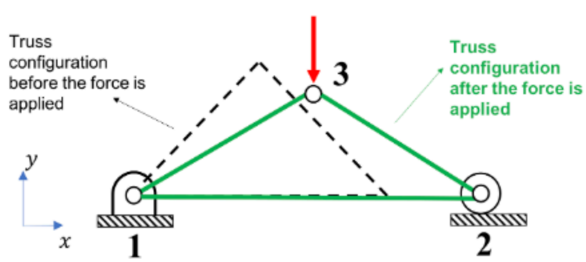
MP2.1. Using linear system of equation to solve physical problems

Simulations and design using the finite element method

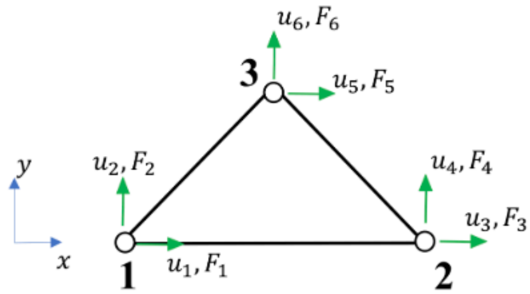
The finite element method (FEM) is a numerical technique for finding approximate solutions of many physics and engineering problems by discretization of the domain into elements. The technique has many different applications, including problems in stress analysis, fluid mechanics, heat transfer, vibrations, electrical and magnetic fields, etc.

In this MP, we will obtain the solution for a truss system. The cranes illustrated below are good examples of truss systems.

A simple truss system is illustrated below, including 3 rods connected by joints, here called nodes. Node 1 is fixed (cannot "move"). Node 2 can only move in the horizontal direction (along x-axis). Each rod can only deform in its longitudinal direction, but cannot bend or twist. For example, when a vertical force is applied downwards at the node 3, the truss will "move" as indicated.



How can we describe all the possible forces acting on the truss system and the resulting movement of the nodes? To do so, let's define two vectors x and F that represent the movements and forces respectively. Forces can be applied either horizontally or vertically to each node, leading to a displacement (movement) of the node in either the x or y direction respectively.



According to our choice of coordinate system, forces will be positive when pointing up or to the right, otherwise they are negative. The displacement of a node is the vector that connects the position of the node before the force is applied to the new position of the node after the force is applied. In general, we can encode this information in our vectors as follows: u_{2i-1} and u_{2i} is the displacement of node i in the x and y direction respectively and F_{2i-1} and F_{2i} is the force acting upon the i^{th} node in the horiztontal and vertical directions respectively. Following this notation, for the truss example above, we have u_1 and u_2 representing the displacements of node 1, u_3 and u_4 representing the displacements of node 2, and u_5 and u_6 representing the displacements of node 3. Our vector `x` representing the displacements of the truss system is therefore `x = np.array([u1,u2,u3,u4,u5,u6])`.

The forces are recorded in a similar manner. We use F_1 and F_2 to represent the horizontal and vertical forces acting on node 1, F_3 and F_4 to represent the horizontal and vertical forces acting on node 2, and F_5 and F_6 to represent the horizontal and vertical forces acting on node 3. The resulting force array is defined as `F = np.array([F1,F2,F3,F4,F5,F6])`.

Write a code snippet that defines the arrays `x` and `F` representing the displacements and forces for the example above (three connected rods with a vertical force at node 3). Assume that node 2 has displacements $(1, 0)$ and node 3 has displacement $(1, -4)$ when a force of magnitude 13 is applied vertically at node 3.

Your code snippet should define the following variable(s) and/or function(s):

Name	Type	Description
`x`	1d numpy array	contains all the displacements for each node
`F`	1d numpy array	contains all the applied forces for each node

user_code.py

```
import numpy as np
```

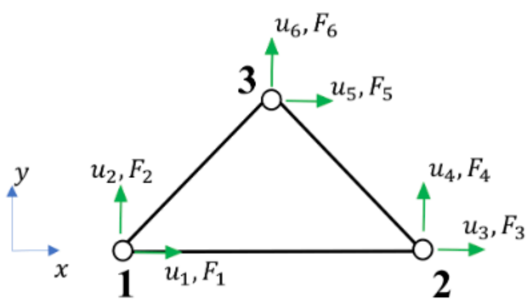
MP2.2. Introducing the linear system of equations

How to obtain the displacements for a given set of forces?

The equilibrium condition of the truss system can be represented by the following system of linear equations:

$$\mathbf{K}\mathbf{x} = \mathbf{F}$$

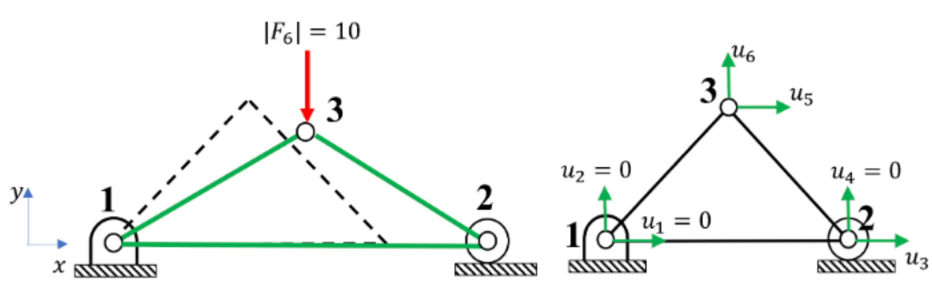
where $\mathbf{K}$ is called the stiffness matrix. The stiffness matrix has information about the geometry of the problem, and the material utilized for each rod. $\mathbf{x}$ and $\mathbf{F}$ are the displacement and force vectors described in the previous page.



For the general example above (truss with three nodes), the linear system of equations is given as:

$$\begin{bmatrix} k_{11} & k_{12} & k_{13} & k_{14} & k_{15} & k_{16} \\ k_{21} & k_{22} & k_{23} & k_{24} & k_{25} & k_{26} \\ k_{31} & k_{32} & k_{33} & k_{34} & k_{35} & k_{36} \\ k_{41} & k_{42} & k_{43} & k_{44} & k_{45} & k_{46} \\ k_{51} & k_{52} & k_{53} & k_{54} & k_{55} & k_{56} \\ k_{61} & k_{62} & k_{63} & k_{64} & k_{65} & k_{66} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \end{bmatrix} = \begin{bmatrix} F_1 \\ F_2 \\ F_3 \\ F_4 \\ F_5 \\ F_6 \end{bmatrix}$$

Without loss of generality, we will now consider the example from the previous page, where node 1 is fixed, node 2 is free to move in the x-direction, and node 3 is subjected to a vertical force with a magnitude of $10\,\mathrm{N}$.



Once we substitute the **known** displacements and force values in the above equations, we get:

$$\begin{bmatrix} k_{11} & k_{12} & k_{13} & k_{14} & k_{15} & k_{16} \\ k_{21} & k_{22} & k_{23} & k_{24} & k_{25} & k_{26} \\ k_{31} & k_{32} & k_{33} & k_{34} & k_{35} & k_{36} \\ k_{41} & k_{42} & k_{43} & k_{44} & k_{45} & k_{46} \\ k_{51} & k_{52} & k_{53} & k_{54} & k_{55} & k_{56} \\ k_{61} & k_{62} & k_{63} & k_{64} & k_{65} & k_{66} \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ u_3 \\ 0 \\ u_5 \\ u_6 \end{bmatrix} = \begin{bmatrix} F_1 \\ F_2 \\ 0 \\ F_4 \\ 0 \\ -10 \end{bmatrix}$$

The entries u_3, u_5, u_6 are the **unknown** displacements we want to find, and F_1, F_2, F_4 are the **unknown** reaction forces corresponding to the prescribed (known) displacements. Assuming we know all the entries k_{ij} of the stiffness matrix, how can we solve the linear system of linear equations $\mathbf{Kx} = \mathbf{F}$ above? Note that both $\mathbf{x}$ and $\mathbf{F}$ have **unknown** quantities.

- (a)

```
x = scipy.linalg.solve(K,F)
```

- (b)

```
P, L, U = scipy.linalg.lu(K)
y = scipy.linalg.solve_triangular(L, np.dot(P.T, F), lower=True)
x = scipy.linalg.solve_triangular(U, y)
```

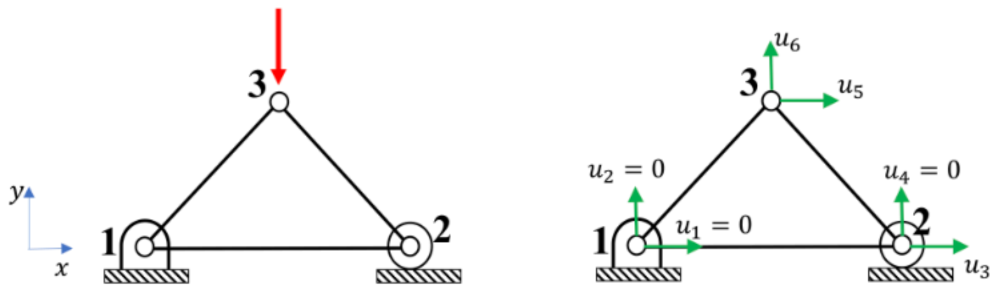
- (c) Swap columns and rows of $\mathbf{K}$ and solve a smaller system of linear equations
- (d) None of the above

Save & Grade Save only

MP2.3. Strategy to solve KU=P

Partitioning the stiffness matrix

The linear system of equations $\mathbf{Kx} = \mathbf{F}$ cannot be solved directly in this format, since there are unknowns in both the displacement ($\mathbf{x}$) and the force vector ($\mathbf{F}$). In the following, we will propose a strategy to solve this type of system. We will use the corresponding numerical values for the stiffness matrix $\mathbf{K}$ for the example below, without loss of generality.



$$\mathbf{Kx} = \mathbf{F} \implies \begin{bmatrix} 20 & 10 & -10 & 0 & -10 & -10 \\ 10 & 10 & 0 & 0 & -10 & -10 \\ -10 & 0 & 20 & -10 & -10 & 10 \\ 0 & 0 & -10 & 10 & 10 & -10 \\ -10 & -10 & -10 & 10 & 20 & 0 \\ -10 & -10 & 10 & -10 & 0 & 20 \end{bmatrix} \begin{bmatrix} u_1 = 0 \\ u_2 = 0 \\ u_3 \\ u_4 = 0 \\ u_5 \\ u_6 \end{bmatrix} = \begin{bmatrix} F_1 \\ F_2 \\ 0 \\ F_4 \\ 0 \\ -10 \end{bmatrix}$$

Let's swap rows and columns 3 and 4, such that the known quantities "stay together". In general, rows and columns of $\mathbf{K}$ are reorganized such that the equations associated with the nodes that have prescribed (known) displacements and unknown reaction forces are positioned in the first rows.

$$\hat{\mathbf{K}}\hat{\mathbf{x}} = \hat{\mathbf{F}} \implies \begin{bmatrix} 20 & 10 & 0 & -10 & -10 & -10 \\ 10 & 10 & 0 & 0 & -10 & -10 \\ 0 & 0 & 10 & -10 & 10 & -10 \\ -10 & 0 & -10 & 20 & -10 & 10 \\ -10 & -10 & 10 & -10 & 20 & 0 \\ -10 & -10 & -10 & 10 & 0 & 20 \end{bmatrix} \begin{bmatrix} u_1 = 0 \\ u_2 = 0 \\ u_4 = 0 \\ u_3 \\ u_5 \\ u_6 \end{bmatrix} = \begin{bmatrix} F_1 \\ F_2 \\ F_4 \\ 0 \\ 0 \\ -10 \end{bmatrix}$$

The two linear system of equations above are equivalent (you can check that!). We will learn soon how to solve the system $\hat{\mathbf{K}}\hat{\mathbf{x}} = \hat{\mathbf{F}}$ in an efficient way. But first, how can we construct the matrix $\hat{\mathbf{K}}$ if we have the matrix $\mathbf{K}$?[1]

To accomplish this, we will provide the array `equation_numbers` which contains the order in which the displacements u_i appear in the array $\hat{\mathbf{x}}$. For the example above, we have `equation_numbers = np.array([1,2,4,3,5,6])`. Note that `equation_numbers` is **1-indexed**.

In your code snippet, define the function reorder_rows_columns, that takes as argument the matrix $\mathbf{K}$ and the array equation_numbers and returns the matrix $\hat{\mathbf{K}}$.

```
def reorder_rows_columns(K,equation_numbers):
    # construct the matrix Khat
    return Khat
```

The setup code provides K and equation_numbers from the example above. You can use it for debugging purposes (since you know what $\hat{\mathbf{K}}$ should look like), by calling reorder_rows_columns(K,equation_numbers). However we will test your function using different input arguments.

The setup code provides the following variable(s) and/or function(s):

Name	Type	Description
K	2d numpy array	stiffness matrix corresponding to the example (to help debugging)
equation_numbers	1d numpy array	ordering of the displacements in xhat

Your code snippet should define the following variable(s) and/or function(s):

Name	Type	Description
reorder_rows_columns	function	change the order of rows and columns of a given matrix

(1): In typical FEA algorithms, the matrix $\hat{\mathbf{K}}$ is constructed directly, without the need of constructing the matrix $\mathbf{K}$ first.

user_code.py

```
import numpy as np

def reorder_rows_columns(K,equation_numbers):
    # construct the matrix Khat
    return Khat
```

Restore original file

MP2.4. Strategy to solve KU=P (continue)

Partitioning the stiffness matrix

Now that you know how to construct the matrix $\hat{\mathbf{K}}$, we can rewrite the linear system of equations as:

$$\hat{\mathbf{K}}\hat{\mathbf{x}} = \hat{\mathbf{F}} \implies \begin{bmatrix} \mathbf{K}_{pp} & \mathbf{K}_{pf} \\ \mathbf{K}_{fp} & \mathbf{K}_{ff} \end{bmatrix} \begin{bmatrix} \mathbf{x}_p \\ \mathbf{x}_f \end{bmatrix} = \begin{bmatrix} \mathbf{F}_p \\ \mathbf{F}_f \end{bmatrix}$$

Using the same example from the previous page

$$\begin{bmatrix} 20 & 10 & 0 & -10 & -10 & -10 \\ 10 & 10 & 0 & 0 & -10 & -10 \\ 0 & 0 & 10 & -10 & 10 & -10 \\ -10 & 0 & -10 & 20 & -10 & 10 \\ -10 & -10 & 10 & -10 & 20 & 0 \\ -10 & -10 & -10 & 10 & 0 & 20 \end{bmatrix} \begin{bmatrix} u_1 = 0 \\ u_2 = 0 \\ u_4 = 0 \\ u_3 \\ u_5 \\ u_6 \end{bmatrix} = \begin{bmatrix} F_1 \\ F_2 \\ F_4 \\ 0 \\ 0 \\ -10 \end{bmatrix}$$

we obtain the partitioned matrices:

$$\mathbf{K}_{pp} = \begin{bmatrix} 20 & 10 & 0 \\ 10 & 10 & 0 \\ 0 & 0 & 10 \end{bmatrix} \qquad \mathbf{K}_{pf} = \begin{bmatrix} -10 & -10 & -10 \\ 0 & -10 & -10 \\ -10 & 10 & -10 \end{bmatrix}$$

$$\mathbf{K}_{fp} = \begin{bmatrix} -10 & 0 & -10 \\ -10 & -10 & 10 \\ -10 & -10 & -10 \end{bmatrix} \qquad \mathbf{K}_{ff} = \begin{bmatrix} 20 & -10 & 10 \\ -10 & 20 & 0 \\ 10 & 0 & 20 \end{bmatrix}$$

and the known vectors:

$$\mathbf{x}_p = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \qquad \mathbf{F}_f = \begin{bmatrix} 0 \\ 0 \\ -10 \end{bmatrix}$$

The following list provides the dimensions of all arrays defined above:

- `nk`: number of known prescribed displacements
- `nf`: number of unknown displacements (the ones we are trying to solve for)
- `xp`: prescribed (known) displacements with shape `(nk,)`
- `xf`: unknown displacements with shape `(nf,)`
- `Fp`: reaction (unknown) forces with shape `(nk,)`
- `Ff`: applied (known) forces with shape `(nf,)`
- `Kff`: block of stiffness matrix with shape `(nf,nf)`
- `Kfp`: block of stiffness matrix with shape `(nf,nk)`
- `Kpf`: block of stiffness matrix with shape `(nk,nf)`
- `Kpp`: block of stiffness matrix with shape `(nk,nk)`

Write the function partition_stiffness_matrix that takes as argument the stiffness matrix K (the "original" one, prior to swapping), the array equation_numbers, the number of known displacements nk and returns the matrices Kff, Kfp, Kpf and Kpp. The setup code provides the function reorder_rows_columns that you implemented in the previous page (no need to copy the function here).

```
def partition_stiffness_matrix(K,equation_numbers,nk):
    # construct the smaller matrices
    return  Kpp,Kpf,Kfp,Kff
```

Similar to the previous page, you can use the given variables K and equation_numbers for debugging purposes, but we will check your code using a different example. The setup code provides the following variable(s) and/or function(s):

Name	Type	Description
K	2d numpy array	stiffness matrix corresponding to the example above (to help debugging)
equation_numbers	1d numpy array	ordering of the displacements for partition
nk	integer	number of known displacements
reorder_rows_columns	function	change the order of rows and columns of a matrix

Your code snippet should define the following variable(s) and/or function(s):

Name	Type	Description
partition_stiffness_matrix	function	constructs the smaller matrices

user_code.py

```
import numpy as np

def partition_stiffness_matrix(K,equation_numbers,nk):
    # construct the smaller matrices
    return  Kpp,Kpf,Kfp,Kff
```

Restore original file

MP2.5. Solving KU=P

The rearranged linear system of equations

$$\begin{bmatrix} \mathbf{K}_{pp} & \mathbf{K}_{pf} \\ \mathbf{K}_{fp} & \mathbf{K}_{ff} \end{bmatrix} \begin{bmatrix} \mathbf{x}_p \\ \mathbf{x}_f \end{bmatrix} = \begin{bmatrix} \mathbf{F}_p \\ \mathbf{F}_f \end{bmatrix}$$

yields the following two system of linear equations:

$$\mathbf{K}_{fp}\mathbf{x}_p + \mathbf{K}_{ff}\mathbf{x}_f = \mathbf{F}_f$$

$$\mathbf{K}_{pp}\mathbf{x}_p + \mathbf{K}_{pf}\mathbf{x}_f = \mathbf{F}_p$$

We use the first system of equations to solve for the unknown displacement $\mathbf{x}_f$

$$\mathbf{K}_{ff}\mathbf{x}_f = \mathbf{F}_f - \mathbf{K}_{fp}\mathbf{x}_p$$

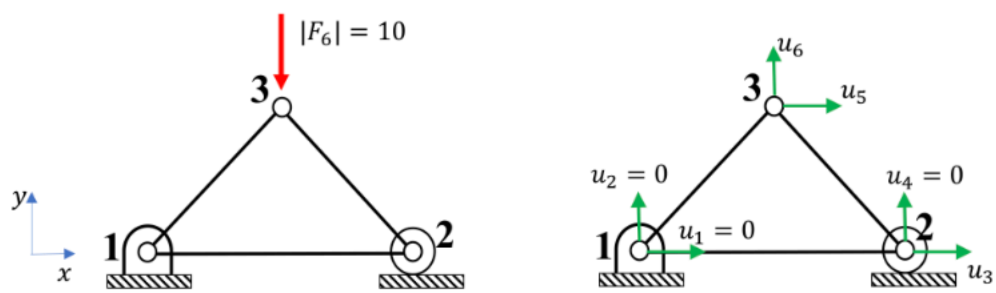
and once $\mathbf{x}_f$ is known we use the second system of equations to obtain the unkown reaction force $\mathbf{F}_p$

$$\mathbf{F}_p = \mathbf{K}_{pp}\mathbf{x}_p + \mathbf{K}_{pf}\mathbf{x}_f$$

Write the function `fea_solve` to calculate the unkown variables defined above. Your function should have the following syntax:

```
def fea_solve(Kpp,Kpf,Kfp,Kff,xp,Ff):
    # do stuff here
    return xf,Fp
```

You can use any existing functions such as `linalg.solve` in your calculations. Use the function `fea_solve` to obtain the displacement `xf` and force `Fp` for the example below.



The setup code provides the variables needed to obtain the partitioned matrices. You will need to construct the arrays `xp` and `Ff` based on the figure above, and the knowledge that you learned in the last pages of this MP.

Once you obtain the solution `xf`, use the provided function `plot_truss(xf)` to plot the before and after configuration of the truss. Store your result in the variable `image_xf`.

We will also check the function `fea_solve` using other examples. Make sure you are not hard-coding the solution!

The setup code provides the following variable(s) and/or function(s):

Name	Type	Description
`K`	2d numpy array	stiffness matrix corresponding to the example above
`equation_numbers`	1d numpy array	ordering of the displacements for partition
`partition_stiffness_matrix`	function	constructs smaller matrices (same function you defined in page 4)
`plot_truss`	function	plot the displacement solution for the truss

Your code snippet should define the following variable(s) and/or function(s):

Name	Type	Description
`fea_solve`	function	solve the linear system of equations
`Ff`	1d numpy array	known applied force
`xp`	1d numpy array	known prescribed displacement
`xf`	1d numpy array	unknown displacement
`image_xf`	image	plot of the displacement response corresponding to the example above

user_code.py

```
import numpy as np
import scipy.linalg as la
import matplotlib.pyplot as plt

def fea_solve(Kpp,Kpf,Kfp,Kff,xp,Ff):
    # do stuff here
    return xf,Fp
```

Restore original file

MP2.6. LU and Triangular Solve

In the previous page, you solved for the unknown displacements $\mathbf{x_f}$ using `linalg.solve`. In this part, we ask you to manually code what happens under the hood when calling `linalg.solve`! Write the following functions:

- `my_lu` which computes and LU factorization (without pivoting) and outputs it into a single matrix $\mathbf{M}$.
- `my_triangular_solve` which implements forward and backward solve.
- `fea_solve` which puts everything together to solve the truss system. You only need to modify the function that you defined in the previous page, so that it now uses your own LU factorization and triangular solve functios.

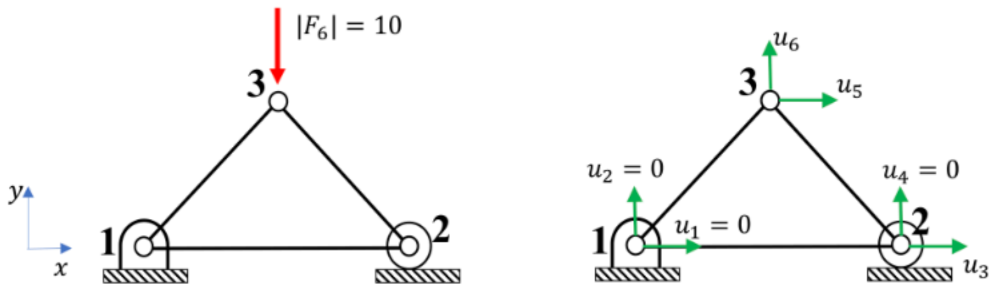
Your functions must be written in the following format for testing purposes:

```
def my_lu(A):
    # LU Factorization
    return M

def my_triangular_solve(M, b):
    # b: 1d array with the right-hand of the linear system of equations
    # implement Forward and Backward substitution
    return x

def fea_solve(Kpp, Kpf, Kfp, Kff, xp, Ff):
    # use my_lu and my_triangular_solve
    # to solve the partitioned system
    return xf, Fp
```

Note You may not use use any canned functionality to implement the solve function. You may not use any functions from `numpy.linalg` or `scipy.linalg`. Some examples of functions that are off limits are `numpy.linalg.inv`, `scipy.linalg.inv`, `scipy.linalg.lu`, `scipy.linalg.lu_factor`, `numpy.linalg.solve`, `scipy.linalg.solve`, `scipy.linalg.solve_triangular`. These are just examples and are not exhaustive. You must write your own LU factorization and triangular solve functions.

The setup code provides variable(s) and/or function(s) corresponding to the example below. They are given to help you with debugging, but we will be testing your functions `my_lu`, `my_triangular_solve` and `fea_solve` using different examples.

The setup code provides variable(s) and/or function(s) corresponding to the example below. They are given to help you with debugging, but we will be testing your functions `my_lu`, `my_triangular_solve` and `fea_solve` using different examples.



Name	Type	Description
`K`	2d numpy array	stiffness matrix corresponding to the example above (to help debugging)
`equation_numbers`	1d numpy array	ordering of the displacements for partition
`partition_stiffness_matrix`	function	constructs smaller matrices (same function you defined in page 4)

Your code snippet should define the following variable(s) and/or function(s):

Name	Type	Description
`my_lu`	function	A function that takes in a matrix A and returns its LU factorizaton A=LU without pivoting
`my_triangular_solve`	function	A function that takes in an LU factorization as a single 2D numpy array and a right-hand side vector and uses the factorization to solve the linear system.
`fea_solve`	function	A function that solves the full linear system corresponding to the truss system

user_code.py

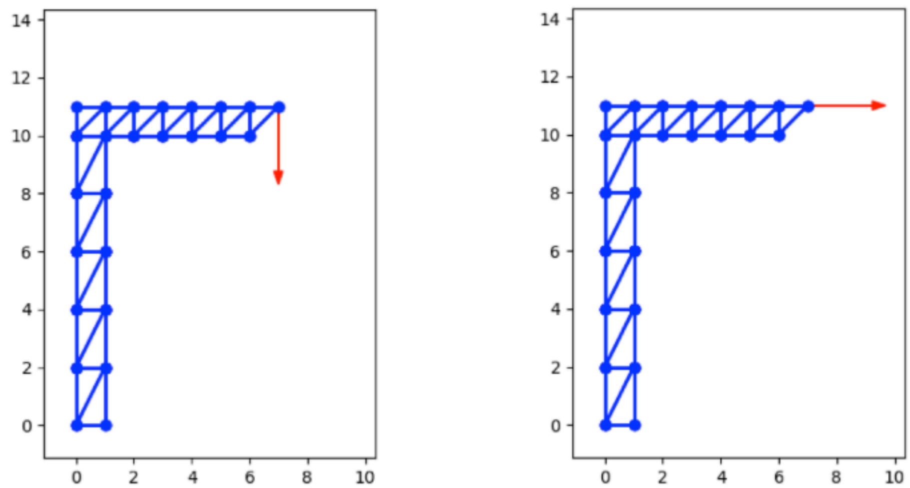
```python
import numpy as np


def my_lu(A):
    # The upper triangular matrix U is saved in the upper part of the matrix M (including the diagonal)
    # The lower triangular matrix L is saved in the lower part of the matrix M (not including the diagonal)
    # Do NOT use `scipy.linalg.lu`!
    # You should not use pivoting

    M = ...
    return M


def my_triangular_solve(M, b):
    # A = LU (L and U are stored in M)
    # A x = b (given A and b, find x)
    # M is a 2D numpy array
    # The upper triangular matrix U is stored in the upper part of the matrix M (including the diagonal)
    # The lower triangular matrix L is stored in the lower part of the matrix M (not including the diagonal)
    # b is a 1D numpy array
    # x is a 1D numpy array
    # Do not use `scipy.linalg.solve_triangular`

    x = ...

    return x


def fea_solve(Kpp, Kpf, Kfp, Kff, xp, Ff):
    # Use my_lu and my_triangular_solve

    xf = ...
    Fp = ...
    return xf, Fp
```

Restore original file

Save & Grade Save only

MP2.7. Solve a Truss System

We now want to solve truss systems under different loading conditions. The figure below indicates two different force configurations applied to the same truss structure. Hence when solving $\mathbf{K}\mathbf{x} = \mathbf{F}$, the stiffness matrix is the same for both configurations.



Force configuration 1 Force configuration 2

For the example above, the setup code provides the already partitioned matrices `kpp`, `kpf`, `kfp` and `kff`, the known prescribed displacement `xp` and the applied force `Ff`. Note here that `Ff` is given as a 2d numpy array, where the first column of the array corresponds to "Force configuration 1" and the second column corresponds to "Force configuration 2".

Write a code snippet that computes the displacement solution `xf_1` corresponding to the force configuration 1, and the displacement solution `xf_2` corresponding to the force configuration 2. Use the `plot_truss(xf)` function given by the setup code to plot these displacement solutions. Store the images in the variables `image_xf_1` and `image_xf_2`.

The setup code provides the functions `my_lu` and `my_triangular_solve`, so you do not need to copy your functions again here. **Note:** You must use these given functions to solve the linear system of equations, since you are not allowed to use any canned functionality such as `numpy.linalg.inv`, `scipy.linalg.inv`, `scipy.linalg.lu`, `scipy.linalg.lu_factor`, `numpy.linalg.solve`, `scipy.linalg.solve`, `scipy.linalg.solve_triangular`.

Name	Type	Description
Kpp	2d numpy array	subset of stiffness matrix
Kpf	2d numpy array	subset of stiffness matrix
Kfp	2d numpy array	subset of stiffness matrix
Kff	2d numpy array	subset of stiffness matrix
Ff	2d numpy array	first column with applied force of configuration 1 and second column with applied force of configuration 2
xp	1d numpy array	known prescribed displacements
my_lu	function	function that takes in a matrix A and returns its LU factorizaton A=LU without pivoting
my_triangular_solve	function	function that takes in an LU factorization as a single 2D numpy array and a right-hand side vector and uses the factorization to solve the linear system.
plot_truss	function	plot the displacement solution for the truss

Your code snippet should define the following variable(s) and/or function(s):

Name	Type	Description
xf_1	1d numpy array	displacement response corresponding to force configuration 1
xf_2	1d numpy array	displacement response corresponding to force configuration 2
image_xf_1	image	displacement response corresponding to force configuration 1
image_xf_2	image	displacement response corresponding to force configuration 2

user_code.py

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