
getfem examples

Release 20.12

Dec 29, 2020

Contents:

1	Indices and tables	1
2	Tutorial	3
2.1	Plotting mesh	3
2.2	Introduction to FEM Analysis with Python	5
2.3	study of cantilever beam	11
2.4	Damping model with one degree of freedom	20
2.5	Assembly examples in Python	33

CHAPTER 1

Indices and tables

- `genindex`
- `modindex`
- `search`

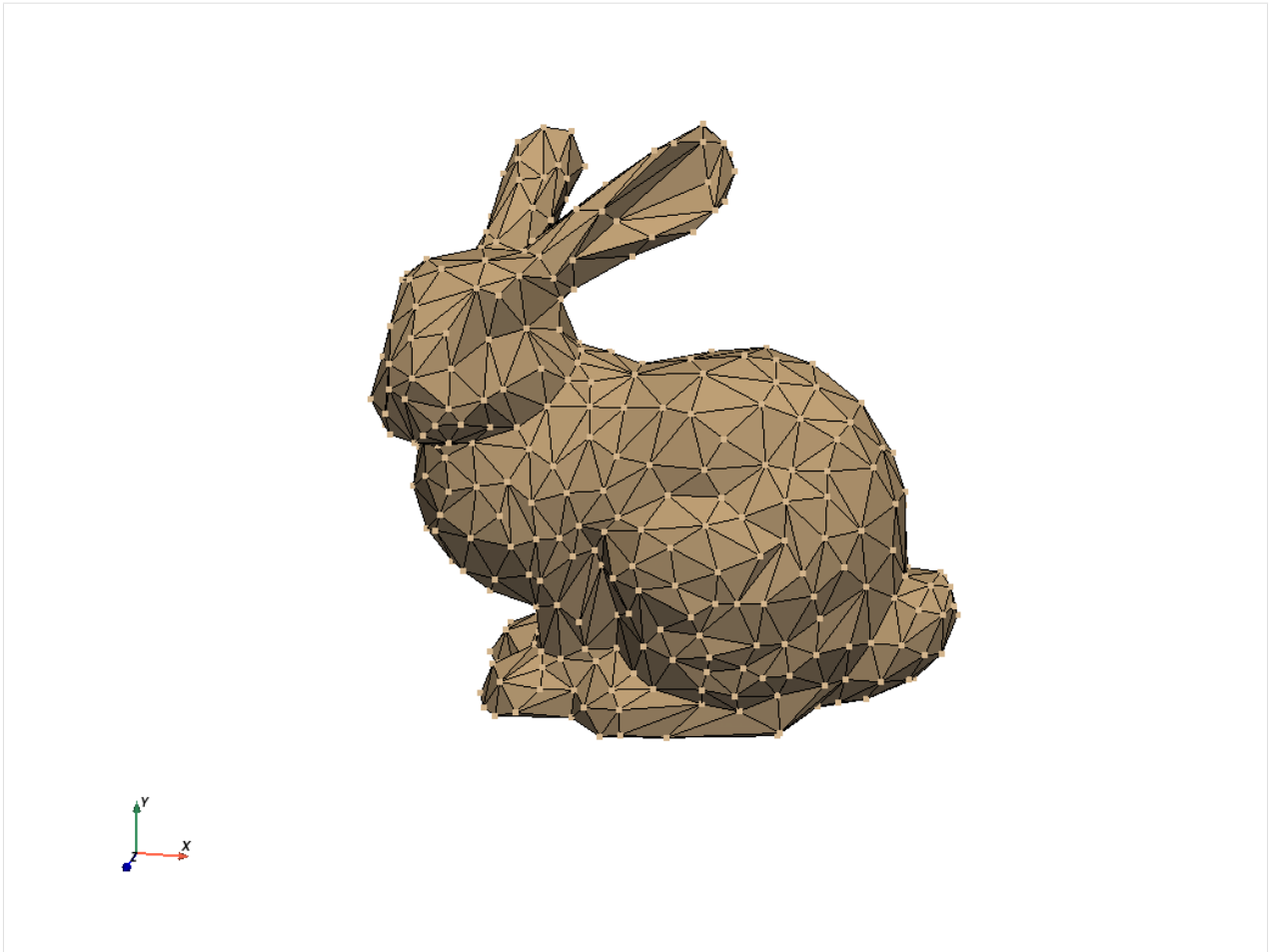
2.1 Plotting mesh

```
[1]: import pyvista as pv
      from pyvista import examples

      from pyvirtualdisplay import Display
      import os
      display = Display(visible=0, size=(1280, 1024))
      display.start()

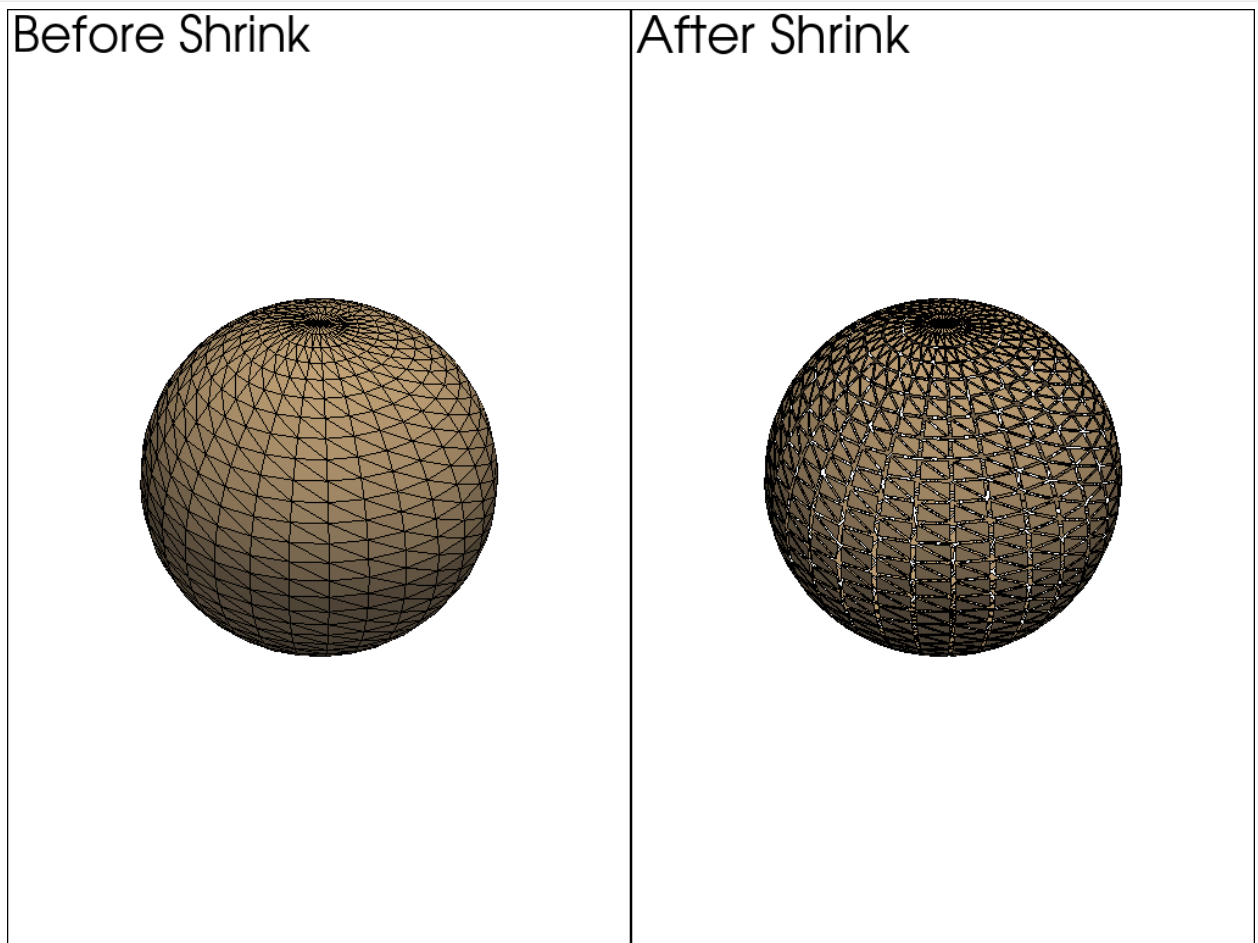
[1]: <pyvirtualdisplay.display.Display at 0x7f8dac50ff40>
```

```
[2]: mesh = examples.download_bunny_coarse()
      cpos = [(0.2, 0.3, 0.9), (0, 0, 0), (0, 1, 0)]
      mesh.plot(cpos=cpos, show_edges=True, color=True)
```



```
[2]: [(0.2, 0.3, 0.9),  
      (0.0, 0.0, 0.0),  
      (0.0, 1.0, 0.0)]
```

```
[3]: mesh = pv.Sphere()  
      shrunk_mesh = mesh.shrink(shrink_factor=0.8)  
      p = pv.Plotter(shape=(1, 2))  
      p.subplot(0, 0)  
      p.add_text("Before Shrink")  
      p.add_mesh(mesh, color="tan", show_edges=True)  
      p.subplot(0, 1)  
      p.add_text("After Shrink")  
      p.add_mesh(shrunk_mesh, color="tan", show_edges=True)  
      p.show()
```

```
[3]: [(2.8332709368378763, 2.8332709368378763, 2.8332709368378763),
      (0.0, 0.0, 0.0),
      (0.0, 0.0, 1.0)]
```

```
[ ]:
```

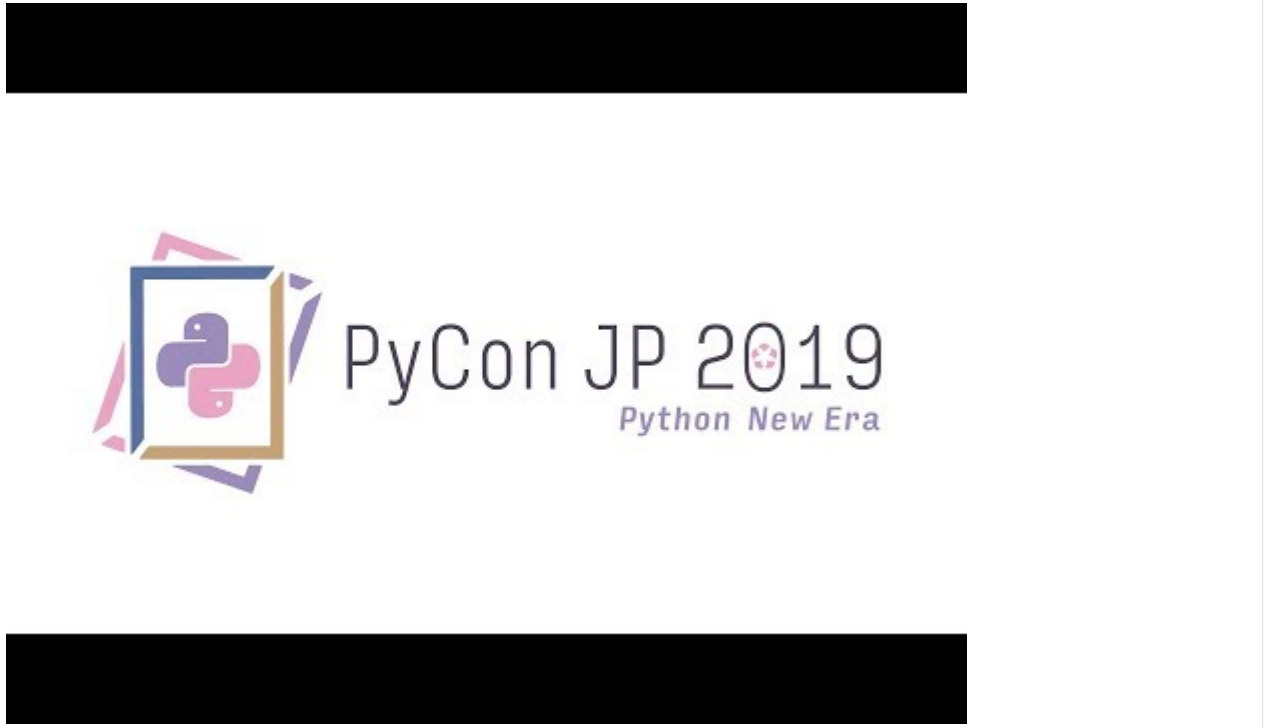
2.2 Introduction to FEM Analysis with Python

This tutorial aims to show using Python to pre-processing, solve, and post-processing of Finite Element Method analysis. It uses a finite element method library with a Python interface called [GetFEM](#) for preprocessing and solving. We will load vtk file by using [meshio](#) and visualize by [matplotlib](#) in pre-processing and post-processing. This tutorial was used in the [PyConJP 2019](#) talk. You can watch the talk on YouTube below. This tutorial is based on the following [official GetFEM page tutorial](#).

```
[1]: from IPython.display import YouTubeVideo

      YouTubeVideo("6JuB1GiDLQQ", start=512)
```

```
[1]:
```



2.2.1 Installation

GetFEM including its python interface can be installed from a terminal by executing aptitude update and aptitude install python3-getfem++.

```
sudo aptitude install python3-getfem++
```

The additional packages in [requirements.txt](#) are required for this tutorial. You do not need to build these environments because they are already configured in the [Dockerfile](#).

2.2.2 The problem setting

The problem refers to “Poisson’s Equation on Unit Disk” published by Math Works’s homepage.

$$\Delta u = 1 \text{ on } \Omega, u = 0 \text{ on } \partial\Omega$$

|pdedemo1_01|

2.2.3 How to use GetEM

We take the following steps when using GetFEM to solve finite element problems. See [this page](#) for more information on using GetFEM.

- define a [MesherObject](#)
- define a [Mesh](#)
- define a [MeshFem](#)
- define a [MeshIm](#)

- define a `Model` and set it up
- solve `Model` object
- get value from `Model` object

2.2.4 Initialization

GetFEM can be imported as follows (numpy has also to be imported).

```
[2]: import getfem as gf
import numpy as np
```

2.2.5 Mesh generation

We use GetFEM's `MesherObject` to create a mesh from the geometric information to be analyzed. This object represent a geometric object to be meshed by the experimental meshing procedure of GetFEM. We can represent a disk by specifying a radius, a 2D center, and using "ball" geometry.

```
[3]: center = [0.0, 0.0]
radius = 1.0

mo = gf.MesherObject("ball", center, radius)
```

We can make mesh object `mesh` by calling the experimental mesher of GetFEM on the geometry represented by `mo`. The approximate element diameter is given by `h` and the degree of the mesh is K ($K > 1$ for curved boundaries).

```
[4]: h = 0.1
K = 2
mesh = gf.Mesh("generate", mo, h, K)
```

2.2.6 Boundary selection

To define a boundary condition, we set a boundary number on the outer circumference of the circle.

```
[5]: outer_faces = mesh.outer_faces()
OUTER_BOUND = 1
mesh.set_region(OUTER_BOUND, outer_faces)
```

2.2.7 Mesh draw

We visualize the created mesh to check its quality. We can output mesh objects, but matplotlib can only display triangles. Therefore, we make a `Slice` object with a slice operation of `("none",)`, which does not cut the mesh. With a refinement of 1, this serves to convert the mesh to triangles.

```
[6]: sl = gf.Slice(("none",), mesh, 1)
```

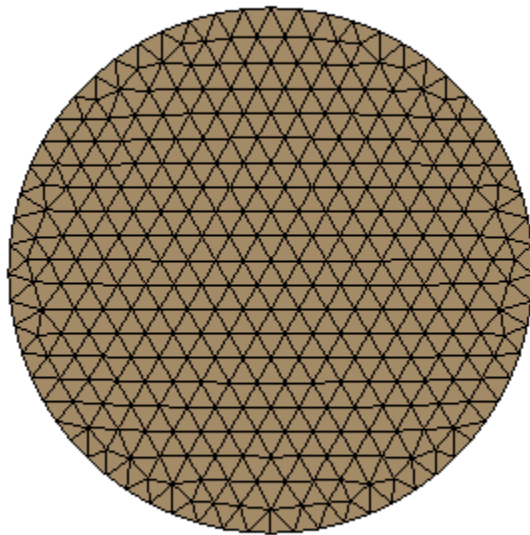
We can export a slice to VTK file by using `export_to_vtk` method.

```
[7]: sl.export_to_vtk("sl.vtk", "ascii")
```

We can render VTK files using Paraview or mayavi2. In order to display in the jupyter notebook this time, we read in `meshio` and draw in `matplotlib`.

```
[8]: import pyvista as pv
      from pyvirtualdisplay import Display

      display = Display(visible=0, size=(1280, 1024))
      display.start()
      p = pv.Plotter()
      m = pv.read("sl.vtk")
      p.add_mesh(m, show_edges=True)
      pts = m.points
      p.show(window_size=[512, 384], cpos="xy")
      display.stop()
```



```
[8]: <pyvirtualdisplay.display.Display at 0x7f6be0527c40>
```

2.2.8 Definition of finite element methods and integration method

We define the finite element and integration methods to use. We create a `MeshFem` that defines the degree of freedom of the mesh as rank 1 (scalar).

```
[9]: mfu = gf.MeshFem(mesh, 1)
```

Next we set the finite element used. Classical finite element means a continuous Lagrange element. Setting `elements_degree` to 2 means that we will use quadratic (isoparametric) elements.

```
[10]: elements_degree = 2
      mfu.set_classical_fem(elements_degree)
```

The last thing to define is an integration method `mim`. There is no default integration method in GetFEM so it is mandatory to define an integration method. Of course, the order of the integration method has to be chosen sufficient

to make a convenient integration of the selected finite element method. Here, the square of `elements_degree` is sufficient.

```
[11]: mim = gf.MeshIm(mesh, pow(elements_degree, 2))
```

2.2.9 Model definition

The model object in GetFEM gathers the variables of the models (the unknowns), the data and what are called the model bricks. The model bricks are some parts of the model (linear or nonlinear terms) applied on a single variable or linking several variables. A model brick is an object that is supposed to represent a part of a model. It aims to represent some integral terms in a weak formulation of a PDE model. They are used to make the assembly of the (tangent) linear system (see [The model object](#) for more details).

$$[K] \{u\} = \{F\}$$

```
[12]: md = gf.Model("real")
md.add_fem_variable("u", mfu)
```

2.2.10 Poisson's equation

To define Poisson's equation, we have to define a Laplacian term and RHS source term. We can add the Laplacian term (which is called a brick in GetFEM) by using `add_Laplacian_brick()`.

```
[13]: md.add_Laplacian_brick(mim, "u")
```

```
[13]: 0
```

If you want to define constants in GetFEM, we use the `add_fem_data()` method.

```
[14]: F = 1.0
md.add_fem_data("F", mfu)
```

We can set constant values with the `set_variable()` method. Here we pass a vector (ndarray) the size of the degrees of freedom.

```
[15]: md.set_variable("F", np.repeat(F, mfu.nbdof()))
```

We define the term RHS with the `add_source_term_brick()` method, using the constant `F` just defined.

```
[16]: md.add_source_term_brick(mim, "u", "F")
```

```
[16]: 1
```

Finally, we set the Dirichlet condition at the boundary.

```
[17]: md.add_Dirichlet_condition_with_multipliers(mim, "u", elements_degree - 1, OUTER_
↪BOUND)
```

```
[17]: 2
```

2.2.11 Model solve

Once the model is correctly defined, we can simply solve it by:

```
[18]: md.solve()
```

```
[18]: (0, 1)
```

2.2.12 Export/visualization of the solution

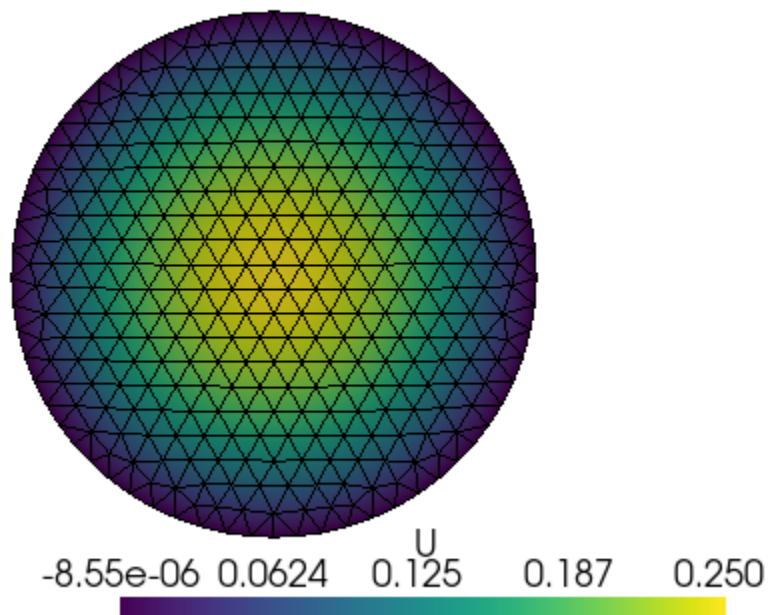
The finite element problem is now solved. We can get the solution u by using `variable` method.

```
[19]: U = md.variable("u")
```

We can output the computed u with the mesh of the `Slice` object.

```
[20]: sl.export_to_vtk("u.vtk", "ascii", mfu, U, "U")
```

```
display = Display(visible=0, size=(1280, 1024))
display.start()
p = pv.Plotter()
m = pv.read("u.vtk")
p.add_mesh(m, show_edges=True)
pts = m.points
p.show(window_size=[512, 384], cpos="xy")
display.stop()
```



```
[20]: <pyvirtualdisplay.display.Display at 0x7f6b9815f1c0>
```

2.2.13 Exact solution

The exact solution to this problem is given by the following equation:

$$u(x, y) = \frac{1 - x^2 - y^2}{4}$$

```
[21]: evalue = mfu.eval("(1-x*x-y*y)/4")
```

We can calculate the error for the L2 and H1 norms by using `compute`:

```
[22]: L2error = gf.compute(mfu, U - evalue, "L2 norm", mim)
      H1error = gf.compute(mfu, U - evalue, "H1 norm", mim)
      print("Error in L2 norm : ", L2error)
      print("Error in H1 norm : ", H1error)
```

```
Error in L2 norm :  1.965329030567132e-06
Error in H1 norm :  0.00010936971229957788
```

As you can see, the size of the error is within the acceptable range.

```
[ ]:
```

2.3 study of cantilever beam

Consider a cantilever beam using the Finite Element Method library GetFEM++.

2.3.1 Creating a Model

Now we are ready, import the library.

```
[1]: import getfem as gf
      import numpy as np
      import numpy.testing as npt
      import pandas as pd
      import pyvista as pv
      from IPython.display import Markdown

      from pyvirtualdisplay import Display
      import os
      display = Display(visible=0, size=(1280, 1024))
      display.start()

[1]: <pyvirtualdisplay.display.Display at 0x7ff96dbb9d90>
```

The review cases are as follows. GetFEM++ uses FEM_PRODUCT and IM_PRODUCT to two-dimensional the finite element method and the integration method, respectively. For quadratic elements, the Gaussian integration point is 3. IM_GAUSS1D(K) represents the integration point of $K/2 + 1$ points. The element uses plane strain elements. Set up these meshes, the finite element method and the integration method.

```
[2]: cases = [  
    "case11",  
    "case12",  
    "case13",  
    "case14",  
    "case21",  
    "case22",  
    "case23",  
    "case24",  
    "case31",  
    "case32",  
    "case33",  
    "case34",  
    "case41",  
    "case42",  
    "case43",  
    "case44",  
]
```

```
[3]: xs = [  
    4,  
    4,  
    4,  
    16,  
    4,  
    4,  
    4,  
    16,  
    4,  
    4,  
    4,  
    16,  
    4,  
    4,  
    4,  
    16,  
]  
ys = [  
    1,  
    2,  
    4,  
    8,  
    1,  
    2,  
    4,  
    8,  
    1,  
    2,  
    4,  
    8,  
    1,  
    2,  
    4,  
    8,  
]  
]
```



```
[4]: fem_names = [
    "FEM_PK(1, 2) ",
    "FEM_PK(1, 2) ",
    "FEM_PK(1, 2) ",
    "FEM_PK(1, 2) ",
    "FEM_PK(1, 1) ",
    "FEM_PK(1, 1) ",
    "FEM_PK(1, 1) ",
    "FEM_PK(1, 1) ",
    "FEM_PK(1, 1) ",
    "FEM_PK(1, 1) ",
    "FEM_PK(1, 1) ",
    "FEM_PK(1, 1) ",
    "FEM_PK(1, 1) ",
    "FEM_PK(1, 1) ",
    "FEM_PK_WITH_CUBIC_BUBBLE(1, 1) ",
    "FEM_PK_WITH_CUBIC_BUBBLE(1, 1) ",
    "FEM_PK_WITH_CUBIC_BUBBLE(1, 1) ",
    "FEM_PK_WITH_CUBIC_BUBBLE(1, 1) ",
]
```

```
[5]: methods = [
    "IM_GAUSS1D(4) ",
    "IM_GAUSS1D(4) ",
    "IM_GAUSS1D(4) ",
    "IM_GAUSS1D(4) ",
    "IM_GAUSS1D(2) ",
    "IM_GAUSS1D(2) ",
    "IM_GAUSS1D(2) ",
    "IM_GAUSS1D(2) ",
    "IM_GAUSS1D(0) ",
    "IM_GAUSS1D(0) ",
    "IM_GAUSS1D(0) ",
    "IM_GAUSS1D(0) ",
    "IM_GAUSS1D(0) ",
    "IM_GAUSS1D(4) ",
    "IM_GAUSS1D(4) ",
    "IM_GAUSS1D(4) ",
    "IM_GAUSS1D(4) ",
]
```

```
[6]: pd.options.display.float_format = "{:.2f}".format
data = []
columns = ["Case Name", "Mesh", "Finite Element Method", "Integration Method"]
for case, x, y, fem_name, method in zip(cases, xs, ys, fem_names, methods):
    data.append([case, str(x) + "x" + str(y), fem_name, method])
df = pd.DataFrame(data=data, columns=columns)
Markdown(df.to_markdown())
```

[6]:

	Case Name	Mesh	Finite Element Method	Integration Method
0	case11	4x1	FEM_PK(1, 2)	IM_GAUSS1D(4)
1	case12	4x2	FEM_PK(1, 2)	IM_GAUSS1D(4)
2	case13	4x4	FEM_PK(1, 2)	IM_GAUSS1D(4)
3	case14	16x8	FEM_PK(1, 2)	IM_GAUSS1D(4)
4	case21	4x1	FEM_PK(1, 1)	IM_GAUSS1D(2)
5	case22	4x2	FEM_PK(1, 1)	IM_GAUSS1D(2)
6	case23	4x4	FEM_PK(1, 1)	IM_GAUSS1D(2)
7	case24	16x8	FEM_PK(1, 1)	IM_GAUSS1D(2)
8	case31	4x1	FEM_PK(1, 1)	IM_GAUSS1D(0)
9	case32	4x2	FEM_PK(1, 1)	IM_GAUSS1D(0)
10	case33	4x4	FEM_PK(1, 1)	IM_GAUSS1D(0)
11	case34	16x8	FEM_PK(1, 1)	IM_GAUSS1D(0)
12	case41	4x1	FEM_PK_WITH_CUBIC_BUBBLE(1, 1)	IM_GAUSS1D(4)
13	case42	4x2	FEM_PK_WITH_CUBIC_BUBBLE(1, 1)	IM_GAUSS1D(4)
14	case43	4x4	FEM_PK_WITH_CUBIC_BUBBLE(1, 1)	IM_GAUSS1D(4)
15	case44	16x8	FEM_PK_WITH_CUBIC_BUBBLE(1, 1)	IM_GAUSS1D(4)

Mesh

The overall size of the model is $L = 10$ mm in length, $h = 1$ mm in height, and $b = 1$ mm in depth. In general, a slender ratio of 1: 10 is considered a beam element.

















```
[7]: L = 10.0
b = 1.0
h = 1.0
meshs = []
for case, x, y in zip(cases, xs, ys):
    X = np.arange(x + 1) * L / x
    Y = np.arange(y + 1) * h / y
    mesh = gf.Mesh("cartesian", X, Y)
    meshs.append(mesh)
    mesh.export_to_vtk("mesh_" + case + ".vtk", "ascii")
```

Outputs an image of each mesh.

```
[8]: p = pv.Plotter(shape=(4, 4))

for i in range(4):
    for j in range(4):
        p.subplot(i, j)
        mesh = pv.read("mesh_" + cases[i * 4 + j] + ".vtk")
        p.add_text(cases[i * 4 + j], font_size=10)
        p.add_mesh(mesh, color="tan", show_edges=True)

p.show(cpos="xy")
```

case11 	case12 	case13 	case14 
case21 	case22 	case23 	case24 
case31 	case32 	case33 	case34 
case41 	case42 	case43 	case44 

```
[8]: [(5.0, 0.5, 19.41486882686712),
      (5.0, 0.5, 0.0),
      (0.0, 1.0, 0.0)]
```

Region

Sets the area on the left side of the mesh where the Dirichlet condition is set. The right side sets the area for setting the Neumann condition.

```
[9]: TOP_BOUND = 1
      RIGHT_BOUND = 2
      LEFT_BOUND = 3
      BOTTOM_BOUND = 4

      for mesh in meshes:
          fb1 = mesh.outer_faces_with_direction([0.0, 1.0], 0.01)
          fb2 = mesh.outer_faces_with_direction([1.0, 0.0], 0.01)
          fb3 = mesh.outer_faces_with_direction([-1.0, 0.0], 0.01)
          fb4 = mesh.outer_faces_with_direction([0.0, -1.0], 0.01)
          mesh.set_region(TOP_BOUND, fb1)
          mesh.set_region(RIGHT_BOUND, fb2)
          mesh.set_region(LEFT_BOUND, fb3)
          mesh.set_region(BOTTOM_BOUND, fb4)
```

Finite Element Method

Create a MeshFem object and associate the mesh with the finite element method.

```
[10]: fems = []
      for fem_name in fem_names:
          fems.append(gf.Fem("FEM_PRODUCT(" + fem_name + ", " + fem_name + ")"))

[11]: mfus = []
      for mesh, fem in zip(meshs, fems):
          mfu = gf.MeshFem(mesh, 2)
          mfu.set_fem(fem)
          mfus.append(mfu)
```

Integral method

Associate the integration method with the mesh.

```
[12]: ims = []
      for method in methods:
          ims.append(gf.Integ("IM_PRODUCT(" + method + ", " + method + ")"))

[13]: mims = []
      for mesh, im in zip(meshs, ims):
          mim = gf.MeshIm(mesh, im)
          mims.append(mim)
```

Variable

Define the model object and set the variable “u”.

```
[14]: mds = []
      for mfu in mfus:
          md = gf.Model("real")
          md.add_fem_variable("u", mfu)
          mds.append(md)
```

Properties

Define properties as constants for the model object. The Young’s modulus of steel is $E = 205000 \times 10^6 N/m^2$. Also set Poisson’s ratio $\nu = 0.0$ to ignore the Poisson effect.

```
[15]: E = 10000 # N/mm2
      Nu = 0.0

      for md in mds:
          md.add_initialized_data("E", E)
          md.add_initialized_data("Nu", Nu)
```

Plane Strain Element

Defines the plane strain element for variable ‘u’.

```
[16]: for md, mim in zip(mds, mims):
    md.add_isotropic_linearized_elasticity_brick_pstrain(mim, "u", "E", "Nu")
```

Boundary Conditions

Set the Dirichlet condition for the region on the left side.

```
[17]: for (md, mim, mfu, fem) in zip(mds, mims, mfus, fems):
    if fem.is_lagrange():
        md.add_Dirichlet_condition_with_simplification("u", LEFT_BOUND)
    else:
        md.add_Dirichlet_condition_with_multipliers(mim, "u", mfu, LEFT_BOUND)
```

Set the Neumann boundary condition on the right side.

```
[18]: F = 1.0 # N/mm2
for (md, mfu, mim) in zip(mds, mfus, mims):
    md.add_initialized_data("F", [0, F / (b * h)])
    md.add_source_term_brick(mim, "u", "F", RIGHT_BOUND)
```

2.3.2 Solve

Solve the simultaneous equations of the model object to find the value of the variable 'u'.

```
[19]: for md in mds:
    md.solve()
```

The constraint on the left end has a displacement of 0.0.

```
[20]: for md, mfu, case in zip(mds, mfus, cases):
    u = md.variable("u")
    dof = mfu.basic_dof_on_region(LEFT_BOUND)
    npt.assert_almost_equal(abs(np.max(u[dof])), 0.0)
```

2.3.3 Review results

Output and visualize the results of each case to vtk files.

```
[21]: for md, mfu, case in zip(mds, mfus, cases):
    u = md.variable("u")
    mfu.export_to_vtk("u_" + case + ".vtk", "ascii", mfu, u, "u")
```

Computation of Theoretical Solutions

Calculates the deflection at each coordinate for comparison with the theoretical solution. The theoretical solution for a cantilever beam subjected to a concentrated load is as follows.

$$w(x) = \frac{FL^3}{3EI}$$

```
[22]: I = b * h ** 3 / 12
      w = F * L ** 3 / (3 * E * I)
      w
```

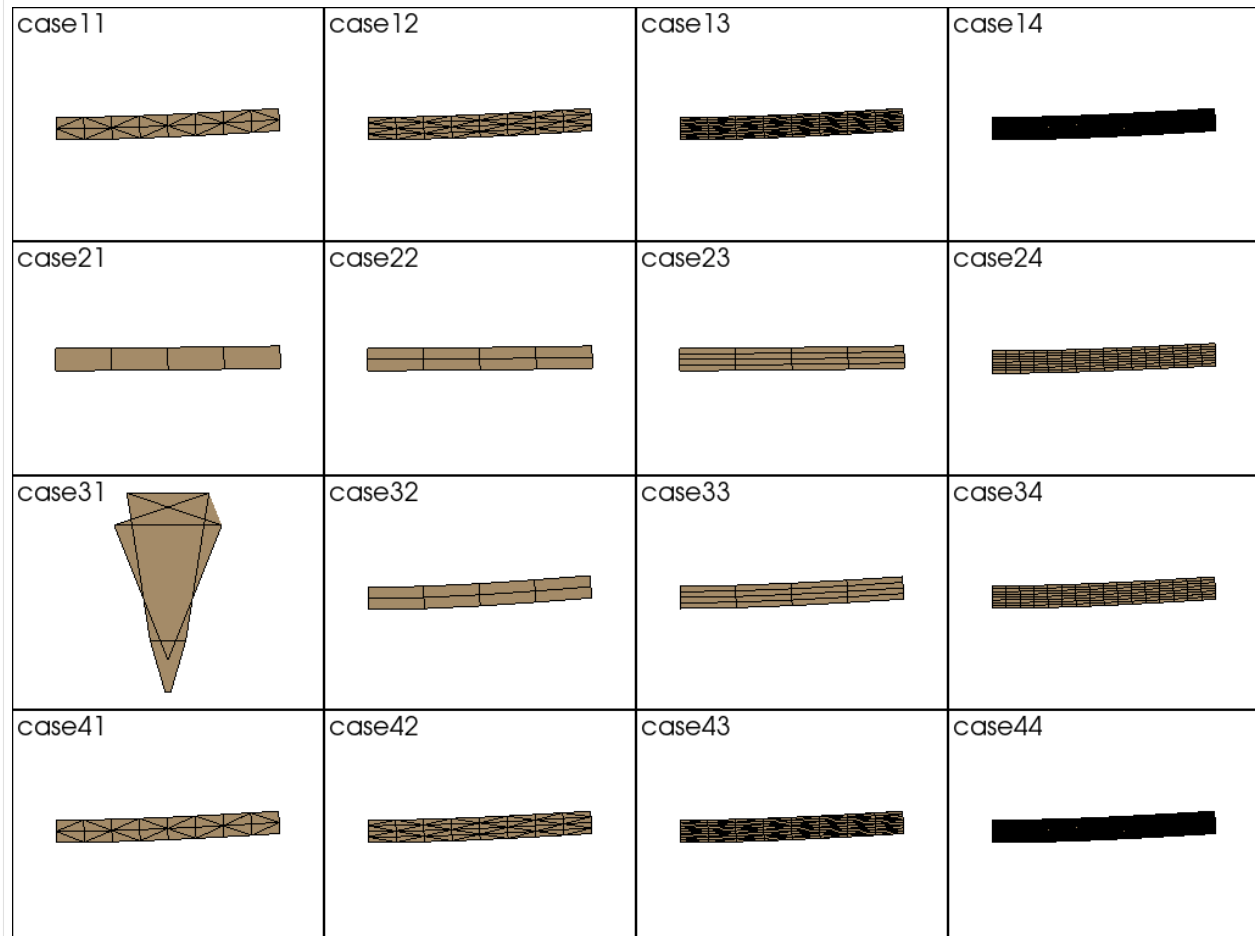
```
[22]: 0.4
```

deformation diagram of case11

```
[23]: p = pv.Plotter(shape=(4, 4))

for i in range(4):
    for j in range(4):
        p.subplot(i, j)
        mesh = pv.read("u_" + cases[i * 4 + j] + ".vtk")
        p.add_text(cases[i * 4 + j], font_size=10)
        p.add_mesh(mesh.warp_by_vector("u"), color="tan", show_edges=True)

p.show(cpos="xy")
```



```
[23]: [(5.015016078948975, 0.7012025117874146, 19.565021099812608),
      (5.015016078948975, 0.7012025117874146, 0.0),
      (0.0, 1.0, 0.0)]
```

Comparison with Theoretical Solutions

```
[24]: b = 1.0
      I = b * h ** 3.0 / 12.0
      dmax = 1.0 / 3.0 * (F * L ** 3) / (E * I)
      dmax
```

```
[24]: 0.4
```

Calculates the ratio of deformation to theoretical solution for each case.

```
[25]: data = []

      columns = [
          "case name",
          "mesh",
          "finite element method",
          "integration method",
          "ratio with theory",
      ]

      for case, x, y, fem_name, method, md, mfu in zip(
          cases, xs, ys, fem_names, methods, mds, mfus
      ):
          u = md.variable("u")
          dof = mfu.basic_dof_on_region(RIGHT_BOUND)
          ratio = "{:.3f}".format(max(u[dof] / dmax))
          data.append([case, str(x) + "x" + str(y), fem_name, method, ratio])
      df = pd.DataFrame(data=data, columns=columns)
      Markdown(df.to_markdown())
```

```
[25]:
```

	case name	mesh	finite element method	integration method	ratio with theory
0	case11	4x1	FEM_PK(1, 2)	IM_GAUSS1D(4)	0.999
1	case12	4x2	FEM_PK(1, 2)	IM_GAUSS1D(4)	1
2	case13	4x4	FEM_PK(1, 2)	IM_GAUSS1D(4)	1
3	case14	16x8	FEM_PK(1, 2)	IM_GAUSS1D(4)	1.006
4	case21	4x1	FEM_PK(1, 1)	IM_GAUSS1D(2)	0.244
5	case22	4x2	FEM_PK(1, 1)	IM_GAUSS1D(2)	0.244
6	case23	4x4	FEM_PK(1, 1)	IM_GAUSS1D(2)	0.244
7	case24	16x8	FEM_PK(1, 1)	IM_GAUSS1D(2)	0.841
8	case31	4x1	FEM_PK(1, 1)	IM_GAUSS1D(0)	6.52355e+10
9	case32	4x2	FEM_PK(1, 1)	IM_GAUSS1D(0)	1.317
10	case33	4x4	FEM_PK(1, 1)	IM_GAUSS1D(0)	1.056
11	case34	16x8	FEM_PK(1, 1)	IM_GAUSS1D(0)	1.021
12	case41	4x1	FEM_PK_WITH_CUBIC_BUBBLE(1, 1)	IM_GAUSS1D(4)	0.999
13	case42	4x2	FEM_PK_WITH_CUBIC_BUBBLE(1, 1)	IM_GAUSS1D(4)	1
14	case43	4x4	FEM_PK_WITH_CUBIC_BUBBLE(1, 1)	IM_GAUSS1D(4)	1
15	case44	16x8	FEM_PK_WITH_CUBIC_BUBBLE(1, 1)	IM_GAUSS1D(4)	1.006

```
[ ]:
```

2.4 Damping model with one degree of freedom

In this example of one degree of freedom with damping, we use here python interface, translate this program for another interface or in C++ is easy. We also explain about how to use GWFL(Generic Weak Form Language) in problem.

2.4.1 The probelm setting

A single truss element is used to verify the damping of a single-degree-of-freedom system. Node 1 is fully constrained, and the damped free vibration is obtained after giving an instantaneous forced displacement of 1 mm in the x direction of node 2.

2.4.2 Building the program

Let us begin by loading GetFEM and fixing the parameter of the problem.

```
[1]: import getfem as gf
import numpy as np

E = 1.0e05 # Young Modulus (N/mm^2)
rho = 8.9e-09 # Mass Density (ton/mm^3)
A = 1.0 # Cross-Sectional Area (mm^2)
L = 1000.0 # Length (mm)
```

We consider that the length of mesh is 1D and it has 1 convex. We generate the mesh of the one degree of freedom using empty mesh of GetFEM (see the documentation of the Mesh object in the python interface).

```
[2]: mesh = gf.Mesh("empty", 1)
```

```
[3]: mesh.add_convex?
```

```
Signature: mesh.add_convex(GT, PTS)
Docstring:
Add a new convex into the mesh.

The convex structure (triangle, prism,...) is given by `GT`
(obtained with GeoTrans('...')), and its points are given by
the columns of `PTS`. On return, `CVIDs` contains the convex #ids.
`PTS` might be a 3-dimensional array in order to insert more than
one convex (or a two dimensional array correctly shaped according
to Fortran ordering).
File:      /usr/lib/python3/dist-packages/getfem/getfem.py
Type:      method
```

```
[4]: cvid = mesh.add_convex(gf.GeoTrans("GT_PK(1, 1)"), [[0, L]])
```

We can check the mesh information using print function. We can see that the mesh has 2 point and 1 convex.


```
[5]: print(mesh)
```

```
BEGIN POINTS LIST

  POINT COUNT 2
  POINT  0  0
  POINT  1 1000

END POINTS LIST

BEGIN MESH STRUCTURE DESCRIPTION

  CONVEX COUNT 1
  CONVEX 0      'GT_PK(1,1)'      0  1

END MESH STRUCTURE DESCRIPTION
```

If you want to build a regular mesh quickly with multi convexs, we can use following Mesh object constructor.

```
[6]: mesh = gf.Mesh("cartesian", [0, L])
```

```
[7]: print(mesh)
```

```
BEGIN POINTS LIST

  POINT COUNT 2
  POINT  0  0
  POINT  1 1000

END POINTS LIST

BEGIN MESH STRUCTURE DESCRIPTION

  CONVEX COUNT 1
  CONVEX 0      'GT_PK(1,1)'      0  1

END MESH STRUCTURE DESCRIPTION
```

Boundary selection

We have to select the different parts of the boundary where we will set some boundary conditions, namely boundary of the fix boundary (0.0, 0.0, 0.0) and the deformed boundary of the (1000.0, 0.0, 0.0).

```
[8]: fb1 = mesh.outer_faces_with_direction([-1.0], 0.01)
      fb2 = mesh.outer_faces_with_direction([1.0], 0.01)

      LEFT = 1
      RIGHT = 2
```

(continues on next page)

(continued from previous page)

```
mesh.set_region(LEFT, fb1)
mesh.set_region(RIGHT, fb2)
```

Mesh draw

In order to preview the mesh and to control its validity, the following instructions can be used:

```
[9]: mesh.export_to_vtk("m.vtk")
```

An external graphical post-processor has to be used (for instance, pyvista).

```
[10]: import pyvista as pv
      from pyvirtualdisplay import Display

      display = Display(visible=0, size=(1280, 1024))
      display.start()
      p = pv.Plotter()
      m = pv.read("m.vtk")
      p.add_mesh(m, line_width=5)
      pts = m.points
      p.add_point_labels(pts, pts[:, 0].tolist(), point_size=10, font_size=10)
      p.show(window_size=[512, 384], cpos="xy")
      display.stop()
```



```
[10]: <pyvirtualdisplay.display.Display at 0x7feb22082ac0>
```

```
[11]: print(mesh)
```

```

BEGIN POINTS LIST

  POINT COUNT 2
  POINT  0  0
  POINT  1 1000

END POINTS LIST

BEGIN MESH STRUCTURE DESCRIPTION

  CONVEX COUNT 1
  CONVEX 0      'GT_PK(1,1)'      0  1

END MESH STRUCTURE DESCRIPTION
BEGIN REGION 1
0/1
END REGION 1
BEGIN REGION 2
0/0
END REGION 2

```

Definition of finite element methods and integration method

We will define three finite element methods. The first one, `mfu` is to approximate the displacement field. This is a vector field. This is defined in Python by

```
[12]: mfu = gf.MeshFem(mesh, 1)

elements_degree = 1
mfu.set_classical_fem(elements_degree)
```

```
[46]: print(mfu)
```

```

BEGIN MESH_FEM

QDIM 1
  CONVEX 0 'FEM_PK(1,1)'
  BEGIN DOF_ENUMERATION
    0:  0 1
  END DOF_ENUMERATION
END MESH_FEM

```

where the 1 stands for the dimension of the vector field. The second line sets the finite element used. `classical_fem` means a continuous Lagrange element and remember that `elements_degree` has been set to 1 which means that we will use quadratic (isoparametric) elements.

The last thing to define is an integration method `mim`. There is no default integration method in GetFEM so this is mandatory to define an integration method. Of course, the order of the integration method have to be chosen sufficient to make a convenient integration of the selected finite element method. Here, the square of `elements_degree` is sufficient.

```
[13]: mim = gf.MeshIm(mesh, elements_degree * 2)
```

```
[47]: print(mim)
```

```
BEGIN MESH_IM  
  
  CONVEX 0 'IM_GAUSS1D(2)'  
END MESH_IM
```

Model definition

The model object in *GetFEM* gather the variables of the models (the unknowns), the data and what is called the model bricks. The model bricks are some parts of the model (linear or nonlinear terms) applied on a single variable or linking several variables. They are used to make the assembly of the (tangent) linear system (see [The model object](#) for more details).

followingThis is not strictly mandatory to use the model object since one may use directly the assembly procedures and build by it own the (tangent) linear system. The model object allows a rapid build of the model since most classical parts of a model are pre-programmed: standard boundary conditions, standard partial differential equations, use of multipliers to prescribe a constraint ... Moreover, some bricks are designed to extend the possibilities of the standard bricks (generic assembly bricks, explicit matrix brick ...). Thus, it is recommended to use the framework of the model object.

There are two versions of the model: the real one and the complex one. Complex models have to be reserved for special applications (some electromagnetism problems for instance) where it is advantageous to solve a complex linear system.

Let us declare a real model with the one variables corresponding to the three fields to be computed:

```
[14]: model = gf.Model("real")  
model.add_fem_variable("u", mfu)
```

Truss deformation problem

Let us now begin by truss deformation problem. The equation on the deformation u and boundary condition can be written as follows:

$$\frac{d}{dx} \left\{ EA \frac{du}{dx} \right\} = F$$

is expressed also:

$$\nabla(D\nabla u) = F$$

Where $D = EA$.

We use directly a GWFL term `add_linear_term(md mim, "E*A*Grad_u.Grad_Test_u")`. See [Compute arbitrary terms - high-level generic assembly procedures - Generic Weak-Form Language \(GWFL\)](#) for more details on GWFL.

$$\int_0^L EA \frac{du(x)}{dx} \frac{dv(x)}{dx} dx = \int_0^L Fv(x) dx$$

v it the test function of u .

```
[15]: model.add_linear_term?
```

```
Signature: model.add_linear_term(mim, expression, region=None, *args)
Docstring:
Synopsis: ind = Model.add_linear_term(self, MeshIm mim, string expression[, int_
↳region[, int is_symmetric[, int is_coercive]])

Adds a matrix term given by the assembly string `expr` which will
be assembled in region `region` and with the integration method `mim`.
Only the matrix term will be taken into account, assuming that it is
linear.

The advantage of declaring a term linear instead of nonlinear is that
it will be assembled only once and no assembly is necessary for the
residual.

Take care that if the expression contains some variables and if the
expression is a potential or of first order (i.e. describe the weak
form, not the derivative of the weak form), the expression will be
derivated with respect to all variables.

You can specify if the term is symmetric, coercive or not.
If you are not sure, the better is to declare the term not symmetric
and not coercive. But some solvers (conjugate gradient for instance)
are not allowed for non-coercive problems.
`brickname` is an optional name for the brick.
File:      /usr/lib/python3/dist-packages/getfem/getfem.py
Type:      method
```

```
[16]: model.add_initialized_data("D", [E * A])
model.add_linear_term(mim, "(D*Grad_u).Grad_Test_u")
```

```
[16]: 0
```

Returned integer of `add_linear_term` show the index of the brick.

The equation of Model object will be:

$$\begin{bmatrix} \frac{EA}{L} & -\frac{EA}{L} \\ -\frac{EA}{L} & \frac{EA}{L} \end{bmatrix} \begin{Bmatrix} u_0 \\ u_1 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \end{Bmatrix}$$

You can assemble equation using `assembly` method. After assembling, you can check stiffness tangent matrix and rhs of equation using `tangent_matrix` and `rhs` method.

```
[17]: model.assembly()
SM = model.tangent_matrix()
rhs = model.rhs()
print(SM)
print(rhs)
```

```
matrix(2, 2)
( (r0, 100) (r1, -100) )
( (r0, -100) (r1, 100) )

[0. 0.]
```

```
[18]: SM = gf.Spmat("empty", 2, 2)
SM.add(0, 0, E * A / L)
SM.add(0, 1, -E * A / L)
```

(continues on next page)

(continued from previous page)

```

SM.add(1, 0, -E * A / L)
SM.add(1, 1, E * A / L)
print(SM)
# model.add_explicit_matrix("u", "u", SM)

matrix(2, 2)
( (r0, 100) (r1, -100) )
( (r0, -100) (r1, 100) )

```

The following program allows to take into account the whole elastic deformation equation. Note the use of specific brick to prescribe the Dirichlet condition on the left boundary. There is several option to prescribe a Dirichlet condition (see Dirichlet condition brick).

```
[19]: model.add_Dirichlet_condition_with_simplification?
```

```

Signature:
model.add_Dirichlet_condition_with_simplification(
    varname,
    region,
    dataname=None,
)
Docstring:
Adds a (simple) Dirichlet condition on the variable `varname` and
the mesh region `region`. The Dirichlet condition is prescribed by
a simple post-treatment of the final linear system (tangent system
for nonlinear problems) consisting of modifying the lines corresponding
to the degree of freedom of the variable on `region` (0 outside the
diagonal, 1 on the diagonal of the matrix and the expected value on
the right hand side).
The symmetry of the linear system is kept if all other bricks are
symmetric.
This brick is to be reserved for simple Dirichlet conditions (only dof
declared on the correspondning boundary are prescribed). The application
of this brick on reduced dof may be problematic. Intrinsic vectorial
finite element method are not supported.
`dataname` is the optional right hand side of the Dirichlet condition.
It could be constant (but in that case, it can only be applied to
Lagrange f.e.m.) or (important) described on the same finite
element method as `varname`.
Returns the brick index in the model.
File:      /usr/lib/python3/dist-packages/getfem/getfem.py
Type:      method

```

```
[20]: model.add_Dirichlet_condition_with_simplification("u", LEFT)
```

```
[20]: 1
```

The equation of Model object will be:

$$\begin{bmatrix} 1 & 0 \\ -\frac{EA}{L} & \frac{EA}{L} \end{bmatrix} \begin{Bmatrix} u_0 \\ u_1 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \end{Bmatrix}$$

```

[21]: model.assembly()
SM = model.tangent_matrix()

```

(continues on next page)

(continued from previous page)

```

rhs = model.rhs()
print(SM)
print(rhs)

matrix(2, 2)
( (r0, 1) )
( (r0, -100) (r1, 100) )

[0. 0.]

```

The following program allows to take into account the rhs of source force. Note the use of specific brick to prescribe the source term brick on the rhs of equation.

```
[22]: model.add_source_term_brick?
```

```

Signature: model.add_source_term_brick(mim, varname, dataexpr, region=None, *args)
Docstring:
Synopsis: ind = Model.add_source_term_brick(self, MeshIm mim, string varname, string_
↳dataexpr[, int region[, string directdataname]])

Add a source term to the model relatively to the variable `varname`.
The source term is
represented by `dataexpr` which could be any regular expression of the
high-level generic assembly language (except for the complex version
where it has to be a declared data of the model).
`region` is an optional mesh region
on which the term is added. An additional optional data `directdataname`
can be provided. The corresponding data vector will be directly added
to the right hand side without assembly. Note that when region is a
boundary, this brick allows to prescribe a nonzero Neumann boundary
condition. Return the brick index in the model.
File:      /usr/lib/python3/dist-packages/getfem/getfem.py
Type:      method

```

```
[23]: model.add_initialized_data("F", [1.0])
ind = model.add_source_term_brick(mim, "u", "F", RIGHT)
```

The equation of Model object will be:

$$\begin{bmatrix} \frac{1}{EA} & 0 \\ -\frac{EA}{L} & \frac{EA}{L} \end{bmatrix} \begin{Bmatrix} u_0 \\ u_1 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 1 \end{Bmatrix}$$

```
[24]: model.assembly()
SM = model.tangent_matrix()
rhs = model.rhs()
print(SM)
print(rhs)

matrix(2, 2)
( (r0, 1) )
( (r0, -100) (r1, 100) )

[0. 1.]

```

If you know the explicit rhs. You can add it using `add_explicit_rhs`:

```
[25]: model.add_explicit_rhs?
```

```
Signature: model.add_explicit_rhs(varname, L)
Docstring:
Add a brick representing an explicit right hand side to be added to
the right hand side of the tangent linear system relatively to the
variable `varname`. The given rhs should have the same size than the
dimension of `varname`. The rhs can be changed by the command
Model.set_private_rhs(). If `dataname` is specified instead of
`L`, the vector `L` is defined in the model as data with the given name.
Return the brick index in the model.
File:      /usr/lib/python3/dist-packages/getfem/getfem.py
Type:      method
```

```
[26]: # model.add_explicit_rhs("u", [0.0, 1.0])
```

Model solve

Once the model is correctly defined, we can simply solve it by:

```
[27]: model.solve()
```

```
[27]: (0, 1)
```

Export of the solution

The finite element problem is now solved. We can get the value of displacement variable using `variable` method.

```
[28]: U = model.variable("u")
print(U)
```

```
[0.   0.01]
```

The model is simple enough to make sure the results are correct.

```
[29]: U = [0.0, L / E * A]
print(U)
```

```
[0.0, 0.01]
```

Delete of brick

In the next section, we calculate a dynamic analysis, so we delete the static load brick. `delete_brick` method delete a variable or a data from the model.

```
[30]: model.delete_brick?
```

```
Signature: model.delete_brick(ind_brick)
Docstring: Delete a variable or a data from the model.
File:      /usr/lib/python3/dist-packages/getfem/getfem.py
Type:      method
```



```
[31]: model.delete_brick(ind)
```

The equation of Model object will be:

$$\begin{bmatrix} 1 & 0 \\ -\frac{EA}{L} & \frac{EA}{L} \end{bmatrix} \begin{Bmatrix} u_0 \\ u_1 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \end{Bmatrix}$$

```
[32]: model.assembly()
SM = model.tangent_matrix()
rhs = model.rhs()
print(SM)
print(rhs)
```

```
matrix(2, 2)
( (r0, 1) )
( (r0, -100) (r1, 100) )

[0. 0.]
```

The model tools for the integration of transient problems

Although time integration scheme can be written directly using the model object by describing the problem to be solved at each iteration, the model object furnishes some basic tools to facilitate the writing of such schemes. For example you can add Newmark scheme using `add_Newmark_scheme` method:

```
[33]: beta = 1.0 / 4.0
gamma = 1.0 / 2.0
model.add_Newmark_scheme("u", beta, gamma)
```

We use average constant acceleration ($\beta = \frac{1}{4}, \gamma = \frac{1}{2}$) which is unconditionally stable in linear model.

Some intermediate variables are added to the model to represent the time derivative (and the second order time derivative for second order problem). For instance, if `u` is a variable, `Dot_u` will represent the first order time derivative of `u` and `Dot2_u` the second order one. One can refer to these variables in the model to add a brick on it or to use it in GWFL, the generic weak form language. However, these are not considered to be independent variables, they will be linked to their corresponding original variable (in an affine way) by the time integration scheme. Most of the schemes need also the time derivative at the previous time step and add the data `Previous_Dot_u` and possibly `Previous_Dot2_u` to the model.

Some data are added to the model to represent the state of the system at previous time steps. For classical one-step schemes (for the moment, only one-step schemes are provided), only the previous time step is stored. For instance if `u` is a variable (thus represented at step n), `Previous_u`, `Previous2_u`, `Previous3_u` will be the data representing the state of the variable at the previous time step (step $n - 1$, $n - 2$ and $n - 3$).

Mass matrix

The element consistent mass matrix is given by the following equation using the mass density and the test function of the element:

$$M = \int_0^L \rho u \cdot v dx$$

Mass brick adds a mass matrix on the tangent linear system with respect to a certain variable. The function which adds this brick to a model is:

```
[34]: model.add_mass_brick?

Signature: model.add_mass_brick(mim, varname, dataexpr_rho=None, *args)
Docstring:
Synopsis: ind = Model.add_mass_brick(self, MeshIm mim, string varname[, string_
↳dataexpr_rho[, int region]])

Add mass term to the model relatively to the variable `varname`.
If specified, the data `dataexpr_rho` is the
density (1 if omitted). `region` is an optional mesh region on
which the term is added. If it is not specified, it
is added on the whole mesh. Return the brick index in the model.
File:      /usr/lib/python3/dist-packages/getfem/getfem.py
Type:      method
```

```
[35]: # model.add_initialized_data("rho", rho)
# model.add_mass_brick(mim, "Dot2_u", "rho")
```

Of course we can also add the mass matrix using Generic Weak-Form Language (GWFL).

```
[36]: model.add_initialized_data("rho", [rho])
model.add_linear_term(mim, "rho*Dot2_u.Test_Dot2_u")
```

```
[36]: 2
```

Explicit mass matrix is

$$M = \frac{\rho AL}{6} \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$$

And the equation is

$$\begin{bmatrix} 1 & 0 \\ \frac{\rho AL}{6} & \frac{\rho AL}{6} \end{bmatrix} \left\{ \begin{array}{c} \frac{\partial^2 u_0}{\partial t^2} \\ \frac{\partial^2 u_1}{\partial t^2} \end{array} \right\} + \begin{bmatrix} 1 & 0 \\ -\frac{EA}{L} & \frac{EA}{L} \end{bmatrix} \left\{ \begin{array}{c} u_0 \\ u_1 \end{array} \right\} = \left\{ \begin{array}{c} 0 \\ 0 \end{array} \right\}$$

Angular frequency without damping of this model is:

$$\omega_u = \sqrt{\frac{k}{m}}$$

$$\omega_d = \frac{\sqrt{4mk - c^2}}{2m}$$

with

$$m = \frac{\rho AL}{3}$$

$$k = \frac{EA}{L}$$

Damping matrix

The damping matrix is given by the following equation using the viscous damping μ and the test function v .

$$C = \int_0^L \mu u \cdot v dx$$

However it is hard to compute the μ , we usually use Rayleigh damping.

$$C = \alpha \cdot M + \beta \cdot K$$

```
[37]: m = rho * A * L / 3.0
      k = E * A / L
      omega_u = np.sqrt(k / m)
      omega_u
```

```
[37]: 5805.8474978713775
```

```
[38]: h = 0.02
      alpha = 2.0 * omega_u * h
      beta = 2.0 * h / omega_u * h
      c = alpha * m + beta * k
      omega_d = np.sqrt(4.0 * m * k - c ** 2) / (2.0 * m)
      omega_d
```

```
[38]: 5804.639291409139
```

we can also add the matrix using Generic Weak-Form Language (GWFL) to `Dot_u`.

```
[39]: model.add_initialized_data("alpha", alpha)
      model.add_initialized_data("beta", beta)
      model.add_linear_term(
          mim, "alpha*rho*Dot_u.Test_Dot_u+beta*D*(Grad_Dot_u.Grad_Test_Dot_u)"
      )
```

```
[39]: 3
```

The equation is

$$\begin{bmatrix} \frac{1}{6} & 0 \\ \frac{\rho AL}{6} & \frac{\rho AL}{3} \end{bmatrix} \begin{Bmatrix} \frac{\partial^2 u_0}{\partial t^2} \\ \frac{\partial^2 u_1}{\partial t^2} \end{Bmatrix} + \left(\alpha \begin{bmatrix} \frac{1}{6} & 0 \\ \frac{\rho AL}{6} & \frac{\rho AL}{3} \end{bmatrix} + \beta \begin{bmatrix} \frac{1}{L} & 0 \\ -\frac{EA}{L} & \frac{EA}{L} \end{bmatrix} \right) \begin{Bmatrix} \frac{\partial u_0}{\partial t} \\ \frac{\partial u_1}{\partial t} \end{Bmatrix} + \begin{bmatrix} \frac{1}{L} & 0 \\ -\frac{EA}{L} & \frac{EA}{L} \end{bmatrix} \begin{Bmatrix} u_0 \\ u_1 \end{Bmatrix} = \begin{Bmatrix} 0 \\ 0 \end{Bmatrix}$$

The implicit Newmark scheme for second order problems

The equation of Newmark theme is following (n is the step of time):

$$\left(K + \frac{\gamma}{\beta \Delta t} C + \frac{1}{\beta \Delta t^2} M \right) u_{n+1} = f_{n+1} + M \left(\left(\frac{1}{2\beta} - 1 \right) \frac{\partial^2 u_n}{\partial t^2} + \frac{1}{\beta \Delta t} \frac{\partial u_n}{\partial t} + \frac{1}{\beta \Delta t^2} u_n \right) + C \left(\left(\frac{\gamma}{2\beta} - 1 \right) \Delta t_n \frac{\partial^2 u_n}{\partial t^2} + \left(\frac{\gamma}{\beta} - 1 \right) \frac{\partial u_n}{\partial t} + u_n \right)$$

We can set the time step dt using `set_time_step` method.

```
[40]: model.set_time_step?

Signature: model.set_time_step(dt)
Docstring:
Set the value of the time step to `dt`. This value can be change
from a step to another for all one-step schemes (i.e for the moment
to all proposed time integration schemes).
File:      /usr/lib/python3/dist-packages/getfem/getfem.py
Type:      method
```

It is generally recommended to use $1/20$ of the desired period for the time step. So we use $1/20$ of the natural period of the model.

```
[41]: T0 = 2.0*np.pi*np.sqrt(m/k)
      dt = T0 / 20.0
      model.set_time_step(dt)
```

We consider the case where the left side is fully constrained and the right side is subjected to a prescribed displacement of 1 mm.

```
[42]: U0 = np.array([0.0, 1.0]) # mm
      V0 = np.array([0.0, 0.0]) # mm/sec
      A0 = np.array([0.0, 0.0]) # mm/sec^2
      model.set_variable("Previous_u", U0)
      model.set_variable("Previous_Dot_u", U0)
      model.set_variable("Previous_Dot2_u", A0)
```

Typically, the solve on the different time steps will take the following form:

```
[43]: T = np.array([])
      U = np.array([])
      for t in np.arange(0.0, 0.004, dt):
          model.solve()
          T = np.append(T, model.get_time())
          U = np.append(U, model.variable("u")[1])
          model.shift_variables_for_time_integration()
```

Note that the call of the method:

```
[44]: model.shift_variables_for_time_integration?

Signature: model.shift_variables_for_time_integration()
Docstring:
Function used to shift the variables of a model to the data
corresponding of their value on the previous time step for time
integration schemes. For each variable for which a time integration
scheme has been declared, the scheme is called to perform the shift.
This function has to be called between two time steps.
File:      /usr/lib/python3/dist-packages/getfem/getfem.py
Type:      method
```

is needed between two time step since it will copy the current value of the variables (u and Dot_u for instance) to the previous ones (Previous_u and Previous_Dot_u).

The comparison between the damping rate of the analytical solution and the FEM amplitude is plotted in the bottom figure. Both are in good agreement.

$$u(t + nT) = u(t) * \exp\left(-2\pi n h \frac{\omega_u}{\omega_d}\right)$$

```
[45]: import matplotlib.pyplot as plt

      fig = plt.figure()
      ax = fig.add_subplot(1, 1, 1)
      ax.plot(T, U, label="FEM", color="black")
      n = np.arange(0, 5)
```

(continues on next page)

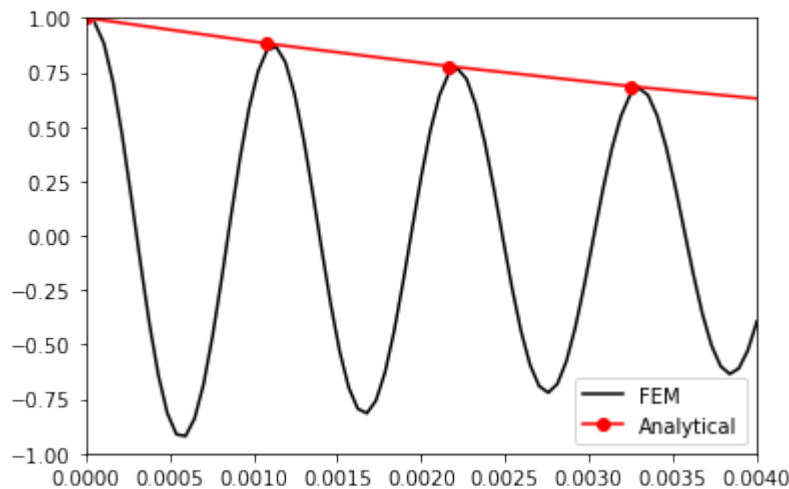
(continued from previous page)

```

ax.plot(
    n * 2.0 * np.pi / omega_d,
    1.0 * np.exp(-2.0 * np.pi * n * h * omega_u / omega_d),
    color="red",
    marker="o",
    label="Analytical",
)
ax.set_xlim(0.000, 0.004)
ax.set_ylim(-1.000, 1.000)
ax.legend()

```

[45]: <matplotlib.legend.Legend at 0x7feaae70cfd0>



This is a small model example, but you can calculate the more complicated element in the same way.

[]:

2.5 Assembly examples in Python

This is the use with Python interface of the C++ assembly examples in Compute arbitrary terms - high-level generic assembly procedures - Generic Weak-Form Language (GWFL).

```

[1]: import getfem as gf
import numpy as np

elements_degree = 1
# elements_degree = 2

mesh = gf.Mesh("cartesian", np.arange(0.0, 2.0, 1.0))

```

mf is supposed to be an already declared `gf.MeshFem` object and mim a already declared `gf.MeshIm` object on the same mesh.

```

[2]: mf = gf.MeshFem(mesh, 1)
mf.set_classical_fem(elements_degree)
print(mf)

```

```
BEGIN MESH_FEM

QDIM 1
  CONVEX 0 'FEM_PK(1,1)'
  BEGIN DOF_ENUMERATION
    0: 0 1
  END DOF_ENUMERATION
END MESH_FEM
```

```
[3]: mim = gf.MeshIm(mesh, elements_degree*2)
      print(mim)
```

```
BEGIN MESH_IM

  CONVEX 0 'IM_GAUSS1D(2)'
END MESH_IM
```

As a first example, if one needs to perform the assembly of a Poisson problem

$$-\operatorname{div} \nabla u = f, \text{ in } \Omega,$$

the stiffness matrix is given

$$K_{i,j} = \int_{\Omega} \nabla \varphi_i \cdot \nabla \varphi_j dx,$$

and will be assembled by the following code:

```
[4]: md = gf.Model("real")
      md.add_fem_variable("u", mf)
      md.add_nonlinear_term(mim, "Grad_u.Grad_Test_u")
      md.assembly(option="build_matrix")
      K = md.tangent_matrix()
      print(K)

matrix(2, 2)
( (r0, 1) (r1, -1) )
( (r0, -1) (r1, 1) )
```

Note that the value of the variable do not really intervene because of the linearity of the problem. This allows to pass "u" as the value of the variable which will not be used. Note also that two other possible expressions for exactly the same result for the assembly string are "Grad_Test2_u.Grad_Test_u" (i.e. an order 2 expression) or "Norm_sqr(Grad_u)/2" (i.e. a potential). In fact other possible assembly string will give the same result such as "Grad_u.Grad_u/2" or "[Grad_u(1), Grad_u(2)].[Grad_Test_u(1), Grad_Test_u(2)]" for two-dimensional problems. However, the recommendation is preferably to give an order 1 expression (weak formulation) if there is no particular reason to prefer an order 0 or an order 2 expression.

```
[5]: md = gf.Model("real")
      md.add_fem_variable("u", mf)
      md.add_nonlinear_term(mim, "Grad_u.Grad_u/2")
      md.assembly(option="build_matrix")
      K = md.tangent_matrix()
      print(K)
```

```
matrix(2, 2)
( (r0, 1) (r1, -1) )
( (r0, -1) (r1, 1) )
```

```
[6]: # for two-dimensional problems
# md = gf.Model("real")
# md.add_fem_variable("u", mf)
# md.add_nonlinear_term(mim, "[Grad_u(1), Grad_u(2)].[Grad_Test_u(1), Grad_Test_u(2)]"
# → ")
# md.assembly(option="build_matrix")
# K = md.tangent_matrix()
# print(K)
```

As a second example, let us consider a coupled problem, for instance the mixed problem of incompressible elasticity given by the equations

$$\begin{aligned} -\operatorname{div}(\mu(\nabla u + (\nabla u)^T) - pI_d) &= f, \text{ in } \Omega, \\ -\operatorname{div} u &= 0. \end{aligned}$$

where u is the vector valued displacement and p the pressure. The assembly of the matrix for the whole coupled system can be performed as follows:

```
[7]: epsilon = 1.; E = 21E6; nu = 0.3;
clambda = E*nu/((1+nu)*(1-2*nu));
cmu = E/(2*(1+nu));

mf_u = gf.MeshFem(mesh, 1)
mf_u.set_classical_fem(elements_degree)
mf_p = gf.MeshFem(mesh, 1)
mf_p.set_classical_fem(elements_degree)

md = gf.Model("real")
md.add_fem_variable("u", mf_u)
md.add_fem_variable("p", mf_p)
md.add_initialized_data("mu", cmu)
md.add_nonlinear_term(mim, "2*mu*Sym(Grad_u):Grad_Test_u"
    "- p*Trace(Grad_Test_u) - Test_p*Trace(Grad_u)")
md.assembly(option="build_matrix")
K = md.tangent_matrix()
print(K)

matrix(4, 4)
( (r2, 0.5) (r3, -0.5) )
( (r2, 0.5) (r3, -0.5) )
( (r0, 0.5) (r1, 0.5) (r2, 1.61538e+07) (r3, -1.61538e+07) )
( (r0, -0.5) (r1, -0.5) (r2, -1.61538e+07) (r3, 1.61538e+07) )
```

where, here, mf_u and mf_p are supposed to be some already declared `getfem::mesh_fem` objects defined on the same mesh, mim a already declared `getfem::mesh_im` object and μ is the Lamé coefficient. It is also possible to perform the assembly of the sub-matrix of this system separately.

Let us see now how to perform the assembly of a source term. The weak formulation of a volumic source term is

Let us see now how to perform the assembly of a source term. The weak formulation of a volumic source term is

$$\int_{\Omega} f v dx$$

where f is the source term and v the test function. The corresponding assembly can be written:

```
[8]: F = np.ones(mf_u.nbdof())

md = gf.Model("real")
md.add_fem_variable("u", mf_u)
md.add_initialized_fem_data("f", mf_u, F)
md.add_nonlinear_term(mim, "f*Test_u")
md.assembly("build_rhs")
rhs = md.rhs()
print(rhs)

[-0.5 -0.5]
```

if the source term is describe on a finite element `mf_data` and the corresponding vector of degrees of freedom `F`. Explicit source terms are also possible. For instance:

```
[9]: region = -1 # ALL
md = gf.Model("real")
md.add_fem_variable("u", mf_u)
md.add_nonlinear_term(mim, "sin(X(1))*Test_u", region)
# for two-dimensional problems
# md.add_nonlinear_term(mim, "sin(X(1)+X(2))*Test_u", region)
md.assembly("build_rhs")
rhs = md.rhs()
print(rhs)

[-0.15767352 -0.30191429]
```

where `region` is the mesh region number.

As another example, let us describe a simple nonlinear elasticity problem. Assume that we consider a Saint-Venant Kirchhoff constitutive law which means that we consider the following elastic energy on a body of reference configuration Ω :

$$\int_{\Omega} \frac{\lambda}{2} (\text{tr}(E))^2 + \mu \text{tr}(E^2) dx$$

where λ, μ are the Lamé coefficients and E is the strain tensor given by $E = (\nabla u + (\nabla u)^T + (\nabla u)^T \nabla u) / 2$.

This is possible to perform the assembly of the corresponding tangent problem as follows:

```
[10]: md = gf.Model("real")

md.add_fem_variable("u", mf_u)
md.add_initialized_data("lambda", clambda)
md.add_initialized_data("mu", cmu)
md.add_nonlinear_term(mim, "lambda*sqr(Trace(Grad_u+Grad_u'+Grad_u'*Grad_u)) "
                      "+ mu*Trace((Grad_u+Grad_u'+Grad_u'*Grad_u) "
                      "* (Grad_u+Grad_u'+Grad_u'*Grad_u)) ")
md.assembly("build_rhs")
rhs = md.rhs()
print(rhs)
md.assembly("build_matrix")
print(K)

[0. 0.]
matrix(4, 4)
( (r2, 0.5) (r3, -0.5) )
( (r2, 0.5) (r3, -0.5) )
```

(continues on next page)

(continued from previous page)

```
( (r0, 0.5) (r1, 0.5) (r2, 1.61538e+07) (r3, -1.61538e+07) )
( (r0, -0.5) (r1, -0.5) (r2, -1.61538e+07) (r3, 1.61538e+07) )
```

and to adapt a Newton-Raphson algorithm to solve that nonlinear problem. Of course the expression is rather repetitive and it would be preferable to define some intermediate nonlinear operators. However, note that repeated expressions are automatically detected and computed only once in the assembly.

The last example is the assembly of the stiffness matrix of an order four problem, the Kirchhoff-Love plate problem:

```
[11]: h = 1.0 # mm
      D = 2.0*h**3*E/(3*(1-nu**2))

      md = gf.Model("real")

      md.add_fem_variable("u", mf)
      md.add_initialized_data("D", D)
      md.add_initialized_data("nu", nu)
      md.add_nonlinear_term(mim, "D*(1-nu)*(Hess_u:Hess_Test_u) - "
                           "D*nu*Trace(Hess_u)*Trace(Hess_Test_u)")
      md.assembly(option="build_all")
      K = md.tangent_matrix()
```

with D the flexion modulus and ν the Poisson ratio.

```
[ ]:
```