Tutor

Dr. Alberto Cruz Alcalde
Departament d'Enginyeria Química i Química Analítica

Grau en Enginyeria Química

Treball Final de Grau

Development of a Python tool for the automated resolution of material balances

Mar Pradas Aguarón

June 2025

UNIVERSITAT DE BARCELONA

CONTENTS

SUMMARY

Macroscopic mass balances are a fundamental tool in chemical engineering for analyzing and designing processes efficiently. They constitute the first stage to be analysed in the study of chemical processes. These balances can be formulated as global balances (considering all components together) or component-wise balances, incorporating accumulation, inputs, outputs, and generation terms. Their resolution depends on factors such as the presence of chemical reactions, steady or unsteady-state conditions, and whether the process is continuous or batch.

Traditionally, these calculations are performed manually or using spreadsheets, which can be inefficient tedius, as each type of problem requires setting up a new sheet. This approach is also prone to errors, especially in complex systems involving multiple streams, recirculations, or chemical reactions. More advanced programs like Aspen offer powerful capabilities but often require numerous parameters and physical properties that may not always be available. Moreover, for preliminary calculations, such a level of detail may be unnecessary.

To address these limitations and offer an intermediate solution, this project aims to develop a Python-based computational tool that automates the resolution of steady-state macroscopic mass balances for continuous processes using a sequential solving approach. The tool handles three types of process blocks: reactors (with reactions and generation terms), separators (input-output only), and splitters (used for recirculations and purges, imposing composition equality). The algorithm allows users to draw the process diagram, input the data into a table, and iteratively solve the blocks using pre-programmed calculation rules until convergence is reached. The final output is a completed table showing the resulting flows and compositions.

The tool has been validated through classical case studies in chemical engineering introductory courses, comparing results with manual calculations and evaluating computational efficiency and accuracy. This project seeks to provide a practical and reliable solution to facilitate mass balance calculations, enhancing their applicability in both academic and industrial contexts.

Keywords: Macroscopic mass balance, Python, Chemical process simulation, Sequential solving, Calculation automation.

RESUM

Els balanços macroscòpics de matèria són una eina fonamental en l'enginyeria química per analitzar i dissenyar processos de manera eficient. Representen la primera etapa a analitzar en l'estudi dels processos químics. Aquests balanços es poden formular com a balanços globals (tenint en compte tots els components conjuntament) o com a balanços per components, incorporant termes d'acumulació, entrades, sortides i generació. La seva resolució depèn de factors com la presència de reaccions químiques, si el sistema es troba en estat estacionari o transitori, i si el procés és continu o discontinu.

Tradicionalment, aquests càlculs es realitzen de manera manual o mitjançant fulls de càlcul, la qual cosa pot ser ineficient i tediosa, ja que cada tipus de problema requereix configurar un nou full. Aquest mètode també és propens a errors, especialment en processos complexos amb múltiples corrents, recirculacions o reaccions químiques. D'altra banda, programes més avançats com Aspen, ofereixen capacitats, però sovint requereixen nombrosos paràmetres i propietats físiques que no sempre són conegudes. A més, per a càlculs preliminars, potser no cal entrar en tant de detall.

Per fer front a aquestes limitacions i oferir una solución intermèdia, aquest projecte té com a objectiu desenvolupar una eina computacional basada en Python que automatitzi la resolució dels balanços macroscòpics de matèria en estat estacionari per a processos continus, utilitzant un enfocament de resolució seqüencial. L'eina gestiona tres tipus de blocs de procés: reactors (amb una reacció i termes de generació), separadors (només entrades i sortides), i divisors (utilitzats per a recirculacions i purgues, on s'imposa la igualtat de composició entre els corrents). L'algoritme permetrà dibuixar el diagrama del procés, registrar les dades a una taula, i resoldre iterativament els blocs mitjançant regles de càlcul preprogramades fins a assolir la convergència. El resultat final és una taula completada amb els cabals i composicions resultants.

L'eina ha estat validada mitjançant estudis de cas d'enginyeria química, comparant els resultats obtinguts amb càlculs manuals i avaluant tant l'eficiència com la precisió computacional.

Aquest projecte pretén oferir una solució pràctica i fiable que faciliti els càlculs de balanços de matèria, millorant-ne l'aplicabilitat tant en l'àmbit acadèmic com en l'industrial.

Paraules clau: Balanç macroscòpic de matèria, Python, Resolució seqüencial, Simulació de processos químics, Automatització de càlculs

SUSTAINABLE DEVELOPMENT GOALS

Through this project, the aim is to contribute to the achievement of several Sustainable Development Goals (SDGs) set out in the United Nations 2030 Agenda.

Firstly, the project is related to **SDG 4: Quality Education**, as it provides a potential didactic tool that facilitates learning and understanding of key concepts in chemical engineering. Students and educators can use it to support problem-solving, fostering more efficient, accessible, and interactive education.

It is also linked to **SDG 9: Industry, Innovation and Infrastructure**, by promoting the integration of programming and automation into a traditionally manual and technical discipline. This project encourages the digitalisation of basic engineering tasks and fosters the development of open, adaptable digital solutions that can evolve over time.

Moreover, the use of this tool can extend beyond education. It can also be useful for professionals performing preliminary estimations in industrial contexts where high precision is not yet required. In such cases, the tool provides a lightweight alternative that does not demand prior knowledge of commercial software packages, which are often complex and costly.

Although more indirectly, it can also be connected to **SDG 12: Responsible Consumption and Production.** By digitising exercises that would otherwise be done manually or on paper, it contributes to reducing paper consumption and promoting more sustainable habits in both academic and professional environments.

Finally, by being developed in Python (an open-source programming language) and designed to be freely accessible, the tool contributes to the democratisation of technical knowledge. It empowers a wider range of users to access, understand, and apply chemical engineering principles without barriers related to cost or software licenses.

1. INTRODUCTION

Chemical engineering focuses on the design, analysis, and optimisation of processes in which materials undergo physical or chemical transformations. One of the fundamental tools used to understand and describe these processes is the macroscopic material balance. This principle, based on the law of conservation of mass, establishes that matter can neither be created nor destroyed, it can only be transported, accumulated, or transformed through reactions.

From a macroscopic point of view, a material balance refers to the analysis of a defined system or control volume, tracking how much mass (or a specific component) enters, exits, accumulates, or is generated within that system over time. This approach deliberately avoids microscopic or molecular-level details, focusing instead on the overall behaviour of the system in terms of measurable quantities such as flow rates and concentrations.

Material balances are typically formulated using general balance equations that relate input, output, accumulation, and generation or consumption terms. These concepts allow engineers to evaluate how a process behaves under different operating conditions, and to identify key variables required for proper design, control, and optimisation.

The use of macroscopic balances extends to virtually all areas of chemical engineering and serves as a foundational tool in many tasks, including the design of chemical reactors, the analysis of separation operations, the optimisation of recycle and purge systems, process control and diagnostics, and environmental assessments. In industrial practice, these balances help ensure efficient use of raw materials, identify inefficiencies, and provide essential information for process optimisation and safety assessments. Their role is especially critical in continuous processes, where real-time monitoring and adjustments are based on the outcome of dynamic or steady-state balances.

Beyond their industrial relevance, material balances play a crucial pedagogical role in chemical engineering education. They are typically one of the first engineering tools introduced in undergraduate programs, due to their accessibility and conceptual clarity. Students learn to define system boundaries, identify relevant flows, classify unit operations, and construct consistent mass

balance equations before being introduced to more complex subjects such as thermodynamics, transport phenomena, or reaction engineering. Mastering these balances fosters systematic thinking, a strong intuition for conservation laws, and the ability to visualise the interconnected nature of chemical process units.

In academic settings, the formulation and resolution of mass balances are often done using manual calculations or spreadsheets. While these methods are useful for small systems, they become inefficient and error-prone as the complexity of the process increases—especially when multiple units, recycle loops, or chemical reactions are involved.

Although the formulation of mass balances is rooted in a simple physical principle, their application can be mathematically complex. The difficulty increases with the number of components, units, and process interconnections. Engineers must often solve large systems of nonlinear equations, evaluate stoichiometric constraints, and consider steady vs. unsteady states, batch vs. continuous operations, and the presence or absence of reactions.

This complexity has traditionally been managed with spreadsheets, which allow custom equation input but become tedious and inflexible for large systems, or with commercial simulation software, such as Aspen Plus, which offer powerful capabilities but require extensive training, detailed input data, and access to expensive licenses. Thus, there is a recognised gap between educational tools and professional software that can hinder learning or slow down early-stage project development. This motivates the need for alternative solutions—lightweight, transparent, and accessible tools that allow students and professionals to focus on the logic of the balances themselves, rather than on the software interface or data preparation.

1.1. GENERAL EXPRESSION OF THE MACROSCOPIC MATERIAL BALANCE

Having introduced the conceptual and practical relevance of macroscopic material balances, it is now essential to formalise their general expression. These balances are governed by the principle of mass conservation, which states that matter is neither created nor destroyed, only transformed [Perry y Green, 2008]. This principle forms the basis for developing mathematical equations that allow the quantification of mass flows and transformations in a system over time.

In this section, the general expression of the material balance is presented, followed by its simplifications under different conditions, that are crucial for selecting the appropriate formulation and solving strategy in each specific case.

The generic terms of the balance are related in such a way that:

Accumulation = Input - Output + Generation by chemical reaction

We mainly differentiate two types of equations, which can be expressed in the international system units of mol/s or kg/s:

- **Global Balance:** Applied to the entire system without distinguishing individual components. This global balance, expressed in molar units, is shown in Eq. 1. A more compact version of this global balance is often used, as seen in Eq. 2.

$\sum_{j=1}^{S} \frac{dnj}{dt} = \sum_{j=1}^{S} \sum_{m=1}^{T} wm,j + \sum_{j=1}^{S} Rj$ [1]→ $\frac{dn}{dt} = \sum_{m=1}^{T} wm + \sum_{j=1}^{S} \sum_{i=1}^{R} vijRi$ [2]

- **Component Balance:** Considers the amount of a specific component in a system (Eq. 3 for molar units, Eq. 4 for mass units).

$\frac{dnj}{dt} = \sum_{m=1}^{T} wm,j + Rj = \sum_{m=1}^{T} wm,j + \sum_{i=1}^{R} vijRi$ [mol/s] [3]

$\frac{dnj}{dt} = \sum_{m=1}^{T} wm,j + Rj \cdot Mj$ [kg/s] [4]

Where:

- n: mol or kg that accumulate in the system.
- w: molar (mol/s) or mass (kg/s) flow that enters (positive sign) or exits (negative sign) the system.
- j= components (1...S)
- m= streams (1...T)
- i= chemical reactions (1...R)
- Rj: molar flow (mol/s) of component j generated or consumed by chemical reactions it participates in. It is directly related to the extensive reaction rates in which component j is involved.
- Mj: molecular mass (kg/mol) of component j.
- vij: stoichiometric coefficient of component j in reaction i, with the following sign rule: positive if a product and negative if a reactant.

- Ri: extensive reaction rate of reaction i in which component j is involved.

The general expression of the material balance can be simplified depending on the specific conditions of the system under study. Some terms can be cancelled depending on the process nature and the system state. The following details the different cases:

- **System state:**
    - Steady state: In this case, the accumulation term (A) is zero (A=0), since the amount of matter in the system does not change over time.
    - Unsteady state: Here, the accumulation term is not zero, as the amount of matter in the system varies over time due to the entry and exit of flow or internal reactions.
- **Process type:**
    - Batch process: No material flow in or out during the process, so the input (E) and output (S) terms are zero (E=0, S=0).
    - Continuous process: There are material flows in and out, so the terms E and S are non-zero.
- **Chemical reactions and balance nature:**
    - System without chemical reaction: There is no generation or consumption of matter due to chemical reactions, so the generation term (G) is zero (G=0).
    - Global balance in mass units or balance in molar units in a system where no global variation of moles occurs between reactants and products in the chemical reactions involved (G=0).
    - System with chemical reaction: When chemical reactions occur in the system, the generation term is non-zero, as matter is being formed or consumed. In this case, the material balance must consider the stoichiometry of the involved reactions to determine the amount of each species generated or consumed.

It is very important to note that when chemical reactions are involved in a system, component and global balances should preferably be performed in molar units. If no chemical reactions are involved, the balances can be performed in mass or molar units. Moreover, in non-steady-state reactive systems where the volume changes over time, it is very convenient to

perform a global mass balance in mass units (the generation term is zero), as this will allow explicit knowledge of the volume variation over time.

1.1.1. DEGREE OF PROGRESS OF A CHEMICAL SYSTEM

The extensive conversion of a reaction, X, is defined as the ratio between the difference in the number of moles of a chemical compound after a certain reaction time and its initial number of moles, divided by the stoichiometric coefficient of the compound. The extensive conversion of a reaction is defined by Eq. 5.

$$X = \frac{nj - njo}{vj} \text{ [mol] [5]}$$

It is specific to the reaction and independent of the component considered.

The relation between the moles of a species and its conversion is presented in Eq. 6. It can be easily deduced from the Eq. 5.

$$nj = njo + \sum_{i=1}^{R} vijXi \text{ [6]}$$

where X is the extensive conversion of the i-th reaction. Each term vijXi corresponds to the moles of j converted in the i-th reaction.

On the other hand, the intensive conversion of a reaction is defined as the variation of extensive conversion with the volume. The intensive form of conversion, related to reactor volume, is given in Eq. 7.

$$\xi = \frac{dX}{dV} = (if\ V\ is\ constant) = \frac{cj - cjo}{vj} \text{ [mol/m}^3\text{] [7]}$$

Where V is the reactor volume.

For R reactions, the composition is related to the conversion by Eq. 8.

$$cj = cj0 + \sum_{j=1}^{R} vij\xi i \text{ [8]}$$

Finally, we find the relative conversion with respect to the reactant that is exhausted first (limiting reactant), as it limits the progress of the reaction. This is calculated using Eq.9.

$$Xj = \frac{nj0 - nj}{nj0} = \frac{wj0 - wj}{wj0} = (if\ V\ is\ constant)\frac{cj0 - cj}{cj0} \text{ [dimensionless] [9]}$$

1.1.2. REACTION RATE CONCEPT

The extensive reaction rate, R, is defined as the variation of extensive conversion with time, or also as the variation of the number of mol of a component with time, affected by the inverse of its stoichiometric coefficient. The extensive reaction rate is defined in Eq. 10.

$$R = \frac{dX}{dt} = \frac{1}{vj}\frac{dnj}{dt} \text{ [mol/s] [10]}$$

Similarly, the intensive reaction rate, r, is defined as the variation of the extensive reaction rate with volume and also as the variation of extensive conversion with time, inversely affected by the volume. The intensive rate, dependent on reactor volume, is described in Eq. 11.

$$r = \frac{dR}{dV} = \frac{1}{V}\frac{dX}{dt} = \frac{1}{V}\frac{1}{vj}\frac{dnj}{dt} = (if\ V\ is\ constant)\frac{d\xi}{dt} = \frac{1}{vj}\frac{dcj}{dt} \text{ [mol/s·m}^3\text{] [11]}$$

1.1.3. CONCEPTS OF FLOW DIAGRAM, BYPASS, RECIRCULATION, AND PURGE

The flow diagram of a chemical process (system) is the graphical representation of the unit operations (subsystems) involved, where material flows between them are indicated by arrows. It is an important document, and its presentation should be clear, comprehensive, precise, and complete. There are different types of flow diagrams: block diagram, pictorial diagram, and symbolic diagram, drawn according to a standard industrial drawing regulation.

A bypass stream is a small portion of the inlet stream to a system (composed of subsystems) that is added or directly diverted to the system's output stream in order to provide a series of components or qualities that have been lost during the physical-chemical treatment performed in the process. This means that the bypass and fresh streams have the same composition, but their flow rates differ substantially, so the bypass stream is usually a small fraction of the fresh stream in the system.

In many chemical processes that aim to reuse unreacted components and also in chemical systems that reach thermodynamic equilibrium, recirculation streams are used, which are reintroduced into the reaction system to improve conversion in that reactor. In many cases, it is necessary to enable, along with the recirculation stream, a purge stream that prevents the accumulation of inert or by-product components at the system's exit. The composition of the recirculation and purge streams are identical, but the recirculation flow rate is much higher than the purge flow rate.

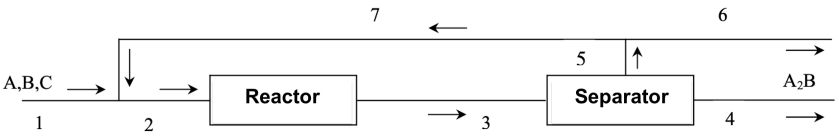


Figure 1. Process flow diagram with reactor, separator, recycle, and purge stream

1.2. STRATEGY FOR SOLVING BALANCES IN PFD: DEGREES OF FREEDOM

All process flow problems share one aspect: they begin with known variables or data to calculate other unknown variables or unknowns. This calculation requires reducing the problem to a set of equations based on macroscopic material balances.

Once all components of the problem are recognized and all their mathematical equations are established, it is very useful to organize all the available information in a PFD, which will show the distribution of the provided information. At this point, you can calculate the number of independent equations (Ne), the number of unknowns (Ni) and three cases can be distinguished:

- Ne>Ni →Multiple solutions
- Ne<Ni → Parametric solution
- Ne=Ni →Unique solution

This allows us to verify whether the problem provides sufficient information for its solution before setting up any equations. This procedure is known as the degrees of freedom (DoF) analysis. While it does not guarantee that the problem is fully well-posed, it helps identify many cases where a suitable solution is not possible. The difference between the number of unknowns and equations defines the degrees of freedom. The degrees of freedom of a process can be calculated using Eq. 12.

$$DoF = Ni - Ne \quad [12]$$

Known the DoF of a problem, the three cases can be established:

- DoF<0 [Over-specified problem] ⟶ Information must be removed
- DoF>0 [Under-specified problem] ⟶ Not all variables can be known
- DoF=0 [Properly specified problem] ⟶ Material balance equations can be set up.

The degrees of freedom can be calculated by considering:

- Ni (number of unknowns): total or partial flow rates, mole fractions, reaction rates, etc.
- Ne (number of equations): mass balances (per component and total), equilibrium relations, process specifications, and composition equality conditions between streams.

This type of analysis helps determine whether sufficient information is available before setting up formal equations and allows the establishment of a structured resolution strategy, starting with units that have DoF = 0.

Problems with a single degree of freedom typically involve selecting a calculation basis, usually defined by the magnitude of an external stream, to which all calculations are referenced.

In general, to successfully solve material balance problems, it is advisable to start studying the system from the outside in, i.e., initially performing balances for the entire experimental device and then for each unit or operation of the process (in one of them, the balances will no longer be mathematically independent and will not need to be performed).

After analyzing the degrees of freedom of the system and the subsystems involved, this may sometimes require relocating the calculation base to facilitate the mathematical solution of the problem.

The process is correctly specified if DoF = 0, in which case the solution is unique. The first set of equations to be solved corresponds to the unit with DoF = 0, and the system will be resolved sequentially until all unknowns are determined. If no unit has DoF = 0, the global balance equations should be solved first. However, if the global DoF is not zero, it is preferable to avoid using it and instead relocate the calculation basis to a unit with DoF = 1, so that the unit then has DoF = 0 and can be solved.

In this project, an explicit calculation of the degrees of freedom for each process unit has not been implemented. However, the logic used in the tool is designed in a way that implicitly follows the principles of DoF analysis, through a sequential solving strategy:

- Reactors are solved first, thanks to the use of temporary variables, which allow calculating the output streams based on inputs, conversion, and reaction rate. This ensures that the reactor can be solved as if it had DoF = 0.
- Next, with the reactor solutions, separators are solved, which involve only material balances without reactions.
- Finally, splitters are addressed, which are special units where composition equality between streams is imposed, introducing additional constraints.

This order of calculation ensures that each step has sufficient information to proceed. The system uses iterations to adjust the variables that depend on interconnected units, especially in the presence of recirculation loops.

Therefore, although a formal DoF analysis is not performed, the developed method applies its principles in a structured and controlled manner, ensuring that the system can be successfully solved through a sequential approach.

1.3. SEQUENTIAL OR SIMULTANEOUS METHOD

In the analysis and solution of material balances in chemical processes, two main approaches can be used: the sequential method and the simultaneous method.

Most commonly used industrial software, such as Aspen Plus, by default implements the modular sequential method, where each process unit is solved individually, using the values calculated in previous units and propagating the results to the next ones [Adams, 2022]. This approach is particularly efficient in systems where the units can be solved in a logical order, without circular dependencies or complex recycling.

In contrast, the simultaneous method sets up all the equations of the system together and solves them in block form using numerical algorithms, which is useful in processes with strong interdependencies, multiple recycling, or nonlinear constraints. However, this method requires more computational effort and mathematical formulation, so it is typically used in optimization problems and advanced design.

The choice of solution method is also linked to the degree of freedom analysis, as a system with the correct number of equations and unknowns can be solved sequentially, while an under-specified system (with more unknowns than equations) or an over-specified system (with redundant equations) may require additional adjustments.

Since the sequential method allows for modular calculations, simplifies the interpretation of results, and is more intuitive in its implementation, it is the preferred option for most industrial processes that operate in a steady-state and without a lot of circular interactions.

2. OBJECTIVES

The general objective is developing a computational tool in Python for the automated resolution of material balances in chemical processes, allowing the structured input of data, visualization of the process flow diagram, and obtaining results using the sequential method.

Specific Objectives

- Design an interface to input data via a table, specifying the inlet and outlet streams, as well as the compositions and flow rates involved.
- Implement a module for the graphical visualization of the flow diagram, representing the connections between process units clearly and intuitively.
- Apply the modular sequential method for solving material balances, ensuring a structured and efficient calculation of unknown variables.
- Integrate a function that completes the initial table with the obtained results, providing a structured report of the material balances to facilitate interpretation and analysis.
- Evaluate the tool's accuracy and efficiency by solving case studies representative of industrial processes.

This project seeks to optimize the analysis and resolution of material balances in chemical systems, providing a flexible and accessible tool that facilitates its application in both academic and industrial settings.

3. METHODOLOGY

To carry out this project, various tools and resources have been used that combine programming, documentation, and academic materials from the field of chemical engineering. The methodology followed has focused on applying theoretical knowledge from introductory chemical engineering courses and translating it into a computational framework using accessible programming environments.

The development of the tool has been carried out entirely using the Python programming language, an open-source language widely used in scientific and engineering fields [Python.org, 2025]. Python was chosen for its simplicity, readability, and the availability of numerous scientific libraries that facilitate numerical computation and data handling. Its use also aligns with the educational purpose of the project, as it is freely available and widely adopted in academic environments.

To write and manage the source code, the Visual Studio Code (VS Code) editor was used. VS Code is a lightweight but powerful code editor developed by Microsoft that supports Python natively and offers useful extensions such as syntax highlighting, integrated terminal, debugging tools, and Git integration. Its modular structure and flexibility have allowed for efficient management of different parts of the project in an organised and scalable way.

In addition to the technical tools, the theoretical foundation of the project is based on materials from the “Introducció a l’Enginyeria Química” course, where key concepts such as the formulation of material balances, process flow diagrams, recycle and purge systems, and degrees of freedom were studied. Several solved exercises and problems from that course have been used as reference cases to test the functionality and accuracy of the developed tool.

The bibliographic references consulted include introductory chemical engineering textbooks that have provided complementary theoretical context.

During the development phase, several iterations were carried out to define the essential process blocks (reactors, separators, and splitters) and to translate the solving logic into a step-

by-step resolution algorithm. The design of the interface, as well as the internal data structure, was guided by a combination of academic problem-solving strategies and best practices in software development.

The chosen methodology ensures a balance between technical rigour and educational accessibility, allowing the resulting tool to be useful both as a learning aid and as a preliminary process analysis tool.

4. PROGRAM DEVELOPMENT

In this part of the project is shown how the software works. Not all parts of the software are shown, there are only explained the basics of the code in order to understand how it operates. The full code is the Appendix 1.

4.1. GENERAL ARCHITECTURE OF THE PROGRAM

The tool has been entirely developed in Python, featuring a graphical user interface built with the Tkinter library. The project is structured into independent modules that interact with each other to enable data input, flow diagram visualization, and the iterative and automated resolution of material balances.

The four main modules are:

- **main.py:** Serves as the entry point of the application. It initializes the main graphical window and loads both the data table and the flow diagram. It is the primary script that invokes all other modules.
- **data_table.py:** Manages the creation and editing of the compound and stream table, both in terms of flow rates and mole fractions. It includes functions for detecting missing values, updating totals, and maintaining synchronization between the two tables. A dedicated button is also provided for manually adding compounds as the system is being built.

- **interface.py:** Responsible for the graphical construction of the process flow diagram (PFD). It allows the user to add blocks, sources, and connections, delete them, initiate the balance resolution, and synchronize all information with the data table. For reactors, it prompts the user to input a chemical reaction; for splitters, it requests the recycle ratio. When a new connection is created, the system automatically asks for the name of the stream via a pop-up window in order to register it in the table. This module also manages the calculation order between blocks.
- **calculations.py:** Implements the complete logic required to solve the material balances. It establishes a sequential resolution order among the blocks, identifies all necessary parameters extracted from the diagram or the table, solves each block based on its specific equations, handles both temporary and dynamic variables, and applies an iterative method until convergence is achieved.

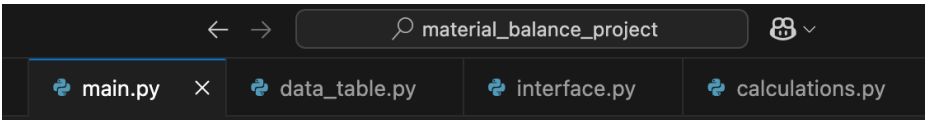


Figure 2. VS Code environment showing the four main Python modules.

All modules are fully synchronized and interdependent, requiring mutual data exchange to perform their respective tasks. The overall workflow consists of first building the flow diagram, then entering all known data into the table, and finally executing the resolution, after which the previously unknown values (initially shown as “-”) are automatically filled in.

The program includes two types of messages to aid in user interaction and debugging:

- **Visual messages** (via messagebox.showinfo) displayed in the graphical interface to inform the user of relevant events (e.g., successful resolution, missing chemical reaction, or the need to click a target block to create a connection).

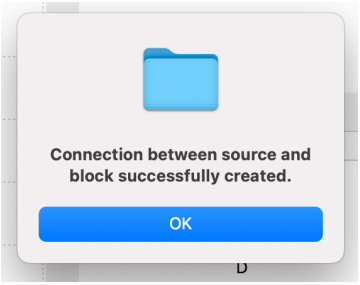


Figure 3. Visual messages shown in the graphical interface during simulation.

- **Internal messages** (via print("...")) output to the terminal, providing detailed feedback on the resolution process, including generated equations, results, iteration logs, and all intermediate steps. These are particularly helpful for identifying errors or for understanding the origin of the final table values.

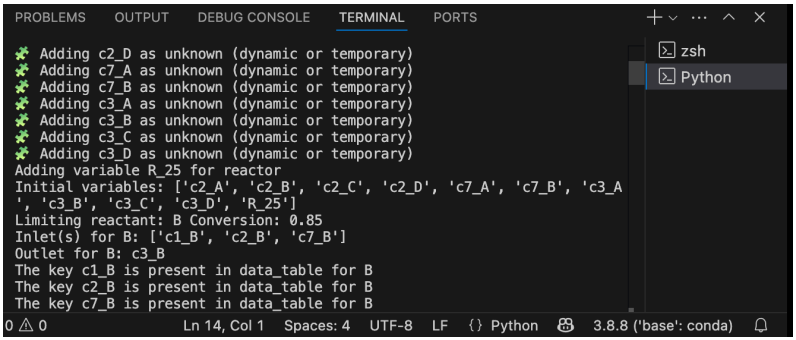


Figure 4. Internal messages displayed in the terminal for debugging and process tracking.

4.2. GRAPHICAL INTERFACE AND USER STRUCTURE

The application features a graphical user interface divided into two main sections:

Flow Diagram (left panel):

This section allows the user to visually construct the process using predefined blocks placed on a grid. The available block types are:

- **Reactors:** Include a user-defined chemical reaction with specified stoichiometric coefficients.

- **Separators:** Perform the separation of compounds while maintaining mass balance between input and output, assuming no generation.
- **Splitters:** Handle recirculation and purge streams by enforcing composition equality across the outlet streams. The recycle ratio R is requested from the user upon block creation.
- **Sources:** Represent system boundaries, such as external feed streams or final products.

Once created, blocks can be connected via process streams. The user is prompted to assign a name to each stream through a pop-up dialog. These streams are then automatically registered in the data table.

Data Table (right panel):

This area includes two distinct tables:

- **Main Table:** Displays the molar flow rates (in kmol/h) of each compound per stream, including the total flow and the conversion factor (a value between 0 and 1) for each compound, when applicable.
- **Mole Fraction Table:** Displays the mole fractions of each compound per stream, as well as the total mole fraction per stream.

Additionally, the interface includes several control buttons that allow the user to:

- Add new blocks to the diagram.
- Add new compounds to the data table.
- Remove existing connections between blocks.
- Launch the automated resolution of the system.

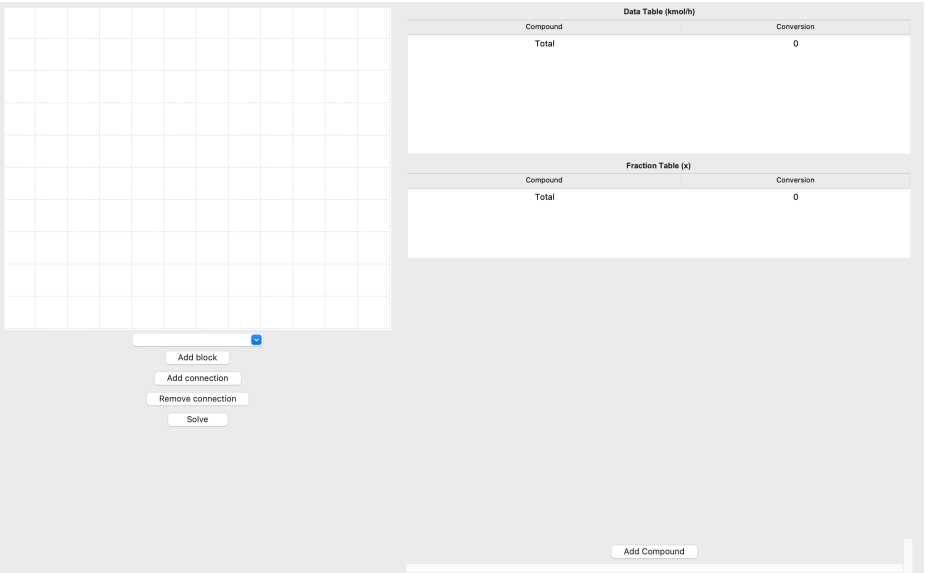


Figure 5. Initial view of the complete graphical interface when launching the program.

4.3. DATA MANAGEMENT (DATA_TABLE.PY)

The data_table.py module contains the DataTable class, responsible for managing the visualization and editing of system data, including flow rates and mole fractions. This component plays a key role, acting as a bridge between the graphical representation of the process and the resolution of the material balances.

This script creates two interactive tables embedded within the graphical user interface: one for flow rates and another for mole fractions. Additionally, it incorporates logic for synchronization between both tables, validation mechanisms, and automated calculations. It is designed to facilitate both user data input and computational handling.

Main Flow Rate Table (tree)

This table represents the molar flow rate (in kmol/h) of each compound in each process stream. Each row corresponds to a compound, and each column to a process stream or associated conversion. The main functionalities include:

- **Add compounds:** A dedicated button allows the user to add new compounds. This automatically creates a new row in both tables, initialized with "-" values.

- **Add streams:** Whenever a new connection is created between process blocks, the corresponding stream is automatically added as a new column.
- **Manual editing:** A double-click on any cell allows the user to enter a value, which is stored as a float if valid.
- **Total row:** Automatically updated to display the sum of flow rates for each stream.

Mole Fraction Table (fraction_tree)

This parallel table displays the mole fraction of each compound in each stream. It is essential for calculations involving known total flows and partial compositions, as well as for validating data consistency:

- **Structural synchronization**: Whenever a new compound or stream is added, both tables are updated to maintain structural consistency.
- **Total calculation**: Includes a special row labeled "Total_x" that displays the sum of mole fractions for each stream, serving as a visual validation tool.
- **Manual entry detection**: Users can manually input mole fractions, which are flagged as “manual” and prioritized in subsequent calculations.

Both tables occupy a fixed area within the graphical interface. As new columns (streams) are added, the width of existing columns is dynamically adjusted to maintain a compact and coherent layout.

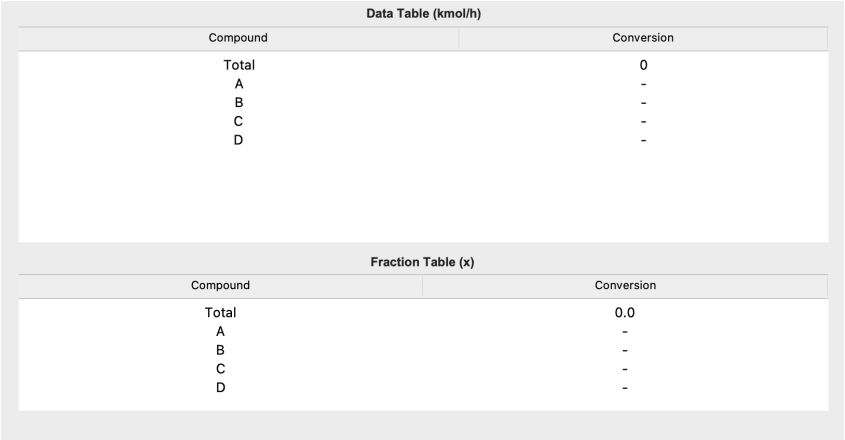
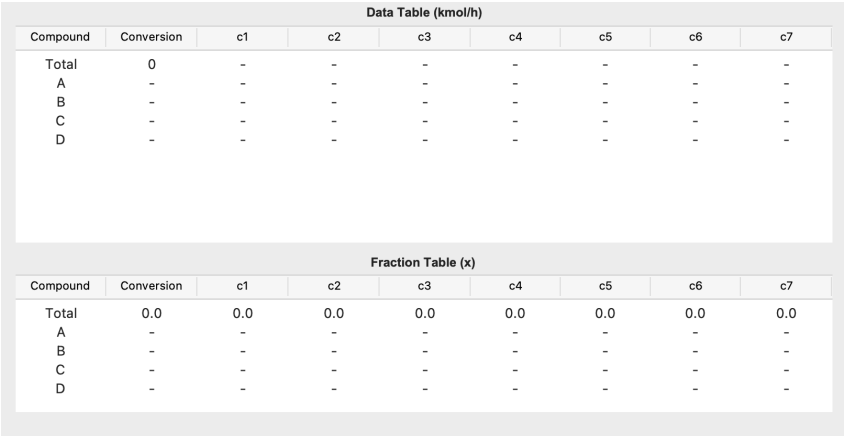


Figure 6. Empty data tables at program start, before adding any streams. The table occupies a fixed display space.



Data Table (kmol/h)

Compound	Conversion	c1	c2	c3	c4	c5	c6	c7
Total	0	-	-	-	-	-	-	-
A	-	-	-	-	-	-	-	-
B	-	-	-	-	-	-	-	-
C	-	-	-	-	-	-	-	-
D	-	-	-	-	-	-	-	-

Fraction Table (x)

Compound	Conversion	c1	c2	c3	c4	c5	c6	c7
Total	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
A	-	-	-	-	-	-	-	-
B	-	-	-	-	-	-	-	-
C	-	-	-	-	-	-	-	-
D	-	-	-	-	-	-	-	-

Figure 7. Data tables after adding seven streams. Column widths are redistributed within the fixed table space.

The DataTable class includes several methods to ensure data integrity and support balance resolution:

- **add_compound:** Adds a new row to both tables and initializes its values.
- **add_stream:** Adds a new column, resizes existing ones, and synchronizes both tables.
- **on_double_click:** Enables manual editing of cells in the main table and stores the input as float values.
- **update_totals:** Recalculates and updates the “Total” row of flow rates.
- **get_missing_values:** Returns a list of cells containing unknown values ("-"), useful to determine whether the system requires solving.
- **update_table_with_resolved_values:** After resolution, populates the tables with the calculated results.
- **update_fraction_totals:** Recalculates mole fraction totals and updates the "Total_x" row.
- **sync_fraction_table_structure:** Ensures both tables maintain a consistent structure of compounds and streams.

4.4. BLOCK CREATION AND CONNECTION (INTERFACE.PY)

The interface.py module contains the FlowDiagram class, which is responsible for the graphical construction of the process flow diagram and its interaction with the user. This component serves as the central axis between the graphical definition of the system and the automated resolution of material balances.

The diagram is rendered on a canvas organized into a 12-column by 10-row grid, where different types of process blocks can be placed visually. This grid provides a clear and structured layout, enabling an intuitive and orderly representation of the system.

Available Block Types:

- **Reactor:** When added, the user is prompted to define a chemical reaction (e.g., A + B → C). The system parses the involved compounds, automatically adds them to the data table, and stores the stoichiometric coefficients for use in subsequent calculations.
- **Splitter:** Allows the definition of a recycle stream through a dialog window in which the user enters the recycle ratio R=Frecycle/Fpurge
- **Separator:** Represents a physical separation without chemical reaction, assuming a standard input–output material balance.
- **Source:** Represents external inputs or open-ended outputs of material. These blocks can be used to define feed streams or final product outlets.

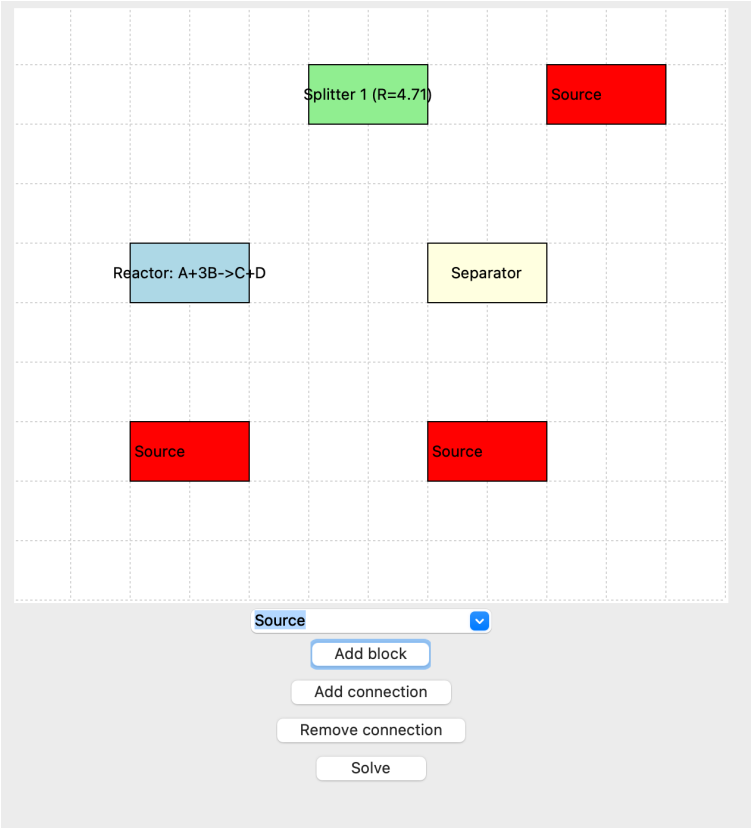


Figure 8. Insertion of process blocks into the flow diagram using the graphical interface.

Connections:

The user can connect blocks using process streams. During this process:

- The system prompts the user to first select the source block, then the destination block, and finally to assign a name to the stream.
- An arrow is drawn on the canvas to visually represent the connection, and it is automatically labeled with the stream name.
- The newly created stream is automatically added as a new column in the data table to ensure data synchronization.

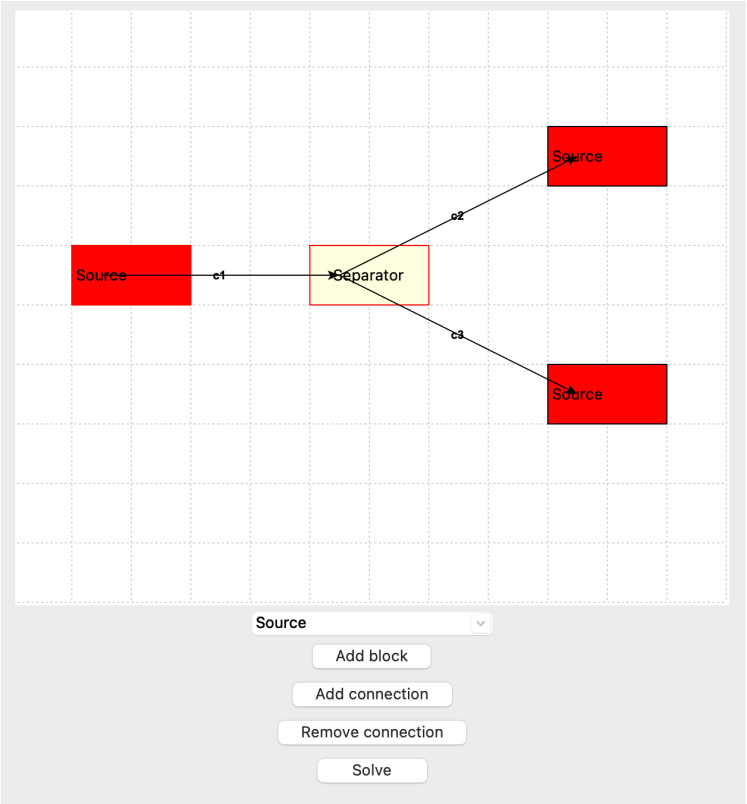


Figure 9. Process flow diagram with connections added between blocks.

Key functionalities:

- **Add blocks and sources:** Through a dropdown menu and a click on the grid, the user can place blocks visually.
- **Input reactions and recycle ratios:** Automatically prompted dialogs allow the user to define chemical reactions for reactors or the recycle ratio for splitters when the block is created.
- **Delete blocks with right-click:** Blocks and sources can be removed both visually and from the system logic.
- **Manage connections:** Dedicated functions allow users to create or delete connections between blocks.

- **"Solve" button:** Initiates the iterative resolution of material balances and displays results via graphical pop-ups and console output to assist in debugging.

All blocks and connections are tightly integrated with the data table (DataTable), ensuring consistency between the graphical model and the numerical data. For example:

- Streams created in the diagram automatically appear as new columns in the tables.
- Compounds involved in a reaction are automatically generated and added.
- The calculation order is determined by the hierarchical structure of the connections.

This modular and visual architecture makes the application intuitive for the user and easily scalable for future developments (e.g., new block types, improved algorithms, etc.).

Main functions of the FlowDiagram class:

- **resize_grid**: Draws the base grid on the canvas, visually defining the cells where blocks can be placed.
- **add_block + select_cell_for_block + place_block_on_grid**: Activate the block placement mode. Once the user selects a cell, a block of the selected type is positioned on the canvas.
- **create_reactor(reaction):** Visually creates a reactor, displayed as a blue rectangle labeled "Reactor: {user-defined reaction}." The block is registered and parsed to extract the involved compounds.
- **update_data_table_for_reaction(reaction):** Automatically adds the identified reactants and products to the data table if they are not already present. This ensures that the chemical information is fully synchronized.
- **parse_reaction(reaction):** Interprets the chemical expression provided by the user (e.g., A + B -> C + D), extracting the compound names and their corresponding stoichiometric coefficients for both reactants and products.
- **create_splitter(col, row):** Visually creates a splitter, displayed as a green rectangle with the label "Splitter: R={recycle ratio}" entered by the user. This value is stored internally for later use during calculations.
- **create_separator(col, row):** Creates a separator, represented as a yellow rectangle labeled "Separator," to model physical separation processes without chemical reaction.
- **create_source(col, row):** Adds an external source, displayed as a red block. It can represent either an open inlet or outlet of the system.

- **start_connection + select_block_for_connection + create_arrow:** Initiate the connection mode between blocks. The user selects a source block and a destination block; an arrow is drawn on the canvas, and the stream name is requested via a popup dialog. This stream is automatically registered in the data table.
- **remove_connection:** Deletes the last connection created, removing both the arrow and its label from the canvas.
- **delete_block_on_right_click:** Enables the deletion of blocks or sources from the diagram via right-click. This enhances flexibility during system construction.
- **resolve_balances + debug_resolver:** Check for unknown values ("-") in the data table. If any are found, the iterative balance resolution method is executed. The system provides feedback through graphical messages indicating whether the system has been solved or if no unknown values remain to calculate.

4.5. MATERIAL BALANCE RESOLUTION (CALCULATIONS.PY)

The calculations.py module contains the MaterialBalanceSolver class, responsible for executing material balances across the entire system of blocks and streams defined by the flow diagram and the data table. This class represents the core of the iterative resolution algorithm.

The class operates using two main sources of information: data_table and flow_diagram.

The balances are applied iteratively, block by block, according to the specific equations associated with each type of unit (reactor, separator, splitter). The system attempts to compute unknown values from the available data, reconciling flow rates and mole fractions.

To carry out the iterative resolution and manage uncertainty due to incomplete data, the system uses two types of variables: temporary and dynamic. While both originate from unknown values in the data table, their roles and behavior during the iterative cycle are distinct and complementary.

Temporary variables are an internal tool used by the system to close mass balances and generate a provisional solution when there is insufficient information to do so directly. These variables appear primarily during the resolution of reactors, which are the first blocks to be solved. When missing values are detected in the conversion equation for the limiting reactant, temporary variables are automatically generated within the solve_reactor_balance function and assigned an

arbitrary initial value (typically 5.0). This provisional value allows the system to proceed with calculations in subsequent blocks, and through iterative refinement, the temporary solution is expected to converge. These variables are only valid within a single iteration. Their initial and final values are stored in initial_temporary_results and final_temporary_results, respectively, which allows the system to compute the variation between iterations and determine whether convergence has been achieved.

Dynamic variables, on the other hand, represent actual unknowns within the system. For instance, if the flow rate of a given compound in a particular stream is not known, this quantity is treated as a dynamic variable. At the start of the resolution, the reset_dynamic_variables function scans the data table to detect all missing values (represented by "-"). These values are then filtered to exclude any already defined as temporary variables, and the resulting list is stored in self.dynamic_variables. These variables remain fixed throughout a single iteration: once a value is calculated (e.g., in a reactor), it is used as input for the resolution of other blocks, such as separators or splitters, without being altered within the same iteration. Therefore, unlike temporary variables, dynamic variables form part of the system's structural unknowns and are resolved numerically through systems of equations. At the beginning of each new iteration, dynamic variables are reset and recalculated, maintaining algorithmic consistency and robustness.

Temporary variables are not reset and are the only ones used to compare values between iterations, which is critical for evaluating convergence.

Additionally, the class initializes three tracking lists:

- initial_temporary_results and final_temporary_results: used to compare the values of temporary variables across iterations.
- initial_mole_fractions: stores user-defined mole fractions to ensure that they are enforced as constraints during the resolution.

Resolution Process (solve_balances)

This method constitutes the core of the system and performs the iterative resolution:

1. Starts with iteration = 0, incrementing up to a maximum of 30 iterations.
2. Saves all manually defined mole fractions.
3. Detects missing values (unknowns).

4. Stores the initial values of temporary variables (from the previous iteration).
5. Resets the values of dynamic variables.
6. Executes the resolution block by block:
    - solve_reactor_balance
    - solve_separator_balance
    - solve_splitter_balance
7. Stores the final values of temporary variables.
8. Updates mole fractions based on the new flow values.
9. Verifies convergence by comparing changes in temporary variables across iterations (convergence if all changes < 1e-4).

Block Resolution Methods

solve_reactor_balance

1. Identifies the inlet and outlet streams.
2. Interprets the chemical reaction:
   - Reactants and products are parsed, along with their stoichiometric coefficients (positive for products, negative for reactants).
3. Detects the limiting reactant:
   - If a conversion value is defined for any compound, it is treated as the limiting reactant.
4. Applies the conversión equation:
   - If all inlet flows for the limiting reactant are known:
     $Foutlet = Finlet \cdot (1 - X)$ [13]
   - If not, unknown inlets are defined as temporary variables and the equation:
     Outlet_rl=sum(inlets)*(1-conversion) [14] is applied
5. Applies the general material balance for each compound:
   $\sum Finlets - \sum Foutlets = v \cdot R$ [15]
   Where v is the stoichiometric coefficient and R the global reaction rate.
6. Solves the system of equations using numpy.linalg.lstsq.
   This function is part of the NumPy library and is designed to solve linear systems of the form:
   $Ax = b$

Where:

A is the matrix of coefficients obtained from the balance equations

x is the vector of unknowns (flow rates)

b is the vector of known terms

The function computes the least-squares solution, which minimizes the Euclidean norm

$||Ax-b||_2$. This is particularly useful in chemical engineering problems, where systems may be overdetermined (more equations tan unknowns), undetermined, or affected by measurement errors. Using lstsq ensures numerical stability and provides a practical solution even when an exact one does not exist, making it well-suited for real-world engineering applications.

7. Stores the results.

This process is repeated in each iteration until the values of temporary variables converge.

solve_separator_balance

1. Identifies inlet and outlet streams
2. For each compound, constructs a mass conservation equation (Eq. 16):

   $\sum F_{inlet} = \sum F_{outlet}$ [16]

3. If manual mole fractions are provided:
   a. If only one is known, the other is deduced so that the total equals 1.
   b. If two are known, the following relationship is imposed to mantain compositional consistency (Eq. 17):

      $x1 \cdot F2 = x2 \cdot F1$ [17]

4. Solves the system of equations using numpy.linalg.lstsq.
5. Stores the results.

solve_splitter_balance

The splitter block divides a stream into two (typically purge and recycle) with a defined recycle ratio (Eq. 18.) $R = F_{recycle}/F_{purge}$ [18].

1. Identifies inlet and outlet streams, determining which represents recycle and which purge.
2. Retrieves the user-defined recycle ratio R.
3. Computes base mole fractions (composition must be identical in all outlet streams):

a. If manual fractions are available, they are normalized.
b. Otherwise, mole fractions are computed from inlet flows.

4. Applies a material balance for each compound:

- Conservation equations are created similarly to other blocks.

5. Imposes composition equality between outlet streams (Eq. 19):

$x_i \cdot F_j = x_j \cdot F_i$ [19]

6. Enforces the recycle ratio with Eq. 20.

$F_{recycle} = R \cdot F_{purge}$ [20]

7. Solves the system of equations using numpy.linalg.lstsq.
8. Stores the results.

Auxiliary Functions

- **get_streams_for_block:** identifies the inlet and outlet streams for a given block.
- **get_limiting_reactant:** identifies the compound with a defined conversion, used as the limiting reactant.
- **get_reaction_coefficients:** parses and stores the stoichiometric coefficients of the chemical reaction.
- **update_fractions_from_flows:** computes mole fractions from flow rates.
- **update_flows_from_fractions:** computes flow rates from mole fractions and total flow.
- **reconcile_flows_and_fractions:** synchronizes mole fractions and flow rates based on available information:
    - If total flow and some mole fractions are known →

      $F_i = x_i \cdot F_{total}$ [21]
    - If two or more partial flows are known →

      $F_{total} = \sum F_i$ [22]

      $x_i = F_i / F_{total}$ [23]
    - If only one manual fraction exists (binary assumption) →

      $x_{other} = 1 - x_{manual}$ [24]

  Additionally, it updates the “Total” and “Total_x” rows in the respective tables.

- **enforce_recycle_ratio:** identifies recycle and purge streams and imposes the equation (Eq. 20).

This class is a critical and sophisticated component of the system, incorporating rigorous variable management and applying numerical methods to solve linear systems of equations. It is designed to operate with incomplete datasets and to compute a coherent and consistent global solution.

5. PROGRAM VALIDATION

To verify the reliability and usefulness of the developed tool, it was applied to several typical case studies in chemical engineering. The results obtained were compared with step-by-step manual calculations, validating both the numerical accuracy and the consistency of mole fractions and total balances.

The key aspects validated were:

- **Convergence**: the system reaches a stable solution in fewer than 30 iterations in all cases.
- **Balance consistency**: the sum of inputs and outputs for each compound matches within the predefined error margin.

Although the current tool is designed for continuous steady-state processes with a single reaction per reactor block, its modular architecture allows for future expansion, such as:

- Incorporation of simultaneous reactions and the calculation of selectivity or yield.
- Addition of energy balances.
- Further functionalities as needed.

5.1. CASE STUDIES USED

To ensure the correct functionality of the tool, several representative examples were created using realistic configurations commonly found in chemical engineering courses or projects. The case studies include:

- Case A: simple process with one reactor and one separator.
- Case B: process with a reactor and a separator, featuring total recycle of unreacted compounds.
- Case C: system with a reactor, separator, and splitter, simulating both recycle and purge of by-products.

In all cases, the system successfully generated a completed data table with flow rates per compound and stream, and mole fractions consistent with the input information. Example results are shown below.

CASE A

This case illustrates a simple process involving a chemical reaction and a separation step. Figure X shows the process flow diagram, constructed using the graphical interface of the tool. The system consists of two source blocks introducing compounds A and B into the process; a reactor, where a chemical reaction takes place that consumes A and B to produce C and D; a separator, which splits the reactor's output stream into two separate streams based on a defined separation logic; and finally, two sink blocks, representing the final output destinations of the system.

To the right of the figure X, the data table is automatically completed after solving the system. In this case, no iterations were needed, as the sequential solving logic allows the reactor to be resolved first. Since all necessary inputs are available, both inlet streams (c1 for B and c2 for A) and the conversion of the limiting reactant (0.7), the reactor can be solved directly. Once the output stream (c3) is calculated, it serves as the input for the separator.

In the separator, a component-based separation rule is applied: components A and B are directed to stream c5, while C and D are directed to stream c4.

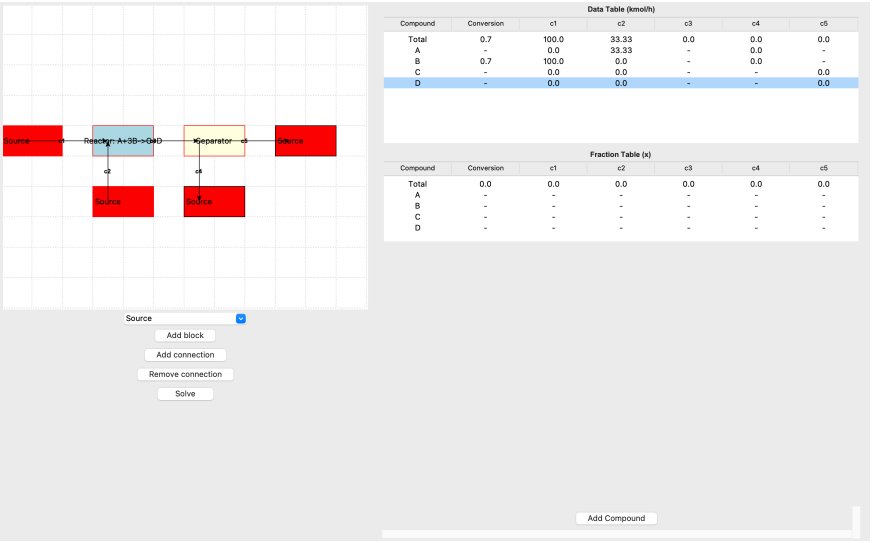


Figure 10. Case A – Initial configuration with the process flow diagram created and input data filled in the table.

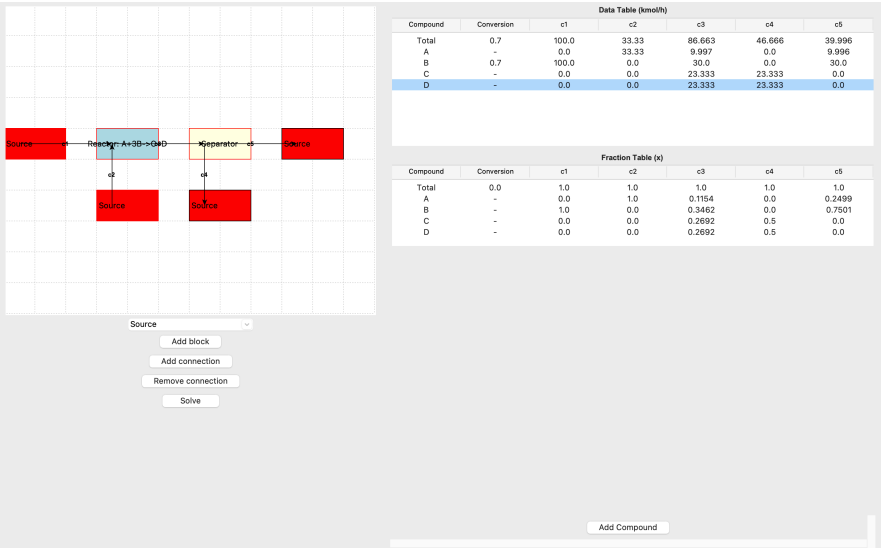


Figure 11. Case A – Results after pressing the "Solve" button, showing the completed table with calculated values.

CASE B

This second case represents a process involving a chemical reaction, separation, and **total recycle of unreacted compounds.** Figure X shows the flow diagram, composed of two source blocks introducing compounds A and B; a reactor, where A and B are partially converted into C and D; a separator, and a sink block for the final output.

In this example, compounds A and B enter the reactor through three streams: c1 and c2 (fresh feed) and c5 (total recycle stream). The products C and D generated in the reactor are separated in the separator block, which distributes the output flow: C and D exit through stream c4, while unreacted A and B are recycled via stream c5, thus closing the recycle loop.

The conversion of the limiting reactant has been set to 0.7 (70%). However, unlike the previous case, not all reactor inputs were defined initially (specifically, c5), so the system required iterations to converge. The tool applied a sequential iterative solving procedure until consistency was reached between the recycled streams and the conditions imposed by the material balance. Figure X shows the completed table with the resulting flows and compositions.

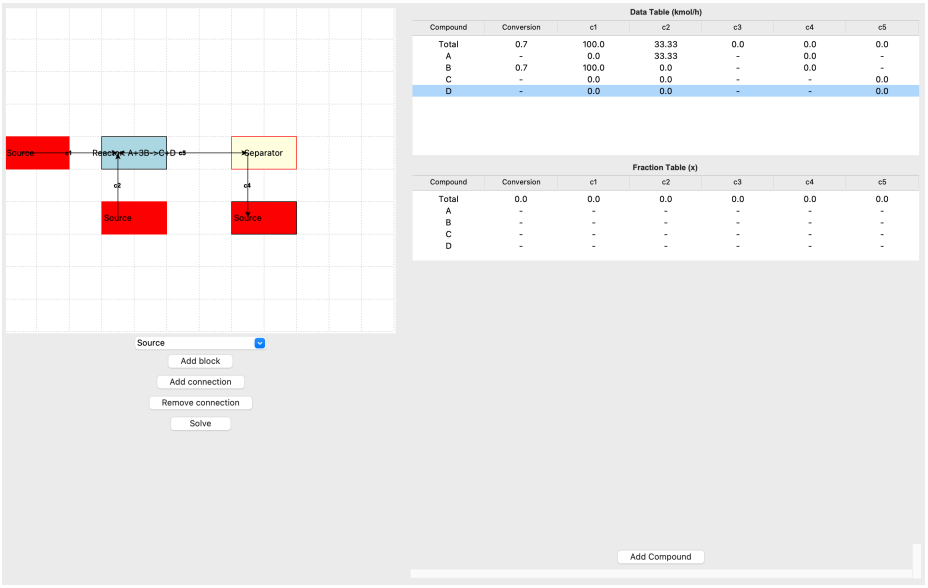


Figure 12. Case B – Initial configuration with the process flow diagram created and input data filled in the table.

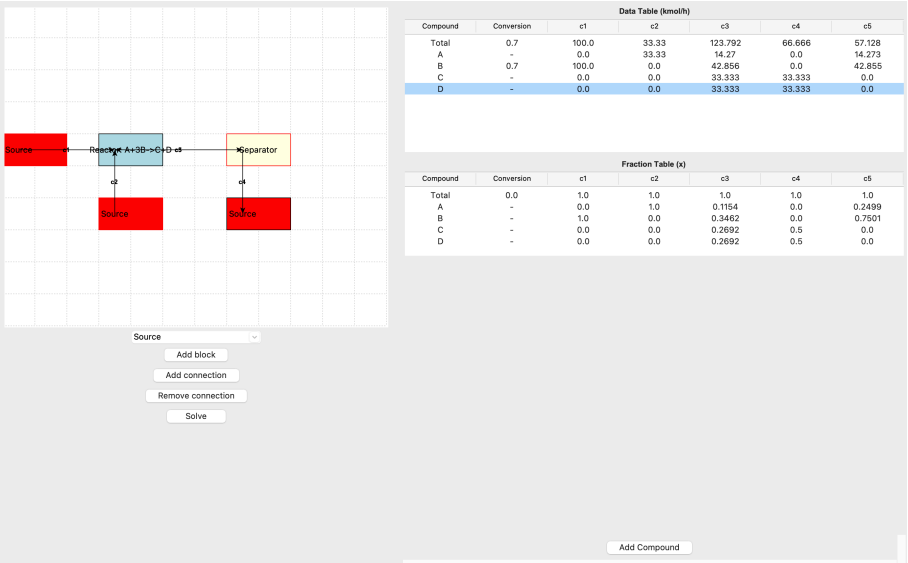


Figure 13. Case B – Results after pressing the "Solve" button, showing the completed table with calculated values.

CASE C

This third case represents a complete system with a reactor, separator, and splitter, allowing the simultaneous simulation of recycle of unreacted reactants and **purge of by-products**. This is a common configuration in industrial processes where the aim is to maximise the conversion of the limiting reactant without allowing the accumulation of inert or undesired compounds in the system.

The process begins with a fresh feed stream introducing compounds A (c2) and B (c1). Unlike the previous two cases, where A and B were fed in stoichiometric proportions, here A is provided in 10% excess. These components enter a reactor, where they are partially converted into the desired product C and a by-product D. The reactor outlet flows into a separator, which separates the unreacted reactants (c5) from the products formed (c4).

The stream containing unreacted reactants (c5) is then directed to a splitter, which divides the flow into two parts: one is recycled back to the reactor (c7), and the other is discharged as a purge stream (c6). This strategy prevents the progressive accumulation of by-product D in the system.

The conversion of the limiting reactant has been set to 0.85 (85%). Since the recycle stream depends on the splitter distribution and directly affects the reactor's input, the system requires iterations to reach convergence. The tool solves this loop using its sequential solving algorithm, adjusting values until both the global and component-wise mass balances are satisfied.

Figure X contains the initial data and Figure X shows the completed table with the resulting flows and compositions.

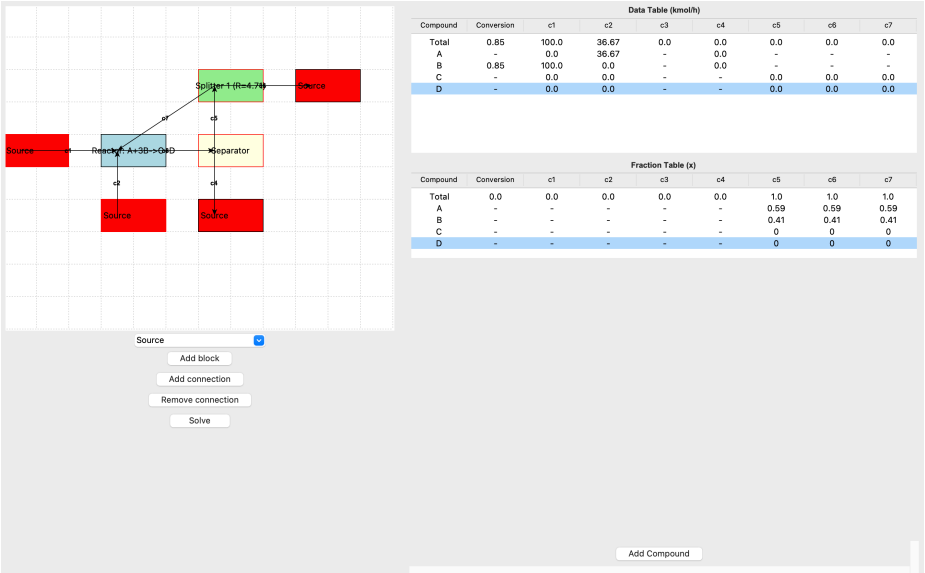


Figure 14. Case C – Initial configuration with the process flow diagram created and input data filled in the table.

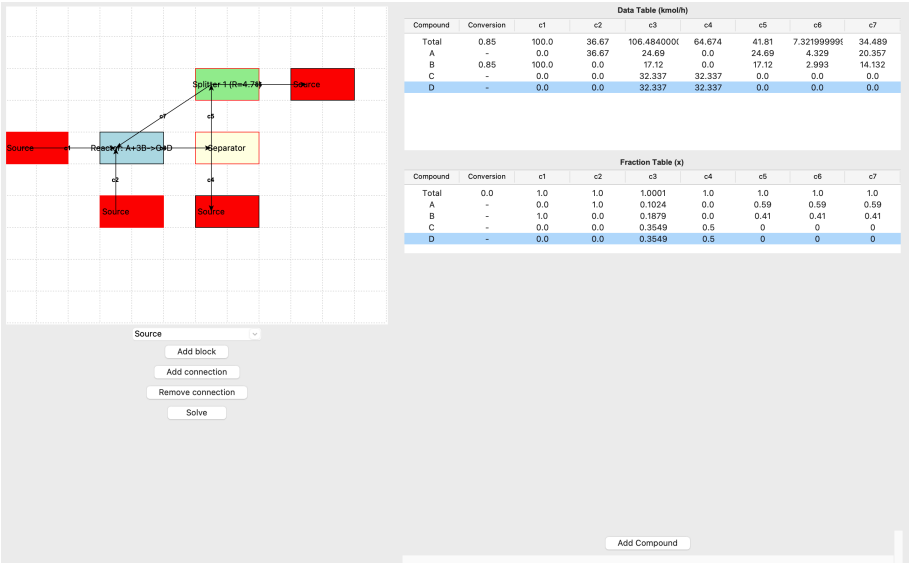


Figure 15. Case C – Results after pressing the "Solve" button, showing the completed table with calculated values.

5.2. COMPARISON WITH MANUAL CALCULATIONS

To confirm the system's reliability, results obtained with the tool were compared to step-by-step manual calculations for each case. Flow rates and mole fractions matched with a minimum precision of three decimal places, even in scenarios involving temporary variables or initial estimations.

Example – Case A:

Manual calculation for compound A at the reactor outlet:

$wB3 = wB1 \cdot (1 - X) = 100 \cdot (1 - 0.7) = 30$ kmol/h

$wB1 - wB3 - 3R = 0 \rightarrow R = (100 - 30)/3 = 23{,}33$

$wA2 - wA3 - R = 0 \rightarrow wA3 = 33{,}33 - 23{,}33 =$ **10 kmol/h**

Tool-calculated value: wA3= 9,997 kmol/h ⟶ Full numerical agreement.

This validation confirms that the system:

- Correctly implements stoichiometric balances.

- Properly applies conversion and compound relationships.
- Reproduces results expected in educational or technical environments.

5.3. CONVERGENCE AND PERFORMANCE OBSERVATIONS

The iterative resolution method is based on comparing temporary variable values across successive iterations. Convergence is considered achieved when all variations are below

1×10-4:

- In simple systems without recycle (Case A), convergence is achieved in a single iteration since no temporary variables are created.
- For medium-complexity systems with recycle (Case B), convergence is reached after 10 iterations.
- For more complex systems involving both recycle and purge (Case C), 22 iterations were required.

In none of the cases did the algorithm reach the predefined maximum of 30 iterations, which was chosen as a safety margin to ensure solvability in potentially more complex configurations.

The computational time remains virtually instantaneous (<1 s) for all tested scenarios. This represents a significant advantage over manual resolution, where solving times may range from 10–15 minutes in simple systems to over an hour for more complex ones.

This performance confirms that:

- The sequential resolution order (reactor → separator → splitter) ensures computational stability and efficiency.
- The combined management of dynamic and temporary variables prevents ill-defined systems.
- The algorithm is robust and capable of converging even when partial or redundant data is provided.

6. CONCLUSIONS AND FUTURE WORK

The development of this mass balance resolution tool has resulted in a complete, intuitive, and robust solution for solving macroscopic mass balances in steady-state continuous processes.

The project successfully met its main objective: to create a modular and functional computational tool that can assist both students and professionals in understanding and applying core concepts of chemical process analysis.

The main conclusions of this work are as follows:

- The system correctly solves sequential balances across reactors, separators, and splitters, handling recycle loops and conversion logic through a modular and scalable algorithm.
- The integration of an interactive flow diagram with editable data tables enhances usability and accessibility, even for users with no prior programming experience.
- The use of temporary and dynamic variables enables efficient convergence in systems involving recycle and purge streams, achieving consistent results in a small number of iterations.
- The tool has been validated through case studies, where the outputs matched the results of manual calculations, demonstrating both the accuracy and reliability of the implemented method.
- The modular design and code architecture allow for future functional expansion without compromising the current system.

This version of the tool achieves the initially proposed goals and serves as a solid starting point for further development. As a natural continuation of this project, and to extend the tool's applicability to more complex industrial cases, future versions may incorporate:

- Support for multiple simultaneous reactions within reactor blocks.
- Implementation of chemical equilibrium conditions.

- Integration of energy balances, enabling combined analysis of mass and energy in chemical systems.
- Calculation of reaction selectivity and product yields, especially relevant in systems with parallel or competitive reactions.

In summary, the tool developed in this project represents a complete and functional contribution for solving mass balance problems in steady-state systems. It lays the groundwork for further enhancements, while already being a practical and didactic resource for chemical engineering education and preliminary process analysis in real-world scenarios.

REFERENCES AND NOTES

1. Adams, T. A. II. *Learn Aspen Plus in 24 Hours*; 2nd ed.; McGraw-Hill Education: New York, 2022.
2. Perry, R. H.; Green, D. W. *Perry's Chemical Engineers' Handbook*; 7th ed.; McGraw-Hill: New York, 2008.
3. Gutiérrez González, J. M.; Chamarro Aguilera, M. E.; Maestro Garriga, A.; Sans Mazón, C.; Torres i Castillo, R. *Introducción a la Ingeniería Química: Balances Macroscópicos*; Edicions de la Universitat de Barcelona: Barcelona, 2020.
4. Izquierdo, J. F.; Costa López, J.; Martínez de la Ossa, E.; Rodríguez, J.; Izquierdo, M. *Introducción a la Ingeniería Química: Problemas Resueltos de Balances de Materia y Energía*; 2nd ed.; Reverté: Barcelona, 2011.
5. Chamarro Aguilera, M. E. *Lecture Notes for "Introduction to Chemical Engineering";* Universitat de Barcelona: Barcelona, 2023. *Unpublished teaching material*.
6. Microsoft Corporation. *Visual Studio Code* (Version 1.79) [Computer software]; https://code.visualstudio.com/ (accessed Jun 12, 2025).
7. Python Software Foundation. *The Python Language Reference, Release 3.11*; https://docs.python.org/3.11/reference/ (accessed Jun 12, 2025).
8. Python Software Foundation. *Python*; https://www.python.org (accessed Jun 12, 2025).
9. Reklaitis, G. V.; Schneider, D. R. *Introduction to Material and Energy Balances*; Wiley: New York, 1983.
10. Felder, R. M.; Rousseau, R. W. *Elementary Principles of Chemical Processes*, 3rd ed.; John Wiley & Sons: New York, 2004.

ACRONYMS

n: mol or kg that accumulate in the system.

w: molar (mol/s) or mass (kg/s) flow that enters (positive sign) or exits (negative sign) the system.

j= components (1...S)

m= streams (1...T)

i= chemical reactions (1...R)

Rj: molar flow (mol/s) of component j generated or consumed by chemical reactions it participates in. It is directly related to the extensive reaction rates in which component j is involved.

Mj: molecular mass (kg/mol) of component j.

vij: stoichiometric coefficient of component j in reaction i, with the following sign rule: positive if a product and negative if a reactant.

Ri: extensive reaction rate (mol/s) of reaction i in which component j is involved.

xj: molar fraction of component j in a given stream.

V: system or control volume (m^3) where accumulation may occur.

Ne: number of independent equations in the system.

Ni: number of unknown variables.

DoF: degrees of freedom of the system.

PFD: process flow diagram.

R: recycle ratio

c: stream label

X: conversion (fraction of limiting reactant converted, $0 \leq X \leq 1$)

APPENDICES

APPENDIX 1: FULL PYTHON CODE

Main.py:

```
import tkinter as tk
from interface import FlowDiagram
from data_table import DataTable

def main():
        root = tk.Tk()
        root.title("Material Balance Simulator")
        root.geometry("1200x600")

        data_table = DataTable(root)
        flow_diagram = FlowDiagram(root, data_table)
        root.mainloop()

if __name__ == "__main__":
        main()
```

Data_table.py:

```
import tkinter as tk
from tkinter import ttk, simpledialog, messagebox

class DataTable:
   def __init__(self, root):
      self.root = root

      self.frame_table = ttk.Frame(self.root, padding=10, width=800, height=400)
      self.frame_table.pack_propagate(False)
      self.frame_table.pack(side=tk.RIGHT, fill=tk.BOTH, expand=False)
```

```
        self.label_title = ttk.Label(self.frame_table, text="Data Table (kmol/h)",
font=("Arial", 12, "bold"))
        self.label_title.pack()

        self.canvas = tk.Canvas(self.frame_table, width=780, height=370)
        self.canvas.pack(side=tk.TOP, fill=tk.BOTH, expand=True)

        self.scroll_y = ttk.Scrollbar(self.frame_table, orient=tk.VERTICAL,
command=self.canvas.yview)
        self.scroll_x = ttk.Scrollbar(self.frame_table, orient=tk.HORIZONTAL,
command=self.canvas.xview)
        self.scroll_y.pack(side=tk.RIGHT, fill=tk.Y)
        self.scroll_x.pack(side=tk.BOTTOM, fill=tk.X)

        self.table_frame = ttk.Frame(self.canvas, width=780, height=370)
        self.table_frame.pack_propagate(False)

        self.canvas_frame = self.canvas.create_window((0, 0),
window=self.table_frame, anchor="nw")

        self.columns = ["Compound", "Conversion"]
        self.fixed_width = 780

        self.tree = ttk.Treeview(self.table_frame, columns=self.columns,
show="headings", height=10)
        self.tree.pack(side=tk.TOP, fill=tk.X, expand=True)

        self.tree.heading("Compound", text="Compound")
        self.tree.heading("Conversion", text="Conversion")
        self.tree.column("Compound", width=150, anchor="center", stretch=True)
        self.tree.column("Conversion", width=80, anchor="center", stretch=True)

        self.label_frac_title = ttk.Label(self.table_frame, text="Fraction Table (x)",
font=("Arial", 12, "bold"))
        self.label_frac_title.pack(side=tk.TOP, pady=(10, 2))
```

```
        self.fraction_tree = ttk.Treeview(self.table_frame, columns=self.columns,
show="headings", height=10)
        self.fraction_tree.pack(side=tk.TOP, fill=tk.X, expand=True)

        self.fraction_tree.heading("Compound", text="Compound")
        self.fraction_tree.heading("Conversion", text="Conversion")
        self.fraction_tree.column("Compound", width=150, anchor="center",
stretch=True)
        self.fraction_tree.column("Conversion", width=80, anchor="center",
stretch=True)

        self.table_data = {}
        self.fraction_data = {}
        self.manual_fractions = {}

        self.tree.insert("", "end", iid="Total", values=("Total", "0"))
        self.table_data["Total"] = {"Conversion": 0}

        self.fraction_tree.insert("", "end", iid="Total_x", values=("Total", "0"))
        self.fraction_data["Total_x"] = {"Conversion": 0}

        self.btn_add_compound = ttk.Button(self.frame_table, text="Add Compound",
command=self.ask_add_compound)
        self.btn_add_compound.pack(pady=5)

        self.tree.bind("<Double-1>", self.on_double_click)
        self.fraction_tree.bind("<Double-1>", self.on_fraction_double_click)

        self.canvas.config(yscrollcommand=self.scroll_y.set,
xscrollcommand=self.scroll_x.set)
        self.scroll_y.config(command=self.canvas.yview)
        self.scroll_x.config(command=self.canvas.xview)
        self.table_frame.bind("<Configure>", lambda e:
self.canvas.configure(scrollregion=self.canvas.bbox("all")))

        if "Total" not in self.tree.get_children():
```

```
            self.tree.insert("", "end", iid="Total", values=("Total", "0"))
        if "Total" not in self.table_data:
            self.table_data["Total"] = {}

    def display_table(self, frame):
        self.frame_table.pack_forget()
        self.frame_table.pack(side=tk.RIGHT, fill=tk.BOTH, expand=True)

    def ask_add_compound(self):
        compound_name = simpledialog.askstring("New Compound", "Enter the
compound name:")
        if compound_name:
            self.add_compound(compound_name)
        else:
            messagebox.showwarning("Error", "You must enter a compound name.")

    def add_compound(self, compound_name):
        if compound_name not in self.table_data:
            values = [compound_name] + ["-"] * (len(self.columns) - 1)
            self.tree.insert("", "end", iid=compound_name, values=values)
            self.table_data[compound_name] = {col: "-" for col in self.columns[1:]}
        else:
            if compound_name not in self.tree.get_children():
                values = [compound_name] + ["-"] * (len(self.columns) - 1)
                self.tree.insert("", "end", iid=compound_name, values=values)

        row_id_x = f"{compound_name}_x"
        if row_id_x not in self.fraction_data:
            values = [compound_name] + ["-"] * (len(self.columns) - 1)
            self.fraction_tree.insert("", "end", iid=row_id_x, values=values)
            self.fraction_data[row_id_x] = {col: "-" for col in self.columns[1:]}
        else:
            if row_id_x not in self.fraction_tree.get_children():
                values = [compound_name] + ["-"] * (len(self.columns) - 1)
                self.fraction_tree.insert("", "end", iid=row_id_x, values=values)
```

```
        self.sync_fraction_table_structure()

    def add_stream(self, stream_name):
        if stream_name and stream_name not in self.columns:
            self.columns.append(stream_name)
            self.tree["columns"] = self.columns
            self.fraction_tree["columns"] = self.columns

            column_width = max(80, self.fixed_width // len(self.columns))

            for col in self.columns:
                self.tree.heading(col, text=col)
                self.tree.column(col, width=column_width, anchor="center",
stretch=True)
                self.fraction_tree.heading(col, text=col)
                self.fraction_tree.column(col, width=column_width, anchor="center",
stretch=True)

            for item in self.tree.get_children():
                current_values = list(self.tree.item(item, "values"))
                current_values.append("-")
                self.tree.item(item, values=current_values)
                if item not in self.table_data:
                    self.table_data[item] = {}
                self.table_data[item][stream_name] = "-"

            for item in self.fraction_tree.get_children():
                current_values = list(self.fraction_tree.item(item, "values"))
                current_values.append("-")
                self.fraction_tree.item(item, values=current_values)
                if item not in self.fraction_data:
                    self.fraction_data[item] = {}
                self.fraction_data[item][stream_name] = "-"

            self.update_fraction_totals()
```

```
        self.canvas.configure(scrollregion=self.canvas.bbox("all"))

    self.sync_fraction_table_structure()

  def get_stream_value(self, stream_name):
    for item in self.tree.get_children():
      values = self.tree.item(item, "values")
      if values[0] == stream_name:
        return values[1]
    return None

  def update_stream_values(self, stream_name, values):
    col_index = self.columns.index(stream_name)
    for item in self.tree.get_children():
      if item != "Total":
        values_list = list(self.tree.item(item, "values"))
        values_list[col_index] = values.pop(0)
        self.tree.item(item, values=values_list)
        self.table_data[item][stream_name] = values_list[col_index]

  def on_double_click(self, event):
    col_id = self.tree.identify_column(event.x)
    row_id = self.tree.identify_row(event.y)

    if not col_id or not row_id:
      return

    col_index = int(col_id[1:]) - 1
    column_name = self.tree["columns"][col_index]
    old_value = self.tree.item(row_id, "values")[col_index]

    new_value = simpledialog.askstring("Edit Value", f"Enter new value for
{column_name}:", initialvalue=old_value)

    if new_value is not None:
      try:
```

```
            value = float(new_value)
        except ValueError:
            value = "-"

        values = list(self.tree.item(row_id, "values"))
        values[col_index] = str(value)
        self.tree.item(row_id, values=values)

        if row_id not in self.table_data:
            self.table_data[row_id] = {}
        self.table_data[row_id][column_name] = value

        if row_id == "Total":
            self.table_data["Total"][column_name] = value

            from calculations import MaterialBalanceSolver
            solver = MaterialBalanceSolver(self, None)
            solver.update_flows_from_fractions()

        self.update_totals()

    def update_totals(self, skip_columns=None):
        if skip_columns is None:
            skip_columns = []

        col_ids = self.tree["columns"][1:]
        totals = [0.0] * len(col_ids)

        for i, col in enumerate(col_ids):
            if col in skip_columns:
                totals[i] = self.table_data["Total"].get(col, 0)
                continue
            for item in self.tree.get_children():
                if item == "Total":
                    continue
```

```
            val = self.tree.item(item, "values")[i+1]
            try:
                if isinstance(val, str) and val.replace(".", "", 1).isdigit():
                    val = float(val)
                if isinstance(val, (int, float)):
                    totals[i] += val
            except Exception:
                continue

        total_values = ["Total"] + [str(round(t, 2)) for t in totals]
        self.tree.item("Total", values=total_values)

        for col, total in zip(col_ids, totals):
            self.table_data["Total"][col] = total

    def get_missing_values(self):
        missing_values = []
        for item in self.table_data:
            for col, value in self.table_data[item].items():
                if value is None or value == "-":
                    missing_values.append((item, col))
        return missing_values

    def update_table_with_resolved_values(self):
        for item in self.table_data:
            if item not in self.tree.get_children():
                continue

            for column, value in self.table_data[item].items():
                if isinstance(value, (int, float)):
                    values = list(self.tree.item(item, "values"))
                    for i, col in enumerate(self.columns):
                        if col == column:
                            values[i] = value
                    self.tree.item(item, values=values)
```

```
  def update_fraction_totals(self):
    col_ids = self.fraction_tree["columns"][1:]
    totals = [0.0] * len(col_ids)

    for item in self.fraction_tree.get_children():
      if item != "Total_x":
        values = self.fraction_tree.item(item, "values")[1:]
        for i, val in enumerate(values):
          try:
            if val != "-" and val !="":
              totals[i] += float(val)
          except ValueError:
            continue

    total_values = ["Total"] + [str(round(t, 4)) for t in totals]
    self.fraction_tree.item("Total_x", values=total_values)
    self.fraction_data["Total_x"] = {col: round(t, 4) for col, t in zip(col_ids, totals)}

  def sync_fraction_table_structure(self):
    print("Synchronizing fraction table structure...")

    self.fraction_tree["columns"] = self.columns
    for col in self.columns:
      self.fraction_tree.heading(col, text=col)
      self.fraction_tree.column(col, width=80, anchor="center", stretch=True)

    for item in self.tree.get_children():
      if item == "Total":
        continue

      row_id_x = f"{item}_x"
      if row_id_x not in self.fraction_data:
        row_values = []
        for col in self.columns:
          if col == "Compound":
```

```
                row_values.append(item)
            else:
                row_values.append("-")
        self.fraction_tree.insert("", "end", iid=row_id_x, values=row_values)
        self.fraction_data[row_id_x] = {col: "-" for col in self.columns[1:]}

    else:
        current_values = list(self.fraction_tree.item(row_id_x, "values"))
        while len(current_values) < len(self.columns):
            current_values.append("-")
        self.fraction_tree.item(row_id_x, values=current_values)

if "Total_x" not in self.fraction_data:
    total_values = []
    for col in self.columns:
        if col == "Compound":
            total_values.append("Total")
        else:
            total_values.append("0")
    self.fraction_tree.insert("", "end", iid="Total_x", values=total_values)
    self.fraction_data["Total_x"] = {col: 0 for col in self.columns[1:]}

else:
    current_values = list(self.fraction_tree.item("Total_x", "values"))
    while len(current_values) < len(self.columns):
        current_values.append("0")
    self.fraction_tree.item("Total_x", values=current_values)

self.update_fraction_totals()

def on_fraction_double_click(self, event):
    selected_item = self.fraction_tree.selection()
    if not selected_item:
        return

    col_id = self.fraction_tree.identify_column(event.x)
```

```
        row_id = self.fraction_tree.identify_row(event.y)

        if col_id and row_id and row_id != "Total_x":
            col_index = int(col_id[1:]) - 1
            column_name = self.fraction_tree["columns"][col_index]
            old_value = self.fraction_tree.item(row_id, "values")[col_index]

            new_value = simpledialog.askstring("Edit Fraction", f"Enter new value for
{column_name}:", initialvalue=old_value)

            if new_value is not None:
                values = list(self.fraction_tree.item(row_id, "values"))
                values[col_index] = new_value
                self.fraction_tree.item(row_id, values=values)

                try:
                    float_val = float(new_value)
                    self.fraction_data[row_id][column_name] = float_val
                except:
                    self.fraction_data[row_id][column_name] = "-"

                self.update_fraction_totals()

                compound = row_id.replace("_x", "")
                stream = column_name
                self.manual_fractions.setdefault(f"{compound}_x", {})[stream] = True
```

Interface.py:

```
import tkinter as tk
from tkinter import ttk, Canvas, messagebox, simpledialog
from calculations import MaterialBalanceSolver
from data_table import DataTable

class FlowDiagram:
    def __init__(self, root, data_table=None):
```

```
        self.root = root
        self.data_table = DataTable(self.root) if data_table is None else data_table
        self.material_solver = MaterialBalanceSolver(self.data_table, self)

        self.num_columns = 12
        self.num_rows = 10
        self.grid_size = 50
        self.canvas_width = self.num_columns * self.grid_size
        self.canvas_height = self.num_rows * self.grid_size

        self.frame_diagram = ttk.Frame(self.root, padding=10)
        self.frame_diagram.pack(side=tk.LEFT, fill=tk.Y, expand=False)

        self.canvas = Canvas(self.frame_diagram, width=self.canvas_width,
height=self.canvas_height, bg="white")
        self.canvas.pack()

        self.block_type = tk.StringVar()
        self.dropdown = ttk.Combobox(self.frame_diagram,
textvariable=self.block_type, state="readonly",
                                     values=["Reactor", "Splitter", "Separator", "Source"])
        self.dropdown.pack()

        self.btn_add_block = ttk.Button(self.frame_diagram, text="Add block",
command=self.add_block)
        self.btn_add_block.pack(pady=2)

        self.btn_add_connection = ttk.Button(self.frame_diagram, text="Add
connection", command=self.start_connection)
        self.btn_add_connection.pack(pady=2)

        self.btn_remove_connection = ttk.Button(self.frame_diagram, text="Remove
connection", command=self.remove_connection)
        self.btn_remove_connection.pack(pady=2)

        self.btn_solve = ttk.Button(self.frame_diagram, text="Solve",
command=lambda: self.debug_resolver())
```

```
        self.btn_solve.pack(pady=2)

        self.resize_grid()

        self.blocks = []
        self.sources = []
        self.connections = []
        self.selected_block = None
        self.creating_connection = False
        self.selected_block_coords = None
        self.connection_start = None

        self.table_frame = ttk.Frame(self.root, padding=10)
        self.table_frame.pack(side=tk.RIGHT, fill=tk.BOTH, expand=True)

        self.data_table.display_table(self.table_frame)

        self.canvas.bind("<Button-3>", self.delete_block_on_right_click)

        self.splitter_counter = 0
        self.splitter_ratios = {}
        self.splitter_names = {}

    def resize_grid(self):
        self.canvas.delete("grid")
        for col in range(self.num_columns + 1):
            x = col * self.grid_size
            self.canvas.create_line(x, 0, x, self.canvas_height, fill="lightgray", dash=(2,
2), tags="grid")
        for row in range(self.num_rows + 1):
            y = row * self.grid_size
            self.canvas.create_line(0, y, self.canvas_width, y, fill="lightgray", dash=(2,
2), tags="grid")

    def delete_block_on_right_click(self, event):
        for block in self.blocks + self.sources:
```

```
        region = self.canvas.bbox(block[0])

        if region[0] <= event.x <= region[2] and region[1] <= event.y <= region[3]:
            self.canvas.delete(block[0])
            self.canvas.delete(block[1])
            if block in self.blocks:
                self.blocks.remove(block)
            elif block in self.sources:
                self.sources.remove(block)
            messagebox.showinfo("Block Deleted", "The block has been successfully
deleted.")
            return

    messagebox.showwarning("Error", "Right-click on a block or source to delete
it.")

  def add_block(self):
    type = self.block_type.get()

    if not type:
      messagebox.showwarning("Error", "Please select a block type first.")
      return

    self.selected_block_coords = None

    self.btn_add_block.config(text="Click on the grid to place the block")

    self.canvas.bind("<Button-1>", self.select_cell_for_block)

  def select_cell_for_block(self, event):
    if self.selected_block_coords is None:
      col = event.x // self.grid_size
      row = event.y // self.grid_size
      self.selected_block_coords = (col, row)

      self.place_block_on_grid()
```

```
            self.btn_add_block.config(text="Add block")

            self.canvas.unbind("<Button-1>")
        else:
            messagebox.showwarning("Error", "A cell has already been selected.")

    def place_block_on_grid(self):
        if not self.selected_block_coords:
            return

        col, row = self.selected_block_coords
        type = self.block_type.get()

        if type == "Reactor":
            self.reaction = simpledialog.askstring("Reactor", "Enter the chemical
reaction (e.g., A -> B + C):")
            if self.reaction:
                self.create_reactor(self.reaction)
            else:
                messagebox.showwarning("Error", "A reaction must be entered.")
        elif type == "Splitter":
            self.create_splitter(col, row)
        elif type == "Separator":
            self.create_separator(col, row)
        elif type == "Source":
            self.create_source(col, row)

    def create_reactor(self, reaction):
        col = self.selected_block_coords[0]
        row = self.selected_block_coords[1]

        x, y = col * self.grid_size, row * self.grid_size
        block = self.canvas.create_rectangle(x, y, x + 2 * self.grid_size, y +
self.grid_size, fill="lightblue", tags="Reactor")
```

```
        text = self.canvas.create_text(x + self.grid_size, y + self.grid_size / 2,
text=f"Reactor: {reaction}")
        self.blocks.append([block, text, x, y, "Reactor"])

        self.update_data_table_for_reaction(reaction)

    def update_data_table_for_reaction(self, reaction):
        reactants, products, _ = self.parse_reaction(reaction)

        for reactant_name, _ in reactants:
            self.data_table.add_compound(reactant_name)

        for product_name, _ in products:
            self.data_table.add_compound(product_name)

    def parse_reaction(self, reaction):
        def parse_compound(compound_str):
            compound_str = compound_str.strip()
            if compound_str[0].isdigit():
                num = ""
                i = 0
                while i < len(compound_str) and compound_str[i].isdigit():
                    num += compound_str[i]
                    i += 1
                coeff = int(num)
                name = compound_str[i:].strip()
            else:
                coeff = 1
                name = compound_str
            return name, coeff

        reactants = []
        products = []
        coefficients = []

        parts = reaction.split("->")
```

```
        if len(parts) == 2:
            reactant_str, product_str = parts
            reactants = [parse_compound(r) for r in reactant_str.split("+")]
            products = [parse_compound(p) for p in product_str.split("+")]

            coefficients = [c for _, c in reactants + products]

        return reactants, products, coefficients


    def create_splitter(self, col, row):
        self.splitter_counter += 1
        splitter_name = f"Splitter {self.splitter_counter}"

        recycle_ratio = simpledialog.askfloat("Recycle ratio", f"Enter the recycle ratio
R for {splitter_name} (F_recycle / F_purge):", minvalue=0.0)
        if recycle_ratio is None:
            messagebox.showinfo("Cancelled", f"{splitter_name} was not created.")
            return

        x, y = col * self.grid_size, row * self.grid_size
        block = self.canvas.create_rectangle(x, y, x + 2 * self.grid_size, y +
self.grid_size, fill="lightgreen", tags="Splitter")
        text = self.canvas.create_text(x + self.grid_size, y + self.grid_size / 2,
text=f"{splitter_name} (R={recycle_ratio})")

        block_data = [block, text, x, y, "Splitter"]
        self.blocks.append(block_data)
        self.splitter_ratios[id(block_data)] = recycle_ratio
        self.splitter_names[block_data] = splitter_name


    def create_separator(self, col, row):
        x, y = col * self.grid_size, row * self.grid_size
        block = self.canvas.create_rectangle(x, y, x + 2 * self.grid_size, y +
self.grid_size, fill="lightyellow", tags="Separator")
        text = self.canvas.create_text(x + self.grid_size, y + self.grid_size / 2,
text="Separator")
```

```
        self.blocks.append([block, text, x, y, "Separator"])

    def create_source(self, col, row):
        x, y = col * self.grid_size, row * self.grid_size
        source = self.canvas.create_rectangle(x, y, x + 2 * self.grid_size, y +
self.grid_size, fill="red", tags="Source")
        text = self.canvas.create_text(x + self.grid_size / 2, y + self.grid_size / 2,
text="Source")
        self.sources.append([source, text, x, y])

    def start_connection(self):
        self.creating_connection = True
        self.connection_start = None
        messagebox.showinfo("Connection mode", "Select the source first, then the
destination.")
        self.btn_add_connection.config(text="Click on the source of the connection",
state=tk.DISABLED)

        self.canvas.bind("<Button-1>", self.select_block_for_connection)

    def select_block_for_connection(self, event):
        if self.connection_start is None:
            for block in self.blocks + self.sources:
                region = self.canvas.bbox(block[0])

                if region[0] <= event.x <= region[2] and region[1] <= event.y <=
region[3]:
                    self.connection_start = block
                    self.canvas.itemconfig(block[0], outline="red")
                    self.btn_add_connection.config(text="Click on the destination of the
connection")
                    return

            messagebox.showwarning("Error", "Click on a block or source to select it.")
        else:
            for block in self.blocks + self.sources:
                region = self.canvas.bbox(block[0])
```

```
            if region[0] <= event.x <= region[2] and region[1] <= event.y <=
region[3]:
                self.create_arrow(self.connection_start, block)

                self.connection_start = None
                self.canvas.itemconfig(block[0], outline="black")

                if block in self.blocks:
                    message = f"Connection between {self.connection_start[4]} and
{block[4]} successfully created."
                else:
                    message = f"Connection between {self.connection_start[4]} and
source successfully created."

                messagebox.showinfo("Connection created", message)

                self.btn_add_connection.config(text="Add connection",
state=tk.NORMAL)
                self.creating_connection = False
                self.canvas.unbind("<Button-1>")
                return

            messagebox.showwarning("Error", "Click on another block or source to
create the connection.")

    def create_arrow(self, block1, block2):
        x1, y1 = block1[2], block1[3]
        x2, y2 = block2[2], block2[3]

        start_center = (x1 + self.grid_size / 2, y1 + self.grid_size / 2)
        end_center = (x2 + self.grid_size / 2, y2 + self.grid_size / 2)

        arrow = self.canvas.create_line(start_center[0], start_center[1],
end_center[0], end_center[1], arrow=tk.LAST)
```

```
        stream_name = simpledialog.askstring("Stream name", "Enter the name of
the stream:")
        if stream_name:
            self.data_table.add_stream(stream_name)

            self.data_table.update_stream_values(stream_name, ["-"] *
len(self.data_table.table_data))

            label = self.canvas.create_text((start_center[0] + end_center[0]) / 2,
(start_center[1] + end_center[1]) / 2,
                                            text=stream_name, font=("Arial", 10, "bold"))

            self.connections.append({
                'start': block1,
                'end': block2,
                'arrow': arrow,
                'label': label,
                'direction': {'start': 'outlet', 'end': 'inlet'}
            })

            if block1 in self.blocks and block2 in self.sources:
                message = f"Connection between block and source successfully created."
            elif block1 in self.sources and block2 in self.blocks:
                message = f"Connection between source and block successfully created."
            else:
                message = f"Connection between block and block successfully created."

            messagebox.showinfo("Connection created", message)

            self.btn_add_connection.config(text="Add connection", state=tk.NORMAL)
            self.creating_connection = False
            self.canvas.unbind("<Button-1>")

    def remove_connection(self):
        if self.connections:
            last_connection = self.connections.pop()
```

```
            self.canvas.delete(last_connection[2])
            self.canvas.delete(last_connection[3])
            messagebox.showinfo("Remove connection", "Last connection deleted.")
        else:
            messagebox.showwarning("Warning", "There are no connections to
delete.")

    def debug_resolver(self):
        print("SOLVE button pressed")
        self.resolve_balances()

    def resolve_balances(self):
        print("Entering resolve_balances")
        missing_values = self.data_table.get_missing_values()
        print("Missing values found:", missing_values)

        if missing_values:
            self.material_solver.solve_balances()
            self.data_table.update_table_with_resolved_values()
            print("Balances resolved")
            messagebox.showinfo("Resolved", "The balances have been solved.")
        else:
            print("No missing values")
            messagebox.showinfo("Balance resolution", "No values to calculate. All
values are already entered.")
```

Calculations.py:

```
import numpy as np
import re
from data_table import DataTable

class MaterialBalanceSolver:
    def __init__(self, data_table, flow_diagram):
        self.data_table = data_table
        self.flow_diagram = flow_diagram
        self.temporary_variables = []
```

```
    self.dynamic_variables=[]
    self.initial_dynamic_variables = None
    self.max_iteration = 30
    self.initial_temporary_results = {}
    self.final_temporary_results = {}
    self.initial_mole_fractions = {}

  def solve_balances(self):
    print("Solving balances...")

    iteration = 0
    while iteration < self.max_iteration:
      print(f"\n Iteration {iteration + 1}")
      iteration += 1

      if iteration == 1:
        self.initial_mole_fractions = {}

        for comp in self.data_table.fraction_data:
          for stream, val in self.data_table.fraction_data[comp].items():
            was_manual = self.data_table.manual_fractions.get(comp,
{}).get(stream, False)
            if was_manual:
              if comp not in self.initial_mole_fractions:
                self.initial_mole_fractions[comp] = {}
              self.initial_mole_fractions[comp][stream] = val
              print(f" Saving initial manual fraction: {comp} in {stream} = {val}")

      self.reset_dynamic_variables()

      self.initial_temporary_results = {}
      for var in self.temporary_variables:
        try:
          stream, comp = var.split("_")
          value = self.data_table.table_data.get(comp, {}).get(stream)
          self.initial_temporary_results[var] = value
```

```
            print(f" Saving initial value of {var}: {value}")
        except Exception as e:
            print(f" Error retrieving initial value of {var}: {e}")

    for var in self.dynamic_variables:
        stream, comp = var.split("_")
        self.data_table.table_data[comp][stream] = "-"
        self.data_table.tree.set(comp, stream, "-")
        print(f" Clearing dynamic variable: {var}")

    self.final_temporary_results = {}

    for block in self.flow_diagram.blocks:
        type = block[4]
        try:
            if type == "Reactor":
                self.solve_reactor_balance(block, iteration)
            elif type == "Separator":
                self.solve_separator_balance(block)
            elif type == "Splitter":
                self.solve_splitter_balance(block)
        except Exception as e:
            print(f" Error solving block {block}: {e}")

    self.update_fractions_from_flows()
    self.data_table.update_fraction_totals()

    converged = False
    changes = []
    for var in self.temporary_variables:
        old = self.initial_temporary_results.get(var)
        new = self.final_temporary_results.get(var)

        print(f" Variable: {var}")
        print(f"    Old: {old} ({old.__class__.__name__})")
```

```
            print(f"    New: {new} ({new.__class__.__name__})")

            if old is not None and new is not None:
                diff = abs(new - old)
                changes.append(diff)
                print(f" Change in {var}: {old} → {new} (Δ = {diff})")

        if changes and all(d < 1e-4 for d in changes):
            print(" Convergence reached in temporary variables.")
            converged = True
        else:
            print(f" Variation in temporary variables: {changes}")

        if converged:
            break

    self.update_fractions_from_flows()
    self.data_table.update_fraction_totals()

  def reset_dynamic_variables(self):
    if self.initial_dynamic_variables is None:
        missing_values = self.data_table.get_missing_values()
        missing_values_set = set(f"{stream}_{comp}" for comp, stream in
missing_values)

        self.initial_dynamic_variables = [
            var for var in missing_values_set if var not in self.temporary_variables
        ]

        print(f" Initial dynamic variables detected:
{self.initial_dynamic_variables}")
    else:
        print(f" Reusing saved dynamic variables: {self.initial_dynamic_variables}")

    self.dynamic_variables = self.initial_dynamic_variables.copy()
```

```
  def solve_reactor_balance(self, reactor_block, current_iteration):
    inlets, outlets = self.get_streams_for_block(reactor_block)
    print("Inlets:", inlets, "Outlets:", outlets)

    components = list(self.data_table.table_data.keys())
    if "Total" in components:
      components.remove("Total")

    coefficients = self.get_reaction_coefficients(reactor_block)
    print("Reaction coefficients:", coefficients)

    variables = []

    for stream in inlets + outlets:
      for comp in components:
        var_name = f"{stream}_{comp}"
        if var_name in self.temporary_variables or var_name in
self.dynamic_variables:
          if var_name not in variables:
            variables.append(var_name)
            print(f" Adding {var_name} as unknown (dynamic or temporary)")

    R_var_name = f"R_{reactor_block[0]}"
    if R_var_name not in variables:
      variables.append(R_var_name)
      print(f"Adding variable {R_var_name} for reactor")

    print(f"Initial variables: {variables}")

    equations = []
    results = []

    limiting_reactant, conversion = self.get_limiting_reactant()
    print("Limiting reactant:", limiting_reactant, "Conversion:", conversion)
```

```
    if limiting_reactant:
        inlet_rls = [f"{inlet}_{limiting_reactant}" for inlet in inlets]
        outlet_rl = f"{outlets[0]}_{limiting_reactant}"

        print(f"Inlet(s) for {limiting_reactant}: {inlet_rls}")
        print(f"Outlet for {limiting_reactant}: {outlet_rl}")

        for inlet_rl in inlet_rls:
            if inlet_rl.split("_")[0] not in
self.data_table.table_data.get(limiting_reactant, {}):
                print(f"Error: The key {inlet_rl} is not found in data_table for
{limiting_reactant}")
            else:
                print(f"The key {inlet_rl} is present in data_table for
{limiting_reactant}")

        known_inlets = [inlet_rl for inlet_rl in inlet_rls if inlet_rl not in variables]
        num_known_inlets = len(known_inlets)

        rl_equation = [0] * len(variables)

        if num_known_inlets == len(inlet_rls):
            inlet_value_rl = None
            for inlet_rl in inlet_rls:
                inlet_key = inlet_rl.split("_")[0]
                inlet_value_rl = self.data_table.table_data.get(limiting_reactant,
{}).get(inlet_key, "-")
                if isinstance(inlet_value_rl, (int, float)):
                    break

            if isinstance(inlet_value_rl, (int, float)):
                outlet_value_rl = inlet_value_rl * (1 - conversion)
                results.append(outlet_value_rl)
            else:
                print(f"Error: The value of inlet {inlet_rls[0]} is not properly defined.")
                outlet_value_rl = 0
```

```
            results.append(outlet_value_rl)

        if outlet_rl not in variables:
            variables.append(outlet_rl)
        rl_equation[variables.index(outlet_rl)] = 1
        print(f"Conversion equation for {limiting_reactant}: {outlet_rl} =
{inlet_value_rl} * (1 - {conversion})")

    else:
        rl_constant = 0
        for inlet_rl in inlet_rls:
            inlet_key = inlet_rl.split("_")[0]
            val = self.data_table.table_data.get(limiting_reactant, {}).get(inlet_key,
"-")
            var_name = f"{inlet_key}_{limiting_reactant}"

            if isinstance(val, (int, float)):
                rl_constant += val * (1 - conversion)
                print(f"Given {inlet_rl}, value {val}, contribution to constant: {-val *
(1 - conversion)}")

            elif var_name in self.initial_temporary_results:
                temp_value = self.initial_temporary_results[var_name]
                rl_constant += temp_value * (1 - conversion)
                self.data_table.table_data[limiting_reactant][inlet_key] = temp_value
            else:
                if current_iteration == 1:
                    initial_value = 5.0
                    print(f" Assigning initial value to {var_name} (iteration 1)")
                else:
                    initial_value = self.initial_temporary_results.get(var_name, 5.0)
                    print(f" Using value from temporary_results for {var_name}:
{initial_value}")

                self.data_table.table_data[limiting_reactant][inlet_key] = initial_value
                rl_constant += initial_value * (1 - conversion)
```

```
                if var_name not in self.temporary_variables:
                    self.temporary_variables.append(var_name)

                stream_1 = inlet_key
                coef_rl = coefficients.get(limiting_reactant, 1)
                for comp, coef in coefficients.items():
                    if comp == limiting_reactant:
                        continue
                    nuevo_var = f"{stream_1}_{comp}"
                    if nuevo_var in self.temporary_variables:
                        continue
                    actual = self.data_table.table_data.get(comp, {}).get(stream_1, "-")
                    if actual in [None, "-"]:
                        try:
                            val_est = round((coef / coef_rl) * 5.0, 5)
                            self.data_table.table_data[comp][stream_1] = val_est
                            self.temporary_variables.append(nuevo_var)
                            self.initial_temporary_results[nuevo_var] = val_est
                            print(f" Assigning stoichiometric value {val_est} to
{nuevo_var}")
                        except ZeroDivisionError:
                            print(f"Division by zero while calculating value for
{nuevo_var}")

            if outlet_rl not in variables:
                variables.append(outlet_rl)
            idx_outlet_rl = variables.index(outlet_rl)
            rl_equation[idx_outlet_rl] = 1

            results.append(rl_constant)

            print(f"Conversion equation for {limiting_reactant}: {outlet_rl} =
sum(inlets) * (1 - {conversion})")

        equations.append(rl_equation)
```

```
    for comp in components:
        constant = 0
        equation = [0.0] * len(variables)

        for stream in inlets + outlets:
            var_name = f"{stream}_{comp}"
            val = self.data_table.table_data.get(comp, {}).get(stream, "-")
            sign = 1 if stream in inlets else -1

            if isinstance(val, (int, float)) and var_name not in
self.temporary_variables:
                constant -= sign * val
            elif var_name in self.temporary_variables:
                val_temp = self.initial_temporary_results.get(var_name, 5.0)
                constant -= sign * val_temp
            else:
                if var_name not in variables:
                    variables.append(var_name)
                idx = variables.index(var_name)
                equation[idx] = sign

        if comp in coefficients:
            idx = variables.index(R_var_name)
            equation[idx] = coefficients[comp]

        equations.append(equation)
        results.append(constant)

    final_variables = [v for v in variables if v not in self.temporary_variables]
    filtered_equations = []
    for eq in equations:
        new_eq = []
        for var in final_variables:
            idx = variables.index(var)
            new_eq.append(eq[idx] if idx < len(eq) else 0.0)
```

```
            filtered_equations.append(new_eq)

        try:
            A = np.array(filtered_equations, dtype=float)
            b = np.array(results, dtype=float)
            print(f"Matrix A:\n{A}")
            print(f"Vector b:\n{b}")
            x = np.linalg.lstsq(A, b, rcond=None)[0]

            for i, var in enumerate(final_variables):
                stream, comp = var.split("_")
                value = round(x[i], 3)
                print(f"Assigning {value} to {comp} in {stream}")
                self.data_table.table_data[comp][stream] = value
                self.data_table.tree.set(comp, stream, str(value))

                if var in self.temporary_variables:
                    self.final_temporary_results[var] = value

            self.data_table.update_totals()

        except Exception as e:
            print(f" Error solving reactor: {e}")

    def solve_separator_balance(self, block):
        inlets, outlets = self.get_streams_for_block(block)
        print(f"Inlets: {inlets} Outlets: {outlets}")

        components = list(self.data_table.table_data.keys())
        if "Total" in components:
            components.remove("Total")

        variables = []
        equations = []
        results = []
```

```
    for comp in components:
        print(f"\n Processing balance for compound: {comp}")
        constant = 0

        for stream in inlets + outlets:
            var_name = f"{stream}_{comp}"
            val = self.data_table.table_data.get(comp, {}).get(stream, "-")
            if not isinstance(val, (int, float)) or var_name in self.temporary_variables:
                if var_name not in variables:
                    variables.append(var_name)
                    print(f" Adding variable {'temporary' if var_name in
self.temporary_variables else 'unknown'}: {var_name}")
            else:
                sign = 1 if stream in inlets else -1
                constant -= sign * val
                print(f" Known flow of {comp} in {stream}: {val} → constant =
{constant}")

        equation = [0.0] * len(variables)

        for stream in inlets + outlets:
            var_name = f"{stream}_{comp}"
            sign = 1 if stream in inlets else -1
            if var_name in variables:
                idx = variables.index(var_name)
                equation[idx] = sign
                print(f" Coefficient {sign} for {var_name} (temporary: {'yes' if var_name
in self.temporary_variables else 'no'})")

        equations.append(equation)
        results.append(constant)
        print(f" Equation generated for {comp}: {equation} = {constant}")

    for stream in outlets:
        print(f"\n--- Processing stream: {stream} ---")
        fractions = []
```

```
        manual_fractions = []

        for comp in components:
            x_val = self.data_table.fraction_data.get(f"{comp}_x", {}).get(stream)
            if isinstance(x_val, (int, float)):
                fractions.append((comp, x_val))

                values = self.data_table.fraction_tree.item(f"{comp}_x", "values")
                index = self.data_table.columns.index(stream)
                visual_value = values[index] if index < len(values) else "-"

                flow_value = self.data_table.table_data.get(comp, {}).get(stream, "-")

                was_manual = self.data_table.manual_fractions.get(f"{comp}_x",
{}).get(stream, False)

                if was_manual or ((visual_value not in ["", "-", None]) and (flow_value in
["", "-", None])):
                    try:
                        val_float = float(visual_value)
                        if 0 <= val_float <= 1:
                            manual_fractions.append((comp, val_float))
                    except:
                        pass

        print(f"Manual fractions in {stream}: {[f[0] for f in manual_fractions]}")

        if len(manual_fractions) == 1:
            manual_comp, manual_x = manual_fractions[0]
            missing_comp = [c for c in components if c != manual_comp][0]
            missing_x = round(1.0 - manual_x, 4)
            print(f" Automatically deduced fraction: {missing_comp} in {stream} =
{missing_x:.4f}")

            manual_var = f"{stream}_{manual_comp}"
            missing_var = f"{stream}_{missing_comp}"
```

```
            for v in [manual_var, missing_var]:
                if v not in variables:
                    variables.append(v)

            eq = [0.0] * len(variables)
            manual_idx = variables.index(manual_var)
            missing_idx = variables.index(missing_var)
            eq[manual_idx] = missing_x
            eq[missing_idx] = -manual_x
            equations.append(eq)
            results.append(0.0)

            print(f" Constructed equation: {missing_x:.4f}*{manual_comp}_{stream}
= {manual_x:.4f}*{missing_comp}_{stream}")

        elif len(manual_fractions) == 2:
            comp_1, x1 = manual_fractions[0]
            comp_2, x2 = manual_fractions[1]

            var_1 = f"{stream}_{comp_1}"
            var_2 = f"{stream}_{comp_2}"

            for v in [var_1, var_2]:
                if v not in variables:
                    variables.append(v)

            eq = [0.0] * len(variables)
            idx_1 = variables.index(var_1)
            idx_2 = variables.index(var_2)
            eq[idx_1] = x2
            eq[idx_2] = -x1
            equations.append(eq)
            results.append(0.0)

            print(f" Equation (2 manual fractions): {x2:.4f}*{comp_1}_{stream} =
{x1:.4f}*{comp_2}_{stream}")
```

```
        else:
            print(f" No implicit equations generated in {stream} (manual fractions =
{len(manual_fractions)})")

    print("\n=== EQUATION SUMMARY BEFORE SOLVING ===")
    print(f"Variables: {variables}")
    for i, eq in enumerate(equations):
        print(f"Equation {i+1}: {eq} = {results[i]}")

    if not variables:
        print("No unknown variables. Nothing to solve.")
        return

    try:
        for eq in equations:
            while len(eq) < len(variables):
                eq.append(0.0)

        A = np.array(equations, dtype=float)
        b = np.array(results, dtype=float)
        x = np.linalg.lstsq(A, b, rcond=None)[0]
        print("\n=== SYSTEM SOLUTION ===")
        print(x)

        for i, var in enumerate(variables):
            stream, comp = var.split("_")
            value = round(x[i], 3)
            print(f"Assigning {value} to {comp} in {stream}")
            self.data_table.table_data[comp][stream] = value
            self.data_table.tree.set(comp, stream, str(value))

            if var in self.temporary_variables:
                self.final_temporary_results[var] = value

        self.data_table.update_totals()
```

```
            self.update_fractions_from_flows()

        except Exception as e:
            print(f" Error solving balances: {e}")
        pass

    def solve_splitter_balance(self, block):
        inlets, outlets = self.get_streams_for_block(block)
        print("Inlets:", inlets, "Outlets:", outlets)

        components = list(self.data_table.table_data.keys())
        if "Total" in components:
            components.remove("Total")

        variables = []
        equations = []
        results = []

        splitter_id = id(block)
        recycle_ratio = self.flow_diagram.splitter_ratios.get(splitter_id, 1.0)
        print(f" Using recycle ratio R = {recycle_ratio} for splitter {splitter_id}")
        splitter_name = self.flow_diagram.splitter_names.get(block[0],
f"Splitter_{splitter_id}")
        print(f" Using recycle ratio R = {recycle_ratio} for {splitter_name} (ID
{splitter_id})")

        fractions = {}
        base_fraction = {}

        print("🔍 Searching for manual fractions...")

        for comp in components:
            row_id = f"{comp}_x"
            for stream in inlets + outlets:
                x = self.initial_mole_fractions.get(row_id, {}).get(stream)
                if isinstance(x, (int, float)):
```

```
            base_fraction[comp] = x
            print(f" Initial manual fraction detected: {comp}_x in {stream} = {x}")
        if comp not in base_fraction:
            base_fraction[comp] = None

    if any(isinstance(v, (int, float)) for v in base_fraction.values()):
        manual_total = sum(v for v in base_fraction.values() if isinstance(v, (int,
float)))
        if manual_total == 0:
            print(" Invalid manual fractions: total sum 0.")
            return
        for comp in components:
            val = base_fraction.get(comp)
            fractions[comp] = round(val / manual_total, 6) if isinstance(val, (int,
float)) else 0.0
        print(" Base fractions (normalized):", fractions)

    else:
        print(" No manual fractions found, calculating from inlet flows...")

        total_inlet = {comp: 0 for comp in components}
        for comp in components:
            for stream in inlets:
                val = self.data_table.table_data.get(comp, {}).get(stream, "-")
                if isinstance(val, (int, float)):
                    total_inlet[comp] += val

        total_inlet_sum = sum(total_inlet.values())
        if total_inlet_sum == 0:
            print(" No inlet flow defined. Fractions cannot be calculated.")
            return

        fractions = {
            comp: round(total_inlet[comp] / total_inlet_sum, 6) for comp in
components
        }
```

```
        print(" Mole fractions calculated from inlet flows:", fractions)

    for comp in components:
        constant = 0
        equation = [0.0] * len(variables)
        has_information = False

        for stream in inlets + outlets:
            var_name = f"{stream}_{comp}"
            val = self.data_table.table_data.get(comp, {}).get(stream, "-")
            sign = 1 if stream in inlets else -1

            if isinstance(val, (int, float)) and var_name not in
self.temporary_variables:
                constant -= sign * val
                if val != 0:
                    has_information = True
            else:
                if var_name not in variables:
                    variables.append(var_name)
                    print(f" Adding variable {'temporary' if var_name in
self.temporary_variables else 'unknown'}: {var_name}")
                    for eq in equations:
                        eq.append(0.0)
                    equation.append(0.0)
                idx = variables.index(var_name)
                equation[idx] = sign
                has_information = True

        if has_information:
            equations.append(equation)
            results.append(constant)
        else:
            print(f" Equation for {comp} discarded (all zero)")

    for outlet in outlets:
```

```
        print(f"\n Composition equality in outlet: {outlet}")

        valid_components = [comp for comp in components if fractions.get(comp)
not in [None, 0]]

        if len(valid_components) < 2:
            print(f" Not enough valid components to enforce composition equality in
{outlet}")
            continue

        for i in range(len(valid_components)):
            for j in range(i + 1, len(valid_components)):
                comp_i = valid_components[i]
                comp_j = valid_components[j]
                xi = fractions[comp_i]
                xj = fractions[comp_j]

                if xi == 0 or xj == 0:
                    continue

                var_i = f"{outlet}_{comp_i}"
                var_j = f"{outlet}_{comp_j}"

                for v in [var_i, var_j]:
                    if v not in variables:
                        variables.append(v)

                eq = [0.0] * len(variables)
                idx_i = variables.index(var_i)
                idx_j = variables.index(var_j)
                eq[idx_i] = xj
                eq[idx_j] = -xi
                equations.append(eq)
                results.append(0.0)

                print(f" = {xj:.4f}·{var_i} = {xi:.4f}·{var_j}")
```

```
    self.enforce_recycle_ratio(components, outlets, variables, equations, results,
recycle_ratio=recycle_ratio)

    print("\n Variables:", variables)
    print("📐 Number of equations:", len(equations))

    if not variables:
        print(" No unknown variables. Nothing to solve")
        return

    try:
        for equation in equations:
            while len(equation) < len(variables):
                equation.append(0.0)

        A = np.array(equations, dtype=float)
        b = np.array(results, dtype=float)
        print(f"Matrix A:\n{A}")
        print(f"Vector b:\n{b}")
        x = np.linalg.lstsq(A, b, rcond=None)[0]
        print("\n System solution:", x)

        for i, var in enumerate(variables):
            stream, comp = var.split("_")
            value = round(x[i], 3)
            print(f" Assigning {value} to {comp} in {stream}")
            self.data_table.table_data[comp][stream] = value
            self.data_table.tree.set(comp, stream, str(value))

            if var in self.temporary_variables:
                self.final_temporary_results[var] = value

        self.data_table.update_totals()
        self.update_fractions_from_flows()
```

```
        except Exception as e:
            print(" Error solving splitter:", e)

        pass

    def get_streams_for_block(self, block):
        inlets = []
        outlets = []
        for conn in self.flow_diagram.connections:
            name = self.flow_diagram.canvas.itemcget(conn["label"], "text")
            if conn["end"] == block:
                inlets.append(name)
            elif conn["start"] == block:
                outlets.append(name)
        return inlets, outlets

    def get_limiting_reactant(self):
        for comp, values in self.data_table.table_data.items():
            if comp == "Total":
                continue
            conversion = values.get("Conversion")
            if isinstance(conversion, (int, float)):
                return comp, float(conversion)
        return None, None

    def get_reaction_coefficients(self, reactor_block):
        reaction_text = self.flow_diagram.canvas.itemcget(reactor_block[1], "text")
        if ":" in reaction_text:
            reaction = reaction_text.split(":", 1)[1].strip()
        else:
            reaction = reaction_text

        reactants, products = reaction.split("->")
        coef_dict = {}
```

```
        def parse_side(side, sign):
            for term in side.split("+"):
                term = term.strip()
                match = re.match(r"(\d*)\s*([A-Za-z]\w*)", term)
                if match:
                    coeff_str, name = match.groups()
                    coeff = int(coeff_str) if coeff_str else 1
                    coef_dict[name] = coef_dict.get(name, 0) + sign * coeff

        parse_side(reactants, -1)
        parse_side(products, 1)

        return coef_dict

    def update_fractions_from_flows(self):
        for stream in self.data_table.columns[2:]:
            total = 0
            caudales = {}

            for compound in self.data_table.table_data:
                if compound == "Total":
                    continue
                value = self.data_table.table_data[compound].get(stream, "-")
                if isinstance(value, (int, float)):
                    caudales[compound] = value
                    total += value

            if total <= 0:
                continue

            for compound in self.data_table.table_data:
                if compound == "Total":
                    continue

                row_id_x = f"{compound}_x"
```

```
            already_written = self.data_table.fraction_data.get(row_id_x,
{}).get(stream)

            if not isinstance(already_written, (int, float)) and compound in caudales:
                frac = round(caudales[compound] / total, 4)

                if row_id_x in self.data_table.fraction_tree.get_children():
                    values = list(self.data_table.fraction_tree.item(row_id_x, "values"))
                    index = self.data_table.columns.index(stream)
                    values[index] = frac
                    self.data_table.fraction_tree.item(row_id_x, values=values)

                self.data_table.fraction_data[row_id_x][stream] = frac

    def update_flows_from_fractions(self):
        print(" Updating flows from mole fractions...")

        columns = self.data_table.columns[2:]

        for stream in columns:
            total_val = self.data_table.table_data.get("Total", {}).get(stream)
            if not isinstance(total_val, (int, float)) or total_val <= 0:
                continue

            for compound in self.data_table.table_data:
                row_id = f"{compound}_x"
                x_val = self.data_table.fraction_data.get(row_id, {}).get(stream)
                if isinstance(x_val, (int, float)):
                    caudal = round(x_val * total_val, 3)
                    self.data_table.table_data[compound][stream] = caudal
                    self.data_table.tree.set(compound, stream, str(caudal))

        self.data_table.update_totals()


    def reconcile_flows_and_fractions(self):
```

```
        print("\n Reconciling flows and mole fractions...")

        compounds = [c for c in self.data_table.table_data if c != "Total"]
        streams = self.data_table.columns[2:]

        for stream in streams:
            print(f"\n Analysing stream: {stream}")
            total = self.data_table.table_data.get("Total", {}).get(stream)

            if isinstance(total, (int, float)) and total > 0:
                print(f" Total found in {stream}: {total}")
                has_fractions = any(
                    isinstance(self.data_table.fraction_data.get(f"{comp}_x",
{}).get(stream), (int, float))
                    for comp in compounds
                )
                if has_fractions:
                    print(f" Known fractions in {stream}, calculating flows..")
                    for compound in compounds:
                        x = self.data_table.fraction_data.get(f"{compound}_x", {}).get(stream)
                        if isinstance(x, (int, float)):
                            caudal = round(x * total, 4)
                            self.data_table.table_data[compound][stream] = caudal
                            self.data_table.tree.set(compound, stream, str(caudal))
                            print(f"   → Flow of {compound} in {stream}: {caudal}")
                else:
                    print(f" No fractions in {stream}. Flows not recalculated.")

            else:
                valid_flows = []
                for compound in compounds:
                    w = self.data_table.table_data.get(compound, {}).get(stream)
                    if isinstance(w, (int, float)) and w > 0:
                        valid_flows.append((compound, w))
```

```
            if len(valid_flows) >= 2:
                total_flows = sum(w for _, w in valid_flows)
                self.data_table.table_data["Total"][stream] = round(total_flows, 4)
                self.data_table.tree.set("Total", stream, str(round(total_flows, 4)))
                print(f" Total recalculated in {stream} by summing flows:
{total_flows}")

                for compound, w in valid_flows:
                    x = round(w / total_flows, 4)
                    row_id_x = f"{compound}_x"
                    self.data_table.fraction_data[row_id_x][stream] = x
                    self.data_table.fraction_tree.set(row_id_x, stream, str(x))
                    print(f"   → Fraction of {compound} in {stream}: {x}")

            elif len(valid_flows) == 1:
                print(f" Only one flow defined in {stream}. Neither total nor fractions
are recalculated.")

            else:
                print(f" Insufficient data in {stream}. Nothing recalculated.")

        self.data_table.update_totals()
        self.data_table.update_fraction_totals()
        print("\n Reconciliation completed.")

    def enforce_recycle_ratio(self, components, outlets, variables, equations,
results, recycle_ratio):
        if len(outlets) != 2:
            print(" Cannot impose recycle ratio: exactly 2 outlets are required.")
            return

        recycle_outlet = None
        purge_outlet = None

        for outlet in outlets:
            contains_temporaries = any(
```

```
            f"{outlet}_{comp}" in self.temporary_variables for comp in components
        )
        if contains_temporaries:
            recycle_outlet = outlet
        else:
            purge_outlet = outlet

    if not recycle_outlet or not purge_outlet:
        print(" Recycle ratio not enforced: recycle and purge not identified.")
        return

    eq = [0.0] * len(variables)
    has_valid_components = False

    for comp in components:
        var_r = f"{recycle_outlet}_{comp}"
        var_p = f"{purge_outlet}_{comp}"

        val_r = self.data_table.table_data.get(comp, {}).get(recycle_outlet)
        val_p = self.data_table.table_data.get(comp, {}).get(purge_outlet)

        include = False
        if (var_r in self.temporary_variables) or (var_p in self.temporary_variables):
            include = True
        elif (isinstance(val_r, (int, float)) and val_r != 0) or (isinstance(val_p, (int,
float)) and val_p != 0):
            include = True

        if include:
            for v in [var_r, var_p]:
                if v not in variables:
                    variables.append(v)
                    for e in equations:
                        e.append(0.0)
                    eq.append(0.0)
```

```
            idx_r = variables.index(var_r)
            idx_p = variables.index(var_p)

            eq[idx_r] += 1.0
            eq[idx_p] += -recycle_ratio
            has_valid_components = True

    if has_valid_components:
        equations.append(eq)
        results.append(0.0)
        print(f" Recycle ratio enforced: {recycle_outlet} = {recycle_ratio:.2f} ×
{purge_outlet}")
    else:
        print(" Recycle ratio not enforced: no valid components.")
```