Simulation Environment of a Gas Absorber System

By:

Osama Al Mazloum (7333584),

Shudip Faiyaz (7762218),

Ahmed Ibrahim (6912371),

Faiyaz Mohammad Zaman (7804931),

Mohammed Sayeem (7493900).

Research report submitted to Dr. David Taylor and Emily Cossar in partial fulfilment of the requirements for the course CHG 4343

University of Ottawa

Department of Chemical and Biological Engineering

Ottawa, Ontario, Canada

December 10th, 2018

Executive Summary

A Java program was designed to simulate the diffusion of a solute from a stagnant gas-side to a stagnant liquid-side for an isothermal and isobaric packed absorption column. The governing equations used to model the system incorporate standard assumptions such as: the gas and liquid phases are immiscible, isothermal conditions and isobaric conditions. The program enables the user to compute the height of the column by specifying a set of parameters. By varying the solute-free inlet liquid flow rate, the program can determine the optimal liquid flow rate and height. An object-oriented programming approach was used to develop a robust, versatile and extendable set of methods and classes which can be utilized to conduct comparative analysis of various inlet conditions and the program can be embedded in a larger process simulation environment similarly to UNISIM.

The simulation was architected to mimic as close as possible the modular features of a gas absorption system. When the program is run, the user is invited to set parameters in an `input.txt` file. These values include the choice of packing material, where there are three: `BerlSaddles`, `PallRings`, `RashigRings`. It also includes the inlet gas/liquid flow rates and fractions, equilibrium mixture coefficients to generate the equilibrium equation, the molecular weight of the substances (solute, liquid and gas), the thermodynamic properties of the fluids and settings for the numerical methods (tolerance, maximum iterations, number of steps …). These inputs are processed via a series of methods which scans the file, removes white spaces, tokenizes the words, separates the names/values as well as loads them in an array, analyzes the array of token to assign values to the right instance variables, where the values are respect specified bounds. This herein exception handling is done in a try-catch block. The stored values are used to calculate the outlet liquid/gas flow rates and fractions, the Schmidt numbers and instantiate a `PackingMaterial` with a `PackingMaterial` object containing the properties of the selected packing material. The general binary mixture is tested to verify that the gas solute fractions are larger than the operating lines for the full range of solute liquid fractions. The height of the column is computed in the `AbsorptionSimulation` class, where to compute the mole fraction at the liquid phase interface a root finding method was used being Ridder's method. The computation of the column height also requires a numerical integration, where three were utilizes being: Trapezoid, Simpson, Modified-Simpson. Therefore, three set of results were generated for every numerical integration. This feature is beneficial to the user since these numerical integration differ in accuracy and performance depending on the curvature of the function it approximates. The use of interface and polymorphism for the numerical methods enables the `AbsorptionSimulation` class to use numerous numerical methods and these numerical methods can be reused with ease in other classes. The `Optimization` class was designed to optimize find the optimal height of the column, where the user controls the number of liquid flow rates to be tested and the range of liquid fraction to be scanned. The optimized heights for every numerical integration methods are reported in table outputted via a the UserOutputs class which outputs the table to an `xls` file. The validation campaign of the exception handling, the methods and classes of the program were conducted using an Excel simulator and a series of validation main methods. The resulting campaign revealed that the output from the program corresponds to the output of the Excel simulator. By simulating various cases, it seemed the Simpson method was deemed more efficient for converging to an optimum height and liquid flow rate. As for the anonymous code validation, a validation campaign was also done by comparing the outputted values to the same Excel Simulator, the values were the same with infinitesimal differences. However, this program lacked input exception handling, an output file and a minimal number of parameters can be manipulated by the user.

Table of Contents

List of Figures

List of Tables

Nomenclature

D_{AB}	Diffusion of gas in liquid $\left(\frac{m^2}{s}\right)$
D_c	Diameter of the column (m)
D_p	Nominal packing size (m)
dZ_V	Tower height differential calculated from the gas stream (m)
dZ_L	Tower height differential calculated from the liquid stream (m)
f_p	Mass transfer packing factor
G_L	Mass flux of liquid stream $\left(\frac{kg}{s \cdot m^2}\right)$
G_V	Mass flux of gas stream $\left(\frac{kg}{s \cdot m^2}\right)$
$k'_x a$	Film mass transfer coefficient in the liquid phase $\left(\frac{kgmol}{s \cdot m^3}\right)$
$k'_y a$	Film mass transfer coefficient in the gas phase $\left(\frac{kgmol}{s \cdot m^3}\right)$
L	Liquid flowrate $\left(\frac{kgmol}{h}\right)$
L_f	Solute free liquid flowrate $\left(\frac{kgmol}{h}\right)$
L_1	Outlet liquid flowrate $\left(\frac{kgmol}{h}\right)$
L_2	Inlet liquid flowrate $\left(\frac{kgmol}{h}\right)$
$\overline{MW_L}$	Average molecular weight of liquid stream $\left(\frac{kg}{kmol}\right)$
$\overline{MW_G}$	Average molecular weight of gas Stream $\left(\frac{kg}{kmol}\right)$
MW_L	Molecular weight of liquid stream $\left(\frac{kg}{kmol}\right)$
MW_G	Molecular weight of gas stream $\left(\frac{kg}{kmol}\right)$
$N_{Sc,\,L}$	Liquid Schmidt number
$N_{Sc,\,G}$	Gas Schmidt number
ρ_L	Density of liquid $\left(\frac{kg}{m^3}\right)$
ρ_G	Density of gas $\left(\frac{kg}{m^3}\right)$
S	Cross-sectional area of the column (m^2)
V	Gas flowrate $\left(\frac{kgmol}{h}\right)$
V_f	Solute free gas flowrate $\left(\frac{kgmol}{h}\right)$

V_1	Inlet gas flowrate $\left(\frac{\text{kgmol}}{\text{h}}\right)$
V_2	Outlet gas flowrate $\left(\frac{\text{kgmol}}{\text{h}}\right)$
X_{A1}	Mole fraction of solute in the outlet liquid stream
X_{A2}	Mole fraction of solute in the inlet liquid stream
x_{AL}	Bulk Concentration of Solute A in the Liquid Phase
x_{Ai}	Mole fraction of solute A in liquid at interface
Y_{A1}	Mole fraction of solute in the inlet gas stream
Y_{A2}	Mole fraction of solute in the outlet gas stream
y_{AG}	Bulk Concentration of Solute A in the Gas Phase
y_{Ai}	Mole fraction of solute A in gas at interface
$(1 - x_A)_{iM}$	Log mean of the concentration of the liquid at interface
$(1 - x_A)_{iM}$	Log mean of the concentration of the gas at interface
μ_L	Viscosity of liquid (Pa·s)
μ_V	Viscosity of Gas (Pa·s)
Z_V	Tower height calculated from the gas stream (m)
Z_L	Tower height calculated from the liquid stream (m)

1. Introduction

In the era when advancement of digital tools and computational evaluations are shaping all industries, the manipulation of computer-aided simulations are becoming an integral part of constructing industrial processes and innovations. Currently, a chemical engineer can employ the power of automated simulations to expeditiously model the governing complexities of unit operations to realistic bounds. With the use of a program, a virtual environment can be created to optimize the operation and design parameters without the need of a physical model and reduce the cost of testing. Software-based tools have led to more efficient data analysis methods and broadened the scope of process optimization.

A method of implementing computational analysis with the help of object-oriented programming (OOP). OOP is one the most efficient programming paradigm that has been created in order to manipulate data based on the concept of “objects”. The main feature about programming languages is the classification of variables to store data and linking them to associated procedures that can be executed in the form of functions and methods. But the specialty of objects being incorporated for these procedures create an integrated system of methods that can call on each other throughout the entire operation. They can be accessed within a complex internal structure serving as references to a single instance of the object of a class in an enormous assortment of memory. Furthermore, OOP can be used to provide a layer of abstraction to separate internal code from the input, giving the user the choice to specify variables within set of realistic parameters as a package, rather than ensuring the inputted data are not unrelated to the system.

When implementing OOP for unit operations, such as absorption columns, the general attributes and functionality for processes as well as specific variations can be unified for one simulation. For instance, the packing within a column dictates the variation of the physical properties of the separation process although they achieve the same goal. The inheritance characteristics within the program can implement these variations by distinguishing the child classes to represent the parameters of the different packing materials. Therefore, OOP possess the appropriate characteristics to exemplify computational manipulation of chemical engineering processes. The process that will be described throughout this report has used the program called Java in order to establish the governing performance of absorption columns and aid in the process design and optimization.

2. Background

A simulator design code for absorption tower was written JAVA for this design project. Absorption tower is a unit operation that is utilized for the removal of gas impurities or solute from a rising gas stream using a liquid solvent. In this project, the main objective was to determine a column height with optimum liquid flow rate. The simulator was required to output all flow rates and composition with the tower heights for vapor and liquid side based on the inputs of vapor flow rate with the mole fraction of the solute, liquid flow rate of the solvent and recovery percentage of the solute in the liquid solvent.

2.1. Design Assumptions

The absorption column was assumed to be operating at isothermal temperature of 25 ºC. The pressure drop along the column was considered negligible, and the change in equilibrium relationship with pressure was neglected. Viscosity, fluid density and diffusion coefficient were assumed constant with stream composition. The solvent carrier gas was considered immiscible in the liquid, and it was assumed the liquid did not evaporate into gas phase resulting the solute free flows of gas and vapor to be constant.

2.2. Governing Equations

The governing equations to solve the absorber tower system was developed using overall material balance, solute material balance, operating line and equilibrium modelling equation.

2.3. Overall material balance

An overall material balance was performed on the absorber tower as shown in ***Figure 5***. The material balance is shown below by EQ1.

$$L'\left(\frac{x_{A2}}{1-x_{A2}}\right)+V'\left(\frac{y_{A1}}{1-y_{A1}}\right)=L'\left(\frac{x_{A1}}{1-x_{A1}}\right)+V'\left(\frac{y_{A2}}{1-y_{A2}}\right) \quad \text{EQ1}$$

Where,

$$V'=V(1-y_A) \quad \text{EQ1a}$$

$$L'=L(1-y_A) \quad \text{EQ1b}$$

2.4. Operating line

The operating line was determined from the inputs to the simulation and performing the overall material balance. The operating line was found by the following equation.

$$L'\left(\frac{x}{1-x}\right)+V'\left(\frac{y_{A1}}{1-y_{A1}}\right)=L'\left(\frac{x_{A1}}{1-x_{A1}}\right)+V'\left(\frac{y}{1-y}\right) \quad \text{EQ2}$$

2.5. Equilibrium Modelling Data

The balance between the solute in its gas and liquid phase is explained by the equilibrium relationship. For this design, the equilibrium data was modelled by a quadratic equation for isothermal operation at an initial pressure of 1.7 atm. The equilibrium relationship is shown below.

$$y_{Ai,25°C}=1.9762x_{Ai}^2+0.0812x_{Ai} \quad \text{EQ3}$$

2.6. Numerical method

Root finding method such as Ridders' method was implemented to find the interface composition by solving a non-linear equation derived from the operating line and equilibrium line. The overall development of this non-linear equation is presented Appendix **Error! Reference source not found.**. The Ridders' method is shown below.

$$x_R=x_M+\frac{(x_M-x_L)sign[f(x_L)-f(x_U)]f(x_M)}{\sqrt{f(x_M)^2-f(x_L)(x_U)}} \quad \text{EQ4}$$

From an initial false variable, the Ridders' method seeks for a solution. A midpoint XM is calculated from a given lower bound, XL and upper bound, XU. Then using XL, XU and XM, an estimated root, XR, is calculated using EQ4. The value of XR is then used to find if it is larger or smaller than the midpoint. The root is found by repeating the whole procedure until a converging point is reached in a given tolerance range. The bound for selecting the bounds are shown in ***Figure 6***.

2.7. Numerical Integration

Numerical integration was used to calculate the tower height of the absorber column. The numerical integration used in this design were trapezoid, Simpson's rule and modified Simpson's rule, and the equations for them are shown below respectively [4].

$$I = \frac{(b-a)}{2n}[f(x_0) + 2\sum_{i=1}^{n-1} f(x_i) + f(x_n)] \quad \text{EQ5}$$

$$I = \frac{(b-a)}{3n}[f(x_0) + 4\sum_{i=1,3,5}^{n-1} f(x_i) + 2\sum_{i=2,4,6}^{n-2} f(x_i) + f(x_n)] \quad \text{EQ6}$$

$$I = \frac{(b-a)}{48}[17f(x_0) + 59f(x_1) + 43f(x_2) + 49f(x_3) + 48\sum_{i=4}^{n-4} f(x_i) + 49f(x_{n-3}) + 43f(x_{n-2}) + 59f(x_{n-1}) + 17f(x_n)] \quad \text{EQ7}$$

3. Design & Discussion

3.1. Initialization

When the simulation is commenced, its first step is to identify a `main` method, if not found, an error would be thrown stating that no main method can be found in the package. This `main` method is rendered `public` to enable java runtime to execute its commands. When the same java runtime commences, there are no object of this class present; thus, it must be `static`. It does not return anything, therefore by default its return type is `void`. The same Java main method accept a single argument of type `String` array, also referred to as Java command line arguments. This herein main method is found in the `MainSimulation` class, which it contains a single line, where an object of `SimulationEnvironnement` class is instantized to initialize the simulator. As a standalone program, this minimalistic main method was designed to firstly encapsulate the embodied objects inside of the `SimulationEnvironnement` class. It is a concrete class, which contains the instances of four classes being: `UserInputs`, `AbsorptionSimulation`, `Optimization` and `UserOutputs`. These classes represent the core of this simulation and segregates the four major components of this program. As the name of these classes entice, the `UserInputs` class contains the methods to prompt the user to enter the file path to a file containing

their data input, it processes this dataset, computes a mass balance and conducts the necessary exception handling to catch a collection of mistakes and/or mismatches. The `AbsorptionSimulation` class utilizes the processed data from the `UserInputs` class, calls the necessary numerical methods classes, captures the parameters of the chosen packing material to compute the height of the absorption column and the error percentage between the computed liquid-side and gas-side height. The `Optimization` class searches for an optimized solute-free liquid flow rate, which minimizes the described error percentage. The `User-Outputs` class compiles the generated data from the `AbsorptionSimulation` class and `Optimization` class to generate a table summarizing the simulation output in an `Output.xls` file.

3.2. Simulation

The gas absorption simulation module requires the user to input their design constraints via an `input.txt` file. The template of the `input.txt` file contains 28 variables that the user must initialize manually. The first setting is the packing material choice (`packingMaterialChoice`), which the user is invited to enter either 0, 1, or 2 for either Rashig ring, Berl saddle and Pall ring respectively. To compute the composition and flowrates of the outlet liquid and gas stream, 5 variables need to be specified to obtain a unique solution being the inlet liquid flow rate (`l_2`), the inlet gas flow rate (`v_1`), inlet gas composition (`y_A1`), inlet liquid composition (`x_A2`). The user can set a value for the molecular weight of the solute (`mw_A`), liquid solvent (`mw_L`) and gas solvent (`mw_V`). Substances have a unique molecular weight and specific thermodynamic properties (viscosity, density, diffusion coefficient) for a set temperature and pressure. Thus, the user is enabled to input the viscosity of both fluids (`u_L` and `u_V`), the density of both fluids (`p_L` and `p_V`), the diffusion coefficient of the solute through either phase (`d_ABL` and `d_ABV`). Additionally, the user can model a general binary mixture via a polynomial equation up to degree 5 (`equilibriumPolynomialCoefficientN`, where N = 0,1,2,3,4,5). The last set of instance variables consist in the `input.txt` file are the settings for the numerical methods and optimization such as: the lower/upper bound liquid fraction used in the Ridder's method (`root_Finder_X_L`, `root_Finder_X_U`), the tolerance for the Ridder's method (`root_Finder_Tolerance`), the maximum number of iterations for the Ridder's method (`root_Finder_Maximum_Iterations`), the number of steps for the Simpson's method (`numerical_Integration_Number_Of_Steps`), the number steps for the optimization procedure (`optimizer_Number_Of_Steps_Liquid_Flow_Rate`), the minimum liquid flow rate for the optimization (`optimizer_Minimum_Liquid_Flow_Rate`). These settings are crucial to enable the user to easily manipulate the processing speed of the program and the resulting precision of the overall iterative process. Once the user is done inputting values to the `input.txt` file, they are ready to proceed to simulation in accordance to the exception handling embedded in the `UserInputs` class.

3.2.1. UserInputs.class

The `UserInputs` class contains the methods to read the `input.txt` file and instances variables to store its values. To do-so, it requires four methods notably: `getFileContent()`, `parseFileContents(…)`, `removeEmptyStrings(...)`, `interpretToken(…)`. The `getFileContent()` method is of `String` type and is the first method executed, when the `UserInputs` class is instantized. It contains a scanner reader to prompt the user to enter the file path to their `input.txt` file. The file path must be specific to a file and not solely a folder. This setup enables the user to prepare numerous input.txt files and choose which file to process for a run by manually changing the file path. The herein scanner reader commands are also incorporated in a try-catch block. This block scans the full content of the file, otherwise it will report an error on the console stating that the desired file is not along the specified file path. If the file path is correctly inputted and it points to an `input.txt` file, the content of this file will be stored in a local variable of `String` type being, `fileContents`. The `getFileContent()` method returns the loaded `fileContents` variable. At this point, every variable and respective value are interpreted as one token, whom the name and value are separated by an equal sign. To parse these tokens, the `parseFileContents(…)` is called, where its parameter is the file content returned by the `getFileContent()` method. To remove whitespaces in the file content, the `replace(" ", "")` method is used. Then, an array of type String (`tokens`) is instantiated by `fileContents.split("=|\n")`, where the `split("=|\n")` method separated the names and values into two separate tokens. The names and values are stored in alternating order in the `tokens` array. The `removeEmptyStrings(…)` method is invoked to remove all the empty strings in the processed `tokens` array. The exception handling for the `tokens` array is done via `interpretToken(…)` method, where its parameters are two instances variables of `String` type being `name` and `values`.

A for-loop is setup inside of `parseFileContents(…)`, to loop every pair of name and values in the tokens array through the `interpretToken(…)` method. This method has a try-catch block containing an if-statement for every variable initialized in the `input.txt` file. Every if-statement contain two conditions, one to check the name of the `token` (`equalsIgnoreCase(name)`) the other to check if the value is within the boundaries set by the Boolean method. For instance, if the `name` of a given `token` doesn't match with any of the names of the if-statement, an error will be written on the console to warn the user of an invalid variable type. Otherwise, if a name matches; its corresponding value must be within the boundaries of the Boolean method or an error statement would be printed to the console to report the value is out of bounds. When both constraints are `truth` for a given variable, the `Double.parseDouble(value)` method is used to convert the `value` variable from a `String` type

to Double type and the resulting variable is stored in its appropriate instance variable via a barebone mutator. The mutators for these variables serve solely this purpose and as a result, they were rendered private. If the names of the variables in the input.txt file are correctly spelled, and the inputted values are within bounds, the instances variables associated to the input.txt file would all have a value.

Once the values have been extracted from the text file and stored in its appropriate instance variables, these instance variables can be utilized to compute necessary parameters for the height calculations. However, these parameters are computed in the UserInputs class since they are necessary to validate the general binary mixture and they remain constant throughout the iterative process of computing the height of the absorption column. It was worthwhile creating instance variables for these computed parameters and allowing other classes to access their values only via accessors. Since the ability to mutate these instance variables outside of the UserInputs class is not necessary, the methods needed to compute these values are rendered private and final. Among these parameters, based on the stored value of the packingMaterialChoice instance variable, an object of PackingMaterial is instantiated and downcasted to a child class of either BerlSaddles, PallRings, RashigRings. This is accomplished by the methodsetPackingMaterials() method. It has two local variables being one of type PackingMaterial, and another of type Double which stores the choice of packing material via an accessor. Using if-statements, the myPackingMaterialObjects local variable is instantiated by the appropriate child class of PackingMaterial class. The resulting myPackingMaterialObjects local variable is returned by the method (See section 3.3.2 PackingMaterial class).

The Schmidt number for both the gas-side and liquid-side are computed via the calculate-SchmidtGas() and calculateSchmidtLiquid(). Similarly, using the values stored in the instances variables, the V_f, L_f, V_2, Y_A2, X_A1, L_1 were calculated in subsequent order using the governing equations EQ1, EQ1a, EQ1b, and EQ1 from the Background section 2.2,2.3 and 2.4.These governing equations were rearranged to calculate these flow rates and fractions in the following methods: calculateV_f(), calculateL_f(), calculateV_2(), calculateY_A2(), calculateX_A1(), calculateL_1(), calculateSchmidtGas(), calculateSchmidt-Liquid(). These parameters were necessary to compute the slope and y-intercept of the operating line

via the calculateSlopeofOperatingLine() and calculateintercept-ofOperatingLine() methods. Both the operating line and equilibrium equation are necessary to validate the inputted equilibrium equation inputted by the user.

The correctGeneralBinaryMixture() method returns true for any value of liquid fraction between X_A1 and X_A2 if the computed vapor fraction from the equilibrium equation is higher than the one of the operating line. If this is not the case, the method would return false. The herein method starts by storing the coefficients via accessors of the equilibrium equations, and the slope/Y-Intercept of the operating line in local variables. The values of X_A1 and X_A2 are also stored in local variables to define the boundaries of the liquid fractions to be tested. An array of liquid fractions (my_X_AI_test) is initialized to a length of 1000 values which will represent the liquid fractions in ascending order within the liquid fraction boundaries. An array of vapor fractions for the operating line (my_Y_A_OL_test) and the equilibrium equation (my_Y_AI_EPC_test) is set to have a length equal to my_X_AI_test. A Boolean local variable (booleanCorrectGeneralBinaryMixture) is set initially to true. There's also an array of Boolean type (isCorrectGeneralBinaryMixture) that stores a Boolean value at every iteration. A for-loop is setup with a nested if-else statement; for every iteration, the vapor fraction will be calculated for both equations using the liquid fraction from my_X_AI_test and stored in their respective arrays. For the flow control of the if-statement, if my_Y_AI_EPC_test[j] is larger or equal to my_Y_A_OL_test[j], both isCorrectGeneralBinaryMixture and booleanCorrectGeneral-BinaryMixture will be set to true. Otherwise, these two Boolean variables will be set to false and the for-loop would be halted by break. If the for-loop scans the full set of values, the correctGeneralBinary-Mixture() method will return true and or else, it will return false. It is assumed in the design of this method that if a 1000 value of liquid fractions over a pre-defined range are tested and the method returns true; it is also assumed that the values between the tested values also respect the constraint. There's also a printCorrect-GeneralBinary-Mixture() method that has the isCorrectGeneralBinaryMixture variable as a parameter and prints a message stating that the binary mixture is acceptable if the Boolean parameter is true, otherwise an error message is printed, and the program is halted.

The UserInputs class contains two constructors one being a default and another a copy. The default constructor contains the instances variables storing the values from the input.txt file, which are initialized with

the values of the default case. The methods to read the file are also within the default constructor to overwrite the values of its respective instance variables. It also contains the instance variables to store the values of the calculated parameters by calling the appropriate method. The copy constructor ha an object of type `UserInputs` as a parameter and stores all the values from UserInputs in its own instance variables. However, it also has the `printCorrectGeneralBinaryMixture()` method, which the default constructor doesn't have it. In summary, the `UserInputs` class processes the data from the `input.txt` file, validates the inputted values of the user, compute parameters necessary for the latter height column calculations. The design of the class has been conceptualized to encapsulate its methods from other classes and only the values that were processed by the embedded exception handling can be accessed.

3.2.2. PackingMaterial.class

The `UserInputs` class has a private instance variable of type `PackingMaterial` class, where this class has 3 instance variables being the packing factor (`f_p`), the nominal packing size (`d_p`) and the coefficient of the nominal packing size (`coefficient_d_p`). These are the three variables necessary in the absorber simulation. The `coefficient_d_p` variable and `d_p` variable is necessary to compute the cross-sectional area (`S`) of the column via the `calculateS()` method. This concrete class has three child classes (`BerlSaddles, PallRings, RashigRings`) that inherits its instance variable and `calculateS()` method.

3.2.3. AbsorptionSimulation.class

The `AbsorptionSimulation` class contains the methods and variables to compute the height of the column and associated statistical analysis. To-do so, the height is computed for both the liquid-side and gas-side by integrating the graph shown in **Figure *1*** and ***Figure* 2**. The area under the curve is approximated via numerical methods, where in this program both heights are generated using for the three numerical integration methods. Thus, this class reports the height on the liquid-side, the height on the gas-side, the highest height of the latter and the error percentage for every numerical integration method (three sets of results). The following parameters of these equations presented in appendix 10.4.1 must be recalculated at every iterative step such as the weighted-average molecular weight on the gas side (`g_weightedMW_V`), the weighted-average molecular weight on the liquid side (`g_weightedMW_L`), the gas flow rate (`g_V`), the liquid flow rate (`g_L`), the gas mass velocity (`g_G_V`), the liquid mass velocity (`g_G_L`), the film mass transfer in gas/liquid phase multiplied by the specific area (`g_kya`, `g_kxa`), the liquid fraction at the interface (`g_X_Ai`), the gas fraction at the interface (`g_Y_Ai`) and the log mean of the concentration of the gas/liquid at the interface (`g_LogMeanYA` and `g_LogMeanXA`) .

These computed parameters are all stored in global variables, where the values are updated after every iteration of the Simpson method. The purpose of making these variables global was to reduce the number of local variables needed to be created for their computation due to the procedural dependencies between these variables. It must be noted that these instance variables to approximate the features of a global variable in this class, they are rendered `static`, knowingly global variables are non-existent in object-oriented programming languages such as Java. The necessary values from `UserInputs` class are stored in global variables via accessors such the molecular weights of the substances (`g_MW_A`, `g_MW_L`, `g_MW_G`), the fractions (`g_Y_A2`, `g_Y_A1`, `g_X_A2`, `g_X_A1`), the solute-free flow rates (g_L_f, g_V_f), the surface area of the packing material (g_S), the packing factor (g_F_p), the Schmidt numbers (`g_Schmidt_Gas`, `g_Schmidt_Liquid`), the viscosity of the liquid solvent (`g_ourU_L`), the equilibrium coefficients (`g_A5_EPC` …) and settings for the numerical methods (`g_Root_Finder_X_L`, `g_Root_Finder_X_U`, `g_Root_Finder_Tolerance`, `g_Root_FinderMaximum_Iterations`, `g_Numerical_Integration_Number_Of_Steps`) . As for the other global variables, they are set to zero, but `g_X_AL` is set equal to `g_X_A2` and `g_Y_AG` set equal to g_Y_A1. With these values, the following variables can be computed sequentially: `g_weightedMW_V`, `g_weightedMW_L`, `g_V`, `g_L`, `g_G_V`, `g_G_L`, `g_kya`, `g_kxa`. The computation for the variables needed to compute both heights are done simultaneously. These methods are stored in order of ascending dependency in `setParameters()` method. The following step is to compute `g_X_Ai`, which requires root finding to solve equation 10.16. This equation is segregated to parts and is found in `findX_AI(double x)`. This class implements an interface `RootFinder`, which the `AbsoroptionSimulation` class inherits its method `findX_AI(double x)`. This class also overrides this method via polymorphism to store the equation. This class calls a `RidderSolver` class, which computes the root finding processes for the `findX_AI(double x)` of the interface `RootFinder`. It has a method called `calcualteRoot`, its necessary parameters are: `xLowerBound`, `xUpperBound`, `Tolerance`, `maxIterations` and a parameter of `RootFinder` type. This method is called upon in calculateX_Ai(…) where the parameters are `g_Root_Finder_X_L`, `g_Root_Finder_X_U`, `g_Root_Finder_Tolerance`, `g_Root_Finder_Maximum_Iterations`, and `this` parameter. These global variables containing the setting values obtained by `UserInputs` class. The `this` parameter is used to instantiate the object of `RootFinder` type to an instance of `AbsorptionSimulation` class. Therefore, the object `myRoot` in the `calcualteRoot` uses the overridden version of `findX_AI(double x)` to find `X_Ai`. This architect utilizing both interfaces and polymorphism enables a user to reuse the `RidderSolver` class and the `RootFinder` interface in order classes to solve root finding problems for any given equation. Furthermore, it facilitates the process of implementing alternative root finding methods, where the engineer would be required to recreate a new class this store this new root finding method and simply reuses the `RootFinder` interface. The resulting X_Ai value is stored in its respective global variable by using the mutator

setG_X_Ai(…). With g_X_Ai, g_Y_Ai was computed using calculateY_Ai() which contained the equilibrium equation. Likewise, the g_LogMeanXA and g_LogMeanYA are computed using equation. These parameters enable the calculation of dZV and dZL using equation 10.20 and equation 10.21 respectively. The methods calculateDZV() and calculateDZL() both have the same parameters X_AL and Y_AG. These methods also call the following methods to obtain the values it needs to return dZ: setG_X_AL(X_AL), setG_Y_AG(Y_AG), setParemeters(), calculateX_Ai(),calculateY_Ai(), logMeanYA(), logMeanXA().

Similarly, to the architect use for the root finding of X_Ai, NumericalIntegration interface is implemented in the herein class. This interface contains a single method of Double[] type being calculateHeightDZ(…) with parameters x and y to capture both the gas and liquid fraction at every iteration of the numerical integration computation. This method is overridden in the AbsorptionSimulation class, where a local variable of Double[] type (arrayofDZ) is sized to 2 ; arrayofDZ[0] and arrayofDZ[1] which store respectively calculateDZL(x, y) and calculateDZV(x, y). By doing-so, the infinitesimal height for the liquid-side and gas-side are computed simultaneously, when the overridden version calculateHeightDZ(…) is called by the solvers. The three numerical integral solvers are TrapezoidSolver, SimpsonSolver and ModifiedSimpsonSolver. These classes contain each a method to compute the numerical integration respectively being: calculateNumerical-IntegrationSimpson(…), calculateNumerical-IntegrationTrapezoid(…) and calculateNumericalIntegrationModifiedSimpson(…). These methods are invoked in three methods being: calculateHeightSimpson(), calculateHeightTrapezoid() and calculateHeightModifiedSimpson(). In all these methods they have the same signature being: numberofPhases, numberofSteps, xLowerBound, xUpperBound, yLowerBound, yUpperBound, myHeight. Since the heights are calculated for both the liquid-side and the gas-side, the numberofPhases are set to 2. The numberofSteps is obtained from g_NumericalIntegration_Number_Of_Steps. The boundary parameters are set by g_X_A2, g_X_A1, g_Y_A2, g_Y_A1. The last parameter myHeight is of type NumericalIntegration, where the inputted parameter is this as well. Similarly, myHeight is instantiated by AbsoprtionSimulation class through the this parameter and these methods will return both heights for each numerical integration. To compute the error between both heights, each numerical integration has an associated method to calculate the percentage of error being: calculateHeightErrorPercentageSimpson(), calculateHeightErrorPercentageModifiedSimpson(), calculateHeightErrorPercentageTrapezoid(). The printNonOptimizedResults() is implemented in this class to print the results from every solver including the errors to the console. In the constructor of the AbsorptionSimulation class,

the following methods are called: `calculateHeightSimpson()`, `calculateHeightErrorPercentageSimpson()`, `calculateHeightTrapezoid()`, `calculateHeightErrorPercentageTrapezoid()`, `calculateHeightModifiedSimpson()`, `calculateHeightErrorPercentageModifiedSimpson()` and `printNonOptimizedResults()`. This class was designed so that when the its invoked in the `SimulationEnvironnement` class; it obtains the appropriate data from the UserInputs class, additional numerical methods can be easily incorporated and all the methods to compute and print the results are executed.

3.2.4. Optimization.class

The optimization class was designed to search for the optimal height and corresponding solute-free liquid flow rate. To optimize these design parameters for the gas absorber, an array of liquid flow rate must need to be generated. The `LiquidFlowRateArray(…)` method execute this task and has a single parameter being `myUserInputs` which is of type `UserInputs`. It calls the minimum liquid flow rate and number of steps via accessors of the `UserInputs` class. These values are stored in local variables namely: `numberofSteps()` and `minimumLiquidFlowRate()`. The maximum liquid flow rate is by multiplying the V_f by 10. This will ensure a broad range of liquid flow rates would be tested for the optimization and the maximum flow rate will be by default relative to the amount of gas to be processed by the absorber column. The step size can be calculated using the number of steps, the minimum and maximum liquid flow rates.

A local variable of `Double[]` type being array `arrayLiquidFlowrateArray` is sized to the number of steps and via a for-loop is loaded with the list of the liquid flow rates to test. The results are generated and stored in a two-dimensional array called `OptimizedArrayResultsRangeofLiquidFlowrate(…)` with two parameters being `myUserInputs` which is of type `UserInputs` and `myAbsorptionSimulation` which is of type `Absorption-Simulation`. A local variable of Double[] type `myLiquidFlowrateArray` stores the list of liquid flow rates to test and its length is used to size the first dimension of the array `myOptimizedArrayResults-RangeofLiquidFlowrate`. Its second dimension is set to 15 representing the number of data points stored for every liquid flowrate iteration. This method only contains a single for-loop, where it loops thorough the first dimension of the array being the liquid flow rates. Within the loop, the `myAbsorptionSimulation` object invokes the mutator of `L_f`, where `L_f` is changed at every loop via `myLiquidFlowrateArray[i]`. The value of `X_A2` is obtained in a local variable being `myX_A2`. This is necessary to compute the new `L_2` at every iteration if `X_A2` is not equal to zero. The methods to compute the 15 data points are hard coded in the for-loop. At every loop, the iteration is stored, the current solute-free liquid flow rate, the inlet liquid flow rate and for every numerical integration method the height on the liquid side, the height

on the vapor side, the largest height of the latter as well as the error percentage of the heights are stored. Once the for-loop is complete, the method returns the loaded 2D array.

To search for the optimal solution, the `indexofMinimumErrorPercentage(…)` was designed to scan the previously described 2D array and return at which iteration was the optimal solution obtained for every numerical integration method. To find the optimal solution, the iteration that stored the smallest error percentage was deemed the most optimal. In a for-loop nested with if-statements, the 2D array was scanned for its full length of liquid flow rates. Local variables were coded to store the minimum liquid flow rates and the iteration corresponding to the optimal case. These variables were initialized with the iterations and liquid flow rate of the first liquid flow rate data set stored in the previously described 2D array. The if-statements contained one constraint being if the error percentage at any given iteration is smaller than the current minimum error percentage, the current minimum error percentage and iteration local variables would be update for a given numerical integration method. Otherwise, the for-loop will continue to the next iteration. Once the full range of liquid flow rates have been scanned, a local variable of `Double[]` type `arrayIndexofMinimumErrorPercentage` stores the iteration containing the optimal results for all 3 numerical integration methods. The `indexofMinimumError-Percentage(…)` method returns this list.

The `summaryofOptimizedResults(…)` has a single parameter being `OptimizedArrayResults-RangeofLiquidFlowrate(…)`. This method captures the list of optimal iterations in a local variable of `Double[]` type being `myArrayIndexofMinimumErrorPercentage` and a 2D array to store the full results in `myOptimizedArrayResultsRangeofLiquidFlowrate`. The iterations are segregated and stored in the following local variables of `Double` type: `indexOptimizedTrapezoid`, `indexOptimizedSimpson` and `indexOptimizedModifiedSimpson`. A local variable of `Double[][]` type being `mySummaryOptimizedResults` is sized to 3 for the first dimension representing the number of numerical integration methods and 7 for the second dimension representing the data points to be stored. These data points include the optimal iteration, the corresponding optimal inlet liquid flowrate, the height of the column on the liquid-side, the height of the column on the gas-side, the largest height of the latter and the corresponding error percentage. These values are loaded to the `mySummaryOptimizedResults` by extracting the necessary data points from `myOptimizedArrayResultsRangeofLiquidFlowrate` using the optimal iteration for every numerical integration method. This loaded array is returned by the `summaryofOpti-mizedResults`. The `printSummaryOptimizedResults` method is similar to the `printNon-OptimizedResults()` and prints the results of `mySummaryOptimizedResults` as a decimal formatted list to the console. This class contains a single constructor with two parameters being `myAbsorption-Simulation` and `myUserInputs` of type `AbsorptionSimulation` and `UserInputs`. Inside the

constructor, all of the methods of this class are invoked, and they are all executed in the `Simulation-Environnement` class when this `Optimization` class is instantiated.

3.2.5. UserOutputs.class

The `UserOutput` class was designed to output the results from both the `Optimization` class and `AbsorptionSimulation` class to an Excel file. It consists of a single default constructor, three private instance variables (`resultsNames`, `resultsValues`, `filepathoutputs`). When this class is invoked in the `SimulationEnvironnement` class, an object of `UserOutputs` class is created, the constructor receives different objects that are used by the implemented concrete method in the Output class with the help of the private instant variable. Jxl library is used in the output to connect java and excel for making java output in an excel file. Therefore, the `UserOutput` class consist of a constructor that is responsible for receiving object parameters, a instant variable responsible and method that is responsible of getting the values in a single array, thus printing the array in the excel with the help of the jxl library that connects java and excel.

Initially, the constructor of `UserOutputs` assigns the global variable `filepathOutput` to the `filepathOutput` that was initiated in the beginning of the inputted Scanner function. This to make sure that the file path of the output is ready before execution of the code. After that the constructor goes through a method called `getResultNames()` that is responsible in initiating the String values of the `Double[] resultsName` global variables. The `resultsName` global variable will serves as the heading subtitle to print the outputted variables. This is mainly because, the output can be either hardcoded or it can be specified by the UserOutputs constructor. After that the constructor accepts objects, `AbsorptionSimulation` object, an `Optimization` object and `UserInput` object. All three objects are passed into a method `getResultsValues` that gets the calculated optimized results from both AbsorptionSimulation and Optimization classes and return it to be stored in the instant variable `double[] resultValues` array in the constructor. This is mainly because the `resultValues` array gets all the computed values, and make sure that it matches with the String values in the excel. This is done by printing the `resultName` and the `resultValues` array in order using jxl library. After that the constructor calls the printPart1() that is responsible connecting with jxl library with java to print `resultName` array and `resultValue` array. Therefore, the constructors of the code serve as a major part in getting the values from both `Optimizaiton` and `AbsorptionSimulation` objects to be outputted in the specified file path.

The `printPart1` method is a concrete method that is implemented inside the UserOutput class that is responsible for printing both resultsNames and resultsValues that were calculated and passed by the Optimization and `AbsorptionSimulation` objects. First, java opens a `WriteableWorkbook` that gets assigned to a null

value. This null value is essential since it can be used in the exception handling if the User has forgotten to initialize the `WritableWorkbook` instance variable. The `Writablebook` is then initialized to create a Workbook in the `newFile` path that was specified in the beginning by the user. The new file path must point to output file of type xls. Only xls would be possible for an output since xlsx extension file is not readable by java excel library. Another catch is implemented to see whether the `filepath` points to an existing `filepath` that is founded in the hardrive. After a workbook is created in the excel, a sheet is created to a zero excel sheet. A zero excel sheet represents the excel sheet that can be created by the workbook. After creating an excel sheet, a for loop is used to print out all the `resultName` array and the `resultValue` array. The `resultName` array is used to print the String values that are represents the subtitle heading of the optimized values in the array. After that the code will create a label for a string value that will be Instantiate to a new label that has (number of columns, number of rows, values) as the constructor number. The structure of arrays in excel is the inverse of the structure of array in java when dealing with a 2D array. Thus, a for loop is made to copy all the results in the String value to be copied in the excel. The same logic applies when printing a number value, except for values an object of Number is creating to instantiate so that it can print the results in (columns, 1 = number of rows, `numbervalue`). After that the workbook is closed this would not allow any further modification. Therefore, the printPart1 creates a `WorkBook` that initializes an `excelsheet` to zero making two objects one object is responsible for printing the String which is called Label and the other object is called Number object that is used in printing the `resultNames` and `resultValues`. The constructor of the `outputArray` accepts 3 different objects, UserInput, Optimization and AbsorptionSimulation object. These different objects are passed to the x which assigns the global variable array `resultValue` values of optimization. Also, the same thing occurs in the `resultNames` where the constructor calls a method to get an array of all the names that are required to be printed in the excel. After that the array are stored in a private instance variable that can be accessed by printPart1 method that is responsible for printing the array to the specified output file path.

3.3. Mathematical Modelling

Mathematical modelling and simulation were carried out on JAVA programming language to design an absorption column. The numerical methods were required to solve the non-linear design equations using static methods instead of instance method. Numerical methods were coded in a versatile manner so that it can be implemented in different situations or design problems without being instantiated as object. The robustness of the code increases as the static method regarding the numerical method could be called outside the driver or simulator class to solve any nonlinear function without any modification or change to the code. This static method have local variables instead of instance variable, and local variables are not required to be saved in the memory stack. The local variables are only needed temporarily and utilized

when that specific function is called upon. In addition to numerical method, numerical integration are also implemented in the design simulation. Numerical integration allows to replace tabulated data points and complex function with an approximation function which is simple to integrate [4].

3.3.1. Numerical method for non-linear equation

The purpose of numerical method in this design is to solve for root of complex polynomial function of equilibrium data provided by the user for any binary mixture. In this design numerical root finding technique known as Ridders method was implemented. A class called `RootFinder` was implemented as interface. The `RootFinder` had abstract method `findX_AI` which would take an input variable x. Any class that would try to utilize the advantage of this `RootFinder` interface would have to implement `Rootfinder` and have a concrete method `findX_AI` . The class could pass itself to the Rootfinder using this keyword in the constructor. In short, RootFinder interface uses multiple inheritance, and take advantage of polymorphism to apply root finding methodology for any nonlinear equation.

`RootFinder` interface class plays an essential role for solving the interface composition `Xa_i`. The interface composition is found from a non-linear equation derived from a line that passes perpendicularly from the operating line with a negative slope and intersects the equilibrium line. The interface composition is found from the coordinates in the equilibrium line from a given y-coordinate and a x-coordinate from the operating line. This non –linear equation is developed inside the concrete method `findX_AI` in the `Absorption simulation class`. The non-linear equation is then passed as an object of the `RootFinder` interface to the numerical method to solve for the `Xa_i` interface composition.

3.3.2. Ridders' Method

The `RidderSolver` class has a static method called `calculateRoot` which will accept lower bound, upperbound, tolerance, maximum iteration and an object of RootFinder. The object of RootFinder plays an important role for this method. The advantage of interface was utilized here. The `findX_AI` method in the `RootFinder` is overridden by the `findX_AI` method in the AbsorptionSimulation class; the non-linear equation present in the `findX_AI` method in the absorption simulation class is passed to the `calculateRoot` method. Since the non-linear equation is passed to the `calculateRoot` method, there is no need to code a non-linear equation in the `calculateRoot` method resulting the code to be

object oriented and very efficient and robust as it can be applied to any non-linear equation in any other class in a very similar manner.

Using the lower bound (`XL`) and upper bound (`XU`), value of midpoint (`XM`) is calculated using the static `calculateXM` method. An estimate of the root, `XR`, was calculated using the static `calcualteXR` method. In this method, the non-linear function is called by using `myRoot.find_XAI` to find values at lower bound, upper bound and midpoint `XM`. The calculate `XR` method returned a value of `XR`. Before entering the Do while loop, the variable `error, numberofIterations,` and `newXR` were initialized to 1.00, 0 and 0 respectively. Within the Do while loop, the algorithm checks for the root by using if and elseif statement between the bounds and condition provided. ***Figure 6*** depicts the conditions:

The conditions were as follows:

1. If XR was less than XM, and product of the non-linear function in the findX_AI method of the absorption column at XU and XR was less than 0, then XU was set equal to XR.
2. If XR was less than XM, and product of the non-linear function in the findX_AI method of the absorption column at XR and XM was less than 0, then XL was set equal to XR, and XU was set equal to XM.
3. If XR was less than XM, and product of the non-linear function in the findX_AI method of the absorption column at XM and XU was less than 0, then XL was set equal to XM.
4. If XR was greater than XM, and product of the non-linear function in the findX_AI method of the absorption column at XU and XR was less than 0, then XU was set equal to XM.
5. If XR was greater than XM, and product of the non-linear function in the findX_AI method of the absorption column at XR and XM was less than 0, then XL was set equal to XM, and XU was set equal to XR.
6. If XR was greater than XM, and product of the non-linear function in the findX_AI method of the absorption column at XM and XU was less than 0, then XL was set equal to XR.
7. If none of the above conditions is satisfied, XU would be returned.

For instance, if the first condition was satisfied, the upper bound would be XR, and the lower bound would stay the same XL. Then the `newXR` variable would be calculated using `calculateXR` method which would use XL as lower bound and XR as upper bound. The error variable would be calculated using newXR , and the XR previously found in the beginning of the Do while loop using `calculateXR`

method . If the value of error is greater than the provided tolerance, the loop would be repeated until the error is less than the tolerance, and at that point the XR is set equal to the `newXR`. Then this updated XR is returned, and this is the root of the non-linear equation. For this design, this value of XR is the value of the interface composition X_AI.

3.3.3. Numerical Integration

In this design project, one of the main objectives was to find the height of the tower based on the vapour side and the liquid side. For each coordinate along the operating line, interface composition X_AI, and Y_AI were computed, and using this value in the design calculation differential tower height for liquid and vapor side were calculated. The differential tower height for liquid was plotted as a function of mass fraction, x, and differential tower height for vapor were plotted as a function of mole fraction in ***Figure 1*** and ***Figure 2*** respectively. The area under the curve is the total height of the tower. At this point it is very crucial that numerical integration techniques were used. Various numerical integration techniques such as trapezoid rule, Simpson's rule and modified Simpson were used to find the tower height. The results were then compared to see which technique produced the height with less error within in a given tolerance. The purpose of implementing all these techniques were to allow flexibility to use any one of the techniques based on the shape of the curve. Similar to numerical method, interface was utilized for numerical integration. The purpose was to make it more versatile and flexible so that it could be called to solve any numerical integration outside the main simulator. The `NumericalIntegration interface` had an abstract method called `calculateHeightDz` which accepts two input variable x and y. The method returned an array. Through an array, tower height for both liquid side and vapor side could be returned, which made the code more robust and efficient.

3.3.4. Trapezoid, Simpson's and Modified Simpson's

Numerical integration for linear function works better using trapezoid rule [4]. For the purpose of this design, a class called `TrapezoidSolver` was made. The method `calulateNumerical-IntegrationTrapezoid` was made a static method and it returned an array. In this array both the height for liquid side and the vapour side could be returned. This static method would receive an input of number of phases, numberofSteps, lowerbounds for both the vapor and liquid, upperbound for both the liquid and the vapor, and finally an object of `Numerical Integration`, which is again utilizing the

advantage of interface. All this input are not hard coded for the purpose of making this code more flexible and robust, so that the user can modify any of the inputs in the future while trying to use this `TrapezoidSolver` class for other problems.

The step size for both the vapor and liquid were computed based on the upper bound, lower bound and the number of steps passed to the method. A local array named as sum, and a local array called heights were sized equal to value of number of phases. For our design purpose there are two phases, so the size of the sum and height array would be two elements. Then a for loop is coded, and the objective of this for loop is to store the value for the first interaction and last iteration for the liquid phase and the vapor phase in sum[0] , and sum[1] respectively. Then using the second for loop all the iterations without the first and last iterations were calculated. Then all the values of iterations for the liquid was stored in array sum [0], and for vapor sum [1]. Using a step size in x, and y, new x and y values were calculated, and the loop was repeated. The height of tower for liquid side was stored in the array Heights [0], and Heights [1] was used to store the height of the tower height for vapor side. Finally, the heights were returned. As explained before, this trapezoid solver accepts an object of `NumericalIntegration myHeight`. Whenever a concrete method for `calculateHeightDZ` is present and passed inside the static method of the `TrapezoidSolver` class, the `calculateHeightDz` in the `NumericalIntegration` would be overridden.

In a similar manner to trapezoid, Simpson's and modified Simpson's were coded in the `SimpsonSolver,` and the `ModifiedSimpsonSolver` respectively. The only modification for the Simspson solver was using the equation EQ6 for simpson alogrithm, and in case of ModifiedSimpsonSolver equation EQ7 was used. The purpose of using all these three methods was to check error between the three techniques and allowing the user to choose the value of the height with lower error. When the function is a curve instead of a linear function, simpson's alogrithm provides more accurate value for the integral. The simulator that is coded is not necessarily limited to the default equilibrium data provided, and the user can use a different binary mixture which may result a complex polynomial function resulting the shape of the curve in figure 1 and 2 be more non-linear or polynomial. Complex polynomial model could be integrated with modified Simpson with more accuracy. Overall, all these numerical integrations considered for this design was to increase the robustness and efficiency of the code to provide better result with more accuracy.

4. Validation

The validation expedition of the program was undertaken to ensure that the code was meeting the requirements of the project. It was precisely conducted to confirm the exception handling works as expect, all methods and classes operate as prescribed and the goals of the projects are met. In this validation campaign, the process can be separated into two parts being the arithmetic operations (Part 1) and the height computation (Part 2). The output values of the final iteration from the code were validated with calculations that were carried out in an Excel spreadsheet. The Excel spreadsheet that was designed for this section carried the calculation process that has been described in the Background section of the report. With simple data linking and formula functions, the spreadsheet was designed very similarly to the Java code to calculate absorption column heights. Furthermore, it used macros and Visual Basic to undergo a iterations for trapezoidal and Simpsons methods for the integration for the total height. For both the Java an Excel simulation, the tolerance of the Ridder's method was set to 10^{-6}, the maximum number of iterations for the Root finder was set to 1000 and the number of steps for the numerical integration was set to 1000. These numerical methods remained consistent throughout the validation campaign. For this campaign three cases are tested found in table 1-3.

Table 1. Base case of imputed values provided by client for gas absorption simulation.

Input Variable	Value	Units
V_1	15	kmol/h
Y_{A1}	0.12	-
L_2	40	kmol/h
X_{A2}	0	-
Recovery	92	%
Packing	Raschig Rings	-

Table 2. Second case of inputted values for gas absorption simulation.

Input Variable	Value	Units
V_1	21	kmol/h
Y_{A1}	0.23	-
L_2	65	kmol/h
X_{A2}	0	-
Recovery	98	%
Packing	Beer Saddles	-

Table 3. Third case of inputted values for gas absorption simulation.

Input Variable	Value	Units
V_1	22	kmol/h
Y_{A1}	0.18	-
L_2	50	kmol/h
X_{A2}	0	-
Recovery	95	%
Packing	Pall Rings	-

4.1. Iteration Arithmetic Calculations

4.1.1. Calculated Values in `UserInputs`

4.1.1.1. Solute-free Flow Rates

This class is responsible for using the input variables from the .txt file and pass along the initial key parameter values. Through the application of appropriate mass balances and scanning methods, it dictates the inlet and outlet conditions for the vapour and liquid streams of the absorption column.

Table 4. Validation table for the free-solute flow rate from `UserInputs` class.

	`L_f`			**`V_f`**		
Case	Excel	Java Solution	%Error	Excel	Java Solution	%Error
Case 1	40	40	0	13.2	13.2	0
Case 2	65	65	0	16.17	16.17	0
Case 3	50	50	0	18.04	18.04	0

Table 4 shows the values of the free-solute fluid flow rates that were assumed to be constant throughout the absorption column. These values were calculated in `UserInputs` class and passed through the iterative process in `AbsorptionSimulation` class to calculate the flow rates according to the mass balance for every portion of the dissected mass transfer model. It was essential that the values calculated displayed minimal error from the Excel simulation, as seen in the table, since they dictated the mass balance. Therefore, validating and calculating the values of `L_f` and `V_f` was the priority of the design of the code.

4.1.1.2. Operating Line

The operating line was calculated in the `UserInput` class for the sake of validating the general binary mixture. If the operating line at any point of the process produced a value below the equilibrium line as set in the

`UserInput` class as well, an exception was placed to warn the user to use a different binary mixture as it would not be possible to operate the absorption process.

Table 5. Validation table for operating line from `UserInputs` class.

	SlopeofOperatingLine			**YinterceptofOperatingLine**		
Case	Excel	Java Solution	%Error	Excel	Java Solution	%Error
Case 1	2.747098333	2.747098321	4.17142E-07	0.010791367	0.010791367	1.28601E-13
Case 2	3.300921082	3.300918323	8.35838E-05	0.005938549	0.005938549	1.46056E-14
Case 3	2.417196819	2.417197061	1.00146E-05	0.010856454	0.010856454	2.71638E-13

Table 5 shows the values of the slope and intercept of the operating line that was produced by the Java simulation in comparison to the Excel calculations. They were calculated based on the inlet fluid compositions and the calculated outlet compositions. The points were set to be (`xA_1`, `yA_1`) and (`xA_2`, `yA_2`). The error percentage for these values ranged from 8.36E-05% to 1.46E-14% which proved that the Java code was consistent with the Excel calculations.

4.1.1.3. Mole fractions (x_{AL} and y_{AG})

Table 6. Validation table for the mole fractions from `UserInputs` class.

	xA_L (calc)			**yA_G (I/O)**		
Case	Excel	Java Solution	%Error	Excel	Java Solution	%Error
Case 1	0.039754177	0.039754177	4.1714E-07	0.12	0.12	0
Case 2	0.067878463	0.06787852	8.35838E-05	0.23	0.23	0
Case 3	0.069975082	0.069975075	1.00146E-05	0.18	0.18	0

Table *6* 6 presents the solutions for the calculated and inputted values of the mole fractions of A in the vapour and liquid streams. Since the validation procedure was implemented for the final iteration to find the height, the values correspond to the outlet mole liquid fraction and inlet vapour fraction, i.e, at the bottom of the absorption column. In this case, the liquid fraction was calculated, and the vapour fraction was based on File I/O. The calculation error percentage ranged from 1.00E-05% to 4.17E-07% which proved that the implementation of the mass balance in the `UserInput` method have accurate results relative to the Excel values. In addition, the File I/O system was also able to record the inlet conditions properly.

4.1.2. Calculated Values in AbsorptionSimulation

The `AbsorptionSimulation` method is used to calculate the iteration variables that numerically represent the mass transfer of A from the vapour stream to the liquid stream via absorption. It has been utilised to calculate this transfer between each iteration of the height of absorption column.

4.1.2.1. Liquid and Vapour Flow rates

Table 7. Validation table for of flow rates from `AbsorptionSimulation` class.

	L			**V**		
Case	Excel	Java Solution	%Error	Excel	Java Solution	%Error
Case 1	0.011571111	0.011571111	1.72695E-08	0.0041667	0.004166667	3.747E-13
Case 2	0.019370388	0.019370389	6.0867E-06	0.0058333	0.005833333	2.97381E-14
Case 3	0.014933889	0.014933889	7.53497E-07	0.0061111	0.006111111	3.83216E-13

Table 7 shows the comparison for the flow rates for the outlet liquid and inlet vapour streams. This was calculated in the `AbsorptionSimulation` method from the `L_f` from `UserInputs` that was assumed to remain constant throughout the column by design, and the changing `xA_L` and `yA_G` mole fractions with height. The calculation error ranged from 6.09E-06% to 1.73E-08% for `L` which proved that the implementation of the mass balance in the `AbsorptionSimulation` method was accurate. Along with the slight deviations from approximations for `V`, it can be observed that `UserInputs` had successfully linked the required objects to `AbsorptionSimulation`.

4.1.2.2. Average Molar Weights of Fluid Streams

Table 8. Validation table for molar weights from `AbsorptionSimulation` class.

	MW_L			**MW_V**		
Case	Excel	Java Solution	%Error	Excel	Java Solution	%Error
Case 1	19.05480122	19.05480123	2.26532E-08	29.97	29.97	4.26752E-13
Case 2	19.7868764	19.78687788	7.46365E-06	31.73	31.73	1.0077E-13
Case 3	19.84145139	19.84145121	9.19341E-07	30.93	30.93	3.44589E-13

Table 8 compares the average molar weights calculated within `AbsorptionSimulation` based on the molar fractions and the variables of the molar weights for A, liquid phase, and vapour phase passed on from `UserInputs`. The calculation error ranged from 7.46E-06% to 2.27E-08% for calculated `MW_L`, and approximation deviations for `MW_V`.

4.1.2.3. Mass Fluxes, G_L and G_V

Table 9. Validation table for mass flux from `AbsorptionSimulation` class.

	G_L			G_V		
Case	Excel	Java Solution	%Error	Excel	Java Solution	%Error
Case 1	0.499076451	0.499076451	3.9923E-08	0.282659179	0.282659179	6.87361E-13
Case 2	3.470268976	3.470269446	1.35503E-05	1.67584732	1.67584732	2.11995E-13
Case 3	6.036378419	6.036378318	1.67284E-06	3.850615914	3.850615914	4.38252E-13

Table *9* contains the validation table for the mass velocities that have been calculated in `AbsorptionSimulation` from the product of fluid flow rates and molar weights per unit surface area of the packing material. This surface area is calculated in `PackingMaterial` based on the selection that was made by the user and called in `UserInputs`. The mass velocities that were calculated had an error percentage that ranged from 2.12E-13% to 1.36E-05%, indicating that the surface area of the packing material that are calculated have minimal deviations from the Excel calculations.

4.1.2.4. Mass Transfer Coefficients, $k'_x a$ and $k'_y a$

Table 10. Validation table for mass transfer coefficients from `AbsorptionSimulation` class.

	k_xa			k_ya		
Case	Excel	Java Solution	%Error	Excel	Java Solution	%Error
Case 1	0.143570118	0.143570118	5.29246E-09	0.020462429	0.020462429	1.99611E-08
Case 2	0.586906018	0.58690603	2.02159E-06	0.177028567	0.177028579	6.77517E-06
Case 3	0.57339268	0.573392679	2.51645E-07	0.27350978	0.273509778	8.36419E-07

Table 10 shows the calculated values of the mass transfer coefficients in both the liquid and the vapour phases from the `AbsorptionSimulation`. Unlike the prior validations, these values have a higher error range due to their dependence on other calculated parameters in addition to input variables. The mass transfer coefficients are calculated from a complex equation connecting the surface area calculated from `PackingMaterial` and packing factor of the packing material, the calculated Schmidt Numbers and viscosities for both liquid and vapour phases from `UserInputs`, and the mass velocities from `AbsorbtionSimulation`. The error percentage ranged from 5.29E-09% to 6.78E-06%, which proved that code could efficient function to carry out calculation procedures that linked several classes very efficiently.

4.1.3. Calculated Values at Interface, `AbsorptionSimulation` and `RidderSolver`

Henceforth, the validation section will progress into finding the interphase equilibrium points and manipulating the data to predict the height of the absorption column based on the mass transfer between phases.

4.1.3.1. Log-mean difference between bulk and interface concentrations, $(1-x_A)_{iM}$ and $(1-y_A)_{iM}$.

Table 11. Validation table for log-mean differences between bulk and interface concentrations from `AbsorptionSimulation` class.

	`logMeanXA()`			**`logMeanYA()`**		
Case	Excel	Java Solution	%Error	Excel	Java Solution	%Error
Case 1	0.952274597	0.952274596	3.76122E-08	0.9336129	0.933612915	8.50902E-09
Case 2	0.902119955	0.902119899	6.29123E-06	0.8604307	0.860430654	1.79284E-06
Case 3	0.897390227	0.897390233	6.9036E-07	0.8849992	0.884999247	1.78389E-07

Table 11 shows the log-mean difference between the bulk concentrations (`xA_L`, `yA_G`) and the interface concentrations (`xA_i`, `yA_i`). The error percentage ranged from 8.51E-09% to 6.29E-06% depending on the case being tested.

4.1.3.2. Slope between bulk and interface concentrations, $M(x_{Ai}, y_{Ai})$ and $P(x_{AL}, y_{AG})$

Table 12. Validation table for slope between PM from AbsorptionSimulation class.

	Slope of PM, m		
Case	Excel	Java Solution	%Error
Case 1	-6.878781565	-6.878781566	1.44345E-08
Case 2	-3.162108741	-3.162108733	2.55188E-07
Case 3	-2.067477775	-2.067477777	7.28022E-08

Table 12 shows the calculated values of the slope of the line that connected the interface and the bulk composition points. This value depended on the interface and bulk compositions in the `AbsorptionSimulation` and the gradient between these points. The table displays an error in the range of 1.44345E-08% to 2.55188E-07% which can be attributed to the approximation in the root finder as these were calculated at the actual computed interface compositions.

4.1.3.3. Intercept of PM

Table 13: Validation table for y-intercept for the line PM from `AbsorptionSimulation` class.

	Intercept of PM, b		
Case	Excel	Java Solution	%Error

Case 1	0.393460298	0.3934603	6.57331E-07
Case 2	0.444639066	0.444639261	4.38962E-05
Case 3	0.324671927	0.324671913	4.41909E-06

Table 13 shows the calculated values of the intercept of the line that connected the interface and the bulk composition points. This value depended on the interface and bulk compositions in the `AbsorptionSimulation` and the gradient between these points. The table displays an error in the range of 6.57E-07% to 4.39E-05%, indicating that the Java code can calculate the parameters of PM to compute the interface compositions.

4.1.3.4. Determining Interfacial Compositions using Ridders Method

Table 14: Validation table for root-finding to determine interface point from `RidderSolver` class.

	f(xA_i)=0		
Case	Excel	Java Solution	%Error
Case 1	-1.40616E-09	-7.97973E-17	-99.99999433
Case 2	-1.63857E-08	-6.16834E-11	-99.6235536
Case 3	-7.03681E-11	-3.49731E-11	-50.29969614

Table 14 shows the roots that were calculated using the Ridder's method for the numerical root-finding process in `RidderSolver`. The Ridder's method was used to vary the `xA_i` value in between xA_L_n and xA_L_{n+1} for the same equilibrium relation using both Java and Excel. The Excel had values that were almost twice as large as the roots that were determined using java. Although the tolerances were set to the identical values, Java proved to be substantially adept at determining the equilibrium `xA_i` values that satisfies the interfacial compositions of the system.

4.1.3.5. Interfacial Compositions, x_{Ai} and y_{Ai}

Table 15: Validation table for interfacial compositions for the liquid and vapour phases from `RidderSolver(xA_i)` and `AbsorptionSimulation (yA_i)`.

	xA_i			**yA_i**		
Case	Excel	Java Solution	%Error	Excel	Java Solution	%Error
Case 1	0.055652393	0.055652394	9.85341E-07	0.0106396	0.010639639	1.55218E-06
Case 2	0.127230853	0.12723091	4.45903E-05	0.0423213	0.042321291	7.82956E-05
Case 3	0.134471948	0.134471943	4.02727E-06	0.0466542	0.04665416	7.11198E-06

Table 15 shows the xA_i that produced the root for function of the equilibrium interface concentrations from RidderSolver. yA_i values were then calculated in AbsorptionSimulator from the xA_i according the given phase equilibrium relation in UserInputs. The error between the values of xA_i ranged from 9.85341E-07% to 4.45903E-05%, while yA_i ranged from 1.55218E-06% to 7.82956E-05%, which can be attributed to the deviations in approximations throughout the numerous calculations and iterations.

4.1.3.6. Calculated Height Differential from AbsorptionSimulation

Table 16. Validation table for the height differentials for liquid and vapour phases from AbsorptionSimulation class.

	dZ_V			dZ_L		
Case	Excel	Java Solution	%Error	Excel	Java Solution	%Error
Case 1	4.471398776	4.471398781	1.22542E-07	11.379678	11.37967752	2.41452E-06
Case 2	1.776360452	1.776360613	9.08753E-06	4.8727135	4.872713653	3.86521E-06
Case 3	3.684062668	3.684062614	1.47349E-06	7.9377792	7.937779	3.03357E-06

Table 16 shows the final value for the iteration process that provides the portion of the height of the column that is required for the associated mass transfer to occur. It depends on the bulk concentrations of the fluids streams at that position and their respective interface compositions that was obtained from RidderSolver and the equilibrium relation. This was calculated in the AbsorptionSimulation method and stored in an array called arrayofDZ. This array containing the fractions of the total height of the absorption column was the final iterative value in the class. This was the final parameter that had to be calculated before the numerical integration methods were used to calculate the total and optimized height for a given system. For the final iteration, the error percentages in the height differentials for both the liquid and the vapour phase varied from 1.23E-07 to 9.09E-06, proving that the absorption column designed for this project was accurate according to the Excel simulation.

4.2. Calculated Heights using Numerical Integration Methods

Table 16: Outputted total liquid phase heights of absorption column using `TrapeziodSolver`, `SimpsonSolver` and `ModifiedSimpsonSolver`.

	Z_L					
	Trapezoidal			**Simpsons**		
Case	Excel	Java Solution	%Error	Excel	Java Solution	%Error
Case 1	1.211940816	1.20859515	0.276058549	1.2119367	1.209404375	0.208948523
Case 2	1.288825301	1.281603745	0.560320803	1.2887848	1.283749956	0.390665081
Case 3	1.549373337	1.544210226	0.333238651	1.5493524	1.545538642	0.246153821
	Modified Simpsons					
Case 1	1.206575894	1.210162259	0.297234929			
Case 2	1.275771322	1.284126807	0.654935935			
Case 3	1.540866868	1.546460078	0.36299111			

Table 17. Outputted total liquid phase heights of absorption column using `TrapeziodSolver`, `SimpsonSolver` and `ModifiedSimpsonSolver`.

	Z_V					
	Trapezoidal			**Simpson**		
Case	Excel	Java	%Error	Excel	Java	%Error
Case 1	1.17736811	1.1712592	0.51886221	1.17736435	1.17296594	0.37358091
Case 2	1.16990220	1.16349301	0.54784029	1.16986882	1.16533053	0.38793127
Case 3	1.47691128	1.47186842	0.34144610	1.47690368	1.47312626	0.25576608
	Modified Simpsons					
Case 1	1.172329864	1.175693424	0.286912477			
Case 2	1.158931281	1.165959636	0.606451481			
Case 3	1.469202894	1.474298216	0.346808614			

Table 17 and 18 represent a tabulated output for the total integrated height of the absorption column using three different methods of numerical integration, Trapezoidal method, Simpsons method and Modified Simpsons method. These values were iterated through the methods called `TrapezoidSolver`, `SimpsonSolver` and

`ModifiedSimpsonSolver` methods respectively with loop through `AbsorptionSimulation` to use `calculateDZ` while stepping through the operating line for the fluid compositions. In terms of the error to signify the inconsistencies from the Excel Simulation, Trapezoidal method had an error range from 0.56% to 0.27%, Simpsons method had an error range from 0.21% to 0.39%, and Modified Simpsons method had an error range from 0.61% to 0.29%. This was expected as Simpsons method for numerical integration was favoured for the extent of linearity of the differential steps through the mass transfer for this absorption column design specifications. It was also expected that Trapezoidal method would produce values that were less accurate than Simpsons method as it assumed the over-linearity of the differential functions. And finally, Modified Simpsons method performs optimally with cases of local maxima's and minima's by being thorough with the extremities of a differential function; as this was not the case for this design, Modified Simpsons had a high range of error. Therefore, the most accurate results for the height of the absorption column for this project were produced by employing the Simpsons Method for numerical integration.

5. Simulation Results

Base Case

Input

An acceptable general binary mixture has been inputted.

- - - Part 1: Non-opmitized Validation Section - - -

[Non-Optimized] Height Liquid [Trapezoid] (m): 1.2085262283030616
[Non-Optimized] Height Gas [Trapezoid] (m): 1.1739524250058786
[Non-Optimized] Height Error Percentage [Trapezoid] (%): 2.860823578957767
[Non-Optimized] Height Liquid [Simpson] (m): 1.209358427571381
[Non-Optimized] Height Gas [Simpson] (m): 1.174761432679566
[Non-Optimized] Height Error Percentage [Simpson] (%): 2.8607726297729843
[Non-Optimized] Height Liquid [Modified-Simpson] (m): 1.2094927590518265
[Non-Optimized] Height Gas [Modified-Simpson] (m): 1.175064293558333
[Non-Optimized] Height Error Percentage [Modified-Simpson] (%): 2.8465210093926805

- - - Part 1 Complete - - -

Output

- - - Part 1.5: Non-opmitized Validation Section - - -

X_AL = 0.03975417706932962
Y_AG = 0.11999999999999902
L = 0.011571111111111108
V = 0.004166666666666661
weightedMW_L = 19.05480122911465
weightedMW_V = 29.969999999999985

G_L = 0.49907645148987245
G_V = 0.2826591789312056
kxa = 0.14357011789333837
kya = 0.020462429219854757
X_Ai = 0.05565239389301146
Y_Ai = 0.010639639179242984
logMeanXA = 0.9522745962379409
logmeanYA = 0.9336129148947592
m = -6.878781566067364
b = 0.39346030039834584
f(x_Ai=0) = -3.357349120935993E-13
dZV = 4.47139878142027
dZL = 11.379677523472544
ZV Trap = 1.1739524250058786
ZL Trap = 1.2085262283030616
ZV Simp = 1.174761432679566
ZL Simp = 1.209358427571381
ZV ModSimp = 1.175064293558333
ZL ModSimp = 1.2094927590518265

- - - Part 1.5 Complete - - -

- - - Optimization Validation Section - - -
[Optimized] Number of Iterations [Trapezoid]: 30
[Optimized] Height on Liquid Side [Trapezoid] (m): 1.203
[Optimized] Height on Gas Side [Trapezoid] (m): 1.177
[Optimized] Highest Height [Trapezoid] (m): 1.203
[Optimized] Percentage Error for Liquid and Gas Side [Trapezoid] (%): 2.11
[Optimized] Solute-Free Liquid Flowrate [Trapezoid] (kgmole/h): 39.6
[Optimized] Inlet Liquid Flowrate [Trapezoid] (kgmole/h): 39.6
[Optimized] Number of Iterations [Simpsons]: 31

[Optimized] Height on Liquid Side [Simpson] (m): 1.223
[Optimized] Height on Gas Side [Simpson] (m): 1.16
[Optimized] Highest Height [Simpson] (m):1.223
[Optimized] Percentage Error for Liquid and Gas Side [Simpson] (%):5.194
[Optimized] Solute-Free Liquid Flowrate [Simpson] (kgmole/h): 40.92
[Optimized] Inlet Liquid Flowrate [Simpson] (kgmole/h): 40.92
[Optimized] Number of Iterations [Modified-Simpson]: 32
[Optimized] Height on Liquid Side [Modified-Simpson] (m): 1.243
[Optimized] Height on Gas Side [Modified-Simpson] (m): 1.143
[Optimized] Highest Height [Modified-Simpson] (m): 1.243
[Optimized] Percentage Error for Liquid and Gas Side [Modified-Simpson] (%): 8.053
[Optimized] Solute-Free Liquid Flowrate [Modified-Simpson] (kgmole/h): 42.24
[Optimized] Inlet Liquid Flowrate [Modified-Simpson] (kgmole/h): 42.24
- - - Part 2: Optimization Complete! - - -

Graph

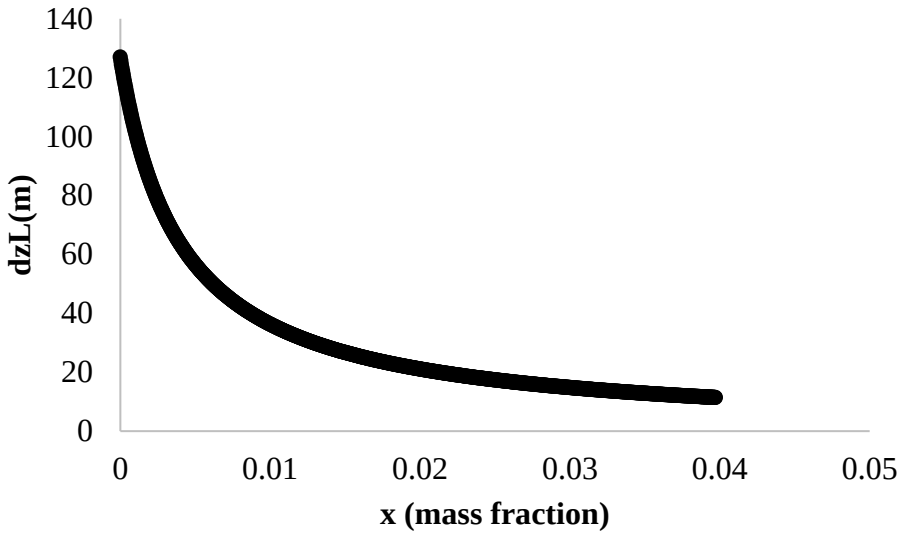


Figure 1 : The tower height for liquid side against mass fraction for the base case obtained from the simulator.

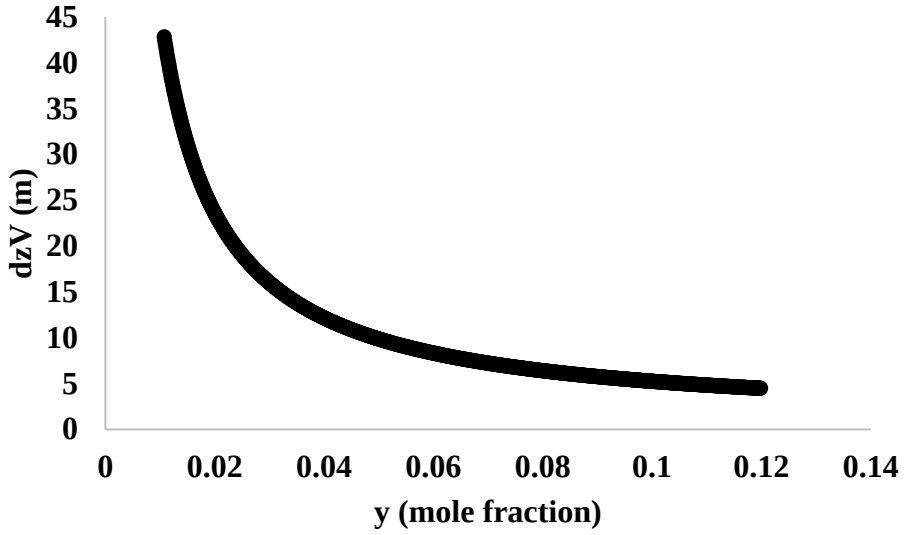


Figure 2: The tower height for vapour side against mole fraction for the base case obtained from the simulator.

Random Case

Input

packingMaterialChoice = 2.0

L_2 = 50.00
V_1 = 22.00

y_A1 = 0.18
x_A2 = 0.00
recoveryPercentage = 95.00

equilibriumPolynomialCoefficient5 = 0.0000
equilibriumPolynomialCoefficient4 = 0.0000
equilibriumPolynomialCoefficient3 = 0.000
equilibriumPolynomialCoefficient2 = 1.9762
equilibriumPolynomialCoefficient1 = 0.0812
equilibriumPolynomialCoefficient0 = 0.00

mw_A = 44.05
mw_L = 18.02
mw_V = 28.05

d_ABV = 1.35e-05
u_V = 9.85e-06
p_V = 1.261

d_ABL = 1.24e-09
u_L = 8.90e-04
p_L = 997.000

root_Finder_X_L = 0.0
root_Finder_X_U = 0.5
root_Finder_Tolerance = 1.00E-06
root_Finder_Maximum_Iterations = 1000.0
numerical_Integration_Number_Of_Steps = 1000.00
optimizer_Number_Of_Steps_Liquid_Flow_Rate = 100.0
optimizer_Minimum_Liquid_Flow_Rate = 0
An acceptable general binary mixture has been inputted.

Output
- - - Part 1: Non-opmitized Validation Section - - -

[Non-Optimized] Height Liquid [Trapezoid] (m): 1.5442102263522668
[Non-Optimized] Height Gas [Trapezoid] (m): 1.4718684243789795
[Non-Optimized] Height Error Percentage [Trapezoid] (%): 4.684712012571832
[Non-Optimized] Height Liquid [Simpson] (m): 1.5455386419099737
[Non-Optimized] Height Gas [Simpson] (m): 1.4731262647611445
[Non-Optimized] Height Error Percentage [Simpson] (%): 4.685251807055573
[Non-Optimized] Height Liquid [Modified-Simpson] (m): 1.545411591424089
[Non-Optimized] Height Gas [Modified-Simpson] (m): 1.4733471096253055
[Non-Optimized] Height Error Percentage [Modified-Simpson] (%): 4.663125487002229

- - - Part 1 Complete - - -

- - - Part 1.5: Non-opmitized Validation Section - - -

X_AL = 0.06997507533201838
Y_AG = 0.17999999999999777
L = 0.014933888888888882
V = 0.006111111111111095
weightedMW_L = 19.841451210892437
weightedMW_V = 30.929999999999968
G_L = 6.036378317891852
G_V = 3.8506159138243805
kxa = 0.5733926789609666
kya = 0.2735097775905522
X_Ai = 0.13447194263873516
Y_Ai = 0.046654160116438415
logMeanXA = 0.8973902332072022
logmeanYA = 0.8849992472631864
m = -2.0674777765481687
b = 0.3246719131262566
f(x_Ai=0) = -3.4973135498717056E-11
dZV = 3.6840626135382593
dZL = 7.937778999505164
ZV Trap = 1.4718684243789795
ZL Trap = 1.5442102263522668
ZV Simp = 1.4731262647611445
ZL Simp = 1.5455386419099737
ZV ModSimp = 1.4733471096253055
ZL ModSimp = 1.545411591424089

- - - Part 1.5 Complete - - -

- - - Optimization Validation Section - - -

[Optimized] Number of Iterations [Trapezoid]: 1
[Optimized] Height on Liquid Side [Trapezoid] (m): 0.322
[Optimized] Height on Gas Side [Trapezoid] (m): 8.525
[Optimized] Highest Height [Trapezoid] (m): 8.525

[Optimized] Percentage Error for Liquid and Gas Side [Trapezoid] (%): 96.225
[Optimized] Solute-Free Liquid Flowrate [Trapezoid] (kgmole/h): 1.804
[Optimized] Inlet Liquid Flowrate [Trapezoid] (kgmole/h): 1.804
[Optimized] Number of Iterations [Simpsons]: 1
[Optimized] Height on Liquid Side [Simpson] (m): 0.322
[Optimized] Height on Gas Side [Simpson] (m): 8.532
[Optimized] Highest Height [Simpson] (m):8.532
[Optimized] Percentage Error for Liquid and Gas Side [Simpson] (%):96.225
[Optimized] Solute-Free Liquid Flowrate [Simpson] (kgmole/h): 1.804
[Optimized] Inlet Liquid Flowrate [Simpson] (kgmole/h): 1.804
[Optimized] Number of Iterations [Modified-Simpson]: 1
[Optimized] Height on Liquid Side [Modified-Simpson] (m): 0.322
[Optimized] Height on Gas Side [Modified-Simpson] (m): 8.534
[Optimized] Highest Height [Modified-Simpson] (m): 8.534
[Optimized] Percentage Error for Liquid and Gas Side [Modified-Simpson] (%): 96.226
[Optimized] Solute-Free Liquid Flowrate [Modified-Simpson] (kgmole/h): 1.804
[Optimized] Inlet Liquid Flowrate [Modified-Simpson] (kgmole/h): 1.804

- - - Part 2: Optimization Complete! - - -

---- The excel can be found in the specified folder! ----

Exception Handling

This absorption procedure is impossible.

Input
packingMaterialChoice = 2.0

l_2 = 80.00
v_1 = 80.00
y_A1 = 0.80
x_A2 = 0.00
recoveryPercentage = 95.00

equilibriumPolynomialCoefficient5 = 0.0000
equilibriumPolynomialCoefficient4 = 0.0000
equilibriumPolynomialCoefficient3 = 0.000
equilibriumPolynomialCoefficient2 = 1.9762
equilibriumPolynomialCoefficient1 = 0.0812
equilibriumPolynomialCoefficient0 = 0.00

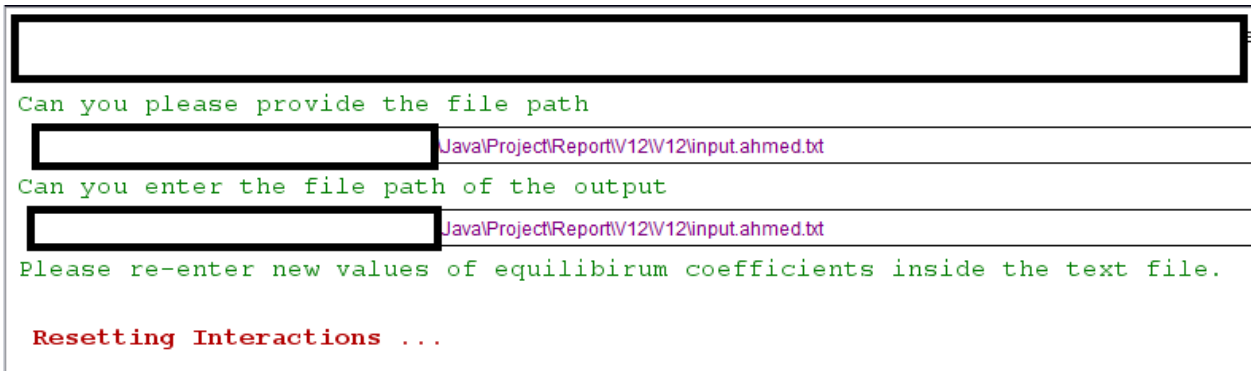
mw_A = 44.05
mw_L = 18.02
mw_V = 28.05

d_ABV = 1.35e-05
u_V = 9.85e-06
p_V = 1.261

d_ABL = 1.24e-09
u_L = 8.90e-04
p_L = 997.000

root_Finder_X_L = 0.0
root_Finder_X_U = 0.5
root_Finder_Tolerance = 1.00E-06
root_Finder_Maximum_Iterations = 1000.0
numerical_Integration_Number_Of_Steps = 1000.00
optimizer_Number_Of_Steps_Liquid_Flow_Rate = 100.0
optimizer_Minimum_Liquid_Flow_Rate = 0

Output



6. Extensions and Improvements

While validation campaigns were carried to ensure the program was as robust, user-friendly, versatile, the design of the code can be further extended and improved for it to offer more features, be more efficient, more user-friendly, more modular, more object-oriented and all the above. These modifications were not incorporated in the program due to their respective complexity and it simply being out of the scope of the assigned project description.

6.1. Extensions

The current program was designed in manner for it be extendable for additional methods, instance variables and classes to the package of the project. This program simulates a gas absorber system and it can easily be implemented into process simulation with slight modifications to the program's architect. For instance: the inlet gas stream can be provided by a plug flow reactor and the inlet liquid stream can be obtained by cooling tower. Instead of the inlet data and proprieties of the substances (solute and solvents) to be extracted from an `Input.txt` file, these can hypothetically be obtained by objects of `Reactor` type and `CoolingTower` type vi their accessors. These values would be stored in the instances variables of the default constructor of `UserInputs` class. However, the user would be required to enter the equilibrium mixture coefficients to size the gas absorber via file I/O unless another input method is implemented. The governing equations used to model the gas absorption process assumed isobaric/isothermal conditions, non-immiscible fluids and the effects of solubilization of the solute in either fluids have not been accounted for. Some processes would require information on the temperature gradient through the column height to compute for instance an energy balance, or pressure gradient to compute the pressure drop through the column. The complexity of incorporating either of these features can introduce a large set of differential equations and/or non-homogeneous equations which would require advance numerical methods to approximate solutions [5-7]. Thus, for the sake of this project, these features were not incorporated. The current simulation program has a specific set of variables to initialize in the `input.txt` files, where outlet liquid/gas flow rates or fractions cannot be specified since they are computed via methods in `UserInputs` class by utilizing the inlet parameters and recovery percentage. Through the incorporation of more classes and methods, the program can be altered to so that any combination of design parameters can be used to generate the rest of the parameters until the degree of freedom reaches zero. This functionality is well used in process simulation software such as: UNISIM, ASPEN… For instance, if the user wants to reuse a pre-built absorption tower that has a height, but they would like to know the capacity of this column to separate a certain solute using various fluids in the gas and liquid stream. The program can be reverse engineering for instance to calculate the inlet and/or outlet flow rates/fractions if the degree of freedom reaches zero.

6.2. Improvements

Certain limitations of the program can be suppressed or eliminated through a series of improvements to make the simulation more robust and versatile. Throughout the code, arrays were utilized store datasets; it might have been worthwhile using array lists or hash tables to improve the usage of linked data structures in the program. The `UserOutputs` class could have been improved to output more tables into a single sheet, graphical output. The program could have been re-engineered to accommodate an Excel input instead of a `fileinputstream`, which would have been worthwhile. For instance, data for runs can be extracted from multiple sheets from a single excel file as opposed to multiple barebone text files.

7. Validation of Emerald Absorption Simulator

7.1. Design Structure of the code

The Anonymous Code consists of 19 different classes that describes an absorption simulation program. The absorption column simulation begins in the Driver.java class. It first takes the input values from the Data.txt text file as String. These Strings are used to build an ArrayList of type String and builds an array based on it. Using certain symbols as delimiters, the values for the different 6 input parameters have been obtained by reading the first part of each String that are stored in the array. The remainder of String for each lines is not read. The simulation, then, goes through the exception handling. The Driver.java class constructs an object in the LiquidStream.java class, VapourStream.java class and Tower.java class. Eventually, the results obtained from these classes are stored in arrays and eventually printed to the user in the console. In other words, the Driver.java main class gets the inputs from Data.txt, assigns it to build in ArrayList, runs the exception handling and lastly goes through the LiquidStream.java class, VapourStream.java and the Tower.java to calculate the height of the absorption tower.

The Main class consists three main parts, the input, exception handling and objects that are responsible for the absorption tower simulation. In the input, the code scan for the data.txt file. The anonymous code scans Data.txt using a Scanner that extends new FileInputStream to read the Data.txt file and store it in an ArrayList. This ArrayList has a fixed length of 6 inputs and the simulator checks if the stored array length is less than 6. The first exception is to determine whether the file is in the proper location. This is done by the FileNotFoundException. If the file is not in the correct place or it does not exist, this exception will be thrown. Through another custom exception called InvalidInputException, the simulator will check if some data is missing in the input file. The simulator stores the read values to the ArrayList without checking if the values matches with the respected String. Also, the program checks if

the values are negative and throws an exception if such values are inputted. Thus, if one of the inputted values is negative custom exception called NegativeNumberException will be thrown. The program, then, checks if the recovery percentage are above 100 or below zero and throws an Invalid Fraction Exception. The same exception is thrown if the mole fraction of solute in the liquid stream and in the gas stream is between 0 and 1. The program in the main class also checks if the number was inputted in the right format. Otherwise, the simulator will throw a Number Format Exception. The final exception occurs inside the simulator during calculation. Once the simulator goes through the input exception handling, it runs the absorption tower simulation which consists of three important methods. The first class making an object of LiquidStream, VapourStream and lastly a Tower class. If the calculated liquid is below Lmin (minimum liquid flow rate), the absorption tower system is rejected and an exception is thrown. This exception is called LowLiquidException. Therefore, the main class goes through input, exception handling for the inputs and the absorption column.

Liquid and the Vapor Stream:

The LiquidStream constructor uses 7 hard-coded parameters to compute the mass transfer coefficient kxa and kya. Using these mass transfer coefficients, it finds the mole fraction of solute in each stream at the gas-liquid interface. Using these two value, the log mean of the mole fraction of solute in each stream at the interface is found. The LiquidStream is an object that is used to calculate the values associated to a liquid state such as the mass transfer coefficient in the liquid stream. These methods are implemented in a similar fashion in the VapourStream.java class. Both the VapourStream.java and LiquidStream.java inherit from the Stream.java class. The VapourStream.java responsible for calculating the kya and the liquid stream responsible for calculating the kxa. Though, the equilibrium line is used in the VapourStream class. However, the equilibrium formula is not found in the LiquidStream that only consist of two methods of kxa and log mean x. Therefore both the LiquidStream class and VapourStream class computes the liquid and vapour mass transfer coefficient, calculates the logmean x and y respectively, however the equilibrium line calculation is implemented in the VapourStream class.

The Tower.java class is a class that is used to calculate the required parameters such as height in both the liquid and vapour side. The Tower constructor accepts 3 objects and 1 primitive data type: a VapourStream object, a LiquidStream object, a PackingMaterial object and a double called fractionRecovery. The Tower.java class implements a root function that calculates the intersection between the equilibrium line and the operating line. However, the equilibrium relation is not specified by

the user, but rather it is directly incorporated in the code. The Tower.java class implements a function integration that accepts a double x and a double y and returns an array of double that calculates the height ZV and ZL. The tower height also solve for the Lmin which calculates the minimum flow rate that is required to accomplish the required percent recovery. The Tower class implements a function FindHeight that is responsible for calculating the height of the column and the tower. The Tower class optimizes the height using the optimizer function incorporated in the Tower class.

The anonymous code had three major parts, the Drive class is responsible for calling the program, the liquid Vapor stream class and the Tower class. The Drive class takes the input from data.txt file and assign it to an ArrayList list to access the values for the calculation of different parameters. Then, the code goes through number of exceptions to check whether the inputted values are the right values. The main methods, then, calls the VapourStream.java and LiquidStream.java to calculate the different parameters associated with each liquid and stream classes. Lastly the main methods makes a Tower object that is responsible in calculation of the final desired parameters that are outputted by the software.

7.2. Modification

Although the anonymous code contains multiple ways of handling various inputs, some improvements should be made in that regards. There are various features that are not well implemented in the codes. These are features that are required in order to accommodate more possible results with different inputs. In addition, even if the anonymous code handles a large number of exceptions, it does not handle all of them. These are found both in the input classes and the classes containing calculated methods and variables passed in them.

The features that needs to be added comes from the fact that many methods and variables passed in classes are hard coded. Hard coded variables and methods do not allow expansion to a code. For instance, the absorption tower simulator deals with binary mixtures: liquid and gas. Each of these binary mixtures of liquid and gas have a different equilibrium relationship, different molecular weights, different densities, and different viscosities. These parameters should not be hard coded, but rather should be inserted into their respective classes as a static double, should the user want to modify these parameters in the future. Another way to resolve this issue can be by giving the user the option to insert these values in a text file and have the input class capture these values from it. This text file does not need to be the same as the main input text file, which contains the main simulation parameters. Similarly, the equilibrium

relationship should be placed inside a static method and return the value of `y_Ai` from a value of `x_Ai` same as usual.

Another change that can be added to the absorption column simulator is the interpretation of the String values. The interpretation of the String values serves to match the input with a certain String. This would make sure that every string name is matched to its corresponding value thus it would eliminate any wrong values inputted by the user since it does not assume certain patterns to follow when inputting the result. For instance, inputting the value #92.0:

Another feature, which was not well implemented are the simulator constraints for the numerical methods used and for optimization. In this case, in the Tower.java class, the number of steps for the trapezoidal rule has been hard-coded to a value 10,000, where it should have been made into a static integer. Another example of this is found in the UpperBound.java class in the `calculateXMax()` method, where the static method for root finder is hard-coded to be bound between 0 and 1. In this method, the bounds 0 and 1 should have been made into a static double. The hard coded constraints have also been found in the `optimize()` method in the Tower.java class. Here, the constraints have again been hard-coded: the precision for values checked is set to 0.1, the number of iterations has been setup as a for-loop which is limited to exactly 4 loops. The final element that creates bad code is the bound for the multiplier for `Lmin` is set to be exactly in between the value of 1.2 and 1.5. A suggestion to correct this issue is to use a do-while loop, where the upper bound and lower bound of the `Lmin` multiplication factor and the precision of values checked should be setup as a static double. Another feature that should be included is the indication in the console that the simulator is doing something.

7.3. Exception Handling

The second issue, the anonymous code needs to improve on are the exceptions. There are many exceptions that needs to be handled properly and some other exceptions that are missing. The results of these tests are shown in. The following are the exceptions that needs to be handled:

1. The anonymous code should be throwing an exception if there is no gas or liquid flow. Currently if a value of zero is inputted for the flow rate, the simulator still runs and computes the height. Since, there is no flow rate in the liquid, this will end up outputting a value of infinity for the height. This is because, when there is no flow of liquid in the tower, this would make absorption impossible. Therefore, this case must have an exception.

2. There are no exceptions set in place if the flow rate of liquid and gas are very small. When these flow rates are very small, the final height is not computed. Therefore, the simulator should be able to handle this exception.

3. There should be an exception preventing the input of very small value of recovery and mole fractions for both the liquid and gas side. When a small value of recovery and mole fraction are inputted,

4. There are no exception put in place if the recovery is set to 0. If no recovery has been set in place, the simulator should not be able to run. This is because if there is no decided recovery, absorption cannot take place. Therefore, a height of zero will be returned. In such cases, the simulator should not be able to run.

5. There are no exception put in place if the recovery is set to a very low number. A very small recovery will this time not return a height of 0, but no value for height can be obtained from it.

6. There are no exception that handles large numbers for the input flow rates. There should be an upper bound exception put in place in order to prevent having results that do not make sense.

7. From the above, there are no max iterations exception put in place for the Ridders method. A small flow rate will make it close to impossible to solve. However, if there are no bound in the code for such values, the simulator will run until it has achieved such value. This can be seen if a very small flow rate has been inputted. The

8. Some parameters that are not well defined may lead to a very large height. Large heights obtained have not been handled properly. The outputted exception handling are not defined properly, for example checking if the height of the tower is realistics compared to the given values that are inputted in the absorption simulator.

9. If the semicolon is removed then there is no exception to handle this type of error. For example if the flow rate semicolon was removed and a number was inputted, then java will print a StringIndexOutOfBoundException which defines that the string is out of bound. This shows that there is no exceptions that handles this type of error, thus the java project will throw exception.

10. The Next exception handling problem is when a space is left out in the data.txt text file such as an space is entered in the file data.txt, then java will throw a StringIndexOutOfBound error. This is not handled properly since if a user enters a space in the data.txt then the program will crash not allowing any continuation of the program. Therefore, leaving a space in data.txt text file will cause the program to crash not allowing the program to run and continue executing.

Table 18. Validation table for anonymous group Emerald

	Input	Output

1.	Changing in the input.txt file:
	from
	`#12.0: inlet gas flow rate`
	To
	`#0: inlet gas flow rate`
		`The absorption tower's height is NaNm with a diameter of 0.7500m. Using heuristics, the optimal height NaNm is reached at an initial liquid flow rate of NaNkmol/h.`

2.	Changing in the input.txt file:
	from
	`#12.0: inlet gas flow rate`
	To
	`#0.000000001: inlet gas flow rate`
		(no outputted results)

3.	Changing in the input.txt file: from `#92: percentage recovery [0-100]` To	`The absorption tower's height is 0.0000m with a diameter of 0.7500m.`

	#0: percentage recovery [0-100]	Using heuristics, the optimal height NaNm is reached at an initial liquid flow rate of 0.0000kmol/h.
4.	Changing in the input.txt file: from #0.12: inlet mole fraction in the gas phase [0-1] To #0.0001: inlet mole fraction in the gas phase [0-1]	The absorption tower's height is NaNm with a diameter of 0.7500m. Using heuristics, the optimal height NaNm is reached at an initial liquid flow rate of 0.1404kmol/h.
5.	Changing in the input.txt file: from #92: percentage recovery [0-100] To #0.00000001: percentage recovery [0-100]	The absorption tower's height is NaNm with a diameter of 0.7500m. Using heuristics, the optimal height NaNm is reached at an initial liquid flow rate of 0.0000kmol/h.

6.	Changing in the input.txt file:
	from
	`#12.0: inlet gas flow rate`
	`#40.0: inlet liquid flow rate`
	To
	`#12000000000000: inlet gas flow rate`
	`#40000000000000: inlet liquid flow rate`
		`The absorption tower's height is 664.6411m with a diameter of 0.7500m. Using heuristics, the optimal height 764.3339m is reached at an initial liquid flow rate of 3340084997890888.0000kmol/h.`

7.	Changing in the input.txt file:
	from
	`#12.0: inlet gas flow rate`
	To (no more semicolon)
	`#12.0 inlet gas flow rate`
		`Exception in thread "main" java.lang.StringIndexOutOfBoundsException: String index out of range: -2`
		`at java.lang.String.substring(Unknown Source)`
		`at Driver.main(Driver.java:37)`

8.	#12.0: inlet gas flow rate
	(enter)
	#0.12: inlet mole fraction in the gas phase [0-1]
	#40: inlet liquid flow rate
	#0.0: inlet mole fraction in the liquid phase [0-1]
	#92: percentage recovery [0-100]
	Raschig: Packing type "Raschig" "Pall" "Berl"
		`Exception in thread "main" java.lang.StringIndexOutOfBoundsException: String index out of range: -1`
		`at java.lang.String.substring(Unknown Source)`
		`at Driver.main(Driver.java:37)`

9.0	#-15.0: inlet gas flow rate #0.12: inlet mole fraction in the gas phase [0-1] #40.0: inlet liquid flow rate #0.0: inlet mole fraction in the liquid phase [0-1] #92.0: percentage recovery [0-100] #Raschig: Packing type "Raschig" "Pall" "Berl"	`Invalid input: NegativeNumberException: Values cannot be negative` `Please input the desired value of the Inlet Gas Flow Rate.`

10.	#15.0: inlet gas flow rate #-0.12: inlet mole fraction in the gas phase [0-1] #40.0: inlet liquid flow rate #0.0: inlet mole fraction in the liquid phase [0-1] #92.0: percentage recovery [0-100] #Raschig: Packing type "Raschig" "Pall" "Berl"	Invalid input: NegativeNumberException: Values cannot be negative Please input the desired value of the Inlet Gas Flow Rate.
11.	#15.0: inlet gas flow rate #0.12: inlet mole fraction in the gas phase [0-1] #-40.0: inlet liquid flow rate #0.0: inlet mole fraction in the liquid phase [0-1] #92.0: percentage recovery [0-100] #Raschig: Packing type "Raschig" "Pall" "Berl"	Invalid input: NegativeNumberException: Values cannot be negative Please input the desired value of the Inlet Gas Flow Rate. .
12	#15.0: inlet gas flow rate #0.12: inlet mole fraction in the gas phase [0-1] #40.0: inlet liquid flow rate #-0.30: inlet mole fraction in the liquid phase [0-1] #92.0: percentage recovery [0-100] #Raschig: Packing type "Raschig" "Pall" "Berl"	Invalid input: NegativeNumberException: Values cannot be negative Please input the desired value of the Inlet Gas Flow Rate.

13	#15.0: inlet gas flow rate #0.12: inlet mole fraction in the gas phase [0-1] #40.0: inlet liquid flow rate #0.30: inlet mole fraction in the liquid phase [0-1] #-92.0: percentage recovery [0-100] #Raschig: Packing type "Raschig" "Pall" "Berl"	`Invalid input: NegativeNumberException: Values cannot be negative` `Please input the desired value of the Inlet Gas Flow Rate.`

Table 19: The following table summarizes the exceptions that have been handled by the simulator

Number	Varied Variable	Value	Anonymous Code	Excel Output
1	+FlowRate	20	1.3512	1.35167
2	-FlowRate	10	1.0453	1.045
3	+MoleFractionIn	0.24	1.1904	1.1903
4	-MoleFractionIn	0.6	1.2234	1.223
5	-Inlet Liquid Flow	35	1.2920	1.2985
6	+LiquidFlowRate	45	1.1367	1.1415
7	+RecoveryPercentage	85	0.9004	0.900
8	RecoveryPercentage	95	1.4475	1.4484

8. Conclusion

The objective of this project was to design a Java program to simulate a gas absorption process. The program enables the user to compute the height of the column by specifying a set of parameters. By varying the solute-free inlet liquid flow rate, the program can determine the optimal liquid flow rate and height. The program was designed to mimic the components of a gas absorber system via object-oriented program; where this program has 16 highly interactive classes. The simulated system assumes gas and liquid phases are immiscible and the computed Schmidt number is assumed constant for both the gas and liquid phases. The program starts by prompting the user to assign values to a collection of variables necessary to simulate the absorber; it computes the height of the column for a given inlet liquid flow rate using three different numerical integration methods. Using the same numerical integrations and parameters set by the user, the solute-free liquid flow rate is fluctuated to optimize the mass transfer between both phases of the system and the optimal column height as well as solute-free liquid flow rate is reported.

A validation campaign was undergone to explore the validity of the described program. Three cases were tested to confirm the exception handling works as expect, all methods and classes operate as prescribed and the goals of the projects are met. The outputs of the code were validated using an Excel simulator and main validation methods. For the sake of the optimization campaign, the settings of the numerical methods were setup to be identical for both the Java and Excel simulation. For the computation of parameters These governing equations were rearranged to calculate these flow rates and fractions in the following methods: `V_f`, `L_f`, `V_2`, `Y_A2`, `X_A1` and more resulted in insignificant differences between the Excel and Java Output. This was truth for the output of all methods from the program. The anonymous code was robust through the validation campaign, however lacked important features related to the project requirement such as: enabling the user to input a general binary mixture of their liking, lacking input exception handling, an output file and it has a minimal number of parameters that can be manipulated by the user. Besides these short-comings, their program does efficiently utilize effective data structure and object-oriented programming features.

9. References

1. C. J. Geoankopolis, *Transport Processes and Unit Operations*, 3rd Edition, Wiley Publishers, **1993**
2. [W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, *Numerical Recipes: The Art of Scientific Computing*, 3rd Edition, *W.*
3. A. Sowinsky, *CHG3331: Mathematical Modelling and Numerical Methods Course Pack*, **2016**.
4. S. C. Chapra, R. P. Canale, *Numerical Methods for Engineers*, 7th Edition, McGraw-Hill Higher Education, New York City, **2015**.
5. E. Dumont and H. Delmas, "Mass transfer enhancement of gas absorption in oil-in-water systems: A review," *Chem. Eng. Process. Process Intensif.*, vol. 42, no. 6, pp. 419–438, **2003**.
6. W. D. Deckwer, "Non-isobaric bubble columns with variable gas velocity," *Chem. Eng. Sci.*, vol. 31, no. 4, pp. 309–317, **1976**.
7. I. Pentchev, K. Paev, and I. Seikova, "Dynamics of non-isothermal adsorption in packed bed of biporous zeolites," *Chem. Eng. J.*, vol. 85, no. 2–3, pp. 245–257, **2002**.

8. Appendices

a. Appendix A: Overall Schematic of Program Design and Architecture

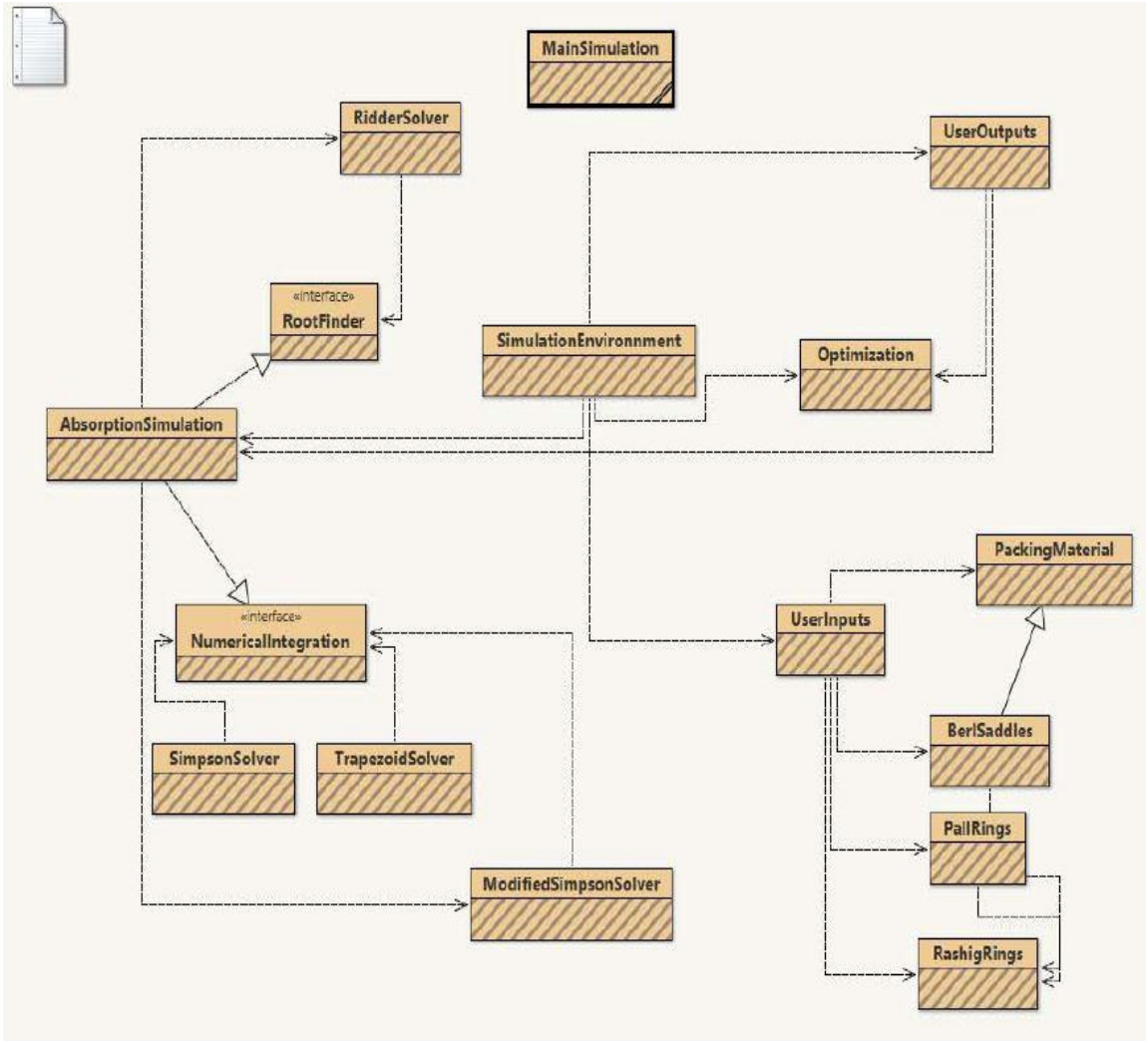


Figure 3. UML Diagram of Gas Absoprtion Program.

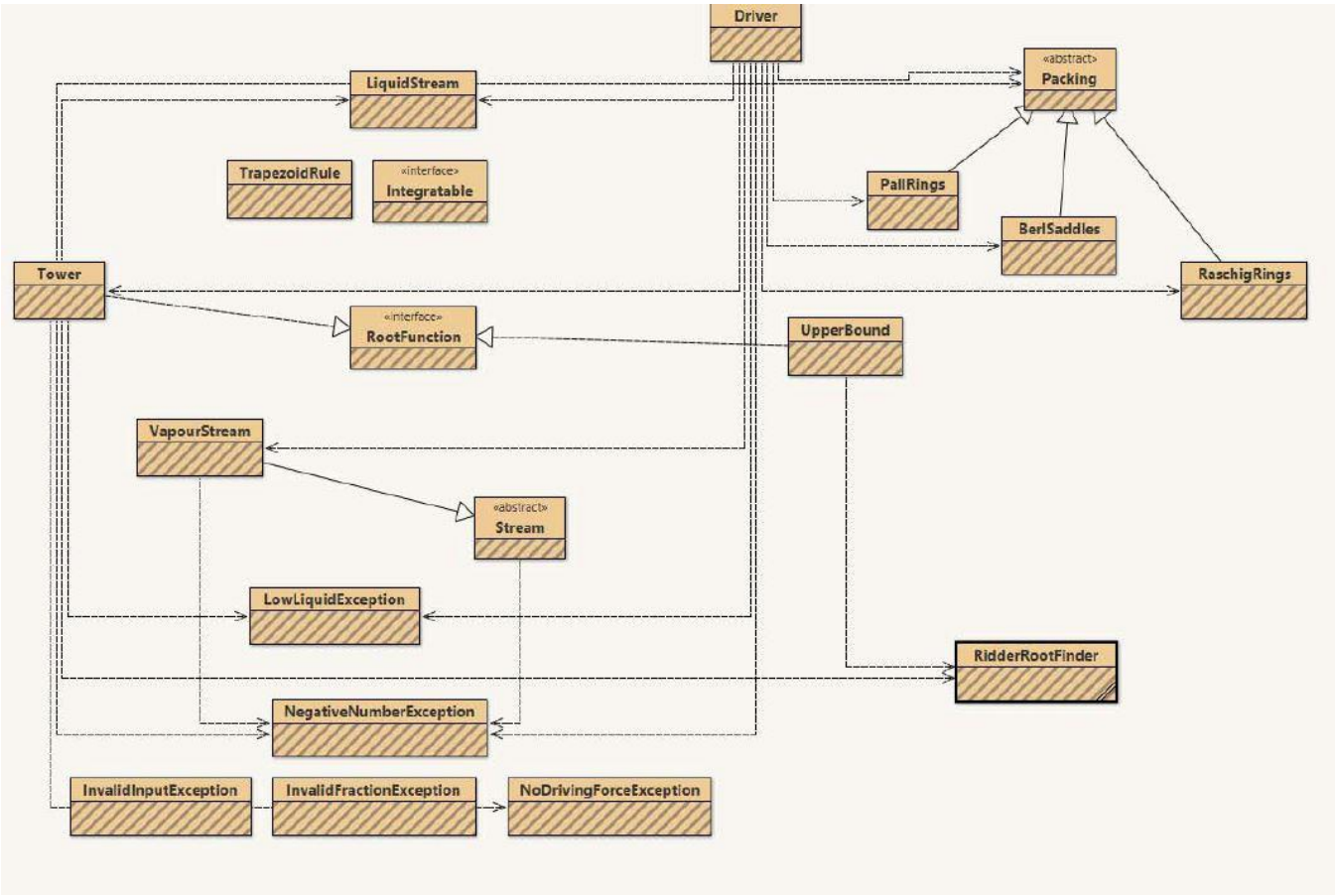


Figure 4. UML Diagram of Anonymous Code.

b. Appendix B: Figures

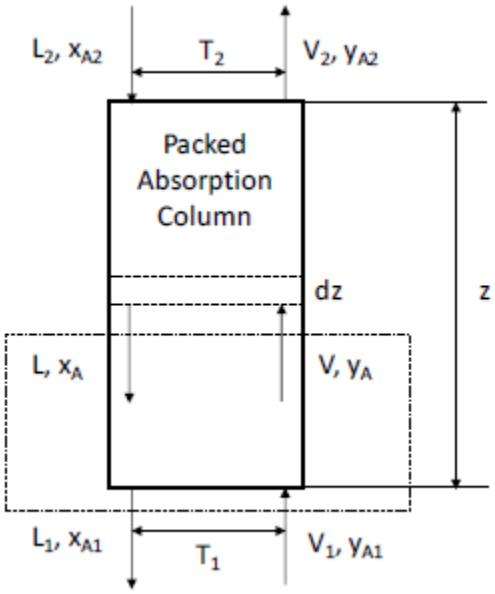


Figure 5: Process Schematic for the absorber design system [1].

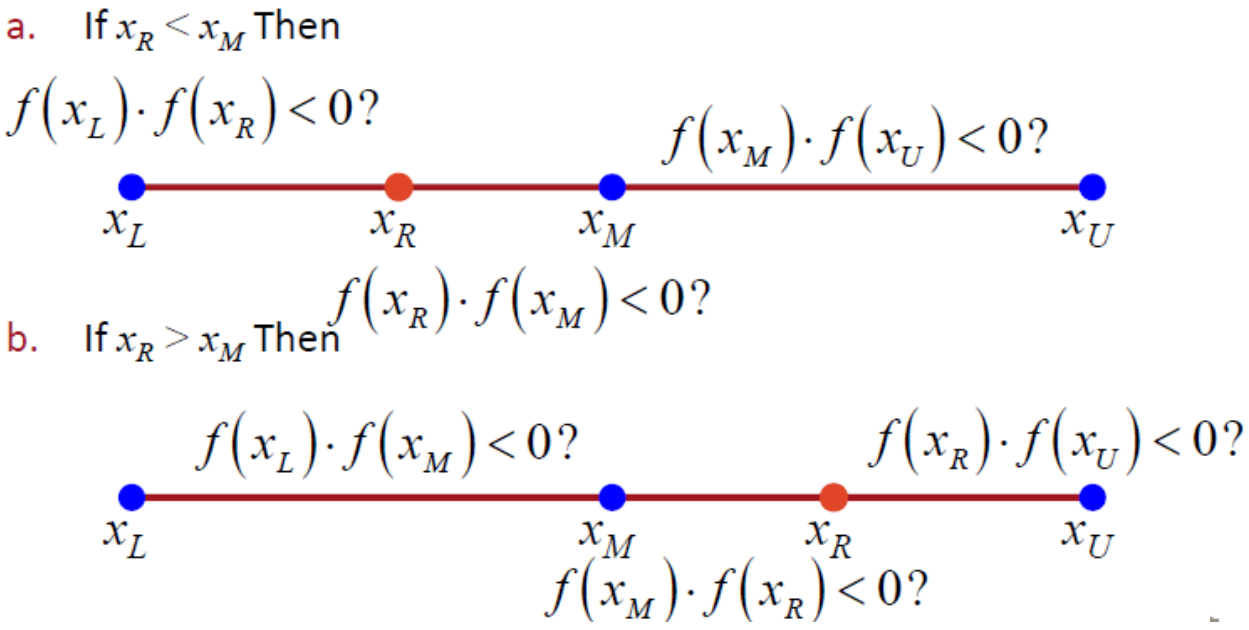


Figure 6. The range of bounds for Ridders' algorithm [2 & 3].

c. Appendix C: Program Code

```java

public class MainValidationPart3 {

	public static void main(String[] args) {

		// This method verifies the some of the computed values in
		// AbsorptionSimulation.class
		UserInputs myUserInputs = new UserInputs();
		AbsorptionSimulation myAbsorptionSimulation = new AbsorptionSimulation(myUserInputs);

		double[] arrayUserAbsorptionSimulationComputationResults = new double[14];

		arrayUserAbsorptionSimulationComputationResults[0] = myAbsorptionSimulation.getG_X_A2();
		arrayUserAbsorptionSimulationComputationResults[1] = myAbsorptionSimulation.getG_Y_A1();
		arrayUserAbsorptionSimulationComputationResults[2] = myAbsorptionSimulation.getG_L();
		arrayUserAbsorptionSimulationComputationResults[3] = myAbsorptionSimulation.getG_V();
		arrayUserAbsorptionSimulationComputationResults[4] = myAbsorptionSimulation.getG_weightedMW_V();
		arrayUserAbsorptionSimulationComputationResults[5] = myAbsorptionSimulation.getG_weightedMW_L();
		arrayUserAbsorptionSimulationComputationResults[6] = myAbsorptionSimulation.getG_G_V();
		arrayUserAbsorptionSimulationComputationResults[7] = myAbsorptionSimulation.getG_G_L();
		arrayUserAbsorptionSimulationComputationResults[8] = myAbsorptionSimulation.calculate_kxa();
		arrayUserAbsorptionSimulationComputationResults[9] = myAbsorptionSimulation.calculate_kya();
		arrayUserAbsorptionSimulationComputationResults[10] = myAbsorptionSimulation.calculateY_Ai();
		arrayUserAbsorptionSimulationComputationResults[11] = myAbsorptionSimulation.calculateX_Ai();
		arrayUserAbsorptionSimulationComputationResults[12] = myAbsorptionSimulation.logMeanYA();
		arrayUserAbsorptionSimulationComputationResults[13] = myAbsorptionSimulation.logMeanXA();

		String[] arrayUserAbsorptionSimulationComputationString = new String[14];

		arrayUserAbsorptionSimulationComputationString[0] = "Bulk Concentration of A in the liquid phase (-):
";
		arrayUserAbsorptionSimulationComputationString[1] = "Bulk Concentration of A in the Gas Phase (-): ";
		arrayUserAbsorptionSimulationComputationString[2] = "Liquid FLowrate (kmol/s): ";
		arrayUserAbsorptionSimulationComputationString[3] = "Gas FLowrate (kmol/s): ";
		arrayUserAbsorptionSimulationComputationString[4] = "Average Molecular Weight on the Gas-side (g/mol)
";
		arrayUserAbsorptionSimulationComputationString[5] = "Average Molecular Weight on the Liquid-side
(g/mol): ";
		arrayUserAbsorptionSimulationComputationString[6] = "Mass Flux on Gas-side (kg/s*m^2): ";
		arrayUserAbsorptionSimulationComputationString[7] = "Mass Flux on Liquid-side (kg/s*m^2): ";
		arrayUserAbsorptionSimulationComputationString[8] = "Film Mass Transfer in the Liquid Phase Multiplied
by the Specific Area (kmol/s*m^3): ";
		arrayUserAbsorptionSimulationComputationString[9] = "Film Mass Transfer in the Gas Phase Multiplied by
the Specific Area (kmol/s*m^3): ";
		arrayUserAbsorptionSimulationComputationString[10] = "Mole Fraction at Interface in the Gas Phase (-)
";
		arrayUserAbsorptionSimulationComputationString[11] = "Mole Fraction at Interface in the Liquid Phase
(-): ";
		arrayUserAbsorptionSimulationComputationString[12] = "Log Mean of the Concentration at Interface in
the Gas-side Phase (-): ";
		arrayUserAbsorptionSimulationComputationString[13] = "Log Mean of the Concentration at Interface in
the Liquid-side Phase (-):";

		String typeValidationString = "[Non_Optimized - Computed Data] ";

		for (int i = 0; i < arrayUserAbsorptionSimulationComputationResults.length; i++) {
			System.out.println(typeValidationString + arrayUserAbsorptionSimulationComputationString[i]
					+ arrayUserAbsorptionSimulationComputationResults[i]);
		}

	}

}

```

```java

public class MainValidationPart2 {

	public static void main(String[] args) {

		// Validation Method to verify if the computed values of UserInputs.class
		UserInputs myUserInputs = new UserInputs();

		String filePathInputString = "File Path of Input file: ";
		String filePathInputResult = myUserInputs.getFilePath();

		String filePathOutputString = "File Path of Output file: ";
		String filePathOutputResult = myUserInputs.getfilepathoutput();

		String isGeneralBinaryMixtureCorrectString = "Boolean of the General Mixture Validation (False or True): ";
		Boolean isGeneralBinaryMixtureCorrectResults = myUserInputs.isGeneralBinaryMixtureCorrect();

		double[] arrayUserInputsComputationResults = new double[12];

		arrayUserInputsComputationResults[0] = myUserInputs.getMyPackingMaterial().calculateS();
		arrayUserInputsComputationResults[1] = myUserInputs.getMyPackingMaterial().getF_p();
		arrayUserInputsComputationResults[2] = myUserInputs.getV_f();
		arrayUserInputsComputationResults[3] = myUserInputs.getL_f();
		arrayUserInputsComputationResults[4] = myUserInputs.getV_2();
		arrayUserInputsComputationResults[5] = myUserInputs.getY_A2();
		arrayUserInputsComputationResults[6] = myUserInputs.getX_A1();
		arrayUserInputsComputationResults[7] = myUserInputs.getL_1();
		arrayUserInputsComputationResults[8] = myUserInputs.getSchmidtGas();
		arrayUserInputsComputationResults[9] = myUserInputs.getSchmidtLiquid();
		arrayUserInputsComputationResults[10] = myUserInputs.getSlopeofOperatingLine();
		arrayUserInputsComputationResults[11] = myUserInputs.getYinterceptofOperatingLine();

		String[] arrayUserInputsComputationString = new String[12];

		arrayUserInputsComputationString[0] = "Column Cross-Sectionnal Area (m^2): ";
		arrayUserInputsComputationString[1] = "Packing Factor (m^(-1))";
		arrayUserInputsComputationString[2] = "Solute-free gas flow rate (kmole/h): ";
		arrayUserInputsComputationString[3] = "Solute-free liquid flow rate (kmole/h): ";
		arrayUserInputsComputationString[4] = "Outlet gas Flow Rate (kmol/h): ";
		arrayUserInputsComputationString[5] = "Outlet Gas Solute Fraction (-): ";
		arrayUserInputsComputationString[6] = "Outlet Liquid Solute Fraction (-): ";
		arrayUserInputsComputationString[7] = "Outlet Liquid Flow Rate (kmol/h): ";
		arrayUserInputsComputationString[8] = "Schmidt Number of Gas-side (-): ";
		arrayUserInputsComputationString[9] = "Schmidt Number of Liquid-side (-): ";
		arrayUserInputsComputationString[10] = "Slope of Operating Line (-): ";
		arrayUserInputsComputationString[11] = "Y_intercept of Operating Line (-): ";

		String typeValidationString = "[UserInputs - Computed Data] ";

		for (int i = 0; i < arrayUserInputsComputationResults.length; i++) {
			System.out.println(
					typeValidationString + arrayUserInputsComputationString[i] + arrayUserInputsComputationResults[i]);
		}

		System.out.println(typeValidationString + filePathInputString + filePathInputResult);
		System.out.println(typeValidationString + filePathOutputString + filePathOutputResult);
		System.out.println(
				typeValidationString + isGeneralBinaryMixtureCorrectString + isGeneralBinaryMixtureCorrectResults);

	}
}

```

```java

public class MainValidationPart1 {

	public static void main(String[] args) {

		// Validation Method to verify if the inputs found in the values match with
		// values in the input.txt file
		UserInputs myUserInputs = new UserInputs();

		String[] arrayFileInputsString = new String[28];

		arrayFileInputsString[0] = "Choice of Packing Material: ";
		arrayFileInputsString[1] = "Inlet Liquid Flow Rate (kmol/h): ";
		arrayFileInputsString[2] = "Inlet Gas Flow Rate (kmol/h): ";
		arrayFileInputsString[3] = "Inlet Solute Fraction on gas-side (-): ";
		arrayFileInputsString[4] = "Inlet Solute Fraction on liquid-side (-): ";
		arrayFileInputsString[5] = "Recovery Percentage of Solute (%): ";
		arrayFileInputsString[6] = "5th Degree Equilibrium Polynomial Coefficient (-): ";
		arrayFileInputsString[7] = "4th Degree Equilibrium Polynomial Coefficient (-): ";
		arrayFileInputsString[8] = "3rd Degree Equilibrium Polynomial Coefficient (-): ";
		arrayFileInputsString[9] = "2nd Degree Equilibrium Polynomial Coefficient (-): ";
		arrayFileInputsString[10] = "1st Degree Equilibrium Polynomial Coefficient (-): ";
		arrayFileInputsString[11] = "0th Equilibrium Polynomial Coefficient (-): ";
		arrayFileInputsString[12] = "Molecular Weight of the Solute (g/mol): ";
		arrayFileInputsString[13] = "Molecular Weight of the Liquid Solvent (g/mol): ";
		arrayFileInputsString[14] = "Molecular Weight of the Gas Solvent (g/mol): ";
		arrayFileInputsString[15] = "Diffusion Coefficient of Solute Through Gas (m^2/s): ";
		arrayFileInputsString[16] = "Viscocity of Gas (Pa*s): ";
		arrayFileInputsString[17] = "Density of Gas (kg/m^3): ";
		arrayFileInputsString[18] = "Diffusion Coefficient of Solute Through Liquid (m^2/s): ";
		arrayFileInputsString[19] = "Density of Liquid (kg/m^3): ";
		arrayFileInputsString[20] = "Viscocity of Liquid (Pa*s): ";
		arrayFileInputsString[21] = "Root Finder: Liquid Fraction Lower Bound (-): ";
		arrayFileInputsString[22] = "Root Finder: Liquid Fraction Upper Bound (-): ";
		arrayFileInputsString[23] = "Root Finder: Tolerance (-): ";
		arrayFileInputsString[24] = "Root Finder: Maximum Iterations (-): ";
		arrayFileInputsString[25] = "Numerical Integration: Number of Steps (-): ";
		arrayFileInputsString[26] = "Optimizer: Number of Steps for Liquid Flow Rates (-): ";
		arrayFileInputsString[27] = "Optimizer: Minimum Liquid Flow Rates (-): ";

		double[] arrayFileInputsResults = new double[28];

		arrayFileInputsResults[0] = myUserInputs.getPackingMaterialChoice();
		arrayFileInputsResults[1] = myUserInputs.getL_2();
		arrayFileInputsResults[2] = myUserInputs.getV_1();
		arrayFileInputsResults[3] = myUserInputs.getY_A1();
		arrayFileInputsResults[4] = myUserInputs.getX_A2();
		arrayFileInputsResults[5] = myUserInputs.getRecoveryPercentage();
		arrayFileInputsResults[6] = myUserInputs.getEquilibriumPolynomialCoefficient5();
		arrayFileInputsResults[7] = myUserInputs.getEquilibriumPolynomialCoefficient4();
		arrayFileInputsResults[8] = myUserInputs.getEquilibriumPolynomialCoefficient3();
		arrayFileInputsResults[9] = myUserInputs.getEquilibriumPolynomialCoefficient2();
		arrayFileInputsResults[10] = myUserInputs.getEquilibriumPolynomialCoefficient1();
		arrayFileInputsResults[11] = myUserInputs.getEquilibriumPolynomialCoefficient0();
		arrayFileInputsResults[12] = myUserInputs.getMw_A();
		arrayFileInputsResults[13] = myUserInputs.getMw_L();
		arrayFileInputsResults[14] = myUserInputs.getMw_V();
		arrayFileInputsResults[15] = myUserInputs.getD_ABV();
		arrayFileInputsResults[16] = myUserInputs.getU_V();
		arrayFileInputsResults[17] = myUserInputs.getP_V();
		arrayFileInputsResults[18] = myUserInputs.getD_ABL();
		arrayFileInputsResults[19] = myUserInputs.getU_L();
		arrayFileInputsResults[20] = myUserInputs.getP_L();
		arrayFileInputsResults[21] = myUserInputs.getRoot_Finder_X_L();
		arrayFileInputsResults[22] = myUserInputs.getRoot_Finder_X_U();
		arrayFileInputsResults[23] = myUserInputs.getRoot_Finder_Tolerance();
		arrayFileInputsResults[24] = myUserInputs.getRoot_Finder_Maximum_Iterations();
		arrayFileInputsResults[25] = myUserInputs.getNumerical_Integration_Number_Of_Steps();
		arrayFileInputsResults[26] = myUserInputs.getOptimizer_Number_Of_Steps_Liquid_Flow_Rate();
		arrayFileInputsResults[27] = myUserInputs.getOptimizer_Minimum_Liquid_Flow_Rate();

		String typeValidationString = "[UserInputs - File Input Data] ";

```

Mai

```
74
75

76
77
78
79
80
81
```

```java

public class MainValidationPart1 {

	public static void main(String[] args) {

		// Validation Method to verify if the inputs found in the values match with
		// values in the input.txt file
		UserInputs myUserInputs = new UserInputs();

		String[] arrayFileInputsString = new String[28];

		arrayFileInputsString[0] = "Choice of Packing Material: ";
		arrayFileInputsString[1] = "Inlet Liquid Flow Rate (kmol/h): ";
		arrayFileInputsString[2] = "Inlet Gas Flow Rate (kmol/h): ";
		arrayFileInputsString[3] = "Inlet Solute Fraction on gas-side (-): ";
		arrayFileInputsString[4] = "Inlet Solute Fraction on liquid-side (-): ";
		arrayFileInputsString[5] = "Recovery Percentage of Solute (%): ";
		arrayFileInputsString[6] = "5th Degree Equilibrium Polynomial Coefficient (-): ";
		arrayFileInputsString[7] = "4th Degree Equilibrium Polynomial Coefficient (-): ";
		arrayFileInputsString[8] = "3rd Degree Equilibrium Polynomial Coefficient (-): ";
		arrayFileInputsString[9] = "2nd Degree Equilibrium Polynomial Coefficient (-): ";
		arrayFileInputsString[10] = "1st Degree Equilibrium Polynomial Coefficient (-): ";
		arrayFileInputsString[11] = "0th Equilibrium Polynomial Coefficient (-): ";
		arrayFileInputsString[12] = "Molecular Weight of the Solute (g/mol): ";
		arrayFileInputsString[13] = "Molecular Weight of the Liquid Solvent (g/mol): ";
		arrayFileInputsString[14] = "Molecular Weight of the Gas Solvent (g/mol): ";
		arrayFileInputsString[15] = "Diffusion Coefficient of Solute Through Gas (m^2/s): ";
		arrayFileInputsString[16] = "Viscocity of Gas (Pa*s): ";
		arrayFileInputsString[17] = "Density of Gas (kg/m^3): ";
		arrayFileInputsString[18] = "Diffusion Coefficient of Solute Through Liquid (m^2/s): ";
		arrayFileInputsString[19] = "Density of Liquid (kg/m^3): ";
		arrayFileInputsString[20] = "Viscocity of Liquid (Pa*s): ";
		arrayFileInputsString[21] = "Root Finder: Liquid Fraction Lower Bound (-): ";
		arrayFileInputsString[22] = "Root Finder: Liquid Fraction Upper Bound (-): ";
		arrayFileInputsString[23] = "Root Finder: Tolerance (-): ";
		arrayFileInputsString[24] = "Root Finder: Maximum Iterations (-): ";
		arrayFileInputsString[25] = "Numerical Integration: Number of Steps (-): ";
		arrayFileInputsString[26] = "Optimizer: Number of Steps for Liquid Flow Rates (-): ";
		arrayFileInputsString[27] = "Optimizer: Minimum Liquid Flow Rates (-): ";

		double[] arrayFileInputsResults = new double[28];


		arrayFileInputsResults[0] = myUserInputs.getPackingMaterialChoice();
		arrayFileInputsResults[1] = myUserInputs.getL_2();
		arrayFileInputsResults[2] = myUserInputs.getV_1();
		arrayFileInputsResults[3] = myUserInputs.getY_A1();
		arrayFileInputsResults[4] = myUserInputs.getX_A2();
		arrayFileInputsResults[5] = myUserInputs.getRecoveryPercentage();
		arrayFileInputsResults[6] = myUserInputs.getEquilibriumPolynomialCoefficient5();
		arrayFileInputsResults[7] = myUserInputs.getEquilibriumPolynomialCoefficient4();
		arrayFileInputsResults[8] = myUserInputs.getEquilibriumPolynomialCoefficient3();
		arrayFileInputsResults[9] = myUserInputs.getEquilibriumPolynomialCoefficient2();
		arrayFileInputsResults[10] = myUserInputs.getEquilibriumPolynomialCoefficient1();
		arrayFileInputsResults[11] = myUserInputs.getEquilibriumPolynomialCoefficient0();
		arrayFileInputsResults[12] = myUserInputs.getMw_A();
		arrayFileInputsResults[13] = myUserInputs.getMw_L();
		arrayFileInputsResults[14] = myUserInputs.getMw_V();
		arrayFileInputsResults[15] = myUserInputs.getD_ABV();
		arrayFileInputsResults[16] = myUserInputs.getU_V();
		arrayFileInputsResults[17] = myUserInputs.getP_V();
		arrayFileInputsResults[18] = myUserInputs.getD_ABL();
		arrayFileInputsResults[19] = myUserInputs.getU_L();
		arrayFileInputsResults[20] = myUserInputs.getP_L();
		arrayFileInputsResults[21] = myUserInputs.getRoot_Finder_X_L();
		arrayFileInputsResults[22] = myUserInputs.getRoot_Finder_X_U();
		arrayFileInputsResults[23] = myUserInputs.getRoot_Finder_Tolerance();
		arrayFileInputsResults[24] = myUserInputs.getRoot_Finder_Maximum_Iterations();
		arrayFileInputsResults[25] = myUserInputs.getNumerical_Integration_Number_Of_Steps();
		arrayFileInputsResults[26] = myUserInputs.getOptimizer_Number_Of_Steps_Liquid_Flow_Rate();
		arrayFileInputsResults[27] = myUserInputs.getOptimizer_Minimum_Liquid_Flow_Rate();


```

```
74
75
76
77              String typeValidationString = "[UserInputs - File Input Data] ";
78
79
80
81
82              for (int i = 0; i < arrayFileInputsResults.length; i++) {
83                      System.out.println(typeValidationString + arrayFileInputsString[i] +
arrayFileInputsResults[i]);
84              }
85
86
87      }
88
89
90
91  }
92
```

```java

public class MainValidationPart2 {

	public static void main(String[] args) {

		// Validation Method to verify if the computed values of UserInputs.class
		UserInputs myUserInputs = new UserInputs();

		String filePathInputString = "File Path of Input file: ";
		String filePathInputResult = myUserInputs.getFilePath();

		String filePathOutputString = "File Path of Output file: ";
		String filePathOutputResult = myUserInputs.getfilepathoutput();

		String isGeneralBinaryMixtureCorrectString = "Boolean of the General Mixture Validation (False or True): ";
		Boolean isGeneralBinaryMixtureCorrectResults = myUserInputs.isGeneralBinaryMixtureCorrect();

		double[] arrayUserInputsComputationResults = new double[12];

		arrayUserInputsComputationResults[0] = myUserInputs.getMyPackingMaterial().calculateS();
		arrayUserInputsComputationResults[1] = myUserInputs.getMyPackingMaterial().getF_p();
		arrayUserInputsComputationResults[2] = myUserInputs.getV_f();
		arrayUserInputsComputationResults[3] = myUserInputs.getL_f();
		arrayUserInputsComputationResults[4] = myUserInputs.getV_2();
		arrayUserInputsComputationResults[5] = myUserInputs.getY_A2();
		arrayUserInputsComputationResults[6] = myUserInputs.getX_A1();
		arrayUserInputsComputationResults[7] = myUserInputs.getL_1();
		arrayUserInputsComputationResults[8] = myUserInputs.getSchmidtGas();
		arrayUserInputsComputationResults[9] = myUserInputs.getSchmidtLiquid();
		arrayUserInputsComputationResults[10] = myUserInputs.getSlopeofOperatingLine();
		arrayUserInputsComputationResults[11] = myUserInputs.getYinterceptofOperatingLine();

		String[] arrayUserInputsComputationString = new String[12];

		arrayUserInputsComputationString[0] = "Column Cross-Sectionnal Area (m^2): ";
		arrayUserInputsComputationString[1] = "Packing Factor (m^(-1))";
		arrayUserInputsComputationString[2] = "Solute-free gas flow rate (kmole/h): ";
		arrayUserInputsComputationString[3] = "Solute-free liquid flow rate (kmole/h): ";
		arrayUserInputsComputationString[4] = "Outlet gas Flow Rate (kmol/h): ";
		arrayUserInputsComputationString[5] = "Outlet Gas Solute Fraction (-): ";
		arrayUserInputsComputationString[6] = "Outlet Liquid Solute Fraction (-): ";
		arrayUserInputsComputationString[7] = "Outlet Liquid Flow Rate (kmol/h): ";
		arrayUserInputsComputationString[8] = "Schmidt Number of Gas-side (-): ";
		arrayUserInputsComputationString[9] = "Schmidt Number of Liquid-side (-): ";
		arrayUserInputsComputationString[10] = "Slope of Operating Line (-): ";
		arrayUserInputsComputationString[11] = "Y_intercept of Operating Line (-): ";

		String typeValidationString = "[UserInputs - Computed Data] ";

		for (int i = 0; i < arrayUserInputsComputationResults.length; i++) {
			System.out.println(
					typeValidationString + arrayUserInputsComputationString[i] + arrayUserInputsComputationResults[i]);
		}

		System.out.println(typeValidationString + filePathInputString + filePathInputResult);
		System.out.println(typeValidationString + filePathOutputString + filePathOutputResult);
		System.out.println(
				typeValidationString + isGeneralBinaryMixtureCorrectString + isGeneralBinaryMixtureCorrectResults);

	}
}
```

```java

public class MainValidationPart3 {
	public static void main(String[] args) {
		// This method verifies the some of the computed values in
		// AbsorptionSimulation.class
		UserInputs myUserInputs = new UserInputs();
		AbsorptionSimulation myAbsorptionSimulation = new AbsorptionSimulation(myUserInputs);

		double[] arrayUserAbsorptionSimulationComputationResults = new double[14];

		arrayUserAbsorptionSimulationComputationResults[0] = myAbsorptionSimulation.getG_X_A2();
		arrayUserAbsorptionSimulationComputationResults[1] = myAbsorptionSimulation.getG_Y_A1();
		arrayUserAbsorptionSimulationComputationResults[2] = myAbsorptionSimulation.getG_L();
		arrayUserAbsorptionSimulationComputationResults[3] = myAbsorptionSimulation.getG_V();
		arrayUserAbsorptionSimulationComputationResults[4] = myAbsorptionSimulation.getG_weightedMW_V();
		arrayUserAbsorptionSimulationComputationResults[5] = myAbsorptionSimulation.getG_weightedMW_L();
		arrayUserAbsorptionSimulationComputationResults[6] = myAbsorptionSimulation.getG_G_V();
		arrayUserAbsorptionSimulationComputationResults[7] = myAbsorptionSimulation.getG_G_L();
		arrayUserAbsorptionSimulationComputationResults[8] = myAbsorptionSimulation.calculate_kxa();
		arrayUserAbsorptionSimulationComputationResults[9] = myAbsorptionSimulation.calculate_kya();
		arrayUserAbsorptionSimulationComputationResults[10] = myAbsorptionSimulation.calculateY_Ai();
		arrayUserAbsorptionSimulationComputationResults[11] = myAbsorptionSimulation.calculateX_Ai();
		arrayUserAbsorptionSimulationComputationResults[12] = myAbsorptionSimulation.logMeanYA();
		arrayUserAbsorptionSimulationComputationResults[13] = myAbsorptionSimulation.logMeanXA();

		String[] arrayUserAbsorptionSimulationComputationString = new String[14];

		arrayUserAbsorptionSimulationComputationString[0] = "Bulk Concentration of A in the liquid phase (-): ";
		arrayUserAbsorptionSimulationComputationString[1] = "Bulk Concentration of A in the Gas Phase (-): ";
		arrayUserAbsorptionSimulationComputationString[2] = "Liquid FLowrate (kmol/s): ";
		arrayUserAbsorptionSimulationComputationString[3] = "Gas FLowrate (kmol/s): ";
		arrayUserAbsorptionSimulationComputationString[4] = "Average Molecular Weight on the Gas-side (g/mol): ";
		arrayUserAbsorptionSimulationComputationString[5] = "Average Molecular Weight on the Liquid-side (g/mol): ";
		arrayUserAbsorptionSimulationComputationString[6] = "Mass Flux on Gas-side (kg/s*m^2): ";
		arrayUserAbsorptionSimulationComputationString[7] = "Mass Flux on Liquid-side (kg/s*m^2): ";
		arrayUserAbsorptionSimulationComputationString[8] = "Film Mass Transfer in the Liquid Phase Multiplied by the Specific Area (kmol/s*m^3): ";
		arrayUserAbsorptionSimulationComputationString[9] = "Film Mass Transfer in the Gas Phase Multiplied by the Specific Area (kmol/s*m^3): ";
		arrayUserAbsorptionSimulationComputationString[10] = "Mole Fraction at Interface in the Gas Phase (-): ";
		arrayUserAbsorptionSimulationComputationString[11] = "Mole Fraction at Interface in the Liquid Phase (-): ";
		arrayUserAbsorptionSimulationComputationString[12] = "Log Mean of the Concentration at Interface in the Gas-side Phase (-): ";
		arrayUserAbsorptionSimulationComputationString[13] = "Log Mean of the Concentration at Interface in the Liquid-side Phase (-):";

		String typeValidationString = "[Non_Optimized - Computed Data] ";

		for (int i = 0; i < arrayUserAbsorptionSimulationComputationResults.length; i++) {
			System.out.println(typeValidationString + arrayUserAbsorptionSimulationComputationString[i]
					+ arrayUserAbsorptionSimulationComputationResults[i]);
		}

	}

}

```

1.1 Appendix F - Ethics Form

Personal Ethics Agreement Concerning University Assignments

We submit this assignment and attest that we have applied all the appropriate rules of quotation and referencing in use at the University of Ottawa, http://web5.uottawa.ca/mcs-smc/academicintegrity/documents/2011/academic-integrity-students-guide.pdf. We attest that this work conforms to the regulations on academic integrity of the University of Ottawa. We understand that this assignment will not be accepted or graded if it is submitted without the signatures of all group members.

Shudip Faiyaz — 7762218

Name, Capital letters — **Student number**

S. F

_____ — 12/10/2018

Signature — **Date**

MOHAMMED SAYEEM — 7493900

Name, Capital letters — **Student number**

M. Sayeem

_____ — 12/10/2018

Signature — **Date**

Ahmed Ibrahim — 6912371

Name, Capital letters — **Student number**

A Ibrahim

_____ — 12/10/2018

Signature — **Date**

Faiyaz Zaman — *7804931*

Name, Capital letters — **Student number**

M. M. Faiyaz

_____ — 12/10/2018

Signature — **Date**

Osama Almazloum — *7333584*

Name, Capital letters — **Student number**

O. A

_____ — 12/10/2018

Signature — **Date**