Assignment 3: Helping a Fellow Engineer Design a Truss

Due: Friday 24 September 11:59pm

Review

In Assignment 1 (and the associated in-class exercises) we became familiar with creating a program to solve an IEA-type simple truss statics problem based on a force balance per the method of joints. It likely cost you more time to correctly translate the mathematical model into Python code than to solve the problem using pencil and paper. It was just for practice. The savings would have come from implementing your solution as a function so that a user could enter any angle for the applied force and have your code easily compute the result (much the way online automated homework systems like Mastering Engineering work). But we had not learned about functions yet. For more programming practice you might consider implementing your Assignment 1 solution as a `function`, and using `if` statements to verify that the input angle is in the appropriate domain.

Assignment 2 depended on knowledge from IEA and Physics I and made you familiar with vectors and vector algebra in a programming environment. More program design decisions were left to you; for example, how to loop over the elements of an array:

```
import numpy as np
vector = np.array([11, 92, 33])
n = vector.shape[0]
print("vector is", vector, "with", n, "elements")

print()
print("Loop by array element index -- usually preferred")
for i in range(0, n):
    print("Python index:", i, "  Element:", vector[i])

print()
print("or loop over the elements themselves")
i = 0            # initialize element Python index (0 to n-1)
for x in vector:
    print("Python index:", i, "  Element:", x)
    i += 1       # increments the Python index
```

and how to implement the mathematical formula for the `cross_product` in code.

Assignment 3 will leave even more program design decisions to you. When you discuss algorithms with each other be sure to embrace the diversity of your approaches and not lock in to what someone else suggests. There is no “one right way”.

<u>Assignment Overview</u>

Assignment 3 begins with Part 1 where you put together the pieces you learned in Lessons 4 and 5 on LU factorization and the associated in-class exercises. You will first polish functions **`forward_sub`** and **`back_sub`** and modify **`lu_factor`** in your **`linsolve`** module to make sure they work as needed and are appropriately commented (including useful docstrings). You will then add another function, **`solve`**, to your **linsolve.py** file that uses the first three functions to solve any linear system of the form **<u>A</u> <u>x</u> = <u>b</u>** where **<u>A</u>** is invertible and has the proper layout for LU decomposition, and **<u>x</u>** and **<u>b</u>** are vectors that are conformable with **<u>A</u>**.

Part 2 returns to IEA to revisit a simple truss, except that this truss has 11 members, making its mathematical model unwieldy for a manual solution.

The truss is part of a jib crane that supports a hoist on an I-beam rail used to move nuclear fuel from the nuclear reactor to spent fuel storage racks in a spent fuel pool. Spent nuclear fuel has been depleted of its uranium fuel that provided nuclear fission energy in a nuclear reactor (typically over four to six years). During refueling the spent fuel must be moved from the reactor to a spent fuel pool: a pool approximately 40 feet deep that uses the water both as coolant and shielding for the highly radioactive spent fuel. A typical pool and transfer crane are illustrated in Figure 1.

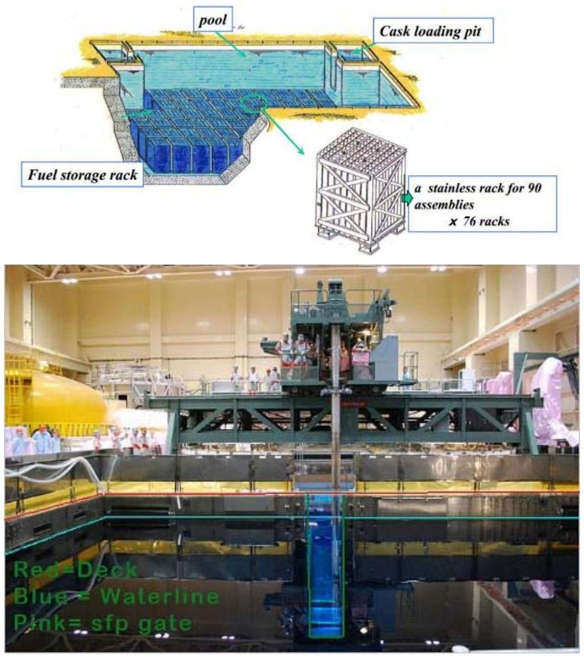

Figure 1: spent fuel pool storage racks and transfer crane

Figure 2 illustrates the truss and hoist system, including some dimensional indications. Figure 3 shows an example of a working hoist that travels along an I-beam.

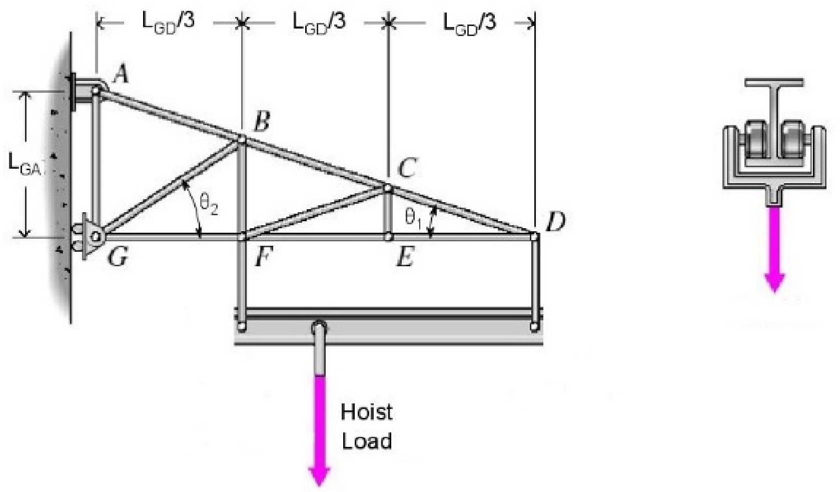

Figure 2: truss and hoist beam system

Figure 3: a working hoist

Solving for the tensions and reactions in the eleven member simple truss requires solving a 14x14 system of equations – a problem that will take too long and is too prone to mistakes if done by hand. The efficiency and reliability of using a proven algorithm translated into working code that has been thoroughly tested should be clear. Your task is to provide that code to a fellow engineer tasked with designing the truss.

The truss design engineer has already developed a mathematical model for the truss shown in Figure 2. The matrix and vectors in Figure 4 summarize that model as derived by a method of joints force balance analysis of the truss:

- matrix $\underline{\underline{\mathbf{G}}}$ models the geometry of the truss system of members and supports;
- vector $\underline{\mathbf{t}}$ is an array of truss member tension magnitudes (when an element is positive; compression magnitude when negative) and support reactions;
- vector $\underline{\mathbf{f}}$ is an array of the horizontal and vertical components of the external applied loads at each joint.

$$\underline{\underline{\mathbf{G}}} = \begin{bmatrix} c_1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ -s_1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ -c_1 & c_1 & 0 & 0 & 0 & 0 & 0 & -c_2 & 0 & 0 & 0 & 0 & 0 & 0 \\ s_1 & -s_1 & 0 & 0 & 0 & 0 & 0 & -s_2 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -c_1 & c_1 & 0 & 0 & 0 & 0 & 0 & 0 & -c_1 & 0 & 0 & 0 & 0 \\ 0 & s_1 & -s_1 & 0 & 0 & 0 & 0 & 0 & 0 & -s_1 & -1 & 0 & 0 & 0 \\ 0 & 0 & -c_1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & s_1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & c_1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & s_1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & c_2 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & s_2 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad \underline{\mathbf{t}} = \begin{bmatrix} T_{AB} \\ T_{BC} \\ T_{CD} \\ T_{DE} \\ T_{EF} \\ T_{FG} \\ T_{AG} \\ T_{BG} \\ T_{BF} \\ T_{CF} \\ T_{CE} \\ A_x \\ A_y \\ G_x \end{bmatrix} \quad \underline{\mathbf{f}} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ F_D \\ 0 \\ 0 \\ 0 \\ F_F \\ 0 \\ 0 \end{bmatrix}$$

Figure 4: linear system model for the truss in Figure 2. Note the following notation used:

s_k = $\sin(\theta_k)$, k = 1, 2
c_k = $\cos(\theta_k)$, k = 1, 2
T_i = tension in truss member i
A_x, A_y, G_x = reaction forces in positive horizontal (x) and vertical (y) directions
F_j = vertical external load applied at joint j

The truss design engineer wants Python code that can quickly analyze the forces in the truss elements based on the position of the hoist along the beam. In particular, the engineer wants five functions to:

- define an applied load vector (the "right hand side" $\underline{\mathbf{f}}$ vector in the linear system);
- define a truss geometry matrix (the $\underline{\underline{\mathbf{G}}}$ matrix in the $\underline{\underline{\mathbf{G}}}\,\underline{\mathbf{t}} = \underline{\mathbf{f}}$ linear system);
- shuffle the rows of $\underline{\underline{\mathbf{G}}}$ such that LU decomposition is applicable (based on shuffles identified manually outside of the code);
- identify the element of the $\underline{\mathbf{t}}$ vector with the largest magnitude; and
- call all of the above functions given some input geometry and load specifications to compute the tension or compression in the truss member with the largest force magnitude.

As you will see, the truss design engineer briefly started to write the code but has passed the project on to you.

Instructions and Deliverables

For this assignment you will need to write two Python module files. The requirements on each of these files are described in detail in the following two parts. When you submit using Submitty there will be a separate box to upload the two files individually. Please recall that, as always, you have only 20 chances to submit your modules to Submitty without penalty. This is a computationally open-ended assignment so you will likely not be able to "test Submitty's numbers" by repeatedly submitting. Instead you will need to test your logic, test small sample cases that you know the solution to, and correct typos without ever knowing if your code really works for all possible cases. (Even Submitty will test only a finite number of cases.)

As always, you are strongly encouraged to design your own test codes (ideally in a separate file in the same directory, with a line of code like "`from hw03 import …`") to verify that your code works before you submit it. Test your code piece by piece. One of the strong arguments for using Python is the ability to rapidly prototype code; use the strengths of the tool you're using. Equally important, you are also always encouraged to do your analyses and outline your pseudocode on paper before writing any actual Python code.

Recall from Assignment 2 a method for streamlining the testing of your code using a "debug" variable. For example, consider the following function.

```
def quadratic_formula (a, b, c):
    debug = True      # set to False before submitting
    discriminant = b**2 - 4.0*a*c
    if (debug):
        print("a, b, c =", a, b, c)
        print("discriminant =", discriminant)
    if (discriminant < 0.0):
        raise RuntimeError("Complex roots not allowed.")
    x1 = ( -b + discriminant**(1/2) ) / (2.0*a)
    x2 = ( -b - discriminant**(1/2) ) / (2.0*a)
    if (debug):
        print("roots are", x1, x2)
    return x1, x2
```

While developing and testing this code you can set `debug = True` and get lots of printed output. You could even use multiple debug variables (`debug_1`, `debug_2`, etc.) to report different amounts or sections of information. Then you can set `debug = False` to suppress the print statements (e.g., before you submit your file to Submitty).

Also note the use of Python's `raise` statement and `RuntimeError` exception. You will use this "as is" in Part 1 of this assignment to stop your code from executing when a failure is detected, as well as to describe the failure in what is effectively a print statement. We will learn more about raising exceptions later in the semester.

Part 1: A module for solving linear systems

Create a Python module file for solving linear systems via LU decomposition. You should reuse and build upon your LU decomposition code from Lesson 5 and your solvers for upper and lower triangular systems from Lesson 4. Please follow the requirements listed below to maximize your score.

- Name the module file **linsolve.py**.

- The module should contain four functions (their order is not important) with the following names: `lu_factor`, `forward_sub`, `back_sub`, and `solve`.

- The `lu_factor` function performs the LU decomposition and must have the following opening definition (which is different from the in-class exercise):
  ```
     def lu_factor(A, tol=1.0e-15):
  ```
  where **A** is the square 2-dimensional NumPy array that is being decomposed. The parameter `tol` is a keyword argument, and `1.0e-15` is its default value. This means if a user calls your function as
  ```
     lu_factor(A)
  ```
  then `tol` defaults to the value `1.0e-15`. Alternatively, the user could specify their own tolerance by calling your function as, e.g.,
  ```
     lu_factor(A, tol=2.345e-6)
  ```
  For this assignment you may use the default tolerance.

  The `tol` parameter should be used in a criterion that decides if a diagonal element is zero during LU decomposition; for example, you could use
  ```
     if ( abs(A[k,k]) <= tol ):
  ```
  at the appropriate point or points during the decomposition. (You may have to carefully sketch and think about this to discern what "appropriate point or points" means here.) If a diagonal entry is less than `tol` then the LU decomposition should be considered to have failed and `lu_factor` should return `False`. In all other cases, the `lu_factor` function should return `True`. (Again, note that this is a change from the in-class exercise where `lu_factor` simply returned `None`.)

- The `forward_sub` function should solve lower-triangular systems and must have the opening definition
  ```
     def forward_sub(lu_fac, b):
  ```
  where `lu_fac` is the square 2-dimensional NumPy array that has been decomposed using function `lu_factor`, and **b** is a 1-dimensional NumPy array.

- The `back_sub` function should solve upper-triangular systems and must have the opening definition
  ```
     def back_sub(lu_fac, y):
  ```
  where `lu_fac` is the square 2-dimensional NumPy array that has been decomposed using `lu_factor`, and **y** is a 1-dimensional NumPy array.

- The solve function must have the following opening definition:
  ```
     def solve(A, b, tol=1.0e-15):
  ```

where **A** is expected to be a 2-dimensional square NumPy array and **`b`** is expected to be a 1-dimensional NumPy array with the same number of elements as the rows of matrix **A**. Matrix **A** and vector **`b`** represent the system model matrix and right-hand side (or output) vector in the linear system $\underline{\underline{A}}\ \underline{x} = \underline{b}$. The **`tol`** keyword argument provides a value for the tolerance that can be passed to the **`lu_factor`** function.

The **`solve`** function should return the solution **`x`** as a 1-dimensional NumPy array. It is expected that the array **A** will be modified during the call to solve, as it will be during the LU decomposition process. Finally, the solve function must make appropriate use of the **`lu_factor`**, **`forward_sub`**, and **`back_sub`** functions described above.

Your function **`solve`** must appropriately detect if the LU decomposition fails. If it fails, your program should terminate after sending a message to the console by using the following statement:

```
raise RuntimeError("Function lu_factor failed.")
```

You can see (and try running) an example of how this statement is used in the quadratic formula root finder discussed above.

- All of the functions in your **linsolve** module must have multiline docstrings. That is, their docstrings should have a one-line concise explanation, followed by a blank line, and then additional details, as described in class. See also https://www.python.org/dev/peps/pep-0257/.

Part 2: Functions for analyzing the tensions in the specified truss

For Part 2 of this assignment you will create a module that imports your **`linsolve`** module to estimate the tensions in the truss specified in the Assignment Overview section.

- Name the module file **hw03.py**.

- The design engineer started to write the code but passed the job to you. All that the truss design engineer got to was an incomplete function to specify the right hand side load vector. Appendix A contains that code which you need to complete. The incomplete code is also provided as file **`AppendixA.py`** on LMS where you got this document from.

- Create a function to return the geometry matrix $\underline{\underline{G}}$ as specified by the truss design engineer earlier in this document. Use the following opening definition:

  ```
  def truss_geometry(L_GA, L_GD):
  ```

  Your function should use the geometry parameters to compute all of the elements of $\underline{\underline{G}}$. Store $\underline{\underline{G}}$ in a 2D numpy array that the function returns. You must define the 14x14 array exactly as the truss design engineer provided it. It is not an array of unknown size or with unknown coefficients except to the extent that you need to compute the values of c_1, c_2, s_1, and s_2 based on the arguments input to the function.

  Note that Python will allow you to extend certain statements over multiple lines. For example, you could code a 2x2 matrix as

```
matrix  =  np.array( [[1.0, 2.0],
                      [3.0, 4.0]] )
```
so that you can visually line up the elements to more easily verify an accurate transcription.

- If you use matrix **G** as provided your code will correctly fail. (HINT: what's the only failure your code detects?) Before calling the solve function you will need to swap various rows of G to put it into a useable form. You will need to manually figure out what order to put the rows, but the actual swapping of rows must be done in code using a function with the following opening definition:
  ```
  def shuffle_Ab(A, b, order):
  ```
  This function will return a new matrix and a new vector (in that order, without overwriting the originals) with their rows in the order specified in the 1D numpy array **`order`**.

  For example, if **A** is the 2D numpy array [[1, 2],[3, 4],[5, 6]] and **b** is the 1D numpy array [7, 8, 9] and if **`order`** is the 1D numpy array [1, 0, 2] (note the Python indexing!), then the shuffled 2D numpy array for **A** will be [[3, 4],[1, 2],[5, 6]] and the shuffled 1D numpy array for **b** will be [8, 7, 9]. That is, the order[i] row of **A** is put in the i-th row of the shuffled matrix. Additional details are given in Appendix C.

  This function must work for any generic **A**, **b**, and **`order`** that make sense. (That is, you can assume that the number of rows in **A** equals the number of elements in **b** and the number of elements in **`order`**. You cannot assume that **A** will be 14x14, nor that **b** and **`order`** will have 14 elements.)

- Create a function using **`for`** loops and **`if`** blocks to determine which element of a 1D numpy array has the largest magnitude (in absolute value). Use the following opening definition:
  ```
  def max_magnitude(x):
  ```
  Your function will return the Python integer index and the value of the element with the maximum magnitude in that order. Note that the actual value of the element must be returned (whether positive or negative), not the positive magnitude. Thus, if the 0th element of the array had the maximum magnitude of -100.0, this function would return `0, -100.0` (in that order).

  When passed an array of truss member forces, this function will return the Python index of the member with the largest tension or compression, as well as the amount of tension (if positive) or compression (if negative) in that member.

  Note that your function must use an **`if`** statement in a **`for`** loop to find the maximum tension or compression. Do not use a NumPy or other library function to directly find the maximum value. You may use the Python function **`abs`** where the statement **`y = abs(x)`** will assign the absolute value of scalar **`x`** to scalar variable **`y`**.

- Finally, create the function that will use all of the above functions to determine the Python index of and force in the truss member with the largest tension or compression given specifications on the geometry and loading of the system. Use the following opening definition:

`def maximum_force_member(L_GA, L_GD, load, position, L_beam, W_beam, order):`

where the parameters are the same as used in the functions above. Just like function **`max_magnitude`**, this function will return the Python index of the member with the largest tension or compression, as well as the amount of tension (if positive) or compression (if negative) in that member.

You must use **`solve`** from your **`linsolve`** module to solve the linear system. Do not use a library function to get the solution.

- Make sure all of your functions have appropriate multiline docstrings and comments.

- Just because Submitty gives you a score is not a guarantee of getting that score on the assignment. If a manual inspection of your code shows less than useful comments and docstrings, or the inappropriate use of NumPy or other library functions instead of writing and using your own code as specified, your score may be reduced – even to zero – at the discretion of the NuMPY team. You may use **`math`** library functions such as **`cos`**, **`sin`**, **`acos`**, **`atan`**, **`pi`**, etc., as needed.

Appendix A: the incomplete code the truss design engineer started to write

```
debug = True

def applied_load(load, position, L_beam, W_beam):
    '''Define vector of forces applied to truss based on hoist position and loads.

    Input:  load     - weight that the hoist carries plus the hoist [kN]
            position - position of hoist along beam [meters]
            L_beam   - length of the I beam that the hoist transits [meters]
            W_beam   - weight of the I beam
    Output: f        - 14 element 1D numpy array of applied forces at joints [kN]

    Define 14 element right side vector of applied forces at joints.

    The weight of the I beam is assumed to be equally shared at the
    two joints F and D.

    The weight of the load (including the hoist) is assumed to be linearly
    distributed between the two joints F and D.  For example,
     - position = 0        => 100% of load is at joint F,   0% at joint D;
     - position = L_beam/4 =>  75% of load is at joint F,  25% at joint D;
     - position = L_beam   =>   0% of load is at joint F, 100% at joint D.
    '''

    # figure out the equations to specify the elements of f, and then make f

    if (debug):
        print("hoist load [kN] =", load)
        print("hoist position [m] =", position)
        print("I beam length [m], weight [kN] =", L_beam, W_beam)
        print("truss load vector f [kN]:")
        print(f)
        print()

    return f
```

Appendix B: revisiting Assignment 1 (the simple three-bar truss)

To most easily compute the truss member and support reaction forces in Assignment 1 you needed to manually determine the order to solve the force balance equations obtained from the method of joints. You have now learned the computational tools needed to solve the same problem more easily as a linear system. One way to write that force balance model for the Assignment 1 three bar simple truss is as follows.

$$\begin{array}{ccccccc} & & \underline{\underline{\mathbf{A}}} & \underline{\mathbf{x}} & = & \underline{\mathbf{b}} \\ \begin{array}{l} \Sigma F_{y,B} \\ \Sigma F_{y,C} \\ \Sigma F_{x,C} \\ \Sigma F_{x,A} \\ \Sigma F_{y,A} \\ \Sigma F_{x,B} \end{array} & : & \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & \cos(60) & 0 & 0 & 0 & 0 \\ 0 & \cos(150) & 1 & 0 & 0 & 0 \\ 0 & \cos(330) & 0 & 1 & 0 & 0 \\ 1 & \cos(240) & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix} & \begin{bmatrix} T_{AB} \\ T_{AC} \\ T_{BC} \\ A_x \\ A_y \\ B_x \end{bmatrix} & = & \begin{bmatrix} 0 \\ Fsin(\theta) \\ Fcos(\theta) \\ 0 \\ 0 \\ 0 \end{bmatrix} \end{array}$$

Since you know the solution to this system you can use this as one possible test case to check that your code for this assignment works.

Also notice the patterns that you see in the $\underline{\underline{\mathbf{A}}}$ matrix. For example, note that the geometry information in columns 1 and 3 (using human indexing) pairs with reaction forces. The angles in column 2 come in pairs as well, opposite each other by 180 degrees. Similar patterns occur in other truss problems, and patterns are evident in many linear system problems in general. Looking for anomalies in patterns can help you detect typos and missing terms in a matrix model.

If you use this test problem we again strongly recommend you use a separate file in the same directory with a statement like "**`from hw03 import`** …" so that you don't pollute your **`linsolve`** and **`hw03`** modules with test code that you'll need to comment out before submitting to Submitty.

Appendix C: swapping rows in a matrix

The example in Appendix B has non-zero elements along the main diagonal (a condition for the LU factorization to succeed), and furthermore it will not generate zeros on the main diagonal during the LU factorization. Thus, LU-factoring the above will succeed.

The LU-factorization of the given 14x14 matrix will not succeed because there are zeros on the main diagonal. (Try it.) You will need to figure out what order to shuffle the rows of $\underline{\underline{\mathbf{G}}}$ into so that the LU-factorization will succeed. This can be done by manually swapping rows, tracking your swaps, and setting up the corresponding **`order`** vector. This can also be done by writing a short code along the following lines and manually adjusting the swaps until you are satisfied.

```
from hw03 import shuffle_Ab
order = np.array([0, 1, 2,…])
# make the matrix
# shuffle the matrix
# print the shuffled matrix and check it; adjust `order` as needed until you are satisfied
```