Computational Methods Chapter 4
Solving Systems of Linear Equations

4.1 Background

1. Many engineering problems can be posed as solving systems of linear equations:
   1. Trusses (Newton's Laws)
   2. Circuits (Kirchoff's Law)
   3. Motion of objects with internal connections and linkages (ordinary differential equations)
   4. Stress/strain/deformation (partial differential equations)
   5. Fluid Motion (partial differential equations)
   6. etc…
2. Cramer's Rule – directly solve for 3x3 (but not bigger)

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

$$x_1 = \frac{\begin{vmatrix} b_1 & a_{12} & a_{13} \\ b_2 & a_{22} & a_{23} \\ b_3 & a_{32} & a_{33} \end{vmatrix}}{\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix}} \qquad x_2 = \frac{\begin{vmatrix} a_{11} & b_1 & a_{13} \\ a_{21} & b_2 & a_{23} \\ a_{31} & b_3 & a_{33} \end{vmatrix}}{\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix}} \qquad x_3 = \frac{\begin{vmatrix} a_{11} & a_{12} & b_1 \\ a_{21} & a_{22} & b_2 \\ a_{31} & a_{32} & b_3 \end{vmatrix}}{\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix}}$$

3. "Clever" substituting (2x2, 3x3… not bigger)
4. Direct and Iterative Methods

In general we are trying to solve:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$$

5. Upper Triangular Form
   1. $$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ 0 & a_{22} & a_{23} & a_{24} \\ 0 & 0 & a_{33} & a_{34} \\ 0 & 0 & 0 & a_{44} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$$
   2. Which can be solved by:

$$x_4 = \frac{b_4}{a_{44}}$$

3. $$x_3 = \frac{b_3 - a_{34} x_4}{a_{33}}$$

$$x_i = \frac{b_i - \sum_{j=i+1}^{j=n} a_{ij} x_j}{a_{ii}} \qquad i = n-1,\ n-2,\ \ldots,\ 1$$

4. This is called backsubstitution

6. Lower Triangular Form
    1. $$\begin{bmatrix} a_{11} & 0 & 0 & 0 \\ a_{21} & a_{22} & 0 & 0 \\ a_{31} & a_{32} & a_{33} & 0 \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$$
    2. Which can be solved by:

$$x_1 = \frac{b_1}{a_{11}}$$

3. $$x_2 = \frac{b_2 - a_{21} x_1}{a_{22}}$$

$$x_i = \frac{b_i - \sum_{j=1}^{j=i-1} a_{ij} x_j}{a_{ii}} \qquad i = 2,\ 3,\ \ldots,\ n$$

4. This is forward substitution.

7. Diagonal Form
    1. $$\begin{bmatrix} a_{11} & 0 & 0 & 0 \\ 0 & a_{22} & 0 & 0 \\ 0 & 0 & a_{33} & 0 \\ 0 & 0 & 0 & a_{44} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$$
    2. Which can be solved by:
    3. $x_i = \frac{b_i}{a_{ii}}$
8. As you might guess… the problems are very seldom like above… *BUT we can make them look this way!*

Developing an algorithm for **back substitution** (this assumes *a* in upper triangular form).

$$x_i = \frac{b_i - \sum_{j=i+1}^{j=n} a_{ij} x_j}{a_{ii}} \qquad i = n-1,\, n-2,\, \ldots,\, 1$$ HOW DO I CODE THIS!?

Starting index of *1*:

$x_n = b_n/a_{nn}$ (Do this part outside the loop)

```
For Loop –  Range = i = n-1 to 1 (stepping backward)
          sum = get_sum(a,x,i)          This is a function call to get_sum
          $x_i = ( b_i – sum)/a_{ii}$

function get_sum(a,x,i):
          sum = 0       Initialize the sum
          For Loop – Range  j = i+1 to j = n
                    sum = sum + $a_{ij}$ $x_j$
          return sum
```

Starting index of *0 (python)*:

$x_{n-1} = b_{n-1}/a_{n-1,n-1}$ (Do this part outside the loop)

```
For Loop –  Range = i = n-2 to 0 (stepping backward)
          sum = get_sum(a,x,i)          This is a function call to get_sum
          $x_i = ( b_i – sum)/a_{ii}$

function get_sum(a,x,i):
          sum = 0       Initialize the sum
          For Loop – Range  j = i+1 to j = n-1
                    sum = sum + $a_{ij}$ $x_j$
          return sum
```

Now with specific values here is what it would happen in code (not coded this way, but just to explain):

This assumes a starting index of 0, n = 4…

$x_3 = b_3/a_{3,3}$ *(Do this part outside the loop)*

For Loop – Range = i = 2 to 0 (stepping backward)

```
    i=2:
    sum = get_sum(a,x,2)          This is a function call to get_sum
    $x_2 = ( b_2 – sum)/a_{22}$

    function get_sum(a,x,2):
        sum = 0    Initialize the sum
        For Loop – Range  j = 3 to j = 3

        j = 3:
            sum = 0 + $a_{23} x_3$
        return sum

    i=1:
    sum = get_sum(a,x,1)          This is a function call to get_sum
    $x_1 = ( b_1 – sum)/a_{11}$

    function get_sum(a,x,1):
        sum = 0    Initialize the sum
        For Loop – Range  j = 2 to j = 3

        j = 2:
            sum = 0 + $a_{12} x_2$
        j = 3:
            sum = sum + $a_{13} x_3$

        return sum

    i=0:
    sum = get_sum(a,x,0)          This is a function call to get_sum
    $x_0 = ( b_0 – sum)/a_{00}$

    function get_sum(a,x,0):
        sum = 0    Initialize the sum
        For Loop – Range  j = 1 to j = 3

        j = 1:
            sum = 0 + $a_{01} x_1$
        j = 2:
            sum = sum + $a_{02} x_2$
        j = 3:
            sum = sum + $a_{03} x_3$
        return sum
```

Gauss Elimination Example (3x3 in augmented form):
Gauss = Forward elim. + Normalizing Pivots as we go + back substitution

$$x_1-3x_2+x_3=4$$
$$2x_1-8x_2+8x_3=-2$$
$$-6x_1+3x_2-15x_3=9$$

$$\begin{bmatrix}1 & -3 & 1 & 4\\ 2 & -8 & 8 & -2\\ -6 & 3 & -15 & 9\end{bmatrix} \quad \begin{matrix}-2R_1+R_2\rightarrow R_2'\\ 6R_1+R3\rightarrow R_3'\end{matrix} \quad \begin{matrix}-2\,[1 \;\; -3 \;\; 1 \;\; 4]\\ [-2 \;\; 6 \;\; -2 \;\; -8]\\ [2 \;\; -8 \;\; 8 \;\; -2]\\ [0 \;\; -2 \;\; 6 \;\; -10]=R_2'\end{matrix} \quad \begin{matrix}6\,[1 \;\; -3 \;\; 1 \;\; 4]\\ [6 \;\; -18 \;\; 6 \;\; 24]\\ [-6 \;\; 3 \;\; -15 \;\; 9]\\ [0 \;\; -15 \;\; -9 \;\; 33]=R_3'\end{matrix}$$

$$\begin{bmatrix}1 & -3 & 1 & 4\\ 0 & -2 & 6 & -10\\ 0 & -15 & -9 & 33\end{bmatrix} \quad \frac{R_2'}{-2}\rightarrow R_2'' \quad [0 \;\; -2 \;\; 6 \;\; -10]/-2=[0 \;\; 1 \;\; -3 \;\; 5]=R_2''$$

$$\begin{bmatrix}1 & -3 & 1 & 4\\ 0 & 1 & -3 & 5\\ 0 & -15 & -9 & 33\end{bmatrix} \quad 15R_2''+R_3'\rightarrow R_3'' \quad \begin{matrix}15\,[0 \;\; 1 \;\; -3 \;\; 5]\\ [0 \;\; 15 \;\; -45 \;\; 75]\\ [0 \;\; -15 \;\; -9 \;\; 33]\\ [0 \;\; 0 \;\; -54 \;\; 108]=R_3''\end{matrix}$$

$$\begin{bmatrix}1 & -3 & 1 & 4\\ 0 & 1 & -3 & 5\\ 0 & 0 & -54 & 108\end{bmatrix} \quad \frac{R_3''}{-54}\rightarrow R_3''' \quad [0 \;\; 0 \;\; -54 \;\; 108]/-54=[0 \;\; 0 \;\; 1 \;\; -2]$$

$$\begin{bmatrix}1 & -3 & 1 & 4\\ 0 & 1 & -3 & 5\\ 0 & 0 & 1 & -2\end{bmatrix} \quad \begin{matrix}(1)x_3=-2\rightarrow x_3=-2\\ (1)x_2+(-3)(-2)=5\rightarrow x_2=-1\\ (1)x_1+(-3)(-1)+(1)(-2)=4\rightarrow x_1=3\end{matrix}$$

<u>Generalize</u>
We start with:

$$\begin{bmatrix}a_{11} & a_{12} & a_{13} & a_{14}\\ a_{21} & a_{22} & a_{23} & a_{24}\\ a_{31} & a_{32} & a_{33} & a_{34}\\ a_{41} & a_{42} & a_{43} & a_{44}\end{bmatrix}\begin{bmatrix}x_1\\ x_2\\ x_3\\ x_4\end{bmatrix}=\begin{bmatrix}b_1\\ b_2\\ b_3\\ b_4\end{bmatrix}$$ and we are trying to make it look like:

$$\begin{bmatrix}a_{11} & a_{12} & a_{13} & a_{14}\\ 0 & a_{22}' & a_{23}' & a_{24}'\\ 0 & 0 & a_{33}'' & a_{34}''\\ 0 & 0 & 0 & a_{44}'''\end{bmatrix}\begin{bmatrix}x_1\\ x_2\\ x_3\\ x_4\end{bmatrix}=\begin{bmatrix}b_1\\ b_2\\ b_3\\ b_4\end{bmatrix}$$ where a_{22}' has been modified from it's value.

The first equation is the *pivot* equation and a_{11} is the pivot coefficient. Here is how we can use it to operate on the other rows/equations to get the matrix in the upper triangular form. We also define $m_{21} = a_{21}/a_{11}$.

$$\begin{aligned}
&a_{21}x_1+a_{22}x_2+a_{23}x_3+a_{24}x_4=b_2\\
-m_{21}&(a_{11}x_1+a_{12}x_2+a_{13}x_3+a_{14}x_4=b_1)\\
&0+(a_{22}-m_{21}a_{12})x_2+(a_{23}-m_{21}a_{13})x_3+(a_{24}-m_{21}a_{14})x_4=b_2-m_{21}b_1\\
&a_{22}'x_2+a_{23}'x_3+a_{24}'x_4=b_2'
\end{aligned}$$

The same operation is performed for each row (eliminating the first term in each equation/row.

Another example:

$$\begin{bmatrix} 0.143 & 0.357 & 2.01 \\ -1.31 & 0.911 & 1.99 \\ 11.2 & -4.30 & -0.605 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}=\begin{bmatrix} -5.173 \\ -5.458 \\ 4.415 \end{bmatrix}$$

Augmented form
It's helpful when actually implementing Gauss elimination to represent the problem as a single matrix in augmented form:

$$\begin{bmatrix} 0.143 & 0.357 & 2.01 & -5.173 \\ -1.31 & 0.911 & 1.99 & -5.458 \\ 11.2 & -4.30 & -0.605 & 4.415 \end{bmatrix}$$

Normalizing Pivots
One process that “helps” with this solution is to *normalize* the row that you are working on.. this means dividing the entire row by the *pivot* element. So for the first row as the pivot row:

$$\begin{bmatrix} 1.00 & 2.50 & 14.1 & -36.2 \\ -1.31 & 0.911 & 1.99 & -5.46 \\ 11.2 & -4.30 & -0.605 & 4.42 \end{bmatrix}$$

Note we have also rounded to three digits… we will keep this convention throughout this example (to make a point!).

Now you have to try to do the forward elimination on *row 2:*

$$\begin{bmatrix} 1.00 & 2.50 & 14.1 & -36.2 \\ 0.00 & 4.19 & 20.5 & -52.9 \\ 11.2 & -4.30 & -0.605 & 4.42 \end{bmatrix}$$

Now do forward elimination on *row 3*

$$\begin{bmatrix} 1.00 & 2.50 & 14.1 & -36.2 \\ 0.00 & 4.19 & 20.5 & -52.9 \\ 0.00 & -32.3 & -159 & 409 \end{bmatrix}$$

Now normalize the second row

$$\begin{bmatrix} 1.00 & 2.50 & 14.1 & -36.2 \\ 0.00 & 4.19 & 20.5 & -52.9 \\ 0.00 & -32.3 & -159 & 409 \end{bmatrix}$$

Now do elimination on the third row

$$\begin{bmatrix} 1.00 & 2.50 & 14.1 & -36.2 \\ 0.00 & 1.00 & 4.89 & -12.6 \\ 0.00 & 0.00 & -1.00 & 2.00 \end{bmatrix}$$

Now do normalize the third row

$$\begin{bmatrix} 1.00 & 2.50 & 14.1 & -36.2 \\ 0.00 & 1.00 & 4.89 & -12.6 \\ 0.00 & 0.00 & 1.00 & -2.00 \end{bmatrix}$$

Show that $x_3 = -2.00$, $x_2 = -2.82$, $x_1 = -0.95$

Also plug these values back into the original equations to see if they are correct??

Correct answers are $x_3 = -3$, $x_2 = 2$, $x_1 = 1$ – **Do you get the right answers!?**

Rounding error was an issue! Of course it will be less of an issue on the computer where we can keep the numbers in double precision.

Also… there's another problem: There are large (2 orders of magnitude differences in the first column):

$$\begin{bmatrix} 0.143 & 0.357 & 2.01 & -5.173 \\ -1.31 & 0.911 & 1.99 & -5.458 \\ 11.2 & -4.30 & -0.605 & 4.415 \end{bmatrix}$$ Note – 0.143, -1.31, 11.2…

$$\begin{bmatrix} 1.00 & 2.50 & 14.1 & -36.2 \\ 0.00 & 4.19 & 20.5 & -52.9 \\ 11.2 & -4.30 & -0.605 & 4.42 \end{bmatrix} \text{ Goes to: } \begin{bmatrix} 1.00 & 2.50 & 14.1 & -36.2 \\ 0.00 & 4.19 & 20.5 & -52.9 \\ 0.00 & -32.3 & -159 & 409 \end{bmatrix}$$

The value -0.605 went to -159… a loss of three decimal places! We will fix this later with partial pivoting.

Now how to do the algorithm for Gauss?

This assumes a starting index of 0!

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & b_0 \\ a_{10} & a_{11} & a_{12} & b_1 \\ a_{20} & a_{21} & a_{22} & b_2 \end{bmatrix}$$ *This is what we will write our algorithm for*

Assume we have matrix *a* (which is 3 x 3) in *a.* Assume we have an augmented form with the RHS vector augmented onto *a* called *ab*. Matrix *ab* will be 3 x 4.

1. **Augment:** We need a function called *augment* to make *ab (the augmented matrix)*

   Note: we have already done this… python numpy made this easy.

```
augment(a,b)
      initialize a new matrix ab
      for i= 0 to 2
            for j=0 to 3
                  if j < 3
                        $ab_{ij} = a_{ij}$
                  else
                        $ab_{ij} = b_i$
      return ab
```

Generalize it!

```
augment(a,b)
      for i= 0 to n-1
            for j=0 to n
                  if j < n
                        $ab_{ij} = a_{ij}$
                  else
                        $ab_{ij} = b_i$
      return ab
```

2. **Normalize:** We need a function *normalize* that divides the entire row by the pivot element – note for Gauss elimination this only needs to proceed from the index to the right within this row.

   We will send in the entire ab matrix and the index of the row we need to normalize. For example purposes we will send in an index of *1.* This means we will be "normalizing" by the value ab_{11} in row 2 (*which has an index of 1!*):

```
normalize(ab,1)
      ab_new = ab   #just get a new version of the original
      norm = $a_{1,1}$
      for j = 1 to 3
            $ab\_new_{1,j} = ab_{1,j}$ / norm
      return ab_new
```

   Here is exactly what will happen above:

```
      j = 1:   $ab\_new_{1,1} = ab_{1,1}$ / norm  (this should give us 1.00000...)
      j = 2:   $ab\_new_{1,2} = ab_{1,2}$ /  norm
      j = 3:   $ab\_new_{1,3} = ab_{1,3}$ /  norm
```

Generalize it!

```
normalize(ab,i)
```

```
ab_new = ab   #just get a new version of the original
norm = $a_{i,i}$
for j = i to n
      $ab\_new_{i,j}$ = $ab_{i,j}$ / norm
return ab_new
```

3. **Forward Elimination:** Let's say we are pivoting on $ab_{0,0}$ that means we are trying to eliminate $ab_{1,0}$ and $ab_{2,0}$.

$$\begin{bmatrix} ab_{00} & ab_{01} & ab_{02} & ab_{03} \\ ab_{10} & ab_{11} & ab_{12} & ab_{13} \\ ab_{20} & ab_{21} & ab_{22} & ab_{23} \end{bmatrix}$$

Assuming $ab_{0,0}$ is normalized we need to do the following:

$$-ab_{10}\left[1.00 \quad ab_{01} \quad ab_{02} \quad ab_{03}\right]+\left[ab_{10} \quad ab_{11} \quad ab_{12} \quad ab_{13}\right]=$$
$$\left[0.00 \quad ab_{11}-ab_{10}\left(ab_{01}\right) \quad ab_{12}-ab_{10}\left(ab_{02}\right) \quad ab_{13}-ab_{10}\left(ab_{03}\right)\right]$$

Here's the loop to do what is above (note this is to eliminate $\underline{ab_{10}}$)

```
for j = 1 to 3
      $ab_{1,j}$ = $ab_{1,j}$ - $ab_{1,0}$ ($ab_{0,j}$)

      j = 1: $ab_{1,1}$ = $ab_{1,1}$ − $ab_{1,0}$ ($ab_{0,1}$)
      j = 2: $ab_{1,2}$ = $ab_{1,2}$ − $ab_{1,0}$ ($ab_{0,2}$)
      j = 3: $ab_{1,3}$ = $ab_{1,3}$ − $ab_{1,0}$ ($ab_{0,3}$)
```

Generalize it! – if the pivot row is *i*, the pivot element is $\underline{ab_{i,i}}$ *and the element being eliminated is* $\underline{ab_{i+1,i}}$

```
for j = i+1 to n
      $ab_{i+1,j}$ = $ab_{i+1,j}$ − $ab_{i+1,i}$ ($ab_{i,j}$) Compare to example above
```

Note this only eliminates… to do this even more generally… have to work through all rows below i.

Also: In code we should go ahead and perform the operations above on element just below i,i…

The process above needs to be applied to every row below the pivot row, $ab_{0,0}$ *is the pivot*

```
for k = 1 to 2
      for j = 1 to 3
            $ab_{k,j} = ab_{k,j} - ab_{k,0}\ (ab_{0,j})$
```

k = 1:

```
      j = 1: $ab_{1,1} = ab_{1,1} - ab_{1,0}\ (ab_{0,1})$
      j = 2: $ab_{1,2} = ab_{1,2} - ab_{1,0}\ (ab_{0,2})$
      j = 3: $ab_{1,3} = ab_{1,3} - ab_{1,0}\ (ab_{0,3})$
```

k = 2:

```
      j = 1: $ab_{2,1} = ab_{2,1} - ab_{2,0}\ (ab_{0,1})$
      j = 2: $ab_{2,2} = ab_{2,2} - ab_{2,0}\ (ab_{0,2})$
      j = 3: $ab_{2,3} = ab_{2,3} - ab_{2,0}\ (ab_{0,3})$
```

Generalize it!

```
for k = i+1 to n-1
      for j = i+1 to n
            $ab_{k,j} = ab_{k,j} - ab_{k,i}\ (ab_{i,j})$
```

We need to normalize after each pivot row and we need to use all rows as pivot rows except the last row… so

```
ab = augment(a,b)
for i = 0 to n-2
      normalize(ab,i)
      for k = i+1 to n-1
            for j = i+1 to n
                  $ab_{k,j} = ab_{k,j} - ab_{k,i}\ (ab_{i,j})$
```

Gauss Elimination with Partial Pivoting

Gauss Elimination works best when *the pivot with the largest possible absolute value is chosen.*

So the *first* pass we would identify the largest value in the first column, going back to the augmented form for a previous example:

$$\begin{bmatrix} 0.143 & 0.357 & 2.01 & -5.173 \\ -1.31 & 0.911 & 1.99 & -5.458 \\ 11.2 & -4.30 & -0.605 & 4.415 \end{bmatrix} \rightarrow \begin{bmatrix} 11.2 & -4.30 & -0.605 & 4.415 \\ -1.31 & 0.911 & 1.99 & -5.458 \\ 0.143 & 0.357 & 2.01 & -5.173 \end{bmatrix}$$

Seems simple enough… we just swap rows. How would an algorithm need to work to do this?

(Assuming an starting index of zero)

$$\begin{bmatrix} ab_{00} & ab_{01} & ab_{02} & ab_{03} \\ ab_{10} & ab_{11} & ab_{12} & ab_{13} \\ ab_{20} & ab_{21} & ab_{22} & ab_{23} \end{bmatrix}$$

```
index = 0 # We are going to place the best row first – keep track in index
initialize order vector – order[nx1]          order = [0,1,2]
max = abs($a_{0,0}$)
```

```
for i = 1 to 2 #examine all rows
      if abs($a_{i,0}$) > max
            index = i
            max = abs($a_{i,0}$)
#now do a row swap
# index = 2
tmp = order[0]   #temporarily store original val. Of what was in row 0
  # tmp = 0
order[0]= order[index]   #put new row 0 in correct place in order vector
  # order[0] = order[2] = 2
order[index] = tmp   #complete row swap
  # order[2] = 0
  # order = [2,1,0]
```

Here are the actual values for the above matrix if we carried this out:

```
index = 0
order = [0, 1, 2]
max = abs($a_{0,0}$) (0.143)
for i = 1 to 2 #examine all rows
i = 1:
      if abs($a_{1,0}$) (1.31) > max (0.143)
            index = i (1)
            max = abs($a_{1,0}$) (1.31)
i = 2:
      if abs($a_{2,0}$) (11.2) > max (1.31)
            index = i (2)
            max = abs($a_{2,0}$) (11.2)

tmp = order[0] (0)
order[0]= index (2) #here's how we actually swap rows
order[2]= tmp (0)

# order vector will now look like: [2,1,0]
# so for the moment following would be true.…

$a_{order[0],0}$ = 11.2 while $a_{0,0}$ is still 0.143
```

After this row exchange the matrix will look like:

$$\begin{bmatrix} ab_{order[0],0} & ab_{order[0],1} & ab_{order[0],2} & ab_{order[0],3} \\ ab_{order[1],0} & ab_{order[1],1} & ab_{order[1],2} & ab_{order[1],3} \\ ab_{order[2],0} & ab_{order[2],1} & ab_{order[2],2} & ab_{order[2],3} \end{bmatrix} \leftarrow \text{This is how we will deal with matrix in code}$$

Which acheives :

$$\begin{bmatrix} 0.143 & 0.357 & 2.01 & -5.173 \\ -1.31 & 0.911 & 1.99 & -5.458 \\ 11.2 & -4.30 & -0.605 & 4.415 \end{bmatrix} \rightarrow \begin{bmatrix} 11.2 & -4.30 & -0.605 & 4.415 \\ -1.31 & 0.911 & 1.99 & -5.458 \\ 0.143 & 0.357 & 2.01 & -5.173 \end{bmatrix}$$

Note this process can process each time you move to a new pivot row… but you only need to proceed below the pivot.

Gauss-Jordan Elimination

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b_1' \\ b_2' \\ b_3' \\ b_4' \end{bmatrix}$$

So the solution will be contained in the ***b'*** vector.

In terms of an algorithm, this is very similar to Gauss elimination… we should include pivoting here, but note pivoting now would include all elements *above* and *below* the pivot. And now elimination is carried out *above* and *below* the pivot also.

First augment the matrix:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & b_1 \\ a_{21} & a_{22} & a_{23} & a_{24} & b_2 \\ a_{31} & a_{32} & a_{33} & a_{34} & b_3 \\ a_{41} & a_{42} & a_{43} & a_{44} & b_4 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 0 & 0 & 0 & b_1' \\ 0 & 1 & 0 & 0 & b_2' \\ 0 & 0 & 1 & 0 & b_3' \\ 0 & 0 & 0 & 1 & b_4' \end{bmatrix}$$

As you might guess… programming this is not too different than Gauss Elimination with Pivoting… the ranges of all the for loops change and there is additional logic required to not do operations on the pivot row itself.

LU Decomposition
It is possible to find $[a]^{-1}$ the inverse of [a]. But it is a time consuming calculation with many operations required. *More operations means more round-off error*. If the inverse will be used many times… it is sometimes worth it, but even in these cases.. there is an alternative called LU Decomposition. Here is how it works:

$[L]$ = Lower Triangular
$[U]$ = Upper Triangular

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 & 0 & 0 \\ L_{21} & L_{22} & 0 & 0 \\ L_{31} & L_{32} & L_{33} & 0 \\ L_{41} & L_{42} & L_{43} & L_{44} \end{bmatrix} \begin{bmatrix} 1 & U_{12} & U_{13} & U_{14} \\ 0 & 1 & U_{23} & U_{24} \\ 0 & 0 & 1 & U_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This would be helpful for several reasons.

Define $[U][x]=[y]$
Equation 1

so that $[L][y]=[b]$
Equation 2

So the process is to:
1. Solve Eq. 2 for [y] (using forward elimination)
2. Solve Eq. 1 for [x] (using back substitution)

To be clear:

How do you get [L] and [U]? Please see the algorithm development for LU decomposition using *Crout's Method.*

Of course we would only do this if we were going to solve this problem with the same [a] many times… this does come up in real problems… an example is below… if the reaction forces change only the RHS vector changes...

Matrix Inversion

There are two ways to get $[a]^{-1}$ LU decomposition and Gauss-Jordan elimination. Please read your book for an example of how to use LU Decomposition. Here is how Gauss-Jordan would work:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & 1 & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & a_{24} & 0 & 1 & 0 & 0 \\ a_{31} & a_{32} & a_{33} & a_{34} & 0 & 0 & 1 & 0 \\ a_{41} & a_{42} & a_{43} & a_{44} & 0 & 0 & 0 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 0 & 0 & 0 & a_{11}' & a_{12}' & a_{13}' & a_{14}' \\ 0 & 1 & 0 & 0 & a_{21}' & a_{22}' & a_{23}' & a_{24}' \\ 0 & 0 & 1 & 0 & a_{31}' & a_{32}' & a_{33}' & a_{34}' \\ 0 & 0 & 0 & 1 & a_{41}' & a_{42}' & a_{43}' & a_{44}' \end{bmatrix}$$

So the a' matrix is the inverse. To do this one just needs to do forward and backward elimination and normalize the pivots. Once this is done you are left with an identity matrix where the a matrix was originally.

ITERATIVE METHODS

In these methods you write the unknown for each equation (row) in terms of all the other unknowns of the problem. For example:

$$\begin{array}{lcl} a_{11}x_1+a_{12}x_2+a_{13}x_3+a_{14}x_4=b_1 & \rightarrow & x_1=[b_1-(a_{12}x_2+a_{13}x_3+a_{14}x_4)]/a_{11} \\ a_{21}x_1+a_{22}x_2+a_{23}x_3+a_{24}x_4=b_2 & & x_2=[b_2-(a_{21}x_2+a_{23}x_3+a_{24}x_4)]/a_{22} \end{array}$$

To get the process started you *assume* values for $x_1 - x_n$. The general form of the *x* equations above would be:

$$x_i=\frac{1}{a_{ii}}\left[b_i-\sum_{j=1,\,j\neq i}^{j=n} a_{ij}x_j\right] \qquad i=1,2,\ldots,n$$

Equation 3

This technique when it works is very fast… all we really have to do is run equation 4 repeatedly. It is many fewer operations than the non-iterative techniques. It is almost like doing only the back substitution step in Gauss elimination. There is an issue… it will not always *converge*. It will converge when:

$$|a_{ii}|>\sum_{j=1,\,j\neq i}^{j=n}|a_{ij}|$$

This has to be true for each row… so one would typically reorder a matrix to make this *as true as possible*! In other words make it so that a_{ii} for each row is optimized to be as large as possible. This is called a *diagonally dominant* matrix.

Jacobi method

Let $x_i^{(2)}$ be the value of x_i for the second iteration. It will be based on all of the x values from the first iteration $x_i^{(1)}$. So using Eq. 4 above…

$$x_i^{(2)}=\frac{1}{a_{ii}}\left[b_i-\sum_{j=1,\,j\neq i}^{j=n} a_{ij}x_j^{(1)}\right] \qquad i=1,2,\ldots,n$$

In general

$$x_i^{(k+1)}=\frac{1}{a_{ii}}\left[b_i-\sum_{j=1,\,j\neq i}^{j=n} a_{ij}x_j^{(k)}\right] \qquad i=1,2,\ldots,n$$

The *x* values are iterated until the *approximate relative fractional error* is less than a stopping criterion, ϵ_s

$$\left|\frac{x_i^{(k+1)}-x_i^{(k)}}{x_i^{(k+1)}}\right|<\epsilon_s$$

This must be true for all rows.

Note the updates for x-values all occur at the end of each iteration. Even though you actually have new values theoretically have new values $x_j^{(k+1)}$ for values of *j* less than *i*.

Gauss-Siedel

The only difference between Jacobi and Gauss-Siedel (GS) is that GS uses the best possible values for the x values… that means using the $x_j^{(k+1)}$ for values of *j* less than *i*. This converges faster than Jacobi… so this is very commonly used for many engineering problems. So for example:

$$x_1^{(k+1)}=\frac{1}{a_{11}}\left[b_1-\sum_{j=1,j\neq i}^{j=n} a_{1j}x_j^{(k)}\right]$$

$$x_i^{(k+1)}=\frac{1}{a_{ii}}\left[b_i-\left(\sum_{j=1}^{j=i-1} a_{ij}x_j^{(k+1)}+\sum_{j=i+1}^{j=n} a_{ij}x_j^{(k)}\right)\right] \qquad i=1,2,\ldots,n-1$$

$$x_n^{(k+1)}=\frac{1}{a_{nn}}\left[b_1-\sum_{j=1}^{j=n-1} a_{nj}x_j^{(k+1)}\right]$$

The *x* values are iterated until the *approximate relative fractional error* is less than a stopping criterion, ϵ_s

$$\left|\frac{x_i^{(k+1)}-x_i^{(k)}}{x_i^{(k+1)}}\right|<\epsilon_s$$

This must be true for all rows.

<u>Tridiagonal Systems</u>

Sometimes (heat transfer, stress-strain, and other problems) the matrix to be solved is tridiagonal.

$$\begin{bmatrix} A_{11} & A_{12} & 0 & 0 & 0 \\ A_{21} & A_{22} & A_{23} & 0 & 0 \\ 0 & A_{32} & A_{33} & A_{34} & 0 \\ 0 & 0 & A_{43} & A_{44} & A_{45} \\ 0 & 0 & 0 & A_{54} & A_{55} \end{bmatrix}$$

Often these are more larger than 100x100… <u>*so there are a lot of zeros*</u>. We should not store those values or waste time using them in calculations. So there are algorithms to solve these problems. One such algorithm is the *Thomas algorithm.*

$$d_i=A_{ii}$$
$$a_i=A_{i,i+1}$$
$$b_i=A_{i-1,i}$$

d= diagonal, a = above, b = below. The *[d]* vector has a length of n, while the *[a]* and *[b]* vectors are each of length *n-1.*

$$\begin{bmatrix} d_1 & a_1 & 0 & 0 & 0 \\ b_2 & d_2 & a_2 & 0 & 0 \\ 0 & b_3 & d_3 & a_3 & 0 \\ 0 & 0 & b_4 & d_4 & a_4 \\ 0 & 0 & 0 & b_5 & d_5 \end{bmatrix}$$

As the book shows:

$$\begin{bmatrix} 1 & a'_1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & a'_2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & a'_3 & 0 & 0 & 0 & 0 \\ & & \cdots & \cdots & \cdots & & & \\ & & & \cdots & \cdots & \cdots & & \\ 0 & 0 & 0 & 0 & 0 & 1 & a'_{n-2} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & a'_{n-1} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \cdots \\ \cdots \\ x_{n-2} \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} B'_1 \\ B''_2 \\ B''_3 \\ \cdots \\ \cdots \\ B''_{n-2} \\ B''_{n-1} \\ B''_n \end{bmatrix}$$

Basically you normalize each row as you proceed (starting at the second row) and then eliminate the below diagonal element, b_{i+1} just below the d_i. Then once the matrix is in the form above… use back substitution to get [x]. Note both the forward elimination and back substitution steps do not need to span the entire rows and columns… instead they are very quick calculations that only use the *b, d,* and *a* values.

ERROR, RESIDUAL, NORMS, CONDITION NUMBER

Error and Residual

For [a][x] = [b], if the true solution is [x_{TS}], then the error in a numerical solution ([x_{NS}]) is

$$[e]=[x_{TS}]-[x_{NS}]$$

Of course we don't know x_{TS}, so… we can't directly calculate *[e]*, but an estimate can be calculated.

Another way of assessing a solution is the *residual, [r]*

$$[r]=[b]-[b_{NS}]=[a][x_{TS}]-[a][x_{NS}]$$

The residual measures the effect of the error in *[x_{NS}]* on the right-hand-side vector *[b]*.

Norms and Condition Numbers

It's easy to understand the *scale* of a scalar number using absolute values. The *scale* of a matrix is harder to assess and there are many ways to describe this. A norm is a way to do this. A norm must satisfy the following:

1. The norm is designated: $\|[a]\|$ and is positive and only equal to zero, if all the elements of the matrix are zero.
2. If you scale a matrix up by multiplying it by a scalar number, α, the norm should also scale up by α: $\|\alpha[a]\|=|\alpha|\,\|[a]\|$.
3. $\|[a][x]\|\leq\|[a]\|\,\|[x]\|$
4. $\|[a+b]\|\leq\|[a]\|+\|[b]\|$ triangle inequality

All norms should follow the rules above.

Vector Norms

For a vector *[v]*.

Infinity Norm
Equal to the largest magnitude component of *[v]*.

$$\| v \|_\infty = max|v_i| \qquad i=1,2,3,\ldots n$$

1-norm
The sum of the absolute values of the vector elements.

$$\| v \|_1 = \sum_{i=1}^{n} |v_i|$$

Euclidean 2-Norm
Square root of the sum of the squares of the vector elements

$$\| v \|_2 = \left(\sum_{i=1}^{n} v_i^2 \right)^{1/2}$$

Example vector: $\begin{bmatrix} 0.2 & -5 & 3 & 0.4 & 0 \end{bmatrix}$

$$\| v \|_\infty = 5$$

$$\| v \|_1 = \sum_{i=1}^{n} |v_i| = 0.2+5+3+0.4+0=8.6$$

$$\| v \|_2 = \left(\sum_{i=1}^{n} v_i^2 \right)^{1/2} = (0.2^2+5^2+3^2+0.4^2+0^2)^{1/2} = (34.2)^{1/2} = 5.8$$

Matrix Norms

For a matrix *[a]*.

Infinity Norm
Equal to the largest value of summing up the magnitudes of each row of *[a]*.

$$\| a \|_\infty = max \sum_{j=1}^{n} |a_{ij}| \qquad i=1,2,3,\ldots n$$

1-norm
Equal to the largest value of summing up the magnitudes of each column of *[a]*.

$$\| a \|_1 = max \left[\sum_{i=1}^{n} |a_{ij}| \right] \qquad j=1,2,\ldots n$$

2-norm (spectral norm)

$$\| a \|_2 = max \left(\frac{\| a v \|}{\| v \|} \right)$$

Note [v] is the vector corresponding to the eigenvalue of the matrix, λ. Note there are muliple eigenvalues. We will talk more about this in the next chapter.

Euclidean Norm
Square root of the sum of the squares of the matrix elements (note does not need to be square…) matrix that is *m x n.*

$$\| a \|_{Euclidean} = \left(\sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij}^{2} \right)^{1/2}$$

Example Matrix:

$$[a] = \begin{bmatrix} 0.2 & -5 & 3 & 0.4 & 0 \\ -0.5 & 1 & 7 & -2 & 0.3 \\ 0.6 & 2 & -4 & 3 & 0.1 \\ 3 & 0.8 & 2 & -0.4 & 3 \\ 0.5 & 3 & 2 & 0.4 & 1 \end{bmatrix}$$

$$\| a \|_{\infty} = max \sum_{j=1}^{n} |a_{ij}| \qquad i = 0.5+1+7+2+0.3 = 10.8$$

$$\| a \|_{1} = max \left[\sum_{i=1}^{n} |a_{ij}| \right] = 3+7+4+2+2 = 18$$

$$\| a \|_{Euclidean} = \left(\sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij}^{2} \right)^{1/2} = 0.2^2+5^2+3^2+0.4^2+.5^2+1^2+7^2+2^2+0.3^2+0.6^2+2^2+4^2+3^2$$
$$+0.1^2+3^2+0.8^2+2^2+0.4^2+3^2+0.5^2+3^2+2^2+0.4^2+1^2$$

<u>Relationship of norms to error and residuals (be patient… big conclusion at the end!!)</u>

$$[r] = [b] - [b_{NS}] = [a][x_{TS}] - [a][x_{NS}] = [a]([x_{TS}] - [x_{NS}]) = [a][e]$$

Which means if [a] is invertible

$$[e] = [a]^{-1} [r]$$

Recalling the property of norms: $\| [a][x] \| \le \| [a] \| \, \| [x] \|$

$$\| [e] \| = \| [a]^{-1} [r] \| \le \| [a]^{-1} \| \, \| [r] \|$$

Equation 4

Also

$$\| [r] \| = \| [a] \, [e] \| \le \| [a] \| \, \| [e] \| \quad \text{or} \quad \frac{\| [r] \|}{\| [a] \|} \le \| [e] \|$$

Equation 5

If Eqs. 5-6 are combined:

$$\frac{\| [r] \|}{\| [a] \|} \le \| [a]^{-1} \| \, \| [r] \|$$

Equation 6

Now define the relative error as $\frac{\|[e]\|}{\|[x_{TS}]\|}$ and the relative residual as $\frac{\|[r]\|}{\|[b]\|}$

Note the error, [e], cannot actually be calculated, *but the residual can be calculated*

$[r]=[a][x_{TS}]-[a][x_{NS}]=[b]-[a][x_{NS}]$

Using the value of [r], one can put bounds on the relative error as follows, from Eqs. 6-7

$$\frac{1}{\|[x_{TS}]\|}\frac{\|[b]\|}{\|[b]\|}\frac{\|[r]\|}{\|[a]\|}\le\frac{\|[e]\|}{\|[x_{TS}]\|}\le\frac{\|[a]^{-1}\|\,\|[r]\|}{\|[x_{TS}]\|}\frac{\|[b]\|}{\|[b]\|}$$

Equation 7

Rearrange to get:

$$\frac{1}{\|[a]\|}\frac{\|[b]\|}{\|[x_{TS}]\|}\frac{\|[r]\|}{\|[b]\|}\le\frac{\|[e]\|}{\|[x_{TS}]\|}\le\|[a]^{-1}\|\frac{\|[b]\|}{\|[x_{TS}]\|}\frac{\|[r]\|}{\|[b]\|}$$

Equation 8

Again use the fact that: $\|[a][x]\|\le\|[a]\|\,\|[x]\|$ Which means that

$\|[a][x_{TS}]\|=\|[b]\|\le\|[a]\|\,\|[x_{TS}]\|$ or that $\frac{\|[b]\|}{\|[a]\|\,\|[x_{TS}]\|}\le1$

Also

$\|[x_{TS}]\|=\|[a]^{-1}[b]\|\le\|[a]^{-1}\|\,\|[b]\|$ or that $\frac{\|[a]^{-1}\|\|[b]\|}{\|[x_{TS}]\|}\ge1$

The conclusion from above is that:

$\frac{\|[b]\|}{\|[x_{TS}]\|}\le\|[a]\|$ and that $\frac{\|[b]\|}{\|[x_{TS}]\|}\ge\frac{1}{\|[a]^{-1}\|}$

So if we were to substitute $\|[a]\|$ for in the RHS of Eq. 9… because this would only make the inequality <u>*more true.*</u>

And if we were to substitute $\frac{1}{\|[a]^{-1}\|}$ for $\frac{\|[b]\|}{\|[x_{TS}]\|}$ in the LHS of Eq. 9… this would only make the inequality <u>*more true.*</u>

So finally: $\frac{1}{\|[a]\|\|[a]^{-1}\|}\frac{\|[r]\|}{\|[b]\|}\le\frac{\|[e]\|}{\|[x_{TS}]\|}\le\|[a]^{-1}\|\|[a]\|\frac{\|[r]\|}{\|[b]\|}$ <u>***THIS IS THE BIG CONCLUSION!***</u>

Why is this important… LHS = lower bound on relative error and the RHS = upper bound on error!

So if you multiply $\frac{1}{\|[a]\|\|[a]^{-1}\|}$ by the relative error, $\frac{\|[r]\|}{\|[b]\|}$, you get the lowest possible relative error, $\frac{\|[e]\|}{\|[x_{TS}]\|}$.

If you mutiply $\|[a]\|\|[a]^{-1}\|$ by the relative error, $\frac{\|[r]\|}{\|[b]\|}$, you get the largest possible relative error, $\frac{\|[e]\|}{\|[x_{TS}]\|}$.

This combination of $\|[a]\|\|[a]^{-1}\|$ is hence very important in knowing what kind of error to expect in solving a linear algebra problem.

This known as the *condition number*.

$$Cond[a]=\|[a]\|\|[a]^{-1}\|$$

- $Cond[I]=1$ *Cond* of all other matrices > 1
- When *Cond* is close to 1,
  - $\frac{1}{Cond[a]}\frac{\|[r]\|}{\|[b]\|}\le\frac{\|[e]\|}{\|[x_{TS}]\|}\le Cond[a]\frac{\|[r]\|}{\|[b]\|}$
  - The relative error bounds are close to one another… and the relative error is of the same order of magnitude (factor of 10) as the relative residual.
- If *Cond[a]* is much larger than 1, then … the precision of the solution will be very limited (relative error bounds are very large).

$$[a]=\begin{bmatrix} 0.2 & -5 & 3 & 0.4 & 0 \\ -0.5 & 1 & 7 & -2 & 0.3 \\ 0.6 & 2 & -4 & 3 & 0.1 \\ 3 & 0.8 & 2 & -0.4 & 3 \\ 0.5 & 3 & 2 & 0.4 & 1 \end{bmatrix}$$

$$[a]^{-1}=\begin{bmatrix} -0.7079 & 2.5314 & 2.4312 & 0.9666 & -3.9023 \\ -0.1934 & 0.3101 & 0.2795 & 0.0577 & -0.2941 \\ 0.0217 & 0.3655 & 0.2861 & 0.0506 & -0.2899 \\ 0.2734 & -0.1299 & 0.1316 & -0.1410 & 0.4489 \\ 0.7815 & -2.8751 & -2.6789 & -0.7011 & 4.2338 \end{bmatrix}$$

Find Cond[a] using the infinity norm.