Computational practicum: Q1 – Lecture 3
Solving systems of equations in $\mathbb{R}^n$

Zoë J.G. Gromotka, Ferdinand Grozema, Artur M. Schweidtmann, Tanuj Karia

With support from Sophia Rupprecht and Giacomo Lastrucci

Computational Practicum
Dept. Chemical Engineering
Delft University of Technology

TU Delft

Process Intelligence RESEARCH

Delft Institute of Applied Mathematics

Wednesday, September 17th, 2025

Recap last lecture

Solution of nonlinear equations (e.g., Van der Waals EoS)

- Numerical techniques:
  - bisection method — $see\ algorithm\ in\ Lecture\ 2$
  - fixed-point iterative method — $f(x) = 0 \xrightarrow{rewrite} g(x) = x \xrightarrow{iterate} x^{k+1} = g\left(x^k\right)$
  - Newton-Raphson method — $f(x) = 0 \xrightarrow{iterate} x^{k+1} = f\left(x^k\right) - \frac{f(x^k)}{f'(x^k)}$

- Python built-in functions: (different advanced methods used in the background)
  - scipy.optimize.fsolve
  - scipy.optimize.root

Learning goals of this lecture

After successfully completing this lecture, you are able to….

- apply basic linear algebra operations and the respective python commands.
- explain the Jacobi and Gauss-Seidel methods for approximating the solution of linear systems of equation.
- explain the Newton-Raphson method for approximating the solution of nonlinear systems of equation.
- implement iterative methods to approximate the solutions of linear and nonlinear systems of equation in Python.

Lecture outline

- Algebra refresher
  - Vectors, vector operations, and vector norms
  - Matrices and matrix operations
  - Determinant
  - Eigenvalues and eigenvectors
  - Matrix rank
  - Matrix norms
- Approximating solutions of linear systems of equation
- Approximating solutions of nonlinear systems of equation

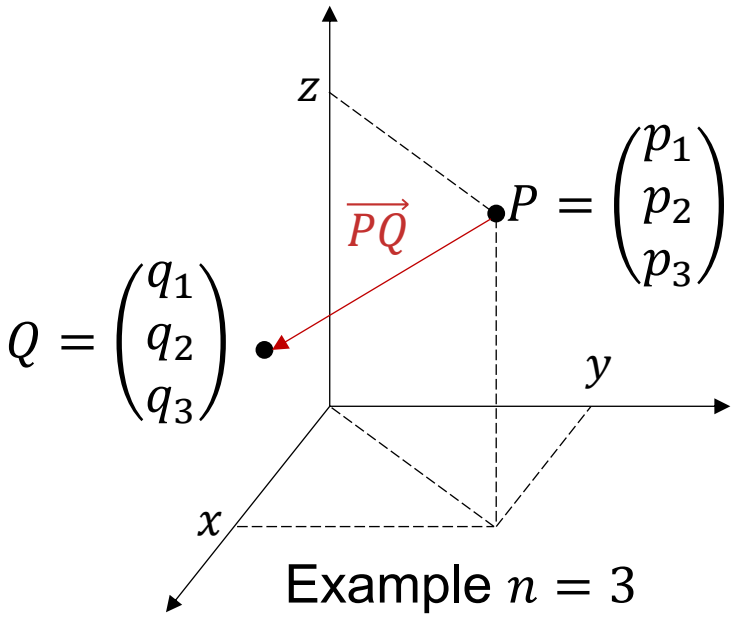
Vectors in $\mathbb{R}^n$

- Point in n-dimensional space:

$$\mathbb{R}^n := \left\{ \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \middle| x_i \in \mathbb{R}, 1 \leq i \leq n \right\}$$

- Euclidian vector: $\overrightarrow{PQ} = \begin{pmatrix} q_1 - p_1 \\ q_2 - p_2 \\ q_3 - p_3 \end{pmatrix}$

→here: $\overrightarrow{PQ} \in \mathbb{R}^3$ `numpy.array([x1, x2, ..., xn])`



Example $n = 3$

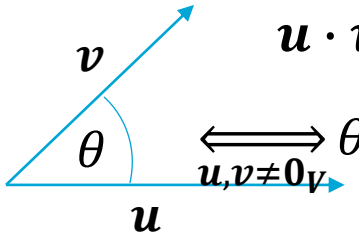
Vector operations: Dot and cross product

Let V be a vector space over the field K (we assume $K = \mathbb{R}$) and the vectors $\boldsymbol{u}, \boldsymbol{v}$ elements of V.

- **Dot product**:
  - $\boldsymbol{u} \cdot \boldsymbol{v} = \sum_{i=1}^{n} u_i v_i = u_1 v_1 + u_2 v_2 + \ \ldots + u_n v_n$
  - $\begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} \cdot \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} u_1 & u_2 & u_3 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = u_1 v_1 + u_2 v_2 + u_3 v_3$

```
numpy.dot(u,v)
```

```
numpy.vdot(u,v)
```

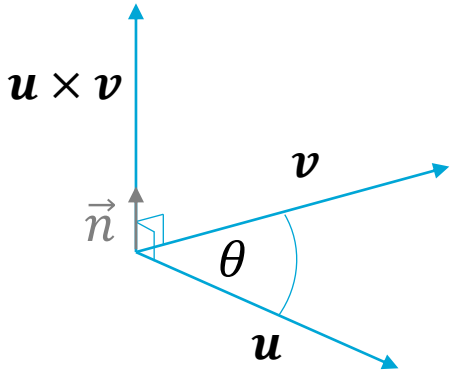


$$\boldsymbol{u} \cdot \boldsymbol{v} = \|\boldsymbol{u}\|\|\boldsymbol{v}\| \cos\theta$$

$$\underset{\boldsymbol{u},\boldsymbol{v} \neq \boldsymbol{0}_V}{\Longleftrightarrow} \theta = \arccos\left(\frac{\boldsymbol{u} \cdot \boldsymbol{v}}{\|\boldsymbol{u}\|\|\boldsymbol{v}\|}\right)$$

- **Cross product**:
  - $\boldsymbol{u} \times \boldsymbol{v} = \|\boldsymbol{u}\|\|\boldsymbol{v}\| \sin\theta \, \boldsymbol{n}$
  - $\begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} \times \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} u_2 v_3 - u_3 v_2 \\ u_3 v_1 - u_1 v_3 \\ u_1 v_3 - u_3 v_1 \end{bmatrix}$

```
numpy.cross(u,v)
```



The magnitude of the cross product equals the area of a parallelogram.

Example dot product

Consider methane combustion: $CH_4^{(g)} + 2\ O_2^{(g)} \rightarrow CO_2^{(g)} + 2\ H_2O^{(g)}$

Question: What is the standard reaction enthalpy given standard enthalpy of formation for each component?

```
    import numpy as np

    H_f_s_CH4 = -74.8 #kJ/mol - Enthalpy of formation
    H_f_s_O2 = 0 #kJ/mol
    H_f_s_CO2 = -383.509 #kJ/mol
    H_f_s_H2O = -241.818 #kJ/mol

    reaction_enthalpies = np.array([H_f_s_CH4, H_f_s_O2, H_f_s_CO2, H_f_s_H2O])
    stoechiometric_coeff = np.array([-1,-2,1,2])

    standard_reaction_enthalpy = np.dot(stoechiometric_coeff, reaction_enthalpies)

    print("The standard reaction enthalpy of Methane combustion is %f
    kJ/mol."%standard_reaction_enthalpy)
... The standard reaction enthalpy of Methane combustion is -792.345000 kJ/mol.
```

Exothermic

Linear dependency

Let V be a vector space over the field K (we assume $K = \mathbb{R}$). If the vectors $\boldsymbol{v_1}, \boldsymbol{v_2}, \ldots, \boldsymbol{v_n}$ are elements of V and $\alpha_1, \alpha_2, \ldots, \alpha_n$ scalars of $\mathbb{R}$,

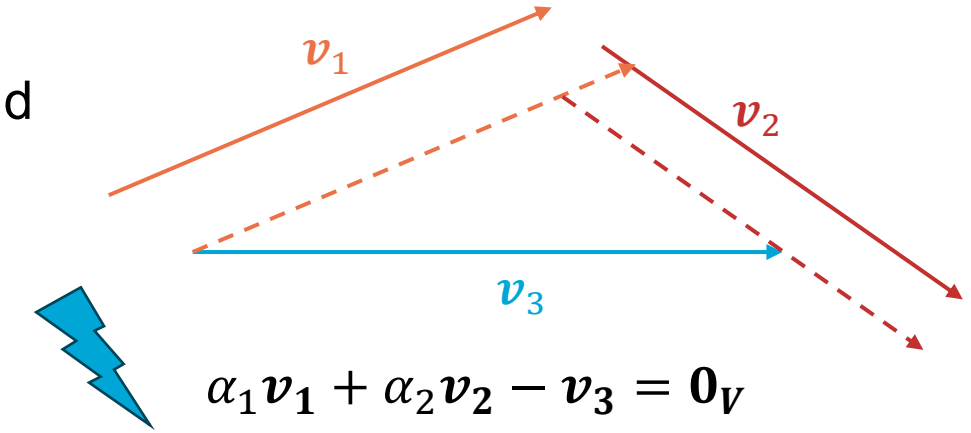
- a linear combination (or superposition) is the sum

$$\sum_{i=1}^{n} \alpha_i \boldsymbol{v}_i \,.$$

- the vectors $\boldsymbol{v_1}, \ldots, \boldsymbol{v_n}$ are <u>linearly independent</u> if

$$\sum_{i=1}^{n} \alpha_i \boldsymbol{v}_i = 0 \quad \Rightarrow \quad \alpha_1 = \alpha_2 = \cdots = \alpha_n = 0$$

holds.




```
    import numpy as np

    v1 = np.array([1,2,2])
    v2 = np.array([3,6,5])
    v3 = np.array([2,4,3])

    check = v1 - v2 + v3
    print(check)
... [0 0 0]
```

Vector norms

- Let V be a vector space over $\mathbb{R}$. The function $\|\cdot\|: V \to \mathbb{R}$ is a norm if for all $\boldsymbol{x}, \boldsymbol{y} \in V$ and $\lambda \in \mathbb{R}$:

  (1) $\|\boldsymbol{x}\| \geq 0$ and $\|\boldsymbol{x}\| = 0 \Leftrightarrow \boldsymbol{x} = \boldsymbol{0}$,

  (2) $\|\lambda \boldsymbol{x}\| = |\lambda| \|\boldsymbol{x}\|$,

  (3) $\|\boldsymbol{x} + \boldsymbol{y}\| \leq \|\boldsymbol{x}\| + \|\boldsymbol{y}\|$.

- Vector norms:
  - $\|\boldsymbol{x}\|_2 := \sqrt{\sum_{i=1}^{n} |x_i|^2}$ → Euclidian norm
  - $\|\boldsymbol{x}\|_1 := \sum_{i=1}^{n} |x_i|$ → L_1-norm
  - $\|\boldsymbol{x}\|_\infty := \max_{i=1,\ldots,n} |x_i|$ → max-norm (a.k.a. inf-norm)

```
    import numpy as np
    from numpy import linalg as LA
    a = np.array([1,2,4])
    print(a)
    print(f"Euclidian-norm: {LA.norm(a,ord = 2)}")
    print(f"1-norm: {LA.norm(a,ord = 1)}")
    print(f"inf-norm: {LA.norm(a,np.inf)}")
... [1 2 4]
    Euclidian-norm: 4.582575694955840
    1-norm: 7.0
    inf-norm: 4.0
```

Lecture outline

- Algebra refresher
  - Vectors, vector operations, and vector norms
  - Matrices and matrix operations
  - Determinant
  - Eigenvalues and eigenvectors
  - Matrix rank
  - Matrix norms
- Approximating solutions of linear systems of equation
- Approximating solutions of nonlinear systems of equation

Matrices

$\boldsymbol{A} \in K^{m \times n}$ is a $(m \times n)$-Matrix $\boldsymbol{A} = \begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \dots & a_{mn} \end{pmatrix}$ with entries $a_{ij} \in K$

- Row index i, $1 \leq i \leq m$
- Column index j, $1 \leq j \leq n$

First row

```
A = numpy.array([[a11, a12], [a21, a22]])
```

- Square matrix if $m = n$ s.t. $\boldsymbol{A} \in K^{n \times n}$: $\boldsymbol{A} = \begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nn} \end{pmatrix}$
- Identity matrix $\boldsymbol{I}_n = \begin{pmatrix} 1 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & 1 \end{pmatrix}$

```
I = numpy.identity(n)
```

Be careful!

Special cases of ‘matrices’ in Numpy are not the same as arrays (normal column vectors)

- “Row vector” if $m = 1$ s.t. $\boldsymbol{v} \in \mathbb{R}^{1\times n}$: $\boldsymbol{v} = (v_1 \quad \dots \quad v_n)$

```
v = numpy.array([[v1, v2]])
```

- “Column vector” if $n = 1$ s.t. $\boldsymbol{u} \in \mathbb{R}^{m\times 1}$: $\boldsymbol{u} = \begin{pmatrix} u_1 \\ \vdots \\ u_m \end{pmatrix}$

```
u = numpy.array([[u1], [u2]])
```

```
regular_vector = np.array([2,4,6])
column_vector_2d = np.array([[2],[4],[6]])
row_vector_2d = np.array([[2,4,6]])

print(f"What are the shapes?:\nregular_vector: {regular_vector.shape}\n\
column_vector_2d: {column_vector_2d.shape}\nrow_vector_2d: {row_vector_2d.shape}\n")
print(f"How to access the entry 4:\nregular_vector: {regular_vector[1]}\n\
column_vector_2d: {column_vector_2d[1][0]}\n\
row_vector_2d: {row_vector_2d[0][1]}\n")
print(f"What happens if we access the 0-th dimension?\n\
column_vector_2d: {column_vector_2d[0]}\nrow_vector_2d: {row_vector_2d[0]}\n")
```

Example in numpy

```
    regular_vector = np.array([2,4,6])
    column_vector_2d = np.array([[2],[4],[6]])
    row_vector_2d = np.array([[2,4,6]])

    print(f"What are the shapes?:\nregular_vector: {regular_vector.shape}\n\
    column_vector_2d: {column_vector_2d.shape}\nrow_vector_2d: {row_vector_2d.shape}\n")
    print(f"How to access the entry ‘4’:\nregular_vector: {regular_vector[1]}\n\
    column_vector_2d: {column_vector_2d[1][0]}\n\
    row_vector_2d: {row_vector_2d[0][1]}\n")
    print(f"What happens if we access the 0-th dimension?\n\
    column_vector_2d: {column_vector_2d[0]}\nrow_vector_2d: {row_vector_2d[0]}\n")
```

```
... What are the shapes?:
    regular_vector: (3,)
    column_vector_2d: (3, 1)
    row_vector_2d: (1, 3)
```

```
... How to access the entry ‘4’:
    regular_vector: 4
    column_vector_2d: 4
    row_vector_2d: 4
```

```
... What happens if we access the 0-th dimension?
    column_vector_2d: [2]
    row_vector_2d: [2 4 6]
```

Matrix operations

For $\boldsymbol{A} = (a_{ij}) \in K^{m \times n}$ and $\boldsymbol{B} = (b_{ij}) \in K^{m \times n}$, $\alpha \in K$, the following operations can be applied:

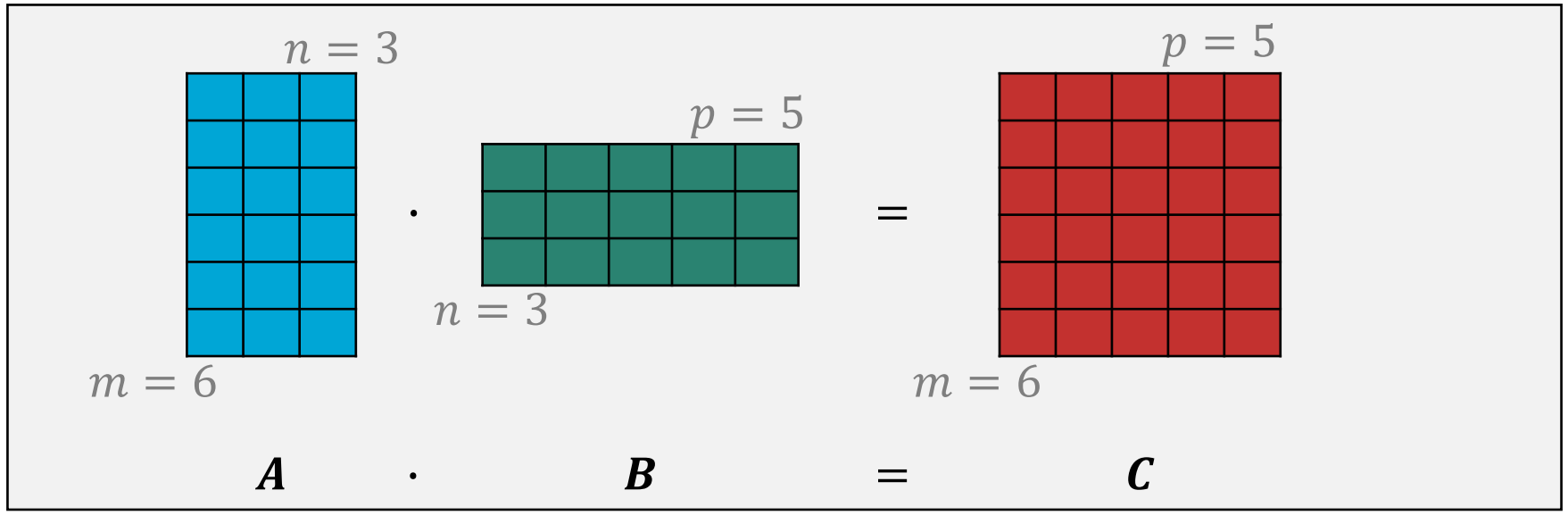
- **Addition:** $\mathbf{C} = \boldsymbol{A} + \boldsymbol{B} \rightarrow c_{ij} = a_{ij} + b_{ij}\ \forall_{i,j}$ `np.add(x1,x2)` `+ (for np.ndarray objects)`
- **Scalar multiplication**: $\alpha \cdot \boldsymbol{A} \coloneqq (\alpha \cdot a_{ij})$ `numpy.multiply(a,A)` `* (for numpy.ndarray objects)`
- **Transposition**: $(\boldsymbol{A}^T)_{ij} = \boldsymbol{A}_{ji}$ `numpy.transpose(A)` `A.T (for numpy.ndarray objects)`
  - $(\alpha \cdot \boldsymbol{A})^T = \alpha \cdot (A^T)$
  - $(\boldsymbol{A} + \boldsymbol{B})^T = \boldsymbol{A}^T + \boldsymbol{B}^T$
  - $(\boldsymbol{A}^T)^T = \boldsymbol{A}$
- **Matrix vector multiplication**: $\boldsymbol{c} = \boldsymbol{A} \cdot \boldsymbol{v}$ `numpy.matmul(A,v)`

Matrix multiplication

If $\boldsymbol{A}$ is a $(m \times n)$-matrix and $\boldsymbol{B}$ is a $(n \times p)$-matrix, the matrix product $\boldsymbol{C} = \boldsymbol{AB}$ yields a $(m \times p)$-matrix $\boldsymbol{C}$ such that

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj}\,,$$

i.e., the c_{ij}-th entry of $\boldsymbol{C}$ is the **dot product** of the i-th row of $\boldsymbol{A}$ and the j-th column of $\boldsymbol{B}$.



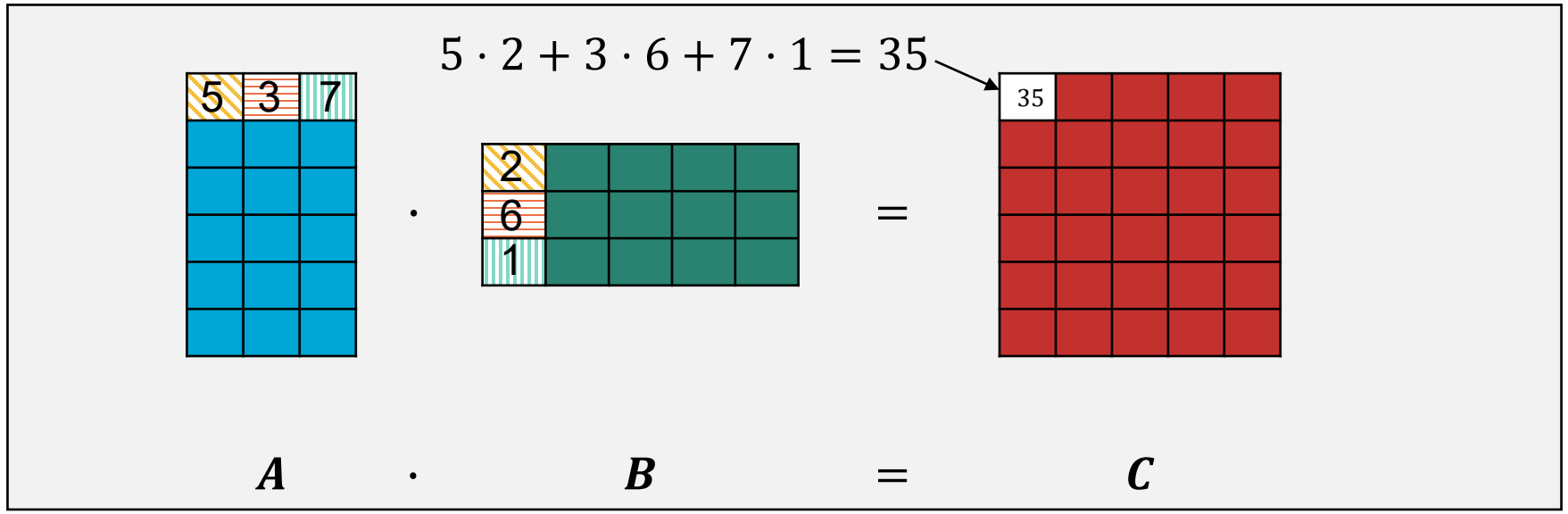
`numpy.matmul(A,B)` `A@B`

Matrix multiplication

If $\boldsymbol{A}$ is a $(m \times n)$-matrix and $\boldsymbol{B}$ is a $(n \times p)$-matrix, the matrix product $\boldsymbol{C} = \boldsymbol{AB}$ yields a $(m \times p)$-matrix $\boldsymbol{C}$ such that

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj}\,,$$

i.e., the c_{ij}-th entry of $\boldsymbol{C}$ is the **dot product** of the i-th row of $\boldsymbol{A}$ and the j-th column of $\boldsymbol{B}$.



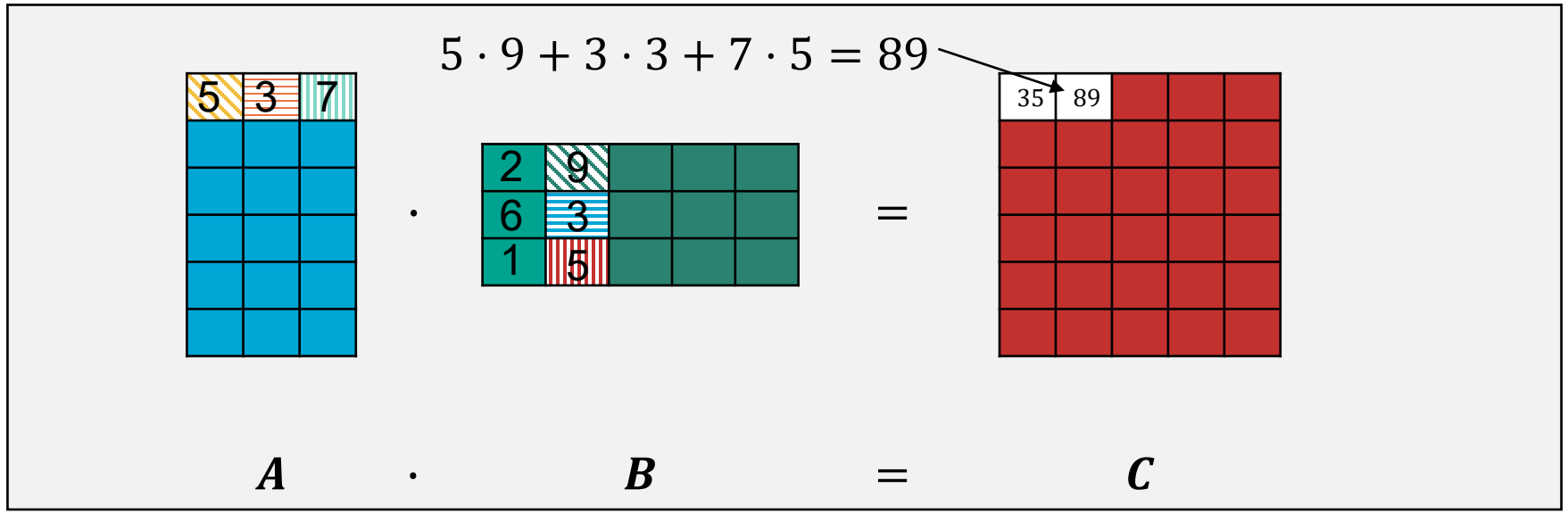
`numpy.matmul(A,B)` `A@B`

Matrix multiplication

If $\boldsymbol{A}$ is a $(m \times n)$-matrix and $\boldsymbol{B}$ is a $(n \times p)$-matrix, the matrix product $\boldsymbol{C} = \boldsymbol{AB}$ yields a $(m \times p)$-matrix $\boldsymbol{C}$ such that

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj}\,,$$

i.e., the c_{ij}-th entry of $\boldsymbol{C}$ is the **dot product** of the i-th row of $\boldsymbol{A}$ and the j-th column of $\boldsymbol{B}$.



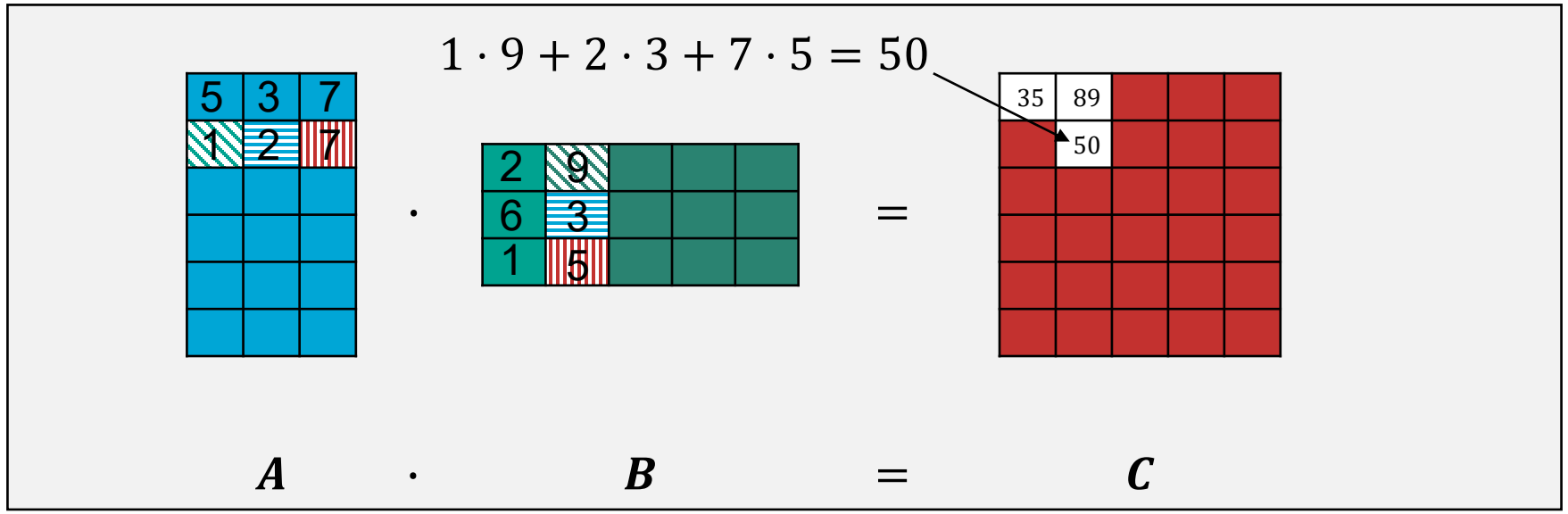
`numpy.matmul(A,B)` `A@B`

Matrix multiplication

If $\boldsymbol{A}$ is a $(m \times n)$-matrix and $\boldsymbol{B}$ is a $(n \times p)$-matrix, the matrix product $\boldsymbol{C} = \boldsymbol{AB}$ yields a $(m \times p)$-matrix $\boldsymbol{C}$ such that

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj}\,,$$

i.e., the c_{ij}-th entry of $\boldsymbol{C}$ is the **dot product** of the i-th row of $\boldsymbol{A}$ and the j-th column of $\boldsymbol{B}$.



`numpy.matmul(A,B)` `A@B`

Important properties of matrix multiplication

- Non-commutativity: $\boldsymbol{AB} \neq \boldsymbol{BA}$
- Distributivity: $\boldsymbol{A}(\boldsymbol{B}+\boldsymbol{C}) = \boldsymbol{AB}+\boldsymbol{AC}$ and $(\boldsymbol{B}+\boldsymbol{C})\boldsymbol{A} = \boldsymbol{BA}+\boldsymbol{CA}$

Lecture outline

- Algebra refresher
  - Vectors, vector operations, and vector norms
  - Matrices and matrix operations
  - Determinant
  - Eigenvalues and eigenvectors
  - Matrix rank
  - Matrix norms
- Approximating solutions of linear systems of equation
- Approximating solutions of nonlinear systems of equation

Determinant of a matrix

If you don't remember this, practise it! *here*

- Recall determinant of a square 2×2 matrix:

$$\det\left(\begin{bmatrix} 5 & 7 \\ 4 & 9 \end{bmatrix}\right) = 5 \cdot 9 - 4 \cdot 7 = 17$$

```
numpy.linalg.det(A)
```

- How does it work for a larger matrix?

Determinant cont'd

If you don't remember this, practise it! *here*

Let $\boldsymbol{A}$ be a square $n \times n$ matrix.

- Use the **cofactor expansion rule**: $\det(\boldsymbol{A}) = \sum_{j=1}^{n} a_{ij} C_{ij}$ for any $i = 1,2, \dots, n$
- Checkerboard pattern: $\begin{bmatrix} + & - & + & - \\ - & + & - & + \\ + & - & + & - \\ - & + & - & + \end{bmatrix}$ → $C_{ij} = (-1)^{i+j} \det(\boldsymbol{A}_{ij})$

$\boldsymbol{A}_{ij}$ obtained from deleting i–th row and j–th column of $\boldsymbol{A}$

- Example: Let $\boldsymbol{A}$ be a square 3×3 matrix $\boldsymbol{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$. The determinant evaluates to

$$\det(\boldsymbol{A}) = a_{11} C_{11} + a_{12} C_{12} + a_{13} C_{13}$$

$$\det(\boldsymbol{A}) = a_{11} \det \begin{pmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{pmatrix} - a_{12} \det \begin{pmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{pmatrix} + a_{13} \det \begin{pmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{pmatrix}$$

$$\det(\boldsymbol{A}) = a_{11}(a_{22}a_{33} - a_{23}a_{32}) - a_{12}(a_{21}a_{33} - a_{23}a_{31}) + a_{13}(a_{21}a_{32} - a_{22}a_{31})$$

Lecture outline

- Algebra refresher
  - Vectors, vector operations, and vector norms
  - Matrices and matrix operations
  - Determinant
  - Eigenvalues and eigenvectors
  - Matrix rank
  - Matrix norms
- Approximating solutions of linear systems of equation
- Approximating solutions of nonlinear systems of equation

Eigenvalues and eigenvectors

- An **eigenvector** $\boldsymbol{v}$ of a matrix $\boldsymbol{A} \in K^{n \times n}$ is the solution to the equation $\boldsymbol{A} \cdot \boldsymbol{v} = \lambda \boldsymbol{v}$.

  Graphically, this implies that the transformation performed by multiplying $\boldsymbol{A}$ and $\boldsymbol{v}$ is equal to stretching $\boldsymbol{v}$ by a constant factor, i.e., resulting in a multiple of $\boldsymbol{v}$ itself.

- The **eigenvectors** $\boldsymbol{v_1}, \ldots, \boldsymbol{v_n}$ of matrix $\boldsymbol{A} \in K^{n \times n}$ are computed by solving

$$(\boldsymbol{A} - \lambda_i \boldsymbol{I_n}) \cdot \boldsymbol{v_i} = \boldsymbol{0}, \qquad i = 1, \ldots, n.$$

- The above equation has solutions $\boldsymbol{v_i}$ different from the null vector if (and only if) the factor $(\boldsymbol{A} - \lambda_i \boldsymbol{I_n})$ is zero.

- The roots $\lambda_1, \lambda_2, \ldots, \lambda_n \in \mathbb{C}$ of the resulting **characteristic polynomial**

$$\det(\boldsymbol{A} - \lambda \boldsymbol{I}_n) = 0$$

  are referred to as the **eigenvalues** of matrix $\boldsymbol{A}$.

```
λ, v = np.linalg.eig(A)
```

Example: Schrödinger equation in matrix form

$$\begin{pmatrix} H_{11} & H_{12} & \dots & H_{1n} \\ H_{12} & H_{22} & \dots & H_{2n} \\ \vdots & & \ddots & \vdots \\ H_{n1} & H_{n2} & \dots & H_{nn} \end{pmatrix} \cdot \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix} = E \cdot \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix}$$

$$\Psi = c_1\phi_1 + c_2\phi_2$$

$$H_{ij} = \langle \phi_i | \hat{H} | \phi_j \rangle$$

$$\mathbf{HC} = E\mathbf{C}$$

H : Hamiltonian matrix
C : vector describing the wavefunction
E : energy belonging to the wavefunction

Example: Schrödinger equation in matrix form

Example: Hückel model for ethylene:

$$\begin{vmatrix} \alpha - E & \beta \\ \beta & \alpha - E \end{vmatrix} = 0$$

$$\Psi = c_1\phi_1 + c_2\phi_2$$

$$H_{ij} = \langle \phi_i | \hat{H} | \phi_j \rangle$$

Expand determinant gives characteristic polynomial:

$$\begin{vmatrix} \alpha - E & \beta \\ \beta & \alpha - E \end{vmatrix} = (\alpha - E)^2 - \beta^2 = \alpha^2 - 2\alpha E + E^2 - \beta^2 = 0$$

Example: Schrödinger equation in matrix form

Characteristic polynomial:

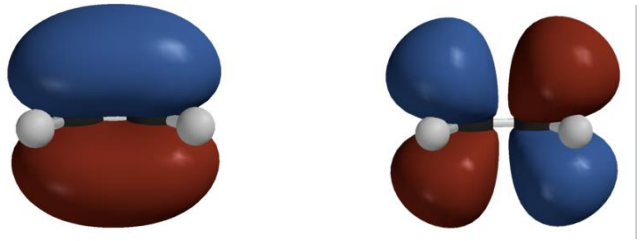
$$\alpha^2 - 2\alpha E + E^2 - \beta^2 = 0$$

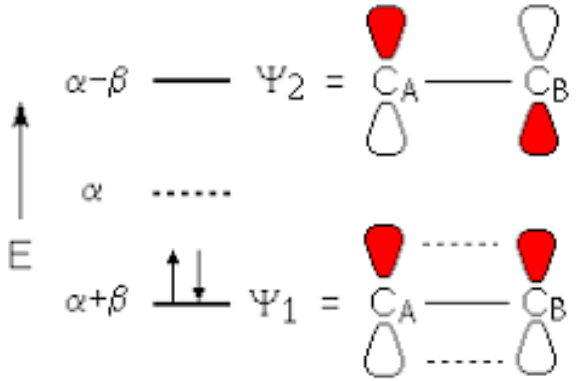
$$\Psi = c_1\phi_1 + c_2\phi_2$$

$$H_{ij} = \langle \phi_i | \hat{H} | \phi_j \rangle$$

Solution:

$$E = \alpha \pm \beta$$

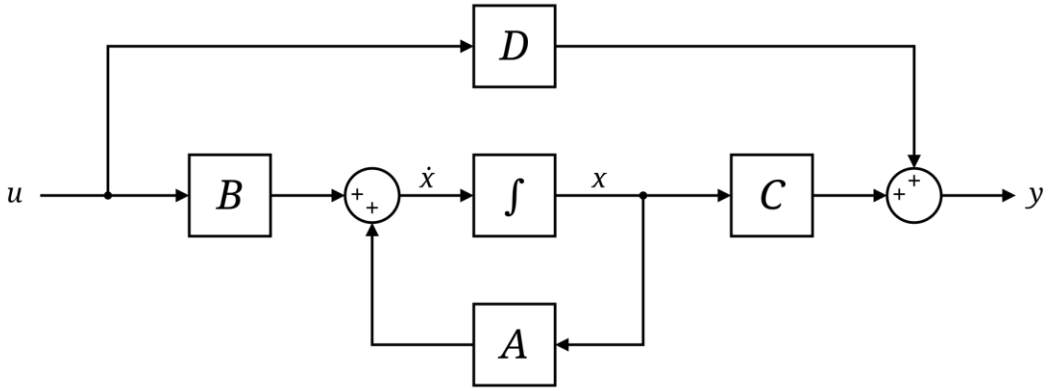




Excurse: State-space formulation (c.f. Process dynamics and control course)

Continuous time-invariant state-space model

$$\dot{\boldsymbol{x}}(t) = \boldsymbol{A}\boldsymbol{x}(t) + \boldsymbol{B}\boldsymbol{u}(t)$$
$$\boldsymbol{y}(t) = \boldsymbol{C}\boldsymbol{x}(t) + \boldsymbol{D}\boldsymbol{u}(t)$$

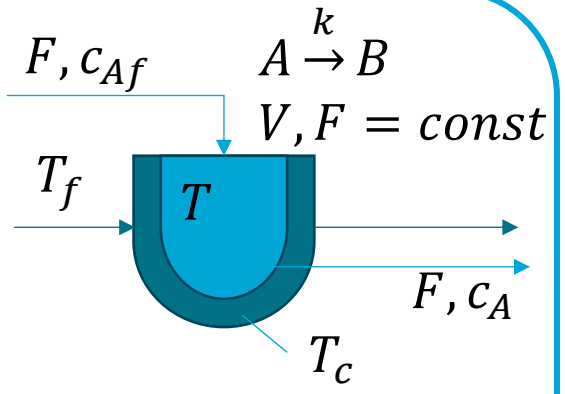


with state vector $\boldsymbol{x}$, control inputs $\boldsymbol{u}$, outputs $\boldsymbol{y}$

System matrix $\boldsymbol{A}$, input matrix $\boldsymbol{B}$, output matrix $\boldsymbol{C}$, feedforward matrix $\boldsymbol{D}$

Figure: https://en.wikipedia.org/wiki/State-space_representation

Example: Stability of CSTR system matrix



$$\frac{dc_A(t)}{dt} = \frac{F}{V}\left(c_{Af}(t) - c_A(t)\right) - r(t)$$

$$\frac{dT(t)}{dt} = \frac{F}{V}\left(T_f(t) - T(t)\right) - \frac{\Delta H}{\rho c_p} r(t) - \frac{UA}{\rho c_p V}\left(T(t) - T_c(t)\right)$$

$$r(t) = k_0 e^{\frac{-E}{RT(t)}} c_A(t)$$

linearization (′) at steady state condition

$$\dot{\boldsymbol{x}}(t) = \boldsymbol{A}\boldsymbol{x}(t) + \boldsymbol{B}\boldsymbol{u}(t)$$

$$\boldsymbol{y}(t) = \boldsymbol{C}\boldsymbol{x}(t) + \boldsymbol{D}\boldsymbol{u}(t)$$

$$\begin{bmatrix} \frac{dc_A{}'}{dt} \\ \frac{dT'}{dt} \end{bmatrix} = \begin{bmatrix} -4.5 & -0.7 \\ 57.3 & 3.1 \end{bmatrix} \begin{bmatrix} c_A{}' \\ T' \end{bmatrix} + \begin{bmatrix} 0 & 1 \\ 0.2 & 0 \end{bmatrix} \begin{bmatrix} T_c{}' \\ c_{Af}{}' \end{bmatrix}$$

$$\begin{bmatrix} c_A{}' \\ T' \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} c_A{}' \\ T' \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} T_c{}' \\ c_{Af}{}' \end{bmatrix}$$

with states $\boldsymbol{x} = \begin{bmatrix} c_A \\ T \end{bmatrix}$ and control inputs $\boldsymbol{u} = \begin{bmatrix} T_c \\ c_{Af} \end{bmatrix}$

Example: Stability of CSTR system matrix cont'd

Recall PDC course: The zeros of the characteristic polynomial $\det(\lambda \boldsymbol{I} - \boldsymbol{A})$, i.e., the eigenvalues of the system matrix $\boldsymbol{A}$, contain information about the dynamic behavior of the described system.

Thus, we compute:

```
    import numpy as np
    import scipy

    system_matrix_A = np.array([[-4.5, -0.7], [57.3, 3.1]])
    l, _ = scipy.linalg.eig(system_matrix_A)
    print(l)
... [-0.7+5.06655702j -0.7-5.06655702j]
```

$\rightarrow \lambda_{1,2} = \boxed{-0.7} \pm 5.06655702j$

The real parts of the zeros of the characteristic polynomial must be negative if the system described by matrix $\boldsymbol{A}$ is stable. → Holds true!

Example based on CSTR Model by MathWorks: https://www.mathworks.com/help/mpc/gs/cstr-model.html

Definite matrices

Let $\boldsymbol{M} \in \mathbb{R}^{n \times n}$ be a real, square, and **symmetric** (i.e., $\boldsymbol{M}^T = \boldsymbol{M}$) matrix. Then

$\boldsymbol{M}$ is **positive-definite** … $\Leftrightarrow \boldsymbol{x}^T \boldsymbol{M} \boldsymbol{x} > \boldsymbol{0} \; \forall \; \boldsymbol{x} \in \mathbb{R}^n \backslash \{\boldsymbol{0}\}$.

$\Leftrightarrow$ all eigenvalues of $\boldsymbol{M}$ are > 0.

$\boldsymbol{M}$ is **positive-semidefinite** … $\Leftrightarrow \boldsymbol{x}^T \boldsymbol{M} \boldsymbol{x} \geq \boldsymbol{0} \; \forall \; \boldsymbol{x} \in \mathbb{R}^n$.

$\Leftrightarrow$ all eigenvalues of $\boldsymbol{M}$ are ≥ 0.

$\boldsymbol{M}$ is **negative-definite** … $\Leftrightarrow \boldsymbol{x}^T \boldsymbol{M} \boldsymbol{x} < \boldsymbol{0} \; \forall \; \boldsymbol{x} \in \mathbb{R}^n \backslash \{\boldsymbol{0}\}$.

$\Leftrightarrow$ all eigenvalues of $\boldsymbol{M}$ are < 0.

$\boldsymbol{M}$ is **negative-semidefinite** … $\Leftrightarrow \boldsymbol{x}^T \boldsymbol{M} \boldsymbol{x} \leq \boldsymbol{0} \; \forall \; \boldsymbol{x} \in \mathbb{R}^n$.

$\Leftrightarrow$ all eigenvalues of $\boldsymbol{M}$ are ≤ 0.

Lecture outline

- Algebra refresher
  - Vectors, vector operations, and vector norms
  - Matrices and matrix operations
  - Determinant
  - Eigenvalues and eigenvectors
  - Matrix rank
  - Matrix norms
- Approximating solutions of linear systems of equation
- Approximating solutions of nonlinear systems of equation

Matrix rank

The rank of a matrix is equal to the minimum number of linearly independent rows or columns.

For square matrices, the number of independent rows and columns is equal, i.e., column and row rank are identical.

In general, a matrix $\boldsymbol{A} \in K^{m \times n}$ is of full rank if $\mathrm{rank}(\boldsymbol{A}) = \min(m, n)$. Otherwise, $\boldsymbol{A}$ is rank deficient, i.e., $\mathrm{rank}(\boldsymbol{A}) < \min(m, n)$.

Example:

```
    import numpy as np

    matrix_A = np.array([[5,2,4], [1,9,3], [2,10,3], [11,4,0]])
    rank = np.linalg.matrix_rank(matrix_A)
    print("Rank of matrix A: %i" %rank)
... Rank of matrix A: 3
```

$$\boldsymbol{A} = \begin{bmatrix} 5 & 2 & 4 \\ 1 & 9 & 3 \\ 2 & 10 & 3 \\ 11 & 4 & 0 \end{bmatrix}$$

→ Matrix $\boldsymbol{A}$ has full rank since $\min(4,3) = 3$.

Lecture outline

- Algebra refresher
  - Vectors, vector operations, and vector norms
  - Matrices and matrix operations
  - Determinant
  - Eigenvalues and eigenvectors
  - Matrix rank
  - Matrix norms
- Approximating solutions of linear systems of equation
- Approximating solutions of nonlinear systems of equation

Matrix norms

Let $\boldsymbol{A} \in K^{n\times n}$ be $\boldsymbol{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$. A matrix norm is a function $\|\cdot\|: \mathbb{R}^{n\times n} \to \mathbb{R}_+, \boldsymbol{A} \to \|\boldsymbol{A}\|$ that satisfies conditions (1) to (3) on slide 9. Recalling the vector norms from slide 9, for $1 \leq p \leq \infty$, the matrix norm is defined as

$$\|\boldsymbol{A}\|_p = \max_{\boldsymbol{v}\neq\boldsymbol{0}} \frac{\|\boldsymbol{A}\boldsymbol{v}\|_p}{\|\boldsymbol{v}\|_p}$$

"How large is matrix A?"

- $\|\boldsymbol{A}\|_1 := \max_j \sum_{i=1}^{n} |a_{ij}|$ → L_1-norm (maximum absolute column sum) `scipy.linalg.norm(A,1)`
- $\|\boldsymbol{A}\|_\infty := \max_i \sum_{j=1}^{n} |a_{ij}|$ → L_{inf}-norm (maximum absolute row sum) `scipy.linalg.norm(A,np.inf)`
- $\|\boldsymbol{A}\|_2 := \sigma_{\max}(\boldsymbol{A})$ → Spectral norm ($\sigma_{\max}$ is the largest singular value of $\boldsymbol{A}$) `scipy.linalg.norm(A,2)`
- $\|\boldsymbol{A}\|_F := \sqrt{\sum_{i,j=1}^{n} |a_{ij}|^2}$ →Frobenius norm `scipy.linalg.norm(A,'fro')`

Matrix exponential

Note: This will become important in lecture 5!

If you don't remember this, practise it! *here*

Imagine we would like to compute the exponential of a matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$: $e^{\boldsymbol{A}}$.

Important properties:

- A **diagonal matrix** is a matrix where all nonzero entries lie on the diagonal.
- A matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ is **diagonizable** if there exists an invertible matrix $\boldsymbol{P}$ and a diagonal matrix $\boldsymbol{D}$ such that

$$\boldsymbol{A} = \boldsymbol{P}\boldsymbol{D}\boldsymbol{P}^{-1} \Leftrightarrow \boldsymbol{P}^{-1}\boldsymbol{A}\boldsymbol{P} = \boldsymbol{D}$$

- If $\boldsymbol{P}$ is invertible, $e^{\boldsymbol{P}\boldsymbol{D}\boldsymbol{P}^{-1}} = \boldsymbol{P}e^{\boldsymbol{D}}\boldsymbol{P}^{-1}$ holds.

The exponential of a diagonal matrix $\boldsymbol{D} \in \mathbb{R}^{n \times n}$ is equal to applying the exponential function to all entries on the diagonal s.t.

$$e^{\boldsymbol{D}} = e^{\begin{bmatrix} d_{11} & \cdots & 0 & \cdots & 0 \\ 0 & \ddots & d_{ii} & \ddots & 0 \\ 0 & \cdots & 0 & \cdots & d_{nn} \end{bmatrix}} = \begin{bmatrix} e^{d_{11}} & \cdots & 0 & \cdots & 0 \\ 0 & \ddots & e^{d_{ii}} & \ddots & 0 \\ 0 & \cdots & 0 & \cdots & e^{d_{nn}} \end{bmatrix}.$$

```
scipy.linalg.expm(A)
```

Lecture outline

- Algebra refresher
- Approximating solutions of linear systems of equation
  - Definition of a linear system
  - Approximation method I: Jacobi method
  - Approximation method II: Gauss-Seidel method
- Approximating solutions of nonlinear systems of equation

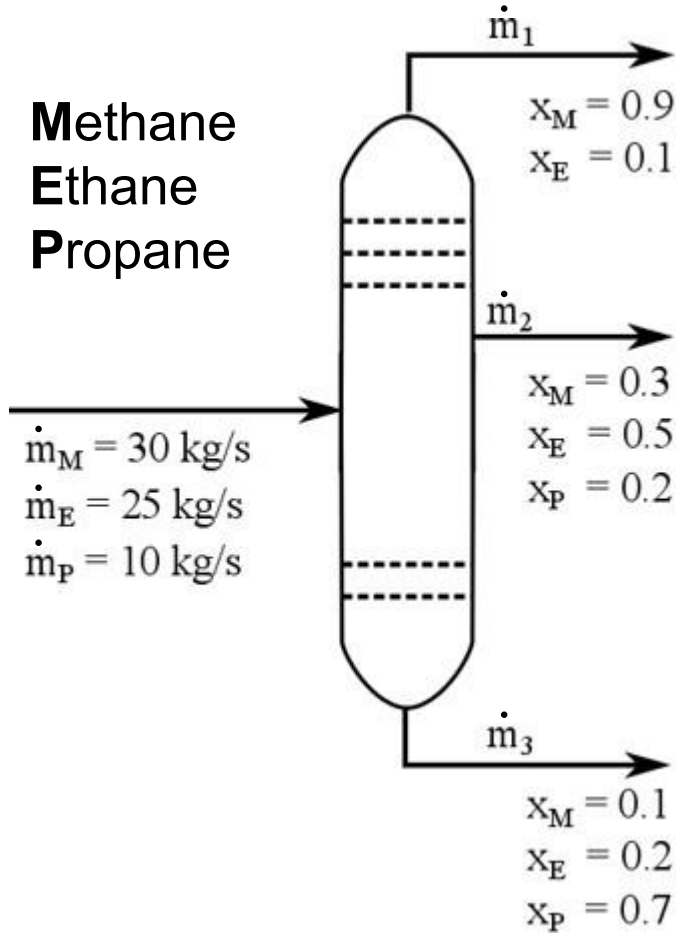
An exercise: distillation column

- Consider a distillation column. The feed consist of a ternary mixture of Methane, Ethane, and Propane with component mass streams $\dot{m}_M, \dot{m}_E$, and $\dot{m}_P$,in kg/s.
- The component compositions x_i are given for the outlet streams $\dot{m}_1, \dot{m}_2$, and $\dot{m}_3$.
- System of equations from component mass balances:

$$0.9\dot{m}_1 + 0.3\dot{m}_2 + 0.1\dot{m}_3 = 30$$
$$0.1\dot{m}_1 + 0.5\dot{m}_2 + 0.2\dot{m}_3 = 25$$
$$0.0\dot{m}_1 + 0.2\dot{m}_2 + 0.7\dot{m}_3 = 10$$

- Matrix-vector equation

$$\begin{pmatrix} 0.9 & 0.3 & 0.1 \\ 0.1 & 0.5 & 0.2 \\ 0 & 0.2 & 0.7 \end{pmatrix} \begin{pmatrix} \dot{m}_1 \\ \dot{m}_2 \\ \dot{m}_3 \end{pmatrix} = \begin{pmatrix} 30 \\ 25 \\ 10 \end{pmatrix}$$



Systems of linear equations

- Consider set of n linearly independent equations:

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \ldots + a_{1n}x_n = b_1 \quad (1)$$
$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \ldots + a_{2n}x_n = b_2 \quad (2)$$
$$\vdots \qquad\qquad \vdots \qquad\qquad \vdots$$
$$a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \ldots + a_{nn}x_n = b_n \quad (n)$$

- Can be written as matrix equation $\boldsymbol{Ax} = \boldsymbol{b}$ with

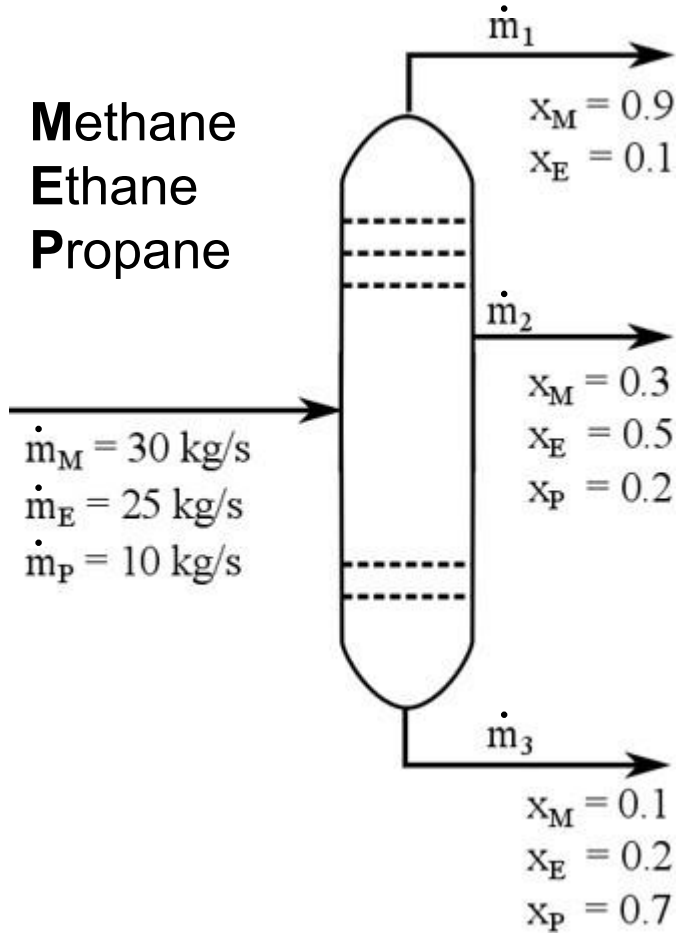
$$\boldsymbol{A} = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{bmatrix}, \boldsymbol{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}, \boldsymbol{b} = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}$$

- Task: Determine the unknown $x_1, \ldots, x_n$

An exercise: distillation column

- Consider a distillation column. The feed consist of a ternary mixture of Methane, Ethane, and Propane with component mass streams $\dot{m}_M, \dot{m}_E$, and $\dot{m}_P$,in kg/s.
- Matrix-vector equation

$$\begin{pmatrix} 0.9 & 0.3 & 0.1 \\ 0.1 & 0.5 & 0.2 \\ 0 & 0.2 & 0.7 \end{pmatrix} \begin{pmatrix} \dot{m}_1 \\ \dot{m}_2 \\ \dot{m}_3 \end{pmatrix} = \begin{pmatrix} 30 \\ 25 \\ 10 \end{pmatrix}$$



- **Solution: just invert the matrix:**

$$\boldsymbol{Ax = b}$$
$$\boldsymbol{A^{-1}Ax = A^{-1}b}$$
$$\boldsymbol{x = A^{-1}b}$$

Excursion: Matrix inversion to solve $x = A^{-1}b$

- If for a matrix $\boldsymbol{A} \in K^{n \times n}$ there exists a matrix $\boldsymbol{B} \in K^{n \times n}$ s.t. $\boldsymbol{A} \cdot \boldsymbol{B} = \boldsymbol{I}_n$, matrix $\mathbf{A}$ is invertible. If $\boldsymbol{A} \cdot \boldsymbol{B} = \boldsymbol{I}_n$ holds, $\boldsymbol{B}$ is called the inverse of $\boldsymbol{A}$.
- The inversion of a matrix $\boldsymbol{A} \in K^{n \times n}$ is…

```
numpy.linalg.inv(A)
```

- … possible if $\boldsymbol{A}$ is non-singular. A non-singular matrix $\boldsymbol{A}$ has the following properties:
  - The determinant of $\boldsymbol{A}$ is nonzero: $\det(\boldsymbol{A}) \neq 0$
  - Matrix $\boldsymbol{A}$ has full rank: $\mathrm{rank}(\boldsymbol{A}) = n$
  - The system $\boldsymbol{Ax} = \boldsymbol{b}$ has a unique solution $\boldsymbol{x} \in \mathbb{R}^n$ for every $\boldsymbol{b} \in \mathbb{R}^n$.
  - The homogeneous system $\boldsymbol{Ax} = \mathbf{0}$ only evaluates to the trivial solution $\boldsymbol{x} = \mathbf{0}$.
- … impossible if $\boldsymbol{A}$ is singular.
- … **computationally challenging** ($\mathcal{O}(n^{2.5})$ to $\mathcal{O}(n^{2.2732})$[1]) if the matrix is sparse and large, i.e., most elements are zero.

Eberly et al. Faster Inversion and Other Black Box Matrix Computations Using Efficient Block Projections. arXiv:cs/0701188

Direct vs indirect methods for solving $\boldsymbol{Ax = b}$

Direct methods

- Solve the system of linear equations by performing a finite number of operations to find an exact solution
- Common algorithms: Gaussian Elimination, LU Decomposition, Cholesky Decomposition
- Yields an exact solution
- Suitable for small to medium-sized systems or dense matrices.
- Computationally expensive for large systems.

Indirect (a.k.a. Iterative) methods

- approximate the solution gradually by iterating
- Common algorithms: Jacobi, Gauss-Seidel, Conjugate Gradient
- Can handle very large systems, especially when the matrix is sparse.
- Often uses much less memory than direct methods.
- May converge slowly or not at all if the system is ill-conditioned.

Gaussian elimination

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

Extended Matrix

$$\left(\begin{array}{cccc|c} a_1 & a_2 & \dots & a_n & f_1 \\ b_1 & b_2 & \dots & b_n & f_2 \\ \vdots & & \ddots & \vdots & \vdots \\ z_1 & z_2 & \dots & z_n & f_n \end{array}\right) \longrightarrow \left(\begin{array}{cccc|c} a'_1 & a'_2 & \dots & a'_n & f'_1 \\ 0 & b'_2 & \dots & b'_n & f'_2 \\ \vdots & & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & z'_n & f'_n \end{array}\right) \longrightarrow \left(\begin{array}{cccc|c} 1 & 0 & \dots & 0 & x_1 \\ 0 & 1 & \dots & 0 & x_2 \\ \vdots & & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & x_n \end{array}\right)$$

Gaussian Elimination

Backward Substitution

Solution

- interchange two lines
- multiply/divide a line by a non-null scalar
- add/substract to one line a scalar multiple of another line

Lecture outline

- Algebra refresher
- Approximating solutions of linear systems of equation
  - Definition of a linear system
  - Approximation method I: Jacobi method
  - Approximation method II: Gauss-Seidel method
- Approximating solutions of nonlinear systems of equation

Approximation I: Jacobi method

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} m_1 \\ m_2 \\ m_3 \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \end{pmatrix}$$

the i-th equation $$\sum_{j=1}^{n} a_{ij} m_j = f_i$$

We can solve for m_i assuming that all other masses are unchanged

$$m_i^{(k)} = \frac{1}{a_{ii}} \left(f_i - \sum_{j \neq i} a_{ij} m_j^{(k-1)} \right)$$

Approximation I: Jacobi method

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} m_1 \\ m_2 \\ m_3 \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \end{pmatrix}$$

Initial guess $m_1^{(0)} = 10 \qquad m_2^{(0)} = 8 \qquad m_3^{(0)} = 24$

Improve the guess

$$m_i^{(k+1)} = \frac{1}{a_{ii}} \left(f_i - \sum_{j \neq i} a_{ij} m_j^{(k)} \right)$$

first iteration

$$\begin{cases} m_1^{(1)} = \frac{1}{a_{11}} \left(f_1 - a_{12} m_2^{(0)} - a_{13} m_3^{(0)} \right) \\ m_2^{(1)} = \frac{1}{a_{22}} \left(f_2 - a_{21} m_1^{(0)} - a_{23} m_3^{(0)} \right) \\ m_3^{(1)} = \frac{1}{a_{33}} \left(f_3 - a_{31} m_1^{(0)} - a_{32} m_2^{(0)} \right) \end{cases}$$

Approximation I: Jacobi method

Improve the guess

$$m_i^{(k+1)} = \frac{1}{a_{ii}} \left(f_i - \sum_{j \neq i} a_{ij} m_j^{(k)} \right)$$

n-th iteration

$$\begin{cases} m_1^{(n)} = \frac{1}{a_{11}} \left(f_1 - a_{12} m_2^{(n-1)} - a_{13} m_3^{(n-1)} \right) \\ m_2^{(n)} = \frac{1}{a_{22}} \left(f_2 - a_{21} m_1^{(n-1)} - a_{23} m_3^{(n-1)} \right) \\ m_3^{(n)} = \frac{1}{a_{33}} \left(f_3 - a_{31} m_1^{(n-1)} - a_{32} m_2^{(n-1)} \right) \end{cases}$$

Approximation I: Jacobi method

- When do we stop iterations?
  - We need a stopping condition!!

$$\left| \begin{pmatrix} f_1 \\ f_2 \\ f_3 \end{pmatrix} - \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} m_1^{(k)} \\ m_2^{(k)} \\ m_3^{(k)} \end{pmatrix} \right| < \text{tolerance}$$

$$1E^{-6}$$

```
np.linalg.norm(f-np.dot(A,m))
```

Approximation I: Jacobi method

$$\begin{pmatrix} 0.9 & 0.3 & 0.1 \\ 0.1 & 0.5 & 0.2 \\ 0.0 & 0.2 & 0.7 \end{pmatrix} \cdot \begin{pmatrix} m_1 \\ m_2 \\ m_3 \end{pmatrix} = \begin{pmatrix} 30.0 \\ 25.0 \\ 10.0 \end{pmatrix}$$

When applied to the distillation column this algorithm should converge in about 15 iterations to the correct solution with a tolerance of 1E-8

```
==============================
== Jacobi Solver for        ==
== linear systems           ==
==============================

Tolerance required 1.00e-08

Iteration 01 :  m_1 = 24.444444   m_2 = 38.000000   m_3 = 8.571429   | res = 5.87e+00
Iteration 02 :  m_1 = 19.714286   m_2 = 42.628571   m_3 = 2.106122   | res = 1.49e+00
Iteration 03 :  m_1 = 18.889796   m_2 = 45.379592   m_3 = 1.320117   | res = 7.63e-01
Iteration 04 :  m_1 = 18.060123   m_2 = 45.859929   m_3 = 1.182878   | res = 1.33e-01
Iteration 05 :  m_1 = 17.915260   m_2 = 45.943797   m_3 = 1.158915   | res = 2.33e-02
Iteration 06 :  m_1 = 17.889966   m_2 = 45.958441   m_3 = 1.154731   | res = 4.06e-03
Iteration 07 :  m_1 = 17.885550   m_2 = 45.960998   m_3 = 1.154001   | res = 7.09e-04
Iteration 08 :  m_1 = 17.884779   m_2 = 45.961444   m_3 = 1.153873   | res = 1.24e-04
Iteration 09 :  m_1 = 17.884644   m_2 = 45.961522   m_3 = 1.153851   | res = 2.16e-05
Iteration 10 :  m_1 = 17.884620   m_2 = 45.961536   m_3 = 1.153847   | res = 3.78e-06
Iteration 11 :  m_1 = 17.884616   m_2 = 45.961538   m_3 = 1.153846   | res = 6.59e-07
Iteration 12 :  m_1 = 17.884616   m_2 = 45.961538   m_3 = 1.153846   | res = 1.15e-07
Iteration 13 :  m_1 = 17.884615   m_2 = 45.961538   m_3 = 1.153846   | res = 2.01e-08
Iteration 14 :  m_1 = 17.884615   m_2 = 45.961538   m_3 = 1.153846   | res = 3.51e-09

Iteration has  converged  in  14  iteration res = 3.508647e-09
```

Solving linear systems of equations numerically: Jacobi method

The matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ is decomposed into $\boldsymbol{A} = \boldsymbol{D} + \boldsymbol{L} + \boldsymbol{U}$ such that

$$\boldsymbol{D} = \begin{bmatrix} a_{11} & 0 & \cdots & \cdots & 0 \\ 0 & a_{22} & \cdots & \cdots & 0 \\ 0 & 0 & a_{33} & \cdots & \vdots \\ \vdots & \vdots & \vdots & \ddots & 0 \\ 0 & 0 & \cdots & 0 & a_{nn} \end{bmatrix}, \boldsymbol{L} = \begin{bmatrix} 0 & 0 & \cdots & \cdots & 0 \\ a_{21} & 0 & \cdots & \cdots & 0 \\ a_{31} & a_{32} & 0 & \cdots & \vdots \\ \vdots & \vdots & \vdots & \ddots & 0 \\ a_{n1} & a_{n2} & \cdots & a_{nn-1} & 0 \end{bmatrix},$$

$$\text{and } \boldsymbol{U} = \begin{bmatrix} 0 & a_{12} & \cdots & \cdots & a_{1n} \\ 0 & 0 & a_{23} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & a_{n-1n} \\ 0 & 0 & \cdots & 0 & 0 \end{bmatrix}.$$

Condition: No zero values on main diagonal of matrix $\boldsymbol{A}$.

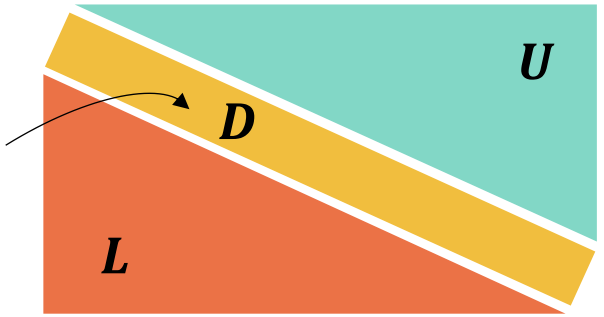
Jacobi method cont'd

- Problem reformulation:

$$\boldsymbol{Ax = b} \xLeftrightarrow{\boldsymbol{A=D+(L+U)}} \boldsymbol{Dx + (L+U)x = b}$$

$$\Leftrightarrow \boldsymbol{Dx = b - (L+U)x}$$

$$\Leftrightarrow \boldsymbol{x = D^{-1}(b - (L+U)x)}$$



- Recall the fixed-point method from Lecture 2 (x=g(x))

→ We can formulate an update rule

$$\boldsymbol{x^{(k+1)} = D^{-1}(b - (L+U)x^k)}$$

$\frac{1}{a_{ii}}$ on diagonal

$$\Leftrightarrow x_i^{(k+1)} = \frac{b_i - \sum_{j \neq i} a_{ij} x_j^{(k)}}{a_{ii}} \text{ for } i = 1, \ldots, n$$

Condition for convergence of iterative methods

Let $\boldsymbol{A} = \boldsymbol{M} + \boldsymbol{N} \in \mathbb{R}^{n\times n}$ be nonsingular and $\boldsymbol{b} \in \mathbb{R}^n$. If $\boldsymbol{M}$ is non-singular and the spectral radius of $\boldsymbol{M}^{-1}\boldsymbol{N}$, $\rho(\boldsymbol{M}^{-1}\boldsymbol{N})$, is less than 1, then the iterates $\boldsymbol{x}^k$ defined by

$$\boldsymbol{M}\boldsymbol{x}^{k+1} = \boldsymbol{b} - \boldsymbol{N}\boldsymbol{x}^k$$

converge to $\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b}$ for any starting vector $\boldsymbol{x}^0$.

The **spectral radius** of a matrix $\boldsymbol{K}$ is defined as the maximum of the absolute values of its eigenvalues

$$\rho(\boldsymbol{K}) = \max\{|\lambda_1|, \dots, |\lambda_n|\}.$$

For the Jacobian iterative method ($\boldsymbol{D}\boldsymbol{x}^{(k+1)} = \boldsymbol{b} - (\boldsymbol{L}+\boldsymbol{U})\boldsymbol{x}^k$), this means:

$$\boldsymbol{M} = \boldsymbol{D} \qquad \boldsymbol{N} = \boldsymbol{L} + \boldsymbol{U} \qquad \boldsymbol{b} = \boldsymbol{b}$$

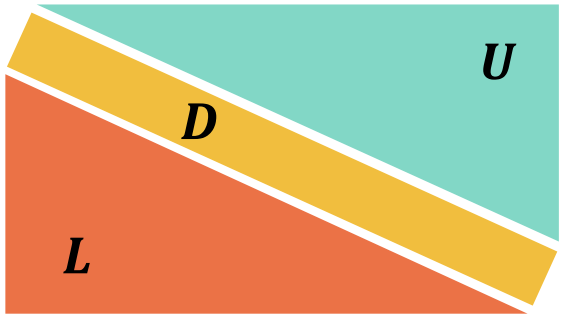
$$\rho(\boldsymbol{M}^{-1}\boldsymbol{N}) = \rho(\boldsymbol{D}^{-1}(\boldsymbol{L} + \boldsymbol{U}))$$

Convergence of the Jacobi method

- The Jacobi method is especially suitable for strictly row diagonal-dominant matrices, i.e., the values on the diagonal are larger than the sum of the other row entries s.t.

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}| \text{ for } i = 1, \ldots, n.$$

$$\boldsymbol{A} = \boldsymbol{D} + \boldsymbol{L} + \boldsymbol{U}$$



- But why?

  If the diagonal entries are the largest within the row, the condition for convergence,

$$\rho\left(\boldsymbol{D}^{-1}(\boldsymbol{L} + \boldsymbol{U})\right) < 1,$$

  can be met more easily. The eigenvalues of matrices of the form $\boldsymbol{D}^{-1}(\boldsymbol{L} + \boldsymbol{U}) =$

$$\begin{bmatrix} 0 & \frac{a_{12}}{a_{11}} & \cdots & \frac{a_{1n}}{a_{11}} \\ \frac{a_{21}}{a_{22}} & 0 & \cdots & \frac{a_{2n}}{a_{22}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{a_{n1}}{a_{nn}} & \frac{a_{n2}}{a_{nn}} & \cdots & 0 \end{bmatrix}$$

  where all entries on the diagonal are zero and all off-diagonal entries are << 1 are bound to be smaller than one[1].

[1] CHI-KWONG LI AND FUZHEN ZHANG. EIGENVALUE CONTINUITY AND GERSGORIN'S THEOREM. Electronic Journal of Linear Algebra, ISSN1081-3810. A publication of the International Linear Algebra Society Volume 35, pp. 619-625, December2019

Jacobi method: live coding

- Open Colab:

  Jacobi method using fixed-point iteration

```
Fixed_point_iteration(C, g, x0, max_iter, eps)
```

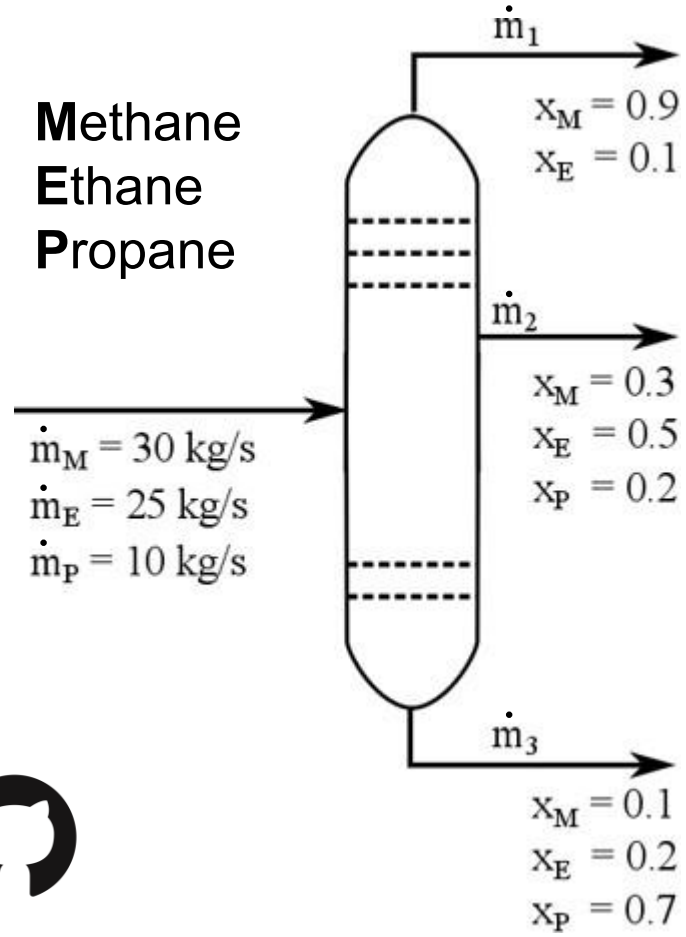
  Compare to numpy built-in method numpy.linalg.solve

```
y = np.linalg.solve(a_matrix, b_vector)
```

  → numpy.linalg.solve uses the LAPACK (linear algebra package) routine

  - Designed for, e.g, solving linear systems of equation and eigenvalue problems
  - Suitable for dense matrices, but not necessarily sparce matrices

- Find more in the Github repository of the course: https://github.com/process-intelligence-research/computational_practicum_lecture_coding/tree/main



Lecture outline

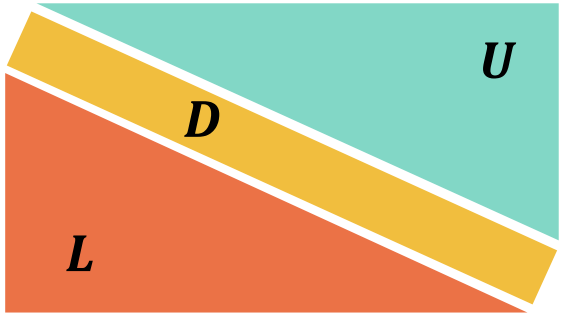
- Algebra refresher
- Approximating solutions of linear systems of equation
  - Definition of a linear system
  - Approximation method I: Jacobi method
  - Approximation method II: Gauss-Seidel method
- Approximating solutions of nonlinear systems of equation

Another indirect solution method: Gauss-Seidel method

- Problem reformulation:

$$\boldsymbol{Ax} = \boldsymbol{b} \overset{\boldsymbol{A=D+L+U}}{\Longleftrightarrow} \boldsymbol{Dx} + \boldsymbol{Lx} + \boldsymbol{Ux} = \boldsymbol{b}$$

$$\Leftrightarrow \boldsymbol{Dx} = \boldsymbol{b} - \boldsymbol{Lx} - \boldsymbol{Ux}$$



Formulate as a fixed-point problem: $\boldsymbol{x}^{(k+1)} = (\boldsymbol{D}+\boldsymbol{L})^{-1}\boldsymbol{b} - (\boldsymbol{D}+\boldsymbol{L})^{-1}\boldsymbol{U}\boldsymbol{x}^{(k)}$

- 1st entry Jacobi method: $x_1^{(k+1)} = \frac{b_1 - \sum_{j=2}^{n} a_{1j} x_j^{(k)}}{a_{11}}$
- Observation: we know entries $x_{j<i}^{(k+1)}$ already when determining $x_i^{(k+1)}$ → use this knowledge!

$$x_i^{(k+1)} = \frac{b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)}}{a_{ii}} \text{ for } i = 2, \dots, n-1$$

- Final entry: $x_n^{(k+1)} = \frac{b_n - \sum_{j=1}^{n-1} a_{ij} x_j^{(k+1)}}{a_{nn}}$

Another indirect solution method: Gauss-Seidel method

Final update rule:

$$x_i^{(k+1)} = \frac{b_i - \sum_{j<i} a_{ij} x_j^{(k+1)} - \sum_{j>i} a_{ij} x_j^{(k)}}{a_{ii}} \text{ for } i = 1, \dots, n$$

or as matrix expression: $\boldsymbol{x}^{(k+1)} = \boldsymbol{D}^{-1}\left(\boldsymbol{b} - \boldsymbol{L}\boldsymbol{x}^{(k+1)} - \boldsymbol{U}\boldsymbol{x}^{(k)}\right)$.

! Updates $x_i^{(k+1)}$ depend on updates $x_j^{(k+1)}$ for $j < i$.

→ Ordering of x_i affects convergence behavior.

Jacobi algorithm

$$m_i^{(k+1)} = \frac{1}{a_{ii}} \left(f_i - \sum_{j \neq i} a_{ij} m_j^{(k)} \right)$$

Gauss-Seidel algorithm

$$m_i^{(k+1)} = \frac{1}{a_{ii}} \left(f_i - \sum_{j<i} a_{ij} m_j^{(k+1)} - \sum_{j>i} a_{ij} m_j^{(k)} \right)$$

Gauss-Seidel vs. Jacobi

- Equations for x_i cannot be solved in parallel for Gauss-Seidel method.
- If the matrix $\boldsymbol{A} \in \mathbb{R}^{n \times n}$ is **strictly row-diagonally dominant**, both the Jacobi and the Gauss-Seidel method converge[1].
- If $\boldsymbol{A}$ is symmetric and **positive definite**, then the **Gauss-Seidel method converges**[2]

BUT

- If **both** $\boldsymbol{A}$ and $2\boldsymbol{D} - \boldsymbol{A}$ are symmetric and **positive definite, then the Jacobi method converges**[2]

[1] https://www.damtp.cam.ac.uk/research/afha/lectures/Part_II_NumAn/Lect_17_slides_alt.pdf, Slide 16
[2] https://www.damtp.cam.ac.uk/research/afha/lectures/Part_II_NumAn/Lect_17_slides_alt.pdf, Slide 20

Lecture outline

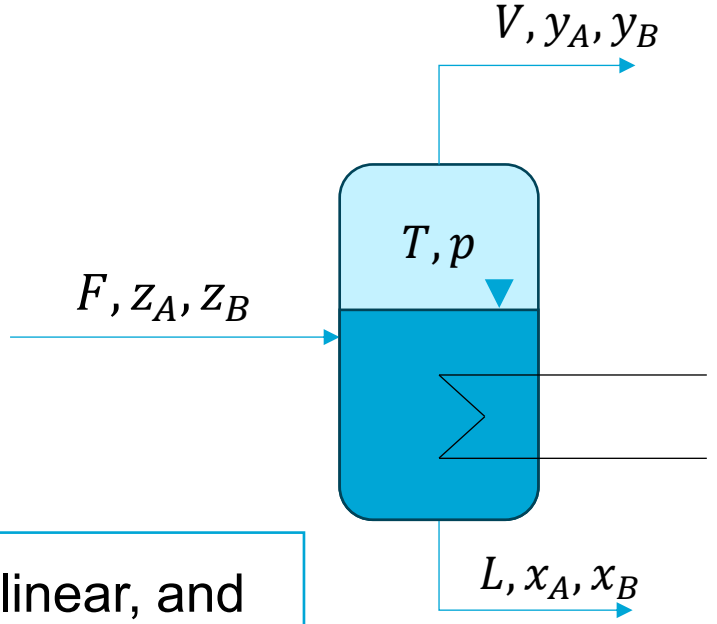
- Algebra refresher
- Approximating solutions of linear systems of equation
- Approximating solutions of nonlinear systems of equation
  - What are systems of nonlinear equations?
  - Newton-Raphson method

Example: Flash unit

Example: Consider a pT-flash unit with $\boldsymbol{x} = [V, L, x_A, x_B]$ assuming given F, z_A, z_B, T, p. Taking Raoult's and Dalton's law into account, the vapor pressure fraction can be expressed as $y_i = \frac{x_i p_i^{\text{sat}}(T)}{p}$.

Nonlinear terms!

$$
\begin{aligned}
F z_A &= V K_A x_A + L x_A \\
F z_B &= V K_B y_B + L x_B \\
x_A + x_B &= 1 \\
K_A x_A + K_B x_B &= 1
\end{aligned}
\qquad \text{with } K_i = \frac{p_i^{\text{sat}}(T)}{p}
$$



In chemical engineering, most of the physical phenomena are nonlinear, and methods for solving linear systems are not valid.

How can we solve systems of nonlinear equations numerically?

Excurse: Raoult's and Dalton's law

Raoult's Law[1]

Raoult's law applies to *ideal* mixtures of liquids. Raoult's law states that the partial vapor pressure of each volatile component in an ideal solution is proportional to its mole fraction in the liquid phase.

For a single component i in an *ideal* solution, the partial vapor pressure p_i of component i is defined as

$$p_i = p_i^* x_i$$

where x_i is the mole fraction of component i in the liquid phase and p_i^* is the equilibrium vapor pressure of the pure component i.

Dalton's law[2]

The total pressure of a mixture of N gases (that are not reacting with each other) is equal to the sum of the partial pressures p_i of the single components i in the gas phase:

$$p_{total} = \sum_{i=1}^{N} p_i$$

[1] https://en.wikipedia.org/wiki/Raoult%27s_law

[2] https://en.wikipedia.org/wiki/Dalton%27s_law

Lecture outline

- Algebra refresher
- Approximating solutions of linear systems of equation
- Approximating solutions of nonlinear systems of equation
  - What are systems of nonlinear equations?
  - Newton-Raphson method

Newton-Raphson for multiple equations

- Coupled nonlinear equations

$$f_1(x_1, x_2) = 0$$
$$f_2(x_1, x_2) = 0$$

f1 and f2 are non-linear functions

Taylor expansion

$$f_1(x_1, x_2) = f_1(x_1^{(0)}, x_2^{(0)}) + \frac{\partial f_1}{\partial x_1}|_{x^{(0)}} \cdot (x_1 - x_1^{(0)}) + \frac{\partial f_1}{\partial x_2}|_{x^{(0)}} \cdot (x_2 - x_2^{(0)}) + \ldots$$
$$f_2(x_1, x_2) = f_2(x_1^{(0)}, x_2^{(0)}) + \frac{\partial f_2}{\partial x_1}|_{x^{(0)}} \cdot (x_1 - x_1^{(0)}) + \frac{\partial f_2}{\partial x_2}|_{x^{(0)}} \cdot (x_2 - x_2^{(0)}) + \ldots$$

Newton-Raphson for multiple equations

- Coupled nonlinear equations

$$f_1(x_1, x_2) = 0$$
$$f_2(x_1, x_2) = 0$$

f1 and f2 are non-linear functions

$$f_1(x_1, x_2) = f_1(x_1^{(0)}, x_2^{(0)}) + \frac{\partial f_1}{\partial x_1}|_{x^{(0)}} \cdot (x_1 - x_1^{(0)}) + \frac{\partial f_1}{\partial x_2}|_{x^{(0)}} \cdot (x_2 - x_2^{(0)}) + \dots$$
$$f_2(x_1, x_2) = f_2(x_1^{(0)}, x_2^{(0)}) + \frac{\partial f_2}{\partial x_1}|_{x^{(0)}} \cdot (x_1 - x_1^{(0)}) + \frac{\partial f_2}{\partial x_2}|_{x^{(0)}} \cdot (x_2 - x_2^{(0)}) + \dots$$

0

Newton-Raphson for multiple equations

$$f_1(x_1, x_2) = 0$$
$$f_2(x_1, x_2) = 0$$

f1 and f2 are non-linear functions

Taylor expansion

$$\frac{\partial f_1}{\partial x_1}|_{x^{(0)}} \cdot (x_1 - x_1^{(0)}) + \frac{\partial f_1}{\partial x_2}|_{x^{(0)}} \cdot (x_2 - x_2^{(0)}) = -f_1(x_1^{(0)}, x_2^{(0)})$$
$$\frac{\partial f_2}{\partial x_1}|_{x^{(0)}} \cdot (x_1 - x_1^{(0)}) + \frac{\partial f_2}{\partial x_2}|_{x^{(0)}} \cdot (x_2 - x_2^{(0)}) = -f_2(x_1^{(0)}, x_2^{(0)})$$

Newton-Raphson for multiple equations

$$f_1(x_1, x_2) = 0$$
$$f_2(x_1, x_2) = 0$$

f1 and f2 are non-linear functions

$$\delta_1^{(0)} = x_1 - x_1^{(0)}$$
$$\delta_2^{(0)} = x_2 - x_2^{(0)}$$

Newton-Raphson for multiple equations

$$f_1(x_1, x_2) = 0$$
$$f_2(x_1, x_2) = 0$$

f1 and f2 are non-linear functions

Taylor expansion

$$\frac{\partial f_1}{\partial x_1}|_{x^{(0)}} \cdot \delta_1^{(0)} + \frac{\partial f_1}{\partial x_2}|_{x^{(0)}} \cdot \delta_2^{(0)} = -f_1(x_1^{(0)}, x_2^{(0)})$$
$$\frac{\partial f_2}{\partial x_1}|_{x^{(0)}} \cdot \delta_1^{(0)} + \frac{\partial f_2}{\partial x_2}|_{x^{(0)}} \cdot \delta_2^{(0)} = -f_2(x_1^{(0)}, x_2^{(0)})$$

Newton-Raphson for multiple equations

$$f_1(x_1, x_2) = 0$$
$$f_2(x_1, x_2) = 0$$

f1 and f2 are non-linear functions

Taylor expansion

$$\begin{pmatrix} \frac{\partial f_1}{\partial x_1}\Big|_{x^{(0)}} & \frac{\partial f_1}{\partial x_2}\Big|_{x^{(0)}} \\ \frac{\partial f_2}{\partial x_1}\Big|_{x^{(0)}} & \frac{\partial f_2}{\partial x_2}\Big|_{x^{(0)}} \end{pmatrix} \cdot \begin{pmatrix} \delta_1^{(0)} \\ \delta_2^{(0)} \end{pmatrix} = \begin{pmatrix} f_1^{(0)} \\ f_2^{(0)} \end{pmatrix}$$

Jacobian

Newton-Raphson for multiple equations

- Iterative solution of system of non-linear equations:
  - Each iteration requires solving a systems of **linear equations**

$$\mathcal{J} \cdot \boldsymbol{\delta} = -\mathbf{f} \quad \rightarrow \quad \begin{matrix} \delta_1^{(0)} \\ \delta_2^{(0)} \end{matrix}$$

$$x_1^{(1)} = x_1^{(0)} + \delta_1^{(0)} \qquad x_1^{(n+1)} = x_1^{(n)} + \delta_1^{(n)}$$

$$x_2^{(1)} = x_2^{(0)} + \delta_2^{(0)} \qquad x_2^{(n+1)} = x_2^{(n)} + \delta_2^{(n)}$$

Newton-Raphson for multiple equations

$$f_1(x_1, \dots x_k) = 0$$

$$\dots\dots$$

$$f_k(x_1, \dots x_k) = 0$$

$$\begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \dots & \frac{\partial f_1}{\partial x_k} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \dots & \frac{\partial f_2}{\partial x_k} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_k}{\partial x_1} & \frac{\partial f_k}{\partial x_2} & \dots & \frac{\partial f_k}{\partial x_k} \end{pmatrix} \cdot \begin{pmatrix} \delta_1 \\ \delta_2 \\ \vdots \\ \delta_k \end{pmatrix} = - \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_k \end{pmatrix}$$

Systems of nonlinear equations: Newton-Raphson

- Recall lecture 2: The Newton-Raphson method for solving univariate nonlinear equations using the tangent line:

$$x^{(k+1)} \approx x^{(k)} - \frac{f(x^{(k)})}{f'(x^{(k)})}$$

→ This can be generalized to systems of nonlinear equations!

- A system of nonlinear equations of dimension n can be rewritten as a set of multivariate functions, i.e., $\boldsymbol{F}: D \subseteq \mathbb{R}^n \to \mathbb{R}^n$ with $\boldsymbol{F}(\boldsymbol{x}) = \boldsymbol{0}$.
- While applying the Newton-Raphson method to a single nonlinear function f requires the derivative f', here we need the Jacobian matrix (matrix of first-order partial derivatives):

$$\boldsymbol{J} \in \mathbb{R}^{n \times n} \text{ with } \boldsymbol{J}_{ij} = \frac{\partial F_i}{\partial x_j}$$

Systems of nonlinear equations: Newton-Raphson

- The derivation of the updating rule is equivalent to the one for univariate functions:

$$\boldsymbol{F}\left(\boldsymbol{x}^{k+1}\right) = \boldsymbol{F}\left(\boldsymbol{x}^{k}\right) + \boldsymbol{J}\left(\boldsymbol{x}^{k}\right)\left(\boldsymbol{x}^{k+1} - \boldsymbol{x}^{k}\right) + o\left(\left(\boldsymbol{x}^{k+1} - \boldsymbol{x}^{k}\right)^{2}\right) = \boldsymbol{0}$$

$$\boldsymbol{x}^{k+1} - \boldsymbol{x}^{k} \approx -\boldsymbol{J}^{-1}(\boldsymbol{x}^{k})\boldsymbol{F}\left(\boldsymbol{x}^{k}\right)$$

We neglect the remainder of the approximation

- Then, we arrive at the resulting updating rule:

$$\boldsymbol{x}^{k+1} = \boldsymbol{x}^{k} - \boldsymbol{J}^{-1}(\boldsymbol{x}^{k})\boldsymbol{F}\left(\boldsymbol{x}^{k}\right)$$

- To solve with Newton-Raphson, you need:
  - The function value evaluated in the point $\boldsymbol{x}^{k}$
  - The Jacobian of the function computed in the point $\boldsymbol{x}^{k}$

Systems of nonlinear equations: Newton-Raphson

- Note that the matrix inversion computation is very expensive, especially for large systems

$$\boldsymbol{x}^{k+1} = \boldsymbol{x}^{k} - \boldsymbol{J}^{-1}(\boldsymbol{x}^{k})\boldsymbol{F}(\boldsymbol{x}^{k})$$

- You can instead get the updated solution $\boldsymbol{x}^{k+1}$ by solving the linear system:

$$\boldsymbol{F}(\boldsymbol{x}^{k}) + \boldsymbol{J}(\boldsymbol{x}^{k})(\boldsymbol{x}^{k+1} - \boldsymbol{x}^{k}) = \boldsymbol{0}$$

- To solve it you can use an iterative method for linear systems (e.g., Gauss-Seidel), i.e., approximate the solution of $\underbrace{\boldsymbol{J}(\boldsymbol{x}^{k})}_{\text{“}\boldsymbol{A}\text{”}}\cdot\underbrace{(\boldsymbol{x}^{k+1} - \boldsymbol{x}^{k})}_{\text{“}\boldsymbol{x}\text{”}} = \underbrace{-\boldsymbol{F}(\boldsymbol{x}^{k})}_{\text{“}\boldsymbol{b}\text{”}}$ at each iteration k.

- Newton-Raphson method for systems of nonlinear equations results in a nested iterative method

Newton Raphson: live coding

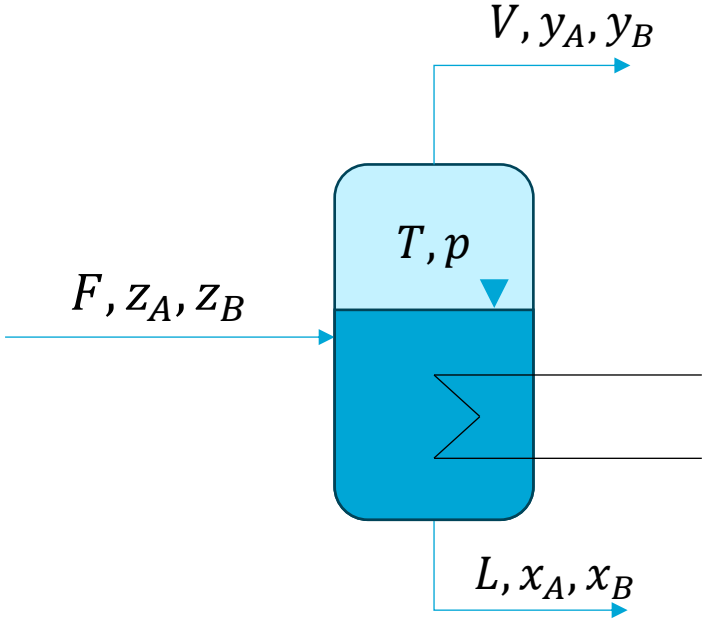
Open Colab:

Newton-Raphson method with inversion of the Jacobian

```
x_netwon_raphson = newton_raphson(x_start, max_iter)
```

Scipy built-in method scipy.optimize.root

```
x_root = scipy.optimize.root(function_value_pTflash, x_start, jac = jacobian_pTflash)
```



Find more in the Github repository of the course: https://github.com/process-intelligence-research/computational_practicum_lecture_coding/tree/main

Quasi-Newton method

Recall the Newton-Raphson method for approximating the solution of a system of nonlinear equations:

$$\boldsymbol{F}\left(\boldsymbol{x}^{k+1}\right) \approx \boldsymbol{F}\left(\boldsymbol{x}^{k}\right) + \boldsymbol{J}\left(\boldsymbol{x}^{k}\right)\left(\boldsymbol{x}^{k+1} - \boldsymbol{x}^{k}\right) = \boldsymbol{0}$$

$$\boldsymbol{x}^{k+1} - \boldsymbol{x}^{k} \approx -\boldsymbol{J}^{-1}(\boldsymbol{x}^{k})\boldsymbol{F}\left(\boldsymbol{x}^{k}\right)$$

We discussed methods so far:

a) Computing the inverse of the Jacobian matrix explicitly,

b) solving the linear systems of equations $\boldsymbol{J}\left(\boldsymbol{x}^{k}\right)\left(\boldsymbol{x}^{k+1} - \boldsymbol{x}^{k}\right) \approx -\boldsymbol{F}\left(\boldsymbol{x}^{k}\right)$ iteratively,

but there is another option:

c) we approximate the inverse of the Jacobian itself. → Quasi-Newton method

Quasi-Newton method cont'd

We seek to approximate the solution of $\boldsymbol{F}(\boldsymbol{x}_{k+1}) \approx \boldsymbol{J}(\boldsymbol{x}_k)(\boldsymbol{x}_{k+1} - \boldsymbol{x}_k) + \boldsymbol{F}(\boldsymbol{x}_k) = 0$

We assume that there exists an approximation $\boldsymbol{H}_{k+1}$of the inverse of the Jacobian $\boldsymbol{J}^{-1}(\boldsymbol{x}_k)$ s.t.

$$\boldsymbol{H}_{k+1}\boldsymbol{y}_k = \boldsymbol{s}_k$$

with $\boldsymbol{s}_k = (\boldsymbol{x}_{k+1} - \boldsymbol{x}_k)$ and $\boldsymbol{y}_k = \boldsymbol{F}(\boldsymbol{x}_{k+1}) - \boldsymbol{F}(\boldsymbol{x}_k)$.

1. If iteration 0: Start with an initial guess of $\boldsymbol{H}_0$.
2. Solve $\boldsymbol{p}_k \approx -\boldsymbol{H}_k\boldsymbol{F}(\boldsymbol{x}_k)$ with $\boldsymbol{p}_k = \boldsymbol{x}_{k+1} - \boldsymbol{x}_k$.
3. Set $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \boldsymbol{p}_k$ and determine $\boldsymbol{F}(\boldsymbol{x}_{k+1})$. Evaluate $\boldsymbol{s}_k$ and $\boldsymbol{y}_k$.
4. Using Broyden's method[1], a common update rule for $\boldsymbol{H}$ for the upcoming iteration is defined as

$$\boldsymbol{H}_{k+1} = \boldsymbol{H}_k + \frac{(\boldsymbol{s}_k - \boldsymbol{H}_k\boldsymbol{y}_k)\boldsymbol{y}_k^T}{\boldsymbol{y}_k^T\boldsymbol{y}_k}.$$

→ No need to compute the inverse of the Jacobian matrix at every iteration!

[1] https://www.ams.org/journals/mcom/1965-19-092/S0025-5718-1965-0198670-6/

Learning goals of this lecture

After successfully completing this lecture, you are able to….

- apply basic linear algebra operations and the respective python commands.
- explain the Jacobi and Gauss-Seidel methods for approximating the solution of linear systems of equation.
- explain the Newton-Raphson method for approximating the solution of nonlinear systems of equation.
- implement iterative methods to approximate the solutions of linear and nonlinear systems of equation in Python.

Thank you very much for your attention!