AMOSS: IMPROVING SIMULATION SPEED AND NUMERICAL STABILITY OF LARGE-SCALE MIXED CONTINUOUS/CONDITIONAL STOCHASTIC DIFFERENTIAL SIMULATIONS

Deon Pretorius

Amoss: Improving Simulation Speed and Numerical Stability of Large-Scale Mixed Continuous/Conditional Stochastic Differential Simulations

by

Deon Pretorius

A dissertation in partial fulfillment
of the requirements for the degree

Master of Engineering (Control Engineering)

in the

Department of Chemical Engineering
Faculty of Engineering, the Built Environment and Information Technology

University of Pretoria
Pretoria

January 2021

SYNOPSIS

Amoss is an equation-orientated stochastic simulation platform, developed on open-source software. It is designed to facilitate the development and simulation of Sasol value chain models using the Moss methodology. The main difficulties with the original Moss methodology was that plant recycles were difficult to incorporate and that plant or model changes meant rebuilding the entire Moss model. The first version of automatic-Moss was developed by Edgar Whyte in an effort to address these problems. It was successful as a proof of concept, but generated simulations were numerically unstable and very slow. A second version of the tool was to be developed to address numerical stability and simulation speed.
The stochastic simulations stemming from Amoss models are large-scale and contain mixed continuous/conditional algebraic equation sets, with first order stochastic differential equations. Additionally, optimal flow allocation as a disjunctive optimisation is often encountered. The complexity of these factors makes finite difference approximation the main solution.
The equation ordering, simulation approach and code generation features of the Amoss tool were investigated and re-implemented. A custom equation ordering method, which uses interval arithmetic and weighted maximal matching for numerically stable matching, followed by Dulmage-Mendelsohn decomposition and Cellier's tearing, was implemented. For implicitly ordered systems, a fixed-point iterative Newton method, where conditional variables are separated from continuous variables for solving stability, was implemented. The optimal allocation problem with heuristic allocation was generalised to plants with recycles. Fast simulation code utilising parallel processing, efficient solving and function evaluation, efficient intermediate data storage and fast file writing, was implemented. Amoss simulations are now substantially faster than the industry equivalent and can reliably model Moss methodology problems.

KEYWORDS: stochastic simulation, simulation speed, numerical stability, equation ordering, simulation approach, code generation

ACKNOWLEDGEMENTS

I would like to thank my project mentor, Carl Sandrock, for the time set aside for guidance, feedback and counsel throughout the project. I would like to thank my predecessor, Edgar Whyte, for the outstanding groundwork done on the Amoss platform. I would like to thank Gerrit Streicher, for the effort put into the development of the industrial Moss models and his feedback on the Amoss project. Finally, I would like to thank Sasol for their continued financial support.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Part I

Background

CHAPTER 1

INTRODUCTION

1.1 Moss

Sasol is a large chemical provider located in South Africa and is one of the world's largest producers of liquids and gasses from coal. The Fischer-Tropsch process to convert low grade coal to synthetic fuels revolutionized the petrochemical industry in 1955, and was pioneered by Sasol. Of late Sasol has also heavily modernised its classical processes and become a manufacturer of many types of chemicals. Sasol was estimated to be worth $23 billion in 2009. It is a top 5 South African publicly listed company and finds itself listed on both Johannesburg and New York stock exchange. (Meyer *et al*, 2011)

Sasol has historically faced many challenges. One of the major difficulties has been decision-making on the highly integrated and dynamic processes found on its plants. The challenges experienced can be summarised as follows (Meyer *et al*, 2011):

1. Production facilities are highly integrated and complex.
2. Chemical and fuel production contains continuous, semi batch and batch processes.
3. Many factors affect fuel and chemical components
4. Production information is limited (quality and availability).

5. Different businesses are involved with various goals.

The original approach to accommodate these factors in plant decisions has been to use average-based models. The first method makes use of plant mass balance models, often implemented on spreadsheets. The second method incorporates a linear program (LP) to processes for planning and economic optimisation. The biggest problem experienced with these methods is that the non-linear and stochastic nature of Sasol's processes are not adequately taken into account. This problem was to be addressed by the Moss methodology. (Meyer *et al*, 2011)

The Moss (modelling of operations using stochastic simulation) methodology was introduced by Sasol's operations research team and makes use of discrete-event stochastic simulation to model the complex Sasol processes. The idea is to take continuous petrochemical systems and consider the flow of liquids and gases as blocks of volume. These blocks of volume can be tracked through the process at regular time intervals. This can be combined with actual operating rules and failure events, which occur on the plants. This concept proved highly successful for decision making on plants and was shown to clearly be an improvement over average-based methods. The Moss methodology can be summarised as follows (Meyer *et al*, 2011):

1. Translation of continuous plant to a flow sheet model.
2. Translation of plant events to discrete events (incidents). This includes fitting probability density functions to plant data and including operating rules.
3. Identification of failures, the frequency of occurrence and time required to fix these failures. These failures are included as simulation input.
4. Set-up optimisation problems, like fuel blending.
5. Validation of model against current conditions.
6. Analysis of the simulated scenarios for decision-making support.

Because of the complexity of the Sasol plants, an "off-the-shelf" model was not available. In addition to this, no investigated software tool could successfully incorporate all the factors required by the highly complex Moss models. The main commercial products investigated were AnyLogic (The AnyLogic Company, 2017) and Simul8 (SIMUL8 Corporation, 2017).

Anylogic is a modular orientated simulation platform with an interactive GUI. There is support for stochastic variables, development of statistical models and stochastic simulation. It can accommodate dynamics in systems, but cannot solve algebraic loops. This makes addition of equations stemming from operating instructions impossible. In addition to this, multi-component flow models are not accommodated.

Simul8 is a discrete-event simulation platform. It satisfies the requirement for discrete-event accommodation, but lacks accommodation for continuous system dynamics. Handling continuous flow was found to be problematic.

In addition to the simulation troubles experienced with the commercial products investigated, the high licensing fee of these types of products was an issue. Due to the incapability of commercial software to incorporate the complexity of the Moss models, the operations research team at Sasol were forced to manually build the highly complex Moss simulations. Use was made of the VBA programming language. These models were difficult to derive, took a large amount of time to develop and had to be entirely rebuilt when a plant change occurred. However, despite the large amounts of challenges faced by the Moss modelling methodology, many of the Sasol operations were successfully modelled. These models could now accurately and reliably model actual plant behaviour, in a far superior manner to previous methods.

1.2 Justification of Amoss

The Moss methodology was a success, but had various shortcomings. The biggest problems faced by the Moss methodology were development time and difficulty in accommodation of plant recycles. Because of the Moss model complexity, developing a full simulation in VBA could take months at a time. In addition to this, Sasol processes often go through changes. This is highly problematic, since the VBA simulations would have to be mostly rebuilt. Secondly, a model based on the Moss methodology was simple to develop when only forward flow was encountered. However, introducing plant recycles made manual model development very difficult, because streams would have to be simultaneously solved.

Because of the challenges faced by the Moss methodology, the University of Pretoria was contacted. The existing optimisation and steady-state models were handed over to the university as reference. The Amoss project or Automatic-Moss was aimed at being an extension of the Moss simulation methodology, with the main focus of automatically generating the Moss models for simulation. If the Moss models were generated directly, both the challenge of development time and the problems associated with plant recycles could be addressed. The deliverables agreed upon by Sasol and the University of Pretoria, aimed at measuring the success of the Amoss project, are the following (Whyte, 2018):

Reduction in development time As discussed, it currently takes too long to build the Moss models. Amoss should reduce development time significantly and ideally not be heavily affected by model complexity.

Generic application Amoss must be able to model a Sasol value chain irrelevant of complexity and configuration. There is also a need to convert the legacy models to the new modelling solution. One of the measures of success of the Amoss model will be how easily this conversion can be done.

Development flexibility Once the basic model exists, a modification to the model should be simple to incorporate.

Simulation flexibility During simulation, the ability to activate or deactivate a particular portion of the plant may be required. The idea is that significant changes can coexist in a single model, instead of having to build separate models.

Acceptable accuracy The simulation must be sufficiently accurate to answer questions arising from various scenarios. The accuracy must be independent of model complexity.

Fit for purpose The modelling environment must be focused on supporting the Sasol Moss methodology and also have the ability to add features.

Linear scalability The resulting simulation speed must scale linearly i.e. if the number of equations describing the system is increased by double, the simulation may at most take twice as long to simulate.

Quick learning curve A quick learning curve is expected for the end user. It is accepted that a steeper learning curve will be experienced by a developer.

Software cost Simulation packages can be quite expensive and it will make little sense for Sasol to purchase a tool that only partially solves the problem. The initial cost of the tool, as well as the annual maintenance cost, must thus be low. It is also required that continual support be provided by the University of Pretoria for further development of the tool.

Version control of model development As more than one user can use the same model at any time, the solution must allow for simultaneous modification and development in a controlled manner.

Debug capability The solution must be able to guide the user to quickly and easily to locate a bug in a faulty model. The solution must be able to replicate an error situation in the same replication and scenario in which it has occurred.

Cause identification Sometimes a model gives counter-intuitive results. It is difficult to expect the modeller to identify the reason for an outcome of this nature. The tool must assist in identifying causes for results and assist in determining bottlenecks.

Fast simulation time The simulation time must be as short as possible as this will allow for an increased number of replications per scenario as well as more runs during validation and verification of the model.

Software package stability Amoss must be stable for all cases. The value of each variable must be calculated at each time increment for all replications of all scenarios.

1.3 Amoss 1.0

The first master's student involved with the Amoss project was Edgar Whyte. His dissertation was completed in January 2018. He was tasked with developing the Amoss platform from the ground up as a proof of concept. The first version of the tool, Amoss 1.0, was developed. Most deliverables were addressed, although not necessarily optimally due to time constraints. It was decided that Amoss 1.0 would be developed in the Python programming language. The advantages of developing Amoss 1.0 in Python include the following (Whyte, 2018):

- Python has a large variety of libraries created and maintained by a large community.
- The Python Software Foundation License (PSFL) allows Python to be used and redistributed without paying royalties.
- The PSFL does not place any restriction on the type of licensing restrictions on software developed in Python. It is thus possible to build Amoss on open source software, but not make it open source.
- Because of Python's large community, support is available for new Python programmers.
- Python's accessibility and support enables more people to be part and contribute to the Amoss project.

The three main packages used for model development and simulation in Amoss 1.0 were (Whyte, 2018):

OpenModelica The OpenModelica Connection Editor (OMEdit) was used for process diagram development. This decision makes sense, because the OMEdit environment comes with a fully connective flow sheet interface, a simple way of defining custom units and a simple export file flow sheet representation, which can easily be read. This representation was then used to do equation generation for plant continuous and ordinary differential equations.

Atom The Atom text editor was used to allow the user to define the plant operating instructions. This was mainly due to Atom's ability to identify Python syntax. The operating instructions could then be developed with Python syntax in the Atom text editor. The operating instructions were then used to do equation generation for the additional equations.

Python The main code base was developed in Python. This included stochastic input generation, equation generation, equation ordering, simulation generation, parallel processing, simulation, storing simulation results and the graphical user interface (GUI).

Making use of the above packages, Amoss 1.0 could be used as a generic simulation tool aimed at addressing the deliverables. A high-level description of the work-flow is illustrated in Figure 1.1 and follows the following steps (Whyte, 2018):

Step 1 requires the user to draw a process diagram in OMEdit and provide all unit specific data.

Step 2 takes the OMEdit diagram, which contains process connectivity and operational unit information and parses it to a network graph.

Step 3 iterates through the graph and creates the necessary continuous equations and differential equations describing the system.

Step 4 tears the continuous equations using the block lower triangular form and symbolically solves the smaller blocks. This is referred to as pre-solving. A subset of the complete model equations is solved, before the entire model is solved for simulation. Pre-solving is used to reduce the border width of the final ordering, making it easier to solve the total system.

Step 5 requires the user to define the operating instructions, which are equations describing how the plant units are operated. In addition to this, the user is required to define stochastic variables.

Step 6 takes the pre-solved continuous equations and differential equations in Step 4 and concatenates it with the operating instruction equations and stochastic inputs in Step 5. This total system is then torn using the block lower triangular form, producing a full final system of equations.

Step 7 simulates the combined process and generates results.

Amoss 1.0 addressed most of the deliverables, but was far from being a well-rounded tool. The deliverables sufficiently addressed are as follows (Whyte, 2018):

Reduction in development time This is the largest improvement offered by Amoss 1.0. To be able to automatically generate a simulation from a flow diagram and operating instructions will drastically reduce development time. In the original Moss methodology models had to manually be derived, which could take months. In addition to this, the Moss methodology had an inability to accommodate for flow diagrams where plant recycles were encountered. A model which could take months to develop in Moss can be developed in about a week using Amoss 1.0.

Generic application Amoss 1.0 was developed as a generic stochastic simulation platform. A process diagram and operating instruction file can be used to develop a model of a process with almost any configuration. It is only required that the units used be defined in Amoss 1.0.

Development flexibility Changing the operating instructions in the original Moss methodology posed a huge problem. This is because the additional equations added by the operating instructions would require the original Moss model to again be symbolically derived. This could set the developer back significantly. However, making a change to an existing Amoss 1.0 model simply requires modifying the flow diagram (and the affected input tables) and the operating instructions. This takes minimal time and effort.

Simulation flexibility In Amoss 1.0, sections of the plant can be activated and deactivated by simply adding an if-block in the operating instructions. This if-block will be defined in a way that inputs to a plant section will be set to zero (off) or the allocated values (on). This simple modification is possible, because of the equation orientated approach followed by Amoss 1.0.

Acceptable accuracy On the condition that the Newton solver converges to a solution, an acceptable tolerance can be specified. An absolute tolerance of 1×10^{-6} was selected. Due to this, the accuracy of the algebraic equations can be guaranteed. On the other hand, the accuracy of integration is not considered. Because Euler integration is used, the accuracy of integration can vary. If the time constants of the buffer tanks are small, accuracy can be poor. However, if the buffer tank time constants are large the integration can be sufficient.

Fit for purpose Amoss 1.0 was built from the Moss methodology and was developed with guidance of Sasol. Amoss 1.0 is written entirely in Python, which has a large community. Thus, it is relatively simple for a person with moderate programming experience to contribute to the Amoss project.

Linear scalability Linear scalability as defined by Sasol initially did not take model complexity into account. A complex model, which in the case of Amoss models would contain numerous recycles or many unsafe assignments, would introduce large regions to simultaneously solve or numerical instability due to possible zero division. Linear scalability is thus not an achievable goal. It is also uncommon in practice to find problems that scale linearly. Even solving a set of linear equations has a worst-case time of $O(n^3)$. Since the systems encountered in Amoss are non-linear in nature, even worse scaling is expected.

Quick learning curve The learning curve is low to moderate. The most difficult part is to learn basic Python syntax for creating the operating instructions and learning to create OpenModelica flow sheets. Other requirements to create a simulation are editing text files and simple spreadsheets. A modeller with no prior knowledge of Python grammar or OpenModelica was able to create small process models within a week.

Software cost The software cost for Amoss 1.0 is very low compared to other commercial software. There is no licensing fee, with the only expense being the funding cost for the University of Pretoria. The Amoss 1.0 package is developed entirely on open source software like Python and OpenModelica and thus carries no additional cost for redistribution.

Version control of model development The University of Pretoria has agreed to continue development beyond the delivery date at the end of 2017. Version control of the project is done with git using Bitbucket as a cloud repository. A developed model can be shared using git and the repository. Alternatively, models can simply be shared using Google Drive.

Debug capability Rudimentary debugging is added in the form of the information window of the GUI. Common errors a modeller could make have been identified and will trigger the display of an appropriate error message, recommending a course of action. An example would be the degree of freedom check. In the event of the degree of freedom not being zero, the number of overspecified or underspecified variables is determined. This would trigger the message ”DOF is not zero. Please add/remove n inputs.”

Cause identification Rudimentary cause identification can be added by identifying when an if-, elif- or else- condition has changed. This identification is possible, because of the way that if-blocks are parsed and converted to equations. The statements of an if-block are parsed to equations using a base if-variable which evaluates to 1 if the condition is true and 0 if the condition is false. A list linking the created if-variables to the statement is made available to the modeller.

There are two main deliverables, which were not met. These are *fast simulation time* and *software package stability*. The problems related to the two deliverables can be summarised as follows (Whyte, 2018):

Fast simulation time The generated simulations of Amoss 1.0 are particularly slow. Even for smaller test processes, Amoss 1.0 simulated more than an order of magnitude slower than the developed VBA code by the Sasol operations research team. It should be noted that the slow simulation speed is a result of the numerical instability and large number of equations to solve simultaneously. This is a consequence of the equation ordering method not performing well and resulting zero division.

Software package stability The stability of the Amoss 1.0 front-end as a whole should also be addressed. The GUI is considered unstable and many problems are experienced. An example problem encountered is that the simulation settings do not properly save in a single session. In addition to this, the GUI is prone to crashing.

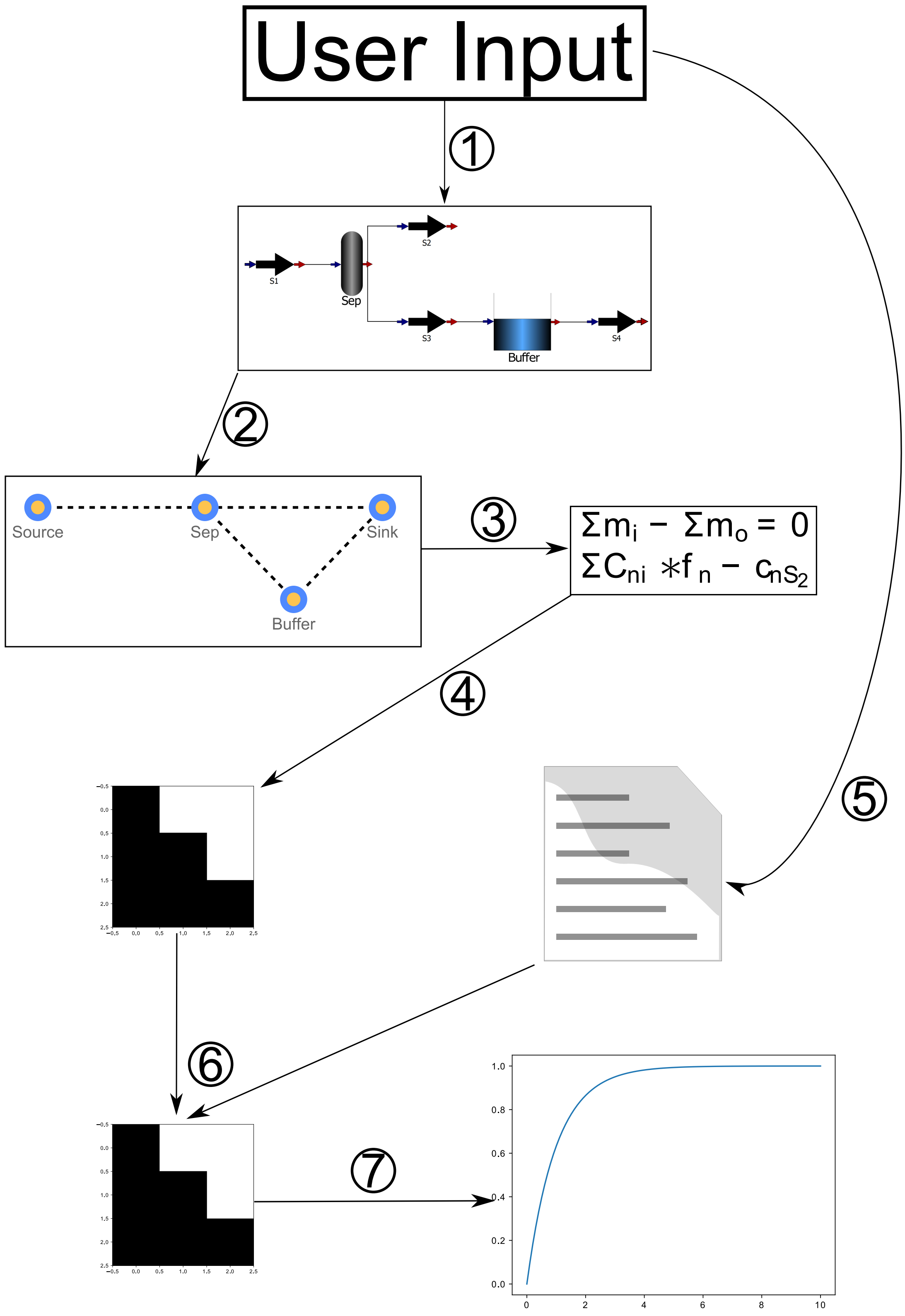


Figure 1.1: Amoss 1.0 work flow (Whyte, 2018), with steps discussed in Section 1.3.

1.4 Amoss 2.0

Considering Amoss 1.0, there is significant room for improvement. Major stumbling blocks encountered were slow simulation speed and numerical instability. Simulations often encountered floating-point errors from zero division or generated infeasible results, such as negative flows or values incorrect by orders of magnitude. A second version of the tool, Amoss 2.0, is required. It should be an improvement regarding the simulation speed and numerical stability of Amoss 1.0. It has also become a priority of the Amoss project to develop the tool as a stand-alone or independent package. The deliverables to be addressed by Amoss 2.0 are as follows:

Fast simulation time The simulation time should be as short as possible. This would allow for more replications per scenario to be run. In turn, this would increase the reliability of the Monte Carlo analysis.

Numerical stability The generated simulations should be as numerically stable as possible. The goal is to not encounter zero division.

Acceptable accuracy The simulation accuracy should be sufficient to answer the questions arising from various scenarios.

Linear scalability Simulation time should scale approximately linearly with model complexity.

Reduction in development time Simulation development time should be reduced from what was required for the Moss simulation development method.

Independent package The tool should be developed with the aim of becoming a stand-alone or independent package. That is to say, that only open source Python packages should be used as dependencies.

To achieve a faster and more numerically stable simulation three main factors will be revisited. These are:

Equation ordering The equation ordering method used to order the augmented algebraic equation set is critical for stable and fast simulation. If an inadequate ordering is obtained, zero division or many equations to simultaneously solve can be encountered.

Simulation approach The simulation approach is essential for fast and stable simulation. Alternative simulation approaches could simulate models in a faster or more numerically stable manner.

Code generation The simulation code should be generated to execute effectively and quickly. Numerical instability can not truly be addressed in code generation, but simulation code which is memory efficient and stable should be generated.

CHAPTER 2

THEORY

2.1 Problem

Mathematical problems stemming from the Moss methodology consist of various elements. Generally, problems are stochastic differential algebraic equation systems with continuous and discrete variables, time-series inputs and possible disjunctive constraint optimisation. This section aims to be a high-level summary of problem formulation and solution.

2.1.1 Formulation

To simplify mathematical notation, variables are grouped as input variables (i), continuous variables (x) and discrete variables (y). Continuous variables have the domain $x \in R$ and discrete variables have the domain $y \in \{d_1, d_2, d_3, \ldots, d_n\}$, where d_i is a possible value assignment. Input variables can have various domains, depending on input definition.

Time-invariant stochastic input variables are generated using user-provided tabular probability data. The probability distribution p is specified as being either continuous or discrete. Stochastic input variables are distributed in time by p, as shown in Equation 2.1.

$$i(t) \sim p \tag{2.1}$$

Custom input variables are provided as tabular time-series data. Input variables of this type are described by Equation 2.2, where c is an interpolation function of the tabular data.

$$i(t) = c(t) \tag{2.2}$$

Non-linear continuous algebraic equations (f) are generated from the flowsheet model and certain operating instructions. Equations are of the form in Equation 2.3.

$$f(i, x, y) = 0 \tag{2.3}$$

Ordinary differential equations (g) are encountered due to integrators in the flowsheet model. Equations can be written in the residual form in Equation 2.4.

$$g(\frac{dx}{dt}, i, x, y) = 0 \tag{2.4}$$

Discrete equations (h) are generated from the operating instructions. Equations can be written in the form shown in Equation 2.5.

$$h(i, x, y) = 0 \tag{2.5}$$

Optimal flow allocation can be specified in the operating instructions. This instruction requires economic optimal distribution of mass flow to unit operations. It imposes disjunctive constraints on the relevant continuous variables (x_{opt}), as shown in Equation 2.6 to Equation 2.8. Additionally, the problem obtains the economic minimisation objective described in Equation 2.9.

$$\sum x_{opt} \leq x_{available} \tag{2.6}$$

$$x_{opt} \leq x_{max} \tag{2.7}$$

$$x_{opt} = 0 \text{ or } x_{opt} \geq x_{min} \tag{2.8}$$

$$\min_{x_{opt}} f(x_{opt}) \tag{2.9}$$

2.1.2 Solution

Due to model complexity, the main solution has been a Monte Carlo simulation approach, with fixed-step ordinary differential equation simulation. For each replication (r), at each time-step, input variables (i) are sampled. Continuous (x) and discrete (y) variables are solved from problem equations and/or optimisation, using the sampled inputs. Derivatives of continuous variables ($\frac{dx}{dt}$) are then determined and used for integration, using the explicit Euler method. After each time-step and after each replication, simulation results are stored. This is used for analysis after the complete simulation. The overall method is illustrated in Algorithm 1.

Algorithm 1 Overall solution approach

```
for $r$ in replications do
    for $t$ in time do
        Sample $i(t)$ from Equation 2.1 and Equation 2.2.
        if not optimal flow allocation then
            Solve $x$ and $y$ from Equation 2.3 and Equation 2.5.
        else
            Solve $x$ and $y$ by optimising Equation 2.9,
            with Equation 2.3 and Equation 2.5 as equality constraints,
            and Equation 2.6 to Equation 2.8 as inequality constraints.
        end if
        Solve $\frac{dx}{dt}$ from Equation 2.4.
        Increase $x$ by $\frac{dx}{dt} * dt$.
        Save results of $x$ and $y$ at time $t$.
    end for
    Save $r$ results.
end for
```

2.2 Stochastic simulation

2.2.1 Probability distribution

A probability distribution is a description of the possible set of outcomes for a sample space with a method to determine probabilities. Closely related to the probability distribution are probability density function and probability mass function. A probability density function is a function used to calculate probabilities and to specify the probability distribution of a continuous random variable (Montgomery & Runger, 2003: 698). For a continuous random variable X, a probability density function is such that:

1. $f(x) \geq 0$
2. $\int_{-\infty}^{\infty} f(x)\,dx = 1$
3. $P(a \leq X \leq b) = \int_a^b f(x)\,dx =$ area under $f(x)$ for any a and b

(Montgomery & Runger, 2003: 99)

The cumulative distribution function of a continuous random variable X can be determined using Equation 2.11.

$$F(x) = P(X \leq x) = \int_{-\infty}^{x} f(u)du \tag{2.10}$$

for $-\infty \leq x \leq \infty$
(Montgomery & Runger, 2003: 102)

A probability mass function is a function that provides probabilities for the values in the range of a discrete random variable (Montgomery & Runger, 2003: 698). For a discrete random variable X, with possible values $x_1, x_2, x_3, \ldots, x_n$, a probability mass function is such that:

1. $f(x_i) \geq 0$
2. $\sum_{i=1}^{n} f(x_i) = 1$
3. $f(x_i) = P(X = x_i)$

(Montgomery & Runger, 2003: 62)

The cumulative distribution function of a discrete random variable X can be determined using Equation 2.11.

$$F(x) = P(X \leq x) = \sum_{x_i \leq x} f(x_i) \tag{2.11}$$

(Montgomery & Runger, 2003: 64)

2.2.2 Input generation

Distribution data for simulation is provided as a user input, consisting of variable values (x), probabilities ($f(x)$) and specification whether the distribution is discrete or continuous. Generating non-uniform stochastic inputs using distribution data is necessary for stochastic simulation and sequential Monte Carlo analysis.

The inversion principle is essential for non-uniform variable generation for both continuous and discrete distributions. Given a cumulative distribution function F, its inverse F^{-1} is defined by Equation 2.12, where *inf* denotes the infimum. The infimum is the greatest lower bound of a dataset and serves the function of providing a unique F^{-1} value for every u. If u is a uniform $[0, 1]$ random variable, then F^{-1} has a distribution F. (Devroye, 1986)

$$F^{-1}(u) = \mathit{inf}\{x : F(x) = u,\, 0 < u < 1\} \tag{2.12}$$

The inversion principle, as illustrated in Equation 2.12, states that given a cumulative distribution function F, its inverse F^{-1} can be used for random variable generation with the original distribution properties left intact. To illustrate this concept, consider the example probability distribution shown in Figure 2.1. If the probability distribution represents a continuous random variable, trapezoidal integration can be used to obtain the cumulative distribution shown in Figure 2.2. Alternatively, if the probability distribution represents a discrete random variable, the probabilities can be added to obtain the cumulative distribution shown in Figure 2.3.

The inversion principle is simple to illustrate graphically. As an example, a randomly generated input u has a value of 0.2. This u value can be used as the y-axis value for interpolation using the cumulative distribution graphs. The relevant x value can then be determined. For the continuous cumulative distribution shown in Figure 2.2, an x value of 2.67 is obtained. Using the discrete cumulative distribution shown in Figure 2.3, an x value of 3.00 is obtained. It is important to note that the infimum function in Equation 2.12 implies that if multiple x values are possible from interpolation, the minimum x or left graphical value should be used.

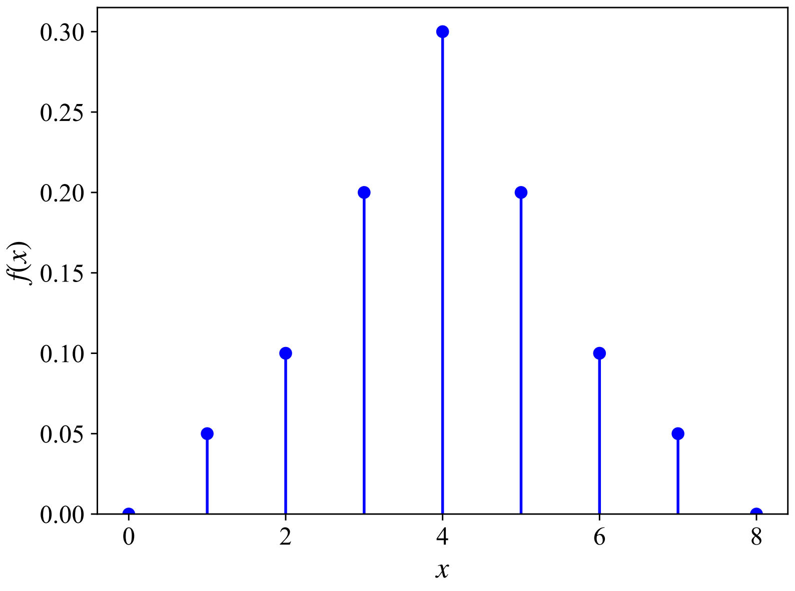


Figure 2.1: Example probability distribution

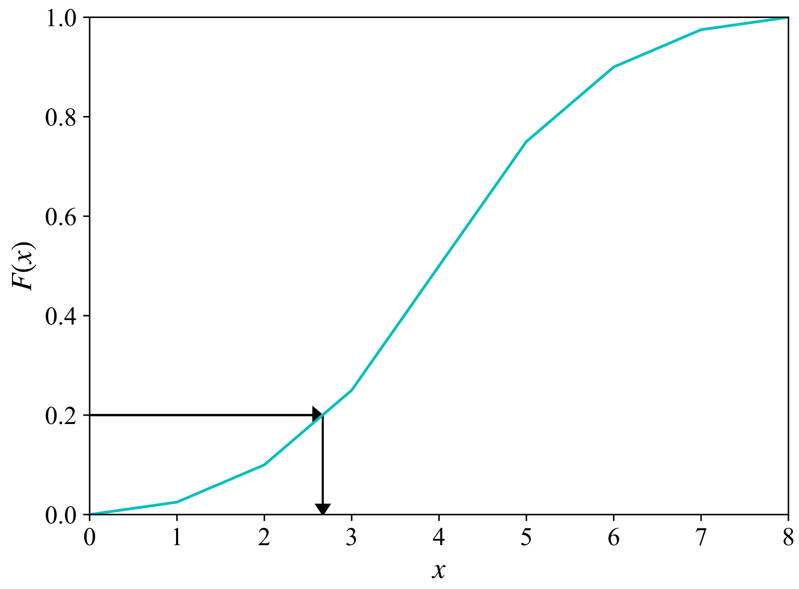


Figure 2.2: Example continuous cumulative distribution function

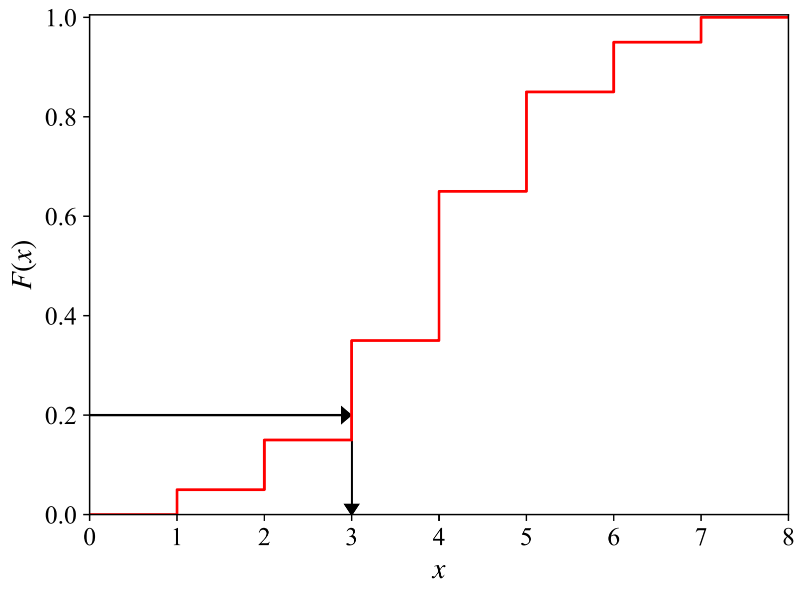


Figure 2.3: Example discrete cumulative distribution function

The interpolation approach to the inversion principle is easy to implement and was used by Whyte (2018) to generate non-uniform values for stochastic simulation in Amoss 1.0. With n denoting the number of time-steps, the steps followed to generate data for a random variable x for a single replication are as follows:

1. Generate cumulative distribution data for continuous or discrete random variable x, using user input distribution data.
2. Select a seed for random number generation.
3. Generate a vector U of size n consisting of pseudo-random values in the interval $[0, 1]$, using a random number generator.
4. Generate vector X of size n, consisting of stochastic input x values throughout the simulation, by interpolating the cumulative distribution for each value u in vector U.

2.2.3 Monte Carlo method

Monte Carlo methods are algorithms that rely on repeated random sampling to obtain numerical results. Randomness is used to simulate effects which are deterministic in nature. These methods are used to solve difficult mathematical and physical problems which could be impossible to solve otherwise. Monte Carlo methods are mainly used in three classes of problems: optimisation, numerical integration and generating draws from a probability distribution (Kroese *et al*, 2014). Mooney (1997) discusses the basic Monte Carlo procedure as follows:

1. Determine a distribution function for a stochastic input variable.
2. Sample the stochastic variable in a way that will approximate real behaviour.
3. Calculate an estimated value for an expected variable $\mathbb{E}(X)$, which is dependent on the stochastic input.
4. Repeat steps 2 and 3 for as many replications (r) as desired.
5. Draw a frequency distribution of the resulting μ_r values which are the Monte Carlo estimates of μ.

This method is best illustrated by an example. The classical example followed is that of approximating the value of π. The basic procedure is as follows (Kalos & Whitlock, 2009):

1. Draw a square and make a quadrant within it.

2. Generate n number of random coordinate points to distribute in the square.
3. Count the number of points in the quadrant (i.e. radius smaller than 1).
4. The ratio of the number of points in the quadrant to the total number of points can be used to approximate the ratio of the two areas.
5. The ratio of the areas is approximately $\frac{\pi}{4}$, so the area ratio should be multiplied by 4 to approximate π.
6. Repeat steps 2 to 5 for the desired number of replications (r).
7. Generate statistical data from the various replicated runs.

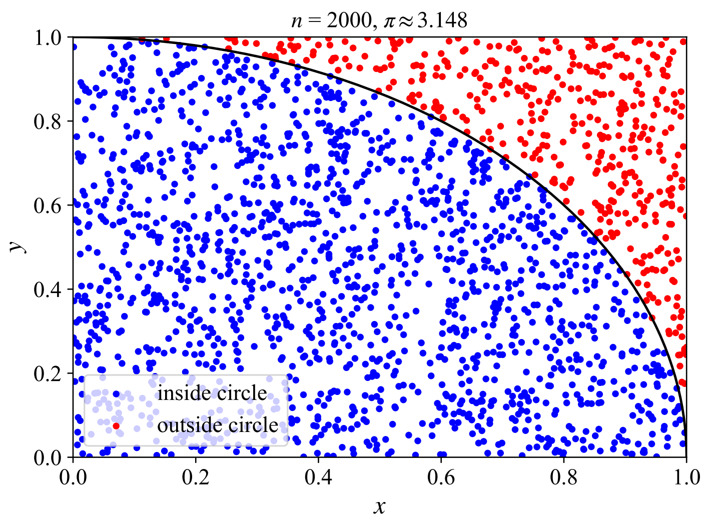


Figure 2.4: Monte Carlo method for π approximation

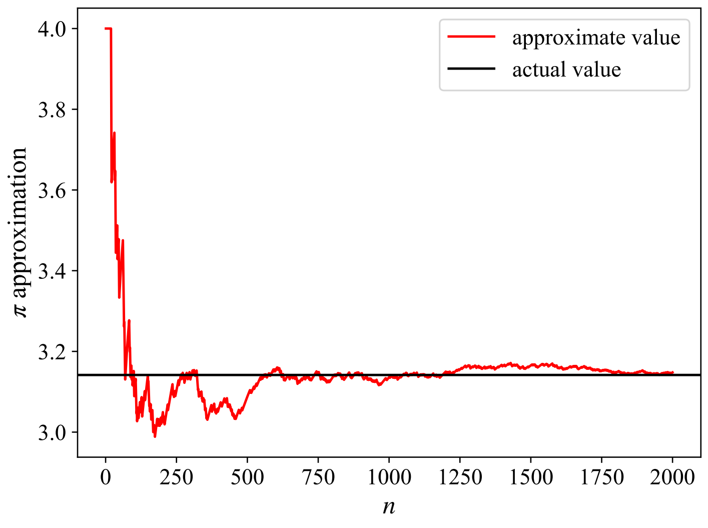


Figure 2.5: Convergence of Monte Carlo method for π approximation

A single replication (r = 1) of the Monte Carlo method with 2000 samples (n = 2000) is shown in Figure 2.4 and Figure 2.5. Commonly, multiple replications are done to generate distribution data and the distribution mean used to more accurately approximate the variable. Korn, Korn & Kroisandt (2010) refer to this experimental-based Monte Carlo method as the crude Monte Carlo method. The crude Monte Carlo method approximates the arithmetic mean of an expected variable $\mathbb{E}(X)$ using Equation 2.13. $X_i(\omega)$ are the independent experimental results, with X as probability distribution.

$$\frac{1}{n}\sum_{i=1}^{n} X_i(\omega),\ N \in \mathbb{N} \tag{2.13}$$

The strong law of large numbers is fundamental to the validity of the crude Monte Carlo method and is shown in Theorem 1.

Theorem 1. *Let $(X_n)_{n\in N}$ be a sequence of integrable, real value random variables that are independent, identically distributed (i.i.d.) and defined on a probability space $(\Omega, \mathcal{F}, \mathbb{P})$. Let further:*

$$\mu = \mathbb{E}(X_1) \tag{2.14}$$

Then we have for $\mathbb{P}$-almost all $\omega \in \Omega$

$$\frac{1}{n}\sum_{i=1}^{n} X_i(\omega) \xrightarrow{n\to\infty} \mu \tag{2.15}$$

i.e. the arithmetic mean of the (realisations of) X_i tends to the theoretical mean of every (X_i), its expectation μ (Korn et al*, 2010).*

The strong law of numbers implies almost sure convergence. We can now consider the accuracy of the crude method. The unbiased Monte Carlo estimator is defined in Theorem 2.

Theorem 2. *Let $(X_n)_{n\in N}$ be a sequence of integrable, real value random variables that are independent, identically distributed (i.i.d.) and defined on a probability space $(\Omega, \mathcal{F}, \mathbb{P})$. Then the Monte Carlo estimator*

$$\bar{X}_N := \frac{1}{N}\sum_{i=1}^{N} X_i,\ N \in \mathbb{N} \tag{2.16}$$

is an unbiased estimator for $\mu = \mathbb{E}(X)$, i.e. we have

$$\mathbb{E}(\bar{X}_N) = \mu \tag{2.17}$$

(Korn et al*, 2010)*

To gain insight into the error we consider the difference in the standard deviation between μ and $\bar{X}_N$. The variance is shown in Equation 2.18. The standard deviation is thus of the order $O(1/\sqrt{N})$. As a consequence, to increase the accuracy of the mean by one digit (reducing the standard deviation by a factor of 0.1) requires increasing the number of runs by a factor of 100.

$$\text{Var}(\bar{X}_N - \mu) = \text{Var}(\bar{X}_N) = \frac{1}{N^2}\sum_{i=1}^{N}\text{Var}(X_i) = \frac{\sigma^2}{N} \tag{2.18}$$

The use of standard deviation for error of a Monte Carlo estimator can be justified by using the central limit theorem. The central limit theorem is shown in Theorem 3.

Theorem 3. *Let $(X_n)_{n\in N}$ be a sequence of integrable, real value random variables that are independent, identically distributed (i.i.d.) and defined on a probability space $(\Omega, \mathcal{F}, \mathbb{P})$. Assume further that they all have a finite variance $\sigma^2 = Var(X)$. Then the normalized and centralized sum of these variables converges in distribution towards the standard normal distribution i.e. we have*

$$\frac{\sum_{i=1}^{N} X_i - N\mu}{\sqrt{N}\sigma} \xrightarrow{D} \mathcal{N}(0,1) \ as\ N \to \infty \tag{2.19}$$

(Korn et al*, 2010)*

Looking at the central limit theorem, it can be seen that the crude Monte Carlo estimator is approximately $\mathcal{N}(\mu, \sigma^2/N)$ distributed for large N values. From this, the 2σ approximate 95%-confidence interval shown in Equation 2.20 can be derived. Typically, the true standard deviation is not known. The standard deviation can be approximated from the gathered data using Equation 2.21. (Korn *et al*, 2010)

$$\left[\frac{1}{N}\sum_{i=1}^{N} X_i - 2\frac{\sigma}{\sqrt{N}}, \frac{1}{N}\sum_{i=1}^{N} X_i + 2\frac{\sigma}{\sqrt{N}}\right] \tag{2.20}$$

$$\bar{\sigma}_N = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(X_i - \bar{X_N})^2} \tag{2.21}$$

When considering the generated Amoss models, Monte Carlo methods are essential. When the Moss methodology was originally introduced, a large improvement over the existing methods was the use of continuous stochastic variables and Monte Carlo methods rather than simply assuming average variable values. This analysis provides better mean values and reliable confidence intervals. In addition to this, discrete stochastic variables can compensate for real plant phenomena like disturbances.

2.2.4 Stochastic differential equations

Dlouhỳ, Fábry & Kuncová (2005) describe stochastic simulations as simulations which track the evolution of variables that change stochastically (randomly) with a particular probability. Stochastic differential equations have been researched in some detail with significant research in the fields of Itô and Stratonovich differential equations. The stochastic elements in these differential equation forms are described by Wiener processes. Durrett (1996) describes a Wiener process, denoted by W_t, as a continuous-time stochastic process with the following properties:

- $W_0 = 0$
- W has increments independent of previous values
- W has Gaussian increments or is normally distributed
- W has continuous paths or W_t is continuous in t

A d-dimensional stochastic differential equation system can be described in Itô differential form by Equation 2.22. Alternatively a system can be described as a Stratonovich differential in the form given by Equation 2.23. It should be noted that W_t is an m-dimensional process with time-independent scalar Wiener process components.

$$dX_t = f(X_t)dt + g(X_t)dW_t \tag{2.22}$$

$$dX_t = f(X_t)dt + g(X_t) \circ dW_t \tag{2.23}$$

The alternative forms of stochastic differential equations are equivalent and conversion can be done from one form to the other. If we have an Itô system described by Equation 2.22, it can be converted to

$$dX_t = \bar{f}(X_t)dt + g(X_t) \circ dW_t$$

with

$$\bar{f}_i(X_t) = f_i(X_t) - \frac{1}{2}\sum_{j=1}^{d}\sum_{i=1}^{m} g_{jk}(X_t)\frac{\partial g_{ik}}{\partial X_j}(X_t), \qquad i = 1, \ldots, d.$$

for the multivariable case. (Picchini, 2007)

2.2.5 Finite difference method

Although exact methods exist, very few systems describable by Stratonovich or Itô calculus are explicitly solvable. A method commonly used to approximate stochastic differential equations is the so-called finite difference method. The finite difference method approximates continuous time as discrete time. To this end, let's divide the interval $[0, T]$ (where T is the final time) into a number of subintervals N as shown in Figure 2.6.

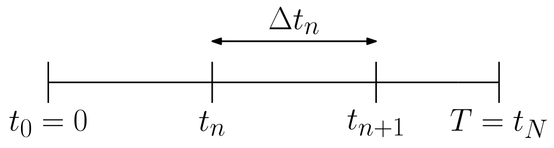


Figure 2.6: Discrete time approximation of continuous time (Duffy & Kienitz, 2009: 108)

We have $N + 1$ mesh points (Duffy & Kienitz, 2009: 107):

$$0 = t_0 < t_1 < \ldots < t_n < t_{n+1} < \ldots < t_N = T$$

In this case, we define a set of subintervals (t_n, t_{n+1}) of size $\Delta t_n \equiv t_{n+1} - t_n,\ 0 \leq n \leq N-1$. We speak of a non-uniform mesh if the sizes of the subintervals are not the same. The most common use case is a uniform mesh, where the N subintervals have the same size, namely $\Delta t = T/N$ (Duffy & Kienitz, 2009: 107). Consider the Itô stochastic differential equation shown in Equation 2.22, with an initial value $X_0 = x_0$. If we approximate the continuous interval $[0, T]$ using a uniform mesh, the continuous function X can be approximated by the discrete function Y. We obtain the recursive relation shown in Equation 2.24 to Equation 2.26 (Duffy & Kienitz, 2009: 108).

$$Y_{n+1} = Y_n + f(Y_n)\Delta t + g(Y_n)\Delta W_n \tag{2.24}$$

$$\Delta W_n = W_{t_{n+1}} - W_{t_n} \tag{2.25}$$

$$Y_0 = x_0 \tag{2.26}$$

This relation is called the Euler-Maruyama integration scheme and is a popular approximation method used for the solution of stochastic differential equations of the Itô calculus description (Duffy & Kienitz, 2009: 108). It is important to note that the Euler-Maruyama method can not directly be applied to the generated problems. This is because the generated models in general can not be described as Itô or Stratonovich differential equation systems. The reasons are as follows:

Noise The system noise is not necessarily described by Wiener processes. Any form of continuous or discrete noise can be used for simulation purposes.

Custom inputs Custom user inputs, such as step and ramp inputs are allowed.

Hybrid system The models generated for the greatest part are a combination of continuous and conditional equations. Conditional equations cause irregular jumps in system values during simulation.

Optimisation Optimal allocation or distribution of flow can form part of the simulation.

However, the finite difference approximation method can provide a good solution to the complex simulation problem. We start with a uniform mesh approximation of the interval $[0, T]$, with $\Delta t = T/N$. Prior to simulation, the stochastic and custom inputs can be generated for every t_n in $[0, T]$. The iterative simulation scheme described in Algorithm 2 can then be implemented.

Algorithm 2 Finite difference approximation scheme

```
for $t_n$ in $[0, T]$ do
    1. Sample - Sample the stochastic and custom inputs.
    2. Root find or optimise - Use the sampled input values to solve the root finding or optimisation problem, depending on which is applicable.
    3. Integrate - After root finding or optimisation, an integration scheme similar to the Euler-Maruyama method could be implemented for integration of the differential equations.
end for
```

2.3 Equation ordering

2.3.1 Background

The simulation problems generated from Amoss models are quite complex and consist of large unordered algebraic equation sets, stemming from the flow sheet model and operating instructions. The main problem to address is reducing the large amount of non-linear algebraic equations to solve simultaneously. Seeing as the number of equations can be in the order of hundreds, equation ordering methods should be investigated.

The most commonly used method of ordering systems of non-linear equations is tearing. Tearing encompasses a broad range of algorithms, with the general goal of ordering equation sets into a computationally efficient and numerically stable form. This entails rewriting equations into a sequentially solvable form and reducing the number of equations to solve simultaneously. (Baharev, Domes & Neumaier, 2017)

2.3.2 Incidence matrices

An incidence matrix is a matrix that indicates the relationship between two sets of objects. Consider a first class X and a second class Y. An incidence matrix representation of X and Y will have one row for each element of X and one column for each element of Y. For every matrix entry, if a relationship exists between the row x and the column y, the entry will have a value of 1. Otherwise, the entry will be 0. (Gross & Yellen, 2005: 97)

Incidence matrices can be used to represent system of equations and are often used in solution of the equation ordering problem. The first class is the equation set, which is indicated by the rows. The second class is the variable set, which is indicated by the columns. If a variable is present in an equation, the entry will contain a 1. Otherwise, the entry will be 0. This is easily illustrated by an example. Consider the following arbitrary unordered system:

$$\frac{\mathrm{d}m}{\mathrm{d}t} - F_i + F_o = 0$$

$$F_i - W(P_u - P) = 0$$

$$F_o - f'L(P - P_a) = 0$$

$$p - \frac{m}{V} = 0$$

$$Re - 1700p = 0$$

$$P - \frac{mRT}{MV} = 0$$

$$\frac{1}{\sqrt{f'}} + 2\log_{10}\left(\frac{\epsilon/D}{3.7} + \frac{2.51}{Re\sqrt{f'}}\right) = 0$$

The system inputs are m, P_u, P_a, ϵ, R, D, W, M, V and L. The variables to solve are $\frac{\mathrm{d}m}{\mathrm{d}t}$, F_i, F_o, P, f', p and Re. There are 7 equations and 7 variables to solve, which indicates that the system is fully specified and solvable. The incidence matrix representation of this system is shown below:

$$\begin{array}{c|ccccccc|}
 & \frac{\mathrm{d}m}{\mathrm{d}t} & F_i & F_o & P & f' & p & Re \\
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
2 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
3 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
4 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
5 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
6 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
7 & 0 & 0 & 0 & 0 & 1 & 0 & 1
\end{array}$$

It should be noted that for Amoss generated models, the algebraic equation sets will necessarily be fully specified and have a square incidence matrix representation. This is because simulation requires a completely specified system.

2.3.3 Bipartite graph

The bipartite graph (or bigraph) finds its origin in graph theory and is defined as a graph whose vertices can be divided into two disjoint sets, U and V, such that every edge E connects a vertex in U to V. The vertex sets U and V are called the parts of the graph. A bipartite graph is often denoted by the notation $G = (U, V, E)$. (Asratian, Denley & Häggkvist, 1998)

Bipartite graphs are a representation form often used to describe systems of equations. The vertex sets, being U and V, can be used to represent the equation set and variable set, respectively. The edges, which are the connections between vertices, indicate an equation-variable relationship and are synonymous with the incidence values of an incidence matrix. A difference which should be noted between bipartite graph and incidence matrices, is that it is possible to indicate directionality in a bipartite graph. An incidence matrix representation can be converted to an undirected bipartite graph representation easily. If we consider the example system in Chapter 2.3.2, we obtain the equivalent bipartite graph representation shown in Figure 2.7.

2.3.4 Identifying unsafe eliminations

For stable simulation, it is important to consider numerical stability in equation ordering or tearing. A significant problem experienced historically in general simulation tools, is zero division. This is due to most tearing methods not taking unsafe pairings of equations and variables into account. Baharev, Schichl & Neumaier (2016b) discuss a solution to this problem, considering possible variable-equation matches.

A definition should be given to safe assignments. If an equation $f_i(x) = 0$ can be solved symbolically for the variable x_j and the solution is unique, explicit and numerically stable, it is considered a safe assignment. Considering these three requirements should be sufficient to identify an unsafe elimination.

Unsafe variable assignments should be noted, but are not necessarily detrimental to simulation. The implicit and non-unique cases simply require that the variable be solved by a root finding method, with the equation as residual. Numerical instability is a more substantial problem and is difficult to address. As an example, consider the equation:

$$x_1 - x_2 x_3 = 0$$

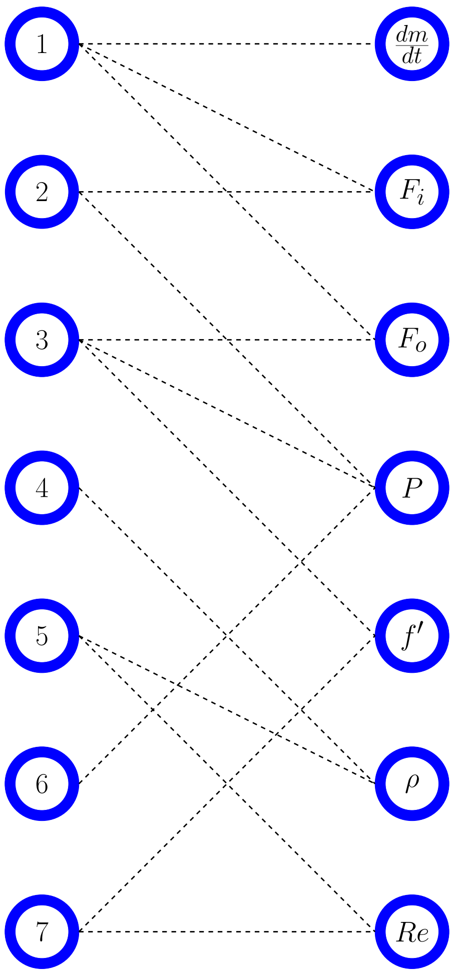


Figure 2.7: Bipartite graph of example system

Making each of the possible variable assignments (x_1, x_2 or x_3) the subject of the equation, three possible solutions are obtained:

$$x_1 = x_2 x_3$$

$$x_2 = \frac{x_1}{x_3}$$

$$x_3 = \frac{x_1}{x_2}$$

If it was known that $-1 \leq x_2 \leq 1$, assigning the equation to x_3 could lead to zero division and numerical instability. Baharev, Schichl, *et al* (2016b) takes the following approach to identifying numerically troublesome eliminations. Firstly, feasible upper and lower bounds for variables are provided. Approximate feasible bounds are sufficient. Making use of interval arithmetic, possible solution ranges are evaluated. If a solution range lies within $r \subseteq [-M, M]$, with an arbitrarily large M, the elimination is considered numerically safe.

The interval arithmetic approach is not perfect, but evaluation of the numerical stability of pairings is necessary. Interval arithmetic tends to overestimate possible ranges. Consequently, this approach can be conservative, especially if conservative variable ranges are provided. A problem that can occur is that a numerically unsafe elimination is the only possible variable assignment. This is especially a problem in sparse systems.

2.3.5 Maximum matching

Maximum matching is an important concept in equation ordering and tearing. To understand maximum matching, preliminary definitions are necessary. Given an undirected graph $G = (V, E)$, a matching is a subset of edges that $M \subseteq E$ such that for all nodes $v \in V$, at most one edge of M is incident on v. A node $v \in V$ is matched by a matching M if some edge in M is incident in v, otherwise, v is unmatched. A perfect matching is a matching in which every node is matched. A maximum matching is a matching of maximum cardinality, which entails that as many as possible nodes are matched. When a bipartite graph is equivalently represented as an incidence matrix, this corresponds to rearranging rows and columns, such that as many as possible non-zero entries are on the diagonal. Maximum matching is often also referred to as maximum transversal or maximum assignment. (Baharev, Schichl & Neumaier, 2016a)

Matching should be put into practical perspective. It is normal in practice, when a small manageable amount of equations are encountered, to try to write equations into a sequentially solvable form where a single equation solves a particular variable. In terms of the equation ordering problem, a maximum matching of equations and variables significantly simplifies obtaining a sequentially solvable problem. Now, considering that the variables and equations sets generated from the combined models initially have no form, starting with variable-equation assignment seems logical.

Maximum matching algorithms have been researched and are implemented by most tools which implement equation ordering methods. It is often the starting point for tearing methods like Dulmage-Mendelsohn decomposition, which is discussed in Section 2.3.7. Maximum matching algorithms mostly follow two methodologies. The first most common implementation is a greedy algorithm, which implements a breadth or depth first search method. This solves the problem in polynomial time. The second, which is used less often, is an optimisation approach. The problem is often referred to as maximum bipartite matching and the fastest general known algorithm to date runs in $O(\sqrt{V}E)$ time (Micali & Vazirani, 1980).

2.3.6 Weighted maximum matching

An alternative to maximum matching discussed in Section 2.3.5 is weighted maximum matching. The problem experienced in traditional maximum matching is that additional information like the suitability of a matching is not taken into account. In our case, it would be ideal to compensate for the numerical stability of a match. Making use of interval arithmetic, numerically safe and unsafe matches can be determined and a weight assigned to each matching. The weighted maximum matching objective would then be to optimise for the most numerically stable matching which assigns a variable to each equation.

The maximum weighted bipartite matching problem, also known as the linear assignment problem, is an optimisation problem to find the maximum weighted matching of a weighted bipartite graph. The first algorithm historically used to solve this problem is the Hungarian algorithm. The original algorithm has a running time of $O(V^4E)$. It uses a modified shortest path search in the augmenting path algorithm. If the Bellman-Ford algorithm is used for this step, the running time of the Hungarian algorithm becomes $O(V^2E)$, or the edge cost can be shifted with a potential to obtain $O(V^2 \log V + VE)$ running time with the Dijkstra algorithm and Fibonacci heap (Fredman & Tarjan, 1987). An alternative algorithm, which has shown better results in practice for the assignment problem, is the so-called Jonker-Volgenant algorithm, with a time running time of $O(V^3E)$ (Jonker & Volgenant, 1987).

2.3.7 Dulmage-Mendelsohn decomposition

Dulmage-Mendelsohn decomposition is a tearing method that is well-established and is often used in tools like OpenModelica and Dymola as part of the equation ordering method (Baharev, Domes, *et al*, 2017). The first step of the Dulmage-Mendelsohn decomposition method is determining a maximum matching or a weighted maximum matching, as an improvement. For the systems generated in Amoss certain things can be assumed. An equal number of equations and variables are necessary for simulation, which implies that all incidence matrices will be square. In addition to this, it is necessary that the system be solvable, which implies that a maximum matching be a perfect matching.

There are two distinct Dulmage-Mendelsohn decomposition methods: coarse decomposition and fine decomposition. Given the input matrix A, the coarse Dulmage-Mendelsohn decomposition yields a row permutation P and a column permutation Q such that

$$PAQ = \begin{bmatrix} A_{11} & & & \\ A_{21} & & & \\ A_{31} & A_{32} & & \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix}, \tag{2.27}$$

where

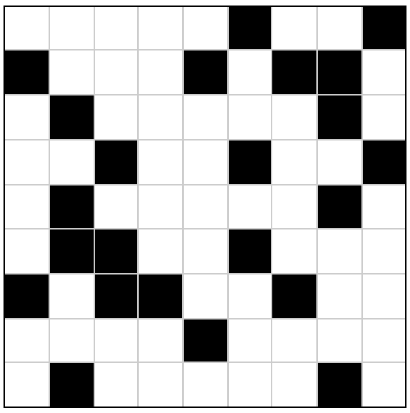
$$\begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix} \tag{2.28}$$

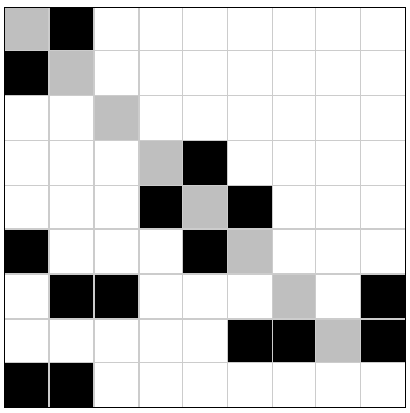
is either absent or it is rectangular and has more rows than columns, similarly

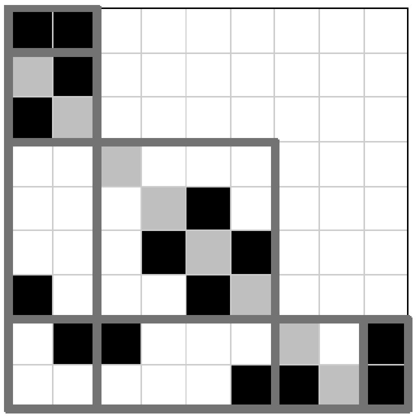
$$\begin{bmatrix} A_{43} & A_{44} \end{bmatrix} \tag{2.29}$$

is either absent or it is rectangular and has more columns than rows. The blocks A_{21}, A_{32} and A_{32} are square with a zero-free diagonal. The matrix 2.28 corresponds to the structurally overdetermined part and the matrix 2.29 to the structurally underdetermined part of the system $Ax = b$. Both matrix 2.28 and matrix 2.29 will be absent if $Ax = b$ is structurally well-defined. (Baharev, Schichl, *et al*, 2016a)

The fine Dulmage-Mendelsohn decomposition yields a row permutation P and a column permutation Q such that A_{21}, A_{32} and A_{43} have possibly smaller irreducible square blocks on the diagonal, with each block having a zero free diagonal. A_{32} also becomes block lower triangular. Irreducible here is meant to suggest that repeating the Dulmage-Mendelsohn decomposition on these blocks would not further decompose them into smaller blocks. An example illustrating how Dulmage-Mendelsohn decomposition works is shown in Figure 2.8. Take note that the zero-free diagonals of the blocks are marked in grey. (Baharev, Schichl, *et al*, 2016a)


(a) Input matrix


(b) Maximum matching


(c) Coarse DM decomposition

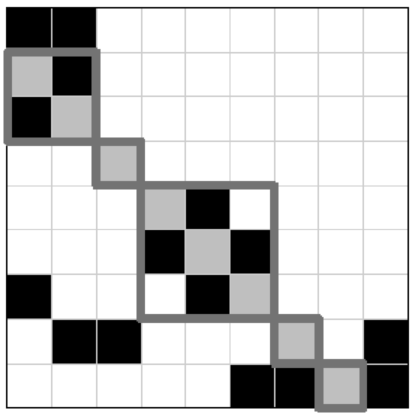

(d) Fine DM decomposition

Figure 2.8: Dulmage-Mendelsohn decomposition example with zero-free diagonals marked in grey (Baharev, Schichl & Neumaier, 2016a)

The primary application of tearing is the simulation of fully specified technical systems (Baharev, Domes, *et al*, 2017). Most mainstream modelling environments will give an error if a system is not fully specified. A special case of the Dulmage-Mendelsohn tearing and the one of primary concern to the Amoss code generation is block lower triangular decomposition or BLT decomposition. If the input matrix is square and structurally non-singular (all submatrices of 2.27 are absent except for A_{32}), a block triangular matrix is obtained as output. The coarse Dulmage-Mendelsohn decomposition leaves A intact, and the fine Dulmage-Mendelsohn decomposition permutes A into a block triangular form, where each diagonal block is square and irreducible, with a zero free diagonal (Baharev, Domes, *et al*, 2017).

It is common practice to perform BLT decomposition first, before implementing a finer tearing technique. This is referred to as partitioning and precedence ordering in literature. The advantage gained from using this method, is that the smaller blocks are computationally easier to handle and tear. In addition to this, considering the number of equations of the Amoss models, it will be greatly advantages to sequentially solve subsystems of equations, instead of a large system of equations. BLT decomposition achieves this, while many tearing methods aim to achieve bordered block triangular form, which has a large single solve area. This can significantly increase solving time, because the number of equations scale as the number of subsystems increase.

However, it should be noted that BLT decomposition followed by tearing can obtain sub-optimal results. An example is shown in Figure 2.9. The variables which need to be guessed for solution of the system are marked in grey. It is clear that the BLT decomposition followed by tearing has 2 unsolved variables, while the optimal tearing only has a 1 unsolved variable. Taking into account the difficult nature of the equation ordering problems faced with the Amoss models, this is a practical trade-off that will be made.

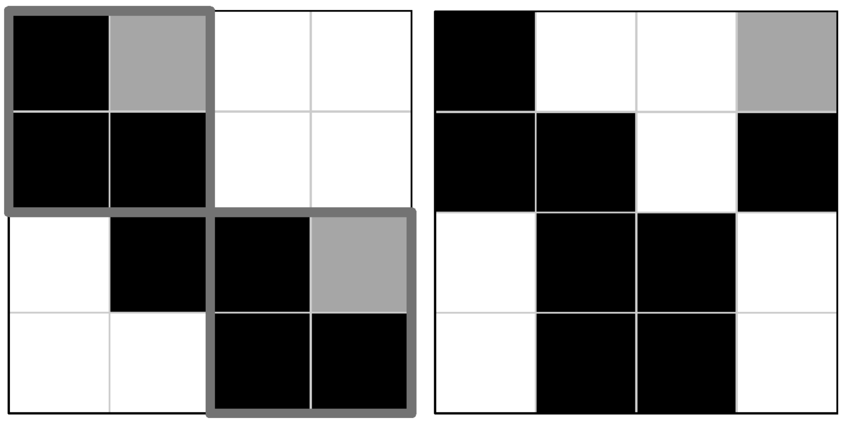

(a) BLT tearing (b) Optimal tearing

Figure 2.9: BLT tearing versus optimal tearing (Baharev, Schichl & Neumaier, 2016a)

2.3.8 Cellier's tearing

After incidence matrix decomposition to block lower triangular form, finer tearing methods should be applied to blocks, to reduce the number of equations to solve. Heuristic tearing methods are commonly used, with Cellier's tearing being a popular choice. The Cellier's tearing method is an essential part of simulation packages like OpenModelica. The traditional Cellier's tearing method contains two sub-routines. The first is maximum matching, where equations and variables are matched, with the Tarjan algorithm being used. The second is variable tearing, where the variables to solve are determined. These routines are called alternately until the system is solved. (Cellier & Kofman, 2006)

The Tarjan algorithm is a graph theory maximum matching algorithm. Its original use was the detection of strongly connected components. Cellier implemented a modified Tarjan algorithm for matching in his algorithm. A bipartite graph representation of the system is required. Two sets of nodes are present, the first representing the equation set and the second representing the variable set. A variable is connected to an equation with an edge if it occurs in the equation. During the matching procedure, matched edges will be coloured red, edges which cannot be matched anymore will be coloured blue and edges not considered will remain black. The matching rules can be summarised as follows (Cellier & Kofman, 2006):

1. For acausal equations with only one black line attached, colour that line red and all other connections ending in that variable blue. Renumber the equation using the lowest number starting from one.

2. For all unknown variables, if the variable only has one line attached to it, colour that line red, follow it back to the equation it points to, and colour all the connections coming from that equation in blue. Renumber the equation using the highest free number starting from n, where n is the number of equations.

Using this colouring technique a full matching of equations to variables is determined. After equation-variable matching using the Tarjan algorithm, heuristic tearing commences. Using the original bigraph (the uncoloured bigraph), a tearing variable should be determined. A tearing variable is the variable that will be solved for. The steps for determining a tearing variable is as follows (Cellier & Kofman, 2006):

1. Determine the equations with the largest number of black lines attached to them.
2. For every one of these equations, follow the black lines, and determine the variables with the most black lines attached to them.
3. For each of these equations determine how many additional equations can be made causal if that variable is assumed known.
4. Choose one of those variables as the next tearing variable that allows the largest number of equations to be made causal.

This process is repeated until a system is made solvable. There are many comments to be made on the original Cellier's tearing method. Starting with the matching algorithm, the Tarjan algorithm has many problems. For instance, difficulties will be encountered if there is an algebraic loop. The algorithm will loop in an effort to obtain a solution endlessly and fail. Secondly, the Cellier's tearing method makes no accommodation for the solvability of the equation-variable matches. Numerical instability, with the possibility of zero division, is possible. Finally, discrete variable matches, which occur in hybrid systems, are not taken into account. (Täuber *et al*, 2014)

The main suggestions to improve the algorithm are as follows. A more robust maximum matching algorithm than the Tarjan algorithm should be used. Many superior algorithms exist, but taking numerical stability into account can be difficult. A possibility is to use weighted maximum matching to accommodate for numerical instability of matches, as discussed in Section 2.3.6. Discrete variables should be taken into account from both a matching and tearing heuristic side. Discrete variable matches should be pre-assigned, with continuous variable maximum matching being done independently. From the tearing side, changing rule 3 of the tearing variable allocation heuristic is recommended to disallow discrete variables as possible tearing variables (Täuber *et al*, 2014).

2.3.9 Desirable forms of incidence matrices

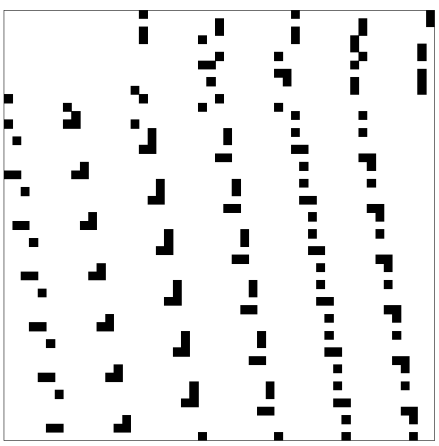
Equation tearing methods have the ultimate goal of obtaining functional forms of incidence matrices. These desirable forms can be used to generate simulations of the represented systems.

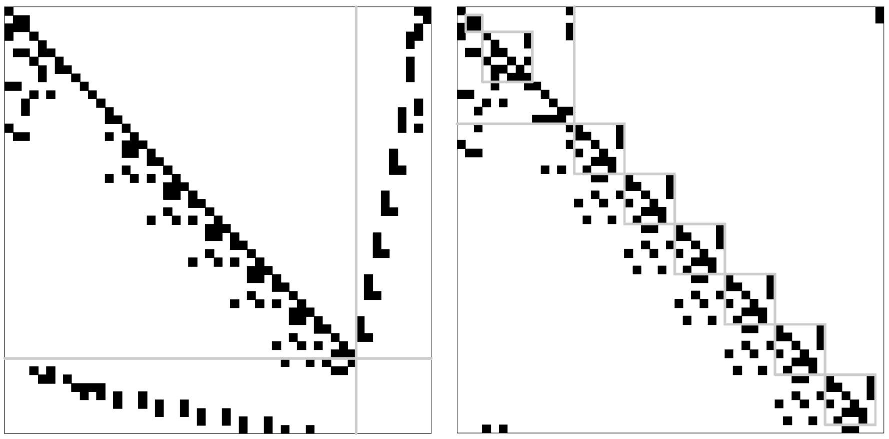
There are two incidence matrix forms commonly used for simulation generation. These are the bordered block lower triangular form and spiked lower triangular form. The bordered block lower triangular form of an input matrix A is described by Equation 2.30. The leading sub-matrix L is a block lower triangular matrix whose diagonal blocks are square and structurally non-singular. (Baharev, Schichl, *et al*, 2016a)

$$A = \begin{bmatrix} L & B \\ C & D \end{bmatrix} \tag{2.30}$$

Spiked lower triangular matrices are nearly lower triangular matrices, where some of the columns have entries above the diagonal. These columns are called spiked columns or simply spikes. The diagonal entry in a non-spike column must be nonzero. Furthermore, for any pair of spike columns, referred to as left and right spikes, the following property must hold: The set of rows in the left spike is either contained in or disjoint from the set of rows of the right spike. This implies that bordered block lower triangular forms can be formed on the diagonal at the spikes where the blocks are either properly nested or disjoint. These diagonal blocks are recursively bordered lower triangular forms. Thus, the spiked lower triangular form can be referred to as a nested bordered lower triangular form. (Baharev, Schichl, *et al*, 2016a)

As an example, consider Figure 2.10. The input matrix can be transformed to the bordered block lower triangular form or the spiked lower triangular form as shown (Baharev, Schichl, *et al*, 2016a). The bordered lower triangular form separates the original problem into two distinctive regions: a sequentially solvable region (the leading sub-matrix L) and a simultaneous solution region (the border). It is the simplest form to use for simulation generation. The spiked lower triangular form breaks the original problem into a series of sub-problems to solve. Each block can be torn to bordered block lower triangular form, each with its own sequentially solvable region and simultaneous solution region. Implementing simulation using this form is more difficult, but faster simulation is possible.


(a) Input matrix



(b) Bordered block lower triangular (c) Spiked lower triangular

Figure 2.10: Desirable incidence matrix forms example (Baharev, Schichl & Neumaier, 2016a)

2.4 Root finding

2.4.1 Newton's method

Solving systems of non-linear equations is an important problem to address regarding the simulations encountered in Amoss. The Newton method, often also called the Newton-Raphson method, and its variants are the most popular solution to the general non-linear root finding problem. Given a system F, which is a function of the vector x, with N variables and N equations, the general root finding problem can be expressed by Equation 2.31. Take note that $F : R^N \to R^N$. F is commonly referred to as the non-linear residual or simply the residual (Kelley, 2003).

$$F(x) = 0 \tag{2.31}$$

An important concept for root finding is the Jacobian matrix $F'(x)$. The Jacobian matrix is a square matrix representation of the gradient of the non-linear residual F with regard to each component of x. If the components of F are differentiable at $x \in R^N$, the Jacobian matrix is defined by Equation 2.32 (Kelley, 2003).

$$F'(x)_{ij} = \frac{\partial (F)_i}{\partial (x)_j}(x) \tag{2.32}$$

The Newton's method is a fixed-point iterative method. The traditional multivariable Newton sequence is shown in Equation 2.33 (Kelley, 2003).

$$x_{n+1} = x_n - F'(x_n)^{-1}F(x_n) \tag{2.33}$$

The local convergence of Newton's method should be determined. The standard assumptions are as follows (Kelley, 2003):

1. Equation 2.33 has a solution x^*
2. $F' : \Omega \to R^{N\times N}$ is Lipschitz continuous near x^*
3. $F'(x^*)$ is non-singular

Lipschitz continuity near x^* requires a $\gamma > 0$ (the Lipschitz constant) such that Equation 2.34 holds for all x, y sufficiently near x^* (Kelley, 2003).

$$\|F'(x) - F'(y)\| \leq \gamma\|x - y\| \tag{2.34}$$

The convergence theorem of Newton's method is stated in Theorem 4 (Kelley, 2003).

Theorem 4. *Let the standard assumptions hold. If x_0 is sufficiently near x^*, then the Newton sequence exists (i.e. $F'(x_n)$ is non-singular for all $n \geq 0$) and converges to x^* and there is a $K > 0$ such that*

$$\|e_{n+1}\| \leq K\|e_n\|^2 \tag{2.35}$$

for n sufficiently large.

Considering the Newton sequence and its convergence, certain things become apparent. From a practical point, it is clear that the Jacobian matrix calculation is critical to both the speed and stability of Newton's method. If the Jacobian matrix can not be calculated easily or the Jacobian is not well behaving, the Newton method will be difficult to implement or diverge if implemented. However, if a fast, stable and well-behaving Jacobian matrix calculation is available, given that the assumptions of Theorem 4 hold, Newton's method should converge rapidly. It is expected that the root finding problems encountered in Amoss simulations will have unique solution, well behaving Jacobian matrices and obey the convergence requirements. The fact that Newton's method exhibits second order convergence, implies that very fast and highly accurate root finding is possible.

2.4.2 Hybrid system solving

The systems encountered in Amoss are for the most part of the mixed continuous/discrete type. These systems, often also called hybrid systems, are difficult to solve with conventional solving methods. If an attempt is made to solve the hybrid system directly using a solver, most solvers will fail. This is mainly due to the fact that most solvers make the assumption of convexity and continuous differentiable gradient. The disjunctions caused by discrete variables have the effect of non-convexity and discontinuous gradients, which are non-differentiable. For this reason alternative approaches need to be investigated.

Modelica is a modelling language where this problem is commonly encountered. Otter, Elmqvist & Mattsson (1999) discuss this problem as approached by the OpenModelica tool. As an example, consider the hybrid system shown in Equation 2.36. For simplicity a linear continuous system has been selected. y are unknown discrete variables of type boolean, integer and/or discrete real. x are unknown continuous variables. A is assumed to be square and regular for all values of y, to ensure a feasible unique solution. f is a function of discrete relations of x, y and of the unknowns y. An example is shown in Equation 2.37. For simplicity, it is also assumed that the equations describing f are sequentially solvable (Otter *et al*, 1999).

$$\begin{gathered} y := f(\text{relation}(x, y), y) \\ A(y)x = b(y) \end{gathered} \tag{2.36}$$

$$\begin{gathered} y_1 := x_1 > x_2 \\ y_2 := y_1 \text{ and } x_3 < 0 \end{gathered} \tag{2.37}$$

Two main approaches of solving this problem were proposed. The first is a simple fixed-point iteration scheme as shown in Algorithm 3. The idea is to do an outer fixed-point iteration for solution of the discrete variables y and with each iteration solve the continuous variables x. If the y values after an iteration matches the previous iteration value, a solution has been found. The advantage of this scheme is its simplicity. This scheme has been shown in Dymola modelling to converge quickly, usually within 3 or 4 iterations. However, convergence is not guaranteed. (Otter *et al*, 1999)

The second solution approach is an exhaustive search scheme as shown in Algorithm 4. Because the discrete variables can only take on a countable number values, all combinations of variables can be tested. The main advantage of this algorithm is that if a solution to the hybrid system exists convergence is guaranteed. The main disadvantage of this scheme is that the number of possible combinations can be huge and that determining the continuous solution for every combination is computationally expensive. For this reason the fixed-point iteration scheme in Algorithm 3 is often preferred. (Otter *et al*, 1999)

Algorithm 3 Fixed point iteration scheme for hybrid systems (Otter, Elmqvist & Mattsson, 1999)

```
x := x (when last event occurred)
y := y (from last event)
**loop**
    last($y$) $:= y$
    $y := f($relation$(x, y), y)$
    if $y ==$ last($y$) then exit;
    $A := A(y)$
    $b := b(y)$
    $\langle$solve $Ax = b$ for $x\rangle$
**end loop**
**return** $x, y$
```

Algorithm 4 Exhaustive search scheme for hybrid systems (Otter, Elmqvist & Mattsson, 1999)

```
x := x (when last event occurred)
y := y (from last event)
**while** $\langle$not all values for relation($x$) tried$\rangle$ **do**
    lastRelation$(x, y) := \langle$next possible value set$\rangle$
    last($y$) $:= y$
    $y := f($lastRelation$(x, y), y)$
    $A := A(y)$
    $b := b(y)$
    $\langle$solve $Ax = b$ for $x\rangle$
    if relation$(x, y) ==$ lastRelation$(x, y)$ then exit;
**end while**
**return** $x, y$
```

2.5 Disjunctive programming

Disjunctive programming problems are commonly encountered in Amoss models. Taking into account that mixed-integer non-linear programming (MINLP) solvers are more established than disjunctive programming solvers, converting the disjunctive programming problems to MINLP problems is a reasonable approach. Generalized disjunctive programming (GDP) is a generalization of disjunctive programming, which is an alternative to the MINLP problem formulation. Generalized disjunctive programming makes use of disjunctions and logic propositions, instead of purely algebraic equations and inequalities as used in MINLP. The generalized disjunctive program is of the form shown in Equation 2.38 (Lee & Grossmann, 2000).

$$\begin{aligned}
&\min Z = \sum_{k\in K} c_k + f(x) \\
&s.t. \quad r(x) \leq 0 \\
&\bigvee_{j\in J_k} \begin{bmatrix} Y_{jk} \\ g_{jk}(x) \leq 0 \\ c_k = \gamma_{jk} \end{bmatrix}, \quad k \in K \\
&\Omega(Y) = True \\
&x \geq 0,\ c_k \geq 0, \quad Y_{jk} \in \{true, false\}
\end{aligned} \tag{2.38}$$

$x \in R^n$ is a vector of the continuous variables. Y_{jk} are boolean variables. $c_k \in R^1$ are continuous variables. γ_{jk} are fixed charges. f is the term for the continuous variables in the objective function. It has a domain $R^n \to R^1$. r are constraints that hold regardless of discrete decisions. It has a domain $R^n \to R^q$. $f(x)$ and $r(x)$ are convex. A disjunction is composed by the $\bigvee$ operator. Each disjunction contains boolean variables Y_{jk}, a set of convex non-linear inequalities $g_{jk}(x)$ and cost variable c_k. $g_{jk}(x)$ has a domain $R^n \to R^1$. If Y_{jk} is true then $g_{jk} \leq 0$ and $c_k = \lambda_{jk}$ are enforced. J_k is an index set of the terms of every disjunction, $J_k = \{j|j = 1, 2, , m_k\}$, $k \in K$. $\Omega(Y) = True$ are the logic terms of the Boolean variables. It is important to note that the problem P, the functions $f(x)$, $r(x)$ and $g_{jk}(x)$ are assumed to be convex and bounded. (Lee & Grossmann, 2000)

The GDP problem shown in Equation 2.38 can be reformulated using big-M reformulation to the MINLP problem shown in Equation 2.39, by replacing the Boolean variables Y_{jk} by binary variables y_{jk} and the big-M constraints. The logic constraints (Y) are converted to linear inequalities. M_{jk} are the big-M parameters that render the inequalities g_{jk} redundant when $y_{jk} = 0$. (HP Williams, 1985)

$$\begin{aligned}
&\min Z = \sum_{k\in K}\sum_{j\in J_k} \gamma_{jk} y_{jk} + f(x) \\
&s.t. \quad r(x) \leq 0 \\
&g_{jk}(x) \leq M_{jk}(1 - y_{jk}), \quad j \in J_k, \quad k \in K \\
&\sum_{j\in J_k} y_{jk} = 1, \quad k \in K \\
&Ay \leq a \\
&x \geq 0, \quad y_{jk} \in \{0,1\}, \quad j \in J_k, \quad k \in K
\end{aligned} \tag{2.39}$$

In Lee & Grossmann (2000) a new approach termed the complex Hull reformulation was introduced. The convex Hull is defined as the tightest possible relaxation of a disjunctive constraint. This is the preferable transformation, considering that the relaxed problem is solved using an MINLP solver. The generalized convex Hull reformulation method is shown in Equation 2.40.

$$\begin{aligned}
&\min Z = \sum_{k\in K}\sum_{j\in J_k} \gamma_{jk} \lambda_{jk} + f(x) \\
&s.t. \quad r(x) \leq 0 \\
&x = \sum_{j\in J_k} \nu^{jk}, \quad k \in K \\
&\sum_{j\in J_k} \lambda_{jk} = 1, \quad k \in K \\
&(\lambda_{jk} + \epsilon) g_{jk}(\nu^{jk}/(\lambda_{jk} + \epsilon)) \leq 0, \quad j \in J_k, \quad k \in K \\
&0 \leq \nu^{jk} \leq \lambda_{jk} U_{jk}, \quad j \in J_k, \quad k \in K \\
&A\lambda \leq a \\
&x, \quad \nu^{jk} \geq 0, \quad \lambda_{jk} \in \{0,1\}, \quad j \in J_k, \quad k \in K
\end{aligned} \tag{2.40}$$

The types of disjunctive constraints encountered in Amoss are simple linear binary disjunctions. Reformulation of these types of constraints is simplified significantly. To illustrate this, we consider the simple example shown in Equation 2.41. This disjunctive constraint creates two separate solutions spaces as shown in Figure 2.11 (Bilodeau, 2015).

$$\begin{bmatrix} Y_1 \\ 0 \leq x_1 \leq 3 \\ 0 \leq x_2 \leq 4 \end{bmatrix} \bigvee \begin{bmatrix} Y_2 \\ 5 \leq x_1 \leq 9 \\ 4 \leq x_2 \leq 6 \end{bmatrix} \tag{2.41}$$

The disjunctive constraint shown in Equation 2.41 can be relaxed to the constraints shown in

Equation 2.42 using big-M formulation (Bilodeau, 2015).

$$\begin{aligned}
y_1 + y_2 &= 1 \\
y_1, y_2 &\in \{0, 1\} \\
-M(1-y_1) + 0 \le x_1 &\le 3 + M(1-y_1) \\
-M(1-y_1) + 0 \le x_2 &\le 4 + M(1-y_1) \\
-M(1-y_2) + 5 \le x_1 &\le 9 + M(1-y_2) \\
-M(1-y_2) + 4 \le x_2 &\le 6 + M(1-y_2)
\end{aligned} \tag{2.42}$$

Alternatively, the disjunctive constraint in Equation 2.41 can be relaxed using convex Hull reformulation to obtain the form shown in Equation 2.43 (Bilodeau, 2015).

$$\begin{aligned}
& y_1 + y_2 = 1 \\
& y_1,\, y_2 \in \{0, 1\} \\
& x_1 = x_{11} + x_{12} \\
& x_2 = x_{21} + x_{22} \\
& 0 \le x_{11} \le My_1 \\
& 0 \le x_{21} \le My_1 \\
& 0 \le x_{12} \le My_2 \\
& 0 \le x_{22} \le My_2 \\
& 0 \le x_{11} \le 3y_1 \\
& 0 \le x_{21} \le 4y_1 \\
& 5y_2 \le x_{12} \le 9y_2 \\
& 4y_2 \le x_{22} \le 6y_2
\end{aligned} \tag{2.43}$$

Although the big-M reformulation and convex Hull reformulation represent the same disjunctive programming problem, the relaxed problem solution spaces differ. To illustrate this, if we arbitrarily select $M = 7$ and use substitution, the big-M reformulation and convex Hull reformulation in Equation 2.44 and Equation 2.45 are obtained, respectively (Bilodeau, 2015).

$$\begin{aligned}
-7(1-y_1) \le x_1 &\le 3 + 7(1-y_1) \\
-7(1-y_1) \le x_2 &\le 4 + 7(1-y_1) \\
-7(1-y_2) + 5 \le x_1 &\le 9 + 7(1-y_2) \\
-7(1-y_2) + 4 \le x_2 &\le 6 + 7(1-y_2)
\end{aligned} \tag{2.44}$$

$$\begin{aligned}
0 &\leq x_{11} \leq 7y_1 \\
0 &\leq x_{21} \leq 7y_1 \\
0 &\leq x_{12} \leq 7y_2 \\
0 &\leq x_{22} \leq 7y_2 \\
0 &\leq x_{11} \leq 3y_1 \\
0 &\leq x_{21} \leq 4y_1 \\
5y_2 &\leq x_{12} \leq 9y_2 \\
4y_2 &\leq x_{22} \leq 6y_2
\end{aligned} \tag{2.45}$$

The solution spaces of the big-M reformulation and convex Hull reformulation are shown in Figure 2.12 and Figure 2.13, respectively. It is clear that the big-M reformulation problem has a larger solution space. The larger solution space leads to an increase in solving difficulty and consequently solving time for the big-M reformulation problem, using an MINLP solver (Bilodeau, 2015).

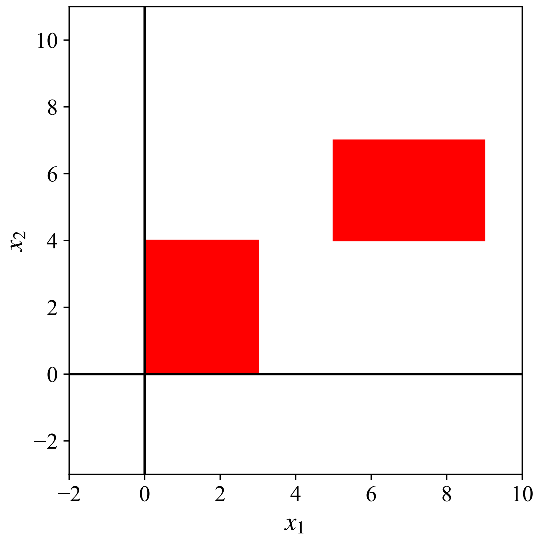


Figure 2.11: Example disjunctive solution space (Bilodeau, 2015)

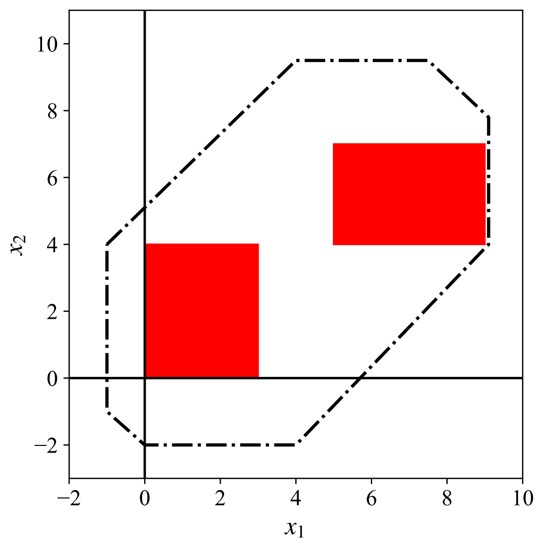


Figure 2.12: Example disjunctive big-M relaxed solution space (Bilodeau, 2015)

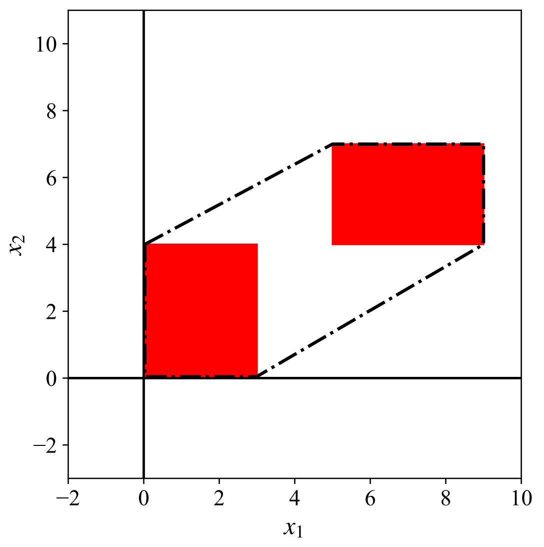


Figure 2.13: Example disjunctive convex Hull relaxed solution space (Bilodeau, 2015)

2.6 Numerical integration

A first order ordinary differential equation (ODE) is commonly of the form shown in Equation 2.46 (Bradie, 2006). Higher order ODE systems are not considered separately. This is because conversion of higher order systems to first order systems can simply done by introducing dummy variables and sequential substitution. Take for example Equation 2.47. If we introduce the dummy variable $x(t)$ as in Equation 2.48, we can rewrite Equation 2.47 as Equation 2.49 and Equation 2.50.

$$y'(t) = f(t, y(t)) \tag{2.46}$$

$$y''(t) = \lambda y(t) \tag{2.47}$$

$$x(t) = y'(t) \tag{2.48}$$

$$x'(t) = \lambda y(t) \tag{2.49}$$

$$y'(t) = \lambda x(t) \tag{2.50}$$

The initial value problem for ODEs is of particular importance. Given an ODE of the form in Equation 2.46, with an initial value $y(t_0) = y_0$, the problem is to solve $y(t)$ for $t > t_0$. Few ODE systems have a symbolic or closed-form solution. For this reason, numerical integration methods are often used for initial value problems regarding ODEs. Various numerical integration methods exist, including implicit, explicit, one-step and multi-step methods. The most often used integration methods are the family of explicit Runge-Kutta methods. Methods include, in increasing order of accuracy, the explicit Euler method, the explicit trapezoidal method and the RK4 or classical Runge-Kutta method. (LeVeque, 1998: 93–102)

Solution of initial value problems regarding ODEs is necessary for simulation of Amoss models. The generated ODE systems are stochastic in nature, consist of hybrid continuous/conditional equations and could contain intermediate optimisation. The nature of the generated models breaks the smoothness assumptions of higher order integration methods. Due to this, only the explicit Euler method, which is a first order method, will be considered. The Euler method is shown in Equation 2.51 (Süli & Mayers, 2003: 317).

$$y_{n+1} = y_n + hf(t_n, y_n) \tag{2.51}$$

There are two types of errors which are used as measures of the accuracy of numerical integration: local truncation error and global truncation error. Local truncation error is the error made by a single iteration. The global truncation error is the error made by multiple iterations or in other words, the cumulative sum of the local truncation error (Süli & Mayers, 2003: 317). Order of accuracy, which is based on the global truncation error, is the rate of convergence of a numerical approximation of a differential equation to the actual solution. A numerical method is said to be n^{th} order accurate if its global truncation error is proportional to step size h to the n^{th} power (LeVeque, 1998: 3–5). The order of accuracy of the Euler method is 1 (LeVeque, 1998: 149).

Another factor to investigate is stability. The first type of stability to discuss is that of zero-stability. The Euler method is zero-stable. A zero-stable method is a method where the error (both global and local) tends to 0 as the step size h tends to 0. Often this limit cannot be computed and this stability definition proves impractical. This is because evaluation of very small h values becomes very computationally expensive. In practice, an h value which is as large as possible while still maintaining the accuracy requirements should be selected. A more meaningful stability measure of a numerical integration method is absolute stability. (LeVeque, 1998: 149).

Consider the test problem shown in Equation 2.52. When the Euler method is selected, the iterative solution in Equation 2.53 is obtained. The method is absolutely stable if $|1 + h\lambda| \leq 1$. Otherwise, it is unstable. Although there are two parameters (h and λ), the product of the two ($z = h\lambda$) is of importance. The Euler method is stable for $-2 \leq z \leq 0$ and thus the interval of absolute stability is [-2, 0]. The region of absolute stability is more commonly used. This is the interval of absolute stability in the complex z plane. It is illustrated graphically with the real component of z on the x-axis and the imaginary component on the y-axis. This allows the possibility that λ values be imaginary, which is not relevant to the ODE systems solved in Amoss. (LeVeque, 1998: 151–152)

$$y'(t) = \lambda y(t) \tag{2.52}$$

$$y_{n+1} = (1 + h\lambda)y_n \tag{2.53}$$

Euler's integration method is simple, with a small stability region and a low order of accuracy. The overall simulation approach makes use of finite difference approximation and frequent re-evaluation of the entire model. Due to this, the Euler method should be sufficient. The use of small time-steps will ensure absolute stability and sufficient integration accuracy.

2.7 Automatic or algorithmic differentiation

When dealing with code that evaluates numerical values of certain functions, it is often important to determine accurate values for the derivatives as well. This is most important for our root finding method. Taking into account that the Newton method requires calculation of a Jacobian matrix for each iteration, using a reliable method for derivative information is essential. The method that is most often used for reliably determining gradients in modern computing is algorithmic or automatic differentiation.

Automatic differentiation, also referred to as algorithmic differentiation can be seen as a set of techniques used to evaluate the derivative of a computer program. It is superior to alternative gradient calculation methods in terms of accuracy and more significantly efficiency. Automatic differentiation exploits the fact that any computer program is executed by sequentially doing basic operations (addition, subtraction, multiplication, division, etc.) and elementary operations (exp, log, sin, cos, etc.). Making use of the chain rule, derivatives of any order can be calculated to working precision, almost as fast as the original function evaluation is done. (Griewank & Walther, 2008)

It is important to distinguish automatic differentiation from both numerical differentiation (method of finite differentiation) and symbolic differentiation. Numerical differentiation is a derivative approximation method that uses function evaluations to approximate the gradient over a finite step or difference. Two commonly used gradient approximation formulas are shown in Equation 2.54 and Equation 2.55. Symbolic differentiation entails analytically deriving the derivative function, directly by substitution, and numerically evaluating the derived function. (Griewank & Walther, 2008)

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \tag{2.54}$$

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} \tag{2.55}$$

Numerical differentiation is often used, but has certain shortcomings. From Equation 2.54 and Equation 2.55, it is clear that if h is small cancellation error reduces the accuracy of the approximation. If h is not small then truncation error reduces the accuracy of the approximation. This effect is even more severe for higher order derivatives. Even if h is optimally selected, the derivative accuracy can be a problem. Symbolic differentiation suffers from two main problems. The first is that determining an analytical derivative function is not always possible or is at least difficult. Even in the case that obtaining an analytical solution is possible, it can occur that the given function cannot be differentiated symbolically. The second is that the analytical function obtained can often be complex and computationally expensive to evaluate. (Griewank & Walther, 2008)

Algorithmic differentiation comes in many variants, with the classical forward and reverse modes being the most common. The forward mode traverses an evaluation trace in a forward direction. An input variable is chosen, with each derivative then being calculated with regard to the independent variable making use of the chain rule. An alternative to this is traversing an evaluation trace in the reverse direction. This is referred to as reverse mode. The output variable is selected and its sensitivity with regard to each input is determined using the chain rule. (Griewank & Walther, 2008)

Algorithmic differentiation is best explained by an example. Consider the function $y = f(x_1, x_2)$ shown in Equation 2.56. The goal is to determine the derivative of y with respect to x_1 for $x_1 = 1.5$ and $x_2 = 0.5$. Evaluating y, the evaluation trace shown in Table 2.1 is obtained. The evaluation trace can be represented by the computational graph shown in Figure 2.14. The notation $\dot{\nu}_i = \partial \nu_i / \partial x_1$ is introduced to simplify the forward mode derivative calculation. Calculating the derivative using the forward mode, we obtain the forward-derivative evaluation trace shown in Table 2.2. The notation $\bar{\nu}_i = \partial y / \partial v_i$ is introduced to simplify the backward mode derivative calculation. Calculating the derivative using the backward mode, we obtain the backward-derivative evaluation trace shown in Table 2.3. (Griewank & Walther, 2008)

$$y = [\sin(x_1/x_2) + x_1/x_2 - \exp(x_2)] \cdot [x_1/x_2 - \exp(x_2)] \tag{2.56}$$

$$\begin{tikzcd}
\nu_{-1} \arrow[rr] & & \nu_1 \arrow[rr] \arrow[drrr] & & \nu_2 \arrow[rr] & & \nu_5 \arrow[dr] & \\
& \nu_0 \arrow[ur] \arrow[rr] & & \nu_3 \arrow[rr] & & \nu_4 \arrow[ur] \arrow[rr] & & \nu_6
\end{tikzcd}$$

Figure 2.14: Computational graph of example evaluation trace (Griewank & Walther, 2008)

Table 2.1: Example evaluation trace (Griewank & Walther, 2008)

ν_{-1}	=	x_1	=	1.5000		
ν_0	=	x_2	=	0.5000		
ν_1	=	ν_{-1}/ν_0	=	$1.5000/0.5000$	=	3.0000
ν_2	=	$\sin(\nu_1)$	=	$\sin(3.0000)$	=	0.1411
ν_3	=	$\exp(\nu_0)$	=	$\exp(0.5000)$	=	1.6487
ν_4	=	$\nu_1 - \nu_3$	=	$3.0000 - 1.6487$	=	1.3513
ν_5	=	$\nu_2 - \nu_4$	=	$0.1411 + 1.3513$	=	1.4924
ν_6	=	$\nu_5 \cdot \nu_4$	=	$1.4924 \cdot 1.3513$	=	2.0167
y	=	ν_6	=	2.0167		

Table 2.2: Example forward-derivative evaluation trace (Griewank & Walther, 2008)

ν_{-1}	=	x_1	=	1.5000		
$\dot{\nu}_{-1}$	=	$\dot{x}_1$	=	1.0000		
ν_0	=	x_2	=	0.5000		
$\dot{\nu}_0$	=	$\dot{x}_2$	=	0.0000		
ν_1	=	ν_{-1}/ν_0	=	$1.5000/0.5000$	=	3.0000
$\dot{\nu}_1$	=	$(\nu_{-1} - \nu_1 \cdot \dot{\nu}_0/\nu_0)$	=	$1.0000/0.5000$	=	2.0000
ν_2	=	$\sin(\nu_1)$	=	$\sin(3.0000)$	=	0.1411
$\dot{\nu}_2$	=	$\cos(\nu_1) \cdot \dot{\nu}_1$	=	$-0.9900 \cdot 2.0000$	=	-1.9800
ν_3	=	$\exp(\nu_0)$	=	$\exp(0.5000)$	=	1.6487
$\dot{\nu}_3$	=	$\nu_3 \cdot \dot{\nu}_0$	=	$1.6487 \cdot 0.0000$	=	0.0000
ν_4	=	$\nu_1 - \nu_3$	=	$3.0000 - 1.6487$	=	1.3513
$\dot{\nu}_4$	=	$\dot{\nu}_1 - \dot{\nu}_3$	=	$2.0000 - 0.0000$	=	2.0000
ν_5	=	$\nu_2 - \nu_4$	=	$0.1411 + 1.3513$	=	1.4924
$\dot{\nu}_5$	=	$\dot{\nu}_2 - \dot{\nu}_4$	=	$-1.9800 + 2.0000$	=	0.0200
ν_6	=	$\nu_5 \cdot \nu_4$	=	$1.4924 \cdot 1.3513$	=	2.0167
$\dot{\nu}_6$	=	$\dot{\nu}_5 \cdot \nu_4 + \nu_5 \cdot \dot{\nu}_4$	=	$0.0200 \cdot 1.3513 + 1.4924 \cdot 2.0000$	=	3.0118
y	=	ν_6	=	2.0167		
$\dot{y}$	=	$\dot{\nu}_6$	=	3.0118		

Table 2.3: Example backward-derivative evaluation trace (Griewank & Walther, 2008)

ν_{-1}	=	x_1	=	1.5000		
ν_0	=	x_2	=	0.5000		
ν_1	=	ν_{-1}/ν_0	=	$1.5000/0.5000$	=	3.0000
ν_2	=	$\sin(\nu_1)$	=	$\sin(3.0000)$	=	0.1411
ν_3	=	$\exp(\nu_0)$	=	$\exp(0.5000)$	=	1.6487
ν_4	=	$\nu_1 - \nu_3$	=	$3.0000 - 1.6487$	=	1.3513
ν_5	=	$\nu_2 - \nu_4$	=	$0.1411 + 1.3513$	=	1.4924
ν_6	=	$\nu_5 \cdot \nu_4$	=	$1.4924 \cdot 1.3513$	=	2.0167
y	=	ν_6	=	2.0167		
$\bar{\nu}_6$	=	$\bar{y}$	=	1.0000		
$\bar{\nu}_5$	=	$\bar{\nu}_6 \cdot \nu_4$	=	$1.0000 \cdot 1.3513$	=	1.3513
$\bar{\nu}_4$	=	$\bar{\nu}_6 \cdot \nu_5$	=	$1.0000 \cdot 1.4924$	=	1.4924
$\bar{\nu}_4$	=	$\bar{\nu}_4 + \bar{\nu}_5$	=	$1.4924 + 1.3513$	=	2.8437
$\bar{\nu}_2$	=	$\bar{\nu}_5$	=	1.3513		
$\bar{\nu}_3$	=	$-\bar{\nu}_4$	=	-2.8437		
$\bar{\nu}_1$	=	$\bar{\nu}_4$	=	2.8437		
$\bar{\nu}_0$	=	$\bar{\nu}_3 \cdot \nu_3$	=	$-2.8437 \cdot 1.6487$	=	-4.6884
$\bar{\nu}_1$	=	$\bar{\nu}_1 + \cos(\nu_1)$	=	$2.8437 + 1.3513 \cdot (0.9900)$	=	1.5059
$\bar{\nu}_0$	=	$\bar{\nu}_0 - \bar{\nu}_1 \cdot \nu_1/\nu_0$	=	$-4.6884 - 1.5059 \cdot 3.0000/0.5000$	=	-13.7239
$\bar{\nu}_{-1}$	=	$\bar{\nu}_1/\nu_0$	=	$1.5059/0.5000$	=	3.0118
$\bar{x}_2$	=	$\bar{\nu}_0$	=	-13.7239		
$\bar{x}_1$	=	$\bar{\nu}_{-1}$	=	3.0118		

2.8 Parallel processing

Parallel processing or computing entails executing many calculations or processes simultaneously (Gottlieb & Almasi, 1989). The idea is to reduce a large problem into smaller sub-problems, which can be executed on multiple processors at the same time and thus be executed significantly faster. Parallelism has become core to high-performance computing in recent times. This is due to most modern computers having multi-core processors (Asanovic *et al*, 2006).

Traditionally computers only had single processors to perform the actions designated by a particular program. Computers with multiple processors or multiprocessor computers have become a norm in recent times. If a process can be divided into at least as many sub-processes as there are processors (p), a given process can be processed in $1/p^{th}$ of the time. This can lead to great performance increases. In practice this is rarely obtained, because practical factors like dividing a problem into perfect sub-problems and even writing to disk pose practical limitations. (Allen & Wilkinson, 2005)

The ideal case of parallel processing is when processes can be divided into entirely independent parts and executed simultaneously. This is referred to as embarrassingly parallel or naturally parallel processes. These types of problems require no advanced techniques for parallelisation. Ideally, there should be no communication between these types of processes. Figure 2.15 illustrates the disconnected computational graph of embarrassingly parallel processes (Allen & Wilkinson, 2005). Monte Carlo simulations are well-known naturally parallel processes, due to the independent nature of the individual replications. This suggests that Amoss simulations could benefit greatly from parallel processing.

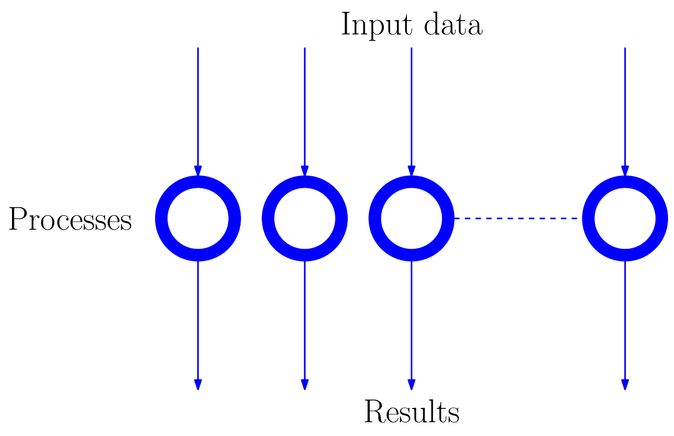


Figure 2.15: Disconnected computational graph of embarrassingly parallel processes (Allen & Wilkinson, 2005)

2.9 Result writing

Result writing is an essential part of simulation. Slow result writing can easily become a simulation bottleneck, especially considering the large amount of data generated during Amoss Monte Carlo simulations. For result writing, it is important to consider the file format used and compression methods if available.

2.9.1 CSV

Comma-separated values (CSV) files are simple text files using commas as delimiter between values. Data is stored in plain text, with each line representing a record. The CSV format was one of the first widely used data storage formats. However, data storage is memory inefficient, especially for large data sets. Due to the representation method, internal compression is not possible (Shafranovich, 2005). Moss simulation results were initially stored as CSV files.

2.9.2 HDF5

The hierarchical data formats (HDF) are a set of data formats, specifically designed for storage of large data sets. It was originally developed by the National Center for Supercomputing Applications and is currently supported by the HDF group. HDF5 is widely used and has mature support. Compression methods are well supported. (The HDF Group, 2019)

2.9.3 Parquet

Parquet is a fast and well established columnar file format. The initial development of Parquet was a joint-effort between Twitter and Cloudera. It is currently a top-level Apache Software Foundation (ASF)-sponsored project. Parquet is commonly used for storage of large data sets. Compression is well supported. (Apache Software Foundation, 2019)

2.9.4 Feather

Feather is a lightweight fast to read and write file format. It makes use of the Apache Arrow columnar memory specification and is maintained by the Apache Arrow community. The file format is not yet mature, so long term data storage is not recommended due to instability. Compression as a feature is not yet available. (Wickham, 2016)

2.9.5 Snappy

Compression methods can be used to reduce file size when writing to disk. This can improve write speed and aid with storage space of large amounts of data. During simulation, file writing occurs between replications, so storing results quickly is important. Writing time is more important than high compression ratios. For this reason, Snappy compression is ideal. Snappy (previously known as Zippy) is a fast data compression and decompression library written in C++ by Google. It does not aim for maximum compression, but rather high compression speeds and reasonable compression (Google, 2019).

2.10 Amoss 1.0

Amoss 1.0 was developed by Edgar Whyte. It was developed as a generic equation-oriented stochastic simulation platform to address the problems associated with the Moss methodology. An automated method of generating and simulating Moss models was introduced. Model generation was done using flow sheet model and user input operating instructions. It proved successful as a proof of concept, but the generated simulations were numerically unstable and slow. (Whyte, 2018)

The primary goal was to generalize Moss simulation generation. The Moss methodology provided a sequential modular approach to generate stochastic simulations by hand. Simulations were developed in this manner by Sasol's operations research team in the VBA language. However, creating simulations took a considerable amount of time and simulations had to be rebuilt every time a plant change occurred. Additionally, complex plant recycles could not be accommodated in a simple way. (Whyte, 2018)

2.10.1 Equation ordering

A significant improvement of Amoss 1.0 was how the flow sheet model and operating instructions were handled. Equations were generated, augmented and ordered into a solvable form for simulation using equation ordering methods. If a plant change was made, the flow sheet model would simply have to be changed. Thus, there was a shift from a sequential modular to equation-orientated simulation approach. (Whyte, 2018)

Zero division was expected to be a problem if an inadequate ordering was obtained. Mass flows and tank levels could viably be zero. Due to this, safe eliminations, as discussed in Section 2.3.4, had to be considered. The `safe-elimination` repository was developed upon request by Baharev (2017a) to determine unsafe eliminations. The `sdopt-tearing` repository, also developed by Baharev (2017b), was used for equation ordering. Using the `hessenberg` function in `rpc_api.py`, the bordered block lower triangular form of incidence matrices could be obtained, taking into account unsafe eliminations. The ordering method proved functional, but various problems were experienced. The problems experienced were as follows (Whyte, 2018):

Many equations to simultaneously solve Ordering to bordered block lower triangular form led to many equations having to be solved simultaneously. This slowed down the root finding method and consequently simulations. This is a result of the conservative nature of the algorithm, which adds any unsafe variable matches to the variables to simultaneously solve.

Numerical instability Although unsafe eliminations were considered, the obtained orderings often led to numerically unstable simulations.

External dependency The repository was not installable as a Python package. Because of this, the repository as a whole had to exist in the code base.

The equation ordering results could be categorized into three groups: explicit ordering, implicit ordering and optimisation problem. An explicit ordering corresponds to a sequentially solvable ordering with no equations to simultaneously solve. An implicit ordering corresponds to an ordering where equations need to be simultaneously solved. If an optimisation goal formed part of the model, the ordering would be marked as an optimisation problem instead of a simulation problem.

2.10.2 Simulation approach

The main focus of Amoss 1.0 was to generate Monte Carlo simulations of Moss models. The augmented models were found to be difficult to simulate for various reasons. The steps used to obtain a model are as follows (Whyte, 2018):

1. Translate real plant to a flow sheet model.
2. Gather statistical data of plant units with statistical variance and fit probability density function.
3. Determine operating rules, which contain protocol followed in event of failures and normal operating procedures.

4. Set up optimisation problems like optimal flow allocation.

Augmenting the equations generated from each of the above steps, a highly complex model could be generated. The simulation elements contained in the model can be summarised as follows:

First order ordinary differential equations buffer tank levels

Continuous equations flow sheet model and operating instructions

Conditional equations if-blocks in the operating instructions

Stochastic elements unit efficiency, weather effects, plant disturbances, mass flow losses etc.

Disjoint constraint optimisation optimal distribution of mass flow to reactors

In addition to the complexity of the combined model, the number of equations could be in the order of hundreds. These large combined problems are difficult to solve and have been the topic of research for almost a decade. Whyte (2018) attempted to solve the problem, using a finite difference approach, which has been the main solution.

Algorithm

The simulation approach to the implicitly ordered problem is shown in Algorithm 5. The first step is sampling of the stochastic variable. The second step entails solving the simultaneous solution region of the combined continuous and conditional equation set in a single root finding step. The third step entails evaluating the remaining continuous and conditional equations. The fourth and final step is where Euler integration is applied to the first order differential equations.

The simulation approach to the explicitly ordered and optimisation problems are similar to the implicitly ordered problem. The explicitly ordered problem approach is shown in Algorithm 6. There is no root finding step, since the system is sequentially solvable. The optimisation problem simulation approach is shown in Algorithm 7. It should be noted that the root finding step is replaced with an optimisation step. The residual equations of the root finding problem become returns to the disjunctive optimisation problem.

The simulation approaches had varying success. Simulations stemming from explicit orderings proved successful, but the implicit ordering and optimisation problem simulations were problematic. Implicit ordering problems often led to numerically unstable and slow simulations. Simulations of the optimisation problem often led to counter-intuitive results.

Algorithm 5 Whyte (2018) implicit ordering simulation approach

```
for t in time_span do
    stochastic_var = sample_stochastic(t)
    solved_var = RootFind(stochastic_var, integrate_var)
    combined_var, integrate_f = Evaluate(stochastic_var, integrate_var, solved_var)
    integrate_var = integrate_var + integrate_f$\Delta t$
end for
```

Algorithm 6 Whyte (2018) explicit ordering simulation approach

```
for t in time_span do
    stochastic_var = sample_stochastic(t)
    combined_var, integrate_f = Evaluate(stochastic_var, integrate_var)
    integrate_var = integrate_var + integrate_f$\Delta t$
end for
```

Algorithm 7 Whyte (2018) optimisation problem simulation approach

```
for t in time_span do
    stochastic_var = sample_stochastic(t)
    solved_var = Optimise(stochastic_var, integrate_var)
    combined_var, integrate_f = Evaluate(stochastic_var, integrate_var, solved_var)
    integrate_var = integrate_var + integrate_f$\Delta t$
end for
```

Optimal flow allocation

One of the most difficult features to incorporate in the generated simulations was optimal flow allocation. It is often possible that mass flow can be distributed through multiple pipes. An operating instruction specifying optimal mass flow distribution can be specified. The problem is made difficult by the fact that some lines require a minimum flow or are constrained by a maximum flow. The optimal flow allocation problem can be described by Equation 2.57. An objective function should be selected, with an economic objective being a common choice. It was decided that the user would assign economic priority to each of the streams. An objective function of the form in Equation 2.58, with a decaying factor d, was selected for minimization. (Whyte, 2018)

$$\begin{aligned} \min_{x} \quad & f(x) \\ \text{s.t.} \quad & \sum x \leq x_{available} \\ & h(x) \leq h_{max} \\ & g(x) = 0 \text{ or } g(x) \geq g_{min} \end{aligned} \tag{2.57}$$

$$f(x) = -d^0 x_0 - d^1 x_1 - d^2 x_2 - \ldots - d^n x_n \tag{2.58}$$

A special case of the general optimisation is possible when the minimum and maximum constraints are local to the point of distribution. This significantly simplifies the optimisation problem, which can be reduced to a series of if-statements with priority allocation. Work was done on this matter for the case where no plant recycles are involved. The `allocate` function was developed for use in the operating instructions, to generate simulations with priority mass flow allocation. The `allocate` function has the following inputs (Whyte, 2018):

available (type: float) mass flow available to distribute

minimum_priority (type: list of ints) priority order in which minimum constraints should be satisfied

minimum_constraints (type: list of floats) minimum constraints of allocations

maximum_priority (type: list of ints) priority order in which maximum constraints should be satisfied

maximum_constraints (type: list of floats) maximum constraints of allocations

For simulation, the `allocate` operating instruction is converted to a set of equations using the algorithm illustrated in Figure A.1. The special case of the optimal flow allocation problem in the forward direction simulated with accepted results. However, two main problems were encountered regarding optimal flow allocation. Firstly, problems were experienced when implementing the special case when a dependency existed between the inputs and the outputs, i.e a plant recycle was encountered. Secondly, simulation of the general case of the optimal flow allocation generated counter-intuitive results.

The disjunctive programming problem describing the general case of optimal flow allocation could be reformulated as an MINLP, using big-M or convex Hull reformulation. It was decided that reformulation and MINLP solving would be too difficult to implement. Equation 2.57 was reduced to Equation 2.59 for simplicity. The maximum constraints ($h(x)$) were removed and the disjunctive minimum constraints ($g(x)$) were reduced to binary constraints that could only take on values of zero or g_{min} (Whyte, 2018). This modelling decision should be criticised. Equation 2.59 is a poor approximation of the disjunctive programming problem and incorrect or counter-intuitive results were to be expected.

$$\begin{aligned} \min_{x} \quad & f(x) \\ \text{s.t.} \quad & \sum x \leq x_{available} \\ & g(x) = \{0; g_{min}\} \end{aligned} \tag{2.59}$$

2.10.3 Code generation

Amoss 1.0 implemented a robust code generation method, although the generated code was not as efficient as possible. The simulation code was set up to do Monte Carlo simulation of each scenario. Simulation code was generated as a non-executable `.py` Python file and imported to the greater Python code base for simulation (Whyte, 2018). This design choice is non-ideal, because effective parallel processing requires simulation runs to be independent. Because a large part of the code base is shared, a dependence will exist among various runs.

Parallel processing was implemented, with simulation of the various scenarios being parallelised. The `Celery` package was used for parallel processing (Whyte, 2018). The decision to parallelise scenarios and not the replications of scenarios limited the potential effect of parallel processing. After simulation, the results were written to HDF5 with compression. The HDF5 files were substantially smaller than the CSV files previously generated. This led to lower write times (Whyte, 2018).

Amoss 1.0 code generation also made use of the `CasADi` package, with great success. `CasADi` is an automatic differentiation framework developed in C++. It was mainly used for simple Newton root finding during simulation. The simulation time was significantly reduced by its use. Additionally, the `CasADi` interface to `IPOPT` was used for optimisation of the optimal flow allocation problem. (Whyte, 2018)

Part II

Implementation

CHAPTER 3

TESTS, BENCHMARKS AND VALIDATION

When the deliverables to address in Amoss 2.0 are considered, as discussed in Section 1.4, it becomes clear that tests, benchmarks and validation should be implemented. Tests, which are run upon any code changes, should be in place to ensure the integrity of the code base as development commences. *Fast simulation time*, *reduction in development time* and *linear scalability* are outcomes which can objectively be measured and quantified. Benchmarks could be used to determine the performance relative to these deliverables. *Acceptable accuracy* as a goal could be evaluated using result validation.

3.1 Hardware and software

All tests and benchmarks will be run on a machine with the hardware and software specifications shown in Table 3.1.

Table 3.1: Hardware and software specifications

Component	Specification
Processor	Intel Core i7-2800 CPU @ 2.40 GHz
Disk	Seagate SATA HDD 160 GB
Operating system	Windows 10 Professional @ 2018 Microsoft Corporation
RAM	32 GB DDR3
Python version	3.7.4

3.2 Test processes

The Amoss tool should be able to accommodate various simulation features. For this reason, test processes with a gradual increase in difficulty and simulation features were created. These test processes will be tested for code generation and simulation upon any code changes. This approach is advantageous for various reasons. Firstly, the integrity of the code base is ensured. Secondly, if a code change breaks a simulation feature, this feature can be determined and the code change altered. Additionally, the test processes can be used for benchmarking, as discussed in Section 3.4. The simulation features which are gradually introduced are:

- The number of *components* in each stream to simulate.
- The number of *recycles* in the flow diagram.
- The number of *buffer tank* units in the flow diagram.
- The number of *separator* units in the flow diagram.
- The number of *cracker* units in the flow diagram.
- The number of *plant* units in the flow diagram.
- The number of *stochastic inputs* in the simulation.
- Whether the simulation contains a *forward heuristic allocation problem.*
- Whether the simulation contains a *heuristic allocation problem* with a plant *recycle*.

- Whether the simulation contains an *optimal allocation problem*.

The number of *equations* of the generated models can be used as a rough indicator of the overall test process complexity. The simulation features of the test processes are summarized in Table 3.2.

Table 3.2: Test process simulation features

Feature	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Components	1	1	5	1	1	1	1	5	5	5	5	5	5	1
Recycles	0	0	0	1	2	3	4	1	2	4	4	1	1	0
Buffer tanks	0	1	1	1	1	1	1	1	1	1	1	1	1	0
Separators	1	1	1	1	1	1	1	1	1	1	1	1	1	0
Crackers	1	1	1	1	1	1	1	1	1	1	1	1	1	0
Plants	1	1	1	1	1	1	1	1	1	1	1	1	0	0
Stochastic inputs	0	0	0	0	0	0	0	0	0	2	3	0	0	0
Heuristic allocation (Forward)	0	0	0	0	0	0	0	0	0	0	0	1	0	0
Heuristic allocation (Recycle)	0	0	0	0	0	0	0	0	0	0	0	0	1	0
Optimal allocation	0	0	0	0	0	0	0	0	0	0	0	0	0	1
Equations	16	26	52	28	30	42	44	78	89	107	89	140	147	10

3.3 Industrial process

The best indication of the success of Amoss 2.0 would be its performance when solving an industrial-scale problem. The ultimate goal of the tool is to be able to solve real plant simulation problems. To date, no fully developed industrial process models exist in Amoss. However, work has been done to create a process model of the most difficult Sasol process. This model is partially built, but is already more complex than any of the test processes. This process was used in Whyte (2018) to quantify the performance of Amoss 1.0.

The partially built process has features summarised in Table 3.3. Take note that the full industrial process contains multiple stochastic inputs, as well as heuristic optimal allocation with recycles. These features should increase the model complexity and ultimately simulation time slightly.

Table 3.3: Features of partially built industrial process

Feature	Number
Components	4
Recycles	4
Buffer tanks	3
Separators	4
Crackers	3
Plants	6
Equations	370

3.4 Benchmarks

Benchmarks for the various deliverables should be determined. This will serve as performance indicator of how successfully the outcomes were addressed. The benchmarks for the deliverables are as follows:

Fast simulation time The simulation time per replication (s/rep) of the test processes will be measured and compared.

Reduction in development time Since the reduction in development time is difficult to quantify, the code generation time (s) of the test processes will be measured and compared. Model development should not take long, depending on the complexity of the process and experience of the modeller.

Linear scalability The simulation time per replication (s/rep) of the test processes as a function of the number of equations will be compared.

3.5 Validation

The requirements of *acceptable accuracy* should be clarified. When considering the accuracy of a single run, there are two factors to take into account: root solving accuracy and integration accuracy. Root solving accuracy is ensured by setting the absolute tolerance of the relevant root solver. Integration accuracy is not truly considered. This is mainly due to the chosen simulation approach. The finite difference method samples inputs, re-evaluates the model and integrates numerically at each time-step. It is expected that the time constants of the buffer tanks are much larger than the selected step size for integration. Euler integration should be sufficient for integration accuracy.

Because root solving and integration accuracy are ensured for a single run, *acceptable accuracy* will be relaxed to simple quantitative checks. Basic sanity checks to determine the possible validity of a single run will be implemented. The basic requirements are as follows:

No invalid values Streams, levels or component rates should only have positive values. In addition to this, no values should tend to infinity or not be a number (nan).

No basic plant constraint violations If flow allocation is implemented, selected streams should remain within given constraints. Additionally, buffer tank ratios (ratio indicating how full a buffer tank is) should remain between 0 and 1.

In addition to basic checks, a qualitative comparison to actual plant results or the currently developed VBA simulation could be used to illustrate *acceptable accuracy*.

CHAPTER 4

EQUATION ORDERING

4.1 Background

After generation of the model variables and equations using the operating instructions and flow sheet model, the task remains to determine an ordered set of algebraic equations for simulation. The starting point of equation ordering in Amoss 2.0 is the mixed algebraic equation set consisting of continuous and discrete equations and variables, as well as variable bounds. The continuous variables and equations stem from the flow sheet model and certain operating instructions. The discrete variables and equations stem from if-statements, min and max functions in the operating instructions. The variable bounds are reasonable expected bounds provided by the user.

In Amoss 1.0, the `sdopt-tearing` repository developed by Baharev (2017b) was used for equation ordering. The ordering method proved functional, but problems were experienced. Problems include that many equations had to be solved simultaneously after ordering to bordered block lower triangular form, the obtained orderings led to numerically unstable simulations and that the repository was not installable as a Python package. This is discussed in Section 2.10.1. Due to the problems experienced, it was decided that custom equation ordering code would be developed for Amoss 2.0. *Fast simulation time*, *numerical stability*, *reduction in development time* and *independent package* are the outcomes to be addressed using the developed equation ordering code.

4.2 Overview

A single superior equation ordering method has not been identified in research. This is due to the often conflicting goals of equation ordering. The first goal of equation ordering is to achieve a numerically stable ordering (*numerical stability*). The second goal is to achieve as little as possible variables to solve simultaneously (*fast simulation time*). And finally, it is important to obtain a numerically stable ordering with as little as possible variables to solve simultaneously in a reasonable amount of time, i.e. a fast ordering method (*reduction in development time*). Due to these factors, practical trade-offs have to be made.

On the selection of a class of equation ordering methods it was decided that strictly heuristic ordering methods will be investigated. This was decided based on the size of the systems encountered in Amoss. A reasonable ordering can be obtained in a short amount of time making use of heuristic ordering methods. To obtain numerical stability in an ordering, a new approach making use of interval mathematics and weighted maximum matching will be used to obtain equation-variable pairings. Incidence matrix decomposition followed by heuristic tearing will be used to ensure that few equations need to be solved simultaneously. The equation ordering method can be summarised in the following steps:

1. Pre-condition equations
2. Determine safe equation-variable pairings
3. Do equation-variable matching
4. Decompose ordering problem into sequentially solvable sub-problems
5. Apply tearing to sub-problems
6. Determine desirable form of incidence matrix
7. Generate input for code generation

4.3 Pre-condition equations

Throughout the equation ordering and simulation method, it is important that equations and variables be either of the continuous or conditional type. Conditional equations are a subset of discrete equations, where the variable solved by the equation can only take on a boolean value of *True* (1) or *False* (0). Following this template simplifies the equation ordering method and improves the performance of the solving methods.

The first step of the equation ordering method is to convert all discrete equations to equivalent conditional and continuous equations. The allowed discrete equations, which are not conditional, are binary min or max functions. An example of a min function is shown in Equation 4.1. An example of a max function is shown in Equation 4.2.

$$x_1 = \min(x_2, x_3) \tag{4.1}$$

$$x_4 = \max(x_5, x_6) \tag{4.2}$$

Conditioning min or max functions proves to be quite simple. The conditional variables y_1 and y_2, defined by Equation 4.3 and Equation 4.4, are introduced. Equation 4.1 and Equation 4.2 can now be rewritten as Equation 4.5 and Equation 4.6, respectively. A single discrete min or max function is converted to a conditional and continuous equation.

$$y_1 = x_2 < x_3 \tag{4.3}$$

$$y_2 = x_5 > x_6 \tag{4.4}$$

$$x_1 = y_1 x_2 + (1 - y_1) x_3 \tag{4.5}$$

$$x_4 = y_2 x_5 + (1 - y_2) x_6 \tag{4.6}$$

4.4 Safe equation-variable pairings

Prior to equation-variable matching, safe and unsafe pairings need to be determined. For the known variables, continuous and conditional, an expected interval is assigned. The continuous variables take on the default bounds specified by the user, while the conditional variables take on the interval $[0, 1]$.

Once reasonable bounds have been identified for all variables, interval arithmetic can be used to determine unsafe pairings. The procedure followed consists of solving every equation for each possible variable match. The solutions are then tested using interval mathematics, to determine whether the equation-variable pairing can lead to zero division. The original approach, suggested by Ali Baharev, is discussed Section 2.3.4. The original method for determining whether a variable is a safe elimination is illustrated in Algorithm 8.

The algorithm employed by Ali Baharev is computationally intensive and quite slow. For instance, if a solution does not have a denominator, it is clear that no interval will cause zero division and interval math is not required. Additionally, the entire solution, numerator and denominator, are evaluated and compared to a very large interval $(-M, M)$. The interval arithmetic is difficult and unnecessarily expensive. An improved, more computationally efficient, method for determining whether a variable is a safe elimination was determined and is illustrated in Algorithm 9. This method breaks up the solution into a numerator and denominator. If the denominator is a number, the pairing is safe. If the denominator is symbolic, only the denominator is evaluated using interval arithmetic. If zero is in the denominator's possible interval, zero division is possible and the pairing is unsafe.

Algorithm 8 Original algorithm for determining safe elimination

```
$M = $ 1E10
solution = solve(equation, variable)
if len(solution) = 1 then
    solution_interval = interval_math(solution, parameter)
    is_safe = solution_interval in $(-M, M)$
else
    is_safe = False
end if
return is_safe
```

Algorithm 9 Improved algorithm for determining safe elimination

```
solution = solve(equation, variable)
if len(solution) = 1 then
    numer, denom = fraction(solution)
    if denom is not a number then
        denom_interval = interval_math(solution, parameter)
        is_safe = 0 not in denom_interval
    else
        is_safe = True
    end if
else
    is_safe = False
end if
return is_safe
```

4.5 Equation-variable matching

Once the safe variables for each equation have been determined, equations and variables should be matched. The aim is to obtain as many as possible numerically safe pairings, while ensuring that as many as possible variables and equations are matched. Firstly, a cost matrix is generated from the safe and unsafe variables in each equation. We assign the smallest value for safe elimination (i.e. -10), a second slightly larger value for unsafe eliminations (i.e. -5) and infeasible pairings are marked with an arbitrary large positive value (i.e. 50). Consider the system of equations shown below, with $x_i \in [0, 100]$:

$$x_2 + 2x_3x_4 - 13 = 0$$

$$5x_1 + 2x_1x_2 - 15 = 0$$

$$x_1 + 2x_4 + 3x_2x_4 - 10 = 0$$

$$x_3 + x_4 + 5x_2x_5 - 12 = 0$$

$$x_1 + 4x_1x_4x_5 + x_5 - 10 = 0$$

A summary of all equation variables and safe variable assignments is shown in Table 4.1. This system can be converted to the cost matrix shown in Equation 4.7.

Table 4.1: Example system feasible and safe variable assignments

Equation	Feasible	Safe
1	x_2, x_3, x_4	x_2
2	x_1, x_2	x_1
3	x_1, x_2, x_4	x_1, x_4
4	x_2, x_3, x_4, x_5	x_3, x_4
5	x_1, x_4, x_5	x_1, x_5

$$C = \begin{bmatrix} 50 & -10 & -5 & -5 & 50 \\ -10 & -5 & 50 & 50 & 50 \\ -10 & -5 & 50 & -10 & 50 \\ 50 & -5 & -10 & -10 & -5 \\ -10 & 50 & 50 & -5 & -10 \end{bmatrix} \tag{4.7}$$

After setting up the cost matrix, the combinatorial optimisation problem known as the linear assignment problem should be solved. Rows and columns of the cost matrix are only allowed to be assigned once. The objective is to minimize the sum of the cost matrix values. Due to Amoss models necessarily being fully specified, the obtained maximum matching should be a perfect matching.

The linear assignment problem is solved in Amoss 2.0 by making use of the `linear_sum_assignment` function, which is part of the `scipy.optimize` module. The implementation makes use of the Hungarian algorithm to solve the weighted maximum matching problem. Weighted maximum matching and the Hungarian algorithm are discussed in greater depth in Section 2.3.6. Applying weighted maximum matching to the cost matrix in Equation 4.7, we obtain the equation-variable assignments shown in Table 4.2.

Using this method as the first step of equation ordering adds an increased numerical stability to the final ordering. The aim is to minimize the possibility of simulation instability due to zero division. This approach is not perfect, since it is not always possible to have exclusively safe variable assignments. However, it proves to be a good practical first step.

Table 4.2: Example system variable assignments

Equation	Variable
1	x_2
2	x_1
3	x_4
4	x_4
5	x_5

4.6 Incidence matrix decomposition

Once equation-variable matching has been achieved, the next proposed step is decomposition of the incidence matrix. The incidence matrix is generated from the continuous and conditional equations and variables. An incidence matrix describes the relationship of two unique sets of objects to one another. In the case of equation ordering, the relationship of equations and variables is described. This is discussed in more detail in Section 2.3.2. The aim of decomposition is to re-order the original incidence matrix into block lower triangular form. This equivalently means breaking up the original problem into a series of sequentially solvable sub-problems. This is discussed in more detail in Section 2.3.7.

The `CSparse` library developed by Timothy A. Davis in the C++ language, is a renowned sparse matrix linear algebra library (Davis, 2006). It contains many algorithmically optimised algorithms, including a very fast Dulmage-Mendelsohn decomposition implementation. Making use directly of this library and its Dulmage-Mendelsohn implementation was not possible for two reasons. Firstly, because the Amoss project is programmed exclusively in Python, using a C++ library is possible with tools like `Cython`, but is difficult to implement. Secondly, the original Dulmage-Mendelsohn algorithm achieves matching making use of maximum matching and does not have accommodation for weighted maximum matching.

However, Richard Lincoln, who did follow-up work on the `CSparse` package, wrote a Python translated version of the package (Lincoln, 2012). This code was included in the code base and modified to make use of weighted maximum matching, instead of maximum matching. Since, the `CSparse` package is only used for the Dulmage-Mendelsohn decomposition, the code was also truncated to only contain the decomposition method. This modified implementation is used exclusively to obtain the decomposed incidence matrices in Amoss 2.0.

To illustrate the overall ordering method, consider an example using an existing test process. Test process 11 was selected. It has the unmatched incidence matrix shown in Figure 4.1. Black dots are used to indicate incidences and white dots are used to indicate the absence of incidences. When weighted maximum matching is applied to the unmatched incidence matrix, the matched incidence matrix in Figure 4.2 is obtained. Take note that the main diagonal is completely populated. Safe variable matches are indicated on the diagonal by blue dots and unsafe matches are indicated on the diagonal by red dots. Take note that the main diagonal should be completely populated. Finally, Dulmage-Mendelsohn decomposition is applied to the matched incidence matrix and the block lower triangular incidence matrix in Figure 4.3 is obtained. Take note that the blocks are indicated by grey.

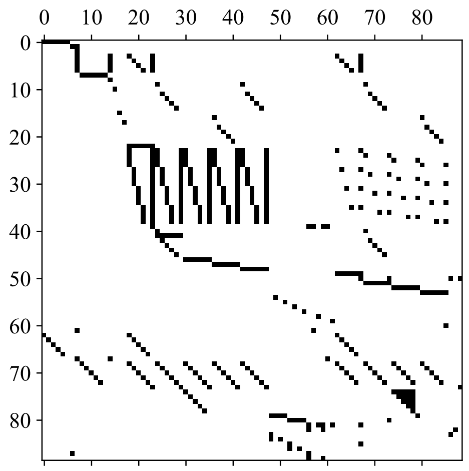


Figure 4.1: Unmatched incidence (test process 11). Incidences are indicated by black.

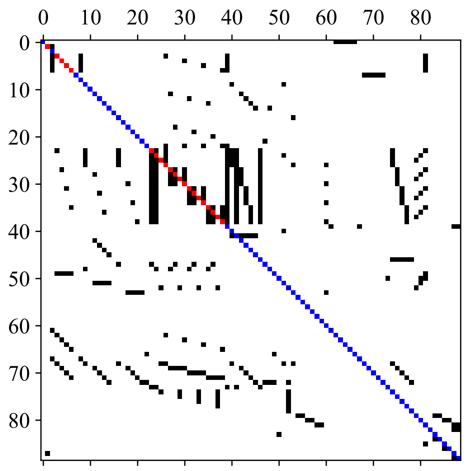


Figure 4.2: Matched incidence (test process 11). Safe and unsafe matches are indicated by blue and red, respectively.

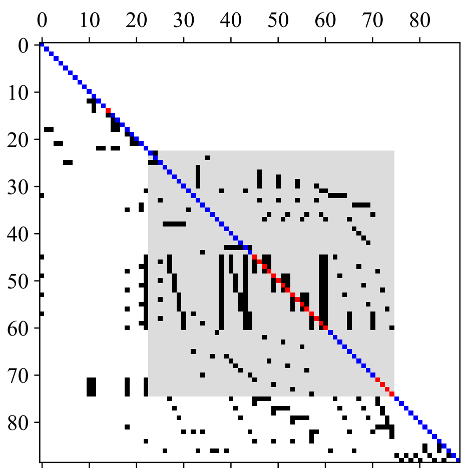


Figure 4.3: Block lower triangular incidence (test process 11). Grey indicates blocks.

4.7 Sub-problem tearing

After division of the large equation ordering problem into smaller sub-problems using Dulmage-Mendelsohn decomposition, the task remains to identify variables to solve in each equation. This task is accomplished in Amoss 2.0 by making use of Cellier's tearing. Cellier's tearing is a heuristic tearing method, which has been quite successfully implemented as part of modelling tools, such as OpenModelica and Dymola. The traditional implementation of Cellier's tearing is graph-based and discussed in detail in Section 2.3.8. No packages that allow implementation of Cellier's tearing to a sub-system was found. This is due to the method rarely existing on its own and mostly being a part of larger equation ordering methods. It was decided that a custom Cellier's tearing algorithm would be implemented.

The original Cellier's tearing method has fairly significant flaws. Firstly, conditional equations and variables are not taken into account. This is a problem, because it is highly undesirable that conditional variables and equations become the residual variable or equations. This would change the nature of the continuous root finding problems and make the problems substantially more difficult to solve. Secondly, variable matching is done recursively, using maximum matching, which again is not desirable. It would make more sense to preserve the original equation-variable matches, obtained from the weighted maximum matching step. Recursive weighted maximum matching would also prove computationally impractical. Finally, because the main approach taken for the Amoss 2.0 equation ordering method is the incidence matrix approach, an incidence matrix implementation rather than a graphical implementation will be used. The custom incidence matrix implementation for Cellier's tearing is shown in Listing 4.1.

Applying Cellier's tearing to the block lower triangular incidence shown in Figure 4.3, the spiked lower triangular incidence shown in Figure 4.4 is obtained. This form has the unique property that the original system is broken into sequentially solvable sub-problems. The simultaneous solution regions are indicated using grey.

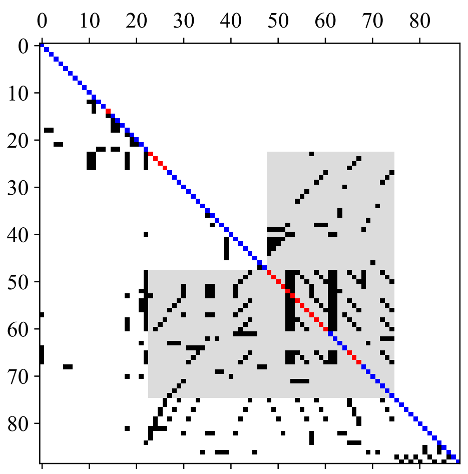


Figure 4.4: Spiked lower triangular incidence (test process 11). Grey indicates simultaneous solution regions.

```python
def cellier_tearing(match_incidence, discrete_indices):
    # Copy incidence matrix
    copy_incidence = numpy.array(match_incidence)

    # Do diagonal pivoting
    row_perm, col_perm = dm_perm(copy_incidence)
    decomposed_incidence = copy_incidence[row_perm, :][:, col_perm]
    is_lower_triangular = numpy.all(is_solved(decomposed_incidence))
    if is_lower_triangular:
        return row_perm, []

    # Create empty assigned index
    solve_index = []
    l = copy_incidence.shape[0]
    unassigned_index = list(range(l))

    for n in range(l):
        # Determine row and column sum
        row_sum = copy_incidence.sum(axis=1)
        max_row_location = numpy.where(row_sum == numpy.max(row_sum))[0]
        max_row_location =\
        [i for i in max_row_location if unassigned_index[i] not in discrete_indices]
        col_sum = copy_incidence[:, max_row_location].sum(axis=0)

        # Determine best index
        i = max_row_location[col_sum.argmax()]

        # Append index
        solve_index.append(unassigned_index[i])
        del unassigned_index[i]

        # Delete row and col entries
        copy_incidence = numpy.delete(numpy.delete(copy_incidence, i, 0), i, 1)

        # Determine causal equations
        row_perm, col_perm = dm_perm(copy_incidence)
        decomposed_incidence = copy_incidence[row_perm, :][:, col_perm]
        is_lower_triangular = numpy.all(is_solved(decomposed_incidence))
        if is_lower_triangular:
            ordered_index = [unassigned_index[i] for i in row_perm]
            break
    return ordered_index, solve_index
```

Listing 4.1: Custom Cellier's tearing implementation

4.8 Desirable incidence matrix form

There are two incidence matrix forms commonly used for simulation generation. These forms are discussed in Section 2.3.9 and are:

Spiked lower triangular form The spiked lower triangular incidence matrix shown in Figure 4.4 contains blocks representing sequentially solvable sub-problems, each with its own explicit region and a sequentially solvable region. This form can be seen as a local ordering. It is often used for simulation and can be quite efficient. However, several sequential solvers would have to be used if multiple blocks are encountered. This is difficult to implement. Additionally, if the algebraic equation set is not used for root finding, but rather forms part of an optimisation, the form cannot be used.

Bordered block lower triangular form The most common form of incidence matrix used for simulation is the bordered block lower triangular form. The bordered block lower triangular form can be seen as a global ordering. This form divides the entire equation set into two distinct regions: an explicitly solvable region and a simultaneous solution region. This requires only a single solver to be generated, which is simpler to implement. Additionally, if the algebraic equation set forms part of a greater optimisation, the simultaneous solution region can simply be added to the optimisation equality constraints.

For simplicity and taking into account that optimisation can be encountered, the bordered block lower triangular form was selected as the desired incidence matrix form. To determine the bordered block lower triangular form from the spiked lower triangular form is quite simple. Any column in the spiked lower triangular form that contains an incidence above the diagonal is moved to the right or the simultaneous solution region. The remaining entries to the left form a lower triangular matrix and make up the explicitly solvable region. The bordered block lower triangular form of the spiked lower triangular incidence in Figure 4.4 is shown in Figure 4.5. Grey is used to indicate the simultaneous solution region.

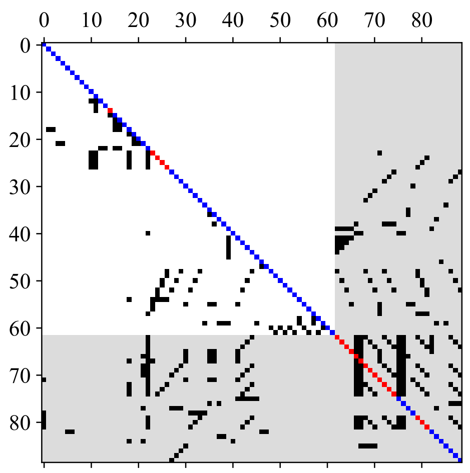


Figure 4.5: Bordered block lower triangular incidence (test process 11). Grey indicates the simultaneous solution region.

4.9 Code generation input

Using the bordered block lower triangular incidence matrix, an input should be generated, which can be used for code generation. The first class necessary is an *Equation* class. This class stores the information of an ordered equation. The properties are as follows:

equation (type: SymPy equation) The equation to solve.

variable (type: SymPy symbol) The variable assigned to solve in the equation.

equation_variable (type: list of SymPy symbols) All variables in the equation.

is_safe (type: boolean) Whether the equation-variable matching is safe or not.

is_conditional (type: boolean) Whether the equation is conditional or not.

Three possibilities exist for the equation ordering result. The first is an explicit ordering. This occurs if an explicit solution, i.e. a solution with no simultaneous solution region, is determined. The second is an implicit ordering. This occurs if the ordering solution contains a simultaneous solution region. Finally, if an optimisation goal is part of the simulation problem, an ordering with an optimisation goal is obtained. The three classes, *Explicit*, *Implicit* and *Optimisation*, are created to contain the information necessary for code generation of each ordering result.

CHAPTER 5

SIMULATION APPROACH

5.1 Background

The simulation problems generated from Amoss models are difficult to simulate for various reasons. To understand how the problem arises, model generation should be considered. Amoss models are generated from the flow sheet model, operating instructions, statistical data and optimisation objectives. These facets are combined to generate a highly complex model. The simulation elements include: first order ordinary differential equations, continuous equations, conditional equations, stochastic elements and disjoint constraint optimisation. Model generation is discussed in detail in Section 2.10.2.

The generated models are not only complex, but the number of equations can be in the order of hundreds. Simulating these large complex models has been the topic of research for over a decade. Whyte (2018) attempted to solve this problem, using a finite difference approach, which has been the main solution to this problem. The simulation algorithms are discussed in Section 2.10.2. The simulation approaches to the implicit ordering, explicit ordering and optimisation problems are shown in Algorithm 5, Algorithm 6 and Algorithm 7, respectively.

There are two main flaws with the simulation approaches followed. The first is a substantial flaw in Algorithm 5. For the implicit ordering problem, a single root find of the combined continuous-conditional system is attempted. The root finding method used is a simple Newton method. Due to the conditional variables, the system will experience discontinuities in the gradient, which makes failure using a Newton method and almost all root finding methods highly likely. This approach led to numerical instability and slow simulations. The second flaw is related to the optimisation of Algorithm 7. A rough approximation of the disjunctive optimisation was attempted. This is discussed in more detail in Section 5.6.3.

Optimal flow allocation using heuristic rules is also a major point of discussion. This was one of the primary concerns identified at the start of the project. Optimal flow using heuristic rules in the forward direction was attempted and successfully implemented in Amoss 1.0. This is discussed in Section 5.6.1. However, a solution to this problem in event of system recycles was not obtained. This is necessary for simulation of most actual systems. Due to the issues experienced in the simulation approach, it was decided that it would be revisited in Amoss 2.0. *Fast simulation time*, *numerical stability* and *acceptable accuracy* are the outcomes to be addressed by development of the simulation approach.

5.2 Overview

The simulation approach consists of many facets which should be addressed. For the implicit ordering problem it is essential that a stable (*numerical stability*), fast (*fast simulation time*) and sufficiently accurate (*acceptable accuracy*) method for root finding of the combined system be selected. When considering the simulation speed of the optimal flow allocation problem, it is clear that the heuristic optimisation be used when possible. The solution time of the disjunctive optimisation problem will be orders of magnitude larger than evaluation of if-conditions. Conditions for heuristic optimisation need to be determined. A solution to the heuristic allocation problem in event of recycle should be determined. If this can be attained, *fast simulation time* of large-scale problems with optimal flow allocation should be possible. The following simulation approach aspects were investigated:

- Simulation inputs
- Numerical integration
- Root finding
- Optimal flow allocation

5.3 Inputs

Input handling is important for simulation. There are two main types of inputs allowed: stochastic and custom inputs. These inputs are generated differently, but are handled effectively in the same manner throughout simulation. The simulation inputs are generated prior to each replication. Throughout simulation, at every time-step t, the variable value is simply read from the pre-generated input.

5.3.1 Stochastic

Stochastic inputs form the basis of Monte Carlo analysis, as discussed in Section 2.2.3. User provided distribution data consisting of variable values (x), probability values (p) and whether the distribution is continuous or discrete is provided. Prior to simulation of a replication, the stochastic variable values for the entire replication run is generated. Stochastic input generation was implemented by Whyte (2018) in Amoss 1.0 and is discussed in more detail in Section 2.2.2.

5.3.2 Custom

Custom inputs have been added as a new feature in Amoss 2.0. As an example, consider a simulation of 100 hours, with a step size of 1 hour. The user can provide a time related data set representing a custom variable input, with an example shown in Table 5.1. The user should also state whether the value changes are continuous or discrete.

Table 5.1: Example custom input

Time (hr)	Value
0	0
20	4
40	10
60	6
80	2
100	0

Using the nature of the value changes of a given data set, the input data can be extended over the entire simulation time-span. If the value changes are continuous, the custom input shown in Figure 5.1 is obtained. Alternatively, if the value changes are discrete, Figure 5.2 is obtained. This feature allows the user to effectively generate input data of any form, including step disturbances, ramp disturbances, time dependant stochastic inputs, etc.

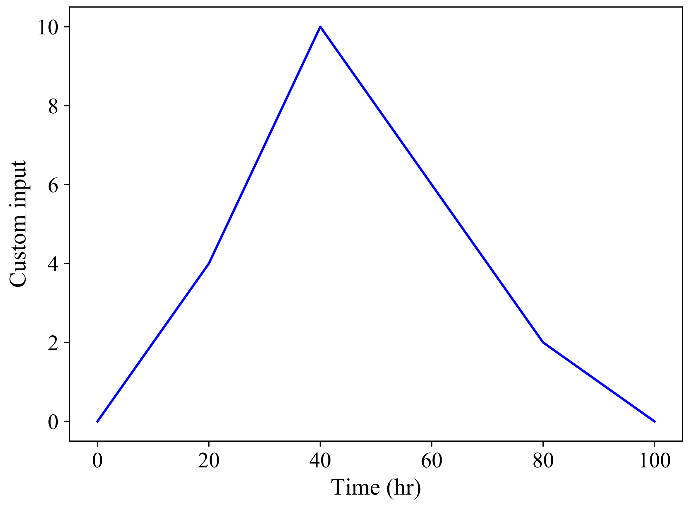


Figure 5.1: Custom input with continuous value changes

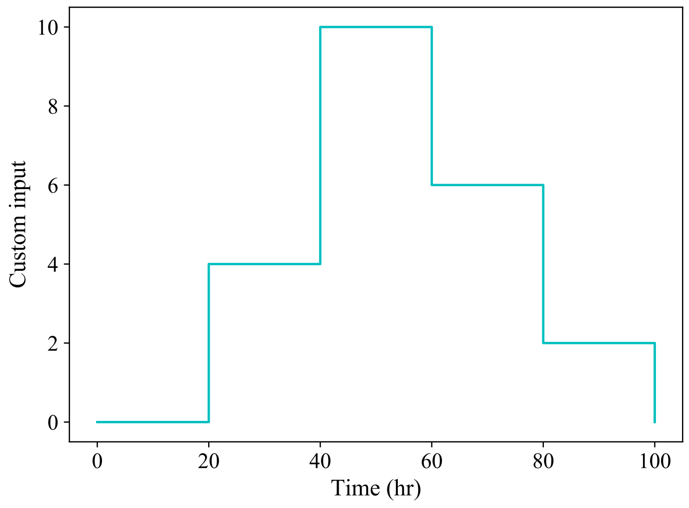


Figure 5.2: Custom input with discrete value changes

5.4 Numerical integration

Ordinary differential equations (ODEs) are a norm in Amoss generated models. This is a result of buffer tank accumulation inherently being described by the mass flow continuity equation. The finite difference approach used for simulation requires numerical integration at every time-step. The explicit Euler method shown in Equation 5.1, which was used by Whyte (2018), was selected for numerical integration. Higher order methods can not be used, because the smoothness assumptions do not hold. This is due to the augmented systems containing stochastic elements, conditional equations and possibly optimisation goals. This is discussed in more detail in Section 2.6.

$$y_{n+1} = y_n + hf(t_n, y_n) \tag{5.1}$$

5.5 Root finding

5.5.1 Continuous system

To simulate as fast and as robustly as possible, it is important that appropriate root finding methods are selected. The first root finding problem commonly encountered is continuous equation root finding. It should be noted that the combined Amoss models often also contain both conditional equations and flow optimisations. The alternative cases will be discussed in more detail later.

It is often possible to quickly determine the root of a continuous system when an initial guess near the solution is available. This can be achieved in many ways, with Newton solvers and its variants being common choices. The properties of Newton solvers are discussed in more detail in Section 2.4.1.

Root finding in Amoss simulations is done at every time-step. This is due to the finite difference approach used. The nature of the root finding problem is thus dynamic and not static. If a valid initial root is determined, each previous time-step root can be used as a guess starting value for the next time-step. This implies that throughout the simulation, a different number of root finding steps may be required, based on how close the previous root is to the next solution. The initial value method is discussed in Section 5.5.3.

Because speed is of utmost importance, a simple Newton root finding scheme is a good choice for root finding during the dynamic simulation. Jacobian evaluation is critical to Newton root finding and it was decided that automatic differentiation be used to determine gradients. Automatic differentiation is much more computationally efficient than traditional methods, while maintaining working accuracy. Automatic differentiation is discussed in more detail in Section 2.7.

5.5.2 Hybrid system

As discussed in Section 5.1, one of the main problems experienced with root finding in Amoss 1.0 is that conditional equations often form part of the equation set to be solved. This is the most common scenario encountered in the implicitly ordered problem, because if-blocks are commonly present in the operating instructions. This creates a significant problem for continuous root finding methods like Newton methods, because discontinuities are introduced in the resulting Jacobian. This means that the solving strategy will likely fail, especially when the number of equations to solve becomes significant.

A practical solution to this problem, which makes use of a fixed-point iteration for the conditional variables, has been implemented by the OpenModelica tool kit and is discussed in Section 2.4.2. The main idea is to separate the conditional and continuous equation sets after equation ordering. The conditional equations and variables are iteratively solved in a fixed-point iteration, with continuous root finding occurring at every iteration. This should work well, since the conditional variables need only take on a boolean value. The implemented fixed-point iteration scheme for Newton root finding is shown in Algorithm 10, with x denoting the continuous variables and y denoting the conditional variables.

Algorithm 10 Fixed-point iteration scheme for hybrid systems with Newton solving

```
$y_{check} = 0$
$y_{calc} = y_{prev}$ (from last event)
while $y_{check} \neq y_{calc}$ do
    $y_{check} = y_{calc}$
    $x = \text{Newton}(x, y_{check})$
    $y_{calc} = \text{Conditional}(x, y_{check})$
end while
return $x, y_{calc}$
```

5.5.3 Initial value

The combined root finding problems of Amoss models are expected to have a single unique root at each time-step. Thus, the root finding scheme in Section 5.5.2 is not expected to be highly sensitive to the initial value. From time zero onward, the previous root value can be used as an initial guess. However, it is important to determine a reasonable starting point for the first root solve at time zero. Because reasonable lower and upper bounds are provided for each variable, a starting root value for the residual continuous variables at the mid-point of the lower and upper bounds can be used. The remaining continuous and conditional variables can be evaluated sequentially using the combined model. This method is clearly sub-optimal, because starting root values can be quite far from the solution. However, it is a practical and easy to implement method of generating initial values for root finding. Considering that the Newton solver accuracy is set beforehand, this method works sufficiently.

5.6 Optimal flow allocation

Incorporating the optimisation problem stemming from mass flow allocation has been a major problem of Amoss simulations. It is often the case in the Moss methodology that flow distribution through multiple pipes is possible. The operating instruction stemming from this is set with the ambiguous goal of optimally distributing mass flow economically. The difficulty of this problem is increased when it is taken into account that for the operation of some lines a minimum flow is required and that maximum plant constraints can be added. This problem can be stated as an optimisation problem with an economic objective considering the distribution of streams with a particular availability constraint, maximum constraints and disjoint minimum constraints. Optimal flow allocation is discussed in more detail in Section 2.10.2. The general optimal flow allocation problem is described by Equation 5.2 (Whyte, 2018).

$$\begin{aligned} \min_{x} \quad & f(x) \\ \text{s.t.} \quad & \sum x \leq x_{available} \\ & h(x) \leq h_{max} \\ & g(x) = 0 \text{ or } g(x) \geq g_{min} \end{aligned} \tag{5.2}$$

5.6.1 Special case: Forward flow

A special case of the problem stated in Equation 5.2 is possible if the conditions below are met:

1. maximum constraints ($h(x) \leq h_{max}$) are imposed on the available streams (x_i)
2. disjunctive minimum constraints ($g(x) = 0$ or $g(x) \geq g_{min}$) are imposed on the available streams (x_i)

3. no mass accumulating units i.e. buffer tanks are situated between input flow ($x_{available}$) and available streams (x_i)

This simplifies the optimisation problem, which can be reduced to a series of if-statements with priority allocation. Whyte (2018) did the original work on this matter, specifically when no recycles are involved. A function for use in the operating instructions called `allocate` was created. This is discussed in Section 2.10.2.

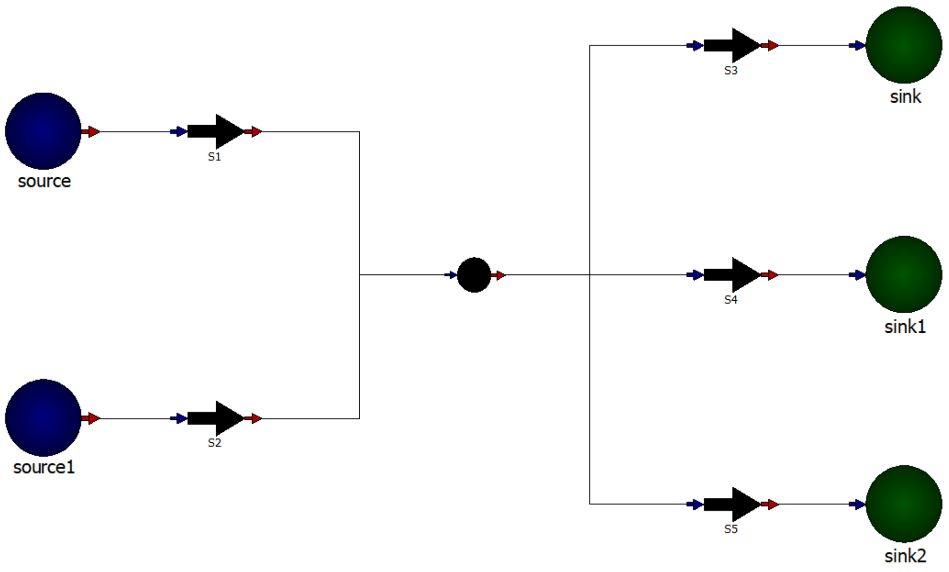


Figure 5.3: Flow diagram illustrating forward flow allocation problem (Whyte, 2018)

Figure 5.3 will be used to illustrate a typical example of this special case optimisation. For the example we have $S_{avail} = S_1 + S_2$ to optimally distribute. The minimum disjoint constraints are $S_3 \geq 10$ or $S_3 = 0$ and $S_4 \geq 10$ or $S_4 = 0$. The maximum constraints are $S_3 \leq 30$ and $S_4 \leq 40$. It should be noted that no constraints are imposed on S_5, because it is a necessary degree of freedom for the stream mass balance to hold. The allocate function can be used in model operating instructions as shown below:

```
S_avail = S1_total + S2_total
S3_min_p = 0
S4_min_p = 1
S3_max_p = 1
S4_max_p = 0
S3_total, S4_total = allocate(S_avail, [S3_min_p, S4_min_p], [10, 20], [S3_max_p, S4_max_p],
    [30, 40])
```

The allocate operating instruction is converted to a set of continuous and conditional equations using the algorithm developed by Whyte (2018), illustrated in Figure A.1. Making use of the developed equation ordering method in Chapter 4, the model description shown Listing B.1 is obtained. The result is an explicitly ordered solution, as expected.

If for example the variables S_1 and S_2 take on an arbitrary value of 25, we have S_{avail} of 50 to distribute between the streams S_3, S_4 and S_5. The values determined from the equation set is $S_3 = 10$, $S_4 = 40$, $S_5 = 0$. The algorithm followed will ensure that the minima are allocated, but because S_4 has the higher maximum priority, its maximum capacity will be reached first.

5.6.2 Special case: Flow with recycle

One of the biggest problems faced historically is accommodation for the special case of the optimal allocation when a recycle is involved. The logic in the forward direction is difficult to maintain when the input and output streams affect one another. Take for example Figure 5.4. We have $S_{avail} = S_1 + S_2 + R_1$ to optimally distribute. R_1 is calculated as $R_1 = 0.1(S_3 + S_4)$. In addition to this, we again have the minimum disjoint constraints $S_3 \geq 10$ or $S_3 = 0$ and $S_4 \geq 10$ or $S_4 = 0$ and the maximum constraints $S_3 \leq 30$ and $S_4 \leq 40$.

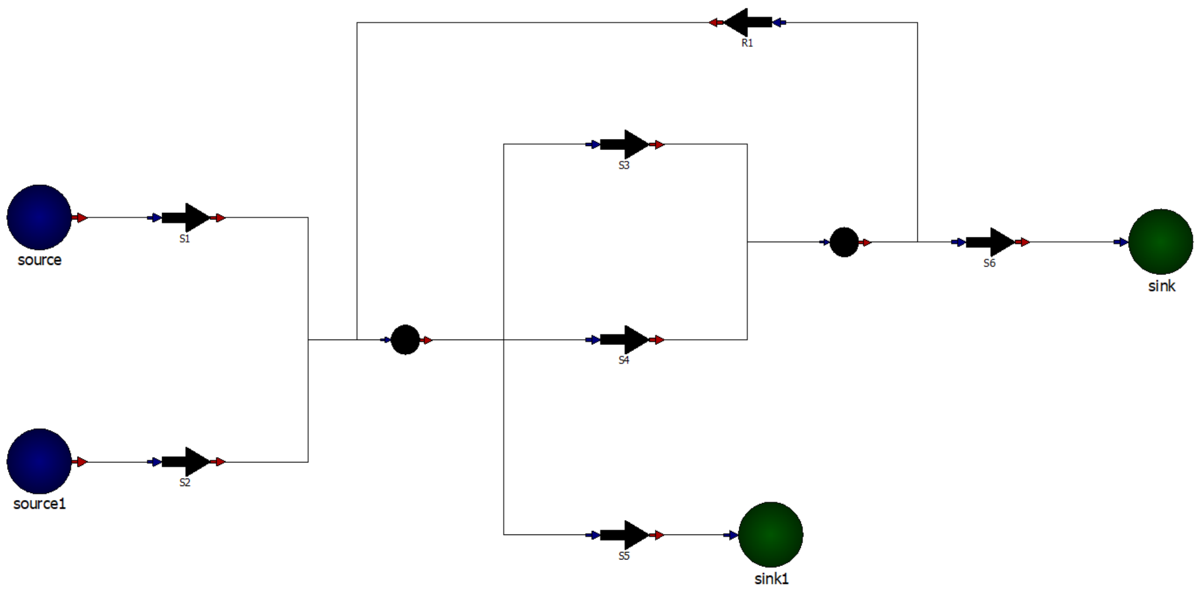


Figure 5.4: Flow diagram illustrating flow allocation problem with recycle

Although the examples in Figure 5.3 and Figure 5.4 look similar, there is a significant difference. This difference is that the recycle R_1 adds to the inflow of the node for allocation. This creates an additional relationship between the input and output of the flow allocation problem. Because the allocate instruction is converted to a simple set of continuous and conditional equations, it can be used in the operating rules in the conventional way. Thus, we have the operating rules:

```
R1_total = 0.1*(S3_total + S4_total)
S_avail = S1_total + S2_total + R1_total
S3_min_p = 0
```

```
S4_min_p = 1
S3_max_p = 1
S4_max_p = 0
S3_total, S4_total = allocate(S_avail, [S3_min_p, S4_min_p], [10, 20], [S3_max_p, S4_max_p],
    [30, 40])
```

Taking into account that an additional relationship between the input and output is added, it is expected that the equation ordering method will yield an implicit solution. Making use of the developed equation ordering method in Chapter 4, the model description shown Listing B.2 is obtained. As expected, the solution is an implicit ordering. This is an important result. Due to the newly introduced fixed-point iteration scheme for root finding in Section 5.5.2, it is possible to solve the heuristic allocation problem with a recycle. This will be handled in the same manner as any if-condition in the operating instructions.

5.6.3 General case

When the special case is not applicable, solving the flow allocation problem as an optimisation problem is necessary. This can be done by having the user assign an economic priority to each of the streams. This makes practical sense, because the streams lead to unit operations with varying economic influence on the overall plant. The objective function $f(x)$ of Equation 5.2 should be defined. Whyte (2018) selected an objective function of the form in Equation 5.3, with a decaying factor d, which can be arbitrarily chosen.

$$f(x) = -d^0 x_0 - d^1 x_1 - d^2 x_2 - \ldots - d^n x_n \tag{5.3}$$

Whyte (2018) could not sufficiently address the general case of the optimal flow allocation problem in Amoss 1.0, as discussed in Section 2.10.2. The largest problem is that the disjunctive constraints on $g(x)$ make the problem solution space disjunctive and render the original problem unsolvable by most MINLP solvers. Whyte (2018) made a crude approximation of the disjunctive programming problem in Equation 5.2. As expected, counter-intuitive results were obtained by using this formulation.

A more commonly followed approach, as discussed in Section 2.5, is to reformulate the disjunctive program as an MINLP problem using either the big-M reformulation or complex Hull reformulation. Complex hull formulation leads to smaller solution spaces and thus faster solution time. The constraint $g(x)$ can be represented by Equation 5.4.

$$\begin{bmatrix} Y_1 \\ 0 \leq x_1 \leq 0 \end{bmatrix} \bigvee \begin{bmatrix} Y_2 \\ g_{min} \leq x_1 \leq \infty \end{bmatrix} \tag{5.4}$$

Making use of big-M reformulation, we obtain the disjunctive constraint relaxation shown in Equation 5.5.

$$\begin{gathered} y_1 + y_2 = 1 \\ y_1, y_2 \in \{0, 1\} \\ -M(1 - y_1) \leq y_1 \leq M(1 - y_1) \\ -M(1 - y_2) + g_{min} \leq y_1 \end{gathered} \tag{5.5}$$

The convex Hull reformulation of the problem is shown in Equation 5.6. Equation 5.6 can be simplified to Equation 5.7.

$$\begin{aligned} & y_1 + y_2 = 1 \\ & y_1,\ y_2 \in \{0, 1\} \\ & x_1 = x_{11} + x_{12} \\ & 0 \leq x_{11} \leq My_1 \\ & 0 \leq x_{12} \leq My_2 \\ & 0 \leq x_{11} \leq 0 \\ & g_{min} y_2 \leq x_{12} \end{aligned} \tag{5.6}$$

$$\begin{aligned} & y_1 + y_2 = 1 \\ & y_1,\ y_2 \in \{0, 1\} \\ & 0 \leq x_1 \leq My_2 \\ & g_{min} y_2 \leq x_1 \end{aligned} \tag{5.7}$$

Because Amoss disjunctive constraints are linear in nature, the relaxed solution region will be linear. As an example, if we have $M = 200$ and $g_{min} = 50$, we obtain the relaxed solution regions for the big-M and convex Hull reformulation shown in Figure 5.5.

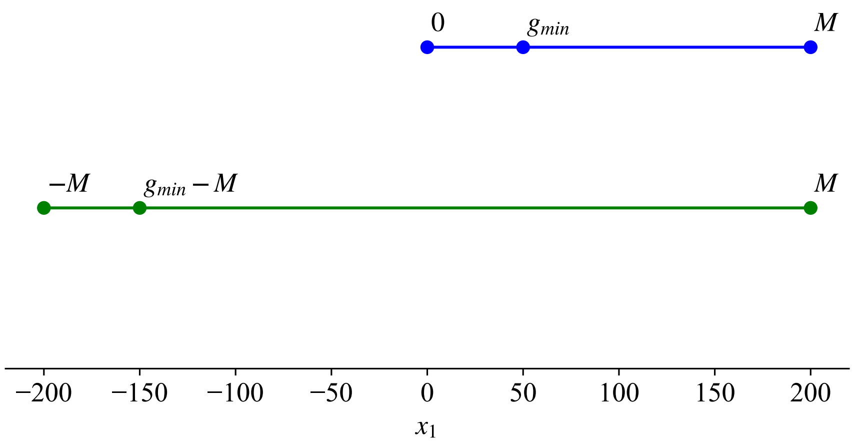


Figure 5.5: Relaxed solution region of Amoss disjunctive constraints. Blue and green denote the convex Hull and big-M reformulations, respectively.

It is clear that the relaxed solution region for the big-M reformulation is twice that of the convex Hull reformulation. This means that the reformulated problem using big-M formulation will take significantly longer to solve using an MINLP solver. Thus, the convex Hull reformulation will be used to relax the disjunctive programming problem in Equation 5.2. Substituting the objective function $f(x)$ and doing convex Hull reformulation we obtain the alternative form of the allocation problem shown in Equation 5.8.

$$\begin{aligned} \min_{x,y} \quad & f(x) = -d^0x_0 - d^1x_1 - d^2x_2 - \ldots - d^nx_n \\ \text{s.t.} \quad & \sum x \leq x_{available} \\ & h(x) \leq h_{max} \\ & y_1 + y_2 = 1 \\ & y_1,\, y_2 \in \{0,1\} \\ & 0 \leq x \leq My_2 \\ & g_{min}y_2 \leq x_1 \end{aligned} \tag{5.8}$$

CHAPTER 6

CODE GENERATION

6.1 Background

After equation ordering, discussed in Chapter 4, has been applied to the model algebraic variables and equations, the simulation approach, discussed in Chapter 5, should be implemented for simulation of the model. The starting point for code generation in Amoss 2.0 is the equation ordering result, stochastic and custom inputs and the integration variables and equations. The equation ordering result is provided as an *Explicit*, *Implicit* or *Optimisation* object. The stochastic and custom inputs are generated inputs to be sampled during simulation. The integration variables and equations are the equations stemming from Euler integration of the buffer tank ODEs.

There were minor flaws regarding the code generation method in Amoss 1.0, discussed in Section 2.10.3. Simulation code could have been generated more efficiently, but with the major faults in the simulation approach, these changes would have had little effect. Due to the large changes in the simulation approach and equation ordering methods, the code generation had to be rebuilt entirely. Rebuilding the code generation does allow the opportunity to re-evaluate the most effective implementation of the simulation approach. *Fast simulation time*, *reduction in development time* and *independent package* are the outcomes to address using the new code generation method.

6.2 Overview

Code generation entails creating simulation code for a generated model. The simulation code will be used to simulate the various scenarios and multiple replications for Monte Carlo analysis. It is important that the code generation method be robust and reliably generate code for the *Explicit*, *Implicit* and *Optimisation* equation ordering objects. Fast simulating code (*fast simulation time*) should be generated reasonably quickly (*reduction in development time*). For the simulation of the various equation ordering results, efficient solving and evaluation tools need to selected. The generated simulations also have the potential to be run in parallel and parallel processing methods should be looked into. Ideally, the Python standard library and widely used open-source packages should be used (*independent package*). The code generation aspects looked into are:

- Template
- Solver and evaluation
- Parallel processing

6.3 Template

Amoss Monte Carlo simulations require running multiple replications of multiple scenarios of a developed model. Amoss 1.0 generated simulation code as non-executable `.py` Python files, which were imported and used by the greater Python code base. This design decision created an unwanted dependency between simulation of various scenarios. The simulation code should be generated in such a way that any replication of any scenario can be simulated independently by using it. To achieve this utility, code will be generated as an executable `.py` Python file, which will be provided simulation arguments using the command line. The layout of the generated code is shown in Listing 6.1. The functions in the general code layout are as follows:

generate_stochastic_input Generate stochastic inputs for simulation from distribution data.

generate_customs_input Generated custom inputs for simulation from custom input data.

initialise_solver Initialise fixed-point iterative Newton solver or MINLP solver.

initialise_function Initialise evaluation functions.

initial_value Initial value function, starting with the residual variable values at the mid-point of the reasonable bounds. The other continuous and conditional variables are sequentially evaluated.

mass_balance Simulation of a single time-step. Stochastic and custom inputs are sampled, the relevant solving is done, followed by evaluation of the remaining variables and Euler integration.

run_replication Main function, where a single replication of a single scenario is executed.

The main function executed in the generated code is `run_replication`. The function code is shown in Listing 6.2. The steps followed to simulate a single replication of a scenario are as follows:

1. Create variable dictionary and read scenario values.
2. Initialise solver and evaluation function.
3. Generate stochastic and custom inputs for simulation.
4. Determine initial values for simulation.
5. Do simulation and store results.
6. Write results.

```
import *

def generate_stochastic_input(file, replication_number):
    ...
    return stochastic_input

def generate_custom_input(file, scenario_number, replication_number):
    ...
    return custom_input

def initialise_solver(tolerance):
    ...
    return solver

def initialise_function():
    ...
    return function

def initial_value(variable_dict, stochastic_input, custom_input):
    ...
    return variable_dict

def mass_balance(variable_dict, solver, function, stochastic_input, custom_input, t, dt):
    ...
    return variable_dict

def run_replication(arguments):
    ...
    return None

if __name__ == '__main__':
    run_replication(arguments)
```

Listing 6.1: Generated code layout

```
def run_replication(arguments):
    # Create variable dictionary and read scenario values
    variable_dict = read_start_value(file)
    scenario_dict = read_scenario(scenario_number)
    variable_dict.update(scenario_dict)

    # Initialise solver and evaluation function
    solver = initialise_solver(tolerance)
    function = initialise_function()

    # Generate stochastic and custom inputs for simulation
    stochastic_input = generate_stochastic_input(file, replication_number)
    custom_input = generate_custom_input(file, scenario_number, replication_number)

    # Determine initial values for simulation
    variable_dict = initial_value(variable_dict, stochastic_input, custom_input)

    # Do simulation and store results
    result_storage = empty
    for t in range(number_of_hours):
        variable_dict = mass_balance(variable_dict, solver, function, stochastic_input,
                                     custom_input, t, dt)
        result_storage.append(variable_dict)

    # Write results
    result_storage.write_results(file)
    return None
```

Listing 6.2: `run_replication` function

6.4 Solver and function

The main tool used for solving and function evaluation is the `CasADi` Python package. `CasADi` is an open-source tool, with the main function of providing automatic differentiation. It is written entirely in self-contained C++ and is often used for non-linear optimisation and optimal control problems. Models are communicated using its symbolic framework (Andersson *et al*, 2019). `CasADi` was first used in the Amoss 1.0 by Whyte (2018) and proved a valuable addition, especially for Newton root finding.

6.4.1 Function

A basic use of the `CasADi` tool is simple evaluation of explicitly ordered sets of equations. The solution of these systems translates to evaluation of a multi-input multi-output function. The `function` class in `CasADi` can be used for function evaluation. Functions are created at runtime and can be numerically evaluated afterwards. With functions being pre-compiled to C++, function evaluation throughout simulation is expected to be very fast.

6.4.2 Root finding

The main advantage of the `CasADi` tool is the automatic differentiation feature. Automatic differentiation can be described as an effective method for determining derivative information of a computer program. It is discussed in more detail in Section 2.7. `CasADi` provides the `rootfinder` class for non-linear root finding. A very fast Newton solver, which heuristically makes use of forward and reverse-mode automatic differentiation to determine Jacobian information is provided. Similar to `function` objects, `rootfinder` objects are pre-compiled to C++. It is expected that using the pre-compiled automatic differentiation Newton solver for iterative root finding of implicitly ordered problems will be the greatest speed improvement provided by using the `CasADi` tool.

6.4.3 MINLP

The mixed-integer non-linear programming (MINLP) problems in Amoss caused by the optimal flow allocation problem are complex and difficult to solve. `CasADi` provides the `nlpsol` class for solution of non-linear programming problems. The `nlpsol` class itself does not contain an MINLP solver, but the open-source `Bonmin` solver is available as a plugin. `Bonmin`

(Basic Open-source Nonlinear Mixed INteger programming) is a solver designed for solving general MINLP and is part of the COIN-OR project. When the optimisation objective is convex, the `Bonmin` solution is exact. However, if the objective is non-convex, a good heuristic solution can be obtained. The MINLP problems in Amoss stem from disjunctive optimisation problems and are expected to have non-convex objectives, due to the reformulation methods. The recommended method for solution of non-convex problems in `Bonmin` is the branch-and-bound method. The `nlpsol` object is pre-compiled, but due to the external solver being called at each use, no compilation speed advantage is expected.

6.5 Parallel processing

It is typical of Monte Carlo type simulations to be embarrassingly parallel in nature. In the case of Amoss simulations, replications of simulation scenarios are entirely independent of one another. That is to say that no common data is accessed and that no communication is necessary between replications, during simulation. This creates an opportunity for parallel processing. Parallel processing utilises the various processors of a computer to run independent instructions concurrently. Parallel processing is discussed in more detail in Section 2.8.

In Amoss 1.0, the simulation scenarios were parallelised for parallel processing. Parallel processing was implemented using `Celery`. Only parallelising the simulation scenarios and not the replications of the scenarios, limits the effect of parallel process. Additionally, `Celery` no longer provides support for Windows (Celery, 2018). As an improvement, in Amoss 2.0 replications of scenarios are parallelised for parallel processing. Parallel processing is now accomplished by spawning subprocesses using the `subprocess` module in the Python standard library.

Given s scenarios and r replications, the total number of runs to do is $s \times r$. On a machine with n virtual cores, n number of scenario-replication subprocesses are spawned simultaneously, each with its own global interpreter lock (GIL) executing in its own terminal. After completion of n parallel replications, n new replications are spawned. This approach works well, because there is little time difference between different replications. If $s \times r$ is perfectly divisible by n, the simulation time is theoretically divided by n.

Part III

Results and conclusions

CHAPTER 7

RESULTS AND DISCUSSION

7.1 Intermediate result storage

During simulation, it is necessary to store simulation data, before writing the results to hard disk. With many simulations this is not a practical problem, but Amoss simulations often contain hundreds of variables, which need to be stored every hour for thousands of hours. The sheer amount of data generated requires effective storage of data in memory before writing the simulation results.

A method which could be used, is intermittently writing results and clearing the memory. This can come at the sacrifice of simulation time, because conversion of data to a writable format can take significant time. The optimal method of simulation with regard to simulation time is to store data in memory throughout the simulation for a single replication, convert the results to a writable format and write to disk only once.

Thus, it is important to select a data type for intermittent storage. This data type should be memory efficient, fast to write to and quick to convert to a writable data type. A two-dimensional spatial requirement is also necessary, because of the time and variable axes. The data types to investigate are:

- Two-dimensional `numpy` array. The C contiguous (row major) and F contiguous (column major) orderings will be considered.

- A list of Python lists.
- A list of Python tuples.
- A list of Python dictionaries.

During simulation, variable values are most easily represented as a dictionary. The dictionary effectively represents the state of all variables. This information can be added to the selected data type for intermittent storage. To test memory usage, write time and conversion time, 500 variables were randomly generated every hour for 70 000 hours. The variable data was written every hour to each data type, to mimic how data will be written during simulation.

The memory usage of each of the data types was determined using the `pympler` tool. The memory usage of the various data types is shown in Figure 7.1. In addition to memory usage, it is necessary to determine how much time is spent on writing data to the relevant data type. The time spent writing data to the relevant data type is shown in Figure 7.2.

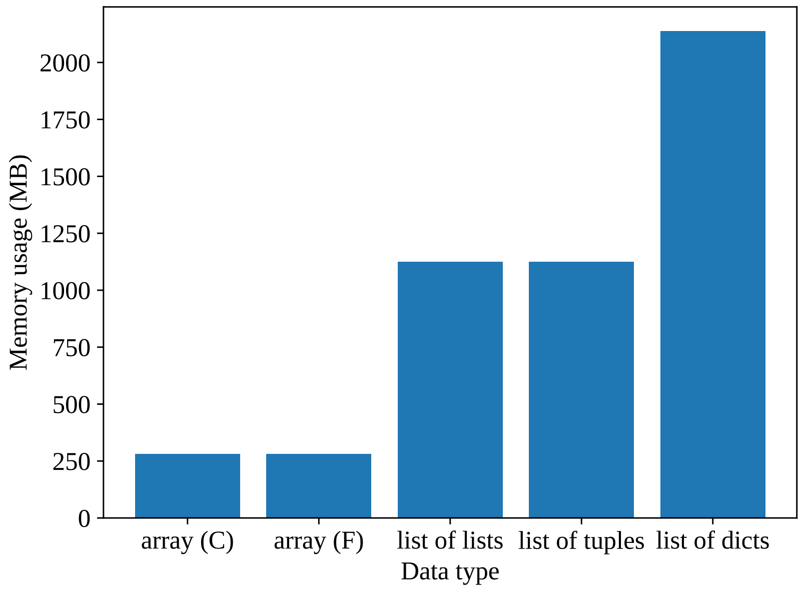


Figure 7.1: Memory usage of data types

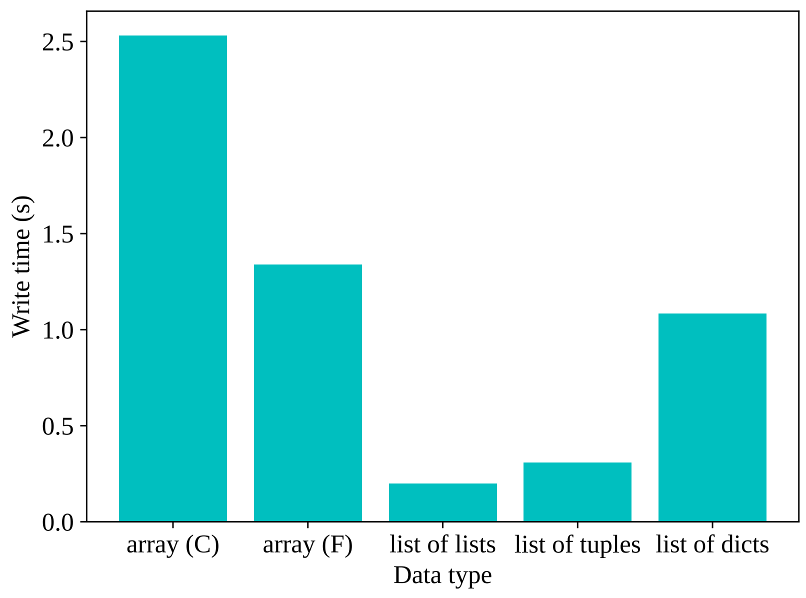


Figure 7.2: Write time to data types

An important factor to take into account is how long each intermediate data format takes to be converted to a writable format. The data type selected for file writing is the widely used `Pandas` dataframe object. When constructing dataframes using `Pandas`, all non-array iterable data formats are internally converted to `numpy` arrays. The conversion time to `numpy` arrays is shown in Figure 7.3. Final conversion from `numpy` arrays to `Pandas` dataframes is very fast and is illustrated in Figure 7.4.

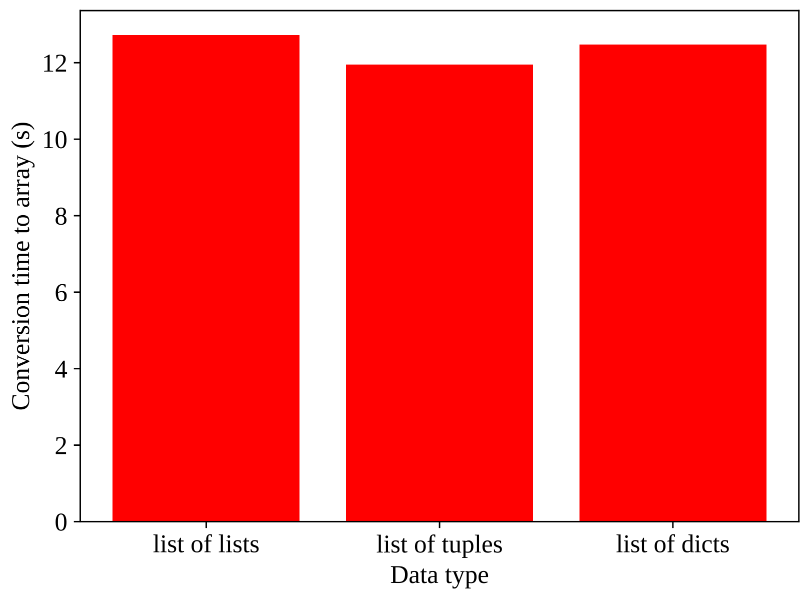


Figure 7.3: Conversion time from data types to `numpy` array

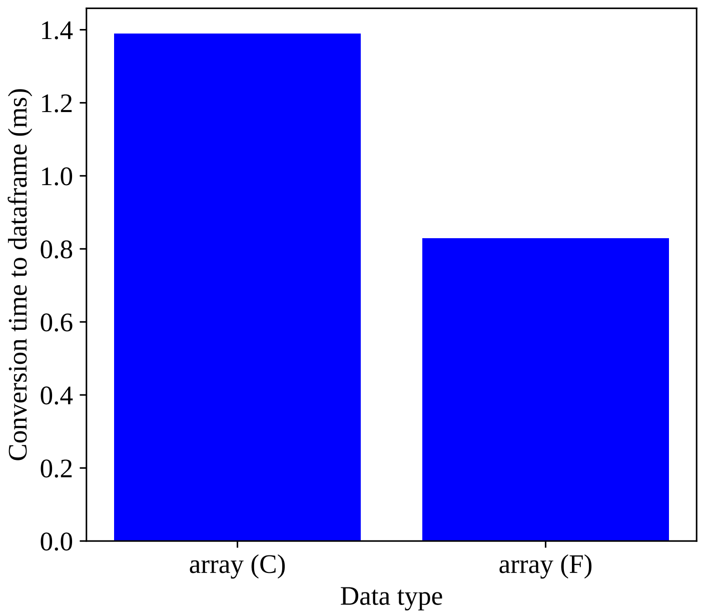


Figure 7.4: Conversion time from `numpy` array to `Pandas` dataframe

From Figure 7.1 it is clear that `numpy` arrays take up much less memory than the standard Python data types. The `numpy` arrays are approximately 5 times smaller than the list of lists and list of tuples and about 10 times smaller than the list of dicts. This result makes sense, because `numpy` arrays are generated in C and are data type specific. Note that parallel processing will be implemented for simulation, which makes memory usage very significant. As an example take a simulation on a machine with 8 processors, where 8 simulations are run in parallel. If 8 `numpy` arrays of 70 000 $\times$ 500 are generated, about 2.2 GB of memory will be used up. Thus, it is clearly impractical to use the other data types, when considering memory usage.

From Figure 7.2 it is clear that write time to `numpy` arrays, especially the C ordering, is slower than the other formats. The calculations involved with simulation are expected to be substantially slower than the write times and more than likely will render these differences insignificant. More significantly than the write times, there is no conversion time when constructing a `Pandas` dataframe from a `numpy` array. Conversion time to `numpy` arrays is slow, as illustrated in Figure 7.3. The absence of conversion time, combined with the memory efficiency, makes the `numpy` array data type the clear best choice for intermediate data storage. Considering the array orderings, it is clear that the F ordering is superior in terms of write time and conversion time to a `Pandas` dataframe.

7.2 Result writing

During simulation, data is stored to an intermediate storage data type as discussed in Section 7.1. After simulation, the intermediate data storage is converted to a writable data type for results writing. It is important that results be written as quickly as possible and take up as little space as possible on disk. Because of the large amount of result data generated, write speed and result storage size can become a problem.

The `Pandas` dataframe object was selected as the writable data type. This is mainly due to `Pandas` widespread use, various available file formats and compression filters. It is necessary to determine is a suitable file format for result writing. The file formats selected for investigation are Feather, Parquet, HDF5 and CSV. The file formats are discussed in more detail in Section 2.9.

Compression should be considered. Because file writing occurs between replications, writing results as quickly as possible is priority. Thus, writing time is more important than high compression ratios. For that reason, only the Snappy compression method will be considered. Snappy compression is discussed in Section 2.9.5 and is focused on fast reasonable compression.

To test the write time, read time and storage file size of each of the file formats, a 70 000 $\times$ 500 `Pandas` dataframe was generated, written to each file format and read. This should be a sufficient representation of simulation result storage and reading. The write time of the various file formats is shown in Figure 7.5. The file size of the various file formats is shown in Figure 7.6. The read time of the various file formats is shown in Figure 7.7.

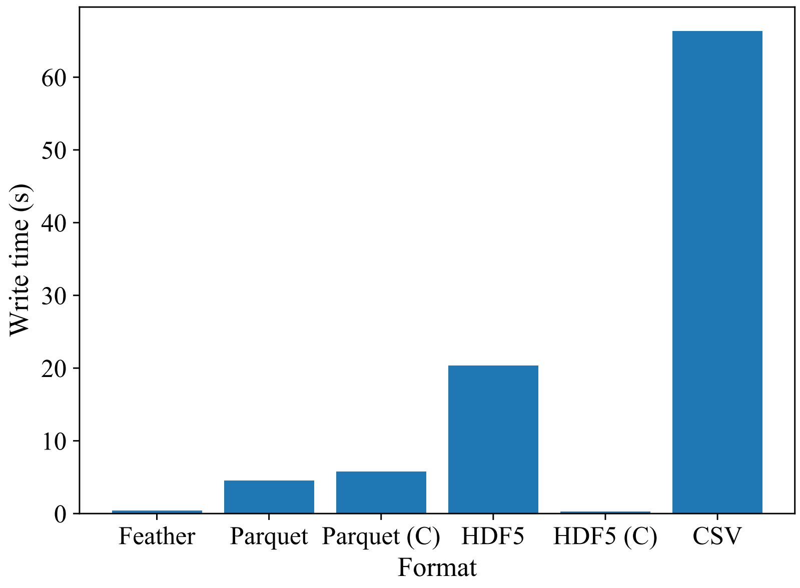


Figure 7.5: Write time of file formats. Snappy compression is indicated by C.

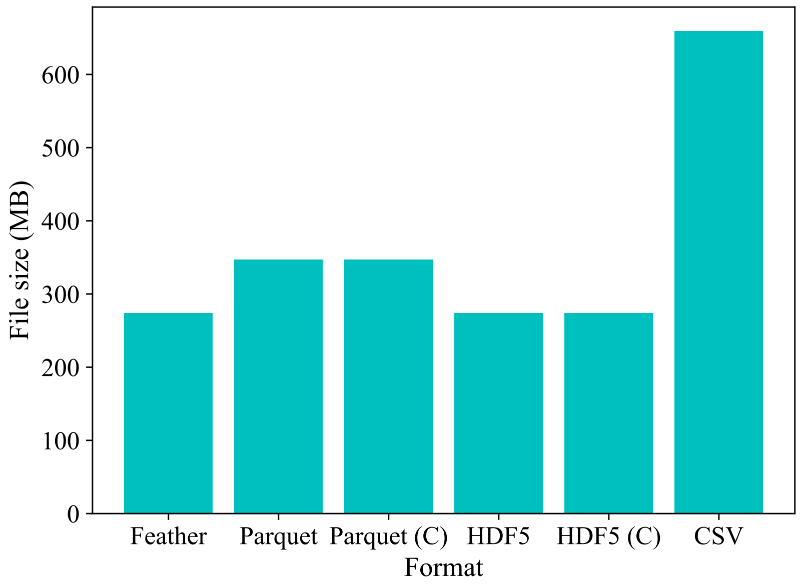


Figure 7.6: File size of file formats. Snappy compression is indicated by C.

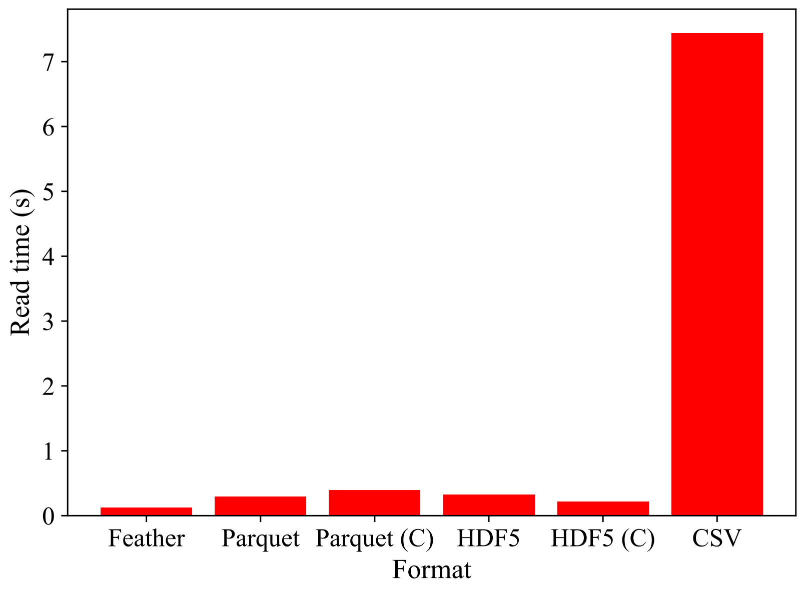


Figure 7.7: Read time of file formats. Snappy compression is indicated by C.

The results from Figure 7.5, Figure 7.6 and Figure 7.7 clearly illustrate the inferiority of the CSV file format for result storage of large data sets, with it being the worst file format in all categories. From Figure 7.5, it is clear that the fastest file formats to write to are feather and the Snappy compressed HDF5 format, with the feather file writing in 0.41 s and the compressed HDF5 file writing in 0.27 s. Figure 7.7 also suggests that these two formats are the fastest to read from, with the feather file reading in 0.12 s and the compressed HDF5 file reading in 0.22 s. Figure 7.6 shows that the smallest file sizes belong to the HDF5 files and feather, with the files being approximately 273 MB in size.

All factors suggest that the Snappy compressed HDF5 format and feather file formats are the best file formats for result writing. It is important to note that the feather format is new and slightly unstable for long-term data storage. The feather format also does not yet have accommodation for compression. Because of the instability of the feather file format and faster write time, the Snappy compressed HDF5 file format was selected as the file format for result writing. However, in future it is expected that this evaluation can change. If compression can be introduced and the long-term storage stability could be improved, the feather format could prove superior to the compressed HDF5 format.

7.3 Benchmarks

With `numpy` arrays (F ordering) as the intermediate data type and Snappy compressed HDF5 as the file format for result writing, the benchmarks in Chapter 3.4 were run for the test processes. The test processes are discussed in Chapter 3.2. The simulations were run for 70 000 hours, with a step size of 1 hour. The machine used for benchmarking has the hardware and software specifications shown in Table 3.1. Because the machine has 8 virtual cores, 8 replications were run in parallel.

7.3.1 Code generation time

The code generation benchmark relates to the *reduction in development time* deliverable. Code generation only constitutes a small portion of development time, because model development prior to code generation is expected to take in the order of a week. Thus, considering that during model development code generation should only occur once, very high code generation speed is not required. That being said, in event of plant or operating instruction changes, fast code generation can be advantageous.

The code generation time of the test processes are shown in Figure 7.8. The average code generation time is approximately 8 s, with the longest code generation taking 37 s. Model development prior to code generation takes in the order of a week. Considering that code generation only occurs once, it is clear that code generation is not a bottleneck.

From Figure 7.8, it can be observed that code generation of test process 8 and test process 12 is substantially slower than that of the other test processes. Various factors could be the cause. It is suspected that the model equations are difficult to solve symbolically (using `SymPy`). This would result in a bottleneck when determining the safe equation-variable pairings during equation ordering.

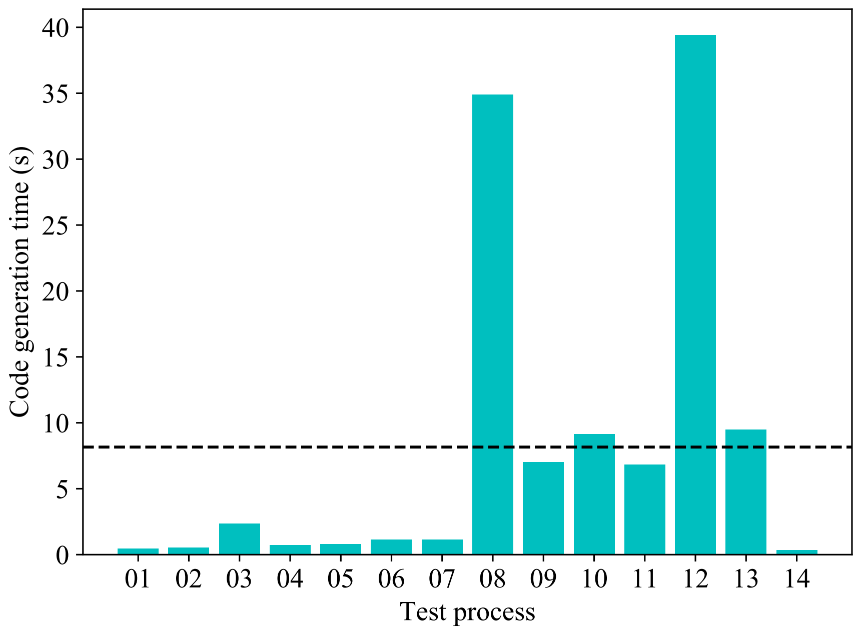


Figure 7.8: Benchmark code generation time

7.3.2 Simulation time

Simulation time relates to the *fast simulation time* deliverable. Fast simulation time is critical. After code generation has been done, the next step is to do Monte Carlo analysis on the various simulation scenarios. Monte Carlo analysis, as discussed in Section 2.2.3, depends on the strong law of large numbers, which implies that more simulation replications will lead to more reliable analysis results. Thus, faster simulation time of a single replication allows more replications to be run and better analysis to be done. Additionally, more simulation scenarios can be run.

The simulation time of the test processes are shown in Figure 7.9. Take note that test process 14 was timed, but not included in the benchmark. Its slow simulation time of 278.3 s/rep is two orders of magnitude slower than the other test processes. This illustrates the impractical slow time obtained by solving the allocation problem using the MINLP approach. The heuristic allocation formulation should always be used when possible.

From Figure 7.9, the simulation time is low. The average simulation time is 2.2 s/rep, with the longest simulations taking about 4.2 s/rep. Considering that for the implicit ordering problems, every hour of 70 000 hours an iterative Newton root finding step is implemented, this is an exceptional result. For all test processes, the basic validations discussed in Section 3.5 were met.

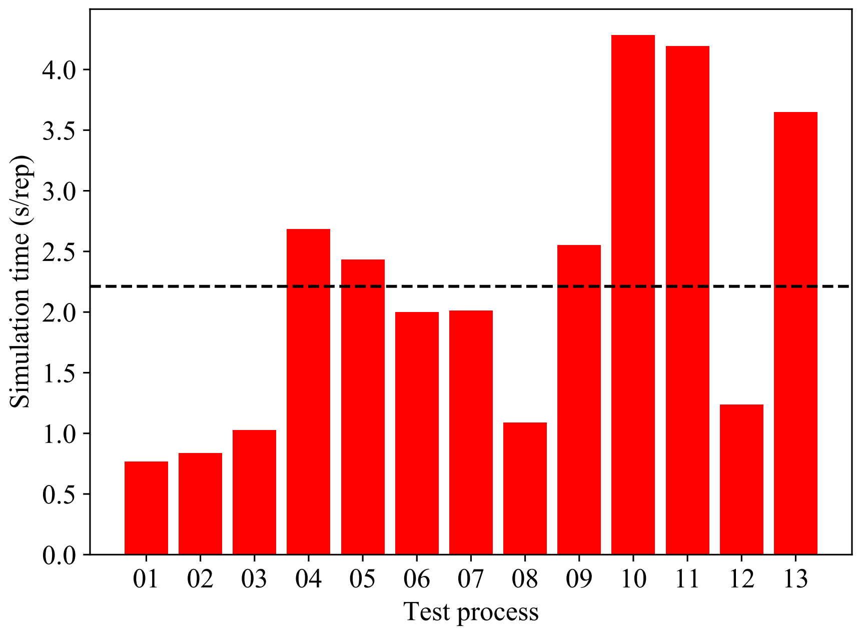


Figure 7.9: Benchmark simulation time

7.3.3 Linear scalability

The *linear scalability* of the simulation time is an important deliverable. Linear scaling implies that there is at most a linear relationship between the simulation time (s/rep) and the number of equations. In other words, if the number of equations of a model doubles, the simulation time should not increase by a factor of more than two. This scaling is important, because inefficient scaling will make simulation of larger systems infeasible.

Simulation time (s/rep) as a function of the number of equations (N) of each test process is plotted in Figure 7.10. To test linearity, the linear trend line $y = 0.1N$ was plotted. There are no points above the linear trend line. It is clear from the plotted points and the linear trend line, that the simulation time scaling is of a lower order than 1. This is an excellent result. To determine an approximate order for the scaling, a non-linear trend line with an unknown power was fitted. The non-linear trend line $y = 0.46N^{0.38}$ was obtained. Thus, the simulation time scaling is of approximately order 0.4.

The number of equations in a system is not an absolute measure of model complexity. The limiting step for simulation is root finding. Thus, the main factor in determining simulation time is not the number of equations, but rather the number of equations to simultaneously solve during root finding. This is supported by Figure 7.11, which illustrates the simulation time (s/rep) as a function of the number of equations to simultaneously solve. Although the data points are not well spread out, there is a clear increase in simulation time as the number of equations to root find increases. The reason for the decent scaling of simulation time is the success of the equation ordering method. Because large systems of equations are reduced to only have a few equations to solve simultaneously, adding more equations to a model does not necessarily increase simulation time substantially.

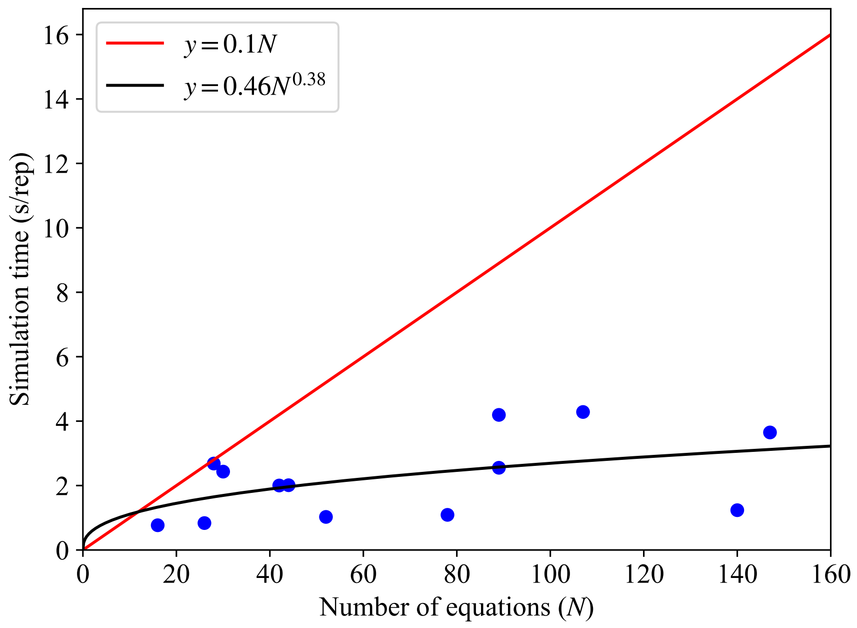


Figure 7.10: Benchmark linear scalability

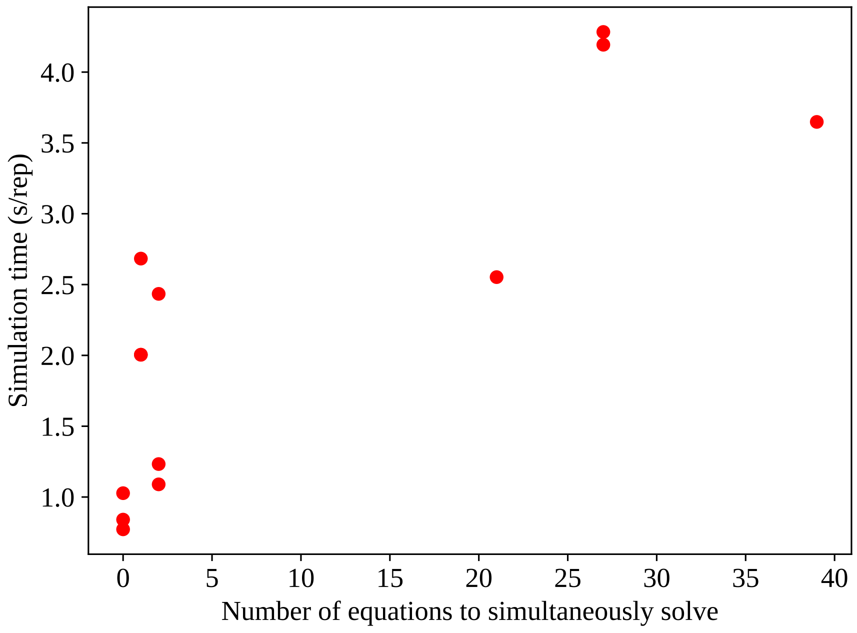


Figure 7.11: Benchmark simulation time versus number of equations to simultaneously solve

7.4 Industrial process

Amoss 2.0 was also tested using the partially built industrial process, discussed in Section 3.3. The results are positive and can be summarised as follows:

Equation ordering The system of 370 equations was reduced to 25 equations to simultaneously solve. This is a good result and a significant reduction. Taking into account that heuristic tearing methods were used, this is more than acceptable. The incidence matrices illustrating the equation ordering method are shown in Appendix C.

Code generation Code generation was completed in approximately 30 s, which is quite fast. Taking into account that the code generation only occurs once, this is a good result.

Simulation time The most impressive result is the newly achieved simulation time. In Whyte (2018) the partially built industrial process simulated in approximately 24 min/rep. The fully built Sasol process simulated in about 1.7 min/rep. The current implementation simulates the partially built industrial process in 3.8 s/rep or 0.063 min/rep. This is more than two orders of magnitude faster than Amoss 1.0 in Whyte (2018) and approximately 27 times faster than the Sasol simulation at the same time. It should be noted that in the meantime, the Sasol model has become more complex and simulates more slowly.

Result validation The simulation results contain no negative values for steams or component rates. Zero division is not encountered and no plant constraints are violated. This suggests that the partially built model results are valid.

The benchmark results illustrate the progress made from Amoss 1.0 to Amoss 2.0. It is recommended that the industrial process build be completed and updated to the version currently in use at Sasol. This would allow more elaborate simulation and result comparison.

CHAPTER 8

CONCLUSIONS AND RECOMMENDATIONS

8.1 Conclusions

The Amoss 2.0 simulation engine is a great improvement over that of Amoss 1.0, which is supported by the benchmark results of Chapter 7. This is a direct consequence of revisiting the simulation factors, discussed in Section 1.4. The changes made are as follows:

Equation ordering A custom equation ordering method was implemented. This method starts by pre-conditioning the system of equations to only consist of continuous and conditional equations. Afterwards, safe and unsafe variable matchings are determined using interval arithmetic. Weighted maximum matching is then applied to determine the most numerically stable equation-variable matching. The matched system is decomposed using Dulmage-Mendelsohn decomposition and torn using Cellier's tearing. Finally, the system is transformed to the bordered block lower triangular form used for code generation.

Simulation approach Various aspects of the existing simulation approach were investigated. A major change was made to the solution of the implicitly ordered problem. Because of the root finding difficulties, a fixed-point iterative Newton solve technique was implemented for solution of the combined continuous-conditional system. The optimal flow allocation problem was reinvestigated. The heuristic allocation method, which has been applied to the forward direction, was extended for use when recycles are encountered. The general case of the optimal flow allocation problem was reformulated. With regard to simulation inputs, accommodation was made for custom inputs.

Code generation Code generation was reimplemented to accommodate the new simulation approach. The simulation code template was generated such that all replications and scenarios can be run using the same code. The `CasADi` tool was selected for function evaluation, Newton root finding and MINLP solution. The evaluation functions and solvers are pre-compiled to C++ and numerically evaluated as simulation progresses. Parallel processing using subprocesses was implemented, to exploit the embarrassingly parallel nature of the replications and scenarios.

Result storage Storing simulation results intermediately to `numpy` arrays (F ordering) was implemented. Writing completed simulation results as HDF5 files with Snappy compression was implemented.

The goal of revisiting the above factors was to address the Amoss 2.0 deliverables in Section 1.4. All deliverables have been sufficiently addressed. The progress with regard to each of the deliverables are as follows:

Fast simulation time The simulation time of the tool was reduced significantly, with the average test process simulation time being 2.2 s/rep and the partially built industrial process simulating at 3.8 s/rep. The partially built industrial process simulation time was reduced by two orders of magnitude from Amoss 1.0 and is 27 times faster than the Sasol developed simulation. This great increase in simulation speed is a result of the improvements made to the equation ordering method, the fixed-point iterative root finding method for implicitly ordered systems, the `numpy` array data type being used for intermediate data storage, using the HDF5 file format for result writing, Snappy compression, parallel processing using subprocesses and the speed of the `CasADi` Newton solver and function evaluations.

Numerical stability The numerical stability of simulations using the tool was greatly improved. The test processes and partially built industrial process simulations are free of invalid values and zero division throughout all tested simulation runs. This is due to the equation ordering method using interval arithmetic to determine safe and unsafe equation-variable pairings, followed by weighted maximum matching and the fixed-point iterative root finding method used to simulation implicitly ordered systems.

Acceptable accuracy The simulation accuracy of the tool was increased by improving numerical stability. Because zero division is no longer encountered, reasonable simulation values are encountered. The implicitly ordered system values are root solved using a Newton strategy with an absolute tolerance of 1×10^{-6}. This ensures acceptable simulation accuracy.

Linear scalability The simulation time of the tool scales very well. The simulation time does not only scale linearly with the number of equations, but scales at approximately an order of 0.4. This is a direct result of the success of the custom equation ordering method.

Reduction in development time Model development time using the tool is very low. The tool itself has a slight learning curve and it is expected that a new user would take approximately a week to develop a model, with no prior experience of using the tool. This development time will of course decrease as the user becomes more accustomed the tool's use. The average code generation time is only 8 s, which is much lower than model development time. The low code generation time means that flow sheet or operating instruction changes can quickly be implemented and simulated.

Independent package The equation ordering, code generation and simulation parts of the tool are now independent. The external equation ordering repository, `sdopt-tearing`, was removed from the code base and custom equation ordering code developed. Exclusive use is made of the Python standard library and other widely-used open source packages.

8.2 Recommendations

8.2.1 Near future

Legacy model building For the first time, since the start of the Amoss project, the simulation engine is numerically stable and simulation is fast. Additionally, it is possible to do almost any simulation by using the stochastic and custom inputs. Taking these factors into account, it is important that time be spent rebuilding the legacy models in Amoss 2.0. These simulations are expected to be substantially faster and more stable than the Sasol developed counterparts. Additionally, simulation results could be compared to industrial results.

Independent package The simulation engine of Amoss 2.0 is independent of non-Python external packages. However, the other sections of the tool have an external dependency on OpenModelica, Atom and Microsoft Excel. Work should be done to remove all external dependencies from the Amoss project. The concurrent projects are and should continue working towards making Amoss a stand-alone package.

Additional stochastic input accommodation In addition to basic time-independent stochastic inputs using distribution data, different types of stochastic inputs are made use of by the Sasol simulations. An example is time-dependent failure stochastic input. For instance a stochastic input can be generated from plant information on mean time to failure and mean time to repair. It is currently possible to manually include this type of input as a custom input, but machinery should be included to automatically generate these inputs.

Result analysis In the current state of the tool, only model development and simulation is accommodated. This creates the need to convert the generated HDF5 simulation results to CSV for separate analysis on the Sasol operation research team's side. This mitigates many of the advantages gained from fast simulation and result writing. Monte Carlo analysis, result graphing and user feedback should be provided in the Amoss tool.

8.2.2 Future

Desired incidence matrix form In the developed equation ordering method, the bordered block lower triangular incidence matrix form was used for code generation. This form effectively divides the entire system into a single explicit region and single simultaneous solution region. The main reason for its selection is its simplicity and the fact that optimisation can easily be accommodated. However, it is expected that the spiked lower triangular incidence matrix form will simulate faster and scale better. This form comes from tearing the blocks of the Dulmage-Mendelsohn decomposition. It effectively breaks up the system into sequentially solvable sub-problems, each with its own explicit region and simultaneous solution region. Thus, instead of having to solve over the entire system, smaller sub-problems are sequentially solved. This is more difficult to implement for code generation.

C code generation A decision was made in Amoss 1.0 and Amoss 2.0 to generate simulation code in the Python programming language. There were many considerations, with the main factor being ease of development. However, a major drawback of Python code is execution speed. It is an interpreted language and consequently runs slower than a compiled language. A large speed improvement is possible by rather generating simulation code in a compiled language, such as C. CasADi can still be used for root finding and supports both C code generation and a C++ API.

Data distribution In the event that the tool becomes frequently used by multiple users, it becomes practical to have a decent data distribution platform in place. As an example, consider the Sasol operation research team. It is important that data be distributed among team members. If the team members are assigned different models, it is unnecessary to develop and simulate on each computer to obtain model data and simulation results. Ideally, model data and simulation results should be actively shared between team members.

Parallel processing over a cluster Stochastic simulations of the Amoss tool are embarrassingly parallel, with replication and scenarios being independent. This property was used to implement parallel processing on a single computer or local cluster. However, it is possible to extend this parallel processing to a cluster of computers. The more processors are available in the cluster, the more independent replications can be run simultaneously. This will even further improve simulation time. However, it is not expected that it will be easy to implement based on the networking restrictions in Sasol.

BIBLIOGRAPHY

Allen, M and Wilkinson, B (2005) *Parallel Programming, Techniques and Applications Using Networked Workstations and Parallel Computers*.

Andersson, JAE, Gillis, J, Horn, G, Rawlings, JB and Diehl, M (2019) "CasADi – A software framework for nonlinear optimization and optimal control" *Mathematical Programming Computation*, *11*, (1): 1–36 DOI: `10.1007/s12532-018-0139-4`.

Apache Software Foundation (2019) *Apache Parquet* URL: `parquet.apache.org` (visited on 01/05/2020).

Asanovic, K, Bodik, R, Catanzaro, BC, Gebis, JJ, Husbands, P, Keutzer, K, Patterson, DA, Plishker, WL, Shalf, J, Williams, SW, *et al* (2006) *The landscape of parallel computing research: A view from berkeley* tech. rep. Technical Report UCB/EECS-2006-183, EECS Department, University of California.

Asratian, AS, Denley, TM and Häggkvist, R (1998) *Bipartite graphs and their applications*, vol. 131 Cambridge university press.

Baharev, A, Schichl, H and Neumaier, A (2016a) "Decomposition Methods for Solving Sparse Nonlinear Systems of Equations" URL: `http://reliablecomputing.eu/baharev_tearing_survey.pdf`.

Baharev, A, Schichl, H and Neumaier, A (2016b) "Ordering matrices to bordered lower triangular form with minimal border width" URL: `http://reliablecomputing.eu/baharev_tearing_exact_algorithm.pdf`.

Baharev, A (2017a) *safe-elimination* https://github.com/baharev/safe-eliminations.

Baharev, A (2017b) *sdopt-tearing* https://github.com/baharev/sdopt-tearing.

Baharev, A, Domes, F and Neumaier, A (2017) "A robust approach for finding all well-separated solutions of sparse systems of nonlinear equations" *Numerical Algorithms*, *76*, (1): 163–189.

Bilodeau, C (2015) *Disjunctive inequalities* URL: https://optimization.mccormick.northwestern.edu/index.php/Disjunctive_inequalities (visited on 07/11/2019).

Bradie, B (2006) *A friendly introduction to numerical analysis*, Pearson Education India.

Celery (2018) *Celery* https://github.com/celery/celery.

Cellier, FE and Kofman, E (2006) *Continuous system simulation*, Springer Science & Business Media chap. 1.

Davis, TA (2006) *Direct methods for sparse linear systems*, vol. 2 Siam.

Devroye, L (1986) *Nonuniform random variate generation*, vol. 1 Springer.

Dlouhỳ, M, Fábry, J and Kuncová, M (2005) *Simulace pro ekonomy*, Oeconomica.

Duffy, D and Kienitz, J (2009) "Monte Carlo Frameworks" *Building Customisable High Performance C++ Applications John Wiley and Sons Chichester*,

Durrett, R (1996) *Stochastic calculus: a practical introduction*, CRC press: pp. 245–250.

Fredman, ML and Tarjan, RE (1987) "Fibonacci heaps and their uses in improved network optimization algorithms" *Journal of the ACM (JACM)*, *34*, (3): 596–615.

Google (2019) *snappy* URL: https://google.github.io/snappy/ (visited on 01/05/2019).

Gottlieb, A and Almasi, G (1989) *Highly parallel computing*, Benjamin/Cummings Redwood City, CA.

Griewank, A and Walther, A (2008) *Evaluating derivatives: principles and techniques of algorithmic differentiation*, vol. 105 Siam.

Gross, JL and Yellen, J (2005) *Graph theory and its applications*, Chapman and Hall/CRC.

Jonker, R and Volgenant, A (1987) “A shortest augmenting path algorithm for dense and sparse linear assignment problems” *Computing*, *38*, (4): 325–340.

Kalos, MH and Whitlock, PA (2009) *Monte carlo methods*, John Wiley & Sons.

Kelley, CT (2003) *Solving nonlinear equations with Newton’s method*, vol. 1 Siam.

Korn, R, Korn, E and Kroisandt, G (2010) *Monte Carlo methods and models in finance and insurance*, CRC press.

Kroese, DP, Brereton, T, Taimre, T and Botev, ZI (2014) “Why the Monte Carlo method is so important today” *Wiley Interdisciplinary Reviews: Computational Statistics*, *6*, (6): 386–392.

Lee, S and Grossmann, IE (2000) “New algorithms for nonlinear generalized disjunctive programming” *Computers & Chemical Engineering*, *24*, (9-10): 2125–2141.

LeVeque, RJ (1998) “Finite difference methods for differential equations” *Draft version for use in AMath*, *585*, (6): 112.

Lincoln, R (2012) *CSparse.py* `https://github.com/rwl/CSparse.py`.

Meyer, M, Hylton, R, Fisher, M, van der Merwe, A, Streicher, G, van Rensburg, JJ, van den Berg, H, Dreyer, E, Joubert, J, Bonthuys, G, Rossouw, R, Louw, W, van Deventer, L, Wykes, C and Cawood, E (2011) “Innovative Decision Support in a Petrochemical Production Environment” *Interfaces*, *41*, 79–92.

Micali, S and Vazirani, VV (1980) “An $O(V^2E)$ algorithm for finding maximum matching in general graphs” in: *21st Annual Symposium on Foundations of Computer Science (sfcs 1980)* IEEE: pp. 17–27.

Montgomery, DC and Runger, GC (2003) *Applied statistics and probability for engineers*, 3rd ed. John Wiley & Sons.

Mooney, CZ (1997) *Monte carlo simulation*, vol. 116 Sage Publications.

Otter, M, Elmqvist, H and Mattsson, SE (1999) “Modeling of mixed continuous/discrete systems in modelica” *Tech. Rep.*

Picchini, U (2007) *SDE Toolbox – Simulation and Estimation of Stochastic Differential Equations with Matlab* version 1.4.1 Matlab URL: `http://sdetoolbox.sourceforge.net/manual.pdf`.

Shafranovich, Y (2005) *Common Format and MIME Type for CSV Files* URL: `https://www.ietf.org/rfc/rfc4180.txt#page-1` (visited on 01/05/2020).

SIMUL8 Corporation (2017) *Simul8 simulation software - for visual process simulation modeling* URL: `https://www.simul8.com/` (visited on 12/2017).

Süli, E and Mayers, DF (2003) *An introduction to numerical analysis*, Cambridge university press.

Täuber, P, Ochel, L, Braun, W and Bachmann, B (2014) "Practical realization and adaptation of Cellier's tearing method" in: *Proceedings of the 6th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools* ACM: pp. 11–19.

The AnyLogic Company (2017) *Simulation modeling software tools & solutions for business anylogic* URL: `https://www.anylogic.com/` (visited on 12/2017).

The HDF Group (2019) *The HDF5 Library File Format* URL: `https://www.hdfgroup.org/solutions/hdf5/` (visited on 01/05/2020).

Whyte, E (2018) "AMOSS: Automatic Modeling Operations using Stochastic Simulation" University of Pretoria URL: `https://repository.up.ac.za/handle/2263/71047`.

Wickham, H (2016) *Feather: A Fast On-Disk Format for Data Frames for R and Python, powered by Apache Arrow* URL: `https://blog.rstudio.com/2016/03/29/feather/` (visited on 01/05/2020).

Williams, HP (1985) *Model building in mathematical programming*, John Wiley & Sons.

APPENDIX A

ALLOCATE ALGORITHM

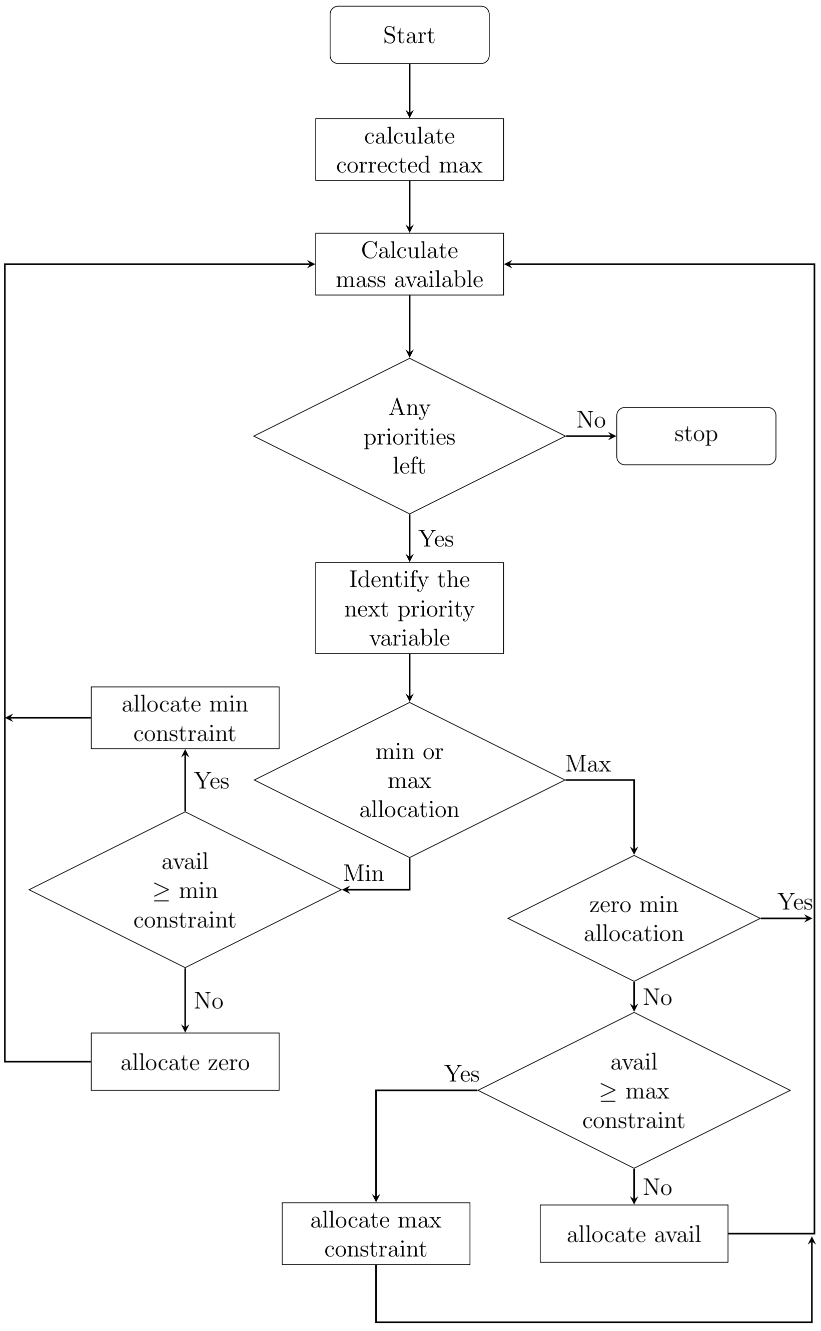


Figure A.1: Algorithm followed to generate equations from allocate instruction (Whyte, 2018)

APPENDIX B

MODEL DESCRIPTION

```
1. Details
  Type: Explicit
  Number of equations/variables: 45

2. Combined equations
  S_avail = S1_comp1 + S2_comp1
  allocate0_avail = S_avail
  allocate0_min_avail_p0 = allocate0_avail
  S4_comp1_max = 40
  S3_comp1_max = 30
  S4_comp1_min = 20
  S3_comp1_min = 10
  S4_min_p = 1
  S4_comp1_min_p1 = S4_min_p == 1
  S4_comp1_min_p0 = S4_min_p == 0
  S3_max_p = 1
  S3_comp1_max_p1 = S3_max_p == 1
  S3_comp1_max_p0 = S3_max_p == 0
  S4_max_p = 0
  S4_comp1_max_p1 = S4_max_p == 1
  S4_comp1_max_p0 = S4_max_p == 0
  S4_comp1_cor_max = S4_comp1_max - S4_comp1_min
  S3_min_p = 0
  S3_comp1_min_p1 = S3_min_p == 1
```

```
allocate0_min_p1 = S3_comp1_min*S3_comp1_min_p1 + S4_comp1_min*S4_comp1_min_p1
S3_comp1_min_p0 = S3_min_p == 0
allocate0_min_p0 = S3_comp1_min*S3_comp1_min_p0 + S4_comp1_min*S4_comp1_min_p0
allocate0_min_p0_if = allocate0_min_p0 <= allocate0_min_avail_p0
allocate0_min_avail_p1 = allocate0_avail - allocate0_min_p0
allocate0_min_p1_if = allocate0_min_p1 <= allocate0_min_avail_p1
allocate0_min_al_1 = allocate0_min_p1*allocate0_min_p1_if
allocate0_min_al_0 = allocate0_min_p0*allocate0_min_p0_if
allocate0_avail_after_min = allocate0_avail - allocate0_min_al_0 - allocate0_min_al_1
allocate0_max_avail_p0 = allocate0_avail_after_min
S3_comp1_cor_max = S3_comp1_max - S3_comp1_min
allocate0_max_p1 = S3_comp1_cor_max*S3_comp1_max_p1 + S4_comp1_cor_max*S4_comp1_max_p1
allocate0_max_p0 = S3_comp1_cor_max*S3_comp1_max_p0 + S4_comp1_cor_max*S4_comp1_max_p0
if_min_0 = allocate0_max_p0 <= allocate0_max_avail_p0
allocate0_max_avail_p1 = allocate0_avail_after_min - allocate0_max_p0
if_min_1 = allocate0_max_p1 <= allocate0_max_avail_p1
allocate0_max_al_1 = -allocate0_max_avail_p1*if_min_1 + allocate0_max_avail_p1 +
  allocate0_max_p1*if_min_1
allocate0_max_al_0 = -allocate0_max_avail_p0*if_min_0 + allocate0_max_avail_p0 +
  allocate0_max_p0*if_min_0
S4_comp1 = S4_comp1_max_p0*allocate0_max_al_0 + S4_comp1_max_p1*allocate0_max_al_1 +
  S4_comp1_min_p0*allocate0_min_al_0 + S4_comp1_min_p1*allocate0_min_al_1
S4_total = S4_comp1
S3_comp1 = S3_comp1_max_p0*allocate0_max_al_0 + S3_comp1_max_p1*allocate0_max_al_1 +
  S3_comp1_min_p0*allocate0_min_al_0 + S3_comp1_min_p1*allocate0_min_al_1
S3_total = S3_comp1
S2_total = S2_comp1
S5_comp1 = S1_comp1 + S2_comp1 - S3_comp1 - S4_comp1
S5_total = S5_comp1
S1_total = S1_comp1
```

Listing B.1: Model description of forward heuristic allocation

```
1. Details
  Type: Implicit
  Number of equations/variables: 49

2. Continuous equations
  S4_comp1_max = 40
  S3_comp1_max = 30
  S4_comp1_min = 20
  S3_comp1_min = 10
  S4_min_p = 1
  S3_max_p = 1
  S4_max_p = 0
  S4_comp1_cor_max = S4_comp1_max - S4_comp1_min
  S3_min_p = 0
  allocate1_min_p1 = S3_comp1_min*S3_comp1_min_p1 + S4_comp1_min*S4_comp1_min_p1
  allocate1_min_p0 = S3_comp1_min*S3_comp1_min_p0 + S4_comp1_min*S4_comp1_min_p0
  S3_comp1_cor_max = S3_comp1_max - S3_comp1_min
  allocate1_max_p1 = S3_comp1_cor_max*S3_comp1_max_p1 + S4_comp1_cor_max*S4_comp1_max_p1
  allocate1_max_p0 = S3_comp1_cor_max*S3_comp1_max_p0 + S4_comp1_cor_max*S4_comp1_max_p0
  S2_total = S2_comp1
  S1_total = S1_comp1
  R1_comp1 = 0.1*S3_comp1 + 0.1*S4_comp1
  S_avail = R1_comp1 + S1_comp1 + S2_comp1
  allocate1_avail = S_avail
  allocate1_min_avail_p0 = allocate1_avail
  allocate1_min_avail_p1 = allocate1_avail - allocate1_min_p0
  allocate1_min_al_0 = allocate1_min_p0*allocate1_min_p0_if
  allocate1_min_al_1 = allocate1_min_p1*allocate1_min_p1_if
  allocate1_avail_after_min = allocate1_avail - allocate1_min_al_0 - allocate1_min_al_1
  allocate1_max_avail_p0 = allocate1_avail_after_min
  allocate1_max_avail_p1 = allocate1_avail_after_min - allocate1_max_p0
  allocate1_max_al_0 = -allocate1_max_avail_p0*if_min_0 + allocate1_max_avail_p0 +
    allocate1_max_p0*if_min_0
  allocate1_max_al_1 = -allocate1_max_avail_p1*if_min_1 + allocate1_max_avail_p1 +
    allocate1_max_p1*if_min_1
  S4_total = S4_comp1
  S6_comp1 = -R1_comp1 + S3_comp1 + S4_comp1
  S6_total = S6_comp1
  S3_total = S3_comp1
  S5_comp1 = R1_comp1 + S1_comp1 + S2_comp1 - S3_comp1 - S4_comp1
  S5_total = S5_comp1
  R1_total = R1_comp1

```

```
3. Residual variables
  S4_comp1, S3_comp1

4. Residual equations
  -S4_comp1 + S4_comp1_max_p0*allocate1_max_al_0 + S4_comp1_max_p1*allocate1_max_al_1 +
     S4_comp1_min_p0*allocate1_min_al_0 + S4_comp1_min_p1*allocate1_min_al_1
  -S3_comp1 + S3_comp1_max_p0*allocate1_max_al_0 + S3_comp1_max_p1*allocate1_max_al_1 +
     S3_comp1_min_p0*allocate1_min_al_0 + S3_comp1_min_p1*allocate1_min_al_1

5. Conditional equations
  S4_comp1_min_p1 = S4_min_p == 1
  S4_comp1_min_p0 = S4_min_p == 0
  S3_comp1_max_p1 = S3_max_p == 1
  S3_comp1_max_p0 = S3_max_p == 0
  S4_comp1_max_p1 = S4_max_p == 1
  S4_comp1_max_p0 = S4_max_p == 0
  S3_comp1_min_p1 = S3_min_p == 1
  S3_comp1_min_p0 = S3_min_p == 0
  allocate1_min_p0_if = allocate1_min_p0 <= allocate1_min_avail_p0
  allocate1_min_p1_if = allocate1_min_p1 <= allocate1_min_avail_p1
  if_min_0 = allocate1_max_p0 <= allocate1_max_avail_p0
  if_min_1 = allocate1_max_p1 <= allocate1_max_avail_p1
```

Listing B.2: Model description of heuristic allocation with recycle

APPENDIX C

PARTIALLY BUILT INDUSTRIAL PROCESS

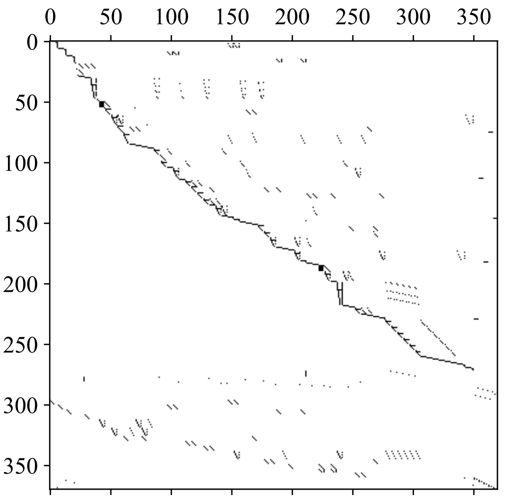


Figure C.1: Unmatched incidence (industrial process). Incidences are indicated by black.

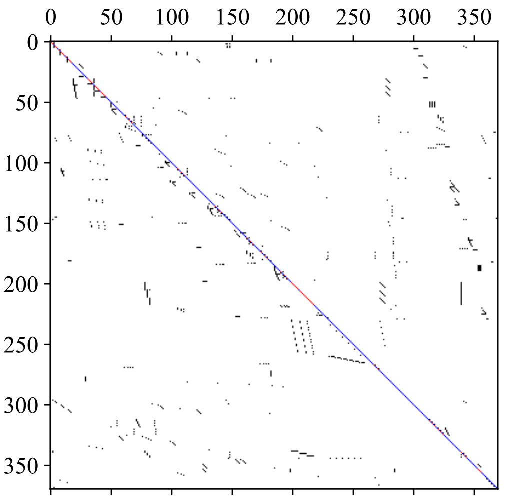


Figure C.2: Matched incidence (industrial process). Safe and unsafe matches indicated by blue and red, respectively.

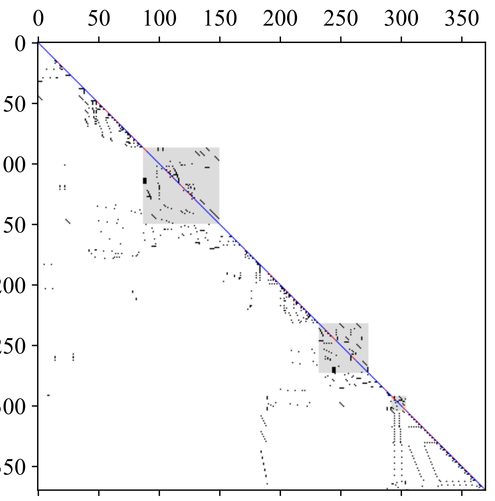


Figure C.3: Block lower triangular incidence (industrial process). Grey indicates blocks.

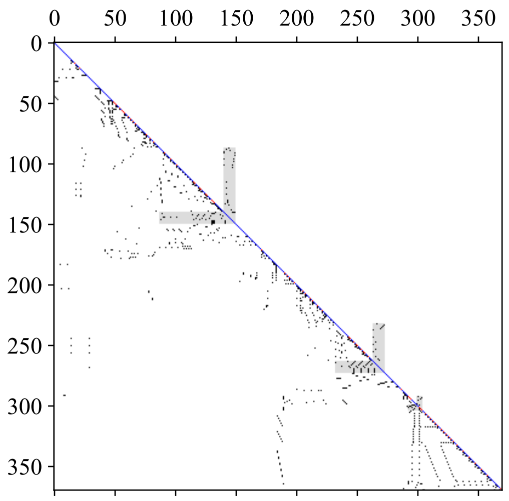


Figure C.4: Spiked lower triangular incidence (industrial process). Grey indicates simultaneous solution regions.

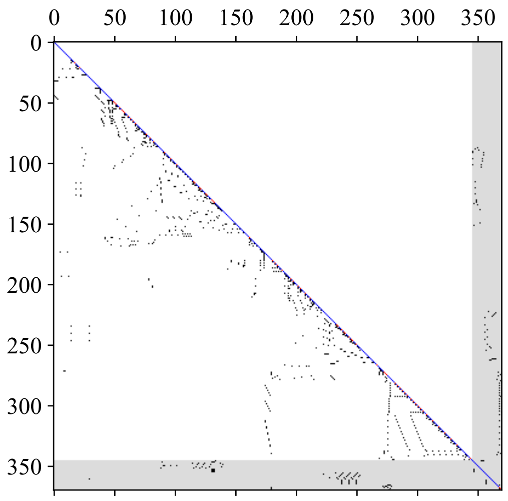


Figure C.5: Bordered block lower triangular incidence (industrial process). Grey indicates the simultaneous solution region.