# neon user documentation

**Darcy Beurle**

**Sep 16, 2019**

# Contents

# CHAPTER 1

# Introduction

neon is a finite element solver written in C++. It aims to solve industrial style finite element problems exploiting modern multicore architectures such as GPUs to accelerate simulations. The finite element problem definition is specified through a `json` format input file. Information regarding material, parts, constitutive models, boundary conditions and solution techniques are specified in this input file.

The mesh file is also given in `json` format which makes it trivial to parse and provides a standardised mesh format that includes syntax highlighting and code folding features.

A containerised version of neon is also available at Dockerhub, which allows the build environment to be simply reproduced.

# CHAPTER 2

# Meshing

In order to construct the system of equations describing the deformation of a body, it must first be spatially discretised by a mesh. This involves providing a CAD geometry file to a pre-processor (such as gmsh) to obtain the body in terms of coordinates and node indices.

Neon uses a utility `gmshreader` to perform the conversion from Gmsh `.msh` format to a `json` representation with the file extension `.mesh`. This program can be found here. It is important to note here that *neon* expects the mesh indices as zero-based.

The mesh file produced must be in the same directory as the executable when reading the input file. The examples show the use of this, including the original Gmsh geometry and mesh data.

## 2.1 Element Options

### 2.1.1 Numerical Integration

The evaluation of the integrals in the finite element method use numerical quadrature rules specialised for each element. For computational efficiency reasons, these could be under-integrated or fully-integrated. However, in the current state reduced integration rules produce a rank-deficient element stiffness matrix and are therefore not recommended for use until stabilisation is applied. For each mesh type, the element options can be specified

```
"element_options" : {
    "quadrature" : "full"
}
```

with reduced integration selected when `"quadrature" : "reduced"`.

# CHAPTER 3

# Section Properties

Sections relate the three-dimensional geometric properties to a lower dimensional approximation of a beam.

## 3.1 Beam

For each beam a name, type and the corresponding dimensions are required.

### 3.1.1 Rectangle

A rectangular profile can be specified by defining `rectangle` as the profile type. The dimensions `width` and `height` are required. For example

```
"profiles" : [{
  "name" : "steel_bar",
  "type" : "rectangle",
  "width" : 200,
  "height" : 200
}]
```

### 3.1.2 Circle

A circular profile can be specified using `circle`. The `diameter` is required in this case. For example

```
"profiles" : [{
  "name" : "steel_bar",
  "type" : "circle",
```

(continues on next page)

(continued from previous page)

```
  "diameter" : 200
}]
```

### 3.1.3 Hollow Circle

Likewise, a hollow circular profile, (e.g. a pipe) can be specified using `hollow_circle`. An `inner_diameter` and `outer_diameter` are required, and the `inner_diameter` must be a smaller value than the `outer_diameter`.

```
"profiles" : [{
  "name" : "steel_bar",
  "type" : "hollow_circle",
  "inner_diameter" : 100,
  "outer_diameter" : 200
}]
```

# CHAPTER 4

# Material Properties

Constitutive models are used to model a material and require a number of material parameters. These properties are tied closely to the constitutive model. As an example, it is possible for multiple bodies to have the same material parameters but different constitutive models. To handle this particular case, each `materials` definition has a unique identifier `name`. For example

```
"materials" : [
{
    "name" : "steel"
},
{
    "name" : "aluminium"
}
]
```

where the `name` field acts as the unique key for each material. An error will be produced for duplicated names. This material field can have properties

```
"materials" : [{
    "name" : "steel",
    "elastic_modulus" : 134.0e3,
    "poissons_ratio" : 0.3
}]
```

and it is up to the constitutive model and the material model to check if the required material properties exist inside the material class. This means you can include irrelevant material properties and only the required values are used. The user should check the constitutive model definition for the list of required material parameters.

# CHAPTER 5

# Procedures

There are several procedures available to handle structural analysis in the finite element framework. The most general procedure is the continuum theory, which solves the momentum equation over a three dimensional body. However, the computational cost of this can be high which led to the development of specialised beam and shell theory. These reduced models alleviate the burden of a three-dimensional simulation with some restrictions.

In this section, an outline of specialised theory in `neon` is documented along with the modelling assumption the theory contains.

## 5.1 Beam

When approximating the structural member with beam theory, additional information regarding the cross-sectional area and the second moment of area must be given in order find an approximate solution. This is accomplished for all beam theories by specifying a `profile` and the orientation of this profile in space for every finite element.

Each element group with the same profile is grouped together inside a `mesh` object in the input file

```
"meshes" : [{
    ...
    "section" : [{
        "name" : "<name of element group in mesh file>",
        "profile" : "<name of profile definition>",
        "tangent" : [0.0, 1.0, 0.0],
        "normal" : [1.0, 0.0, 0.0]
    },
    ...
    ],
}]
```

### 5.1.1 Linear (C0)

In order to avoid the requirement of higher order shape functions (see Beam Theory (C1)), a reduced form of the momentum equation is solved rather than discretising the Euler-Bernoulli fourth order differential equation. This involves the computation of four separate stiffness matrices; axial, torsional, bending and shear. A specialised integration scheme is automatically applied to this analysis to avoid shear locking through under integration of selected stiffness matrices.

### 5.1.2 Euler-Bernoulli (C1)

If the Euler-Bernoulli beam theory is directly formulated in the finite element method, the resulting shape functions are required to be C1 continuous, resulting in a higher order interpolation. This theory will be implemented after the C0 theory.

## 5.2 Continuum

The evaluation of the integrals in the finite element method use numerical quadrature rules specialised for each element. For computational efficiency reasons, these could be under-integrated or fully-integrated. However, in the current state reduced integration rules produce a rank-deficient element stiffness matrix and are therefore not recommended for use until stabilisation is applied. For each mesh type, the element options can be specified

```
"element_options" : {
    "quadrature" : "full"
}
```

with reduced integration selected when `"quadrature" : "reduced"`.

# CHAPTER 6

# Constitutive Models

Constitutive models play an important role in the solution of the equations of motion. They are required to close the system such that the system is solvable. These models relate quantities of displacement to the stress in the body. Each model of material deformation requires some material properties and can effect the solution time, stability and accuracy of the overall finite element simulation.

All of the units are assumed to be consistent and it is the user's responsibility to make sure this happens.

## 6.1 Isotropic Linear Elasticity

The simplest form of constitutive model for linear materials is the isotropic linear elasticity model. This is specified using `"isotropic_linear_elasticity"`. The required parameters are

- `"poissons_ratio"` for the Poisson's Ratio of the material
- `"elastic_modulus"` for the elasticity modulus (slope of stress-strain curve)

In addition, the elasticity constants can also be specified using `"shear_modulus"` and `"bulk_modulus"` if this is more convenient for the user.

The model is specified by

```
"constitutive" : {
    "name" : "isotropic_linear_elasticity"
}
```

## 6.2 Small Strain J2 Plasticity

The small strain J2 plasticity model introduces material non-linearity into the finite element formulation. This model extends the `"isotropic_linear_elasticity"` model to include:

- `"YieldStress"` stress when the material begins plastic flow

- `"IsotropicHardeningModulus"` the hardening modulus of the material

This model is used by specifying

```
"constitutive" : {
    "name" : "J2Plasticity",
    "finite_strain" : false
}
```

where the finite strain version of the J2 model is not required. Further modifications to the J2 model are given in subsequent subsections.

### 6.2.1 Isotropic Chaboche Damage

This non-linear isotropic ductile damage model is based on the `"J2Plasticity"` model. It is mainly controlled by the following material parameters:

- `"kinematic_hardening_modulus"` the kinematic hardening modulus of the material
- `"softening_multiplier"` a scaling factor multiplied by the back stress in the plastic yield function
- `"plasticity_viscous_exponent"` $n$: relates the plastic multiplier to the plastic yield function exponentially
- `"plasticity_viscous_denominator"` $K$: scales the plastic yield function $\dot{\lambda}_{vp} = \langle \frac{f_{vp}}{K} \rangle^n$
- `"damage_viscous_exponent"` $s$: relates the damage multiplier to the damage yield function exponentially
- `"damage_viscous_denominator"` $S$: scales the damage yield function $\dot{\lambda}_d = \langle \frac{f_d}{S} \rangle^s$

This model is used by specifying

```
"constitutive" : {
      "name" : "J2Plasticity",
      "damage" : "isotropic_chaboche",
      "finite_strain" : false
  }
```

This constitutive model is implemented only in an infinitesimal strain framework.

## 6.3 Neo-Hooke Elasticity

The Neo-Hooke model is a common hyperelastic material model which can be used to model non-linear elasticity. Only two material parameters required, which are

- `"poissons_ratio"` for the Poisson's ratio of the material
- `"elastic_modulus"` for the elasticity modulus (slope of stress-strain curve)

alternatively the material parameters

- `"bulk_modulus"`
- `"shear_modulus"`

can be used, where the `"bulk_modulus"` acts as the penalty parameter for volumetric strain term.

This model is used by specifying

```
"constitutive" : {
    "name" : "neohooke"
}
```

Note that this constitutive model invokes the finite strain solver, which requires an incremental approach and additional Newton-Raphson solver iterations.

## 6.4 Affine Microsphere

The affine microsphere model uses a Langevin statistical mechanics model for polymer chains and derives a continuum model suitable for a finite element implementation. This model uses numerical integration over a unit sphere to compute this homogenisation and requires a quadrature rule to be specified. Note that higher order rules increase the computation time. This model is suited to larger deformations than the Neo-Hooke model.

Material parameters are

- `"bulk_modulus"`
- `"shear_modulus"`
- `"segments_per_chain"` Number of segments per chain (experimental)

To specify the unit sphere quadrature rules

- `"BO21"` 21 point symmetric scheme
- `"BO33"` 33 point scheme
- `"FM900"` 900 point scheme. Note this is computationally expensive

The model is used by specifying

```
"constitutive" : {
    "name" : "microsphere",
    "type" : "affine",
    "statistics" : "langevin",
    "quadrature" : "BO21"
}
```

## 6.5 Gaussian Affine Microsphere

The Gaussian affine microsphere model re-derives the affine microsphere model using a Gaussian chain description and significantly reduces the complexity of the model.

Material parameters are

- `"bulk_modulus"`
- `"shear_modulus"`
- `"segments_per_chain"` Number of segments per chain (not required)

To specify the unit sphere quadrature rules

- `"BO21"` 21 point symmetric scheme
- `"BO33"` 33 point scheme
- `"FM900"` 900 point scheme. Note this is computationally expensive

The model is used by specifying

```
"constitutive" : {
    "name" : "microsphere",
    "type" : "affine",
    "statistics" : "gaussian",
    "quadrature" : "BO21"
}
```

CHAPTER 7

# Boundary Conditions

An essential part of defining the model is the specification of appropriate boundary conditions. Boundary conditions are applied to the mesh and included in the mathematical description of the system. There are a few different types of boundary conditions, which are specialised for each particular type of simulation.

Each boundary condition is applied over a specified time. That means, for each boundary condition the user must specify *at least* two data points inside a vector type

```
"boundaries" : [{
    "name" : "x-sym",
    "type" : "displacement",
    "time" : [0.0, 1.0],
    "x" : [0.0, 0.0]
}]
```

where multiple boundary conditions can be added by inserting extra elements into the array.

The specified time and values are automatically linearly interpolated if a time step happens to fall between two values. For convenience the adaptive time step algorithm will always produce a solution at the specified time points to enable easy experimental comparisons where only some discrete time points are evaluated.

Since boundary conditions can vary over time, it is possible to specify cyclic type boundary conditions that follow a periodic curve. Currently, only a sinusoidal periodic type is implemented and it can be specified with

```
"boundaries" : [{
    "name" : "x-sym",
    "type" : "displacement",
    "generate_type" : "sinusoidal",
    "x" : [0.05, 0.10],
    "period" : [10,10],
    "phase" : [0,0],
    "number_of_cycles" : [2,1]
}]
```

This may be used to generate block loading where each block follows a specific sinusoidal wave that corresponds to one entry of `"x"`, `"period"`, `"phase"` and `"number_of_cycles"`. In this case the time step is taken to be

equal to

```
"time" : {
    "period" : 30,
    "increments" : {
        "initial" : 0.1
    }
```

and the time steps are generated automatically based on the given boundary condition parameters.

## 7.1 Essential (Dirichlet) Type

These boundary conditions are required to ensure there is no rigid body motion of the system under load.

| Simulation type | Specification |
|---|---|
| Solid mechanics | `"displacement"` specified with degree of freedom `"x"`, `"y"` or `"z"` |
| Thermal | `"temperature"` specified by `"value" : []` |

## 7.2 Natural (Neumann) Type

In contrast to Dirichlet boundary conditions, Neumann boundary conditions specify a derivative over a boundary. These can be used to model tractions, pressure, heat flux, gravitational loads and internal heat generation. Because of their diverse meaning, this section will be split into each simulation type possible.

### 7.2.1 Solid Mechanics

| Keyword | Specification |
|---|---|
| `"traction"` | Specified with degree of freedom `"x"`, `"y"` and/or `"z"` |
| `"body_force"` | Specified with degree of freedom `"x"`, `"y"` and/or `"z"` |
| `"pressure"` | Specified by `"value" : []` |

Note here, that `"pressure"` is a non-follower load and does not rotate with the deformation of the body.

### 7.2.2 Thermal

Thermal analysis specifies a heat flux which is defined to the normal of the surface. In this case only a scalar value needs to be specified.

| Keyword | Specification |
|---|---|
| `"flux"` | Specified by `"value" : []` |

A special case occurs for the thermal analysis as a mixed (or Robin type) boundary condition. This type is a combination of a Neumann and a Dirichlet condition and is used to model natural convection and is known as Newton Cooling. This is specified by

```
{
    "name" : "fins",
    "type" : "newton_cooling",
    "time" : [0.0, 5000.0],
    "heat_transfer_coefficient" : [50.0, 50.0],
    "ambient_temperature" : [323.0, 323.0]
}
```

CHAPTER 8

# Solution Methods

## 8.1 Linear Equilibrium and Transient

Solving linear problems involves one invocation of a linear solver and one assembly step and is therefore inexpensive to perform. These routines are automatically selected based on the problem and deserve no special discussion.

## 8.2 Non-linear Equilibrium

In the non-linear finite element framework, the solution of the non-linear equations requires linearisation and an incremental approach to determine the solution. This means that multiple invocations of a linear solver are required to reach a specified equilibrium.

To perform the solution stage, neon implements the full Newton-Raphson method such that an updated tangent matrix is computed in each iteration. Since the assembly of the tangent stiffness matrix is implemented in parallel, the computational cost is very low in comparison to the cost of a linear solve. Other finite element solvers will avoid the computation of stiffness matrix due to the computational cost at the expense of improved convergence properties.

The iterative nature of a non-linear problem requires the use of tolerances to determine if the results are sufficiently converged. For this, non-linear simulation cases need to specify the relative displacement, force residuals and the maximum number of Newton-Raphson iterations to perform before a cutback. For additional control, the relative or absolute tolerances can be chosen based on the physics of the problem

```
"nonlinear_options" : {
    "displacement_tolerance" : 1.0e-5,
    "residual_tolerance" : 1.0e-4,
    "absolute_tolerance" : true,
    "linear_iterations" : 15
}
```

Methods to improve the properties of the Newton-Raphson could be implemented on top of the current non-linear solvers, such as line searching algorithms to improve convergence properties.

## 8.3 Non-linear Implicit Dynamic

The code to solve the implicit dynamic problem has been implemented but not yet tested. If required, please provide a test case to the developers.

## 8.4 Natural Frequency

The generalised Eigenvalue problem can be solved by specifying

```
"steps" : [{
    ...
    "solution" : "natural_frequency",
    ...
}]
```

Note that loads do not need to be specified for free vibration analysis but the restraints must be specified. Since this solution method builds a mass matrix, the material for the mesh must specify a material density.

For a selection of valid Eigenvalue solvers, please refer to the solvers section of this manual.

# CHAPTER 9

# Solvers

## 9.1 Linear systems

Linear solvers typically take up the largest portion of the solution time in the finite element method. Very large scale systems of million degrees of freedom should use iterative solvers as their memory usage is small in comparison to their direct solver brethren. For problems where it is possible to use a direct solver then this is the suggested default.

Neon uses a selection of linear solvers, most of which employ shared memory parallelisation or have GPU-accelerated counterparts. Direct solvers require no options apart from the solver name. The type of solver (symmetric or unsymmetric) is automatically deduced based on boundary conditions, constitutive model and other algorithmic choices.

Table 1: Direct solvers available `"type" : "keyword"`

| Type keyword | Details |
|---|---|
| `"direct"` | Inbuilt from the Eigen library (LLT or LU) |
| `"PaStiX"` | Interfaces to the PaStiX library (LLT or LU) |
| `"MUMPS"` | Interfaces to the MUMPS library (LDLT or LU) |

An example of using the `"PaStiX"` solver is

```
"linear_solver" {
    "type" : "PaStiX"
}
```

To specify an iterative solver require additional fields due to the white-box nature of the methods. If these are not set, then defaults will be chosen for you. The following table demonstrates the defaults, where each iterative solver currently uses a diagonal pre-conditioner due to ease of parallelisation

Table 2: Iterative solvers defaults

| Iterative solver option | Default value |
|---|---|
| `"device"` | `"cpu"` |
| `"tolerance"` | `1.0e-6` (Relative tolerance) |
| `"maximum_iterations"` | `1.0e-6` (Maximum iterations before failure) |

and the option meanings

Table 3: Iterative solvers available "Type" : "keyword"

| Additional options | Details |
|---|---|
| "device" | "cpu" and "gpu" |
| "backend" | Provide options for the acceleration framework |

Note than when selecting "gpu" the binary must have compiled in support that is enabled through the -DENABLE_OPENCL=1 or -DENABLE_CUDA CMake flags.

An example of an iterative solver definition using the CUDA iterative linear solver

```
"linear_solver" {
    "type" : "iterative",
    "device" : "gpu",
    "tolerance" : 1.0e-6,
    "maximum_iterations" : 1500,
    "backend" : {
       "type" : "cuda",
       "device" : 0
    }
}
```

The "type" field can also contain "opencl" to use the OpenCL kernels for computation. Because of the multi-device nature of OpenCL a "device" and a "platform" are then required to select the desired device

```
"linear_solver" {
    "type" : "iterative",
    "device" : "gpu",
    "tolerance" : 1.0e-6,
    "maximum_iterations" : 1500,
    "backend" : {
       "type" : "opencl",
       "platform" : 0,
       "device" : 0
    }
}
```

All linear solvers use double floating point precision which may incur performance penalties on GPU devices however testing shows a significant increase in performance is still obtained with double precision due to the higher memory bandwidth. A single precision version of Krylov subspace solvers is not recommended due to round-off error in computing the search direction.

## 9.2 Eigenvalue problems

The solution of an eigenvalue problem arises in natural frequency analysis and buckling analysis in structural dynamics. Usually only the solution of a few eigenvalues are required and this becomes computationally feasible for large sparse matrices. These eigenvalues and eigenvectors correspond to the natural frequency and the mode shape of the structure for natural frequency analysis. For the linear elastic buckling load of a structure, the eigenvalues provide the limit load and the deformed shape.

Unlike direct methods for linear systems, eigenvalues have to be solved for in an iterative fashion. There are several different algorithms available depending on the problem.

An example of an eigenvalue solver definition

```
"eigen_solver" {
    "type" : "lanczos",
    "values" : 15,
    "spectrum" : "lower"
}
```

where the "type" field indicates what algorithm to use ("arpack", "lanczos" and "power_iteration") are available. The "values" keyword determines how many eigenvalues are to be solved for. This should be much less than the total number of degrees of freedom in the system. Finally the "spectrum" keyword indicates from which end of the spectrum the values will be computed, where "lower" indicates Eigenvalues from the lowest frequency and "upper" computes the higher frequency Eigenvalues.

CHAPTER 10

# Test Suite

To ensure correctness of the code and verify that the implementation is correct, multiple levels of testing are performed. At the finest level, unit tests are run to ensure functions inside the code perform as expected. These building blocks are assembled to form part of the larger code which relies on the correctness of the smaller blocks. In the middle, the larger blocks are then tested to ensure correctness. Finally integration tests are done to test the end-to-end execution for testing against analytical solutions.

CHAPTER 11

# License

neon is covered by the MIT open-source license and uses several other libraries, refer to each dependency for their license.

CHAPTER 12

# Contact

Questions? Bugs? Comments? Open up an issue on GitHub or submit a pull request.