README CME 211 HW4

Thomas Brink (06446502)

October 29, 2021

1 Introduction

In this homework, we provide code for a 'truss' class. This class takes in input files on the beams and joints of a truss and analyzes the geometry and forces that are at play in the truss. We provide two files: *main.py* and *truss.py*.

2 main.py

In the *main.py* file, we provide the handling of user inputs and call the truss class to provide output. That is, when running our code, we call *main.py* and some input arguments, which results in some sort of output.

Here's an example of command line code to analyze the truss geometry and forces given some user inputs:

$ python3 main.py [joints file] [beams file] [optional plot file]

Note that the plot file is optional. When given by the user, a figure will be created that plots the truss geometry (2D).

The output of the command line example is either a table containing numbered Beams and the corresponding forces as columns or some sensible error message. We provide more insight into these error messages when discussing the *truss.py* file.

3 truss.py

In *truss.py*, we provide code for the *Truss* class. This class allows users to create instances of a truss with given joints and beams data. The code for this class consists of 6 functions and some introductory import statements, which we consequently discuss one by one.

a) Import statements: we import the *matplotlib*, *numpy*, *scipy*, and *warnings* modules. We use these model for plotting, data handling and manipulation, (sparse) matrix algebra, and warning handling, respectively.

b) *__init__*: the first function is the *__init__* function, which we use to initialize an instance of the truss class given the user inputs and to call the other functions in the truss class so as to perform our analysis on the initialized instance. The *__init__* function takes three inputs (not including *self*); a file name for joints data, a file name for beams data, and an optional name to save the truss geometry plot to. By default, this argument is set to 0, which leads to not plotting the geometry.

c) *load_data*: this second function we use loads in the joints and beams data given by the user by storing beams data in a numpy array and the joints data in a dictionary (with joint number as key and the data for this joint as provided in the input file as value). Also, we raise exceptions if errors occur while loading in the data and add the index of a joint to the value in the dictionary (so as to make ordering of the joints easier, which we will use in constructing the matrix system for our analysis).

d) *plot_geometry*: this function plots the x- and y-coordinates of joints and plots lines between these points in case a beam between these joints exists. We save the plot to a file with the name that is given by the user as input.

e) *calculate_forces*: this function is the main calculation function in the class, and is used to construct the vector of external forces and matrix of force coefficients that are used in the linear system

$$Ax = b, \tag{1}$$

where A is the matrix with force coefficients, b the vector of external forces, and x a vector with beam and reaction forces. Note that x is unknown and finding x is the main goal of the class. The dimensions of A should be $n \times n$, where $n = 2\times$ (the number of joints), or $n =$ (the number of beams) $+2\times$ (the number of joints with a reaction force.

In the *calculate_forces* function, we check whether these dimensions are correct and raise an exception otherwise. Namely, if the dimensions don't add up, we are not able to find x. Note that x is a vector with beam forces and reaction forces, while b is a vector with external forces for each x- and y-component of each joint.

We compute b by creating a sparse matrix that reads in the external force data in the joints data file. We compute A coefficients for beam forces by calculating the difference in x- and y-components between joints connecting a beam, scaled by the length of the beam. For the starting joint, we compute

$$\text{coeff} = \frac{x_1 - x_2}{\text{length of beam}}, \tag{2}$$

and similar for the y-component. For the ending joint of a beam, we compute

$$\text{coeff} = \frac{x_2 - x_1}{\text{length of beam}}. \tag{3}$$

After filling in the beam force coefficients, we only need the coefficients for the reaction forces. In our joints input data, we have the *zerodisp* column that indicates which joints have a reaction force. We add a coefficient 1 to the x- and y-component of a joint in the respective reaction force column of A. Just as with b, we save A in sparse format. We use a *csr*-type sparse matrix, which allows us to immediately go to the next function, which solves the linear system.

f) *compute_static_equilibrium*: this function uses the matrix of force coefficients A and the vector of external force coefficients b to solve for x in linear system 1. In case the coef-

ficient matrix is singular (det = 0), we cannot solve the system and raise an exception. Otherwise, we solve the system and save the forces in vector x.

g) *__repr__*: the last function is the string representation function, which allows us to print the output of our truss analysis directly when using the *print*(*instance*) command. We use the specific formatting as outlined in the assignment.

Putting the two code files together, we have obtained a complete set of code that allows us to perform analysis on a truss instance given joints and beams data!