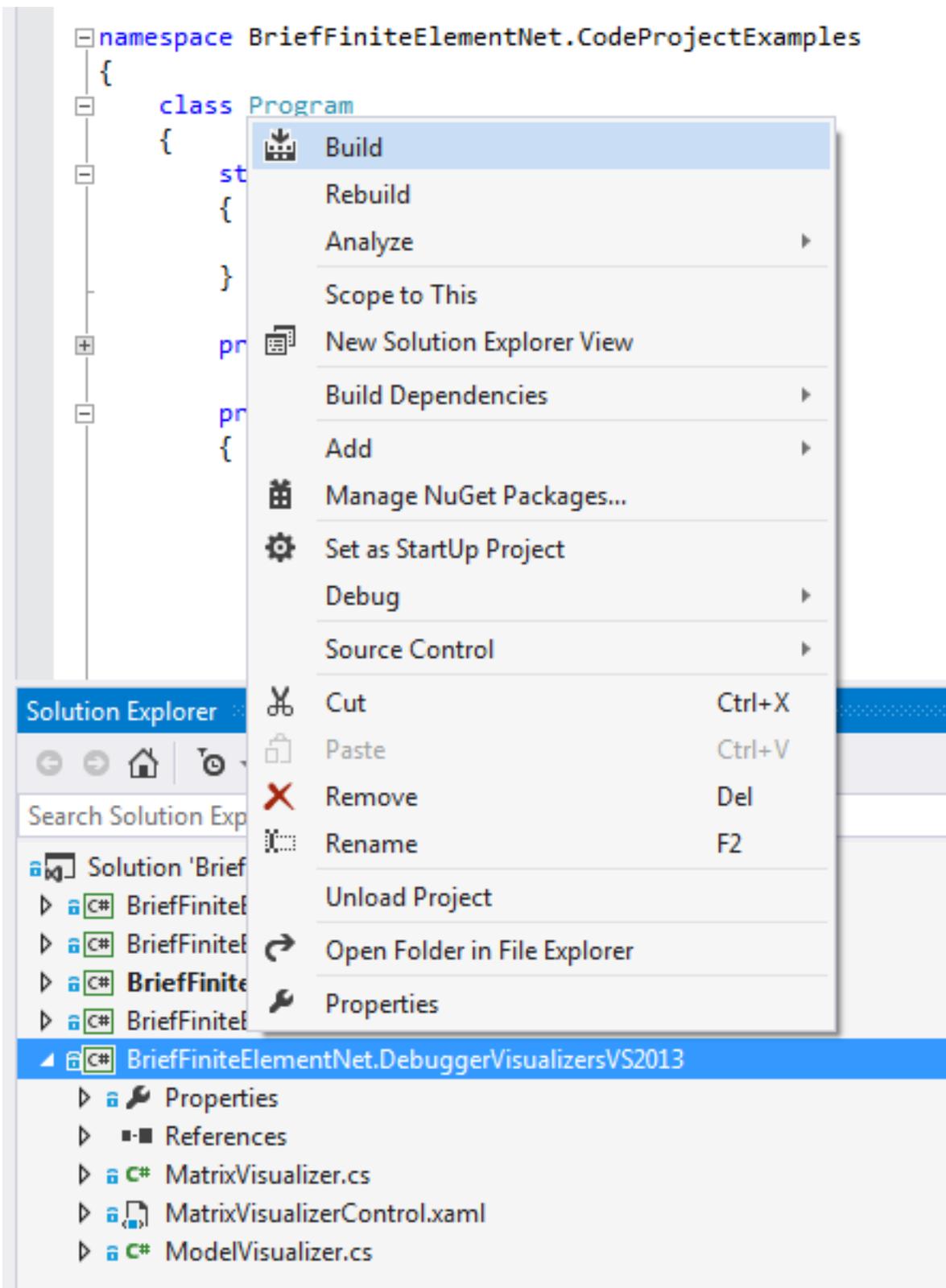
After opening the solution file, there is a project named 'BriefFiniteElementNet.DebuggerVisualizersVS20XX' where VS20XX matches the Visual Studio version number running on local computer, simply right click on it and click Build to build it like this image:

after successful build of project it does automatically copy appropriated files into the "C:\Users\{YOUR_USER_NAME}\Documents\Visual Studio 20XX\Visualizers" using Post-Build events of project, and there is no need to do anything manually. The files that will copied to that address with build are:

- BriefFiniteElementNet.DebuggerVisualizers.dll
- BriefFiniteElementNet.dll
- BriefFiniteElementNet.Controls.dll
- BriefFiniteElementNet.Common.dll
- HelixToolkit.Wpf.dll
- DynamicDataDisplay.dll

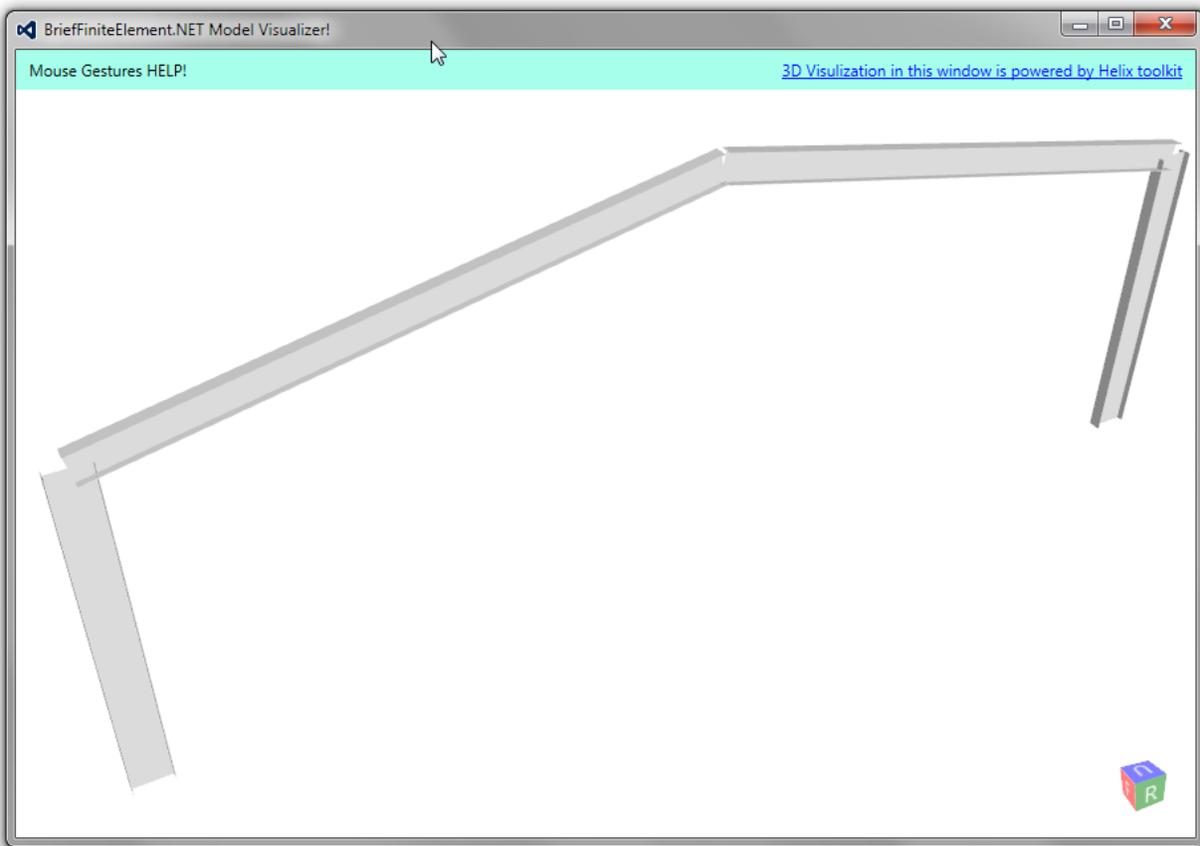
Next time you debug your code, when move mouse to a variable with `Model` type you will see a magnifier icon like this:

and you should simply click it to visualize and see your model like this:



```
}  
  
private static void Example2()  
{  
    var model = new Model();  
  
    var n1 = new Node(-10, 0, 0);  
    var n2 = new Node(-10, 0, 6);  
    var n3 = new Node(0, 0, 8);  
    var n4 = new Node(10, 0, 6);  
    var n5 = new Node(10, 0, 0);  
  
    model.Nodes.Add(n1, n2, n3, n4, n5);  
  
    var secAA = SectionGenerator.GetISetion(0.24, 0.67, 0.01, 0.006);  
    var secBB = SectionGenerator.GetISetion(0.24, 0.52, 0.01, 0.006);  
  
    var e1 = new FrameElement2Node(n1, n2);  
    var e2 = new FrameElement2Node(n2, n3);  
    var e3 = new FrameElement2Node(n3, n4);  
    var e4 = new FrameElement2Node(n4, n5);  
  
    e1.Geometry = e4.Geometry = secAA;  
    e2.Geometry = e3.Geometry = secBB;  
  
    e1.UseOverridedProperties =  
        e2.UseOverridedProperties = e3.UseOverridedProperties = e4.UseOverridedProperties = false;  
  
    model.Elements.Add(e1, e2, e3, e4);  
}
```

model (BriefFiniteElementNet.Model)



9.1 Force

Force object represents a general concentrated force in 3D (3 force component and 3 moment components).

9.1.1 F_x

F_x represents the X component of force

9.1.2 F_y

F_y represents the Y component of force

9.1.3 F_z

F_z represents the Z component of force

9.1.4 M_x

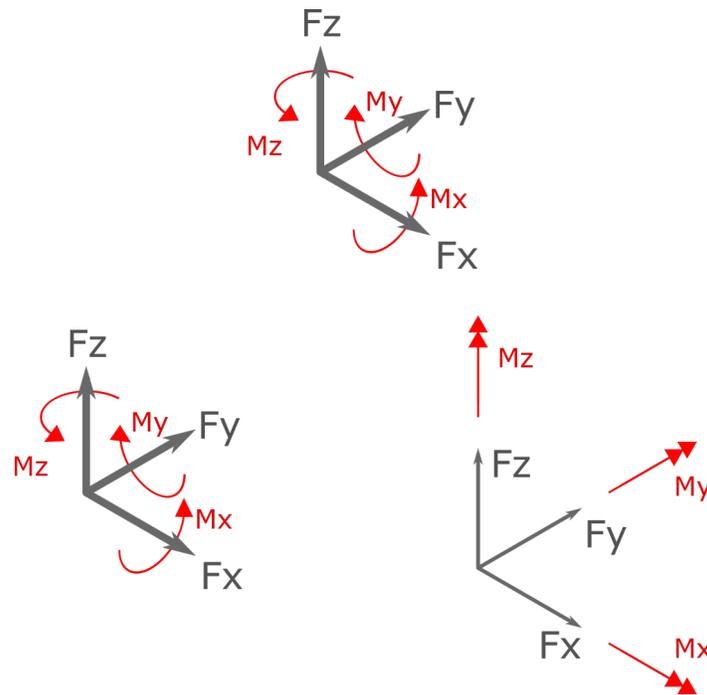
M_x represents the X component of moment

9.1.5 M_y

M_y represents the Y component of moment

9.1.6 Mz

Mz represents the Z component of moment



Examples —

```
var force1 = new Force(); force1.Fx = 10;//x component of force equal to 10 [N] force1.Mz = 15;//z
component of moment equal to 15 [N.m]
```

```
var force2 = new Force(10,0,0,15,0,0);//using constructor, parameters are fx,fy,fz,mx,my,mz //force2
is equal to force1
```

9.2 Displacement

Displacement object represents a general displacement in 3D (6 DoF, 3 straight displacement and 3 rotational).

9.2.1 Dx

Dx represents the X component of displacement (Δx)

9.2.2 Dy

Dy represents the Y component of displacement (Δy)

9.2.3 Dz

Dz represents the Z component of displacement (Δz)

9.2.4 RX

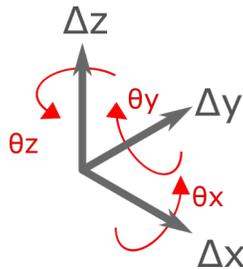
R_x represents the X component of rotation (θ_x)

9.2.5 Ry

R_y represents the Y component of rotation (θ_y)

9.2.6 Rz

R_z represents the Z component of rotation (θ_z)



Note that unit of rotations are Radians.

9.3 LoadCase

A `LoadCase` represents a group of loads. The `LoadCase` struct have a `nature` property (an enum type) and a `title` property (with string type). `LoadNature` can be: `Default`, `Dead`, `Live`, `Snow`, `Wind`, `Quake`, `Crane` and `Other`.

9.4 LoadCombination

A `LoadCombination` Represents a load combination which consists of a set of `Loads` and a `Factor` for each `Load`.

9.4.1 Examples

9.5 Point

`Point` object Represents a point in 3D space

9.5.1 X

X represents the X location of point

9.5.2 Y

Y represents the Y location of point

9.5.3 Z

Z represents the Z location of point

9.6 Vector

`Vector` object Represents a vector in 3D space

9.6.1 X

X represents the X component of vector

9.6.2 Y

Y represents the Y component of vector

9.6.3 Z

Z represents the Z component of vector

Brief Finite Element .NET (or BFE.NET or BFE) is an object oriented framework that enables .NET developers to do some brief LINEAR Finite Element modelling and analysis using .NET objects.

If you like to learn by writing code, we'd recommend one of our *Brief Finite Element .NET* guides to get you started with BFE.NET.