



Implementation of an NH₃ Storage Model Based SCR-Simulation Algorithm

Andreas Holmqvist

Bachelor's Thesis

Electrical Engineering

Vaasa 2016



BACHELOR'S THESIS

Author: Andreas Holmqvist
Degree Programme: Electrical Engineering, Vaasa
Specialization: Automation
Supervisor: Ray Pörn

Title: *Implementation of an NH₃ Storage Model Based SCR-Simulation Algorithm*

Date 26.4.2016 Number of pages 55 Appendices 3

Abstract

This thesis was commissioned by Wärtsilä Catalyst Systems. The SCR-process is a chemical process, which is used to reduce the amount of nitrogen oxides (NO_x) in the exhaust gas of diesel and gas engines. The dynamics of the chemical reactions that are involved in the SCR-process have a great impact on the functionality of the process and must be modeled in order to be taken into consideration in the control system. The objective of the thesis project was to create a practical implementation of a real-time SCR-simulation algorithm, which simulates the internal states of the chemical reactions involved in the SCR-process. Based on the simulation and the state observer, two practically unmeasurable process variables are calculated: the amount of stored ammonia (NH₃), and its distribution profile across the SCR-reactor.

The essential task of the thesis project was to create a reliable and stable application that beyond the SCR-simulation would also include a number of other functionalities such as communication diagnostics and multi SCR-simulation. A Raspberry Pi computer was chosen as a simulation platform for the thesis project and the programming language of choice was Python. The result of the thesis project was a full-scale simulation application which is fully integrable with the control system of Wärtsilä's SCR-systems. Beyond the development process as well as a functional description of the created application, the thesis also covers and describes the real SCR-tests that were conducted in Bermeo, Spain and the optimization of the application.

Language: English Key words: NH₃, NO_x, Python, SCR, Simulation

EXAMENSARBETE

Författare: Andreas Holmqvist
Utbildningsprogram och ort: Elektroteknik, Vasa
Profilerings: Automation
Handledare: Ray Pörn

Titel: *Implementering av en NH₃ lagringsmodellbaserad SCR-simuleringsalgoritm*

Datum 26.4.2016

Sidantal 55

Bilagor 3

Abstrakt

Detta examensarbete utfördes åt Wärtsilä Catalyst Systems. SCR-processen är en kemisk process som används för att reducera mängden kväveoxider (NO_x) i avgaserna från diesel- och gasmotorer. Dynamiken hos de kemiska reaktionerna som ingår i SCR-processen har en stor inverkan på processens funktionalitet och måste modelleras för att kunna tas i beaktande i styrsystemet. Målet med examensarbetet var att skapa en praktisk implementation av en simuleringsalgoritm som i realtid används för att simulera de kemiska reaktionernas interna tillstånd. Baserat på simuleringen och tillståndsobservatören, så beräknas två, i praktiken omätbara processvariabler: mängden lagrad ammoniak (NH₃) och dess lagringsprofil genom SCR-reaktorn.

Den grundläggande uppgiften i examensarbetet var att skapa en pålitlig och stabil applikation som förutom SCR-simuleringen, även innehöll en mängd annan funktionalitet såsom kommunikationsdiagnostik och multi-SCR-simulering. En Raspberry Pi dator valdes som simuleringsplattform och som programmeringsspråk valdes Python. Resultatet av examensarbetet var en fullständig simuleringsapplikation som till fullo kan integreras i styrsystemet hos Wärtsiläs SCR-system. Förutom utvecklingsprocessen och en funktionell beskrivning av applikationen, så täcker och beskriver examensarbetet även de verkliga SCR-testerna som utfördes i Bermeo, Spanien samt applikationsoptimeringen.

Språk: engelska

Nyckelord: NH₃, NO_x, Python, SCR, Simulering

OPINNÄYTETYÖ

Tekijä: Andreas Holmqvist
Koulutusohjelma ja paikkakunta: Sähkötekniikka, Vaasa
Suuntautumisvaihtoehto: Automaatiotekniikka
Ohjaaja: Ray Pörn

Nimike: *NH₃-varastointimalliin perustuva SCR-simulointialgoritmin implementointi*

Päivämäärä 26.4.2016 Sivumäärä 55 Liitteet 3

Tiivistelmä

Tämä opinnäytetyö suoritettiin Wärtsilä Catalyst Systemsille. SCR-prosessi on kemiallinen prosessi, jota käytetään vähentämään diesel- ja kaasumoottoreiden pakokaasuissa olevaa typpioksidien (NO_x) määrää. SCR-prosessin kemiallisten reaktioiden dynamiikalla on suuri vaikutus prosessin toiminnallisuuteen, mutta ne on simuloitava, jotta ne voidaan ottaa huomioon ohjausjärjestelmässä. Opinnäytetyön tarkoitus oli luoda käytännöllinen implementointi reaaliaikaisesta simulointialgoritmista, jota käytetään SCR-prosessin kemiallisten reaktioiden sisäisten tilojen simulointiin. Simuloinnin ja tilahavaitsijan perusteella lasketaan kaksi käytännössä mittaamatonta prosessivariaabelia: SCR-reaktorissa varastoidun ammoniakkin (NH₃) määrä ja sen talletusprofiili.

Opinnäytetyön perimmäinen tehtävä oli luoda luotettava ja tasainen applikaatio, joka SCR-simuloinnin ohessa sisälsi myös muita toimintoja, kuten kommunikaation diagnostiikkaa ja moni-SCR-simulointia. Simulointialustaksi valittiin Raspberry Pi-tietokone ja ohjelmointikieleksi valittiin Python. Tämän opinnäytetyön tuloksena oli täydellinen simulointiapplikaatio, joka integroituu täydellisesti Wärtsilän SCR-järjestelmien ohjausjärjestelmiin. Kehitysprosessin ja applikaation käytännöllisen kuvauksen ohessa, tämä opinnäytetyö käsittää myös Espanjan Bermeossa tehdyt varsinaiset SCR-testit ja applikaation optimointi.

Kieli: englanti

Avainsanat: NH₃, NO_x, Python, SCR, Simulointi

Table of contents

1	Introduction.....	1
1.1	Thesis background	1
1.2	Thesis objectives and boundaries	2
1.3	Wärtsilä Oyj	3
1.3.1	Catalyst Systems	4
2	Theory	6
2.1	NO _x emissions	6
2.2	Selective catalytic reduction - SCR	7
2.2.1	Process overview	8
2.2.2	Reductants and chemical behavior	11
2.2.3	SCR modeling	13
2.3	Object oriented programming.....	17
2.4	Python	19
2.4.1	Typing system and syntax	19
2.4.2	Libraries and modules	20
2.5	Raspberry Pi.....	21
2.5.1	Hardware	21
2.5.2	Raspbian	22
3	Project realization.....	24
3.1	Methods and approach	24
3.1.1	Implementation development	25
3.1.2	Real SCR tests and result analysis	26
3.1.3	Implementation optimization	26
3.2	Model implementation.....	26
3.2.1	Code architecture.....	27
3.2.2	General functionality.....	29
3.2.3	Program flow and simulation logic	30
3.2.4	Modbus communication	32
3.3	Simulation versus Matlab	35
3.4	Real SCR tests	37
3.4.1	Test definition and setup	37
3.4.2	Test results.....	39
3.5	Code optimization.....	45
3.5.1	Profile tests and results.....	46
3.5.2	Optimization results	49
4	Results and conclusion.....	52
5	Discussion	54
	References.....	56

Appendix

List of abbreviations

ASDC	Ammonia Storage Distribution Control
CPU	Central Processing Unit
CSO	Combined SCR and OXI
CSTR	Continuous Stirred Tank Reactor
DOC	Diesel Oxidation Catalyst
NH ₃	Ammonia
NOR	NO _x -Reducer
NO _x	Nitrogen Oxides
OO(P)	Object Oriented (Programming)
OXI	Oxidation Catalyst
PLC	Programmable Logic Controller
R&D	Research & Development
SCR	Selective Catalytic Reduction
SO _x	Sulphur Oxides
TCP/IP	Transmission Control Protocol / Internet Protocol
TWC	Three Way Catalyst
UNIC	Unified Control

List of figures

Figure 1. Wärtsilä's NOR product layout.	5
Figure 2. SCR Catalyst.	9
Figure 3. Process diagram over Wärtsilä's SCR-system NOR.	10
Figure 4. Schematic description of ammonia storage distribution control.	16
Figure 5. Code example 1.	18
Figure 6. Code example 2.	19
Figure 7. The Raspberry Pi 2 B+. (Upton, 2015)	22
Figure 8. Raspbian user interface.	23
Figure 9. Script start up and initialization logic.	29
Figure 10. Simulation logic. N corresponds to the number of SCRs simulated.	31
Figure 11. Modbus handshake functionality.	33
Figure 12. Calculated outgoing NO _x and ammonia by Python and Matlab.	35
Figure 13. Calculated differences between the Matlab application and the Python application.	36
Figure 14. Test network topology.	38
Figure 15. Real SCR test control set up.	38
Figure 16. Calculated ammonia coverage (green) and ammonia coverage set point (red).	39
Figure 17. NO _x measured by analyzer (green) versus by model calculated NO _x (red).	40
Figure 18. NH ₃ measured by analyzer (green) versus by model calculated NH ₃ (red).	41
Figure 19. Calculated ammonia coverage (green) and ammonia coverage set point (red) in a situation where the model was influencing the urea control.	42
Figure 20. NO _x measured by analyzer (green) versus by model calculated NO _x (red) in a situation where the model was influencing the urea control.	43
Figure 21. NH ₃ measured by analyzer versus by model calculated NH ₃ in a situation where the model was influencing the urea control.	44
Figure 22. Code profile report. Report ordered by actual time.	46
Figure 23. Code profile report. Report ordered by actual time.	47
Figure 24. Bottleneck.	48
Figure 25. Code profile report after optimization. Report ordered by actual time.	49
Figure 26. Code optimization results.	50
Figure 27. Estimated actual iteration time per number of simulated SCR-systems.	51

List of tables

Table 1. System components in the process diagram in figure 3.	10
Table 2. Symbol descriptions of equations 2.3.3.1 - 2.3.3.5.	14
Table 3. Functions and corresponding input and output data.....	27
Table 4. The SCR class' methods and their corresponding input and output data.	28
Table 5. Time stamp variable descriptions.....	30
Table 6. Application requirements.	52

1 Introduction

This thesis was commissioned by Wärtsilä Catalyst Systems of the Environmental Solutions organization in Vaasa, Finland. The goal was to create a practical implementation of a real-time SCR-simulation algorithm together with the control system of an SCR-system. The thesis describes the implementation of the modeling algorithm as it was implemented in the programming language Python, all auxiliary functionalities that were created, the tests that were conducted in office and in Bermeo, Spain, as well as the optimization of the code. Furthermore, this thesis also covers and describes relevant theory subjects that were studied throughout this project.

1.1 Thesis background

The dynamics of the chemical reactions that are involved in the SCR-process are quite complex and the classical feed-forward control of the process is unable to take these dynamics into consideration during process control. In order to take these dynamics into consideration in the control system, they have to be mathematically modeled and simulated. An SCR-modeling algorithm had already been created and tested by Catalyst Systems R&D by the time this thesis project was launched. But as the model had been implemented in Matlab, it was not an implementation that could be used together with a real SCR-system, due to Matlab requiring a PC and a license. The Matlab application also lacked crucial functionality that would be required for it to be successfully implemented together with the control system of the SCR. Functionality for Modbus TCP/IP communication, multi-SCR-simulation and data validation were the major deficiencies of the Matlab application.

The original goal of creating an SCR-simulation algorithm was to master dynamic situations caused by engine load alterations. Engine load alterations cause hefty changes in many of the process variables that are crucial to the SCR-process, and as such these sudden changes will also affect the dynamics of the process itself. By creating a simulation algorithm of the process, these rapid changes in the dynamics of the process can to some extent be mastered and thus the negative impacts on the exhaust emissions during, and after, these situations can be limited. Thus, this thesis project was started, in order to create an implementation of the model that would be possible to use in practice.

1.2 Thesis objectives and boundaries

The main purpose of the thesis was to create a real-time practical implementation of an SCR-simulation algorithm in the programming language Python. A Raspberry Pi was to be used as a testing platform and as the main device running the application and interacting with the control system of the SCR-process. The project would also include application testing together with a real Wärtsilä CSO SCR-system and a Wärtsilä engine, in order to validate the simulation functionality and the auxiliary functionalities of the application. Therefore, the main purpose of the thesis was not to just implement the simulation algorithm in another programming language, but also to create auxiliary functionality that would make it possible to use the simulation application safely together with a real CSO-system.

Automated control systems controlling industrial processes must under all circumstances be robust, reliable and safe. As such, software applications must be created in such a manner that they can interact the control system in a safe and reliable way. On top of this, new software additions must also fulfill various technical requirements, so that the actual functionality corresponds to the expected functionality. Therefore, there were some requirements that the application had to fulfill. These requirements were listed as follows by Catalyst Systems:

- Support for simulation of up to twelve individual SCR-systems.
- Support for communication over Modbus TCP/IP.
- Support for re-initialization of Modbus connection upon failure.
- Support for validation of simulation data.
- Ability to recover simulation of a SCR-system that returns from failure mode.
- Iteration time under 3.5 milliseconds when simulating one SCR-system.
- Iteration time under 50 milliseconds when simulating twelve SCR-systems.
- Support portability to other operating systems.

The reason for the application supporting simulation of twelve individual SCR-systems, is that the overall PLC-application that controls Wärtsilä's SCR-systems is capable of controlling up to twelve CSO-systems. Therefore, it was only natural that the application would also support simulation of up to twelve SCR-systems. The iteration benchmark had been set at 50 milliseconds when simulating twelve SCR-systems. It had prior to this thesis

project been noted that trying to simulate the SCR-process with a time step above this set benchmark would make the model unstable and the simulated data would not correspond to actual values.

The expected result from the thesis project was in other words a full-scale SCR-simulation application that could independently interact with the control system, while also complying with the requirements stated above. The last requirement listed above, was more or less fulfilled just by choosing Python as programming language, as scripts written in Python can be run on multiple other operating systems that have a Python interpreter installed.

1.3 Wärtsilä Oyj

Wärtsilä is a Finland-based global corporation that offers its customers complete lifecycle power solutions for marine, off shore and land based applications. Wärtsilä operates in over 200 locations distributed across approximately 70 countries and employs totally around 17700 workers all over the globe. The Wärtsilä organization is divided into three major business units:

- Marine Solutions
- Energy Solutions
- Services

Marine Solutions provides its customers in the marine, oil and gas industry with solutions for ship machinery, propulsion and ship maneuvering. The products of Marine Solutions include engines and power generating sets, propulsion equipment, control systems and various engine auxiliary systems such as exhaust gas after treatment systems.

Energy solutions provides its customers in the decentralized energy market and in the oil and gas industry with power plant solutions for baseload, peaking and industrial self-generation purposes. The product portfolio ranges from single engines to full-scale turn-key power plant solutions.

Services is the biggest business segment of the three and provides its customers in all types of industrial and business segments with maintenance, reconditioning, support and services throughout the full lifecycle of their installations. They also support the two other business units in various ways. (Wärtsilä, 2015)

1.3.1 Catalyst Systems

Catalyst Systems is a section in the Environmental Solutions organization of the Marine Solutions business segment. Catalyst Systems sells, designs, manages and develops Wärtsilä's own exhaust gas after treatment systems for the reduction of nitrogen oxides in the exhaust gas from Wärtsilä's engines. The products are suited for both new installments and retrofits for marine and power plant applications that run on gaseous and/or liquid fuels. As marine and power plant applications may differ quite heavily, Catalyst Systems offers two different SCR-systems, each corresponding to the application. The marine SCR-system is called NOR (NO_x-Reducer) and the power plant SCR-system is called CSO (Combined SCR and OXI). Both systems are based on the selective catalytic reduction (SCR) technology, which means that the systems reduce the level of NO_x in the exhaust gas by means of catalytic elements and a reducing agent. Both the NOR and the CSO are modularized systems that are compliant with various NO_x emission limits and can be installed together with other after treatment systems such as scrubber systems for SO_x-removal. The standard scope of delivery for one SCR-system include the following main components:

- SCR-reactor
- Catalyst elements
- Soot blowing unit
- Urea injection and mixing unit
- Urea pump unit
- Air unit
- Urea dosing unit
- Control and automation unit

Other optional modules include:

- Mixing duct
- NO_x monitoring system
- Urea tank
- Compressed air unit

The SCR-reactor, catalyst elements, soot blowing unit, urea injection and mixing unit as well as the dosing unit are modules and components that are engine specific. This means that every specific NOR- and CSO-system will require one or several pieces of these modules and components. The rest of the modules listed above are centralized, which then in turn means that they can under certain circumstances be shared by a number of independent NOR- and CSO-systems. Figure 1 below illustrates Wärtsilä's NOR product layout with both standard and optional components.

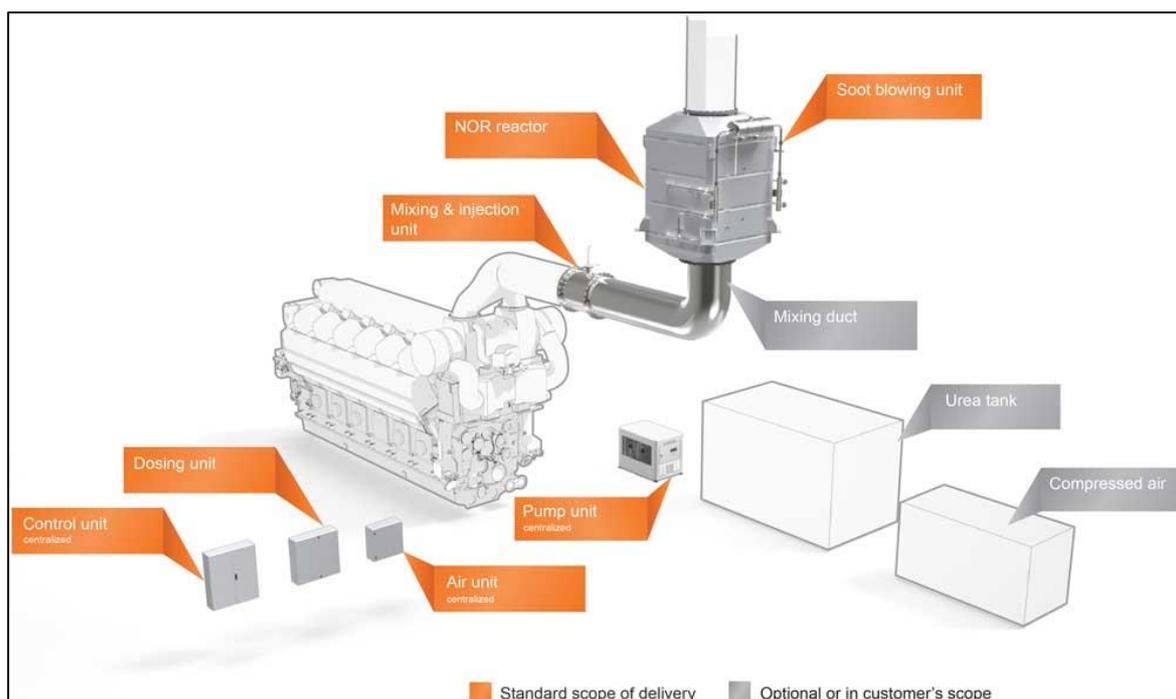


Figure 1. Wärtsilä's NOR product layout. (Wärtsilä, 2014, p. 11)

Generally, the NOR and the CSO are different versions of the same product. The main differences lie in the automatic control system and in the reactor. The NOR is controlled by Wärtsilä's own automation platform UNIC, while the CSO is controlled by a PLC-system. The CSO will also include an oxidation catalyst (OXI) layer in the reactor which is used to oxidize various components in the exhaust gas before they are released into the atmosphere. In all other aspects, the two version are more or less identical in terms of both components and functionality.

2 Theory

This section describes necessary theoretical subjects that were thoroughly studied during the project process and the writing of this thesis. It provides a general introduction to relevant theory regarding NO_x-emissions, selective catalytic reduction, object oriented programming, Python and Raspberry Pi. Together these theory subjects create a basic knowledge foundation, which is required for a good understanding of the work done in this thesis. Note that theory chapter 2.2 *Selective Catalytic Reduction* and its subchapters focus on SCR-technology in large diesel and gas applications found in power plant applications and on marine vessels.

2.1 NO_x emissions

Nitrogen oxides (NO_x) are gaseous emissions that are formed when air is heated to high temperatures, for instance in combustion engines and when lightning strikes the surface of the earth. The general definition of NO_x today includes the compounds nitric oxide (NO) and nitrogen dioxide (NO₂). NO and NO₂ are compounds that are of major concern in terms of air pollution as they are the only nitrogen oxides that are emitted in substantial quantities. NO_x emissions are therefore often legislated and local emission limits must not be exceeded. The emission limits often depend on country, location and application. As of today, the European Union and the USA impose the heftiest regulations concerning the amounts of NO_x emitted from larger diesel engines. (Chatterjee & Rusch, 2014, p. 33; Peavy, et. al., 1958, p. 455)

Typically, the concentration of NO_x in untreated diesel exhausts ranges from 50 to 1000 ppm, depending on combustion temperature, engine load and fuel quality. The NO₂/NO ratio in the NO_x-gases emitted from a diesel engine varies from 10% - 20%. Nitric oxide is formed when the nitrogen (N) in the air reacts with atmospheric oxygen (O) in combustion processes with high temperatures and pressure, as can be seen in the reaction formula 2.1.1 below. (Eastwood, 2000, p. 10-11; Khair & Majewski, 2006, p. 123)



Nitrogen dioxide on the other hand, is formed when nitric oxide is oxidized by the oxygen in the surrounding air, as can be seen in the reaction formula 2.1.2 below.



Normally this reaction occurs spontaneously in the mixture of nitric oxide and air when the exhaust gases from the engine have been discharged into the exhaust gas duct or into the atmosphere. In other words, nitrogen dioxide is not directly formed in the combustion process. Despite this, NO_x-gases are still considered to be directly emitted from a source, making them primary pollutants. Even though nitric oxide and nitrogen dioxide share a lot of common attributes, they are not equivalents in terms of environmental impact. Nitric oxide is odorless and only moderately toxic to humans and animals, while nitrogen dioxide is both toxic and carry a distinct and unpleasant odor. NO_x-emissions are generally considered to be primary pollutants as they are emitted directly from a source. Aside from this, they are also involved in other atmospheric reactions which form secondary pollutants such as acid rain and smog. Due to this concern it can be argued whether or not NO_x-gases can also be considered as secondary pollutants. They will also take part in the depletion of the earth's ozone layer. (Eastwood, 2000, p. 11-12)

2.2 Selective catalytic reduction - SCR

NO_x that is emitted from gasoline engines can be effectively reduced through three way catalysts (TWC). However, TWC's must operate under stoichiometric conditions, making it impossible to use them together with lean-burn engines such as gas and diesel engines. Selective catalytic reduction (SCR) is an active and well established exhaust gas after treatment technology for the reduction of NO_x in diesel and gas engine exhausts. SCR-systems are mainly used in power generation, marine and truck applications where heavy duty engines are used. The fuels that may be used in these applications together with a SCR system include various industrial gases, natural gas, crude, light and heavy fuel oil. The SCR-process is primarily a chemical process that utilizes catalytic elements as well as a reducing agent to reduce the levels of NO_x in the exhaust gases to locally accepted levels. (Johnson, 2014, p. 1; Khair & Majewski, 2006, p. 416)

SCR-systems hold one significant advantage over other established diesel exhaust after treatment technologies commercially available such as NO_x-adsorbers, as they have the highest NO_x conversion ratio (>90%) of all technologies available. Despite this, SCR-systems also hold some disadvantages as they have relatively high operational costs and require both space and a sophisticated control system. Furthermore, SCR systems that uses

urea and/or ammonia as reductant generate under certain conditions unwanted ammonia emissions, known as ammonia slip. (Khair & Majewski, 2006, p. 405)

2.2.1 Process overview

An SCR-system consists of a set of standard components that are crucial to the functionality of the SCR-process, irrespective of system manufacturer. These components are:

- Reactor
- Catalyst elements
- Injection and mixing unit
- Pump unit
- Reducing agent tank
- Control unit

The main component in a SCR-system is the reactor, where most of the chemical reactions involved in the SCR-process take place. The reactor is a steel casing with an inlet and an outlet installed on the exhaust pipe system of the engine. The main purposes of the reactor is to both house the catalyst elements and to provide a suitable environment for the chemical reactions of the SCR-process. The reactions that take place inside the reactor must occur under suitable temperatures, which is why in order to minimize the heat dissipation, heat insulation is usually installed around the reactor housing. (Personal communication with Wärtsilä employee.)

A chemical catalyst can be defined as a substance that accelerates the reaction rate of one or several chemical reactions, without directly taking part of the reactions between the reactants. Thus, it is not a product of the reaction(s) and should normally not even be altered by the reactions. When two or more potential reactants are to chemically react with each other, they must first pass through an energy barrier called activation energy (E). The activation energy is the threshold potential energy which is needed for the mentioned reactions to take place and transform into a final product. A catalyst lowers the activation needed for the reactants to chemically react with each other, thus speeding up the reaction. Thus, the catalyst elements that are used in SCR-systems, are the most critical component in terms of chemical functionality. The elements are often brick-shaped cuboids with a

varying amount of cells that usually take the form of square-type honeycombs. As the cells go right through the catalyst elements, they allow the exhaust gas to flow right through, while at the same time allowing the chemical reactions of the NO_x -reduction to take place on the surrounding cell walls. The elements are packed together in a metal frame installed inside the reactor in a number of layers. Figure 2 below contains a picture of an SCR-catalyst element. (Chatterjee & Rusch, 2014, p. 55; Khair & Majewski, p. 371, 368)

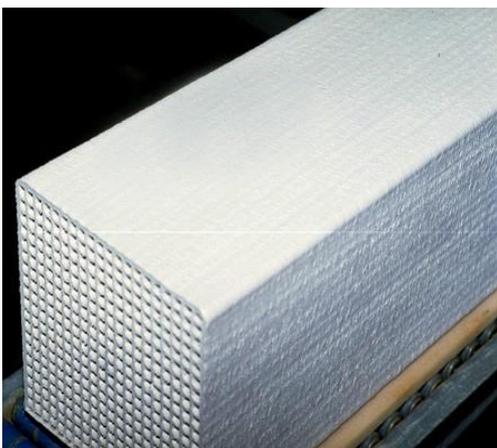


Figure 2. SCR Catalyst. (Wärtsilä internal documentation)

In order to reduce the amounts of NO_x in the exhausts, some sort of reducing agent needs to be added to the exhaust gases as the reduction is a chemical reaction taking place in the reactor. The reducing agent is injected through a nozzle installed in the piping between the SCR-reactor and the engine. It is of high importance that the exhaust gas and the reducing agent are well mixed, which is why in some cases the reducing agent is injected together with pressurized air in order to atomize the reducing agent which then in turn improves the mixing. Furthermore, a vortex mixer can also be installed in the duct before the injection point in order to induce turbulence to the exhaust gas flow. Turbulence further improves the mixing by distributing the agent more evenly across the cross section of the duct. (Chatterjee & Rusch, 2014, p. 54)

The reducing agent is generally stored in a tank in the proximity of the SCR-system. The pump unit feeds the injection system through a pipe system with the reducing agent from the storage tank at a suitable pressure. Due to the activeness of the SCR-process, a sophisticated control system is needed to control the different components used in the system, such as the pump(s), the injection and the valves, as well as for monitoring different process variables such as temperatures, levels of NO_x and exhaust gas flow. The automatic control system is often located in an automation unit in the proximity of the

SCR-system. The device that controls the SCR-system can vary depending on the complexity of the system as well as on the application itself, but for larger applications a PLC-system is most commonly used. (Internal communication with Wärtsilä employee)

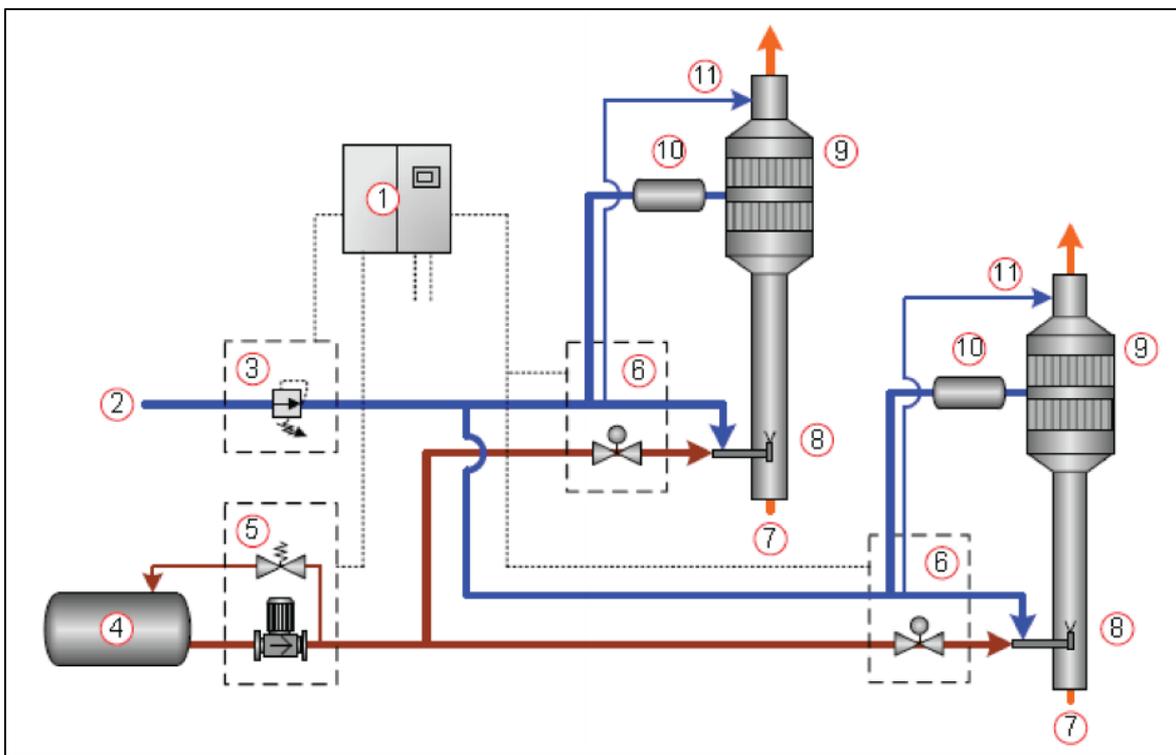


Figure 3. Process diagram over Wärtsilä's SCR-system NOR. (Wärtsilä, 2014, p.13)

The figure above is an example of what a process diagram over an SCR-system with two engines can look like. The numbers in the figure above correspond to the components in table 1 below. Note that the components in the table and in the figure above only corresponds to Wärtsilä's SCR-system. Other manufacturers of SCR-systems may use other terminology or completely different components.

1	Control unit	7	Incoming exhaust gas
2	Pressurized air supply	8	Injection and mixing unit
3	Air unit	9	Reactor
4	Reducing agent tank	10	Soot blowing unit
5	Pump unit	11	NO _x -sensor purge
6	Dosing unit		

Table 1. System components in the process diagram in figure 3. (Wärtsilä, 2014, p. 13)

2.2.2 Reductants and chemical behavior

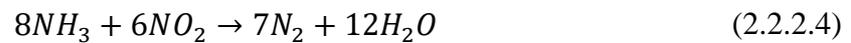
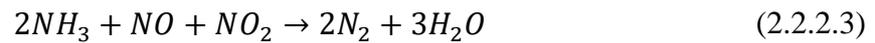
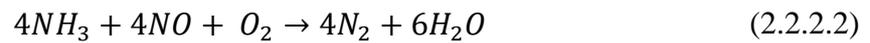
Generally, there are two forms of reductants that can be used in SCR systems: ammonia (NH_3) and urea ($\text{CO}(\text{NH}_2)_2$). The ammonia used in SCR applications can take the form of pure anhydrous ammonia, which is an ammonia substance that does not contain any water and aqueous ammonia, which is a water-ammonia solution. Anhydrous ammonia is toxic and of a hazardous nature due to its high vapor pressure. Aqueous ammonia on the other hand is less toxic in comparison with anhydrous ammonia and is somewhat easier and safer to handle. (Khair & Majewski, p. 416-417)

As the toxic and hazardous nature of ammonia implies safety and handling problems, a safer and more convenient SCR-reductant is desirable. Urea is today the most recognized alternative reactant to anhydrous ammonia and aqueous ammonia, as it holds both commercial and technical advantages over the two forms of ammonia. Urea is both non-toxic and safe to handle and is practically commercially available all over the world due to its widespread use in the food processing industry, as well as a fertilizer. Urea is a solid substance under normal conditions, however, when used as a reductant in SCR-systems it takes the form of a liquid as a urea-water solution with a concentration of approximately 32,5 weight %. (Khair & Majewski, p. 418)

However, the choice of reductant comes to the same end. Urea will start to thermally decompose and undergo hydrolysis in the same moment as it is injected into the exhaust gas duct, provided that the urea is heated up to temperatures higher than 160°C . When the urea undergoes decomposition and hydrolysis it will primarily form ammonia and carbon dioxide according to reaction formula 2.2.2.1 below. (Khair & Majewski, p. 418)

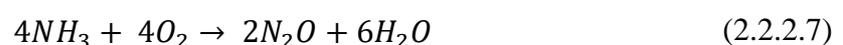


As mentioned earlier, most of the chemical reactions involved in the SCR-process take place in the SCR-reactor, or, more specifically, on the cell walls of the catalyst elements that are located in the SCR-reactor. After the reducing agent has been injected into the exhaust gas flow it enters the SCR-reactor. When the exhaust gas and the reducing agent reach the catalyst elements the reducing agent will begin to be adsorbed by the catalyst elements. The selective catalytic reduction will then start to take place in accordance with the reaction formulas below:



These three formulas above are the key chemical reactions that take place in the SCR-process. As can be seen in the three formulas above, the incoming ammonia reacts with the nitrogen oxides and nitrogen dioxides. The products of the reactions are, in comparison with the reactants much less harmful as the reactions generate various amounts of molecular nitrogen and water. Formula 2.2.2.2 is generally considered to be the standard reaction formula in SCR-systems. However, formula 2.2.2.3 is generally considered to be the preferred way of reducing NO_x, as it is the fastest reaction of the three (often referred to as fast-SCR.), reducing both NO and NO₂ at the same time. As mentioned earlier, the amount of NO₂ in NO_x-gases is much less than NO, which creates problems as the preferred formula 2.2.2.3 requires equal amounts of NO and NO₂. In order to enhance the reduction of NO_x through formula 2.2.2.3, a diesel oxidation catalyst (DOC) can be installed before the SCR-reactor. The DOC forms more NO₂ by oxidizing the nitrogen oxide in the exhaust, thus promoting the reduction of NO_x through reaction formula 2.2.2.3. (Johnson, 2014, p. 10)

Naturally, an insufficient ammonia or urea injection results in low NO_x-reduction, while a high injection rate results in high NO_x-reduction but also a high amount of ammonia emitted into the atmosphere. As previously stated, the SCR-process may under certain circumstances produce ammonia slip, either due to bad mixing, excess dosing or ammonia stored on the catalyst elements being released before reacting with the NO_x. The ammonia slip is highly unwanted as ammonia may react with oxygen in accordance with the reaction formulas below:



As can be seen in the formulas above, the reductant may in this case actually form the pollutant it is designed to reduce. Ammonia is however much less likely to react with oxygen according to formula 2.2.2.6 in comparison with NO_x. Despite this, ammonia

emissions are still of great concern. Ammonia can also form toxic products such as various ammonium sulphates. This applies especially to SCR-systems in diesel applications, as diesel generally contain higher sulphur (S) levels, which naturally amounts to higher sulphur oxide (SO_x) levels in the exhausts. These sulphates are also of great matter as they will deposit in the catalyst elements and on the inside walls of the reactor. The fouling of the catalyst elements can have severe negative impacts on the chemical functionality catalyst elements as they will at first mask the active sites of the elements and furthermore cause clogging in the reactor and the catalyst elements. (Eastwood, 2000, p. 207-208; Hsieh & Wang, 2014, p. 427; Johnson, 2014, p. 11; Khair & Majewski, p. 416-417)

2.2.3 SCR modeling

The automatic control of an SCR-process comes with a lot of challenges due to the complex dynamics of the chemical reactions in the process. The process also contains one parameter that has a significant impact on the functionality of the whole process; the ammonia coverage ratio of the catalyst elements. The ammonia coverage ratio is the main reason for simulating the process, as it is practically impossible to measure and must therefore be modeled. The ammonia coverage ratio is defined as the total amount of ammonia momentarily adsorbed by the catalyst elements, that is to say the total amount of ammonia inside the reactor. Furthermore, adding transient engine operation to the mix and the result is a mathematically complex and nonlinear process. This makes the design and control of the urea injection quite a challenge by conventional means, as both NO_x and ammonia emissions are preferably to be simultaneously kept at a minimum. (Hsieh & Wang, 2014, p. 425-426, 441)

The SCR process can be condensed in three major stages. In the first stage, urea is injected into the exhaust gas duct and starts to undergo thermal decomposition and hydrolysis, ultimately forming ammonia. In the second stage, the ammonia enters the reactor and is then adsorbed by the catalyst elements. In the third and the final stage of the process, the adsorbed ammonia reacts with the NO_x and forms nitrogen and water. All of these stages are vital in terms of functionality of the SCR-process. In order to make it possible for the control system to take these stages and their corresponding dynamics into consideration on behalf of the injection control, the corresponding reaction rates must be mathematically modeled and estimated. The reaction rates of each corresponding reaction is modeled by Arrhenius equations as can be seen below. (Hsieh & Wang, 2014, p. 426-427)

NO reduction:

$$R_1 = K_1 e^{-\frac{E_1}{RT}} C_{NO} C_{O_2} \theta \Theta V^2 \quad (2.2.3.1)$$

NO and NO₂ reduction:

$$R_2 = K_2 e^{-\frac{E_2}{RT}} C_{NO} C_{NO_2} \theta \Theta V^2 \quad (2.2.3.2)$$

NH₃ adsorption:

$$R_{4F} = K_{4F} e^{-\frac{E_{4F}}{RT}} C_{NH_3} (1 - \theta) \Theta V \quad (2.2.3.3)$$

NH₃ desorption:

$$R_{4R} = K_{4R} e^{-\frac{E_{4R}}{RT}} \theta \Theta V \quad (2.2.3.4)$$

NO oxidation:

$$R_5 = K_5 e^{-\frac{E_5}{RT}} C_{NO} C_{CO_2} V^2 \quad (2.2.3.5)$$

The symbols in the equations above correspond to the variables in the table below.

Symbol	Description	Unit
R_i	Reaction rate of the corresponding reaction.	mole/s/m ³
K_i	Rate constant of the corresponding reaction.	(Element type dependent)
E_i	Activation energy.	J
T	Temperature.	K
R	Universal gas constant.	(~8.314) J mole ⁻¹ K ⁻¹
C_x	Mole concentration of the species in the subscript.	mole/m ³
V	Catalyst volume.	m ³
θ	Ammonia coverage ratio.	- (0-1)
Θ	Total ammonia adsorption capacity.	mole

Table 2. Symbol descriptions of equations 2.3.3.1 - 2.3.3.5. (Hsieh & Wang, 2014, p. 428)

In order to avoid partial differential equations, the SCR-reactor can be assumed to be a continuous stirred tank reactor (CSTR) according to Hsieh and Wang (2014, p. 429) so that a zero-dimensional model can be created. Naturally, this a great simplification of the dynamics of the process. However, this allows a fast simulation of the internal states of the chemical reactions. Because of this CSTR assumption, the different states of the SCR-process can be considered to be homogenous. Based on this CSTR assumption and the mass conservation law, the dynamic equations that describe the different states of the SCR-process can be created in accordance with the formulas below:

$$VC_{NO}^{\dot{}} = FC_{NO,in} - R_1 - 0.5R_2 - R_5 - FC_{NO} \quad (2.2.3.6)$$

$$VC_{NO_2}^{\dot{}} = FC_{NO_2,in} - 0.5R_2 - R_5 - FC_{NO_2} \quad (2.2.3.7)$$

$$VC_{NH_3}^{\dot{}} = FC_{NH_3,in} - R_{4F} + R_{4R} - FC_{NH_3} \quad (2.2.3.8)$$

$$VM_{NH_3}^{\dot{}} = R_{4F} - R_{4R} - R_1 - R_2 - R_3 \quad (2.2.3.9)$$

F is the exhaust gas mass flow rate (kg/s), $C_{x(in)}$ is the concentration of the species in the subscript and R_i is the reaction rates in formulas 2.3.3.1 – 2.3.3.5. These equations describe how the amounts and concentrations of NO_x and NH_3 in the SCR-process change with respect to time. It is based on these four equations that the stored amount of ammonia in the reactor, outgoing NO_x and ammonia can be calculated. (Personal communication with Wärtsilä employee; Hsieh & Wang, 2014, p. 429)

Based on the reaction rates in equations 2.3.4.1 through 2.3.4.5, the most important input variables are inlet and outlet NO_x concentration, inlet and outlet NH_3 concentration, exhaust gas flow rate, reactor temperature and the catalyst ammonia coverage ratio. The NO_x concentrations, NH_3 concentrations, the temperatures as well as the exhaust flow rate are to be seen as input data to the SCR model, while the ammonia coverage ratio is the main output data of interest. Furthermore, calculated outgoing NO_x and NH_3 may also be output data of interest as they may be used to validate the functionality of the model itself by comparing them to real, measured values. (Personal communication with Wärtsilä employee; Hsieh & Wang, 2014, p.430-431)

The input variables can generally be measured by sensors or other equipment directly from the process. As can be noted by examining formulas 2.3.3.1 - 2.3.3.5, the SCR model will also require other variables and constants than the input variables listed above. Mole concentrations can be easily calculated as the exhaust flow rate and the incoming NO_x and NH_3 levels are measured and given as input data to the model. The constants catalyst volume and activation energy are rather easy to measure or otherwise estimate. The ammonia storage capacity can also be considered to be a constant, but unlike the other constants, the storage capacity is quite challenging to estimate. (Personal communication with Wärtsilä employee; Hsieh & Wang, 2014, p. 428-430)

The main objective of modeling the SCR-process is to create a state observer of the SCR-process so that the ammonia coverage ratio can be estimated. Depending on implementation, the state observer can then be used in various ways to complement the control system. This might allow the dynamics of the process to be mastered, which can minimize both NO_x and ammonia emissions during dynamic engine operation. Classic SCR control through feed-forward controllers can under certain conditions perform rather well. However, during dynamic situations, such as hefty engine load alteration, studies have implied that these types of controllers may not perform well due to the substantial changes in temperature, level of NO_x and exhaust gas flow rate these dynamic situations cause. The consequences of the controllers not handling these dynamic situations well, may have severe impacts on the NO_x and/or the ammonia emissions during and for some time after these dynamic situations have ended. (Hsieh & Wang, 2014, p. 441)

By designing a control system with the aim of controlling the ammonia stored in the catalyst elements, process control can be improved during dynamic engine operation. This type of SCR-control approach is called ammonia storage distribution control (ASDC). ASDC focuses on controlling the process based on the calculated amount of ammonia stored in the reactor rather than controlling the process based on the measured level of NO_x in the incoming exhaust gas. The goal is to have a high amount of ammonia stored in catalyst elements in the upstream part of the reactor for efficient NO_x reduction and to have lean ammonia storage in the catalyst elements in the downstream part of the reactor, this in order to store the excess ammonia. A simplified schematic representation of this can be seen in figure 4 below. (Personal communication with Wärtsilä employee; Hsieh & Wang, 2014, p. 441-442)

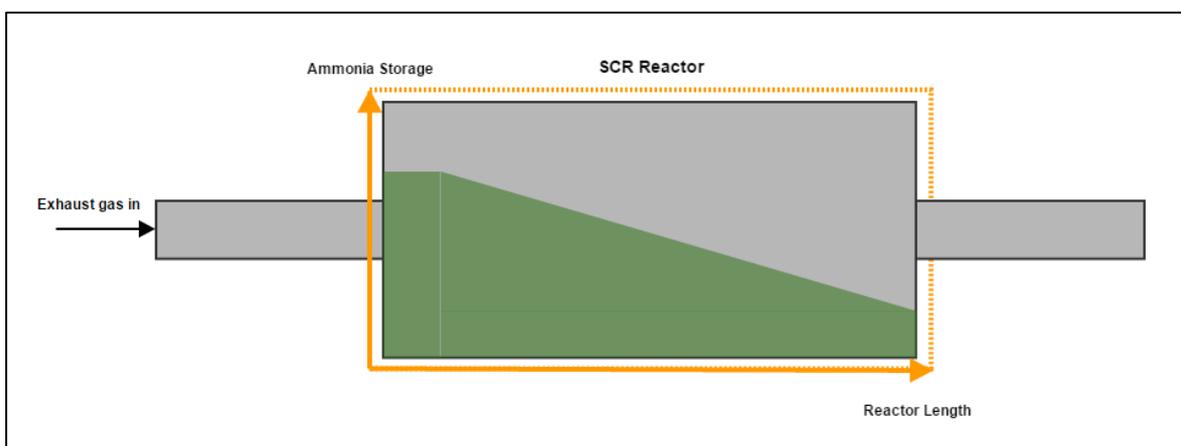


Figure 4. Schematic description of ammonia storage distribution control. (Hsieh & Wang, 2014, p. 442)

This control method approach requires that the process and the reactor are divided into several different models. Alternatively, the process and the reactor can also be treated as a single model split up in several different sections, where the outgoing NO_x and NH₃ from one section function as the incoming NO_x and NH₃ to the next section. Regardless of method, the goal is to primarily reduce NO_x in the upstream part of the reactor and to store ammonia slip in the downstream part of the reactor. (Personal communication with Wärtsilä employee; Hsieh & Wang, 2014, p.443)

2.3 Object oriented programming

Object oriented programming (OOP) is a programming paradigm that focuses on viewing and treating programming concepts as independent software structures called objects rather than as processes. The globally most popular, and used programming languages today include several object oriented languages such as Java, C++, C#, Python and Ruby. In other words, OOP is today one of the most popular and widespread programming paradigms. A general perception about OOP among professional programmers is that OOP to some extent resembles the way humans view and solve real-world problems. (Cass & Diakopoulos, 2015; Sintès, 2002, p.11)

The definition of an object is an instance of a class that consists of internal variables (known as attributes) and functions (known as methods). The attributes are the characteristics of the class and can either be specified for every specific object created through input when initialized, or be a constant of the class, and thus also constant for every specific object created. Methods are the internal functions of the class and define the specific behavior of each object. They can perform various types of operations: calculate internal variables, return variable or attribute values or perform various actions on itself and other objects. (Sintès, 2002, p. 12-13)

Figure 5 below contains a code example of an object written in Python. This class is simple and defines a car. The class consists of two methods that define its behavior. The first method initializes its internal attributes when a specific instance or object of this type is created. Two of the attributes are given as input parameters when a specific instance of the object is created: make and model. This method also has a third attribute that defines the number of wheels. This attribute is not given as an input parameter since all cars always have four wheels, in comparison to the two other attributes that vary from one car to

another. Note that the argument *self* can be found as an input parameter to the two methods and when attributes are given their respective value. This parameter only specifies that the attribute or method is an internal variable.

```

class car(object):
    def __init__(self, make, model):
        self.make=make
        self.model=model
        self.number_of_wheels=4
    def return_attributes(self):
        return self.make, self.model

car1 = car('Mercedes', 'C')
car2 = car('Ford', 'Escort')

# Calls upon the return_attributes method for each of the two car objects.
make1, model1 = car1.return_attributes()
make2, model2 = car2.return_attributes()

```

Figure 5. Code example 1.

The second method returns, when called upon the text strings that were given as inputs when the object was created. As can be seen in the last line of the figure, the return attribute method of the object *car2* is called. The method will return the text strings to the variables *make2* and *model2*. The variable *make2* will after this, contain the string *Ford* and the variable *model2* will contain the string *Escort*.

By programming with objects, the programmer is allowed to work with concrete and understandable terminology of the programmer's own choice. Because of this, more natural software can be created as the programmer can focus on functionality rather than on the implementation itself and/or on its syntax. OOP also makes it possible to isolate specific parts of the code and work with them individually without making changes in other parts of the code. This allows the programmer to focus on smaller bits of code at a time, and will also allow the programmer to test and validate these bits of the code individually. This will then in turn tend to create more reliable code, as bugs and syntax errors are more easily found and corrected. (Sintes, 2002, p. 18)

2.4 Python

Python is an open source, high-level, object-oriented, general purpose programming language. Python is today one of the most popular programming languages in the world and is used in all various types of applications such as GUI development and data analysis. Python focuses on high readability and simple syntax in order to allow the programmer to focus on solving problems rather than on coding. Python also supports integration with external functionality coded in other languages such as C and multi paradigm functionality. (Lutz, 2011, p. xxxix; Zandbergen, 2013, p. 3-4)

Python is an interpreted programming language, as opposed to other popular programming languages such as C and C++, which are compiled languages. The difference between compiled languages and interpreted programming languages is that interpreted languages are parsed line by line by an interpreter, while code written in a compiled language will require a compiler to build the code before the code can be executed. As Python is an interpreted programming language, code execution will be a bit slower than code written in a compiled language. (Lutz, 2011, p. xxxix)

2.4.1 Typing system and syntax

Python is a dynamically typed programming language, which means that the programmer is not forced to declare data types for every variable or constant created. Instead the data type will be automatically determined by the Python interpreter for every variable and constant upon creation. Python also allows variables to change their data types throughout the code. An example of this can be seen in figure 6 below where the variable *x* at first will be of the data type *string*. When given a new value, the data type of the variable *x* will change from *string* to *integer*. (The Python Guru, 2015)

```
x = 'Hello World!'      # Creates the variable x and gives it a text string.
type(x)                # Checks the data type of the variable x.
>> str                 # Data type = string.

x = 10                  # Gives x a new value.
type(x)                # Checks the data type of the variable x.
>> int                 # Data type = integer.
```

Figure 6. Code example 2.

As previously mentioned, Python focuses on high-readability. This is realized through Python's simple, yet powerful syntax. A program written in Python is divided into so-called *logical lines* which normally take place within the boundaries of one physical line of the code. In some cases, a logical line may take place on several physical lines of the code provided that the syntax allows line joining through backslash-token (\). Python utilizes indentation through spaces and tabs at the beginning of a logical line to differentiate blocks of code from other code blocks. The level of indentation is determined by the number of spaces between the beginning of the physical line and the first non-blank character on said line. Other programming languages, such as C, utilize brackets to do this. Note that Python will not require the end of an expression to be terminated with a semicolon, as is very common in other programming languages. (Python, 2016a)

The names of variables and constants can in general be chosen freely by the programmer, provided that the programmer does not choose the same name for a variable that is reserved for any of the so-called keywords that Python uses. The keywords are special identifiers that Python has reserved for its own syntax. These keywords include, among others, *and*, *while*, *for*, *class*, *break*, *not*, *from*, *def*, *try*, *except*, *if* and *return*. (Python, 2016a)

2.4.2 Libraries and modules

One of Python's greatest features is its large standard library. The standard library consists of modules both written in C and Python. The modules written in C are often modules that give the programmer access to system functionality, such as file I/O, which would otherwise be unavailable. The modules that are written in Python generally include functionalities for solving standardized problems that Python programmers may encounter in their everyday work. The modules included in the standard library are among others, modules that consist of mathematical functions, modules for online code debugging, modules for file reading and data logging. (Python, 2016c)

Thanks to a large community, and the general open source consensus, Python also has a large amount of user developed modules and function libraries, complementing the standard library. These are commonly referred to as *packages*. Similar to the modules in the standard library, these available packages might be written in C or directly in Python. These packages are often specifically created for certain tasks and problems. An example

of a package used in this thesis project is the *PyModbus*-package, which is a Python implementation of the communication protocol Modbus. (Python, 2016c)

2.5 Raspberry Pi

Raspberry Pi is a series of credit card sized single-board computers, developed and manufactured by The Raspberry Pi Foundation. The Raspberry Pi was originally intended and developed to be a cheap device that could be used in educational projects and to promote basic computer science and programming. Today, the Raspberry Pi can be used in all types of professional and leisure projects. (Halfacree & Upton, 2014, p. 1-9)

2.5.1 Hardware

The Raspberry Pi comes in three different models with different hardware and computing power; A, B and B+. As well as different models, different revisions of all three models featuring updated hardware have also been released. All of the three models share the same *system-on-chip* (SoC) module; Broadcom's BCM-2835. This integrated circuit (IC) integrates a computer's basic components into a single chip and provides the computer with its general-purpose computing, graphics and IO functionalities. All models also share a number of interface ports, such as USB for connecting keyboards and mice to the Pi, HDMI for video and audio output, and general-purpose IO pins. (Halfacree & Upton, 2014, p. 13-16)

Model A is the series' budget model and the least powerful model. Model A has a memory of 256 MB and a 700 MHz CPU. Model B shares the same CPU with model A and has a memory of 512 MB. Model B also includes two USB ports and an Ethernet port which allows the Pi to be connected to a local network and the internet. Model B+ is the latest model of the Pi series and comes in two different revisions; 1 and 2. The Raspberry Pi 2 B+ is the most powerful model of the current Pi models available and is illustrated in figure 7 below. This model has a total memory of 1 GB and has a 900 MHz quad core ARM Cortex -A7 CPU. Exactly like model B, the Pi 2 B+ also include four USB ports as well as an Ethernet port. (Halfacree & Upton, 2014, p. 15-18)

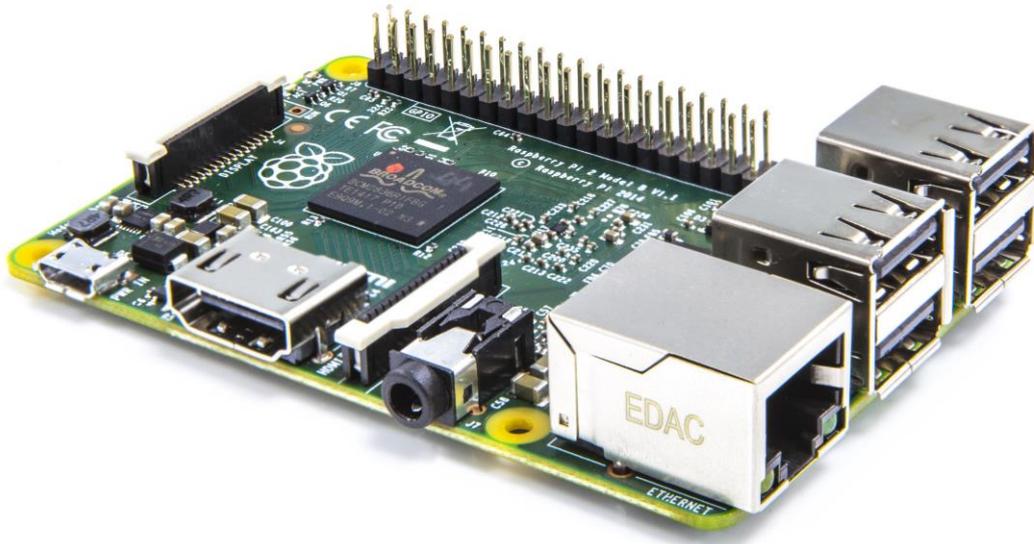


Figure 7. The Raspberry Pi 2 B+. (Upton, 2015)

None of the Raspberry Pi models are equipped with storage devices. In order to make an operating system run on the Pi an external storing device is needed. All of the models are equipped with an SD-card reader which allows the Pi to store and read data from an SD-card. SD-cards are solid-state storage systems that are often found in digital cameras and other small devices where there is a need to store data for longer periods of time. (Halfacree & Upton, 2014, p. 29)

2.5.2 Raspbian

A Raspberry Pi does not generally come with a standard operating system installed. Instead, users must download and install the operating system of choice on the Pi's SD-card. Even though there is no standard operating system, it is however designed to run operating systems based on GNU/Linux. At this moment there is a number of Linux distributions that have been ported or created specifically for the Pi. These include, among others, Raspbian, Pidora and Arch Linux. (Halfacree & Upton, 2014, p. 20-21)

Raspbian is an implementation of the Linux distribution Debian and is today the one of the most popular operating systems for Raspberry Pi. Alike other Linux distributions, Raspbian is a free and open-source operating system, created and optimized to run on a Raspberry Pi. Raspbian comes with a great number of software and packages pre-installed and ready to use at first startup. Pre-installed packages and software include, among others, educational packages, graphical packages and Raspbian's own Python development

environment IDLE. Raspbian's user interface resembles other Debian-based Linux distributions. The desktop environment of Raspbian is known as *Lightweight X11 Desktop* (LXDE) and is a simple point-and-click interface similar to Microsoft Windows and Apple's OS X. (Halfacree & Upton, 2014, p. 46-49)

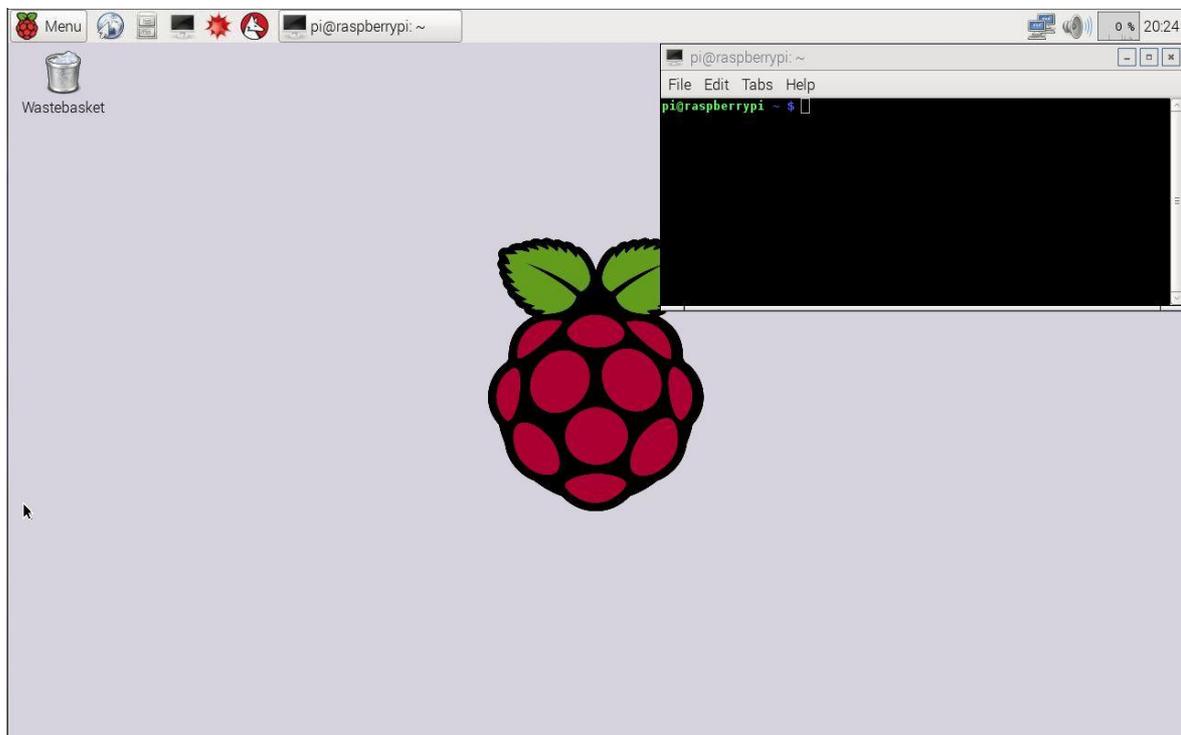


Figure 8. Raspbian user interface.

3 Project realization

This chapter and its subchapters cover the actual work done in the thesis project as well as the methods used and approaches taken, with emphasis on the former. The different approaches going into the different phases of the project are defined, explained and motivated thoroughly in the first subchapter. The following subchapter covers the results from both the actual work and the various tests that were conducted throughout this thesis project. The final subchapter describes the optimization of the application and the optimization results.

3.1 Methods and approach

The initial work done on behalf of this project consisted mainly of studying background material regarding modeling the SCR-process, the actual SCR-model created by Catalyst Systems R&D and the programming language Python. General SCR-technology was not studied; essential knowledge had been acquired through working with SCR-systems before this project started. The practical work in this project started when the initial work and the study of background information had been completed. The practical work of this thesis project can be condensed into three stages:

- Implementation development and functionality validation
- Real SCR tests and result analysis
- Optimization of the implementation

Implementation development and functionality validation differs from the other two stages, as this stage more or less spanned over the whole project process, while the other two were smaller parts of the project, spanning over shorter timeframes. The three stages are more described in-depth in the following subchapters.

3.1.1 Implementation development

An implementation plan was created before the actual programming began. In the plan, the implementation development process was divided into the eight stages below:

1. Translate the original model from Matlab code to Python.
2. Implement the model in Python.
3. Implement, test and validate Modbus communication between the Pi and the PLC.
4. Validate functionality through online tests versus the Matlab application.
5. Implement communication diagnostics functionality.
6. Implement simulation logic and data validation.
7. Implement other secondary functionality.
8. Expand implementation to support up to twelve individual SCR-systems.

Naturally, a modular approach was taken, utilizing the advantages of programming with an OOP language. The plan was to divide the code into smaller bits, so that focus would lie on individual parts of the code instead of on the whole functionality. This approach was considered to be a good way of working on this project, as focusing on individual parts of the code was deemed to ensure a reliable and well-functioning implementation. Troubleshooting would also be easier to conduct, which would both save time and further enhance the reliability of the application.

Translating the model from Matlab code and implementing it in Python was the first step in the process. The basic Modbus communication was implemented after the model had been implemented. The reason for implementing communication functionality before validating the functionality of the model was that online tests versus Matlab using data loggers were desired. Online tests versus the original Matlab application were seen as a good option to validate the mathematical functionality of the Python model as graphs and figures of different process variables could be created and analyzed. As a matter of course, the first three steps of the process were repeated numerous of times during the first two months of this project as various problems appeared.

3.1.2 Real SCR tests and result analysis

The implementation was tested using a real Wärtsilä CSO-system in Bermeo, Spain. The tests focused on testing the interaction of the implementation together with the PLC and on simulating a real process during dynamic engine operation. The interaction tests were conducted to validate that the implementation was stable and safe to use with a real SCR-system. Simulation tests were conducted to investigate whether the implementation could improve the process control during dynamic engine operation in comparison to conventional feed-forward control. Both simulation data and real process data were logged during the simulation tests. The logged data was then used to analyze the interaction of the implementation with the process and to determine the impact of the model on the process control during dynamic engine operation.

3.1.3 Implementation optimization

Optimizing the code was the last stage of the thesis project. When the implementation had been successfully created in Python, tested and validated, it was considered to be too slow for unknown reasons. Wärtsilä had set a limit of the total simulation time to 50 milliseconds when simulating twelve SCR-systems. The implementation at this time was running at around 20 milliseconds per SCR, which would, theoretically, amount a total iteration time of about 240 milliseconds, while simulating twelve SCR-systems. In other words, the code needed to be optimized so that it would comply with the set benchmark. A so-called code profiler was used to test the code and generate reports of time spent in various functions, operations and methods. The reports were then used to find bottlenecks, and correct flaws and thus optimize the code by a significant margin.

3.2 Model implementation

This subchapter and its subsections describe the implementation, its architecture and its functionality as it was created. Subchapter *3.2.1 Code architecture* defines how the implementation was put together on a macro-level, while subchapters *3.2.2 General functionality* and *3.2.3 Program flow and simulation logic* describe the functionality more in depth. Subchapter *3.2.4 Modbus communication* describes how the communication between the Pi and the PLC was realized and how communication diagnostics was implemented.

3.2.1 Code architecture

The application is based on one class and six methods. The functions are the general-purpose operations of the implementation, mainly comprising necessary auxiliary functionality. It is through the functions that the implementation gains the fundamental ability to interact with the SCR-process in a reliable way. These are mainly used and called from the outside of the actual simulation and calculation functionality of the implementation. The functions, their input and output data are listed in the table below:

Function name	Input data	Output data
Main	Number of SCRs	Simulation step
Global_constants	-	Global constants
Variable_initialization	-	Global variables
Modbus_initialization	PLC IP-address	-
Modbus_handshake	-	Connection status
Reset	-	-

Table 3. Functions and corresponding input and output data.

The function *main* contains the actual *while true*-loop. This function is used to initialize the SCR-objects and to call the other functions and methods according to stated simulation logic in the while-loop. The functions *Global_constants* and *Variable_initialization* are only called once as they define and initialize global constants such as SCR-specific Modbus addresses and various variables. The function *Modbus_initialization* is used to initialize the connection to the PLC. *Modbus_handshake* is a function used to diagnose the communication between the Pi and the PLC. *Reset* is a function that is only called under the circumstance that the communication has failed between the Pi and the PLC and re-initialization has failed. This function restarts the whole application in order to try to re-initialize the Modbus connection.

The class defines an SCR-object and consists of six methods and a number of internal attributes that define its behavior. As real SCR-systems share the same behavior and are non-reliant on other SCRs, it was considered natural to treat individual SCR-processes as individual objects in the code as well. Dissimilarities such as Modbus addresses for fetching SCR-specific process values, are specified through input parameters when a specific object is created. The six methods of the SCR class and their corresponding input and output data are found in the table below.

Method name	Input data	Output data
<code>__init__</code>	SCR-specific Modbus addresses	-
<code>Read_Modbus</code>	-	SCR-specific process data
<code>Write_Modbus</code>	SCR-specific simulation data, data validity	-
<code>Write_Modbus_zero</code>	-	-
<code>F_model</code>	SCR-specific process data, simulation step	SCR-specific simulation data
<code>Current_theta_sp</code>	SCR-specific process data	Ammonia coverage set point

Table 4. The methods of the SCR-class and their corresponding input and output data.

The method `__init__` is the constructor method of the class. This method defines and initializes an object's specific attributes when an object is created through input parameters and local constants. The `Read_Modbus` method is used to fetch SCR-specific process data from the PLC. `Write_Modbus` and `Write_Modbus_zero` are methods used to write simulated data to the PLC. `Write_Modbus` is used to write simulated data to the PLC, provided the data is considered valid. `Write_Modbus_zero` is used to flush the PLC's Modbus registers from data, should the data to be considered as non-valid, or, if an SCR goes into failure mode or is otherwise shut down.

The methods `F_model` and `Current_theta_sp` are perhaps the most interesting methods as they contain the actual SCR-simulation functionality and process control functionality of the implementation. The main purpose of `F_model` is to simulate the process, calculate outgoing NO_x and NH₃ and most importantly, calculate the current ammonia storage profile of the SCR. `Current_theta_sp` can be seen as a sub-method of `F_model`, as it is the only method that calls for this method. `Current_theta_sp` is used to calculate the set point for the ammonia storage distribution, based on both real SCR data and simulated data. The `Current_theta_sp`-method can be utilized to calculate both a set point for the upstream part of the reactor as well as a set point for the downstream part of the reactor.

The simulation itself, and the main functionality of the implementation, are in other words found in the objects themselves. Therefore, most of the execution time will be spent inside the methods of the objects, provided that the implementation is running normally and that the Pi can communicate with the PLC. The functions are only to be used under certain circumstances, except for the main-function which contains the actual while-loop and the simulation logic.

3.2.2 General functionality

When the application is started, it will first initialize and define various global constants, variables, as well as the Modbus connection to the PLC. Before it proceeds to the simulation loop it will diagnose and validate that the Modbus connection between the PLC and the Pi has been successfully established. It will also check that the communication is functioning properly by trying to do a *communication handshake* with the PLC. If the connection is deemed valid and functional, it will proceed to the simulation loop. The figure below illustrates this chain of events.

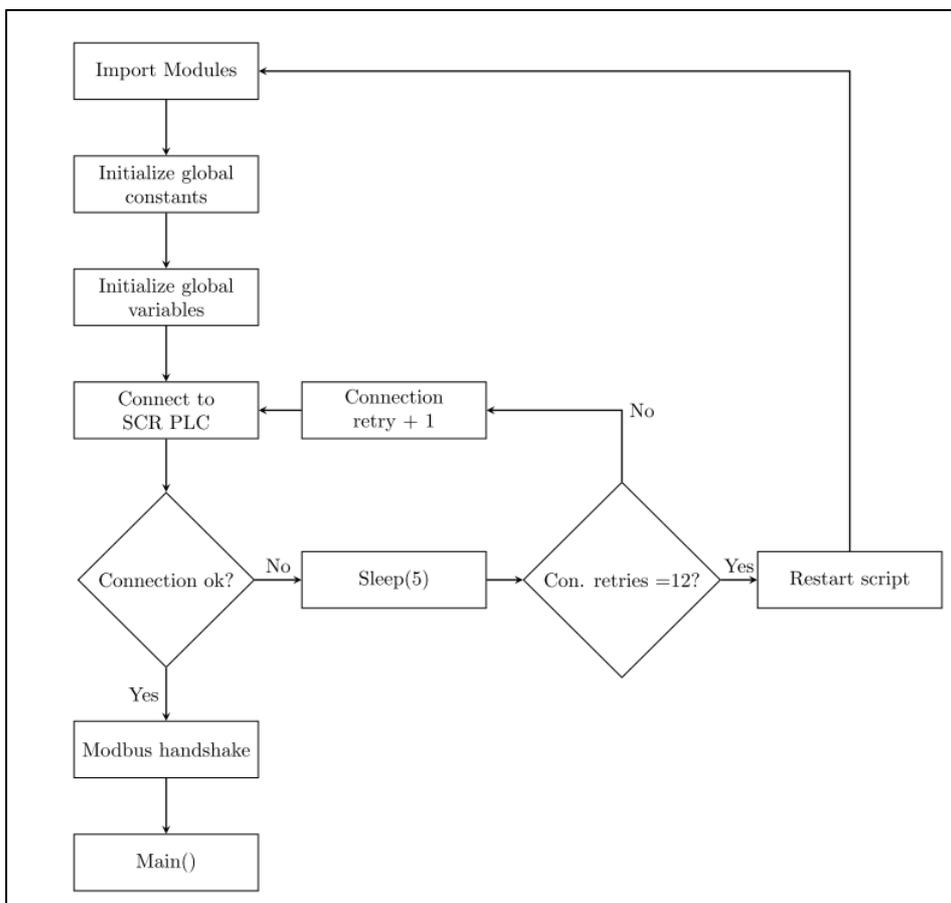


Figure 9. Script start up and initialization logic.

However, in case the Modbus connection is not deemed as valid, the application will start a re-initialization loop. When trying to re-initialize the Modbus connection it will at first pause for five seconds (*sleep*) and then check how many times the script has tried to re-initialize the connection. If it has tried to re-initialize the connection less than twelve times, it will call itself in order to try again. However, if the script has tried to re-initialize the connection more than twelve times, it will restart the whole script, since simulated data are considered to be non-valid due to the time lost trying to re-initialize the connection during

the time the script has tried to re-initialize the connection. Naturally, if the script is still unable to initialize a Modbus connection to the PLC, it will be stuck in this loop until a connection has been established. By letting the application pause for five seconds, other network components are given some time to resolve possible problems, which might have caused the Modbus connection to fail. In addition, this will ensure that the network is not overburdened with connection requests.

3.2.3 Program flow and simulation logic

The simulation logic is stated in the while-loop found in the function *main*. In addition to the while-loop, this function also contains the functionality that initializes the specific SCR-objects themselves. The function will at first, when called upon initialize n SCR-objects, where n is the value stored in the global constant *Number_of_Engines*. In other words, *Number_of_Engines* determines the number of SCRs that are to be simulated. The application supports simulation of up to twelve SCRs. After initializing the SCR-objects, the application will move on, registering various time stamps, which can be seen in the table below.

Time stamp variable	Description
Time_stamp_SCRn	Determine how long an SCR-system has been running in normal mode. Used to validate data. SCR specific.
Simulation_time_stamp	Calculate the simulation time step.
Modbus_RW_time_stamp	Determine when the application is allowed to fetch data from the PLC and to write data to the PLC.
Modbus_handshake_time_stamp	Determine when the application is allowed to do a handshake with the PLC.

Table 5. Time stamp variable descriptions.

After registering various time stamps, the application will proceed to enter the actual *while true*-loop, which is the main event loop of the application. In this loop, all simulation, read/write and handshake logic is stated. It is also from this loop most function and method calls are made. Calculating the simulation step is the first thing that happens inside the loop. The simulation step variable is the time step that describes the amount of time spent since the last simulation took place. This variable is crucial to the functionality of the simulation algorithm and is given as an input to the simulation method. It is then used in the internal calculations, as the time aspect must be taken into consideration when simulating the process.

After calculating the simulation step, the application will move on to executing the simulation logic for SCR1 by checking if it is possible to read and write data. As previously mentioned, this is determined by a time stamp variable. When entering the loop for the first time, this is automatically allowed. In case the application does not allow data to be read from the PLC, it will continue to check in what mode the SCR1 is running in. In case the SCR is running in automatic mode (=8), the application will simulate the SCR in question with data read from the last iteration where it to read and wrote data. If not, the application will continue onto the next SCR and repeat the same logic. However, if the application is allowed to read and write data, it will move on to read data from the PLC and check in what mode SCR1 is running in. In case SCR1 is not running in automatic mode, it will proceed to the next SCR. If it is running in automatic mode, it will proceed to simulate SCR1 with the data it just read. After simulating SCR1, it will check if the data is valid. Data validity is determined by an SCR-specific time stamp variable that keeps track of how long an SCR-system has been running in automatic mode, thus simulated. If the data is considered to be valid, the application will proceed to write the data to the PLC and then proceed to the next SCR. If the data is not considered to be valid, it will just proceed to the next SCR. The figure below is a flowchart describing the simulation logic.

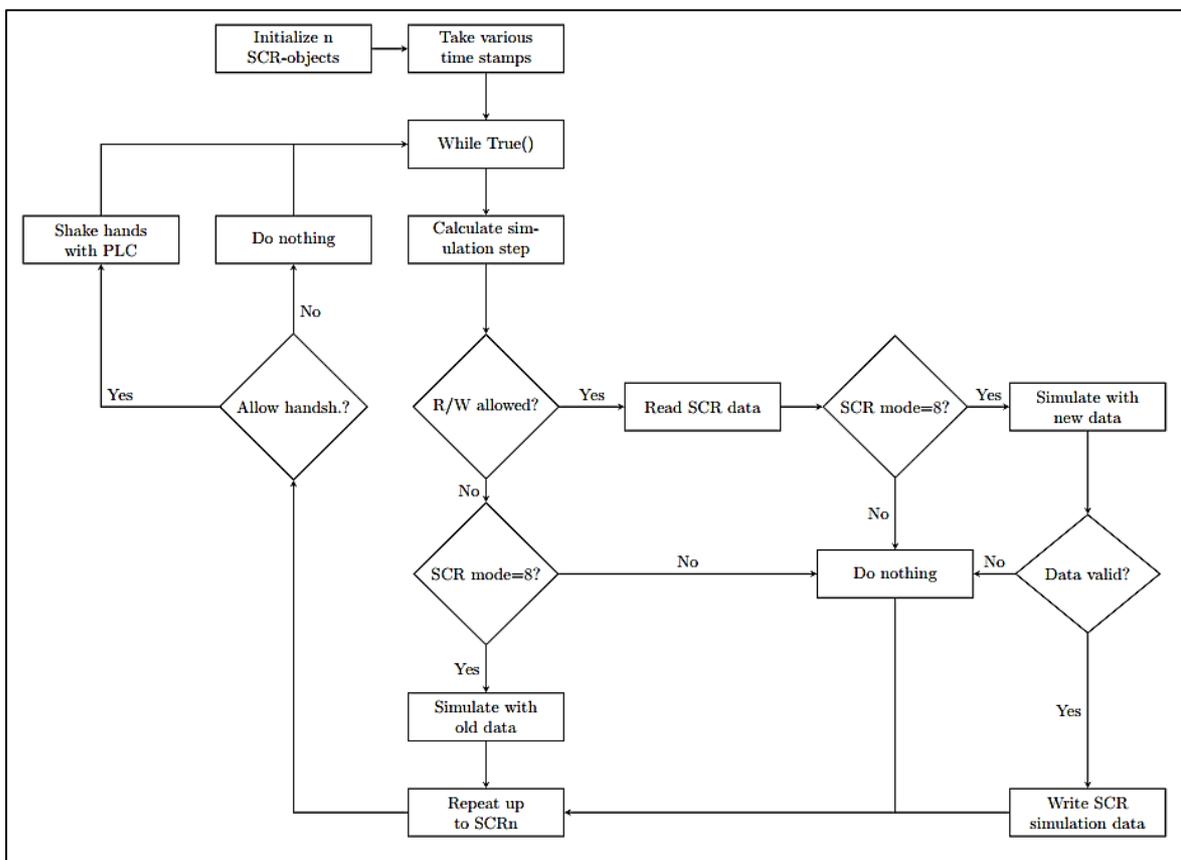


Figure 10. Simulation logic. N corresponds to the number of SCRs simulated.

The application will execute the exact same logic for every SCR simulated, which is stated between the *Calculate simulation step-* and the *Repeat up to SCRn*-boxes in the flowchart in figure 10. When this logic has been executed for every SCR, the application will proceed to check if it is allowed to do a handshake with the PLC. If a handshake is allowed, it will proceed to shake hands with the PLC and then complete the loop. If a handshake is not allowed, it will complete the loop. Having completed the loop, the application will restart the same chain of events and execute the same logic for an infinite number of iterations or until manually stopped. As the application checks the run mode for every SCR, regardless of running mode, it will always detect if an SCR goes into failure mode or returns from failure mode. This makes the application more flexible, as it will not simulate SCRs running in other modes than automatic mode and thus freeing up computing power and speed for the other SCRs. At the same time, the application is also able to re-initialize simulation of an SCR returning from failure or shut down mode.

3.2.4 Modbus communication

The Modbus communication was realized through the Python implementation of Modbus TCP/IP, called *PyModbus*. The communication functionality of the application is found both inside the SCR class and in two of the general purpose functions. The connection to the PLC is initialized in the function *Modbus_Initialization*. This function does not take any input parameters, nor does it return any output data. The purpose of this function is only to initialize the connection when called upon, and re-initialize it in case the connection has failed. The PLC's IP-address is given as a text string when initializing the connection. This variable together with the global constant *Number_of_Engines* are the only constants that needs to be specified by a user before taking the whole implementation into use.

The other function of the implementation that contains communication functionality is the *Modbus_handshake*-function. This function is used to diagnose the established connection between the Raspberry and the PLC by sending so called *acknowledgement numbers* back and forth. The fundamental function is simple: the PLC writes a number between 0 and 1000 to the Pi, which then in turn takes the incoming number, increments it by one and then returns it to the PLC. The PLC will then once again, increment the number by one and return it to the Pi. Both the PLC and the Pi store the incoming value from the previous loop and compares it with the current incoming value. In case the new incoming value

differs from the old value, then the communication can be assumed to be working properly.

However, if the new incoming value is exactly the same as the old value, then it can be assumed that the communication has run into a problem, as the Modbus register has not been updated since the last handshake. The function will then try to shake hands with the PLC for ten more times. If the handshakes still fail ten times more, it assumes that the communication has failed, and proceeds to call the Modbus initialization function in order to try and re-initialize the connection. The flowchart in the figure below illustrates the functionality of the Modbus handshake function.

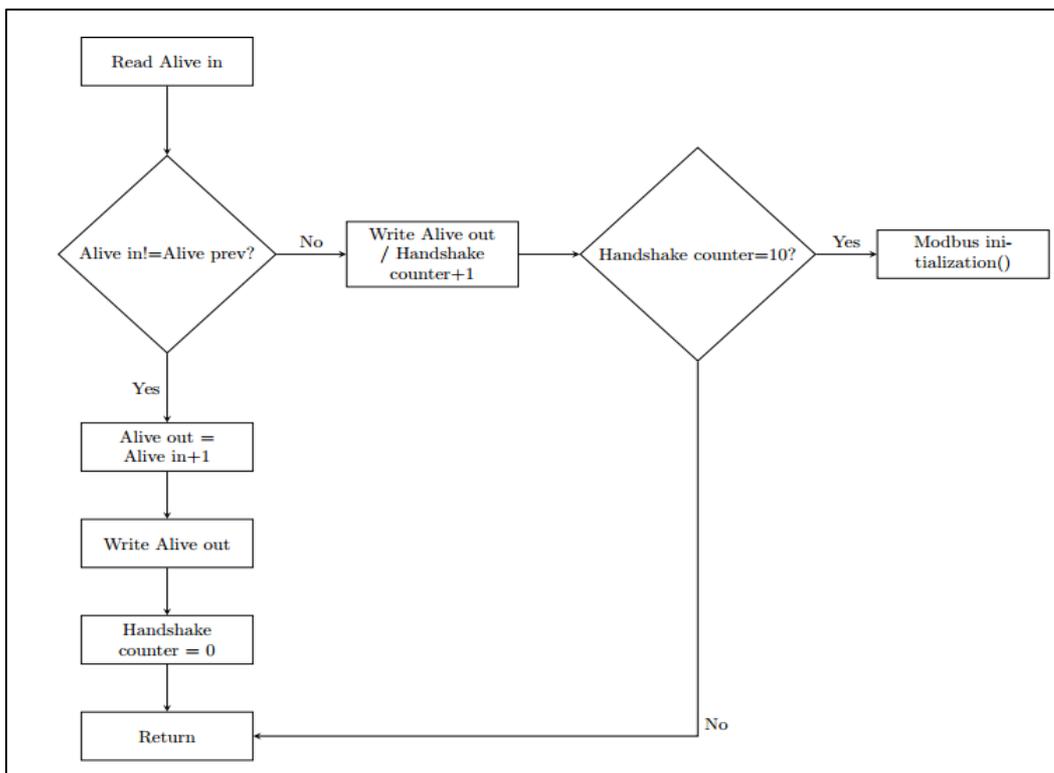


Figure 11. Modbus handshake functionality.

The exact same functionality is also implemented in the PLC. This makes it possible for the PLC to diagnose the connection between the PLC and the Pi and take necessary actions in case the communication has failed. This also makes it possible to create status signals and alarms that tell the engine operators if the communication is functioning as intended or alert them if the communication fails.

The reading of SCR-specific process data is realized through the *read_Modbus*-method, which is called once every second for every SCR simulated. When calling the *read_Modbus*-method, the SCR-specific mode status register will first be read. In case the specific SCR is running in normal mode, the application will then proceed to read the rest of the process variables for the specific SCR. If the SCR is running in normal mode, the method will only send two individual read requests to the PLC, despite requiring up to twelve process variables. This is because it is designed to speed up the code in case any of the SCRs simulated are not running in normal mode. An SCR not running in normal mode will not be simulated by the application, thus computing power for the remaining simulated SCRs running in normal mode will be freed up. This also gives the application the ability to detect whether a specific SCR goes into, or returns from shutdown mode.

The actual process variables read from the PLC are read with a single command, despite reading a total number of 16 registers from the PLC. This is realized through *PyModbus*'s functionality, which supports the reading of multiple registers simultaneously within a certain range. By doing this, the overall communication between the Pi and the PLC was to some degree optimized, as reading each register individually would require the Pi sending read-requests for every register, which would be utmost time-consuming. All process variables required by the implementation were packed together successively in order to enhance this functionality. Writing data is also done in the same fashion, i.e. all simulation data that is written to the PLC is written through a single command.

Reading data from the PLC and writing data to the PLC is allowed once per second. The simulation data and the process data will therefore be updated once per second for every SCR simulated. The Modbus handshake is also allowed once per second. The time stamps determine when one second has passed since the last time it read data from the PLC and the last time it wrote data to the PLC. During iterations where the application is not allowed to read and write data, the it will proceed to simulate with old stored data. Neither process variables nor simulation data are expected to undergo hefty changes under one second, therefore it is considered unnecessary to read or write data more frequently than once per second.

3.3 Simulation versus Matlab

Online tests were conducted together with the original Matlab application, in order to validate the functionality of the Python implementation created. The tests were conducted using a Modbus server providing input and output registers and a custom made Matlab script for data logging. The output registers contained all process variables of interest, from which the two application fetched process information. Fixed values were given to all process variables, with the exception of the urea flow measurement signal. The fixed values corresponded to real values that could appear in real situations. The intention was to only alter the value for the urea flow measurement signal as this variable alone would have a great impact on the models.

The calculated outgoing ammonia and NO_x were the variables of interest during these tests. It was assumed that the calculated outgoing ammonia and NO_x would correspond in the two applications, given that the functionality of the Python application and the Matlab application were identical. Figure 13 below illustrates the outgoing NO_x and ammonia from both the Python application and the Matlab application plotted together with the urea flow measurement signal (blue).

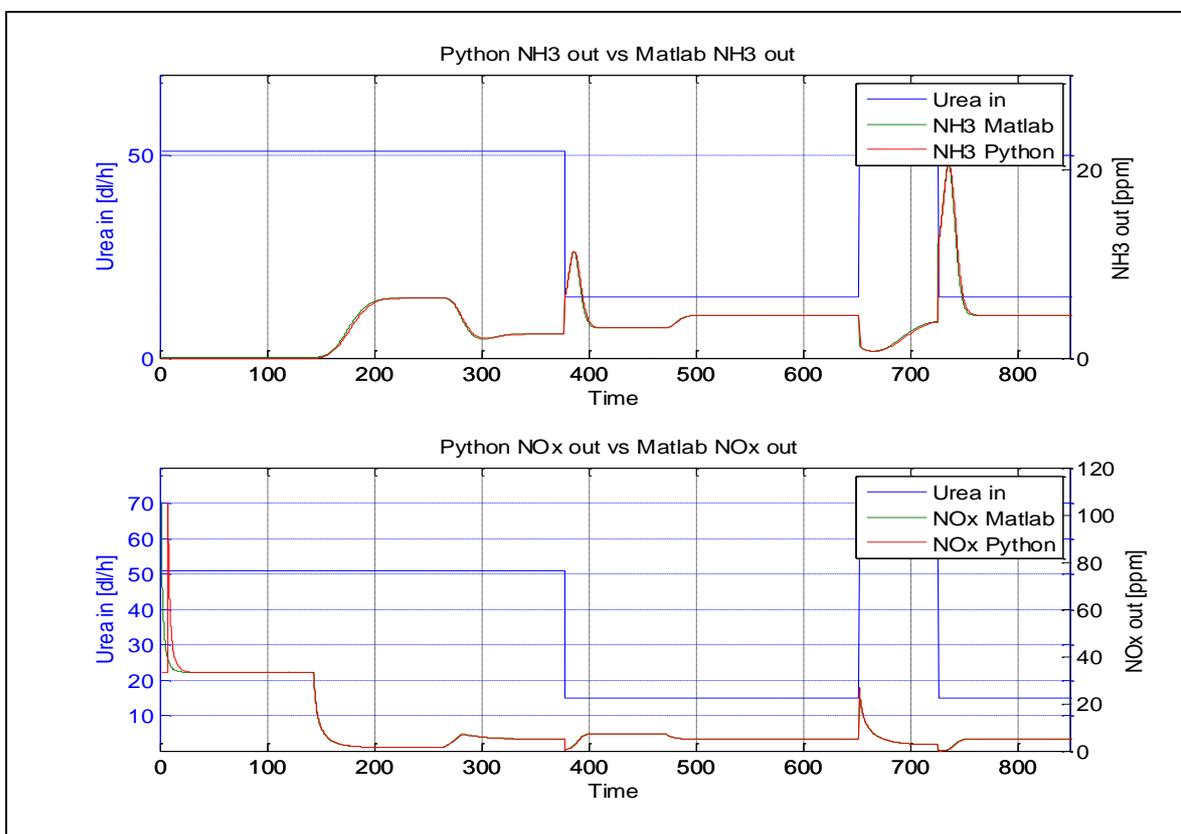


Figure 12. Calculated outgoing NO_x and ammonia by Python and Matlab.

Both the calculated outgoing NO_x and the calculated outgoing ammonia by the two applications seem to be almost overlapping each other. By examining the figure more closely however, it can be noted that there are still some differences between the two applications as the lines seem to differ from each other at certain occasions. Especially at the beginning of the figure, there seems to be a relatively great difference between the calculated outgoing NO_x -values of the two applications. Figure 14 below is created with the same data that was logged during the same tests as in figure 13 above and illustrates the mathematical differences between the two applications.

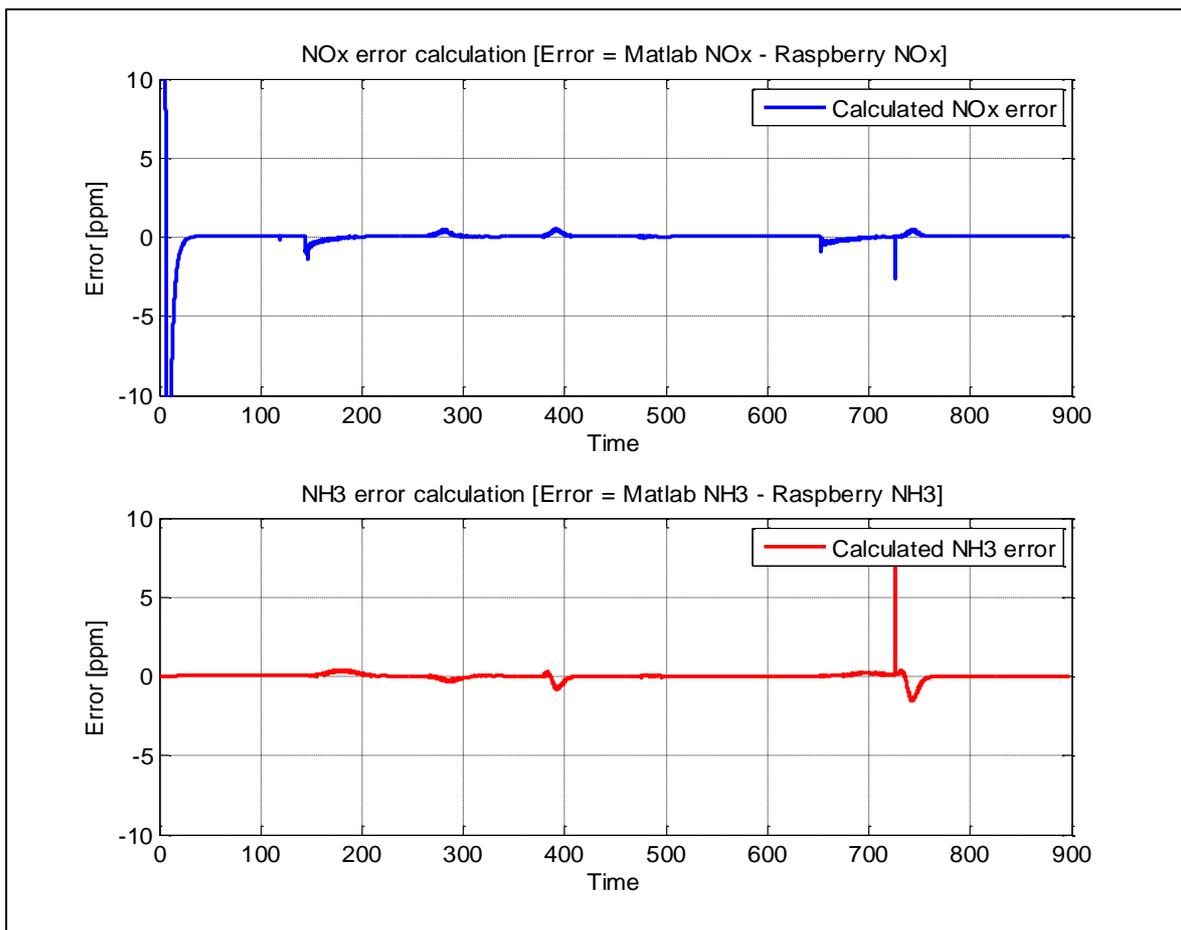


Figure 13. Calculated differences between the Matlab application and the Python application.

The overall differences between the two applications seem to be relatively small as the calculated differences between the corresponding NO_x and ammonia values are quite stable around zero. There are however situations where the corresponding NO_x and ammonia values differ substantially. Both the overall calculation differences and the sudden spikes of calculation errors can be assumed to at least partially be caused by the test set up itself. As the data logger was able to log the Matlab script's simulation data locally it was able to

register the Matlab script's calculations with a high precision, as opposed to the Python script's calculations that were logged over Modbus TCP/IP which required that the simulated data was rounded off. It can also be assumed that the computing power of the PC running the Matlab script might have had an impact on the precision of the calculations in comparison with the Pi. Based on the results from these tests, the Python model could still be considered to be functioning as intended despite the minor differences between the applications.

3.4 Real SCR tests

Real SCR tests were conducted on the implementation and on the model itself in Bermeo, Spain. The tests were mainly conducted in order to validate the functionality of the model and to ensure that the implementation could interact with a real SCR-system in a safe and reliable way. This subchapter and its subsections define, illustrate and presents how the tests were conducted and the actual results from running the implementation together with the SCR-system.

3.4.1 Test definition and setup

The tests focused on SCR- and implementation functionality during dynamic engine operation, i.e. during hefty engine load alterations. The main objective was to study the functionality of the SCR-process, and the emissions, when lowering the load from a high set point to a low set point. Previous tests had shown that the model was most effective in these situations. Nevertheless, the model was also tested on steady loads as well as in situations where the load was raised from a low set point to a high set point. Load alterations were also registered with the model not influencing the process control for reference.

The tests were conducted using a Wärtsilä 20V34SG gas engine and a full scale Wärtsilä CSO-system. As the objective was to test the SCR-system's and the SCR-model's behavior together with a running engine, the CSO-system and the SCR-model were set in automatic mode. The engine load set point was the only parameter that was altered by a human operator, all other variables and parameters were either calculated or measured from the process itself. This was considered as a good way of influencing the process, since the load alterations cause hefty changes in many of the process variables that are important in terms

of the functionality of the SCR-system. The figure below is a simplified illustration of how the system was connected with the SCR-system during the tests.

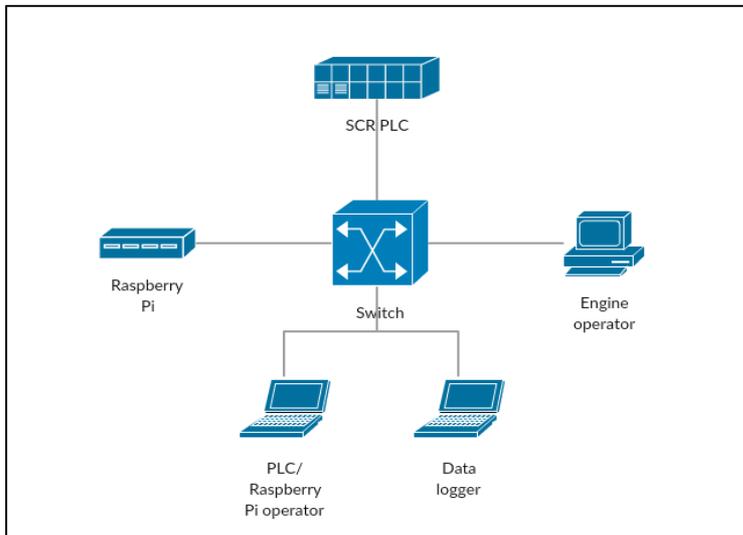


Figure 14. Test network topology.

Data logging was realized using a custom made Matlab script and a local Modbus client connected to the CSO-system's PLC. All process and simulation variables of interest were logged during the tests for result analysis. The calculated ammonia coverage set point and the simulated ammonia coverage value were the variables that were utilized to allow the model to affect the process control. This was realized by calculating the difference between the set point and the process value and a P-controller. The calculated output value from the P-controller was then added to the calculated urea flow set point of the feed forward controller in the PLC. The P-controllers gain value was set to 100 000, due to both the coverage set point and the actual coverage value being quite small. Otherwise the impact of the model on the process control would have been minimal at most, thus a high gain value was required. The figure below is a simplified illustration of the control set up.

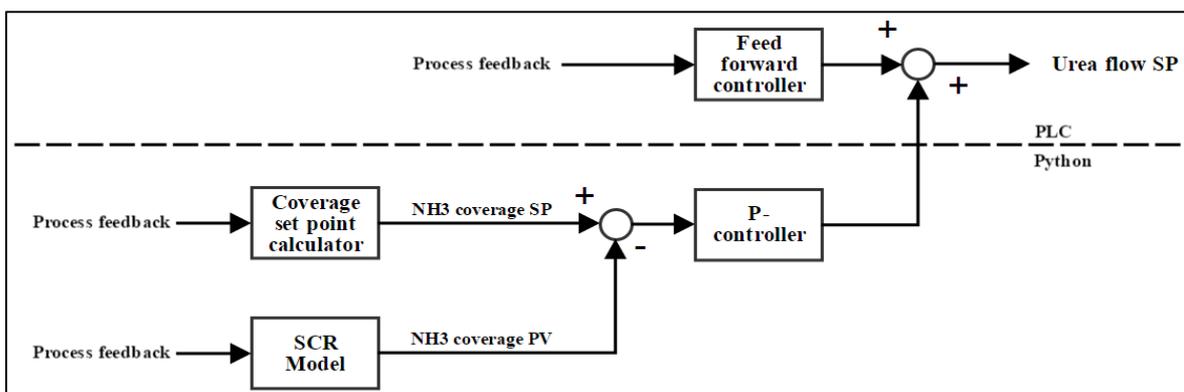


Figure 15. Real SCR test control set up.

As can be seen in the figure above, the feed forward controller that is used to calculate the urea flow set point remained in use during the tests. Therefore, full scale ammonia storage distribution oriented SCR-control tests were not conducted. Nevertheless, the SCR-model still had a great impact on the process control, as it was allowed to both raise and lower the urea flow set point.

3.4.2 Test results

The tests included running the SCR with the application both influencing and not influencing the urea flow control. The purpose behind this was to create engine load reference steps, where the functionality of the SCR-process with the model not influencing the urea control could be registered. The variables of interest during these tests were both calculated and measured outgoing NO_x and NH_3 , as well as the calculated NH_3 coverage value, NH_3 coverage set point and the difference between them. As previously mentioned, the calculated outgoing NO_x and NH_3 are mainly used to validate the functionality of the model by comparing them to the corresponding measured values. The first figure below shows the calculated NH_3 coverage value and the calculated difference between the coverage value, as well as the coverage set point, during a reference load step.

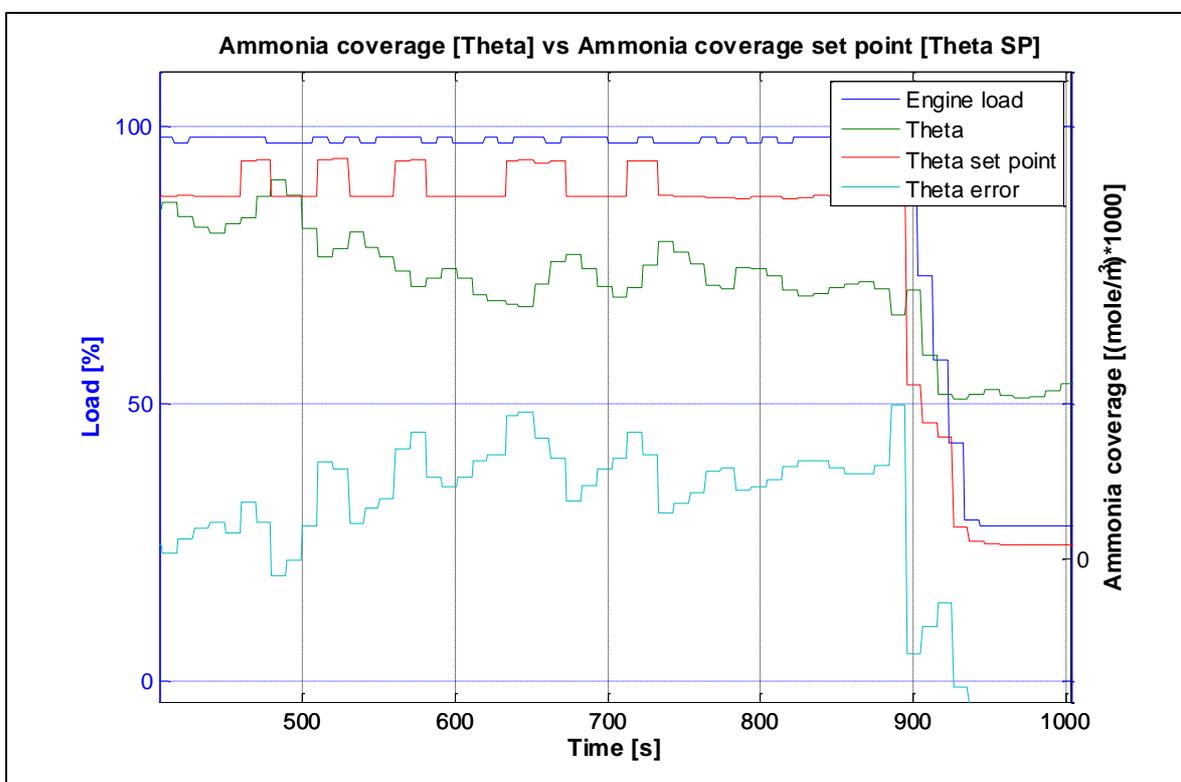


Figure 16. Calculated ammonia coverage (green) and ammonia coverage set point (red).

As can be expected, the calculated NH_3 coverage value is nowhere near or actively approaching the coverage set point, as the model is not influencing the urea control in the captured situation in the figure above. Nevertheless, the model and the application are still able to calculate the coverage value and a set point. Naturally, the model calculates a quite high coverage set point during high engine load. This due to the temperature of the incoming exhausts, the incoming NO_x and the exhaust gas mass flow are all quite high. When the engine load is lowered, all three variables will immediately start to decrease along with the coverage set point. This behavior can be observed during the moments when the engine load is decreasing. The calculated outgoing NO_x and the measured outgoing NO_x is plotted in the figure below. This figure is created using values from the same situation found in figure 16 above.

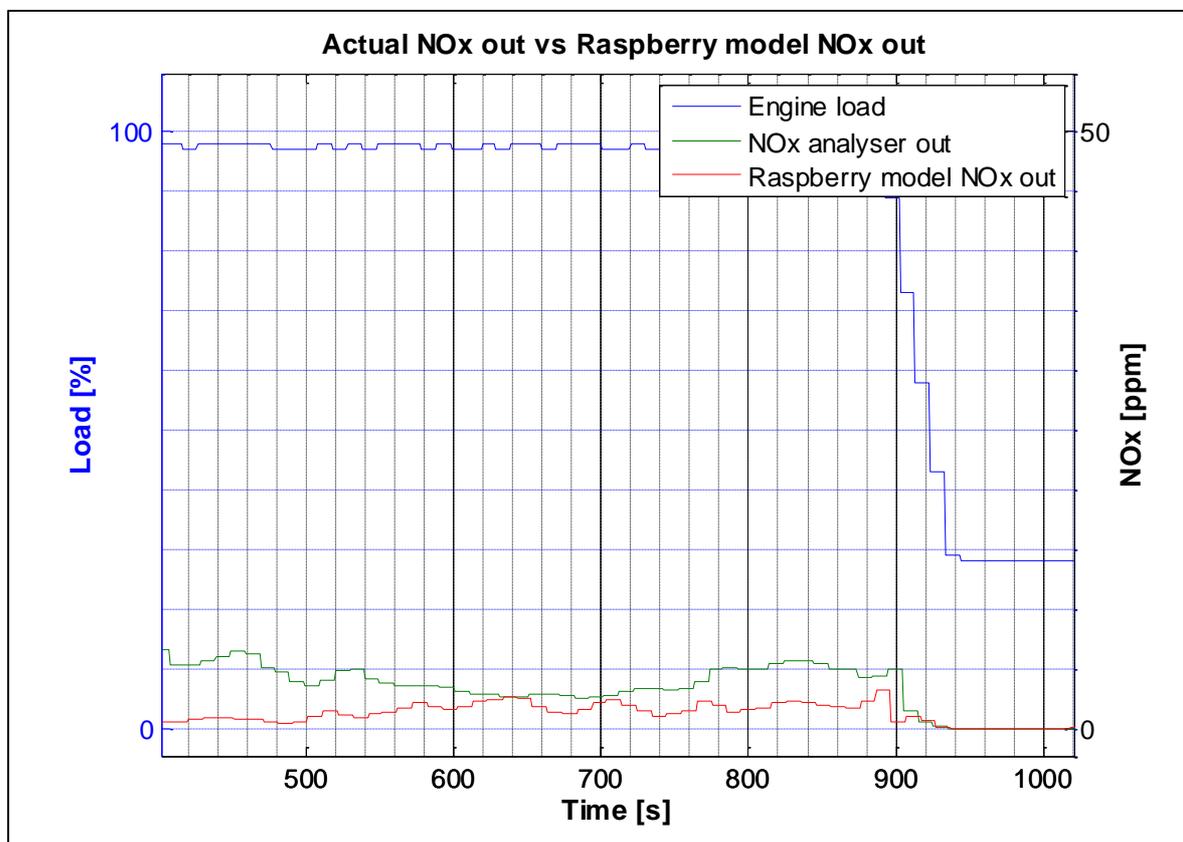


Figure 17. NO_x measured by analyzer (green) versus by model calculated NO_x (red).

The calculated outgoing NO_x is not following exactly the actual, measured outgoing NO_x , as the model seems to predict that the NO_x is reduced more effectively in comparison with the actual NO_x -reduction. The overall behavior of the calculated NO_x still follows the measured outgoing NO_x however. Thus, the calculated NO_x can be considered to be able to give a reasonable picture of the overall functionality of both the model and the process,

which to a certain degree corresponds to the actual functionality of the process. As mentioned earlier, the calculated outgoing NO_x is used for purpose of validating the functionality of the model, by pointing at how calculated values compare to actual measured variables. Thus, despite that the calculated NO_x -reduction does not corresponding directly to the actual NO_x -reduction, this does not imply that the model or the implementation are not working properly, as the calculated NO_x is not a variable used in the control aspect of the application. Figure 18 below illustrates the calculated outgoing ammonia and the measured outgoing ammonia. This figure is created using values from the same situation found in figures 16 and 17 above.

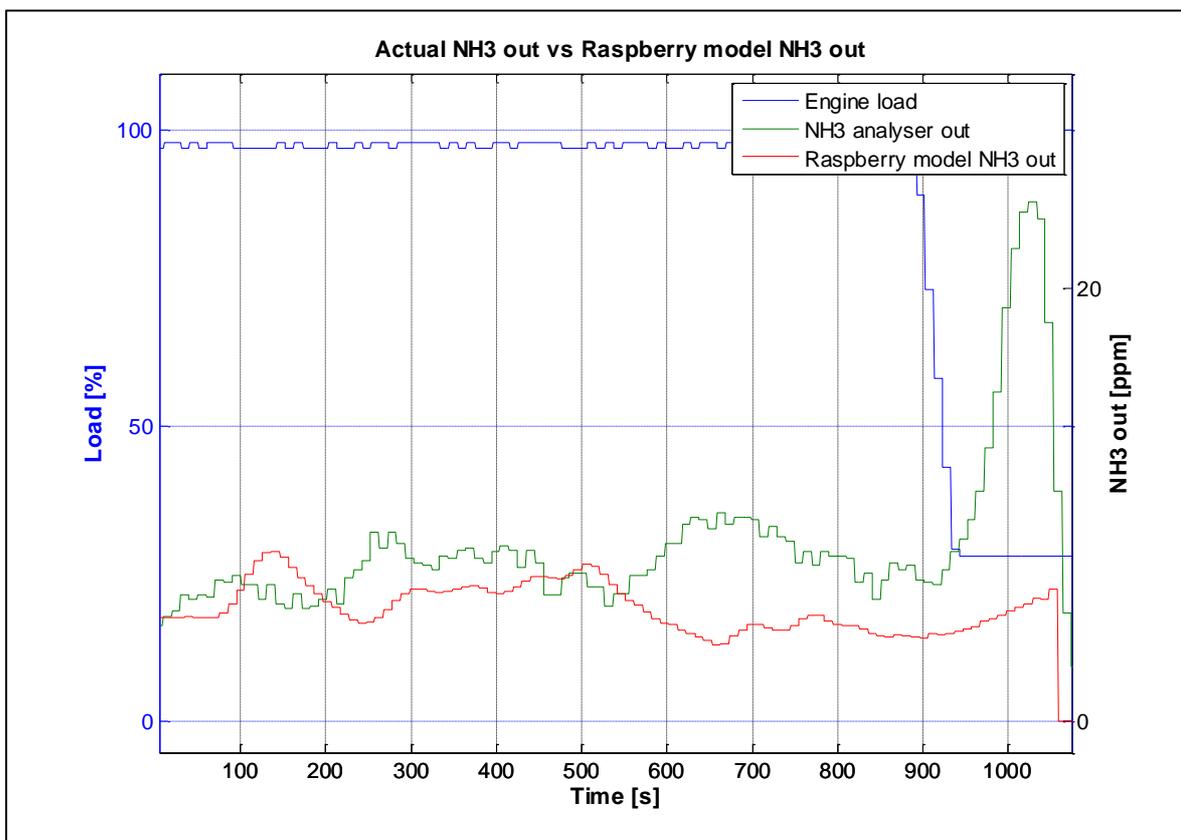


Figure 18. NH_3 measured by analyzer (green) versus by model calculated NH_3 (red).

The outgoing ammonia seems to be harder to predict in comparison to the NO_x . The overall level and value of calculated NH_3 can be considered to be relatively near the actual outgoing NH_3 during high load engine operation. The overall behavior of the calculated outgoing NH_3 does not however conform with the actual behavior of the outgoing NH_3 . The model also failed to predict the sudden spike of ammonia after the engine load step. This problem was expected as the previous tests with the corresponding Matlab application had shown this exact same behavior.

The application was allowed to influence the urea flow control directly after the system had settled, having finished the reference load step. Figure 19 below illustrates the ammonia coverage value and the coverage set point after the reference load step.

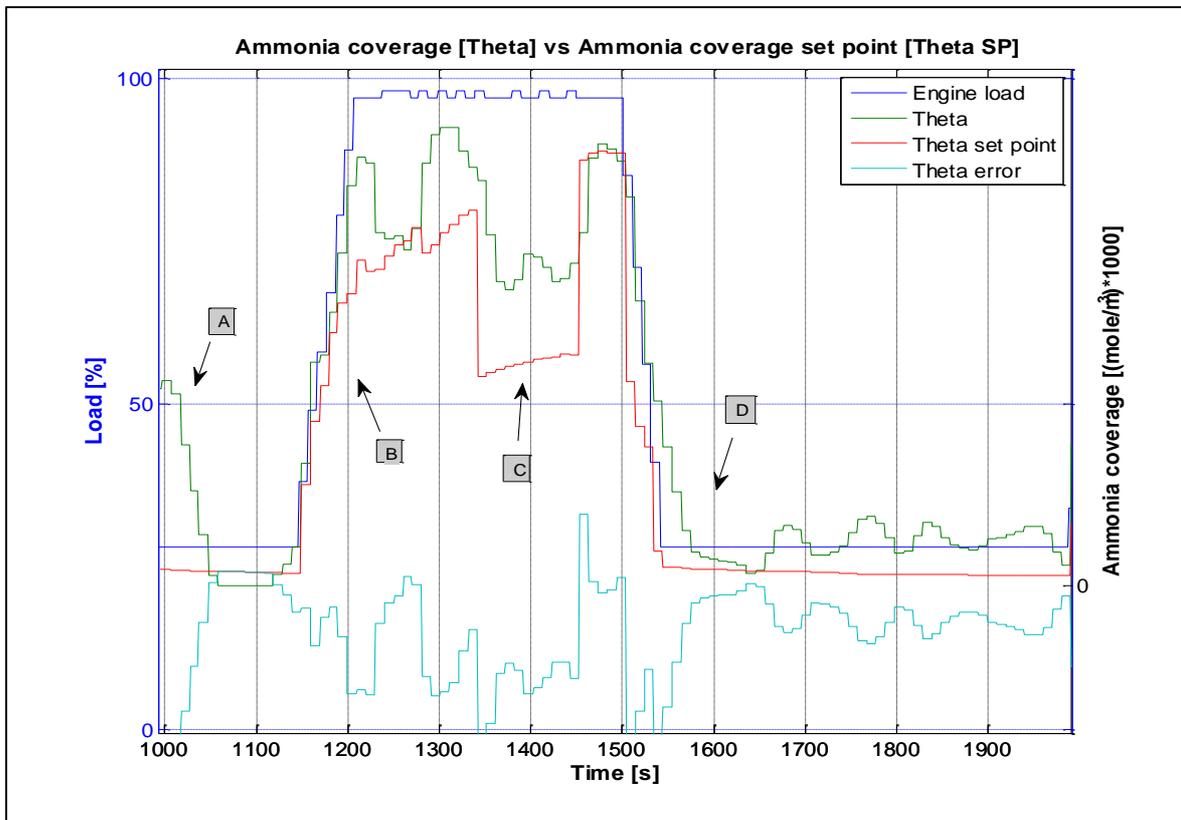


Figure 19. Calculated ammonia coverage (green) and ammonia coverage set point (red) in a situation where the model was influencing the urea control.

The application is allowed to influence the urea flow control approximately at point A in the figure above. As can be seen, the calculated coverage value will immediately at this point start to approach the coverage set point. The calculated coverage value at point A is much higher than the calculated coverage set point. This leads to the calculated error between the two variables lowering the urea flow set point, thus lowering the amount of incoming NH_3 to the reactor. During the load step at point B, the application starts calculating higher set points as the incoming exhaust gas temperature, the incoming level of NO_x and exhaust mass flow are all raising. However, the calculated coverage value seems to be able to adjust quite well according to the coverage set point in this situation. The influence of the model can be considered as quite modest, as it is the conventional feed forward controller that mainly controls the urea flow set point in this situation and as the calculated error between coverage value and set point is fairly small.

At point C, the coverage set point calculator of the application seems to have encountered a calculation problem. The calculated set point changes steeply in an otherwise stable situation. Logically, there is nothing that would support that the coverage set point would undergo hefty changes in similar situations. Thus, it can be concluded that this sudden coverage set point dip is a result of a calculation error in the application. The cause of this error is still unknown, as the reasons behind it could be syntax errors or internal value problems in various matrices. Nevertheless, the application recovered quite fast and well from this situation, as can be seen at point D in the figure. During the reverse load step at point D, the coverage value adjusts fairly well according to the descending coverage set point. Overall, the application seems to be able to adjust the coverage value relatively well according to the coverage set point. However, the control is not perfect as there seems to be some oscillations in the coverage value in situations where the engine load value is settling after a load change. It is also worth to note that the P-controllers gain value was not changed throughout these tests, so it is possible that a lower gain value would minimize these coverage value oscillations. The calculated outgoing NO_x from the same situation is illustrated in the figure below.

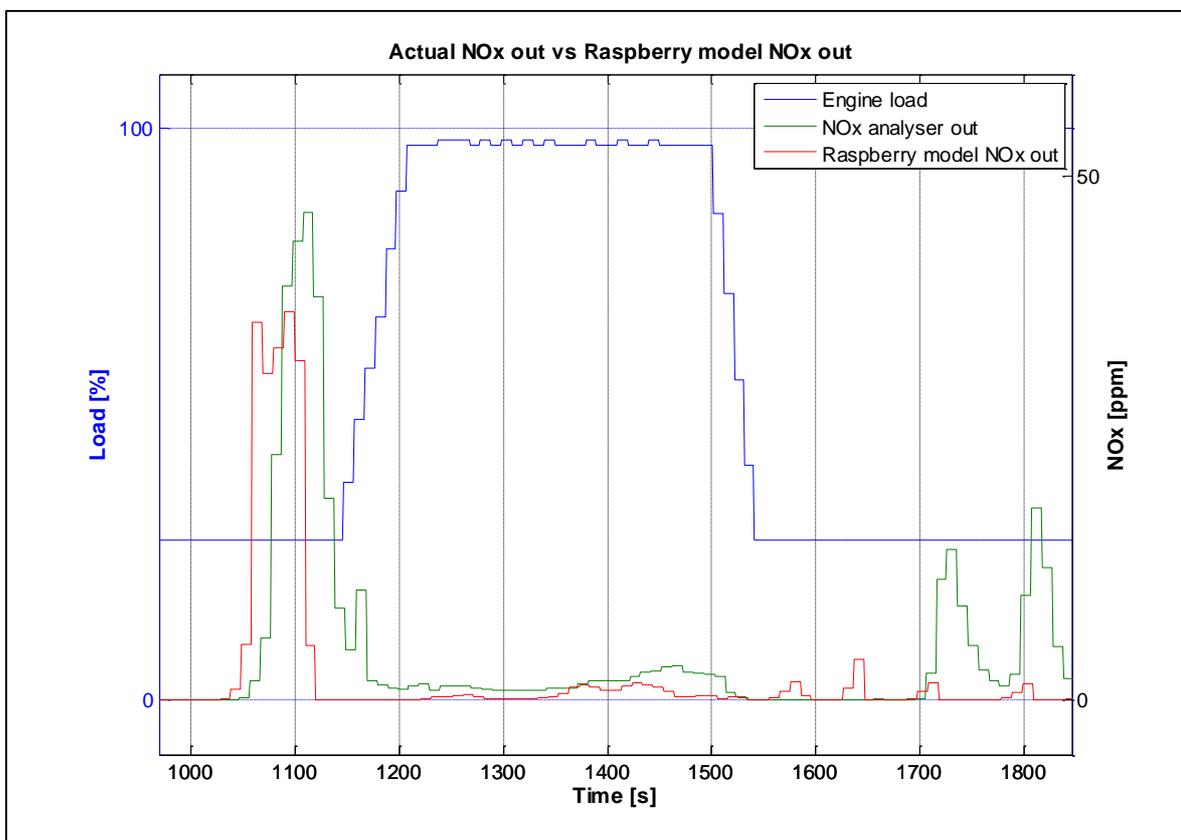


Figure 20. NO_x measured by analyzer (green) versus by model calculated NO_x (red) in a situation where the model was influencing the urea control.

The calculated NO_x in the same situation as in figure 18 follows the same behavior as the calculated outgoing NO_x in figure 17. In other words, the calculated NO_x follows the actual outgoing NO_x quite well overall, albeit not perfectly. The application seems to be calculating a more efficient NO_x -reduction than the factual NO_x -reduction, as in the previous situation. However, the application successfully predicted the sudden peak of NO_x right before the engine load step. The oscillations in the actual outgoing NO_x after the reverse load step could not however be predicted by the application. The calculated outgoing NH_3 from the same situation is illustrated in the figure below.

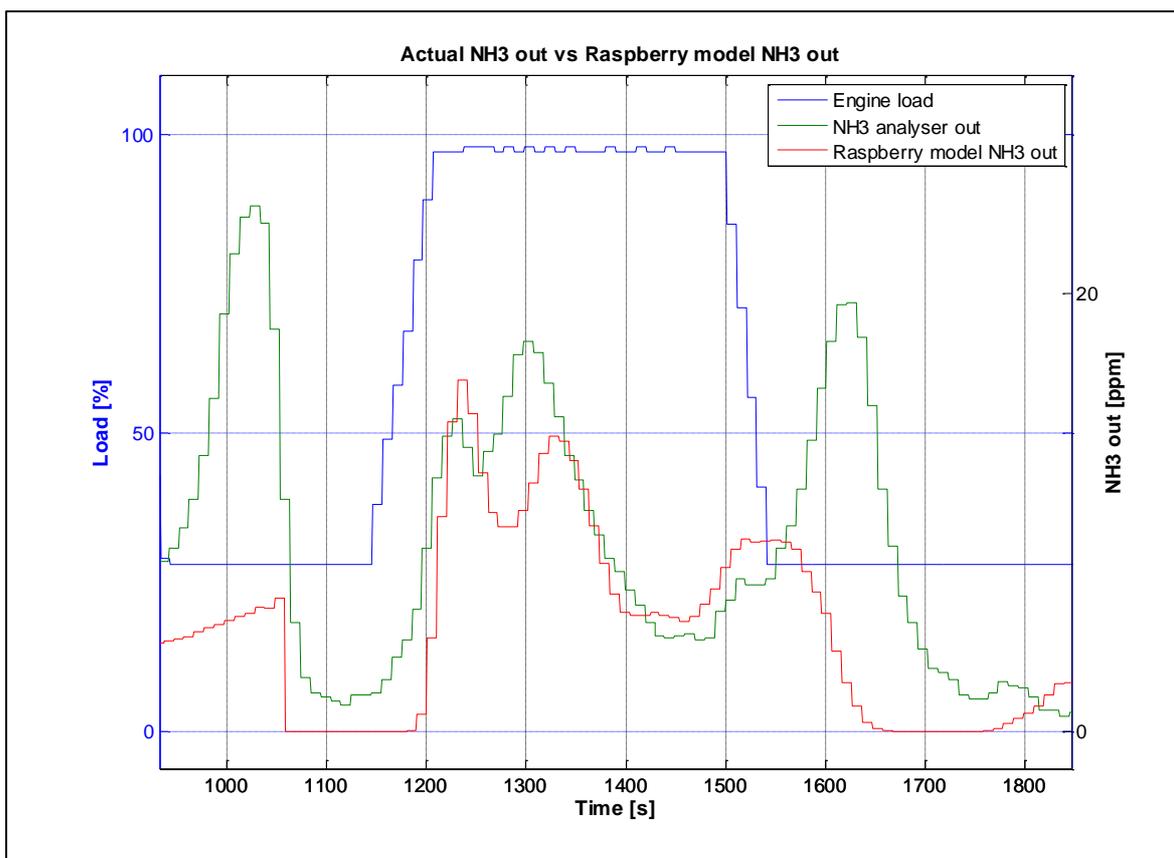


Figure 21. NH_3 measured by analyzer versus by model calculated NH_3 in a situation where the model was influencing the urea control.

As previously mentioned, calculating outgoing ammonia seems to be harder than calculating the outgoing NO_x . The calculated outgoing ammonia differs from the actual outgoing ammonia by a great margin several times in the situation in the figure above. However, the application seems to be able to calculate the outgoing ammonia during stable high load engine operation. During stable low load engine operation and load settling however, the application is unable to predict the peaks of ammonia, as can be seen in the figure above.

Aside from functionality, these tests indirectly indicate that the application is able to interact with a running SCR-control system and a real SCR-process, which was also one of the objectives with running the application together with a real SCR-process. Thus, the aspiration of validating the application's ability to interact with the process and the control system in a stable and safe way, can be considered accomplished.

The impact of the application on the process control is quite hard to estimate, as the conventional feed-forward controller was still in use during the tests. The application is however able to simulate the process and based on the simulation calculate both a coverage set point and a coverage value. As the actual process responded directly when the model was allowed to influence the urea control, it can be assumed that the coverage value and the coverage set point can be utilized in various ways in the control system. Figures 18 and 19 above, imply that the model might reduce the amount of NO_x and NH_3 simultaneously in the moments after a load alteration. This assumption can be made based on the reference load steps where the application was not influencing the urea flow control. In these steps, it seems that the spike amplitudes of both NO_x and NH_3 emitted are higher than the corresponding spikes where the application is influencing the urea flow control. However, this assumption might not be fully correct, as there are many variables that may vary in comparison with other situations and thus affecting the SCR-process. In addition to this, the conventional feed forward controller was still in use during these tests, as previously mentioned.

3.5 Code optimization

This subchapter and its subsections describe the optimization of the application. The reason for optimizing the code was that the original implementation overhead was considered to be too high. Python's built in deterministic code profiler (*cProfiler*) was used in order to create code reports describing the amount of time spent in various functions and methods. According to Python (2016b), the *cProfiler* interpreter adds some overhead to the profiled code, making the execution somewhat slower. Thus, the execution time according to the reports will not directly correspond to the actual execution time when running the application with the conventional Python interpreter. Nevertheless, the reports can still be used to optimize the code. The reports were used to find software bottlenecks that were the most time-consuming and what possibly could be optimized in order to make the application run faster and comply with the set benchmark at 3.5 milliseconds per simulated

SCR per iteration. Reducing overhead would improve the overall stability of the application, and more importantly, improve the precision of the SCR-simulations. As the simulation algorithms can be executed more frequently in an application with low overhead, the simulation can then return more precise data.

3.5.1 Profile tests and results

The profile tests were conducted using *cProfiler's* own Python interpreter running the application. The application was altered in such a way that the actual while-loop found in the main-function was replaced by a for-loop with a fixed loop counter at 1000 iterations. The application would then simulate an SCR 1000 times, while also reading and writing data as well as conducting handshakes with the PLC. The first test was conducted on the original, unaltered application created. Figure 22 below contains a part of the profile report created by the profiler, where the ten most time-consuming methods and functions are listed, ordered by actual time. Actual time only includes the time spent inside the actual functions or methods.

```
Tue Jan 5 09:05:32 2015    profile_output.txt

439318 function calls (439115 primitive calls) in 23.237 seconds

Ordered by: internal time
List reduced from 1065 to 10 due to restriction <10>

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
 1000   15.837    0.016   19.609    0.020 /home/pi/NH3_StorageModel:712(f_model)
   181    2.272    0.013    2.272    0.013 {method 'recv' of '_socket.socket'
objects}
   8000    0.964    0.000    1.070    0.000 {sum}
 60018    0.674    0.000    0.674    0.000 {max}
120322    0.504    0.000    0.504    0.000 {min}
   2000    0.501    0.000    0.731    0.000 /home/pi/NH3_StorageModel:850
(current_theta_sp)
150001    0.400    0.000    0.400    0.000 {math.exp}
     1    0.364    0.364   22.519   22.519 /home/pi/NH3_StorageModel:920(main)
   101    0.342    0.003    0.342    0.003 {method 'take' of 'numpy.ndarray'
objects}
   349    0.101    0.000    0.101    0.000 {numpy.core.multiarray.array}
```

Figure 22. Code profile report. Report ordered by actual time.

As can be seen in the report above, the application required 23.237 seconds to run 1000 iterations of the application, amounting to approximately 23.3 milliseconds per iteration. The method *f_model*, which appears on the first row of the report turned out to be the most time consuming part of the application. As can be seen, both the cumulative time and the actual time of this method are by far exceeding the corresponding metrics for other methods and functions in the report. This method alone consumed 15.837 seconds, or

approximately 68% of the total 23.237 seconds spent running the application. Thus finding the actual problem inside this method would save a lot of time. According to the report above, the Modbus-communication also seemed to be quite time consuming, as the function *recv* of *_socket.socket* alone consumed approximately 2.272 seconds. By studying the implementation, it was discovered that the functionality that wrote simulation data to the PLC had been designed in such a way that every variable was written individually. Because of this, the Pi was forced to send write requests for every single variable written to the PLC. This was fixed by writing multiple registers simultaneously. The figure below contains the second profile report created after this minor problem had been corrected.

```

Tue Jan 19 17:11:00 2016    profile_result_model2.txt

440317 function calls (440114 primitive calls) in 20.446 seconds

Ordered by: internal time
List reduced from 1059 to 15 due to restriction <10>

ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
 1000   15.425   0.015   18.165   0.018 /home/pi/NH3_StorageModel:724(f_model)
   87    1.063   0.012    1.063   0.012 {method 'recv' of '_socket.socket'
        objects}
60018    0.671   0.000    0.671   0.000 {max}
  2000    0.464   0.000    0.688   0.000 /home/pi/NH3_StorageModel:861
        (current_theta_sp)
120322    0.464   0.000    0.464   0.000 {min}
150001    0.396   0.000    0.396   0.000 {math.exp}
     1    0.353   0.353   19.730   19.730 NH3_StorageModel:931(main)
  8000    0.168   0.000    0.273   0.000 {sum}
  1516    0.127   0.000    0.127   0.000 {numpy.core.multiarray.array}
   980    0.070   0.000    0.312   0.000 /usr/lib/pymodules/Python2.7/numpy/
        lib/function_base.py:2861(median)
   980    0.070   0.000    0.070   0.000 {method 'mean' of 'numpy.ndarray'
        objects}

```

Figure 23. Code profile report. Report ordered by actual time.

As can be seen in the report, the total time for running 1000 iterations was reduced by approximately 2.8 seconds from 23.237 seconds to 20.446 seconds. This was accomplished by reducing the number of calls for the *recv* of *_socket.socket*-function, and by writing data to the PLC with one command only. This would amount to an iteration time of approximately 20.4 milliseconds.

Finding the problem in the *f_model*-method was naturally the main target of attention after the communication had been fixed. Upon further analysis of the reports in figure 22, it was discovered that the *f_model*-method's actual time stood for approximately 85% of the total cumulative time consumed. This implied that the actual bottleneck of the code was to be found inside the internal code of the *f_model*-method and not in any of the external

functions called from the method. By studying the internal code of the *f_model*-method, it was discovered that the dynamic typing system of Python caused the method to be very time-exhausting. The figure below contains the part of the code where the bottleneck eventually was found.

As can be seen in the figure below, the variables *temp* and *self.Urea_corr_i* are given their values by calculating the median values of two arrays, which is realized through the Python extension *NumPy* (*np*), a module that supports array handling functionalities. By utilizing the *NumPy* to calculate these two variables, the Python interpreter also automatically chooses the data type for these two variables to be *numpy.float64*, a high precision data type exclusive to the *NumPy*-extension. After calculating the values of the two variables, the interpreter will then calculate the variable *self.Urea_corr_c* by using these two variables and the constant *K_Urea_corr* of the data type *float*. As the calculation involves variables and a constant of different data types, the Python interpreter was forced to convert the data type of the constant, for it to make it comply with the other two variables in the calculation.

```
temp = temp1 - temp2
temp = np.median([-10.0, temp, 10.0])

if DNC == 0:
    self.Urea_corr_i = np.median([-0.5, self.Urea_corr_i+K_Urea_corr \
                                *K_i_Urea_corr*temp*real_step, 0.5])
else:
    self.Urea_corr_i = 0

self.Urea_corr_c = (K_Urea_corr*temp+(self.Urea_corr_i))+1.0
```

Figure 24. Bottleneck.

Converting data types on the fly is very time-consuming for the Python interpreter, especially when converting to data types with high precision. This data type conversion did not appear in any of the reports created, as code profiler is unable to detect these types of forced conversions. Nevertheless, the solution could easily be applied as Python allows hard coded data type conversions of variables that have been assigned *NumPy* data types. Therefore, the datatypes of the variables *temp* and *self.Urea_corr_i* were converted to conventional Python floating point numbers, thus complying with the other data types used in the calculation.

3.5.2 Optimization results

A final profile test was conducted on the fully optimized application to validate that the overhead of the application indeed had been reduced. The profile test report can be seen in the figure below.

```

Thu Jan 28 08:38:58 2016    profile_output3.txt

328829 function calls (328626 primitive calls) in 6.666 seconds

Ordered by: internal time
List reduced from 1076 to 15 due to restriction <10>

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
  1000    2.787    0.003    5.242    0.005 NH3_StorageModel:365(f_model)
   200    0.477    0.000    0.705    0.000 NH3_StorageModel:518(current_theta_sp)
120322    0.370    0.000    0.370    0.000 {min}
     1    0.327    0.327    5.931    5.931 NH3_StorageModel:588(main)
    25    0.317    0.013    0.317    0.013 {method 'recv' of 'socket.socket'
objects}
30000    0.302    0.000    0.302    0.000 {round}
   3019    0.230    0.000    0.230    0.000 {numpy.core.multiarray.array}
60018    0.152    0.000    0.152    0.000 {max}
   1978    0.134    0.000    0.611    0.000 /usr/lib/pymodules/Python2.7/numpy/lib
/function_base.py:2861(median)
   1978    0.132    0.000    0.132    0.000 {method 'mean' of 'numpy.ndarray'
objects}

```

Figure 25. Code profile report after optimization. Report ordered by actual time.

As can be seen, the total time required for running 1000 iterations was reduced to approximately 6.67 seconds. This amounts to a total 71%-time reduction, when comparing the original application and the fully optimized application. As previously discussed, the execution time returned from *cProfiler* reports do not directly correspond to the actual execution time when running an application with the actual Python interpreter. Therefore, further tests were required in order to try and estimate the actual execution time of the application. These tests were conducted using the conventional Python interpreter and time stamping. A time stamp was taken directly when the application was started. This time stamp was then used to calculate the time-span needed for the application to complete 1000 iterations. This was done by calculating the time between said time stamp and the value of the internal clock after completing the 1000 iterations. The results from these tests can be seen in the bar chart below.

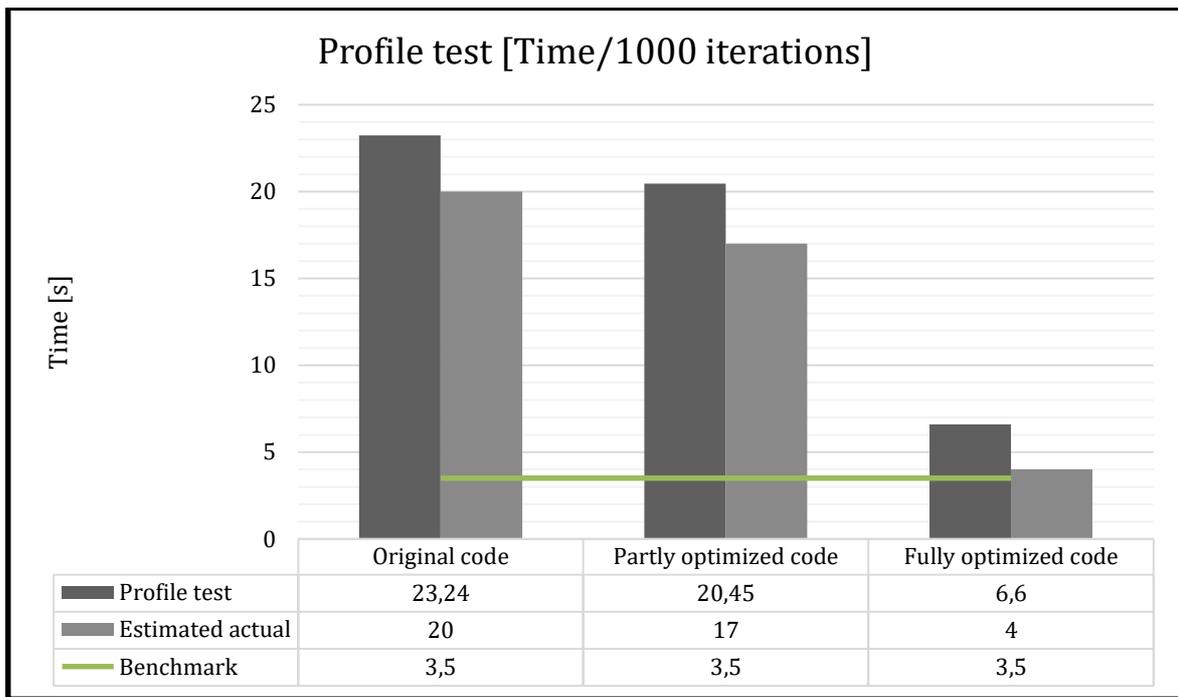


Figure 26. Code optimization results.

As expected, the actual estimated execution times were somewhat lower than execution times in the reports. As can be seen in the figure above, the benchmark had been set at 3.5 milliseconds per iteration per SCR. This would theoretically amount to an estimated iteration time of 42 milliseconds when simulating twelve SCR-systems. Despite the significant improvement in execution time, the fully optimized application does not fully comply with the set benchmark, running at approximately 4 milliseconds per iteration per SCR. Nevertheless, the overhead of the fully optimized application is still considered to be low enough to successfully simulate multiple SCR-systems within reasonable iteration times and with reasonable precision and stability. An iteration time of 4 milliseconds per SCR will theoretically amount to a total iteration time of 48 milliseconds, thus complying with the set benchmark at 50 milliseconds per iteration when simulating twelve SCR-system. Tests were conducted in order to validate this. The figure below was created using the results from these tests.

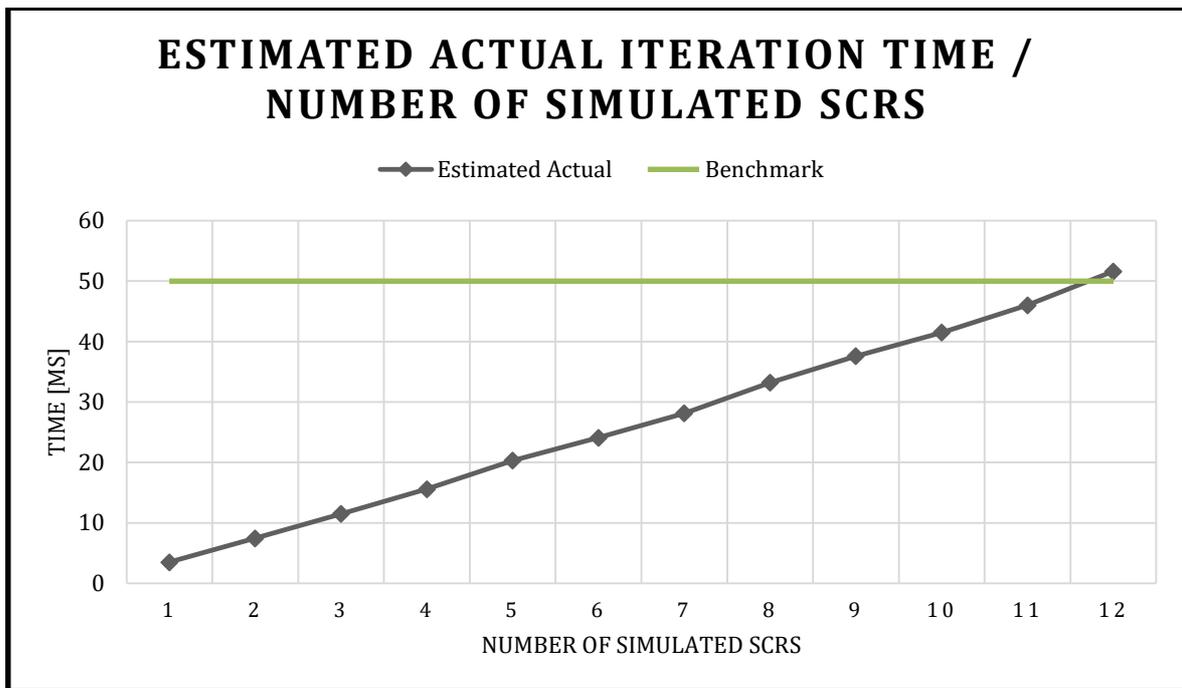


Figure 27. Estimated actual iteration time per number of simulated SCR-systems.

As can be seen in the figure above, the tests revealed that simulating twelve SCR-systems required a bit more time than the set benchmark, running at approximately 51.6 milliseconds per iteration. Nevertheless, the overhead of the application is still low enough to comply with the set benchmark when simulating less than twelve SCR-systems. Overhead seems to be added a bit more than the theoretical 4 milliseconds per SCR per iteration. However, the actual iteration time when simulating 12 SCR-systems is quite near this theoretical iteration time, running at approximately 51.6 milliseconds per iteration. Therefore, it can be concluded that other functionalities seem to be consuming more time with corresponding to the amount of SCRs simulated.

4 Results and conclusion

The goal of the thesis was to create a practical implementation of the SCR-simulation algorithm created by Catalyst Systems R&D. The application that was created, tested, and optimized in accordance with the material presented in chapter 3 *Project realization* and its subchapters, fully corresponds to the final results of the work in this project. Therefore, it can be concluded that the work resulted in a full-scale simulation application, which can safely interact and communicate with the control system of the CSO. Beyond this, the application is also able to handle various exceptional situations, such as communication failures and mode changes in specific SCR-systems. Creating an application that would comply with the requirements stated in chapter 2.1 *Thesis objectives and boundaries* was the main problem of the thesis project. Table 6 below explains how the final application created meets these requirements.

Requirement	Fulfilled?	Comment
Support for simulation of up to twelve individual SCR-systems.	Yes.	The application supports simulation of up to twelve individual SCR-systems.
Support for communication over Modbus TCP/IP.	Yes.	Communication realized by using <i>PyModbus</i> .
Support for re-initialization of Modbus connection upon failure.	Yes.	The application is able to re-initialize the connection upon failure.
Support for validation of simulation data.	Yes.	The application is able to check if simulated data is valid or not.
Ability to recover simulation of an SCR-system that returns from failure mode.	Yes	The application is fully able to recover simulation of an SCR-system returning from failure mode.
Iteration time under 3.5 milliseconds when simulating one SCR-process.	No.	The application does not comply with this benchmark. Further optimization needed to meet this requirement. However, the application is very close to meeting this requirement.
Iteration time under 50 milliseconds when simulating twelve SCR-processes.	No.	The application does not comply with this benchmark. Further optimization needed to meet this requirement. However, the application is very close to meeting this requirement.
Support portability to other operating systems.	Yes.	Scripts written in Python can freely be used on a variety of operating systems.

Table 6. Application requirements.

As can be seen in the table above, the application met all of the requirements, except for the two benchmark requirements. As the actual execution time of the application when simulating one SCR-process was estimated to approximately 4 milliseconds per iteration, it is still considered to be fully possible to achieve this set benchmark by further optimization. Theoretically, by achieving this benchmark the application would then also automatically comply with the second benchmark set at 50 milliseconds per iteration when simulating twelve SCR-processes. It is also worth noting that the application has never been run on any other platform than the Raspberry Pi. Running the application on a more powerful system could also improve the execution time of the application, thus making the application comply with the benchmarks.

Beyond the technical requirements stated in the table 6 above, the main goal was also to create a stable and safe application. The tests that were conducted in Bermeo, show indirectly that the application indeed is stable and can safely interact with the control system of the CSO and furthermore, the SCR-process itself. This assumption is made based on the logged data from the conducted tests (appendix 1-3), that prove that the application can autonomously interact with the control system and simulate a real SCR-process without exceptions and errors. Numerous of unregistered tests conducted using only a PLC-system also confirms this assumption.

Simplicity was also prioritized throughout the whole application development process. This was realized by giving variables and constants logical names that also correspond to the same variables and constants found in the Matlab application. This simplifies making changes in the code and adding new features and functionalities. Therefore, further development of the application can be easily achieved, even by persons who have not been part of the project.

5 Discussion

This thesis project has in many regards been multifaceted, spanning over several technical subjects. Gathering information for creating a knowledge foundation has therefore been quite a challenge. Nevertheless, the book *Urea-SCR Technology for deNO_x After Treatment of Diesel Exhausts* (Nova & Tronconi, 2014), and especially its chapter about SCR-modeling, have been fundamental during the early stages of this project as a basic understanding of modeling SCR-processes was needed. Despite being quite helpful, the book did not explain or cover any theory on how to implement the SCR-model, or how to create a simulation application in practice. Therefore, consultation with colleagues and superiors regarding expected and desired technical functionalities, requirements and control system interaction has also been fundamental throughout the whole project.

Learning a new programming language proved to be a challenge, mostly due to the lack of experience and knowledge regarding Python and OOP in general. As opposed to SCR-simulation and model implementation, theory regarding Python and OOP could be found quite easily, facilitating the learning process. Both official and unofficial documentation on Python have been very important throughout the whole project, as minor programming-related problems have appeared on a regular basis. Despite these minor problems, Python proved to be an easy language to learn, in virtue of the large amount of information available on the subject and to its simple syntax.

With the benefit of hindsight, OOP should have been studied more thoroughly, as the initial approach was to create a sequential-oriented application. This came quite naturally, as I had previous to this thesis project solely programmed in sequential-oriented languages. Luckily, most of the code created in the sequential-oriented application could still be used in the OO-application after some minor modifications. Creating a new OO version of the application proved to be the right decision, as the complexity of the code was reduced, and the readability of the code was improved significantly. Nevertheless, a considerable amount of time could have been saved, had the initial approach been to create a OO type of application.

Code optimization was an unexpected part of this thesis project. Mostly due to the fact that the execution time of the application was not foreseen to become a problem. Nevertheless, the application as it was implemented after the tests in Bermeo, implied that the application

required optimization, even though the application was executing fast enough for simulating one SCR with sufficient precision. The *cPython*-package turned out to be invaluable, as it would have been very tedious to guess what operations were the most time consuming. Had it not been for the lack of time during the application optimization stage, the application would also have been optimized using multi-threading. A multi-threaded application runs on several processor cores instead of one. This would make it possible to create an application where certain tasks are run on a specific processor core. In this thesis project, this could have been utilized in order to simulate multiple SCR-systems simultaneously on several processor cores. For instance, the application could have been split in such a way that the SCRs 1 – 3 are simulated on the first CPU core, SCRs 4 – 6 on the second CPU core, SCRs 7 – 9 on the third CPU core and SCRs 10 – 12 on the fourth CPU core. Theoretically, this would have improved both execution time and calculation precision for every specific SCR-system simulated.

The application created in this project is as of now ready to be used in practice. However, the application still needs to undergo numerous tests, mostly to validate that the application is stable enough to be used for longer periods of time. The online tests conducted in Bermeo were all conducted during a single afternoon. As there were many other matters that required my attention in Bermeo, there was simply not enough time to test the application in order to validate that the application indeed is stable enough to be used for continuous simulation under a longer period of time. In spite of this, the application is still considered to be stable and reliable enough to be used in practice.

The application was from the beginning of the thesis project, intended to be integrated with the PLC-application of the CSO. However, the application can also, after some minor modifications be integrated with the UNIC-application of the NOR. Beyond the obvious area of usage as a real-time SCR state-observer application, the application can therefore also be used as a fundament for both a PLC and UNIC software test bench. As the essential functionality of the application is to simulate the process itself, it would not require much work to take the application into a new area of usage as a software testing platform. This would also require that models would be created of the other components of the system such as the dosing unit and the pump unit. As the application and Python is capable of reading Matlab-data, logged data could be used in this test bench for SCR-simulation and software testing.

References

- Abelli, B. 2004. *Programmeringens grunder*. Lund, Sweden: Studentlitteratur.
- Cass, S. & Diakopoulos, N., 2015. *Interactive: The Top Programming Languages 2015*. [Online] <http://spectrum.ieee.org/static/interactive-the-top-programming-languages-2015> [Retrieved: 14.11.2015]
- Chatterjee, D. & Rusch, K., 2014. *SCR Technology for Off-highway (Large Diesel Engine) Applications*. In: I. Nova & E. Tronconi ed. *Urea-SCR Technology for deNOx After Treatment of Diesel Exhausts*. New York, USA: Springer Verlag New York Inc.
- Crispin, A.J., 1997. *Programmable Logic Controllers and their Engineering Applications*. Maidenhead, England: McGraw-Hill Publishing Company.
- Eastwood, P., 2000. *Critical Topics in Exhaust Gas Aftertreatment*. Hertfordshire, England: Research Studies Press LTD.
- Halfacree, G. & Upton, E., 2014. *Raspberry Pi User Guide*. Chichester, West Sussex, United Kingdom: John Wiley & Sons Ltd.
- Hsieh, M-F. & Wang, J., 2014. *Diesel Engine SCR Systems: Modeling, Measurements and Control*. In: I. Nova & E. Tronconi, ed., *Urea-SCR Technology for deNOx After Treatment of Diesel Exhausts*. New York, USA: Springer Verlag New York Inc.
- Johnson, T.V., 2014. *Review of Selective Catalytic Reduction (SCR) and Related Technologies for Mobile Applications*. In: I. Nova & E. Tronconi ed. *Urea-SCR Technology for deNOx After Treatment of Diesel Exhausts*. New York, USA: Springer-Verlag New York Inc.
- Khair, M.K. & Majewski, W.A., 2006. *Diesel Emissions and Their Control*. Warrendale, Pennsylvania, USA: SAE International.
- Lutz, M., 2011. *Programming Python*. Sebastopol, California, USA: O'Reilly Media, Inc.

Nova, I. & Tronconi, E. ed., 2014. *Urea-SCR Technology for deNO_x After Treatment of Diesel Exhausts*. New York, USA: Springer-Verlag New York Inc.

Peavy, H.S., Rowe, D.R. & Tchobanoglous, G., 1985. *Environmental Engineering*. Singapore: McGraw-Hill Book Co.

The Python Guru., 2015. *Getting started with Python*. [Online]
<http://thePythonguru.com/getting-started-with-Python/> [Retrieved: 8.12.2015]

Python., 2016a. *The Python Language Reference: Lexical analysis*. [Online]
https://docs.Python.org/2/reference/lexical_analysis.html [Retrieved: 31.1.2016]

Python., 2016b. *The Python Profilers*. [Online]
<https://docs.Python.org/2/library/profile.html#module-cProfile> [Retrieved: 12.3.2016]

Python., 2016c. *The Python Standard Library*. [Online] <https://docs.Python.org/2/library/>
[Retrieved: 31.1.2016]

Sintes, T. 2002. *Teach Yourself Object Oriented Programming in 21 days*. Indianapolis, Indiana, USA: SAMS.

Upton, E. 2015. *Raspberry Pi 2*. [Online] <https://www.raspberrypi.org/blog/raspberry-pi-2-on-sale/> [Retrieved: 8.2.2016]

Wärtsilä., 2014. *Wärtsilä Environmental Product Guide*. [Online]
<http://cdn.wartsila.com/docs/default-source/product-files/exhaust-gas-cleaning/product-guide-o-env-environmental-solutions.pdf?sfvrsn=14> [Retrieved: 7.10.2015]

Wärtsilä., 2015. *This is Wärtsilä*. [Online] <http://www.wartsila.com/about> [Retrieved: 3.10.2015]

Zandbergen, P.A., 2013. *Python Scripting for ArcGIS*. Redlands, California, USA: Esri Press.

