**NOTES**

**Lecture 1: Introduction: Computation, Programming and Chemical Engineers**

**Q. What is Computation:** the action of mathematical calculation.

**Q. History of Computers**

**1. Early Mechanical Aids (Before 1800s)**

* **Abacus (c. 2700–2300 BCE):** First known device for arithmetic, used in Mesopotamia, Egypt, China, and elsewhere.
* **Antikythera mechanism (c. 100 BCE):** Ancient Greek device to predict astronomical positions and eclipses.
* **Napier’s Bones (1617):** John Napier’s rods to simplify multiplication and division.
* **Pascaline (1642):** Blaise Pascal’s mechanical calculator for addition/subtraction.
* **Leibniz Step Reckoner (1673):** Gottfried Leibniz’s machine for multiplication/division.

**2. Pre-Computer Era (1800s)**

* **Jacquard Loom (1801):** Joseph Jacquard’s loom used punched cards to control weaving patterns → precursor to data storage.
* **Charles Babbage’s Difference Engine (1822):** Designed to calculate polynomial functions.

**Considered as a father of Computers**

* **Analytical Engine (1837):** Proposed general-purpose programmable computer; never built in his lifetime.
* **Ada Lovelace (1843):** Wrote the first algorithm intended for Babbage’s Analytical Engine →

**World’s first computer programmer.**

* **Hollerith Tabulating Machine (1890):** Herman Hollerith used punched cards for the U.S. Census → led to IBM (International Business Machines).

**3. First Generation (1940s–1950s) – Vacuum Tube Computers**

* **Electronic Numerical Integrator And Computer (ENIAC) (1945):** First general-purpose electronic computer; 18,000 vacuum tubes.
* **Electronic Discrete Variable Automatic Computer (EDVAC) (1949):** Introduced the stored-program concept (from John von Neumann).
* **Universal Automatic Computer (UNIVAC I) (1951):** First commercial computer sold in the U.S.
* Characteristics: Huge, expensive, used vacuum tubes, machine language, punch cards.

**4. Second Generation (1956–1963) – Transistors**

* Transistors replaced vacuum tubes → smaller, faster, more reliable.
* Used **assembly language** instead of just machine code.
* Examples: IBM 1401, IBM 7090, CDC 1604.

**5. Third Generation (1964–1971) – Integrated Circuits**

* Integrated Circuits (ICs) replaced transistors.
* Introduction of **operating systems** and **high-level languages** (FORTRAN, COBOL).
* Examples: IBM System/360, PDP-8.

**6. Fourth Generation (1971–Present) – Microprocessors**

* Microprocessors (Intel 4004 in 1971) allowed an entire CPU on a single chip.
* Rise of **personal computers (PCs)**:
  + Apple II (1977), IBM PC (1981), Macintosh (1984).
* Operating systems: MS-DOS, Windows, macOS, Linux.
* Networking → Internet revolution (1990s).

**7. Fifth Generation (Present & Future) – AI & Beyond**

* Based on **parallel processing, AI, quantum computing, nanotechnology**.
* Current trends:
  + **Cloud computing, IoT, supercomputers, AI systems.**
  + **Quantum computers** (Google, IBM, etc.) promising exponential power.

**Q. What is Chemical Engineering**

Chemical engineering is the branch of engineering that designs, develops, and operates industrial-scale processes that convert raw materials into useful products. It applies principles of chemistry, biology, physics, and mathematics to manufacture a vast range of products, from life-saving pharmaceuticals to everyday consumer goods.

**Chemical engineering vs. chemistry**

A key difference between chemical engineering and chemistry lies in the scale of their work.

* **A chemist** works in a laboratory to discover and study new molecules and chemical reactions, typically on a small scale.
* **A chemical engineer** takes these lab-scale discoveries and designs the commercial-scale processes and equipment needed to manufacture the products safely, economically, and efficiently.

**Core principles**

Chemical engineering utilizes fundamental concepts such as mass and energy balances, thermodynamics, fluid mechanics, transport phenomena, chemical reaction engineering, and unit operations to design and optimize processes.

**Applications and products**

Chemical engineers are crucial in producing various goods and technologies, including chemicals (fertilizers, plastics), energy (petroleum refining, biofuels), environmental solutions (waste treatment, sustainable manufacturing), food and consumer products, healthcare items (pharmaceuticals, vaccines), and materials for various industries.

**Key roles and duties**

Chemical engineers undertake roles such as designing processes for new or existing plants, overseeing and optimizing plant operations, conducting research and development to enhance processes and products, ensuring safety and compliance with regulations, and performing economic analysis to create profitable and efficient processes.

**Q. What is Programming Language**

A programming language is a set of specific instructions, a vocabulary, and grammatical rules that humans use to communicate with computers to create software, websites, and other applications. It acts as a tool for translating human intentions into commands that a machine can understand and execute, allowing us to build and control technology.

**Key applications of programming**

**Process simulation and modelling**

Programming allows chemical engineers to model and simulate complex chemical processes and equipment on a computer.

* **Predictive modelling:** By applying governing physical and mathematical models, engineers can simulate how a system will behave and predict outcomes, reducing the need for costly and time-consuming physical trial-and-error.
* **Customization:** While commercial software like Aspen Plus and CHEMCAD are widely used, coding enables engineers to build custom models or modify existing ones to fit unique process needs.
* **Dynamic simulation:** Programming is used for dynamic modelling to analyse and predict how processes will react to changes over time.

**Optimization**

Programming is a powerful tool for optimizing chemical processes to maximize efficiency, yield, and profit while minimizing costs.

* **Complex calculations:** Languages like MATLAB and Python are used to solve complex optimization problems that balance factors such as reaction rates, energy consumption, and raw material costs.
* **Advanced techniques:** With programming, engineers can implement advanced mathematical programming techniques like Mixed-Integer Non-Linear Programming (MINLP) for complex synthesis problems.

**Data analysis and machine learning**

The modern chemical industry generates massive amounts of data from sensors, experiments, and plant operations. Programming is essential for analyzing this "big data".

* **Data processing:** Scripts can be used to clean, organize, and process large datasets from disparate sources, a task that would be tedious and error-prone to perform manually.
* **Advanced analytics:** Machine learning algorithms can be applied to process data to predict equipment failures, optimize formulations, and troubleshoot operational problems.
* **Reproducibility:** A major benefit of using code for analysis is that it creates a reproducible record of the work, allowing collaborators or future team members to easily inspect and repeat the analysis.

**Automation and process control**

Chemical engineers use programming to automate plant operations and develop sophisticated control systems.

* **Custom controls:** Scripts can be written to tune controllers (like PID loops) for automated systems or to integrate different pieces of hardware and software.
* **Robotics:** In research and manufacturing, programming controls robotic systems that can automate repetitive tasks like mixing and material handling with high precision.
* **Digital twins:** The development of "digital twins"—virtual models of a physical plant—relies on programming and enables engineers to monitor and manage processes in real time.

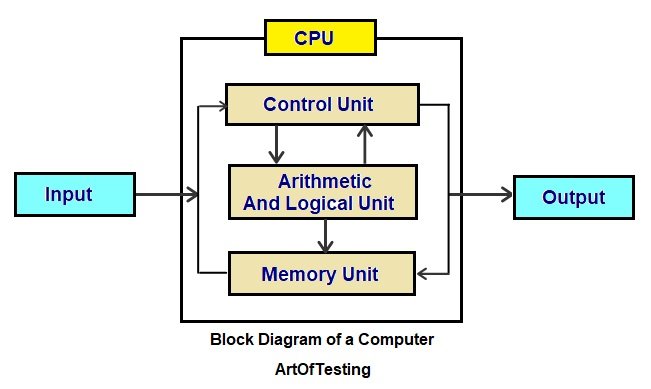
**Programming languages for chemical engineers**

A chemical engineer's choice of language often depends on the specific task, but some of the most beneficial options include:

* **Python:** A versatile, easy-to-learn, and widely used language. Its large ecosystem of libraries (like NumPy, SciPy, and Pandas) makes it excellent for numerical computations, data analysis, and machine learning.
* **MATLAB:** A popular, powerful tool for numerical analysis, visualization, and programming, especially for scientific and engineering purposes. SCILAB is Free Alternative of MATLAB.
* **C/C++/C#:** Often used for computationally intensive tasks like high-performance molecular simulations and for building specialized simulation software packages.
* **VBA (Visual Basic for Applications):** The macro programming language for Microsoft Excel, useful for automating repetitive tasks and calculations in spreadsheets.

**Lecture 2: Computer Hardware**

* **Definition**: Physical components of a computer system (contrast with software).

****

**Block Diagram of Computer: Explained with Components**

A block diagram of a computer shows its main parts and how they work together. It makes it easy to understand how data moves between the CPU, memory, input/output devices, and storage.  
  
The components in a block diagram of a computer typically include:

1. **Input Unit** – Devices like a keyboard and a mouse that send data to the computer.
2. **Central Processing Unit (CPU)** – The brain that processes data, including:
   * **Arithmetic Logic Unit (ALU)** – Performs calculations and logical operations.
   * **Control Unit (CU)** – Manages data flow and instructions.
   * **Memory Unit**: Very fast but small storage (registers & cache)
3. **External Memory Storage** – Stores data and instructions, including:
   * **Primary Memory (RAM)** – Temporary storage for active processes.

* A separate hardware component, connected to CPU via the **memory bus**.
* Temporarily stores data and instructions while the computer is running.
* Volatile → data is lost when power is off.
* Larger than cache/registers but slower.
  + **Secondary Memory (HDD/SSD)** – Permanent storage for data.

**Hierarchy of Memory**

1. **Registers (inside CPU)** → fastest, smallest.
2. **Cache (inside or very close to CPU)** → very fast, small.
3. **RAM (outside CPU, but connected)** → slower, bigger.
4. **Storage (HDD/SSD)** → much slower, much larger.
5. **Output Unit** – Devices like monitors and printers that display or produce results.

These components are connected to show data flow (input → CPU → memory → output).

**Input**

All the data received by the computer goes through the [input unit](https://artoftesting.com/input-devices-of-computer). The input unit comprises different devices like a mouse, keyboard, scanner, etc. In other words, each of these devices acts as a mediator between the users and the computer.

The data that is to be processed is put through the input unit. The computer accepts the raw data in binary form. It then processes the data and produces the desired output.

#### **Major Functions of the Input Unit**

The 3 major functions of the input unit are-

* Take the data to be processed by the user.
* Convert the given data into machine-readable form.
* And then, transmit the converted data into the main memory of the computer. The sole purpose is to connect the user and the computer. In addition, this creates easy communication between them.

**CPU – Central Processing Unit**

* Central Processing Unit or the CPU, is the brain of the computer. It works the same way a human brain works. As the brain controls all human activities, similarly the CPU controls all the tasks. Moreover, the CPU conducts all the arithmetical and logical operations in the computer. **Definition**: Electronic circuit that executes instructions of a computer program.
* **Major components**:
  1. **Control Unit (CU):** Directs the flow of data/instructions, manages fetch–decode–execute cycle.
  2. **Arithmetic Logic Unit (ALU):** Performs mathematical (addition, subtraction, multiplication, division) and logical (AND, OR, NOT, comparisons) operations.
  3. **Registers:** Small, high-speed memory locations (e.g., accumulator, instruction register, program counter).
  4. **Cache:** Very fast memory closer to CPU than RAM.
* **Functions of CPU**:
  1. Fetch instructions from memory.
  2. Decode the instruction.
  3. Execute operation via ALU.
  4. Store result back in memory/register.
* **Analogy**: CPU = human brain (CU = mind, ALU = calculator, Registers = short-term memory).

Now the CPU comprises of two units, namely – ALU (Arithmetic Logic Unit) and CU (Control Unit). Both of these units work in sync. The CPU processes the data as a whole.

Let us see what particular tasks are assigned to both units.

### **ALU** **– Arithmetic Logic Unit**

* **Definition**: Subsystem of CPU that performs all arithmetic and logic operations.
* **Operations**:
  + **Arithmetic**: +, −, ×, ÷, increment, decrement.
  + **Logic**: <, >, =, AND, OR, NOT.
* **Inputs & Outputs**:
  + Takes data from registers, processes it, sends result back to registers/memory.
* **Example**:
  + Instruction: A + B → CU fetches A & B → ALU adds them → result stored in accumulator.

The Arithmetic Logic Unit is made of two terms, arithmetic and logic. There are two primary functions that this unit performs.

1. Data is inserted through the input unit into the primary memory. Performs the basic arithmetical operations on it, like addition, subtraction, multiplication, and division. It performs all sorts of calculations required on the data. Then, it sends back data to the storage.
2. The unit is also responsible for performing logical operations like AND, OR, Equal to, Less than, etc.  In addition to this, it conducts merging, sorting, and selection of the given data.

### **CU – Control Unit**

The control unit as the name suggests is the controller of all the activities/tasks and operations. All this is performed inside the computer.

The memory unit sends a set of instructions to the control unit. Then the control unit in turn converts those instructions. After that these instructions are converted to control signals.

These control signals help in prioritizing and scheduling activities. Thus, the control unit coordinates the tasks inside the computer in sync with the input and output units.

### **Memory Unit**

All the data that has to be processed or has been processed is stored in the memory unit. The memory unit acts as a hub of all the data. It transmits it to the required part of the computer whenever necessary.

The memory unit works in sync with the CPU. This helps in faster accessing and processing of the data. Thus, making tasks easier and quicker.

#### **Types of Computer Memory**

There are two types of computer memory-

##### **Primary Memory**

This type of memory cannot store a vast amount of data. Therefore, it is only used to store recent data. The data stored in this is temporary. It can get erased once the power is switched off. Therefore, is also called temporary memory or main memory.

RAM stands for Random Access Memory. It is an example of primary memory. This memory is directly accessible by the CPU. It is used for reading and writing purposes. For data to be processed, it has to be first transferred to the RAM and then to the CPU.

##### **Secondary Memory**

As explained above, the primary memory stores temporary data. Thus it cannot be accessed in the future. For permanent storage purposes, secondary memory is used. It is also called permanent memory or auxiliary memory. The hard disk is an example of secondary memory. Even in a power failure data does not get erased easily.

**Output**

There is nothing to be amazed by what the output unit is used for. All the information sent to the computer once processed is received by the user through the output unit. Devices like printers, monitors, projectors, etc. all come under the output unit.

The output unit displays the data either in the form of a soft copy or a hard copy. The printer is for the hard copy. The monitor is for the display. The output unit accepts the data in binary form from the computer. It then converts it into a readable form for the user.

**4. Microprocessors**

* **Definition**: A CPU on a single chip.
* **History**:
  + Intel 4004 (1971): First microprocessor (4-bit).
  + Intel 8086 (1978): Basis of modern PCs.
* **Characteristics**:
  + Contains ALU, CU, registers, cache on one chip.
  + Executes millions of instructions per second.
* **Generations of microprocessors**:
  + 4-bit (Intel 4004)
  + 8-bit (Intel 8080)
  + 16-bit (Intel 8086, 80286)
  + 32-bit (Intel 80386, 80486)
  + 64-bit (modern processors: Intel Core, AMD Ryzen).
* **Applications**: PCs, smartphones, cars, washing machines, IoT devices.

**5. Integrated Circuits (ICs)**

* **Definition**: Electronic circuit built on a small semiconductor (usually silicon) chip.
* **Components inside ICs**: Transistors, resistors, capacitors, diodes.
* **Types of ICs**:
  + **SSI (Small Scale Integration):** <100 transistors (logic gates).
  + **MSI (Medium Scale Integration):** 100–1000 transistors.
  + **LSI (Large Scale Integration):** 1000–100,000 transistors (microprocessors).
  + **VLSI (Very Large Scale Integration):** >100,000 transistors (modern CPUs, GPUs).
  + **ULSI (Ultra-Large Scale Integration):** Millions/billions of transistors.
* **Importance**:
  + Enabled miniaturization of computers.
  + Reduced cost, increased speed, improved reliability.
* **Real-world examples**:
  + Microchips in smartphones, RAM, GPUs, processors.

**Lecture 3: Programming basics: variables, constants, data types**

## **1. Introduction to Programming**

* **What is programming?**
  + Process of writing instructions for a computer to perform specific tasks.
* **Why basics matter?**
  + Variables, constants, and data types are the foundation of all programming languages (Python, SCILAB, C, Java, etc.).
* **Relevance for SCILAB & Python:**
  + Both languages rely on the same concepts but with different syntax.

## **2. Variables**

### **Definition**

* A variable is a **named storage location in memory** used to hold data that can change during program execution.
* Think of it as a "box" with a label (the variable name).

### **Properties**

* Must have a name (identifier).
* Must have a data type (depends on language).
* Stores a value, which can change.

### **Rules for naming variables**

* Should start with a letter (a–z, A–Z) or underscore \_.
* Cannot start with a number.
* Cannot use reserved keywords (if, for, while, etc.).
* Should be meaningful (e.g., temperature, not x1).

### **Examples**

* **Python**:
* age = 25
* name = "Alice"
* temperature = 36.5
* **SCILAB**:
* age = 25;
* name = "Alice";
* temperature = 36.5;

## **3. Constants**

### **Definition**

* Constants are values **that do not change** during program execution.

### **Types of constants**

1. **Numeric constants**: 10, 3.14, -5
2. **Character constants**: 'a', 'Z'
3. **String constants**: "Hello", "SCILAB", "Python"
4. **Boolean constants**: True, False

### **Examples**

* **Python**:
* PI = 3.14159 # convention: constants written in UPPERCASE
* GRAVITY = 9.8
* **SCILAB**:
* PI = 3.14159;
* GRAVITY = 9.8;

## **4. Data Types**

### **Definition**

* Data type defines the **kind of value** a variable can hold and the **operations** allowed on it.

### **Common Data Types**

1. **Integer** – whole numbers (e.g., 5, -23, 1000)
   * Python: int
   * SCILAB: default numeric type is double, but integers can be represented.
2. **Float/Real numbers** – decimal values (e.g., 3.14, -0.001, 9.81)
   * Python: float
   * SCILAB: 1.5, -3.2
3. **String** – sequence of characters (e.g., "Hello")
   * Python: "Hello", 'World'
   * SCILAB: "Hello"
4. **Boolean** – truth values (True/False)
   * Used in conditions & logic.
5. **Complex numbers** – numbers with real & imaginary parts.
   * Python: 3 + 4j
   * SCILAB: 3 + 4\*%i

### **Examples**

* **Python**:
* a = 10 # integer
* b = 3.14 # float
* c = "Hello" # string
* d = True # boolean
* e = 2 + 3j # complex
* **SCILAB**:
* a = 10; // integer
* b = 3.14; // float
* c = "Hello"; // string
* d = %T; // boolean true
* e = 2 + 3\*%i; // complex

## **5. Difference between Variables and Constants**

| **Feature** | **Variable** | **Constant** |
| --- | --- | --- |
| Value changes? | Yes | No |
| Declaration | Normal assignment | Fixed/predefined value |
| Example | x = 5 → later x = 10 | PI = 3.14 (unchanged) |

## **6. Summary & Closing**

* **Variables** → Named memory locations for storing data.
* **Constants** → Fixed values that don’t change.
* **Data Types** → Define the kind of data stored (int, float, string, boolean, complex).
* These concepts are **language-independent** → same in SCILAB, Python, C, etc.

1. **Quick Quiz:**
   * Is PI = 3.14 a variable or constant?
   * Which data type is "123"?
2. **Live Demo in Python & SCILAB:**
   * Declare different variables.
   * Change a variable’s value.
   * Print constant values.
   * Show error if syntax rules are broken.

**Lecture 4: Input/Output Operations in SCILAB/Python**

* **Why I/O is important?**
  + Programs are not useful unless they can **interact with users or other systems**.
  + **Input**: Data given to a program (keyboard, files, sensors).
  + **Output**: Data produced by a program (screen, files, devices).
* Both **SCILAB** and **Python** provide simple functions for I/O.

### **PYTHON**

* **Basic printing**:
* print("Hello, World")
* print("Value of x is", 10)
* print("Sum =", 5 + 3)
* **Formatted printing**:
* name = "Alice"
* age = 25
* print(f"My name is {name} and I am {age} years old") # f-string
* print("Value of pi = {:.2f}".format(3.14159)) # format method
* **Multiple outputs**:
* x, y = 10, 20
* print("x =", x, " y =", y)

### **SCILAB**

* **Basic display**:
* disp("Hello, World");
* x = 10;
* disp(x);
* **Concatenating text and numbers**:
* name = "Alice";
* age = 25;
* disp("Name: " + name);
* disp("Age: " + string(age));
* **Formatted output (using printf)**:
* printf("My name is %s and I am %d years old\n", name, age);
* printf("Value of pi = %.2f\n", %pi);

## **3. Input Operations**

### **PYTHON**

* **Basic input**:
* name = input("Enter your name: ")
* print("Hello", name)
* **Numeric input (casting required)**:
* age = int(input("Enter your age: "))
* pi = float(input("Enter value of pi: "))
* print("Next year you will be", age + 1)
* **Example program**:
* a = int(input("Enter first number: "))
* b = int(input("Enter second number: "))
* print("Sum =", a + b)

### **SCILAB**

* **Basic input**:
* name = input("Enter your name: ", "string");
* disp("Hello " + name);
* **Numeric input**:
* age = input("Enter your age: ");
* disp("Next year you will be " + string(age + 1));
* **Example program**:
* a = input("Enter first number: ");
* b = input("Enter second number: ");
* printf("Sum = %d\n", a + b);

| **Feature** | **Python** | **SCILAB** |
| --- | --- | --- |
| Print text | print("Hello") | disp("Hello"); or printf("Hello\n"); |
| Print variable | print("x =", x) | disp(x); or printf("x = %d", x); |
| Input string | name = input("Enter name: ") | name = input("Enter name: ", "string"); |
| Input integer/float | int(input(...)) or float(...) | input("Enter number: "); |
| String formatting | f-strings, .format() | printf with %d, %f, %s specifiers |

## **Classroom Activities**

1. **Python exercise**:
   * Ask the user for their name and age, then print:  
     "Hello Alice, you are 25 years old."
2. **SCILAB exercise**:
   * Ask the user for two numbers and display their sum, product, and average.

**Lecture 5: Input/Output Operations in SCILAB/Python**

* **Definition**: Operators are special symbols that perform operations on variables and values.
* **Operands**: The data on which an operator acts.
  + Example: in a + b, + is operator, a and b are operands.
* **Categories we focus on**:
  + Arithmetic operators
  + Relational (comparison) operators
  + Logical operators

**Arithmetic operator:**

* **Python**

a, b = 10, 3

print(a + b) # 13

print(a / b) # 3.333...

print(a // b) # 3

print(a % b) # 1

print(a \*\* b) # 1000

* **SCILAB**

| **Operator** | **Meaning** | **Example** | **Result** |
| --- | --- | --- | --- |
| + | Addition | 5 + 3 | 8 |
| - | Subtraction | 5 - 3 | 2 |
| \* | Multiplication | 5 \* 3 | 15 |
| / | Right division | 5 / 2 | 2.5 |
| \ | Left division | 2 \ 4 | 2 |
| ^ | Power | 2^3 | 8 |
| mod() | Modulus function | mod(5,2) | 1 |

**Relational Operators**

**Python**

| **Operator** | **Meaning** | **Example** | **Result** |
| --- | --- | --- | --- |
| == | Equal to | 5 == 5 | True |
| != | Not equal to | 5 != 3 | True |
| > | Greater than | 5 > 3 | True |
| < | Less than | 5 < 3 | False |
| >= | Greater or equal | 5 >= 5 | True |
| <= | Less or equal | 3 <= 5 | True |

**SCILAB**

| **Operator** | **Meaning** | **Example** | **Result** |
| --- | --- | --- | --- |
| == | Equal to | 5 == 5 | %T |
| ~= | Not equal to | 5 ~= 3 | %T |
| > | Greater than | 5 > 3 | %T |
| < | Less than | 5 < 3 | %F |
| >= | Greater or equal | 5 >= 5 | %T |
| <= | Less or equal | 3 <= 5 | %T |

**Logical Operators**

**Python**

| **Operator** | **Meaning** | **Example** | **Result** |
| --- | --- | --- | --- |
| and | True if both true | (x>5 and y<10) | True/False |
| or | True if at least one | (x>5 or y<10) | True/False |
| not | Negation | not(x>5) | True/False |

**SCILAB**

| **Operator** | **Meaning** | **Example** | **Result** |
| --- | --- | --- | --- |
| & | Logical AND | (x>5 & y<10) | %T/%F |
| ` | ` | Logical OR | `(x>5 |
| ~ | Logical NOT | ~(x>5) | %T/%F |

 **Python:**

* Input two numbers and check if the first is greater than the second **and** if both are even.

 **SCILAB:**

* Input a number and check if it is positive, negative, or zero using relational & logical operators.

 **Challenge:**

* Write a condition that checks if a student’s marks are between 40 and 100 (inclusive).

**Lecture 6: Flowcharts, pseudocode, problem solving approach**

## **1. Introduction**

* Programming is not just about writing code → it’s about solving problems systematically.
* To solve a problem effectively, we use:
  1. **Problem-solving approach** (understand, plan, implement, test).
  2. **Pseudocode** (structured English-like steps).
  3. **Flowcharts** (graphical representation of logic).

**Objective of the lecture:**

* Learn the problem-solving cycle.
* Understand pseudocode.
* Represent logic with flowcharts.

## **2. Problem-Solving Approach in Programming**

Steps followed by programmers:

1. **Problem Definition**
   * Understand what is being asked.
   * Define inputs, outputs, and constraints.
   * Example: Find the sum of two numbers.
     + Input: two numbers.
     + Output: their sum.
2. **Analysis**
   * Break down problem into smaller parts.
   * Identify operations needed.
3. **Algorithm Design**
   * Step-by-step procedure to solve the problem.
4. **Implementation**
   * Write the algorithm in a programming language (Python, SCILAB).
5. **Testing & Debugging**
   * Check if the program gives correct results.
6. **Documentation & Maintenance**
   * Write comments, explain code, and update when needed.

## **3. Pseudocode**

* **Definition:** A simplified, English-like representation of an algorithm.
* **Not a programming language** → no strict syntax.
* **Helps bridge the gap** between algorithm and real code.

**Characteristics:**

* Easy to understand.
* Independent of programming language.
* Uses keywords like INPUT, OUTPUT, BEGIN, END, IF, WHILE, FOR.

### **Examples**

1. **Add two numbers**

BEGIN

INPUT A, B

SUM = A + B

OUTPUT SUM

END

1. **Find largest of two numbers**

BEGIN

INPUT A, B

IF A > B THEN

OUTPUT A

ELSE

OUTPUT B

ENDIF

END

1. **Find factorial of a number**

BEGIN

INPUT N

FACT = 1

FOR I = 1 TO N DO

FACT = FACT \* I

ENDFOR

OUTPUT FACT

END

## **4. Flowcharts**

* **Definition:** Diagrammatic representation of a program’s logic using standard symbols.
* **Why flowcharts?**
  + Easy to visualize logic.
  + Good for planning and communication.

### **Basic Symbols**

* **Oval (Ellipse):** Start/End.
* **Parallelogram:** Input/Output.
* **Rectangle:** Process/Calculation.
* **Diamond:** Decision/Condition.
* **Arrows:** Flow of control.

### **Examples**

1. **Add two numbers**

[Start] → [Input A, B] → [Sum = A+B] → [Output Sum] → [End]

1. **Check even or odd**

[Start] → [Input N] → [N % 2 == 0?] → Yes → [Output "Even"]

No → [Output "Odd"] → [End]

1. **Factorial (loop)**

[Start] → [Input N] → [FACT = 1, I = 1] → [I <= N?] → Yes → [FACT = FACT\*I] → [I = I+1] → loop back

No → [Output FACT] → [End]

## **5. Comparison: Pseudocode vs Flowchart**

| **Aspect** | **Pseudocode** | **Flowchart** |
| --- | --- | --- |
| Format | Textual (English-like) | Graphical |
| Clarity | Easy to write quickly | Easy to visualize |
| Best for | Detailed stepwise design | Overview of logic |
| Flexibility | More detailed | Less detailed for large problems |

## **6. Classroom Activity / Examples**

* Students design **flowchart + pseudocode** for:
  1. Calculating area of a circle.
  2. Checking whether a number is positive, negative, or zero.
  3. Printing first 10 natural numbers.

**Lecture 7: First programs: unit conversion problems**

## **1. Introduction**

* Transition from planning tools (flowcharts & pseudocode) → **actual coding**.
* First programs should be **simple, practical, and relatable**.
* Unit conversions are **everyday problems** (temperature, length, weight, time, etc.).

**Objectives:**

* Write simple programs in SCILAB and Python.
* Practice using variables, input/output, arithmetic operations.
* Build problem-solving confidence.

## **2. Steps in Writing a Program**

1. Define the problem clearly (what input? what output?).
2. Write pseudocode / flowchart.
3. Implement in code.
4. Test with different inputs.

Example: Convert Celsius → Fahrenheit.

* Input: Celsius temperature.
* Formula: F = (C \* 9/5) + 32.
* Output: Fahrenheit value.

## **3. Example 1: Temperature Conversion**

### **Pseudocode**

BEGIN

INPUT Celsius

Fahrenheit = (Celsius \* 9/5) + 32

OUTPUT Fahrenheit

END

### **Flowchart (explain on board/slide)**

[Start] → [Input Celsius] → [F = (C\*9/5)+32] → [Output F] → [End]

### **PYTHON**

# Celsius to Fahrenheit

celsius = float(input("Enter temperature in Celsius: "))

fahrenheit = (celsius \* 9/5) + 32

print("Temperature in Fahrenheit:", fahrenheit)

### **SCILAB**

// Celsius to Fahrenheit

celsius = input("Enter temperature in Celsius: ");

fahrenheit = (celsius \* 9/5) + 32;

disp(fahrenheit, "Temperature in Fahrenheit:");

## **4. Example 2: Length Conversion**

* Convert kilometers → miles.
* Formula: miles = km \* 0.621371.

### **PYTHON**

km = float(input("Enter distance in kilometers: "))

miles = km \* 0.621371

print("Distance in miles:", miles)

### **SCILAB**

km = input("Enter distance in kilometers: ");

miles = km \* 0.621371;

disp(miles, "Distance in miles:");

## **5. Example 3: Weight Conversion**

* Convert kilograms → pounds.
* Formula: pounds = kg \* 2.20462.

### **PYTHON**

kg = float(input("Enter weight in kilograms: "))

pounds = kg \* 2.20462

print("Weight in pounds:", pounds)

### **SCILAB**

kg = input("Enter weight in kilograms: ");

pounds = kg \* 2.20462;

disp(pounds, "Weight in pounds:");

## **6. Example 4: Time Conversion**

* Convert minutes → hours.

### **PYTHON**

minutes = int(input("Enter time in minutes: "))

hours = minutes / 60

print("Time in hours:", hours)

### **SCILAB**

minutes = input("Enter time in minutes: ");

hours = minutes / 60;

disp(hours, "Time in hours:");

Ask students to implement on their own:

1. Convert meters → centimetres.
2. Convert litters → millilitres.
3. Convert Fahrenheit → Celsius.

**Lecture 8: Conditional Statements in SCILAB & Python**

## **1. Introduction**

* **Why conditional statements?**
  + Programs often need to **make decisions** based on input or computation.
  + Conditional statements allow the program to **execute different actions depending on conditions**.
* **Decision-making in programming:**
  + Example: Check if a number is positive, negative, or zero.
  + Input → Decision → Output

## **2. Types of Conditional Statements**

1. **Simple if statement** → Executes a block if condition is True.
2. **if-else statement** → Executes one block if True, another if False.
3. **if-elseif-else / nested if** → Multiple conditions are tested in sequence.

## **3. Syntax**

### **Python**

if condition:

# code block

elif condition2:

# code block

else:

# code block

### **SCILAB**

if condition then

// code block

elseif condition2 then

// code block

else

// code block

end

**Notes:**

* Python uses **colon : and indentation**.
* SCILAB uses **then and end** to mark blocks.

## **4. Simple** if **Statement**

**Python Example: Check positive number**

num = float(input("Enter a number: "))

if num > 0:

print("Number is positive")

**SCILAB Example:**

num = input("Enter a number: ");

if num > 0 then

disp("Number is positive");

end

## **5. if-else Statement**

**Python Example: Check even or odd**

num = int(input("Enter a number: "))

if num % 2 == 0:

print("Even number")

else:

print("Odd number")

**SCILAB Example:**

num = input("Enter a number: ");

if mod(num,2) == 0 then

disp("Even number");

else

disp("Odd number");

end

## **6. if-elseif-else Statement / Multiple Conditions**

**Python Example: Check sign of number**

num = float(input("Enter a number: "))

if num > 0:

print("Positive number")

elif num < 0:

print("Negative number")

else:

print("Zero")

**SCILAB Example:**

num = input("Enter a number: ");

if num > 0 then

disp("Positive number");

elseif num < 0 then

disp("Negative number");

else

disp("Zero");

end

**Key Points:**

* Multiple conditions tested in order.
* Only the **first True condition executes**.

## **7. Nested if Statements**

**Python Example:** Check age category

age = int(input("Enter age: "))

if age >= 0:

if age < 13:

print("Child")

elif age < 20:

print("Teenager")

else:

print("Adult")

else:

print("Invalid age")

**SCILAB Example:**

age = input("Enter age: ");

if age >= 0 then

if age < 13 then

disp("Child");

elseif age < 20 then

disp("Teenager");

else

disp("Adult");

end

else

disp("Invalid age");

end

**Exercise:**

1. Input marks and print grade based on:
   * = 90 → A
   * = 80 → B
   * = 70 → C
   * else → F
2. Input temperature and print:
   * 40 → “Hot”
   * 20–40 → “Moderate”
   * <20 → “Cold”
3. Check whether a year is **leap year** or not.

**Lecture 9: Nested Conditionals – Chemical Property Checks (Gas vs Liquid Phase)**

## **1. Introduction**

* **Objective:** Learn how to use **nested conditional statements** for more complex decision-making.
* **Why nested conditionals?**
  + Simple if or if-else handles only one condition at a time.
  + Nested if allows **multiple levels of decision-making**.
* **Application in chemical engineering:**
  + Determine **phase of a substance** based on **temperature** and **pressure**.

## **2. Concept of Nested Conditionals**

* **Definition:** Conditional statements placed **inside another conditional statement**.
* **Structure:**
  + Outer condition → evaluates a primary property.
  + Inner condition → further checks another property if the outer condition is True.
* **Advantages:**
  + Handles **hierarchical decisions**.
  + Avoids overly long logical expressions in a single if.

## **3. Example: Check Physical State of Water (Temperature & Pressure)**

* **Scenario:**
  + Input: Temperature (T) in °C and Pressure (P) in atm.
  + Output: Phase (Solid, Liquid, Gas)
* **Decision Rules (Simplified):**
  + At 1 atm:
    - T ≤ 0 → Solid (Ice)
    - 0 < T < 100 → Liquid (Water)
    - T ≥ 100 → Gas (Steam)
  + Optional: Extend to check **supercritical conditions**.

## **4. Pseudocode**

BEGIN

INPUT Temperature, Pressure

IF Pressure == 1 THEN

IF Temperature <= 0 THEN

OUTPUT "Solid phase (Ice)"

ELSEIF Temperature < 100 THEN

OUTPUT "Liquid phase (Water)"

ELSE

OUTPUT "Gas phase (Steam)"

ENDIF

ELSE

OUTPUT "Phase determination at this pressure requires advanced calculation"

ENDIF

END

## **5. Flowchart (Optional Visual)**

* Start → Input T, P → Check P=1 atm?
  + Yes → Nested check on T: ≤0 → Solid; 0–100 → Liquid; ≥100 → Gas
  + No → Output: “Advanced calculation required” → End

## **6. Python Implementation**

# Phase check for water at 1 atm

T = float(input("Enter temperature (°C): "))

P = float(input("Enter pressure (atm): "))

if P == 1:

if T <= 0:

print("Solid phase (Ice)")

elif T < 100:

print("Liquid phase (Water)")

else:

print("Gas phase (Steam)")

else:

print("Phase determination at this pressure requires advanced calculation")

**Key points:**

* Outer if → Pressure check
* Inner if-elif-else → Temperature-based phase determination

## **7. SCILAB Implementation**

T = input("Enter temperature (°C): ");

P = input("Enter pressure (atm): ");

if P == 1 then

if T <= 0 then

disp("Solid phase (Ice)");

elseif T < 100 then

disp("Liquid phase (Water)");

else

disp("Gas phase (Steam)");

end

else

disp("Phase determination at this pressure requires advanced calculation");

end

## **8. Extended Example: Multiple Substances**

* Extend the program to check **phase of ethanol, water, or nitrogen**:
  + Each substance has different melting & boiling points.
  + Outer if → substance type
  + Inner if → temperature comparison for that substance
* **Illustrates:**
  + Nested conditions
  + Multiple decision layers

## **9. Classroom Activity / Exercises**

1. Input **temperature & substance** → determine **phase**.
   * Water: 0–100 °C
   * Ethanol: -114–78 °C
   * Nitrogen: -210–-196 °C
2. Extend **pressure check** to allow simple high-pressure warning.
3. Compare **Python vs SCILAB implementation** and discuss syntax differences.

## **10. Summary / Key Points**

* Nested conditionals allow **multi-level decision-making**.
* **Outer if** → primary property check (e.g., substance or pressure).
* **Inner if-elif-else** → secondary check (e.g., temperature ranges).
* Applications in chemical engineering:
  + Phase determination
  + Safety checks (temperature/pressure limits)
  + Reaction feasibility checks

**Lecture 10: Loops – For Loop Basics (Summation & Factorial)**

## **1. Introduction**

* **Why loops?**
  + Many programming problems require **repeating a set of instructions** multiple times.
  + Writing repetitive code manually is inefficient.
* **Types of loops:**
  + **For loop** – iterate a specific number of times.
  + **While loop** – iterate while a condition is true (covered later).
* **Applications:** Summation, factorial, printing sequences, cumulative calculations.

## **2. Concept of** for **Loop**

* **Definition:** Executes a block of code for a **fixed number of iterations**.
* **Structure:**
  + Initialize a counter
  + Test condition / range
  + Increment counter after each iteration

## **3. Syntax: Python vs SCILAB**

### **Python**

for variable in range(start, stop, step):

# code block

* start → starting value (default 0)
* stop → end value (not included)
* step → increment (default 1)

### **SCILAB**

for variable = start:step:stop

// code block

end

* start:step:stop defines the loop range
* end marks the end of the loop

## **4. Example 1: Summation of First N Natural Numbers**

**Problem:** Calculate sum of first N numbers

### **Python**

N = int(input("Enter N: "))

sum = 0

for i in range(1, N+1):

sum += i # sum = sum + i

print("Sum of first", N, "natural numbers is:", sum)

### **SCILAB**

N = input("Enter N: ");

sum = 0;

for i = 1:N

sum = sum + i;

end

disp(sum, "Sum of first " + string(N) + " natural numbers:");

**Explanation:**

* Loop variable i runs from 1 → N
* Each iteration adds i to sum

## **5. Example 2: Factorial of a Number**

**Problem:** Calculate N! = 1 × 2 × ... × N

### **Python**

N = int(input("Enter a number: "))

fact = 1

for i in range(1, N+1):

fact \*= i # fact = fact \* i

print("Factorial of", N, "is:", fact)

### **SCILAB**

N = input("Enter a number: ");

fact = 1;

for i = 1:N

fact = fact \* i;

end

disp(fact, "Factorial of " + string(N) + ":");

**Explanation:**

* Loop variable multiplies sequentially into fact
* Works for N ≥ 0

## **6. Example 3: Print Multiplication Table**

### **Python**

N = int(input("Enter number: "))

for i in range(1, 11):

print(N, "x", i, "=", N\*i)

### **SCILAB**

N = input("Enter number: ");

for i = 1:10

printf("%d x %d = %d\n", N, i, N\*i);

end

**Purpose:** Shows loops for **repetitive display operations**.

## **7. Key Points**

* **for loop** iterates a **known number of times**.
* Loop variable changes automatically.
* Common uses in chemical engineering: summation of series, factorial, arrays, time steps in simulations.
* Python uses range() function; SCILAB uses start:step:end.

## **8. Classroom Activities / Exercises**

1. Calculate sum of **even numbers** up to N.
2. Compute factorial of a given number **and check if it exceeds 1000**.
3. Print first N **squares and cubes** of natural numbers.
4. Generate **multiplication table** for numbers 1–5.

## **9. Summary / Conclusion**

* Loops allow **repetition of tasks efficiently**.
* **For loops** are ideal for **fixed iterations**.
* Practical applications: summation, factorial, printing sequences, and array operations.
* Essential foundation for **nested loops, simulations, and iterative calculations** in chemical engineering and scientific programming.

**Lecture 11: While Loops & Iterative Calculations – Newton–Raphson Method**

## **1. Introduction**

* **Why loops are important?**
  + Many engineering problems require **repeating calculations until a condition is satisfied**.
  + Examples: convergence in iterative methods, solving nonlinear equations.
* **While loops** are ideal for **unknown number of iterations**.
* **Application in chemical engineering:**
  + Solving equations for reaction conversion, phase equilibria, or root finding.
  + Newton–Raphson method: find roots of nonlinear equations iteratively.

## **2. While Loop Concept**

* **Definition:** Executes a block of code repeatedly **while a condition is True**.
* **Structure:**

### **Python**

while condition:

# code block

### **SCILAB**

while condition do

// code block

end

**Key point:**

* Condition is **evaluated before each iteration**.
* Loop continues **until condition becomes False**.

## **3. Simple Example: Sum of Numbers Until User Enters Zero**

### **Python**

total = 0

num = int(input("Enter a number (0 to stop): "))

while num != 0:

total += num

num = int(input("Enter a number (0 to stop): "))

print("Total sum:", total)

### **SCILAB**

total = 0;

num = input("Enter a number (0 to stop): ");

while num <> 0 do

total = total + num;

num = input("Enter a number (0 to stop): ");

end

disp(total, "Total sum:");

**Explanation:**

* Loop executes **until user enters 0**.
* Example shows **unknown number of iterations**.

## **4. Introduction to Newton–Raphson Method**

* **Purpose:** Solve nonlinear equations of the form f(x)=0f(x) = 0
* **Idea:** Start with an initial guess x0x\_0 and iterate:

xn+1=xn−(f(xn)/(f′(xn)))

* Stop when: ∣xn+1−xn∣<tolerance
* **Applications in chemical engineering:**
  + Solving reaction rate equations
  + Equilibrium compositions
  + Phase fraction calculations

## **5. Pseudocode for Newton–Raphson**

BEGIN

INPUT initial guess x0

INPUT tolerance

LOOP

f\_x = f(x0)

f\_prime = derivative f'(x0)

x1 = x0 - f\_x / f\_prime

IF |x1 - x0| < tolerance THEN

OUTPUT x1

EXIT LOOP

ENDIF

x0 = x1

END LOOP

END

## **6. Python Implementation**

# Newton-Raphson example: solve x^2 - 2 = 0

def f(x):

return x\*\*2 - 2

def f\_prime(x):

return 2\*x

x0 = float(input("Enter initial guess: "))

tolerance = 1e-6

max\_iter = 100

iteration = 0

while iteration < max\_iter:

x1 = x0 - f(x0)/f\_prime(x0)

if abs(x1 - x0) < tolerance:

break

x0 = x1

iteration += 1

print("Root:", x1)

print("Iterations:", iteration)

**Key points:**

* Iterative loop continues **until tolerance is met**.
* Avoids specifying number of iterations in advance.

### **SCILAB**

function y = f(x)

y = x^2 - 2;

endfunction

function y = f\_prime(x)

y = 2\*x;

endfunction

x0 = input("Enter initial guess: ");

tolerance = 1e-6;

max\_iter = 100;

iteration = 0;

while iteration < max\_iter do

x1 = x0 - f(x0)/f\_prime(x0);

if abs(x1 - x0) < tolerance then

break;

end

x0 = x1;

iteration = iteration + 1;

end

disp(x1, "Root:");

disp(iteration, "Iterations:");

## **8. Classroom Exercises / Examples**

1. Solve x3−x−2=0x^3 - x - 2 = 0 using Newton–Raphson in Python and SCILAB.
2. Iteratively calculate **square root of a number** using the formula xn+1=0.5(xn+S/xn).
3. Implement a **loop to sum numbers until cumulative sum exceeds 1000**.

## **9. Summary / Key Points**

* **While loops**: repeat **until a condition is False** → suitable for iterative calculations.
* **Newton–Raphson method**: iterative root-finding method.
* **Structure:**
  + Start with initial guess
  + Compute next iteration
  + Check **convergence (tolerance)**
* **Practical relevance:** solves nonlinear chemical engineering problems efficiently.

**Lecture 12: Loop Control – Break and Continue Statement**

## **1. Introduction**

* **Loops** allow repetitive execution of code.
* Sometimes, we need **more control** over loop execution:
  + **Exit a loop prematurely** → break
  + **Skip certain iterations** → continue
* **Applications in chemical engineering:**
  + Stop a simulation when **convergence is reached**
  + Skip invalid inputs in iterative calculations
  + Break out of large loops to **save computation time**

## **2. Concept of** break **Statement**

* **Definition:** Terminates the **innermost loop immediately**.
* Execution continues **after the loop**.

### **Python**

for i in range(1, 10):

if i == 5:

break

print(i)

**Output:** 1 2 3 4

### **SCILAB**

for i = 1:9

if i == 5 then

break;

end

disp(i)

end

## **3. Concept of** continue **Statement**

* **Definition:** Skips the **rest of the code in current iteration** and moves to the next iteration.

### **Python**

for i in range(1, 6):

if i == 3:

continue

print(i)

**Output:** 1 2 4 5

### **SCILAB**

for i = 1:5

if i == 3 then

continue;

end

disp(i)

end

**Key points:**

* Skips only **current iteration**; loop continues.
* Useful to **ignore invalid inputs** or **skip unnecessary computations**.

## **4. Examples in Chemical Engineering Context (15 minutes)**

### **Example 1: Stop iteration when convergence is reached**

### **Python**

tolerance = 1e-6

max\_iter = 100

x = 0.5

for i in range(max\_iter):

x\_new = x - (x\*\*2 - 2)/(2\*x) # Newton-Raphson

if abs(x\_new - x) < tolerance:

print("Converged at iteration", i+1)

break

x = x\_new

### **SCILAB**

tolerance = 1e-6;

max\_iter = 100;

x = 0.5;

for i = 1:max\_iter

x\_new = x - (x^2 - 2)/(2\*x); // Newton-Raphson

if abs(x\_new - x) < tolerance then

disp("Converged at iteration " + string(i));

break;

end

x = x\_new;

end

### **Example 2: Skip invalid inputs (negative values for sqrt)**

### **Python**

numbers = [4, -3, 9, -1, 16]

for num in numbers:

if num < 0:

continue

print("Square root:", num\*\*0.5)

### **SCILAB**

numbers = [4, -3, 9, -1, 16];

for i = 1:length(numbers)

if numbers(i) < 0 then

continue;

end

disp(sqrt(numbers(i)), "Square root:");

end

## **5. Nested Loops with Break & Continue**

* Break and continue only **affect innermost loop**.
* Example: stop scanning a matrix row if a condition is met

### **Python**

matrix = [[1,2,3],[4,5,6],[7,8,9]]

for row in matrix:

for val in row:

if val > 5:

break

print(val)

### **SCILAB**

matrix = [1,2,3;4,5,6;7,8,9];

for i = 1:3

for j = 1:3

if matrix(i,j) > 5 then

break;

end

disp(matrix(i,j))

end

end

## **6. Classroom Exercises / Examples**

1. Print **all numbers from 1–20**, but **skip multiples of 3** using continue.
2. Sum numbers from 1–50, but **stop the loop if the sum exceeds 200** using break.
3. Loop through a **list of temperatures**, ignore negative temperatures, and **stop when temperature exceeds 100°C**.

## **7. Summary / Key Points**

* **break:** exits a loop completely.
* **continue:** skips current iteration and continues.
* Useful for:
  + **Convergence checks**
  + **Input validation**
  + **Efficient computation**
* Works in both **for** and **while** loops.
* Important in **chemical engineering simulations** and **numerical calculations**.

**Lecture 13: Vapor Pressure Estimation Using Antoine Equation**

## **1. Introduction**

* **Vapor pressure:** The pressure exerted by a vapor in equilibrium with its liquid at a given temperature.
* **Importance in chemical engineering:**
  + Distillation and separation processes
  + Evaporation and boiling point estimation
  + Safety and storage of volatile liquids
* **Antoine equation:** Empirical relation to calculate vapor pressure at a given temperature.

## **2. Antoine Equation**

log10P=A−(B/(C+T))

Where:

* P = vapor pressure (mmHg)
* T = temperature (°C)
* A, B, C = substance-specific Antoine constants

**Notes:**

* Different temperature ranges require **different sets of constants**.
* Can be rearranged to compute **temperature for a given vapor pressure**.

## **3. Problem Statement**

**Objective:**

* Compute **vapor pressure of a liquid** at given temperatures using loops.
* Extend to **multiple temperatures** (e.g., tabulate vapor pressures).

**Example:**

* Substance: Water
* Antoine constants (for 1–100 °C):
  + A=8.07131, B=1730.63, C=233.426
* Compute vapor pressures at T = 20, 40, 60, 80, 100 °C

## **4. Pseudocode**

BEGIN

INPUT Antoine constants A, B, C

INPUT list of temperatures T[]

FOR each temperature t in T[]

P = 10^(A - B / (C + t))

OUTPUT P

ENDFOR

END

* **Optional:** Use **while loop** if user enters temperatures until a sentinel value.

### **Python**

# Antoine equation: log10(P) = A - B / (C + T)

import math

# Antoine constants for water

A = 8.07131

B = 1730.63

C = 233.426

temperatures = [20, 40, 60, 80, 100]

for T in temperatures:

P = 10\*\*(A - B / (C + T))

print(f"At {T} °C, Vapor Pressure = {P:.2f} mmHg")

**Key points:**

* Loop iterates over multiple temperatures.
* 10\*\*x → computes 10 raised to x (antilog base 10).
* :.2f → formats output to 2 decimal places.

### **SCILAB**

A = 8.07131;

B = 1730.63;

C = 233.426;

temperatures = [20, 40, 60, 80, 100];

for i = 1:length(temperatures)

T = temperatures(i);

P = 10^(A - B/(C + T));

printf("At %d °C, Vapor Pressure = %.2f mmHg\n", T, P);

end

**Key points:**

* SCILAB ^ operator used for exponentiation.
* printf formats the output for readability.

## **7. Advanced Application: User Input & While Loop**

* Let user **enter temperatures until they enter -1**:

**Python**

while True:

T = float(input("Enter temperature (°C, -1 to exit): "))

if T == -1:

break

P = 10\*\*(A - B / (C + T))

print(f"Vapor Pressure = {P:.2f} mmHg")

### **SCILAB**

while 1 do

T = input("Enter temperature (°C, -1 to exit): ");

if T == -1 then

break;

end

P = 10^(A - B/(C + T));

disp(P, "Vapor Pressure (mmHg):");

end

## **8. Classroom Exercises / Examples**

1. Compute vapor pressures of **ethanol** at 0, 20, 40, 60, 80 °C using Antoine constants:
   * A = 8.20417, B = 1642.89, C = 230.3
2. Extend program to **output temperature for a target vapor pressure** (rearrange Antoine equation).
3. Plot **vapor pressure vs temperature curve** in Python using matplotlib.

## **9. Summary / Key Points**

* **Antoine equation** allows calculation of **vapor pressure** for liquids.
* Loop structures (**for or while**) allow calculation at **multiple temperatures**.
* break and continue can control iterative input/termination.
* Useful for chemical engineers in:
  + Distillation
  + Evaporation
  + Process safety analysis

**Lecture 14: Arrays in SCILAB & Lists in Python**

## **1. Introduction**

* **Motivation:**
  + Often we need to store **multiple values** together (e.g., temperatures, pressures, concentrations).
  + **Arrays** (SCILAB) and **lists** (Python) allow **organized storage and easy access**.
* **Applications in chemical engineering:**
  + Storing temperature, pressure, or concentration data
  + Iterating over multiple simulation points
  + Plotting graphs (e.g., vapor pressure vs temperature)

## **2. Arrays in SCILAB**

* **Definition:** Ordered collection of elements of the **same type**.
* **1D array (vector):**

A = [1, 2, 3, 4, 5]; // Row vector

B = [1; 2; 3; 4; 5]; // Column vector

* **Access elements:**

disp(A(2)) // 2nd element

* **Modify elements:**

A(3) = 10; // 3rd element becomes 10

* **Length of array:**

n = length(A);

* **2D arrays (matrices):**

M = [1,2,3; 4,5,6; 7,8,9]; // 3x3 matrix

disp(M(2,3)) // element in 2nd row, 3rd column

## **3. Lists in Python**

* **Definition:** Ordered collection of **elements of any type**.
* **Create a list:**

temperatures = [20, 40, 60, 80, 100]

* **Access elements (indexing starts from 0):**

print(temperatures[2]) # 3rd element, 60

* **Modify elements:**

temperatures[2] = 65

* **Length of list:**

len(temperatures)

* **Lists can hold different types:**

data = [20, "Water", True]

## **4. Iterating Over Arrays / Lists**

### **SCILAB**

T = [20, 40, 60, 80, 100];

for i = 1:length(T)

disp(T(i), "Temperature:");

end

**Python**

T = [20, 40, 60, 80, 100]

for temp in T:

print("Temperature:", temp)

* **Use in calculations:** Apply formulas to each element.
* Example: Convert temperatures to Fahrenheit:
* for temp in T:
* F = (temp\*9/5) + 32
* print(F)

## **5. Multi-dimensional Arrays / Lists**

### **SCILAB**

P = [101, 150, 200; 110, 160, 210]; // 2x3 matrix

disp(P(2,3)) // 2nd row, 3rd column

### **Python Nested Lists**

P = [[101,150,200], [110,160,210]]

print(P[1][2]) # 2nd row, 3rd column (210)

* Nested loops can be used to **process each element**.

## **6. Classroom Examples / Applications**

1. **SCILAB:**
   * Store temperatures [20, 40, 60, 80, 100] and convert each to Fahrenheit using a loop.
2. **Python:**
   * Store concentrations [0.1, 0.2, 0.3, 0.4] and calculate reaction rate r = k\*C for k = 0.5.
3. **Multi-dimensional:**
   * Store vapor pressures of multiple substances at different temperatures in a matrix/list and display in tabular form.

## **7. Key Points / Summary**

* **SCILAB arrays:** Elements must be **same type**, indexing starts at 1.
* **Python lists:** Can store **different types**, indexing starts at 0.
* Arrays/lists are essential for **handling multiple data points** efficiently.
* Iteration with loops allows **calculation or transformation of all elements**.
* Foundation for **vectors, matrices, plotting, and simulations** in chemical engineering.

**Lecture 15: Matrices and vectorized operations (SCILAB emphasis)**

## **1. Introduction**

* **Motivation:**
  + Many chemical engineering problems involve **multiple variables** simultaneously (e.g., concentrations, temperatures, pressures).
  + **Matrices** allow storing and manipulating **2D data efficiently**.
  + **Vectorized operations** reduce the need for explicit loops, making calculations **faster and more readable**.
* **Applications:**
  + Reaction network simulations
  + Mass and energy balance calculations
  + Solving linear algebraic equations

## **2. Matrices in SCILAB**

* **Definition:** Rectangular arrangement of numbers in **rows and columns**.

### **Create a matrix**

A = [1, 2, 3; 4, 5, 6; 7, 8, 9] // 3x3 matrix

* **Access elements:**

disp(A(2,3)) // element in 2nd row, 3rd column → 6

* **Modify elements:**

A(1,1) = 10 // changes first element to 10

* **Size of matrix:**

[m,n] = size(A) // m = number of rows, n = number of columns

## **3. Matrix Operations**

* **Addition / Subtraction**

B = [9,8,7; 6,5,4; 3,2,1];

C = A + B

D = A - B

* **Scalar multiplication**

E = 2 \* A

* **Matrix multiplication**

F = A \* B

* **Element-wise multiplication / division**

G = A .\* B // element-wise multiplication

H = A ./ B // element-wise division

* **Transpose of a matrix**

AT = A' // transpose

* **Identity and zero matrices**

I = eye(3,3) // 3x3 identity matrix

Z = zeros(2,4) // 2x4 zero matrix

* **Accessing rows and columns**

row2 = A(2,:) // entire 2nd row

col3 = A(:,3) // entire 3rd column

## **4. Vectorized Operations**

* **Definition:** Perform operations on **entire arrays/matrices** without explicit loops.
* **Benefits:** Faster computation, concise code.

### **Examples**

* Multiply **all elements** of a vector by 2

v = [1,2,3,4];

v2 = 2 .\* v

* Compute **vapor pressures** for a temperature vector (from previous lecture)

T = [20,40,60,80,100];

A = 8.07131; B = 1730.63; C = 233.426;

P = 10 .^ (A - B ./ (C + T)) // vectorized operation

* Compute **concentration decay** for a batch reaction

C0 = 1; k = 0.1; t = [0:1:10];

C = C0 \* exp(-k .\* t) // concentration at each time

## **5. Practical Chemical Engineering Applications**

1. **Reaction network matrix:**
   * Rows → species, Columns → reactions
   * Vectorized calculation of reaction rates
2. **Heat transfer in multiple streams:**
   * Temperature matrix → apply .\* operations for energy balance
3. **Vapor pressure calculation at multiple temperatures for multiple substances:**
4. T = [20,40,60,80,100];
5. A = [8.07131, 8.20417]; B = [1730.63,1642.89]; C = [233.426,230.3];
6. P1 = 10.^(A(1) - B(1)./(C(1)+T));
7. P2 = 10.^(A(2) - B(2)./(C(2)+T));

## **6. Exercises / Classroom Examples**

1. Create a **3x3 concentration matrix** and multiply each element by 0.8 (decay factor).
2. Compute **vapor pressures** for water, ethanol, and methanol for temperatures 20–100°C in steps of 10°C.
3. Multiply two matrices (3x3) **element-wise** and using **matrix multiplication**; observe the difference.
4. Extract the **second row** and **third column** from a given matrix.

## **7. Key Points / Summary**

* **Matrices:** Efficient storage for 2D data; access with (row, column) indexing.
* **Vectorized operations:**
  + Use .\*, ./, .^ for element-wise operations.
  + Avoids explicit loops → faster and cleaner code.
* Essential in chemical engineering:
  + Reaction rate calculations
  + Heat and mass transfer simulations
  + Vapor pressure and concentration profiles

**Lecture 16:** **Introduction to Python NumPy Arrays**

## **1. Introduction**

* **Motivation:**
  + Standard Python lists are flexible but **slow for numerical computations**.
  + **NumPy arrays** provide:
    - Efficient storage of numerical data
    - Vectorized operations (element-wise calculations)
    - Tools for linear algebra, statistics, and more
* **Applications in chemical engineering:**
  + Reaction kinetics
  + Vapor pressure calculations
  + Heat and mass transfer simulations
  + Storing multi-dimensional experimental data

## **2. Installing and Importing NumPy**

# Install (if needed)

!pip install numpy

# Import NumPy

import numpy as np

* np is the standard alias for NumPy.

## **3. Creating NumPy Arrays**

### **1D arrays (vectors)**

import numpy as np

# From a Python list

arr1 = np.array([1,2,3,4,5])

print(arr1)

### **2D arrays (matrices)**

arr2 = np.array([[1,2,3], [4,5,6], [7,8,9]])

print(arr2)

### **Special arrays**

zeros = np.zeros((2,3)) # 2x3 matrix of zeros

ones = np.ones((3,2)) # 3x2 matrix of ones

identity = np.eye(3) # 3x3 identity matrix

## **4. Array Attributes**

* **Shape:** number of rows and columns

print(arr2.shape) # (3,3)

* **Size:** total number of elements

print(arr2.size) # 9

* **Data type of elements:**

print(arr2.dtype)

## **5. Indexing and Slicing**

* **1D array indexing:**

print(arr1[2]) # 3rd element (index starts at 0)

arr1[2] = 10 # modify element

* **2D array indexing:**

print(arr2[1,2]) # 2nd row, 3rd column

* **Slicing:**

print(arr1[1:4]) # elements from index 1 to 3

print(arr2[:,1]) # 2nd column

print(arr2[0,:]) # 1st row

## **6. Vectorized Operations**

* **Element-wise operations:**

a = np.array([1,2,3])

b = np.array([4,5,6])

print(a + b) # [5,7,9]

print(a \* b) # [4,10,18]

print(a / b) # [0.25,0.4,0.5]

print(a \*\* 2) # [1,4,9]

* **Scalar operations:**

print(a \* 2) # [2,4,6]

* **Comparison operators:**

print(a > 2) # [False, False, True]

## **7. Practical Chemical Engineering Examples**

1. **Vapor pressure calculation using Antoine equation (vectorized):**

import numpy as np

T = np.array([20,40,60,80,100])

A, B, C = 8.07131, 1730.63, 233.426

P = 10\*\*(A - B/(C + T))

print("Vapor pressures:", P)

1. **Reaction concentration decay:**

C0 = 1

k = 0.1

t = np.arange(0,11,1) # time from 0 to 10

C = C0 \* np.exp(-k\*t)

print("Concentrations:", C)

* Vectorized computation avoids explicit loops.

## **8. Array Functions**

* **Sum, mean, max, min:**

print(np.sum(C))

print(np.mean(C))

print(np.max(C))

print(np.min(C))

* **Reshape array:**

arr = np.arange(1,10).reshape((3,3))

print(arr)

* **Transpose:**

print(arr.T)

## **9. Classroom Exercises / Examples**

1. Create a **1D array of temperatures** from 0°C to 100°C in steps of 10°C.
2. Compute **vapor pressures** for water and ethanol using vectorized NumPy arrays.
3. Create a **3x3 matrix**, multiply all elements by 0.5 (reaction decay).
4. Compute **sum, mean, and max** of a concentration array.

## **10. Key Points / Summary**

* NumPy arrays are **efficient, fixed-type, multi-dimensional arrays**.
* **Vectorized operations** allow element-wise calculations without loops.
* Essential for **scientific computing** and **chemical engineering simulations**.
* NumPy functions: sum, mean, max, min, reshape, transpose provide **powerful data manipulation tools**.
* Prepares students for **matrix operations, plotting, and advanced calculations**.

**Lecture 17: Solving linear equations (material balance of mixing problem)**

## **1. Introduction**

* **Motivation:**
  + Many chemical engineering problems involve **linear relationships** between flows, concentrations, or heat/mass balances.
  + Examples:
    - Material balances in **mixing tanks**
    - Simultaneous reactions
    - Heat exchanger energy balances
* **Objective of lecture:**
  + Solve systems of linear equations using **Python (NumPy)** and **SCILAB**.
  + Apply to **material balance problem of mixing streams**.

## **2. Linear Equations Refresher**

* **General form:**

a1x+b1y+c1z=d1 ….(1) a2x+b2y+c2z=d2 ….(2) a3x+b3y+c3z=d3 ….(3)

* Can be written as **matrix form:**

A⋅x=b

Where:

A=coefficient matrix, x=[x,y,z]T, b=constants

## **3. Material Balance of Mixing Problem**

**Example:**

* Two streams of water with solute **A** mix in a tank:
  + Stream 1: 10 L/min, 5 g/L
  + Stream 2: 15 L/min, 8 g/L
* Find **concentration in the mixed stream**.

**Formulation:**

* Total flow: F=F1+F2=10+15=25 L/min
* Mass balance for solute A:

F1C1+F2C2=F⋅Cmix

10∗5+15∗8=25∗Cmix

Cmix=(50+120)/25=6.8 g/L

* **More complex case:** multiple tanks or unknown flows → leads to **linear equations**.

## **4. Python Implementation Using NumPy**

**Multiple mixing problem example:**

* **Problem:** Three streams mix to give total flow 50 L/min and total solute 200 g/min. Find individual flows if two flows are unknown.
* **Equations:**

F1+F2+F3=50

C1F1+C2F2+C3F3=200

* Represent in matrix form:

import numpy as np

# Coefficient matrix A

A = np.array([[1, 1, 1],

[5, 8, 10],

[1, 0, -1]]) # example extra equation for unique solution

# Constants vector b

b = np.array([50, 200, 5])

# Solve linear equations

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

print("Flow rates:", F)

**Explanation:**

* np.linalg.solve(A,b) solves **Ax = b** efficiently.
* Works for **any system of linear equations with unique solution**.

### **SCILAB**

// Coefficient matrix

A = [1,1,1; 5,8,10; 1,0,-1];

// Constants vector

b = [50; 200; 5];

// Solve linear equations

F = A\b; // or use F = inv(A)\*b

disp(F, "Flow rates:");

* A\b is **recommended method** instead of inv(A)\*b (numerically stable).

## **6. Classroom Exercises / Examples**

1. **Simple mixing:** Two streams mix; find **unknown flow** using linear equations.
2. **Three streams mixing:** Solve using **Python and SCILAB**.
3. **Multiple solutes:** Solve **simultaneous equations** for flows in a mixing tank with **two solutes**.
4. Check solutions by **substituting back into original equations**.

## **7. Summary / Key Points**

* **Material balance** problems often lead to **linear systems**.
* Matrix form: A⋅x=b
* Python: np.linalg.solve(A,b)
* SCILAB: A\b (preferred) or inv(A)\*b
* Efficient and scalable for **multiple streams, solutes, or tanks**.
* Foundation for **linear algebra applications** in chemical engineering.

**Lecture 18:** **Matrix Operations – Transpose, Determinant, Inverse (SCILAB & Python)**

## **1. Introduction**

* **Motivation:**
  + Matrices are central to **linear algebra**, used in:
    - Material and energy balances
    - Reaction network modeling
    - Process simulations
  + Important **matrix operations:**
    - Transpose → rearranging rows and columns
    - Determinant → check for singularity/solvability
    - Inverse → solve linear systems explicitly

## **2. Matrix Transpose**

* **Definition:** Swap **rows and columns** of a matrix.

### **SCILAB**

A = [1,2,3;4,5,6;7,8,9];

AT = A' // transpose

disp(AT)

**Python**

import numpy as np

A = np.array([[1,2,3],[4,5,6],[7,8,9]])

AT = A.T

print(AT)

* **Applications:**
  + Rearranging matrices for multiplication
  + Switching between row/column vectors

## **3. Determinant of a Matrix (10 minutes)**

* **Definition:** A scalar value that indicates whether a **square matrix is invertible**.
* **Determinant ≠ 0:** Matrix is invertible
* **Determinant = 0:** Matrix is singular → no unique solution

### **SCILAB**

A = [2,1;5,3];

detA = det(A)

disp(detA)

**Python**

import numpy as np

A = np.array([[2,1],[5,3]])

detA = np.linalg.det(A)

print(detA)

* **Chemical engineering applications:**
  + Check if **mixing problem** has a unique solution
  + Linear independence of **reaction stoichiometry matrix**

## **4. Matrix Inverse**

* **Definition:** For square matrix A, **A⁻¹** satisfies:

A⋅A−1=I

* Only exists if **determinant ≠ 0**.

### **SCILAB**

A = [2,1;5,3];

invA = inv(A)

disp(invA)

* **Solve linear equations using inverse:**

b = [5;15];

x = inv(A)\*b

disp(x)

**Python**

import numpy as np

A = np.array([[2,1],[5,3]])

b = np.array([5,15])

invA = np.linalg.inv(A)

x = np.dot(invA, b)

print(x)

* **Note:** For solving linear equations, prefer np.linalg.solve(A,b) in Python or A\b in SCILAB (more numerically stable).

## **5. Summary of Matrix Operations**

| **Operation** | **SCILAB Syntax** | **Python Syntax** | **Use in Chemical Engg** |
| --- | --- | --- | --- |
| Transpose | A' | A.T | Switching row/col vectors |
| Determinant | det(A) | np.linalg.det(A) | Check singularity |
| Inverse | inv(A) | np.linalg.inv(A) | Solve linear systems |

## **Practical Examples / Applications**

**Python**

1. **#Mixing problem (linear system)**
2. A = np.array([[1,1],[5,8]])
3. b = np.array([15, 110])
4. x = np.linalg.inv(A).dot(b)
5. print("Flow rates:", x)
6. #Check Solvability
7. detA = np.linalg.det(A)
8. if detA == 0:
9. print("No unique solution")

**Transpose example**:

* + Convert **row vector of flows** into column vector for matrix multiplication

**SCILAB Example:**

A = [1,1;5,8];

b = [15;110];

disp("Determinant: "+string(det(A)))

disp("Inverse:")

disp(inv(A))

x = inv(A)\*b

disp("Flow rates:")

disp(x)

## **7. Classroom Exercises**

1. Find the **transpose, determinant, and inverse** of a 3×3 matrix.
2. Solve a **2-stream mixing problem** using inverse in Python and SCILAB.
3. Verify: A \* inv(A) = I for a 2×2 or 3×3 matrix.
4. Check determinant of singular matrix (e.g., rows proportional) and discuss significance.

## **8. Key Points / Summary**

* **Transpose:** swaps rows and columns; often used in multiplication or vector alignment.
* **Determinant:** scalar measure; checks invertibility.
* **Inverse:** allows explicit solution of linear systems; exists only if determinant ≠ 0.
* Use **matrix operations** for solving **material balance, reaction networks, and process simulations** efficiently.
* For solving equations, prefer **A\b in SCILAB** or **np.linalg.solve(A,b) in Python** over using inverse.

**Lecture 19:** **Case Study – Solving 3-Equation System for Chemical Process Mass Balance**

## **1. Introduction**

* **Motivation:**
  + Material balances are fundamental in **chemical process design**.
  + Complex processes often require solving **simultaneous linear equations**.
  + This lecture demonstrates **practical application** of matrices, determinants, and linear algebra to a **3-equation system**.
* **Objective:**
  + Solve a 3×3 linear system representing **mass balance in a mixing and separation process**.
  + Implement solutions in **Python and SCILAB**.

## **2. Problem Statement**

**Example Scenario:**

* Three liquid streams mix in a tank, producing a single outlet stream.
* Stream flow rates: F1,F2,F3 (unknown).
* Total flow = 100 L/min.
* Mass balances for solute A, B, and C:

F1+F2+F3=100

2F1+3F2+F3=180 (solute A)

F1+4F2+5F3=220 (solute B)

* **Goal:** Find F1,F2,F3.

## **3. Formulating Matrix Form**

* Coefficient matrix AA, variable vector xx, and constant vector bb:

A=, x= b=

* Matrix equation: A⋅x=b

## **4. Python Implementation**

import numpy as np

# Coefficient matrix

A = np.array([[1, 1, 1],

[2, 3, 1],

[1, 4, 5]])

# Constant vector

b = np.array([100, 180, 220])

# Solve the system

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

print("Flow rates (F1, F2, F3):", F)

**Explanation:**

* np.linalg.solve(A,b) automatically computes solution for a **unique solution** system.
* Output gives **flow rates of all streams**.

## **5. SCILAB Implementation**

// Coefficient matrix

A = [1,1,1; 2,3,1; 1,4,5];

// Constants vector

b = [100;180;220];

// Solve system

F = A\b; // preferred over inv(A)\*b

disp(F, "Flow rates (F1, F2, F3):");

**Explanation:**

* A\b efficiently solves linear system.
* Avoids numerical instability of inv(A)\*b.

## **6. Verification of Solution**

* Substitute solution back into original equations:

np.dot(A, F) # Should equal b

* Ensures **mass balance is satisfied**

## **7. Discussion / Applications**

* **Why 3-equation system?**
  + Many chemical processes involve multiple components, streams, and constraints.
* **Extension to more complex systems:**
  + 4–5 components
  + Multi-stage separation
  + Coupled reaction and separation balances
* **Key learning:**
  + Linear algebra allows **systematic and scalable approach** to process design problems.

## **8. Classroom Exercises**

1. Solve a **three-component mixing problem** with different solute concentrations.
2. Include **four streams** and solve using Python and SCILAB.
3. Compute determinant of coefficient matrix to **check if solution exists** before solving.
4. Extend the case study to compute **average solute concentration** in the outlet stream.

## **9. Key Points / Summary**

* Mass balances in chemical processes often form **linear systems**.
* Representing them in **matrix form** allows efficient computation:
  + Python: np.linalg.solve(A,b)
  + SCILAB: A\b
* Verification by substitution ensures **accuracy of solution**.
* Foundation for **multi-component, multi-stream, and multi-stage chemical process calculations**.

**Lecture 20: Writing Functions in SCILAB**

## **1. Introduction**

* **Motivation:**
  + Functions allow **reusable, modular, and organized code**.
  + Encapsulate **repeated calculations** (e.g., material balances, concentration conversions).
  + Makes code **cleaner and easier to debug**.
* **Applications in chemical engineering:**
  + Mass and energy balance calculations
  + Reaction rate computations
  + Thermodynamic property calculations (vapor pressure, enthalpy)

## **2. Function Basics in SCILAB**

* **Syntax:**

function output = function\_name(input1, input2, ...)

// Function body

output = ... // calculation

endfunction

* **Example:** Simple function to **square a number**

function y = square\_number(x)

y = x^2

endfunction

disp(square\_number(5)) // Output: 25

* **Key Points:**
  + function and endfunction define the function.
  + Inputs are passed as arguments.
  + Output is assigned to the variable after = in function definition.

## **3. Function with Multiple Inputs and Outputs**

* **Syntax for multiple outputs:**

function [out1, out2] = multi\_output\_function(a, b)

out1 = a + b

out2 = a \* b

endfunction

[x\_sum, x\_product] = multi\_output\_function(5, 3)

disp(x\_sum) // 8

disp(x\_product) // 15

* **Applications:**
  + Calculate **total flow and average concentration** in mixing tanks

## **4. Example: Material Balance Function**

* **Problem:** Compute outlet flow and solute concentration for two-stream mixing

function [F\_total, C\_out] = mix\_streams(F1, C1, F2, C2)

F\_total = F1 + F2

C\_out = (F1\*C1 + F2\*C2) / F\_total

endfunction

[F\_total, C\_out] = mix\_streams(10, 5, 15, 8)

disp(F\_total, "Total Flow (L/min):")

disp(C\_out, "Outlet Concentration (g/L):")

* **Explanation:**
  + Encapsulates **material balance formula** in a reusable function.
  + Can now use for **different streams** without rewriting the code.

## **5. Functions Calling Other Functions**

* Functions can **call other functions** for modularity.

function y = square(x)

y = x^2

endfunction

function z = sum\_of\_squares(a, b)

z = square(a) + square(b)

endfunction

disp(sum\_of\_squares(3,4)) // Output: 25

* Useful for **hierarchical calculations** in chemical process modeling.

## **6. Anonymous Functions**

* **Short inline functions** for simple operations

f = @(x) x^2 + 2\*x + 1

disp(f(3)) // Output: 16

* Quick for **simple mathematical expressions** without full function definition.

## **7. Classroom Exercises**

1. Write a function to **calculate vapor pressure** using Antoine equation.
2. Write a function to **compute outlet flow and concentration** for 3 streams mixing.
3. Write a function to **compute factorial** of a number.
4. Write a function that **returns both sum and product** of a vector of numbers.

**Lecture 21: Writing Functions in Python (Arguments, Return Values)**

## **1. Introduction**

* **Motivation:**
  + Functions allow **modular, reusable, and organized code**.
  + Python functions are essential for **encapsulating calculations** such as material balances, reaction rates, and property evaluations.
* **Applications in chemical engineering:**
  + Mass and energy balances
  + Reaction kinetics computations
  + Thermodynamic property calculations (vapor pressure, enthalpy, etc.)

## **2. Function Basics in Python**

* **Syntax:**

def function\_name(parameters):

"""

Optional: docstring describing function

"""

# Function body

return output

* **Example:** Simple function to **square a number**

def square\_number(x):

return x\*\*2

print(square\_number(5)) # Output: 25

* **Key Points:**
  + def keyword defines a function.
  + return passes result back to caller.
  + Python functions can accept **any data type** as arguments.

## **3. Function with Multiple Arguments**

* **Example:** Compute **sum and product of two numbers**

def sum\_and\_product(a, b):

return a + b, a \* b

s, p = sum\_and\_product(5, 3)

print("Sum:", s) # 8

print("Product:", p) # 15

* **Key Points:**
  + Functions can return **multiple values**.
  + Useful for **simultaneous calculations** like total flow and average concentration.

## **4. Keyword and Default Arguments**

* **Keyword arguments:** specify values explicitly

def power(base, exponent):

return base\*\*exponent

print(power(base=2, exponent=3)) # Output: 8

* **Default arguments:** used if not provided by caller

def power(base, exponent=2):

return base\*\*exponent

print(power(4)) # 16, exponent defaults to 2

print(power(2,3)) # 8

## **5. Example: Material Balance Function**

* **Problem:** Compute **outlet flow and concentration** for two streams mixing

def mix\_streams(F1, C1, F2, C2):

F\_total = F1 + F2

C\_out = (F1\*C1 + F2\*C2)/F\_total

return F\_total, C\_out

F\_total, C\_out = mix\_streams(10, 5, 15, 8)

print("Total Flow (L/min):", F\_total)

print("Outlet Concentration (g/L):", C\_out)

* **Explanation:**
  + Encapsulates **material balance formula** in a reusable function.
  + Can now apply for **different stream data** without rewriting calculations.

## **6. Functions Calling Other Functions**

def square(x):

return x\*\*2

def sum\_of\_squares(a, b):

return square(a) + square(b)

print(sum\_of\_squares(3,4)) # Output: 25

* Allows **hierarchical calculations** in complex chemical engineering problems.

## **7. Lambda (Anonymous) Functions**

* **Short inline functions** for simple operations

f = lambda x: x\*\*2 + 2\*x + 1

print(f(3)) # Output: 16

* Useful for **quick calculations** without full function definitions.

## **8. Classroom Exercises**

1. Write a function to **calculate vapor pressure** using Antoine equation.
2. Write a function to **compute outlet flow and concentration** for 3 streams mixing.
3. Write a function that **returns sum and product** of a list of numbers.
4. Use **lambda function** to calculate squares of numbers in a vector.

## **9. Key Points / Summary**

* Functions in Python are defined with def and return values using return.
* Functions can have **multiple arguments, default values, and return multiple outputs**.
* Functions can **call other functions**, enabling modular and hierarchical code.
* **Lambda functions** provide concise syntax for simple calculations.
* Encapsulating **material balances, property calculations, and iterative computations** in functions improves **readability and reusability**.

**Lecture 22: Chemical Engineering Example – Enthalpy Calculation Using Functions in Python**

## **1. Introduction**

* **Motivation:**
  + Chemical engineering calculations often involve repetitive **thermodynamic property evaluations**.
  + Enthalpy, a key property, depends on **temperature, specific heat, and phase**.
  + Using **functions**, these calculations become **modular and reusable**.
* **Objective:**
  + Learn to **write Python functions** for enthalpy calculations.
  + Apply to **different substances and temperature ranges** efficiently.

## **2. Enthalpy Calculation Overview**

* **Definition:** Change in enthalpy for a substance can be calculated as:

ΔH=

* **Simplified case:** Assume **constant specific heat Cp**

ΔH=Cp⋅(T−Tref)

* **Python implementation** allows handling **arrays of temperatures** for multiple conditions.

## **3. Basic Function for Enthalpy**

def enthalpy(T, T\_ref, Cp):

"""

Calculate enthalpy change for a substance with constant Cp

T: temperature [°C or K]

T\_ref: reference temperature [°C or K]

Cp: specific heat [J/(g\*K) or kJ/(kg\*K)]

"""

return Cp \* (T - T\_ref)

# Example usage

delta\_H = enthalpy(T=150, T\_ref=25, Cp=4.18) # Water, J/g

print("Enthalpy change (J/g):", delta\_H)

* **Explanation:**
  + Encapsulates formula in a reusable **function**.
  + Allows **different substances, temperatures, and Cp values**.

## **4. Handling Arrays of Temperatures (Vectorization with NumPy)**

import numpy as np

def enthalpy\_vector(T\_array, T\_ref, Cp):

return Cp \* (T\_array - T\_ref)

T = np.array([25, 50, 75, 100, 125])

delta\_H\_array = enthalpy\_vector(T, T\_ref=25, Cp=4.18)

print("Enthalpy changes (J/g):", delta\_H\_array)

* **Benefit:** Compute **multiple temperature points** efficiently without loops.
* Can be integrated into **process simulation calculations**.

## **5. Function with Phase Check (Optional Advanced)**

* **Idea:** Use different Cp values for **liquid and vapor phase**

def enthalpy\_phase(T, T\_ref, Cp\_liq, Cp\_vap, T\_boiling):

if T <= T\_boiling:

return Cp\_liq \* (T - T\_ref)

else:

# Heat to boiling point + vaporization + heat above boiling

H1 = Cp\_liq \* (T\_boiling - T\_ref)

Hvap = 40.7 # Example: latent heat kJ/mol

H2 = Cp\_vap \* (T - T\_boiling)

return H1 + Hvap + H2

# Example

delta\_H = enthalpy\_phase(T=120, T\_ref=25, Cp\_liq=4.18, Cp\_vap=2.0, T\_boiling=100)

print("Enthalpy (kJ/mol or J/g):", delta\_H)

* **Applications:**
  + Water or steam
  + Chemical process streams with **phase change**

## **6. Classroom Exercises**

1. Write a function to calculate **enthalpy change for ethanol** from 25°C to 78°C (boiling point) assuming constant Cp.
2. Extend the function to handle **phase change at boiling point**.
3. Use a **NumPy array of temperatures** to calculate enthalpy changes for **a range of temperatures from 25°C to 150°C**.
4. Compare **enthalpy changes for two substances** using a single function.

## **7. Key Points / Summary**

* Functions allow **modular, reusable thermodynamic calculations**.
* Python functions can handle **single values and arrays** (vectorization).
* Phase-aware functions enable **accurate modeling of heating/cooling with phase changes**.
* Prepares students for **process simulation and energy balance calculations**.

**Lecture 23: Functions for Raoult’s Law – Bubble Point and Dew Point Calculations**

## **1. Introduction**

* **Motivation:**
  + Vapor-liquid equilibrium (VLE) is crucial in chemical engineering for **distillation, separation, and process design**.
  + **Raoult’s Law** provides a simple model for **ideal mixtures**.
  + Writing **functions in Python** allows **reusable, modular VLE calculations**.
* **Objective:**
  + Learn to create Python functions for **bubble point** and **dew point** calculations.
  + Apply for **multi-component mixtures**.

## **2. Raoult’s Law Basics**

* **Raoult’s Law:**

Pi=xi⋅Pisat(T)

Where:

* Pi = partial pressure of component ii
* xi = mole fraction in liquid
* Pisat(T) = saturation pressure at temperature T
* **Total pressure:**

P=∑iPi=∑ixi⋅Pisat(T)

* **Bubble point:** Temperature at which **first bubble of vapor forms at given pressure**.
* **Dew point:** Temperature at which **first drop of liquid forms from vapor at given pressure**.

## **3. Python Function: Bubble Point**

* **Assumptions:**
  + Ideal solution
  + Known **liquid mole fractions** and **saturation pressures**

def bubble\_point\_pressure(x, Psat):

"""

Calculate total pressure using Raoult's Law

x: liquid mole fractions (list or array)

Psat: saturation pressures at given temperature (same length as x)

"""

P\_total = sum(xi\*Pi for xi, Pi in zip(x, Psat))

return P\_total

# Example

x = [0.5, 0.5] # mole fractions

Psat = [200, 100] # kPa

P\_total = bubble\_point\_pressure(x, Psat)

print("Bubble Point Pressure (kPa):", P\_total)

* **Explanation:**
  + Loops through each component, calculates partial pressures, and sums them.
  + Output is **total pressure at bubble point**.

## **4. Python Function: Dew Point**

* **Dew point calculation:** **Use** **vapor mole fractions**

1P=∑iyiPisat(T)

def dew\_point\_pressure(y, Psat):

"""

Calculate total pressure at dew point using Raoult's Law

y: vapor mole fractions

Psat: saturation pressures at given temperature

"""

P\_total = 1 / sum(yi/Pi for yi, Pi in zip(y, Psat))

return P\_total

# Example

y = [0.4, 0.6]

Psat = [200, 100]

P\_dew = dew\_point\_pressure(y, Psat)

print("Dew Point Pressure (kPa):", P\_dew)

* **Explanation:**
  + Inverts the sum of **y\_i / P\_i^sat** to get total pressure.
  + Applicable for **vapor composition known**.

## **5. Bubble Point Temperature Iteration**

* Often, **Psat depends on T** via Antoine equation:

log10Pisat=A−BC+T\log\_{10} P\_i^{sat} = A - \frac{B}{C+T}

* Can write a **function to compute bubble point temperature iteratively**:

import numpy as np

def Psat\_Antoine(T, A, B, C):

return 10\*\*(A - B/(C + T))

def bubble\_point\_temperature(x, Antoine\_params, P\_total, T\_guess=100):

T = T\_guess

for \_ in range(100):

Psat = [Psat\_Antoine(T, \*params) for params in Antoine\_params]

P\_calc = sum(xi\*Pi for xi, Pi in zip(x, Psat))

T += (P\_total - P\_calc)\*0.1 # simple iteration

return T

* **Explanation:**
  + Iterative method adjusts **T** until **calculated P matches total pressure**.

## **6. Classroom Exercises**

1. Write a function to calculate **bubble point pressure** for a 3-component mixture.
2. Write a function to calculate **dew point pressure** for the same mixture.
3. Extend the bubble point function to **include Antoine equation** for temperature-dependent Psat.
4. Compare results for **binary mixtures** at different compositions.

## **7. Key Points / Summary**

* **Raoult’s Law** provides a simple method for **VLE calculations**.
* Functions allow **modular, reusable, and clean code** for bubble and dew point calculations.
* Iterative methods can handle **temperature-dependent saturation pressures**.
* Python functions can handle **single values or arrays** for process simulations.
* Prepares students for **more complex distillation and separation calculations**.

**Lecture 24: I/O in Python – Reading/Writing Text & CSV Files**

## **1. Introduction**

* **Motivation:**
  + Chemical engineers frequently work with **experimental data, process logs, and simulation outputs** stored in text or CSV files.
  + Python provides **simple and flexible ways** to read and write these files for **data analysis and automation**.
* **Objective:**
  + Learn how to **read and write text and CSV files** in Python.
  + Apply to **storing or retrieving chemical process data**.

## **2. Reading and Writing Text Files**

* **Opening a file:**

# Open a file for writing

f = open("output.txt", "w")

f.write("Hello, Python File I/O\n")

f.write("Data for chemical process\n")

f.close()

* **Reading a text file:**

f = open("output.txt", "r")

content = f.read() # read entire file

print(content)

f.close()

* **Important Modes:**
  + 'r' → read
  + 'w' → write (overwrite)
  + 'a' → append
  + 'r+' → read/write
* **With statement (recommended):** automatically closes file

with open("output.txt", "r") as f:

content = f.read()

print(content)

## **3. Reading and Writing Line by Line**

* **Writing multiple lines:**

lines = ["Stream 1: 10 L/min\n", "Stream 2: 15 L/min\n"]

with open("streams.txt", "w") as f:

f.writelines(lines)

* **Reading line by line:**

with open("streams.txt", "r") as f:

for line in f:

print(line.strip()) # remove newline character

## **4. CSV Files – Introduction**

* **CSV (Comma Separated Values):** Common format for **tabular process data**.
* Python has a built-in **csv module** and **pandas library** for easier handling.

## **5. Using csv Module**

* **Writing CSV:**

import csv

data = [["Stream","Flow(L/min)","Concentration(g/L)"],

["S1", 10, 5],

["S2", 15, 8]]

with open("streams.csv", "w", newline="") as file:

writer = csv.writer(file)

writer.writerows(data)

* **Reading CSV:**

with open("streams.csv", "r") as file:

reader = csv.reader(file)

for row in reader:

print(row)

## **6. Using pandas for CSV**

* **Reading CSV:**

import pandas as pd

df = pd.read\_csv("streams.csv")

print(df)

* **Writing CSV:**

df.to\_csv("streams\_out.csv", index=False)

* **Benefits of pandas:**
  + Easy manipulation of **tabular data**
  + Can **filter, sort, calculate, and plot** data efficiently

## **7. Practical Examples**

1. **Store experimental data:** flow rates and concentrations of multiple streams.
2. **Read a CSV of VLE data** (temperature vs pressure) and calculate bubble point using function from previous lecture.
3. **Append new simulation results** to an existing CSV file.
4. **Compare process streams** using pandas DataFrame operations.

## **8. Classroom Exercises**

1. Write a text file containing **stream flows and concentrations** for 3 streams.
2. Read the file and **compute total flow**.
3. Write a CSV file for **enthalpy vs temperature** data (from previous function).
4. Read CSV using pandas and **calculate average enthalpy**.

## **9. Key Points / Summary**

* File I/O allows **storing, retrieving, and analyzing process data** efficiently.
* **Text files:** simple, flexible, good for logs and small data.
* **CSV files:** structured format, widely used for **tabular chemical engineering data**.
* **Python modules:** csv for basic operations, pandas for advanced data handling.
* Integrating **functions with File I/O** allows automated calculations for **process simulations and experimental data analysis**.

**Lecture 25: File Handling in SCILAB – Data Import & Export**

## **1. Introduction**

* **Motivation:**
  + Chemical engineers often work with **process data** in files: text, CSV, or spreadsheets.
  + SCILAB provides commands to **import and export data** for analysis, plotting, or simulations.
* **Objective:**
  + Learn to **read data from files** and **write data to files** in SCILAB.
  + Apply to **process streams, experimental measurements, or simulation outputs**.

## **2. Reading Text Files**

* **Using read or mgetl for text files:**

// Reading a simple text file

f = mopen('data.txt', 'r'); // open file for reading

line = mgetl(f); // read one line

while line <> -1

disp(line)

line = mgetl(f)

end

mclose(f)

* **Explanation:**
  + mopen(filename, mode) opens file
  + mgetl(f) reads **line by line**
  + mclose(f) closes the file

## **3. Reading Numeric Data**

* **Using read command:**

// File 'streams.txt' contains numeric data: flow rates and concentrations

data = read('streams.txt');

disp(data)

* **Using csvRead for CSV files:**

data = csvRead('streams.csv', ';'); // specify separator if needed

disp(data)

* **Applications:**
  + Reading **experimental measurements**
  + Importing **VLE data** or **thermodynamic tables**

## **4. Writing Data to Text Files**

* **Using mopen and mfprintf**:

F = [10; 15; 20]; // flow rates

C = [5; 8; 12]; // concentrations

f = mopen('output.txt','w');

for i = 1:size(F,1)

mfprintf(f, 'Stream %d: Flow = %f L/min, Conc = %f g/L\n', i, F(i), C(i))

end

mclose(f)

* **Explanation:**
  + mfprintf similar to printf in C
  + Allows **formatted output** for logs or reports

## **5. Writing CSV Files**

* **Using csvWrite:**

data = [F, C]; // combine flow and concentration

csvWrite(data, 'streams\_out.csv', ';') // ';' as separator

* **Applications:**
  + Export **simulation results** for Excel or Python
  + Share **process data with colleagues**

## **6. Reading & Writing Matrices**

* **Save a matrix to a file:**

M = [1 2 3; 4 5 6; 7 8 9];

save('matrix\_data.dat', 'M')

* **Load a matrix from a file:**

load('matrix\_data.dat', '-mat') // loads variable M

disp(M)

* **Explanation:**
  + Useful for **storing simulation matrices**
  + Preserves **SCILAB variable types**

## **7. Practical Examples**

1. **Import stream flow and concentration data** from CSV, compute **total flow**.
2. **Export computed enthalpy values** for multiple temperatures to a text file.
3. **Read experimental VLE data** and use in **bubble point calculations**.
4. **Store simulation matrix results** for future analysis.

## **8. Classroom Exercises**

1. Create a CSV file containing **flow rates of 3 streams**, read it in SCILAB, and compute **average flow**.
2. Write a **text report** of stream data including **flow, concentration, and total flow**.
3. Save a **matrix of temperature and enthalpy** to a .dat file and reload it.
4. Read a CSV of **bubble point temperatures** and **plot using SCILAB plotting commands**.

## **9. Key Points / Summary**

* SCILAB provides multiple ways to **read and write data**: read, csvRead, mopen/mfprintf, csvWrite, save/load.
* File handling is crucial for **experiment data processing, simulation results, and reporting**.
* CSV files are convenient for **tabular data exchange** with Excel or Python.
* Encapsulating file operations with **functions** makes code **modular and reusable**.

**Lecture 26: Plotting in SCILAB – 2D Plots & Least Squares Regression Method**

## **1. Introduction**

* **Motivation:**
  + Visualization is critical in chemical engineering to **analyze experimental data, process trends, and model predictions**.
  + **2D plots** help compare variables (e.g., temperature vs. enthalpy, flow vs. concentration).
  + **Least Squares Regression (LSR)** is a common method to **fit a line to data points**, estimating **linear trends**.
* **Objective:**
  + Learn to create **2D plots in SCILAB**.
  + Apply **least squares regression** to find best-fit lines for experimental data.

## **2. Basic 2D Plot in SCILAB**

* **Syntax:**

x = [1 2 3 4 5];

y = [2 4 5 4 6];

plot(x, y) // basic 2D plot

xlabel("x-axis") // label x-axis

ylabel("y-axis") // label y-axis

title("2D Plot Example")

* **Enhancements:**
  + grid() → show grid
  + plot(x, y, 'r\*') → red stars for points
  + plot(x, y, 'b-') → blue line connecting points

## **3. Adding Multiple Curves**

y2 = [1 3 4 3 5];

plot(x, y, 'r\*') // first curve

plot(x, y2, 'b-') // second curve

legend(["Data1", "Data2"])

* Useful for **comparing experimental vs theoretical data**

## **4. Least Squares Regression**

* **Purpose:** Fit a straight line y=m∗x+c through data points
* **Formulas:**

m=n∑xy−∑x∑yn∑x2−(∑x)2,c=∑y−m∑xn

* **SCILAB Implementation:**

x = [1 2 3 4 5];

y = [2 4 5 4 6];

n = length(x)

m = (n\*sum(x.\*y) - sum(x)\*sum(y)) / (n\*sum(x.^2) - sum(x)^2)

c = (sum(y) - m\*sum(x)) / n

y\_fit = m\*x + c

plot(x, y, 'r\*') // data points

plot(x, y\_fit, 'b-') // fitted line

xlabel("x")

ylabel("y")

title("Least Squares Regression")

legend(["Data", "Fit"])

* **Explanation:**
  + .\* → element-wise multiplication
  + .^2 → element-wise square
  + Combines **experimental points** and **regression line**

## **5. Using** reglin **Function**

* SCILAB provides **built-in linear regression**:

coeff = reglin(x', y') // returns [intercept; slope]

y\_fit = coeff(1) + coeff(2)\*x

plot(x, y, 'r\*')

plot(x, y\_fit, 'b-')

* **Benefit:** simplifies regression for **quick analysis**

## **6. Practical Example – Chemical Process Data**

* **Scenario:** Flow rate vs. outlet concentration of a mixing stream

Flow = [10 15 20 25 30];

Conc = [5.1 7.9 10.2 12.1 14.3];

// Least squares regression

n = length(Flow)

m = (n\*sum(Flow.\*Conc) - sum(Flow)\*sum(Conc)) / (n\*sum(Flow.^2) - sum(Flow)^2)

c = (sum(Conc) - m\*sum(Flow))/n

Conc\_fit = m\*Flow + c

plot(Flow, Conc, 'r\*')

plot(Flow, Conc\_fit, 'b-')

xlabel("Flow rate (L/min)")

ylabel("Concentration (g/L)")

title("Flow vs Concentration")

legend(["Experimental", "Least Squares Fit"])

grid()

* **Interpretation:**
  + Red stars → experimental data
  + Blue line → trend line for process prediction

## **7. Classroom Exercises**

1. Plot **temperature vs enthalpy data** and fit a linear regression line.
2. Fit **bubble point vs mole fraction data** using LSR.
3. Compare **experimental and simulated VLE data** in one plot.
4. Use **reglin function** for multiple datasets and compare slopes.

## **8. Key Points / Summary**

* SCILAB provides **flexible 2D plotting** for chemical engineering data.
* Least Squares Regression is useful to **fit a linear trend** to experimental points.
* Element-wise operations (.\*, .^) are critical for **manual LSR calculations**.
* reglin provides a **quick, built-in regression solution**.
* Visualization aids in **process understanding, design, and comparison with models**.

**Lecture 27: Plotting in Python – Matplotlib Basics & Least Squares Regression**

## **1. Introduction**

* **Motivation:**
  + Visualization is critical for **analyzing chemical engineering data**, such as **flow rates, concentrations, temperatures, and VLE data**.
  + **Matplotlib** is the primary library for 2D plotting in Python.
  + **Least Squares Regression (LSR)** is used to **fit a line to experimental or simulated data**.
* **Objective:**
  + Learn **basic plotting using Matplotlib**.
  + Apply **least squares regression** to **fit a line through data points**.

## **2. Matplotlib Basics**

* **Importing Matplotlib:**

import matplotlib.pyplot as plt

* **Simple 2D plot:**

x = [1, 2, 3, 4, 5]

y = [2, 4, 5, 4, 6]

plt.plot(x, y) # basic line plot

plt.xlabel("x-axis") # label x-axis

plt.ylabel("y-axis") # label y-axis

plt.title("2D Plot Example")

plt.show()

* **Scatter plot for points:**

plt.scatter(x, y, color='red', label='Data points')

plt.xlabel("x")

plt.ylabel("y")

plt.title("Scatter Plot Example")

plt.legend()

plt.grid(True)

plt.show()

## **3. Multiple Curves in One Plot**

y2 = [1, 3, 4, 3, 5]

plt.plot(x, y, 'r\*', label='Data1') # red stars

plt.plot(x, y2, 'b-', label='Data2') # blue line

plt.xlabel("x")

plt.ylabel("y")

plt.title("Multiple Curves")

plt.legend()

plt.grid(True)

plt.show()

* Useful for **comparing experimental vs theoretical data**.

## **4. Least Squares Regression – Manual Calculation**

* **Using NumPy for LSR:**

import numpy as np

x = np.array([1, 2, 3, 4, 5])

y = np.array([2, 4, 5, 4, 6])

n = len(x)

m = (n\*np.sum(x\*y) - np.sum(x)\*np.sum(y)) / (n\*np.sum(x\*\*2) - (np.sum(x))\*\*2)

c = (np.sum(y) - m\*np.sum(x)) / n

y\_fit = m\*x + c

plt.scatter(x, y, color='red', label='Data')

plt.plot(x, y\_fit, color='blue', label='Least Squares Fit')

plt.xlabel("x")

plt.ylabel("y")

plt.title("Least Squares Regression")

plt.legend()

plt.grid(True)

plt.show()

* **Explanation:**
  + x\*\*2 → element-wise square
  + x\*y → element-wise multiplication
  + Blue line represents **linear fit** for trend analysis

## **5. Using NumPy** polyfit **for Regression**

* **Simpler approach:**

coeff = np.polyfit(x, y, 1) # 1 = linear fit

m, c = coeff

y\_fit = m\*x + c

plt.scatter(x, y, color='red', label='Data')

plt.plot(x, y\_fit, color='blue', label='LS Fit')

plt.xlabel("x")

plt.ylabel("y")

plt.title("Linear Regression with polyfit")

plt.legend()

plt.grid(True)

plt.show()

* **Benefits:**
  + Fast and reliable
  + Can extend to **polynomial regression** by changing degree

## **6. Practical Example – Chemical Process Data**

* **Scenario:** Flow rate vs. concentration of mixing stream

Flow = np.array([10, 15, 20, 25, 30])

Conc = np.array([5.1, 7.9, 10.2, 12.1, 14.3])

# Linear regression using polyfit

m, c = np.polyfit(Flow, Conc, 1)

Conc\_fit = m\*Flow + c

plt.scatter(Flow, Conc, color='red', label='Experimental')

plt.plot(Flow, Conc\_fit, color='blue', label='LS Fit')

plt.xlabel("Flow rate (L/min)")

plt.ylabel("Concentration (g/L)")

plt.title("Flow vs Concentration – Least Squares Regression")

plt.legend()

plt.grid(True)

plt.show()

* **Interpretation:**
  + Scatter points → experimental data
  + Line → linear trend for **process prediction**

## **7. Classroom Exercises**

1. Plot **temperature vs enthalpy data** and fit a linear regression line.
2. Fit **bubble point vs mole fraction data** using polyfit.
3. Compare **experimental and simulated VLE data** on one plot.
4. Extend to **polynomial regression** for non-linear experimental data.

## **8. Key Points / Summary**

* **Matplotlib** is the standard Python library for **2D plotting**.
* **Scatter plots** and **line plots** can be combined for clear visualization.
* **Least Squares Regression** can be done manually or using np.polyfit.
* Visualization aids in **process understanding, trend analysis, and model validation**.
* Similar workflow as SCILAB, but with **Python’s NumPy and Matplotlib integration**.

**Lecture 28: Comparative Plotting – Temperature Profile in a Cooling Fin**

## **1. Introduction**

* **Motivation:**
  + In chemical engineering, **heat transfer analysis** often involves **temperature distribution along a fin**.
  + Comparative plotting allows engineers to **visualize differences** between:
    - Analytical solutions vs numerical simulations
    - Different fin materials or geometries
    - Steady-state vs transient conditions
* **Objective:**
  + Learn to create **comparative plots** in Python (Matplotlib) and SCILAB.
  + Apply to **temperature profiles of cooling fins**.

## **2. Temperature Distribution in a Fin**

* **Governing equation (steady-state, 1D fin):**

d2Tdx2−hPkA(T−T∞)=0

* **Analytical solution for a straight fin with tip at adiabatic condition:**

θ(x)=T(x)−T∞Tb−T∞=cosh[m(L−x)]cosh(mL)

Where:

* m=hPkA
* L = fin length
* Tb= base temperature
* T∞ = ambient temperature
* This provides **temperature profile along the fin**, which can be compared **for different materials or boundary conditions**.

## **3. Generating Data for Comparative Plot**

* **Python Example:**

import numpy as np

# Fin parameters

L = 0.1 # m

Tb = 373 # K

Tinf = 293 # K

h = 100 # W/m2K

P = 0.02 # m

k1 = 200 # W/mK (Material 1)

k2 = 400 # W/mK (Material 2)

A = 1e-4 # m2

x = np.linspace(0, L, 50) # discretized fin length

m1 = np.sqrt(h\*P/(k1\*A))

m2 = np.sqrt(h\*P/(k2\*A))

theta1 = np.cosh(m1\*(L - x))/np.cosh(m1\*L)

theta2 = np.cosh(m2\*(L - x))/np.cosh(m2\*L)

T1 = Tinf + theta1\*(Tb - Tinf)

T2 = Tinf + theta2\*(Tb - Tinf)

## **4. Comparative Plotting in Python**

import matplotlib.pyplot as plt

plt.plot(x, T1, 'r-', label='Material 1 (k=200 W/mK)')

plt.plot(x, T2, 'b--', label='Material 2 (k=400 W/mK)')

plt.xlabel('Fin Length (m)')

plt.ylabel('Temperature (K)')

plt.title('Temperature Profile Along Cooling Fin')

plt.legend()

plt.grid(True)

plt.show()

* **Interpretation:**
  + Red solid line → Material 1
  + Blue dashed line → Material 2
  + Shows **effect of thermal conductivity on temperature drop** along the fin

## **5. Comparative Plotting in SCILAB**

L = 0.1; Tb = 373; Tinf = 293; h = 100; P = 0.02;

k1 = 200; k2 = 400; A = 1e-4;

x = linspace(0,L,50);

m1 = sqrt(h\*P/(k1\*A));

m2 = sqrt(h\*P/(k2\*A));

theta1 = cosh(m1\*(L-x))/cosh(m1\*L);

theta2 = cosh(m2\*(L-x))/cosh(m2\*L);

T1 = Tinf + theta1\*(Tb-Tinf);

T2 = Tinf + theta2\*(Tb-Tinf);

plot(x, T1, 'r-')

plot(x, T2, 'b--')

xlabel("Fin Length (m)")

ylabel("Temperature (K)")

title("Temperature Profile Along Cooling Fin")

legend(["Material 1","Material 2"])

grid()

* **Comparison:**
  + Workflow in SCILAB is very similar to Python
  + Differences mainly in **syntax and plotting functions**

## **6. Practical Exercises**

1. Compare temperature profiles for **three different materials** with varying thermal conductivity.
2. Compare **adiabatic vs convective tip boundary conditions**.
3. Compare **analytical solution vs numerical solution** (using finite differences).
4. Plot **temperature drop along fin for different base temperatures**.

## **7. Key Points / Summary**

* Comparative plots are essential for **analyzing the effect of material, geometry, and boundary conditions**.
* Python and SCILAB allow **flexible 2D plotting with multiple curves**.
* Temperature profile along a fin is a **classic heat transfer problem**, demonstrating practical engineering analysis.
* Enables **visual comparison of multiple scenarios**, which aids in **design and optimization**.

# **Lecture 29: Plotting Concentration vs Time for a Batch Reactor**

## **1. Introduction**

* In chemical reaction engineering, batch reactors are commonly used to study **kinetics** and **reaction rates**.
* The concentration of reactants decreases with time, while product concentration increases.
* **Plotting concentration vs. time** helps visualize reaction behaviour and validate kinetic models.

### **General rate law (for a single reactant, A → Products):**

dCAdt=−rA

For first-order reactions:

CA(t)=CA0e−kt

## **2. Key Learning Objectives**

1. Understand how concentration changes with time in a batch reactor.
2. Apply simple kinetic equations (first-order & second-order).
3. Learn plotting techniques in **SCILAB** and **Python (Matplotlib)**.

## **3. Batch Reactor Kinetics Examples**

### **First-order reaction (A → Products):**

CA(t)=CA0e−kt

* CA0: initial concentration (mol/L)
* k: rate constant (1/time)
* t: time

### **Second-order reaction (2A → Products):**

CA(t)=CA0+kt

## **4. Example Problem**

* Initial concentration: CA0=1.0 mol/L
* Rate constant: k=0.1 min−1
* Time range: 0≤t≤50 min

For first-order:

CA(t)=1.0⋅e−0.1t

## **5. Implementation in SCILAB**

// Batch reactor concentration vs time (First-order reaction)

clc; clear; clf();

Ca0 = 1; // initial concentration

k = 0.1; // rate constant

t = 0:1:50; // time vector

Ca = Ca0 \* exp(-k \* t);

plot(t, Ca, 'r-');

xlabel("Time (min)");

ylabel("Concentration (mol/L)");

title("Batch Reactor: Concentration vs Time (First-order)");

## **6. Implementation in Python (Matplotlib)**

import numpy as np

import matplotlib.pyplot as plt

# Parameters

Ca0 = 1.0 # initial concentration (mol/L)

k = 0.1 # rate constant (1/min)

t = np.linspace(0, 50, 100)

# Concentration profile (first-order)

Ca = Ca0 \* np.exp(-k \* t)

# Plot

plt.plot(t, Ca, 'b-', linewidth=2, label="First-order")

plt.xlabel("Time (min)")

plt.ylabel("Concentration (mol/L)")

plt.title("Batch Reactor: Concentration vs Time")

plt.grid(True)

plt.legend()

plt.show()

## **7. Comparison of First vs Second Order**

* Add second-order case in the same plot.

**SCILAB:**

Ca2 = 1 ./ (1/Ca0 + k\*t); // second order

plot(t, Ca, 'r-', t, Ca2, 'b--');

legend("First-order", "Second-order");

**Python:**

Ca2 = 1 / (1/Ca0 + k\*t) # second-order

plt.plot(t, Ca, 'r-', label="First-order")

plt.plot(t, Ca2, 'g--', label="Second-order")

plt.legend()

## **8. Chemical Engineering Insights**

* **First-order**: Exponential decay, common in radioactive decay & hydrolysis reactions.
* **Second-order**: Faster decay, often for **bimolecular reactions**.
* Graphs allow:
  + Identifying reaction order.
  + Estimating **rate constant (k)** from slope.
  + Comparing model predictions with experimental data.

# **Lecture 30: Newton–Raphson Method (Equilibrium Constant Calculation)**

## **1. Introduction**

* In chemical engineering, equilibrium problems often require solving **nonlinear equations**.
* Examples: chemical equilibrium, vapor–liquid equilibrium, reaction extent, etc.
* Analytical solutions may not exist → need **numerical methods**.
* **Newton–Raphson method** is a widely used iterative method for solving nonlinear equations.

## **2. Newton–Raphson Method: Concept**

For a nonlinear equation:

f(x)=0

The Newton–Raphson iteration is:

xn+1=xn−f(xn)f′(xn) = current guess

* f(xn) = function value
* f′(xn) = derivative (slope) at xn

Convergence is fast if:

* Initial guess is close to true root.
* Function is differentiable and slope not too small.

## **3. Application in Chemical Equilibrium**

### **Example Problem**

Reaction:

A↔BA \leftrightarrow B

Equilibrium constant:

K=CBCA

If initial moles of A = 1, and extent of reaction = ξ:

* CA=1−ξ
* CB=ξ

At equilibrium:

K=ξ1−ξ

Rearranged:

f(ξ)=ξ1−ξ−K=0

Derivative:

f′(ξ)=1(1−ξ)2

Newton–Raphson iteration:

ξn+1=ξn−ξn1−ξn−K1(1−ξn)2

## **4. Example Calculation**

Suppose K=4, initial guess ξ0=0.5.

Iteration formula simplifies:

ξn+1=ξn−[ξn1−ξn−4](1−ξn)2

Do 2–3 iterations to show convergence in class.

## **5. Implementation in SCILAB**

// Newton-Raphson Method for Equilibrium Constant

clc; clear;

K = 4; // Equilibrium constant

xi = 0.5; // Initial guess

tol = 1e-6; // Tolerance

maxIter = 50;

for i = 1:maxIter

f = xi/(1 - xi) - K; // function

df = 1/((1 - xi)^2); // derivative

xi\_new = xi - f/df; // update

if abs(xi\_new - xi) < tol then

break;

end

xi = xi\_new;

end

disp("Equilibrium extent of reaction: " + string(xi\_new));

## **6. Implementation in Python**

# Newton-Raphson for equilibrium constant

K = 4.0 # Equilibrium constant

xi = 0.5 # Initial guess

tol = 1e-6

max\_iter = 50

for i in range(max\_iter):

f = xi / (1 - xi) - K

df = 1 / (1 - xi)\*\*2

xi\_new = xi - f / df

if abs(xi\_new - xi) < tol:

break

xi = xi\_new

print("Equilibrium extent of reaction:", xi\_new)

## **7. Chemical Engineering Insights**

* Newton–Raphson is powerful for solving **reaction equilibrium problems** where no closed-form solution exists.
* Useful for **multi-reaction equilibrium** (system of equations).
* **Drawback:** Requires derivative and good initial guess.
* Compared with **Bisection/Secant methods**, Newton–Raphson is **faster** but less stable if guess is poor.

## **8. Classroom Demonstration Ideas**

1. Start with a simple **A ↔ B** system.
2. Show students how to derive **f(ξ)** and **f′(ξ)**.
3. Perform 2 manual iterations on the board.
4. Run SCILAB/Python script to confirm.
5. Discuss extension to **real equilibria** (e.g., gas-phase dissociation, VLE equations).

# **Lecture 32: Numerical Differentiation: Forward & Backward Newton’s Difference Methods**

## **1. Introduction**

* In chemical engineering, we often need **derivatives of experimental data** (e.g., reaction rates from concentration vs. time data, heat transfer coefficients from temperature profiles).
* Analytical differentiation may not be possible when only **discrete data points** are available.
* **Numerical differentiation** uses difference methods to approximate derivatives.

## **2. Basics of Finite Differences**

Let the function values be tabulated:

x0,x1,x2,…,xn with step size h

yi=f(xi)

Define:

* **Forward difference**:

Δyi=yi+1−yi

* **Backward difference**:

∇yi=yi−yi−1

Higher-order differences:

Δ2yi=Δyi+1−Δyi

## **3. Newton’s Forward Difference Formula for Derivatives**

Applicable when we want derivative near the **beginning of the table**.

* Interpolation formula:

f(x)≈y0+pΔy0+p(p−1)2!Δ2y0+…

p=x−x0h

Differentiating with respect to x:

f′(x0)≈1h[Δy0−12Δ2y0+13Δ3y0−… ]

## **4. Newton’s Backward Difference Formula for Derivatives**

Applicable near the **end of the table**.

* Interpolation formula:

f(x)≈yn+p∇yn+p(p+1)2!∇2yn+…

where

p=x−xnh

Differentiating:

f′(xn)≈1h[∇yn+12∇2yn+13∇3yn+… ]

## **5. Example Problem**

Given data:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **x** | **0** | **1** | **2** | **3** |
| y | 1 | 2 | 4 | 7 |

* Step size h=1.
* Compute f′(0) using forward difference, and f′(3) using backward difference.

**Forward difference table:**

Δy0=2−1=1, Δ2y0=(4−2)−(2−1)=1, Δ3y0=0

At x0=0:

f′(0)≈11[1−12(1)+0]=0.5

**Backward difference table at x3 = 3:**

∇y3=7−4=3,∇2y3=(7−4)−(4−2)=1,∇3y3=0\nabla y\_3 = 7-4 = 3, \quad \nabla^2 y\_3 = (7-4) - (4-2) = 1, \quad \nabla^3 y\_3 = 0

At x3=3:

f′(3)≈11[3+12(1)]=3.5

## **6. Implementation in SCILAB**

// Numerical differentiation using forward and backward difference

clc; clear;

x = [0 1 2 3];

y = [1 2 4 7];

h = x(2) - x(1);

// Forward difference derivative at x0

dy1 = y(2) - y(1);

dy2 = (y(3) - y(2)) - (y(2) - y(1));

fwd = (dy1 - 0.5\*dy2)/h;

disp("Forward difference f'(0) = " + string(fwd));

// Backward difference derivative at xn

n = length(x);

dy1b = y(n) - y(n-1);

dy2b = (y(n) - y(n-1)) - (y(n-1) - y(n-2));

bwd = (dy1b + 0.5\*dy2b)/h;

disp("Backward difference f'(3) = " + string(bwd));

## **7. Implementation in Python**

import numpy as np

# Data

x = np.array([0, 1, 2, 3], float)

y = np.array([1, 2, 4, 7], float)

h = x[1] - x[0]

# Forward difference derivative at x0

dy1 = y[1] - y[0]

dy2 = (y[2] - y[1]) - (y[1] - y[0])

fwd = (dy1 - 0.5\*dy2) / h

print("Forward difference f'(0) =", fwd)

# Backward difference derivative at xn

dy1b = y[-1] - y[-2]

dy2b = (y[-1] - y[-2]) - (y[-2] - y[-3])

bwd = (dy1b + 0.5\*dy2b) / h

print("Backward difference f'(3) =", bwd)

## **8. Chemical Engineering Applications**

* Determining **reaction rates** from concentration vs. time data.
* Estimating **heat flux** from temperature profiles.
* Finding **velocity gradients** in fluid mechanics.
* Approximating slopes when only tabulated data is available.

✅ **Summary for Lecture**

* Introduced forward & backward Newton’s difference methods.
* Derived formulas for first derivatives.
* Solved example problem with data table.
* Implemented in SCILAB & Python.
* Discussed applications in chemical engineering.

# **Lecture 33: Numerical Integration: Trapezoidal Rule & Simpson’s Rule**

## **1. Introduction**

* In chemical engineering, we often need to **integrate experimental data** (e.g., reactor design equations, heat duty, concentration–time data).
* Analytical integration may not be possible when only **discrete data points** are available.
* **Numerical integration** provides approximate solutions.
* Two common methods:
  1. **Trapezoidal Rule**
  2. **Simpson’s Rule**

## **2. General Integral**

We want to evaluate:

I=∫abf(x) dx

If f(x) is known only at **n+1 equally spaced points**:

x0,x1,x2,…,xn(h=b−an)

with values

yi=f(xi)

## **3. Trapezoidal Rule**

* Approximates the area under curve using **trapezoids**.
* Formula:

I≈h2[y0+2(y1+y2+⋯+yn−1)+yn]

### **Example:**

∫01(x2+1) dx

* Exact solution = [x33+x]01=43
* With n=2, h=0.5:
  + y0=f(0)=1
  + y1=f(0.5)=1.25
  + y2=f(1)=2

I≈0.52(1+2(1.25)+2)=1.4167 (close to 1.3333)

## **4. Simpson’s Rule**

* Uses **parabolic segments** for better accuracy.
* Requires **n to be even** (subintervals).

### **Formula (Simpson’s 1/3 Rule):**

I≈h3[y0+4(y1+y3+y5+… )+2(y2+y4+… )+yn]

### **Example:**

Same integral ∫01(x2+1) dx, n=2, h=0.5.

I≈0.53[1+4(1.25)+2]=1.3333 (exact!)

## **5. Implementation in SCILAB**

// Numerical integration: Trapezoidal and Simpson’s Rule

clc; clear;

function y = f(x)

y = x.^2 + 1;

endfunction

a = 0; b = 1; n = 2;

h = (b - a) / n;

x = a:h:b;

y = f(x);

// Trapezoidal rule

I\_trap = h/2 \* (y(1) + 2\*sum(y(2:$-1)) + y($));

disp("Trapezoidal Rule = " + string(I\_trap));

// Simpson's 1/3 Rule

if modulo(n,2)<>0 then

error("n must be even for Simpson's rule");

end

I\_simp = h/3 \* (y(1) + 4\*sum(y(2:2:$-1)) + 2\*sum(y(3:2:$-2)) + y($));

disp("Simpson's Rule = " + string(I\_simp));

## **6. Implementation in Python**

import numpy as np

# Function

def f(x):

return x\*\*2 + 1

a, b = 0, 1

n = 2

h = (b - a) / n

x = np.linspace(a, b, n+1)

y = f(x)

# Trapezoidal Rule

I\_trap = h/2 \* (y[0] + 2\*sum(y[1:-1]) + y[-1])

print("Trapezoidal Rule =", I\_trap)

# Simpson's 1/3 Rule

if n % 2 != 0:

raise ValueError("n must be even for Simpson's rule")

I\_simp = h/3 \* (y[0] + 4\*sum(y[1:-1:2]) + 2\*sum(y[2:-2:2]) + y[-1])

print("Simpson's Rule =", I\_simp)

## **7. Chemical Engineering Applications**

* **Reactor design equation**:

V=∫CACA0dCA−rAV = \int\_{C\_A}^{C\_{A0}} \frac{dC\_A}{-r\_A}

* **Heat exchanger duty**:

Q=∫AUΔT dAQ = \int\_A U \Delta T \, dA

* **Diffusion and transport processes** where data is experimental.
* **Tabulated data** (enthalpy vs temperature, Cp vs T).

## **8. Comparison**

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Accuracy** | **Requirement** | **Use case** |
| Trapezoidal | Moderate | Works for any n | Quick estimates |
| Simpson’s 1/3 | High | Requires even n | Smooth functions, better accuracy |

**Lecture 34:** **ODE Solving in SCILAB: ode Function**

## **1. Introduction**

* Many problems in **chemical engineering** involve **ordinary differential equations (ODEs)**:
  + Reactor design (batch reactor concentration vs time).
  + Heat transfer (cooling/heating of bodies).
  + Fluid dynamics (velocity profile).
* Analytical solutions are not always possible → **numerical solvers** are required.
* In **SCILAB**, the built-in function ode is used for solving **initial value problems (IVPs)**.

## **2. General Form of ODEs in SCILAB**

The ODE to solve:

dydt=f(t,y),y(t0)=y0

SCILAB syntax:

y = ode(y0, t0, t, f)

Where:

* y0 → initial value(s) (scalar or vector).
* t0 → initial time.
* t → vector of times where solution is required.
* f → function defining RHS of ODE (dy/dt).

## **3. Example 1: First-order Reaction in a Batch Reactor**

Reaction: A→B

dCAdt=−kCA

Analytical solution:

CA(t)=CA0e−kt

### SCILAB Code:

clc; clear;

// Define rate constant

k = 0.1;

// Define ODE function: dCa/dt = -k\*Ca

function dCa = f(t, Ca)

dCa = -k \* Ca;

endfunction

// Initial conditions

Ca0 = 1.0; // mol/L

t0 = 0; // start time

t = 0:1:50; // time vector

// Solve ODE

Ca = ode(Ca0, t0, t, f);

// Plot

plot(t, Ca, 'b-');

xlabel("Time (min)");

ylabel("Concentration of A (mol/L)");

title("Batch Reactor: ODE Solution using SCILAB ode()");

## **4. Example 2: Heating of a Solid Object**

Equation:

dTdt=UAρCpV(T∞−T)\frac{dT}{dt} = \frac{U A}{\rho C\_p V}(T\_\infty - T)

Suppose:

* UA/ρCpV=0.2 min−1U A / \rho C\_p V = 0.2 \ \text{min}^{-1}
* T∞=350 KT\_\infty = 350 \,K, T(0)=300 KT(0) = 300 \,K

### SCILAB Code:

clc; clear;

k = 0.2; // heat transfer parameter

Tinf = 350; // surrounding temperature

function dT = cooling(t, T)

dT = k \* (Tinf - T);

endfunction

T0 = 300;

t0 = 0;

t = 0:1:50;

T = ode(T0, t0, t, cooling);

plot(t, T, 'r-');

xlabel("Time (min)");

ylabel("Temperature (K)");

title("Heating of a solid object using ode()");

## **5. Example 3: System of ODEs (CSTR with A → B)**

Equations:

dCAdt=CA,in−CAτ−kCA

dCBdt=−CBτ+kCA

### SCILAB Code:

clc; clear;

// Parameters

k = 0.1;

tau = 5;

CAin = 1.0;

// System of ODEs

function dydt = cstr(t, y)

Ca = y(1);

Cb = y(2);

dCa = (CAin - Ca)/tau - k\*Ca;

dCb = -Cb/tau + k\*Ca;

dydt = [dCa; dCb];

endfunction

// Initial conditions

y0 = [0; 0]; // [Ca, Cb]

t0 = 0;

t = 0:1:50;

// Solve

y = ode(y0, t0, t, cstr);

// Extract results

Ca = y(1,:);

Cb = y(2,:);

plot(t, Ca, 'b-', t, Cb, 'r--');

xlabel("Time (min)");

ylabel("Concentration (mol/L)");

legend("C\_A", "C\_B");

title("CSTR dynamics solved with ode()");

## **6. Chemical Engineering Applications of ODE Solvers**

* **Kinetics**: Batch and CSTR reactors.
* **Heat Transfer**: Cooling/heating of objects, unsteady conduction.
* **Mass Transfer**: Diffusion problems.
* **Process Control**: First-order and second-order dynamic systems.

## **7. Classroom Demonstration Strategy**

1. Start with simple **first-order ODE** (reaction kinetics).
2. Show **plot of concentration vs time** (compare with analytical solution).
3. Extend to **heat transfer problem** (Newton’s law of cooling).
4. Introduce **system of ODEs** (CSTR).
5. Highlight importance of SCILAB’s ode for **chemical engineering simulations**.

✅ **Summary for Lecture**

* Introduced **SCILAB ode function** for ODE solving.
* Demonstrated with **batch reactor**, **heat transfer**, and **CSTR** examples.
* Discussed **applications in chemical engineering**.

Would you like me to prepare the **next lecture** on **ODE solving in Python (using scipy.integrate.odeint)** as a parallel continuation to this SCILAB lecture?

# **Lecture 35:** **ODE Solving in Python: scipy.integrate.odeint**

## **1. Introduction**

* Many problems in **chemical engineering** involve **ODEs (ordinary differential equations)**:
  + Reactor dynamics (batch reactor, CSTR).
  + Heat transfer (cooling fins, heating objects).
  + Mass transfer (diffusion models).
* Python’s **SciPy library** provides odeint for solving **initial value problems (IVPs)**.

## **2. General Form**

ODE to solve:

dy/dt=f(y,t), y(t0)=y0

Python syntax:

from scipy.integrate import odeint

y = odeint(func, y0, t)

Where:

* func(y, t) → function returning RHS (dy/dt).
* y0 → initial condition(s) (scalar or vector).
* t → time points for solution.
* Returns y → solution array.

## **3. Example 1: Batch Reactor (First-order reaction)**

dCAdt=−kCA

### Python Code

import numpy as np

from scipy.integrate import odeint

import matplotlib.pyplot as plt

# Parameters

k = 0.1

# ODE function

def dCa\_dt(Ca, t):

return -k \* Ca

# Initial condition

Ca0 = 1.0

t = np.linspace(0, 50, 100)

# Solve

Ca = odeint(dCa\_dt, Ca0, t)

# Plot

plt.plot(t, Ca, 'b-')

plt.xlabel("Time (min)")

plt.ylabel("Concentration of A (mol/L)")

plt.title("Batch Reactor: First-order kinetics")

plt.show()

✅ Compare result with analytical solution CA(t)=CA0e−kt.

## **4. Example 2: Heating of a Solid (Newton’s Law of Cooling/Heating)**

dTdt=k(T∞−T)

### Python Code

k = 0.2

T\_inf = 350

def dT\_dt(T, t):

return k \* (T\_inf - T)

T0 = 300

t = np.linspace(0, 50, 100)

T = odeint(dT\_dt, T0, t)

plt.plot(t, T, 'r-')

plt.xlabel("Time (min)")

plt.ylabel("Temperature (K)")

plt.title("Heating of a Solid Object")

plt.show()

## **5. Example 3: System of ODEs (CSTR with A → B)**

Equations:

dCAdt=CA,in−CAτ−kCA

dCBdt=−CBτ+kCA

### Python Code

# Parameters

k = 0.1

tau = 5

CA\_in = 1.0

def cstr(y, t):

Ca, Cb = y

dCa = (CA\_in - Ca)/tau - k\*Ca

dCb = -Cb/tau + k\*Ca

return [dCa, dCb]

y0 = [0, 0] # Initial concentrations

t = np.linspace(0, 50, 100)

sol = odeint(cstr, y0, t)

Ca = sol[:,0]

Cb = sol[:,1]

plt.plot(t, Ca, 'b-', label="C\_A")

plt.plot(t, Cb, 'r--', label="C\_B")

plt.xlabel("Time (min)")

plt.ylabel("Concentration (mol/L)")

plt.legend()

plt.title("CSTR Dynamics using odeint")

plt.show()

## **6. Applications in Chemical Engineering**

* **Reaction Engineering**: Batch, PFR, CSTR dynamics.
* **Heat Transfer**: Transient conduction, cooling/heating processes.
* **Mass Transfer**: Diffusion models, film theory.
* **Process Control**: First- and second-order dynamic systems.

## **7. Classroom Demonstration Strategy**

1. Begin with **batch reactor** example (single ODE).
2. Extend to **heat transfer** problem (Newton’s law of cooling).
3. Introduce **system of ODEs** (CSTR).
4. Show **plotting of solutions** with matplotlib.
5. Emphasize **SciPy’s power** for engineering problem-solving.

# **Lecture 36: Case Study: Cooling of a Hot Sphere (Unsteady Heat Transfer)**

## **1. Introduction**

* Many **chemical engineering operations** involve unsteady heat transfer:
  + Cooling/heating of catalyst particles.
  + Solid fuel combustion.
  + Quenching of metal spheres.
* For a **small Biot number (Bi < 0.1)**, **lumped capacitance method** can be applied:
  + Temperature inside the solid is **uniform** at any time.
  + Heat transfer is controlled by **convection** at the surface.

## **2. Governing Equation (Lumped Model)**

For a sphere cooling in a fluid:

dTdt=−hAρcpV(T−T∞)

Where:

* T = sphere temperature (K)
* T∞= fluid temperature (K)
* h = convective heat transfer coefficient (W/m²·K)
* A = surface area of sphere = 4πR2
* V = volume of sphere = 43πR3
* ρ = density of solid (kg/m³)
* cp = specific heat (J/kg·K)
* R = sphere radius (m)

## **3. Dimensionless Form**

dθdt=−hAρcpV

Where:

θ=T−T∞

Analytical solution:

T(t)=T∞+(T0−T∞)exp(−hAρcpVt)

## **4. Example Problem**

* **Sphere radius (R):** 0.01 m
* **Initial temperature (T0):** 500 K
* **Fluid temperature (T∞):** 300 K
* **h:** 100 W/m²·K
* **ρ:** 8000 kg/m³
* **cp:** 500 J/kg·K

### **Step 1: Calculate constants**

* Surface area A=4πR2 = 0.00126 m²
* Volume V=43πR3=4.19×10−6 m³
* Mass = ρV= 0.0335 kg
* Heat capacity = mcp=16.75 J/K
* Time constant = τ=ρcpVhA=13333 s

### **Step 2: Cooling equation**

T(t)=300+(500−300)e−t/13333

## **5. Python Implementation**

import numpy as np

import matplotlib.pyplot as plt

from scipy.integrate import odeint

# Parameters

R = 0.01

h = 100

rho = 8000

cp = 500

T\_inf = 300

T0 = 500

A = 4\*np.pi\*R\*\*2

V = (4/3)\*np.pi\*R\*\*3

# ODE function

def dT\_dt(T, t):

return -(h\*A)/(rho\*cp\*V) \* (T - T\_inf)

# Time grid

t = np.linspace(0, 20000, 200)

# Solve

T = odeint(dT\_dt, T0, t)

# Plot

plt.plot(t, T, 'b-', label="Sphere Temp")

plt.axhline(T\_inf, color='r', linestyle='--', label="Fluid Temp")

plt.xlabel("Time (s)")

plt.ylabel("Temperature (K)")

plt.title("Cooling of a Hot Sphere")

plt.legend()

plt.show()

## **6. SCILAB Implementation**

// Parameters

R = 0.01;

h = 100;

rho = 8000;

cp = 500;

Tinf = 300;

T0 = 500;

A = 4\*%pi\*R^2;

V = (4/3)\*%pi\*R^3;

// ODE definition

function dT = sphereCooling(t, T)

dT = -(h\*A)/(rho\*cp\*V) \* (T - Tinf);

endfunction

// Time range

t = 0:100:20000;

// Solve

T = ode(T0, t, sphereCooling);

// Plot

plot(t, T, 'b')

xtitle("Cooling of a Hot Sphere", "Time (s)", "Temperature (K)")

## **7. Applications in Chemical Engineering**

* **Catalyst pellet cooling/heating** in fluidized beds.
* **Quenching operations** in metallurgy.
* **Food processing** (e.g., cooling of spherical fruits/particles).
* **Pharmaceuticals** (cooling of drug pellets in drying).

## **8. Classroom Strategy**

1. Begin with **physical intuition** (sphere cooling in air/water).
2. Derive governing equation via **energy balance**.
3. Show **analytical exponential decay**.
4. Implement in **Python and SCILAB**.
5. Discuss **applications** in chemical engineering.

**Lecture 37: Case Study: Chemical Reactor Kinetics Simulation (1st Order Reaction)**

## **1. Introduction**

* **Chemical reaction kinetics** describe how fast reactants are consumed and products are formed.
* A **first-order reaction** is the simplest kinetic model:

A    →k    Products −rA=kCA

* Appears in:
  + **Batch reactors** (unsteady-state).
  + **CSTRs & PFRs** (steady-state).
  + Examples: decomposition of N₂O₅, radioactive decay, hydrolysis of esters.

## **2. Governing Equation (Batch Reactor)**

Material balance on **species A**:

dCAdt=−kCA

Where:

* CA = concentration of A (mol/L)
* k = reaction rate constant (1/time)

**Analytical solution:**

CA(t)=CA0e−kt

## **3. Example Problem**

* Reaction: A→B
* Parameters:
  + CA0=1.0 mol/L
  + k=0.1 min−1
  + Time: 0–50 min

**Expected behaviour:** Exponential decay of CA.

## **4. Python Simulation (Batch Reactor)**

import numpy as np

from scipy.integrate import odeint

import matplotlib.pyplot as plt

# Parameters

k = 0.1

Ca0 = 1.0

# ODE definition

def dCa\_dt(Ca, t):

return -k \* Ca

# Time grid

t = np.linspace(0, 50, 100)

# Solve ODE

Ca = odeint(dCa\_dt, Ca0, t)

# Plot

plt.plot(t, Ca, 'b-', label="C\_A(t)")

plt.xlabel("Time (min)")

plt.ylabel("Concentration (mol/L)")

plt.title("Batch Reactor: First-order kinetics")

plt.legend()

plt.show()

✅ Produces an **exponential decay curve** for CA(t).

## **5. SCILAB Simulation**

// Parameters

k = 0.1;

Ca0 = 1.0;

// ODE definition

function dCa = batchReactor(t, Ca)

dCa = -k \* Ca;

endfunction

// Time points

t = 0:0.5:50;

// Solve ODE

Ca = ode(Ca0, t, batchReactor);

// Plot

plot(t, Ca, 'b')

xtitle("Batch Reactor: First-order kinetics", "Time (min)", "Concentration (mol/L)")

## **6. Conversion in Reactors**

### **(a) Batch Reactor**

Conversion:

X=1−CACA0

### **(b) CSTR (Continuous Stirred Tank Reactor)**

At steady state:

CA=CA01+kτ

Where τ = residence time (V/F).

### **(c) PFR (Plug Flow Reactor)**

CA=CA0exp(−kτ)

## **7. Comparative Example**

* Take k=0.1 min−1,τ=10 min.
* CA values:
  + Batch (at 10 min): CA=CA0e−kt=0.37
  + CSTR: CA = 0.5.
  + PFR: CA = 0.37.

✅ Shows **different reactor performance** for the same kinetics.

## **8. Applications in Chemical Engineering**

* **Reaction design** (reactor sizing, conversion prediction).
* **Process simulation** (batch vs. continuous).
* **Pharmaceuticals** (drug degradation kinetics).
* **Environmental engineering** (pollutant decay, wastewater treatment).

## **9. Classroom Strategy**

1. Start with **real-life chemical examples** of first-order kinetics.
2. Derive batch reactor ODE and solve analytically.
3. Use **Python/SCILAB simulation** to visualize.
4. Compare **Batch, CSTR, PFR** results.
5. Discuss **engineering implications**: efficiency, reactor design.

## ✅ **Summary**

* First-order kinetics: rA=kCA.
* Batch reactor ODE solved by odeint (Python) and ode (SCILAB).
* Analytical and numerical results match (exponential decay).
* Extended to **CSTR and PFR** models.
* Strong foundation for **reactor design and simulation**.

Would you like me to **extend this case study** in the **next lecture** to include **second-order kinetics (e.g., A+B→Products)** so students see how reaction order changes reactor behaviour?

# **Lecture 38: Fluid Flow Problems: Pressure Drop in Pipes**

## **1. Introduction**

* In **chemical engineering**, fluid transport in pipelines is essential (reactors, heat exchangers, distillation columns).
* Pressure drop in a pipe directly impacts:
  + **Pumping power requirements**.
  + **Flow rates** and energy consumption.
  + **Process efficiency** and safety.
* Understanding pressure drop is a **fundamental skill** for engineers.

## **2. Governing Equation (Darcy–Weisbach Equation)**

ΔP=fLDρu22

Where:

* ΔP= pressure drop (Pa)
* f = Darcy friction factor (dimensionless)
* L = pipe length (m)
* D = pipe diameter (m)
* ρ = fluid density (kg/m³)
* u = average velocity (m/s)

## **3. Reynolds Number**

Re=ρuDμ

* **Laminar flow (Re < 2100):**  
  f=16/Re
* **Turbulent flow (Re > 4000):**  
  f depends on Re and pipe roughness.
  + Use **Colebrook–White equation**:

1f=−2log10(ϵ/D3.7+2.51Ref) (ϵ = pipe roughness).

## **4. Head Loss (Energy Form)**

hf=fLDu22g

Where:

* hf = head loss (m of fluid)
* g = gravitational acceleration

Relation to pumping power:

Ppump=ΔP⋅Q

Where Q = volumetric flow rate.

## **5. Example Problem**

Water at 25°C flows in a pipe:

* D=0.05 m, L=10 m
* u=1 m/s, ρ=1000 kg/m3, μ=0.001 Pa·s

**Step 1: Reynolds number**

Re= ρuD/μ= (100010.05)/0.001=50,000 (turbulent)

**Step 2: Friction factor (Moody/Colebrook)**  
Assume smooth pipe: f≈0.018.

**Step 3: Pressure drop**

ΔP=0.018100.051000122≈1800 Pa

## **Python**

import numpy as np

from scipy.optimize import fsolve

# Parameters

D = 0.05 # m

L = 10 # m

u = 1.0 # m/s

rho = 1000 # kg/m3

mu = 0.001 # Pa.s

eps = 0.0001 # roughness (m)

# Reynolds number

Re = rho \* u \* D / mu

# Colebrook equation solver

def colebrook(f):

return 1/np.sqrt(f) + 2\*np.log10((eps/D)/3.7 + 2.51/(Re\*np.sqrt(f)))

f\_guess = 0.02

f\_solution = fsolve(colebrook, f\_guess)[0]

# Pressure drop

dP = f\_solution \* (L/D) \* (rho\*u\*\*2/2)

print("Re =", Re)

print("Friction factor =", f\_solution)

print("Pressure drop (Pa) =", dP)

## **SCILAB**

// Parameters

D = 0.05; L = 10; u = 1; rho = 1000; mu = 0.001; eps = 0.0001;

// Reynolds number

Re = rho\*u\*D/mu;

// Colebrook equation (implicit)

function y = colebrook(f)

y = 1/sqrt(f) + 2\*log10((eps/D)/3.7 + 2.51/(Re\*sqrt(f)));

endfunction

f\_guess = 0.02;

f = fsolve(f\_guess, colebrook);

// Pressure drop

dP = f\*(L/D)\*(rho\*u^2/2);

disp(Re, "Re =")

disp(f, "Friction factor =")

disp(dP, "Pressure drop (Pa) =")

## **8. Applications in Chemical Engineering**

* **Pipeline design** (oil, natural gas, water).
* **Heat exchangers** (tube-side pressure drop).
* **Process plant utilities** (steam distribution).
* **Safety** (preventing excess energy loss and pump overload).

# **Lecture 39: Heat Exchanger Performance Calculations**

## **1. Introduction**

Heat exchangers are widely used in chemical, petrochemical, and process industries to transfer heat between two fluid streams. Performance analysis helps engineers evaluate **heat duty, outlet temperatures, efficiency, and required heat transfer area**.

## **2. Heat Exchanger Types**

* **Double pipe heat exchanger** – simple, suitable for small duties.
* **Shell-and-tube heat exchanger** – common in industries, flexible design.
* **Plate heat exchanger** – compact, suitable for liquids.

Flow arrangements:

* **Parallel flow** – both fluids move in same direction.
* **Counter flow** – fluids move in opposite directions (most efficient).
* **Cross flow** – fluids move perpendicular to each other.

## **3. Governing Equations**

### **(a) Heat Duty**

Q=mhCp,h(Th,in−Th,out)=mcCp,c(Tc,out−Tc,in)

where:

* m = mass flow rate
* Cp = specific heat
* T = temperature

### **(b) Log Mean Temperature Difference (LMTD)**

ΔTlm=(ΔT1−ΔT2)/ln(ΔT1/ΔT2)

Where,

ΔT1=Th,in−Tc,out, ΔT2=Th,out−Tc,in

**Heat transfer rate:**

Q=UAΔTlm

* U = overall heat transfer coefficient
* A = heat transfer area

### **(c) Effectiveness–NTU Method**

For cases where outlet temperatures are unknown:

ε=Q/Qmax Qmax=Cmin (Th,in−Tc,in)

where

C=m

Number of transfer units (NTU):

NTU=UACmin

Relations between ε, NTU, and heat capacity ratio (Cr=CminCmax) depend on flow arrangement:

* **Counter flow:**

ε=1−exp[−NTU(1−Cr)]1−Crexp[−NTU(1−Cr)]

* **Parallel flow:**

ε=1−exp[−NTU(1+Cr)]1+Cr

## **4. Example Problem**

Hot water at 80∘C (mass flow = 1 kg/s, Cp=4180 J/kg·K) is cooled to 50∘C by cold water at 20∘C (mass flow = 1.5 kg/s, Cp=4180 J/kg·K) in a counterflow heat exchanger. Find heat duty, cold water outlet temperature, and LMTD.

**Solution:**

* Heat lost by hot water:

Q=mhCp(Th,in−Th,out)=1×4180×(80−50)=125,400 W

* Heat gained by cold water:

Q=mcCp(Tc,out−Tc,in)

125,400=1.5×4180(Tc,out−20)

Tc,out=40∘C

* Temperature differences:

ΔT1=80−40=40, ΔT2=50−20=30

ΔTlm=(40−30)ln((40/30))=34.65∘C

## **5. Applications in Chemical Engineering**

* Condensers in distillation columns
* Reboilers in separation processes
* Cooling water exchangers in reactors
* Process-to-process heat recovery

✅ **Key Takeaways:**

* Heat exchanger performance can be analysed by **LMTD method** (if outlet temperatures are known) or **Effectiveness-NTU method** (if outlet temperatures are unknown).
* Choice of method depends on available information.
* Chemical engineering design often requires balancing between efficiency, cost, and operational flexibility.

# **Lecture 40: Diffusion Problems: Unsteady Diffusion in a Film**

## **1. Introduction**

* **Mass transfer by diffusion** is a key transport phenomenon in chemical engineering.
* Applications:
  + Gas absorption in liquids
  + Drying of solids
  + Membrane separation
  + Catalyst pellet reactions
* **Unsteady diffusion** (transient diffusion) occurs when concentration changes with both **position and time**.

## **2. Governing Equation (Fick’s Second Law)**

For **one-dimensional diffusion** in a stagnant film of thickness L:

∂C/∂t=D(∂2C/∂x2)

Where:

* C=C(x,t) → concentration (mol/m³)
* D → diffusivity (m²/s)
* x → position in film
* t → time

**Boundary conditions:**

* At x=0 (surface): C=Cs (constant surface concentration)
* At x=L (bulk): C=C∞

**Initial condition:**

C(x,0)=C∞

## **3. Analytical Solution (Constant Diffusivity)**

For **thin film** and **constant surface concentration**:

=1−

* Infinite series converges quickly; first 3–5 terms usually sufficient.

## **4. Lumped Mass Transfer Approach (High Biot Number)**

If **film is very thin**, we can use **overall mass transfer coefficient kc**:

dC/dt=kc(Cs−C)

* Solution:

C(t)=Cs+(C0−Cs)exp(−kct)

* Analogous to **Newton’s law of cooling in heat transfer**.

## **5. Example Problem**

* Gas diffusing into a stagnant liquid film:
  + Film thickness L=0.001 m
  + Diffusivity D=2×10−9 m²/s
  + Surface concentration Cs=0.01 mol/m³
  + Initial bulk concentration C∞=0

**Objective:** Find concentration at x=0.0005 m, after t=500 s.

* Using first-term approximation of series:

C−00.01−0≈1−4πexp⁡[−π2Dt4L2]sin⁡(πx2L)\frac{C - 0}{0.01 - 0} \approx 1 - \frac{4}{\pi} \exp \left[ -\frac{\pi^2 D t}{4 L^2} \right] \sin\left(\frac{\pi x}{2 L}\right) C≈0.01[1−4πexp⁡(−π2(2e−9)(500)4(0.001)2)sin⁡(π/4)]C \approx 0.01 \left[ 1 - \frac{4}{\pi} \exp \left( -\frac{\pi^2 (2e-9)(500)}{4 (0.001)^2} \right) \sin(\pi/4) \right]

* Students can **compute the numerical value** using a calculator or Python/SCILAB.

## **6. Numerical Solution (Finite Difference Method)**

* Discretize space: x=iΔx
* Discretize time: t=nΔt

**Explicit scheme:**

Cin+1=Cin+DΔt(Δx)2(Ci+1n−2Cin+Ci−1n)C\_i^{n+1} = C\_i^n + \frac{D \Delta t}{(\Delta x)^2} (C\_{i+1}^n - 2 C\_i^n + C\_{i-1}^n)

**Stability criterion:**

DΔt(Δx)2≤0.5

### **Python Example (Explicit FDM)**

import numpy as np

import matplotlib.pyplot as plt

# Parameters

L = 0.001

D = 2e-9

Nx = 11

dx = L/(Nx-1)

dt = 100

Nt = 50

C = np.zeros(Nx)

C\_s = 0.01

C\_infty = 0.0

C[0] = C\_s # surface boundary

for n in range(Nt):

C\_new = C.copy()

for i in range(1,Nx-1):

C\_new[i] = C[i] + D\*dt/dx\*\*2\*(C[i+1]-2\*C[i]+C[i-1])

C\_new[-1] = C\_infty # bulk boundary

C = C\_new.copy()

plt.plot(np.linspace(0,L,Nx),C,'b-o')

plt.xlabel("x (m)")

plt.ylabel("Concentration (mol/m³)")

plt.title("Transient Diffusion in a Film")

plt.show()

## **7. Applications**

* Gas absorption in **scrubbers**
* Drying of solids in **fluidized beds**
* **Membrane separation** (water purification)
* Catalyst pellets (**reaction-diffusion phenomena**)

## **8. Classroom Strategy**

1. Introduce **Fick’s law** and its transient form.
2. Derive **analytical solution for simple cases**.
3. Compare with **lumped mass transfer approximation**.
4. Implement **numerical solution in Python/SCILAB**.
5. Discuss **engineering relevance and applications**.

## ✅ **Summary**

* Unsteady diffusion in a film can be analyzed **analytically** (series solution) or **numerically** (finite difference).
* Boundary and initial conditions are critical.
* Analogies with **heat transfer** help in understanding.
* Practical applications in chemical reactors, separations, and mass transfer operations.

# **Lecture 41: Mass Transfer Coefficient Estimation via Correlations**

## **1. Introduction**

* **Mass transfer coefficient** (kc) quantifies the rate at which a species is transferred between phases (gas–liquid, liquid–solid).
* Important in **chemical engineering operations**:
  + Gas absorption in towers
  + Liquid–liquid extraction
  + Drying and evaporation
  + Packed bed reactors
* Direct calculation of kc from first principles is complex; **empirical correlations** are widely used.

## **2. General Form of Mass Transfer**

NA=kc(Cs−Cb)

Where:

* NA = molar flux (mol/m²·s)
* kc = mass transfer coefficient (m/s)
* Cs = concentration at the interface
* Cb = bulk concentration
* Dimensionless numbers are often used to express kc in **correlations**:
  + **Reynolds number (Re)** → flow regime
  + **Schmidt number (Sc)** → ratio of momentum to mass diffusivity

## **3. Common Mass Transfer Correlations**

### **(a) Gas–Liquid Systems (Sherwood Number)**

Sh=kcL/DAB

Where:

* Sh = Sherwood number (dimensionless)
* L = characteristic length
* DAB = diffusivity (m²/s)

#### **Examples:**

1. **Flow over a flat plate (laminar):**

Sh=0.664Re1/2Sc1/3

1. **Flow over a flat plate (turbulent):**

Sh=0.037Re0.8Sc1/3

1. **Packed bed (Carman–Kozeny type):**

Sh=2+1.1Re0.6Sc1/3

### **(b) Liquid–Solid Systems (Stirred Tanks / Agitated Systems)**

Sh=kcdp/DAB=0.42Re0.5Sc0.33

Where dp = particle diameter

## **4. Reynolds, Schmidt, and Sherwood Numbers**

Re=ρuLμ(dimensionless flow)Re = \frac{\rho u L}{\mu} \quad \text{(dimensionless flow)} Sc=μρDAB(momentum/mass diffusivity)Sc = \frac{\mu}{\rho D\_{AB}} \quad \text{(momentum/mass diffusivity)} Sh=kcLDAB(mass transfer rate)Sh = \frac{k\_c L}{D\_{AB}} \quad \text{(mass transfer rate)}

* Rearranging:

kc=Sh DABL

## **5. Example Problem**

**Problem:**

* Gas flowing in a tube: u=0.5 m/su = 0.5\,\text{m/s}, D=1.0 cmD = 1.0\,\text{cm}, ρ=1.2 kg/m³\rho = 1.2\,\text{kg/m³}, μ=1.8×10−5 Pa\cdotps\mu = 1.8\times10^{-5}\,\text{Pa·s}, DAB=2×10−5 m²/sD\_{AB} = 2\times10^{-5}\, \text{m²/s}
* Calculate kc using laminar flow correlation.

**Step 1: Calculate Re and Sc**

Re=ρuDμ=1.2⋅0.5⋅0.011.8e−5≈333Re = \frac{\rho u D}{\mu} = \frac{1.2 \cdot 0.5 \cdot 0.01}{1.8e-5} \approx 333 Sc=μρDAB=1.8e−51.2⋅2e−5=0.75Sc = \frac{\mu}{\rho D\_{AB}} = \frac{1.8e-5}{1.2 \cdot 2e-5} = 0.75

**Step 2: Sherwood number (laminar flow, flat plate):**

Sh=0.664Re1/2Sc1/3=0.664⋅3330.5⋅0.751/3≈12.2Sh = 0.664 Re^{1/2} Sc^{1/3} = 0.664 \cdot 333^{0.5} \cdot 0.75^{1/3} \approx 12.2

**Step 3: Mass transfer coefficient**

kc=Sh DABL=12.2⋅2e−50.01≈0.0244 m/sk\_c = \frac{Sh \, D\_{AB}}{L} = \frac{12.2 \cdot 2e-5}{0.01} \approx 0.0244 \, \text{m/s}

## **6. Python Implementation**

import numpy as np

# Given

rho = 1.2

u = 0.5

D = 0.01

mu = 1.8e-5

D\_AB = 2e-5

# Dimensionless numbers

Re = rho\*u\*D/mu

Sc = mu/(rho\*D\_AB)

# Sherwood number (laminar)

Sh = 0.664 \* Re\*\*0.5 \* Sc\*\*(1/3)

# Mass transfer coefficient

k\_c = Sh \* D\_AB / D

print("Re =", Re)

print("Sc =", Sc)

print("Sh =", Sh)

print("k\_c (m/s) =", k\_c)

## **7. Applications in Chemical Engineering**

* Gas absorption in **scrubbers** or packed columns
* **Distillation** – vapor–liquid mass transfer
* **Drying operations** of solids
* **Liquid–liquid extraction**
* **Bioreactors** – oxygen transfer in fermentation

## ✅ **Summary**

* **Mass transfer coefficient** (kc) is essential for design of chemical processes.
* Calculated via **empirical correlations** using **dimensionless numbers**.
* **Python/SCILAB implementations** allow easy estimation for different flow systems.
* Correlations vary for **laminar/turbulent flow**, **geometry**, and **phase type**.

# **Lecture 42: Equation of State: van der Waals Gas & Compressibility Factor**

## **1. Introduction**

* **Ideal gas law** (PV=nRT) is an approximation; real gases deviate due to:
  + **Finite molecular size**
  + **Intermolecular forces**
* **van der Waals equation of state (EOS)** accounts for these deviations:

(P+an2V2)(V−nb)=nRT

Where:

* P = pressure (Pa)
* V = volume (m³)
* T = temperature (K)
* n = moles of gas
* R = universal gas constant (8.314 J/mol·K)
* a = attraction parameter (Pa·m⁶/mol²)
* b = volume excluded by molecules (m³/mol)

## **2. Physical Meaning of van der Waals Parameters**

* a → corrects for **intermolecular attraction**, reduces effective pressure.
* b → corrects for **finite molecular volume**, reduces effective volume.
* Reduces to **ideal gas law** if a=0 and b = 0.

## **3. Compressibility Factor (Z)**

* Quantifies deviation from ideal gas:

Z=PVmRT

Where:

* Vm=V/n = molar volume
* For ideal gas: Z=1
* For real gas: Z≠1
* From van der Waals EOS:

Z=PVmRT=VmVm−b−aRTVm

## **4. Reduced Properties & Corresponding States**

* **Critical constants** of van der Waals gas:

Vc=3b, Pc=a27b2, Tc=8a27Rb

* **Reduced variables:**

Pr=P/Pc, Tr=T/Tc, Vr=Vm/Vc

* Van der Waals EOS in reduced form:

(Pr+3Vr2)(3Vr−1)=8Tr

* Allows **generalized compressibility correlations** for all gases.

## **5. Example Problem**

* **Gas:** CO₂
* **van der Waals constants:** a=3.59 Pa·m⁶/mol², b=4.27×10−5 m³/mol
* T=310 K, P=5 MPa T = 310

**Step 1: Solve cubic EOS for molar volume Vm**

(P+aVm2)(Vm−b)=RT

* This is a **cubic equation** in Vm.
* Solve **numerically** using Python/SCILAB.

**Step 2: Calculate compressibility factor**

Z=PVmRTZ

## **6. Python Implementation (Numerical Solution)**

import numpy as np

from scipy.optimize import fsolve

# Given constants for CO2

a = 3.59 # Pa·m6/mol2

b = 4.27e-5 # m3/mol

R = 8.314 # J/mol·K

T = 310

P = 5e6

# van der Waals EOS as a function of Vm

def vdW(Vm):

return (P + a/Vm\*\*2)\*(Vm - b) - R\*T

# Initial guess

Vm\_guess = R\*T/P

# Solve cubic equation

Vm = fsolve(vdW, Vm\_guess)[0]

# Compressibility factor

Z = P\*Vm/(R\*T)

print("Molar volume Vm =", Vm, "m³/mol")

print("Compressibility factor Z =", Z)

## **7. Graphical Representation**

* **Plot Z vs P at constant T**
* Shows:
  + Z<1 → attractive forces dominate (low P)
  + Z>1 → repulsive forces dominate (high P)
* Helps **visualize real gas deviations**.

## **8. Applications in Chemical Engineering**

* High-pressure reactor design
* Supercritical fluid extraction
* Gas liquefaction and compression
* Pipeline transport of natural gas

## **9. Classroom Strategy**

1. Start with **limitations of ideal gas law**.
2. Introduce **van der Waals EOS** and physical meaning of parameters.
3. Define **compressibility factor** and its significance.
4. Solve a **numerical example** using Python/SCILAB.
5. Plot **Z vs P or Z vs Vm** to visualize real gas behavior.

## ✅ **Summary**

* van der Waals EOS corrects ideal gas law for **molecular size** and **intermolecular forces**.
* Compressibility factor ZZ indicates deviation from ideality.
* Cubic EOS can be solved **numerically** for real gases.