Development of an Open Source Chemical Process Simulator

Submitted in partial fulfillment of the requirements of the degree of Master of Technology

by

Rahul Jain
(Roll No. 143020028)

Supervisor:
Prof. Kannan M. Moudgalya

Chemical Engineering Department

INDIAN INSTITUTE OF TECHNOLOGY BOMBAY
2016

Approval Sheet

This thesis entitled "Development of an Open Source Chemical Process Simulator" by Rahul Jain (Roll no. 143020028) may be accepted for being evaluated for the award of the degree of Master of Technology.

....
Prof. Kannan M. Moudgalya (Supervisor)

....
Prof. V. Dalvi (Examiner)

....
Prof. Y. Shastri (Examiner)

....
Prof. D. B. Phatak(Chairperson)

Declaration

I, **Rahul Jain**, Roll No. 143020028, declare that the work presented in this report is carried out by my own. Also, wherever someone else's ideas have been used, they have been cited and nothing has been copied verbatim. Due credit has been given to all the contributors and authors, whose ideas and theories reflecting in this project report.

....
Rahul Jain(143020028)

Date: 06 July 2016

Acknowledgements

Abstract

Openmodelica is a state of art equation oriented modeling environment capable of doing both steady state and dynamics simulation but lacks thermodynamics, and good chemical engineering support. DWSIM is a sequential modular steady state simulator with good thermodynamic library (DTL). An attempt is made to make DTL available for Openmodelica in this work. Python-C API and Socket programming approaches are implemented and the performance compared for tasks like steady state and transient simulations regarding chemical engineering systems in Openmodelica. Socket programing performs much better in all the attempted examples. But both the approaches failed when it comes to solving complex chemical engineering processes. Therefore an attempt is made to develop an in-built thermodynamic engine in openmodelica so that thermodynamic equations can be solved simultaneously with mass and energy balance. This built in thermodynamic engine proved to be successful in solving many complex processes with excellent speed and accuracy.

Contents

List of Tables

5

List of Figures

6

Chapter 1

Introduction

A chemical process simulator is a software through which one can perform chemical engineering calculations. As all the calculations are done by a computer, chemical process simulator provides a robust and fast way to carry out the calculations, rather than doing it manually.

Presently many commercial chemical process simulators are available, which can be used for educational and industrial purposes. But as these commercial process simulators are very expensive, only large scale industries and very few academic institutes can a afford it. Aspen plus, Chemcad, Pro II are some of the commercial chemical process simulators presently available in the market. Due to this reason there is a strong need to develop an open source, free chemical process simulator, so that it can be used by small scale industries and academic institutes.

The open source chemical process simulators that are currently available, have some drawbacks, due to which they cannot replace the commercial process simulators. DWSIM is one such steady state process simulator which is open source and has excellent built in thermodynamic and unit operations, but the only drawback is that it is not fit for design and dynamic simulation. Whereas openmodelica is a state of art simulation and modeling environment, which is excellent for design and dynamic simulations but doesn't have a thermodynamic engine and library of unit operations. Therefor if one can successfully develop a thermodynamic engine in openmodelica, it can become an alternate to the commercial chemical process simulators, which is beneficial for both small scale industries and academic institutes. There are two ways of carrying out the above task.

- Integrating the thermodynamic engine of DWSIM with openmodelica. In this process the thermodynamic calculations will be performed by DWISIM where as the unit operation modeling equations which are mostly mass and energy balances will be solved by openmodelica solvers.

- Developing a new thermodynamic engine from scratch in openmodelica itself. This includes the development of compound database and all the thermodynamic routines and VLE packages required for a powerful thermodynamic engine. In this approach all the thermodynamic equations and the unit operations modeling equations will be solved simultaneously by openmodelica solvers.

Chapter 2

Open Source Chemical Process Simulator

2.1 Chemical Process Simulators

2.1.1 What is it ?

Chemical Process simulator is a software that can be used to design, optimize, analyze and develop chemical processes.Chemical Process simulation is a model based representation of unit operations in software. Basic prerequisites are a thorough knowledge of chemical and physical properties of pure components and mixtures, of reactions, and of mathematical models which, in combination, allow the calculation of a process in computer. Chemical process simulation software describes processes in flow diagrams where unit operations are positioned and connected by product streams. The software has to solve the mass and energy balance to reach a stable operating point. The goal of a process simulation is to reach optimal conditions for an examined process. This is essentially an optimization problem which has to be solved in an iterative process. Chemical Process simulation always use models which introduce approximations and assumptions but allow the description of a property over a wide range of temperatures and pressures which might not be covered by real data. Models also allow interpolation and extrapolation within certain limits and enable the search for conditions outside the range of known properties. A chemical process simulator which is free of cost, with its source available and editable is knows as open source chemical process simulator.

2.1.2 Various Components of a Chemical Process Simulator

- **Component Database**
  A component database is a library of all the components and their physical properties. It contains constant and equations for calculation of physical properties at different pressures and temperatures. Some of these physical property constants are Antoine's coefficients, heat capacity, density, heat of formation, critical temperature and pressure. The simulator accesses all the physical prop-

erty data from this database and hands it over to the solver at run time. Some of the commercially available component databases are DCHEMA and Nist. Chemsep and DWSIM are two open source component database.

- **Thermodynamics**
  All chemical process simulator need a library of routines that calculate thermodynamic data such as VLE parameters, dew and bubble point pressures and temps., vapor and liquid enthalpies, viscosities, densities, and other physical and transport properties of pure component and mixtures .Various thermodynamic models have been developed to fulfill this requirement as a single model cannot predict the properties of all the components.All these thermodynamic models are present in thermodynamic engine of a process simulator.It is the thermodynamic engine that decides how robust a simulator is. Choosing the right model is extremely important because using the wrong model, or estimating a physical property incorrectly, can lead to inaccurate results for the overall simulation.

- **Unit Operation Modules**
  Every process simulator has a library of unit operation modules which replicates the physical/chemical operation of the actual process unit of the plant like Heat Exchangers, Distillation Columns, Pumps, Computer compressors, etc. These unit operation modules consists of a set of equations which relates thermodynamic behavior of input stream variables to that to the output stream variables. Most of the the unit operation involves highly non-linear algebraic equation which has to solved by a solver which does it by some numerical iterative method as mostly analytical solution is not possible.

- **Solver**
  The solver solves all the equations that are given to it by the thermodynamic engine and all unit modules included in the flowsheet. Depending on the flowsheet there may be two types of processes namely acyclic and recycle. In acyclic calculation order is generally straight forward and is in the direction of material flow. In recycle processes the flow sheet is divided into a number of single or nested recycle loops and the minimum number of tear streams (can be recycle streams) are identified . The calculation sequence of module is thus determined as a result of this tearing. The solver solves these equations with the help of some numerical iterative method.

2.1.3 Applications of Chemical Process Simulators in Education and Industry

- **Helps in learning chemical engineering concepts.**
  One can generate various parametric graphs and observe the nature for better understanding of the concepts.

- **Visualization of the whole plant.**
  By looking at the flowsheeting section of a simulator one can visualize the mass and energy flow of a plant.

- **Helps in project work.**
  Students can use a simulator to perform calculations related to their projects which otherwise would be very time consuming if done manually. All the thermodynamic and transport properties at various temperatures and pressures can also be extracted which are required in most of the project work.

- **Process plant synthesis.**
  New process can be checked for feasibility study. Comparative studies can also be performed to decide which process would be more efficient for the production of required compound.

- **Estimation of Parameters.**
  Based on the experimental data various parameters can be estimated by the simulator's build in data fitting techniques. For example binary interaction parameters, rate constants of reaction, order of reactions can be predicted.

- **Used for design of unit operations.**
  One can come out with an optimized design for unit operations for a given load.

- **Optimization of Operating Parameters.**
  Simulators can be used to study the effect of changing various parameters (pressure, temperature, flow rate etc.) on plant efficiency and product purity. This eliminates the need of pilot plants, which were quite cost intensive.

- **Designing control strategies.**
  Control strategies can be designed and tested before applying it on the actual plant.

- **Cost estimation.**
  Cost estimation can be done which gives an idea whether the process is economically feasible or what will be the revenue generated.

- **Safety.**
  All the parametric changes can be done on the simulator instead of applying it directly on the plant which may lead to some hazardous or dangerous situations. Safety systems can also be designed on Simulator.

2.2 Why an Open Source Chemical Process Simulator

2.2.1 Expensive for academic institutions

Commercial process simulator are very expensive even for educational purposes. Most of the chemical engineering institutes exclude process simulation course from their curriculum as Commercial simulators are not affordable.

Simulation is a valuable tool to understand the principles of chemical engineering. It will also help simulate large scale, industrial level, problems, which is not possible by paper and pen. A student who is well trained in simulation will be extremely useful to the industry. This will help to improve employment of chemical engineering students.

2.2.2 Expensive for Industry

Commercial process simulators are not affordable for small and medium scale industries. Hiring consulting firms is not an option because presently all consulting firms use commercial simulators and therefore the consulting charges are very high. Therefore most of them are poorly designed, which reduces the efficiency of the plant. Industry can use an open source simulator for resolving their problems or take help from individuals who knows open source simulation.

Small chemical consulting firms can be established which uses open source simulators, to resolve industrial problems.These consulting firms can be started with almost zero investment as they will be using an open source simulator. The consulting charges will also be reasonably low as commercial simulator are not used.

This way both the consulting firms and the manufacturing firms can be mutually helpful to each other.

2.2.3 Amenable to the conduct of large scale training

As open source simulators are freely available, anyone who is an expert can share his/her expertise without any legal constraints. One such powerfull training technique is spoken tutorial.The Spoken Tutorial project is an initiative of National Mission on Education through ICT, MHRD, Govt. of India to promote IT literacy through Open Source Software. The Spoken Tutorial Project is about teaching and learning a particular FOSS (Free and Open Source Software) like Linux, Scilab, LaTeX, PHP, MySQL, Java, C/C++, LibreOffice etc. via an easy Video tool - Spoken Tutorials.

Spoken tutorials possesses an excellent method called side-by-side method which helps in active learning. In this approaches the learner should open the spoken tutorial in a reduced sized window and open the corresponding software side-by-side. This process leads to effective learning. This is possible only with open source software as the learner can easily attain the software locally and can learn it with the corresponding freely available spoken tutorial [18].

Spoken tutorials are available for two open source process simulators namely, DWSIM[1] and Openmodelica[2]. The series of videos available for each simulator, starts from the basic understanding of the software and covers almost all the aspects as we go towards the end of the series.

Chapter 3

Literature Survey

3.1 Available commercial chemical process simulation environments and their thermodynamic engine

3.1.1 gPROMS ModelBuilder

gPROMS ModelBuilder is an environment for modelers to build, validate and execute steady-state and dynamic process models based on the gPROMS second-generation EO modeling platform originally developed at Imperial College London [Barton and Pantelides, 1994; Oh and Pantelides,1996], and subsequently substantially re-architected and re-written at PSE. Following are the thermodynamic engines that can be used in gPROMS to perform chemical process simulations. [3]

- **Multiflash thermodynamic engine**
  KBC Advanced Technologies' Multiflash is the standard gPROMS physical properties package. It is supplied as an integral part of gPROMS Process Builder, along with the 2000+ component DIPPR 801 database.

  Multiflash is a highly rigorous properties package that supports all commonly used thermodynamic and transport properties, including a wide range of equation of state and activity coefficient thermodynamic models.

  Multiflash is specifically designed for equation-orientated modeling, providing tight convergence of internal iterations and analytical partial derivatives with respect to temperature, pressure and composition.

  PSE can supply Multiflash with the DIPPR database supplied by the AIChE, providing component data for thousands of components.

- **OLI electrolyte properties**
  PSE provides OLI Systems' OLI electrolyte properties package for aqueous electrolytic systems. It is a proprietary thermodynamic package and involves separate licensing cost.

- **CAPE-OPEN physical properties**

  gPROMS provides a CAPE-OPEN physical properties socket into which any CAPE-OPEN compliant physical properties package can be plugged for example Aspentech's Aspen Properties, PROSIM's SIMULIS Thermodynamics, Aspentech's COMThermo and many others.

- **gSAFT**

  Statistical Associating Fluid Theory, or SAFT, is an advanced molecular thermodynamic method that can predict a wide variety of thermodynamic properties of mixtures accurately based on physically-realistic models of molecules and their interactions with other molecules.

3.1.2   Aspen Custom Modeler

Aspen-tech' Aspen custom modeler is a modeling and simulation environment to develop customized model. It is an evolution of SPEEDUP, a first-generation EO tool originally developed at Imperial College London [Pantelides, 1988].

The primary purpose of custom modeler is to built customized process models which are not available in Aspen Plus or Apen Hysys. These models can then be imported to Aspen plus or Hysys and can be integrated with the flowsheet.

The custom modeler attains its thermodynamics from aspen properties which is a proprietary thermodynamic engine developed by Aspen-tech. Aspen properties itself uses two popular databases, namely NIST and DCHEMA.

The thermodynamic equations are calculated separately by aspen properties, whereas the modeling equations are solved separately by aspen custom modeler. [4]

3.1.3   CHEMCAD

Chemstations ' CHEMCAD [5] is a chemical process simulator which can simulate steady state as well as dynamic chemical processes. CHEMCAD has its own thermodynamic engine which is primary based on DIPPR (Design Institute for Physical Properties) databank, which is a proprietary databank created in 1978 under the auspices of the American Institute of Chemical Engineers [6].CHEMCAD take all the property constant for the databank based on which thermodynamic routines are written. CHEMCAD has a comprehensive set of thermodynamic properties and packages. Like Aspen plus one can switch between sequential modular approach and equation oriented approach in CHEMCAD.

3.1.4   Simulis Thermdynamics

ProSim' Simulis Thermodynamics [7] is a calculation server for thermophysical properties and phase equilibria calculations on pure components and mixtures. This thermodynamic server is a standalone package which can be plugged in any other application requiring reliable and accurate thermophysical properties. As it is a standalone thermodynamic package when used with other application or in ProSim itself, the

thermodyanmic properties are calculated by Simulis, whereas the modeling equations has to solved separately.

3.2 Open source simulation environment and associated thermodynamic engine

3.2.1 ASCEND

ASCEND is also an open source simulation environment based on equation oriented approach. ASCEND has developed its own thermodynamic engine based on the book 'The Properties of Gases and Liquids' [19] which provides pure component properties for compounds. This thermodynamic engine is named as fprops [8]. The problem with ASCEND thermodynamic is that is has very limited thermodynamic package support for example their is support for only one thermodynamic activity which is UNIQUAC. Another problem is that the compound database is not intensive.

Chapter 4

DWSIM: An Open Source Steady State Process Simulator

4.1 The Sequential Modular Approach

[20]

4.1.1 What is Sequential Modular Approach

DWSIM is based on sequential modular approach. In this approach there are computer programs for all unit operations. One constructs a complete model of these units by writing modeling equations, keeping in mind some assumptions, which closely replicates the actual process units. The simulator then solves the total process flowsheet by calling each unit according to how they are connected in the flowsheet and iterating where loops are encountered. In this approach the variables(pressure, temperature, flows, composition) to input streams to the are fixed and it will compute the variables of the streams leaving the unit. Apart from specifying the input streams completely one has to specify the unit parameters as well, for example the flash unit requires one to specify the temperature and pressure at which the flashing takes place. The degree of freedom should be zero. The solver used for solving the equations is developed so efficiently that it takes some default initial guess values to calculate the solution through some numerical iterative technique. These guess values are then updated and progressively the system converges.

4.1.2 Advantages and Disadvantages of Sequential Modular Approach

- The models of the different units can be prepared and tested separately, while the unit model can be expressed as a computer program subroutine(procedure).

- The solution method for a single unit can be optimized.

- Because of rigid requirements, input data can be quite easily checked for completeness and consistency.

- Because of the modular structure, new units are easily added to the flowsheeting program.

- Less attractive strategy when dealing with design and optimization problems.

4.2 Good collection of Unit Operations and Flowsheeting Applications

4.2.1 Unit Operations in DWSIM

DWSIM has a rich library of built in unit operations which covers almost all the major unit operations used in any chemical industry. These unit modules can easily be inserted into the flowsheet by just dragging them to the flowsheet. The units can then be connected to input and output material streams and energy streams if required. The flowsheet can then be simulated to calculate the output stream properties. Apart from this if one wants to insert a new unit module, DWSIM provides the facility of building your own custom unit operations. For customizing a module DWSIM provides two options.

- **Spread Sheet Unit Operation**
  This module uses a spreadsheet application like excel or Libre office calc to model the unit which can then be inserted directly into the flowsheet. The spreadsheet consists of three sheets namely input, output and calculation. The variable in the input and output sheets are predefined by DWSIM and the user must not edit it. The variables in the input sheet automatically takes the values from the input material stream at run time. All the modeling equations must be inserted in the calculation sheet which takes values from input sheet and return calculated values to output sheet.

- **Custom unit Operation**
  This module uses a programming language where the user can code the required unit module, after which the module can directly be inserted into the flowsheet. The two programming languages which can be used for this purpose are iron python and visual basic.

4.2.2 Flowsheeting Applications in DWSIM

- **Column profiles**
  Column profiles are very important as they give idea of how various parameters like temperature, pressure, composition, flows vary along the length of the column. By studying these profiles one can decide various design aspects of the column like number of trays, the optimal feed stage etc. DWSIM can generate column profiles of distillation, absorption and extractive columns.

- **Sensitivity Analysis**
  Sensitivity analysis is varying an independent variable over a given range and observing the behavior of dependent variable accordingly. This feature helps optimize parameters. For example the behavior of product composition and reboiler duty can be observed by varying the reflux ratio and the number of trays so that one obtain an optimized value of both. Sensitivity analysis is not confined to only a single unit operation, it can be implemented between any two variables of the entire flowsheet.

- **Multivariable Optimizer**
  It is similar to sensitivity analysis but instead of generating a plot it directly gives the optimized value of the parameter. Another advantage of multi variable optimizer is that, here one can define an objective function like cost, efficiency etc. which can be minimized or maximized.

4.3   Thermodynamics

4.3.1   Thermodynamic packages available in DWSIM

DWSIM in enriched with strong thermodynamic engine which can extract physical property data from the two available databases namely Chemsep and DWSIM. It can also extract properties from any other databases which is in .xml format. DWSIM has a rich set of thermodynamic packages which includes ideal gas, equation of state, activity coefficient models. This makes it versatile to simulate any kind of system in DWSIM.

4.3.2   Flash Algorithm

[17] Flash algorithm are thermodynamic procedures which calculates the properties of output streams after the input stream is flashed, for given operation conditions. In principle, flash calculations are straightforward and involve combining the VLE equations with the component mass balances, and in some cases energy balance also. Following are the four flash algorithms one can use in DWSIM according to the process requirement.

- **PT Flash**
  This is the simplest flash algorithms. One has to specify pressure and temperature for implementing it. It involves overall and component mass balance, VLE equations and mole fraction summation equations.

- **PH Flash**
  This is pressure and enthalpy flash algorithm. It involves overall and component mass balance, VLE equations, mole fraction summation equations and energy balance equation.

- **PV Flash**

  This is the pressure and vapor fraction flash algorithm. One needs to specify the pressure and vapor fraction and the algorithm will calculate temperature and composition of the resulting vapor and liquid streams. The dew point and bubble point calculations of any mixture are performed by this flash. For bubble point set the vapor fraction as zero and for dew point set the vapor fraction as one. It involves overall and component mass balance, VLE equations and mole fraction summation equations.

- **PS Flash**

  This is the pressure and entropy flash algorithm. It involves the same equations as of PH flash, but the energy balance equation in PH flash is replaced by entropy balance here.

4.3.3 Thermodynamic Curves and Parameter Estimation

DWSIM offers the facility to plot various thermodynamic plots like VLE curves, critical pressure, ternary diagrams. One can also obtain the data point of these curves if required. DWSIM can also be used to estimate VLE and LLE binary interaction parameters for almost all thermodynamic packages. One need to provide experimental data and based on that the estimation is done.

4.3.4 Standalone Thermodynamic Library

DWSIM provides a standalone thermodynamic library by the name DTL (DWISM Thermodynamic Library). This library can be used externally. This is a very powerful tool as one can use this library in other simulating software where models can be developed but the thermodynamic engine is missing. There are built in routines in DTL which can be used for all thermodynamic calculations. These routines include calculation of all physical properties as function of temperature, pressure and compositions. All flash algorithms discussed above are also included. Parameters like pure component properties, Binary interaction parameters of components can also be accessed just by calling these routines. DTL comes with two default databases which are Chemsep and DWSIM, but one can use his/her own database provided that the database is in .xml format. Users can also add their own compounds to the default database and can also modify the properties of the default database.

4.4 Drawbacks of DWSIM

- **Not Suitable for Design Problems**

  As DWSIM is based on sequential modular approach it can only follow unidirectional calculation order. Only if all the inputs to a unit is completely specified, the output is calculated. It can't follow backward flow which is to specify the desired output variables and calculate the input variable. Therefore some trial and error method has to be used to solve these kind of problems.

Although DWSIM has an adjust operation through which we can solve these kind of problems but still its not efficient and is time consuming as it is based on trial and error approach.

- **Dynamic Processes Cannot be Simulated**
  DWSIM being a steady state simulator cannot handle dynamic process as its solvers are incapable of solving differential equations. This is a major drawback for small scale industries as most of them consists of batch processes which are dynamic and cannot be solved in DWSIM.

Chapter 5

Openmodelica: An Open Source Dynamic Modeling and Simulation environment

5.1 Equation Oriented Approach

5.1.1 What is it?

This approach refers to collecting all the equations describing the behavior of the system and using some numerical technique to solve them simultaneously. Here we provide the equation system in such a way that all the unknown are to the left hand side of the equation, i.e. its given in the form of x=g(x). For solving the system one has to provide initial estimates to all the unknowns.

5.1.2 Advantages and Disadvantages

- Attractive strategy when dealing with design and optimization problems. As all the equations are solved simultaneously it doesn't matter whether the unknown variable in an input or output variable.

- If the system consists of large number of equations, large number of initial estimates must be provided which may not guarantee for convergence.

- Testing of individual units cannot be done as the whole system is solved simultaneously.

- The approach is less realistic as compared to sequential modular.

5.2   Advanced features of Openmodelica

5.2.1   Declarative Language

Openmodelica offers both equation as well as assignment statements. Equation statements are those in which the direction is not defined, which means the variable on the left side of the equal sign need not be the unknown variable. Openmodelica automatically determines the unknown variable. This makes life easy as implicit functions can directly be written without converting it into explicit form. This feature is rarely available in any other simulation environments. The makes openmodelica library classes more reusable than traditional classes containing assignment statements where the input output causality is fixed. Example the resister equation: R*i = v, can be used in three different ways: i := v/R, v := R*i, R := v/i [12]. Not only the statements but the direction of flow of data need not be specified by the programmer. Think of it as an energy port that allows energy to flow from whatever happens to be the place of higher energy to that where it is lower. This is known as accusal modeling. A user would be able to build up a graphical network from function blocks, then the software builds the mathematical model, he states. Whereas a conventional modeler might model a resistance-capacitance-inductance circuit as a circuit, Openmodelica would model each individual component separately and connect the three without worrying about the flow of data[12].

5.2.2   Object Oriented Language

Openmodelica is a strongly typed object oriented language with a general class concept. Individual generic models can be written which can then be reused and inherited. These models can be directly dragged in any simulation and then connected to the respective location. The initial values and the guess values of the variables of these models can be explicitly specified according to the requirement of the process without changing the actual code of the model. Therefore the user has to code only once and that piece of code can be reused.

Being an object oriented language openmodelica also supports inheritance through which a child class can inherit the properties of a parent class. Now the child class can implement the same behavior of the parent class.

5.2.3   Graphical Modelling

Openmodelica permit users to attach their own graphics to the models. This helps in better visualization of the system. Unlike other simulation software, which exhibits just on signal flow model modelica exhibits a physical model which is easy to understand and visualize. Due to the acausal feature the user just have to connect the different models involved in the flowsheet without worrying about the signal flow. Figure 5.1 show a comparative system of openmodelica versus simulink. Openmodelica offers an integrated modeling and simulation environment where one can first model the unit and then connect them with each other graphically. This relieves the user of
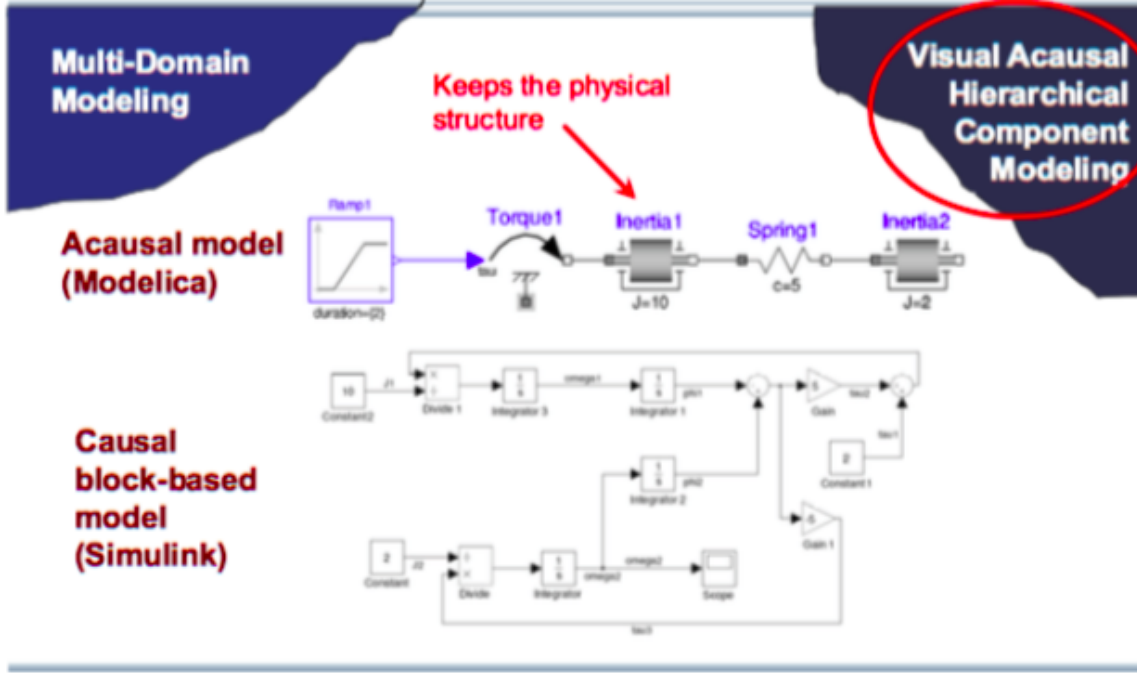
Figure 5.1: Comparison of graphical environment of Openmodelica and Simulink [12]

writing the connecting equations and codes. Openmodelica has a special feature for establishing inter model connection, named connectors. Connector is a special type of class through which users can define what are the incoming or outgoing variables from a model.

Following are two types of variables that can be defined in a connector.

- **Flow variables** Flow variables are those variables comes in or goes out of a model example the current flowing through a resister. In openmodelica flow variables follow the Kirchhoff's current law, which means the total amount of any quantity coming into any node is equal to the amount of quantity going out of that node.

- **Potential variables** Potential variables are variables that replicates the voltage in an electrical circuit. In openmodelica, potential variables follow Kirchhoff's voltage law, which states that the directed sum of the variables around any closed loop should be zero.

Openmodelica offers an excellent graphical facility which is drag and drop composition. Using this one can easily drag and drop models to the simulation environment and connect them through a mouse cursor. Figure 5.2 depicts this feature of open modelica.

5.3 Drawbacks

- As modelica is a generic modeling and simulation environment it has no built

24
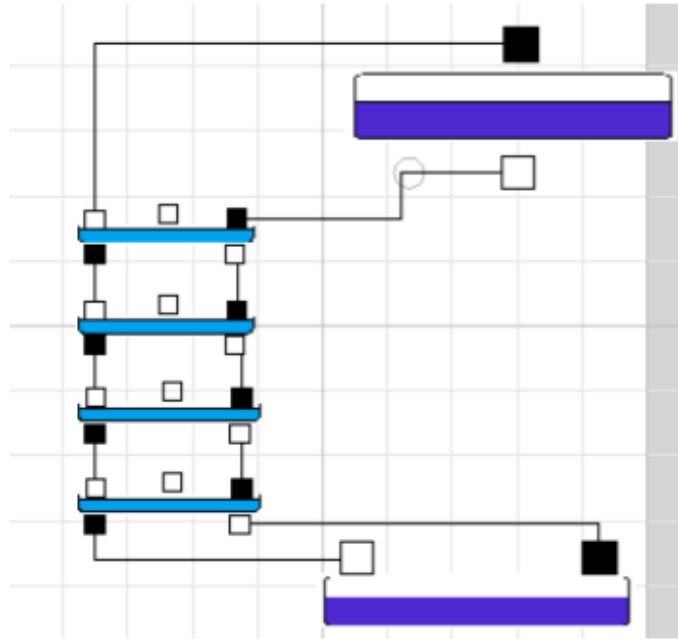
Figure 5.2: A distillation tower model as a connection of tray model in openmodelica

in chemical engineering unit operations.

- Openmodelica does have a thermodynamic engine. Therefore thermodynamic calculations cannot be performed.

- As every unit operation has to be coded user has to provide the solver with good initial guess values otherwise the model may not converge.

Chapter 6

Importing the Thermodynamic engine of DWSIM in Openmodelica

6.1   Motivation

- **Design and Optimization**
  As shown in chapter three DWSIM is not efficient in handling design and optimization problems because it is based on sequential modular approach but has a strong thermodynamic engine. As described earlier DWSIM also has a standalone thermodynamic library (DTL) which can be used externally. On the other hand openmodelica is apt in handling equations but lacks thermodynamics. Therefore, if somehow the thermodynamics of DWSIM is imported to openmodelica, both these problems can be solved.

- **Dynamic Simulation**
  As stated in chapter three DWSIM cannot simulate dynamic systems. On the other hand openmodelica can do this very efficiently due to its strong built in integration solvers. Again if DTL is clubbed with openmodelica, simulation of dynamic chemical processes can easily be carried out.

The above two benefits shows that it may be a fruitful exercise to integrate the DTL with openmodelica. As a result one may obtain a complete process simulator which is capable of solving all type of problems that is simulation, design, optimization with steady state or dynamic behavior.

6.2   Python-C API approach to Integrate Openmodelica with DTL

[9] DTL consists of a file with an extension .dll (dynamic link library) which is written in VB.NET (windows environment). This file is COM (component object model) enabled, which means that any programming language which supports COM can import this library and access the built in thermodynamic subroutines. Openmodelica
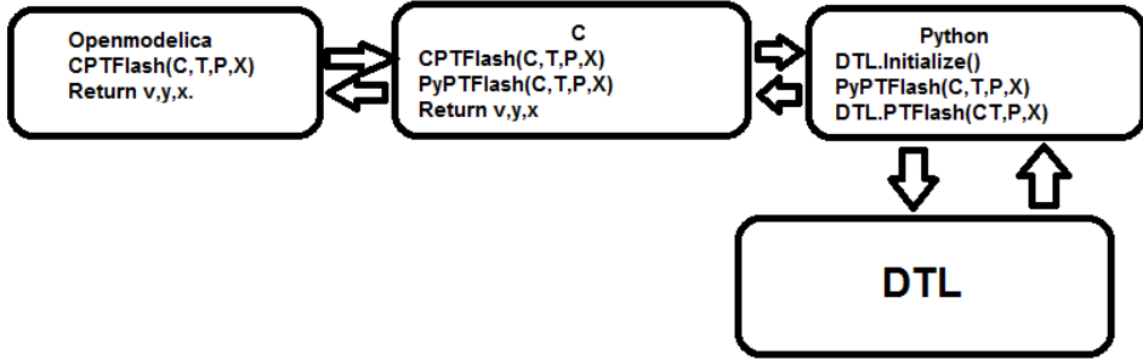
Figure 6.1: Structure of Python-C API approach

is written in C (Linux environment) and the only language whose routines can be called from openmodelica is C. As both these programs are developed in so different environment it becomes hard to integrate the two. As openmodelica supports only C language, it would have been best to call DTL from C but unfortunately due to the difference in their environment both were not compatible. Therefore, there was a need to bring in some glue language which can establish this integration from DTL to C. For this purpose python was chosen as it is an excellent glue language.

The package **win32com** in python can be used to establish connection with COM enabled objects. Therefore, using win32com we can call the DTL library and hence all the thermodynamic routines available in it.

Python-C API is one of the approach through which this integration of openmodelica and DTL is carried out successfully. Figure 6.1 describes the flow of the approach. Following points describes the structure and flow of this approach A.

- First the DTL routines were imported to python. A package in python named win32com.client was used to do that. This package allows python to call routines from a dll file which is registered in the windows registry. Once the dll file is dispatched through win32com.client, python has access to all the COM enabled functions of the dll.

- Once DTL is imported to python now python functions were written to send input parameters to DTL routines and calculate the required thermodynamic properties. These python functions now behaves similarly as DTL routines. The codes for python functions are shown in appendix A.

- These python functions are now called by C through python-C API. In computer programming, an API (Application programming interface) is a set of routines, protocols, and tools for building software applications. An API expresses a software component in terms of its operations, inputs, outputs, and underlying types. This API is responsible for converting C variables to python and vice versa. The codes for C functions are shown in Appendix A

27
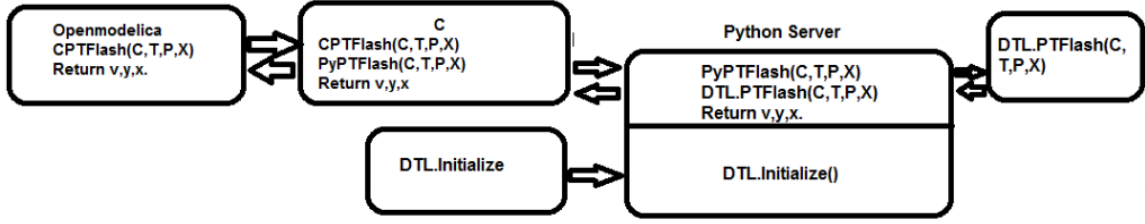
Figure 6.2: Structure of Python-C socket approach

- Finally as openmodelica is compatible with C, the inputs are then sent to C functions through openmodelica external C functions, which in turn calls the python functions, which in turn calls DTL routines. The codes for external function in openmodelica are shown in Appendix A.

6.3 Client-Server (sockets) approach to integrate OpenModelica with DTL

[10] Client-Server or socket approach is another approach through which the integration is successful. Figure 6.2 describes the data flow of the approach. Following points describes the structure and flow of this approach B.

- In this approach also, firstly the the DTL routines are called in python with the help of win32com.client package.

- Similar to python-C approach, functions are written in python which calls DTL to calculate various physical properties.

- Now a python server which consists of all the above functions is created which waits for a C clients to establish connection, receive inputs from it and send the calculated values back to the client. The codes for python server in shown in Appendix B

- Clients are coded in C which establishes connections with the python servers and send and receive data from it. The codes for C client is shown in Appendix B

- Once the connection is established the python server receives the data from C client, contacts DTL, calculate the required property as asked by the C client and sends it back to the client.

- Finally the these C clients are called by openmodelica external C functions giving the required inputs to the client which in turn contacts the python servers for the calculation. The C client receives the calculated values from python servers and transfers it to openmodelica.

28

Table 6.1: Thermodynamic routines and the procedures to call them

Thermodynamic Prop	Thermodynamic Funct	Arguments
Vapur Pressure	VapPres	Comp,T
Enthalpy	Ent	Comp., T,P
Liquid Density	LiqDen	Comp,T
Vapour Density	VapDen	Comp,T
Pres. Temp. Flash	PTFlash	Comp,Z,T,P,Model
Pres. Volume. Flash	PVFlash	Comp,Z,V,P,Model
Pres. Enth. Flash	PHFlash	Comp,Z,H,P,Model
Liquid Viscosity	LiqVis	Comp,T
Vapour Viscosity	VapVis	Comp,T
Surface tension	SurfTen	Comp,T

6.4 Implementation of the two approaches

Table 6.4 lists the thermodynamic functions and their that can be used in open-modelica to receive values from DTL. These functions can be used directly in any simulation.

When, using the socket approach the python server should be up and running during the execution of the simulation.

These thermodynamic functions can be used along with the mass and energy balance written in openmodelica to solve the unit operation modules.

6.5 Comparison of the Two Approaches

The following two models were simulated in openmodelica with different integration approaches to import DTL.

- **Steady State Flash Separator**
  An equimolar mixture of Benzene and Toluene was flashed in a flash separator. The thermodynamic package used was Raoult's law. All the pure component and mixture properties were imported from DTL. To test the capability of the integration methods the composition of the resulting vapor stream was specified, and the temperature at which this composition is attained was left unknown. It was observed that the **Python-C API** approach took thirty seconds to solve the system, whereas the **Client-Server** method took less than one second to simulate.

- **Dynamic Flash**
  A dynamic flash was simulated with the feed as equimolar mixture of benzene and toluene. The thermodynamic package used was again Raoult's Law. It was assumed that the output liquid stream was at the same composition and temperature, as the holdup inside the flash separator. Heat supplied to flash

separator was kept constant. The set of equations involved were mass balance, energy balance and equilibrium equations. The mass and energy balance were differential equations. It was observed that the Python-C API approach took **thirty minutes** to solve, whereas the Client-Server approach takes **four minute** to solve the system.

The above two examples clearly states that Client-Server approach is much faster than Python-C API approach. This is because of two reasons.

- Before you call any routine of DTL one has to call another DTL routine named Initialize. This routine loads all the compounds and their properties from the database, which consumes lot of time. In Python- C API approach this initialize routine was called every time a call was made from openmodelica to C and from C to modelica. When a model is simulated in openmodelica using DTL through Python-C API approach, on each iteration such a call was made and every time the Initialize routine was called. This repeated loading of the database through initialize make the process very slow. Whereas in Client-Server approach, before the server is ready to accept connections from client the initialize routine is already called. Therefore the initialize routine was only called once and not on every single iteration.

- Whenever API is used in any program it makes it slow as there are lots of conversion involved like data type conversions etc. As python-C API approach is based on API it is slow.

Chapter 7

Development of a built in Thermodynamic engine in Openmodelica

7.1   Motivation

The following three reasons justifies the requirement of a built in thermodynamic engine in openmodelica.

- The two approaches as described in the previous chapter imports DWSIM's thermodynamic engine in openmodelica. This means that the thermodynamic engine is sitting outside openmodelica and has to be called each and every time a thermodynamic property is to be calculated. This process is too time consuming, thus making it impossible to solve complex models involving rigorous thermodynamics calculations.

- As DWSIM's thermodynamic is external to openmodelica, it appears to be a black box to openmodelica, where inputs are send and calculated values are obtained which are solved by DWSIM's solver. Openmodelica is unable to access the thermodynamic equations and therefore it is totally unaware of the nature of these equations. On the other hand if these equations are available in openmodelica itself, they can be solved simultaneously with mass and energy balance thereby solving all the equations simultaneously using openmodelica's own solver. This seems to be lot faster as openmodelica now knows the nature of these equations and they are solved internally by openmodelica itself.

- Another key point to be considered is as described in chapter four, openmodelica has a rich set of nonlinear and ODE solvers which are written in C. This makes the solving process too fast. Whereas DWSIM solvers are written in C# which is slower than C in terms of computation time required. Consequently, the calculation of thermodynamic properties , which are highly nonlinear equations, is much faster in openmodelica than DWSIM.

- The problem with DTL approach is that as DTL is a dll file it is supported only in windows. Infact the concept of COM is only supported by windows. Therefore, this approach dosen't work in other operation systems like linux and mac. Whereas, the build in thermodynamics is a part of openmodelica itself and openmodelica is compatible with every operating system. Therefore the built in thermodynamic should be multi platform.

7.2 Development of the three components of a thermodynamic engine

7.2.1 Development of Compound Database

- **The Chemsep Database**
  Chemsep is an open source database, written in **xml** format, of over six hundred compounds with a comprehensive set of thermodynamic properties of each compound. It also has an extensive database of binary interaction parameters for thermodynamic packages like NRTL, Peng Robinson, UNIQUAC, SRK etc. Most of the thermodynamic properties are calculated by using empirical equations which are functions of temperature or pressure. Chemsep database includes the constants which are used in these equations. Therefore, chemsep database is a comprehensive database that can be used to built a powerful and robust thermodynamic engine.

- **Porting Chemsep database into Openmodelica**
  As described in the previous section chemsep database is written in xml format. This xml data has to be re written in such a form which can be understood by openmodelica. Therefore, each compound (including all its thermodynamic properties) is replicated as a single class in openmodelica as shown in figure 7.1. The properties are given abbreviations (as shown in table 7.1) so that they can be called conveniently. The conversion from xml to openmodelica classes is carried out by developing a python script (see Appendix C) which automates this process, thus making it fast and robust.

  Now, one can extract any independent property of a compound by the **.** operator followed by the property relevant abbreviation. For example the critical temperature (Tc) of methane can be accessed by **Methane.Tc**. Similarly, all the properties of any compound can be accessed in the same way as shown in table 7.1.

7.2.2 Development of Thermodynamic Functions

As discussed earlier most of the thermodynamic properties of pure compounds which depends on temperature or pressure are in the form of equations which includes constants and an independent variable (temperature and pressure). These properties are
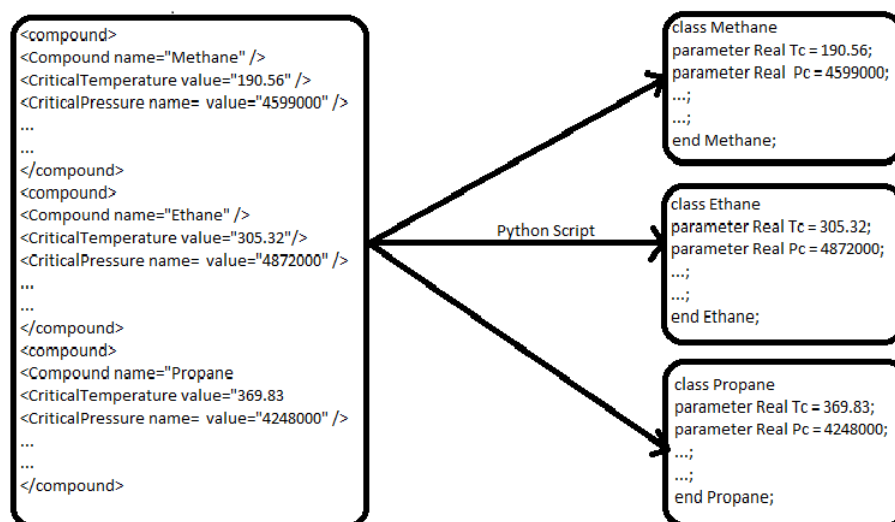
Figure 7.1: Porting chemsep database in Openmodelica

Table 7.1: Independent thermodynamic properties and openmodelica routines to call them.

Thermodynamic Property	Calling procedure
Critical Temperature	Compound.Tc
Critical Pressure	Compound.Pc
Critical Volume	Compound.Vc
Boiling point	Compound.Tb
Melting poind	Compound.Tm
Molecular weight	Compound.MW
Accentric Factor	Compound.AF
Triple Point	Compound.TT
Solubility parameter	Compound.SP
Dipole moment	Compound.DP
Heat of formation	Compound.HOF
Absolute enthalpy	Compound.ABSENT
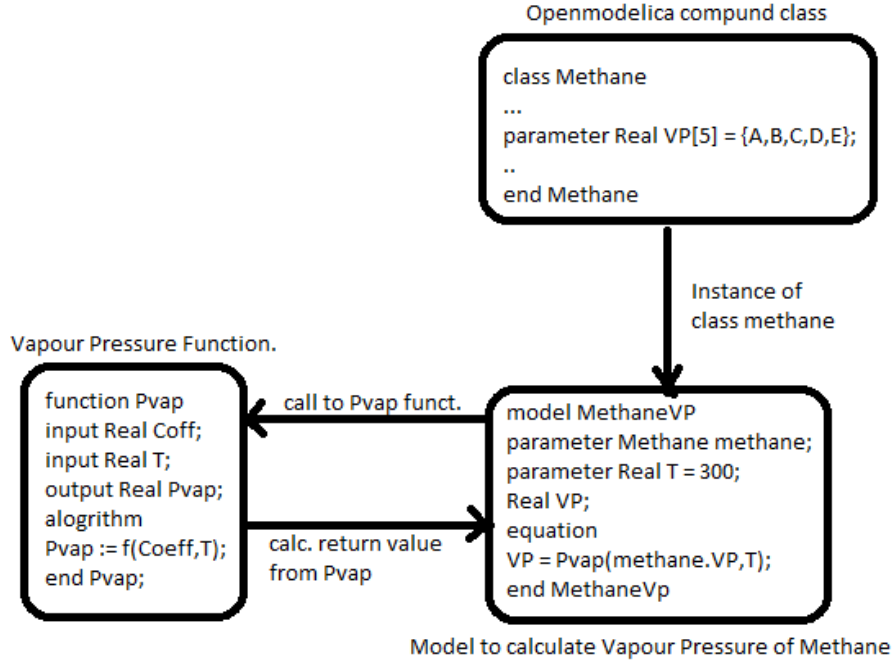
Figure 7.2: Using built in thermodynamic functions (Pvap)

written in the form of functions in openmodelica. The arguments to these functions are the independent variable (temperature, pressure, composition) and the coefficients of the respective compound whose properties have to be calculated. The coefficients can be accessed by instantiating the base compound class. The functions then returns the calculated property. For example the vapor pressure of methane at 300 K can be calculated by first instantiating the base methane class and then calling **Pvap(water.VP, 373)**. Where Pvap is a generic function to calculate the vapor pressure of any compound at any given temperature. The whole process is shown in figure 7.2. Similarly, all other thermodynamic properties can be calculated using their respective functions as shown in table 7.2.

7.2.3 Development of Phase Equilibria models

In a mixture which finds itself in a vapor-liquid equilibria state (VLE), the component fugacities are the same in all phases, that is :

$$f_i^L = f_i^V$$

The fugacity of a component in a mixture depends on temperature, pressure and composition. in order to relate fiV with temperature, pressure and molar fraction, we define the fugacity coefficient,

$$\Phi_i = \frac{f_i^V}{y_i P^*}$$

34

Table 7.2: Dependent thermodynamic properties and openmodelica functions to call them.

Thermodynamic Property	Calling procedure
Liquid density	LiqDen(Compound name,temperature)
Vapor pressure	VP(Compound name,temperature)
Heat of Vaporization	HOV(Compound name,temperature)
Liquid heat capacity	LiqCp(Compound name,temperature)
Liquid viscosity	LiqVis(Compound name,temperature)
Vapor viscosity	VapVis(Compound name,temperature)
Liquid thermal conductivity	LiqK(Compound name,temperature)
Vapor thermal conductivity	VapK(Compound name,temperature)

which can be calculated from PVT data, commonly obtained from an equation of state. For a mixture of ideal gases, $\Phi_i = 1$. The fugacity of the i component in the liquid phase is related to the composition of that phase by the activity coefficient γ_i, which by itself is related to x_i and standard-state fugacity f_i^0 by

$$\gamma_i = \frac{f_i^L}{x_i f_i^0}$$

The standard state fugacity f_i^0 is the fugacity of the i-th component in the system temperature, i.e. mixture, and in an arbitrary pressure and composition. Here, the standard-state fugacity of each component is considered to be equal to pure liquid i at the system temperature and pressure. An Equation of State is used to calculate equilibria, fugacity of the i-th component in the liquid phase is calculated by

$$\gamma_i = \frac{f_i^L}{x_i P^*}$$

with the fugacity coefficient Φ_i calculated by the EOS, just like it is for the same component in the vapor phase. The following EOS models are developed in openmodelica [17].

The binary interaction parameters for each of the EOS and acitivity coefficient models have been extracted from chemsep database where they are saved in a .dat file. The following procedure is used to port all the binary interaction parameters to openmodelica.

- First, the dat file is converted to a csv file as python is very much comfortable with csv format. To accomplish this sublime editor is used which makes this task simple enough.

- This csv is then converted to an openmodelica function by a python script (Appendix C) which converts the compound and the binary interaction parameters as an array.
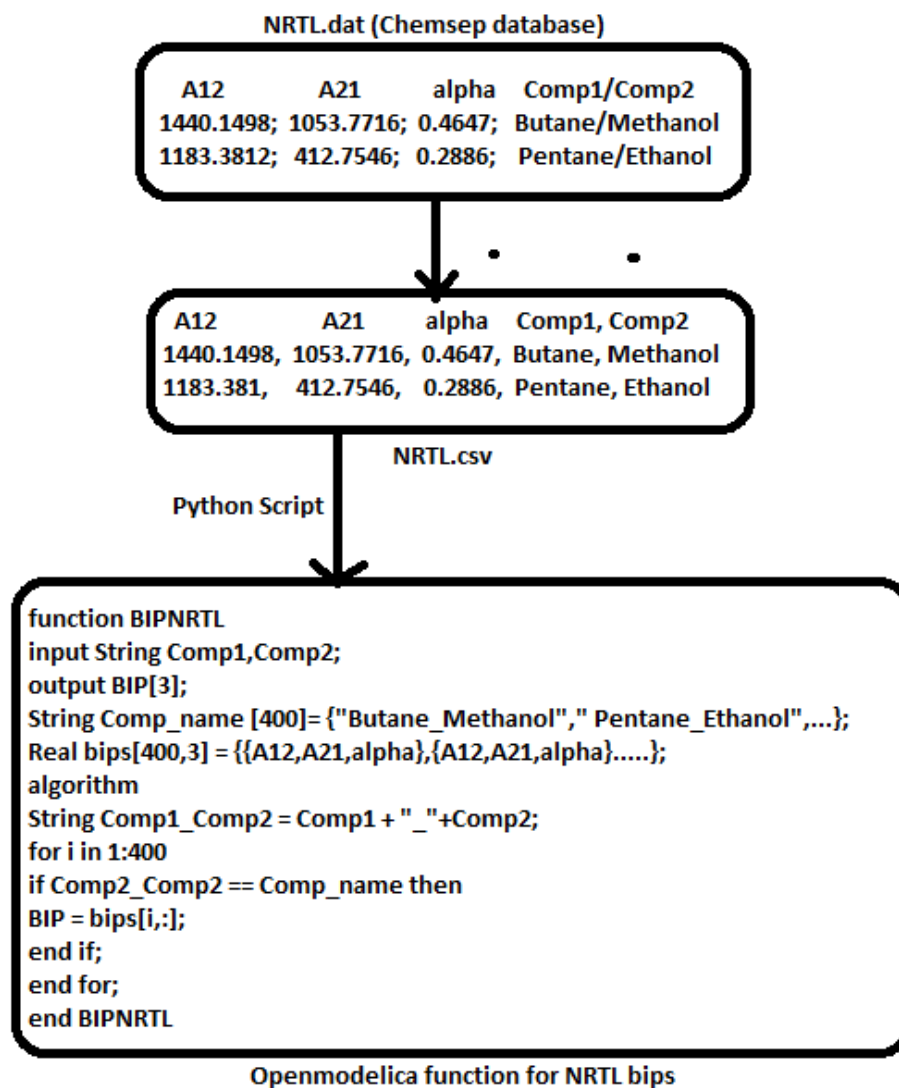
Figure 7.3: Porting Chemsep's binary interaction parameters in openmodelica

Figure 7.3 demonstrate the above process for NRTL activity coefficient model.

- **Peng Robinson**
  The following equations represent the Peng Robinson model [11].

$$a(T) = [1 + (0.37464 + 1.54226\omega - 0.2699\omega^2)(1 - T_r^{(1/2)})]^2 0.45724(R^2 T_c^2)/P_c$$
$$b = 0.07780(RT_c)/P_c$$

  where
  ω acentric factor (obtained by Compound.W)
  T_c critical temperature
  P_c critical pressure
  T_r reduced temperature, T/T_c

$$a_m = \sum_i \sum_j x_i x_j \sqrt{a_i a_j}(1 - k_{ij})$$
$$b_m = \sum_i x_i b_i$$

  where
  $x_{i,j}$ molar fraction of the i or j component in the phase (liquid or vapor)
  $a_{i,j}$ i or j component a constant
  $b_{i,j}$ i or j component b constant
  $k_{i,j}$ binary interaction parameter which characterizes the i-j pair

  The fugacity coefficient obtained with the Peng-Robinson EOS [17] is given by

$$ln\frac{f_i}{x_i P} = \frac{b_i}{b_m}(Z - 1) - ln(Z - B) - \frac{A}{2\sqrt{2}B}\left(\frac{\sum_k x_k a_{ki}}{a_m} - \frac{b_i}{b_m}\right)ln\left(\frac{Z + 2.414B}{Z - 0.414B}\right)$$

  where Z is the phase compressibility factor and can be obtained from

$$Z^3 - (1 - B)Z^2 + (A - 3B^2 - 2B)Z - (AB - B^2 - 2B) = 0$$
$$A = \frac{a_m P}{R^2 T^2}$$
$$B = \frac{b_m P}{RT}$$

- **Soave-Redlich-Kwong Equation of State**
  The Soave-Redlich-Kwong Equation [13] is also a cubic equation of state in volume.
  The a and b parameters are given by

$$a(T) = [1 + (0.48 + 1.574\omega - 0.176\omega^2)(1 - T_r^{(1/2)})]^2 0.42747(R^2 T^2)/P_c$$
$$b = 0.08664(RT_c)/P_c$$

The fugacity is calculated by

$$ln\frac{f_i}{x_iP} = \frac{b_i}{b_m}(Z-1) - ln(Z-B) - \frac{A}{B}(\frac{\sum_k x_k a_{ki}}{a_m} - \frac{b_i}{b_m})ln(\frac{Z+B}{Z})$$

The phase compressibility factor Z is obtained from

$$Z^3 - Z^2 + (A - B - B^2)Z - AB = 0$$
$$A = \frac{a_m P}{R^2 T^2}$$
$$B = \frac{b_m P}{RT}$$

- **NRTL**
Renon and Prausnitz [14] developed the NRTL equation (Non-Random, Two-Liquid) based on the concept of local composition but, unlike Wilson model, the NRTL model is applicable to systems of partial miscibility. The model equations are:

$$ln\gamma_i = \frac{\sum_{j=1}^{n} \tau_{ji} x_j G_{ji}}{\sum_{k=1}^{n} x_k G_{ki}} + \sum_{j=1}^{n} \frac{x_j G_{ij}}{\sum_{k=1}^{n} x_k Gkj}(\tau_{ij} - \frac{\sum_{m=1}^{n} \tau mj x_m G_{mj}}{\sum_{k=1}^{n} x_k G_{kj}})$$
$$G_{ij} = exp(-\tau_{ij}\alpha_{ij})$$
$$\tau_{ij} = \frac{\alpha_{ij}}{RT}$$

γ_i Activity coefficient of component i
x_i Molar fraction of component i
α_{ij} Interaction parameter between i-j ($a_{ij} = aji$) (calculate/mol)
T Temperature (K)
α_{ij} non-randomness parameter for the i-j pair ($a_{ij} = aji$)

Chapter 8

Implementation of the built in thermodynamics in Openmodelica

The following example demonstrates the implementation of the built in thermodynamics in openmodelica.

8.1 Steady state flash of ethanol-water system

8.1.1 The Model

An equimolar feed containing ethanol and water is to be flashed at 1 atm and 355 K. The resulting vapor and liquid compositions are to be calculated. Following are the modeling equations for the same.

Mass balance

$$z_i F = x_i L + y_i V$$

Where F,L,V feed,liquid,vapour flowrate resp. (100kmol/hr) and z_i, x_i, y_i feed,liquid,vapour compositions respectively.
Equilibrium equations

$$y_i = K_i x_i$$

Where K_i is the equilibrium constant obtained from UNIQUAC model.
Summation Equation

$$\sum_{i=1}^{2} y_i = 1$$

UNIQUAC activity coefficient model is used as the phase equilibria model.
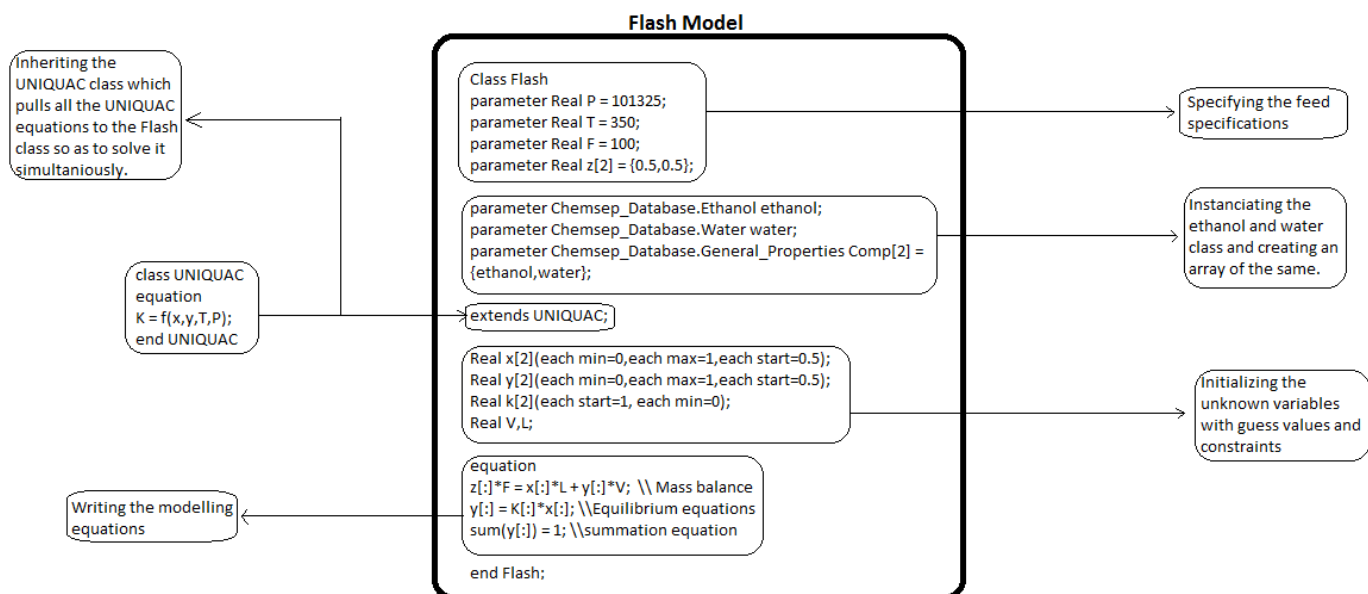
Figure 8.1: Openmodelica code for ethanol-water flash.

8.1.2 Openmodelica code and its explanation

Figure 8.1 describes the code used to simulate the flash. The first four lines specifies variables which carries the feed specifications (temperature, pressure, composition, flowarate). The next set of lines instantiate the compound class ethanol and water and creates an array of the same. The next line **extends UNIQUAC** is used to inherit the already developed UNIQUAC class (which has all the equations for UNIQUAC model). By inheriting a class all the equations and variables of the inherited class are actually dragged and added to the class which inherits it. Therefore, all the UNIQUAC equations are added to the equation section of the Flash class. The next set of variables initializes and defines the unknown variables that needs to be calculated. Finally, the equation section has all the modeling equations required for the simulation of the flash.

8.1.3 How openmodelica handles and solves the model equations

As described in chapter 5, openmodelica follows an equation oriented approach where all the equations are solved simultaneously. Therefore, for chemical engineering problems, openmodelica collects all the model equations along with the thermodynamic equations and then solves them simultaneously. The fact that thermodynamic equations are solved simultaneously along with the modeling equations is demonstrated by the following example.

A simplest flash model is simulated for an ideal system of benzene and toluene. The feed is an equimolar mixture of benzene and toluene, which is flased at 1 atm. As the system is ideal the VLE model is taken to be Raoult's law. The unknown
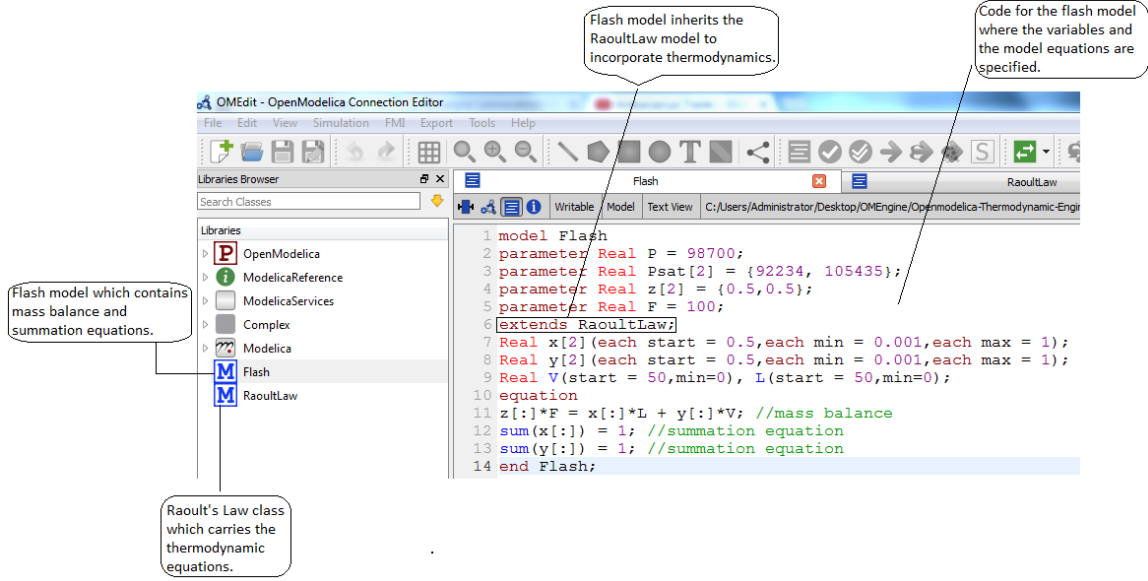
Figure 8.2: Openmodelica code for ideal flash model.

variables that needs to be calculated are compositions and flowrates of liquid and vapor streams. Following are the modelling equations for the flash model.

Mass balance

$$z_i F = x_i L + y_i V \tag{8.1}$$

Summation Equation

$$\sum_{i=1}^{2} y_i = 1 \tag{8.2}$$

Raoult's law

$$y_i P = x_i P_i^{sat} \tag{8.3}$$

Where F,L,V feed,liquid,vapour flowrate resp. (100kmol/hr) and z_i, x_i, y_i feed,liquid,vapour compositions respectively. P_i^{sat} is the saturated vapour pressure

Figure 8.2 shows the openmodelica code for the flash model. It can be seen that this model involves only mass balance and summation(equations 8.1 and 8.2). To incorporate raoult's law, another class named as **RaoultLaw** is inherited by the flash model. Figure 8.3 shows the openmodelica code for this class. It can be seen that this class involves only the thermodynamic equations (equation 8.3). Therefore, at this stage the thermodynamics and the model equations are in separate classes.
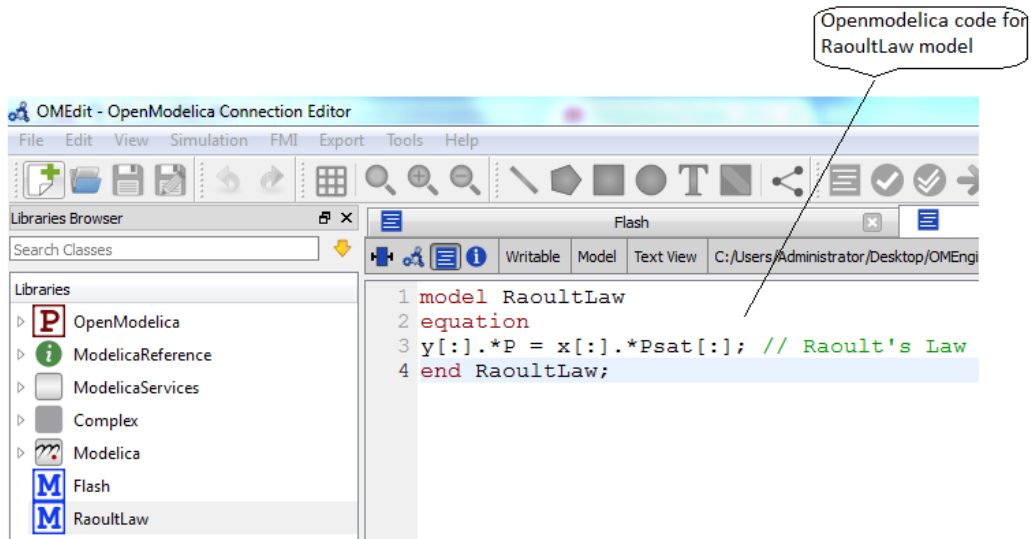
Figure 8.3: Openmodelica code for Raoult's Law thermodynamic model

Before simulating a model, openmodelica instanciate it by itself. It is at this stage that openmodelica's compiler collects all the equations and dumps them in the **equation section** of the instantiated model. Figure 8.4 shows the instance of the flash model as generated by openmodelica. It is clear that now this model involves both the modeling equations and the thermodynamic equations.

After the instance of the main model is generated, openmodelica start solving all the equations present in the **equation** section of the instantiated model. By default openmodelica uses **Dassl** to solve differential algebric equations and a hybrid solver for solving algebraic equations. However, we can choose a different solver from the list of solvers. This is achieved by going in the simulation setup window, where one can specify all the details regarding the simulation for example the time period, the number of intervals for simulation etc. The output of openmodelica's successful simulation is a mat or a csv file.
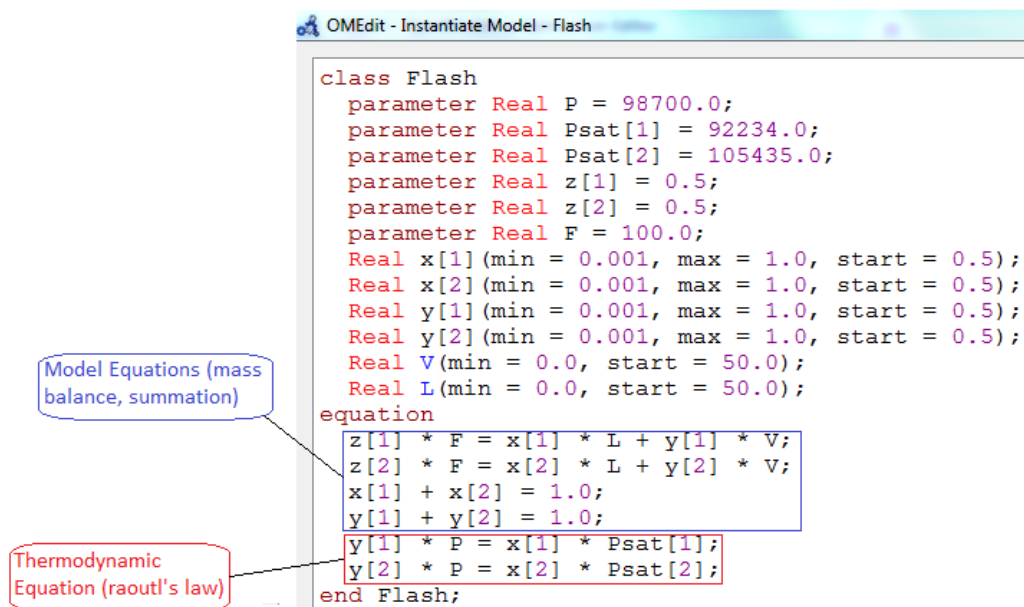
Figure 8.4: Combined model generated by openmodelica

Chapter 9

Solved examples in Openmodelica using the built in thermodynamics

9.1 Generating VLE curve (Txy) for ethanol water system using the UNIQUAC model

- **Bubble point**
  Modified Roult's law (equations 9.1 and 9.2) is the governing equation to generate the bubble point curve.

$$y_1.P = \gamma_1.x_1.Pvap_1 \tag{9.1}$$
$$y_2.P = \gamma_2.x_2.Pvap_2 \tag{9.2}$$

where γ_1 and γ_2 are activity coefficients. y_1, y_2 are vapor phase compositions. x_1, x_2 are liquid phase compositions and $Pvap_1$, $Pvap_2$ are corresponding vapor pressures.

Adding the above two equations and putting $y_1 + y_2 = 1$ we get

$$P = \gamma_2.x_2.Pvap_2 + \gamma_1.x_1.Pvap_1 \tag{9.3}$$

Here γ_1 and γ_2 are complex nonlinear functions of temperature and liquid compositions and $Pvap_1$ and $Pvap_2$ are functions of temperature

The pressure is kept constant at 1 atm. The value of x_1 (mole fraction of ethanol) is varied from 0 to 1 with an interval of 0.1 and for each value of x_1 the corresponding value of temperature is calculated by equation 9.3. Figure 9.1 shows the resulting tx curve as calculated by the developed built in thermodynamics in openmodelica.

- **Due point**
  Modified Roult's law (equations 9.1 and 9.2) is the governing equation to generate the dew point curve.
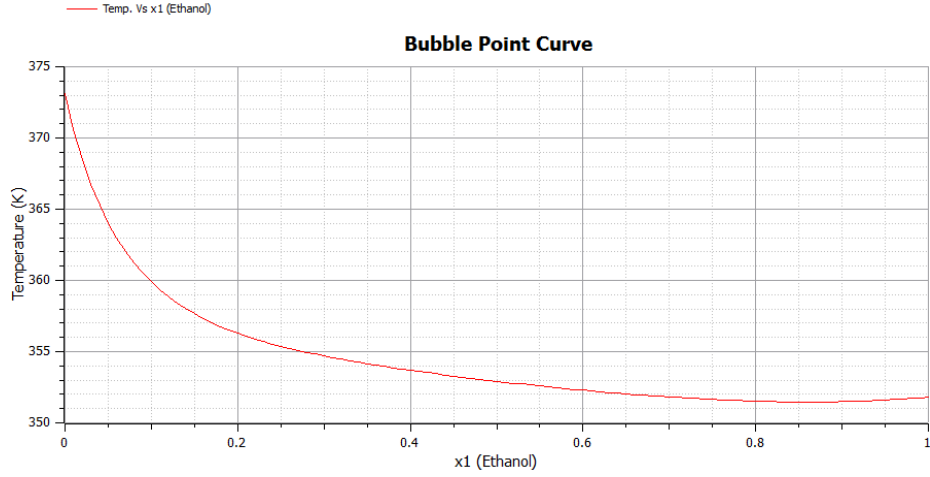
Figure 9.1: Bubble point (Tx) curve for ethanol water system generated in openmodelica.
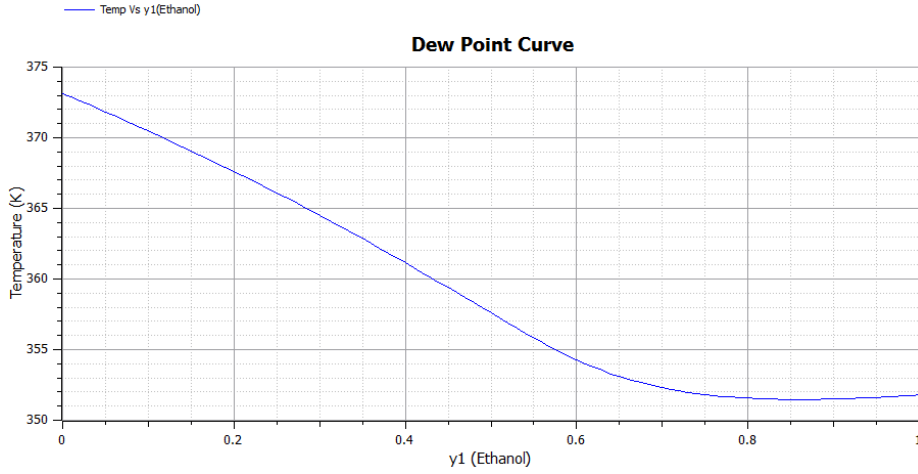
Figure 9.2: Dew point (Ty) curve for ethanol water system generated in openmodelica.

Manipulating the equations 9.1 and 9.2 and putting $x_1 + x_2 = 1$ we get

$$\frac{y_1}{\gamma_1.Pvap_1} + \frac{y_2}{\gamma_2.Pvap_2} = 1 \tag{9.4}$$

The pressure is kept constant at 1 atm. The value of y_1 (mole fraction of ethanol) is varied from 0 to 1 with an interval of 0.1 and for each value of y_1 the corresponding value of temperature if calculated by equation 9.4. Figure 9.2 shows the resulting ty curve as calculated by the developed built in thermodynamics in openmodelica.

- Combining the bubble point and the dew point curves as generated above gives the Txy curve for the system as shown in Figure 9.3
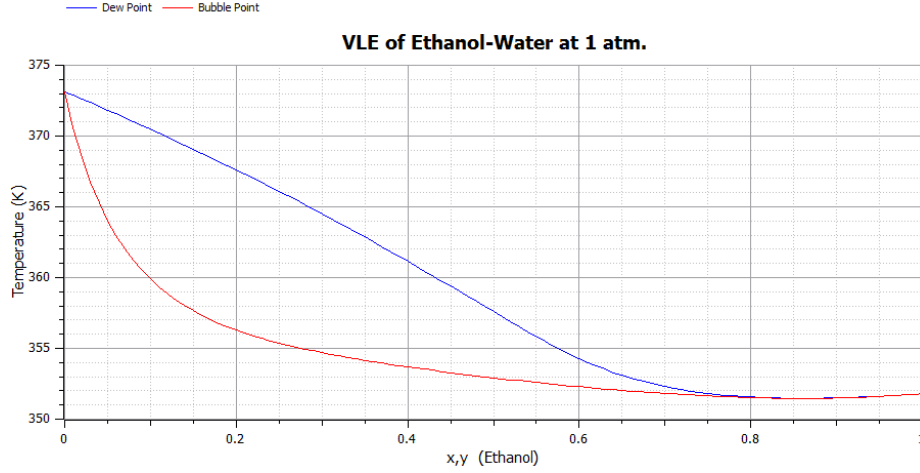
Figure 9.3: Txy curve for ethanol water system generated in openmodelica
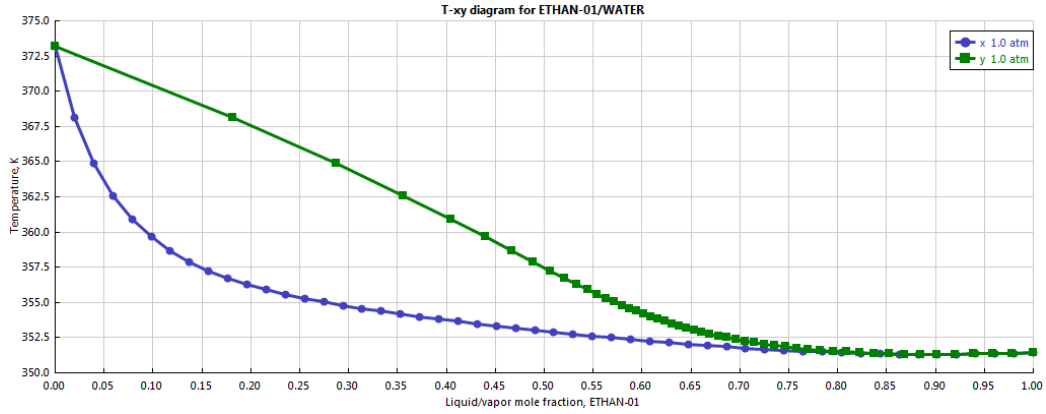
Figure 9.4: Txy curve for ethanol-water system (UNIQUAC) generated in Aspen Plus

This figure is verified by reproducing the same problem in Aspen plus. Figure 9.4 depicts the Txy curve for ethanol water system generated in Aspen plus. It clearly shows that the plot generated in openmodelica completely replicates the one generated in Aspen plus.

Reliable azeotropic data source by American Chemical Society [15] says that for ethanol-water system, at 1 atm, the azeotropic composition and temperatures are 0.96 mole fraction ethanol and 351.4 K. These values are also in agreement with the openmodelica results.

The same simulation when carried out with the imported DWSIM's thermodynamic engine in openmodelica resulted in an execution time of about 20 minutes whereas for the built in thermodynamic engine the execution time was fraction of a second.
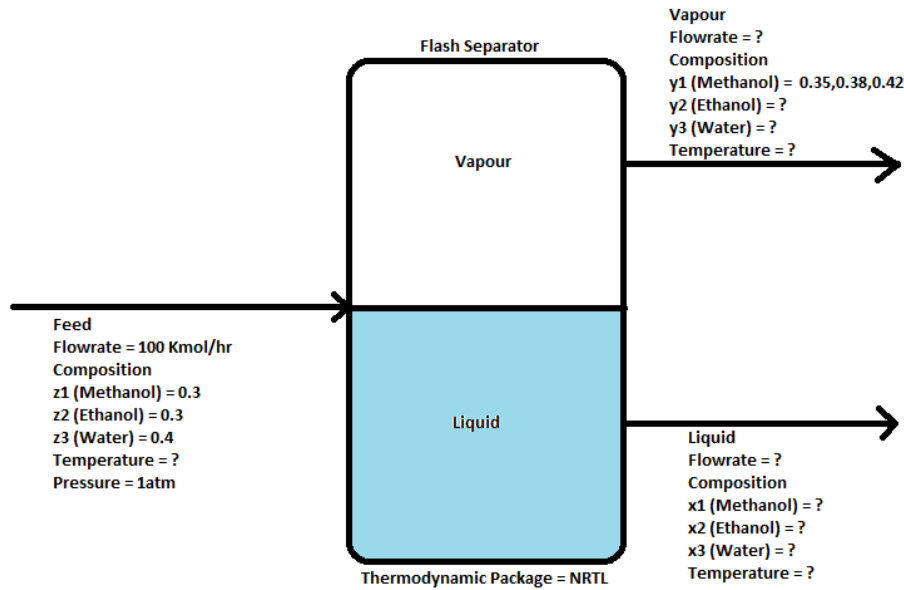
Figure 9.5: Model with problem statement for steady state flash of Methanol-ethanol-water.

9.2 Steady State Flash

9.2.1 The Model

In this model a steady state flash is simulated in openmodelica using the developed thermodynamics. The components involved are methanol, ethanol and water. The thermodynamic package used is NRTL. To check the design efficiency of the developed thermodynamic engine in openmodelica , the output composition of the vapor product is specified, while the temperature at which this desired composition of vapor is attained needs to be calculated. Therefore, it is a design problem. To carry out this simulation in DWSIM, we have to use an adjust operation which uses a trial and error method and is time consuming.

9.2.2 Problem Statement

The flowrate and the composition of the feed is specified. Pressure is kept constant at 1 atm. The problem is divided into three parts with each having different desired vapor compositions. The temperature and all the unknowns compositions and flow rates need to be calculated. The schematic of the problem statement is shown in figure 9.5.

9.2.3 Simulation Details and Equations

The input stream enters at 1 atm and the temperature is to be determined according to the desired vapor composition. The simulation was run for three different vapor

compositions. The minimum and maximum temperatures were taken to be boiling points of pure benzene and toluene and the initial guess for temperature is taken to be average of the two boiling points (Dew point and bubble point). The following equations describe the model.

Material Balances

$$F = V + L$$
$$x_{i,F}F = y_iV + x_iL$$

For i = 1,2,3

Equilibrium relations

$$y_i = K_ix_i$$

For i = 1,2,3

Summation Equation

$$\sum_{i=1}^{3} x_i = 1$$

Where i is the component index. y is the mole fraction in vapor phase and x in the liquid phase. 1,2,3 represents methanol, ethanol and water respectively.

9.2.4 Results

For the three case studies the desired vapor composition is different for each case. According to the VLE behavior the case with richest vapor composition (of more volatile component) should have the lowest temperature and the case with least vapor composition (of more volatile composition) should have the highest temperature. Table 9.1 show the resulting temperature and other compositions for the three different cases. The simulation time taken was almost instant, whereas if DWSIM was used it takes fifteen to twenty seconds to simulate the same.

Comparison of tables 9.1 and 9.2 shows that the results of openmodelica are very close to that of DWSIM.

Table 9.1: Results of simulation in Openmodelica using the build in thermodynamics

Desired Vapor Comp.(Methanol)	Temperature	Liquid Comp.
0.35	351.21	0.194
0.38	350.6	0.23
0.425	349.24	0.271

Table 9.2: Results of simulation in DWSIM

Desired Vapor Comp.(Methanol)	Temperature	Liquid Comp.
0.35	351.26	0.199
0.38	350.211	0.234
0.425	349.12	0.279

9.3 Semi-Batch Steam Distillation of a Binary Organic Mixture

9.3.1 Problem background

This illustrative example involves semi-batch steam distillation of binary mixture. A schematic plot of the steam distillation apparatus is shown in Figure 9.6. The organic mixture is charged into the still initially, and then steam is bubbled through continuously until the desired degree of separation has been reached. There are two different periods in the operation of the still: the heating period, until the boiling point temperature of the organic mixture is reached, and the distillation period. A brief description of the mathematical models for the two periods follows [21].

Heating period

A simple mass balance on the water phase yields

$$\frac{dm_w}{dt} = W_s$$

where W_s is the steam ow-rate in kmol/s and m is the mass SW of water in the still in kmol. It is assumed that all the steam condenses in the distillation vessel and that the organic phase masses remain constant during the heating period.

An energy balance on the still provides the equation for the change of the temperature T in °C

$$\frac{dT}{dt} = \frac{W_s(H_s - H_{lw}) - Q}{m_w c_{pLw} + m(x_1 c_{pL1} + x_2 c_{pL2})}$$

where H_s is the enthalpy of the steam in J/kmol, HLw is the enthalpy of liquid water in J/kmol, Q is the rate of heat transfer to the surroundings in J/s, c_{pLw} is the molar speci c heat of the water in J/kmolK, m is the mass of the organic phase in the still in kmol, x_1 and x_2 are the mole fractions, and c_{pL1} and c_{pL2} are the molar speci c heats of organic compounds No. 1 and 2, respectively, in J/kmolK. The heat transfer
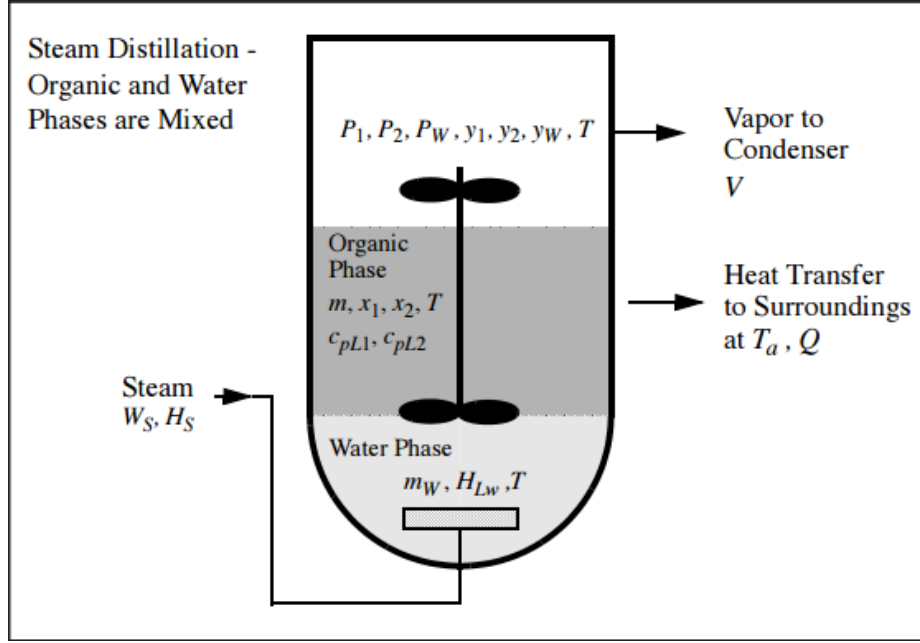
Figure 9.6: Schematic of steam distillation apparatus [21].

rate to the surroundings is calculated from the following equation.

$$Q = UA(T - T_a)$$

where UA is the product of the overall heat transfer coefficient U and the contact area A with the surroundings in J/sK, T_a is the ambient temperature in K, and T is the temperature of the liquid in the still in K.

Assuming ideal liquid behavior, Raoult's law can be used to calculate the vapor mole fraction of the components in the organic phase

$$y_1 = \frac{x_1 P_1}{P}$$
$$y_2 = \frac{x_2 P_2}{P}$$

where P is the total pressure in Pa and P_1 and P_2 are the vapor pressures of the organic compounds in Pa. The mole fraction of the water which is immiscible in the organic phase is given by $y_W = P_W/P$. The heating period continues until the sum of vapor pressures of the organic compounds and the water is equal to the total pressure. Thus, the bubble point equation to be satis ed can be expressed as

$$f(T) = 1 - (y_1 + y_2 + y_w) = 0$$

Distillation Period

During the distillation period, there is output of water vapor from the still.

$$\frac{dm_w}{dt} = W_s - V y_w$$

50

where V is the outlet vapor ow rate. Material balances on the two organic compounds yield two additional differential equations

$$\frac{d(mx_1)}{dt} = -Vy_1$$

$$\frac{d(mx_2)}{dt} = -Vy_2$$

The organic mass in the still at any time is given by: $m = m_x1 + m_x2$. The temperature in the still changes in a manner so that the bubble point equation is satisfied. The energy balance at a particular temperature yields the momentary vapor flow rate

$$V = \frac{W_s(H_S - HLW) - Q}{Hv - [y_w h_{lw} + (y_1 h_{L1} + y_2 h_{L2})]}$$

where H_V is the molar enthalpy of the vapor phase; $h_{Lw}, h_{L1}, and h_{L2}$ are the liquid phase molar enthalpies of water, n-octane and n-decane, respectively. Material balances on the water and organic phases in the still can provide the amount and the mole fractions of the various components in the distillate.

9.3.2 Problem Statement

semi-batch steam distillation of an n-octane (comp. 1) and n-decane (comp. 2) mixture. Initially M = 0.015 kmol of organics with composition $x_1 = 0.725$ is charged into the still. The initial temperature in the still is $T_0 = 25$ °C. Starting at time t = 0, steam at a temperature Tsteam = 99.2 °C is bubbled continuously through the organic phase at the rate of $M_S = 3.85$e-5 kmol/s. All the steam is assumed to condense during the heating period. The ambient temperature is $T_E = 25$ °C and the heat transfer coefficient between the still and the surrounding is UA = 1.05 J/s-K. The ambient pressure is P = 9.839E+04 Pa. Assumptions: 1) Ideal behavior of all components in pure state or mixture; 2) complete immiscibility of the water and the organic phases; 3) ideal mixing in the boiler; and 4) equilibrium between the organic vapor and its liquid at all times. The standard state for enthalpy calculations pure liquids at 0 °C and 1 atm. can be used.

- Calculate and plot the still temperature (T), component mole fractions inside the still $(x_1, x_2, y_1, and y_2)$, and the component mole fractions in the distillate $(x_1 dist and x_2 dist)$ using the data and the initial values provided.

- Determine the lowest n-octane mole fraction in the feed that can yield a distillate concentration of 90% of n-octane. Compute the percent recovery of n-octane in the distillate as function of its concentration in the feed. Vary the feed concentration in the range where the requirement for the n-octane concentration in the distillate is attainable.
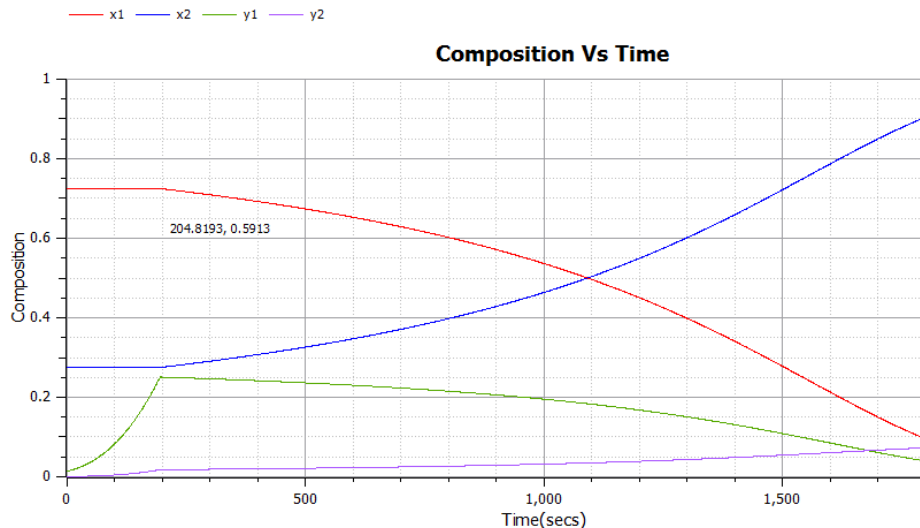
Figure 9.7: Change of organic phase composition during semi-batch steam distillation (as simulated in Openmodelica)

9.3.3 Results

Plots 9.7 and 9.8 shows the results obtained by simulating the example in openmodelica by using the built in thermodynamics.

 Plots 9.9 and 9.10 shows the results from the actual thesis.

It can be seen that the results of openmodelica are in agreement with the original thesis. Therefore, the built in thermodynamics can be claimed to be robust and accurate.

9.4 Batch Distillation

9.4.1 The Model

This case study is taken from the book Chemical Engineering Dynamic Modelling with PC simulation [16]. For this example, a column containing four theoretical plates and reboiler is assumed, together with constant volume conditions in refux drum. The liquid behavior is, however, non-ideal for this water-methanol system. Constant holdup is considered for each tray and the vapour and liquid flowrates are assumed to be constant throughout the column. The efficiency of each tray is considered to be one. The thermodynamic used is NRTL as methanol- water is a non-ideal system. The block diagram of the model is shown in figure 9.11.

9.4.2 Simulation Details and Equations

For convenience, only four stage were used in this model. Here the only thermodynamic calculations which were required were calculation of bubble point temperature
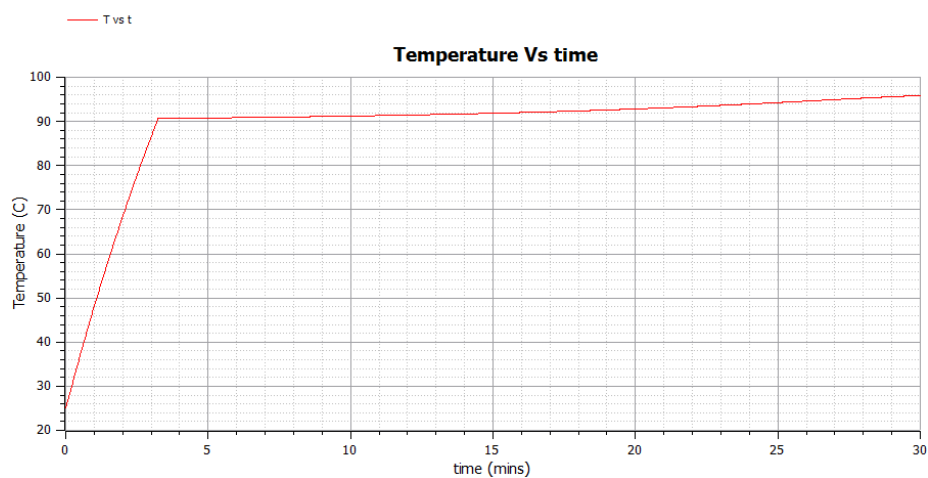
Figure 9.8: Temperature change during semi-batch steam distillation (as simulated in Openmodelica)
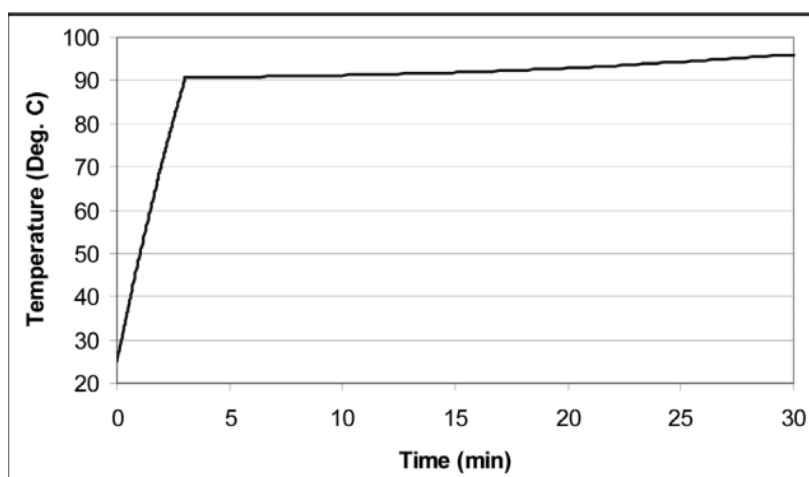
Figure 9.9: Change of organic phase composition during semi-batch steam distillation [21]
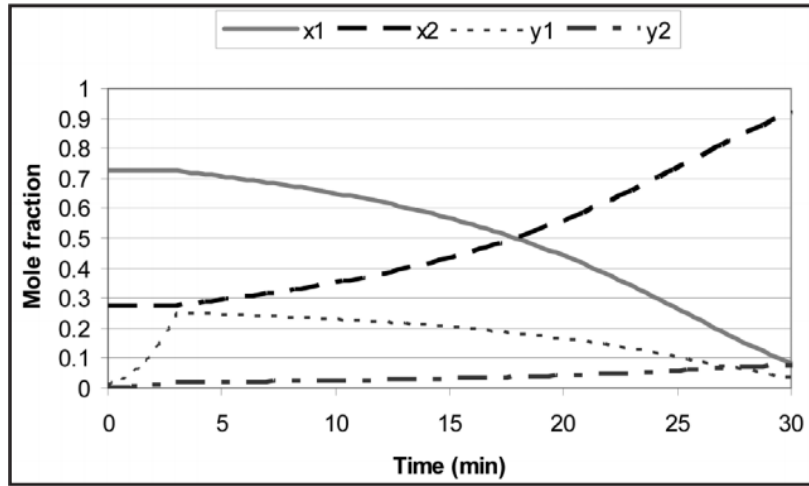
Figure 9.10: Temperature change during semi-batch steam distillation [21]
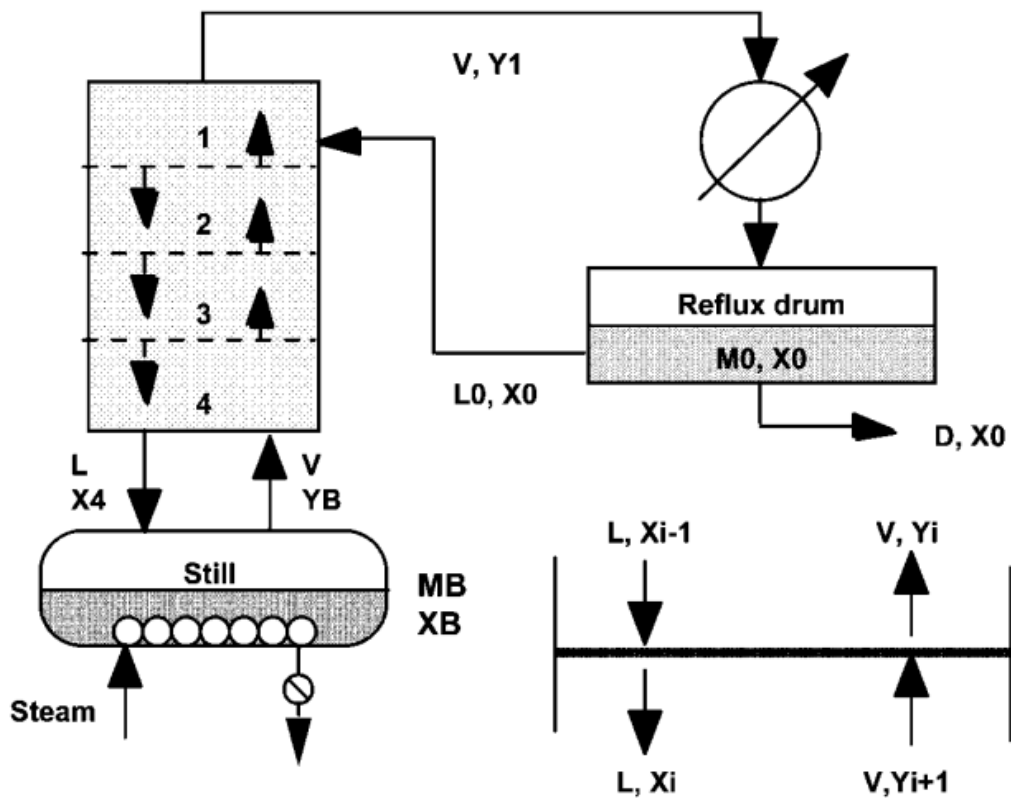
Figure 9.11: The four stage batch distillation column [16]

at each tray and the corresponding vapour compositions. The condenser is a total condenser. Molar holdup on each tray was taken to be 20. Initial molar holdup in the reboiler is 2500. Initial concentration in the reboiler was 0.8 mole fraction Methanol. Vapour boilup rate is 10. The pressure is constant at 1 bar. The column was started with total reflux and then it was reduced to 10 at t = 20. Following equation describe the model.

Component balances for the more volatile component on any plate n.

$$M\frac{dx_n}{dt} = L(x_{n-1} - x_n) + V(y_{n+1} - y_n)$$

Equilibrium equations for the trays.

$$y_{n,i} = K_i x_{n,i}$$

Where n stands for the tray and i for the compound.
Summation Equations

$$y_1 + y_2 = 1$$
$$x_1 + x_2 = 1$$

where 1 is the more volatile component and 2 is the less volatile component.
Component balance for the reboiler

$$\frac{d(M_B x_B)}{dt} = Lx_4 - Vy_B$$

Equilibrium and summation equations for the reboiler is same as that of the trays.
For reflux drum and condenser

$$\frac{dx_0}{dt} = Vy_1 - (L + D)x_0$$
$$\frac{L}{D} = R$$
$$V = L + D$$

9.4.3   Results

Figures 9.12 and 9.13[16] shows the composition of methanol increases very fast in starting as there is total reflux initially and then in increases with a low rate an now the reflux ratio is 10. The roboiler is slowly getting leaner in methanol as the holdup in roboiler is very high as compared to the trays.

Figures 9.14 and 9.15 shows that the temperature of all the trays drastically decreases and becomes less than the reboiler. The reboiler temperature changes slowly because the holdup is very high as compared to trays. One can also notice that as we move upwards along the column the temperature, this is due to the fact that the as we move upwards along the column the composition gets richer in more volatile component.
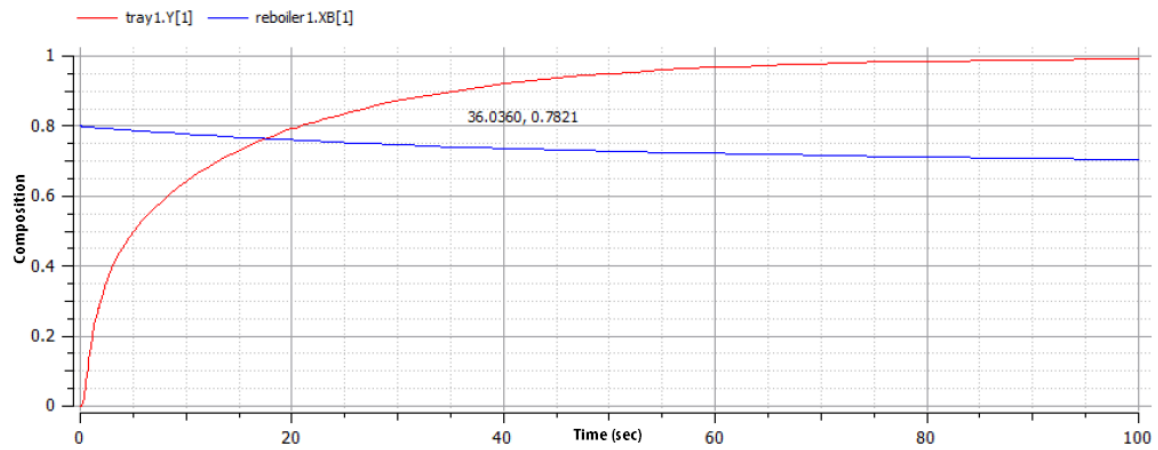
Figure 9.12: Methanol composition profiles of liquid holdup in reboiler and vapour leaving form tray 1 (as generated in openmodelica)
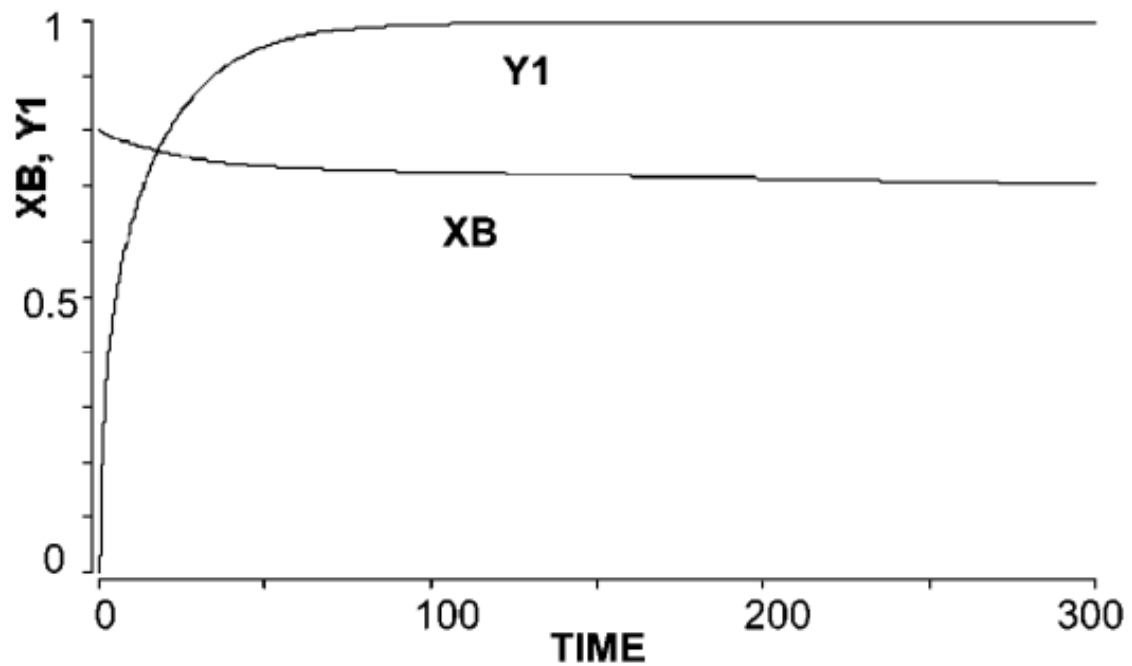
Figure 9.13: Methanol composition profiles of liquid holdup in reboiler and vapour leaving form tray 1 [16]
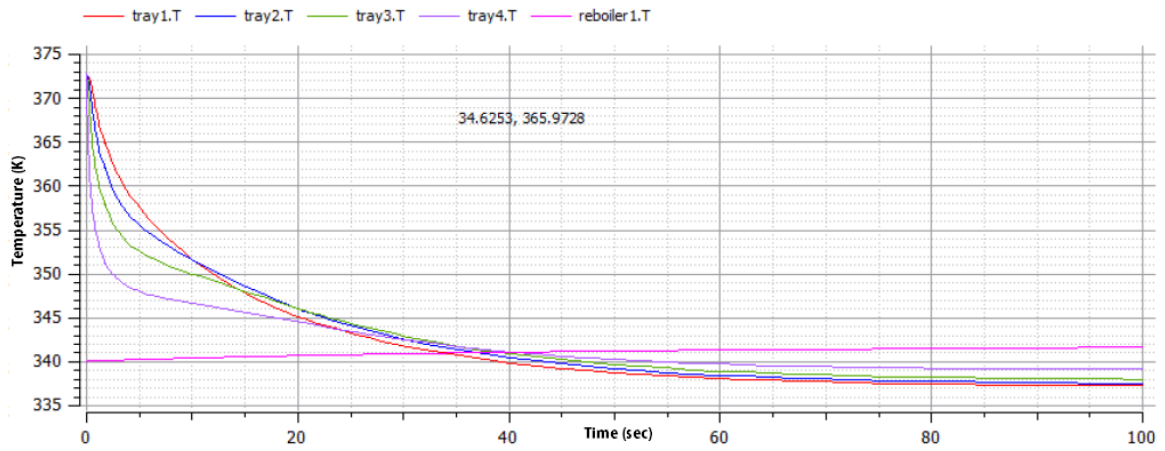
Figure 9.14: Temperature profiles of tray 1,2,3,4 and reboiler with time as generated in openmodelica
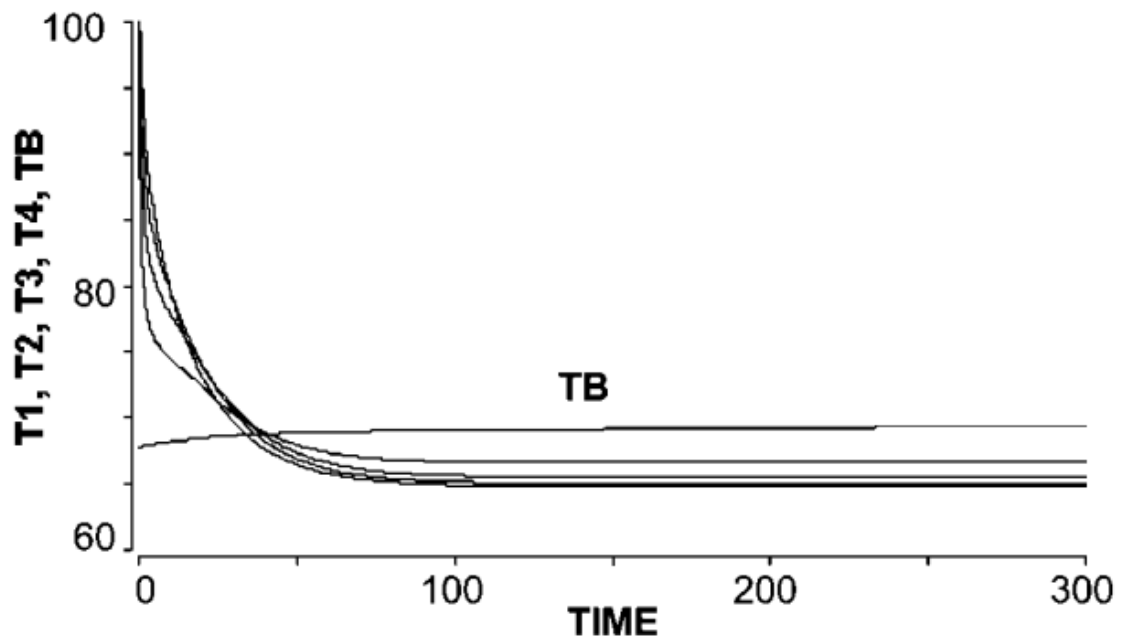
Figure 9.15: Temperature profiles of tray 1,2,3,4 and reboiler with time[16]

57

Chapter 10

Future Work

One of the next objective will be to built a library of steady state and dynamic unit operations in openmodelica. As now openmodelica has its own thermodynamic engine, a library of steady state unit operations should be modeled. Another prime objective is to build a library of dynamic unit operations in openmodelica to carry out calculations of batch processes.

After developing the unit operation library, a GUI should be developed so that an end user could conveniently use the software without getting into the complexities of the code. The GUI will be a replication of the commercial process simulators so the it might be easy for the users to adapt.

A collection of sample examples from various reliable sources should be simulated so as to verify the developed simulator, thus making it flawless. These sources might include examples from standard chemical engineering books and sample problems from commercial simulators like aspen plus.

Another objective is to identify dynamic simulation problems of interest to chemical industry and coding them in OpenModelica. An industrial campaign is planned, to visit small scale industries, who can't afford consultancy services from big consulting firms using commercial simulators, to identify their problems and provide an efficient solution as now we have an open source simulator that can handle dynamic simulations as well.

Appendix A

Codes for Python-C API Approach

A.1 The Python Code

```
def Properties(Comp, Prop, PropType, TorP):
import win32com. client
dtlc = win32com.client.Dispatch(DTL.Thermodynamics.Calculator) dtlc . Ini
if PropType==constProp:
elif
elif
PROPVALUE = dtlc.GetCompoundConstProp(Comp, Prop)  PropType==constPDepProp:
TorP=float (TorP)
PROPVALUE = dtlc.GetCompoundPDepProp(Comp, Prop, TorP)
PropType==constTDepProp:  TorP=float(TorP)
PROPVALUE = dtlc.GetCompoundTDepProp(Comp, Prop, TorP)
pv = float(PROPVALUE)
return pv
def TwoPhaseProperties(PropPack, FlashAlg, P, T, Comp1, Comp2, mf1, mf2, k, l2):
import win32com. client
dtlc = win32com.client.Dispatch(DTL.Thermodynamics.Calculator
dtlc.Initialize()
Compounds=[      1 2          ,        1 2         ]
MoleFractions=[       1 2         ,        1 2        ]
pv=[[1.0 ,1.0 ,1.0 ,1.0]  ,[1.0 ,1.0 ,1.0 ,1.0]  ,[1.0 ,1.0 ,1.0 ,1.0]]
x=[[1.0 ,1.0 ,1.0 ,1.0]  ,[1.0 ,1.0 ,1.0 ,1.0]  ,[1.0 ,1.0 ,1.0,1.0]]
Compounds[0] = Comp1
Compounds [1] = Comp2
MoleFractions [0]=mf1
MoleFractions [1]=mf2
FlashAlg=float (FlashAlg)
P=float(P)
T=float(T)
l=len ( MoleFractions ) for i in range(l):
```

```python
MoleFractions[i]= float(MoleFractions[i])
val=dtlc.PTFlash(PropPack, FlashAlg, P, T,Compounds, MoleFractio pv[0] =
pv[1]= val [2]
pv[2]= val [3]
for i in range(3):
for j in range(4):
x[i][j]=float(pv[i][j])
return x[k][l2]
def BinaryInteractionParameters(PropPack,Comp1,Comp2,l2):
import win32com.client
dtlc = win32com.client.Dispatch(DTL.Thermodynamics.Calculator )
dtlc.Initialize()
x=[1.0 ,1.0 ,1.0]
l2=int(l2)
val=dtlc.GetInteractionParameterSet(PropPack ,Comp1,Comp2)
for i in range(3):
x[i]= float(val[i])
return x[l2]
```

A.2   The C Code

```c
#include "C:\Python27\include\Python.h"

double Property(char *Comp, char *Prop, char *PropType, double TorP)
{
        PyObject *pName, *pModule, *pDict, *pFunc, *pValue, *pTorP,
        *ptype, *pArgs, *pcomp, *pprop;
        double mw = 0.0;

        // Initialize the Python Interpreter
        Py_Initialize();

        // Build the name object
        pName = PyString_FromString("py_function");

        // Load the module object
        pModule = PyImport_Import(pName);

        // pDict is a borrowed reference
        pDict = PyModule_GetDict(pModule);
        pFunc = PyDict_GetItemString(pDict, "Properties");

        // pFunc is also a borrowed reference
        if (PyCallable_Check(pFunc))
        {
```

```
        // Prepare the argument list for the call

        pArgs = PyTuple_New(4);
        ptype = PyString_FromString(PropType);
        pcomp = PyString_FromString(Comp);
        pprop = PyString_FromString(Prop);
        pTorP = PyFloat_FromDouble(TorP);
        PyTuple_SetItem(pArgs, 0, pcomp);
        PyTuple_SetItem(pArgs, 1, pprop);
        PyTuple_SetItem(pArgs, 2, ptype);
        PyTuple_SetItem(pArgs, 3, pTorP);

        pValue = PyObject_CallObject(pFunc, pArgs);
        mw = PyFloat_AsDouble(pValue);

        if (pArgs != NULL)
        {
                Py_DECREF(pArgs);
        }

        else
        {
                pValue = PyObject_CallObject(pFunc, NULL);
        }

        if (pValue != NULL)
        {
                Py_DECREF(pValue);
        }
        else
        {
                PyErr_Print();
        }

        // some code omitted...
    }
    // Clean up
    Py_DECREF(pModule);
    Py_DECREF(pName);
    return mw;
    // Finish the Python Interpreter
    Py_Finalize();


}
```

```
double TwoPhaseProperty(char *propPack, double flashAlg,
double pressure, double temp,
char *comp1, char *comp2, double mf1, double mf2, double i, double j)
{
 PyObject *pName, *pModule, *pDict, *pFunc, *pValue, *palg,
 *pArgs,*pcomp1, *pcomp2, *pmf1, *pmf2, *ptemp, *ppres, *ppack, *pi, *pj;
 double mw = 0.0;
         // Initialize the Python Interpreter
         Py_Initialize();

         // Build the name object
         pName = PyString_FromString("py_function");

         // Load the module object
         pModule = PyImport_Import(pName);

         // pDict is a borrowed reference

         pDict = PyModule_GetDict(pModule);
         pFunc = PyDict_GetItemString(pDict, "TwoPhaseProperties");

         // pFunc is also a borrowed reference
         if (PyCallable_Check(pFunc))
         {
                 // Prepare the argument list for the call

                 pArgs = PyTuple_New(10);
                 ppack = PyString_FromString(propPack);
                 palg = PyFloat_FromDouble(flashAlg);
                 ppres = PyFloat_FromDouble(pressure);
                 ptemp = PyFloat_FromDouble(temp);
                 pcomp1 = PyString_FromString(comp1);
                 pcomp2 = PyString_FromString(comp2);
                 pmf1 = PyFloat_FromDouble(mf1);
                 pmf2 = PyFloat_FromDouble(mf2);

                 pi = PyInt_FromLong(i);
                 pj = PyInt_FromLong(j);

                 PyTuple_SetItem(pArgs, 0, ppack);
                 PyTuple_SetItem(pArgs, 1, palg);
                 PyTuple_SetItem(pArgs, 2, ppres);
                 PyTuple_SetItem(pArgs, 3, ptemp);
                 PyTuple_SetItem(pArgs, 4, pcomp1);
```

62

```
                PyTuple_SetItem(pArgs, 5, pcomp2);
                PyTuple_SetItem(pArgs, 6, pmf1);
                PyTuple_SetItem(pArgs, 7, pmf2);
                PyTuple_SetItem(pArgs, 8, pi);
                PyTuple_SetItem(pArgs, 9, pj);


                pValue = PyObject_CallObject(pFunc, pArgs);
                mw = PyFloat_AsDouble(pValue);

                if (pArgs != NULL)
                {
                        Py_DECREF(pArgs);
                }

                else
                {
                        pValue = PyObject_CallObject(pFunc, NULL);
                }

                if (pValue != NULL)
                {

                        Py_DECREF(pValue);
                }
                else
                {
                        PyErr_Print();
                }

                // some code omitted...
        }
        // Clean up
        Py_DECREF(pModule);
        Py_DECREF(pName);

        return mw;
        // Finish the Python Interpreter
        Py_Finalize();

}


double binaryInteractionParameters(char *propPack, char *comp1,
 char *comp2, double i)
```

```
{
        PyObject *pName, *pModule, *pDict, *pFunc, *pValue,
         *pArgs, *pcomp1, *pcomp2, *ppack, *pi;
        double mw = 0.0;
        // Initialize the Python Interpreter
        Py_Initialize();

        // Build the name object
        pName = PyString_FromString("py_function");

        // Load the module object
        pModule = PyImport_Import(pName);

        // pDict is a borrowed reference

        pDict = PyModule_GetDict(pModule);
        pFunc = PyDict_GetItemString(pDict,
         "BinaryInteractionParameters");

        // pFunc is also a borrowed reference
        if (PyCallable_Check(pFunc))
        {
                // Prepare the argument list for the call

                pArgs = PyTuple_New(4);
                ppack = PyString_FromString(propPack);
                pcomp1 = PyString_FromString(comp1);
                pcomp2 = PyString_FromString(comp2);
                pi = PyInt_FromLong(i);

                PyTuple_SetItem(pArgs, 0, ppack);
                PyTuple_SetItem(pArgs, 1, pcomp1);
                PyTuple_SetItem(pArgs, 2, pcomp2);
                PyTuple_SetItem(pArgs, 3, pi);


                pValue = PyObject_CallObject(pFunc, pArgs);
                mw = PyFloat_AsDouble(pValue);

                if (pArgs != NULL)
                {
                        Py_DECREF(pArgs);
                }

                else
```

```c
                {
                        pValue = PyObject_CallObject(pFunc, NULL);
                }

                if (pValue != NULL)
                {
                        printf("Return_of_call_:_%d\n",
                        PyInt_AsLong(pValue));
                        Py_DECREF(pValue);
                }
                else
                {
                        PyErr_Print();
                }

                // some code omitted...
        }
        // Clean up
        Py_DECREF(pModule);
        Py_DECREF(pName);

return mw;
        // Finish the Python Interpreter
        Py_Finalize();


}
double antoine2(char *comp, double i)
{
        PyObject *pName, *pModule, *pDict, *pFunc, *pValue,
        *pArgs, *pcomp, *pi;
        double mw = 0.0;
        // Initialize the Python Interpreter
        Py_Initialize();

        // Build the name object
        pName = PyString_FromString("py_function");

        // Load the module object
        pModule = PyImport_Import(pName);

        // pDict is a borrowed reference

        pDict = PyModule_GetDict(pModule);
        pFunc = PyDict_GetItemString(pDict, "antoine");
```

```c
// pFunc is also a borrowed reference
if ( PyCallable_Check(pFunc))
{
        // Prepare the argument list for the call

        pArgs = PyTuple_New(2);
        pcomp = PyString_FromString(comp);
        pi = PyInt_FromLong(i);

        PyTuple_SetItem(pArgs, 0, pcomp);
        PyTuple_SetItem(pArgs, 1, pi);


        pValue = PyObject_CallObject(pFunc, pArgs);
        mw = PyFloat_AsDouble(pValue);

        if (pArgs != NULL)
        {
                Py_DECREF(pArgs);
        }

        else
        {
                pValue = PyObject_CallObject(pFunc, NULL);
        }

        if (pValue != NULL)
        {

                Py_DECREF(pValue);
        }
        else
        {
                PyErr_Print();
        }

        // some code omitted...
}
// Clean up
Py_DECREF(pModule);
Py_DECREF(pName);

return mw;
// Finish the Python Interpreter
```

```c
        Py_Finalize ();

}

double calcP(char *comp, char *property, char *propType, double T0){
        double guess[6], Tguess[6], P;
        int i;
        guess[1] = 132300;
        guess[0] = 122310;
        /*clock_t time1 =clock();
        clock_t end;
        printf("%f", time1);*/
        for (i = 0; i < 2; i++)
        {
                Tguess[i] = Property(comp, property, propType, guess[i]);
                /* end = clock();
                printf("time ra time: %f\n", ((double)
                (end −   time1))
        }

        for (i = 2; i < 6; i++){
                if (fabs(Tguess[i − 1] − 339) > 0.01){
                        guess[i] = guess[i − 1] − (Tguess[i − 1] − T0)
                        * ((guess[i − 1] − guess[i − 2]) /
                        (Tguess[i − 1] − Tguess[i − 2]));
                        Tguess[i] = Property(comp, property,
                         propType, guess[i]);
                        /*end = clock();
                        printf("final time: %f\n", ((double)
                        (end −  time1))
                }
                else {
                        P = guess[i − 1];
                        printf("Pfinal else wala: %f\n", P);
                        double T = Property(comp, property,
                        propType, guess[i]);
                        /*end = clock();
                        printf("final time: %f\n",
                        ((double)(end −  time1))/CLOCKS_PER_SEC);
                        printf("\n\n%f", T);*/

                        return P;

                }
        }
```

```
        P = guess[5];
        /*printf("Pfinal: %f", P);
        double T = Property(comp, "boilingPointTemperature"
        , "constPDepProp", P);
        printf("\n\n%f", T);
        printf("%f", time(NULL) - time1); */
        return P;
}

void checkFunction(double p[])
{
p[1]=52;
}
```

Appendix B

Codes for Client-Server (Socket) Approach

B.1   The Python Approach

```python
import socket
import win32com.client
def Main():
    HOST = ''
    PORT = 7000
    dtl = win32com.client.Dispatch("DTL.Thermodynamics.Calculator")
    dtl.Initialize()
    serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    serversocket.bind((HOST,PORT))
    serversocket.listen(2)
    print('Server Listening.....')
    while True:
        connsocket, addr = serversocket.accept()
        print('Connection from',addr)
        if True:
          data = connsocket.recv(4096)
          if not data: break
          strdata = data.decode()
          splitdata = strdata.split(',')
          Nc = int(splitdata[3])
          No = 4+Nc
          P = float(splitdata[1])
          VF = float(splitdata[2])
          Comp = splitdata[4:No]
          Xstr = splitdata[No:len(splitdata)]
          X = [float(i) for i in Xstr]
          PVFlash = dtl.PVFFlash(splitdata[0],0,P,VF,Comp,X)
```

```python
        ptfl = " " + str(PVFlash[2][0]) + " "
        if Nc>2:
         for j in range(3,Nc+1):
            ptfl = ptfl + str(PVFlash[j][0]) + " "
        ptfl = ptfl + PVFlash[Nc+2][0]
        connsocket.send(ptfl)
      else:
        connsocket.close()


    serversocket.close()

if __name__ == '__main__':
    Main()
```

B.2 The C Client

```c
#ifdef WIN32
    #include <winsock.h>
    #include <winsock2.h>
#else
    #include <system/socket.h>
    #include <netinet/in.h>
#endif
#include<stdio.h>
#include<errno.h>
#include<stdlib.h>
#include<string.h>
#include<unistd.h>
#include<math.h>

#define BUFFERSIZE 4096

void error(const char*);

char* PVFFlash(char *Prop, double P, double VF,
int Nc, const char *Comp[], const double x[])


{
    char *Result;
// double Resultss[3];
//   char *token;
   int i = 0;
   int j = 0;
```

70

```
int k = 0;
WSADATA wsa;

//    printf("\nInitialising Winsock...");
   if (WSAStartup(MAKEWORD(2,2),&wsa) != 0)
   {
       // printf("Failed. Error Code : %d",WSAGetLastError());
       // return 1;
   }

  // printf("Initialised.");

struct sockaddr_in sock_addr;
int sock_d = 0;
char recvbuffer[BUFFERSIZE], sendbuffer[BUFFERSIZE],
recvbuffer1[BUFFERSIZE];
strcpy(sendbuffer,Prop);
strcat(sendbuffer,",");
sprintf(sendbuffer+ strlen(sendbuffer),"%lf",P);
strcat(sendbuffer,",");
sprintf(sendbuffer+ strlen(sendbuffer),"%lf",VF);
strcat(sendbuffer,",");
sprintf(sendbuffer+ strlen(sendbuffer),"%d",Nc);
strcat(sendbuffer,",");
for (i=0;i<Nc;i++)
{
   strcat(sendbuffer,Comp[i]);
   strcat(sendbuffer,",");
}
  for (j=0;j<Nc;j++)
{
   sprintf(sendbuffer+ strlen(sendbuffer),"%lf",x[j]);
   if(j<(Nc-1))
  { strcat(sendbuffer,",");}
}

if ((sock_d = socket(AF_INET,SOCK_STREAM,0)) < 0)
   printf("Problem in creating socket");

memset(&sock_addr, 0, sizeof(sock_addr));
sock_addr.sin_family = AF_INET;
sock_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
sock_addr.sin_port = htons(7000);

if(connect(sock_d, (struct sockaddr *) &sock_addr,
```

```
      sizeof(sock_addr)) < 0)
      perror("Problem_in_connecting_to_server");


            sendto(sock_d, sendbuffer, strlen(sendbuffer),
            0, (struct sockaddr*) &sock_addr, sizeof(sock_addr));
            memset(recvbuffer, 0, BUFFERSIZE);
            int size = sizeof(sock_addr);
            if (recvfrom(sock_d, recvbuffer, BUFFERSIZE,
            0, (struct sockaddr*) &sock_addr, &size) == 0)
            perror("The_server_therminated_prematurely");


            Result = recvbuffer;
  //        token = strtok(result1, " ");
  //        while (token != NULL)
  //         {
  //         ResultStr[k] = token;
  //            token = strtok(NULL, " ");
  //            Resultss[k] = round(atof(ResultStr[k])*100000)/100000;
  //            k++;

  //          }
      close(sock_d);
return Result;
}
```

Appendix C

Python script to convert Chemsep database into openmodelica syntax

C.1 Python script to convert Chemsep.xml to openmodelica classes

```python
from xml.dom.minidom import parse
import xml.dom.minidom
DOMTree = xml.dom.minidom.parse("chemsep1.xml")
compounds = DOMTree.documentElement
compound = compounds.getElementsByTagName("compound")
i = 1
for comp in compound:
compName = comp.getElementsByTagName
("CompoundID")[0].getAttribute("value")
CompName = compName.replace(" ","")
CompName = CompName.replace("-","")
CompName = CompName.replace(",","")
CompName = CompName.replace("1","One")
CompName = CompName.replace("2","Two")
CompName = CompName.replace("3","Three")
CompName = CompName.replace("4","Four")
CompName = CompName.replace("5","Five")
CriticalTemp = comp.getElementsByTagName
("CriticalTemperature")[0].getAttribute("value")
CriticalPres = comp.getElementsByTagName
("CriticalPressure")[0].getAttribute("value")
CriticalVol = comp.getElementsByTagName
("CriticalVolume")[0].getAttribute("value")
CriticalComp = comp.getElementsByTagName
("CriticalCompressibility")[0].getAttribute("value")
```

```python
try:
    NormalBoilPoint = comp.getElementsByTagName
    ("NormalBoilingPointTemperature")[0].
    getAttribute("value")
except IndexError:
    NormalBoilPoint = "0"
try:
    NormalMeltingPoint = comp.getElementsByTagName
    ("NormalMeltingPointTemperature")[0].
    getAttribute("value")
except IndexError:
    NormalMeltingPoint = "0"
try:
    TripPntTemp = comp.getElementsByTagName
    ("TriplePointTemperature")[0].
    getAttribute("value")
except IndexError:
    TripPntTemp = "0"
try:
    TripPntPres = comp.getElementsByTagName
    ("TriplePointPressure")[0].
    getAttribute("value")
except IndexError:
    TripPntPres = "0"
MolWt =   comp.getElementsByTagName
("MolecularWeight")[0].
getAttribute("value")
try:
    LiqVolAtBoilPnt = comp.getElementsByTagName
    ("LiquidVolumeAtNormalBoilingPoint")[0].
    getAttribute("value")
except IndexError:
    LiqVolAtBoilPnt = "0"
try:
    AcenFactor = comp.getElementsByTagName
    ("AcentricityFactor")[0].
    getAttribute("value")
except IndexError:
    AcenFactor = "0"
try:
    SolParam = comp.getElementsByTagName
    ("SolubilityParameter")[0].
    getAttribute("value")
except IndexError:
    SolParam = "0"
```

```python
try:
    DipoleMoment = comp.getElementsByTagName
    ("DipoleMoment")[0].
    getAttribute("value")
except IndexError:
    DipoleMoment = "0"
try:
    IGHF = comp.getElementsByTagName
    ("HeatOfFormation")[0].
    getAttribute("value")
except IndexError:
    IGHF = "0"
try:
    GEF = comp.getElementsByTagName
    ("GibbsEnergyOfFormation")[0].
    getAttribute("value")
except IndexError:
    GEF = "0"
try:
    AbsEntropy = comp.getElementsByTagName
    ("AbsEntropy")[0].
    getAttribute("value")
except IndexError:
    AbsEntropy = "0"
try:
    HeatFusionMeltPnt = comp.getElementsByTagName
    ("HeatOfFusionAtMeltingPoint")[0].
    getAttribute("value")
except IndexError:
    HeatFusionMeltPnt = "0"
try:
    HOC = comp.getElementsByTagName
    ("HeatOfCombustion")[0].
    getAttribute("value")
except IndexError:
    HOC = "0"
try:
    UniquacR = comp.getElementsByTagName
    ("UniquacR")[0].
    getAttribute("value")
except IndexError:
    UniquacR = "0"
try:
    UniquacQ = comp.getElementsByTagName
    ("UniquacQ")[0].
```

```python
    getAttribute("value")
except IndexError:
  UniquacQ = "0"
try:
  RacketParam = comp.getElementsByTagName
  ("RacketParameter")[0].
  getAttribute("value")
except IndexError:
  RacketParam = "0"


try:
  LiqDen = comp.getElementsByTagName("LiquidDensity")[0]
  LiqDenEqn = LiqDen.getElementsByTagName("eqno")[0].getAttribute("value")
  A=LiqDen.getElementsByTagName("A")[0].getAttribute("value")
  B=LiqDen.getElementsByTagName("B")[0].getAttribute("value")
  C=LiqDen.getElementsByTagName("C")[0].getAttribute("value")
  D=LiqDen.getElementsByTagName("D")[0].getAttribute("value")
  try:
      E=LiqDen.getElementsByTagName("E")[0].getAttribute("value")
  except IndexError:
      E = "0"
except IndexError:
  LiqDenEqn = "0"
  A = "0"
  B = "0"
  C = "0"
  D = "0"
  E = "0"
try:
  VapPres = comp.getElementsByTagName("VaporPressure")[0]
  VapPresEqn = VapPres.getElementsByTagName("eqno")[0].
  getAttribute("value")
  VA=VapPres.getElementsByTagName("A")[0].getAttribute("value")
  VB=VapPres.getElementsByTagName("B")[0].getAttribute("value")
  VC=VapPres.getElementsByTagName("C")[0].getAttribute("value")
  try:
      VD=VapPres.getElementsByTagName("D")[0].getAttribute("value")
  except IndexError:
      VD = "0"
  try:
      VE=VapPres.getElementsByTagName("E")[0].getAttribute("value")
  except IndexError:
      VE = "0"
except IndexError:
```

76

```python
  VapPresEqn = "0"
 VA = "0"
 VB = "0"
 VC = "0"
 VD = "0"
 VE = "0"
try:
  LiqCp = comp.getElementsByTagName("LiquidHeatCapacityCp")[0]
  LiqCpEqn = LiqCp.getElementsByTagName("eqno")[0].getAttribute("value")
  LCpA=LiqCp.getElementsByTagName("A")[0].getAttribute("value")
  LCpB=LiqCp.getElementsByTagName("B")[0].getAttribute("value")
  LCpC=LiqCp.getElementsByTagName("C")[0].getAttribute("value")
  try:
      LCpD=LiqCp.getElementsByTagName("D")[0].getAttribute("value")
  except IndexError:
      LCpD = "0"
  try:
      LCpE=LiqCp.getElementsByTagName("E")[0].getAttribute("value")
  except IndexError:
      LCpE = "0"
except IndexError:
 LiqCpEqn = "0"
 LCpA = "0"
 LCpB = "0"
 LCpC = "0"
 LCpD = "0"
 LCpE = "0"
try:
  HOV = comp.getElementsByTagName("HeatOfVaporization")[0]
  HOVEqn = HOV.getElementsByTagName("eqno")[0].getAttribute("value")
  HOVA=HOV.getElementsByTagName("A")[0].getAttribute("value")
  HOVB=HOV.getElementsByTagName("B")[0].getAttribute("value")
  HOVC=HOV.getElementsByTagName("C")[0].getAttribute("value")
  try:
      HOVD=HOV.getElementsByTagName("D")[0].getAttribute("value")
  except IndexError:
      HOVD = "0"
  try:
      HOVE=HOV.getElementsByTagName("E")[0].getAttribute("value")
  except IndexError:
      HOVE = "0"
except IndexError:
 HOVEqn = "0"
 HOVA = "0"
 HOVB = "0"
```

```python
  HOVC = "0"
  HOVD = "0"
  HOVE = "0"
if (float(NormalBoilPoint) > 298.15):
  HA = float(HOVA)
  HB = float(HOVB)
  HC = float(HOVC)
  HD = float(HOVD)
  HE = float(HOVE)
  Tr = 298.15/float(CriticalTemp)
  SHOV = HA*(pow((1-Tr),(HB + HC*Tr + HD*pow(Tr,2) + HE*pow(Tr,3))))
  AbsEnthalpy = float(IGHF) - SHOV
else:
  AbsEnthalpy = float(IGHF)
SH = str(AbsEnthalpy)
try:
  VapCp = comp.getElementsByTagName("IdealGasHeatCapacityCp")[0]
  VapCpEqn = VapCp.getElementsByTagName("eqno")[0].getAttribute("value")
  VCpA=VapCp.getElementsByTagName("A")[0].getAttribute("value")
  VCpB=VapCp.getElementsByTagName("B")[0].getAttribute("value")
  VCpC=VapCp.getElementsByTagName("C")[0].getAttribute("value")
  try:
      VCpD=VapCp.getElementsByTagName("D")[0].getAttribute("value")
  except IndexError:
      VCpD = "0"
  try:
      VCpE=VapCp.getElementsByTagName("E")[0].getAttribute("value")
  except IndexError:
      VCpE = "0"
except IndexError:
  VapCpEqn = "0"
  VCpA = "0"
  VCpB = "0"
  VCpC = "0"
  VCpD = "0"
  VCpE = "0"

try:
  LiqVis = comp.getElementsByTagName("LiquidViscosity")[0]
  LiqVisEqn = LiqVis.getElementsByTagName("eqno")[0].
  getAttribute("value")
  LiqVisA=LiqVis.getElementsByTagName("A")[0].getAttribute("value")
  LiqVisB=LiqVis.getElementsByTagName("B")[0].getAttribute("value")
  LiqVisC=LiqVis.getElementsByTagName("C")[0].getAttribute("value")
  try:
```

```python
        LiqVisD=LiqVis.getElementsByTagName("D")[0].getAttribute("value")
    except IndexError:
        LiqVisD = "0"
    try:
        LiqVisE=LiqVis.getElementsByTagName("E")[0].getAttribute("value")
    except IndexError:
        LiqVisE = "0"
except IndexError:
 LiqVisEqn = "0"
 LiqVisA = "0"
 LiqVisB = "0"
 LiqVisC = "0"
 LiqVisD = "0"
 LiqVisE = "0"


try:
  VapVis = comp.getElementsByTagName("VaporViscosity")[0]
  VapVisEqn = VapVis.getElementsByTagName("eqno")[0].
  getAttribute("value")
  VapVisA=VapVis.getElementsByTagName("A")[0].getAttribute("value")
  VapVisB=VapVis.getElementsByTagName("B")[0].getAttribute("value")
  VapVisC=VapVis.getElementsByTagName("C")[0].getAttribute("value")
  try:
      VapVisD=VapVis.getElementsByTagName("D")[0].getAttribute("value")
  except IndexError:
      VapVisD = "0"
  try:
      VapVisE=VapVis.getElementsByTagName("E")[0].getAttribute("value")
  except IndexError:
      VapVisE = "0"
except IndexError:
 VapVisEqn = "0"
 VapVisA = "0"
 VapVisB = "0"
 VapVisC = "0"
 VapVisD = "0"
 VapVisE = "0"


try:
  LiqK = comp.getElementsByTagName("LiquidThermalConductivity")[0]
  LiqKEqn = LiqK.getElementsByTagName("eqno")[0].getAttribute("value")
  LiqKA=LiqK.getElementsByTagName("A")[0].getAttribute("value")
  LiqKB=LiqK.getElementsByTagName("B")[0].getAttribute("value")
  LiqKC=LiqK.getElementsByTagName("C")[0].getAttribute("value")
```

```python
    try:
        LiqKD=LiqK.getElementsByTagName("D")[0].getAttribute("value")
    except IndexError:
        LiqKD = "0"
    try:
        LiqKE=LiqK.getElementsByTagName("E")[0].getAttribute("value")
    except IndexError:
        LiqKE = "0"
except IndexError:
 LiqKEqn = "0"
 LiqKA = "0"
 LiqKB = "0"
 LiqKC = "0"
 LiqKD = "0"
 LiqKE = "0"

try:
  VapK = comp.getElementsByTagName("VaporThermalConductivity")[0]
  VapKEqn = VapK.getElementsByTagName("eqno")[0].getAttribute("value")
  VapKA=VapK.getElementsByTagName("A")[0].getAttribute("value")
  VapKB=VapK.getElementsByTagName("B")[0].getAttribute("value")
  VapKC=VapK.getElementsByTagName("C")[0].getAttribute("value")
  try:
        VapKD=VapK.getElementsByTagName("D")[0].getAttribute("value")
  except IndexError:
        VapKD = "0"
  try:
        VapKE=VapK.getElementsByTagName("E")[0].getAttribute("value")
  except IndexError:
        VapKE = "0"
except IndexError:
 VapKEqn = "0"
 VapKA = "0"
 VapKB = "0"
 VapKC = "0"
 VapKD = "0"
 VapKE = "0"

f = open('File5.txt','a')
f.write('model '+CompName)
f.write('\n')
f.write('extends General_Properties(')
f.write('\n')
f.write('SN ' + '=' + str(i) +',')
f.write('\n')
```

80

```
f.write('name' + '=' + '"'+ CompName + '",')
f.write('\n')
f.write('Tc ' + '=' + CriticalTemp + ',')
f.write('\n')
f.write('Pc ' + '=' + CriticalPres + ',')
f.write('\n')
f.write('Vc ' + '=' + CriticalVol + ',')
f.write('\n')
f.write('Cc ' + '=' + CriticalComp + ',')
f.write('\n')
f.write('Tb ' + '=' + NormalBoilPoint + ',')
f.write('\n')
f.write('Tm ' + '=' + NormalMeltingPoint + ',')
f.write('\n')
f.write('TT ' + '=' + TripPntTemp + ',')
f.write('\n')
f.write('TP ' + '=' + TripPntPres + ',')
f.write('\n')
f.write('MW ' + '=' + MolWt + ',')
f.write('\n')
f.write('LVB ' + '=' + LiqVolAtBoilPnt + ',')
f.write('\n')
f.write('AF ' + '=' + AcenFactor + ',')
f.write('\n')
f.write('SP ' + '=' + SolParam + ',')
f.write('\n')
f.write('DM ' + '=' + DipoleMoment + ',')
f.write('\n')
f.write('SH ' + '=' + SH + ',')
f.write('\n')
f.write('IGHF ' + '=' + IGHF + ',')
f.write('\n')
f.write('GEF ' + '=' + GEF + ',')
f.write('\n')
f.write('AS ' + '=' + AbsEntropy + ',')
f.write('\n')
f.write('HFMP ' + '=' + HeatFusionMeltPnt + ',')
f.write('\n')
f.write('HOC ' + '=' + HOC + ',')
f.write('\n')
f.write('LiqDen =  { '+LiqDenEqn+" ,"+A+" ,"+B+" ,"+C+" ,"+D+" ,"+E+' },')
f.write('\n')
f.write('VP = { '+VapPresEqn+" ,"+VA+" ,"+VB+" ,"+VC+" ,"+VD+" ,"+VE+' },')
f.write('\n')
f.write('LiqCp = { '+LiqCpEqn+" ,"+LCpA+" ,"+LCpB+" ,"+LCpC+",
```

```python
"+LCpD+","+LCpE+'},')
f.write('\n')
f.write('HOV =_{ '+HOVEqn+","+HOVA+","+HOVB+","+HOVC+",
"+HOVD+","+HOVE+'},')
f.write('\n')
f.write('VapCp =_{ '+VapCpEqn+","+VCpA+","+VCpB+","+VCpC+",
"+VCpD+","+VCpE+'},')
f.write('\n')
f.write('LiqVis =_{'+LiqVisEqn+","+LiqVisA+","+LiqVisB+","+LiqVisC+",
"+LiqVisD+","+LiqVisE+'},')
f.write('\n')
f.write('VapVis =_{'+VapVisEqn+","+VapVisA+","+VapVisB+","+VapVisC+",
"+VapVisD+","+VapVisE+'},')
f.write('\n')
f.write('LiqK =_{ '+LiqKEqn+","+LiqKA+","+LiqKB+","+LiqKC+",
"+LiqKD+","+LiqKE+'},')
f.write('\n')
f.write('VapK =_{ '+VapKEqn+","+VapKA+","+VapKB+","+VapKC+",
"+VapKD+","+VapKE+'},')
f.write('\n')
f.write('Racketparam =_'+RacketParam +',')
f.write('\n')
f.write('UniquacR =_'+ UniquacR + ',')
f.write('\n')
f.write('UniquacQ =_'+ UniquacQ + ');')
f.write('\n')
f.write('end _'+CompName+';')
f.write('\n')
f.write('\n')
i = i + 1
f.close()
```

Bibliography

[1] http://spoken-tutorial.org/tutorial-search/?search_foss=DWSIM&
    search_language=English.

[2] http://spoken-tutorial.org/tutorial-search/?search_foss=
    OpenModelica&search_language=English.

[3] http://www.psenterprise.com/gproms/physprops/index.html.

[4] http://www.aspentech.com/email/email_archive/acm_webinar-final.
    pdf.

[5] http://web.thu.edu.tw/g96310005/www/6_thermo.pdf.

[6] http://www.aiche.org/dippr/about.

[7] http://www.prosim.net/en/software-simulis-thermodynamics-mixture-properties-and
    php.

[8] http://ascend4.org/FPROPS.

[9] http://www.tutorialspoint.com/python/python_classes_objects.htm.

[10] David M. Beazley. Python network programming. Technical report, 2010.

[11] Robinson DB Peng DY. A new two-constant equation of state. *Industrial and
     Engineering Chemistry Fundamentals*, 15(1):59–64, 1976.

[12] Peter Fritzson. *Principles of Object-Oriented Modelling and Simulation with
     Modelica 2.1*. Wiley Interscience, 2006.

[13] Soave G. Equilibrium constants from a modified redlich-kwong equation of state.
     Chemical Engineering Science, 27(6):1197–1203, 1972.

[14] Prausnitz JM Renon H. Local compositions in thermodynamic excess functions
     for liquid mixtures. *AICHE Journal*, 14(135):135–144, 1968.

[15] L.H. Horsley. Azeotropic data, iii. *American Chemical Society*, pages 1–1, 1973.

[16] John Ingham. *Chemical Engineering Dynamics Modelling With PC Simulation*.
     VCH Publishers, 1992.

[17] Daniel Medeiros. Dwsim technical document. Technical report, 2015.

[18] Kannan M Moudgalya. Campaign for it literacy through foss and spoken tutorials. 2014.

[19] Polling Reid, Prausnitz. *The Properties of Gases and Liquids*. McGraw-Hill, 1987.

[20] Endre Rev. Process modelling and simulation. Technical report, Budapest University of Technology and Economics, Dept. of Chemical Engineering, 2013.

[21] Mordechai Shacham. Semi batch steam distillation of a binary organic mixture. technical report. Technical report, 2009.