

AMOSS: AUTOMATIC MODELING  
OPERATIONS USING STOCHASTIC  
SIMULATION

Edgar David Whyte

# **A Moss: Automatic Modeling Operations using Stochastic Simulation**

by

**Edgar David Whyte**

A dissertation submitted 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 2018**

---

# SYNOPSIS

Amoss is a generic equation-orientated stochastic simulation platform specifically designed to facilitate the development and simulation of stochastic models, using Sasol's in-house MOSS methodology. In the current situation in Sasol, modelling stochastic processes with recycle streams (feedbacks) using the MOSS methodology is a laborious task. To simulate these models with acceptable accuracy and simulation speed the model equations were derived and ordered manually. This human input leads to long development times and makes it hard to alter an already created model. Amoss aims to automate the development of MOSS simulation hence the name automatic-MOSS and is an extension of the MOSS methodology.

The automation was achieved by automatically creating the bulk of the model equations and leave only the inputs that make each model unique as user inputs. The aspects that make each model unique were identified as the characteristics of the unit operations, how these operational units are connected, the heuristic rules that govern how the process is operated and the stochastic elements. Given these inputs, a stochastic model is automatically created and ordered to the bordered lower triangular form.

The created models are simulated using Euler's algorithm for integration, coupled with automatic differentiation and multidimensional Newton's method to find the roots of the system of equations at each time step. The models created using MOSS are embarrassingly parallel, and simulation speed was increased by employing parallel processing to exploit the decoupled nature of these simulations.

Amoss consists of all the necessary building blocks to create and simulate stochastic simulations and provides a good platform from which improvements in usability and expansions to the MOSS methodology can be made.

**KEYWORDS:** stochastic simulation, Monte Carlo, flow-sheeting, equation orientated, equation ordering

---

# SINOPSIS

Amoss is 'n generiese vergelyking-gebaseerde stochastiese simulatieplatform wat spesifiek ontwerp is om die ontwikkeling van Sasol se MOSS-metodologie te bevorder. Soos dit tans staan in Sasol is dit 'n moeisame taak om 'n stochastiese proses wat 'n hersirkuleer stroom (terugvoerlus) bevat te modelleer volgens die MOSS-metodologie. Om hierdie prosesse met aanvaarbare akuraatheid en simulasiespoed te simuleer is die model vergelykings met die hand afgelei en ge-orden. Hierdie menslike inset het gelei tot lang ontwikkelings tyd en het die verandering van 'n bestaande model gekompliseer. Die doel van Amoss is om die ontwikkeling van MOSS-simulasies te outomatiseer en gevolglik die naam outomatiese-MOSS en dus is Amoss 'n uitbreiding van MOSS.

Outomatisering is bereik deur die grootste gedeelte van die model vergelykings outomaties te genereer en slegs die insette wat elke model uniek maak as 'n gebruikerinset te los. Die geïdentifiseerde gedeeltes wat elke simulatie uniek maak is die karakteristieke eienskappe van die bedryfseenhede, die konnektiwiteit van dié eenhede, die heuristiese reëls wat die bedryf van die eenhede bepaal en die stochastiese elemente. 'n Stochastiese simulatie kan ontwikkel en georden word na die begrensde laer driehoekige vorm gegewe hierdie instette.

Die ontwikkelde modelle word gesimuleer deur Euler se algoritme te gebruik vir integrasie saam met outomatiese differensiasie en multidimensionele Newton se metode om by elke tydstep die wortels van die stelsel van vergelykings te bereken. Die modelle wat deur die MOSS-metodologie ontwikkel word is almal onafhanklik van mekaar en die simulasiespoed is vinniger gemaak deur die simulasies in parallel te prosesseer.

Amoss bevat al die nodige boustene om stochastiese simulasies te genereer en te simuleer en bied 'n goeie platvorm waarop verbeterings gemaak kan word ten op sigte van die gebruiker asook die uitbreiding van die MOSS metodologie.

**SLEUTELWOORDE:** stochastiese simulatie, Monte Carlo, vloeiagramstudie, vergelyking gebaseerde, ge-ordende vergelykings

---

# ACKNOWLEDGEMENTS

I would like to acknowledge and thank Carl Sandrock who guided me through this project and taught me a great deal and Gerrit Streicher for sacrificing a lot of his time to mould Amoss into the platform it is today and setting the path for future developments. Thanks also to Sasol for supporting Amoss financially.

---

# CONTENTS

Synopsis . . . . .	i
Sinopsis . . . . .	ii
Acknowledgements . . . . .	iii
<b>1 Introduction</b>	<b>1</b>
1.1 Background . . . . .	1
1.2 MOSS . . . . .	2
1.2.1 The methodology . . . . .	2
1.3 Justification of Amoss . . . . .	2
1.4 Deliverables . . . . .	3
<b>2 Theoretical background</b>	<b>6</b>
2.1 Stochastic simulation . . . . .	6
2.1.1 The Monte-Carlo Principle . . . . .	6
2.1.2 Discrete event simulation . . . . .	8
2.1.3 Why use stochastic simulation . . . . .	10
2.2 Graph theory basics . . . . .	10
2.2.1 Bipartite graphs . . . . .	12
2.3 Optimisation . . . . .	12
2.3.1 Formulation of an optimisation problem . . . . .	13
2.3.2 Difficulties of optimisation . . . . .	15
2.3.3 Optimisation methods . . . . .	16
2.3.4 Branch and bound . . . . .	18
2.4 Flowsheeting simulation . . . . .	20
2.4.1 Equation orientated vs modular orientated strategy . . . . .	21
2.5 Equation tearing . . . . .	24
2.5.1 Identifying feasible assignments . . . . .	25

2.5.2	Incidence matrix . . . . .	27
2.5.3	Block triangular decomposition . . . . .	28
2.5.4	Bordered lower triangular tearing . . . . .	30
2.5.5	Alternative ordering . . . . .	39
2.5.6	Failure of tearing . . . . .	40
2.6	Non-uniform random variable generation . . . . .	41
2.6.1	Probability functions . . . . .	41
2.6.2	Generation from a probability function . . . . .	43
2.7	Algorithmic- or automatic differentiation (AD) . . . . .	46
2.7.1	CasADi . . . . .	52
2.8	Parallel processing . . . . .	52
2.8.1	Celery . . . . .	53
2.9	Big O notation . . . . .	53
<b>3</b>	<b>Overview</b>	<b>55</b>
3.1	Relation with MOSS . . . . .	55
3.2	What is Amoss . . . . .	55
3.3	Project software . . . . .	57
3.4	Amoss work-flow . . . . .	58
<b>4</b>	<b>Analysis</b>	<b>60</b>
4.1	Process diagram . . . . .	60
4.1.1	Component information . . . . .	62
4.1.2	Input information . . . . .	62
4.1.3	Separator information . . . . .	63
4.1.4	Reactor information . . . . .	64
4.2	Process graph . . . . .	65
4.3	Automatic equation generation . . . . .	66
4.3.1	Component mass balance equations . . . . .	66
4.3.2	Total flow equations . . . . .	67
4.3.3	Mix Point equations . . . . .	68
4.3.4	Reactor equations . . . . .	70
4.3.5	Separator equations . . . . .	71
4.3.6	Buffer equations . . . . .	73
4.4	Pre-solve equations . . . . .	75
4.5	External user equation . . . . .	75
4.5.1	Special operating instruction functions and methods . . . . .	76
4.5.2	Parsing the operating instruction . . . . .	84
4.6	Ordering to bordered lower triangular form . . . . .	95

4.6.1	Model generation . . . . .	98
4.7	Simulation . . . . .	101
<b>5</b>	<b>User interface (UI)</b>	<b>106</b>
5.1	Editing environment . . . . .	106
5.2	Graphical user interface (GUI) . . . . .	109
<b>6</b>	<b>Conclusion and recommendations</b>	<b>113</b>
6.1	Recommendations . . . . .	115
6.1.1	Near future recommendations . . . . .	115
6.1.2	Future . . . . .	118
<b>A</b>	<b>Time simulation data and inputs</b>	<b>119</b>
A.1	Simulation time frequency distributions . . . . .	119
A.2	Simulation time inputs . . . . .	120



---

## LIST OF FIGURES

2.1	1x1 square with a circle segment with radius 1. . . . .	8
2.2	Convergence of $\mu$ to $\pi$ for the first 500 experiments. . . . .	9
2.3	Graphical representation of a graph $G$ with $V = \{1, 2, 3, 4, 5, 6, 7\}$ and $E = \{\{1, 2\}, \{1, 5\}, \{2, 5\}, \{3, 4\}, \{5, 7\}\}$ adapted form Diestel (2000: 2) .	11
2.4	Graphical representation of a bipartite graph $G$ with $V = \{1, 2, 3, 4, 5, 6\}$ and $E = \{\{1, 5\}, \{2, 5\}, \{2, 6\}, \{3, 4\}\}$ . . . . .	12
2.5	Illustrating the concept of feasible region. The dashed lines show the sides where the inequalities are violated. The heavy line shows the feasible region. (Edgar et al., 2001: 15) . . . . .	14
2.6	The Easom function exhibit a large flat plane with a steep exponential. .	16
2.7	The Ackley function showing many local minimums between a global minimum. . . . .	17
2.8	Decomposition of the optimisation problem in Equation 2.4 using the branch and bound method (Edgar et al., 2001: 356) . . . . .	19
2.9	The incidence matrix in Equation 2.17 represented in a bipartite graph. .	28
2.10	Directed graph created from Figure 2.9 using the list described above. . .	30
2.11	Left: Block lower triangular decomposition first and then tearing on the irreducible blocks. Right: Direct tearing of the incidence matrix. The variables that need to be guessed are shown in grey. (Baharev et al., 2016a)	31
2.12	The steps of tearing: Top left a maximal matching $M' \rightarrow$ top right a matching $M$ after removing infeasible edges $\rightarrow$ bottom left create a directed graph $\rightarrow$ bottom right solve the feedback edge set problem. . . . .	32
2.13	Partially reordered matrix when applying the incomplete perutation $\pi = (\rho, \kappa)$ (Baharev et al., 2016b). . . . .	34
2.14	The heuristic Hessenberg ordering by Fletcher & Hall (1993). . . . .	35

2.15	Example of a spiked lower triangular ordered incidence matrix (Baharev et al., 2016a). . . . .	39
2.16	Probability density function of a normal distribution (top) and the corresponding cumulative distribution function (bottom). . . . .	42
2.17	Probability mass function of the binomial distribution (top) and the corresponding cumulative distribution function (bottom) with $p = 0.5$ and $n = 10$ . . . . .	44
2.18	Probability density function where there is a region of zero probability in-between regions which has a probability. . . . .	45
2.19	Probability distribution function (left) and its inverse (right) of Figure 2.18.	46
2.20	Graphical method to find a non-uniform random variable $X$ given a uniform variable $U$ and a cumulative distribution function. . . . .	47
2.21	Diagram of a lighthouse shining a light on a quay-wall (Griewank & Walther, 2003). . . . .	48
2.22	Directed graph showing the computational flow of the lighthouse example.	50
2.23	Embarrassingly parallel problem represented as a disconnected computational graph adapted from Wilkinson & Allen (2005: 80). . . . .	53
3.1	A small unit with a reactor, distillation column and a buffer tank. . . . .	57
3.2	Work-flow of the Amoss project from problem description to simulation with 1) Process digram in Modelica, 2) Process digarm to network graph, 3) Equation generation, 4) Equation tearing using block lower triangular decomposition, 5) External equations from user, 6) System equation (4) and user equations (5) concatenated and tor using bordered block lower triangular decomposition, 7) Stochastic simulation. . . . .	59
4.1	Example of the graphical process interface by utilising OMEdit. . . . .	60
4.2	Effective representation of a <b>Reactor</b> unit. . . . .	70
4.3	Effective representation of a <b>Buffer</b> unit. . . . .	73
4.4	Typical example where mass must be distributed to multiple destinations.	77
4.5	Graphical method followed to generate non-uniform random variables with a continuous example left and a discrete example right. . . . .	84
4.6	The ast created for the assignment variable a. . . . .	86
4.7	The ast created for the assignment variable c. . . . .	86
4.8	The ast created for the if-block. . . . .	86
4.9	The ast created for the augmented assignment variable e. . . . .	86
4.10	Algorithm that is followed by the created allocate equations. . . . .	88
4.11	Algorithm that is followed by allocate_opt to solve the disjunctive constraints.	92
4.12	Example of a square incidence matrix of 171 rows and columns. . . . .	96
4.13	Reorder incidence matrix to the bordered lower triangular form. . . . .	96

4.14	Combined simulation and result write (to disk) time and how it improved by implementing the method on the x-axis. . . . .	102
4.15	Simulation time in Amoss using the system of equations in Equation 4.40 for different $i$ together with a linear regression fit. . . . .	104
4.16	Simulation time in Amoss using the system of equations in Equation 4.41 for different $r$ together with a linear regression fit. . . . .	105
5.1	Adding comments to equations. . . . .	108
5.2	Tab completion of created variables. . . . .	108
5.3	Syntax highlighting. . . . .	108
5.4	Error indication. . . . .	108
5.5	The user interface that appears when Amoss is started. Top the UI and bottom the info window. . . . .	109
5.6	The user interface that appears when a new model is created. . . . .	110
5.7	User interface to edit simulation information and edit the operating instructions. . . . .	110
5.8	User interface to execute different parts of Amoss. The UI top left is when a simulation is not being run, and the UI top right shows the UI when a simulation is being executed. . . . .	111
A.1	Frequency plot for the base implementation. . . . .	119
A.2	Frequency plot for the HFD5 implementation. . . . .	120
A.3	Frequency plot for the bordered lower triangular implementation. . . . .	121
A.4	Frequency plot for the CasADi implementation. . . . .	121
A.5	Process flow diagram of the process used to time the improvements in simulation time. . . . .	122

---

# LIST OF TABLES

2.1	Decomposition into atomic operations of $F$ from Equation 2.38 (Griewank, 2000: 18). . . . .	49
2.2	Example of the forward mode of applying the chain rule (Griewank & Walther, 2003). . . . .	50
2.3	Example of the reverse mode of applying the chain rule (Griewank & Walther, 2003). . . . .	52
4.1	Example tables show how to add the component list (left) and the simulation inputs (right). . . . .	62
4.2	Example table of the information that is required for every <b>Separator</b> . . .	64
4.3	Example table of the information that is required for every <b>Reactor</b> . . . .	65
4.4	<b>Reactor</b> data table used for the equation example . . . . .	71
4.5	Separation data table used for the equation example . . . . .	72
4.6	Example of how a user can define a distribution. . . . .	82
A.1	The component list (left) and the inputs list (right) of the Figure A.5. . .	123
A.2	Separation data table of Figure A.5. . . . .	123
A.3	Reactor data table of Figure A.5. . . . .	123
A.4	Distribution table of Figure A.5. Left (random1) is a continuous- and right (random2) a discrete distribution. . . . .	124
A.5	Scenario table for the simulation in Figure A.5. . . . .	125

---

---

# CHAPTER 1

---

## INTRODUCTION

### 1.1 Background

Sasol is an integrated energy and chemicals industry in South Africa and leads the world in producing liquid fuels from natural gas and coal (Meyer et al., 2011). The whole coal-to-liquid operation can be grouped into many value chains. A value chain normally consists of a group of interlinked plants, designed to produce a basic component which is then distributed and converted by consumer plants into value-added products and then supplied to their respective markets.

These value chains and their external influences are studied and improved as a unit to ensure a global optimisation instead of considering only individual plants to prevent local optimisation. Factors such as limited plant capacity, insufficient plant availability and sub-optimal operational philosophies, are often the main constraints.

A value chain can become quite complex when recycle streams are present. In such a case, a change in one part of the process can have a big impact on a completely different part of the process. In this situation, it is difficult for the plant Subject Matter Experts (SME) to grasp fully and even harder to quantify the cause and effect which is always required for a capital expenditure motivation.

This is a classic case where process simulation can be applied with significant effect. Since the model must be able to simulate plant availabilities, operational philosophies and storage vessels, the model must be dynamic, stochastic and heuristic in nature. The strength of such a simulation lies in the ability to quantify the impact of a change over the whole value chain and in the process reduce the amount of subjective decision-making.

Over the years, Sasol has developed a modelling methodology, Modeling Operations using Stochastic Simulation (MOSS) (Meyer et al., 2011), to determine the impact of modifications on value chains.

## 1.2 MOSS

The MOSS methodology was pioneered by Meyer et al. (2011). MOSS was created to better understand the effect of variability and dynamics in Sasol's decision-making. It uses a discrete-event simulation approach to model how liquids and gases move through an interactive, continuous petrochemical process at regular (discrete) intervals.

In the past, an average-based approach was used to support decisions (Meyer et al., 2011). These models were created in Simio, spreadsheets and/or posed as linear programming problems. These approaches lacked the statistical variability that occurs in everyday operations as well as the non-linear and dynamic nature of the value chains.

Amoss (Automatic-MOSS) is an extension of MOSS and was created because the current method of creating MOSS simulations is hindering their effectiveness. The points listed in section 1.4 list the shortcomings currently experienced in way MOSS simulations are created and are subsequently the requirements of Amoss.

### 1.2.1 The methodology

The list below summarises how a MOSS model is constructed (Meyer et al., 2011):

1. Translation of the process plant to a flowsheet model.
2. Translation of plant data to discrete events (incidents), probability density functions and operating rules are fitted to data.
3. Identification of failures (cause and effect), the frequency they occur and the time that is required to fix these failures. These failures must also feature in a MOSS simulation and is added as a simulation input.
4. Set-up of optimisation problems like blending of fuel.
5. Validation of models against current conditions.
6. Analysis of the simulated scenarios to support decision-making.

## 1.3 Justification of Amoss

The existing implementation of the MOSS simulation method does not easily deal with value chains that have tightly coupled feedback loops where resource optimisation is a prerequisite. It was necessary to change the existing simulation method to simulate these operations with an acceptable level of accuracy. The modification included many complex mathematical formulae that had to be derived, programmed and tested, which added a significant amount of model development time, resulting in a delayed, but accurate set

of results. The resulting time delay to develop a new model triggered the search for an alternative set of stochastic simulation tools.

The commercial software that was considered was: AnyLogic (The AnyLogic Company, 2017) and Simul8 (SIMUL8 Corporation, 2017).

**AnyLogic** is a modular-orientated simulation platform with an interactive GUI. It supports stochastic variables, development of statistical models and stochastic simulation. It can also accommodate system dynamics, but it cannot solve algebraic loops, hence, mass balances can not be stated directly. Another problem with AnyLogic is that it currently does not support multicomponent flow models.

**Simul8** is a discrete event simulation platform. It is best suited to simulate discrete rather than continuous processes. Handling continuous flow is problematic in Simul8 (Concannon, 2006: 29)

The commercial software also has a relatively high licence fee (IM cost) and Sasol would ideally want a solution that can be distributed to any employee that wants to create simulations using the MOSS methodology.

The University of Pretoria was contacted and requested to submit a modeling solution. The existing stochastic model (Streicher, 2013), the existing optimisation and steady-state models were handed to the University as references.

## 1.4 Deliverables

The University of Pretoria's liaison at Sasol (Gerrit Streicher) identified the following points that need to be satisfied in order to make Amoss useful to Sasol's simulation department:

**Reduction in development time.** With the existing simulation method, it takes too long to build or modify an existing simulation model of a complex value chain. Amoss must reduce development time and not be affected by the complexity of the model.

**Generic application.** Amoss must be able to model any value chain in Sasol irrespective of the combination and configuration of plants. There is also need to convert the existing legacy models to the new modelling solution. One of the measurements of success of Amoss will be determined by the ease with which the conversion can be done.

**Development flexibility.** Once the basic model exists, a new request that requires modification to the existing model must be easy and simple to incorporate.

**Simulation flexibility** During simulation the ability to activate or deactivate a particular portion of the value chain is required. The idea is that significant changes can coexist in a single model instead of building separate models for each scenario.

**Acceptable accuracy.** The simulation must be appropriately accurate to answer the questions arising from value chain improvement scenarios. The accuracy must be independent of the complexity of the value chain.

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

**Linear scalability.** The resulting simulation must scale linearly, i.e. if the number of equations describing the process doubles the simulation time may not increase more than double.

**Quick learning curve.** A quick learning curve is required for the end user, but it is accepted that the learning curve for a developer will take longer.<sup>1</sup>

**IM Cost.** Simulation packages are expensive, and it will be difficult to justify purchasing a new simulation tool which will only partially solve Sasol's problem. The initial cost, as well as the annual maintenance cost, must thus be low. It is also vital that there is continual support from the University of Pretoria to develop the model further on the Sasol's request.

**Version control of model development.** As more than one person 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 locate a bug in a faulty model. The solution must also be able to replicate an error situation in the same replication and scenario in which has occurred.

**Cause identification.** Sometimes a model gives counter-intuitive results. It is expected from the modeller to interpret and identify the reason for an outcome which is often not an easy task. The model solution must be able to identify causes for results and assist in finding 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.

---

<sup>1</sup>A user is a person that will use Amoss to model systems whereas a developer is a person that will work on the Amoss project.



**Software package stability.** Amoss must be stable for all cases especially when final scenario runs are done which are predominantly automated. The model must calculate a value for each variable in each time increment for all replications for all the scenarios.

The above set of thoroughly thought requirements sat narrow but well-defined path for the Amoss project. In the remainder of this document chapter 2 covers all the theoretical knowledge that was required to build Amoss, chapter 3 gives an overview of the current Amoss solution, chapter 4 and chapter 5 provide more detail on how Amoss was created and chapter 6 evaluates Amoss against the requirements above and give suggestions on how to proceed with the project.

---

---

# CHAPTER 2

---

## THEORETICAL BACKGROUND

This chapter covers all the theoretical principals that was used to build Amoss. Chapters 3 and 4 discuss how these principles are applied and it is assumed that the reader are then familiar with these concepts.

### 2.1 Stochastic simulation

According to Ripley (2006: 1) simulation is the use of a model instead of experiments on a real system (eg. physical changes to a chemical factory like increasing buffer capacity) to produce results. The use of simulations above experiments can be due to the system not existing yet (eg. design phase of chemical factory) or to predict observations in a “what-if” analysis of an existing system. On a big system like a chemical factory the ability to perform experiments are limited due to its high cost. This is why simulations are extensively used on chemical factories to assist in decision-making and fault-finding.

A stochastic simulation is simply a model which contains a stochastic<sup>1</sup> element. The term Monte-Carlo simulation is sometimes used interchangeably with stochastic simulation. (Ripley, 2006: 1-3)

#### 2.1.1 The Monte-Carlo Principle

The principle of Monte-Carlo simulation is that the behaviour of a stochastic element can be revealed by repeatedly sampling of its distribution function and observing its results. By implementing the Monte-Carlo strategy an “artificial world” is effectively created which mimics the real world as closely as possible. This “artificial world” can then be used to see how these elements behave across different samples. (Mooney, 1997: 3-4)

---

<sup>1</sup>having a probability distribution

The basic Monte-Carlo procedure is as follows (Mooney, 1997: 4):

1. Define an appropriate distribution function which reflects the behaviour of the stochastic element.
2. Take samples from the distribution function in such a way that it will reflect the desired statistical behaviour, e.g. the same sampling strategy, sample size etc.
3. Calculate the expected value ( $\mu = E(X)$ ) dependent on the stochastic elements.
4. Repeat steps two and three for the desired number of replications  $r$ .
5. Draw a relative frequency distribution of the resulting  $\mu_r$  values which are the Monte-Carlo estimates of the distribution of  $\mu$ .

The Monte-Carlo method is based on the strong law of large numbers which states that the arithmetic mean of a sequence of independently, identically distributed random variables  $X$  converges to the expected value  $\mu = E(X)$  (Korn et al., 2010: 57). That is the more samples are drawn (experiment replications) the better the estimate  $\mu$  will become. This can be related to the range of the 95% confidence interval of  $\mu$  which can be calculated using the  $2\sigma$ -rule for an approximate 95% interval in Equation 2.1

$$[\bar{X}_N - \frac{2\bar{\sigma}}{\sqrt{N}}, \bar{X}_N + \frac{2\bar{\sigma}}{\sqrt{N}}] \quad (2.1)$$

where

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

(Korn et al., 2010: 59) The range of the confidence interval is proportional to  $\frac{1}{\sqrt{N}}$  and therefore to decrease the range by a factor of 0.1, 100 additional experiments are required.

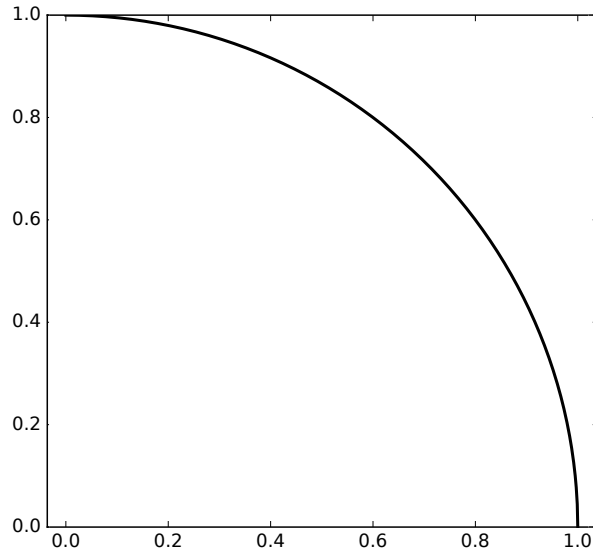
To illustrate the Monte-Carlo principle by Mooney (1997) the classical example to estimate the value of  $\pi$  is used.

If a square of  $1 \times 1$  is taken and a circle segment with radius 1 is drawn within the square (seen in Figure 2.1) the probability of a point  $P = (y_1, y_2)$  being encircled can be used to estimate  $\pi$  when  $y_1$  and  $y_2$  are sampled from a uniform random distribution

Let  $\mathbf{C}$  be the collection of all the points which are encircled and  $X_i = 1$  when  $P_i \in \mathbf{C}$ . The probability that  $P_i$  is within  $\mathbf{C}$  ( $\mathbb{P}(P_i \in \mathbf{C})$ ) is the area the circle segment encircles in the square divided by the total area of the square. Equation 2.2 shows how the  $\mathbb{P}(P_i \in \mathbf{C})$  was calculated

$$\mathbb{P}(P_i \in \mathbf{C}) = \frac{\frac{\pi}{4}r^2}{r^2} = \frac{\pi}{4} \quad (2.2)$$

This is only true because  $P_i$  has the same probability to be anywhere within the square. The fraction of the square encircled by the circle segment is  $\frac{\pi}{4}$  and therefore the probability of  $P_i$  being encircled is also  $\frac{\pi}{4}$ .



**Figure 2.1:** 1x1 square with a circle segment with radius 1.

The steps by Mooney (1997)

1. The distribution that is chosen for this simulation is the uniform random distribution.
2. Two variable  $y_1$  and  $y_2$  was sampled form the distribution at every experiment.
3. The experiment will be conducted for  $N = 100000$ .
4. The estimate for  $\pi$  ( $\mu$ ) is calculated by  $\mu = \frac{4}{N} \sum_{i=1}^N (X_i)$  (Korn et al., 2010: 61).
5. The frequency diagram for the estimates  $\mu$  was drawn in Figure 2.2.

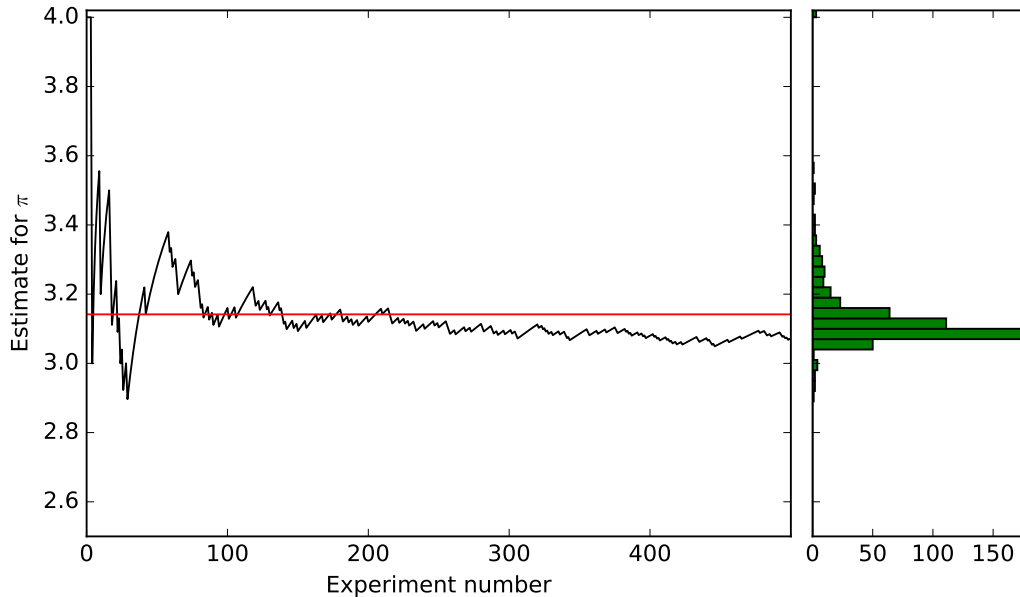
The convergence of  $\mu$  is plotted in Figure 2.2. See how  $\mu$  moves closer to the true value of  $\pi$  as the number of experiments increase.

### 2.1.2 Discrete event simulation

Discrete event simulation is used to model the behaviour of a system in a variety of applications like production, scheduling, traffic and many more. Discrete event simulation consists of two main parts (Kroese et al., 2011: 283):

**System state:** The set of variables or parameters that are required to describe the system.

**Event:** An instantaneous occurrence that may change the system state. The event consists of the *event time* which is the time when the event occurs and the *event type* identifying how the event affects the system state after the event time.



**Figure 2.2:** Convergence of  $\mu$  to  $\pi$  for the first 500 experiments.

In discrete event simulations, the system is only observed at the event times. Between event times the system must behave in a deterministic way. Discrete event simulations require an internal timekeeping mechanism called a simulation clock to advance the simulation time from one event to another. This clock is necessary because of discrete event simulation's dynamic nature. An event list is used to keep track and maintain all the pending events in chronological order. The event list always has the most imminent event at its head. (Kroese et al., 2011: 283)

At the start of the simulation, the events are added to the event list, and the first event is executed. Next, the simulation clock is advanced and the next imminent event executed and removed from the event list. The advance of the simulation clock, execution and removal of events are repeated until a stop criterion is reached (fixed simulation time or last event simulated). (Kroese et al., 2011: 287 - 285)

When implementing discrete event simulation, two approaches can be followed (Kroese et al., 2011: 284):

- Event-orientated approach: Separate subroutines are executed for every event, and the result updates the system state. In this approach, the purpose of the main program is to progress through the event list.
- Process-oriented approach: An event is a deterministic sub-processes rather than a single stochastic event. In the process-orientated approach, a sub-process is inter-

rupted at the event time, and a different sub-process is activated. The event list contains a set of sub-processes.

### 2.1.3 Why use stochastic simulation

Models are used to help in understanding how a real-life process behaves, to predict the behaviour of a process in the future or to aid in decision-making (Ripley, 2006: 3). To make use of a model one has to choose if model analysis or simulation will solve the problem (Ripley, 2006: 3):

**Analysis:** Make use of mathematical analysis of the model to try and understand the behaviour of the model. This method becomes increasingly more difficult the more complex the model becomes.

**Simulation:** Experiment with the model by changing some parameters and observe how it affects the model.

The choice between simulation and analysis depends on the purpose of the model. As a general rule simulation is used to answer “what-if” questions and to aid in decision-making and analysis is usually used to understand the model. (Ripley, 2006: 3)

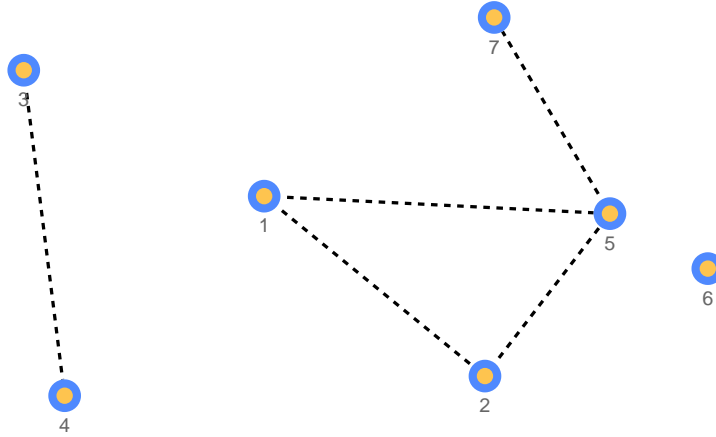
A distinction can also be drawn between two different models: mechanistic or convenient. Stochastic models are usually convenient whereas physical models are mechanistic and deterministic. Well-defined complex physical models are difficult to simulate due to the high computational cost. By making use of a stochastic model, a complex model can be reduced to a convenient model, and the convenient model can be simulated instead. (Ripley, 2006: 3)

In other cases, the choice between a deterministic or a stochastic model is made for you. An example in the petrochemical industry is that the failure rates in an operational unit or their equipment are stochastic in nature. Critical events like pump and compressors failures or even shut-down events like fires can be simulated by sampling the stochastic variables from a distribution which represents these events.

## 2.2 Graph theory basics

A graph is denoted by  $G$  and  $G$  is a pair of sets,  $G = (V, E)$ , where  $E$  is the 2-element subset of  $V$  ( $E \subseteq [V]^2$ ). The set  $V$  is known as the vertices or nodes of the graph  $G$  and  $E$  the edges. A graph can be represented graphically shown in Figure 2.3. An edge is named by an ordered pair of vertices for a directed graph or an unordered pair of vertices for an undirected graph, eg.  $\{x, y\}$  is written as  $xy$  or  $yx$  in an undirected graph. A trivial graph  $G$  is an empty graph ( $G = (\emptyset, \emptyset)$ ) or a graph with a single vertex. For practicality, a graph will always be assumed to be non-trivial. (Diestel, 2000: 2)

A directed graph  $D$  is a graph with a pair  $(V, E)$  together with the maps  $init : E \rightarrow V$  and  $ter : E \rightarrow V$ . The maps  $init$  and  $ter$  assigns an initial (start) vertex and terminal (end) vertex for  $\forall e \in E$ . The edge  $e$  is said to be directed from  $init(e)$  to  $ter(e)$ . (Diestel, 2000: 25)



**Figure 2.3:** Graphical representation of a graph  $G$  with  $V = \{1, 2, 3, 4, 5, 6, 7\}$  and  $E = \{\{1, 2\}, \{1, 5\}, \{2, 5\}, \{3, 4\}, \{5, 7\}\}$  adapted form Diestel (2000: 2)

Two vertices  $x, y$  are said to be adjacent or neighbours if the edge  $xy$  forms part of  $G$ , e.g. vertices 1 and 2 in Figure 2.3 are adjacent because the edge 1,2 exists but vertices 1 and 3 are not adjacent. The two non-equal edges  $e$  and  $f$  ( $e \neq f$ ) are adjacent if they have an end in common, e.g. edge 1,2 and edge 1,5 in Figure 2.3. A set of edges or vertices are called independent if none of the elements are adjacent, e.g. an independent vertex set in Figure 2.3 will be  $\{1, 3, 6, 7\}$  and edge set will be  $\{\{1, 2\}, \{3, 4\}, \{5, 7\}\}$  (Diestel, 2000: 3)

A path is a non-empty graph  $P$  with  $P = (V, E)$  given that

$$V = \{x_0, x_1, \dots, x_k\} \quad E = \{x_0x_1, x_1x_2, \dots, x_{k-1}x_k\}$$

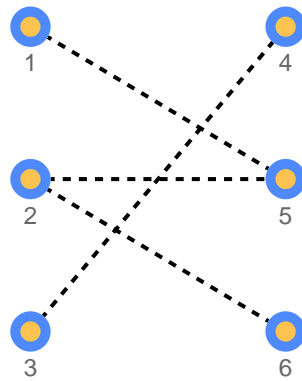
and  $x_i$  are all distinct. A path  $P$  can also be represented as  $P = x_0x_1\dots x_k$ . The length,  $k$ , of a path is equal to the number of edges in the path. (Diestel, 2000: 6-7)

A cycle is a path that starts and ends at the same vertex. If the path  $P = x_0\dots x_{k-1}$  has a length larger or equal to three ( $k \geq 3$ ) then the graph  $C := P + x_{k-1}x_0$  is called a cycle.(Diestel, 2000: 6-7)

A simple cycle does not have any repeating vertices or edges. Two or more simple cycles are called distinct if non of the cycles in question are a cyclic permutation of the one another (Baharev et al., 2016a).

### 2.2.1 Bipartite graphs

The graph  $B = (V, E)$  is called bipartite if the vertex set  $V$  can be divided into two classes. Vertices in the same class may not be neighbours with any of the vertices of the same class but only to vertices in the other class. The graphical representation of a bipartite graph is shown in Figure 2.4. A graph is considered bipartite if and only if it contains no odd cycles. (Diestel, 2000: 14-15)



**Figure 2.4:** Graphical representation of a bipartite graph  $G$  with  $V = \{1, 2, 3, 4, 5, 6\}$  and  $E = \{\{1, 5\}, \{2, 5\}, \{2, 6\}, \{3, 4\}\}$

### Matching in bipartite graph

A matching  $M$  is a set of independent edges in the graph  $B = (V, E)$ .  $M$  is a matching of  $U \subseteq V$  if every vertex in  $U$  is incident with an edge in  $M$ . The vertices in  $U$  are said to be matched by  $M$  whereas the vertices not incident with the edges in  $M$ , i.e.  $V - U$ , are unmatched. (Diestel, 2000: 29)

A maximal matching is the edge set  $M'$  with the highest number of matches for  $B$  were  $|M'| \geq |M|$  for any matching  $M$ , where  $|M|$  is the number of edges in  $M$  (Baharev et al., 2016a). A complete matching is a matching where all the vertices ( $V$ ) are matched.

## 2.3 Optimisation

Optimisation is used to solve problems over a large range of fields from engineering to finance. Optimisation is used to assist in decision-making where the answer is not necessarily intuitive. A typical optimisation problem can be posed as a system represented by a set of equations or experimental data with a single performance criterion subjected to various constraints. This criterion can be to reduce operational costs, increase system performance or to provide a compromise between the two. The goal of optimisation is to find values of the variables in a system that yield the best value of the performance criteria. (Edgar et al., 2001: 4-5)



### 2.3.1 Formulation of an optimisation problem

To formulate an optimisation problem a model representing the system under study together with a suitable performance criterion<sup>2</sup> is required. Every optimisation consists of three essential categories:

1. At least one objective function.
2. Equality constraints.
3. Inequality constraints.

(Edgar et al., 2001: 14)

The mathematical way an optimisation problem is presented is shown in Equation 2.3

$$\text{Minimise: } f(x) \quad \text{objective function} \quad (2.3a)$$

$$\text{Subject to: } h(x) = 0 \quad \text{equality constraints} \quad (2.3b)$$

$$g(x) \geq 0 \quad \text{inequality constraints} \quad (2.3c)$$

which contains all three essential categories of an optimisation problem (Edgar et al., 2001: 16).

Consider the following definitions used in optimisation:

**Feasible solution** of the optimisation problem is a set of variables that satisfy the equality and inequality constraints.

**Optimal solution** is the set of values that simultaneously satisfy the equality and inequality constraints and provides an optimal value of the objective function. The optimal solution is not guaranteed to be unique.

**Feasible region** is the region where the equality and inequality constraints are satisfied.

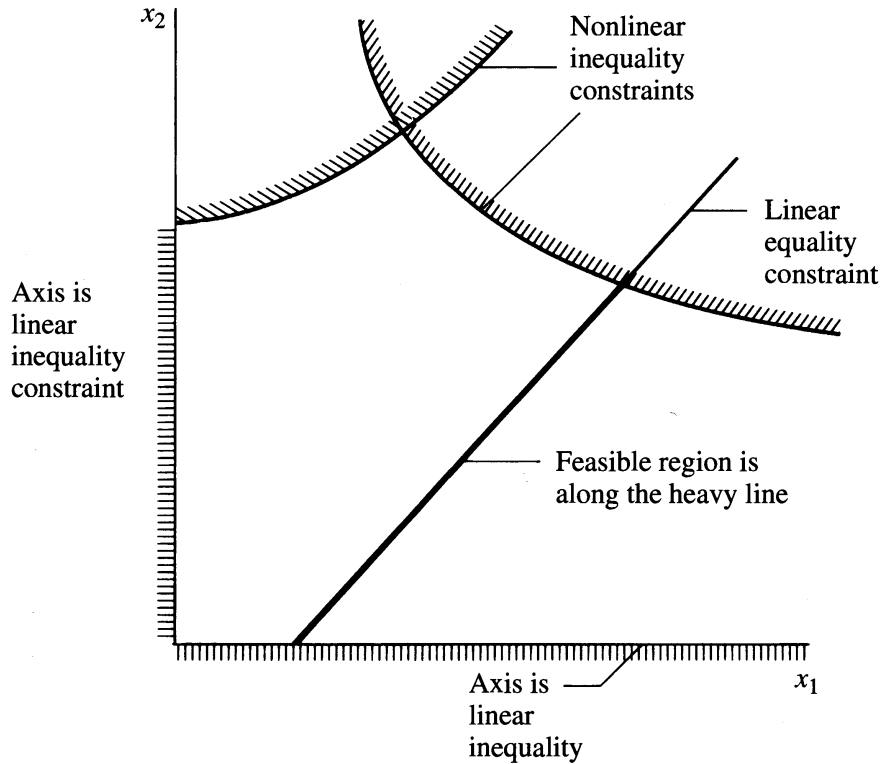
All *feasible solutions* and ultimately the *optimal solution* (if it exists) is found in this region. Figure 2.5 is a graphical illustration of the *feasible region*.

In optimisation the degrees of freedom (DOF) of the system under study is important and will determine if an optimal solution can be found. A DOF analysis separates the problem into three categories (Edgar et al., 2001: 66-67):

1. DOF = 0: The problem is fully determined because the number of independent equality constraints is equal to optimisation variables. In this case, the problem is no longer one of optimisation but simulation.

---

<sup>2</sup>Also known as an objective function.



**Figure 2.5:** Illustrating the concept of feasible region. The dashed lines show the sides where the inequalities are violated. The heavy line shows the feasible region. (Edgar et al., 2001: 15)

2.  $\text{DOF} > 0$ : The problem is undetermined, and the number of optimisation variables are more than the independent equality constraints. In undetermined systems, at least one variable can be optimised.
3.  $\text{DOF} < 0$ : The problem is overdetermined, and the number of optimisation variables are less than the independent equality constraints. For overdetermined problems, the set of equality constraints cannot be satisfied. Some constraints can, therefore, be relaxed. Another option would be to find a solution closest to the feasible region (Edgar et al., 2001: 16).

Edgar et al. (2001: 19) lists four steps to formulate an optimisation problem and two steps to solve it:

1. Analyse the system and list all the variables of interest.
2. Determine the criteria for optimisation and formulate the objective function in terms of the variables identified in step 1 together with its coefficients.
3. Using mathematical expressions develop a system model that relates the input-output relation between variables its associated coefficients. In this step add the equality and inequality constraints.

4. If the problem formulation is too large subdivide it into smaller manageable parts or simplify the objective function or model.
5. Use a suitable optimisation technique to solve the optimisation problem.
6. Check the answer and examine the sensitivity of the result to changes in the problem coefficients or assumptions.

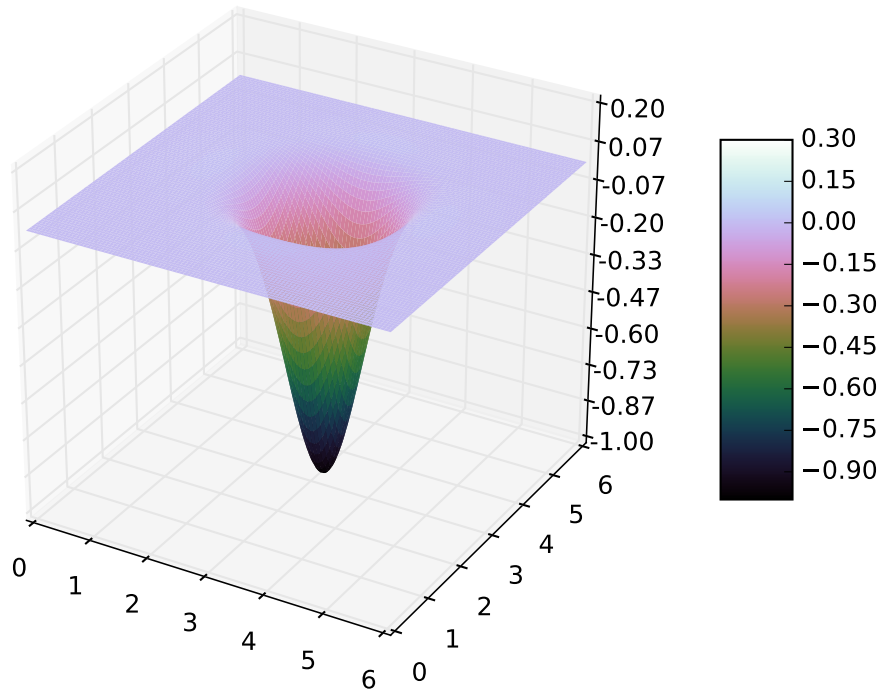
It must be noticed that steps 1-4 (problem formulation) are dependent entirely on the developer of the optimisation problem. If the optimisation problem is not formulated correctly, difficulty and problems can be expected in step 5. Today there are a variety of optimisation packages available that can perform steps 5-6, but if the formulation is not done correctly, the well-known saying of “garbage in, garbage out” can be applied.

### 2.3.2 Difficulties of optimisation

Some optimisation problems are easier to solve than others. Objective functions and constraints that are all linear can be solved more easily than non-linear problems. The advantage of linear problems is that it is explicitly known that the solution is unique if it exists.

Edgar et al. (2001: 26-27) lists five common situations in optimisation that can cause failure in the calculation:

1. The objective function or the constraints have finite discontinuities in the continuous parameter values. Discontinuities are common in objects having fixed prices. Take a desktop computer as an example, the price of the CPU is fixed and jumps discontinuously between different models while the required processing power (measured in GHz) can be continuous.
2. The objective function or the constraints are non-linear. Non-linearity is almost always present.
3. The variables in the objective function or the constraint functions may have strong interactions with one another. For an example the objective function  $f(x_1, x_2) = x_1 \times x_2$  prohibits the determination of a unique value for  $x_1$  and  $x_2$ .
4. The objective function or constraints may exhibit nearly “flat” behaviour in some regions and exponential behaviour in other. An illustration of this behaviour is displayed in Figure 2.6.
5. The objective function may have many local optima where a global optimum is required. An illustration of such a function is shown in Figure 2.7.



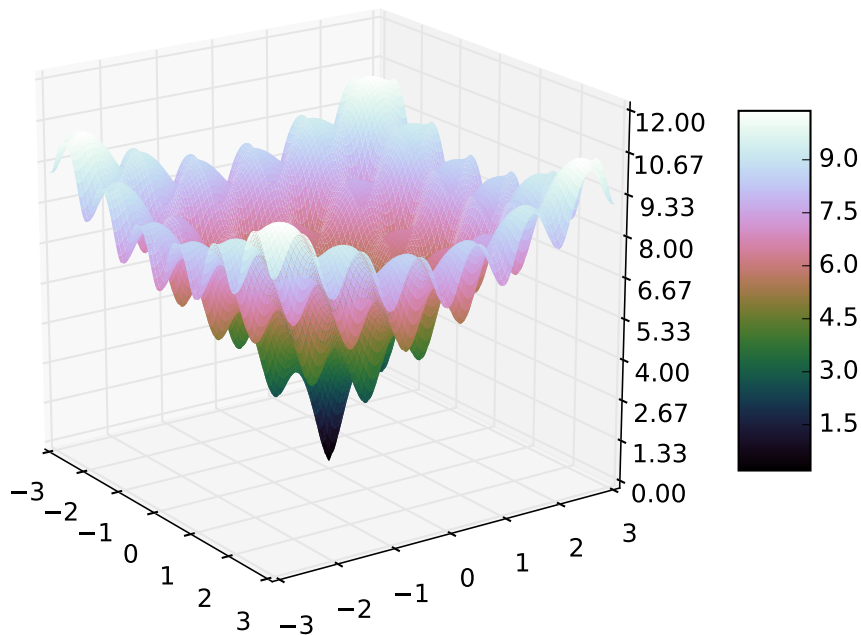
**Figure 2.6:** The Easom function exhibit a large flat plane with a steep exponential.

### 2.3.3 Optimisation methods

There are a variety of different approaches to solving an optimisation problem which will be briefly discussed with the focus on the branch and bound (BB) method because it is used in Amoss. The two principal optimisation methods are: 1) methods that rely on function values *only* (zero order) and 2) techniques that make use of gradient information

The zero order techniques evaluate the objective function and check the results to determine the direction or area to search for the optimum. Under some circumstances, these methods are effective but are mostly ineffective compared to methods exploiting the gradient information. Some of these zero-order methods are:

- The **random search** method which randomly assigns values to the optimisation variables compares the current position to the best previous position and updates the best value when a better solution is found (Edgar et al., 2001: 183).
- The **grid search** method forms a grid of values around a reference point. The point with the most optimal value becomes the next reference point, but now the grid is condensed (consisting of the same number of evaluations but in a smaller are) (Edgar et al., 2001: 183).



**Figure 2.7:** The Ackley function showing many local minimums between a global minimum.

- The **simplex** method selects points at the vertices of the simplex to evaluate the objective function. In 2D space, the simplex is an equilateral triangle, and in 3D space, it becomes a regular tetrahedron and so on. In the simplex method the search direction is determined by moving away from the vertex which has the least optimal value. (Edgar et al., 2001: 185)

The other methods make use of first or second order derivative information to determine a search direction in which to approach the optimum. Some of these methods are:

- The **steepest decent** method uses first-order derivatives to find the direction of steepest descent for minimisation and ascent in maximisation at the current iterative point. A step is made in that direction, and the gradient is evaluated again to find a new direction. (Edgar et al., 2001: 190)
- **Newton's method** makes use of the second order derivatives to determine a search direction. It is analogous to the steepest descent method if the steepest descent is interpreted as a linear approximation of the objective function. Newton's method makes use of the second order derivatives and makes a quadratic approximation of

the objective function. The minimum of the approximated quadratic function is determined which serves as the new direction. The advantage of this method, as opposed to steepest descent, is that it can account for the curvature of the objective function. (Edgar et al., 2001: 197)

### 2.3.4 Branch and bound

The branch and bound (BB) method is a technique that is used to solve mixed integer non-linear programming (MINLP) problems which contain both continuous and integer variables. The method involves the relaxation of the integer constraints and making these variables continuous, e.g. a binary<sup>3</sup> variable  $y$  is relaxed to be a continuous variable bounded between 0 and 1 ( $0 \leq y \leq 1$ ). The relaxed problem is called the NLP relaxation of the MINLP (Edgar et al., 2001: 355, 362).

BB starts by solving the relaxed NLP relaxation using an NLP solver (Edgar et al., 2001: 355, 362). If all the discrete variables in the NLP relaxed problem have integer values, the NLP relaxed problem also solves the MINLP. If one or more of the binary variables have a fractional value the next step in BB is to start branching and create two NLP sub-problems by fixing one of the fractional variables to 0 in the one sub-problem and 1 in the other (see Edgar et al. (2001: 358) for a non-binary example). If one of the sub-problems have an integer solution then a solution to the MINLP is found, and that branch needs no further investigation. If a branch contains a fractional value for a discrete variable that branch may need further branching.

To decide whether a branch may need further branching depends on the best solution of the MINLP at that point. If the NLP sub-problem has a better value for the objective function than the best MINLP, then it warrants further branching. When the NLP sub-problem has a worse value for the objective function, it needs no further branching. The value for the objective function of an NLP will not increase when enforcing the discrete constraint and will always lead to a worse value.

To explain the BB method better consider the mixed integer linear programming (MILP) problem in Equation 2.4 and its solution in Figure 2.8

$$\text{Maximise: } f = 86y_1 + 4y_2 + 40y_3 \quad (2.4a)$$

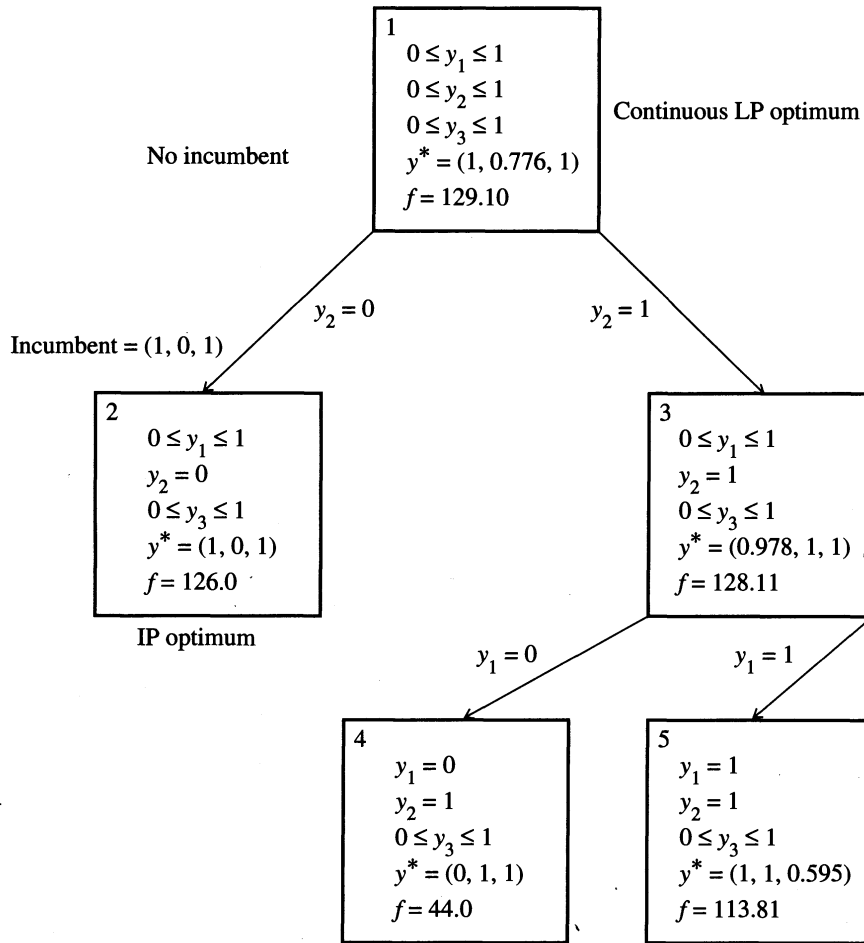
$$\text{subject to: } 774y_1 + 76y_2 + 42y_3 \leq 875 \quad (2.4b)$$

$$67y_1 + 27y_2 + 53y_3 \leq 875 \quad (2.4c)$$

$$y_1, y_2, y_3 = 0, 1 \quad (2.4d)$$

---

<sup>3</sup>Either 0 or 1



**Figure 2.8:** Decomposition of the optimisation problem in Equation 2.4 using the branch and bound method (Edgar et al., 2001: 356)

The relaxed problem was solved in node 1 with  $y_2$  a fractional value therefore branching was required. In node 2  $y_2$  was fixed to 0 which resulted in a solution to the MILP problem with  $f = 126.0$  and a possible optimal solution. In node 3  $y_2$  was fixed to 1 but  $y_1$  is fractional and thus not a solution to the MILP. The objective function of node 2 has a value of  $f = 128.1$  and is higher than the best MILP solution of 126.0 and requires further branching. In node 4  $y_1$  was fixed to 0 and resulted in a solution to the MILP with  $f = 44.0$  but the objective function value is worse than in 126.0 and was discarded as a possible optimal solution. In node 5  $y_1$  is fixed to 1 and resulted in  $f = 113.8$  with  $y_3$  fractional. The objective function value of node 5 is worse than in node 2 and therefore node 5 needs no further branching. The result in node 2 is the solution to the MILP.

Noticed that after every branching the value of the objective function was always worse than the node it branched form.

## Disjunctive programming

Disjunctive programming is a special case of MINLP problem (Edgar et al., 2001: 371). A disjunctive constraint is a type of constraint where a variable has two or more valid regions separated by invalid regions. An example would be a variable  $x$  with the constraint  $x \leq 0$  or  $x \geq 1$  (the region  $(0, 1)$  is invalid). This type of constraint is known as a logical condition, and exactly one of these conditions must be true.

Disjunctive programming problems are solved by reformulating it as a MINLP problem by introducing binary variables using the big-M approach (Edgar et al., 2001: 372). The big-M approach uses an arbitrary large constant denoted by  $M$  to switch between different disjunctive constraints.

An optimisation problem with the disjunctive constraint of  $x$  above can be reformulated in Equation 2.5 by introducing new variables  $y_1$  and  $y_2$  and an equality constraint in Equation 2.5b

$$\text{Minimise: } f(x) \tag{2.5a}$$

$$\text{subject to: } y_1 + y_2 = 1 \tag{2.5b}$$

$$x - 1 \geq -My_1 \tag{2.5c}$$

$$x \leq My_2 \tag{2.5d}$$

$$y_1, y_2 = 0, 1 \tag{2.5e}$$

The problem in Equation 2.5 can now be solved as an MINLP using BB. The constraint in Equation 2.5b ensures that only  $y_1$  or  $y_2$  is 1 at any given point which in effect “switches” constraints on and off. With  $M$  sufficiently large if  $y_1 = 0$  then constraint 2.5c is enforced and 2.5d is switched off. This is achieved by making constraint 2.5d “easy” to be satisfied because  $M$  is large whereas constraint 2.5c is one of the original constraints for  $x$ . The same applies if  $y_2 = 0$  but now constraint 2.5d is enforced.

## 2.4 Flowsheeting simulation

Flowsheeting is the calculation of steady-state heat and mass balances and costing calculations of a chemical process. In flowsheeting a process block diagram (flowsheet) is analysed together with the complete system characteristics. This analysis does not include any dimensions, structural design parameters, instrumentation or piping network system. Flowsheeting simulation involves the analysis of the flowsheet by creating the necessary equations describing the mass and heat flow through the process given equipment parameters (reactor conversion and separator split ratios) and system inputs to calculate the outputs of the flowsheet. The principal approaches to flowsheeting simula-



tion are the sequential modular and equation orientated approaches. (Ludwig & Coker, 2007)

### 2.4.1 Equation orientated vs modular orientated strategy

An equation orientated strategy gathers all the equations of the flowsheeting<sup>4</sup> problem in a single large set of algebraic equations and attempts to solve the equations simultaneously using multidimensional root-finding software. A modular strategy attempts to solve modular units sequentially by tearing the recycle streams and iterating to convergence. (Barton, 2000)

The equation orientated strategy consists of the following steps:

1. Equations and variables are created at the modular level even though it is not used to simulate the process.
2. These equations and variables are combined into one large system of non-linear equations.
3. Some variables are specified to make the degrees of freedom (DOF) of the entire system zero.
4. The system of equations are solved simultaneously by using general root-finding software.

### Advantages and disadvantages of the equation orientated approach

The equation orientated approach has many advantages over the modular approach and it is more suitable for a variety of different modelling requirements, even outside pure flowsheeting problems. The equation orientated approach has also received much academic attention, resulting in improved methods for general root-finding (Barton, 2000).

The advantages of equation based strategies is listed below (Barton, 2000):

- It is much more efficient. Although modular approaches are efficient at solving unidirectional problems (equation orientated approaches can take even greater advantage of such systems), it becomes much more computationally expensive when recycles or design specifications are introduced, due to the high increase in required passes through the flow-sheet because of the nested strategies used in converging the tear streams.
- The barrier between simulation and design specifications are effectively removed. By specifying the design requirements in step 3 of the equation orientated strategy,

---

<sup>4</sup>Flowsheeting is the steady-state simulation of a process (Shacham et al., 1982)

the design specifications can be calculated. Similarly, by specifying the design specifications, the outputs in a simulation can be calculated. In a well-posed problem, the computational load stays the same in both these situations unless the linearity or the block decomposition are significantly changed between the two specifications sets.

- It is much easier to extend or modify existing model libraries because the model is simply viewed as a set of equations and variables. All equations are handled in the same manner and therefore the libraries can simply incorporate additional equations describing new operational units or modifying existing units. Whereas in a modular approach a subroutine with a system of equations and an embedded solution procedure may be required.
- The primary advantage of this strategy is its capability to be extended in a variety of different modelling requirements which includes design, flowsheeting simulation, optimisation and dynamic simulation. This versatility is due to the architecture of this method making it possible to interface with the model using different subroutines.
- The debugging of certain model formulation errors is possible even in situations where a poorly posed problem is not localised to a single modular unit but combination of different equations. The debugging of non-localised (not on modular level) problems is achieved by analysing the entire problem for errors such as singularities, which is not possible in the modular approach.

The advantages of the equation orientated approach seems like it is overwhelmingly superior, but it is not without its drawbacks. The disadvantages of the equation orientated approach are listed below (Barton, 2000):

- The general purpose root-finding software is not as robust and reliable as the sequential modular methods.
- It has a larger demand for computer resources, in particular memory (although the restriction on these hardware pieces has essentially been eliminated with the wide availability of powerful machines).

### **Multidimensional Newton's method**

At the heart of the equation orientated strategy is the multidimensional root-finding software. The ability to find the roots of a multidimensional system of equations is a key to this strategy. One method to finding the roots of a multidimensional system of equations is to use Newton's method. (Barton, 2000)

Newton's method approximates  $f(\mathbf{x}) = \hat{0}$  where  $\mathbf{x}$  is a vector of the independent variables and  $\hat{0}$  the zero vector using a first order Taylor expansion of  $f$ .

$$f(\mathbf{x}) \approx f(\mathbf{a}) + J|_{\mathbf{a}} \cdot (\mathbf{x} - \mathbf{a}) \quad (2.6)$$

where  $\mathbf{a}$  is the point around which the Taylor expansion is applied and  $J|_{\mathbf{a}}$  is the Jacobian matrix of  $f$  evaluated at  $\mathbf{a}$ . (Barton, 2000)

The goal of Newton's method is to find  $x^*$  that satisfy the equality  $f(x^*) = 0$ . With rearrangement of Equation 2.6, Equation 2.7 shows an iterative method to find  $x^*$  known as Newton's method

$$x^{k+1} = x^k - J|_{x^k}^{-1} f(x^k) \quad (2.7)$$

and when  $k$  is sufficiently large  $x^{k+1} \cong x^*$ . (Barton, 2000)

With reference to Equation 2.7 the algorithm to the multidimensional Newton's method is given below:

1.  $k = 0$  ( $x^0$ ) the initial guess.
2. Evaluate  $f(x^k)$ .
3. Check for conversion using some stopping criteria eg. if  $|f(x^k)| \leq \epsilon$  then stop.
4. Evaluate  $J|_{x^k}$ .
5. Get  $J|_{x^k}^{-1}$ .
6. Calculate  $x^{k+1}$  using Equation 2.7.
7.  $k = k + 1$ . Repeat from step 2.

(Barton, 2000) This iterative multidimensional Newton's method converges rapidly close to the solution.

In modern root-finding software the basic Newton method as described above is modified in several ways:

- Any updated  $x^{k+1}$  must obey some physical bounds on the variables of  $\mathbf{x}$  and only a fraction of the full Newton step can be applied to ensure that these bounds are not violated and a term in Equation 2.7 is modified to

$$x^{k+1} = x^k - \alpha J|_{x^k}^{-1} f(x^k) \quad (2.8)$$

where  $0 \leq \alpha \leq 1$ .

- The system of equations to solve in flowsheeting problems are usually sparse and appropriate sparse linear equation solvers are used to solve the linear equation  $J|_a \cdot (x^* - a) = -f(a)$  in step 5 instead of calculating the inverse directly.

- Modifications to the iterative process is required if the Jacobian becomes numerically singular at one or more rows and the inverse  $J|_{x^k}^{-1}$  can not be calculated.

(Barton, 2000)

## 2.5 Equation tearing

The purpose of tearing algorithms is to reduce the amount of computational time that is required to solve a system of equations. This is achieved by ordering the original system of equations  $f(x) = 0$ , where  $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ , and reducing it to a smaller (usually much smaller) system  $H(z) = 0$ . This is achieved by reordering the equation vector  $f$  and the variable vector  $x$  in Equation 2.9 using permutation matrices  $P$  and  $Q$

$$\begin{bmatrix} g \\ h \end{bmatrix} = Pf, \quad \begin{bmatrix} y \\ z \end{bmatrix} = Qx \quad (2.9)$$

so that most of the variables  $y$  can be solved sequentially. The transformation must produce equations  $g_i(y, z) = 0$  (equation  $i$  in equation vector  $g$ ) that can be rewritten in an explicit form (shown in Equation 2.10)

$$y_i = \tilde{g}_i(y_{1:i-1}, z) \quad (2.10)$$

using appropriate symbolic transformations. It is important to note that the transformed equation  $\tilde{g}_i$  contains only variables  $z$  and a row slice of the variable vector  $y$  from the index 1 to  $i - 1$  ( $y_{1:i-1}$ ). (Baharev et al., 2016a)

Given a reordering,  $f(x) = 0$  can be rewritten in Equation 2.11

$$g(y, z) = 0 \quad (2.11a)$$

$$h(y, z) = 0 \quad (2.11b)$$

The requirement for Equation 2.10 is that it must be able to transform  $g(y, z) = 0$  explicit in  $y_i$ . This means that  $g(y, z) = 0$  can be transformed to  $y = \tilde{g}(z)$ . By substituting  $\tilde{g}(z)$  for  $y$  in Equation 2.12a

$$h(y, z) = 0 \quad (2.12a)$$

$$h(\tilde{g}(z), z) = 0 \quad (2.12b)$$

$$H(z) = 0 \quad (2.12c)$$

results in Equation 2.12c which is usually a much smaller system to solve. (Baharev et al., 2016a)

To minimise the number of variables that need to be guessed is the same as maximising the number of variables solved through assignment (making the  $g$  vector as large as possible). The assumption is usually made that a reduction in the number of variables that need to be guessed  $z$  will lead to a reduction in computational time. The time required to solve a system is dependent on many factors, but it would be too difficult to minimise solving time directly, hence the assumption. However, there no guarantee that this reduction will lead to lower computational time and it is even possible for it to lead to an increase in computational time (Baharev et al., 2016a).

### 2.5.1 Identifying feasible assignments

If equation  $i$  in  $f$  ( $f_i(x) = 0$ ) can be solved symbolically for variable  $j$  in  $x$  ( $x_j$ ) and the solution is unique, explicit and numerically stable then, and only then, can the equation variable pair  $(i, j)$  represent a feasible assignment. The more feasible equation variable pairs can be identified, the more flexibility there is in finding the optimal tearing order.

When identifying feasible assignment pairs Baharev et al. (2016b) follows a conservative approach and any assignment pair that does not satisfy the three conditions (unique, explicit and numerically stable) for a feasible assignment is not considered. (Baharev et al., 2016b)

#### Unique and explicit solutions

Baharev et al. (2016b) uses Sympy (Certik et al., 2008) to solve the equations symbolically. A variable is only considered as a feasible assignment if Sympy returns *one* explicit solution<sup>5</sup>. Consider Equation 2.13 for an example

$$x_1 + x_2 \times x_3 = 0 \tag{2.13}$$

When Equation 2.13 is solved for  $x_1$ ,  $x_2$  and  $x_3$ , respectively, Sympy will produce the results in Equation 2.14

$$x_1 = -x_2 \times x_3 \tag{2.14a}$$

$$x_2 = \frac{x_1}{x_3} \tag{2.14b}$$

$$x_3 = \frac{x_1}{x_2} \tag{2.14c}$$

All three the equations (Equation 2.14a to Equation 2.14c) are explicit and unique and the equation variable pairs are *considered* as candidates for feasible assignments (it needs to be proven that it is numerically stable) (Baharev et al., 2016b).

---

<sup>5</sup>Sympy is a mature symbolic-math library and if a closed form solution exists Sympy often finds it (Baharev et al., 2016b)

Not all equations that have an explicit solution are unique. Equation 2.15 is an example of such an equation

$$x_1^2 + x_1 \times x_2 - 1 = 0 \quad (2.15)$$

By solving Equation 2.15 for  $x_1$  will result in Equation 2.16

$$x_1 = -x_2 + \sqrt{x_2^2 + 1} \quad (2.16a)$$

$$x_1 = -x_2 - \sqrt{x_2^2 + 1} \quad (2.16b)$$

The two results in Equation 2.16 are explicit but are not unique because of the two possible solutions and thus the variable equation pair is not considered for a feasible assignment. (Baharev et al., 2016b)

Some equations do not have a closed form solution, or Sympy fails to find one (on a practical level it is the same as not having a closed form solution), and these equation variable pairs are also not considered as feasible assignments (conservative approach) (Baharev et al., 2016b).

### Identify numerically stable solutions

Examples of a numerically unstable solutions would be Equation 2.14b and Equation 2.14c, if  $x_2$  and  $x_3$  are allowed to be zero resulting in zero division. Baharev et al. (2016b) assumes that all the variables have reasonable (not big-M see section 2.3.4) lower and upper bounds. To check for numerically stable assignments Sympy's interval arithmetic implementation was used. Interval arithmetic is a cheap computational way to obtain a guaranteed lower and upper bound on the range of a function over the domain of its variables. Guaranteed here means that the true lower and upper bound (if a bound is found) will *always* be included in the result but the result is not necessarily tight, and the resulting lower and upper bound may be far from the true bounds. To illustrate this point three examples are considered (Baharev et al., 2016b):

Example 1 When the equation  $f(x_1, x_2) = \frac{x_1 - x_2}{x_1 + x_2}$  with  $x_1 \in [3, 9]$  and  $x_2 \in [1, 2]$  is evaluated with interval arithmetic the resulting range is  $[0.09, 2.0]$ . The true range is  $[0.2, 0.8]$ . The obtained range contains the true range but also over estimates it.

Example 2 When the equation  $f(x) = \frac{1}{x^2 - x + 1}$  with  $x \in [0, 1]$  is evaluated with interval arithmetic the resulting range is  $[0.5, \infty]$ . The true range is  $[1, 0.75]$  and again the true range is included but grossly over estimated but when  $f$  is rewritten as  $g(x) = \frac{4}{(2x-1)^2 + 3}$  and  $f(x) = g(x)$  evaluating  $g(x)$  with interval arithmetic obtains the true range. This shows the result of interval arithmetic is dependent on how a function is presented.

Example 3 When the equation  $f(x) = \log(x)$  with  $x \in [-1, 1]$  an exception is raised stating that “logarithm of a negative number” was encountered.

A solution is considered numerically stable if and only if the interval arithmetic evaluation did not raise any exceptions and resulting range  $r$  falls within the bound  $[-M, M]$  ( $r \subseteq [-M, M]$ ) with  $M = 10^{15}$  as a default bound. Therefore, as long as the variables are within their stated bounds performing a floating-point arithmetic will not fail due to zero division of illegal function arguments errors. (Baharev et al., 2016b)

## 2.5.2 Incidence matrix

Tearing algorithms almost always start with the incidence matrix or a representation thereof. The incidence matrix is essentially a mapping of what variables are present in each equation. The primary structural properties of a system of equations is located in the pattern of the non-zero entries, and an incidence matrix makes it possible to represent these entries (Barton, 1995).

Equation 2.18 (Barton, 1995) is an example of a system with five equations and five variables. Equation 2.17 is the corresponding  $5 \times 5$  incidence matrix of Equation 2.18.

$$[A] = \begin{array}{c} f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \end{array} \begin{array}{ccccc} x_1 & x_2 & x_3 & x_4 & x_5 \\ \left[ \begin{array}{ccccc} 1 & & & 1 & \\ & 1 & 1 & 1 & 1 \\ 1 & 1 & & 1 & \\ 1 & & & 1 & \\ 1 & & 1 & & 1 \end{array} \right] \end{array} \quad (2.17)$$

The rows of an incidence matrix represent the equation in the system, and the columns represent the variables. The matrix is filled by allocating a 1 to every index  $ij$  where the equation in row  $i$  contains within it the variable in column  $j$ . All the blank unallocated entries are zero and are called the zero entries.

$$f_1(\mathbf{x}) = x_1 + x_4 - 10 = 0 \quad (2.18a)$$

$$f_2(\mathbf{x}) = x_2^2 \times x_3 \times x_4 - x_5 - 6 = 0 \quad (2.18b)$$

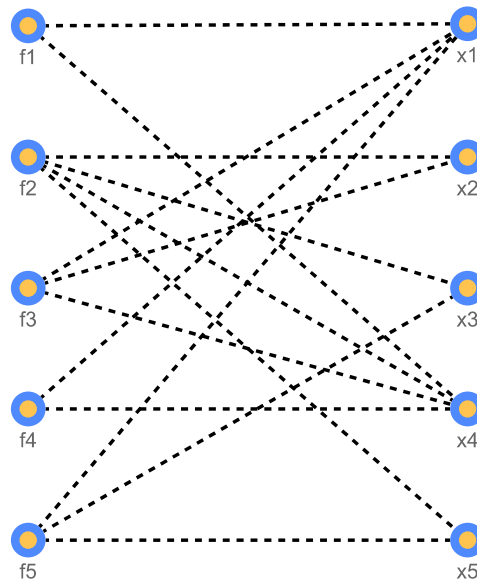
$$f_3(\mathbf{x}) = x_1 \times x_2^{1.7} \times (x_4 - 5) - 8 = 0 \quad (2.18c)$$

$$f_4(\mathbf{x}) = x_4 - 3x_1 + 6 = 0 \quad (2.18d)$$

$$f_5(\mathbf{x}) = x_1 \times x_3 - x_5 + 6 = 0 \quad (2.18e)$$

The incidence matrix can also be represented as a bipartite graph in Figure 2.9.

The graph is constructed by making a single vertex for all the rows and columns in Equation 2.17. Edges are created connecting all the row vertices to the column vertices if there is a non-zero entry (Barton, 1995).



**Figure 2.9:** The incidence matrix in Equation 2.17 represented in a bipartite graph.

### 2.5.3 Block triangular decomposition

The block lower triangular decomposition is a special case of the Dulmage-Mendelsohn decomposition when the incidence matrix is square and structurally non-singular (Baharev et al., 2016a). Amoss strictly performs simulations and requires that the incidence matrix is at least square. If the matrix is not square the DOF is not zero and the simulation may not continue.

The block lower triangular decomposition method uses the graph representation of the incidence matrix to identify subsystems of equations that must be solved simultaneously.

The first task is to identify an output set assignment. The output set assignment assigns a single variable to a unique equation. Take the equation  $x_1 = f(x_2)$  as an example. In this equation  $x_1$  is the output set assignment to the equation  $f(x_1, x_2) = 0$  (Barton, 1995). Equation 2.19 shows an output set assignment for the system of equations in Equation 2.18



$$\begin{matrix} & x_1 & x_2 & x_3 & x_4 & x_5 \\ f_1 & \textcircled{1} & & & 1 & \\ f_2 & & 1 & 1 & 1 & \textcircled{1} \\ f_3 & 1 & \textcircled{1} & & 1 & \\ f_4 & 1 & & & \textcircled{1} & \\ f_5 & 1 & & \textcircled{1} & & 1 \end{matrix} \quad (2.19)$$

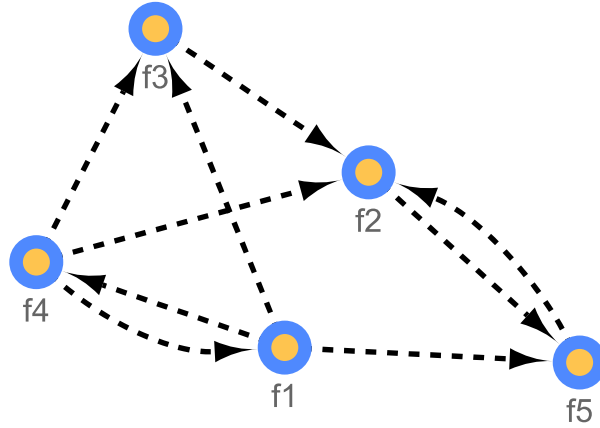
The output set assignment is not unique and in Equation 2.19 the assignment to  $f_1$  could also have been  $x_4$  and as a result the assignment to  $f_4$  will be  $x_1$ . The assignment to  $f_2$ ,  $f_3$  and  $f_5$  will remain the same.

A matching in the bipartite representation of the incidence matrix is similar to an output set assignment (Barton, 1995). Therefore, a maximal matching algorithm can be used to identify an output set assignment automatically. For this strategy to work the system of equations must adhere to two conditions: the number of equations and variables must be equal (fully determined system), and all the equations must be independent. When these conditions are met, it ensures that the maximal matching will also be a perfect matching which produces a valid output set assignment.

Once the output set assignment is obtained by obtaining the perfect matching in the bipartite graph an additional directed graph is created by the following steps (Barton, 1995):

1. Create a directed graph with a vertex for each row in the incidence matrix.
2. Start walking through the matched bipartite graph by staring at a row vertex and walking along the matched edge. This edge will lead to a column vertex. From that column vertex identify all the row vertices connected to the column vertex except the vertex connected through the matched edge.
3. In the directed graph add directed edges from the initial row vertex in step 2 to all the other row vertices identified in step 2.
4. Remove the matched edge from the bipartite graph.
5. Repeat steps 2-4 until there are no matched edges left in the bipartite graph.

Figure 2.10 shows the resulting directed graph of Equation 2.18. The directed graph is used to identify the strongly connected components. The order to solve the subsystems of equations is from the least strongly connected components to the most strongly connected components. All the vertices in strongly connected component set need to be solved simultaneously. (Barton, 1995)



**Figure 2.10:** Directed graph created from Figure 2.9 using the list described above.

For Figure 2.10 the strongly connected components listed from least to most is  $\{f_1, f_4\}$ ,  $\{f_3\}$  and  $\{f_2, f_5\}$ . This means that  $f_1, f_4$  should be solved simultaneously first then  $f_3$  and then  $f_2, f_5$  are solved simultaneously afterwards. The permuted incidence matrix from Equation 2.17 is shown in Equation 2.20

$$\begin{aligned}
 [B] = & \begin{array}{c} f_1 \\ f_4 \\ f_3 \\ f_5 \\ f_2 \end{array} \begin{array}{ccccc} & x_1 & x_4 & x_2 & x_3 & x_5 \\ \left[ \begin{array}{cccccc} \textcircled{1} & 1 & & & & \\ 1 & \textcircled{1} & & & & \\ 1 & 1 & \textcircled{1} & & & \\ 1 & & & & \textcircled{1} & 1 \\ & 1 & 1 & 1 & & \textcircled{1} \end{array} \right] & \end{array} \quad (2.20)
 \end{aligned}$$

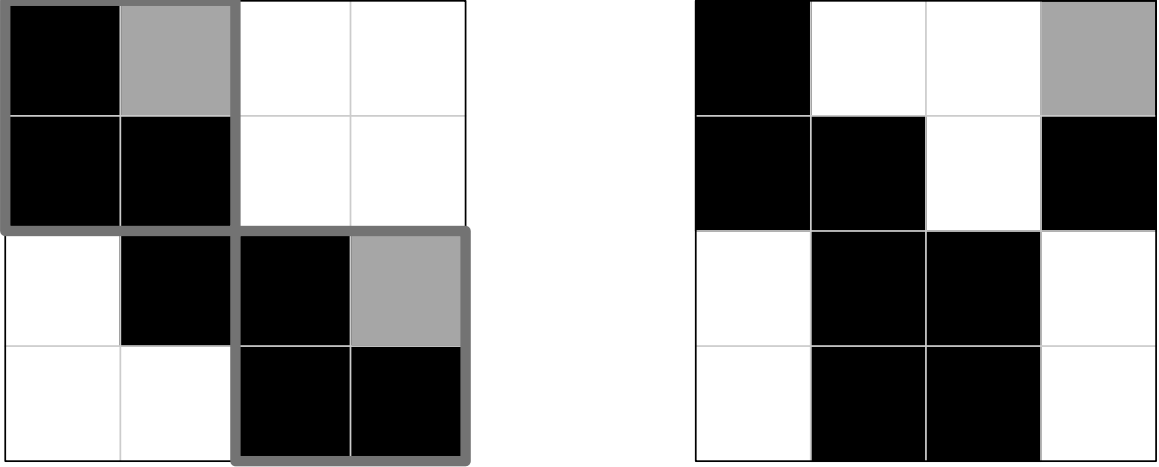
### 2.5.4 Bordered lower triangular tearing

It is common practice to perform a block lower decomposition first and then apply tearing to the irreducible<sup>6</sup> blocks on the diagonal, but this can lead to suboptimal results. It is possible to achieve better results by performing tearing directly to the original incidence matrix (Baharev et al., 2016a).

Figure 2.11 shows an example of a small system that was torn by block lower triangular decomposition first and then tearing of the irreducible blocks against an optimal tearing algorithm performed directly on the original incidence matrix. Figure 2.11 displays the incidence matrices in an alternative way where all the coloured blocks represent a 1 in the incidence matrix and the grey blocks are the variables that needs to be guessed. Notice that in Figure 2.11 optimal tearing requires only one variable to be guessed to solve the system whereas the block lower triangular tearing requires two variables.

Baharev et al. (2016b) developed an optimal tearing as well as a custom branch and

<sup>6</sup>Irreducible in this section means that the blocks from the block lower decomposition cannot be reduced further by applying the Dulmage-Mendelsohn decomposition.



**Figure 2.11:** Left: Block lower triangular decomposition first and then tearing on the irreducible blocks. Right: Direct tearing of the incidence matrix. The variables that need to be guessed are shown in grey. (Baharev et al., 2016a)

bound algorithm which performs tearing on the original incidence matrix. Baharev et al. (2016b) implemented the algorithms in Python (Baharev, 2017a).

### Optimal tearing with integer programming by Baharev et al. (2016b)

The optimal integer programming tearing algorithm requires the bipartite representation of the incidence matrix  $B = (V, E)$  as well as the matrix itself. Refer back to subsection 2.5.2 which explains how to represent the incidence matrix as a bipartite graph. Let  $F \subseteq E$  be the subset of edges containing all the feasible assignment pairs (see subsection 2.5.1).

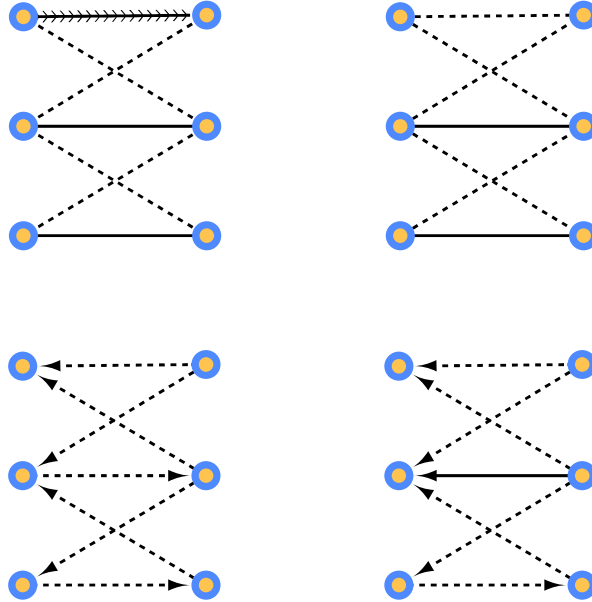
The main idea behind the optimal integer programming tearing algorithm is broken down into the following steps with reference to Figure 2.12:

**Maximal matching:** Find a maximal matching  $M'$  of  $B$  (the solid lines represents matched edges in Figure 2.12). The arrow  $\Rightarrow$  in Figure 2.12 represents an infeasible matched edge.

**Feasible assignments:** Remove the edges in  $M'$  that are not in  $F$  to get the matching  $M$  ( $M = M' \cap F$ ) Baharev et al. (2016a).

**Orientation:**  $B$  is orientated with the edges in  $M$  pointing to the variable nodes and all the other edges towards the equation nodes resulting in the directed graph  $D$  (Baharev et al., 2016a). The variable nodes set is represented by  $C \subset V$  and the equation nodes by  $R \subset V$  with  $C \cap R = \emptyset$  and  $V = R \cup C$ , i.e.  $V$  is partitioned by  $C$  and  $R$ .

**Feedback edge set:** By solving the feedback edge set problem the variables to tear are identified. The purpose of the feedback edge set problem is to identify the minimum amount of edges to reverse (see the solid arrow in Figure 2.12 bottom right) (Baharev et al., 2016a). Go to section 2.5.4 to see how the cycles are created and how an acyclic graph is obtained.



**Figure 2.12:** The steps of tearing: Top left a maximal matching  $M'$   $\rightarrow$  top right a matching  $M$  after removing infeasible edges  $\rightarrow$  bottom left create a directed graph  $\rightarrow$  bottom right solve the feedback edge set problem.

### Feedback edge set problem

The feedback edge set problem will be explained using the directed graph  $D$  from the orientation of  $B$  in the optimal tearing algorithm above.

The way  $B$  is orientated given the matching  $M$  induces a certain cycle structure in  $D$ . An unmatched node  $c \in C$  is a source and unmatched node  $r \in R$  is a sink. Sources and sinks may not be part of a cycle in  $D$  but only the matched nodes of  $M$ . The matched edge of a node  $n$  must partake in all the cycles  $n$  is involved in and by reversing this edge and only this edge must destroy all the cycles  $n$  is involved in as well. Reversing a matched edge  $(u, v) \in M$  of  $D$  effectively removes the edge  $(u, v)$  from  $M$  and  $D$  and adds the new edge  $(v, u)$  to  $D$ .  $M - (u, v)$  is still a matching, and  $u$  and  $v$  becomes a source and sink accordingly (Baharev et al., 2016a).

The subset of edges  $T \subseteq M$  that need to be reversed to make  $D$  acyclic is called a feedback edge set. The minimum feedback edge set problem is the problem to obtain the smallest feedback edge set  $T$  as possible. (Baharev et al., 2016a)

## Integer linear program

The integer program used by Baharev et al. (2016b) to maximise the number of matches  $M$  but ensuring  $D$  is acyclic is given in Equation 2.21 (Baharev et al., 2016b)

$$\max_y \sum_{e \in F} y_e \quad (\text{find the maximal matching}) \quad (2.21a)$$

$$\text{s.t. } \sum_{e \in E} u_{re} y_e \leq 1 \text{ for each } r \in R \quad (\text{equation node may only be matched once}) \quad (2.21b)$$

$$\sum_{e \in E} v_{ce} y_e \leq 1 \text{ for each } c \in C \quad (\text{each variable node may only be matched once}) \quad (2.21c)$$

$$\sum_{e \in E} a_{se} y_e \leq \frac{\ell_s}{2} - 2 \text{ for each } s \in S \quad (\text{no cycles are allowed}) \quad (2.21d)$$

$y_e$  is a binary variable which is 1 if the edge  $e \in M$  and 0 otherwise.  $u_{re}$  is a binary variable and is 1 if node  $r$  is incident to edge  $e$  and 0 otherwise.  $v_{ce}$  is a binary variable which is 1 if node  $c$  is incident to edge  $e$  and 0 otherwise.  $S$  is a set of simple cycles in the cycle matrix  $A = (a_{se})$  (see (Baharev et al., 2016a)). The entry  $a_{se}$  is 1 if the edge  $e$  participates in a simple cycle  $s$  and is 0 otherwise.  $\ell_s$  is the length of the simple cycle  $s$ . (Baharev et al., 2016b)

When the acyclic graph  $D$  is obtained, the system can be ordered in a bordered lower triangular form.

## Custom branch and bound by Baharev et al. (2016b)

To understand the custom branch and bound algorithm the heuristic ordering algorithm of Fletcher & Hall (1993) (which was also the inspiration behind the custom branch and bound) to the lower Hessenberg form is first studied.

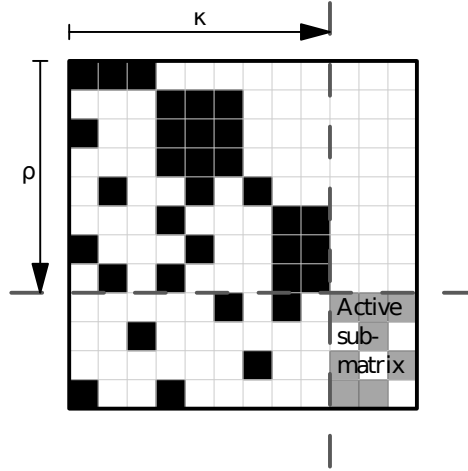
The lower Hessenberg form also results in a block lower triangular matrix, but instead of having square blocks on the diagonal it has fully dense rectangular blocks.

## Algorithm by Fletcher & Hall (1993)

The matrix  $A \in \mathbb{R}^{n \times n}$  is a square irreducible incidence matrix. The algorithm to permute  $A$  to the Hessenberg form searches the active sub-matrix and progressively removes rows and columns based on a heuristic to determine an ordering (Baharev et al., 2016b).

The active sub-matrix is a sub-matrix of  $A$  containing rows and columns that have not been included in the permutation. It is referred to as active because it is in this matrix where further permutations take place. At the start of the algorithm the whole matrix

$A$  is the active sub-matrix and when the algorithm terminates the active sub-matrix is empty. Figure 2.13 shows the location of the active sub-matrix of a partially permuted  $A$ .



**Figure 2.13:** Partially reordered matrix when applying the incomplete permutation  $\pi = (\rho, \kappa)$  (Baharev et al., 2016b).

The removed rows and columns are assembled in the permutation vectors  $\rho$  and  $\kappa$  respectively. The pair  $\pi = (\rho, \kappa)$  is referred to as the incomplete permutation of  $A$ . When  $\pi$  is known the active sub-matrix is also known. It is from this fact that the notation  $r_i(\pi)$  and  $c_j(\pi)$  refers to the number of non-zero entries in row  $i$  and column  $j$  of the *active sub-matrix*, respectively (Baharev et al., 2016b).

Figure 2.14 (Baharev et al., 2016b) shows the algorithm by Fletcher & Hall (1993) to order a square non-singular irreducible incidence matrix  $A$ .

**A as the active sub-matrix** turns the entire incidence matrix  $A$  to the active sub-matrix.

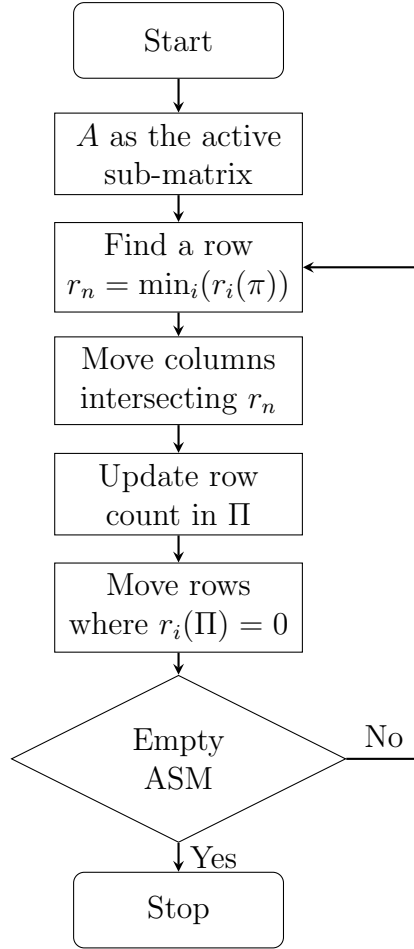
**Find a row  $r_n = \min_i(r_i(\pi))$**  in the active sub-matrix which has the lowest number of non-zero entries. This is the heuristic of this algorithm.

**Move columns intersecting  $r_n$**  to the left and consider them as removed.

**Update row count in  $\Pi$**  updates the row counts in the active sub-matrix taking the removed columns into account and is represented by the incomplete permutation  $\Pi$ .

**Move rows where  $r_i(\Pi) = 0$**  in the active sub-matrix to the top and consider them as removed.

**Empty ASM** checks if the active sub-matrix (ASM) is empty. If it is empty the algorithm terminates otherwise the algorithm repeats until the active sub-matrix is empty.



**Figure 2.14:** The heuristic Hessenberg ordering by Fletcher & Hall (1993).

The algorithm by Fletcher & Hall (1993) is for a square non-singular irreducible incidence matrix  $A$ . Baharev et al. (2016b) proposes the following extension to accommodate rectangular and reducible full rank  $m \times n$  incidence matrices to the lower Hessenberg form: If  $m \geq n$   $A$  is ordered into the form  $A = \begin{bmatrix} A_T \\ A_B \end{bmatrix}$  where  $A_T \in \mathbb{R}^{n \times n}$  is structurally non-singular and in the Hessenberg form. If  $m \leq n$   $A$  then  $A$  is ordered into the form  $A = \begin{bmatrix} A_L & A_R \end{bmatrix}$  where  $A_R \in \mathbb{R}^{m \times m}$  is structurally non-singular and in the Hessenberg form. Note that the  $A_T$  and  $A_R$  do not need to be irreducible but only require a structural full rank.

#### Algorithm proposed by Baharev et al. (2016b)

The algorithm is very similar to that of Fletcher & Hall (1993) but instead of using a heuristic to determine the tiebreaks it explores all the possibilities in a branch and bound algorithm to reduce a cost function. The custom branch and bound algorithm can also handle a rectangular incidence matrix  $A \in \mathbb{R}^{m \times n}$  as long as it has full structural column rank when  $m \geq n$  and row rank when  $m \leq n$ . For this section it is assumed that all the equations in  $A$  can be symbolically transformed into an assignment with any of its variables and a function containing the remaining variables.

Consider the following definition:

**Row elimination** is the process of solving the equations for one of its variables represented by the corresponding row in the incidence matrix.

The algorithm is based on the cost of a row elimination. The cost associated by eliminating a row  $i$  (row cost of row  $i$ ) is the number variables left when row  $i$  is eliminated. Equation 2.22 expresses the row cost of row  $i$  in terms of the non-zero entries in row  $i$

$$z_i = \max(0, r_i(\pi) - 1) \quad (2.22)$$

Due to the assumption that all variables in  $A$  are valid eliminations it means that at most the number of non-zero entries minus one ( $r_i(\pi) - 1$ ) variables need to be guessed if that row is eliminated.

The cost of a particular incomplete permutation  $\pi = (\rho, \kappa)$  is given in Equation 2.23

$$z = z(\rho) = \sum_i z_i; \quad i \in \rho \quad (2.23)$$

The cost is simply the sum of all the eliminated rows in  $\kappa$ . The algorithm seeks to find the permutation  $\pi$  that transforms  $A$  into the lower Hessenberg form with the lowest cost.

### Lower bounds based on the cost function using row and column count

Equation 2.24 gives a lower bound on the cost for the best possible complete permutation possible given an incomplete permutation  $\pi$

$$\underline{z}(\rho) = z(\rho) + \min_i(z_i) \quad (2.24)$$

The lower bound rests on the fact that the cost  $z(\rho)$  is fixed and at least  $\min_i(z_i)$  variables need to be guessed in order to continue with the elimination. Therefore, if all the remaining eliminations have zero cost the minimum cost for the complete permutation is  $\underline{z}(\rho)$ . This means that the lower bound is sharp.<sup>7</sup>

For an  $m \geq n$  incidence matrix  $A$  the optimal cost of a *complete* permutation  $z^*$  can not be lower than the row with the minimum cost as shown in Equation 2.25

$$z^* \geq \min_i(r_i(\pi) - 1) \quad (2.25)$$

where  $i \in$  row indices of  $A$ . A result from Equation 2.25 is that any non-singular square matrix  $A$  will have at least one guessed variable if the minimum row count is equal or

---

<sup>7</sup>The best possible lower bound that can exist.



larger than two ( $\geq 2$ ). Stated otherwise: it is impossible to order  $A$  into a lower triangular form when the minimum row count is  $\geq 2$ .

For an  $m \leq n$  incidence matrix  $A$  a minimum of  $n - m$  variables need to be guessed for any permutation. When  $A$  is underdetermined, the goal is to assign as many variables as possible to the  $m$  available equations. A square non-singular  $m \times m$  sub-matrix  $A_R$  with  $A = [A_L \ A_R]$  is therefore constructed with the objective of finding  $A_R$  that will result in the lowest possible permutation cost of  $A_R$ .  $A_L$  is an  $m \times (n - m)$  sub-matrix which contains the remaining DOF and these variables will have to be guessed in any case. To find the best  $A_R$  the ordering is based on a column cost. Ordering in this fashion stays the same as the algorithm by Fletcher & Hall (1993), but instead of row count ( $r_i(\pi)$ ) column count ( $c_j(\pi)$ ) is used and the ordering starts from the bottom right to the top left of  $A_R$ .

If the last column in a full rank Hessenberg ordering of  $A_R$  has  $c_l$  non-zero entries the permutation cost of  $A_R$  is at least  $c_l - 1$  (see (Baharev et al., 2016b) for the proof). A lower bound on  $c_l$  is given in Equation 2.26

$$c_l \geq \min_j (c_j(\pi)) \quad (2.26)$$

where  $j \in$  column indices of  $A$ . Notice that the lower bound of  $c_l$  is calculated over the entire  $A$  and not just  $A_R$ . This is because the lowest column count in  $A_R$  can not be lower than the lowest column count in  $A$ . As a result of the lower bound on  $c_l$  a lower bound on  $z^*$  is calculated in Equation 2.27

$$z^* \geq \underbrace{\min_j (c_j(\pi) - 1)}_{\text{lower bound on cost}} + \underbrace{(n - m)}_{\text{remaining DOF}} \quad (2.27)$$

with  $j \in$  column indices of  $A$  and  $m \leq n$ .

The lower bounds in Equation 2.25 and Equation 2.27 are sharp because it is possible that after removing the row (when  $m \geq n$ ) or column (when  $m \leq n$ ) that the remaining permutations can have zero cost.

### Column slice relaxation

Column slice relaxation is another method to obtain a lower bound on the cost of a complete permutation  $z^*$ . The previous section used row and column counts to derive the lower bounds whereas column slice relaxation finds a lower bound by the relaxation of partitioned columns.

Let  $\tilde{A}$  be an arbitrary column slice of  $A$ . The columns included in  $\tilde{A}$  must either be eliminated variables or guessed variables and must include all the rows that can be used for elimination. A lower bound for  $z^*$  can be calculated by accumulating the lower

bounds of  $z$  for each column slice in  $A$ . All the columns not in  $\tilde{A}$  will only pose further constraints on the lower bound  $z$  of  $\tilde{A}$  but it is ignored or *relaxed* for the calculation of  $z^*$ .

The usefulness of the lower bound  $z^*$  calculated using column relaxation is greatly dependent on the choice of the partitioning of  $\tilde{A}$ . Baharev et al. (2016b) found the best results when walking the upper envelope<sup>8</sup> of the Hessenberg form by stepping either right or down and moving along a direction as far as possible before changing direction. The matrix is partitioned horizontally and vertically whenever two or more ( $\geq 2$ ) steps in the right direction is possible. The partitioning takes place at these places after the first step right and effectively cuts off the columns which cause the row cost to be larger than one ( $z_i \geq 1$ ).

### Distinguishing features

Features that make the branch and bound algorithm by Baharev et al. (2016b) *custom* is discussed. The branch and bound algorithm uses the lower bounds on the cost function utilising the row and column count as well as column relaxation to determine the best tie breaks. Baharev et al. (2016b) lists the following points that set their algorithm apart from others:

1. The search tree is navigated by a depth-first search order.
2. When branching the best-first rule is used. The node with the lowest score ( $\underline{z}(\rho)$  in Equation 2.24) is explored first, breaking ties arbitrarily. A min-priority queue was implemented to explore the lowest costing rows first.
3. An initial feasible solution is calculated using the algorithm by Fletcher & Hall (1993) before the branch and bound starts. A lower bound on the optimal cost  $z^*$  is calculated using Equation 2.25 and Equation 2.27.
4. When a new complete permutation is found (a leaf node<sup>9</sup> is reached by the branch and bound algorithm) column relaxation is run on the permutation to improve the lower bound.
5. The algorithm keeps track of all the trailing sub-matrices that have been fully discovered by the depth-first search. Whenever an active sub-matrix is encountered that has previously been discovered the optimal ordering and cost is simply retrieved by a bookkeeping method that was implemented.
6. The “back-track rule” of Hernandez & Sargent (1979) was implemented to discard entire sub-trees in the branch and bound procedure by excluding sequences that

---

<sup>8</sup>Walking the lower Hessenberg form from the top left to the bottom right.

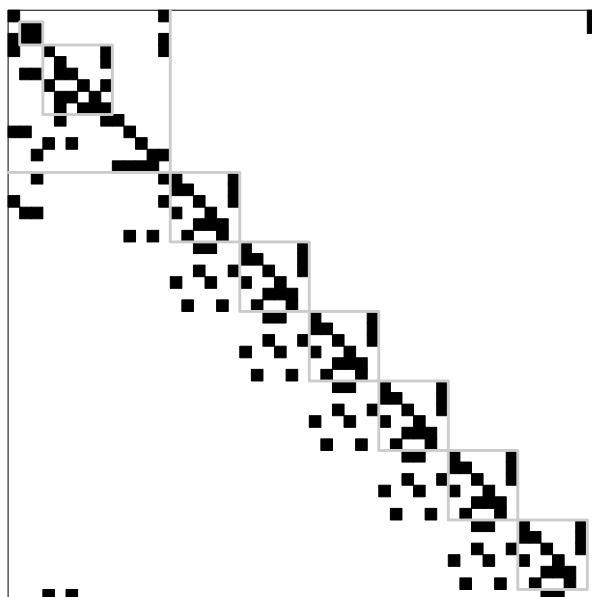
<sup>9</sup>A node with no child nodes. Just like a leaf which is at the end of a branch a leaf node is also at the end of a branch in a graph.

have strict lower bounds on  $z^*$  that are higher than already found orderings. See (Hernandez & Sargent, 1979) for more info.

7. It is possible for the bipartite graph of the incidence matrix to become disconnected. Whenever this happens, the connected components are processed separately.

### 2.5.5 Alternative ordering

An alternative to the bordered lower triangular ordering is the spiked lower triangular ordering. Figure 2.15 is an example of an incidence matrix ordered to the spiked lower triangular form. The spiked lower triangular form is nearly lower triangular with some columns having entries above the diagonal. These columns are called spiked columns. In this form the non-spiked columns must have a non-zero entry on the diagonal however spiked columns are allowed to have a zero entry. (Baharev et al., 2016a)



**Figure 2.15:** Example of a spiked lower triangular ordered incidence matrix (Baharev et al., 2016a).

In Figure 2.15 a block is drawn around each spike. The block starts at the highest point of the spike and is extended left and down until the diagonal is reached and the square closed. These blocks indicate which rows (equations) belong to that particular spike. The spiked lower triangular form also has the following distinguishing feature: for any two given spiked columns (left and right) the rows belonging to the left spike is either contained within the right spike or is completely disjoint from the right. This requirement means that smaller bordered lower triangular forms can be created on the diagonal at the spikes where the blocks are properly nested (the one spike completely contains the other) or disjoint (Baharev et al., 2016a).

The advantage of this form above the conventional bordered lower triangular form is that it is possible to solve smaller system of equations sequentially instead of solving all the guessed variables in a single block. Smaller systems are easier to solve and it is expected that the spiked lower triangular form may be more computationally efficient.

### 2.5.6 Failure of tearing

When using tearing to solve a system of equations, it is possible to obtain completely incorrect results, or it is possible that solving a system of equations without tearing is solved quicker than applying tearing.

The reason for these failures can be due to the residual equation of the guessed variable being too sensitive or too insensitive to the guessed variable. Below these two failures will be discussed together with an example.

#### Too sensitive

As an example consider the system in Equation 2.28

$$x_{i-1} + 10x_i + x_{i+1} = 1.2 \quad i = 1, 2, \dots, 20 \quad (2.28)$$

where  $x_0 = 0.1$  and  $x_{21} = 0.1$ . The only tear variable is  $x_1$  with the last equation ( $i = 20$ ) the residual. In the bordered lower triangular form  $x_1$  is in the border and the rest of the incidence matrix is lower triangular. The solution to the system of equations is  $x_i = 0.1$  for  $i = 1, 2, \dots, 20$ . Baharev et al. (2017) reported that the solvers Dymola and OpenModelica returned completely incorrect results for  $x_{20}$  with Dymola returning  $x_{20} = 32.03$  and OpenModelica  $x_{20} = 85.82$ .

In this example the failure occurred because residual equation is too sensitive to  $x_1$  (the tear variable). Given an initial guess for  $x_1$  the equations in the lower triangular section can be substituted one into the other as shown in Equation 2.29

$$x_{i+1} = -x_{i-1} - 10x_i + 1.2 \quad i = 1, 2, \dots, 19 \quad (2.29)$$

with the residual  $r = -x_{19} - 10x_{20} + 1.1$  a univariate function of  $x_1$  ( $r(x_1) = 0$ ). Due to the factor 10 in Equation 2.29 the error in the guess of  $x_1$  is also roughly multiplied by 10 for every substitution in Equation 2.29. There is a total of 19 substitutions in Equation 2.29 which mean that an error in  $x_1$  will be roughly magnified by a factor of  $10^{19}$  by the time the residual is calculated. This is a catastrophic consequence resulting from the tearing of Equation 2.28. Using a 64-bit floating-point number to represent  $x_1$  the two closest numbers enclosing 0.1 will result in  $x_{20} = 85.82$  and  $x_{20} = -101.03$  after substitution in Equation 2.29. This large error in  $x_{20}$  is a consequence of the roughly

$10^{19}$  magnification of the very small error in  $x_1$  and therefore Equation 2.28 is unsolvable with a 64-bit machine (Baharev et al., 2017).

### Too insensitive

This example is the polar opposite of the example in Equation 2.28. Consider the system in Equation 2.30

$$x_{i-1} + x_i + 15x_{i+1} = 17 \quad i = 1, 2, \dots, 20 \quad (2.30)$$

where  $x_0 = 1$  and  $x_{21} = 1$ . Again  $x_1$  is the tear variable with the last equation ( $i = 20$ ) the residual. In the bordered lower triangular form  $x_1$  is in the border and the rest of the incidence matrix is lower triangular. The solution to the system of equations is  $x_i = 1$  for  $i = 1, 2, \dots, 20$ . Baharev et al. (2017) reported that the solvers Dymola and OpenModelica failed with Dymola not producing any results and OpenModelica giving warnings and returns the initial guess for  $x_1$  (Baharev et al., 2017).

In this example failure occurred because the residual equation is too insensitive to  $x_1$  (the tear variable). Given an initial guess for  $x_1$  the equations in the lower triangular section can be substituted one into the other as shown in Equation 2.31

$$x_{i+1} = \frac{1}{15}(-x_{i-1} - x_i + 17) \quad i = 1, 2, \dots, 19 \quad (2.31)$$

with the residual  $r = -x_{19} - x_{20} + 2$  a univariate function of  $x_1$  ( $r(x_1) = 0$ ). The error for  $x_1$  in the initial guess is divided by 15 for each substitution in Equation 2.31. As a consequence the magnitude of the error is reduced at every substitution. This results in the residual being equal to  $r = 0.0000000000$  (10 decimal points) for either  $x_1 = -1$  or  $x_1 = 3$ . The tearing in this example failed because the residual that is used to update  $x_1$  does not provide any useful information on the error of  $x_1$  (Baharev et al., 2017).

## 2.6 Non-uniform random variable generation

### 2.6.1 Probability functions

#### Continuous

A probability density function  $f(x)$  describes how the probabilities of a continuous random variable  $X$  is distributed across  $X$ . A probability density function of a random variable  $X$  is characterised by the following:

1.  $f(x) \geq 0$  Probability is a positive number.
2.  $\int_{-\infty}^{\infty} f(x)dx = 1$
3.  $P(a \leq X \leq b) = \int_a^b f(x)dx$  for  $a, b \in \mathbb{R}$

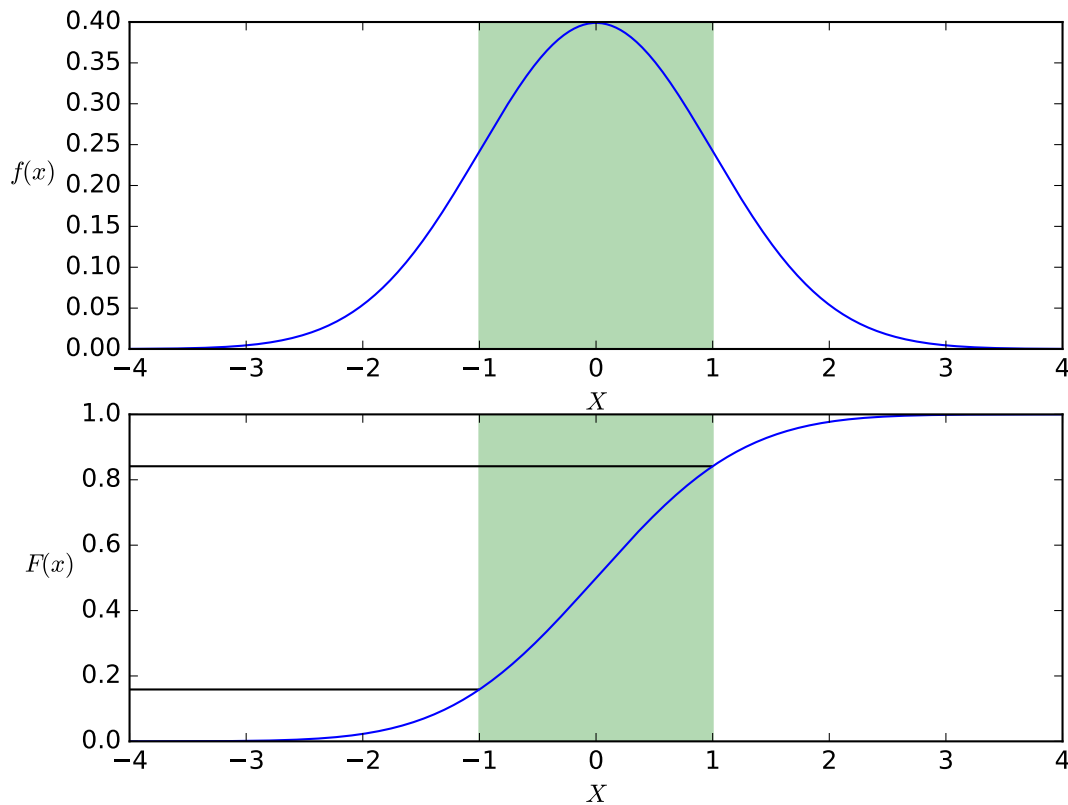
(Montgomery & Runger, 2011: 109)

The cumulative distribution function is calculated in Equation 2.32

$$F(x) = P(X \leq x) = \int_{-\infty}^x f(u)du \text{ for } x \in \mathbb{R} \quad (2.32)$$

(Montgomery & Runger, 2011: 112)

In Figure 2.16 is a normal probability density and cumulative distribution function. Notice that the region with the highest probability in the probability density function in Figure 2.16 (highlighted from  $-1$  to  $1$ ) corresponds to the region with the steepest slope in the cumulative distribution function. This fact will help in the later understanding for generating non-uniform random variables from a given cumulative distribution functions.



**Figure 2.16:** Probability density function of a normal distribution (top) and the corresponding cumulative distribution function (bottom).

## Discrete

A probability distribution describes how the probabilities of a discrete random variable  $X$  is distributed across  $X$ . For discrete random variables, the distribution is often specified in a simple list of possible values along with its probability. Just like the continuous case,

the distribution can be described by a function known as the probability mass function which has the following characteristics:

1.  $f(x_i) \geq 0$  Probabilities are positive.
2.  $\sum_{i=1}^n f(x_i) = 1$  The sum of all the discrete probabilities equals to one.
3.  $f(x_i) = P(X = x_i)$  The evaluation of  $f(x_i)$  is the probability that the discrete random variable  $X = x_i$ .

(Montgomery & Runger, 2011: 68)

The cumulative distribution function is calculated in Equation 2.33

$$F(x) = P(X \leq x) = \sum_{x_i \leq x} f(x_i) \text{ for } x \in X \quad (2.33)$$

(Montgomery & Runger, 2011: 72)

Figure 2.17 shows the binomial probability mass and cumulative distribution function. Similar to the continuous case notice that the region in the probability mass function containing the three random variables with the highest probabilities (4, 5, 6) corresponds to the largest discontinuity jumps in the cumulative distribution function.

## 2.6.2 Generation from a probability function

In stochastic simulations, a source of non-uniform random variables is required. According to Devroye (1986: 28) a non-uniform variable can be created given a uniform random variable generator and a probability density or mass function. To generate these random numbers the cumulative distribution is required.

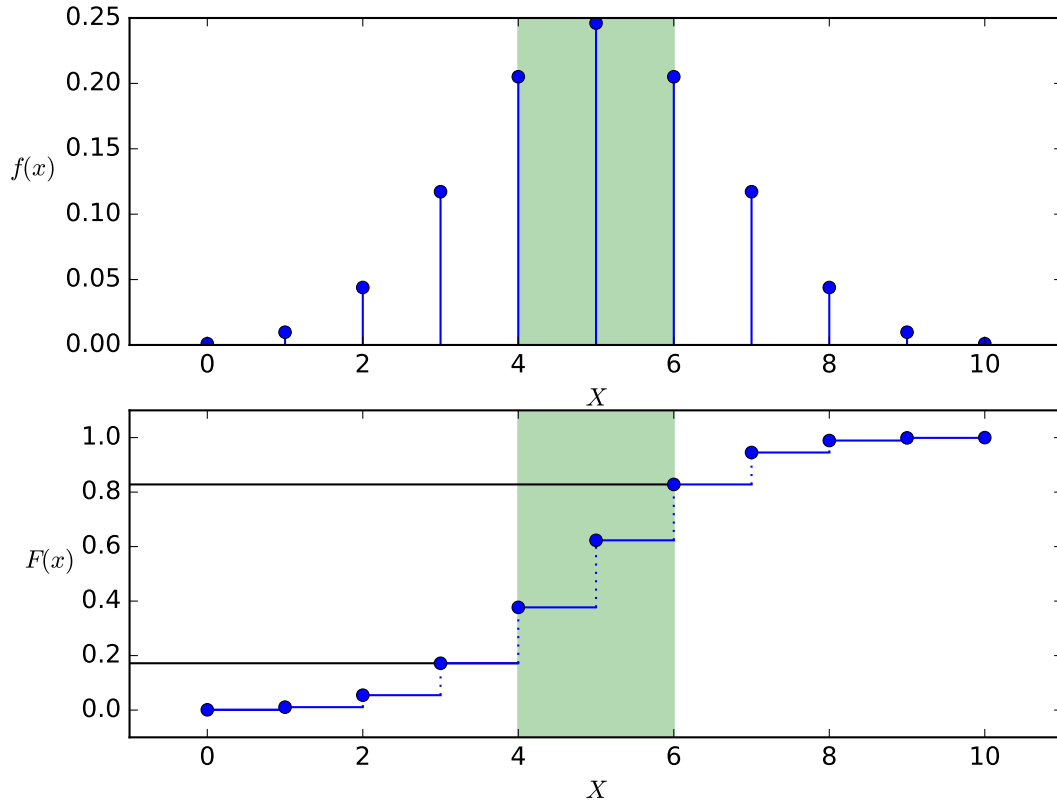
Consider a continuous distribution function  $F$  on  $\mathbb{R}$  the inverse  $F^{-1}$  is defined by Equation 2.34

$$F^{-1}(u) = \inf \{F(x) = u, 0 < u < 1\} \quad (2.34)$$

If  $U$  is uniform random variable on  $[0, 1]$  then  $F^{-1}(U)$  has a distribution function  $F$ . If  $X$  has a distribution function  $F$  then  $F(X)$  is also uniformly distributed on  $[0, 1]$ . (Devroye, 1986: 28)

Equation 2.34 can be used to generate random variable with any arbitrary continuous distribution function  $F$  (Devroye, 1986: 28) and by definition any probability density function where  $F$  can be calculated. A random variable is created by generating a uniform random variable  $U$  and calculating a non-uniform variable  $X$  using  $F^{-1}$ .

The use of the infimum in Equation 2.34 is important to make  $F^{-1}$  valid in all cases of  $F$ . The use of the infimum is easier explained by example. Figure 2.18 is an arbitrary probability density function where there is a region of zero probability. Figure 2.19 shows the cumulative distribution and its inverse of Figure 2.18.



**Figure 2.17:** Probability mass function of the binomial distribution (top) and the corresponding cumulative distribution function (bottom) with  $p = 0.5$  and  $n = 10$ .

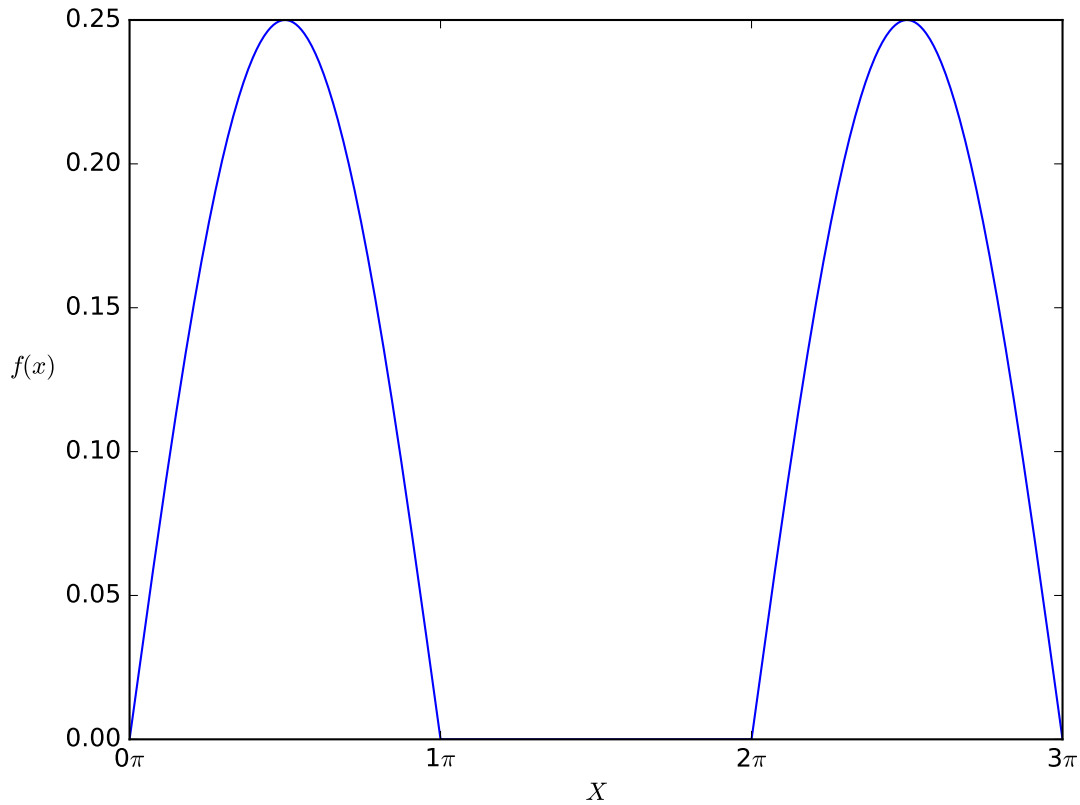
Notice that the cumulative distribution in Figure 2.19 has a region of zero slope (highlighted) and in the inverse corresponds to an infinite slope at 0.5. This is why the infimum is required which will produce a single value when  $F^{-1}(0.5)$  is evaluated. For  $F^{-1}(0.5)$  in Figure 2.19 the set of values  $Z = \{x \in \mathbb{R}, \pi \leq x \leq 2\pi\}$  is possible and the  $\inf \{Z\} = \pi$  which will ensure that the values of zero probability will not be generated.

The generation of non-uniform random variables for a discrete probability mass function is a little different but follows the same logic. If  $U$  is a uniformly generated random variable on  $[0, 1]$ ,  $X$  can be obtained by a monotone transformation of  $U$  such that  $P(X = i) = p_i$ . If  $X$  is defined by Equation 2.35

$$F(X - 1) = \sum_{i < X} p_i < U \leq \sum_{i \leq X} p_i = F(X) \quad (2.35)$$

then  $P(X = i) = F(i) - F(i - 1) = p_i$ .





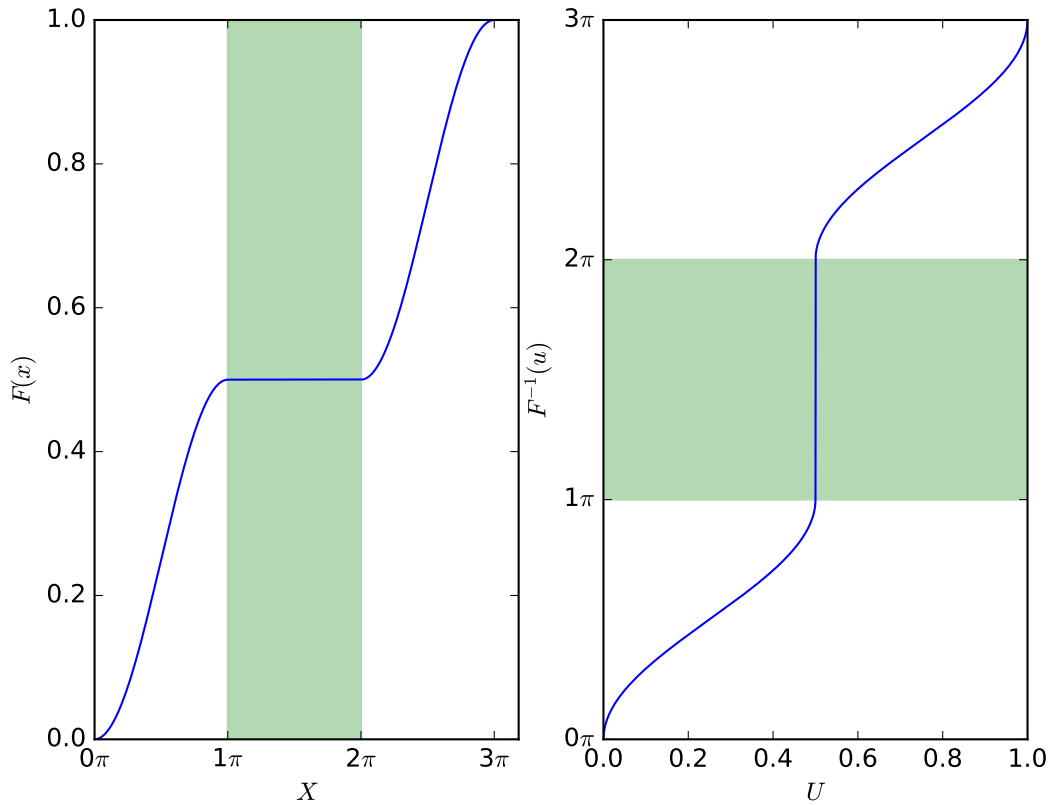
**Figure 2.18:** Probability density function where there is a region of zero probability in-between regions which has a probability.

### Graphical interpretation

The graphical interpretation to generate non-uniform random numbers is discussed. Figure 2.20 is the normal cumulative distribution function used in section 2.6.1. Given a uniform random variable  $U$  it is possible to graphically obtain the non-uniform random variable  $X$ .

With reference to Figure 2.20 the random variable  $X$  is found graphically by first sampling the random variable  $U$  (for Figure 2.20  $U = 0.6$ ). Then locate the position of  $U$  on the  $y$ -axis of the cumulative distribution function. Next, draw a horizontal line from the  $y$ -axis in the positive  $x$ -direction until the curve of the cumulative distribution function is reached. The corresponding  $x$ -value at this point is the generated random variable  $X$ .

The graphical explanation of why it is possible to generate a non-uniform random variable given a uniform random variable is due to the change in slope of the cumulative distribution function. Refer back to Figure 2.16 and see that a high probability in the density function relates to a steep slope in the cumulative distribution and this steep slope causes that a small range of  $X$  in the density function accounts for a large range



**Figure 2.19:** Probability distribution function (left) and its inverse (right) of Figure 2.18.

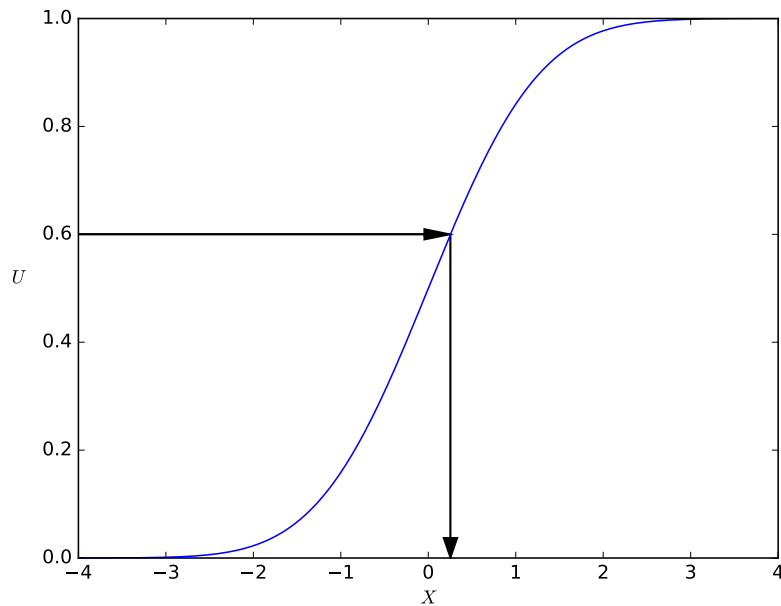
in  $F(x)$ . This result illustrates how the uniform random variable produces non-uniform random variables.  $U$  has the same probability to be anywhere on the  $y$ -axis but because the  $X$  values with the higher probability account for a larger range on the  $y$ -axis the higher probability  $X$  values are generated more frequently.

The discrete case in Figure 2.17 acts very similarly to the continuous case, but the higher probability  $X$  values correlate to higher discontinuous jumps in the cumulative distribution function which also accounts for a higher range in  $F(x)$ . Therefore, the higher probability  $X$  values are generated more frequently.

## 2.7 Algorithmic- or automatic differentiation (AD)

Algorithmic differentiation (also referred to as automatic differentiation) is a technique which numerically calculates values to evaluate derivatives with comparable accuracy and efficiency (Griewank, 2000: 2) with evaluation time a small multiple larger than the time to evaluate the original set of equations (Griewank, 2000: 9).

Before we look at how AD evaluates derivatives let's first establish what AD is and what it is not. AD is not a numerical (using numerical in the conventional sense) method to de-



**Figure 2.20:** Graphical method to find a non-uniform random variable  $X$  given a uniform variable  $U$  and a cumulative distribution function.

termine derivatives nor does it generate derivatives symbolically like computer algebraic packages (Griewank, 2000: 2-3). AD is rather a technique that calculates some intermediate values (numeric) given input values  $x$  to calculate output values  $y$  (Griewank, 2000: 4-5). These intermediate values can then be used to calculate derivatives (Griewank, 2000: 4-5).

In the above statement it is clear that AD is not symbolic but there might be some confusion about it being numerical. It was also stated that AD does not determine derivatives in the “conventional” numerical sense. What is meant by this is that AD *does not approximate* derivatives using a difference approach like in Equation 2.36

$$f'(x) \simeq \frac{f(x+h) - f(x)}{h} \text{ or } f'(x) \simeq \frac{f(x+h) - f(x-h)}{2h} \quad (2.36)$$

(Griewank, 2000: 2). The approach in Equation 2.36 does not provide accurate values: when  $h$  is small then cancellation errors reduces the number of significant figures of  $f'(x)$  but if  $h$  is not small truncation errors in terms like  $h^2 f'''$  become significant (Griewank, 2000: 2). AD does not incur truncation errors and usually yields accurate derivatives (Griewank, 2000: 2).

The efficiency of AD stems from the fact that it is not symbolic. Take the expression  $f(x) = x^2$  for example: AD yields the derivative  $2x$  exactly but it does not create the symbolic expression  $2x$  but instead the numerical value of  $x$  (in memory) is multiplied by 2 and returned as the derivative (Griewank, 2000: 3). This is an important distinction to make especially when equations become larger and more complex. Take Equation 2.37

as an example

$$f(x) = \prod_{i=1}^n x_i \quad (2.37)$$

then

$$\nabla f(x) = \left( \prod_{j \neq i} x_j \right)_{i=1 \dots n}$$

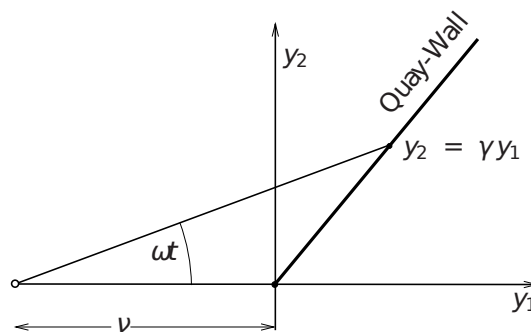
When  $n$  is large in Equation 2.37 the amount of memory required to represent  $\nabla f(x)$  symbolically is very large but AD exploits common sub-expressions in  $\nabla f(x)$  and uses them to evaluate  $\nabla f(x)$  at any point (Griewank, 2000: 3). AD does not first generate symbolic expressions and then simplify them again after but instead calculate intermediate numerical values and calculates  $\nabla f(x)$  using the chain rule (Griewank, 2000: 3-4).

AD uses a forward and reverse mode to calculate the derivatives. The basic principles of these two modes will be discussed in the remainder of this section. This section serves only as an introductory overview of how AD calculates derivatives. These two modes will be illustrated using an example of a lighthouse model whose light beam hits a quay-wall in Figure 2.21. The equations that describes the coordinate  $y = (y_1, y_2)$  of the light beam on the wall are given in Equation 2.38

$$y_1 = \frac{\nu \tan(\omega t)}{\gamma - \tan(\omega t)} \quad (2.38a)$$

$$y_2 = \frac{\gamma \nu \tan(\omega t)}{\gamma - \tan(\omega t)} \quad (2.38b)$$

where  $\gamma$  is the slope of the quay-wall,  $\nu$  the horizontal distance of the lighthouse from the wall,  $\omega$  the angular velocity of the revolving beam and  $t$  the time of the model (Griewank, 2000: 16-17). Equation 2.38 is the equations of the function  $F : \mathbb{R}^4 \rightarrow \mathbb{R}^2$  so that  $Y = [y_1, y_2] = F(X)$  with  $X = [\nu, \gamma, \omega, t]$ .



**Figure 2.21:** Diagram of a lighthouse shining a light on a quay-wall (Griewank & Walther, 2003).

Before the forward and reverse modes are described first consider how AD abstracts a system of equations to a directed graph called an evaluation trace (Griewank, 2000: 5).

This graph is acyclic and can be ordered in a lower triangular incidence matrix with no guessed variables.

### Decomposition into atomic operations

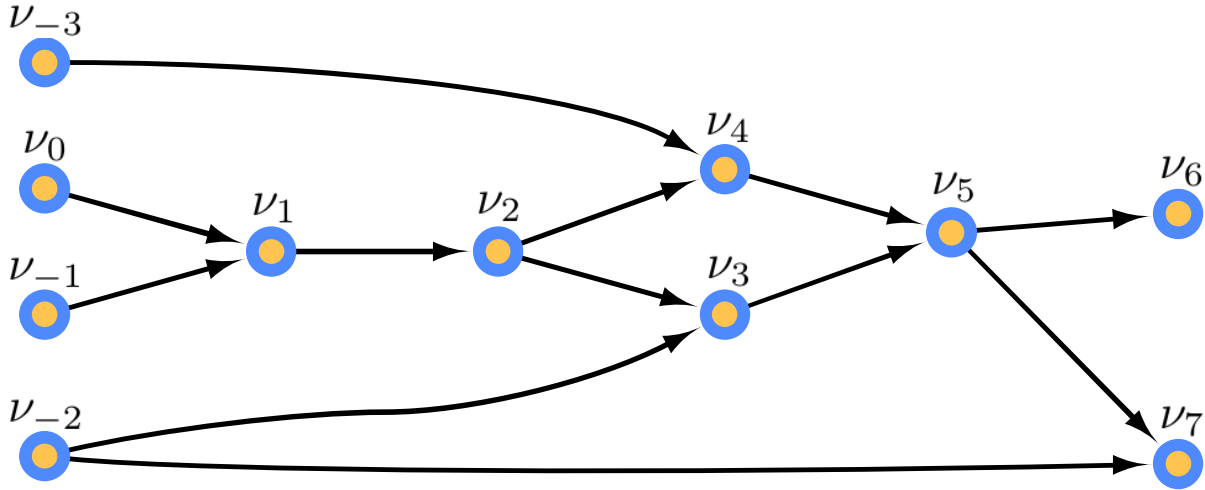
The expressions in AD are decomposed into atomic operations to evaluate derivatives more economically (Griewank, 2000: 17). Take Equation 2.38 as an example the expression  $\tan(\omega t)$  is present in both the equations. Evaluating this expression twice is avoided by assigning it to an intermediate variable (Griewank, 2000: 17). The process of assigning intermediate variables effectively abstracts the direct dependence of the dependent variables  $Y$  from the independent variables  $X$ , but instead is now dependent on an evaluation procedure (Griewank, 2000: 17).

Table 2.1 shows how intermediate variables are assigned to Equation 2.38. The first section in Table 2.1 maps the independent variables  $X$  to the internal variables  $\nu_{-3}, \nu_{-2}, \nu_{-1}, \nu_0$ . The second section is where the actual calculations take place and the third section maps the internal variables to the independent variables  $Y$ . The equation  $\nu_6 = \nu_5$  was introduced to make  $y_1 = \nu_6$  and  $y_2 = \nu_7$  mutually independent. A physical numerical example is also shown. The following values were assigned: the light is 100 m from the wall ( $\nu = 100$ ), the slope of the wall is 1.5 ( $\gamma = 1.5$ ), the light revolves with angular velocity of  $2 \cdot \pi/60 \frac{rad}{s}$  ( $\omega = 2 \cdot \pi/60$ ) and  $t$  is 5 s (Griewank, 2000: 18).

**Table 2.1:** Decomposition into atomic operations of  $F$  from Equation 2.38 (Griewank, 2000: 18).

Variables	Equation	Calculation	Result
$\nu_{-3}$	$= x_1 = \nu$		$= 100.0 \text{ m}$
$\nu_{-2}$	$= x_2 = \gamma$		$= 1.500$
$\nu_{-1}$	$= x_3 = \omega$	$= 2 \cdot \pi/60$	$= 0.1047 \frac{rad}{s}$
$\nu_0$	$= x_4 = t$		$= 5.000 \text{ s}$
$\nu_1$	$= \nu_{-1} \times \nu_0$	$= 0.1047 \times 5.000$	$= 0.5235 \text{ rad}$
$\nu_2$	$= \tan(\nu_1)$	$= \tan(0.5235)$	$= 0.5772$
$\nu_3$	$= \nu_{-2} - \nu_2$	$= 1.500 - 0.5772$	$= 0.923$
$\nu_4$	$= \nu_{-3} \times \nu_2$	$= 100.0 \times 0.5772$	$= 57.72 \text{ m}$
$\nu_5$	$= \nu_4/\nu_3$	$= 57.74/0.923$	$= 62.6 \text{ m}$
$\nu_6$	$= \nu_5$		$= 62.6 \text{ m}$
$\nu_7$	$= \nu_5 \times \nu_{-2}$	$= 62.6 \times 1.500$	$= 93.9 \text{ m}$
$y_1$	$= \nu_6$		$= 62.6 \text{ m}$
$y_2$	$= \nu_7$		$= 93.9 \text{ m}$

The directed graph representing Table 2.1 is shown in Figure 2.22.



**Figure 2.22:** Directed graph showing the computational flow of the lighthouse example.

### Forward mode

Suppose the partial derivative of  $\frac{\partial y_2}{\partial x_1}$  (or  $\frac{\partial \nu_7}{\partial \nu_{-3}}$  according to the decomposition) is required. One way of achieving this would be to calculate all the partial derivatives of all the variables ( $\nu_{-3}$  to  $\nu_7$ ) with respect to independent variable  $\nu_{-3}$ . This was done in Table 2.2 and by applying the chain rule to every line in Table 2.1 the goal of  $\frac{\partial \nu_7}{\partial \nu_{-3}}$  can be calculated. Therefore, in the forward mode a new variable  $\dot{\nu}_i = \frac{\partial \nu_i}{\partial x_1}$  is associated with every variable  $\nu_i$ . (Griewank, 2000: 6)

**Table 2.2:** Example of the forward mode of applying the chain rule (Griewank & Walther, 2003).

Variables	Equation	Calculation	Result
$\dot{\nu}_{-3}$	$= \dot{x}_1$		$= 1.000$
$\dot{\nu}_{-2}$	$= \dot{x}_2$		$= 0.0000$
$\dot{\nu}_{-1}$	$= \dot{x}_3$		$= 0.0000$
$\dot{\nu}_0$	$= \dot{x}_4$		$= 0.0000$
$\dot{\nu}_1$	$= \nu_0 \times \dot{\nu}_{-1} + \nu_{-1} \times \dot{\nu}_0$	$= 5.000 \times 0.0000 + 0.1047 \times 0.0000$	$= 0.0000$
$\dot{\nu}_2$	$= \dot{\nu}_1 / \cos^2(\nu_1)$	$= 0.0000 / \cos^2(0.5235)$	$= 0.0000$
$\dot{\nu}_3$	$= \dot{\nu}_{-2} - \dot{\nu}_2$	$= 0.0000 - 0.0000$	$= 0.0000$
$\dot{\nu}_4$	$= \nu_2 \times \dot{\nu}_{-3} + \nu_{-3} \times \dot{\nu}_2$	$= 0.5772 \times 1.000 + 100.0 \times 0.0000$	$= 0.5772$
$\dot{\nu}_5$	$= (\dot{\nu}_4 - \nu_5 \cdot \dot{\nu}_3) / \nu_3$	$= (0.5772 - 62.6 \times 0.0000) / 0.9228$	$= 0.625$
$\dot{\nu}_6$	$= \dot{\nu}_5$		$= 0.625$
$\dot{\nu}_7$	$= \nu_{-2} \times \dot{\nu}_5 + \nu_5 \times \dot{\nu}_{-2}$	$= 1.500 \times 0.625 + 62.6 \times 0.0000$	$= 0.938$
$\dot{y}_1$	$= \dot{\nu}_6$		$= 0.625$
$\dot{y}_2$	$= \dot{\nu}_7$		$= 0.938$

By applying the chain rule to every line as shown in Table 2.2 to calculate the partial derivative of  $\frac{\partial y_2}{\partial x_1}$  the partial derivative of  $\frac{\partial y_1}{\partial x_1}$  was also calculated in the process. The

calculation of all the output derivatives is a consequence of the forward mode where one sweep of the derivative trace for a specific independent variable (in this case  $x_1$ ) will result in the partial derivative of all the outputs ( $y_1$  and  $y_2$ ) to that specific independent variable.

By analysing Table 2.2 further it can be also be seen that the partial derivative of eg.  $\frac{\partial y_2}{\partial x_2}$  (or  $\frac{\partial \nu_7}{\partial \nu_{-2}}$ ) can just as easily be calculated by simply letting  $\dot{x}_2 = 1$  and  $\dot{x}_1, \dot{x}_3, \dot{x}_4 = 0$  without changing any of the equations in Table 2.2. This is also a consequence of the forward mode and to calculate all the partial derivatives of a function  $F : \mathbb{R}^m \rightarrow \mathbb{R}^n$  will require  $m$  sweeps of the derivative trace. Therefore, if  $n \gg m$  it is more efficient to use the forward mode.

## Reverse mode

The reverse mode is also known as the adjoint mode. In contrast to the forward mode, an output variable ( $Y$ ) is chosen and the partial derivatives calculated with respect to all the intermediate variables ( $\nu_i$ ) instead of choosing an input variable and calculating the partial derivatives calculated of all the intermediate variables with respect to the input variable. The adjoint partial derivatives ( $\bar{\nu}_i$ ) are calculated by moving backwards in the graph (Figure 2.22). (Griewank, 2000: 8)

As an example the partial derivative of  $y_2$  to  $x_1$  is again calculated but in the reverse mode. To calculate this in the reverse mode a variable  $\bar{\nu}_i = \frac{\partial y_2}{\partial \nu_i}$  is associated with every variable  $\nu_i$  (Griewank, 2000: 8). Calculating the derivatives in the reverse mode is a bit more tricky and using the computational graph (e.g. Figure 2.22) is a useful way to find the correct connection. Take  $\bar{\nu}_2$  as an example which was calculated as follows:  $\bar{\nu}_2 = \frac{\partial y_2}{\partial \nu_4} \frac{\partial \nu_4}{\partial \nu_2} + \frac{\partial y_2}{\partial \nu_3} \frac{\partial \nu_3}{\partial \nu_2}$  and because  $\bar{\nu}_4 = \frac{\partial y_2}{\partial \nu_4}$  and  $\bar{\nu}_3 = \frac{\partial y_2}{\partial \nu_3}$  it simplifies to  $\bar{\nu}_2 = \bar{\nu}_4 \frac{\partial \nu_4}{\partial \nu_2} + \bar{\nu}_3 \frac{\partial \nu_3}{\partial \nu_2}$ . The tricky part is to find out which variables (in this example  $\nu_4$  and  $\nu_3$ ) have an effect on  $y_2$  (Griewank, 2000: 8). To find these variables find all the paths from the base variable (in this case  $\nu_2$ ) to the required output variable (in this case  $y_2$ ) and the second vertex in each path is a variable that should be used in the chain rule. For the current example there are two paths ( $P_1 = \{\nu_2, \nu_4, \nu_5, \nu_7\}$  and  $P_2 = \{\nu_2, \nu_3, \nu_5, \nu_7\}$ ) from the base variable  $\nu_2$  and the second vertices are  $\nu_4$  and  $\nu_3$ .

By applying the chain rule to every line as shown in Table 2.3 to calculate the partial derivative of  $\frac{\partial y_2}{\partial x_1}$  the partial derivatives of  $\frac{\partial y_2}{\partial x_2}$ ,  $\frac{\partial y_2}{\partial x_3}$  and  $\frac{\partial y_2}{\partial x_4}$  was also calculated in the process. This is a consequence of the reverse mode where one sweep of the derivative trace for a specific dependent variable (in this case  $y_2$ ) will result in the partial derivative of all the inputs ( $x_1, x_2, x_3$  and  $x_4$ ) to that specific dependent variable.

By analysing Table 2.3 further it can be also be seen that the partial derivative of eg.  $\frac{\partial y_1}{\partial x_i}$  can just as easily be calculated by simply letting  $\bar{y}_1 = 1$  and  $\bar{y}_2 = 0$  without changing any of the equations in Table 2.3. This ability to change only the inputs is also

**Table 2.3:** Example of the reverse mode of applying the chain rule (Griewank & Walther, 2003).

Variables	Equation	Calculation	Result
$\bar{\nu}_7$	$= \bar{y}_2$		$= 1.000$
$\bar{\nu}_6$	$= \bar{y}_1$		$= 0.0000$
$\bar{\nu}_5$	$= \bar{\nu}_7 \times \nu_{-2} + \bar{\nu}_6$	$= 1.0000 \times 1.500 + 0.0000$	$= 1.500$
$\bar{\nu}_4$	$= \bar{\nu}_5 / \nu_3$	$= 1.500 / 0.923$	$= 1.63$
$\bar{\nu}_3$	$= (-\bar{\nu}_5 \times \nu_5) / \nu_3$	$= -(1.500.6) / 0.923$	$= -102$
$\bar{\nu}_2$	$= \bar{\nu}_4 \times \nu_3 - \bar{\nu}_3$	$= 1.63 \times 100.0 - (-102)$	$= 265$
$\bar{\nu}_1$	$= \bar{\nu}_2 / \cos^2(\nu_1)$	$= 265 / \cos^2(0.5235)$	$= 353$
$\bar{\nu}_0$	$= \bar{\nu}_1 \times \nu_{-1}$	$= 353.1047$	$= 37.0$
$\bar{\nu}_{-1}$	$= \bar{\nu}_1 \times \nu_0$	$= 353 \times 5$	$= 1765$
$\bar{\nu}_{-2}$	$= \bar{\nu}_7 \times \nu_5 + \bar{\nu}_3$	$= 1.000 \times 62.6 + (-102)$	$= -39$
$\bar{\nu}_{-3}$	$= \bar{\nu}_4 \times \nu_2$	$= 1.63 \times 0.5772$	$= 0.941$
$\bar{x}_1$	$= \bar{\nu}_{-3}$		$= 0.941$
$\bar{x}_2$	$= \bar{\nu}_{-2}$		$= -39$
$\bar{x}_3$	$= \bar{\nu}_{-1}$		$= 1765$
$\bar{x}_4$	$= \bar{\nu}_0$		$= 37.0$

a consequence of the reverse mode and to calculate all the partial derivatives of a function  $F : \mathbb{R}^m \rightarrow \mathbb{R}^n$  will require  $n$  sweeps of the derivative trace. Therefore, if  $m \gg n$  it is more efficient to use the reverse mode.

The difference in values of  $\frac{\partial y_2}{\partial x_1}$  between the reverse mode is due to rounding error and is equal if rounded to 2 decimal points.

### 2.7.1 CasADi

CasADi (Andersson, 2013) is a tool that can be used for AD. Although it started out as a tool for AD it has expanded since its inception to a tool for gradient-based optimisation and other numerical methods like multidimensional Newton's method.

CasADi itself is *not* an optimisation package but interfaces with well-known optimisation packages like IPOPT (Wächter & Biegler, 2006).

CasADi forms an integral part of Amoss not only due to its ability to evaluate derivatives automatically but how it interfaces with other optimisation packages and numerical methods on behalf of the user.

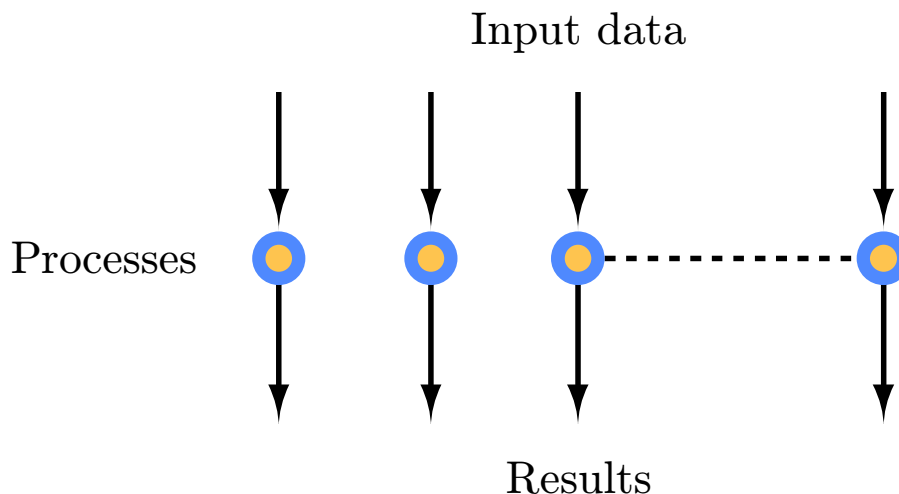
## 2.8 Parallel processing

Parallel processing is a method where a problem is divided into separate parts and computations executed on separate processors simultaneously. Embarrassingly parallel pro-



cessing, or sometimes referred to as ideal processing, is a sub-branch of parallel processing where the computations can be divided into completely independent parts.

To parallelise these problems are usually simple and do not require special techniques and or algorithms to obtain results. These problems usually do not require communication between the separate processes and are completely disconnected from one another as shown in Figure 2.23. (Wilkinson & Allen, 2005: 79 - 80)



**Figure 2.23:** Embarrassingly parallel problem represented as a disconnected computational graph adapted from Wilkinson & Allen (2005: 80).

### 2.8.1 Celery

Celery (Ask Solem et al., 2017) (a Python library) can be used to execute embarrassingly parallel problems easily. A job is created for each computational part and handed to Celery. Celery manages these jobs and distributes the jobs to available processors as they become available. The speed-up factor that can be expected is a small number less than the integer division of the number of separate parts by the number of available processors.

## 2.9 Big O notation

Big-O notation is used to express an asymptotic upper bound on the running time growth of a program given an input size. The upper bound on the running time of a binary search is written as  $O(\log n)$ . This means that the worst-case running time is  $k \log(n)$  for some constant  $k$  and input size  $n$  (when  $n$  is large enough). The big-O notation gives an upper bound on the running time growth and not the absolute running time growth. Big-O indicates that the growth will not be larger than the upper bound but it may also be lower. (Khan Academy, 2017)

The big-O notation of some common problems are:  $O(n)$  for finding an item in an unordered list,  $O(n^2)$  for solving a system of linear equations, bubble- and selection sort algorithms and  $O(n!)$  for solving the travelling salesman problem via brute-force.

---

---

# CHAPTER 3

---

## OVERVIEW

### 3.1 Relation with MOSS

Amoss is an extension of Sasol's MOSS methodology with the goal of improving the effectiveness of these simulations. The problems of MOSS are outlined in section 1.4 and the Amoss project was started to solve these problems.

MOSS is currently used to answer and quantify difficult "what if" questions like: would increase in buffer tank size significantly reduce product loss or would additional process equipment increase production (increasing the production capability of a unit does not guarantee overall production increase due to bottlenecks in the process). For a specific model, a scenario table is created containing all the different "what if" questions and these scenarios are then simulated for some number of replications (experiments). The results from these scenarios are used in decision-making.

### 3.2 What is Amoss

Amoss is an equation orientated discrete event simulation platform to create and simulate flowsheeting problems containing stochastic elements. Although Amoss simulates continuous systems, the differential equations are discretised using a backward difference approximation (Euler's algorithm) and the stochastic elements are sampled at every time step.

The goal of Amoss is to create and simulate stochastic simulations in the most logical and user-friendly way. The emphasis of Amoss was to relieve the user from most of the repetitive work in making simulations and to focus on the aspects that make each simulation unique. The basic aspects that make each simulation unique are:

**Operating units.** The specifications of the operating units are different from process to process, e.g. the split ratios of different distillation columns or conversions in reactors. The operating units available in Amoss are reactors, separators, buffer tanks and mixing points.

**Connectivity.** The mapping that indicates how the different operating units are connected to each other.

**Operating rules.** Heuristic rules that govern how a process should be operated (high-level control).

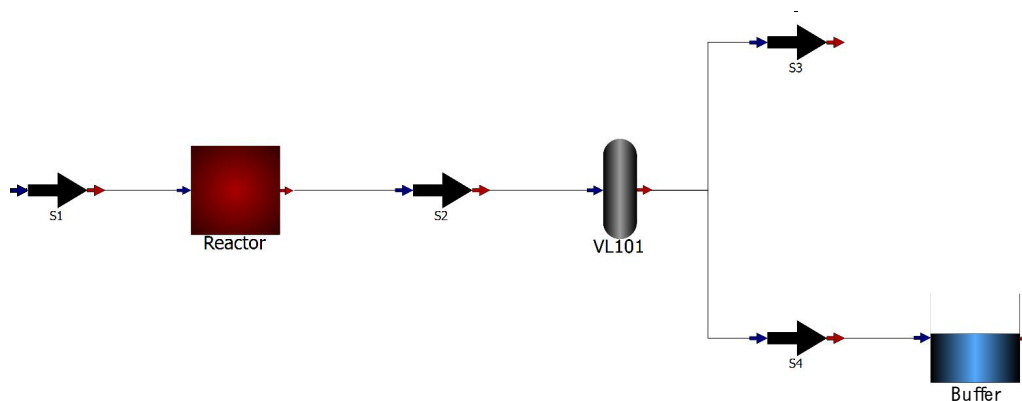
**Stochastic elements.** The variables that have a distribution function (this may include common distributions or a custom user-defined distributions).

Amoss is designed from the ground up to use the equation orientated approach due to the many advantages over the modular approach (see subsection 2.4.1). This also resulted in faster simulation speeds and the ability to handle feedback loops. The modular approach is efficient in solving unidirectional (no feedback loops) systems, but as soon as feedback loops are introduced multiple iterations are required for convergence.

One of the major factors for choosing the equation orientated strategy is its expandability. In the modular approach, the user is limited to the available modules or has to create his/her own. The lack of ability to add operating rules that work across modules makes the modular approach even more restrictive; whereas in the equation oriented approach, these equations are simply added to the larger system of equations where it is treated as just another equation.

To illustrate this point consider the small unit in Figure 3.1. In this unit the distillation column and the buffer is located downstream of the reactor. These downstream unit operations will affect the reactor operation. In the scenario the downstream units are not operational or processing less material (could be due to buffer reaching its capacity or inadequate cooling or steam supply at the distillation column) the feed to the reactor need to be adjusted otherwise the reactor products will be discarded leading to financial loss.

In this example the downstream units dictate the feed to the reactor. In the equation orientated approach equations are created relating the downstream conditions to upstream conditions (eg. when the buffer reaches full capacity stop feed to the reactor), but in the modular approach different modular units will need to be created that can still interface with the rest of the flow-sheet. In this particular example, the dependence of the upstream units on downstream information (flow-rate, temperature, composition etc.) adds an information feedback loop which reduces the modular approach's efficiency.



**Figure 3.1:** A small unit with a reactor, distillation column and a buffer tank.

### 3.3 Project software

Amoss is written in Python (Python Software Foundation, 2017) utilising Open Modelica’s Connection Editor (OMEdit) (Open Source Modelica Consortium, 2017), Microsoft Excel (Microsoft, 2017) and Atom text editor (GitHub Inc, 2017) to facilitate with the user interface.

Python was chosen as the programming language for the reasons listed below:

- 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 distributed without paying royalties. Therefore, developing in Python makes it possible to distribute Amoss among many users without the obligation to pay a licensing fee.
- The PSFL does not place any licencing restrictions on software developed with Python so it is possible to distribute Amoss without making it open source.
- Python is open source software and easily accessible.
- Python has a large community enabling support for new Python programmers.
- Python’s accessibility and support enable a larger population of people the ability to contribute to the Amoss project.

OMEdit is used as a graphical interface for the user to define a process’ connectivity as displayed in Figure 4.1. Microsoft Excel is available to all Sasol’s office employees, and most are familiar with the software; therefore, Microsoft Excel is used as a way for the user to supply specific data structures to Amoss. Atom is used as a platform for the user to supply external equations to the simulation. Atom describes itself as “a hackable text editor” GitHub Inc (2017). This customisability was utilised to give a user-friendly platform to add additional information.

The combination of these software makes Amoss.

### 3.4 Amoss work-flow

The basic work-flow of Amoss is illustrated in Figure 3.2 with an in-depth analysis of each of the steps, 1-7, in Figure 3.2 explained in chapter 4. A high-level description of the Amosswork-flow is described below:

Step 1 requires the user to draw a process diagram in OMEdit together with operational unit specific data in Excel.

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

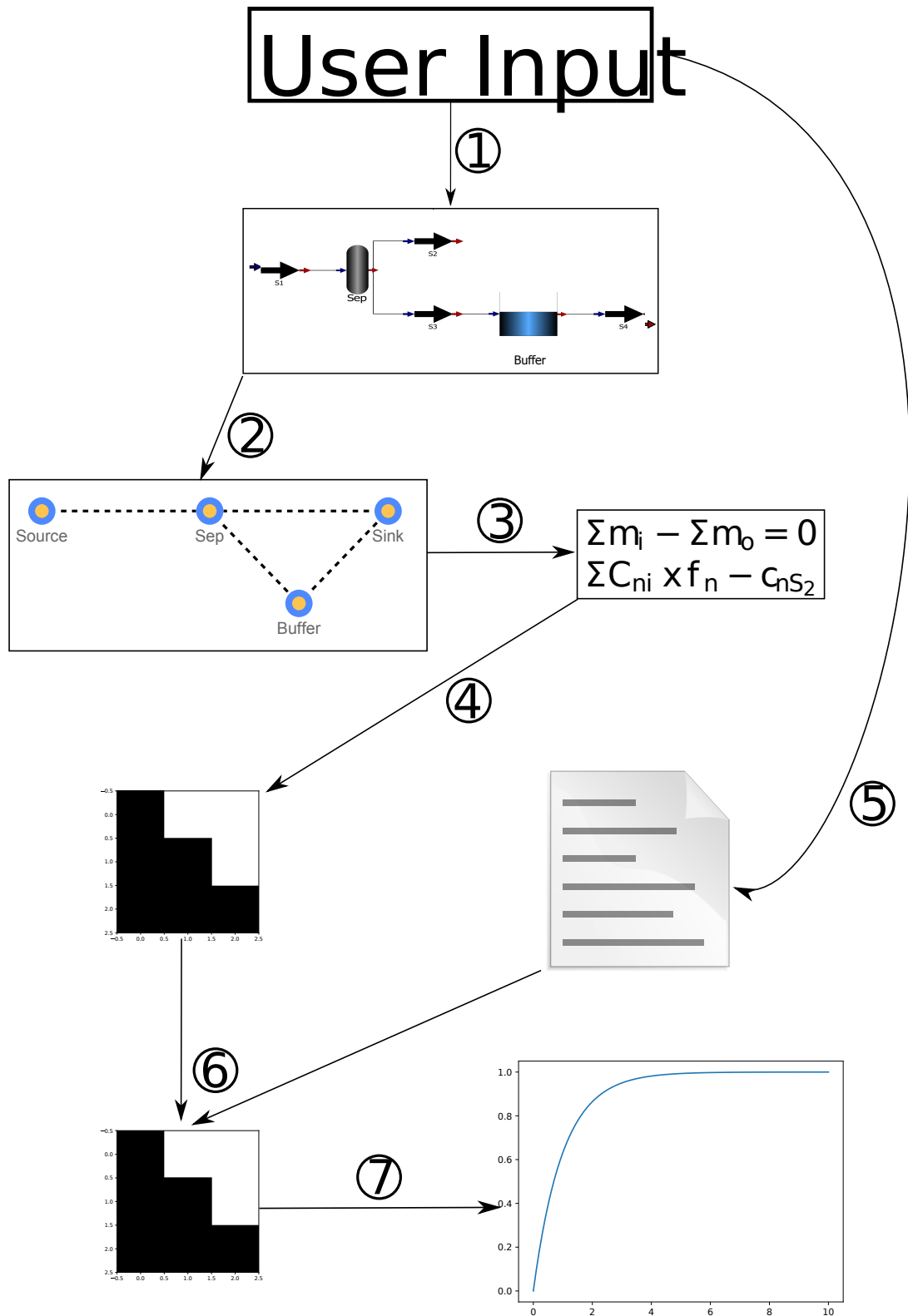
Step 3 iterates through the graph creating all the necessary equations.

Step 4 tears the equations using the block lower triangular method and symbolically solves the smaller blocks. This is referred to a pre-solving the equations, because a subset of the complete model equations are solved before the entire model is solved during the simulation. Pre-solving is done to decrease the border width in Step 6 making it easier to solve the system of equations.

Step 5 requires the user to define additional equations to describe how the unit is operated (operating rules) together with identifying which variables or parameters are stochastic and provide a distribution.

Step 6 takes the system of equations in Step 4 and concatenates it with the equations in Step 5 and tears these equations using the bordered block lower triangular method producing the full system of equations.

Step 7 simulates the process.



**Figure 3.2:** Work-flow of the Amoss project from problem description to simulation with 1) Process digram in Modelica, 2) Process digram to network graph, 3) Equation generation, 4) Equation tearing using block lower triangular decomposition, 5) External equations from user, 6) System equation (4) and user equations (5) concatenated and tor using bordered block lower triangular decomposition, 7) Stochastic simulation.

---

---

# CHAPTER 4

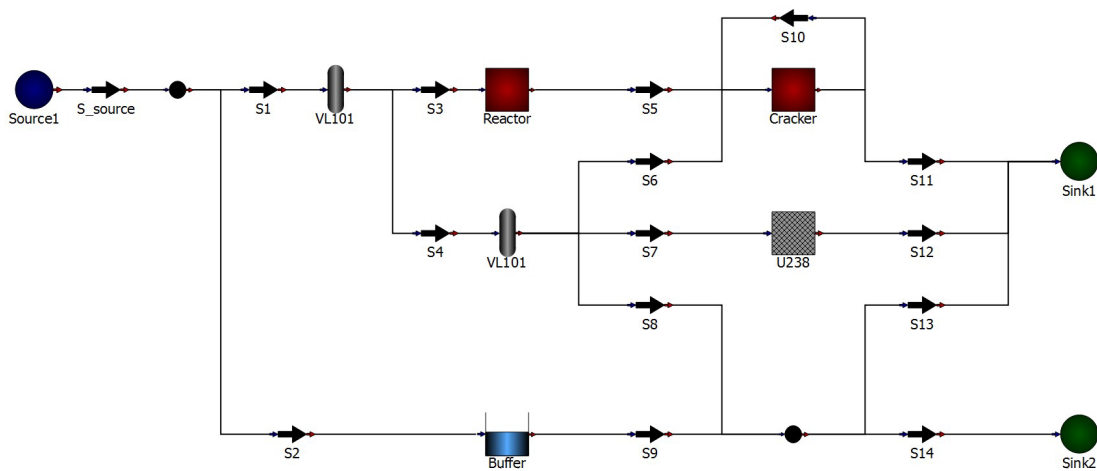
---

## ANALYSIS

In this chapter each aspect of Amoss shown in Figure 3.2 will be discussed and analysed individually.

### 4.1 Process diagram


Figure 4.1 shows an example of a process diagram in OMEdit. Amoss is distributed with an OMEdit package containing different models representing the different unit operations. The icons and the unit types available are discussed below. It is important to use the correct icon that describes the operational unit that is required because in Step 2 this information is translated to different vertex attributes and in Step 3 the unit type will lead to the creation of different equations.





**Figure 4.1:** Example of the graphical process interface by utilising OMEdit.




The model icons and their functionality in a simulation is described below:


**Source**  is the location where mass enters the simulated process (when thinking of the process in isolation this is the location where mass is generated).


**Sink**  is the location where mass exits the simulated process (in isolation, mass is consumed at this location).


**Mix Point**  is an operational unit where streams of different compositions are mixed and the streams leaving the **Mix Point** all have the same composition. It is allowed that more than one stream may exit a **Mix Point** with different flow rates.

**Buffer**  is a tank within the process.

**Reactor**  is an operational unit where components can be converted to different components.

**Pipe**  is the model type used to give flow-streams names and indicate the positive flow direction. All the calculations of Amoss are based on the names given to the **Pipes** and the positive flow direction.

**Separator**  is the operational unit that models a distillation column. This model can split streams into streams with different compositions. It is the opposite of the **Mix Point** where the outputs all have the same composition.

**Visual Unit**  is a unit to indicate that a unit is present visually but concerning mass flow acts the same as a **Mix Point**.

To draw a diagram in OMEdit, the model icons are dragged and dropped in the diagram worksheet. The locations where mass enters (**Source**) the process and exits (**Sink**) the process needs to be indicated by the **Source** and **Sink** models. It is an Amoss *rule* that every unit operation must be separated by adding a **Pipe** model between them. This rule can be observed in Figure 4.1. It is important for the creation of the equations in Amoss that the operational units are separated by a **Pipe**.

To complete a process diagram each of the units needs to be connected with a connector (thin line in Figure 4.1 connecting the **Pipes** to the other units) in OMEdit. The order or direction of the connection does not indicate the positive flow direction; therefore, these connections can be added in any direction. The positive flow direction is determined by the **Pipe** arrow direction and the inlet and outlet of all the other units.

With a careful look at model icons each model icon has a small arrow entering or exiting the model. These arrows indicate the positive flow direction. The blue arrow indicates the inlet to the unit and the red arrow the outlet.

Together with the diagram Amoss also requires additional information to be added in the Excel file as tables in different sheets. All the units, excluding the **Source**, **Sink** and **Pipe**, require additional information together with the component names and the input variables to the simulation. All the information is added to an Excel file called `simulation_description.xlsx`. The Excel sheets names mentioned in the subsections below all belong to the `simulation_description.xlsx` file.

### 4.1.1 Component information

The component information is simply a list of names of the components that are found in the simulation added in the `component_list` sheet. It is important that the list contains *all* the components that take part in the system (if the value is required) and not only the components that are fed through the **Sources**. An example of the component list is in Table 4.1. The component names must be valid Python variable names.

### 4.1.2 Input information

The input data are those variables that are required to make the degrees of freedom (DOF) equal to zero and will always be the flow-rate of a **Pipe**. These inputs are listed in the `inputs` sheet. In each simulation, the component flow-rate, as well as the total flow-rate of each pipe, will be calculated and therefore the inputs can be a total flow-rate or a single component flow-rate.

An example of an input data table is in Table 4.1. To list an input the specific **Pipe** name together with the **Pipe** flow variable (a component name for a component flow-rate and total for the total flow-rate) is required. In Table 4.1 a few inputs are listed with the **Pipe** name under the stream heading and the **Pipe** flow variable under `comp`. In Table 4.1 row 1 the input variable is the total flow-rate in **Pipe** S1 (variable name `S1_total`) and in row 2 is the `comp1` flow-rate in **Pipe** S2 (variable name `S2_comp1`).

**Table 4.1:** Example tables show how to add the component list (left) and the simulation inputs (right).

Components	stream	comp
comp1	S1	total
comp2	S2	comp1
⋮	⋮	⋮
compn	Sn	compn

Below is a guideline for choosing the input variables. This guideline does not cover all the possible valid inputs, but by following this guideline all the selected inputs will be valid. The guideline uses a modular approach to identify the input variables, but because

Amoss is equation orientated a variety of other input variable combinations can also be valid.

A fully defined stream means that every **Pipe** that enters a unit will have a known value for all the component flow-rates as well as the total flow-rates. This guideline will assume that all the streams entering a unit is fully defined. This assumption is not unrealistic and in most cases valid. Most of the time what enters a unit is known but the output needs to be calculated and the calculated output of a unit serves as the input to another.

### The guideline:

- For every **Source** that is encountered the number of inputs that are required is equal to the number of components ( $n_C$ ) listed in the **component.list** sheet. The number of required inputs can simply be all the component flow-rates of the **Pipes** exiting a **Source** or the total flow-rate of a **Pipe** together with  $n_C - 1$  component flow-rates of that **Pipe**.
- For every **Mix Point**, **Reactor** or **Visual Unit** that is used require the number of **Pipes** ( $n_P$ ) connected to the outlet of the unit negative one ( $n_P - 1$ ) as its the number of input variables (this can be a component flow-rate or a total flow-rate).

Always specify stream flows with an *out* direction from these operational units because it guarantees that all inputs are valid. It is important not to list all the streams exiting these model because it will result in an over specified system. The remaining stream will be calculated with a mass balance.

- Every **Buffer** that is used require the same number of input variables as the number of **Pipes** connected to the outlet of the **Buffer** (this can be a component flow-rates or a total flow-rates).

### 4.1.3 Separator information

Every **Separator** that is added in the diagram requires a corresponding entry in the **separation.data** sheet table. The number of rows that are required in the **separation.data** sheet table per each **Separator** is equal to the number of **Pipes** connected to the **Separator** outlet negative one ( $n_P - 1$ ). An example of how this table should look like is shown in Table 4.2.

The number of columns of the table is the number of components plus two ( $n_C + 2$ ). The first two columns contain the name of the **Separator** and the relevant **Pipe** name. The remaining columns indicate which fraction of that component flow-rate entering the **Separator** will report at the **Pipe** listed under the attribute column of that same row.

The automatic set-up of the `separation_data` table based on the component list and the process diagram is not currently a feature of Amoss and is recommended for future improvement.

As an example take row 1 in Table 4.2 with the fraction variable `S6_f1` equal to 0.5 ( $S6\_f1 = 0.5$ ) then 50% of the total mass flow of `comp_1` entering the Separator will report in Pipe `S6`. The same logic applies to `comp_2` and `S6_f2`. The fraction `S6_f2` of the total mass flow of `comp_2` will report in `S6`. As a sanity check the columns of each Separator must sum to a value less than 1, eg. in Table 4.2 the column of row 1 and 2 in must sum to less than 1 and the columns of row 3 must be less than 1.

Currently Amoss has no automated checks to ensure that the columns of each separator sums to a value less than one, but it is recommended as a future improvement.

The fraction entries may also be a rational number and are not limited to symbolic names. The advantage of giving fraction entries symbolic names is the ability to change the fractions during the simulation or different scenarios. If the fractions do not change during the simulation, entering the real numbers will make the simulation set-up cleaner because these values do not need to be defined elsewhere.

**Table 4.2:** Example table of the information that is required for every Separator.

node	attribute	comp_1	comp_2	...	comp_n
Separator	S6	S6_f1	S6_f2	...	⋮
Separator	Sx	Sx_f1	Sx_f2	...	⋮
VL101	S101	S101_f1	S101_f2	...	S101_fn

#### 4.1.4 Reactor information

Every Reactor that is added in the diagram requires a corresponding entry in the `react_data` sheet table. Table 4.3 is an example of a `react_data` table.

The number of rows that are required per Reactor is equal to the number components in the component list ( $n_C$ ). The number of columns of the table is  $n_C + 2$ . Column 1 contains the name of the Reactor and column 2 is the list of components with the word *in* next to it. The remaining column headings also contain the component list but with the word, *out* next to it. The conversion fractions of the components entering the Reactor to the other components are the entries in columns containing the word *out* in the heading.

The automatic set-up of the `reactor_data` table based on the component list and the process diagram is not currently a feature of Amoss and is recommended for future improvement.

The table contains the conversion of each component entering the Reactor (`comp_x` in) to another component (`comp_x` out). For example `R1.c1i.c2o` (short for *Reactor1 comp\_1*

*in comp\_2 out*) is the fractional conversion of *comp\_1* to *comp\_2*. As a sanity check, all the columns for each **Reactor** must sum up to 1.

Currently Amoss has no automated checks to ensure that the columns of each reactor sums to a value of one, but it is recommended as a future improvement.

It is not required for the conversion fractions to be symbolic names, but rational numbers are also allowed. The same advantages hold for either giving the conversions symbolic names or rational fractions, depending on the situation, as in the case of the **Separator** mentioned above.

**Table 4.3:** Example table of the information that is required for every **Reactor**.

node	comp	comp_1 out	comp_2 out	...	comp_n out
Reactor	comp_1 in	R1_c1i_c1o	R1_c1i_c2o	...	R1_c1i_cno
Reactor	comp_2 in	R1_c2i_c1o	R1_c2i_c2o	...	...
Reactor	⋮	⋮	⋮	⋮	⋮
Reactor	comp_n in	R1_cni_c1o	R1_cni_c2o	...	R1_cni_cno
Cracker1	comp_1 in				⋮
Cracker1	⋮		⋮		
Cracker1	comp_n in		...		R2_cni_cno

## 4.2 Process graph

A directed graph  $D = (V, E)$  is extracted from the process diagram and created in NetworkX (NetworkX developers, 2017), a Python library with a variety of graph theory algorithms. All the operational units in the diagram are the vertices, and the pipes indicate an edge between vertices together with the edge direction. The diagram is transformed into a directed graph because it is easier to interact with the graph than the diagram.

A graph makes it possible to iterate through a network and to find the number of edges connected in the in direction and the out direction of a vertex. NetworkX also makes it possible to store information for every vertex and edge known as attributes. Vertices contain the name given in the diagram as well as the unit operational type, e.g. separator, reactor or buffer. It is also possible to add the data tables in Table 4.2 and Table 4.3 for the **Separators** and **Reactors** to their specific vertices. The **Pipe** names in the diagram are the names given to the edges.

The reason the process diagram in OMEdit is used instead of creating a graph directly is that it is easier for a human to understand and interpret a visual representation of the

process rather than a graph table containing the connectivity of the units and their attributes.

## 4.3 Automatic equation generation

The basic equations that describe how mass flows through the network are automatically created by using the directed graph  $D = (V, E)$  extracted from the process diagram. These equations are: the component mass balances, mixing equations around **Mix Points**, component conversion equations for the **Reactors**, the component split equations for the **Separators** and integral equations for the **Buffers**.

Equations are created by iterating through all the vertices in  $D$  and depending on the vertex type (**Separator**, **Reactor**, etc.) different sets of equations are created as discussed below.

All the examples in section 4.3 will use the diagram in Figure 4.1 with two components `comp_1` and `comp_2`. In the subsections, a DOF analysis will also be performed alongside the examples. It should be noted that all the DOF analyses that are done are made with the assumption that the all the edges entering a vertex are fully defined. These DOF analyses serve to support the suggestions made in the guideline for choosing inputs (section 4.1.2)

### 4.3.1 Component mass balance equations

Component mass balances are performed over most of the operational units excluding all **Sources**, **Sinks** and **Pipes**. In the environment consisting solely of  $D$ , **Sources** are locations where mass enters the system and **Sinks** where mass exits the system. No mass balances are performed over **Sources**, **Sinks** and **Pipes** because the mass balances around these units are redundant.

A component mass balance is created by the summation of all the single component flow-rates in the edges entering a vertex and subtracting the summations of the component flow-rates in the exiting edges. This component balance is done for each of the components listed in the component list Table 4.1.

#### Example

The **Mix Point** connected to the **Pipe S\_source** on the left of Figure 4.1 is used as an example of a component balance. The equations that will be created are shown in Equation 4.1 and Equation 4.2

$$S\_source_{comp.1} - (S1_{comp.1} + S2_{comp.1}) = 0 \quad (4.1)$$

$$S\_source_{comp.2} - (S1_{comp.2} + S2_{comp.2}) = 0 \quad (4.2)$$

The number of edges entering a vertex  $N$  is calculated with Equation 4.3 and the number of edges exiting  $N$  is calculated with Equation 4.4. The total number of edges connected to  $N$  is the sum of Equation 4.3 and Equation 4.4

$$n_{Ei} : V \rightarrow \mathbb{N}_0 \quad (4.3)$$

$$n_{Ex} : V \rightarrow \mathbb{N}_0 \quad (4.4)$$

$$n_E = n_{Ei} + n_{Ex} \quad (4.5)$$

The number of equations ( $n_{Eq}$ ) and the number of variables ( $n_{Var}$ ) that will be created at each vertex are dependent on the  $n_C$  and the number of edges ( $n_E$ ) connected to the vertex. The  $n_{Eq}$  and  $n_{Var}$  that will be created per vertex around which component mass balances are performed is calculated in Equation 4.6 and Equation 4.7

$$n_{Eq} = n_C \quad (4.6)$$

$$n_{Var} = n_E(N) \times n_C \quad (4.7)$$

$$\begin{aligned} \text{DOF} &= n_E(N) \times n_C - [\underbrace{n_{Ei}(N) \times n_C}_{\text{fully defined inputs}} + n_C] \quad (4.8) \\ &= (n_E(N) - n_{Ei}(N)) \times n_C - n_C \\ &= (n_{Ex}(N) - 1) \times n_C \end{aligned}$$

where  $N$  is the current vertex.

It is important to notice that  $\text{DOF} > 0$  when  $n_{Ex}(N) \geq 2$  because it will increase the DOF for every vertex around which these equations are applied. More equations or inputs are required to specify the system fully. These additional equations that are required are discussed in the remainder of this section.

### 4.3.2 Total flow equations

For every edge in  $D$ , a total flow equation is created. This equation is simply a summation of all the components in the stream.

#### Example

Equation 4.9 uses S1 in Figure 4.1 for the example equation

$$S1_{\text{comp.1}} + S1_{\text{comp.2}} - S1_{\text{total}} = 0 \quad (4.9)$$

Equation 4.10 and Equation 4.11 calculates the  $n_{Eq}$  and  $n_{Var}$  create per vertex

$$n_{Eq} = n_E(N) \quad (4.10)$$

$$n_{Var} = n_E(N) \quad (4.11)$$

$$\text{DOF} = 0$$

where  $N$  is the current vertex.

The  $\text{DOF} = 0$  because the Total flow equations introduces the same amount of equations and variables.

### 4.3.3 Mix Point equations

These equations ensure that the composition of each edge exiting the vertex is equal to the combined composition of the edges entering the vertex. These equations are created by subtracting the mass fraction of component  $x$  in a single exiting edge from the combined mass fraction of component  $x$  in all the edges entering the vertex  $N$ .

#### Example

Equation 4.12 is the equation that will be created for the Mix Point in Figure 4.1 on the bottom left with entering streams S8 and S9 and exiting streams S13 and S14

$$\frac{\text{S8}_{\text{comp.1}} + \text{S9}_{\text{comp.1}}}{\text{S8}_{\text{total}} + \text{S9}_{\text{total}}} - \frac{\text{S13}_{\text{comp.1}}}{\text{S13}_{\text{total}}} = 0 \quad (4.12)$$

The number of equations that will be generated is dependent on the number of edges exiting a vertex ( $n_{Ex}$ ) and the  $n_C$ . Equation 4.13 and Equation 4.14 calculated the number of equations and variables created

$$n_{Eq} = (n_{Ex}(N) - 1) \times (n_C - 1) \quad (4.13)$$

$$n_{Var} = 0 \quad (4.14)$$

$$\text{DOF} = -(n_{Ex}(N) - 1) \times (n_C - 1) \quad (4.15)$$

where  $N$  is the current vertex. The  $\text{DOF} = 0$  in Equation 4.15 when  $n_{Ex}(N) = 1$  or  $n_C = 1$  and  $\text{DOF} < 0$  when  $n_{Ex}(N) \geq 2$  and  $n_C \geq 2$  therefore reducing the  $\text{DOF}$ .

Around every Mix Point Component mass balance equations and Total flow equations are created in addition to the Mix Point equations. In Equation 4.13 a Mix Point equations is created for only  $n_{Ex}(N) - 1$  edges and  $n_C - 1$  components which means that one exit edge and one component is not used in the creation of the Mix Point equations. The removal of this edge and component is necessary to ensure that the equations around every Mix



Point are all independent together with the Component mass balance equations and Total flow equations.

Equation 4.16 is a DOF analysis of a Mix Point vertex including the Component mass balance equations and Total flow equations

$$\begin{aligned}
 \text{DOF} &= \underbrace{[(n_{Ex}(N) - 1) \times n_C]}_{\text{mass balance}} + \underbrace{[0]}_{\text{total flow}} + \\
 &\quad \underbrace{[-(n_{Ex}(N) - 1) \times (n_C - 1)]}_{\text{mix}} \\
 &= n_C (n_{Ex}(N) - n_{Ex}(N)) - n_C + n_C + n_{Ex}(N) - 1 \\
 &= n_{Ex}(N) - 1
 \end{aligned} \tag{4.16}$$

Equation 4.16 shows that for every Mix Point  $n_{Ex}(N) - 1$  variables need to be specified as inputs as reported in section 4.1.2. The  $\text{DOF} = 0$  when  $n_{Ex}(N) = 1$  otherwise  $\text{DOF} > 0$  when  $n_{Ex}(N) \geq 2$ .

The  $n_{Ex}(N) - 1$  guideline is *not* a rule and different input combinations can exist, but ensuring their validity becomes much harder. An example of another valid input to a Mix Point not covered by the guideline would be to specify all the exit edge's ( $n_{Ex}(N)$ ) total flow-rates as inputs which in turn will require that one of the input edge's variables is unknown (input to the Mix Point is not fully determined). This different input configuration will backpropagate through  $D$  which will leave a Source with one variable that must not be added as an input and a Sink with one variable that should be added. Specifying inputs in this manner, though possible, is not always an intuitive way to think of a simulation and in most cases, the Sources are known; therefore, the guideline is a good place to start when it is unknown what inputs are required.

### Choosing the exit edge to exclude in the Mix Point equations

Equation 4.13 shows that one of the exiting edges and one component is not used to form an equation. By analysing the equations that are created, it was found that including an edge which does not have a corresponding edge variable as an input in the inputs list leads to a system which has a significantly higher stiffness. Take the Mix Point in the example, if all the entering edge variables are fully defined, and  $S14_{\text{total}}$  is specified as the input five equations (two component balances, two total flow equations and Equation 4.12) will need to be solved simultaneously. That is 5 out of 7 equations that need to be solved simultaneously. By only changing the input from  $S14_{\text{total}}$  to  $S13_{\text{total}}$  requires no simultaneous solution (0 out of 7). The reason  $S13_{\text{total}}$  as the input leads to an easier system to solve is because  $S13_{\text{total}}$  features in Equation 4.12. For the example the edge that was excluded was  $S14_{\text{total}}$ .

The number of variables that will require simultaneous solution when an edge is included in the Mix Point equations, but do not have a corresponding edge variable in the inputs list, has a linear relationship with the number of components in the system as shown in Equation 4.17

$$n_{Eq} = 2 \times n_C + 1 \quad (4.17)$$

when  $n_C \geq 2$ .

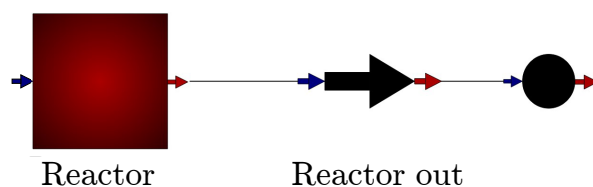
A decrease in stiffness of the simulation makes it easier to solve reducing simulation time. To avoid the edges that will lead to a stiff systems, logic was added to Amoss to check the inputs list and include any edges that have an input variable linked to it.

A similar effect is observed when the input is specified as a component flow instead of the total flow, but the effect is less severe. For the case above selecting either S13<sub>comp\_2</sub> or S14<sub>comp\_2</sub> as an input will require two equations to be solved simultaneously. Currently, the last component in the component list is excluded in the Mix Point equations and no logic is added to select the correct components according to the inputs list. This logic has not been included yet because of the lower impact on stiffness and because it is less likely to specify a component flow as an input relative to a total flow.

#### 4.3.4 Reactor equations

Reactor equations calculates how components are transformed to other components. Around a Reactor, a component mass balance will not hold due to the reaction taking place. This is why the reactor\_data table (Table 4.3) requires the reactor output to be fully specified for all the components.

Reactors are allowed to have multiple edges exiting the Reactor's vertex (see the Cracker top right in Figure 4.1). To enable this ability and create the necessary equations each Reactor unit is substituted with a Reactor unit connected to Mix Point as shown in Figure 4.2



**Figure 4.2:** Effective representation of a Reactor unit.

The Reactor equations are created by taking a component in the out column, e.g. comp\_1 out in Table 4.3 and multiplying all the fractions under that column with the corresponding component in the comp in row (comp\_1 in to comp\_n in) which are the entering component flow-rates to the Reactor and summing them all together. The result

of the summation is the component flow-rate of the component in the out column out of the Reactor.

### Example

Using the Reactor named Reactor in Figure 4.1 and the reactor\_data table in Table 4.4 an example of the Reactor equations are illustrated with Equation 4.18 and Equation 4.19

**Table 4.4:** Reactor data table used for the equation example

node	comp	comp_1 out	comp_2 out
Reactor	comp_1 in	c1i_c1o	c1i_c2o
Reactor	comp_2 in	c2i_c1o	c2i_c2o

$$\text{Reactor\_out}_{\text{comp}_1} = (\text{c1i\_c1o} \times \text{S3}_{\text{comp}_1}) + (\text{c2i\_c1o} \times \text{S3}_{\text{comp}_2}) \quad (4.18)$$

$$\text{Reactor\_out}_{\text{comp}_2} = (\text{c1i\_c2o} \times \text{S3}_{\text{comp}_1}) + (\text{c2i\_c2o} \times \text{S3}_{\text{comp}_2}) \quad (4.19)$$

The number of equations ( $n_{Eq}$ ) and the number of variables ( $n_{Var}$ ) that are created by each Reactor unit is equal to the number of component shown in Equation 4.21

$$n_{Eq} = n_C \quad (4.20)$$

$$n_{Var} = n_C \quad (4.21)$$

$$\text{DOF} = 0$$

A DOF analysis of a Reactor unit is done in Equation 4.22

$$\begin{aligned} \text{DOF} &= \underbrace{n_{Ex}(N) - 1}_{\text{mix}} + \underbrace{0}_{\text{reactor}} \\ &= (n_{Ex}(N) - 1) \end{aligned} \quad (4.22)$$

where  $N$  is the Mix Point vertex.

The DOF analysis indicates that the variables required for inputs can be handled in the same manner as that of the Mix Point equations.

### 4.3.5 Separator equations

Separator equations calculates how components are split into different streams given certain split ratios. The Separator equations that are created for each Separator calculate

the component flow-rates for all the edges listed under the attribute column in the separation\_data table.

To create the equations a single row is selected with a specific exit edge listed under the attribute column in the separation\_data table. The equations for the component flow-rates of the exit edge is created by multiplying the fractions under remaining columns with its corresponding component flow-rate entering the Separator unit. All the equations are created by iterating through all the rows. In addition to the Separator equations Component mass balance equations and Total flow equations are also created.

### Example

Using the Separator called VL101 in Figure 4.1 and the separation\_data table in Table 4.5 Equation 4.23 - 4.26 show the equations that will be created

**Table 4.5:** Separation data table used for the equation example

node	attribute	comp_1	comp_2
VL101	S6	S6_f1	S6_f2
VL101	S7	S7_f1	S7_f2

$$S6_{comp.1} = S4_{comp.1} \times S6\_f1 \quad (4.23)$$

$$S6_{comp.2} = S4_{comp.2} \times S6\_f2 \quad (4.24)$$

$$S7_{comp.1} = S4_{comp.1} \times S7\_f1 \quad (4.25)$$

$$S7_{comp.2} = S4_{comp.2} \times S7\_f2 \quad (4.26)$$

Equation 4.27 and Equation 4.28 shows how many equations and variables are created for every Separator vertex

$$n_{Eq} = (n_{Ex}(N) - 1) \times n_C \quad (4.27)$$

$$n_{Var} = 0 \quad (4.28)$$

$$DOF = n_C - n_{Ex}(N) \times n_C$$

where  $N$  is the Mix Point vertex.

The number of equations created per Separator vertex is equal to the number of edges exiting the vertex negative one (this is a requirement when setting up the separation\_data table) multiplied by the number of components. No new variables are introduced.

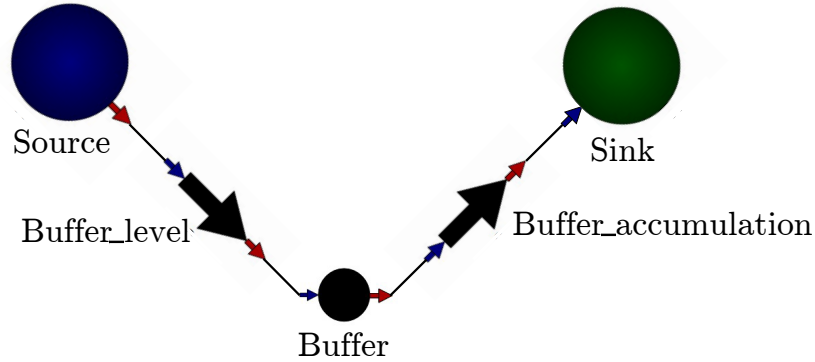
The DOF analysis around every Separator is calculated in Equation 4.29

$$\begin{aligned}
\text{DOF} &= \underbrace{(n_E(N) - 1) \times n_C}_{\text{mass balance}} + \underbrace{0}_{\text{total flow}} + \underbrace{n_C - n_{Ex}(N) \times n_C}_{\text{separation}} - \underbrace{n_{Ei}(N) \times n_C}_{\text{fully defined inputs}} \quad (4.29) \\
&= (n_E(N) - n_{Ex}(N) - n_{Ei}(N)) \times n_C \\
&= 0
\end{aligned}$$

No additional inputs are required for a Separator.

### 4.3.6 Buffer equations

The Buffer equations calculates the composition of the Buffer contents at every times step and subsequently the composition of the edges exiting (assume perfect mixing) the Buffer vertex as well as the level (measured in mass units) of the Buffer by creating the integral equations. To create the necessary equations, each Buffer unit is effectively substituted with a Mix Point with some additional edges connected to it shown in Figure 4.3.



**Figure 4.3:** Effective representation of a Buffer unit.

All the equations associated with the Mix Point equations are created around the effective Buffer unit. From the input line mentioned in section 4.1 it required that all the exiting edges from the Buffer have a corresponding input in the inputs list. On each Buffer vertex, an extra exiting edge is added (shown in Figure 4.3) which is not visible to the user and is the one input that does not need to be specified as an input. With this knowledge the guideline for specifying the inputs for a Mix Point still holds.

It may seem that the edge connected to the Source requires additional inputs to the inputs list, but it is not necessary. The edge exiting the Source represents the contents of the Buffer of the previous time step, and the Buffer\_level variables are carried over between the Euler iteration. The edge entering the Sink, on the other hand, represents the net flow of content in or out of the Buffer. This is the accumulation term of a non-

steady state component mass balance integrated using Euler's algorithm with a single time unit (e.g 1 s, 1 min or 1 h) step size.

When the Amoss project was started it was decided to work on a single time unit step size. That means that if the mass flows rates are given on a per minute basis, the Euler integration will have a one-minute time step and if the flow rates are given on a per hour basis, a one-hour time step will be used. It was also decided to integrate at the beginning of every time step which means that the simulation results that are recorded starts at  $t = 0$  instead of  $t = 1$ .

### Example

Using the Buffer in Figure 4.1 for the example Equation 4.30 and Equation 4.31 shows the integrations variables that are created

$$\text{Buffer\_level}_{\text{comp.1}} = \text{Buffer\_level}_{\text{comp.1}}(t - 1) + \text{Buffer\_accumulation}_{\text{comp.1}} \quad (4.30)$$

$$\text{Buffer\_level}_{\text{comp.2}} = \text{Buffer\_level}_{\text{comp.2}}(t - 1) + \text{Buffer\_accumulation}_{\text{comp.2}} \quad (4.31)$$

The current time Buffer\_level variables is calculated by adding the Buffer\_accumulation variable to the previous time step's Buffer\_level variables.

$$n_{Eq} = (n_{Ex}(N) - 1) \times (n_C - 1) \quad (4.32)$$

$$n_{Var} = 0 \quad (4.33)$$

$$\text{DOF} = -(n_{Ex}(N) - 1) \times (n_C - 1) \quad (4.34)$$

where  $N$  is the Mix Point vertex. Note that  $N$  has the additional edges connected to it. Equation 4.34 is exactly the same as Equation 4.15 because a Buffer is essentially a Mix Point but the additional edges carry different meaning.

In addition to the Buffer equations Component mass balance equations and Mix Point equations are also created. The DOF analysis of the combined equations around a Buffer unit is

$$\begin{aligned} \text{DOF} &= \underbrace{[(n_{Ex}(N) - 1) \times n_C]}_{\text{mass balance}} + \underbrace{[0]}_{\text{total flow}} + \\ &\quad \underbrace{[-(n_{Ex}(N) - 1) \times (n_C - 1)]}_{\text{mix}} \\ &= n_C (n_{Ex}(N) - n_{Ex}(N)) - n_C + n_C + n_{Ex}(N) - 1 \\ &= n_{Ex}(N) - 1 \end{aligned} \quad (4.35)$$

Equation 4.35 is also the same as Equation 4.16 and  $n_{Ex}(N) - 1$  inputs are required but in the case of the **Buffer** unit one edge is not known to the user and from the user perspective  $n_{Ex}(N^*)$  (where  $N^*$  is the **Buffer** vertex excluding the additional added edges) inputs are required

## 4.4 Pre-solve equations

Pre-solving refers to the analytical solving of the system of equations created in section 4.3 before running the simulation. The logic behind this step is to perform most of the solving work before simulations are run to make simulations faster by avoiding numerical solving of equations.

Before any steps are taken to pre-solve the equations, a DOF analysis is done. This analysis ensures at least that the correct number of inputs are specified. At the current date, there is no mechanism in place to guide a user to identify which inputs are required or should be omitted if the DOF  $\neq 0$ . It is possible to use the guideline to check the inputs and in that manner assist the user to select or discard inputs as required, but due to Amoss using the equation orientated approach it would undermine some advantages by using this approach if inputs are required on a modular level. The general guideline provided in section 4.1 stems from a modular approach and is guaranteed to work, but other input combinations are also valid and more difficult to check its validity automatically.

Sympy (Certik et al., 2008) is used to convert all the equations created in section 4.3 to symbolic mathematical expressions. The tearing method described in subsection 2.5.3 is used to tear the system of equations and identify the subsets of equations that require being solved simultaneously. Sympy has a symbolic solver included and it was used to solve the subsets of equations. Initial trials showed that solving a subset larger than around 10 equations was not tractable. This led to the current heuristic of only attempting solution of subsets less than 10.

## 4.5 External user equation

Amoss also provides a platform, called the operating instructions, where relationships and equations can be written to assign a numerical value to the variables listed in the **Inputs** sheet. The operating instructions is also the place where the user can add additional equations.

These equations are used to describe additional process activities and operational philosophy which is impossible to determine from the process connectivity alone. When an input variable or parameter does not stay constant throughout the simulation or a statistical distribution needs to be sampled the operating instructions file is where these equations should be defined.

The operating instructions is saved in a file called “operating\_instructions.txt”.

### 4.5.1 Special operating instruction functions and methods

Special operating instructions functions and methods are available in Amoss and can be used in the operating instructions file. These functions include `allocate()`, `allocate_opt()` and `sample_dist()` (`sample_dist()` is a sampling method from the Distribution object).

`allocate()` and `allocate_opt()` were created due Sasol’s specific request for their functionality. In the MOSS methodology creating the relationships to assign numerical values to the input variables of **Mix Points** are a common phenomenon and is repetitive exercise.

Figure 4.4 is an example of a typical **Mix Point** that is expected with three or more exiting streams. Determining how the feed to the **Mix Point** should be split to mimic current or desired plant behaviour can become difficult because each stream is dependent on the flow limits (maximums and minimums) associated with that specific stream.

The purpose of `allocate()` and `allocate_opt()` is to maximise the flow through the network based on minima and maximum constraints together with allocation priorities. Sasol requested that the functions `allocate()` and `allocate_opt()` should automatically set-up the equations to distribute mass according to a preferred allocation order.

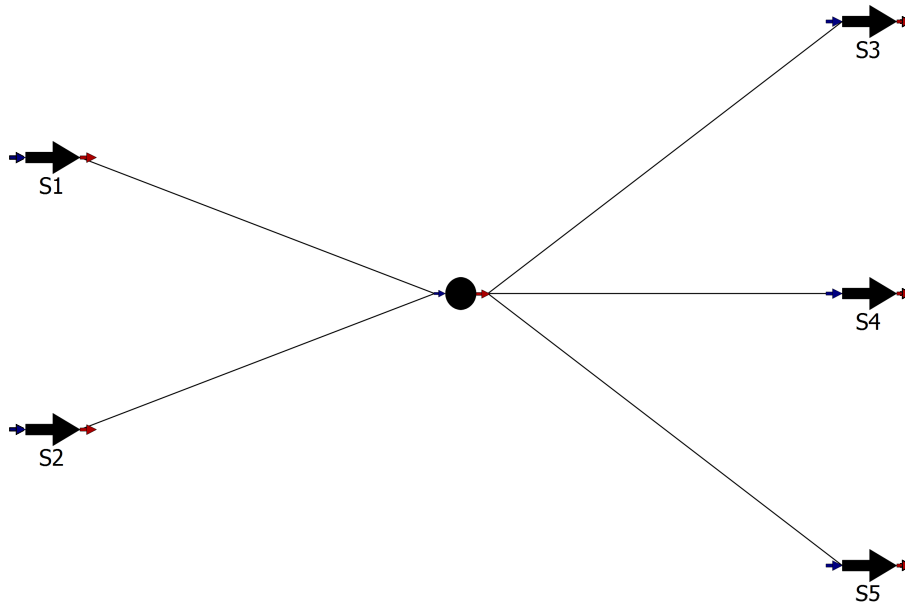
With reference to Figure 4.4 if the preferred allocation order is S3, S4 and then S5 the available mass ( $S1 + S2$ ) is first allocated to S3 until the maximum is reached the remainder to S4 until its maximum is reached and the remainder is allocated to S5. Another constraint frequently encountered when distributing mass is disjoint minimum constraints. These constraints require that the flow must be larger than a minimum or be zero when this minimum cannot be reached. This approach where mass is distributed according to a priority order subject to constraints is common feature in the MOSS methodology.

The allocations can also be viewed as an optimisation problem to minimise the cost through the network given the minimum and maximum constraint with the stream which should be allocated first having the lowest cost with the cost increasing as the priority decreases. If the constraints are local to the distribution point (as it is in Figure 4.4 to the streams S3 and S4) the optimal distribution (according to the priority orders) can be calculated analytically without the use of optimisation software, but when the constraints are not local (downstream or upstream of the distribution point) then a general optimisation is required.

#### **allocate ()**

`allocate()` is used to distribute mass according to a preferred priority order with the minimum and maximum constraints being local to the point of distribution. `allocate()` is one of two instances where the use of multiple assignments are allowed with the other being





**Figure 4.4:** Typical example where mass must be distributed to multiple destinations.

`allocate_opt()`. When using `allocate()` particular care must be taken in the order *assignments* and arguments are written. In Python code the order of variable assignments have no influence on the inner workings of a function, but the operating instructions is not Python code and in the case of `allocate()` the order of variables assignment is important.

The documentation explaining how to use

`allocate(available, minimum_priority, minimum_constraints, maximum_priority, maximum_constraints)` is given below:

**Purpose:** `allocate()` is used to distribute mass and allocate mass flow values to the assignment variables based on priority orders and *local* constraints.

**Assignment order:** In `allocate()` the order of the assignment variables are used as a base to determine the index order of the function arguments e.g. for the assignment:  $x_0, x_1, x_2, \dots, x_n = \text{allocate}(\dots)$  allocation  $x_0$  corresponds to index 0 in all the arguments and allocation  $x_1$  corresponds to index 1 etc.

## Parameters

**available (type: float):** The mass available to distribute. This is a maximum constraint were the sum of the assignment variables must be less than or equal to the available e.g.  $\sum x_0, x_1, x_2, \dots, x_n \leq \text{available}$

**minimum\_priority (type: list of ints):** A list containing the priority order in which the minimum allocations need to be satisfied. The value of index 0 corresponds to the priority of assignment 0 ( $x_0$ ) and the value of index 1 to the priority of assign-

ment 1 ( $x_1$ ) etc. A value of 0 is the highest priority with the priority decreasing as the value increases e.g.

[0, 1, 2, ..., n] indicates that the attempts to satisfy the minimum constraints must be in the order of  $x_0, x_1, x_2, \dots, x_n$  and

[2, 0, 1, ... n] indicates that the attempts must be in the order of  $x_1, x_2, x_0, \dots, x_n$ .

If all the minimum constraints can not be satisfied ( $\text{available} < \sum \text{minimum\_constraints}$ ) then some allocation must be set to zero. This is why the priority order is required. The minimum constraints are attempted to be satisfied according to the priority order and therefore the allocation with the lowest priority will be the first to receive a zero allocation if its minimum constraint is not satisfied.

**minimum\_constraints (type: list of floats):** A list containing the minimum constraints of the allocations e.g.

[10, 20, 30, ..., y] indicates the following constraints:

$$x_0 \geq 10 \text{ or } x_0 = 0$$

$$x_1 \geq 20 \text{ or } x_1 = 0$$

$$x_2 \geq 30 \text{ or } x_2 = 0$$

$$x_n \geq y \text{ or } x_n = 0$$

**maximum\_priority (type: list of ints):** A list containing the priority order in which the maximum allocations need to be satisfied. The value of index 0 corresponds to the priority of assignment 0 ( $x_0$ ) and the value of index 1 to the priority of assignment 1 ( $x_1$ ) etc. A value of 0 is the highest priority and with the priority decreasing as the value increases e.g.

[0, 1, 2, ..., n] indicates that the attempts to satisfy the maximum constraints must be in the order of  $x_0, x_1, x_2, \dots, x_n$  and

[2, 0, 1, ... n] indicates that the attempts must be in the order of  $x_1, x_2, x_0, \dots, x_n$ .

If the minimum constraints have been allocated the remaining mass is distributed based on the maximum priorities. Starting from the highest priorities mass is allocated until all the remaining available mass is zero.

**maximum\_constrains (type: list of floats):** A list containing the maximums constraints of the allocations eg.:

[40, 50, 60, ..., z] indicates the following constraints:

$$x_0 \leq 40$$

$$x_1 \leq 50$$

$$x_2 \leq 60$$

$$x_n \leq z$$

## Returns

**allocations (type: tuple):** The calculated allocations based on the constraints and priority orders.

`allocate()` is very flexible with the priority order of the constraints. The minimum and maximum constraint priorities orders can be different from one another, and it can be changed independently during each time step in the simulation. This changeable priority order makes `allocate()` very versatile with the ability to accommodate different control philosophies across a simulation and can even be extended to change during different scenarios.

Figure 4.4 is used to demonstrate an example of how to use `allocate()`. For this example let the available mass  $(S_1 + S_2) = 55$ . The disjoint minimum constraint are  $S_3 \geq 10$  or  $S_3 = 0$  and  $S_4 \geq 20$  or  $S_4 = 0$  with  $S_3$  having the higher minimum priority. The maximum constraints imposed are  $S_3 \leq 30$  and  $S_4 \leq 40$  with  $S_4$  having the higher maximum priority.

If the guideline on selecting the inputs are followed only two of the three exit streams can be inputs, and no allocation can be done on the remaining stream. In this example, it was chosen to list  $S_3$  and  $S_4$  as inputs, and therefore no allocation can be done on  $S_5$ . The exclusion of  $S_5$  will ensure that the mass balance still holds; therefore, allocations can only be made on input variables.

To calculate the allocation for  $S_3$  and  $S_4$  the following line could be used:

```
S3, S4 = allocate(50, [0, 1], [10, 20], [1, 0], [30, 40])
```

The allocation will result in  $S_3 = 15$  and  $S_4 = 40$ . The algorithm that is followed (Figure 4.4) will ensure that the minima are allocated but because  $S_4$  has the higher maximum priority the maximum constraint on  $S_4$  was reached. The remaining mass that was left after the  $S_4$  allocation was allocated to  $S_3$ .

## `allocate_opt()`

When the constraints are not local to the distribution point `allocate_opt()` can be used to distribute the mass while taking downstream or upstream constraints into account. Similar to `allocate()` the assignment order is also important but it does not form the base order of the function arguments but instead serves as the preferred allocation order.

When the constraints are not local the idea to link a minimum and maximum priority order to an allocation variable does not make sense anymore. This idea of a priority order does not work because the influence on a single constraint variable cannot necessarily be isolated to a single allocation variable and more than one allocation variable can have

an interacting influence on the constraint variable. Due to this, it was decided to have a minimum constraint priority and an allocation priority.

The difference between the `allocate()` minimum constraint priority and `allocate_opt()` minimum priority is that with `allocate()` the minimum priority coincided with a specific allocation variable whereas with `allocate_opt()`, values are assigned to any allocation variable in order to meet the constraint. When all the minimum constraints have been met the flow through the network is optimised based on the maximum allocation priority without exceeding a maximum constraint.

`allocate_opt()` is not as flexible as `allocate()` and a fixed priority order is forced throughout the simulation because the ability to change objective function or the implemented branch and bound procedure (see parsing `allocate_opt()`) interchangeable during each time step is not currently possible. The documentation explaining how to use `allocate_opt(available, minimum_constraints, maximum_constraints)` is given below:

**Purpose:** `allocate_opt()` is used to distribute mass and allocate mass flow values to the assignment variables based on priority orders and constraints.

**Assignment order:** In `allocate_opt()` the order of the assignment variables are used to determine the preferred flow allocations when the minimum constraints are satisfied. The objective function is created by giving a lower cost to the assignments that are written first and increasing the cost progressively e.g. for the assignment:

$x_0, x_1, x_2, \dots, x_n = \text{allocate\_opt}(\dots)$  the objective function is set up in the following manner:  $-x_0 \cdot f^0 - x_1 \cdot f^{-1} - x_2 \cdot f^{-2} - \dots - x_n \cdot f^{-n}$

In the objective function  $f$  is the scale factor and has a value of 2 (2 was somewhat arbitrarily chosen but provided the desired result) giving the first allocation the lowest cost (-1) and progressively increasing the cost ( $-0.5 \rightarrow -0.25 \rightarrow \dots \rightarrow -1/2^n$ ).  $f$  is currently hard coded and not changeable by the user.

## Parameters

**available (type: float):** The mass available to distribute. This is a maximum constraint were the sum of the assignment variables must be less than or equal to the available e.g.  $\sum x_0, x_1, x_2, \dots, x_n \leq \text{available}$

**minimum\_constraints (type: dict or None):** A dictionary with the keys the variables with minimum constraints and the values are the minimum constraints. The order in which the keys are added is used to determine the minimum priority order e.g.

$\{z_0: 10, z_1: 20, z_2: 30\}$  have a minimum priority order of  $z_0, z_1$  and then  $z_2$  (where  $z_0, z_1$  and  $z_2$  may be non-local constraints). It will be attempted to satisfy the  $z_0$  constraint first then  $z_1$  etc.

The minimum constraint dictionary will produce the following disjointed constraints:

$$z0 \geq 10 \text{ or } z0 = 0$$

$$z1 \geq 20 \text{ or } z1 = 0$$

$$z2 \geq 30 \text{ or } z2 = 0$$

The argument `None` will indicate that there are no minimum constraints.

**maximum\_constraints (type: dict or None):** A dictionary with the keys the variables with the maximum constraints and the value are the maximum constraints. For the maximum constraints the order of the dictionary has no meaning. For a maximum dictionary of e.g. `{y0: 40, y1: 50, y2: 60}` the following constraints are created:

$$y0 \leq 40$$

$$y1 \leq 50$$

$$y2 \leq 60$$

The argument `None` will indicate that there are no other maximum constraints

## Returns

**Allocations (type: tuple):** The optimised allocations based on the constraints and priority orders.

In Python, the order in which dictionaries are created are not persistent, but in Amoss the order is exploited to convey meaning in `allocate_opt()` making the function more compact and the operating instructions cleaner.

## sample\_dist()

The distribution object contains all the distributions that are listed in `scipy.stats` together with any user-defined distributions. To define a custom distribution an Excel file called “stochastic\_distributions.xlsx” is required together with a table with values and its corresponding probability. For every new distribution, a new sheet is required, and the sheet name is the name that should be used to sample the distribution in operating instructions. Table 4.6 is an example of a custom user-defined distribution.

To activate the option to sample the distribution discretely set the value next to “Discrete” in the distribution table to 1 (0 indicates that the distribution is to be sampled continuously). The values and its corresponding probability are listed in the columns “values” and “p” respectively.

On a technical level, any distribution defined by the user is discrete. To sample these distributions continuously the probability density function is approximated by linearly interpolating between the data points making it piecewise linear. Therefore, if more

**Table 4.6:** Example of how a user can define a distribution.

Discrete	0
value	p
0	0
1	0.0625
2	0.1250
3	0.1875
4	0.2500
5	0.1875
6	0.1250
7	0.0625
8	0

data points are supplied it will increase the accuracy, but if the required distribution is available in `scipy.stats` it is recommended to use these because analytical equations are used to sample these distributions.

The user-defined distributions are numerical representations; therefore, numerical integration is used to find the cumulative distribution. For the continuous case, the trapezium rule with a variable step size<sup>1</sup> is used for integration. If it is assumed that the piecewise linear approximation is the most accurate approximation of the probability density function, then the trapezium rule will have zero error for the integration of the piecewise linear density function. The piecewise linearity of the approximation is exploited by using a variable step size which is the difference between each of the listed values in Table 4.6. Using the variable step size in this manner eliminates the need to provide a step size making the integration calculation much simpler without the loss of accuracy. In the discrete case, the numerical integration is 100% accurate.

Listing 4.1 shows how a probability density function is integrated using Numpy.

```
1 import numpy
2
3 # calculate the perpendicular height of each trapezium
4 h = values [1:] - values [: -1]
5
6 # calculate the sum of the parallel sides
7 sum_parallel_sides = probabilities [: -1] + probabilities [1:]
8
9 # calculate the area of each trapezium and calculate the cumulative sum to
  get the cumulative distribution
10 cum_integrate = numpy.cumsum(0.5*h*sum_parallel_sides)
11
12 # ensure the cumulative sum equals to 1
```

---

<sup>1</sup>not fixed

```

13 cum_integrate /= cum_integrate[-1]
14
15 # the cumulative sum does not start at zero and a zero is added at the
    start making the cumulative distribution start at zero
16 cum_integrate = numpy.append([0], cum_integrate)

```

**Listing 4.1:** Numerical integration of the piecewise linear approximation of a user-defined probability density function using the trapezium rule.

In line 13 the cumulative distribution is divided by the last value in the cum.integration array to ensure that the approximated  $F$  is bounded by  $[0, 1]$  because this is a requirement of a cumulative distribution.

The discrete integration is far simpler than the continuous case shown in Listing 4.2.

```

1 import numpy
2
3 # obtain the cumulative sum of all the probabilities
4 cum_integrate = numpy.cumsum(probabilities)
5 cum_integrate /= cum_integrate[-1]

```

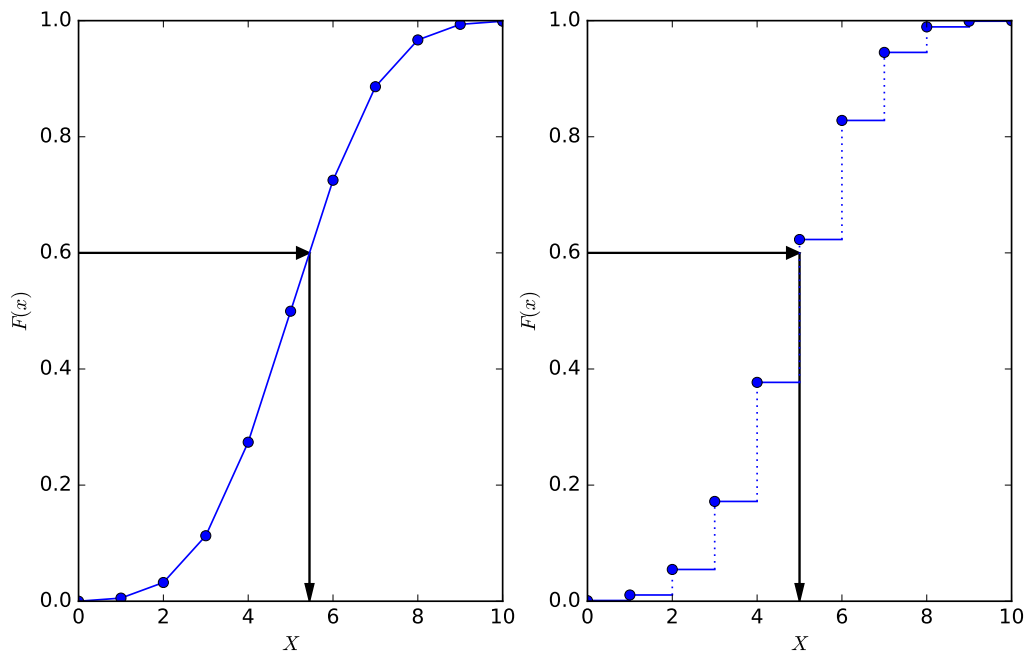
**Listing 4.2:** Numerical integration of a discrete user defined probability mass function.

As a safety measure, the cumulative distribution is scaled to ensure that it is bounded by  $[0, 1]$  if the user made a mistake.

To generate the non-uniform random numbers the same principles in subsection 2.6.2 are applied, but because the cumulative distributions are numerical and also piecewise linear the graphical approach is followed. Figure 4.5 shows the graphical approach that is followed.

A uniform random variable  $U$  on  $[0, 1]$  is created and a horizontal line on  $y = U$  is drawn. The  $x$  value which corresponds to the location where  $y$  crosses the cumulative distribution curve is the non-uniform generated variable. In Python the `random()` function from the `random` library is used to generate  $U$ .

In Amoss the graphical approach is achieved by using the `interp()` method in Numpy. Listing 4.3 shows how the graphical method was implemented in Python. The `cum.integrate` arrays that were created in Listing 4.1 and Listing 4.2 contains only discrete endpoint on of the cumulative distribution curve but because it is piecewise linear the in-between values can be obtained by simple interpolation.



**Figure 4.5:** Graphical method followed to generate non-uniform random variables with a continuous example left and a discrete example right.

```

1 import numpy
2 import random
3
4 # generate a uniform random variable on [0, 1]
5 random_number = random.random()
6
7 # calculate the non-uniform random variable X
8 X = numpy.interp(random_number, cum_integrate, probabilities)

```

**Listing 4.3:** Graphical non-uniform random variable generation implemented in Python using `numpy.interp()`.

## 4.5.2 Parsing the operating instruction

The `operating_instructions.txt` file is parsed using the `ast` (abstract syntax tree) library in Python. The `ast` library makes it possible to parse a string and break it down into node objects with a specific meaning which can be used to identify different forms of grammar. Only three different Python grammar types are allowed in Amoss: assignments, if-blocks and augmented assignment types. A description of these grammar types are given below:

**Assignment (assign)** is when a variable or multiple variables are assigned to an expression. These expressions can range from a mathematical expression to objects and function calls. An assignment is identified by variables followed by the equal operator eg. `a = ...` or `x, y = ...`



**If-block** is the entire collection of the if and else-if and an else conditions together with the equations belonging to each condition.

**Augmented assignment (augassign)** is assignment type where the assignment variable also features in the expression but is not written explicitly eg.  $a = a + b$  is equivalent to the augmented assignment of  $a$  by using the  $+=$  assignment in  $a += b$ .

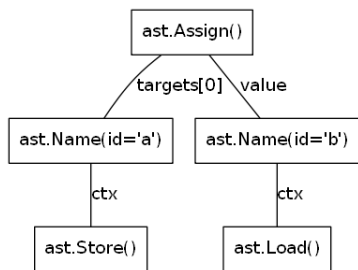
Listing 4.4 is an example that shows how `ast.parse()` parses a string. The string is created in line 3 and contains all the grammar types allowed in Amoss. When `ast.parse()` is called it creates an object with the variable `.body` which contains a list with node objects. The assignment in line 5 and 8 are parsed as `Assign` types and corresponds to index 0 and 1 in `parse_ast.body`. The if-block which stretches from line 11 to 18 is parsed as an `If` type in index 2. Note that all the information of the entire if-block is contained in the `If` object as an `ast`. Last is the `AugAssign` type in line 20 which was parsed and can be found in index 3. A further advantage of using the `ast` library is that it recognises comments within a string and does not parse it.

```
1 import ast
2
3 pars_string = """
4 # simple assignment
5 a = b
6
7 # assignment calling a function
8 c = min(a, 1, 2)
9
10 # if-block consisting of an if-statement, else if-statement and an else
11 if c > 0:
12     d = 5
13
14 elif c <= 0:
15     d = -5
16
17 else:
18     d = 0
19
20 e += d
21 """
22 parse_ast = ast.parse(pars_string)
```

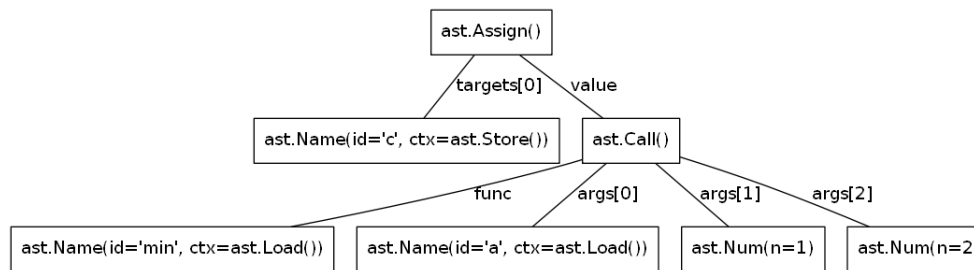
**Listing 4.4:** Example of how the `ast.parse()` methods parse a string into a list of node objects.

The syntax trees that are created by calling `ast.parse(pars_string)` is shown in Figure 4.6 to Figure 4.9. These trees can be navigated in the variable `parse_ast`.

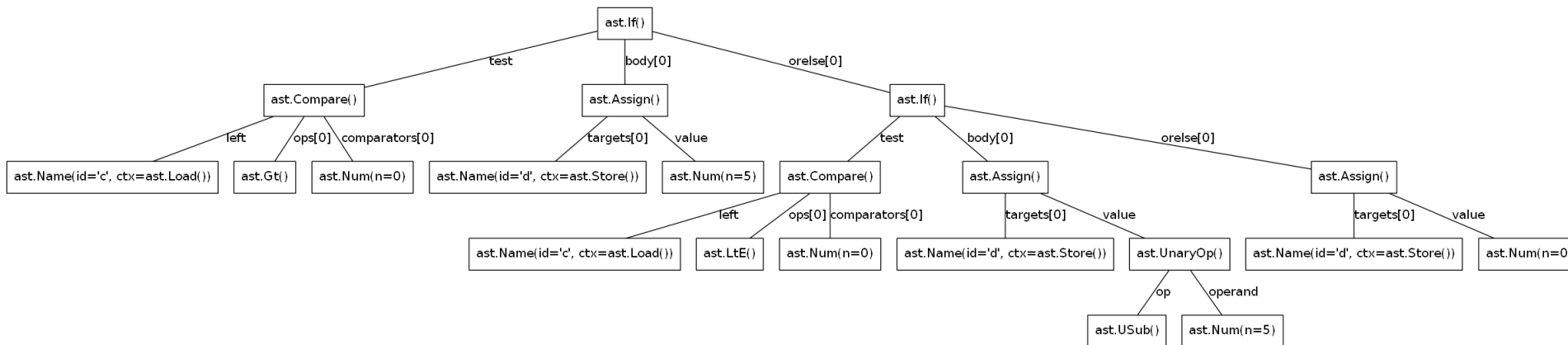
Depending on which of these grammar types are encountered in the iteration through the `.body` list different procedures are followed to emit the operating instructions.



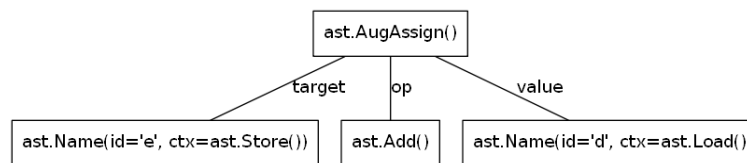
**Figure 4.6:** The ast created for the assignment variable a.



**Figure 4.7:** The ast created for the assignment variable c.



**Figure 4.8:** The ast created for the if-block.



**Figure 4.9:** The ast created for the augmented assignment variable e.

## Assign type

If a node from the parsed operating instructions is identified as an assignment, the node value<sup>2</sup> is further examined to identify if a function call was made. If the node is not an `ast.Call` type it is assumed to be a mathematical equation and is added to the system of equations.

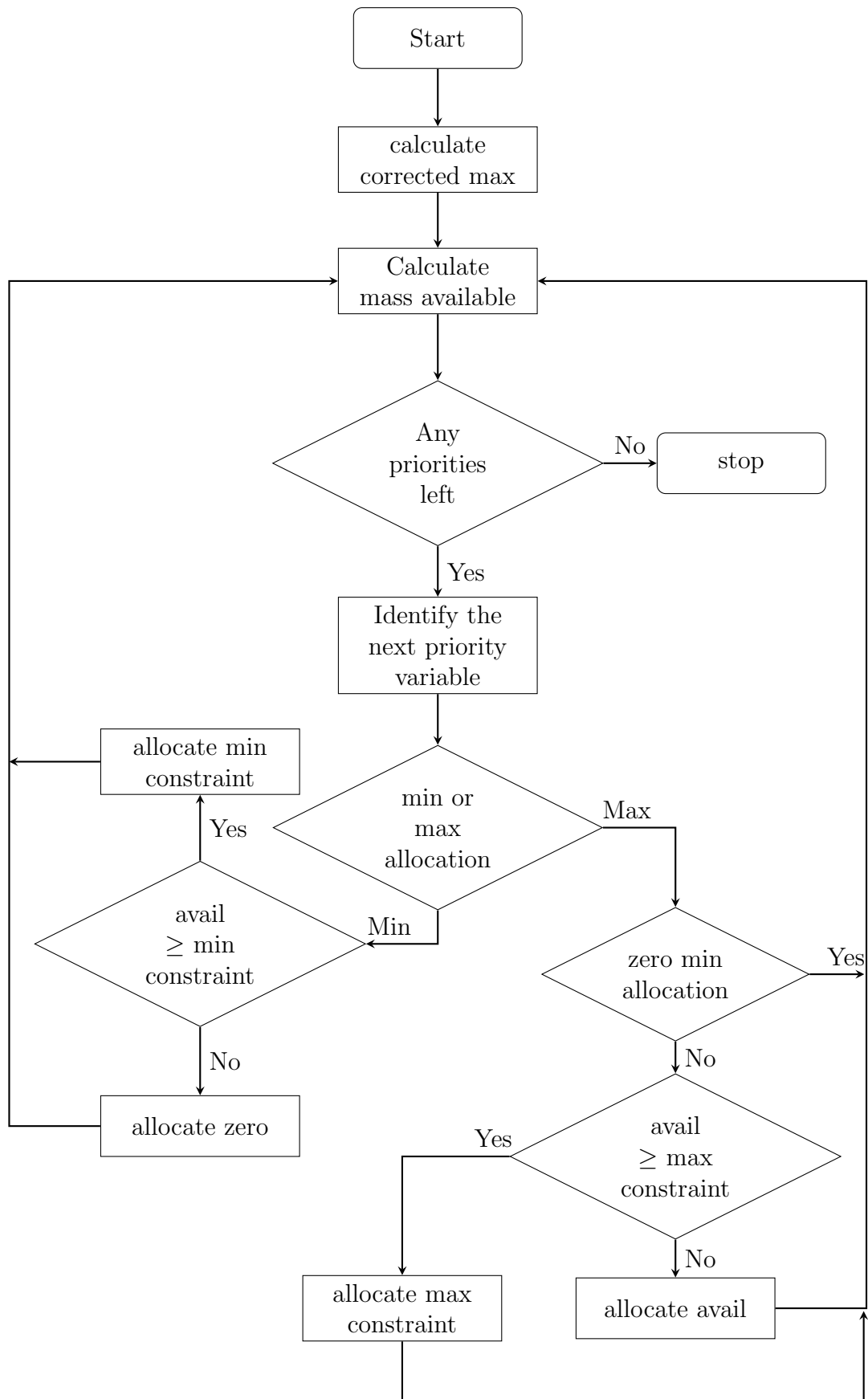
For Amoss only specific function calls are allowed: `min()`, `max()`, `allocate()`, `allocate_opt()` and `sample_dist()`. The function name can be identified by accessing the `id` parameter in `value.func.id`. If the function name is `min` or `max` the assignment is emitted as a normal equation and the assignment variable is stored for use in the code generation (see subsection 4.6.1) part of Amoss. If the function name is `sample_dist` an equation is created which samples the desired distribution and the assignment variable is stored for use in the code generation (see subsection 4.6.1) part of Amoss.

### **allocate()**

When `allocate()` is identified as a function call, extra equations are created which will calculate how the flow is allocated. The exact equations that are formed are a bit cumbersome to write here, and therefore the algorithm which these equations follow will rather be explained. Figure 4.10 is the algorithm diagram which the created equations follow.

---

<sup>2</sup>Right-hand side of the assignment.



**Figure 4.10:** Algorithm that is followed by the created allocate equations.

**Calculate corrected maximum** is a calculation where the minimum constraint of a stream is subtracted from its maximum constraint. This subtraction is done to accommodate the manner in which the streams are allocated. Allocations are performed in two steps: first, the minimum allocations are made and then only after the minima are allocated is the maximum constraint considered. If there is enough mass to reach the maximum constraint the corrected maximum is added to the already allocated minimum constraint value. To make this calculation easier the difference is already taken into account and called the corrected maximum.

**Calculate mass available** is performed at the start of each iteration. This calculation involves all the allocated streams subtracted from the total available mass (first argument for `allocate()`). On the first iteration, no streams have been allocated and the available mass is the total available mass.

**Any priorities left** is a check which is added to terminate the iteration. If all the priorities have been allocated, and there are none left the algorithm terminates, but if there are any unallocated priorities left proceed with the algorithm.

**Identify the next priority variable** identifies the assignment variable with the current iteration's highest priority. As it is explained in section 4.5.1 the assignment variables have different minimum and maximum constraint priority orders. The minimum constraints have the highest priority and are first identified, and the maximum priority order is identified afterwards. In the example in section 4.5.1 the priority order will be S3 minimum, S4 minimum, S4 maximum and S3 maximum.

**Min or max allocation** identifies if the current priority is a maximum or a minimum allocation.

**Avail  $\geq$  min constraint** checks if the available mass is greater than the minimum constraint, but if the available is less than the minimum constraint then the allocation should be zero (**allocate zero**). If the available mass is larger than the minimum, then allocate the minimum constraint (**allocate min constraint**).

**Zero min allocation** is a check to identify if a variable was allocated a zero in the **avail  $\geq$  min constraint** decision. If the variable was allocated a zero, then no additional mass may be allocated to that variable because the minimum constraint can already not be satisfied. An example where this may occur is when one of the variables (call it S10) has a very high minimum constraint which can almost never be met with other variables (call them Sx) with low maximum constraints. By the time the maximum constraints are allocated all the other maxima (Sx maxima) can be satisfied with available mass left. This mass that is left cannot be allocated

to S10 because the high minimum constraint cannot be satisfied and the available mass left will never be enough to satisfy this constraint.

**Avail  $\geq$  max constraint** checks if the available mass is greater than the minimum constraint (corrected maximum). If the available mass is larger than the maximum, then allocate the maximum constraint, but when the available is less than the maximum constraint it works differently to the minimum constraint allocation. In this case the all the available mass is allocated to that variable.

### **allocate\_opt()**

When `allocate_opt()` is identified as a function call an NLP optimisation problem is created as opposed to an additional set of equations in `allocate()`. In `allocate()` the minimum and maximum constraints are local to the point of distribution and it is possible to find the optimal solution analytically, whereas it is not possible in all cases when the constraints are not local. Equation 4.36 shows the NLP problem that is created

$$\text{Minimise: } f(x) \tag{4.36a}$$

$$\text{subject to: } \Sigma x \leq \textit{available} \tag{4.36b}$$

$$h(x) \leq \textit{max} \tag{4.36c}$$

$$g(x) \geq \textit{min} \tag{4.36d}$$

where  $x$  is the allocation variables,  $h$  is the relationship of the allocation variables to the maximum variable constraints and  $g$  the relationship of the allocation variables to the minimum variable constraints.

When two or more `allocate_opt()` calls are made they are combined together under one single optimisation problem as it is presented in Equation 4.36. In this case the order they appear in the operating instructions file is important and it is assumed that they appear in the same order as they are encountered in the graph when moving in a forward manner. The highest priorities are assigned to the allocations in the first `allocate_opt()` call and the lowest to the last allocations in the `allocate_opt()` call that appears last.

Considering the feasible region of Equation 4.36 and notice that it is possible to have *no* feasible region. As an example take the case where *available* is 0 then Equation 4.36b and Equation 4.36d will never be satisfied if  $\Sigma \textit{min} > 0$ . In such a case there is no feasible solution and the problem can not be solved. In section 4.5.1 it was explained that when a situation like this occurs that some variables in Equation 4.36d are removed and replaced by equality constraints of zero. The modified NLP problem is given in Equation 4.37

$$\begin{aligned}
& \text{Minimise: } f(x) \\
& \text{subjected to: } g(y) = 0; \quad y \subseteq x \\
& \quad \quad \quad \Sigma x \leq \textit{available} \\
& \quad \quad \quad h(x) \leq \textit{maxima} \\
& \quad \quad \quad g(z) \geq \textit{minima}; z = \{x\} - \{y\}
\end{aligned} \tag{4.37}$$

The challenge in solving Equation 4.37 is identifying what subset  $y$  of  $x$  must be equal to zero especially when two or more `allocate_opt()` calls are encountered. By recognising that the minimum constraints are in fact disjunctive<sup>3</sup> ( $g(x) \geq \textit{minima}$  or  $g(x) = 0$ ) a big-M approach could be implemented. Another option, which was easier to implement (referring to coding considerations) and was the implemented<sup>4</sup> method in Amoss is to remove the maximum constraint (Equation 4.36c) and turn the minimum constraint (Equation 4.36d) into binary constraints of zero or the minimum. The MINLP of this formulation is given in Equation 4.38

$$\text{Minimise: } f(x) \tag{4.38a}$$

$$\text{subjected to: } \Sigma x \leq \textit{available} \tag{4.38b}$$

$$g(x) = 0, \textit{minima} \tag{4.38c}$$

A branch and bound approach is followed to solve Equation 4.38. The optimal solution of Equation 4.38 will identify the minimum constraints that must equal zero ( $g(y) = 0$ ,  $y \subseteq x$ ). Due to this method, it is of *paramount importance* that the distribution point must be fully bounded by minima (no allocation variables  $x$  may have a path in the process graph that has no minimum constraint) otherwise it is possible that optimisation may falsely identify minima that need to be zero.

Figure 4.11 shows the algorithm that determines the optimal distribution of the combined `allocate_opt()` optimisation problem.

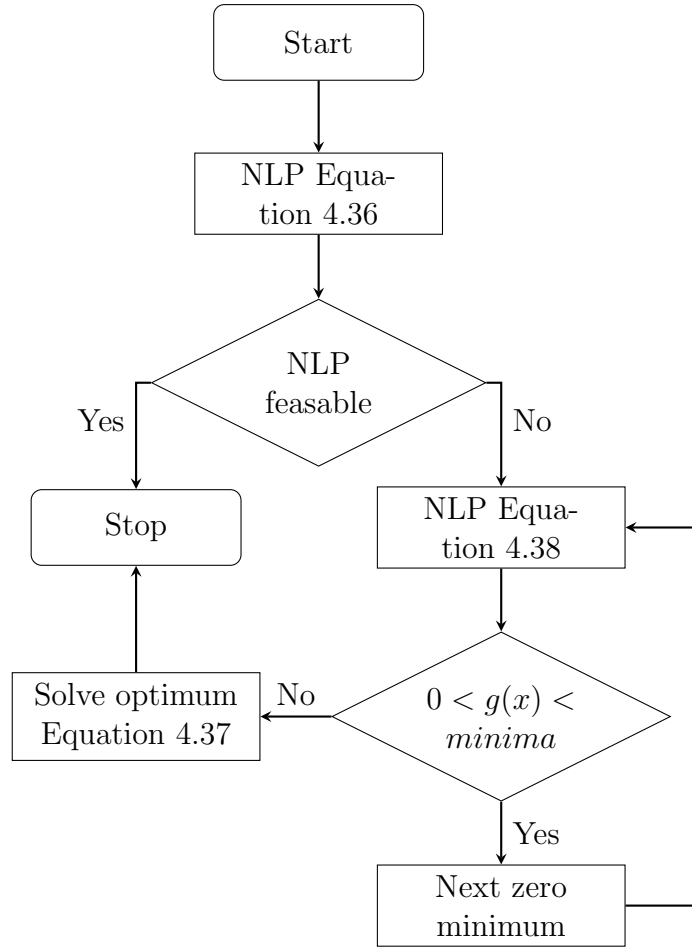
**NLP Equation 4.36** attempts to find the optimum distribution given the constraints.

**NLP feasible** is a check that determines if Equation 4.36 has a feasible solution, and if a feasible solution is found the algorithm terminates. It is expected that in most iteration the solution will be found by Equation 4.36. If an infeasible problem is detected, it is most likely due to the available mass not being enough to meet all the minimum constraints, and the branch and bound procedure is started.

---

<sup>3</sup>See section 2.3.4

<sup>4</sup>At this stage time restrictions for the project was starting to be reached and the method that could be implemented easier was used



**Figure 4.11:** Algorithm that is followed by `allocate_opt` to solve the disjunctive constraints.

**NLP Equation 4.38** solves the relaxed optimisation problem with Equation 4.38c relaxed to  $0 \leq g(x) \leq minima$ .

$0 < g(x) < minima$  checks if *any* minimum constraint is not at its boundary (at 0 or *minima*). If all the minima satisfy Equation 4.38c then continue to find the optimum solution otherwise branching is required.

**Next zero minimum** is the branching step in the branch and bound procedure. It is known that if any of the minima are not at their boundary that the available mass is not enough and that at least one minimum must be zero. Due to this fact it is not necessary to solve both of the binary branches but only the branch which fixes one minimum to zero. Thus, only a single branch needs to be calculated effectively cutting the computational load in half compared to a traditional branch and bound. This situation was exploited in `Amoss` and the remaining branch will not be calculated. The **next zero minimum** block therefore only needs to identify the minimum that needs to be fixed to zero. The decision made in this block is based on the priority of the minimum. The minima that will be fixed to zero is identified



by starting at the lowest priority and working its way up the rank as the iteration continues.

If there is more than one `allocate_opt()` then the minimum with the lowest priority starts at the first `allocate_opt()` and the minimum with the highest priority ends at the minimum with the highest priority in the last `allocate_opt()`. In this configuration, it starts with the minima with the lowest priority equal to zero first and also starting with the allocation that is first encountered in the graph. First chipping away at the first allocation and working it downstream is important because the distribution decided upstream will determine how much mass is available for distribution downstream.

**Solve optimum Equation 4.37** finds the optimal distribution. The subset  $y$  is found in the branch and bound procedure and is all the minima that were set to zero. The cost of implementing the easier branch and bound instead of using the big-M approach is seen in this block which requires one additional optimisation problem to solve to find the optimal distribution. Equation 4.38 is a completely different optimisation problem than the one stated in Equation 4.36 and by solving Equation 4.38 only assists in determining which minima need to be fixed to zero. This information is then used to find the optimal distribution.

## If type

When an if-block is identified the entire block is parsed as an additional set of equations. For every if-blocks in the operating instructions a unique base variable is created based on its location in the operating instructions file. The first if-block in the operating instructions will have the base variable `if0` with the second `if1` etc. These base variable are used to create additional auxiliary variables to parse an if-block into a set of equations.

An if-block consist of the following four parts:

**If-statement** is the first condition found in an if-block and identified by the Python grammar word `if`. It is required that there is only one if-statement per if-block.

**Elif-statement** is the follow-up conditions of the if-statement and any other elif-statements that came prior and is identified by the Python grammar `elif`. There can theoretically be any number of elif-statements, including none.

**Else** is the last condition of the if-block. The else is an implied condition which is `True` if all other conditions before the else is `False`. Else is identified by the Python grammar word `else` and is optional.

**Conditional equations** are the equations that should be evaluated if any of the conditions in the if-block are `True`.

The equations that are created when parsing an if-block is better explained using the example if block in Listing 4.4 which is rewritten in Python grammar in Listing 4.5.

```

1 if c > 0:
2     d = 5
3
4 elif c <= 0:
5     d = -5
6
7 else:
8     d = 0

```

**Listing 4.5:** Extraction of the if-block example in Listing 4.4.

In this example, the base variable will be `if0`. When parsing an if-block, the following equations are created:

- Using the base variable the if-statement is parsed to  $\text{if0}_0 = c > 0$ . The if-statement will always have the `_0` extension to the base variable.
- The elif-statements are parsed from the top to bottom of the if-block. In the example the elif-statement is parsed as  $\text{if0}_1 = c \leq 0$  and  $\text{if0}_0 == 0$ . The extension `_1` is the distance of the elif-statement from the if-statement. Notice that there is a second condition when parsing elif-statements. The second condition will always stay the same except the extension of the auxiliary if variable is one less the extension of the currently parsed if variable. In this case, the extension number was 1 and the second condition extension was 0 (1 - 1). This second condition ensures that only one of the auxiliary if variable evaluates to 1 (one<sup>5</sup>).
- The else is parsed to the equation  $\text{if}_2 = \text{if}_1 == 0$ . The extension of the else `_2` is also the distance of the else from the if-statement. The condition in the else equation is the same as the second condition in the elif equation.
- The conditional equations are parsed based on the condition it belongs to and the auxiliary if variable created for that condition. In this example the following equations will be created  $\text{if0}_0.d = 5$ ;  $\text{if0}_1.d = -5$ ;  $\text{if0}_2.d = 0$ . The equations are simply the auxiliary if variable with the assignment name as the extension.
- Finally, the true value of the assignment is calculated by creating the equation  $d = \text{if0}_0.d \times \text{if0}_0 + \text{if0}_1.d \times \text{if0}_1 + \text{if0}_2.d \times \text{if0}_2$ . The equation is the sum of the conditional equation multiplied by its auxiliary if variable. Due to one auxiliary if variable evaluating as 1 all the other variable are 0, and in this manner, the correct conditional equation is selected.

---

<sup>5</sup>When a statement is True it evaluates to 1 and when it is False 0

The method described above to parse an if-block has a drawback: all conditional equations and all conditions are evaluated and costs computing time. For example if the if-statement in Listing 4.5 is true only two calculations are needed. One to evaluate the condition and the other evaluating the equation. In the method that is implemented seven calculations will always be required no matter which condition is True. The sole reason this was done is because it made it possible to order the if-block as normal equations in section 4.6. Attempts were made to parse an if-block as a single dummy equation, but the problem was that this dummy equation would sometimes feature in a residual equation which did not make sense. In the implemented manner the entire if-block is just another set of equations.

### **Augassign type**

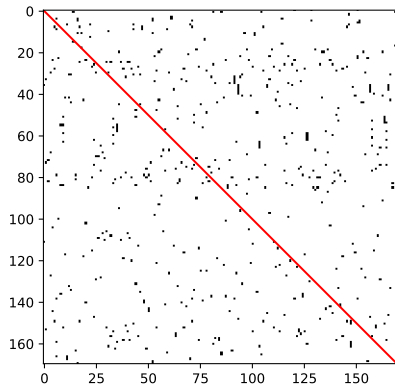
The augmented assignments are parsed as is and are not considered as part of the set of equations. In the simulation these equations are evaluated before the set of equations is solved. The assignments of these equations are considered inputs to the set of equations.

## **4.6 Ordering to bordered lower triangular form**

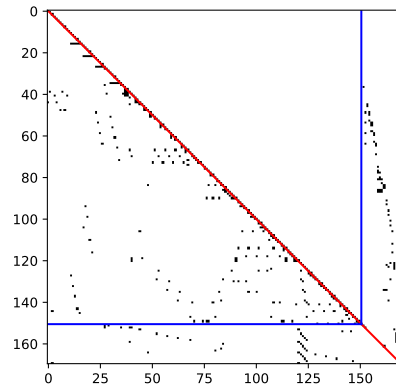
The purpose of ordering the full system of equations (including the parsed equations (subsection 4.5.2)) was to create a system which is easier to solve than the unordered system. Figure 4.12 is an example of an incidence matrix using a test process of Amoss with 171 variables and equations.

To solve a system in the state as it is in Figure 4.12 will require the entire system of 171 variables to be solved simultaneously. Using the bordered lower triangular ordering algorithm of Baharev (2017a) on the system in Figure 4.12 results in the incidence matrix in Figure 4.13. The order system in Figure 4.13 requires only 19 variables that need to be guessed. Notice that the first block in Figure 4.13 is completely lower triangular and these first 152 equations can be solved explicitly and sequentially. The remaining variables have to be solved using a numerical solver, and the remaining equations are the residual equations used in the root-finding software.

Due to the advantage of tearing it was decided to make tearing a part of Amoss. The code written by Baharev (2017a) requires as an input a sparse incidence matrix  $A$ , but the full system of equations are saved in string form. To create  $A$ , a list is created containing all the variables that are created in the automatic equation generator (section 4.3) as well as any new variables introduced in the parsing of the operating instructions. The indices of these variables in this list (called the variable list) correspond to a variable ID (the variable ID is the index plus one). To identify which variables occur in each equation (excluding the created equations for the if-, elif-, and else statements as well



**Figure 4.12:** Example of a square incidence matrix of 171 rows and columns.



**Figure 4.13:** Reorder incidence matrix to the bordered lower triangular form.

as stochastic variable sampling equations) are converted to Sympy expression trees using `sympy.sympify()`. The variables taking part in an equation can very easily be obtained by calling the `.free_symbols` parameter of a Sympy expression. The following example illustrates this:

Consider the Sympy expression `eq = sympy.sympify("a - b*c")`. By calling the `eq.free_symbols` parameter the following Python set will be returned: `{a, b, c}`.

Before the matrix is built, a DOF check is done to ensure that the number of variables and equations are still equal. If the check passes, only then  $A$  will be built.  $A$  is built by iterating through the equations (matrix rows) and finding the variables (matrix columns) partaking in the equation and adding the variable ID as the entry in the correct column (the variable index still corresponds with the column index of  $A$ ). This why the variable IDs can not correspond to the variable list's indices. The variable in the 0'th index will then correspond to a zero entry in  $A$ , effectively not being there. The use of the variable ID will become clear in subsection 4.6.1 when retrieving the equation order.

The bordered lower triangular method by Baharev (2017a) does not establish the feasible assignments automatically and the infeasible assignments need to be indicated by making the corresponding entry (ID) in  $A$  negative for that variable in the corresponding row. To identify the feasible and infeasible assignments Baharev (2017b) made a repository available on GitHub called `safe-eliminations` (Baharev, 2017b). The module requires a sympy equation and bounds for the variable in the equations. The module returns a Python dictionary with the feasible assignment as the key and the symbolically solved equation as the value. When iterating through the equations to build  $A$  each equation is checked for feasible assignments and if any variable is infeasible for that equation the negative value of the ID is entered in  $A$ .

The if-block equations not converted to Sympy expression trees are stored as special

equations and the variables identified and stored when they were parsed. They are added to  $A$  in the same way, except Sympy is not used to find the variables partaking in the equations. The reason why these variable were not sympified is because Sympy cannot create expression trees for all types of logic statements. Below is a list explaining the shortcomings:

- An equation containing an equality condition is evaluated and a Sympy tree is not built eg. `eq = sympy.sympify("a == b")` results in a `False`. This is for any equality.
- Sympy is unable to sympify an equation containing the `and/ or` operator and returns an `TypeError`. It is well able to sympify equations looking like: `sympy.sympify("a == b and a > c")` or `sympy.sympify("True and a > c")` or `sympy.sympify("False or a == b and a > c")`. The result of these exceptions are still only `True` or `False`.
- It can only successfully sympify equations having a single inequality eg. `eq = sympy.sympify("a >= b")`.

It is not feasible to limit the user to a single inequality per statements and that is why these equations are added as special equations to  $A$ . These if-block equations are not strict analytical equations where it is possible to eliminate a variable by symbolic manipulation. Therefore, all the variables in these equations are infeasible assignments except the assignment variable created from the base if-variable and are indicated accordingly by making the IDs negative in  $A$ .

The other equation type that was not sympified was stochastic variable sampling equations. These equations are numerical (not analytical) in the way they are presented in Amoss and these variables are not differentiable, which poses problems when the simulation code is generated (see subsection 4.6.1). These equations are removed from the full system of equations and are regarded as inputs to the system. The consideration of the variables as inputs inadvertently requires these variables to be sampled at every iteration.

The argument against this is that some stochastic variables may not require such frequent sampling and the simulation time could be shortened by only sampling when necessary. It is possible to make the stochastic variables differentiable by breaking it down to its linear parts (piecewise linear), but the uniform random variable will still need to be generated at every time step regardless. Therefore, making these variables differentiable is a futile attempt.

It was possible to sample only as needed when derivative information was not used to solve the system (see subsection 4.6.1 and section 4.7) but the solving speed was greatly increased when derivative information was incorporated to solve the system.

### 4.6.1 Model generation

The results from the lower triangular ordering need to be interpreted and a model of the ordered system needs to be constructed. This model must then be solved using some numerical root-finding software.

#### Interpreting the results

The results from the bordered lower triangular method (Baharev, 2017a) are two permutation matrices  $P$  and  $Q$  so that matrix multiplication  $PAQ$  results in the ordered incidence matrix  $A'$ . Equation 4.39 shows how  $A'$  is subdivided

$$A' = \begin{bmatrix} A'_{\hat{i},\hat{i}} & A'_{\hat{i},n} \\ A'_{m,\hat{i}} & A'_{m,m} \end{bmatrix} \quad (4.39)$$

where  $A'_{\hat{i},\hat{i}}$  is the lower triangular part of  $A'$ . The index  $\hat{i}$  is identified by simply searching through  $A'$  and finding the first column with an entry above the diagonal. The assignment variables can easily be obtained by identifying all the variable IDs that are on the diagonal of  $A'_{\hat{i},\hat{i}}$ . The order of these assignment variables is simply the order in which they appear on the diagonal (the column index). To determine the equation order is not as straightforward and a square  $m \times m$  auxiliary matrix  $R$  was created with a single entry in each row corresponding to the row number. This entry may be in any column. In  $R$ , the entry number corresponds to the equation row number in  $A$  (if the equation was added to  $A$  first it has row number 1 the second has row number 2 etc.). By remembering the order in which the equations were added to  $A$  the equation order can now be obtained by the matrix multiplication of  $PRQ = R'$ . The same as in  $R$ ,  $R'$  also only has one entry per row but the number is not sequential as in  $R$ . The order the values appear in  $R'$  is the equation order. The value in row 1 corresponds to the equation number that needs to be solved first, and the value in row 2 corresponds to the equation that needs to be solved second etc. Now that the variable and equation order is known the ordered system of equations is available and the model can be built.

When the identification of the feasible assignments in safe-eliminations (Baharev, 2017b) was done, it solved each equation for all its variables. This process is very time-consuming when ordering to  $A'$ . Although it is time-consuming, it means that all possible eliminations were already calculated and can be used when building the model. To save some processing time in the model generation part, all the results from safe-eliminations were saved and the correct symbolically solved equation can be matched to corresponding assignment saving time by not redoing the same calculation.

The variables and equations that are not in  $A'_{\hat{i},\hat{i}}$  are the variables that need numerical solving and the equations are the residuals that need their roots determined.

## Building the model

The entire model is built in CasADi (Andersson, 2013). The need for an algorithmic differentiation tool arose when `allocate_opt()` (section 4.5.1) was introduced to Amos because optimisation without derivative information did not succeed. CasADi was also implemented to find the roots of the system of equations with great success (see section 4.7). Depending on the presence of `allocate_opt()` two different models will be built. These two methods are discussed below.

### Absence of `allocate_opt()`

When `allocate_opt()` is not used in the operating instructions no optimisation is required and the system of equations simply needs to be solved. A CasADi function is built which takes the variables that need to be solved numerically as its inputs together with any additional inputs and parameters to the system and returns the residual equations. To show how to solve a system of equations with CasADi Equation 2.18 was ordered to the border lower triangular form (equation  $f_1$  was modified to  $f_1(\mathbf{x}) = x_1 + x_4 - 10 - a$  introducing a parameter  $a$ ) which resulted in  $A'_{ii}$  having the variable order  $x_1, x_2, x_3$  and equation order  $f_1, f_3, f_2$  when all variables are bounded to between 10 and 100 ( $10 \leq x_i \leq 100$  for  $i = 1, \dots, 5$ ). Variables  $x_4$  and  $x_5$  need to be solved numerically with residual equations  $f_4$  and  $f_5$ . Listing 4.6 shows how to use CasADi to solve the  $5 \times 5$  system using CasADi's syntax

```
1 from casadi import SX, Function, rootfinder, vertcat
2
3 # define the CasADi variables
4 x4 = SX.sym("x4")
5 x5 = SX.sym("x5")
6 a = SX.sym("a")
7
8 # equations in  $A'_{ii}$ 
9 x1 = a - x4 + 10
10 x2 = 3.39804717687081*(1/(x1*(x4 - 5)))*0.588235294117647
11 x3 = (x5 + 6)/(x2**2*x4)
12
13 # residual equations
14 r1 = x4 - 3*x1 + 6
15 r2 = x1*x3 - x5 + 6
16
17 # analytical model
18 model = Function('model', [vertcat(x4, x5), vertcat(a)], [vertcat(r1, r2)])
19
20 # create rootfinding object using the newton method
21 newton = rootfinder('newton', 'newton', model)
22
```

```

23 # find roots
24 res = newton([guess_x4, guess_x5], [parameter_a])

```

**Listing 4.6:** Solving of Equation 2.18 when order to bordered lower triangular form (equation  $f_1$  was modified to  $f_1(\mathbf{x}) = x_1 + x_4 - 10 - a$ ).

### Presence of `allocate_opt()`

When `allocate_opt()` is present the system is not solved using root-finding methods, instead the objective function is minimised together with equality and inequality constraints. All the variables identified in the bordered lower triangular ordering that need to be solved numerically are now considered as optimisation variables and the residual equations are now equality constraints to the optimisation.

The procedure is very similar to the case when `allocate_opt()` is absent but instead the model takes only the optimisation variables (not any other parameters and inputs) and outputs the constraints equations. Any other equality or inequality constraints are also added to the model function output. The additional parameters and inputs to the system of equations are added when the optimisation object is defined.

To illustrate how the optimisation works Listing 4.7 is used as an example. The equations in Listing 4.7 is the same as Listing 4.6 except equation  $f_4$  was removed to obtain  $\text{DOF} = 1$  and adding an objective function  $f = -x_4$  (maximise  $x_4$ ). In Listing 4.7 the remaining residual equation  $r_2$  is now an equality constraint. The other covariants  $g_1$  and  $g_2$  are arbitrary inequality constraints. In CasADi the equality and inequality equations are handled in exactly the same manner but the defining of the upper and lower bound on these equations determines if it is an equality or an inequality constraint. In Listing 4.7  $r_1$  is an equality constraint  $r_1 = 0$  (upper and lower bound is 0),  $g_1$  and  $g_2$  are inequality constraint with  $g_1$  constrained between 50 and a 1000 ( $50 \leq g_1 \leq 1000$ ) and  $g_2$  constrained between 10 and a 100 ( $10 \leq g_2 \leq 100$ ). The optimisation variables, parameters and inputs, constraint equations and the objective function are added to the optimisation object via a Python dictionary with the keys "x" the optimisation variables, "p" parameters and inputs, "g" constraint equations and "f" the objective function.

```

1 from casadi import SX, Function, rootfinder, vertcat, nlpsol
2
3 # define the CasADi variables
4 x4 = SX.sym("x4")
5 x5 = SX.sym("x5")
6 a = SX.sym("a")
7
8 # equations in A'
9 x1 = a - x4 + 10
10 x2 = 3.39804717687081*(1/(x1*(x4 - 5)))*0.588235294117647
11 x3 = (x5 + 6)/(x2**2*x4)

```



```

12
13 # equality equations
14 r2 = x1*x3 - x5 + 6
15
16 # objective function (minimise)
17 f = -x4
18
19 # inequality constraints
20 g1 = x1/x5
21 g2 = x4
22
23 # analytical model
24 model = Function('model', [vertcat(x4, x5)], [vertcat(r2, g1, g2)])
25
26 # create optimisation object using IPOPT
27 nlp = {'x': [vertcat(x4, x5)], 'p': vertcat(a), 'g': model(vertcat(x4, x5)),
        'f': f}
28 opt = nlpsol('nlpsol', 'ipopt', nlp)
29
30 # define bounds
31 lb = [0, 50, 10]
32 ub = [0, 1000, 100]
33
34 res = opt(x0=[guess_x4, guess_x5],
35 lb=lb,
36 ub=ub)

```

**Listing 4.7:** Optimising Listing 4.6 by removing  $f_4$ .

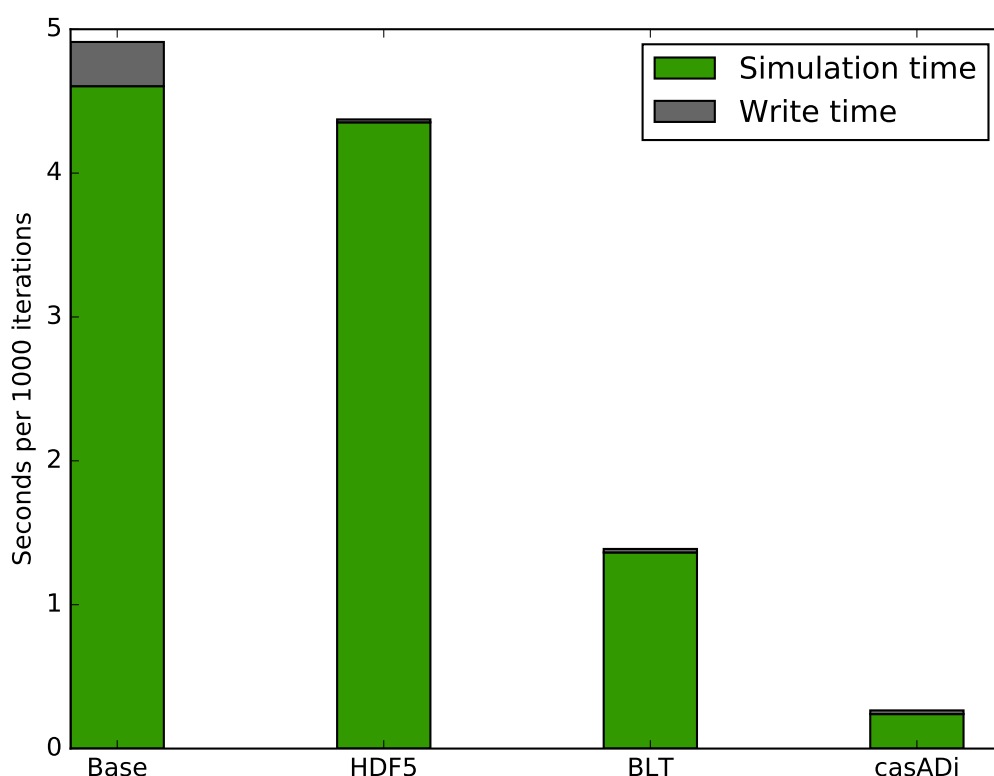
The branch and bound method as described in section 4.5.1 is not shown in Listing 4.7. Listing 4.7 only shows how an optimisation problem is set up in CasADi, but the optimisation problem in Listing 4.7 corresponds to the “NLP Equation 4.36” block in Figure 4.11.

## 4.7 Simulation

In the simulation part of Amoss, the model built in subsection 4.6.1 is executed for every active scenario for the number of replications required and the results written to a file. In this section, the focus will be on the improvements made to the simulation speed by the implementation of different techniques together with factors that influence the simulation speed.

The implementation of the HDF5 (The HDF Group, 2017) file format, bordered lower triangular tearing and CasADi each had a fundamental influence on the simulation time. Figure 4.14 shows how each of these improved the combined simulation time. These

improvements are cumulative meaning that every new improvement is implemented on top of the previous. The base simulation time was recorded when block lower triangular tearing was used instead of the bordered lower triangular tearing and a CSV file was used to save the results. All the tests were run for 1000 iterations and 250 replications. Therefore, the results in Figure 4.14 is an average of 250 data points. The computations were carried out with the following hardware and software configuration. Processor: Intel® Core™ i7-3770 CPU @ 3.40GHz, disk: Samsung SSD Evo 850, operating system: Windows 7 Ultimate © 2009 Microsoft Corporation and Python 3.5.2. The complete defined system that was used is found in section A.2.



**Figure 4.14:** Combined simulation and result write (to disk) time and how it improved by implementing the method on the x-axis.

**HDF5** The HDF5 file format was introduced to increase the write time to disk. It was found that speed-up achieved by implementing the HDF5 file format was purely due to the size reduction. On average an HDF5 file is 15 times smaller than a CSV file and directly responsible for the faster write time.

**BLT** Block lower triangular ordering was used to increase the simulation speed. The simulation speed-up seen by implementing the bordered lower triangular tearing is due to a reduction in the number of variables that need to be solved numerically.

For this particular simulation (see section A.2) the reduction was from 57 to 7 variables. The solver `scipy.optimize.fsolve` with no derivative information supplied was used for both cases.

**CasADi** The implementation of CasADi shows the power of derivative information. CasADi's core is also written in C++ (Andersson, 2013: 72) which is a language with a fast execution time.

## Parallel processing

The simulations Amoss simulates are embarrassingly parallel and the scenarios can be simulated completely independently from one another. In Amoss, scenarios are run in parallel instead of the replications, because the replications of a single scenario depend on the last values of the previous replication as its initial values; therefore, the replications are not independent of one another and not embarrassingly parallel. So it is not possible to parallelise the replications without using other parallel processing techniques. Sasol specifically requested that replication need to follow one another in this manner, but should this constraint be removed and replications are allowed to have their own independent initial values it is possible to run replications in parallel as well.

Celery is used to parallelise the scenarios.

## Linear scaling

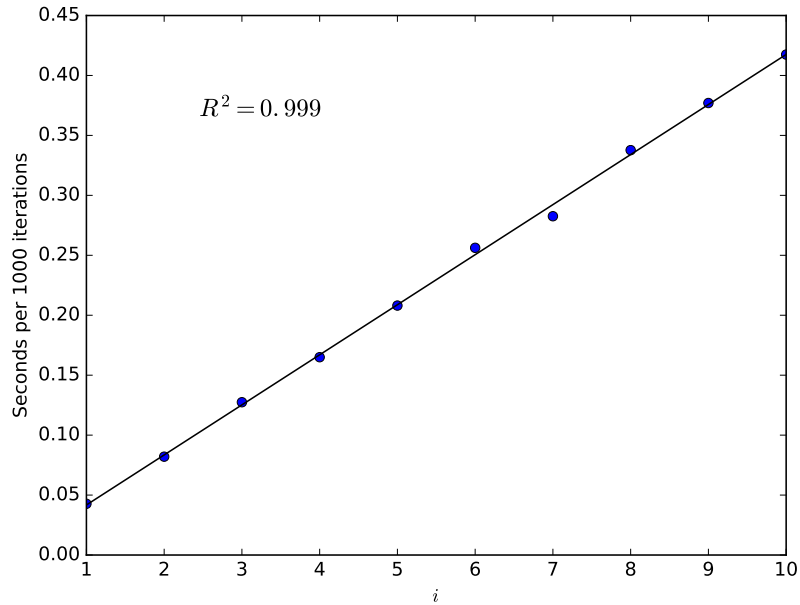
Amoss was tested to see if it does scale linearly with the number of equations. The following set of equations

$$x_i = x_{i-1} + 0.001x_{i-2} + 1.5 \quad (4.40)$$

with  $x_0 = 1$  and  $x_{-1} = 1$  was created for  $i = \{100, 200, 300, 400, 500, 600, 700, 800, 900, 1000\}$  and simulated for 1000 iterations and 250 replications. The results of Equation 4.40 were plotted in Figure 4.15.

Figure 4.15 show that the simulation time increase linearly as the number of equations increase which was expected, but unfortunately this linearity is dependent on the solving difficulty of the system. It was observed through a variety of test processes that different systems of equations will solve faster than others even if the number of equations are roughly the same. Based on observation it seemed that this discrepancy was due to the number of equations that are required to be solved numerically. It was hypothesised that and increase in variables requiring numerical solving will increase simulation time. To test this hypothesis, the following set of equations were used

$$\begin{aligned} x_i &= x_{i-1} + 0.001x_{i+1} + 1.5 & i &= 1 : r + 1 \\ x_i &= x_{i-1} + 0.001x_{i-1} + 1.5 & i &= r + 2 : n \end{aligned} \quad (4.41)$$

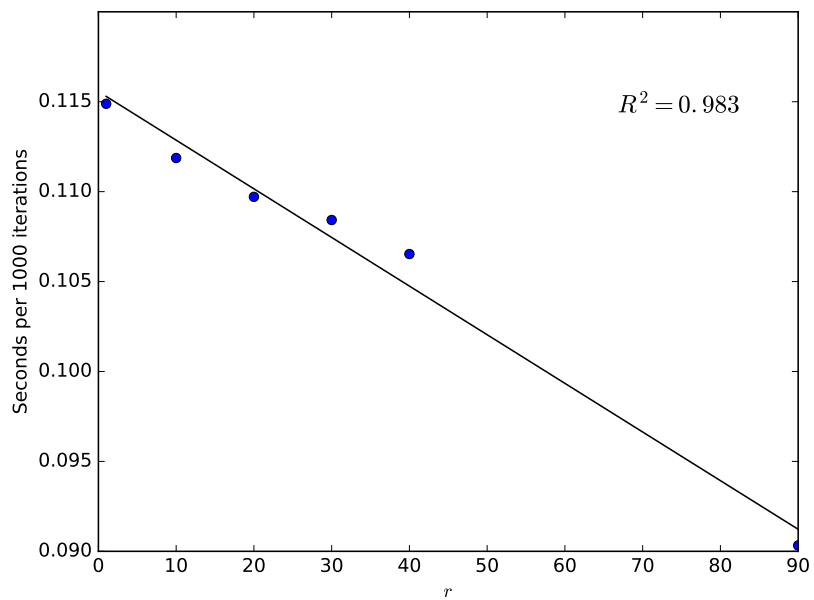


**Figure 4.15:** Simulation time in Amoss using the system of equations in Equation 4.40 for different  $i$  together with a linear regression fit.

with  $x_0 = 1$  and  $r$  controls the number of equations that will need numerical solving and  $n$  is the total number of equations that will be created.

Using  $r = \{1, 10, 20, 30, 40, 90\}$  did not produce the expected result. Instead of seeing an increase in simulation time a slight decrease was observed. Figure 4.16 shows this phenomenon. Due to this result, it is now hypothesised that the difficulty to solve a system of equations is dependent on the factors described in subsection 2.5.6. The sensitivity of the guessed variables to the residual equations can lead to more iterations in the multidimensional Newton method leading to longer simulation time. The decreases observed in Figure 4.16 could be because a majority of the floating-point operations are executed in C++ as opposed to the Python interpreter.

The result of Equation 4.41 shows that it is difficult to predict the simulation time and that some models may simulate a faster than others. Therefore, no estimate can be given on the simulation speed given the number of equations or guessed variables. The only way to predict the simulation time is to quantify the difficulty of the system of equations.



**Figure 4.16:** Simulation time in Amoss using the system of equations in Equation 4.41 for different  $r$  together with a linear regression fit.

---

---

# CHAPTER 5

---

## USER INTERFACE (UI)

The UI is a combination of the `simulation_description.xlsx` file (the documentation of how to populate this file is in section 4.1), the `stochastic_distributions.xlsx` file (explained in section 4.5.1), an editing environment and a graphical user interface. The UI is the gateway for the user to interact with the code written for Amoss.

Currently Sasol supports Microsoft and every engineer has access to Excel. The use of `.xlsx` files are a simple way for the user to input structured and specific data into Amoss and because the user already has access to this software it was the chosen format.

The editing environment is a platform which enables the user to edit the `operating_instructions.txt` file (see section 4.5 for the content that appear in this file). The operating instructions file is `.txt` file which can be edited by any number of text editors. For a better user experience it was chosen to use the Atom text editor (GitHub Inc, 2017). Atom calls itself the “hackable text editor” and it is utilised to create a more interactive editing environment.

The graphical user interface was created to assist the user to execute all the tasks in section 4.2 to section 4.7. The GUI is also used to create new models and change simulation settings.

### 5.1 Editing environment

When creating a model from the ground up, the modeller has a natural feeling of the available variables and what it represents, but because Amoss automatically creates the bulk of the equations and variables, there is a disconnect between the modeller and the model. To bridge this disconnect, it was necessary to create a more interactive editing environment which provides feedback to the modeller on the available variables. This feedback mechanism saves time and eliminates unnecessary errors by identifying spelling

mistakes or undefined variables. To create the desired environment, four key features were identified: the ability to add comments, tab completion of the created variables, syntax highlighting and general error detection (linting) when writing the equations.

The operating instructions file is not incorporated in Amoss as pure code but is parsed (as discussed in subsection 4.5.2) and interpreted as an additional set of equations.

The Atom text editor can fulfil all the mentioned requirements to make a functional and usable environment. With Atom being a community-based project a large variety of packages are available to create the desired environment. A large portion of the requirements was solved by simply using Python grammar.

By simply installing Python grammar and linting packages in Atom, comments can be added without it being parsed and syntax highlighting and linting can be achieved. The tab completion was achieved by defining all the created variables in a separate Python file and importing it into the operating instruction environment using Python syntax and the installed Atom package. These packages will recognise the file and make these variables available. The possibility still exists that a created variable can be misspelled, an undefined variable is used or Python grammar syntax errors can be made. The linter will see the operating instructions as a Python script and will point out any syntax errors or misspelt and undefined variables.

Even though the environment forces a user to define a variable before it is used in other equations, the order that equations appear in the operating instructions file carries no meaning. It is technically still valid to use a variable and define it later in the operating instructions, but the linter will see this as an error. This error is by no means an issue that should be fixed because it will ensure that the user has defined all required variables. This behaviour of the editing environment can also create the illusion that code is executed by from top to bottom tempting a user to re-purpose or update a variable (defining a variable more than once in the operating instructions file). See the rules for the operating instruction environment for more detail.

Figure 5.1 to Figure 5.4 is a summary of all the necessary features that are required to make a user-friendly editing environment.

Due to heavy use of Python syntax and grammar the operating instruction environment may look “too familiar” to an experienced Python programmer which may lead to the feeling or urge to view the operating instructions as a Python script. The operating instructions *must* be viewed as additional equations and not as code and the use of Python objects (lists, dictionaries etc.), libraries and loops (while and for loops) are not allowed.

Below is a list consisting of the rules for using the operating instruction environment:

- Only valid Python variables are allowed.
- Any variable starting with “Amoss\_”, “allocate{ }” or “if{ }” (where { } is a number)

```

3   # the ability to add comments
4   # a is the gravitational acceleration on earth
5   a = 10 # m/s^2

```

**Figure 5.1:** Adding comments to equations.

```

3   S1
4   v S1_comp1 1
5   v S1_comp2 1

```

**Figure 5.2:** Tab completion of created variables.

```

3   if Buffer_level_total >= 50 or S5_total == 0:
4       S1_total = 0
5   else:
6       S1_total = max(S_source_total, 0)

```

**Figure 5.3:** Syntax highlighting.

```

3   ● test1 = S100_total
4   ● test2 = S1total

```

**Figure 5.4:** Error indication.

are forbidden from being created or used. These variables have inner meaning to Amoss and using these variables can lead to errors.

- Update or re-purpose<sup>1</sup> of variables is not allowed because the operating instructions must be seen as equations and when a variable is updated or re-purposed it creates different equations for the same variable which are conflicting and will lead to an ill-defined equation set.
- When using conditionals the variables defined in the if-statement must also be defined in the following elif- and else- statements. All variables that are not defined for all conditions will be zero when the condition evaluates true.
- When making use of the min or max function the equation may only contain the min or max call with no other additional mathematical operations allowed on the

---

<sup>1</sup>Re-purpose refers to using the same variable name but representing a new quantity

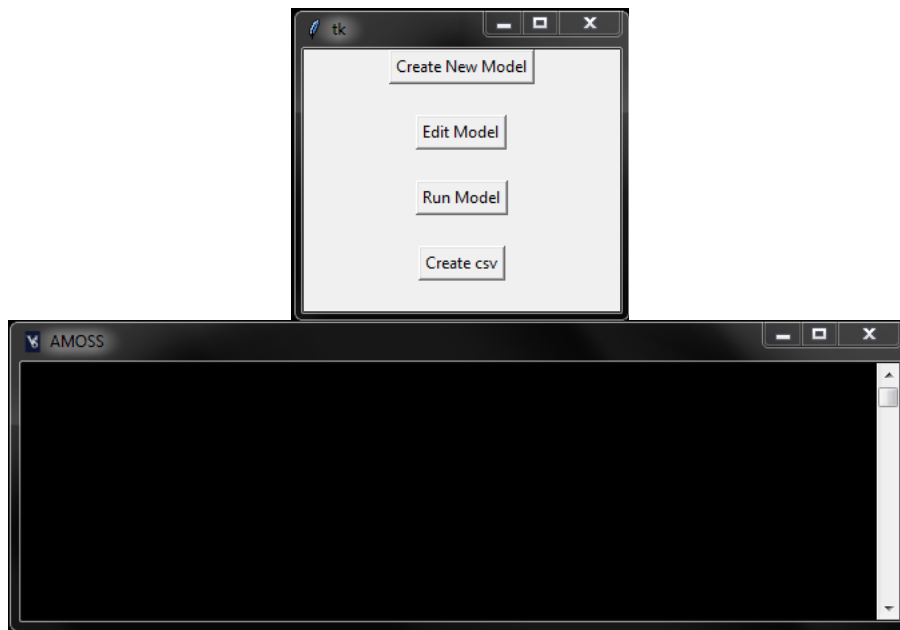


min or max function itself. Mathematical operations are allowed inside the call.

- The use of nested min or max functions is not allowed.
- Multiple variable assignment in a single line (see subsection 4.5.2).
- Mathematical functions or constants ( $\sin$ ,  $\log$ ,  $\pi$ ) are not supported and can not be used.

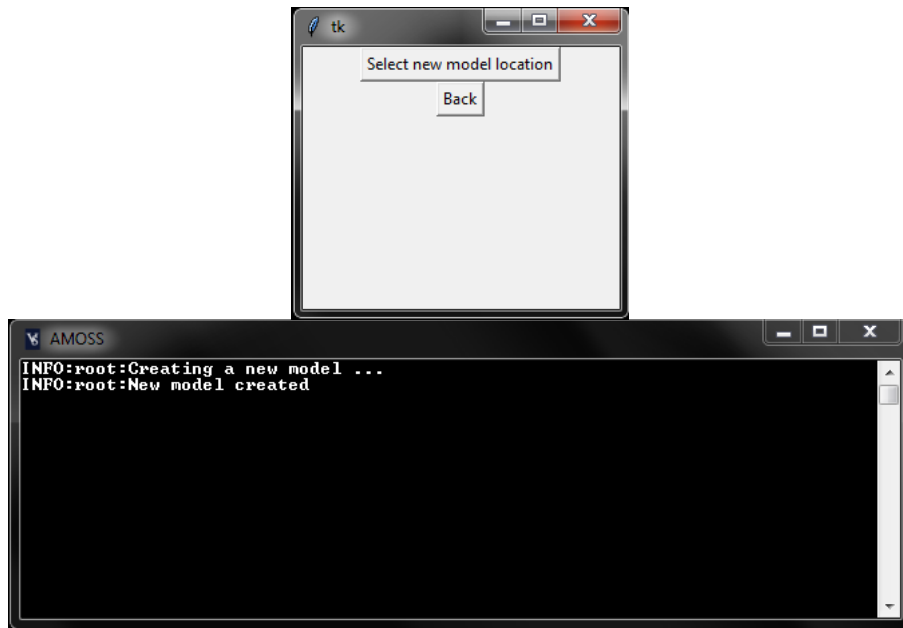
## 5.2 Graphical user interface (GUI)

The GUI consists out of two windows: a navigation and execution window together with a small info window which displays information related to the different tasks. The combination of these two windows are known as the GUI. When starting Amoss the initial windows that are displayed are shown in Figure 5.5.



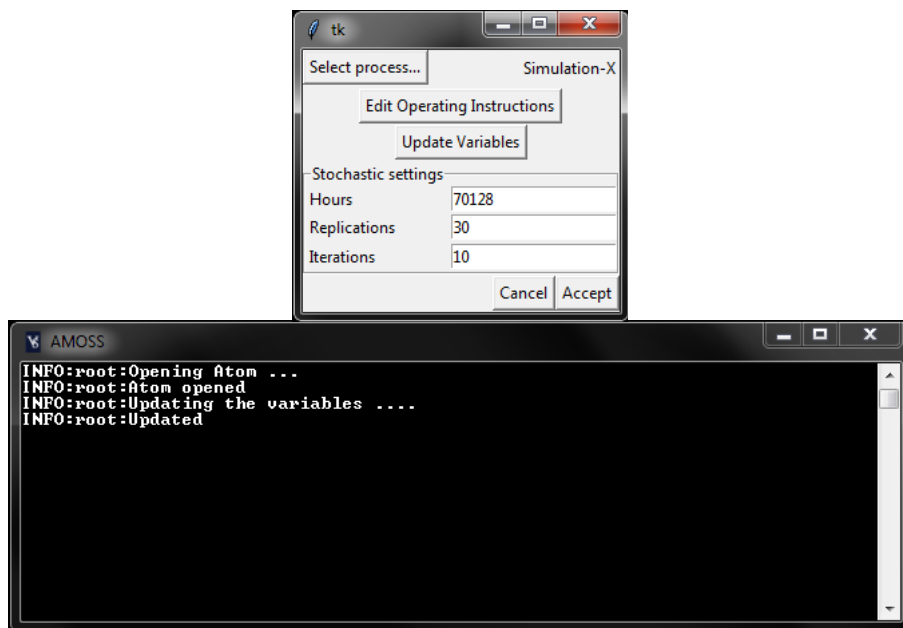
**Figure 5.5:** The user interface that appears when Amoss is started. Top the UI and bottom the info window.

The first button on the GUI (Create New Model) Figure 5.6 is used to create a new blank process in the folder of choice containing all the necessary files to create a full model. These files include: *process.mo* the OpenModelica file where the process diagram is drawn in, *simulation\_description.xlsx* the Excel file containing all the information to create equations, *operating\_instructions.txt* file containing the operating instructions and *stochastic\_distributions.xlsx* the file containing the user-defined distributions. The info window will display “Creating new model” while the new location and its files are created and “New model created” when the process is complete.



**Figure 5.6:** The user interface that appears when a new model is created.

The “Edit Model” (Figure 5.5) button is used to edit the simulation information and the operating instructions. To change the process that needs to be edited the “Select process” button can be used to select another process model. The name of the selected process model name (Simulation-X in this example) is displayed on the right. The next button “Edit Operating Instructions” will open `operating_instructions.txt` with the Atom editor (if installed). The info in the info window will read “Opening Atom” when Atom is opening and “Atom opened” when the Atom editor process is started.



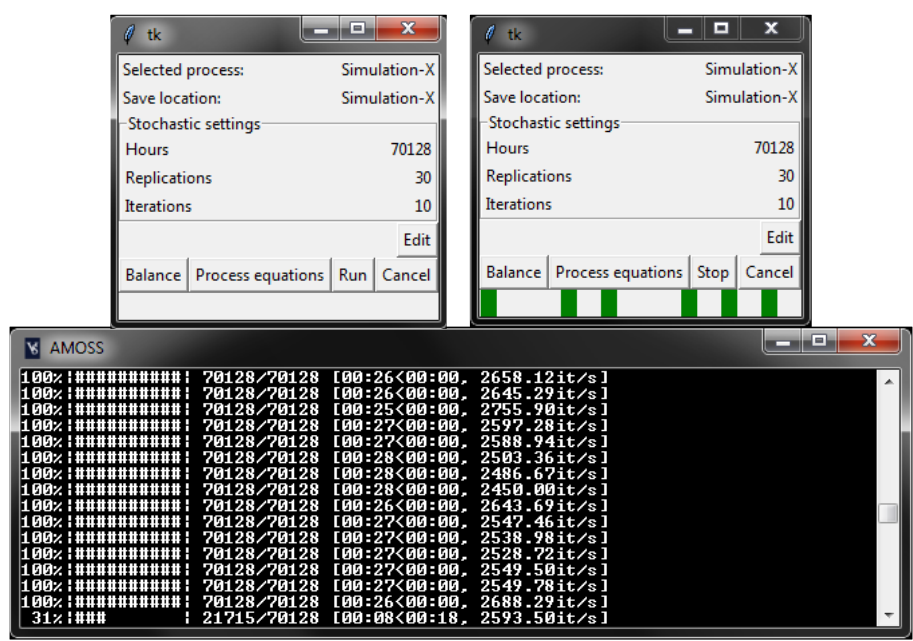
**Figure 5.7:** User interface to edit simulation information and edit the operating instructions.

“Update Variables” will gather all the variables that were automatically created to-

gether with additional variables created in the scenario table and make them available in the Atom environment where the operating instructions are edited. All these variables will appear in the Atom environment as variables that have already been created with tab completion.

The remaining settings are related to the simulation itself. “Hours” indicates the time span of the simulation and “Replications” the number of times a scenario is repeated. “Iterations” is the maximum number of function evaluations allowed in the root-finder (currently not implemented for the CasADi version). As a default 70 128 hours (8 years), 30 replications and a maximum of 10 function evaluation are shown. These values can be changed by entering new values with the keyboard. The “Accept” button will save any changes to the number of Hours, Replication and Iterations and “Cancel” will display the UI in Figure 5.5.

The “Run Model” button (Figure 5.5) will show the UI in Figure 5.8. At the top, the UI displays the model which will be executed as well as the run settings (Hours, Replications and Iterations). To change any of these the “Edit” button will lead to the UI in Figure 5.7 where it can be changed. The “Balancer” button will execute the code that automatically creates the equations (section 4.2 to section 4.4). The “Process equations” button will execute the code that orders the automatically created equations, and the equations in operating instructions to the bordered lower triangular form in section 4.6. The “Balancer” and “Process equations” were separated to make it possible to get access to the automatically created variables in the operating instructions before the equations are ordered.



**Figure 5.8:** User interface to execute different parts of Amoss. The UI top left is when a simulation is not being run, and the UI top right shows the UI when a simulation is being executed.

In the UI (top left) in Figure 5.8 the “Run” button will start the execution of the model (section 4.7). When the button is pressed the UI (top right) will appear, and the “Run” button will change to a “Stop” button to stop the simulation if required. The UI also has a progress bar to show the progress of the simulation. Due to the simulations being run in parallel the progress bar is divided into parts equal to the number of scenarios. Therefore, each segment represents the progress of a single scenario. The info window displays the progress of each replication.

The “Cancel” button will show the UI in Figure 5.5.

---

---

# CHAPTER 6

---

## CONCLUSION AND RECOMMENDATIONS

Amoss successfully provides a platform to create and simulate stochastic simulations. It encompasses a variety of different aspects from automatic equation generation, equation ordering, optimisation and parallel processing to provide this platform. The success of Amoss can be measured by the listed deliverables in section 1.4. The deliverables that are satisfied by Amoss are:

**Reduction in development time.** This is one of the largest improvements Amoss offers. The ability to generate equations automatically given a process diagram together with the equation ordering drastically reduced the development time. In MOSS equations are manually derived for systems having feedbacks, solved symbolically to avoid numerical root-finding software and ordered intuitively in an attempt to reduce the simulation time. Creating simulations in this manner could take upwards of a month whereas a similar simulation can be completed in about a week using Amoss.

**Generic application.** Amoss was designed from the ground up as a general stochastic simulation platform. This is evident in the use of a process diagram and the operating instructions file to create a model of a process with any configuration provided that the operational unit is defined in Amoss. If the operational unit is not defined in Amoss it can be added to the automatic equation generation code and because of the equation orientated approach does not require any additional changes to the downstream code.

**Development flexibility.** Changing the operating instructions or the physical plant in MOSS for an existing model poses a huge problem. In MOSS any change requires

additional equations to be derived and can lead to the symbolically solved equations to be invalid. Depending on the severity of the change it could setback the developer a week to a month. Changing an existing model in Amoss simply requires the modification of the process diagram (and the affected input tables) and the operating instructions with minimal effort and time.

**Simulation flexibility.** In Amoss, sections of the plant can be activated and deactivated by simply adding an if-block in the operating instructions which will set the inputs to the plant section to zero (deactivate) or allocating a value to the inputs (activate). This on-off capability is possible due to Amoss's equation orientated design and the feasible assignments by Baharev (2017a) which will avoid zero division errors.

**Acceptable accuracy.** On condition that the Newton root finder successfully finds a solution under 1000 function evaluations any residual equation will be solved with an absolute error of  $1^{-12}$  or if a single absolute Newton step is less than  $1^{-12}$ . When the Newton root finder succeeds the equations will be satisfied at each Euler step, but this does not guarantee that the integration is within acceptable accuracy. The accuracy of the integration is dependent on the dynamic behaviour of the Buffer tanks. A Buffer with a small time constant will not integrate with acceptable accuracy but a Buffer with a large time constant will.

**Fit for purpose.** Amoss is an extension of the MOSS methodology and was developed with the guidance of Sasol and follows the modelling methodology of Sasol. Amoss is written in Python which is a common language with a large community making it possible for a person with moderate programming experience to contribute to Amoss.

**Linear scalability.** Linear scalability as initially defined by Sasol does not take the model complexity into consideration. The difficulty of the system depends on the complexity of the process and the size of the feasible assignments. A process with a high number of feedback streams or equations with possible zero divisions can lead to a more difficult system to solve independent of the number of equations of the process. It is also uncommon to find problems that scale linearly. Even solving a set of linear equations has a worst-case time of  $O(n^2)$  and it is expected that the Amoss with its non-linear equations will at least scale worse. However, Figure 4.15 shows that simulation time can scale linearly with the number of equations when the difficulty stays constant.

**Quick learning curve.** The learning curve is low to moderate. The most difficult part is to learn basic Python grammar and how to create an OpenModelica flow sheet together with rudimentary coding skills. The other skills required to create a simulation is editing a text file and spreadsheets, as well as modelling knowledge. A

modeller with no prior knowledge of Python grammar or OpenModelica was able to create small process models within a week.

**IM Cost.** The IM cost for Amoss is low relative to other commercial software due to the use of open source software like Python and OpenModelica. The University of Pretoria has also agreed to continue development beyond the delivery date at the end of 2017.

**Version control of model development.** Version control of the project is done with git (Software Freedom Conservancy, 2017) using Bitbucket (Atlassian, 2017) as the cloud repository. It is also possible use git and a repository to put any created models under version control, because all the information required to define a model is text based (even the .mo file).

**Debug capability.** Rudimentary debugging is added in the form of the information window in the GUI. Common errors a modeller would make has been identified and will display an appropriate message and a recommended course of action. An example would be when the DOF is not zero a message will inform the modeller and urge them to add or remove inputs in the inputs list.

**Cause identification.** Rudimentary cause identification is added by identifying when an if-, elif- or else statements of any if-block evaluated true. This identification is a direct consequence of the way if-blocks are parsed and converted to equations. The statements in an if-block are parsed to equations using a base if-variable and evaluate to 1 when true and 0 when false. The if-variable equal to 1 indicates that that particular statement was triggered. A list linking the created if-variables to the statement is made available to the modeller.

## 6.1 Recommendations

This section will discuss some recommendations to improve Amoss. These recommendations are divided into two parts. Near future and future recommendations. The near future recommendations are suggestions that will improve Amoss in the short term whereas the future recommendations will discuss a possible expansion of Amoss.

### 6.1.1 Near future recommendations

Two of the deliverables of Amoss listed in section 1.4 could not be met. These are **fast simulation time** and **software package stability**.

Even though improvements were made in the simulation speed, it is not up to the required standard. When comparing a model built in Amoss against the same model using

the MOSS methodology in VBA the VBA model speed is superior to that of Amoss. The benchmark process (which can not be disclosed here) takes 1.7 minutes to complete a replication of 70 128 hours whereas Amoss simulates the same process in 24 minutes.

This problem is due to the high number of variables (100) that need to be solved numerically as well as an increase in difficulty of the system. A high number of these variables (84) stems from infeasible assignments. When feasible assignments are not taken into account when ordering to the bordered lower triangular form only 14 variables need be solved numerically. The single largest source of these infeasible assignments stems from the Mix Point equations (subsection 4.3.3). The example in Equation 4.12 shows the sources of the possible zero division. The benchmark process recorded a total of 14 Mix Points and Buffers resulting in 92 equations with possible zero division. Therefore, a speed increase can be achieved by reducing the infeasible assignments. Some ways of reducing the infeasible assignments are listed below:

- A crude way to avoid zero division is to add a small constant to the denominator, eg.  $\frac{b}{a+\epsilon}$  where  $\epsilon$  is the small constant. By adding  $\epsilon$  will make all denominators larger than zero only if  $a$  falls within the range  $[0, \infty)$ . This condition is true for all models that do not allow reverse flow. It is important that  $b \rightarrow 0$  when  $a \rightarrow 0$  otherwise the fraction will result in a very large number. Implementing this will lead to a loss of accuracy.
- For the given simulation the bounds for the variables were not accurately given when the feasible assignments were identified. In a simulation consisting of 633 variables assigning proper bounds to all variables will be a tremendous task. A more targeted approach is recommended by identifying the variables that could lead to zero division and requesting accurate bound for these variables only.
- Using the spiked Hessenberg form to identify smaller subsystems of the bordered lower triangular form to solve. This method will not directly reduce the number of infeasible assignments but rather solve smaller subsystems containing a small number of variables that need to be solved numerically.

Improving the stability of Amoss is the other deliverable that needs more attention. Amoss is stable in the sense that it does not crash (close unexpectedly), but the GUI is considered as unstable. More specifically the “Edit Model” section (Figure 5.7). The stochastic setting (Hours, Replications and Iterations) does not save properly. When these settings have been changed the settings that will be used in the simulation is the old settings even though the newly changed number will appear on the screen. Only when the GUI is closed, do these changes take effect and will be changed once a new session is started.



The remaining recommendations are features and checks that will increase the overall usability of Amoss and are listed below:

- Check if all the inputs correspond to a valid created variable. Currently, the inputs are only counted, and the use of invalid variables can lead to a  $DOF = 0$  solution even if one of the input variables does not feature in the model.
- When the DOF analysis is performed in the automatic equation generation (section 4.3) and the  $DOF \neq 0$  it could be useful to assist the user in removing variables when  $DOF < 0$  or identifying variables to add when  $DOF > 0$ . The equation block orientated way to approach this would be to check if the guidelines are followed and identify any input that may be wrong. The equation orientated approach would be to use the maximal matching of the bipartite representation of the incidence matrix to identify these inputs.
- Check that the rows in the reactor table add up to 1 (see subsection 4.3.4).
- Check that the right number of streams are specified in the separation table (see subsection 4.3.5).
- Check that all the input variables are assigned to a value in the scenario table or an equation in the operating instructions file.
- Inform the user if the piecewise integration of a discrete probability mass function is not exactly 1. It is expected that the piecewise integration of a continuous probability density function will not be exactly 1, but when some tolerance value is exceeded there might be an error and the user should be informed. (see section 4.5.1)
- The ability to add multiple variable assignments in a single line (see section 5.1).
- Inform the user when the number of assignment variables and any of the arguments of `allocate()` does not match (see section 4.5.1).
- Inform the user when a minimum constraint is higher than a maximum constraint when using `allocate()` (see section 4.5.1).
- The ability to define optimisation problems in the operating instructions file.
- The ability to use for-loops in the operating instructions file. For-loops could make the operating instructions file cleaner by reducing the number of written equations. When parsing the for-loop, each loop will create additional equations. Rules and restrictions will need to be set in place when using for-loops in Amoss but CasADi can handle the creation of equations via a for-loop.

- Currently a Celery server is started when the “Run” button in the GUI (Figure 5.8) is pressed which produces some overhead and clutters the info window. It will be better to start the Celery server when a new session of Amoss is started.
- Place created models under version control and expand the GUI to facilitate with git.

### 6.1.2 Future

If Amoss becomes a common simulation package in Sasol, it could be useful to link up two or more different models into a single large model. Linking of models can be achieved by adding the equations of the different model together and mapping the sinks that correlate to sources in other simulations.

Currently, the MOSS methodology and, subsequently, Amoss are used extensively to answer what-if questions given a set of operating instructions. A key variable (buffer size, source feed rate etc.) is changed between scenarios and the effect on profit is investigated under the current operating philosophy specified in the operating instructions. It may be interesting to use Amoss in reverse to determine the best operating philosophy given the current process set-up or determining the best operating philosophy after a change to the process. Implementing this capability will involve stochastic optimisation given some objective function.

---

---

# APPENDIX A

---

## TIME SIMULATION DATA AND INPUTS

### A.1 Simulation time frequency distributions

Figure A.1 - Figure A.4 shows the frequency plots of the improvements made in the total simulation time by implementing the HDF5 file format, bordered lower triangular ordering and CasADi.

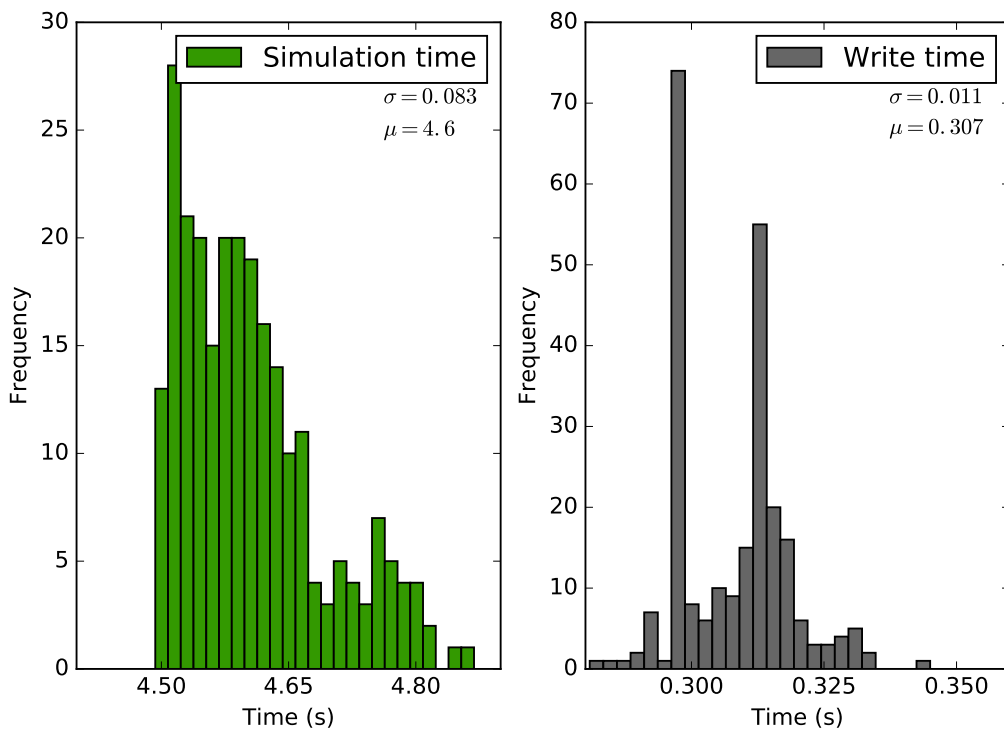
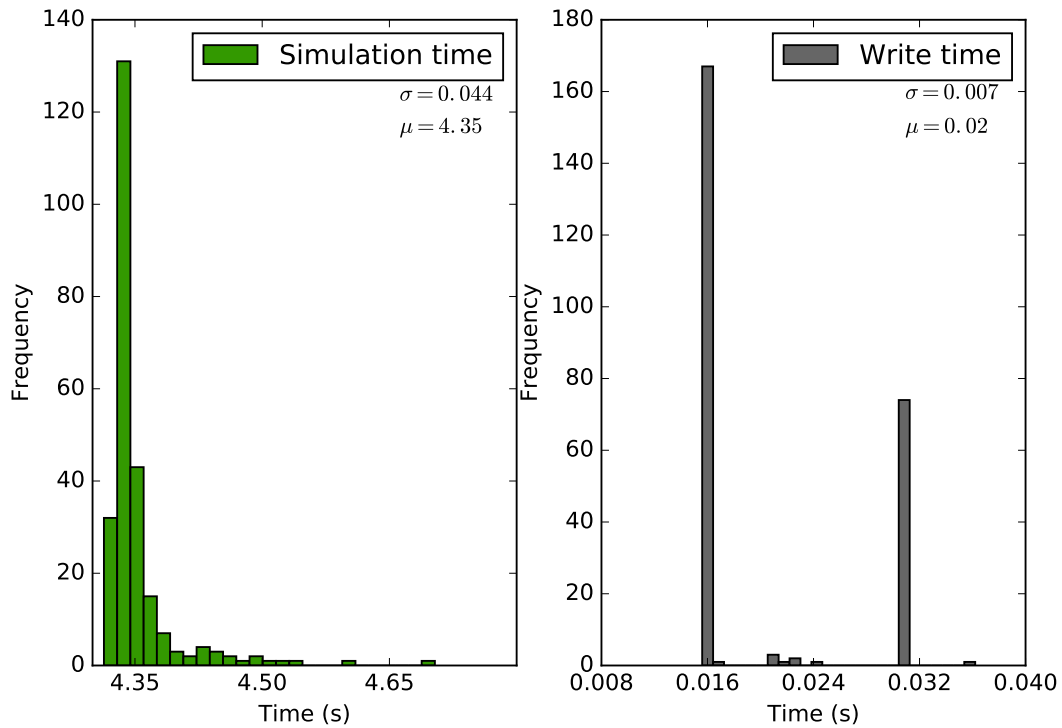


Figure A.1: Frequency plot for the base implementation.



**Figure A.2:** Frequency plot for the HFD5 implementation.

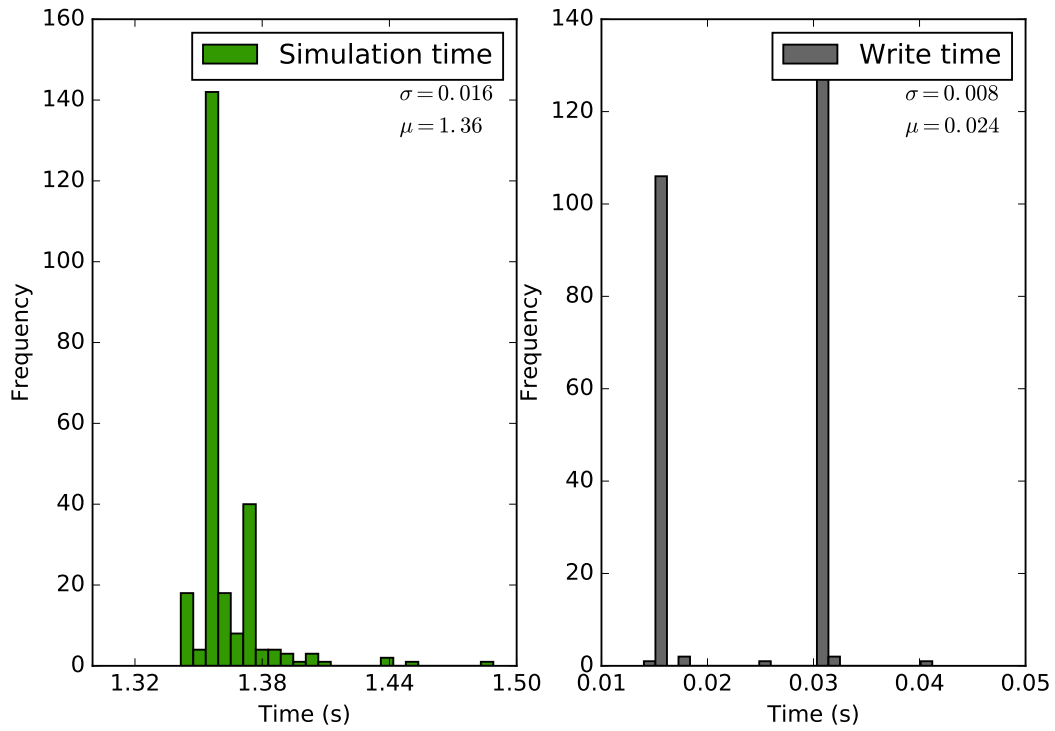
## A.2 Simulation time inputs

The user inputs for the simulation on which Figure 4.14 is based is given below.

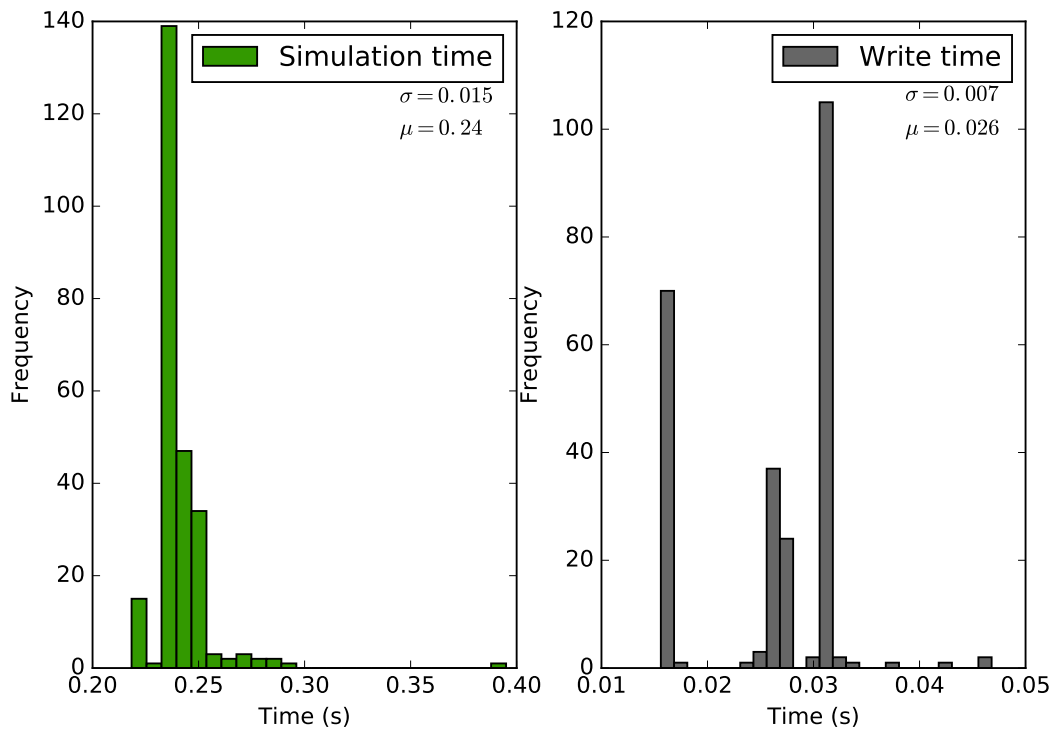
Figure A.5 is a very stiff system with a 4 recycle streams.

The probabilities in Table A.4 were intentionally wrongfully added so that the integral does not add to 1. This was done to test if the correct normalisation is applied.

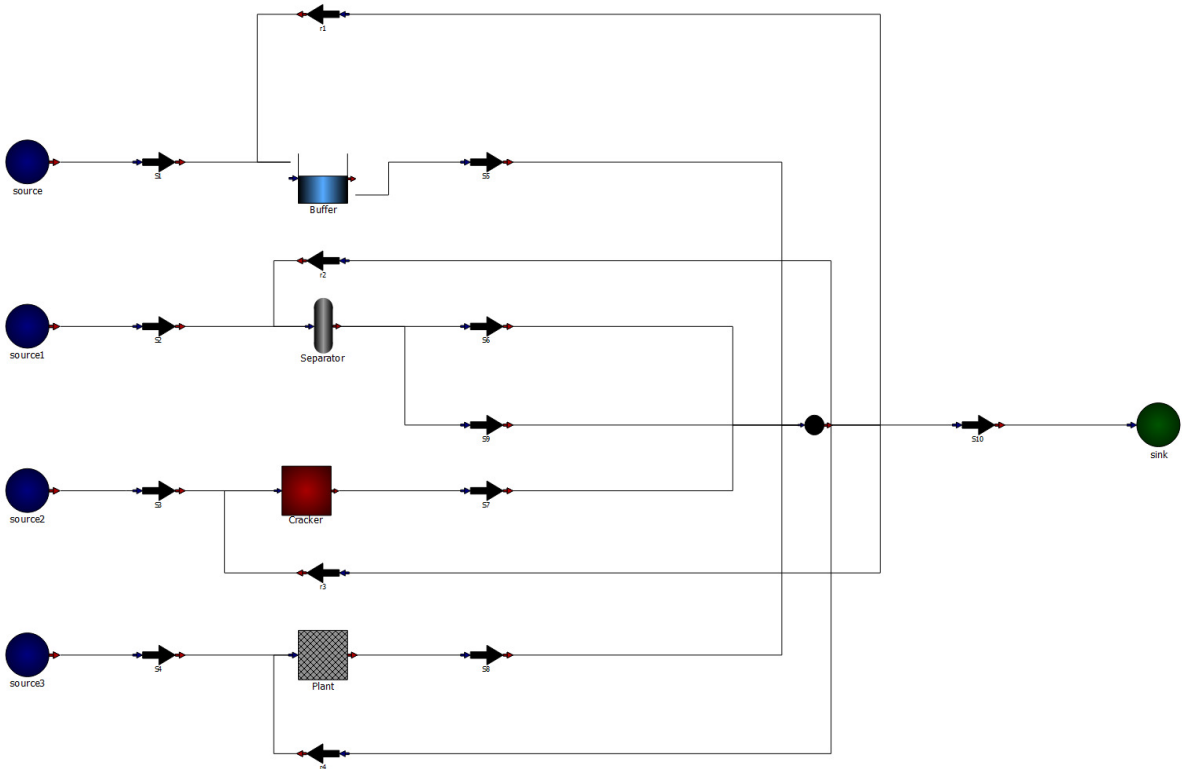
Listing A.1 is a relatively arbitrary set of operating instructions to test the total simulation time. It purposely does not include the `allocate()` or `allocate_opt()` functions because not all the versions of the code timed supports these functions.



**Figure A.3:** Frequency plot for the bordered lower triangular implementation.



**Figure A.4:** Frequency plot for the CasADi implementation.



**Figure A.5:** Process flow diagram of the process used to time the improvements in simulation time.

**Table A.1:** The component list (left) and the inputs list (right) of the Figure A.5.

Components	stream	comp
comp1	S1	comp1
comp2	S1	comp2
comp3	S1	comp3
comp4	S1	comp4
comp5	S1	comp5
	S2	comp1
	S2	comp2
	S2	comp3
	S2	comp4
	S2	comp5
	S3	comp1
	S3	comp2
	S3	comp3
	S3	comp4
	S3	comp5
	S4	comp1
	S4	comp2
	S4	comp3
	S4	comp4
	S4	comp5
	S5	total
	r1	total
	r2	total
	r3	total
	r4	total

**Table A.2:** Separation data table of Figure A.5.

node	attribute	comp1	comp2	comp3	comp4	comp5
Separator	S6	eff_c1	eff_c2	eff_c3	eff_c4	eff_c5

**Table A.3:** Reactor data table of Figure A.5.

node	comp	comp1 out	comp2 out	comp3 out	comp4 out	comp5 out
Cracker	comp1 in	c1.i.c1_o	0	0	0	0
Cracker	comp2 in	c2.i.c1_o	c2.i.c2_o	0	0	0
Cracker	comp3 in	c3.i.c1_o	c3.i.c2_o	c3.i.c3_o	0	0
Cracker	comp4 in	c4.i.c1_o	c4.i.c2_o	c4.i.c3_o	c4.i.c4_o	0
Cracker	comp5 in	c5.i.c1_o	c5.i.c2_o	c5.i.c3_o	c5.i.c4_o	c5.i.c5_o

**Table A.4:** Distribution table of Figure A.5. Left (random1) is a continuous- and right (random2) a discrete distribution.

Discrete   0		Discrete   1	
values	p	values	p
0	1	0	1
1	1	1	1
1	2	2	2
2	2	3	3
2	3	4	2
3	3	5	1
3	2		
4	2		
4	1		
5	1		

```

1
2 r4_total = 0.1
3
4 # calculate a remainder
5 remain1 = to_mix - r1_total - r2_total
6 if remain1 > 0:
7     r3_total = 0.1
8 else:
9     r3_total = 0.1
10
11
12 if r1_total == 0.1:
13     r2_total = 0.1
14 else:
15     r2_total = 0.1
16
17
18 # total flow to MixPoint
19 to_mix = (S5_total + S6_total + S7_total + S8_total + S9_total)
20
21 # if the level of the buffer lower than half the max slowly fill the tank
22 # else if it is higher dump to a level of 1
23
24 if Buffer_fullness_ratio <= 0.5:
25     r1_total = 0.1
26     S5_total = Buffer_level_total*0.9
27     test_stochastic2 = sample_dist(random1)
28 else:
29     S5_total = Buffer_level_total + S1_total - 1
30     r1_total = 0.1
31     test_stochastic2 = sample_dist(random2)
32
33 test_stochastic = sample_dist(norm, 10, 5)

```

**Listing A.1:** The operating instructions for the simulation in Figure A.5.



**Table A.5:** Scenario table for the simulation in Figure A.5.

scenario_nr	1
scenario_active	1
S1_comp1	10
S1_comp2	10
S1_comp3	10
S1_comp4	10
S1_comp5	10
S2_comp1	10
S2_comp2	10
S2_comp3	10
S2_comp4	10
S2_comp5	10
S3_comp1	10
S3_comp2	10
S3_comp3	10
S3_comp4	10
S3_comp5	10
S4_comp1	10
S4_comp2	10
S4_comp3	10
S4_comp4	10
S4_comp5	10
eff_c4	0.6
eff_c3	0.7
eff_c1	0.9
eff_c2	0.8
eff_c5	0.5
c1.i.c1.o	1
c2.i.c1.o	0.1
c2.i.c2.o	0.9
c3.i.c1.o	0.1
c3.i.c2.o	0.2
c3.i.c3.o	0.7
c4.i.c1.o	0.1
c4.i.c2.o	0.2
c4.i.c3.o	0.3
c4.i.c4.o	0.4
c5.i.c1.o	0.1
c5.i.c2.o	0.2
c5.i.c3.o	0.3
c5.i.c4.o	0.39
c5.i.c5.o	0.01
Buffer_max	100
Buffer_min	1

---

---

# BIBLIOGRAPHY

- Andersson, J. October (2013) *A General-Purpose Software Framework for Dynamic Optimization*, PhD thesis Arenberg Doctoral School, KU Leuven, Department of Electrical Engineering (ESAT/SCD) and Optimization in Engineering Center, Kasteelpark Arenberg 10, 3001-Heverlee, Belgium.
- Ask Solem et al. “Homepage — celery: Distributed task queue”, <http://www.celeryproject.org/> November (2017).
- Atlassian “Bitbucket — the git solution for professional teams”, <https://bitbucket.org/> November (2017).
- Baharev, A. “Exact and heuristic methods for tearing”, <https://sdopt-tearing.readthedocs.org/> November (2017)a.
- Baharev, A. “safe-eliminations”, <https://github.com/baharev/safe-eliminations> (2017)b.
- Baharev, A.; SCHICHL, H. and NEUMAIER, A. (2016)a “Decomposition methods for solving sparse nonlinear systems of equations”, *Submitted for publication. Available online: [http://reliablecomputing.eu/baharev\\_tearing-survey.pdf](http://reliablecomputing.eu/baharev_tearing-survey.pdf).*
- Baharev, A.; Schichl, H. and Neumaier, A. (2016)b “Ordering matrices to bordered lower triangular form with minimal border width”, *Submitted for publication. Available online: [http://reliablecomputing.eu/baharev\\_tearing-exact\\_algorithm.pdf](http://reliablecomputing.eu/baharev_tearing-exact_algorithm.pdf).*
- Baharev, A.; Neumaier, A. and Schichl, H. “Failure modes of tearing and a novel robust approach”, in *Proceedings of the 12th International Modelica Conference, Prague, Czech Republic, May 15-17, 2017*, number 132 pages 353–362 Linköping University Electronic Press (2017).

- Barton, P. I. (1995) “Structural analysis of systems of equations”, .
- Barton, P. I. (2000) “The equation oriented strategy for process flowsheeting”, *Department of Chemical Engineering, Massachusetts Institute of Technology, Cambridge, MA, 2139*, 24.
- Certik, O. et al. “SymPy python library for symbolic mathematics”, (2008).
- Concannon, K. (2006) *Simulation modeling with SIMUL8*, Visual Thinking International, .
- Devroye, L. (1986) *Non-uniform random variable generation*, Springer, .
- Diestel, R. (2000) *Graphentheory*, Springer, .
- Edgar, T. F.; Himmelblau, D. M. and Lasdon, L. S. (2001) *Optimization of chemical processes*, McGraw-Hill, .
- Fletcher, R. and Hall, J. (1993) “Ordering algorithms for irreducible sparse linear systems”, *Annals of Operations Research*, 43 (1), 15–32.
- GitHub Inc “Atom”, <https://atom.io/> October (2017).
- Griewank, A. (2000) *Evaluating derivatives: principles and techniques of algorithmic differentiation*, Society for Industrial and Applied Mathematics, .
- Griewank, A. and Walther, A. (2003) “Introduction to automatic differentiation”, *PAMM*, 2 (1), 45–49.
- Hernandez, R. and Sargent, R. (1979) “A new algorithm for process flowsheeting”, *Computers & Chemical Engineering*, 3 (1-4), 363–371.
- Khan Academy “(2) Big-O notation (article) — algorithms — khan academy”, <https://www.khanacademy.org/computing/computer-science/algorithms/asymptotic-notation/a/big-o-notation> December (2017).
- Korn, R.; Korn, E. and Kroisandt, G. (2010) *Monte Carlo methods and models in finance and insurance*, CRC press, .
- Kroese, D. P.; Taimre, T. and Botev, Z. I. (2011) *Handbook of Monte Carlo methods*, Wiley, .
- Ludwig, E. E. and Coker, A. K. (2007) *Ludwig’s applied process design for chemical and petrochemical plants*, Gulf Professional Publishing, .

- Meyer, M.; Hylton, R.; Fisher, M.; van der Merwe, A.; Streicher, G.; van Rensburg, J. J.; 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 (1), 79–92 ISSN 00922102, 1526551X URL <http://www.jstor.org/stable/23016181>.
- Microsoft “Excel 2016 by microsoft - spreadsheet software — office”, <https://products.office.com/en-us/excel?legRedir=true&CorrelationId=febc2792-9fdd-4329-aa1f-905f4b5b857c> October (2017).
- Montgomery, D. C. and Runger, G. C. (2011) *Applied statistics and probability for engineers*, Wiley, .
- Mooney, C. Z. (1997) *Monte carlo simulation*, , 116 Sage Publications, .
- NetworkX developers “Networkx”, <https://networkx.github.io/> October (2017).
- Open Source Modelica Consortium “Omedit openmodelica connection editor openmodelica user’s guide v1.13.0-dev-126-g1311439 documentation”, <https://www.openmodelica.org/doc/OpenModelicaUsersGuide/latest/omedit.html> October (2017).
- Python Software Foundation “Python.org”, <https://www.python.org/> October (2017).
- Ripley, B. D. (2006) *Stochastic simulation*, Wiley-Interscience, .
- Shacham, M.; Macchieto, S.; Stutzman, L. and Babcock, P. (1982) “Equation oriented approach to process flowsheeting”, *Computers and Chemical Engineering*, 6 (2), 7995.
- SIMUL8 Corporation “Simul8 simulation software - for visual process simulation modeling”, <https://www.simul8.com/> December (2017).
- Software Freedom Conservancy “Git”, <https://git-scm.com/> November (2017).
- Streicher, G. “A stochastic simulation model of a continuous value chain operation with feedback streams and optimization”, in *Proceedings of the 2013 Winter Simulation Conference: Simulation: Making Decisions in a Complex World*, pages 3912–3912 Piscataway, New Jersey (2013) IEEE Press.
- The AnyLogic Company “Simulation modeling software tools & solutions for business anylogic”, <https://www.anylogic.com/> December (2017).
- The HDF Group “Hdf group - hdf5”, <https://support.hdfgroup.org/HDF5/> November (2017).

Wächter, A. and Biegler, L. T. (2006) “On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming”, *Mathematical programming*, 106 (1), 25–57.

Wilkinson, B. and Allen, M. (2005) *Parallel programming: techniques and applications using networked workstations and parallel computers*, Pearson/Prentice Hall, .