pBEAM Documentation

Release 0.1.0

S.A. Ning

Technical Report
NREL/TP-5000-58818
September 2013

Contract No. DE-AC36-08GO28308

NATIONAL RENEWABLE ENERGY LABORATORY

pBeam Documentation

Release 0.1.0

S.A. Ning

Prepared under Task No. WE11.0341

National Renewable Energy Laboratory
15013 Denver West Parkway
Golden, CO 80401
303-275-3000 • www.nrel.gov

Technical Report
NREL/TP-5000-58818
September 2013

Contract No. DE-AC36-08GO28308

Cover Photos: (left to right) photo by Pat Corkery, NREL 16416, photo from SunEdison, NREL 17423, photo by Pat Corkery, NREL 16560, photo by Dennis Schroeder, NREL 17613, photo by Dean Armstrong, NREL 17436, photo by Pat Corkery, NREL 17721.

Printed on paper containing at least 50% wastepaper, including 10% post consumer waste.

Table of Contents

List of Figures

List of Tables

1 Introduction

The Polynomial Beam Element Analysis Module (pBEAM) is a finite element code for beam-like structures. The methodology uses Euler-Bernoulli beam elements with 12 degrees of freedom (3 translation and 3 rotational at each end of the element). The basic theory is described in Yang (1986). A unique feature of the code is that section properties can be described as polynomials of any order between nodes (e.g., EIxx could vary quarticly across an element). This allows for higher fidelity in describing variation in structural properties, which means higher accuracy can be achieved with fewer elements. The use of polynomials also allows for higher accuracy because integrals and derivatives are evaluated analytically rather than numerically and have greater flexibility in extending the code by using higher-order shape functions.

pBEAM was originally written to analyze tower/monopiles and rotor blades of wind turbines; however, the approach is general enough to be used for any beam-like structure. pBEAM is written in C++ and can be used directly in C++ or imported as a Python module. The Python module exposes the full class structure of the finite element code, allowing for flexible usage in a object-oriented scripting environment, while the analysis retains the speed advantage of compiled C++. If desired the C++ code could easily be adapted to run as a stand-alone command-line executable that reads data from an input file; however, that type of usage is not included.

pBEAM can estimate structural mass, deflections in all degrees of freedom, coupled natural frequencies, critical global axial buckling loads, and axial stress/strain. All inputs and outputs are given about the elastic center and in principal axes in order to remove cross-coupling terms. Any arbitrary definition of the structural properties can be translated to the elastic center and rotated to the principal axes (see Hansen (2008) for example).

2 Installation

Prerequisites

C++ compiler, Boost C++ Libraries: specifically boost_python-mt, boost_system-mt (and boost_unit_test_framework-mt if you want to run the unit tests), LAPACK, NumPy, and SciPy

---

Download either pBEAM.py-0.1.0.tar.gz or pBEAM.py-0.1.0.zip, and uncompress/unpack it.

Install pBEAM with the following command.

```
$ python setup.py install
```

To verify that the installation was successful, run Python from the command line,

```
$ python
```

and import the module. If no errors are issued, the installation was successful.

```
>>> import _pBEAM
```

pBEAM has a large range of unit tests, but they are only accessible through C++. These tests verify the integrity of the underlying C++ code for development purposes. If you want to run the tests, change the working directory to src/twister/rotorstruc/pBEAM and run

```
$ make test CXX=g++
```

where the name of your C++ compiler should be inserted in the place of g++. The script will build the test executable and run all tests. The phrase "No errors detected" signifies that all the tests passed.

To access an HTML version of this documentation, which contains further details and links to the source code, open docs/index.html.

3 Tutorial

Two examples are included in this section. The first example provides the sectional properties and assumes a linear variation between the sections (see matching *constructor*). The example simulates a blade for the NREL 5-MW reference model. The second example uses the convenience constructor for a beam with cylindrical shell sections (see matching *constructor*). The example simulates the tower for the NREL 5-MW refernece model. A third *constructor* is available for more advanced usage and allows for arbitrary polynomial variation in section properties. This advanced constructor is not demonstrated in these examples, but details are available in the :ref:'documentation <documentation-label>.

3.1 Linear Variation

This example simulates a rotor blade from the NREL 5-MW reference model in pBEAM. First, the relevant modules are imported.

```
import numpy as np
import matplotlib.pyplot as plt

import _pBEAM
```

Next, we define the stiffness and inertial properties. The stiffness and inertial properties can be computed from the structural layout of the blade using a code like PreComp. Section properties are defined using the *SectionData* class.

```
# stiffness / inertial properties

r = np.array([1.501, 1.803, 1.902, 2, 2.105, 2.203, 2.302, 2.868, 3.003, 3.102,
              5.607, 7.005, 8.34, 10.51, 11.76, 13.51, 15.86, 18.52, 19.97,
              22.02, 24.07, 26.12, 28.17, 32.28, 33.53, 36.39, 38.53, 40.49,
              42.54, 43.54, 44.59, 46.54, 48.7, 52.8, 56.22, 58.95, 61.69, 63.06])
EA = np.array([1.626e+10, 1.65e+10, 1.677e+10, 1.685e+10, 1.694e+10, 1.702e+10,
               1.625e+10, 1.208e+10, 1.212e+10, 1.23e+10, 1.228e+10, 1.199e+10,
               1.16e+10, 8.27e+09, 8.035e+09, 8.42e+09, 8.754e+09, 8.972e+09,
               9.058e+09, 9.128e+09, 9.122e+09, 8.947e+09, 8.747e+09, 8.295e+09,
               8.088e+09, 7.532e+09, 7.09e+09, 6.645e+09, 6.079e+09, 5.809e+09,
               5.515e+09, 4.955e+09, 4.353e+09, 3.186e+09, 2.221e+09, 1.503e+09,
               9.426e+08, 7.459e+08])
EI11 = np.array([2.352e+10, 2.352e+10, 2.491e+10, 2.491e+10, 2.491e+10, 2.491e+10,
                 2.372e+10, 1.659e+10, 1.751e+10, 1.821e+10, 1.456e+10, 1.336e+10,
                 1.065e+10, 4.918e+09, 4.769e+09, 4.331e+09, 4.044e+09, 3.423e+09,
                 3.167e+09, 2.873e+09, 2.572e+09, 1.969e+09, 1.58e+09, 1.296e+09,
                 1.101e+09, 7.37e+08, 6.386e+08, 5.837e+08, 4.358e+08, 3.622e+08,
                 3.096e+08, 2.541e+08, 2.034e+08, 1.22e+08, 7.101e+07, 3.817e+07,
                 7.727e+06, 3.664e+06])
EI22 = np.array([2.307e+10, 2.307e+10, 2.328e+10, 2.328e+10, 2.328e+10, 2.328e+10,
                 2.217e+10, 1.555e+10, 1.524e+10, 1.495e+10, 1.848e+10, 1.69e+10,
                 1.342e+10, 6.219e+09, 5.231e+09, 5.255e+09, 5.286e+09, 4.943e+09,
                 4.699e+09, 4.421e+09, 4.142e+09, 3.674e+09, 3.241e+09, 2.603e+09,
                 2.397e+09, 1.995e+09, 1.756e+09, 1.558e+09, 1.292e+09, 1.215e+09,
                 1.107e+09, 9.199e+08, 7.931e+08, 5.797e+08, 4.296e+08, 3.009e+08,
                 9.361e+07, 4.69e+07])
GJ = np.array([1.079e+10, 1.128e+10, 1.182e+10, 1.199e+10, 1.217e+10, 1.234e+10,
               1.192e+10, 8.915e+09, 9.135e+09, 9.261e+09, 7.303e+09, 4.995e+09,
               2.821e+09, 5.891e+08, 4.595e+08, 4.022e+08, 3.556e+08, 2.913e+08,
               2.697e+08, 2.414e+08, 2.157e+08, 1.689e+08, 1.415e+08, 1.191e+08,
               1.019e+08, 7.086e+07, 6.037e+07, 5.437e+07, 4.033e+07, 3.455e+07,
               3.002e+07, 2.53e+07, 2.128e+07, 1.433e+07, 9.588e+06, 6.455e+06,
               4.335e+06, 3.017e+06])
rhoA = np.array([1086, 1102, 1121, 1126, 1132, 1137, 1086, 878.6, 891.2, 896.1,
                 730.2, 608.9, 559.4, 337.1, 332.3, 333.3, 331.2, 324.2, 320.7,
                 315, 308, 271.6, 261.2, 244.8, 237, 218.2, 206.2, 194.8, 157,
                 151, 144.5, 133.1, 122.1, 100.6, 82.32, 67.6, 46.81, 31.18])
rhoJ = np.array([2778, 2905, 3048, 3092, 3139, 3184, 3075, 2415, 2532, 2566,
                 2119, 1595, 1255, 576.9, 558, 554.2, 534, 488.8, 467.5, 436.7,
                 404.6, 331.1, 295.2, 243.2, 222.8, 180.2, 155.6, 135.1, 92.91,
                 85.2, 76.85, 63.57, 52.86, 35.45, 23.82, 16.32, 8.66, 6.038])


# number of sections
nsec = len(r)

p_section = _pBEAM.SectionData(nsec, r, EA, EI11, EI22, GJ, rhoA, rhoJ)
```

Distributed loads can be computed from an aerodynamics code like CCBlade. This example includes only distributed loads, which are defined in *Loads*.

```
# distributed loads

P1 = np.array([106.8, 487.3, 604.7, 715.4, 826.2, 923.1, 1007, 1299, 1326,
               1343, 1440, 1337, 1108, -1252, -2185, -2017, -1736, -1727,
               -1718, -1698, -1665, -1419, -1131, -1051, -937, -673.4,
               -643.8, -617, -553.7, -524.2, -504.1, -484.4, -462.8,
               -417, -374.3, -336.6, -295.3, -0.01844])
P2 = np.array([452.4, 2064, 2561, 3030, 3499, 3909, 4263, 5499, 5612,
               5686, 6098, 5291, 4690, 1.938e+04, 2.563e+04, 2.492e+04,
               2.387e+04, 2.346e+04, 2.308e+04, 2.202e+04, 2.097e+04,
               2.021e+04, 1.959e+04, 1.865e+04, 1.844e+04, 1.796e+04,
               1.734e+04, 1.668e+04, 1.592e+04, 1.556e+04, 1.518e+04,
               1.452e+04, 1.379e+04, 1.23e+04, 1.096e+04, 9802, 8552,
               13.34])
P3 = np.array([-1.065e+04, -1.081e+04, -1.098e+04, -1.104e+04, -1.109e+04,
               -1.114e+04, -1.064e+04, -8611, -8734, -8782, -7156, -5967,
               -5482, -3303, -3256, -3267, -3246, -3177, -3143, -3087,
               -3019, -2662, -2560, -2399, -2323, -2138, -2021, -1909,
               -1539, -1480, -1416, -1305, -1197, -985.7, -806.8, -662.5,
               -458.7, -305.5])



p_loads = _pBEAM.Loads(nsec, P1, P2, P3)  # only distributed loads
```

The tip/base data is defined with a free end in *TipData* and a rigid base for *BaseData*. The blade object is then assembled using the *Beam* constructor that assumes linear variation in properties between sections.

```
# tip/base data
p_tip = _pBEAM.TipData()  # no tip mass
k = np.array([float('inf'), float('inf'), float('inf'),
              float('inf'), float('inf'), float('inf')])
p_base = _pBEAM.BaseData(k, float('inf'))  # rigid base
```

```
# create blade object
blade = _pBEAM.Beam(p_section, p_loads, p_tip, p_base)
```

The constructed blade object can now be used for various computations. For example, the mass and the first five natural frequencies

```
# compute mass and natural frequncies
print 'mass =', blade.mass()
print 'natural freq =', blade.naturalFrequencies(5)
```

which result in

```
>>> mass = 17170.0189
>>> natural freq = [ 0.90910346  1.13977516  2.81855826  4.23836926  6.40037864]
```

Figure 1 shows a plot of the blade displacement in the second principal direction generated from the following commands.

```
# plot displacement in second principal direction (approximately flapwise direction)
disp = blade.displacement()
dy = disp[1]
plt.plot(r, dy)
plt.xlabel('r (m)')
plt.ylabel('$\delta y$ (m)')
# plt.show()
```

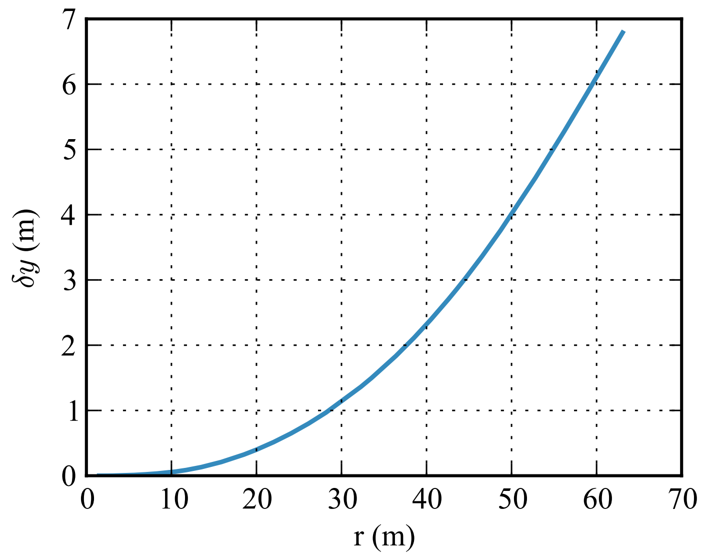

Figure 1. Blade deflection along span in flapwise direction.

Figure 2 shows a plot of the strain along the blade at the location of maximum airfoil thickness on both the pressure and suction side of the airfoil.

```
# evaluate strain at location of max airfoil thickness (upper and lower surface)
xpt = np.array([-0.02676, -0.02716, 0.009337, 0.009381, 0.009428, 0.009472,
                0.009515, -0.02336, 0.01155, 0.01219, -0.1379, -0.2207, -0.4827,
                -0.5969, -0.63, -0.6355, -0.6947, -0.6163, -0.6085, -0.5971,
                -0.5857, -0.5064, -0.4283, -0.3977, -0.3872, -0.3107, -0.2956,
                -0.2811, -0.212, -0.2087, -0.2012, -0.1864, -0.1791, -0.1669,
                -0.1599, -0.1587, -0.1586, -0.1559, -0.02725, -0.02765, 0.00883,
                0.008872, 0.008916, 0.008958, 0.008999, -0.02389, 0.01102,
                0.01165, -0.1384, -0.2211, -0.4831, -0.5972, -0.6303, -0.6357,
                -0.695, -0.6165, -0.6088, -0.5973, -0.5859, -0.5066, -0.4285,
                -0.3979, -0.3874, -0.3108, -0.2957, -0.2813, -0.2121, -0.2088,
                -0.2013, -0.1865, -0.1792, -0.167, -0.16, -0.1588, -0.1586, -0.1559])
ypt = np.array([1.63, 1.654, 1.686, 1.694, 1.702, 1.71, 1.718, 1.738, 1.773, 1.779,
                1.665, 1.465, 1.253, 0.9807, 0.8931, 0.8325, 0.7735, 0.7064, 0.6822,
                0.6488, 0.6125, 0.545, 0.4985, 0.4701, 0.4415, 0.3727, 0.354,
                0.3417, 0.3047, 0.2843, 0.2697, 0.2551, 0.2419, 0.2138, 0.1895,
                0.1694, 0.1488, 0.1382, -1.63, -1.654, -1.686, -1.694, -1.703,
                -1.711, -1.718, -1.738, -1.774, -1.78, -1.674, -1.482, -1.287,
                -1.022, -0.9412, -0.89, -0.8412, -0.7795, -0.7559, -0.7211, -0.6782,
                -0.5917, -0.5272, -0.4972, -0.4647, -0.3841, -0.3647, -0.3522,
                -0.3056, -0.2824, -0.2651, -0.2505, -0.2371, -0.208, -0.1823,
                -0.1605, -0.137, -0.1247])
zpt = np.array([1.501, 1.803, 1.902, 2, 2.105, 2.203, 2.302, 2.868, 3.003, 3.102,
                5.607, 7.005, 8.34, 10.51, 11.76, 13.51, 15.86, 18.52, 19.97,
                22.02, 24.07, 26.12, 28.17, 32.28, 33.53, 36.39, 38.53, 40.49,
                42.54, 43.54, 44.59, 46.54, 48.7, 52.8, 56.22, 58.95, 61.69,
                63.06, 1.501, 1.803, 1.902, 2, 2.105, 2.203, 2.302, 2.868, 3.003,
                3.102, 5.607, 7.005, 8.34, 10.51, 11.76, 13.51, 15.86, 18.52,
                19.97, 22.02, 24.07, 26.12, 28.17, 32.28, 33.53, 36.39, 38.53,
                40.49, 42.54, 43.54, 44.59, 46.54, 48.7, 52.8, 56.22, 58.95,
                61.69, 63.06])

strain = blade.axialStrain(len(xpt), xpt, ypt, zpt)
nstrain = len(strain)/2
plt.figure()
plt.plot(r, strain[:nstrain], label='suction side')
plt.plot(r, strain[nstrain:], label='pressure side')
plt.xlabel('r (m)')
plt.ylabel('$\epsilon$')
plt.legend()
plt.xlim([0, 63.0])
# plt.show()
```

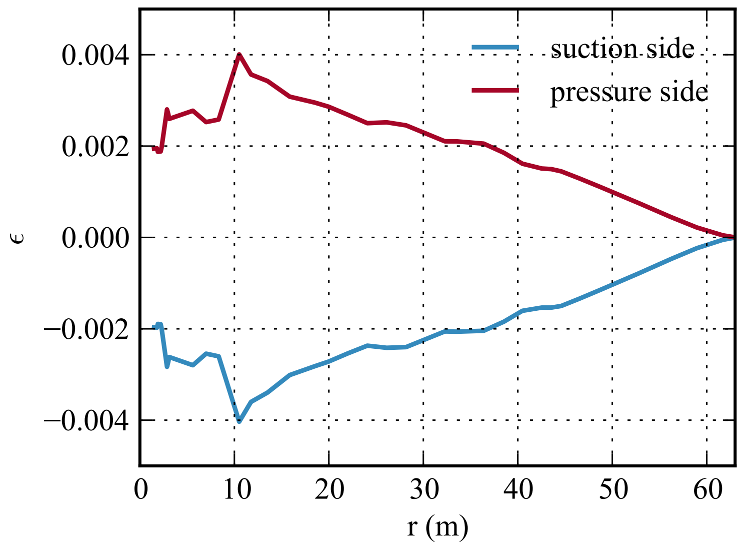

Figure 2. Strain along span at location of maximum airfoil thickness.

3.2 Cylindrical Shell Sections

This example simulates the tower from the NREL 5-MW reference model in pBEAM. First, the relevant modules are imported

```python
import numpy as np
from math import pi
import matplotlib.pyplot as plt

import _pBEAM
```

The basic tower geometry is defined

```python
# tower geometry defintion
z0 = [0, 30.0, 73.8, 117.6]                # heights starting from bottom (m)
d0 = [6.0, 6.0, 4.935, 3.87]               # corresponding diameters (m)
t0 = [0.0351, 0.0351, 0.0299, 0.0247]      # corresponding shell thicknesses (m)
n0 = [5, 5, 5]                             # number of finite elements per section
```

and then discretized so that it is defined at the end of every element (for convenience, a discretized definition could be supplied up front).

```python
# discretize
nodes = int(np.sum(n0)) + 1  # C++ interface requires int

z = np.zeros(nodes)
start = 0
for i in range(len(n0)):
    z[start:start+n0[i]+1] = np.linspace(z0[i], z0[i+1], n0[i]+1)
    start += n0[i]

d = np.interp(z, z0, d0)
t = np.interp(z, z0, t0)
```

The cylindrical shell model only allows for isotropic *material properties*.

```
# material properties

E = 210e9                # elastic modulus (Pa)
G = 80.8e9               # shear modulus (Pa)
rho = 8500.0             # material density (kg/m^3)

material = _pBEAM.Material(E, G, rho)
```

Distributed loads in this example come from wind loading and the tower's weight.

```
# distributed loads

g = 9.81  # gravity

# wind loading in x-direction
Px = np.array([0.0, 133.18, 167.41, 191.71, 211.09, 227.42, 236.04, 240.30,
               241.36, 239.92, 236.47, 231.37, 224.95, 217.64, 209.33, 200.16])
Py = np.zeros_like(Px)
Pz = -rho*g*(pi*d*t)  # self-weight


loads = _pBEAM.Loads(nodes, Px, Py, Pz)
```

Contributions from the rotor-nacelle-assembly (RNA) include mass, moments of inertia, and transfered forces/moments. These are added using the *TipData* class.

```
# RNA contribution

m = 300000.0  # mass
cm = np.array([-5.0, 0.0, 0.0])  # center of mass relative to tip
I = np.array([2960437.0, 3253223.0, 3264220.0, 0.0, -18400.0, 0.0])  # moments of inertia
F = np.array([750000.0, 0.0, -m*g])  # force
M = np.array([0.0, 0.0, 0.0])  # moment

tip = _pBEAM.TipData(m, cm, I, F, M)
```

The base of the tower is assumed to be rigidly mounted in this example. This corresponds to *BaseData* being initialized with infinite stiffness in all directions.

```
# rigid base

inf = float('inf')
k = np.array([inf]*6)

base = _pBEAM.BaseData(k, inf)
```

Finally, the tower object is created using the *cylindrical shell constructor*.

```
# create tower object

tower = _pBEAM.Beam(nodes, z, d, t, loads, material, tip, base)
```

Relevant properties can now be computed from this object in the same manner as in the previous example.

4 Module Documentation

This documentation only details the outward facing classes that are available through the Python module. Other classes and methods are available through the C++ implementation; however, they primarily encapsulate implementation details not necessary for external use. For details on use through C++, refer to the source code and examples in the test suite. The main class in the pBEAM module is *Beam*. All other classes are only helper objects used as inputs to the Beam object.

The HTML version of this documentation is better suited to view the code documentation and contains details on the *methods* contained in the Beam class as well as hyperlinks to the source code.

4.1 Beam

Beam is the main class for pBEAM and defines the complete beam object. Three constructors are provided: a convenience constructor for a beam with cylindrical shell sections and isotropic material (e.g., a wind turbine tower), a constructor for general sections that very linearly between sections, and a constructor where properties vary as polynomials across elements.

Class Summary:

class `_pBEAM.`**`Beam`**(*nodes*, *z*, *d*, *t*, *loads*, *mat*, *tip*, *base*)

A convenience constructor for a beam with cylindrical shell sections and isotropic material (e.g., a wind turbine tower)

Parameters

nodes : int

number of nodes

z : ndarray [nodes] (m)

location of beam sections starting from base and ending at tip

d : ndarray [nodes] (m)

diameter of beam at each z-location

t : ndarray [nodes] (m)

shell thickness of beam at each z-location

loads : *Loads*

loads along beam

material : *Material*

isotropic material properties

tip : *TipData*

properties of offset tip mass

base : *BaseData*

properties of base stiffness boundary condition

class `_pBEAM.`**`Beam`**(*section*, *loads*, *tip*, *base*)

A beam with general section properties with linear variation between nodes.

Parameters

section : *SectionData*

section data (inertial and stiffness properties) along beam

loads : *Loads*

loads along beam

tip : *TipData*

properties of offset tip mass

base : *BaseData*

properties of base stiffness boundary condition

class `_pBEAM.`**`Beam`**(*section*, *loads*, *tip*, *base*, *flag*)

A beam with general section properties with polynomial variation between nodes.

Parameters

section : *PolySectionData*

polynomial section data (inertial and stiffness properties) along beam

loads : *PolyLoads*

polynomial loads along beam

tip : *TipData*

properties of offset tip mass

base : *BaseData*

properties of base stiffness boundary condition

flag : int

can be set to any value, just a flag to allow use of polynomial variation rather than the overloaded constructor above that uses linear variation

Table 1. Methods available for Beam objects.

`mass()`	mass of beam
`naturalFrequencies(n)`	first n natural frequencies
`naturalFrequenciesAndEigenvectors(n)`	first n natural frequencies and eigenvectors
`displacement()`	6 DOF displacement at each node
`criticalBucklingLoads()`	global minimum critical axial buckling loads
`axialStrain(n, x, y, z)`	axial strain along beam
`outOfPlaneMomentOfInertia()`	out of plane moment of inertia

4.2 SectionData

SectionData is a C++ struct that defines the section properties along the beam, assuming a linear variation in properties between the defined sections. For polynomial variation, see *PolySectionData*.

Class Summary:

class `_pBEAM.`**`SectionData`**(*n*, *z*, *EA*, *EIxx*, *EIyy*, *GJ*, *rhoA*, *rhoJ*)

Section data defined along the structure from base to tip.

Parameters

n : int

number of sections where data is defined (nodes)

z : ndarray [n] (m)

location of beam sections starting from base and ending at tip

EA : ndarray [n] (N)

axial stiffness at each section

EIxx : ndarray [n] (N*m**2)

bending stiffness about +x-axis

EIyy : ndarray [n] (N*m**2)

bending stiffness about +y-axis

GJ : ndarray [n] (N*m**2)

torsional stiffness about +z-axis

rhoA : ndarray [n] (kg/m)

mass per unit length

rhoJ : ndarray [n] (kg*m)

polar mass moment of inertia per unit length

Notes

All parameters must be specified about the elastic center and in principal axis (i.e., EIxy, Sx, and Sy are all zero). Linear variation in properties between sections is assumed.

4.3 PolySectionData

PolySectionData is a C++ struct that defines the section properties along the beam, and allows for polynomial variation of properties between the defined sections. For linear variation, *SectionData* is simpler to work with.

Class Summary:

class `_pBEAM.`**`PolySectionData`**(*nodes*, *z*, *nA*, *nI*, *EA*, *EIxx*, *EIyy*, *GJ*, *rhoA*, *rhoJ*)

Polynomial section data defined along the structure from base to tip.

Parameters

n : int

number of sections where data is defined (nodes)

z : ndarray [n] (m)

location of beam sections starting from base and ending at tip

nA : ndarray(int) [n - 1]

nA[i] is the order of the polynomial describing structural properties between nodes i and i + 1 for properties that are area dependent (EA, rhoA)

nI : ndarray(int) [n - 1]

nI[i] is the order of the polynomial describing structural properties between nodes i and i + 1 for properties that are moment of inertia dependent (EIxx, EIyy, GJ, rhoJ)

EA : list(ndarray) [n - 1] (N)

EA[i] is a polynomial of length nA[i] that describes the axial stiffness between nodes i and i + 1

EIxx : list(ndarray) [n - 1] (N*m**2)

EIxx[i] is a polynomial of length nI[i] that describes the bending stiffness about +x-axis between nodes i and i + 1

EIyy : list(ndarray) [n - 1] (N*m**2)

EIyy[i] is a polynomial of length nI[i] that describes the bending stiffness about +y-axis between nodes i and i + 1

GJ : list(ndarray) [n - 1] (N*m**2)

GJ[i] is a polynomial of length nI[i] that describes the torsional stiffness about +z-axis between nodes i and i + 1

rhoA : list(ndarray) [n - 1] (kg/m)

rhoA[i] is a polynomial of length nA[i] that describes the mass per unit length between nodes i and i + 1

rhoJ : list(ndarray) [n - 1] (kg*m)

rhoJ[i] is a polynomial of length nI[i] that describes the polar mass moment of inertia per unit length between nodes i and i + 1

Notes

All parameters must be specified about the elastic center and in principal axis (i.e., EIxy, Sx, and Sy are all zero). Polynomials are expressed as:

EA[i] = [5.0, 3.0, 2.0] which means $EA[i] = 5.0\eta^2 + 3.0\eta + 2.0$

where η is a normalized coordinate s.t. it equals 0 at the base of the given element and 1 at the top of the element. The numpy polynomial class (numpy.poly1d) is useful for multiplying polynomials, etc.

4.4 Loads

Loads is a C++ struct that defines the applied loads along the beam. Three constructors are provided: beams with no external loads, beams with only distributed loads, and beams with distributed loads and point forces/moments. Distributed loads are assumed to vary linear between sections. For polynomial variation in distributed loads, see *PolyLoads*.

Class Summary:

class `_pBEAM.`**`Loads`**

No applied external loads.

class `_pBEAM.`**`Loads`** (*n*, *Px*, *Py*, *Pz*)

Distributed loads along beam from base to tip.

Parameters

n : int

number of sections where forces are defined (nodes)

Px : ndarray [n] (N/m)

force per unit length along beam in the x-direction

Py : ndarray [n] (N/m)

force per unit length along beam in the y-direction

Pz : ndarray [n] (N/m)

force per unit length along beam in the z-direction

Notes

Loads must be given at the corresponding z locations defined in *SectionData* or *PolySectionData*.

class `_pBEAM.`**`Loads`** (*n*, *Px*, *Py*, *Pz*, *Fx*, *Fy*, *Fz*, *Mx*, *My*, *Mz*)

Distributed loads and applied point forces/moments along beam from base to tip.

Parameters

n : int

number of sections where forces are defined (nodes)

Px : ndarray [n] (N/m)

force per unit length along beam in the x-direction

Py : ndarray [n] (N/m)

force per unit length along beam in the y-direction

Pz : ndarray [n] (N/m)

force per unit length along beam in the z-direction

Fx : ndarray [n] (N)

point forces in the x-direction

Fy : ndarray [n] (N)

point forces in the y-direction

Fz : ndarray [n] (N)

point forces in the z-direction

Mx : ndarray [n] (N*m)

point moments in the x-direction

My : ndarray [n] (N*m)

point moments in the y-direction

Mz : ndarray [n] (N*m)

point moments in the z-direction

Notes

Loads must be given at the corresponding z locations defined in *SectionData* or *PolySectionData*.

4.5 PolyLoads

PolyLoads is a C++ struct that defines the applied loads along the beam and allows for polynomial variation of loads between the defined sections. For linear variation in distributed loads, *Loads* is simpler to work with.

Class Summary:

class `_pBEAM.`**`PolyLoads`** (*n*, *nP*, *Px*, *Py*, *Pz*, *Fx*, *Fy*, *Fz*, *Mx*, *My*, *Mz*)

Polynomial variation in distributed loads, and point forces/moments defined along the structure from base to tip.

Parameters

n : int

number of sections where loads are defined (nodes)

nP : ndarray(int) [n - 1]

nP[i] is the order of the polynomial describing the distributed load between nodes i and i + 1

Px : list(ndarray) [n - 1] (N)

Px[i] is a polynomial of length nP[i] that describes the force per unit length in the +x-direction between nodes i and i + 1

Py : list(ndarray) [n - 1] (N)

Py[i] is a polynomial of length nP[i] that describes the force per unit length in the +y-direction between nodes i and i + 1

Pz : list(ndarray) [n - 1] (N)

Pz[i] is a polynomial of length nP[i] that describes the force per unit length in the +z-direction between nodes i and i + 1

Fx : ndarray [n] (N)

point forces in the x-direction

Fy : ndarray [n] (N)

point forces in the y-direction

Fz : ndarray [n] (N)

point forces in the z-direction

Mx : ndarray [n] (N*m)

point moments in the x-direction

My : ndarray [n] (N*m)

point moments in the y-direction

Mz : ndarray [n] (N*m)

point moments in the z-direction

Notes

Polynomials are expressed as:

Px[i] = [5.0, 3.0, 2.0], which means $Px[i] = 5.0\eta^2 + 3.0\eta + 2.0$

where η is a normalized coordinate s.t. it equals 0 at the base of the given element and 1 at the top of the element. The numpy polynomial class (numpy.poly1d) is useful for multiplying polynomials, etc.

Loads must be given at the corresponding z locations defined in *SectionData* or *PolySectionData*

4.6 BaseData

BaseData is a C++ struct that defines the stiffness properties of the base of the beam. Two constructors are available: a convenience constructor for a free-end, and a general constructor for specifying the equivalent spring stiffness in all 6 DOF.

Class Summary:

class `_pBEAM.`**`BaseData`**

A free-end. External spring stiffness is zero in all directions.

class `_pBEAM.`**`BaseData`**(*k*, *infinity*)

A base with equivalent external spring stiffness applied.

Parameters

k : ndarray (N/m)

stifness at base [k_xx, k_txtx, k_yy, k_tyty, k_zz, k_tztz] where tx is the rotational direction theta_x and so forth

infinity : float (N/m)

a value that represents infinity (can be any arbitrary float but it is convenient to use Python's `float('inf')`). used to denote infinitely rigid directions.

4.7 TipData

TipData is a C++ struct that defines the properties of the offset tip mass. Two constructors are available: a convenience constructor for a beam with no offset tip mass, and a general constructor for specifying the properties of the offset tip mass.

Class Summary:

class `_pBEAM.`**`TipData`**

No offset tip mass.

class `_pBEAM.`**`TipData`**`(`*m*, *cm*, *I*, *F*, *M*`)`

Used to model an offset tip mass (e.g., rotor/nacelle/assembly on top of a wind turbine tower)

Parameters

m : float (kg)

mass of object

cm : ndarray (m)

location of object's center of mass relative to beam tip [x, y, z]

I : ndarray (m^4)

area moment of inertia of object about beam tip [Ixx, Iyy, Izz, Ixy, Ixz, Iyz]

F : ndarray (N)

applied force from the object onto the beam tip [Fx, Fy, Fz]

M : ndarray (N*m)

applied moment from the object onto the beam tip [Mx, My, Mz]

4.8 Material

Material is a C++ struct that defines the material properties for an isotropic material.

Class Summary:

class `_pBEAM.`**`Material`**`(`*E*, *G*, *rho*`)`

Material properties for an isotropic material.

Parameters

E : float (N/m**2)

modulus of elasticity

G : float (N/m**2)

shear modulus

rho : float

mass density (kg/m**3)

5 Theory

The methodology details are available in Yang (1986). Usage differs here only in that rather than using precomputed matrices for an assumed variation in structural properties, a polynomial representation is assumed that allows for more flexible usage and exact polynomial integration.

Table 2. Nomenclature for symbols used in this section.

symbol	definition
K	stiffness matrix
M	inertia matrix or moment
N	incremental stiffness matrix
q	displacement vector
F	force vector
η	nondimensional coordinate along element
L	length of element
ρ	mass density
m	mass
A	cross-sectional area
I	area moment of inertia
J	polar area moment of inertia
E	modulus of elasticity
G	shear modulus of elasticity
f	shape function
v	velocity vector
ω	angular velocity vector
V	shear force

5.1 Finite Element Matrices

The governing equation of the structure is given by

$$Kq + M\ddot{q} = F$$

and for buckling analysis

$$[K - N]q = F$$

The finite element matrices are computed from the structural properties of the beam. As mentioned earlier, pBEAM uses a polynomial representation of the structural properties. All polynomials are defined across an element in normalized coordinates (η). For example, the moment of inertia across an element may vary quadratically as $I_2\eta^2 + I_1\eta + I_0$. Computation of the finite element matrices are described below, where all derivatives are with respect to η.

bending stiffness matrix:

$$K_{ij} = \frac{1}{L^3} \int_0^1 EI(\eta) f_i''(\eta) f_j''(\eta) d\eta$$

bending inertia matrix:

$$M_{ij} = L \int_0^1 \rho A(\eta) f_i(\eta) f_j(\eta) d\eta$$

incremental stiffness matrix:

$$N_{ij} = \frac{1}{L} \int_0^1 F_z(\eta) f_i'(\eta) f_j'(\eta) d\eta$$

axial stiffness matrix:

$$K_{ij} = \frac{1}{L} \int_0^1 EA(\eta) f_i'(\eta) f_j'(\eta) d\eta$$

axial inertia matrix:

$$M_{ij} = L \int_0^1 \rho A(\eta) f_i(\eta) f_j(\eta) d\eta$$

Torsional matrices are computed similarly to the axial matrices, except $EA(\eta)$ is replaced with $GJ(\eta)$ and $\rho A(\eta)$ is replaced with $\rho J(\eta)$. Note that although the same notation was used, the axial shape functions are not the same as those for bending. Because section properties are defined as polynomials, each of these derivatives and integrals are done analytically.

5.2 Top Mass

pBEAM assumes that the top of the beam is a free end, but that a mass may exist on top of the beam. This is useful for modeling structures such as an RNA (rotor/nacelle/assembly) on top of a wind turbine tower. The top mass is assumed to be a rigid body with respect to the main beam and thus, does not contribute to the stiffness matrix. It does, however, affect the inertial loading and external forces as discussed below. The top mass can be offset from the beam top by some vector ρ. Although idealized as a point mass, its moment of inertia matrix can also be specified. The tip is both translating and rotating, so the velocity of the tip mass in an inertial reference frame is given by (with reference to the variables in Figure 3):

$$\vec{v}_m = \frac{d\vec{r}}{dt} + \left(\frac{d\vec{\rho}}{dt} \right)_\rho + \vec{\omega} \times \vec{\rho}$$

where the second time derivative is taken in the rotating reference frame. The kinetic energy of the mass is then

$$\begin{aligned} T &= \frac{1}{2} m \vec{v}_m \cdot \vec{v}_m + \frac{1}{2} \vec{\omega}^T I \vec{\omega} \\ &= \frac{1}{2} m \left[(\dot{x} + \dot{\theta}_y \rho_z - \dot{\theta}_z \rho_y)^2 + (\dot{y} + \dot{\theta}_z \rho_x - \dot{\theta}_x \rho_z)^2 + (\dot{z} + \dot{\theta}_x \rho_y - \dot{\theta}_y \rho_x)^2 \right] \\ &+ \frac{1}{2} \left[I_{xx} \dot{\theta_x}^2 + 2 I_{xy} \dot{\theta}_x \dot{\theta}_y + 2 I_{xz} \dot{\theta}_x \dot{\theta}_z + \dots \right] \end{aligned}$$

Using the Lagrangian one can show that

$$(M\ddot{q})_i = \frac{d}{dt} \frac{\partial T}{\partial \dot{q}_i}$$

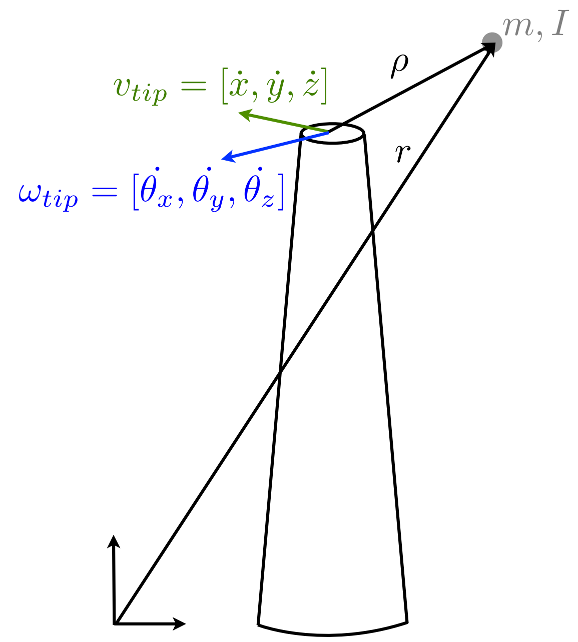
m, I
ρ
$v_{tip} = [\dot{x}, \dot{y}, \dot{z}]$
r
$\omega_{tip} = [\dot{\theta}_x, \dot{\theta}_y, \dot{\theta}_z]$

Figure 3. Diagram of top mass idealized as a point mass with moments of inertia. The center of mass of the top mass is offset by vector ρ relative to the top of the beam. The top of the beam is also potentially translating and rotating.

After taking the derivatives, the inertial matrix contribution from the top mass is given by

$$M_{tip} = \begin{bmatrix} m & & & m\rho_z & & -m\rho_y \\ & I_{xx}+m(\rho_y^2+\rho_z^2) & -m\rho_z & I_{xy}-m\rho_x\rho_y & m\rho_y & I_{xz}-m\rho_x\rho_z \\ & -m\rho_z & m & & & m\rho_x \\ m\rho_z & I_{xy}-m\rho_x\rho_y & & I_{yy}+m(\rho_x^2+\rho_z^2) & -m\rho_x & I_{yz}-m\rho_y\rho_z \\ & m\rho_y & & -m\rho_x & m & \\ -m\rho_y & I_{xz}-m\rho_x\rho_z & m\rho_x & I_{yz}-m\rho_y\rho_z & & I_{zz}+m(\rho_x^2+\rho_y^2) \end{bmatrix}$$

where $q = [x, \theta_x, y, \theta_y, z, \theta_z]$. Note that the current implementation assumes moments of inertia are given about the beam tip, though moments of inertia about its own center of mass are easily translated to the beam tip via the generalized parallel axis theorem.

Finally, the top mass may also apply loads (forces and moments) to the beam. These are simply added to the force vector at the tip of the structure. It is assumed that the weight of the top mass was already added to the force vector.

5.3 Base

The bottom of the beam is assumed to be constrained by linear springs in all 6 coordinate directions. Any of these springs can be chosen to be infinitely stiff, or in other words, rigidly constrained in that direction. This simply adds a diagonal stiffness matrix at the bottom of the beam, and directions that are rigid are removed from the structural matrices.

5.4 Loads

Distributed loads, point forces, and point moments can be specified anywhere in the structure. Distributed loads are specified as polynomials across the elements. For distributed loads in the lateral directions, work equivalent loads are computed at the nodes. Axially distributed loads are integrated starting from the free end of the beam to compute the polynomial variation in axial force.

5.5 Axial Stress

The computation of axial stress is separate from the finite element analysis, but is included in this code for convenience. First, the forces and moments must be computed along the beam. For example the shear force and moments are evaluated as

$$V_i = V_{i+1}(0) + F_{pt\,i+1} + (z_{i+1} - z_i) \int_1^0 q(\eta) d\eta$$
$$M_i = M_{i+1}(0) + M_{pt\,i+1} + (z_{i+1} - z_i) \int_1^0 V_i(\eta) d\eta$$

where F_{pt} and M_{pt} are external point forces and moments along the structure. Note that the integration is actually an indefinite integral, but limits are noted to signify that integration must be done from the tip where forces/moments are known. Finally, the stress is computed as (or use $E(x,y) = 1$ to compute strain):

$$\sigma_{zz}(x,y) = E(x,y)\left(\frac{M_x}{[EI]_x} y - \frac{M_y}{[EI]_y} x + \frac{N_z}{[EA]}\right)$$

Bibliography

Hansen, M.O.L. (2008). *Aerodynamics of Wind Turbines*. 2nd ed. Earthscan.

Yang, T.Y. (1986). *Finite Element Structural Analysis*. Prentice Hall College Div.