Project: Truss Solver

Nicholas Gower

February 19, 2021

1 The Problem

***6–4.** Determine the force in each member of the truss and state if the members are in tension or compression.

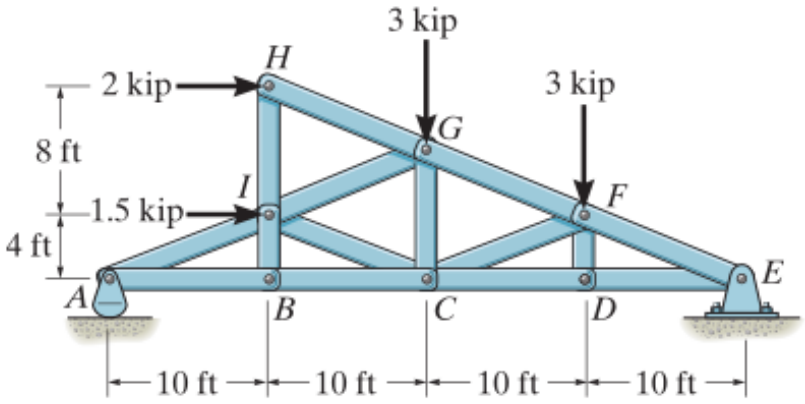

Figure 1: Problem 6.4 in Engineering Mechanics: Statics.

Based on this problem, all forces will be measured in kip, and all distances will be in feet. The origin will be where pin A is located. The +x axis will be parallel with member AB, and the +y axis will be parallel with member BI.

2 Generation of Augmented Matrix

For every pin in the truss, there is one equation in $\mathbb{R}^2$ in the form $\vec{0} = \sum \vec{T} + \sum \vec{F}_{ext}$, where $\sum \vec{T}$ is the sum of all tension forces from members attached to the pin, and $\sum \vec{F}_{ext}$ is the sum of all external applied and reaction forces. If θ represents the angle of a vector measured counterclockwise from the positive x axis, $M\angle\theta = \begin{bmatrix} M\cos(\theta) \\ M\sin(\theta) \end{bmatrix}$, and pin α is connected to pins $[p_1, p_2, p_3..., p_n]$ at angles of $[\theta_1, \theta_2, \theta_3..., \theta_n]$,

$$\sum \vec{T}_\alpha = T_{\alpha 1}\angle\theta_1 + T_{\alpha 2}\angle\theta_2 + T_{\alpha 3}\angle\theta_3 + ... + T_{\alpha n}\angle\theta_n$$

Based on this, I wrote two Matlab lists that describe the positions of each of the pins, and one matrix that describes which pins each pin is connected to.

```
pinsX=[0,10,20,30,40, 30,20,10,10 ]; %Positions of each pin
pinsY=[0,0,0,0,0, 4 ,8 ,12 ,4];
connection=zeros(9,5);

% Pin A=1
% Pin B=2
% Pin C=3
% Pin D=4
% Pin E=5
% Pin F=6
% Pin G=7
% Pin H=8
% Pin I=9

connection(1,:)=[2,9,0,0,0]; % Pin A is connected with pins B and I
connection(2,:)=[1,3,9,0,0]; % Pin B is connected with pins A, C, and I
connection(3,:)=[2,9,7,6,4]; % etc.
connection(4,:)=[3,6,5,0,0];
connection(5,:)=[4,6,0,0,0];
connection(6,:)=[7,3,4,5,0];
connection(7,:)=[8,9,3,6,0];
connection(8,:)=[9,7,0,0,0];
connection(9,:)=[1,2,3,7,8];
```

The validity of the pinsX and pinsY lists can be quickly demonstrated with the command "scatter(pinsX,pinsY)," with the plot shown in Figure 2 being the result.

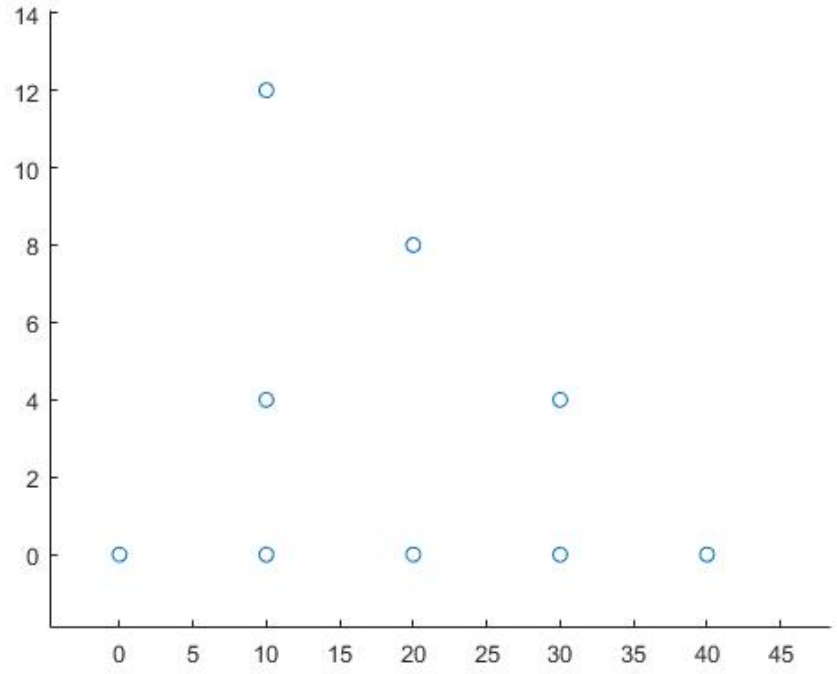

Figure 2: The result of entering "scatter(pinsX,pinsY)" into the Matlab command window.

For every pin-pin connection, there is one tension force, so for every two non-zero values in the connection matrix, there is one tension force.

```
for i=1:arraySize(1)
   for j=1:arraySize(2)
     if connection(i,j) ~=0
        numMembers=numMembers+0.5; %For every two pin-pin connections, there is one member.
      end
   end
end
```

I needed a standardized method of identifying and assigning forces, where the tension force from one pin to another is the same force that is applied when the pins are looked at in reverse. I did this by creating a 2D matrix "memberForceIDsystem" and string list "forceNames, initally containing only the external reaction forces." Whenever a tension force needs an ID, it first checks memberForceIDsystem by entering the two pin IDs in ascending order. If the found ID is zero, it will create a new ID by adding one new name to forceNames, then making the ID equal to the length of forceNames. It will then copy this ID to memberForceIDsystem in the index it just checked, so that the other pin connected to the member can use the same force ID.

```
memberForceIDsystem=zeros(numMembers); %Enter two pins, get column of force in forceMatrix
forceNames=["A","Ex","Ey"];
```

To prepare for program's loop that generates the force matrix, I calculated the total number of unknown forces that exist in the system. Three external and 15 internal forces adds up to 18 unknown forces. Because there are nine pins, there are 18 equations, so the problem is solvable.

```
numNonMemberForces=length(forceNames);
forceMatrix=zeros(2*length(pinsX),numMembers+numNonMemberForces+1); %Augmented matrix
%Will be 18 x 19 matrix, so problem is solvable.
numForces=numMembers+numNonMemberForces;
```

Inside this loop, for every pin, the program generates a 2x19 augmented force matrix. For every pin this pin is connected to, it generates the tension force associated with that connection. The angle of a tension force is calculated using the relative displacement of the pin it's associated with. After doing this, external forces are individually added based on the ID of the pin. At the end of each iteration of the loop, the 2x19 matrix is added to the main matrix in the appropriate location.

```
for i=1:length(pinsX) %For every pin in the truss
   pin=[pinsX(i),pinsY(i)];
   forces=zeros(2,numForces+1);
   connections=connection(i,:);
   for n=1:length(connections) %Adds tension forces
      if connections(n)~=0
      otherPin=[pinsX(connections(n)),pinsY(connections(n))];
      displacement=otherPin-pin;
      magnitude=getMagnitude(displacement);

      xAxis=[1,0];
      angle=acosd(dot(displacement,xAxis)/(magnitude)); %Finds angle of tension vector using dot product
      if displacement(2)<0 %Flips sign of angle if y component is negative
         angle=-angle;
      end
```

```
        pinIDs=[i,connections(n)];
        %forceID=(length(pinsX)+1)*min(pinIDs)+max(pinIDs);
        firstPin=min(pinIDs);
        secondPin=max(pinIDs);
        forceID=memberForceIDsystem(firstPin,secondPin); % Gets force ID from array
        if forceID==0 %If force ID not set yet
           forceID=length(forceNames)+1; %Generate new ID
           memberForceIDsystem(firstPin,secondPin)=forceID;
           %Send new ID to array, so 3rd law pair can use the same ID
           forceName=strcat('T_',number2letter(firstPin),number2letter(secondPin)); %Generate new force name
           forceNames(forceID)=forceName; %Send name of force to name list
        end

        forces(1,forceID)=cosd(angle); %Add tension force with angle
        forces(2,forceID)=sind(angle);

        end
     end

  %Applied forces and non-tension reaction forces

     if i==1 %Pin A
        forces(2,1)=1; % Rocker A reaction force
     elseif i==5 %Pin E
        forces(1,2)=1; % Fixed pin E reaction force
        forces(2,3)=1;
     elseif i==6
        forces(2,length(forces))=3; % -3 kip applied force down
     elseif i==7
        forces(2,length(forces))=3; % -3 kip applied force down
     elseif i==8
        forces(1,length(forces))=-2; % 2 kip applied force right
     elseif i==9
        forces(1,length(forces))=-1.5; % 1.5 kip applied force right
     end


     forceMatrix(i*2-1,:)=forces(1,:); %Adds force equations to main matrix
     forceMatrix(i*2,:)=forces(2,:);
  end
```

This augmented force matrix is split into a square matrix and sum vector to prepare it for solving.

```
A=forceMatrix(:,1:end-1);
b=forceMatrix(:,end);
```

2.1 Resulting Matrices

$$\mathbf{A}\vec{x} = \vec{b}$$

$$\mathbf{A} = \begin{pmatrix}
0 & 0 & 0 & 1 & \frac{5\sqrt{29}}{29} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & \frac{2\sqrt{29}}{29} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -1 & 0 & -\frac{5\sqrt{29}}{29} & 0 & \frac{5\sqrt{29}}{29} & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{2\sqrt{29}}{29} & 1 & \frac{2\sqrt{29}}{29} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -\frac{5\sqrt{29}}{29} & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{2\sqrt{29}}{29} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\frac{5\sqrt{29}}{29} & 0 & 0 & 0 & \frac{5\sqrt{29}}{29} & -\frac{5\sqrt{29}}{29} & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\frac{2\sqrt{29}}{29} & 0 & -1 & 0 & -\frac{2\sqrt{29}}{29} & \frac{2\sqrt{29}}{29} & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{5\sqrt{29}}{29} & -\frac{5\sqrt{29}}{29} & -\frac{5\sqrt{29}}{29} & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -\frac{2\sqrt{29}}{29} & \frac{2\sqrt{29}}{29} & -\frac{2\sqrt{29}}{29} & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{5\sqrt{29}}{29} & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\frac{2\sqrt{29}}{29} & 0 & -1 \\
0 & 0 & 0 & 0 & -\frac{5\sqrt{29}}{29} & 0 & 0 & \frac{5\sqrt{29}}{29} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{5\sqrt{29}}{29} & 0 \\
0 & 0 & 0 & 0 & -\frac{2\sqrt{29}}{29} & 0 & -1 & -\frac{2\sqrt{29}}{29} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{2\sqrt{29}}{29} & 1
\end{pmatrix}$$

$$\vec{b} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 & 0 & 3 & -2 & 0 & -\frac{3}{2} & 0 \end{pmatrix}^T$$

$$\vec{x} = \begin{pmatrix} A & E_x & E_y & T_{AB} & T_{AB} & T_{AI} & T_{BC} & T_{BI} & T_{CI} & T_{CG} & T_{CF} & T_{CD} & T_{DF} & T_{DE} & T_{EF} & T_{FG} & T_{GH} & T_{GI} & T_{HI} \end{pmatrix}^T$$

3 Condition Number of Matrix

Using the following command,

```
>> cond(A)

ans =

22.7176

>>
```

we can see that the condition number of A is 22.7176. This means that the output's error is going to be around 23 times the input's error. To account for this, the output values should have one fewer significant figure than input values. Because it is implied that values used in this problem are exact, I will ignore the condition number when displaying final results.

4 Solving of Matrix

4.1 Summary

Method	Type	Function Call	Is Result correct?	Time(μs)
Cramer's Rule	Custom	cramersRule(A,b)	Yes	389.9
Naive Gaussian Elimination	Custom	naiveGauss(A,b)	No	134.0
Gaussian Elimination w/ Partial Pivoting	Custom	gaussPartialPivot(A,b)	Yes	247.5
LU Method	Custom	luMethod(A,b)	Yes	224.0
A\b	Built-in	inverseWrapper(A,b)	Yes	74.3
A^(-1)*b	Built-in	inverseWrapper2(A,b)	Yes	77.8
inv(A)*b	Built-in	inverseWrapper3(A,b)	Yes	37.4
rref([A,b])	Built-in	rref([A,b])	Yes*	2891.3

*Correct when values rounded to nearest 10^-15.

4.2 Measuring Execution Time

To measure the execution time of each method, I wrote a script that ran my main script 10,000 times.

```
for z=1:10000
run('main');
disp(z)
end
```

I then ran this script with "Run and Time" enabled. I took the total amount of time spent executing each function, and divided it by 10,000 to get the average execution time.

4.3 The Correct Result

When solving this problem manually, we can see that

$$\vec{x} = \begin{pmatrix} A \\ E_x \\ E_y \\ T_{AB} \\ T_{AI} \\ T_{BC} \\ T_{BI} \\ T_{CI} \\ T_{CG} \\ T_{CF} \\ T_{CD} \\ T_{DF} \\ T_{DE} \\ T_{EF} \\ T_{FG} \\ T_{GH} \\ T_{GI} \\ T_{HI} \end{pmatrix} = \begin{pmatrix} 1.5000 \\ -3.5000 \\ 4.5000 \\ 3.7500 \\ -4.0389 \\ 3.7500 \\ 0 \\ 0.2693 \\ 1.4000 \\ -4.0389 \\ 7.7500 \\ 0 \\ 7.7500 \\ -12.1166 \\ -8.0777 \\ -2.1541 \\ -5.9237 \\ 0.8000 \end{pmatrix} kip$$

The positive values in A and E_y indicate that those forces are facing up, and the negative value of E_x indicates that it is pointing left. For every tension force, a positive value indicates tension in the member, and a negative value indicates compression.

Every method listed as producing a correct result on the table produced output equal to $\vec{x}$. The naive Gaussian elimination method produced an output vector containing only "NaN," so it did not produce a result. This is not surprising, as naive Gaussian elimination only works under very specific circumstances. Naive Gaussian elimination only works if each row's pivot value never equals zero. The more zero values there are in the matrix, the more likely a pivot value equal to zero will emerge. With a problem like this one, where no equation contains more than a quarter of the unknowns, making this method work manually is nearly impossible.

However, we can use the lu() method to help us find a matrix that can be solved with naive Gaussian elimination.

$$P^T LU = A$$
$$P^T LUx = b$$
$$LUx = P^{(T)^{-1}} b$$

$$\therefore$$
$$A' = LU$$
$$b' = P^{(T)^{-1}} b$$

```
[l,u,p]=lu(A);
>> naiveGauss(l*u,p'\b)

ans =

1.5000
-3.5000
4.5000
3.7500
-4.0389
3.7500
0
0.2693
1.4000
-4.0389
7.7500
0
7.7500
-12.1166
-8.0777
-2.1541
-5.9237
0.8000

latex(sym(l*u))

ans =
```

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0.3714 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -0.9285 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.3714 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0.9285 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -0.9285 & 0 & 0 & 0.9285 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.9285 & 0 \\ 0 & 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -0.3714 & 0 & -1 & -0.3714 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.3714 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.3714 & 1 & 0.3714 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -0.9285 & 0 & 0 & 0 & 0.9285 & -0.9285 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -0.3714 & 0 & -1 & 0 & -0.3714 & 0.3714 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & -0.9285 & 0 & 0.9285 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.9285 & -0.9285 & -0.9285 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.9285 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -5.5511e-17 & 0 & 0 & 0 & 0 & -0.3714 & 0.3714 & -0.3714 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -0.3714 & 0 & -1 \end{pmatrix}$$

```
>> latex(sym(p'\b))

ans =
```

$$\begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ -1.5000 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 3 \\ 0 \\ 0 \\ 0 \\ -2 \\ 3 \\ 0 \end{pmatrix}$$

This demonstrates that if the rows of the matrix are rearranged such that the largest absolute value of each column is placed on a diagonal cell, naive Gaussian elimination can solve this problem. This is done as part of the function in Gaussian elimination with partial pivoting, so there is little reason to work around the limitations of naive Gaussian elimination.

5 Conclusion

With this project, I've learned how to take a truss problem and produce a series of steps to solve it that apply to every truss problem. Because I wrote my program in such a general way, it would be possible to adapt the program to solve any 2D truss problem in existence. Once I generated a matrix containing all relevant equations, I learned that any of MATLAB's built-in matrix division methods are significantly faster than my custom functions. If I am certain that a problem has only one answer, I am going to use A\b to solve augmented matrices in the future, rather than rref(), which is the slowest of the row reduction functions that I tested.