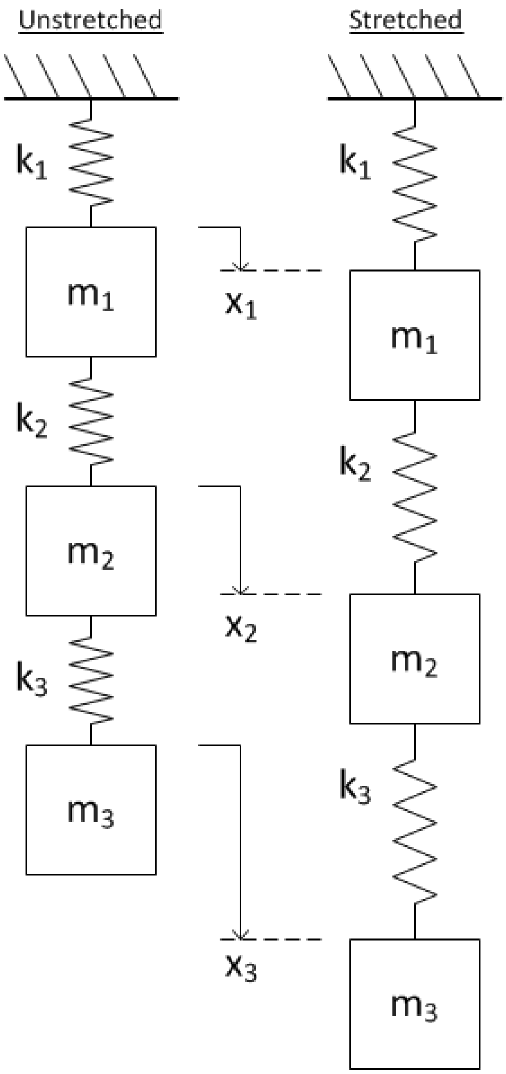
SECTION 3: SYSTEMS OF EQUATIONS

ESC 440 – Computational Methods for Engineers

Introduction

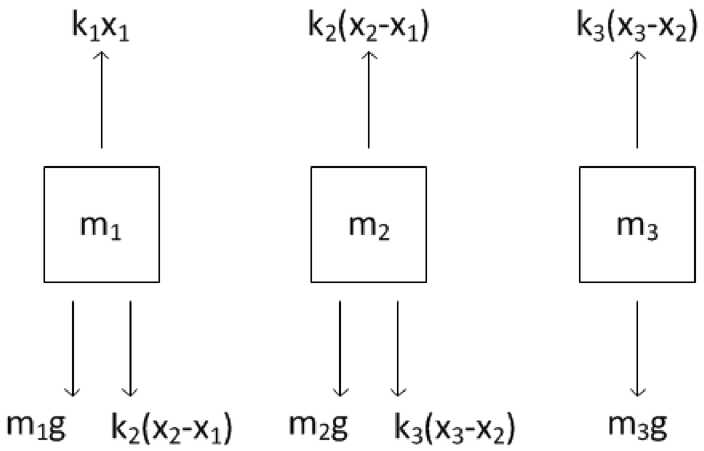
A System of Equations – Example



- Three masses
  - m_1, m_2, and m_3
- Three springs
  - k_1, k_2, k_3
- Connected in series and suspended
- Determine the displacement of each mass from its unstretched position

A System of Equations – Example

- Three unknown displacements: x_1, x_2, x_3
  - Need three equations to find displacements
- Apply Newton's second law to each mass



- Three equations result:

$$m_1\ddot{x}_1 = m_1 g + k_2(x_2 - x_1) - k_1 x_1$$

$$m_2\ddot{x}_2 = m_2 g + k_3(x_3 - x_2) - k_2(x_2 - x_1)$$

$$m_3\ddot{x}_3 = m_3 g - k_3(x_3 - x_2)$$

A System of Equations – Example

- Steady-state, so $\ddot{x}_i = 0,\ \forall i$

$$m_1 g + k_2(x_2 - x_1) - k_1 x_1 = 0$$

$$m_2 g + k_3(x_3 - x_2) - k_2(x_2 - x_1) = 0$$

$$m_3 g - k_3(x_3 - x_2) = 0$$

- Rearranging

$$\begin{aligned} (k_1 + k_2)x_1 \quad & - k_2 x_2 & + 0x_3 &= m_1 g \\ -k_2 x_1 \quad & + (k_2 + k_3)x_2 & - k_3 x_3 &= m_2 g \\ 0x_1 \quad & - k_3 x_2 & + k_3 x_3 &= m_3 g \end{aligned}$$

A System of Equations – Example

- Our system of three equations

$$\begin{aligned}(k_1+k_2)x_1 \quad & - k_2x_2 \quad + 0x_3 = m_1g \\ -k_2x_1 \quad + (k_2+k_3)x_2 \quad & - k_3x_3 = m_2g \\ 0x_1 \quad & - k_3x_2 \quad + k_3x_3 = m_3g\end{aligned}$$

can be put into matrix form

$$\begin{bmatrix}(k_1+k_2) & -k_2 & 0 \\ -k_2 & (k_2+k_3) & -k_3 \\ 0 & -k_3 & k_3\end{bmatrix}\begin{bmatrix}x_1\\x_2\\x_3\end{bmatrix}=\begin{bmatrix}m_1g\\m_2g\\m_3g\end{bmatrix}$$

A System of Equations – Example

$$\begin{bmatrix} (k_1 + k_2) & -k_2 & 0 \\ -k_2 & (k_2 + k_3) & -k_3 \\ 0 & -k_3 & k_3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} m_1 g \\ m_2 g \\ m_3 g \end{bmatrix}$$

- We can rewrite this matrix equation as

$$\mathbf{Ax} = \mathbf{b}$$

- Can apply tools of linear algebra to determine the vector of unknown displacements

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

Matrix notation

Conventions for matrix notation vary greatly. In general, the dimensions of a variable are known from context. These notes will use the following convention:

- Matrices
  - Upper-case, bold variables, e.g. $\mathbf{A}$
- Vectors
  - Lower-case, bold variables, e.g. $\mathbf{x}$
- Hand-written matrices and vectors
  - Underbar, instead of bold, e.g. $\underline{A}$ or $\underline{x}$

Solving Systems of Equations with Python

Before getting into the algorithms used to solve systems of linear equations, we'll take a look at how we can use available Python functions to find a solution.

System as a Matrix Equation

- Our system of equations has the form

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

- This can be written in matrix form as

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

or

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

Solving the Matrix Equation

- Solving our system of equations amounts to solving the matrix equation

$$\mathbf{Ax} = \mathbf{b}$$

for the vector $\mathbf{x}$

- To isolate $\mathbf{x}$ on the left of the equal sign, left multiply by the inverse of the coefficient matrix

$$\mathbf{A}^{-1}\mathbf{Ax} = \mathbf{A}^{-1}\mathbf{b}$$

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$$

Solving the Matrix Equation

- In NumPy's `linalg` module – left-multiply by $\mathbf{A}^{-1}$

```python
# %% solve using matrix inverse

import numpy as np

A = np.array([[7, 3, 8],
              [2, 1, 9],
              [0, 6, 4]])
b = np.array([3, 7, 2])

x = np.linalg.inv(A)@b

print('\n x =', x)
```

```
In [4]: runcell('solve using matrix inverse',
Notes/Python/Section3/linSysSolve.py')

 x = [-0.50359712 -0.28057554  0.92086331]
```

- Use `np.linalg.inv()` for matrix inversion
- Use @ for matrix multiplication
  - * performs element-by-element multiplication
- Note that $\mathbf{b}$ can be a row or column vector
  - Treated as a column vector either way
- Matrix inversion works, but is not always the best way to solve
  - Inefficient, slow
  - Sensitive to numerical error
    - Some systems worse than others

Solving the Matrix Equation

- Instead, use NumPy's `linalg.solve()` function

```python
#%% solve using np.linalg.solve

import numpy as np

A = np.array([[7, 3, 8],
              [2, 1, 9],
              [0, 6, 4]])
b = np.array([3, 7, 2])

x = np.linalg.solve(A,b)

print('\n x =', x)
```

```
In [5]: runcell('solve using np.linalg.solve',
Notes/Python/Section3/linSysSolve.py')

 x = [-0.50359712 -0.28057554  0.92086331]
```

- If $\mathbf{A}^{-1}$ exists, then

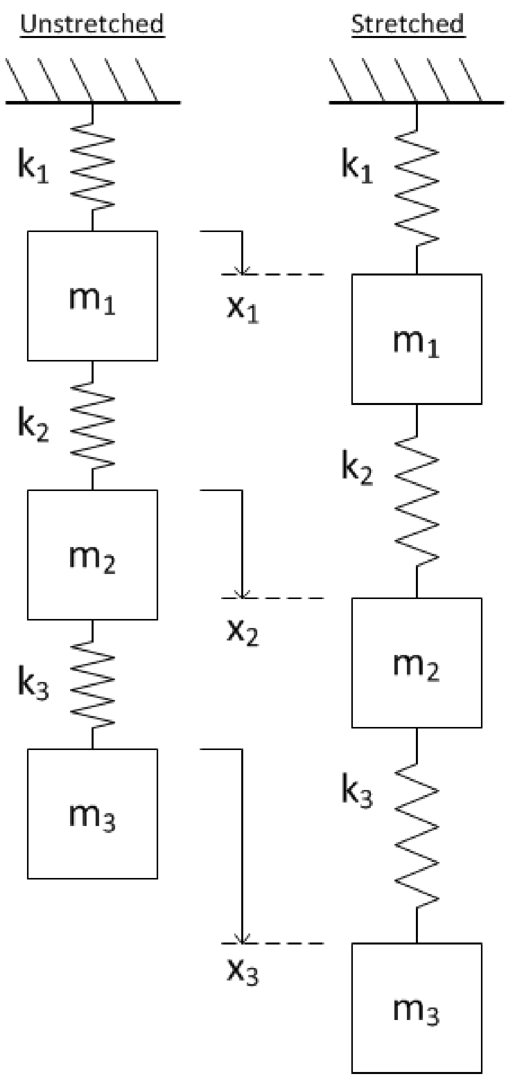
  x = `np.linalg.solve(A, b)`

  is equivalent to

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$$

- Does not calculate $\mathbf{A}^{-1}$
  - Faster, more robust
- Makes use of techniques we'll explore next

Example – Solving Using NumPy



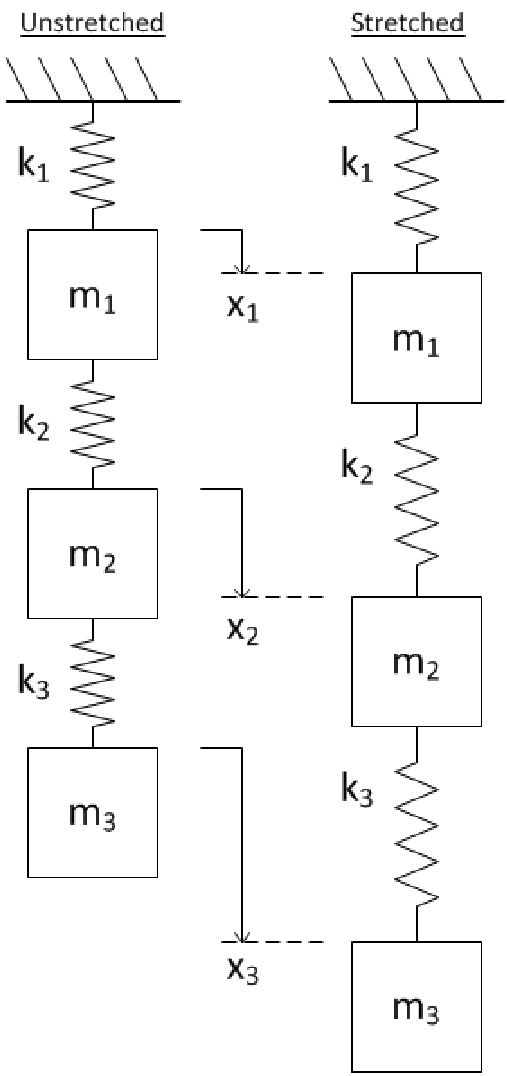
- Our linear system is described by the matrix equation

$$\begin{bmatrix} (k_1+k_2) & -k_2 & 0 \\ -k_2 & (k_2+k_3) & -k_3 \\ 0 & -k_3 & k_3 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} m_1 g \\ m_2 g \\ m_3 g \end{bmatrix}$$

$$\mathbf{Ax} = \mathbf{b}$$

- Find the displacements, $\mathbf{x}$, for the following system parameters
  - $k_1 = 500\frac{N}{m},\ k_2 = 800\frac{N}{m},\ k_3 = 400\frac{N}{m}$
  - $m_1 = 3kg,\ m_2 = 1kg,\ m_3 = 7kg$

Example – Solving Using NumPy




```python
# linSysEx.py

import numpy as np

# spring constants  [N/m]
k1 = 500
k2 = 800
k3 = 400

# masses  [kg]
m1 = 3
m2 = 1
m3 = 7

# grav. accel.  [m/s^2]
g = 9.81

A = np.array([[k1+k2,   -k2,   0],
              [-k2, k2+k3, -k3],
              [0,    -k3,  k3]])

b = np.array([m1*g, m2*g, m3*g])

x1 = np.linalg.inv(A)@b

x2 = np.linalg.solve(A, b)

print('\nSolution using linalg.inv():\n\tx =', x1)
print('\nSolution using linalg.solve():\n\tx =', x2)
```

```
In [235]: runfile('C:/Users/webbky/Box
Section3/linSysEx.py', wdir='C:/Users/
Python/Section3')

Solution using linalg.inv():
    x = [0.21582  0.31392  0.485595]

Solution using linalg.solve():
    x = [0.21582  0.31392  0.485595]
```

$$x_1 = 21.6cm, \quad x_2 = 31.4cm, \quad x_3 = 48.6cm$$

Techniques for Solving Linear Systems

Solving Systems of Linear Equations

- Techniques exist for finding the solution to ***small systems*** of linear equations:
  - ***Graphical method***
  - ***Cramer's rule***
  - ***Elimination of unknowns***
- Not generally useful for numerical solution of larger systems, but they do provide insight
- For numerical solution of ***larger systems*** techniques include:
  - ***Gaussian elimination***
  - ***Jacobi method***
  - ***Gauss-Seidel***

Graphical Solution

- A system of two linear equations with two unknown variables

$$a_{11}x_1 + a_{12}x_2 = b_1$$

$$a_{21}x_1 + a_{22}x_2 = b_2$$

can be thought of as equations of two lines in the $x - y$ plane:

$$x_2 = -\frac{a_{11}}{a_{12}}x_1 + \frac{b_1}{a_{12}}$$

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

Graphical Solution

$$x_2 = -\frac{a_{11}}{a_{12}}x_1 + \frac{b_1}{a_{12}}$$

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

- Solution to this system of equations is the point of intersection (x_1, x_2) of the two lines
  - May not exist
  - May not be unique
  - May exist, but be difficult to determine accurately

Unique Solution

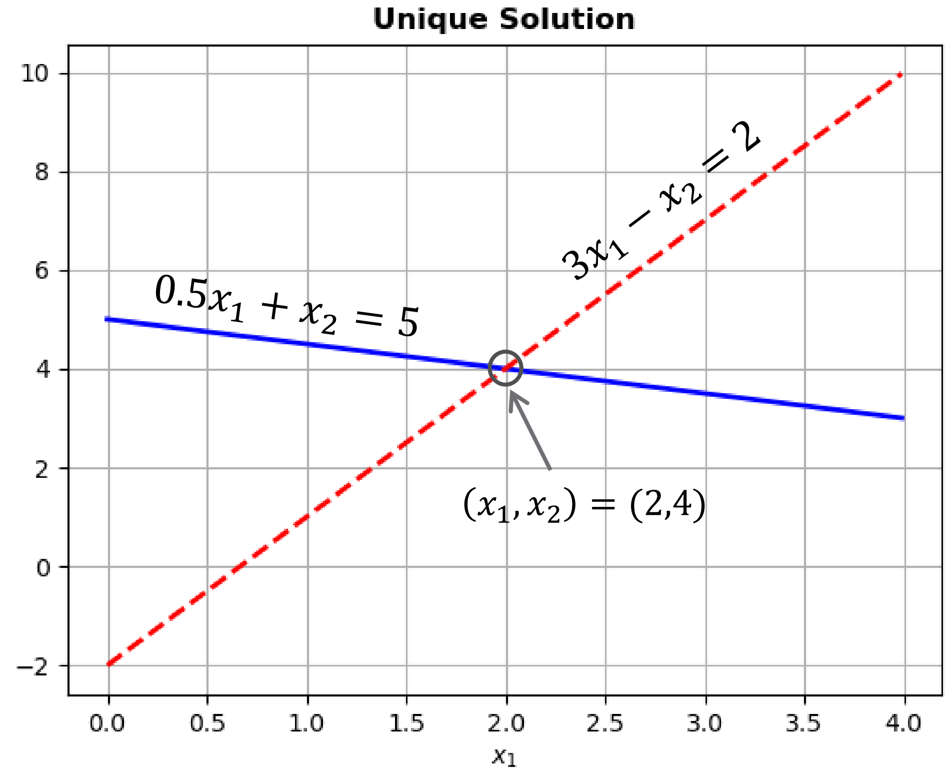
- System of two linear equations:

$$0.5x_1 + x_2 = 5$$
$$3x_1 - x_2 = 2$$

- Represented in matrix form

$$\begin{bmatrix} 0.5 & 1 \\ 3 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 5 \\ 2 \end{bmatrix}$$

$$\mathbf{Ax} = \mathbf{b}$$



- Solution at the point of intersection: $(x_1, x_2) = (2,4)$

No Solution

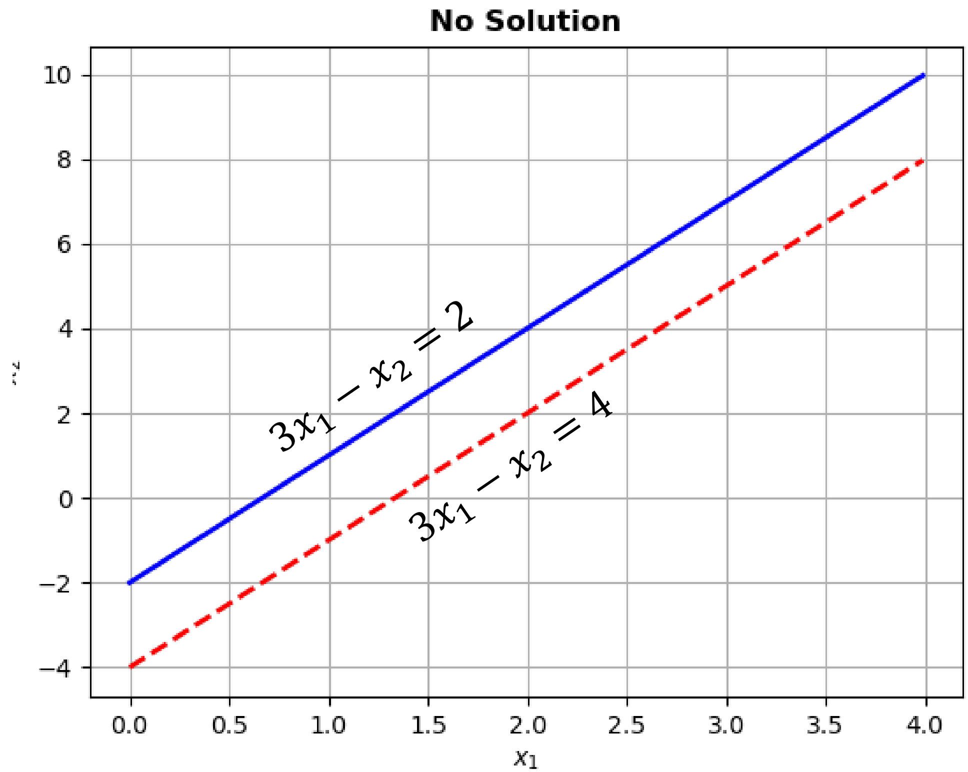
- System of two linear equations:

$$3x_1 - x_2 = 2$$
$$3x_1 - x_2 = 4$$

- Represented in matrix form

$$\begin{bmatrix} 3 & -1 \\ 3 & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2 \\ 4 \end{bmatrix}$$

$$\mathbf{Ax} = \mathbf{b}$$



- Lines don't intersect, so no solution exists

Infinite Solutions

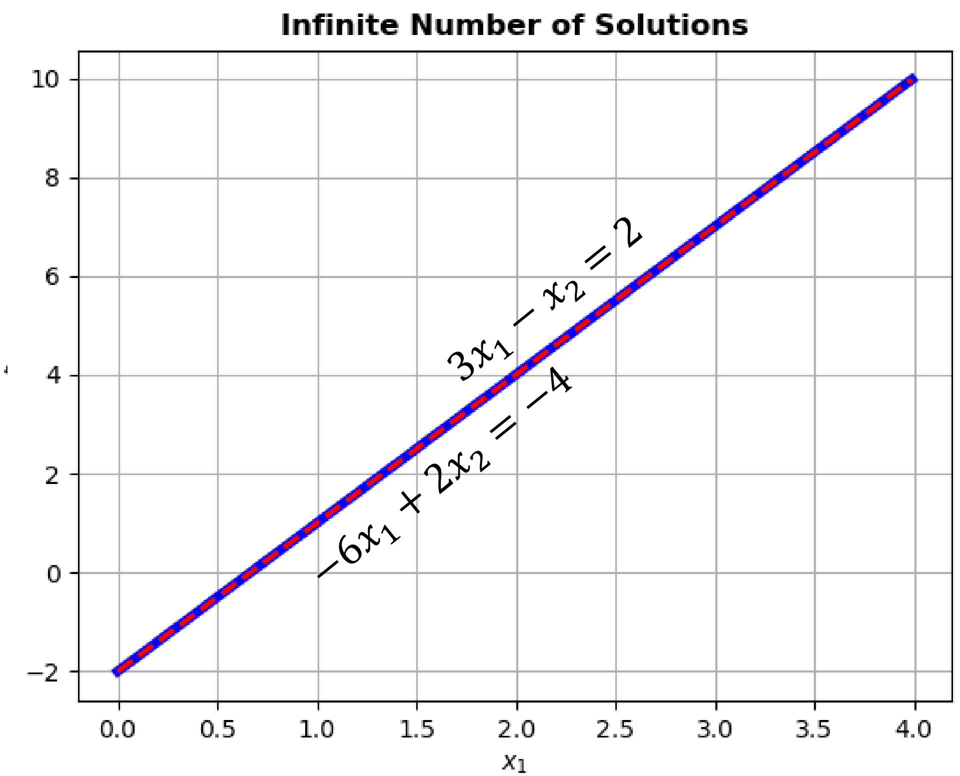
- System of two linear equations:

$$\begin{aligned} 3x_1 \quad - x_2 &= 2 \\ -6x_1 + 2x_2 &= -4 \end{aligned}$$

- Represented in matrix form

$$\begin{bmatrix} 3 & -1 \\ -6 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2 \\ -4 \end{bmatrix}$$

$$\mathbf{Ax} = \mathbf{b}$$



- Solutions at all points along the lines

Ill-Conditioned System

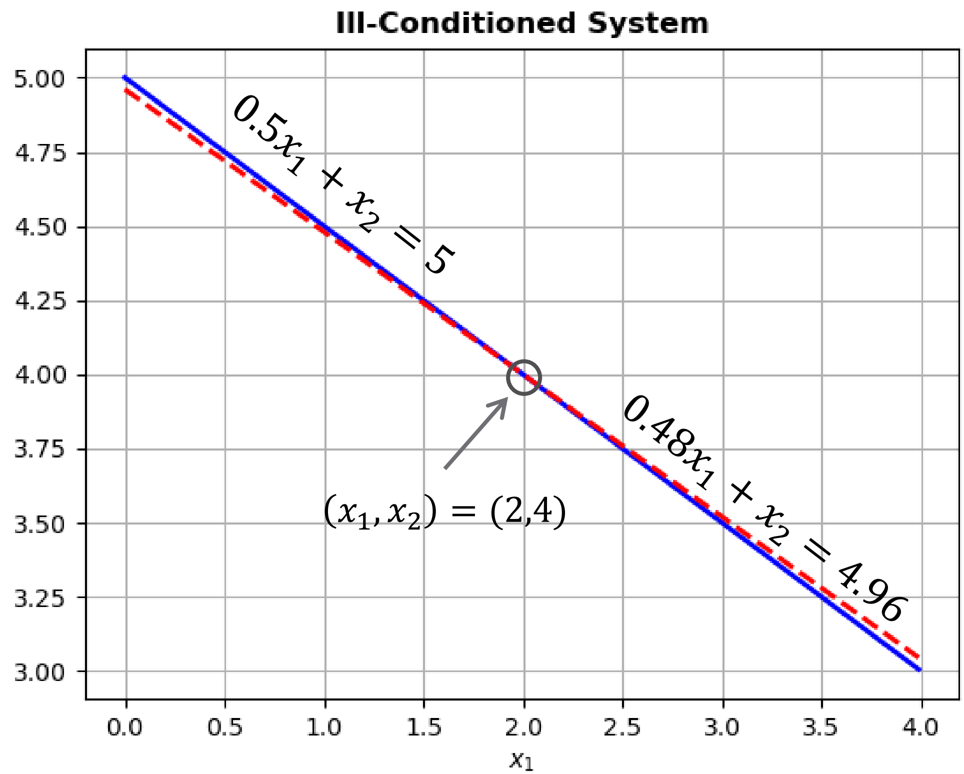
- System of two linear equations:

$$0.5x_1 + x_2 = 5$$
$$0.48x_1 + x_2 = 4.96$$

- Represented in matrix form

$$\begin{bmatrix} 0.5 & 1 \\ 0.48 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 5 \\ 4.96 \end{bmatrix}$$

$$\mathbf{Ax} = \mathbf{b}$$



- Solutions exists, but it is difficult to identify accurately

Singularity and the Coefficient Matrix, $\mathbf{A}$

- Systems with no solutions or infinite solutions are both referred to as ***singular***
- Coefficient matrix, $\mathbf{A}$, is singular
  - $\mathbf{A^{-1}}$, does not exist
  - $\det(\mathbf{A}) = 0$
- For the example with no solutions

$$\det(\mathbf{A}) = \begin{vmatrix} 3 & -1 \\ 3 & -1 \end{vmatrix} = -3 - (-3) = 0$$

- For the example with infinite solutions

$$\det(\mathbf{A}) = \begin{vmatrix} 3 & -1 \\ -6 & 2 \end{vmatrix} = 6 - 6 = 0$$

Ill-Conditioned Systems

- Ill-conditioned systems are nearly-singular
  - $\det(\mathbf{A}) \approx 0$
  - $\mathbf{A}^{-1}$ exists, but may be difficult to determine accurately
  - Solution exists, but it may difficult to determine accurately – either graphically or numerically
- For the previous example of an ill-conditioned system

$$\det(\mathbf{A}) = \begin{vmatrix} 0.5 & 1 \\ 0.48 & 1 \end{vmatrix} = 0.5 - 0.48 = 0.02$$

(This example may be ill-conditioned for graphical solution, but would not be if solving numerically)

Rank of the Coefficient Matrix, $\mathbf{A}$

- ***Rank of a matrix*** – number of linearly-independent rows (or columns) of the matrix
- ***Full-rank*** matrix
  - All rows and columns are linearly-independent
  - Must be square
  - $\det(\mathbf{A}) \neq 0$, $\mathbf{A}^{-1}$ exists
- In both of our singular examples $\mathbf{A}$ is ***rank-deficient***

$$\mathbf{A_1} = \begin{bmatrix} 3 & -1 \\ 3 & -1 \end{bmatrix} \quad \text{and} \quad \mathbf{A_2} = \begin{bmatrix} 3 & -1 \\ -6 & 2 \end{bmatrix}$$

- For a 2×2, rank-deficient matrix, columns and rows represent ***collinear vectors***

Gaussian Elimination

Gaussian Elimination

- Two steps in Gaussian elimination:
  - ***Elimination of unknowns***
  - ***Solution through back-substitution***
- Applies to arbitrarily large systems

$$
\begin{aligned}
a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\
a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\
\vdots \qquad\qquad\qquad & \quad \vdots \\
a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n
\end{aligned}
$$

- The basic algorithm will be introduced using an example system of three equations with three unknowns

Gaussian Elimination – the Basic Algorithm

- The basic algorithm:
  1. ***Forward elimination of unknowns***
     - Reduce to an ***upper-triangular*** system
  2. ***Back-substitution to solve for unknowns***
     - Reduction to an upper-triangular system yields the solution for x_n directly
     - Back-substitute the solution for x_n to solve for x_{n-1}
     - Back-substitute the solution for x_{n-1} to solve for x_{n-2}
     - Continue until all x_i have been determined

Forward Elimination of Unknowns

- We'll use a system of three equations with three unknowns as an example

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

- Create the ***augmented*** system matrix

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

- Each row represents an equation – row operations are operations on the equations

Forward Elimination of Unknowns

- Reduce to an upper-triangular system
  - Eliminate x_i from the $(i+1)^{\text{st}}$ through n^{th} equations for $i = 1 \dots n$

- First eliminate x_1 from the second equation
  - Perform row operations to set the first element on the second row to zero
  - ***Normalize*** the first equation (row) – divide by the leading coefficient, a_{11}
  - Multiply the first equation (row) by the leading coefficient of the second equation (row), a_{21}

$$\begin{bmatrix} a_{21} & \frac{a_{21}}{a_{11}} a_{12} & \frac{a_{21}}{a_{11}} a_{13} & \vdots & \frac{a_{21}}{a_{11}} b_1 \\ a_{21} & a_{22} & a_{23} & \vdots & b_2 \\ a_{31} & a_{32} & a_{33} & \vdots & b_3 \end{bmatrix}$$

Forward Elimination of Unknowns

- Subtract the first row from the second, and replace the first row with its original values

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \vdots & b_1 \\ 0 & a_{22} - \dfrac{a_{21}}{a_{11}} a_{12} & a_{23} - \dfrac{a_{21}}{a_{11}} a_{13} & \vdots & b_2 - \dfrac{a_{21}}{a_{11}} b_1 \\ a_{31} & a_{32} & a_{33} & \vdots & b_3 \end{bmatrix}$$

- Use *prime* notation to indicate a modified coefficient value
  - Add additional *prime* mark for each modification

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \vdots & b_1 \\ 0 & a'_{22} & a'_{23} & \vdots & b'_2 \\ a_{31} & a_{32} & a_{33} & \vdots & b_3 \end{bmatrix}$$

Forward Elimination of Unknowns

- Next, eliminate x_1 from the third equation
  - ***Normalize*** the first row
  - Multiply by the leading coefficient of the third row, a_{31}

$$\begin{bmatrix} a_{31} & \frac{a_{31}}{a_{11}}a_{12} & \frac{a_{31}}{a_{11}}a_{13} & \vdots & \frac{a_{31}}{a_{11}}b_1 \\ 0 & a'_{22} & a'_{23} & \vdots & b'_2 \\ a_{31} & a_{32} & a_{33} & \vdots & b_3 \end{bmatrix}$$

  - Subtract the first row from the third and reset the first row to its original values

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \vdots & b_1 \\ 0 & a'_{22} & a'_{23} & \vdots & b'_2 \\ 0 & a_{32}-\frac{a_{31}}{a_{11}}a_{12} & a_{33}-\frac{a_{31}}{a_{11}}a_{13} & \vdots & b_3-\frac{a_{31}}{a_{11}}b_1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \vdots & b_1 \\ 0 & a'_{22} & a'_{23} & \vdots & b'_2 \\ 0 & a'_{32} & a'_{33} & \vdots & b'_3 \end{bmatrix}$$

Elimination of Unknowns - Terminology

- First row is used for the elimination of x_1 from second and third rows
- In general, i^{th} row used to eliminate the i^{th} unknown from the $(i+1)^{\text{st}}$ through n^{th} rows
  - This is the ***pivot row***
  - $(n-1)$ rows will be pivot rows at some point
  - Leading coefficient in the pivot row, a_{ii}, is the ***pivot element***
- ***Normalization*** involves dividing the pivot row by the pivot element
  - Could this be problematic?

Forward Elimination of Unknowns

- Finally, eliminate x_2 from the third equation
  - ***Normalize*** the second row (the pivot row)
  - Multiply by the leading coefficient of the third row, a'_{32}

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \vdots & b_1 \\ 0 & a'_{32} & \frac{a'_{32}}{a'_{22}}a'_{23} & \vdots & \frac{a'_{32}}{a'_{22}}b'_2 \\ 0 & a'_{32} & a'_{33} & \vdots & b'_3 \end{bmatrix}$$

  - Subtract the second row from the third and reset the second row to its previous values

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \vdots & b_1 \\ 0 & a'_{22} & a'_{23} & \vdots & b'_2 \\ 0 & 0 & a'_{33} - \frac{a'_{32}}{a'_{22}}a'_{23} & \vdots & b'_3 - \frac{a'_{32}}{a'_{22}}b'_2 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \vdots & b_1 \\ 0 & a'_{22} & a'_{23} & \vdots & b'_2 \\ 0 & 0 & a''_{33} & \vdots & b''_3 \end{bmatrix}$$

Back-Substitution

- System is now ***upper-triangular***

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \vdots & b_1 \\ 0 & a'_{22} & a'_{23} & \vdots & b'_2 \\ 0 & 0 & a''_{33} & \vdots & b''_3 \end{bmatrix}$$

- Last row represents a single equation with a single unknown, x_3

$$x_3 = \frac{b''_3}{a''_{33}}$$

- In general, solve for the n^{th} unknown as

$$x_n = \frac{b_n^{(n-1)}}{a_{nn}^{(n-1)}}$$

Back-Substitution

- Next, substitute x_3 into the second equation

$$a'_{22}x_2 + a'_{23}x_3 = b'_2$$

$$a'_{22}x_2 + a'_{23}\frac{b''_3}{a''_{33}} = b'_2$$

and solve for x_2

$$x_2 = \frac{b'_2 - a'_{23}\dfrac{b''_3}{a''_{33}}}{a'_{22}}$$

- In general:

$$x_i = \frac{1}{a_{ii}^{(i-1)}}\left(b_i^{(i-1)} - \sum_{j=i+1}^{n} a_{ij}^{(i-1)}x_j\right)$$

Back-Substitution

- Finally, substitute x_2 and x_3 into the first equation

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 = b_1$$

$$a_{11}x_1 + a_{12}\frac{b_2' - a_{23}'\frac{b_3''}{a_{33}''}}{a_{22}'} + a_{13}\frac{b_3''}{a_{33}''} = b_1$$

and solve for x_1

$$x_1 = \frac{b_1 - a_{12}\frac{b_2' - a_{23}'\frac{b_3''}{a_{33}''}}{a_{22}'} - a_{13}\frac{b_3''}{a_{33}''}}{a_{11}}$$

- In practice we'd solve for x_1 using the general formula

$$x_i = \frac{1}{a_{ii}^{(i-1)}}\left(b_i^{(i-1)} - \sum_{j=i+1}^{n} a_{ij}^{(i-1)}x_j\right)$$

Algorithm Summary

1) Form augmented system matrix
2) ***Elimination of unknowns*** – for $i = 1 \ldots n-1$
   a) Normalize pivot row (i^{th} row)
   b) Multiply pivot row by leading coefficient of j^{th} row, a_{ji} (for $j = (i+1) \ldots n$)
   c) Subtract pivot row from j^{th} row
3) ***Back-substitution***
   a) Determine x_n from the last row: $x_n = \frac{b_n^{(n-1)}}{a_{nn}^{(n-1)}}$
   b) Solve for remaining x_i for $i = (n-1) \ldots 1$:

$$x_i = \frac{1}{a_{ii}^{(i-1)}}\left(b_i^{(i-1)} - \sum_{j=i+1}^{n} a_{ij}^{(i-1)} x_j\right)$$

Partial Pivoting

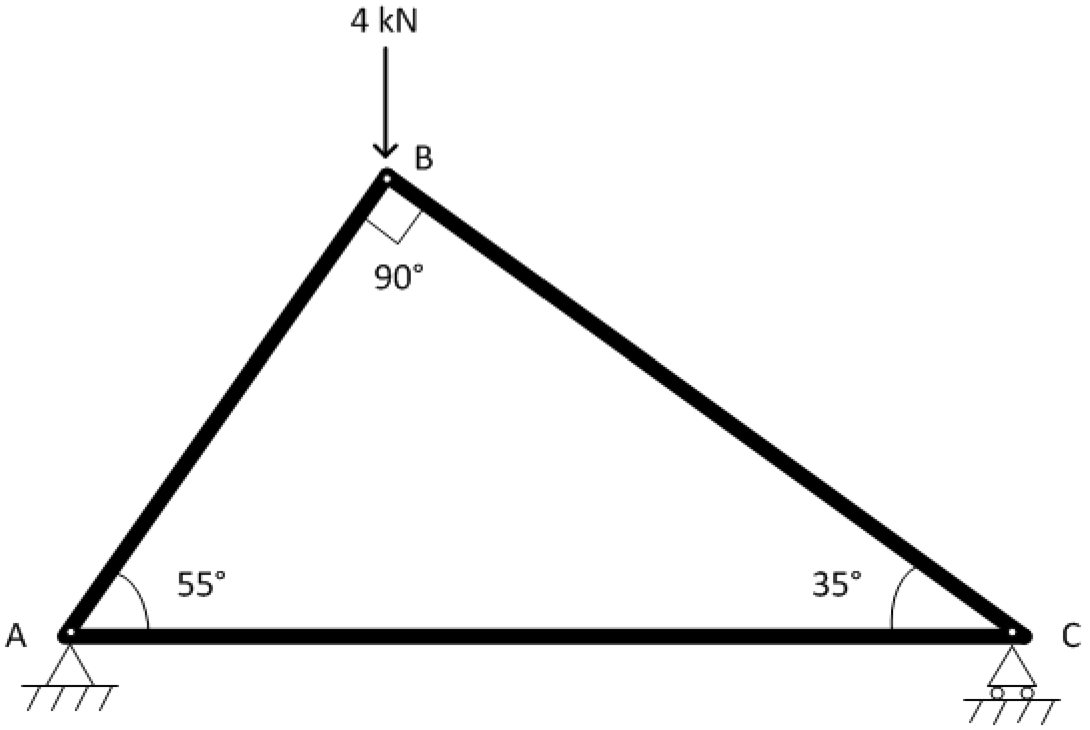
- During forward elimination of unknowns, pivot row is normalized
  - i^{th} row divided by leading coefficient, a_{ii}
  - If $a_{ii} = 0$ → divide-by-zero, algorithm fails
  - If $a_{ii} \approx 0$ → not fatal, but susceptible to roundoff error

- ***Partial pivoting***
  - Prior to normalizing the pivot (i^{th}) row, search all rows from $i \dots n$ for the one with the largest value in the i^{th} column
  - Move to the current pivot row location and continue with algorithm

Gaussian Elimination - Example

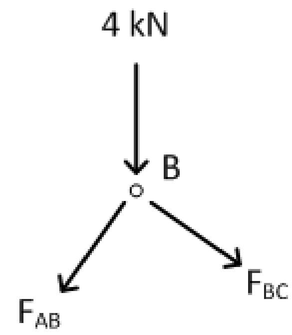
Example – Truss Analysis

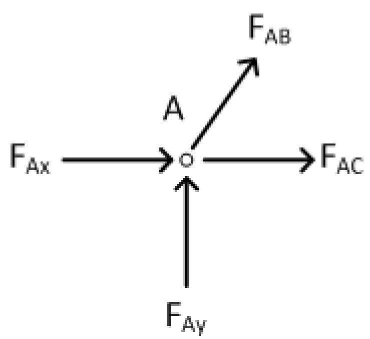
- Simple statically-determinate truss
- Determine all internal and external forces



Example – Truss Analysis

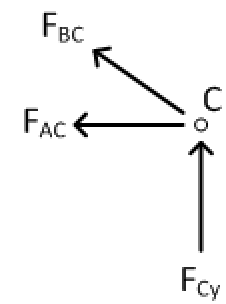
- Force components at each joint must balance





$$-F_{AB}\cos(55°) + F_{BC}\cos(35°) = 0$$

$$-4\ kN - F_{AB}\sin(55°) - F_{BC}\sin(35°) = 0$$



$$F_{Ax} + F_{AC} + F_{AB}\cos(55°) = 0$$

$$F_{Ay} + F_{AB}\sin(55°) = 0$$

$$-F_{AC} - F_{BC}\cos(35°) = 0$$

$$F_{Cy} + F_{BC}\sin(35°) = 0$$

Example – Truss Analysis

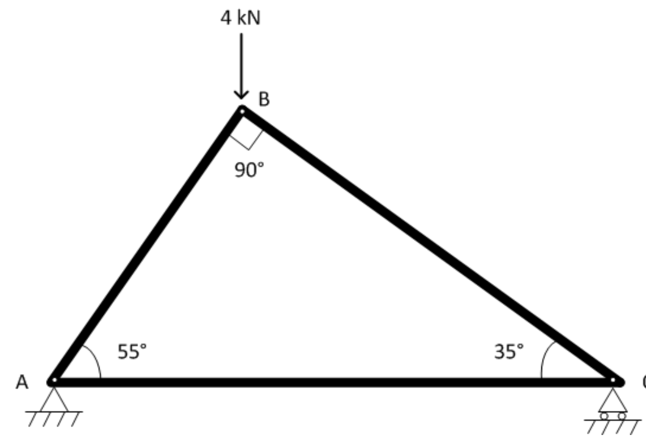
- System of six equations with six unknown internal and external forces

$$\begin{bmatrix} \cos(55°) & 1 & 0 & 1 & 0 & 0 \\ \sin(55°) & 0 & 0 & 0 & 1 & 0 \\ -\cos(55°) & 0 & \cos(35°) & 0 & 0 & 0 \\ \sin(55°) & 0 & \sin(35°) & 0 & 0 & 0 \\ 0 & -1 & -\cos(35°) & 0 & 0 & 0 \\ 0 & 0 & \sin(35°) & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} F_{AB} \\ F_{AC} \\ F_{BC} \\ F_{Ax} \\ F_{Ay} \\ F_{Cy} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ -4000 \\ 0 \\ 0 \end{bmatrix}$$

- Python Gaussian elimination demo…

Example – Truss Analysis


```python
# truss_example.py

import numpy as np
from gausselim import gausselim


theta1 = np.radians(55)
theta2 = np.radians(35)

A = np.array([[np.cos(theta1),  1,                0, 1, 0, 0],
              [np.sin(theta1),  0,                0, 0, 1, 0],
              [-np.cos(theta1), 0,  np.cos(theta2), 0, 0, 0],
              [np.sin(theta1),  0,  np.sin(theta2), 0, 0, 0],
              [0,              -1, -np.cos(theta2), 0, 0, 0],
              [0,               0,  np.sin(theta2), 0, 0, 1]])

b = np.array([0, 0, 0, -4e3, 0, 0])
x = np.linalg.solve(A, b)

x = gausselim(A,b)

print('\n x = \n', x)
```




```
In [42]: runfile('C:/Users
wdir='C:/Users/webbky/Box/
Reloaded modules: gausseli

 x =
 [[-3276.60817716]
 [ 1879.38524157]
 [-2294.3057454 ]
 [    0.        ]
 [ 2684.04028665]
 [ 1315.95971335]]

In [43]:
```

$$F_{AB} = -3.277\ kN \qquad F_{Ax} = 0\ N$$

$$F_{AC} = 1.879\ kN \qquad F_{Ay} = 2.684\ kN$$

$$F_{BC} = -2.294\ kN \qquad F_{Cy} = 1.316\ kN$$

Gaussian Elimination

- Gaussian elimination summary:
  - Create the augmented system matrix
  - Forward elimination
    - Reduce to an upper-triangular matrix
  - Back substitution
    - Starting with x_N, solve for x_i for $i = N \dots 1$

- A ***direct solution*** algorithm
  - Exact value for each x_i arrived at with a single execution of the algorithm
- Alternatively, we can use an ***iterative*** algorithm
  - ***Jacobi method***
  - ***Gauss-Seidel***
  - ***Newton-Raphson***

Linear Systems of Equations – Iterative Solution – Jacobi Method

Jacobi Method

- Consider a system of N linear equations

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{y}$$

$$\begin{bmatrix} a_{1,1} & \cdots & a_{1,N} \\ \vdots & \ddots & \vdots \\ a_{N,1} & \cdots & a_{N,N} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_N \end{bmatrix} = \begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix}$$

- The k^{th} equation (k^{th} row) is

$$a_{k,1}x_1 + a_{k,2}x_2 + \cdots + a_{k,k}x_k + \cdots + a_{k,N}x_N = y_k \qquad (1)$$

- Solve (1) for x_k

$$x_k = \frac{1}{a_{k,k}}\left[y_k - \left(a_{k,1}x_1 + a_{k,2}x_2 + \cdots + a_{k,k-1}x_{k-1} + \right.\right. \qquad (2)$$

$$\left.\left. + a_{k,k+1}x_{k+1} + \cdots + a_{k,N}x_N\right)\right]$$

Jacobi Method

- Simplify (2) using summing notation

$$x_k = \frac{1}{a_{k,k}} \left[y_k - \sum_{n=1}^{k-1} a_{k,n} x_n - \sum_{n=k+1}^{N} a_{k,n} x_n \right], \qquad k = 1 \ldots N \qquad (3)$$

- An equation for x_k
  - But, of course, we don't yet know all other x_n values
- Use (3) as an ***iterative expression***

$$x_{k,i+1} = \frac{1}{a_{k,k}} \left[y_k - \sum_{n=1}^{k-1} a_{k,n} x_{n,i} - \sum_{n=k+1}^{N} a_{k,n} x_{n,i} \right], \qquad k = 1 \ldots N \qquad (4)$$

  - The i subscript indicates iteration number
    - $x_{k,i+1}$ is the updated value from the current iteration
    - $x_{n,i}$ is a value from the previous iteration

Jacobi Method

$$x_{k,i+1} = \frac{1}{a_{k,k}}\left[y_k - \sum_{n=1}^{k-1} a_{k,n}x_{n,i} - \sum_{n=k+1}^{N} a_{k,n}x_{n,i}\right], \qquad k = 1 \dots N \qquad (4)$$

- Old values of x_n, on the right-hand side, are used to update x_k on the left-hand side
- Start with an ***initial guess*** for all unknowns, $\mathbf{x}_0$
- Iterate until adequate ***convergence*** is achieved
  - Until a specified ***stopping criterion*** is satisfied
  - Convergence is not guaranteed

Convergence

- An approximation of $\mathbf{x}$ is refined on each iteration
- Continue to iterate until we're *close* to the right answer for the vector of unknowns, $\mathbf{x}$
  - Assume we've converged to the right answer when $\mathbf{x}$ changes very little from iteration to iteration
- On each iteration, calculate a ***relative error*** quantity

$$\varepsilon_{i+1} = \max\left(\left|\frac{x_{k,i+1} - x_{k,i}}{x_{k,i+1}}\right|\right), \qquad k = 1 \ldots N$$

- Iterate until

$$\varepsilon_i \leq \varepsilon_s$$

where ε_s is a chosen ***stopping criterion***

Jacobi Method – Matrix Form

- The Jacobi method iterative formula, (4), can be rewritten in matrix form:

$$\mathbf{x}_{i+1} = \mathbf{M}\mathbf{x}_i + \mathbf{D}^{-1}\mathbf{y} \tag{5}$$

where $\mathbf{D}$ is the diagonal elements of $\mathbf{A}$

$$\mathbf{D} = \begin{bmatrix} a_{1,1} & 0 & \cdots & 0 \\ 0 & a_{2,2} & 0 & \vdots \\ \vdots & 0 & \ddots & 0 \\ 0 & \cdots & 0 & a_{N,N} \end{bmatrix}$$

and

$$\mathbf{M} = \mathbf{D}^{-1}(\mathbf{D} - \mathbf{A}) \tag{6}$$

  - Recall that the inverse of a diagonal matrix is given by inverting each diagonal element

$$\mathbf{D}^{-1} = \begin{bmatrix} 1/a_{1,1} & 0 & \cdots & 0 \\ 0 & 1/a_{2,2} & 0 & \vdots \\ \vdots & 0 & \ddots & 0 \\ 0 & \cdots & 0 & 1/a_{N,N} \end{bmatrix}$$

Jacobi Method – Example

- Consider the following system of equations

$$-4x_1 + 7x_3 = -5$$
$$2x_1 - 3x_2 + 5x_3 = -12$$
$$x_2 - 3x_3 = 3$$

- In matrix form:

$$\begin{bmatrix} -4 & 0 & 7 \\ 2 & -3 & 5 \\ 0 & 1 & -3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -5 \\ -12 \\ 3 \end{bmatrix}$$

- Solve using the Jacobi method

Jacobi Method – Example

- The iteration formula is

$$\mathbf{x}_{i+1} = \mathbf{M}\mathbf{x}_i + \mathbf{D}^{-1}\mathbf{y}$$

where

$$\mathbf{D} = \begin{bmatrix} -4 & 0 & 0 \\ 0 & -3 & 0 \\ 0 & 0 & -3 \end{bmatrix} \qquad \mathbf{D}^{-1} = \begin{bmatrix} -0.25 & 0 & 0 \\ 0 & -0.333 & 0 \\ 0 & 0 & -0.333 \end{bmatrix}$$

$$\mathbf{M} = \mathbf{D}^{-1}(\mathbf{D} - \mathbf{A}) = \begin{bmatrix} 0 & 0 & 1.75 \\ 0.667 & 0 & 1.667 \\ 0 & 0.333 & 0 \end{bmatrix}$$

- To begin iteration, we need a starting point
  - Initial guess for unknown values, $\mathbf{x}$
  - Often, we have some idea of the answer
  - Here, arbitrarily choose

$$\mathbf{x}_0 = [10 \quad 25 \quad 10]^T$$

Jacobi Method – Example

- At each iteration, calculate

$$\mathbf{x}_{i+1} = \mathbf{M}\mathbf{x}_i + \mathbf{D}^{-1}\mathbf{y}$$

$$\begin{bmatrix} x_{1,i+1} \\ x_{2,i+1} \\ x_{3,i+1} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1.75 \\ 0.667 & 0 & 1.667 \\ 0 & 0.333 & 0 \end{bmatrix} \begin{bmatrix} x_{1,i} \\ x_{2,i} \\ x_{3,i} \end{bmatrix} + \begin{bmatrix} 1.25 \\ 4 \\ -1 \end{bmatrix}$$

- For $i = 0$:

$$\mathbf{x}_1 = \begin{bmatrix} x_{1,1} \\ x_{2,1} \\ x_{3,1} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1.75 \\ 0.667 & 0 & 1.667 \\ 0 & 0.333 & 0 \end{bmatrix} \begin{bmatrix} 10 \\ 25 \\ 10 \end{bmatrix} + \begin{bmatrix} 1.25 \\ 4 \\ -1 \end{bmatrix}$$

$$\mathbf{x}_1 = [18.75 \quad 27.33 \quad 7.33]^T$$

- The relative error is

$$\varepsilon_1 = \max\left(\left|\frac{x_{k,1} - x_{k,0}}{x_{k,1}}\right|\right) = 0.467$$

Jacobi Method – Example

- For $i = 1$:

$$\mathbf{x}_2 = \begin{bmatrix} x_{1,2} \\ x_{2,2} \\ x_{3,2} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1.75 \\ 0.667 & 0 & 1.667 \\ 0 & 0.333 & 0 \end{bmatrix} \begin{bmatrix} 18.75 \\ 27.33 \\ 7.33 \end{bmatrix} + \begin{bmatrix} 1.25 \\ 4 \\ -1 \end{bmatrix}$$

$$\mathbf{x}_2 = [14.08 \quad 28.72 \quad 8.11]^T$$

- The relative error is

$$\varepsilon_2 = \max\left(\left|\frac{x_{k,2} - x_{k,1}}{x_{k,1}}\right|\right) = 0.331$$

- Continue to iterate until relative error falls below a specified stopping condition

Jacobi Method – Example

- Automate with computer code, e.g. Python
- Setup the system of equations

```
# coefficient matrix
A = np.array([[-4, 0, 7],
              [2, -3, 5],
              [0, 1, -3]])

# vector of knowns
y = np.array([-5, -12, 3])
```

- Initialize matrices and parameters for iteration

```
reltol = 1e-6
eps = 1

max_iter = 600
iter = 0

# initial guess for x
x = np.array([10, 25, 10])

D = np.diag(np.diag(A))
invD = np.linalg.inv(D)

M = invD@(D - A)
```

Jacobi Method – Example

- Loop to continue iteration as long as:
  - Stopping criterion is not satisfied
  - Maximum number of iterations is not exceeded

```
while((eps > reltol) and (iter < max_iter)):
    xold = x
    x = M@xold + invD@y

    eps = np.max(abs((x - xold)/x))

    iter = iter + 1
```

- On each iteration
  - Use previous $\mathbf{x}$ values to update $\mathbf{x}$
  - Calculate relative error
  - Increment the number of iterations

Jacobi Method – Example

- Set $\varepsilon_s = 1 \times 10^{-6}$ and iterate:

i	$\mathbf{x}_i$	ε_i
0	$[10 \quad 25 \quad 10]^T$	-
1	$[18.75 \quad 27.33 \quad 7.33]^T$	0.467
2	$[14.08 \quad 28.72 \quad 8.11]^T$	0.331
3	$[15.44 \quad 26.91 \quad 8.57]^T$	0.088
4	$[16.25 \quad 28.59 \quad 7.97]^T$	0.076
5	$[15.20 \quad 28.12 \quad 8.53]^T$	0.070
6	$[16.18 \quad 28.35 \quad 8.37]^T$	0.061
⋮	⋮	⋮
371	$[20.50 \quad 36.00 \quad 11.00]^T$	0.995×10^{-6}

- Convergence achieved in 371 iterations

Linear Systems of Equations – Iterative Solution – Gauss-Seidel

Gauss-Seidel Method

- The iterative formula for the Jacobi method is

$$x_{k,i+1} = \frac{1}{a_{k,k}}\left[y_k - \sum_{n=1}^{k-1} a_{k,n}x_{n,i} - \sum_{n=k+1}^{N} a_{k,n}x_{n,i}\right], \qquad k = 1 \ldots N \qquad (4)$$

- Note that only old values of x_n (i.e. $x_{n,i}$) are used to update the value of x_k
- Assume the $x_{k,i+1}$ values are determined in order of increasing k
  - When updating $x_{k,i+1}$, all $x_{n,i+1}$ values are already known for $n < k$
  - We can use those updated values to calculate $x_{k,i+1}$
  - The ***Gauss-Seidel method***

Gauss-Seidel Method

- Now use the x_n values already updated on the current iteration to update x_k
  - That is, $x_{n,i+1}$ for $n < k$
- ***Gauss-Seidel*** iterative formula

$$x_{k,i+1} = \frac{1}{a_{k,k}}\left[y_k - \sum_{n=1}^{k-1} a_{k,n}x_{n,i+1} - \sum_{n=k+1}^{N} a_{k,n}x_{n,i}\right], \qquad k = 1 \ldots N \qquad (7)$$

- Note that only the first summation has changed
  - For already updated x values
  - x_n for $n < k$
  - Number of already-updated values used depends on k

Gauss-Seidel – Matrix Form

- In matrix form the iterative formula is the same as for the Jacobi method

$$\mathbf{x}_{i+1} = \mathbf{M}\mathbf{x}_i + \mathbf{D}^{-1}\mathbf{y} \tag{5}$$

where, again

$$\mathbf{M} = \mathbf{D}^{-1}(\mathbf{D} - \mathbf{A}) \tag{6}$$

but now $\mathbf{D}$ is the lower triangular part of $\mathbf{A}$

$$\mathbf{D} = \begin{bmatrix} a_{1,1} & 0 & \cdots & 0 \\ a_{2,1} & a_{2,2} & 0 & \vdots \\ \vdots & \vdots & \ddots & 0 \\ a_{N,1} & a_{N,2} & \cdots & a_{N,N} \end{bmatrix}$$

- Otherwise, the algorithm and computer code is identical to that of the Jacobi method

Gauss-Seidel – Example

- Apply Gauss-Seidel to our previous example
  - $x_0 = [10 \quad 25 \quad 10]^T$
  - $\varepsilon_s = 1 \times 10^{-6}$

i	$\mathbf{x}_i$	ε_i
0	$[10 \quad 25 \quad 10]^T$	-
1	$[18.75 \quad 33.17 \quad 10.06]^T$	0.875
2	$[18.85 \quad 33.32 \quad 10.11]^T$	0.005
3	$[18.94 \quad 33.47 \quad 10.16]^T$	0.005
4	$[19.03 \quad 33.61 \quad 10.20]^T$	0.005
⋮	⋮	⋮
151	$[20.50 \quad 36.00 \quad 11.00]^T$	0.995×10^{-6}

- Convergence achieved in 151 iterations
  - Compared to 371 for the Jacobi method

Nonlinear Systems of Equations

We have seen how to apply the Newton-Raphson root-finding algorithm to solve a single nonlinear equation.

We will now extend that algorithm to the solution of a system of nonlinear equations

Nonlinear Systems of Equations

- Consider a system of nonlinear equations
  - Can be represented as a vector of N functions
  - Each is a function of an N-vector of unknown variables

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix} = \mathbf{f}(\mathbf{x}) = \begin{bmatrix} f_1(x_1, x_2, \cdots, x_N) \\ f_2(x_1, x_2, \cdots, x_N) \\ \vdots \\ f_N(x_1, x_2, \cdots, x_N) \end{bmatrix}$$

- As we did when applying Newton-Raphson to find the root of a single equation, we can again approximate this function as linear (i.e., a first-order Taylor series approximation)

$$\mathbf{y} = \mathbf{f}(\mathbf{x}) \approx \mathbf{f}(\mathbf{x}_0) + \mathbf{f}'(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0) \tag{8}$$

  - Note that all variables are N-vectors
    - $\mathbf{f}$ is an N-vector of known, nonlinear functions
    - $\mathbf{x}$ is an N-vector of unknown values – this is what we want to solve for
    - $\mathbf{y}$ is an N-vector of known values
    - $\mathbf{x_0}$ is an N-vector of $\mathbf{x}$ values for which $\mathbf{f}(\mathbf{x}_0)$ is known

Newton-Raphson Method

- Equation (8) is the basis for our Newton-Raphson iterative formula
  - Let it be an equality and solve for $\mathbf{x}$

$$\mathbf{y} - \mathbf{f}(\mathbf{x}_0) = \mathbf{f}'(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0)$$

$$[\mathbf{f}'(\mathbf{x}_0)]^{-\mathbf{1}}[\mathbf{y} - \mathbf{f}(\mathbf{x}_0)] = \mathbf{x} - \mathbf{x}_0$$

$$\mathbf{x} = \mathbf{x}_0 + [\mathbf{f}'(\mathbf{x}_0)]^{-\mathbf{1}}[\mathbf{y} - \mathbf{f}(\mathbf{x}_0)]$$

- This last expression can be used as an iterative formula

$$\mathbf{x}_{i+1} = \mathbf{x}_i + [\mathbf{f}'(\mathbf{x}_i)]^{-\mathbf{1}}[\mathbf{y} - \mathbf{f}(\mathbf{x}_i)]$$

- The derivative term on the right-hand side of (8) is an $N \times N$ matrix
  - The ***Jacobian*** matrix, $\mathbf{J}$

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \mathbf{J}_i^{-\mathbf{1}}[\mathbf{y} - \mathbf{f}(\mathbf{x}_i)] \qquad (9)$$

The Jacobian Matrix

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \mathbf{J}_i^{-1}[\mathbf{y} - \mathbf{f}(\mathbf{x}_i)] \qquad (9)$$

- ***Jacobian matrix***
  - $N \times N$ matrix of partial derivatives for $\mathbf{f}(\mathbf{x})$
  - Evaluated at the current value of $\mathbf{x}$, $\mathbf{x}_i$

$$\mathbf{J}_i = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_N} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_N} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_N}{\partial x_1} & \frac{\partial f_N}{\partial x_2} & \cdots & \frac{\partial f_N}{\partial x_N} \end{bmatrix}_{\mathbf{x}=\mathbf{x}_i}$$

Newton-Raphson Method

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \mathbf{J}_i^{-1}[\mathbf{y} - \mathbf{f}(\mathbf{x}_i)] \tag{9}$$

- We could iterate (9) until convergence or a maximum number of iterations is reached
  - Requires ***inversion*** of the Jacobian matrix
    - Computationally expensive and error prone
- Instead, go back to the Taylor series approximation

$$\mathbf{y} = \mathbf{f}(\mathbf{x}_i) + \mathbf{J}_i(\mathbf{x}_{i+1} - \mathbf{x}_i)$$

$$\mathbf{y} - \mathbf{f}(\mathbf{x}_i) = \mathbf{J}_i(\mathbf{x}_{i+1} - \mathbf{x}_i) \tag{10}$$

  - Left side of (21) represents a difference between the known and approximated outputs
  - Right side represents an increment of the approximation for $\mathbf{x}$

$$\Delta\mathbf{y}_i = \mathbf{J}_i\Delta\mathbf{x}_i \tag{11}$$

Newton-Raphson Method

$$\Delta\mathbf{y}_i = \mathbf{J}_i\Delta\mathbf{x}_i \qquad (12)$$

- On each iteration:
  - Compute $\Delta\mathbf{y}_i$ and $\mathbf{J}_i$
  - Solve for $\Delta\mathbf{x}_i$ using ***Gaussian elimination***
    - Matrix inversion not required
    - Computationally robust
  - Update $\mathbf{x}$

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \Delta\mathbf{x}_i \qquad (13)$$

Newton-Raphson – Example

- Apply Newton-Raphson to solve the following system of nonlinear equations

$$\mathbf{f}(\mathbf{x}) = \mathbf{y}$$

$$\begin{bmatrix} x_1^2 + 3x_2 \\ x_1 x_2 \end{bmatrix} = \begin{bmatrix} 21 \\ 12 \end{bmatrix}$$

  - Initial condition: $\mathbf{x}_0 = [1 \quad 2]^T$
  - Stopping criterion: $\varepsilon_s = 1 \times 10^{-6}$
  - Jacobian matrix

$$\mathbf{J}_i = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{bmatrix}_{\mathbf{x}=\mathbf{x}_i} = \begin{bmatrix} 2x_{1,i} & 3 \\ x_{2,i} & x_{1,i} \end{bmatrix}$$

Newton-Raphson – Example

$$\Delta \mathbf{y}_i = \mathbf{J}_i \Delta \mathbf{x}_i \quad (12)$$

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \Delta \mathbf{x}_i \quad (13)$$

- For iteration i:
  - Compute $\Delta \mathbf{y}_i$ and $\mathbf{J}_i$
  - Solve (12) for $\Delta \mathbf{x}_i$
  - Update $\mathbf{x}$ using (13)

Newton-Raphson – Example

- $\underline{i = 0}$:

$$\Delta\mathbf{y}_0 = \mathbf{y} - \mathbf{f}(\mathbf{x}_0) = \begin{bmatrix} 21 \\ 12 \end{bmatrix} - \begin{bmatrix} 7 \\ 2 \end{bmatrix} = \begin{bmatrix} 14 \\ 10 \end{bmatrix}$$

$$\mathbf{J}_0 = \begin{bmatrix} 2x_{1,0} & 3 \\ x_{2,0} & x_{1,0} \end{bmatrix} = \begin{bmatrix} 2 & 3 \\ 2 & 1 \end{bmatrix}$$

$$\Delta\mathbf{x}_0 = \begin{bmatrix} 4 \\ 2 \end{bmatrix}$$

$$\mathbf{x}_1 = \mathbf{x}_0 + \Delta\mathbf{x}_0 = \begin{bmatrix} 1 \\ 2 \end{bmatrix} + \begin{bmatrix} 4 \\ 2 \end{bmatrix} = \begin{bmatrix} 5 \\ 4 \end{bmatrix}$$

$$\varepsilon_1 = \max\left(\left|\frac{x_{k,1} - x_{k,0}}{x_{k,1}}\right|\right), \qquad k = 1 \dots N$$

$$\boxed{x_1 = \begin{bmatrix} 5 \\ 4 \end{bmatrix}, \quad \varepsilon_1 = 0.8}$$

Newton-Raphson – Example

- $\underline{i = 1}$:

$$\Delta\mathbf{y}_1 = \mathbf{y} - \mathbf{f}(\mathbf{x}_1) = \begin{bmatrix} 21 \\ 12 \end{bmatrix} - \begin{bmatrix} 37 \\ 20 \end{bmatrix} = \begin{bmatrix} -16 \\ -8 \end{bmatrix}$$

$$\mathbf{J}_1 = \begin{bmatrix} 2x_{1,1} & 3 \\ x_{2,1} & x_{1,1} \end{bmatrix} = \begin{bmatrix} 10 & 3 \\ 4 & 5 \end{bmatrix}$$

$$\Delta\mathbf{x}_1 = \begin{bmatrix} -1.474 \\ -0.421 \end{bmatrix}$$

$$\mathbf{x}_2 = \mathbf{x}_1 + \Delta\mathbf{x}_1 = \begin{bmatrix} 5 \\ 4 \end{bmatrix} + \begin{bmatrix} -1.474 \\ -0.421 \end{bmatrix} = \begin{bmatrix} 3.526 \\ 3.579 \end{bmatrix}$$

$$\varepsilon_2 = \max\left(\left|\frac{x_{k,2} - x_{k,1}}{x_{k,1}}\right|\right), \qquad k = 1 \ldots N$$

$$\boxed{x_2 = \begin{bmatrix} 3.526 \\ 3.579 \end{bmatrix}, \quad \varepsilon_2 = 0.418}$$

Newton-Raphson – Example

- $\underline{i = 2}$:

$$\Delta\mathbf{y}_2 = \mathbf{y} - \mathbf{f}(\mathbf{x}_2) = \begin{bmatrix} 21 \\ 12 \end{bmatrix} - \begin{bmatrix} 23.172 \\ 12.621 \end{bmatrix} = \begin{bmatrix} -2.172 \\ -0.621 \end{bmatrix}$$

$$\mathbf{J}_2 = \begin{bmatrix} 2x_{1,2} & 3 \\ x_{2,2} & x_{1,2} \end{bmatrix} = \begin{bmatrix} 7.053 & 3 \\ 3.579 & 3.526 \end{bmatrix}$$

$$\Delta\mathbf{x}_2 = \begin{bmatrix} -0.410 \\ 0.240 \end{bmatrix}$$

$$\mathbf{x}_3 = \mathbf{x}_2 + \Delta\mathbf{x}_2 = \begin{bmatrix} 3.526 \\ 3.579 \end{bmatrix} + \begin{bmatrix} -0.410 \\ 0.240 \end{bmatrix} = \begin{bmatrix} 3.116 \\ 3.819 \end{bmatrix}$$

$$\varepsilon_3 = \max\left(\left|\frac{x_{k,3} - x_{k,2}}{x_{k,2}}\right|\right), \qquad k = 1 \dots N$$

$$\boxed{x_3 = \begin{bmatrix} 3.116 \\ 3.819 \end{bmatrix}, \quad \varepsilon_3 = 0.132}$$

Newton-Raphson – Example

- $\underline{i = 6}$:

$$\Delta\mathbf{y}_6 = \mathbf{y} - \mathbf{f}(\mathbf{x}_6) = \begin{bmatrix} 21 \\ 12 \end{bmatrix} - \begin{bmatrix} 21.000 \\ 12.000 \end{bmatrix} = \begin{bmatrix} -0.527 \times 10^{-7} \\ 0.926 \times 10^{-7} \end{bmatrix}$$

$$\mathbf{J}_6 = \begin{bmatrix} 2x_{1,6} & 3 \\ x_{2,6} & x_{1,6} \end{bmatrix} = \begin{bmatrix} 6.000 & 3 \\ 4.000 & 3.000 \end{bmatrix}$$

$$\Delta\mathbf{x}_6 = \begin{bmatrix} -0.073 \times 10^{-6} \\ 0.128 \times 10^{-6} \end{bmatrix}$$

$$\mathbf{x}_7 = \mathbf{x}_6 + \Delta\mathbf{x}_6 = \begin{bmatrix} 3.000 \\ 4.000 \end{bmatrix} + \begin{bmatrix} -0.073 \times 10^{-6} \\ 0.128 \times 10^{-6} \end{bmatrix} = \begin{bmatrix} 3.000 \\ 4.000 \end{bmatrix}$$

$$\varepsilon_7 = \max\left(\left|\frac{x_{k,7} - x_{k,6}}{x_{k,6}}\right|\right), \qquad k = 1 \dots N$$

$$\boxed{x_7 = \begin{bmatrix} 3.000 \\ 4.000 \end{bmatrix}, \quad \varepsilon_7 = 31.9 \times 10^{-9}}$$

Newton-Raphson – Python Code

- Define the system of equations

```python
f = lambda x: np.array([x[0]**2 + 3*x[1], x[0]*x[1]])
y = np.array([21, 12])
```

- Initialize **x**

```python
x0 = np.array([1, 2])
x = x0
```

- Set up solution parameters

```python
reltol = 1e-6
max_iter = 1000
eps = 1
iter = 0
```

Newton-Raphson – Python Code

- Iterate:
  - Compute $\Delta\mathbf{y}_i$ and $\mathbf{J}_i$
  - Solve for $\Delta\mathbf{x}_i$
  - Update $\mathbf{x}$

```python
while(iter < max_iter) and (eps > reltol):

    J = np.array([[2*x[0], 3], [x[1], x[0]]])
    x_old = x

    # calculate output error term
    Dy = y - f(x_old)

    # use Gaussian elimination to solve for increment to x
    Dx = np.linalg.solve(J, Dy)
    x = x_old + Dx

    eps = np.max(abs((x - x_old)/x))

    iter = iter + 1
```