TrussPy Documentation

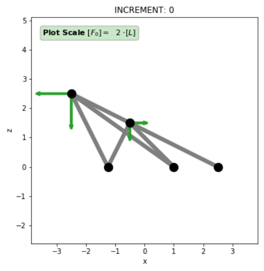
Release 2018.08

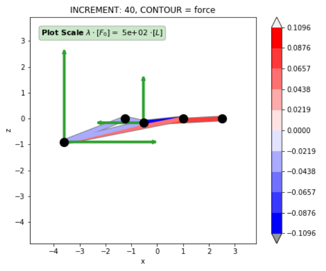
Trusspy

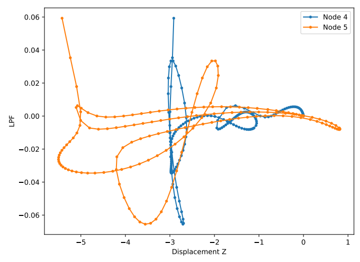
Aug 27, 2018

CONTENTS

TrussPy is a 3D **Truss**-Solver written in **Py**-thon which is capable of material and geometric nonlinearities. It uses an object-oriented approach to structure the code in meaningful classes, attributes and methods. TrussPy contains both multistep functionality (multiple loadcase analysis with sequenced external forces) and an adaptive method to control incremental stepwidths. Input files may be written in Excel or directly in Python. A simple post-processing inside TrussPy is directly available via Matplotlib. Model Plots whether in undeformed or deformed configuration with optional contour plots on element forces are easy to show. They may also be generated for a series of increments and saved as a GIF Movie. Last but not least History (a.k.a. x-y) Plots for a series of increments or Path Plots along a given node path may be generated for nodal properties (displacements, forces) or global quantities like the Load-Proportionality-Factor (LPF).







CHAPTER ONE

INSTALLATION

1. Either use pip to install TrussPy (recommended, **but not yet available!**)

   ```
   pip install trusspy
   ```

2. **or download this repo as ZIP-file, unzip it and add the unzipped location to your PYTHONPATH. On Windows 10:**

   - open startmenu
   - search for "environment variable"
   - edit environment variable for this account
   - **New… (user variable)**
     - variable name: PYTHONPATH
     - variable value: *C:\Path\to\trusspy_package\*
   - to verify the correct path *C:\Path\to\trusspy_package\* this directory should contain the following files and folders:

     ```
     + docs\
     + tests\
       - e101\
       - e102\
       - ...
     + trusspy\
       - core\
       - handlers\
       - model.py
       - ...
     README.rst
     LICENSE
     ```

Hint: *Method 2) is a quick option for using the latest version of trusspy without installing it.*

Not yet familiar with Python but want to have a look at TrussPy in action? Give Python a try, it's not that hard to start with. Install one of the popular scientific Python 3.x distributions

- Anaconda (Windows, Max, Linux) or
- WinPython (Windows)

and you're ready to go. Python 3 Tutorials? Google is your friend (also search for NumPy, SciPy, SymPy and Matplotlib - these packages are already included with the above mentioned distributions). Ready to go? Proceed with 1).

CHAPTER

TWO

GETTING STARTED

The following basic example is written in interactive mode to show how trusspy works. *For mid-sized models it may be more convenient to use a Spreadsheet (Excel) - input file*. All model parameters except allowed incremental quantities are assumed with default values to enable a clean tutorial for the model creation process. We will consider a model with two nodes and one truss. Although this configuration does not include any geometric nonlinear effects it is the most basic example to start with. The left end of the truss (Node 1) is fixed whereas the right end displacement is free in direction x (Node 2). An external force acts on the right end of the truss in direction x. To sum up, this model contains two nodes, one element and one degree of freedom (DOF).

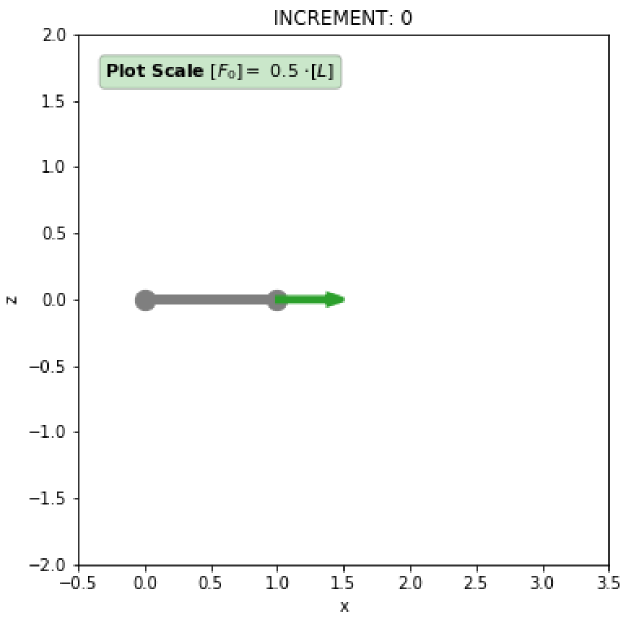


Fig. 2.1: The undeformed model with an **External Force** vector acting on **Node** 2.

First we import trusspy with it's own namespace and create a Model object `M`.

```
import trusspy as tp

# init model
M = tp.Model()
```

Now we create Nodes with coordinate triples and Elements with a list of node connectivities and both material and geometric properties. Both Nodes and Elements are identified with their label.

```
# create nodes
#
#    tp.Node( label,   coord  )
# ----------------------------
N1 = tp.Node(      1,  (0,0,0) )
N2 = tp.Node(      2,  (1,0,0) )

# create element
young = 1
area  = 1
E1 = tp.Element( 1, [1,2], material_properties=[young], geometric_properties=[area] )
```

Mechanical boundary conditions must be supplied for all nodes which contain locked DOF's: *0 = inactive (locked)* and *1 = active (free)*. The same applies to external forces - no **External Force** object has to be added to the **Model** if all components of a node are zero.

```
# create displacement (U) boundary conditions
B1 = tp.BoundaryU( 1,  (0,0,0) )
B2 = tp.BoundaryU( 2,  (1,0,0) )

# create external forces
# F1 = tp.ExternalForce( 1,  (0,0,0) ) # not necessary
F2 = tp.ExternalForce( 2,  (1,0,0) )
```

We have to specify some important **Settings Parameters** concerning the trusspy path-tracing algorithm:

```
M.Settings.dlpf = 0.02 # maximum allowed incremental load-proportionality-factor
M.Settings.du   = 0.02 # maximum allowed incremental displacement component
M.Settings.incs = 50   # maximum number of increments
```

All generated items have to be added to the model `M`. Of course this is cumbersome for bigger models. Alternativly, one may wrap for example the Node object creation inside the `node_add` function of the Model object: `M.Nodes.add_node(tp.Node(label, coord))`. Either way we are able to `build` the model and `run` the job afterwards. The nodal ordering of Nodes, Boundaries and Forces inside the corresponding *add* function doesn't matter. TrussPy will sort all nodal quantities by their node labels in the *build* method.

```
# add items to the model
M.Nodes.add_nodes([N1,N2])
M.Elements.add_element(E1)
M.Boundaries.add_bounds_U([B1,B2])
M.ExtForces.add_forces([F2])

# build model and run job
M.build()
M.run()
```

When the job has finished we may post-process the deformed model and plot the force-displacement curve at Node 2.

```
# show results
M.plot_model(config=['deformed'],
             view='xz',
             contour='force',
             lim_scale=(-0.5,3.5,-2,2),
             force_scale=1.0,
             inc=-1)
M.plot_history(nodes=[2,2], X='Displacement X', Y='Force X')
M.plot_show()
```

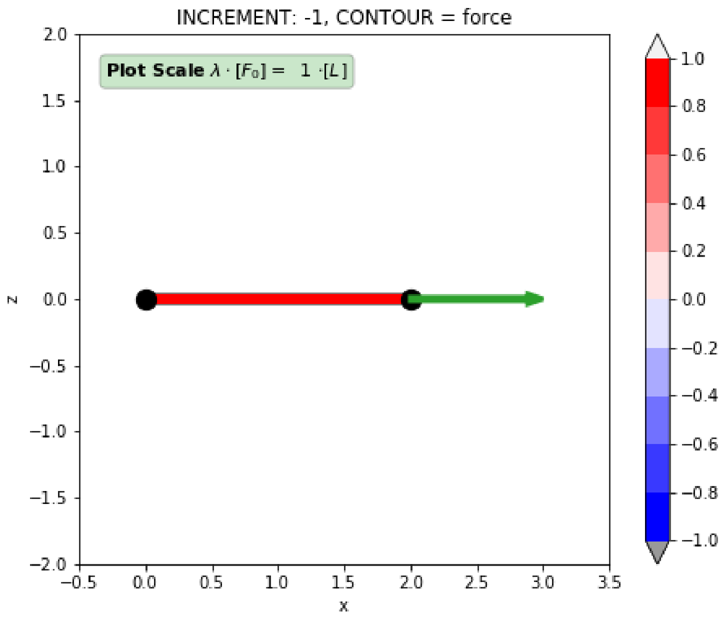


Fig. 2.2: The deformed model with the current **External Force** vector acting on **Node** 2.

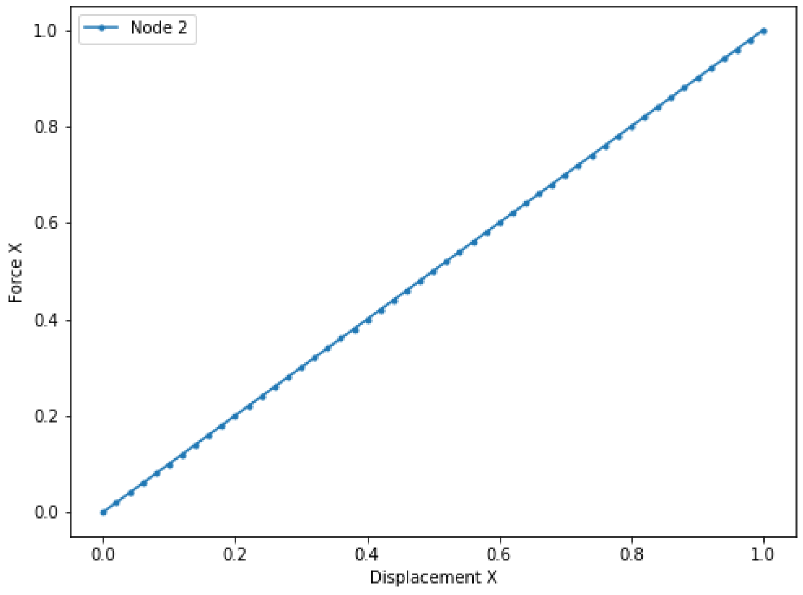


Fig. 2.3: The load-displacement curve for all increments at **Node** 2.

It could also be helpful to show the animated deformation process within a simple GIF file (options should be self-explaining):

```
# show results
M.plot_movie(config=['deformed'],
             view='xz',
             contour='force',
             lim_scale=(-0.5,3.5,-2,2),
             force_scale=1.0,
             cbar_limits=[-1,1])
```

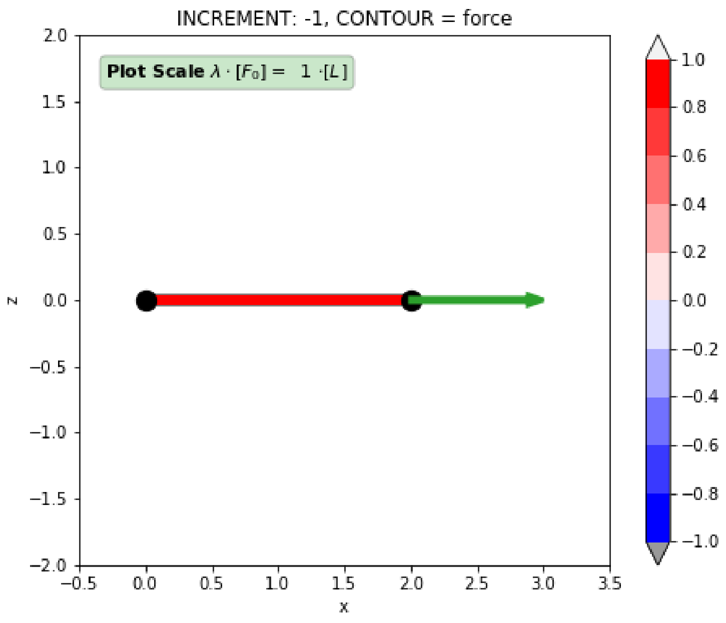


Fig. 2.4: An animation of the deformation process.

Important Note: A **LOT** of assumptions are made to run this model without specifying barely any parameter. Most important ones are incremental displacement values, incremental LPF value and the amount of increments to be solved. These critical parameters are responsible if the model solution will converge or not!

CHAPTER

THREE

THEORY GUIDE

3.1 Truss Element

This section describes the **Kinematics** and **Constitution** of a Truss element `trusspy.elements.element_definition.truss`.

3.1.1 Kinematics

For a truss element the stretch may be calculated as

$$\Lambda = \frac{l}{L} = \sqrt{1 + 2 \left(\frac{\boldsymbol{\Delta X}}{L}\right)^T \left(\frac{\boldsymbol{\Delta U}}{L}\right) + \left(\frac{\boldsymbol{\Delta U}}{L}\right)^T \left(\frac{\boldsymbol{\Delta U}}{L}\right)} \tag{3.1}$$

which follows from

$$\begin{aligned} l^2 &= \boldsymbol{\Delta x}^T \boldsymbol{\Delta x} \\ L^2 &= \boldsymbol{\Delta X}^T \boldsymbol{\Delta X} \end{aligned} \tag{3.2}$$

and enables the Biot strain measure:

$$E_{11} = \Lambda - 1 \tag{3.3}$$

3.1.2 Constitution

The normal force of a truss depends directly on the geometric exactly defined strain measure E_{11}. For the general case of a nonlinear material behaviour the normal force is defined as

$$N = S_{11}(E_{11})\ A + N_0 \tag{3.4}$$

and the derivative

$$\frac{\partial N}{\partial E_{11}} = \frac{\partial S_{11}(E_{11})}{\partial E_{11}}\ A \tag{3.5}$$

For the case of a linear elastic material this reduces to

$$\begin{aligned} S_{11}(E_{11}) &= E\ E_{11} \\ N &= EA\ E_{11} + N_0 \\ \frac{\partial N}{\partial E_{11}} &= EA \end{aligned} \tag{3.6}$$

3.1.3 Kinetics

The nodal (nonlinear) internal force vector may be expressed as a function of the elemental force N and the deformed unit vector $\boldsymbol{n}$. The indices *A* and *E* represent the begin node *A* and end node *E*.

$$\boldsymbol{r} = \begin{bmatrix} \boldsymbol{r}_A \\ \boldsymbol{r}_E \end{bmatrix} = N \begin{bmatrix} -\boldsymbol{n} \\ \boldsymbol{n} \end{bmatrix} \tag{3.7}$$

3.1.4 Stiffness Matrix

The elemental stiffness matrix for a truss can be divided in four block matrices of the same components but with different signs.

$$\boldsymbol{K}_{6,6} = \begin{bmatrix} \boldsymbol{K}_{AA} & \boldsymbol{K}_{AE} \\ \boldsymbol{K}_{EA} & \boldsymbol{K}_{EE} \end{bmatrix} \begin{bmatrix} \boldsymbol{K}_{EE} & -\boldsymbol{K}_{EE} \\ -\boldsymbol{K}_{EE} & \boldsymbol{K}_{EE} \end{bmatrix} \tag{3.8}$$

Whereas a change in theinternal force vector at the end node *E* w.r.t. a small change of the displacements at node *E* is defined as the tangent stiffnes *EE*.

$$\begin{aligned} \boldsymbol{K}_{EE} &= \frac{\partial \boldsymbol{r}_E}{\partial \boldsymbol{U}_E} \\ \boldsymbol{K}_{EE} &= \frac{A}{L} \frac{\partial S_{11}(E_{11})}{\partial E_{11}} \boldsymbol{n} \otimes \boldsymbol{n} + \frac{N}{l} (\boldsymbol{1} - \boldsymbol{n} \otimes \boldsymbol{n}) \end{aligned} \tag{3.9}$$

Continue to *Global Assembly*.

3.2 Global Assembly

The element force contributions have to be assembled into a system equilibrium.

$$\boldsymbol{r}_{System} = \bigcup_{k=1}^{n_{el}} \boldsymbol{r}_k \tag{3.10}$$

The same also applies for the elemental tangent stiffness.

$$\boldsymbol{K}_{T,System} = \bigcup_{k=1}^{n_{el}} \boldsymbol{K}_{T,k} \tag{3.11}$$

3.3 Equilibrium

This section covers the description of global system equilibrium equations and their linearization. For local (element-based) contributions see TrussPy's API.

3.3.1 System Equilibrium

The system vector of nodal equilibrium equations is formulated as the component-wise sum for each DOF of negative internal forces $-\boldsymbol{r}(\boldsymbol{U})$ and prescribed nodal external forces $\boldsymbol{f} = \lambda\, \boldsymbol{f}_0$.

$$\boldsymbol{g}(\boldsymbol{U}, \lambda) = -\boldsymbol{r}(\boldsymbol{U}) + \lambda\, \boldsymbol{f}_0 \leq \varepsilon_{tol} \tag{3.12}$$

with

Table 3.1: Description of system parameters

Symbol	Description
$\boldsymbol{g}$	system vector of (nonlinear) nodal equilibrium equations
$\boldsymbol{r}$	system vector of nodal internal forces
$\boldsymbol{U}$	system vector of nodal displacements
λ	load-proportionality-factor (LPF)
$\boldsymbol{f}_0$	prescribed external nodal load vector (reference loadcase)
ε_{tol}	tolerance vector for allowable numerical violation of the equilibrum state

3.3.2 Linearized System Equlibrium

The linearized equilibrium equations for a given equlibrium state $\boldsymbol{g}(\boldsymbol{U},\lambda)$ are approximated with the help of a 1st order taylor - expansion:

$$\begin{aligned}
\boldsymbol{g}(\boldsymbol{U}+\boldsymbol{dU},\lambda+d\lambda) &= -\boldsymbol{r}(\boldsymbol{U}+\boldsymbol{dU}) && +(\lambda+d\lambda)\boldsymbol{f}_0 && && (3.13)\\
&\approx -\left(\boldsymbol{r}(\boldsymbol{U})+\frac{\partial \boldsymbol{r}}{\partial \boldsymbol{U}}\boldsymbol{dU}\right) && +(\lambda+d\lambda)\boldsymbol{f}_0 && = \boldsymbol{0} && (3.14)\\
&\approx -\boldsymbol{r}(\boldsymbol{U})+\lambda\boldsymbol{f}_0 && +\left(-\frac{\partial \boldsymbol{r}}{\partial \boldsymbol{U}}\boldsymbol{dU}\right)+d\lambda\,\boldsymbol{f}_0 && = \boldsymbol{0} && (3.15)\\
&\approx \quad \boldsymbol{g}(\boldsymbol{U},\lambda) && -\boldsymbol{K}_T(\boldsymbol{U})\,\boldsymbol{dU}+d\lambda\,\boldsymbol{f}_0 && = \boldsymbol{0} && (3.16)
\end{aligned}$$

3.3.3 Newton-Rhapson Iteration and Update of Displacements

The linearized equilibrium equations may also be expressed as a simple linear equation system. The right hand side of this equation enforces a self-correction over incremental updates of the displacement vector.

$$\boldsymbol{K}_T(\boldsymbol{U})-d\lambda\boldsymbol{f}_0=\boldsymbol{g}(\boldsymbol{U},\lambda) \tag{3.17}$$

with

$$\begin{aligned}\boldsymbol{U} &\leftarrow (\boldsymbol{U}+d\boldsymbol{U})\\ \lambda &\leftarrow (\lambda+d\lambda)\end{aligned} \tag{3.18}$$

3.4 Modified system equations

In order to solve the equilibrium equations a equation system with *n+1* DOF's will be formulated. The extra degree of freedom arises from the external load - scale parameter, also denoted as load-proportionality-factor λ, which will be appended to the system vector $\boldsymbol{U}$. The transformation will be shown for both the equilibrium and tangent stiffness.

$$\boldsymbol{V}=\begin{bmatrix}\boldsymbol{U}\\ \lambda\end{bmatrix} \tag{3.19}$$

3.4.1 Modified equilibrium

The modified equilibrium is extended by one extra equation, which is also referred to as **control equation**.

$$\boldsymbol{g}_{mod}=\begin{bmatrix}\boldsymbol{g}(\boldsymbol{V})\\ g_{control}(\boldsymbol{V})\end{bmatrix}=\begin{bmatrix}\boldsymbol{g}(\boldsymbol{U},\lambda)\\ g_{control}(\boldsymbol{U},\lambda)\end{bmatrix} \tag{3.20}$$

This extra control equation is formulated as a linear constraint to limit the control component *j*.

$$g_{control} = V_j - V_{j,max} \tag{3.21}$$

With this definition the modified system equilibrium becomes

$$\boldsymbol{g}_{mod} = \begin{bmatrix} \boldsymbol{g}(\boldsymbol{U},\lambda) \\ g_{control}(\boldsymbol{U},\lambda) \end{bmatrix} = \begin{bmatrix} -\boldsymbol{r}(\boldsymbol{U}) + \lambda\, \boldsymbol{f}_0 \\ V_j - V_{j,max} \end{bmatrix} \leq \boldsymbol{\varepsilon}_{tol} \tag{3.22}$$

3.4.2 Modified tangent stiffness

The modified tangent stiffness $\boldsymbol{K}_{T,mod}$ is now calculated w.r.t. the modified system vector $\boldsymbol{V}$.

$$\begin{gathered} \boldsymbol{K}_{T,mod} = \begin{bmatrix} \frac{\partial \boldsymbol{r}}{\partial \boldsymbol{U}} & -\boldsymbol{f}_0 \\ \frac{\partial g_{control}}{\partial \boldsymbol{U}} & \frac{\partial g_{control}}{\partial \lambda} \end{bmatrix} \\ \boldsymbol{K}_{T,mod} = \begin{bmatrix} \boldsymbol{K}_T & -\boldsymbol{f}_0 \\ \boldsymbol{q}_{c,\boldsymbol{U}} & \boldsymbol{q}_{c,\lambda} \end{bmatrix} \end{gathered} \tag{3.23}$$

with the partial derivatives of the control equation:

$$\begin{aligned} \boldsymbol{q}_{c,\boldsymbol{U}} &= \frac{\partial g_{control}(\boldsymbol{U},\lambda)}{\partial \boldsymbol{U}} \\ \boldsymbol{q}_{c,\lambda} &= \frac{\partial g_{control}(\boldsymbol{U},\lambda)}{\partial \lambda} \end{aligned} \tag{3.24}$$

To illustrate the components of the partial derivatives of the control equation we formulate the first order derivative of the control equation for small changes in $\boldsymbol{dU}$ and $d\lambda$ at a given state $(\boldsymbol{U},\lambda)$.

$$\boldsymbol{g}_{control}(\boldsymbol{U}+\boldsymbol{dU},\lambda+d\lambda) = g_{control}(\boldsymbol{U},\lambda) + dg_{control}(\boldsymbol{U},\lambda) = V_j - V_{j,max} + dV_j - dV_{j,max} = 0 \tag{3.25}$$

with

$$dV_j = \frac{\partial V_j}{\partial \boldsymbol{U}} \quad \boldsymbol{dU} + \frac{\partial V_j}{\partial \lambda} \quad d\lambda \tag{3.26}$$

$$dV_j = \boldsymbol{q}_{c,\boldsymbol{U}} \quad \boldsymbol{dU} + q_{c,\lambda} \quad d\lambda \tag{3.27}$$

The combined partial derivate of the control equation may be expressed as:

$$\boldsymbol{q}_c = \begin{bmatrix} \boldsymbol{q}_{c,\boldsymbol{U}} & q_{c,\lambda} \end{bmatrix} \tag{3.28}$$

and is a row vector with a length of *nDOF+1* entries filled with zeros, except for the *j*-th entry (control component), which is one. Let's assume we have a system of 6 DOF and the control component is identified to *j=4* then $\boldsymbol{q}_c$ looks like

$$\boldsymbol{q}_c = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \tag{3.29}$$

3.4.3 Summary

The final equation system for the *Path-Following Algorithm* may now be formulated as

$$\boldsymbol{K}_{T,mod}\, \boldsymbol{dV} = -\boldsymbol{g}_{mod}(\boldsymbol{V}) \tag{3.30}$$

and in detail

$$\begin{bmatrix} \boldsymbol{K}_T & -\boldsymbol{f}_0 \\ \boldsymbol{q}_{c,\boldsymbol{U}} & \boldsymbol{q}_{c,\lambda} \end{bmatrix} \begin{bmatrix} \boldsymbol{dU} \\ d\lambda \end{bmatrix} = \begin{bmatrix} -\boldsymbol{g}(\boldsymbol{U}, \lambda) \\ g_{control}(\boldsymbol{U}, \lambda) \end{bmatrix} \tag{3.31}$$

The control component j is defined as the index of the biggest component of the incremental system vector $\boldsymbol{dV}$. This component remains fixed during all newton-iterations inside an increment. To initialize the control component j the linear equation system is solved. Last but not least the sign of j is estimated with the help of the determinant of the stiffness matrix.

$$\begin{aligned} \boldsymbol{K}_T\, \boldsymbol{dU} &= d\lambda_{max} \boldsymbol{f}_0 \\ \rightarrow \boldsymbol{dU} &= \dots \end{aligned} \tag{3.32}$$

$$j = \text{index}[\max(\boldsymbol{dU})]\ \text{sign}(\det \boldsymbol{K}_T) \tag{3.33}$$

3.5 Path-Following Algorithm

This section gives an overview of the algorithm for the path-following `trusspy.solvers.tpsolver.pathfollow()` of the equilibrium curve. The algorithm is called once for each step, which contains solutions for a number of increments. During an increment several Newton-Rhapson (NR) iterations are performed to obtain a fullfilled equilibrium. For an easier understanding this description uses a mixed Python and pseudo-code style with omitting namespaces, etc.

```
+ step 1                # <-- path-following algorithm is called the first time
  - increment 1
      * iteration 1
      * iteration 2
      * iteration 3
      * ...
      * iteration n
  - incrmeent 2
      * iteration 1
      * iteration 2
      * iteration 3
      * ...
      * iteration n

+ step 2                # <-- path-following algorithm is called the 2-nd time
+ ...
+ step s                # <-- path-following algorithm is called the s-th time
```

3.5.1 Initialization

Some initializations have to be made before the calculation loops may start. First, there is a init section for the *step* followed by an initialization for each *increment*. If the user doesn't freeze the control component for the whole analysis, the control component is initialized to the incremental load-proportionality-factor *dlpf* which is the last component in *x*.

```python
if j==None:
    j = len(x)
```

If we use the automatic incremental stepwidth control (has **absolutely nothing** to do with the *global* step) a internal variable is pre-defined for later use. This will ensure a stepwidth increase is only performed if the previous *b* increments were successfully solved within the specified number of iterations and without any recycles.

```
b = 3
```

3.5.2 Increment Loop

For each increment several recycles *z* are allowed. The amount of recycle loops is controlled with the *cycl* parameter and defaults to *cycl=4*. The system vector *x* is copied to *x0* so we can go back to the initial system vector if a recycle will be necessary.

3.5.2.1 Initialization - Determination of the control component *j*

A success flag for the pre - NR-iteration is initiated to *False*. The amount of NR-iterations is limited to 1, as we are interested only in the linear solution of the equilibrium equations. Note that it is important to know the direction of the incremental LPF to determine the direction for the upcoming equilibrium state. Is it always positive? No! So we'll use the determinant of the global stiffness matrix at the given state *x* (beginning of the increment) to get the direction information. As we are not yet interested in the control equation - or, with other - we don't know it yet, we just omit the last dimensions by slicing the matrix *dfdx(x'*args)[:-1,:-1]*.

```
sgn = sign( det( dfdx(x)[:-1,:-1] ) )
j   = len(x) * sgn
```

The absolute scalar-valued control parameter for the pre - NR-iteration is further determined as *xmax = x[j] + dxmax[j]*. As Python is indexing arrays with starting *0* we have to take care to get the right value and subtract j by 1 in the component selection. The solution for the pre-identification is then performed with a single NR-iteration.

```
xmax = x[abs(j)-1] + dxmax[abs(j)-1] * sign(j)
solution_pre = newton(f,dfdx,x,nfev_pre,ftol,xtol,*args)

x = solution_pre[0]
nr_success_pre = sultion_pre[1]

Dx = x-x0
```

As we are interested in the incremental *Dx* and not the absolute *x* vector we build the difference. (OK, this is unnecessary but easier to code). Now we look for the biggest componenet in Dx and it's sign. Both informations are stored inside the single variable *j*. Again, taking care of Python's 0-indexing.

```
j = (index(Dx,max)+1) * sign(Dx,max)
```

What we got so far is:

- determination of the control component *j* and
- corresponding solution *x*.

If the solution did converge in this single increment, then the increment is finished. This is indicated by the success flag *nr_success_pre*. Increment results are copied to `trusspy.core.analysis.Analysis` - go to next increment.

3.5.2.2 Recycle Loop (Nonlinear solution process)

If the pre NR-iteration did not converge (*nr_success_pre = False*) a recycle loop is started. For later usage the current control component *j* is copied to *j0*. Also, we reset the system vector *x* to *x0*. Before the NR-iterations `trusspy.solvers.zerosearch.newton()` are called the allowed control component *xmax* is updated with the current values of *j* and *dxmax*.

```
while nr_success_pre is not True:

    x = x0.copy()
    j0 = j

    xmax = x[abs(j)-1] + dxmax[abs(j)-1] *np.sign(j)
    x,nr_success,n,f_norm,x_norm = newton(f,dfdx,x,nfev,ftol,xtol,verbose,*args)

    Dx = x-x0
```

The total incremental system vector is then obtained by the difference between the end and beginning state of the increment (or, alternatively as a sum over all virtual NR-iteration solutions *Dx = sum(dx)*). If the solution has met our tolarences we stop the recycle loop, call the equilibrium function *f(x)* once again with a flag to store internal variables inside the material definition, append the analysis `trusspy.core.analysis.Analysis` to `trusspy.handlers.handler_result.ResultHandler` and proceed to the next increment. Otherwise, when the used and final control component differ each other, once again recycle is performed. Therefore the recycle variable is increased *z = z+1*. The final control component will be used for the recycle of the increment. This procedure is done not more than *cycl* times as long as the control component does not change anymore and the required tolerances are met. Otherwise the calculation stops.

CHAPTER

FOUR

CODE STRUCTURE

TrussPy's code structure is organized in an object-oriented approach, hence it uses classes, attributes, methods and functions. The main idea is to seperate the code in single files (in Python called (sub-)modules) which are grouped in subfolders (again, so-called (sub-)packages). A basic overview of the essential classes is given in the following figure. It all starts with the **Model** class - it is basically the main container in TrussPy. On the next meta-level so-called **Handler** classes are managing the low-level **Core** classes of TrussPy: **Node**, **Element**, **Boundary**, **Analysis** and **Result**.

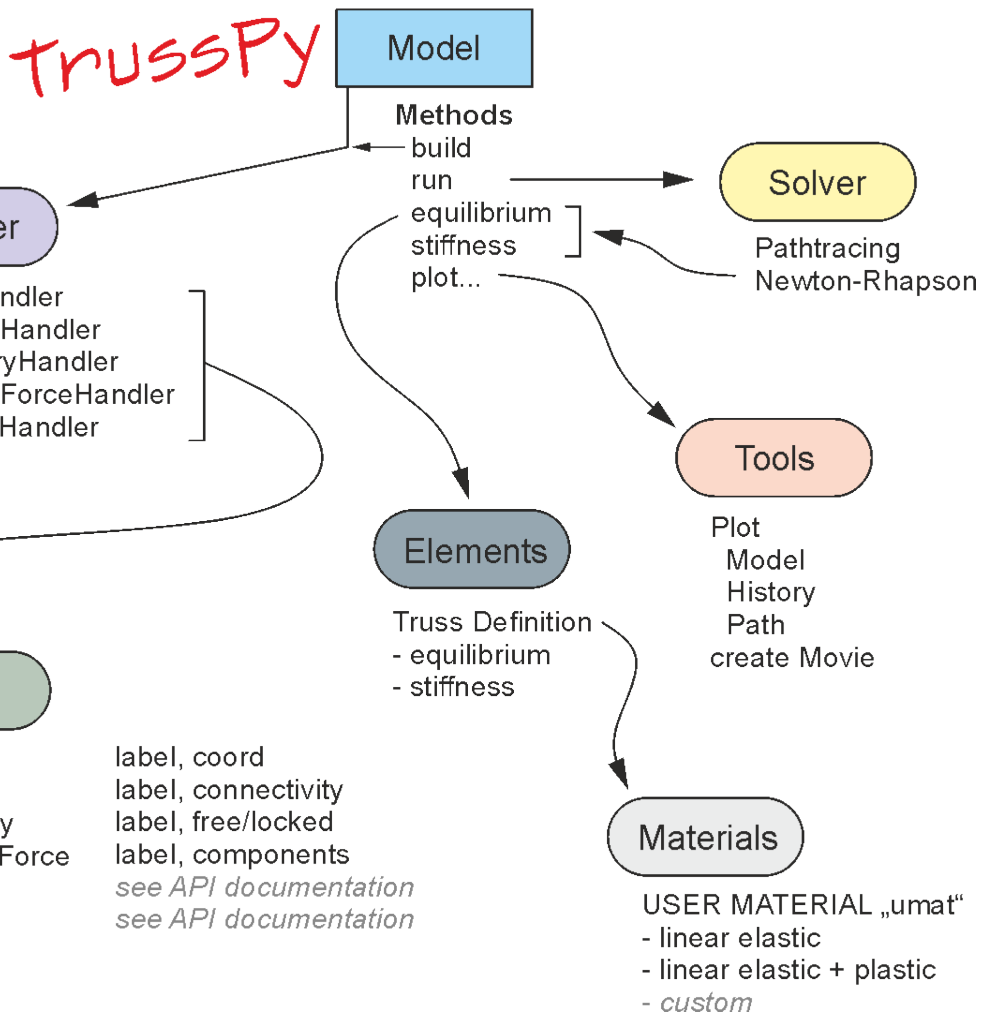


Fig. 4.1: TrussPy's essential code structure

CHAPTER

FIVE

EXAMPLES

5.1 Example 101

Getting Started

5.2 Example 102 “Zweibock”

This model contains two trusses and three nodes. The left and right end nodes are fixed, whereas on the top middle node a reference force is applied in negative z-direction.

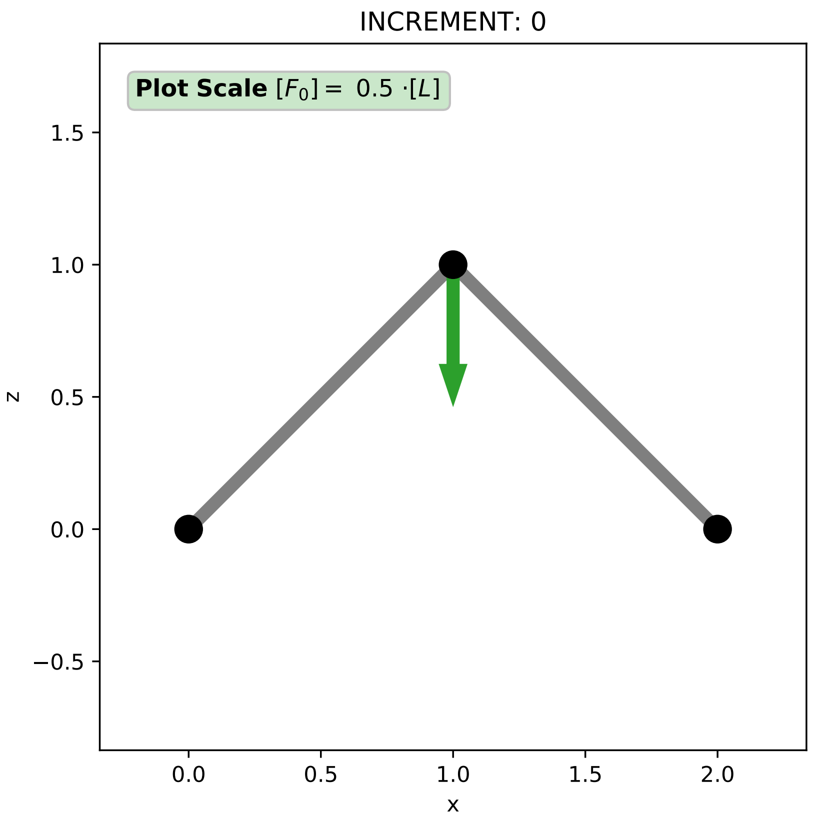


Fig. 5.1: The undeformed configuration of the Model

The model may be generated with the following code and is shown in the undeformed state.

```python
import trusspy as tp

M = tp.Model(logfile=False)

with M.Nodes as MN:
    MN.add_node( 1, coord=( 0, 0, 0))
    MN.add_node( 2, coord=( 1, 0, 1))
    MN.add_node( 3, coord=( 2, 0, 0))

with M.Elements as ME:
    ME.add_element( 1, conn=(1,2), gprop=[1] )
    ME.add_element( 2 ,conn=(2,3), gprop=[1] )

    E = 1      # elastic modulus
    ME.assign_material( 'all', [E])
```

```
with M.Boundaries as MB:
    MB.add_bound_U( 1,  (0,0,0) )
    MB.add_bound_U( 2,  (0,0,1) )
    MB.add_bound_U( 3,  (0,0,0) )

with M.ExtForces as MF:
    MF.add_force( 2,  ( 0,  0,-1) )
```

The calculation of the deformation process is started by calling the *build()* and *run()* methods.

```
M.build()
M.run()

M.plot_model(config=['undeformed'],inc=0)
M.plot_model(config=['undeformed','deformed'],
             view='xz',
             contour='force',
             lim_scale=(-1,3,-2.5,1.5)
             )
fig, ax = M.plot_history(nodes=[2,2],X='Displacement Z',Y='LPF')
```

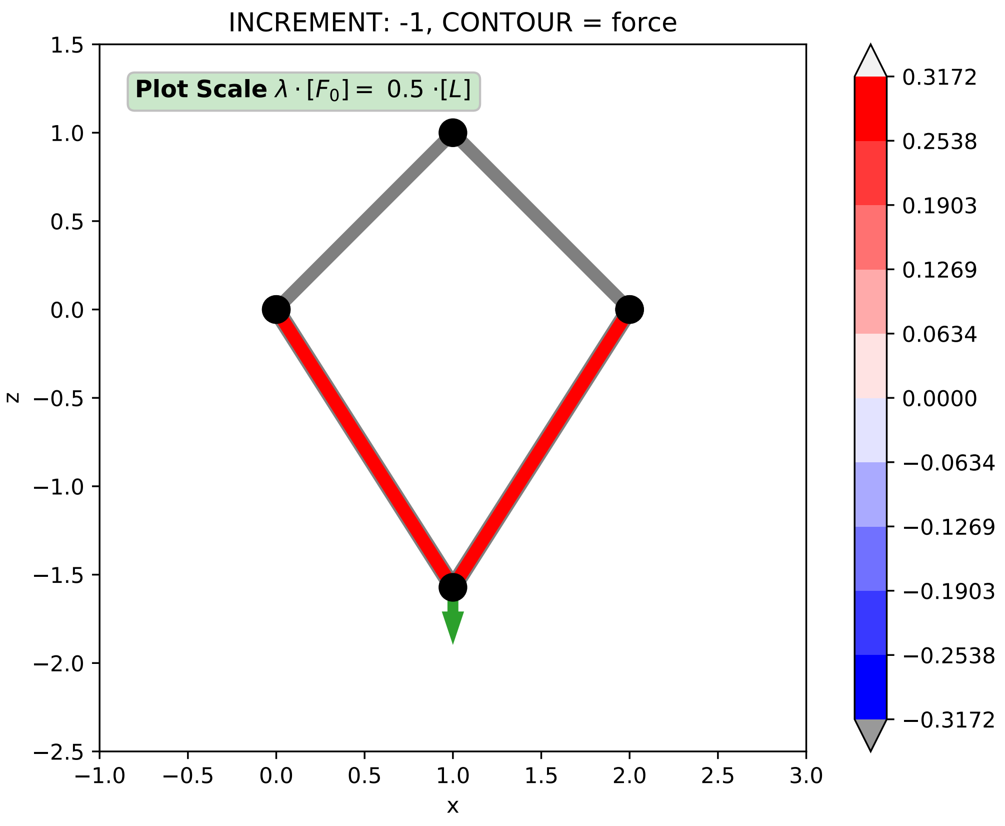


Fig. 5.2: The deformed configuration of the Model

Let's re-run the model with a nonlinear material (plasticity with isotropic hardening). The isotropic yield-modulus *K* and the initial yield stress have to be specified. Additionally, the material has to be changed to *mtype=2*. Results are plotted for Node 2 as a History Plot of z-displacement vs. LPF.

```
with M.Elements as ME:
    E  = 1.0  # elastic modulus
    K  = 0.1  # plastic modulus
    Sy = 0.1  # initial yield stress

    ME.assign_mtype(    'all',  1   )
    ME.assign_material( 'all', [E, K, Sy])

M.build()
M.run()

fig, ax = M.plot_history(nodes=[2,2],X='Displacement Z',Y='LPF',fig=fig,ax=ax)
ax.legend(['Node 2: linear elastic','Node 2: elastic-plastic (isotropic hardening)'])
```

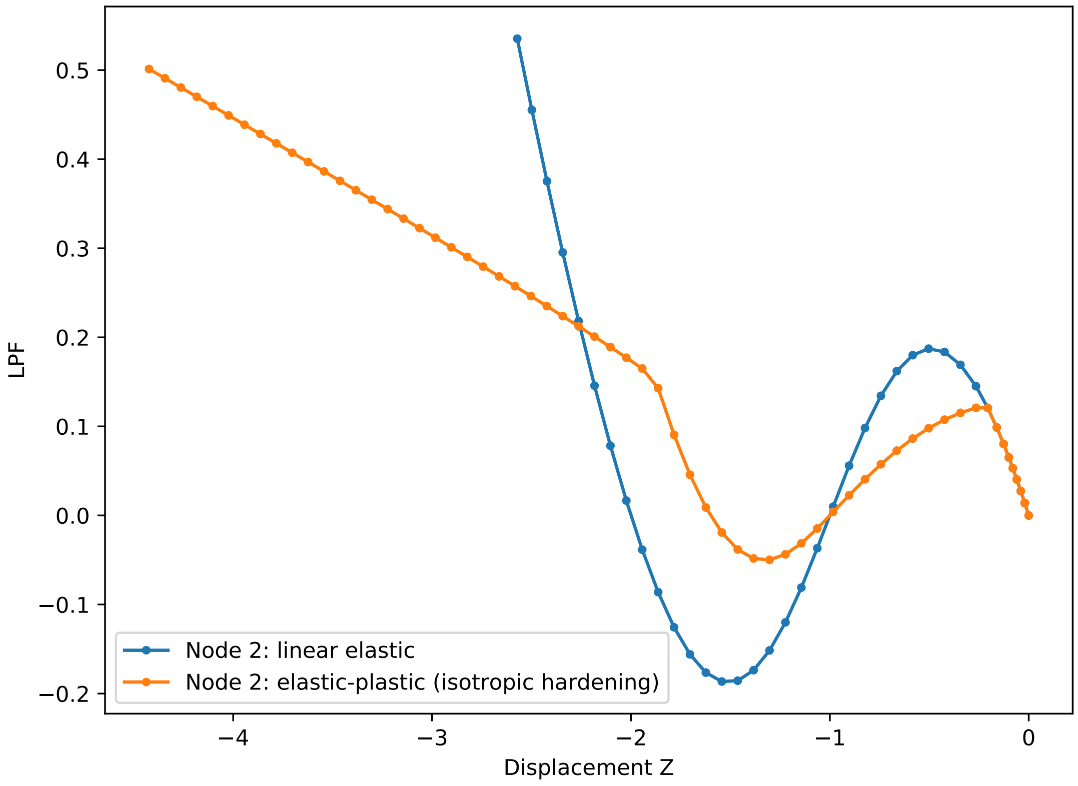


Fig. 5.3: Displacement Z vs. LPF for Node 2

5.3 Example 103 “Zweibock” (with imperfection)

This model contains two trusses and three nodes. The left and right end nodes are fixed, whereas on the top middle node a reference force is applied in negative z-direction. A second version contains 2 DOF (x,z at node 2) with a geometric imperfection at node 2.

5.3.1 No imperfection

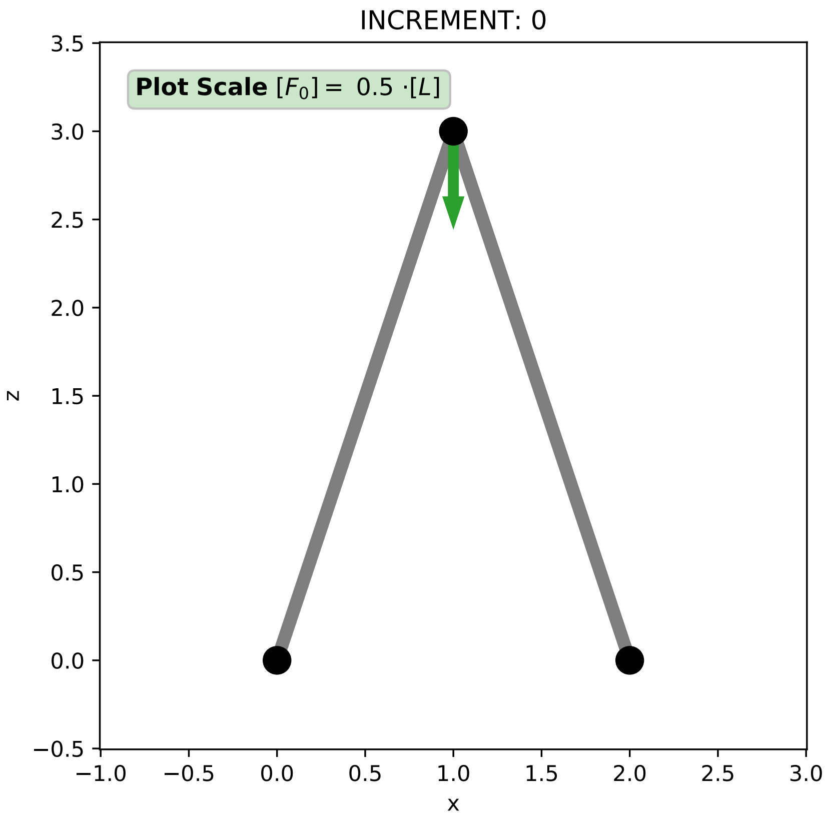


Fig. 5.4: The undeformed configuration of the Model

The model may be generated with the following code and is shown in the undeformed state.

```
import trusspy as tp

M1 = tp.Model(logfile=False)

with M1.Nodes as MN:
    MN.add_node( 1, coord=( 0, 0, 0))
    MN.add_node( 2, coord=( 1, 0, 3))
    MN.add_node( 3, coord=( 2, 0, 0))

with M1.Elements as ME:
    ME.add_element( 1, conn=(1,2), gprop=[1] )
    ME.add_element( 2 ,conn=(2,3), gprop=[1] )

    E = 1      # elastic modulus
    ME.assign_material( 'all', [E])

with M1.Boundaries as MB:
    MB.add_bound_U( 1,  (0,0,0) )
    MB.add_bound_U( 2,  (0,0,1) )
    MB.add_bound_U( 3,  (0,0,0) )

with M1.ExtForces as MF:
    MF.add_force( 2,  ( 0,  0,-1) )

M1.Settings.incs = 150
M1.Settings.du = 0.01
M1.Settings.dlpf = 0.01
M1.Settings.xlimit = (1,10)
M1.Settings.dlpf
M1.Settings.stepcontrol = True
M1.Settings.maxfac = 10
```

The calculation of the deformation process is started by calling the *build()* and *run()* methods.

```
M1.build()
M1.run()

fig, ax = M1.plot_model(config=['undeformed'],inc=0)
fig, ax = M1.plot_model(config=['undeformed','deformed'],
                       view='xz',
                       contour='force',
                       force_scale=2,
                       inc=20
                      )

fig1, ax1 = M1.plot_history(nodes=[2,2],X='Displacement X',Y='Displacement Z')
fig2, ax2 = M1.plot_history(nodes=[2,2],X='Displacement Z',Y='LPF')
```

5.3.2 Geometric imperfection

Let's re-run the model with a geometric imperfection at node 2 (misalignment *dx=0.1*). Results are plotted for Node 2 as a History Plot of z-displacement vs. LPF and x-displacement vs. z-displacement.

```
M2 = tp.Model(logfile=False)

with M2.Nodes as MN:
    MN.add_node( 1, coord=( 0,  0,  0))
    MN.add_node( 2, coord=( 1.1,  0,  3))
    MN.add_node( 3, coord=( 2,  0,  0))

with M2.Elements as ME:
    ME.add_element( 1, conn=(1,2), gprop=[1] )
    ME.add_element( 2 ,conn=(2,3), gprop=[1] )

    E = 1      # elastic modulus
    ME.assign_material( 'all', [E])
```

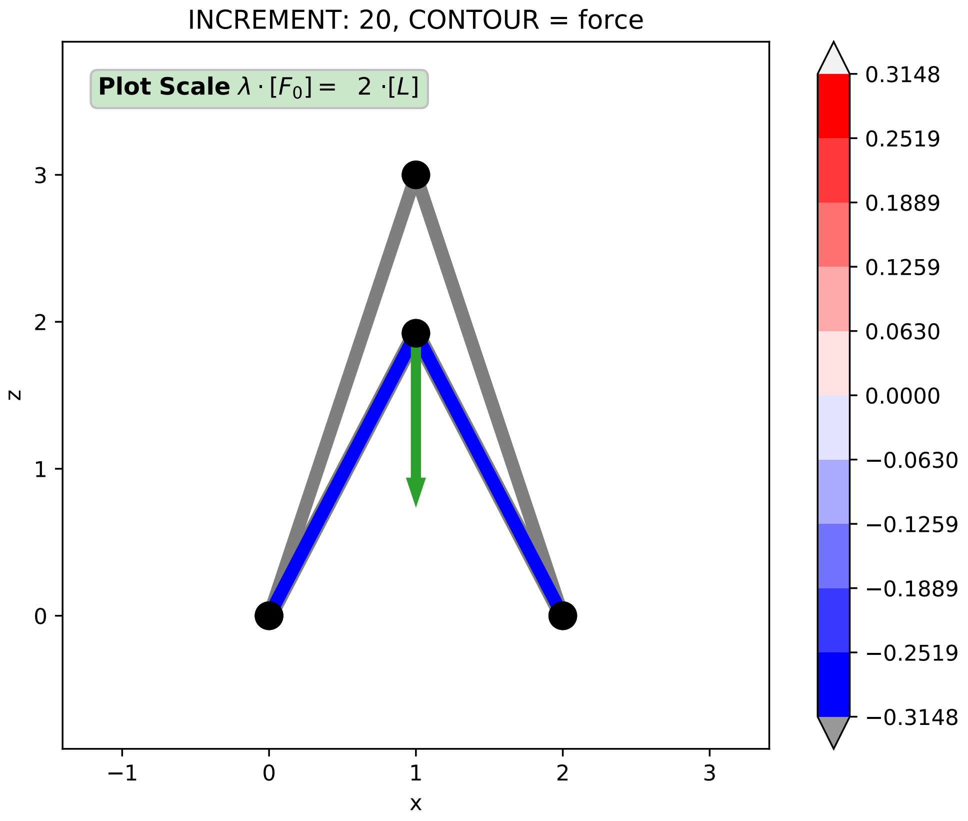


Fig. 5.5: The deformed configuration of the Model

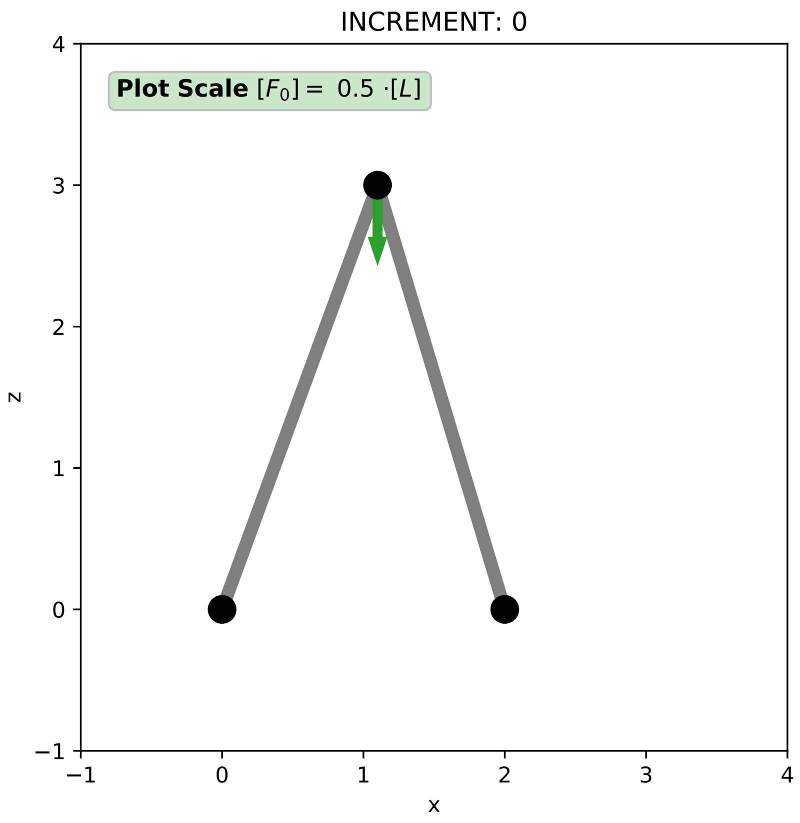


Fig. 5.6: The deformed configuration of the Model with geometric imperfection

```
with M2.Boundaries as MB:
    MB.add_bound_U( 1, (0,0,0) )
    MB.add_bound_U( 2, (1,0,1) )
    MB.add_bound_U( 3, (0,0,0) )

with M2.ExtForces as MF:
    MF.add_force( 2, ( 0, 0,-1) )

M2.Settings.incs = 150
M2.Settings.du = 0.01
M2.Settings.dlpf = 0.01
M2.Settings.xlimit = (2,10)
M2.Settings.dlpf
M2.Settings.stepcontrol = True
M2.Settings.maxfac = 10

M2.build()
M2.run()

fig, ax = M2.plot_model(config=['undeformed'],lim_scale=(-1,4,-1,4),inc=0)
fig, ax = M2.plot_model(config=['undeformed','deformed'],
                       view='xz',
                       contour='force',
                       lim_scale=(-1,4,-1,4),
                       force_scale=10,
                       inc=40
                      )

fig2, ax2 = M2.plot_history(nodes=[2,2],X='Displacement Z',Y='LPF',fig=fig2,ax=ax2)
ax2.legend(['Node 2: basic model (nDOF=1)','Node 2: imperfection (nDOF=2)'])

fig1, ax1 = M2.plot_history(nodes=[2,2],X='Displacement X',Y='Displacement Z',
↪fig=fig1,ax=ax1)
ax1.legend(['Node 2: basic model (nDOF=1)','Node 2: imperfection (nDOF=2)'])
```

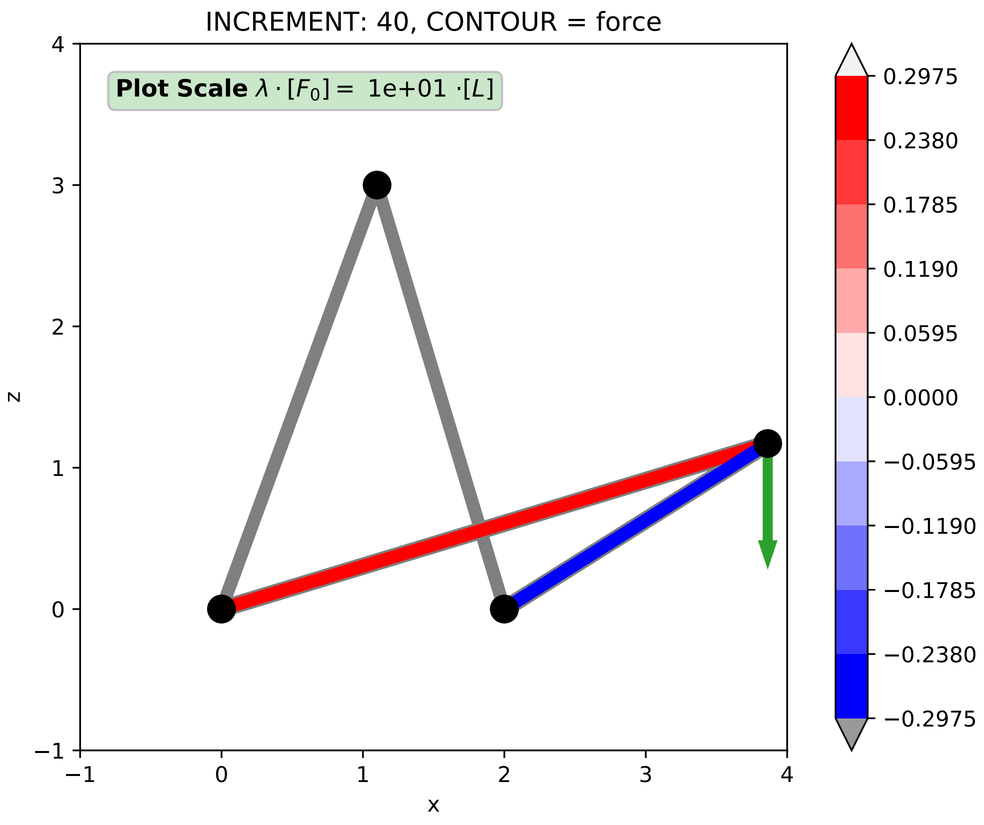


Fig. 5.7: The deformed configuration of the Model with geometric imperfection

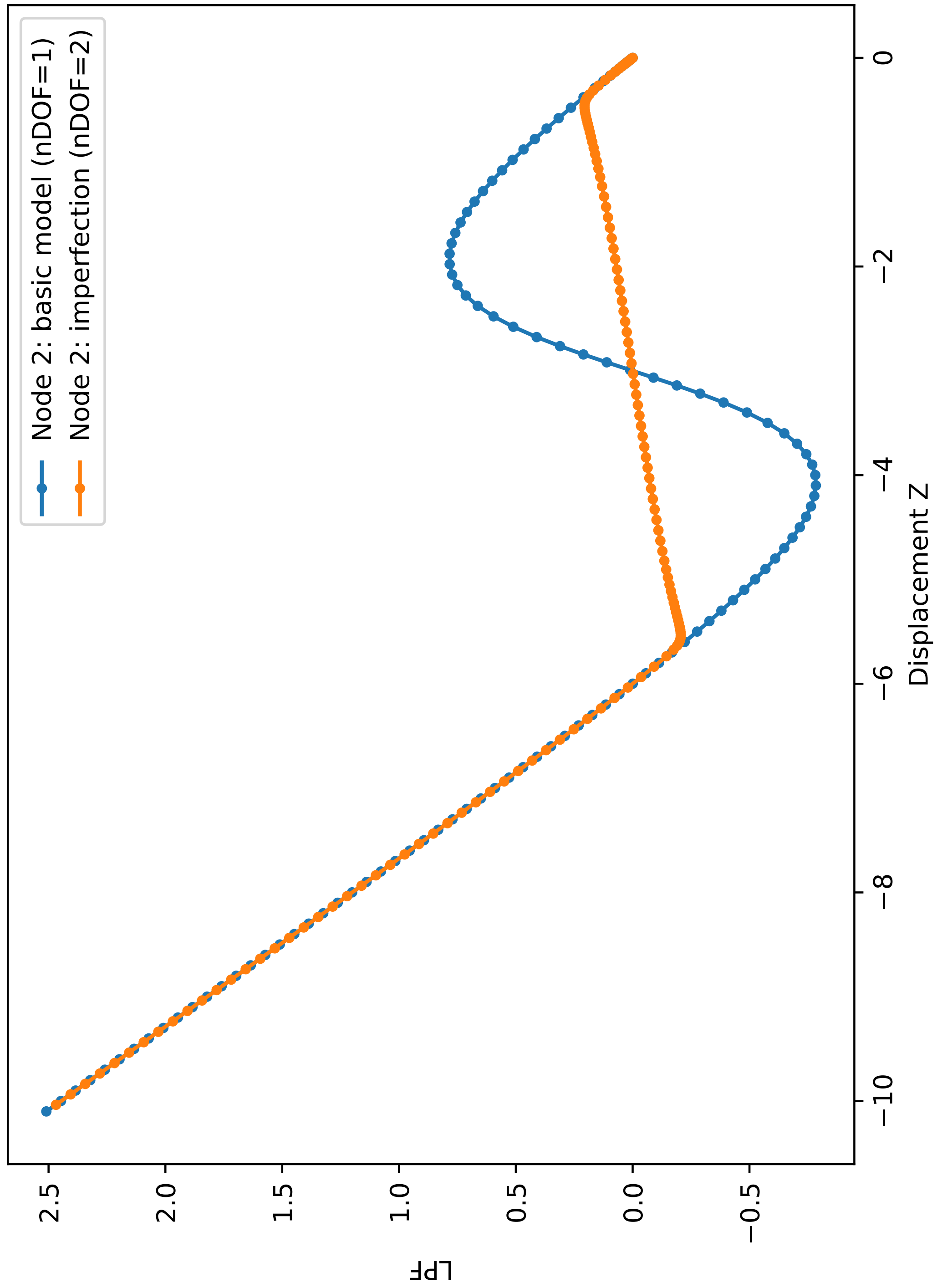


Fig. 5.8: Displacement Z vs. LPF for Node 2

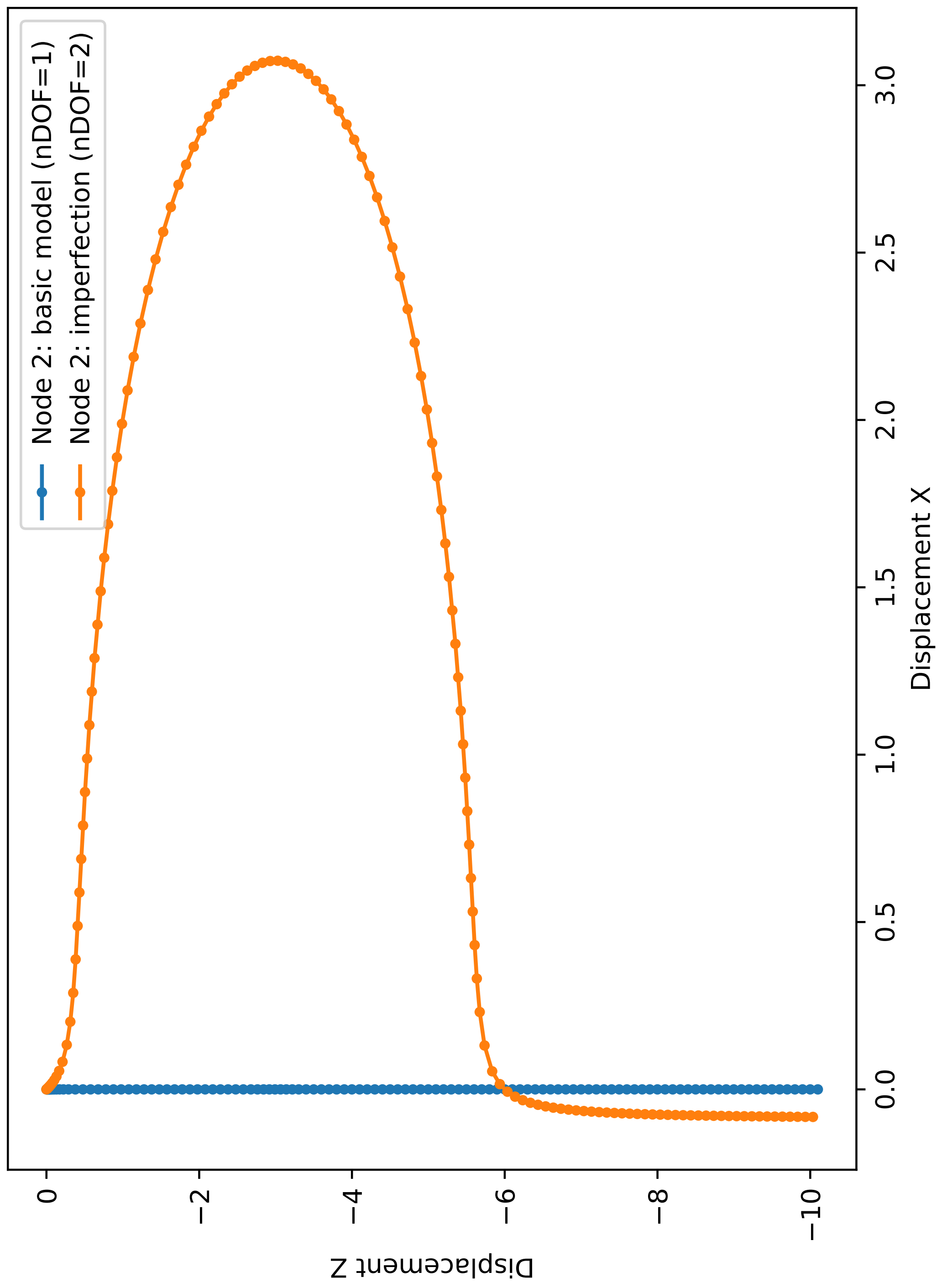
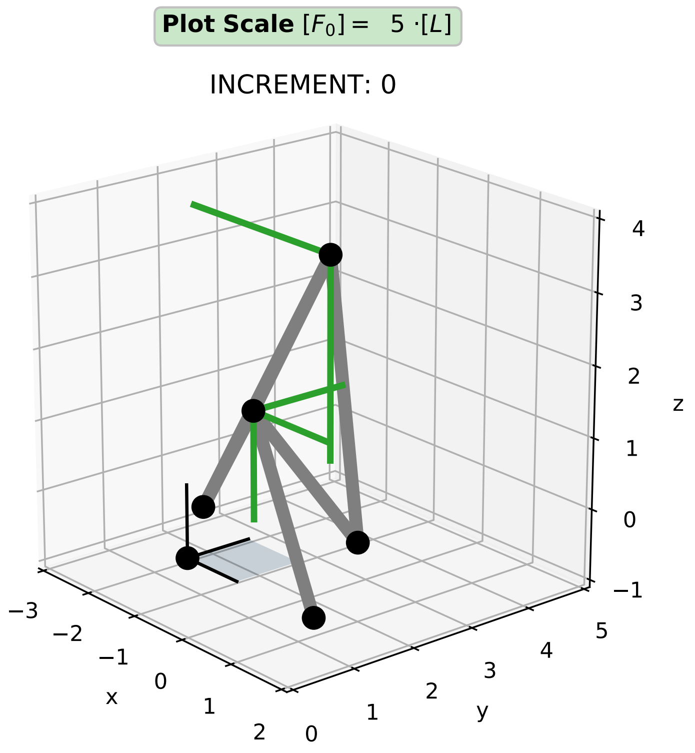


Fig. 5.9: Displacement X vs. Displacement Z for Node 2

5.4 Example NTA-A "3D Truss System"

This example describes a three-dimensional system of trusses with 5 Nodes and 6 Elements (in total 5 active DOF). Given to it's geometry strong geometric nonlinearities are to be excepted when the given reference load is applied. The model is shown in different views: a general 3D-view and views in XZ-,YZ- and XY-planes. The force vectors of the reference load are illustrated in green, where the force scale is denoted separated on each figure. Within the 3D-view the coordinate system is indicated as a black tripod with a big black dot at the origin point. The XY-plane is illustrated with a semi-transparent light blue fill.



5.4.1 Model creation

First we import trusspy with it's own namespace and create a Model object M. By explicitly enforcing a logfile creation all incremental solution informations are stored in a file *analysis.log* instead of printing it to the interactive shell window.

```
import trusspy as tp

# init model
M = tp.Model(logfile=True)
```

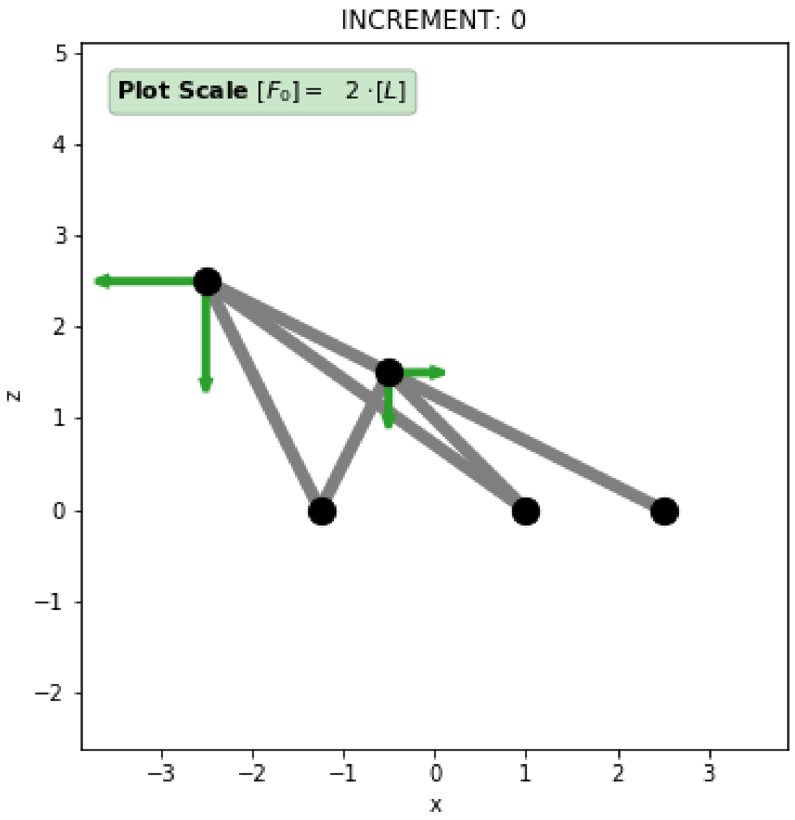
Now we create Nodes with coordinate triples and Elements with a list of node connectivities and both material and geometric properties. Both Nodes and Elements are identified with their label. To keep the code clean we'll use the *with* statement for the model creation process.

```
with M.Nodes as MN:
    MN.add_node( 1, coord=( 2.5 , 0    , 0   ))
    MN.add_node( 2, coord=(-1.25, 1.25, 0   ))
    MN.add_node( 3, coord=( 1    , 2    , 0   ))
    MN.add_node( 4, coord=(-0.5 , 1.5 , 1.5))
    MN.add_node( 5, coord=(-2.5 , 4.5 , 2.5))
```

```
element_type    = 1     # truss
material_type   = 1     # linear-elastic

youngs_modulus = 1.0

with M.Elements as ME:
    ME.add_element( 1, conn=(1,4), gprop=[0.75] )
    ME.add_element( 2 ,conn=(2,4), gprop=[1    ] )
    ME.add_element( 3, conn=(3,4), gprop=[0.5 ] )
    ME.add_element( 4, conn=(3,5), gprop=[0.75] )
    ME.add_element( 5, conn=(2,5), gprop=[1    ] )
    ME.add_element( 6, conn=(4,5), gprop=[1    ] )

    ME.assign_etype(    'all',    element_type    )
    ME.assign_mtype(    'all',   material_type    )
    ME.assign_material( 'all', [youngs_modulus] )
```

Beside Nodes and Elements we have to define Mechanical (U) Boundaries and External Forces. If a node does not contain Boundaries or External Forces the corresponding entries are added automatically by TrussPy.

```
with M.Boundaries as MB:
    MB.add_bound_U( 1,  (0,0,0) )
    MB.add_bound_U( 2,  (0,0,0) )
    MB.add_bound_U( 3,  (0,0,0) )
    MB.add_bound_U( 5,  (1,0,1) )

with M.ExtForces as MF:
    MF.add_force( 4,  ( 1,  1,-1) )
    MF.add_force( 5,  (-2,  0,-2) )
```



Now that the model is finished some additional Settings have to be chosen. Initial allowed incremental system vector components for both the displacement vector and the load-proportionality-factor have to be specified. We use *dlpf = 0.005* and *du = 0.05* (figured out after some trial and error). Both parameters can't be specified automatically as they depend on the model configuration.

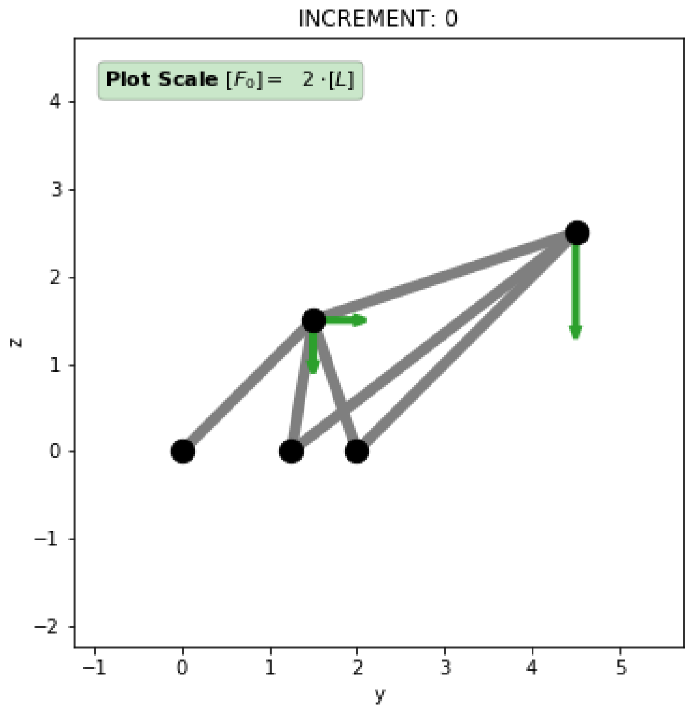
```
M.Settings.dlpf = 0.005
M.Settings.du = 0.05
```

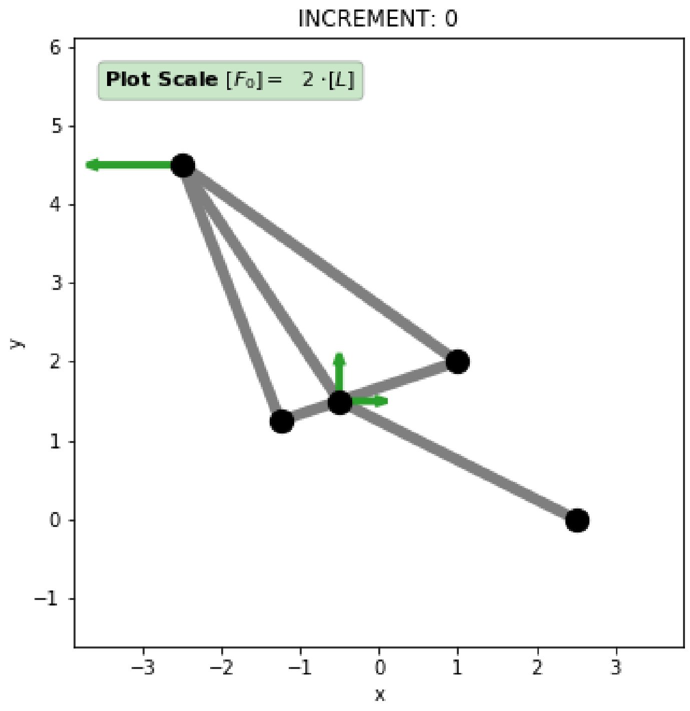
Next the job will be limited to a total amount of 190 increments (again, the total number has been figured out after some job runs to get good looking plots).

```
M.Settings.incs = 163
```

To speed up the calculation and make the model solution process more robust against a poorly defined initinal incremental system vector an automatic incremental step-size control is activated (*stepcontrol = True*). A maximum factor of *maxfac = 4* limits the increase of the incremental values. If the solution converges but the incremental system vector is bigger than the one specified a total overshoot factor of *dxtol = 1.25* is allowed.

```
M.Settings.stepcontrol = True
M.Settings.maxfac = 4

M.Settings.ftol = 8
M.Settings.xtol = 8
M.Settings.nfev = 8
```


INCREMENT: 0
Plot Scale $[F_0] = 2 \cdot [L]$
y
z


INCREMENT: 0
Plot Scale $[F_0] = 2 \cdot [L]$
x
y

```
M.Settings.dxtol = 1.25
```

5.4.2 Build & Run the Model

As the Model creation is finished we may start the calculation process by calling the two Model methods *build()* and *run()*. During the build process the model components will be sorted according to their label. Missing entries (e.g. nodes with zero external force vector) are automatically added. The *run()* method finally starts the calculation.

```
M.build()
M.run()
```

5.4.3 Verify the Results

After the job has finished the logfile contains useful information regarding convergence, recycles and control components. At the end of the logfile the total execution time during the *run()* method was measured with 6.7 seconds.

```
# LOGFILE "analysis.log"

total  cpu time "build":      0.001 seconds
total wall time "build":      0.001 seconds

total  cpu time "run":      6.739 seconds
total wall time "run":      6.737 seconds
```

For example at increment 40 a converged solution was obtained within 4 iterations plus one iteration to get the new control component. The determination of the control component is based on a linear solution (only 1 Newton-Rhapson iteration) for the current allowed incremental LPF. The sign of this allowed incremental LPF is estimated with the sign of the determinant of the stiffness matrix *det(KT)*. The vector norm of the equilibrium equations and the incremental system vector both satisfy the specified tolerance *tol = 1e-8* at the end of the increment. The 5th component of the incremental system vector (+DOF 5) is used as control component for the path tracing algorithm. No overshoot is detected - the control equation was fullfilled. The 2nd biggest displacement is -DOF 4 with a relative displacement of *-0.3168* compared to the allowed value. As the maximum incremental stepwidth was already reached in increment 8 and that there were no convergence problemes during this increment the allowed incremental system vector is whether increased nor decreased by the automatic stepcontrol. The final LPF at the end of increment 40 is *LPF(inc 40) = -0.003547*.

INCREMENT 40								
			Norm		sorted Dx/Dxmax (descending)			
Cyc	NR-It.	Control	Eq.	dx	i: Value	i: Value	i: Value	i: Value
1	pre	6	1.149e+00	4.779e+00	5: 2e+01			
	1		2.499e-03	2.161e-01				
	2		3.514e-06	6.512e-03				
	3		2.143e-11	1.435e-05				
	4		2.072e-16	1.108e-10				
tot	sum	used	final	final	final			
1	4	5	2.072e-16	1.108e-10	5: 1.0000	4: -0.3168	3: 0.2030	2: 0.0502
• final LPF: -0.003547								

In addition to the biggest components of the relative incremental system vector the Result object inside the Model also contains the absolute values. The results are printed below.

```
>>> M.Results.R[40].U - M.Results.R[39].U
array([[ 0.        ,  0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        ],
       [-0.00316   ,  0.01003059,  0.04060547],
       [-0.06335354,  0.        ,  0.2       ]])

>>> M.Results.R[40].Ured - M.Results.R[39].Ured
array([-0.00316   ,  0.01003059,  0.04060547, -0.06335354,  0.2       ])

>>> M.Results.R[40].dVmax
array([0.2 , 0.2 , 0.2 , 0.2 , 0.2 , 0.02])
```

With this information at hand it is shown that DOF 5 is the biggest incremental displacement component (in positive direction). The total displacement vector and LPF factor for increment 40 is accesible via the Model Result:

```
>>> M.Results.R[40].lpf
-0.0035474199465762137

>>> M.Results.R[40].U
array([[ 0.        ,  0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        ],
       [-0.03397299,  1.21931575, -1.66109327],
       [-1.09743012,  0.        , -3.40464559]])

>>> M.Results.R[40].Ured
array([-0.03397299,  1.21931575, -1.66109327, -1.09743012, -3.40464559])
```

The element forces for increment 40 are

```
>>> M.Results.R[40].element_force
array([[ 0.08230657],
       [-0.03135514],
       [-0.109559  ],
       [ 0.05063536],
       [-0.0412221 ],
       [-0.03236385]])
```

and the corresponding strains in the elements are evaluated with the stretches:

```
>>> M.Results.R[40].stretch - 1
array([[ 0.10974209],
       [-0.03135514],
       [-0.219118  ],
       [ 0.06751382],
       [-0.0412221 ],
       [-0.03236385]])
```

The system equilibrium equations are displayed for the whole system and are reshaped to *(nnodes,ndim)*.

```
>>> M.Results.R[40].g.reshape(M.nnodes,M.ndim)
array([[-6.12431121e-02,  5.48915100e-02, -3.25179328e-03],
       [ 9.87502262e-03, -6.06477963e-02,  1.21489498e-02],
       [ 5.49155094e-02, -2.24698313e-02,  1.74510337e-03],
       [ 1.83013327e-16,  4.77048956e-17, -1.47451495e-17],
       [ 8.15320034e-17,  2.46786976e-02,  1.73472348e-17]])
```

If we take only the active DOF from this vector and plot it as flattened array it is shown that the equilibrium is fullfilled. Another check is performed with the interal force and external force vector at increment 40, which shows the same result as *g*.

```
>>> M.Results.R[40].g.take(M.nproDOF1)
array([ 1.83013327e-16,  4.77048956e-17, -1.47451495e-17,  8.15320034e-17,  1.
↪73472348e-17])

>>> (-M.Results.R[40].r + M.Results.R[40].lpf * M.ExtForces.forces).take(M.nproDOF1)
array([ 1.83013327e-16,  4.77048956e-17, -1.47451495e-17,  8.15320034e-17,  1.
↪73472348e-17])
```

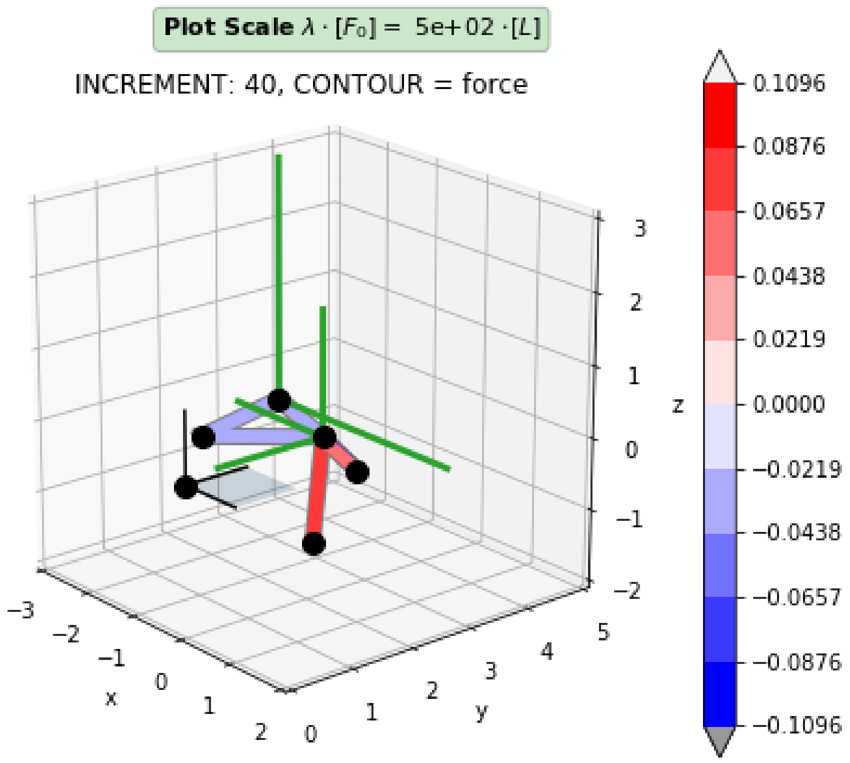
5.4.4 Model Plot and Node History

To visualize the deformed state of the model for increment 40 some model plots are generated. First the undeformed configuration is generated for different views.

```
# undeformed views
fig, ax = M.plot_model(config=['undeformed'],
                       view='3d', #'xy', 'yz', 'xz'
                       contour='force',
                       lim_scale=(-3,2,0,5,-1,4), #3d
                       #lim_scale=1.4, #plane-view
                       force_scale=5.0, #2
                       inc=0)
fig.savefig('model_undeformed_inc0_3d.pdf')
fig.savefig('model_undeformed_inc0_3d.png')
```

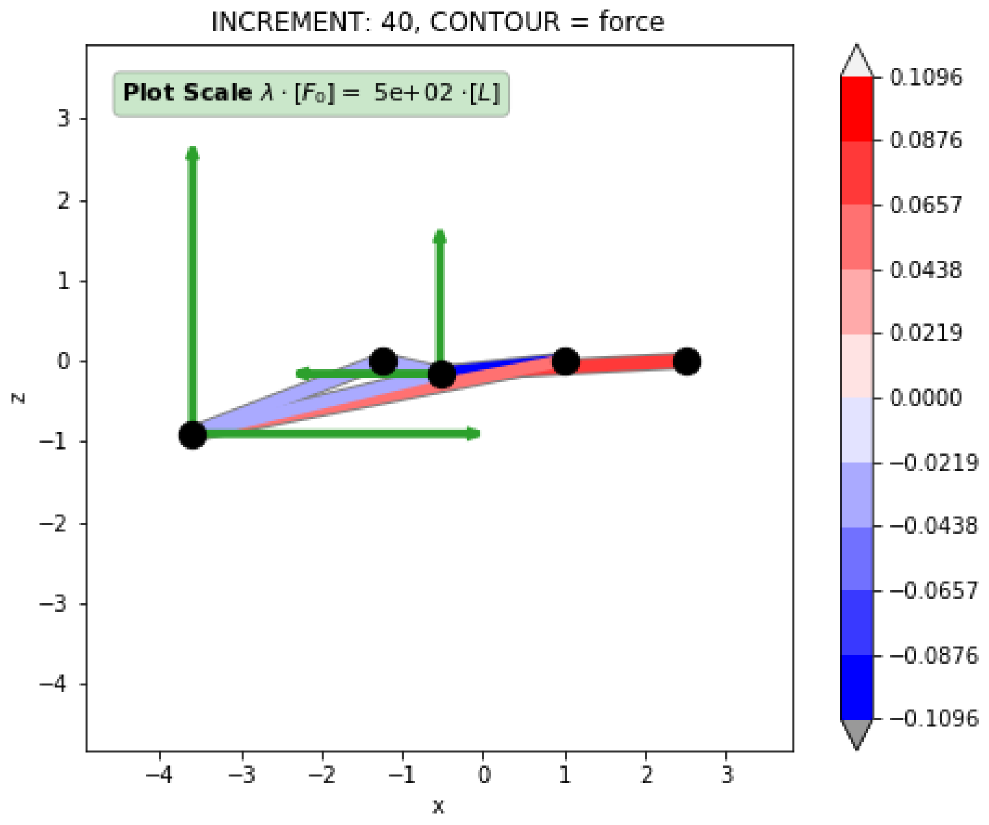
For the deformed model the figures are generated with the following code:

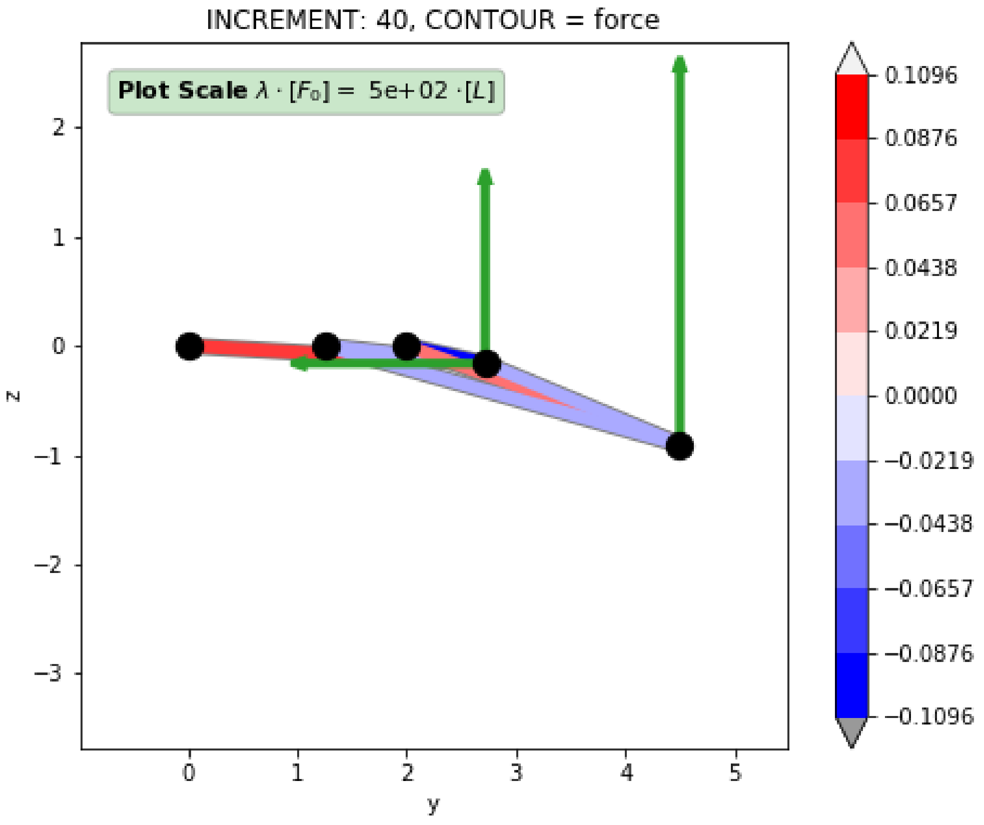
```
fig, ax = M.plot_model(config=['deformed'],
                       view='xz',
                       contour='force',
                       lim_scale=1.3,
                       force_scale=500.0,
                       inc=pinc)

fig, ax = M.plot_model(config=['deformed'],
                       view='3d',
                       contour='force',
                       lim_scale=(-3,2,0,5,-2,3),
                       force_scale=500.0,
                       inc=40)
```

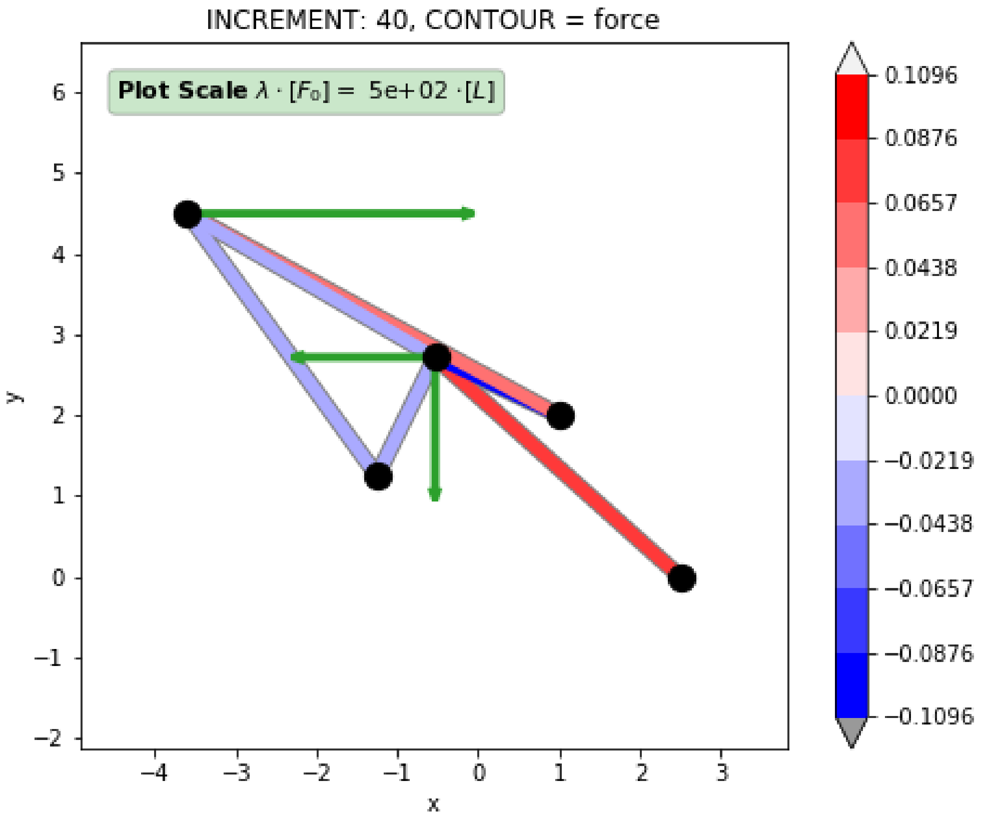


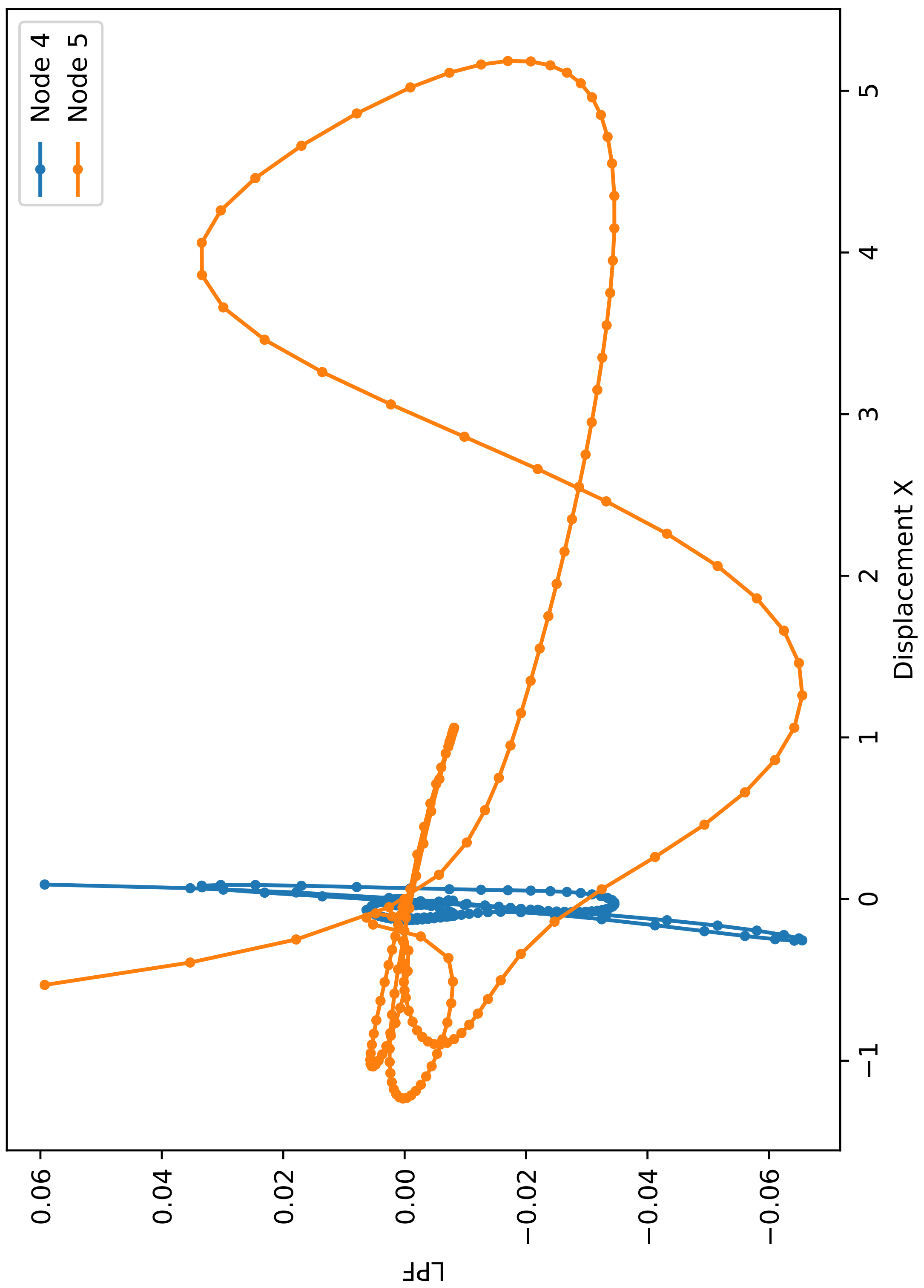
5.4.5 Path-Tracing of the Displacement-LPF curves

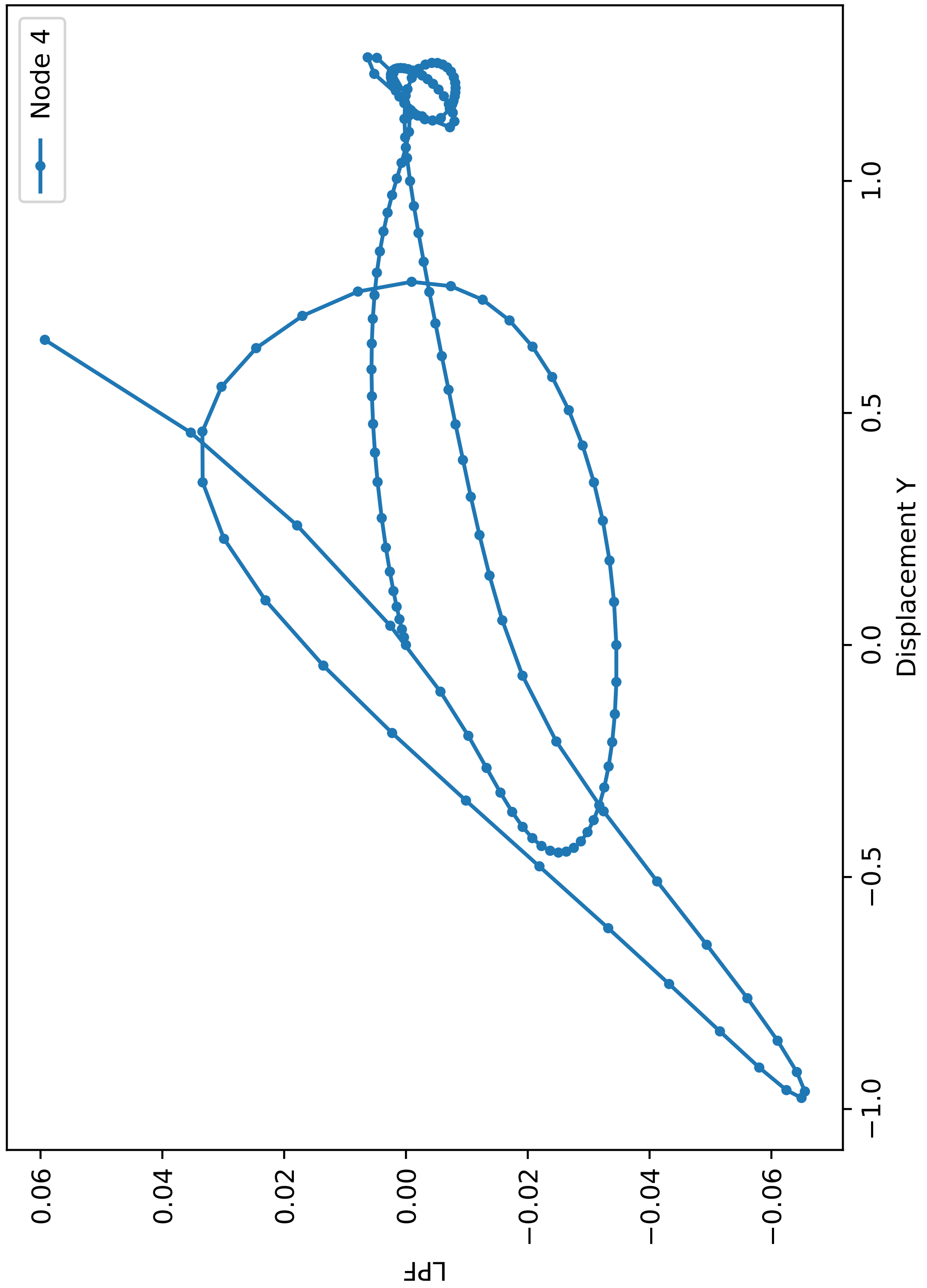
The path-tracing of the deformation process is shown as a History Plot of Displacement-LPF curves for all active DOF. Strong geometrical nonlinearities are observed for all active DOF.

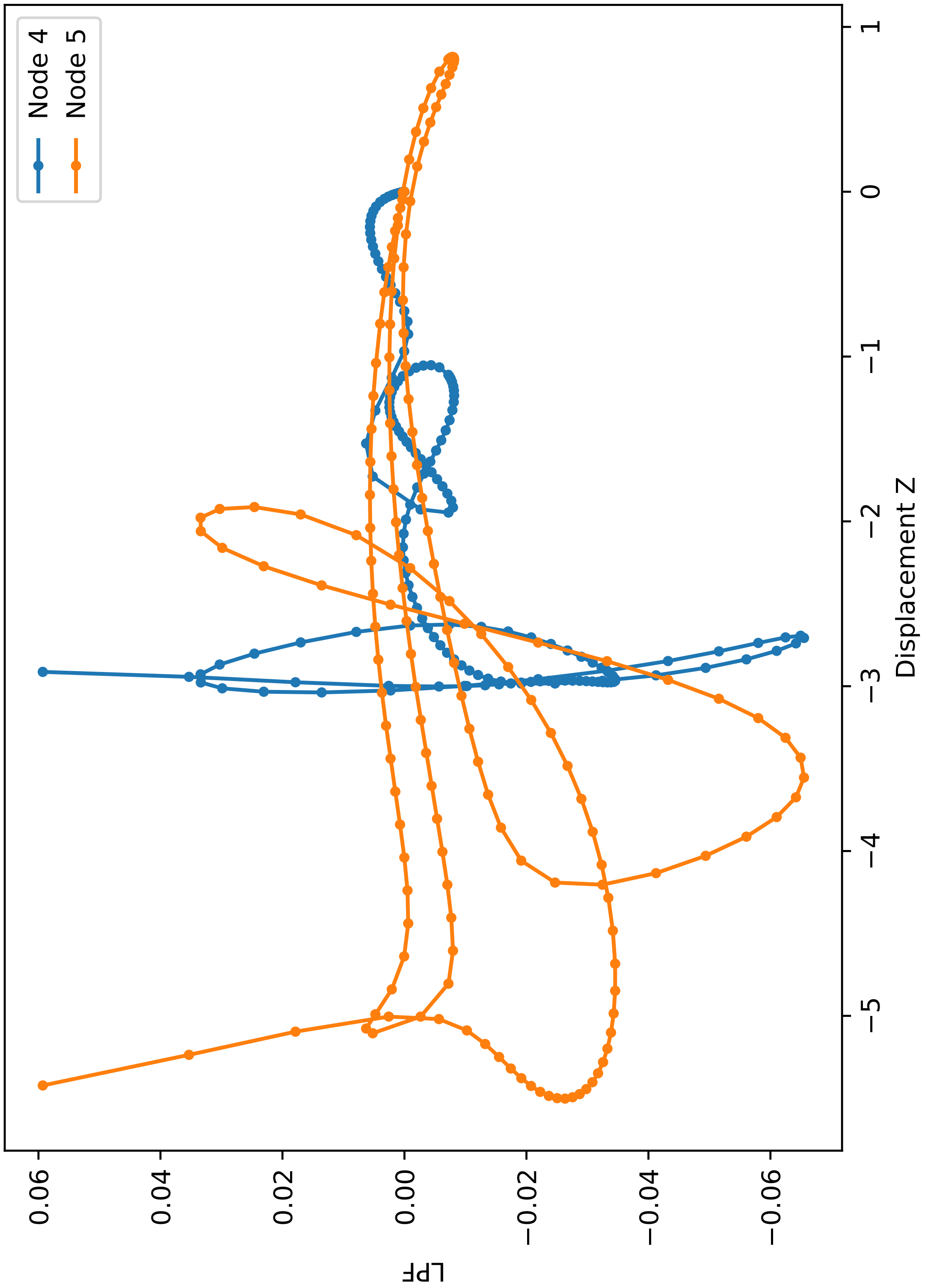

INCREMENT: 40, CONTOUR = force
Plot Scale $\lambda \cdot [F_0] =$ 5e+02 $\cdot [L]$
z
x
3
2
1
0
−1
−2
−3
−4
−4
−3
−2
−1
0
1
2
3
0.1096
0.0876
0.0657
0.0438
0.0219
0.0000
−0.0219
−0.0438
−0.0657
−0.0876
−0.1096


INCREMENT: 40, CONTOUR = force
Plot Scale $\lambda \cdot [F_0] =$ 5e+02 $\cdot [L]$
z
y
0.1096
0.0876
0.0657
0.0438
0.0219
0.0000
−0.0219
−0.0438
−0.0657
−0.0876
−0.1096


INCREMENT: 40, CONTOUR = force
Plot Scale $\lambda \cdot [F_0] =$ 5e+02 $\cdot [L]$
y
x
0.1096
0.0876
0.0657
0.0438
0.0219
0.0000
−0.0219
−0.0438
−0.0657
−0.0876
−0.1096


Node 4
Node 5
Displacement X
LPF
−1
0
1
2
3
4
5
0.06
0.04
0.02
0.00
−0.02
−0.04
−0.06


Node 4
Displacement Y
LPF
−1.0
−0.5
0.0
0.5
1.0
0.06
0.04
0.02
0.00
−0.02
−0.04
−0.06


Node 4
Node 5
Displacement Z
LPF
1
0
−1
−2
−3
−4
−5
0.06
0.04
0.02
0.00
−0.02
−0.04
−0.06

CHAPTER SIX

NOTES

This project evolved out of a homework for the course Nonlinear Structural Analysis (202.482) at Graz University of Technology (Lecturer: Guggenberger, Werner, Ao.Univ.-Prof. Dipl.-Ing. Dr.techn.).

CHAPTER

SEVEN

LICENSE

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search