



# Tuning light knock limits on SG engines

Janne Holtti

Bachelor's thesis  
Automation Technology  
Vaasa 2012



# BACHELOR'S THESIS

Author: Janne Holtti  
Degree programme: Electrical Engineering, Vaasa  
Specialization: Automation technology  
Supervisor: Roger Mäntylä

Title: *Tuning light knock limits on SG engines*

---

Date: 25.4.2012

Pages: 41

---

## **Abstract**

This thesis is about engine knock and knock control in general. The purpose of the thesis work was to develop a semi-automated program for the WECSplorer Wizard plug-in using the Python programming language. The program is to aid in the tuning of the light knock limits on spark ignited gas engines.

This document covers the causes and effects of knocking on an engine and the normal control procedures associated with keeping the engine out of a knocking state and the sensors used in measuring the knock levels.

The basics of Python, wxPython and some general programming paradigms and options have been studied.

The project and its core functionality and development are described. The most critical functions that affect engine safety, which had stricter specifications, are also described in the thesis.

---

Language: English

Keywords: knock, python, wecs, unic, programming

---

The thesis is available in the web library Theseus.fi or in Tritonia, Academic Library, Vaasa

# EXAMENSARBETE

Författare: Janne Holtti  
Utbildningsprogram: Elektroteknik  
Inriktningalternativ: Automationsteknik  
Handledare: Roger Mäntylä

Titel: *Inställning av de svaga knackningens gränsvärden på gasmotorer*

---

Datum: 25.4.2012

Sidantal: 41

---

## **Abstrakt**

Detta examensarbete handlar om motorknack och styrning av motorknack i allmänhet. Målet med examensarbetet var att utveckla ett semi-automatiskt program i Python som skall användas för att underlätta inställningen av den svaga knackningens gränsvärden på gasmotorer.

Dokumentet innehåller information om de orsaker som ger upphov till knackning samt knackningens effekter. Utöver det så tar dokumentet upp de allmänna kontroll systemen och de sensorer som kan användas i mätning av knack.

Dokumentet tar upp Python och wxPython-programmering och vissa allmänna tillvägagångssätt som utnyttjats i programmeringen av programmet.

Till sist tar dokumentet upp projektet och dess funktionalitet och utveckling. Specifikt de kritiska funktioner tas upp som hade specifika krav gällande motorns säkerhet.

---

Språk: engelska

Nyckelord: knack, python, wecs, unic, programmering

---

Examensarbetet finns tillgängligt antingen i web biblioteket Theseus.fi eller i Tritonia, Vetenskapliga bibliotek, Vasa

# OPINNÄYTETYÖ

Tekijä: Janne Holtti  
Koulutusohjelma: Sähkötekniikka  
Suuntautumisvaihtoehto: Automaatiotekniikka  
Ohjaaja: Roger Mäntylä

Nimike: *Kevyen nakutuksen rajan säätäminen kaasumoottoreilla*

---

Päivämäärä: 25.4.2012

Sivumäärä: 41

---

## **Tiivistelmä**

Tämä opinnäytetyö käsittelee moottorien nakutusta ja nakutuksen ohjausta yleisesti. Tavoitteen opinnäytetyössä oli kehittää Pythonissa puoliautomaattinen ohjelma, jota käytettäisiin apuna kaasumoottoreiden kevyen nakutuksen rajan säätämiseen.

Opinnäytetyöstä löytyy tietoa asioista, jotka voivat aiheuttaa nakutusta moottoreissa ja niiden vaikutuksista. Sen lisäksi opinnäytetyö kuvaa tavanomaiset ohjausmenetelmät ja sensorit, joita käytetään nakutuksen mittauksessa.

Opinnäytetyö käy läpi Python- ja wxPython-ohjelmoinnin yleisesti ja tavanomaiset lähestymistavat, joita käytetään ohjelmoinnissa.

Lopuksi työssä kuvataan projekti ja sen kehitystä ja toiminnallisuutta. Erityisesti opinnäytetyö käy läpi kriittiset toiminnot, jotka ohjelman täytyy täyttää, jottei moottorille tapahdu mitään haitallista.

---

Kieli: englanti

Avainsanat: nakutus, python, wecs, unic, ohjelmointi

---

Opinnäytetyö on saatavilla joko ammattikorkeakoulujen verkkokirjastossa Theseus.fi tai Tritoniassa, Vaasan tiedekirjasto.

# Contents

|          |                                    |           |
|----------|------------------------------------|-----------|
| <b>1</b> | <b>Introduction</b>                | <b>1</b>  |
| 1.1      | The company in brief . . . . .     | 1         |
| 1.1.1    | Ship Power . . . . .               | 1         |
| 1.1.2    | Power Plants . . . . .             | 1         |
| 1.1.3    | Services . . . . .                 | 1         |
| 1.1.4    | Industrial Operations . . . . .    | 1         |
| 1.1.5    | Background . . . . .               | 2         |
| 1.2      | Objectives . . . . .               | 2         |
| <b>2</b> | <b>Knock theory</b>                | <b>3</b>  |
| 2.1      | Causes for knocking . . . . .      | 3         |
| 2.2      | Effects of knocking . . . . .      | 4         |
| 2.3      | Knock detection . . . . .          | 5         |
| 2.3.1    | Knock sensors . . . . .            | 5         |
| 2.3.2    | UNIC . . . . .                     | 7         |
| 2.4      | Knock control . . . . .            | 9         |
| 2.4.1    | Signal processing . . . . .        | 10        |
| 2.4.2    | Detection strategy . . . . .       | 12        |
| <b>3</b> | <b>Python</b>                      | <b>14</b> |
| 3.1      | Typing system . . . . .            | 15        |
| 3.2      | Interactivity . . . . .            | 16        |
| 3.3      | Standard library . . . . .         | 18        |
| 3.3.1    | Extending Python . . . . .         | 18        |
| 3.4      | wxPython . . . . .                 | 18        |
| 3.4.1    | wxPython architecture . . . . .    | 19        |
| 3.4.2    | wxPython modules . . . . .         | 19        |
| 3.4.3    | wxPython inheritance . . . . .     | 20        |
| 3.4.4    | wxPython events . . . . .          | 20        |
| <b>4</b> | <b>Development and realization</b> | <b>22</b> |
| 4.1      | Design . . . . .                   | 22        |
| 4.1.1    | Graphical User Interface . . . . . | 22        |

|          |  |           |
|----------|--|-----------|
| 4.1.2    | MVC - Model-View-Controller . . . . .        | 23        |
| 4.1.3    | PEP8 . . . . .                               | 24        |
| 4.2      | Class and module design . . . . .            | 24        |
| 4.2.1    | API module . . . . .                         | 25        |
| 4.2.2    | Engine class . . . . .                       | 26        |
| 4.2.3    | Cylinder class . . . . .                     | 27        |
| 4.2.4    | EventHandler, the main class . . . . .       | 27        |
| 4.2.5    | External data handling . . . . .             | 28        |
| 4.2.6    | Data visualization . . . . .                 | 29        |
| 4.3      | Critical requirements . . . . .              | 30        |
| 4.3.1    | Ignition angles . . . . .                    | 31        |
| 4.3.2    | Setting the knock limits . . . . .           | 32        |
| 4.3.3    | Calculating the light knock limits . . . . . | 32        |
| 4.4      | Reporting . . . . .                          | 34        |
| 4.5      | Documentation . . . . .                      | 34        |
| <b>5</b> | <b>Results</b>                               | <b>36</b> |
| <b>6</b> | <b>Discussion</b>                            | <b>38</b> |
|          | <b>Bibliography</b>                          | <b>40</b> |

# Abbreviations

UNIC = Unified Control

WECS = Wärtsilä Engine Control System

SG = Spark-ignited Gas

DF = Dual Fuel

FFT = Fast Fourier Transform

DFT = Digital Fourier Transform

DSP = Digital Signal Processing

FIR = Finite Impulse Response

API = Application Programming Interface

STL = Standard Library

PyPI = Python Package Index

PIP = Pip Installs Packages

GUI = Graphical User interface

XML = eXtended Markup Language

MFI = Multi-port fuel injection

RST = reStructuredText

# 1 Introduction

## 1.1 The company in brief

Wärtsilä is a global company providing complete solutions for the energy and marine markets. In 2010, Wärtsilä's net sales totalled EUR 4.6 billion with 17,500 employees. The company has operations in 160 locations in 70 countries around the world. Wärtsilä can be divided into four main units working together, Industrial Operations, Ship Power, Power Plants and Services.

### 1.1.1 Ship Power

Ship Power supplies its customers with integrated systems, solutions and products ranging from alarm systems to complete control systems including engine control.

### 1.1.2 Power Plants

Power Plants is a leading supplier of flexible power plants for the power generation markets. Wärtsilä offers solutions for base load power generation, grid stability/peaking, industrial self-generation and for the oil and gas industry.

### 1.1.3 Services

Services supports Wärtsilä customers throughout the lifecycle of their installations and products by optimizing efficiency and performance. Services provides a comprehensive portfolio of services and upgrade products. Services has one of the broadest service networks in the industry.

### 1.1.4 Industrial Operations

Industrial operations encompass Wärtsilä factory, research and development, laboratory and more. Industrial operations is the base of the company supplying the other units with software, technology and new standard solutions which are adapted to customer needs in the other units.



### 1.1.5 Background

In the summer of 2011 I was working as a trainee at Wärtsilä/Services/Electrical and Automation/Product Development. My normal tasks included writing technical specification, system specifications for various products/upgrades and the occasional investigation into an idea for a new solution or product.

As the summer vacation months were drawing closer for my coworkers, my supervisor had an idea for a project for me to work on for the rest of the summer. From there I got in contact with Technical Service and I was given an introduction to the idea. We suspected that the work alone could be enough for a Bachelor's thesis.

## 1.2 Objectives

The objective or end product for my thesis work is a wizard plug-in for the engine configuration tool WECSplorerUT (v1.2). The wizard should automate the process of tuning the light knock limit on gas engines, specifically the SG engines with UNIC engine control system. The wizard could be later on be expanded to the WECS engine control system and DF engines.

The program should have a graphical representation of the incoming knock values so that the user can determine when a knock occurs and based on the values try to calculate an appropriate light knock limit. It should be user configurable in the sense that all the timings and limits should be configurable inside the program.

While tuning the program should automatically raise the ignition angle of the cylinder. Because a knock is very hard to detect programmatically with any certainty the program will require user input based on hearing when the engine is in a knocking state. This will then trigger the calculation and automatically decrease the ignition angle of the cylinder.

The program should also have a reporting function that creates reports of the tuning done to the engine.

My work is based on the WECSplorerUT API, which was developed in 2007 following research done for a Master's thesis. The API has been mostly unused with the exception of a configuration tool for the ESM, which was developed alongside the API.

## 2 Knock theory

The term knock is the name given to the noise that is transmitted through the engine structure when a spontaneous and independent combustion of a proportion of the end-gases occurs ahead of the propagating flame front traveling through the cylinder initiated by the spark. [1]

### 2.1 Causes for knocking

Knocking, a spontaneous combustion, is a result of temperature, pressure and time. If, for instance, the temperature and the pressure of the end gases reach the threshold for self-ignition before the flame front initiated by the spark has had time to propagate to the cylinder walls, a knock will occur.

Knocking creates shock waves and thermal explosions throughout the cylinder leading to audible and physical effects that affect the consumer and the engine. Various steps can be taken to decrease an engine's tendency to knock.

Increased burn rates will allow less time for the end gases to heat up and ignite before combustion is completed. Colder charge and block temperatures will guarantee more time for the end gases to reach the threshold temperature of self-ignition, giving the flame front more time to reach the end gases before knock occurs. Higher octane fuels have a similar effect on the end gases, by raising the activation energy required to self-ignite. Rich mixtures burn faster than lean ones, and they heat up the cylinder less for each consecutive combustion event.

Ignition timing can shift the peak pressure of the combustion event away from top dead center (when the piston is farthest away from the crankshaft), thus avoiding the dramatic rise in cylinder pressure (and temperature) when the two are in close proximity relative to the crank angle [2].

Some of the steps listed above must be addressed in the base engine design, but some can be utilized in an active cylinder-by-cylinder, cycle-by-cycle knock controller to effectively reduce knock only in operating conditions where knock is present, thus retaining the majority of the engine's potential power whenever possible.

## 2.2 Effects of knocking

Knocking has many adverse effects on an engine. These range from mechanical wear to an actual damaging effect on different parts on the engine.

Engine efficiency is also something that is directly affected by knocking. Comparing an engine under normal running conditions with a heavily knocking engine shows that engine efficiency drops by around 13% [3].

When a knocking-related ignition occurs, the cylinder pressure is about twice the pressure at the beginning of normal combustion. The same applies for absolute temperature. This leads to the fact that the possibility of engine seizure will grow higher. However this is highly improbable, because the anti polishing ring (APR) and the piston are made of different materials with different thermal expansion coefficients. Different materials with different coefficients can lead to the fact that the piston top will expand more than the APR during knocking. It must be stated that the occurrence of this is also very improbable, as the temperature difference would have to be in the range of 500°C.

Engine parts may suffer from mechanical deformation because of knocking, due to the increased pressures inside the cylinder chamber. The compression of the piston and oil films, a deformation of the gudgeon pin, the intake valves, the exhaust valves and the connecting rod due to knocking all affect the engine life-span in adverse ways.

## 2.3 Knock detection

Knocking is determined by measuring the levels using one of the methods described in the following subsection in this thesis and by using signal processing techniques described in later sections.

First of all the resonance frequency of the cylinder is determined using Drapers equation [4].

$$f_{m,n} = \frac{c_0 \cdot \sqrt{T} \cdot n_{m,n}}{\pi \cdot B} \quad (2.1)$$

where:

|           |   |
|-----------|---|
| $f_{m,n}$ | = resonant frequency                      |
| $n_{m,n}$ | = non-dimensional mode number             |
| $c_0$     | = phase velocity constant                 |
| $T$       | = combustion mixture temperature          |
| $B$       | = cylinder bore diameter                  |
| $m, n$    | = radial and circumferential mode numbers |

This frequency is the base for the selection of the sensor (mechanical vibration) and the signal processing procedures that are applied to the signal before executing the knock detection strategy.

### 2.3.1 Knock sensors

There are several ways to measure knock oscillations. The ones used on Wärtsilä engines are either combustion pressure sensors or mechanical vibration sensors.

#### Combustion pressure sensor

A direct approach is to measure the combustion pressure. In this approach the sensors measure the pressure inside the combustion chamber of a running engine. This approach provides the best signal to analyze and it is used on some Wärtsilä engines. However, the direct approach is not available on all engines, as the sensor cost is relatively high and each cylinder requires its own sensors [5].

#### Mechanical vibration sensors

This type of knock sensor is a piezoelectric vibration sensor in correct terms. It measures the structure-born vibrations on the cylinder that occur at uncontrolled combustion. As

a result of the forces generated by the vibrations in the engine the piezo-elements in the sensors generate a voltage that can be measured.

These sensors are available in two different types, tuned and broadband. A limitation with the tuned approach is that a different sensor might be required for each engine type, due to variations in the characteristic frequency. This results in a need of a wide selection of the same basic part with slightly different characteristic, which increases the overall cost of a system. To eliminate that cost, the broadband sensor is more common as its bandwidth can be made wide enough to cover expected variations in the cylinder characteristic frequency. These sensors' bandwidth range from 1 to 20 kHz and are no different from the knock sensors found in a regular car. This type of sensor is also what the end result of the thesis work is based on.



*Figure 1. Bosch knock sensor*

### **Ion current measurement**

When using ion current measurement the standard spark plug is the sensor. During the combustion process electrically charged ions and electrons are generated. The intensity of the ionization depends on the flame temperature, the air fuel ratio and the fuel quality [6].

The principle of the measurement is actually really simple. The technique is based on applying a low-voltage DC bias to the spark plug after the ignition coil has discharged. This bias voltage attracts the ions and electrons created in the combustion process to move in one direction, which leads to a measurable ionic current being formed between the two electrodes of the spark plug [7].

The problem with ion current measurement is that it only represents the combustion intensity in a very small volume around the spark plug. Therefore, when the technique is used in knock detection, the accuracy heavily depends on the position of the spark plug [8].

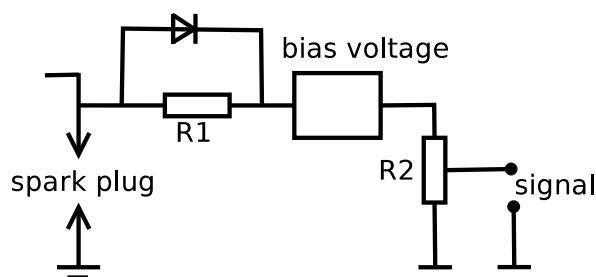


Figure 2. Principle schematic of the ion current measurement

### Light intensity of combustion process

The knock oscillations modulate the intensity of the combustion process. This leads to a modulation of the light intensity and color in the combustion chamber. Therefore light measurement is another way of measuring knocking.

A fiber-glass cable is fed through the spark plug from where the light is forwarded to a photo transistor. The problem with this approach is the buildup of soot, which leads to inaccuracy and variations in the intensity of the light. These are hard to predict or account for in the long run.

### 2.3.2 UNIC

UNIC is the embedded engine control system that through its various modules handles most of the functionality on a modern Wärtsilä engine including starting and stopping, safety functions, load control and more.

UNIC is a modular system that can be customized with different number of modules depending on the engine size and type to get functionality specific to the engine configuration. Presently there are three different base configurations of UNIC. The first is the C1, which is the most basic of the UNIC systems and handles for example the starting and stopping of the engine. After that comes the C2, which handles for example the starting and stopping and the speed/load control. The last of the UNIC base configurations is the C3, which includes all functionalities of the previous configurations and adds safety management to the list. UNIC C3 is required for gas powered engines.

UNIC is designed to be on-engine, and as such it is a very robust system that endures high temperatures and vibrations. Some off-engine versions of UNIC have also been developed for retrofitting non-portfolio engines.

Communication in the UNIC system is handled over CAN-bus (actually two buses, one for redundancy) and all information between the modules travels on the network. The rate of polling the knock values for this thesis was determined to be slow enough in order not to cause any trouble on the bus network. The usual method for tuning involved polling for the knock values every 10 ms versus the 30 ms that the developed program is limited to.

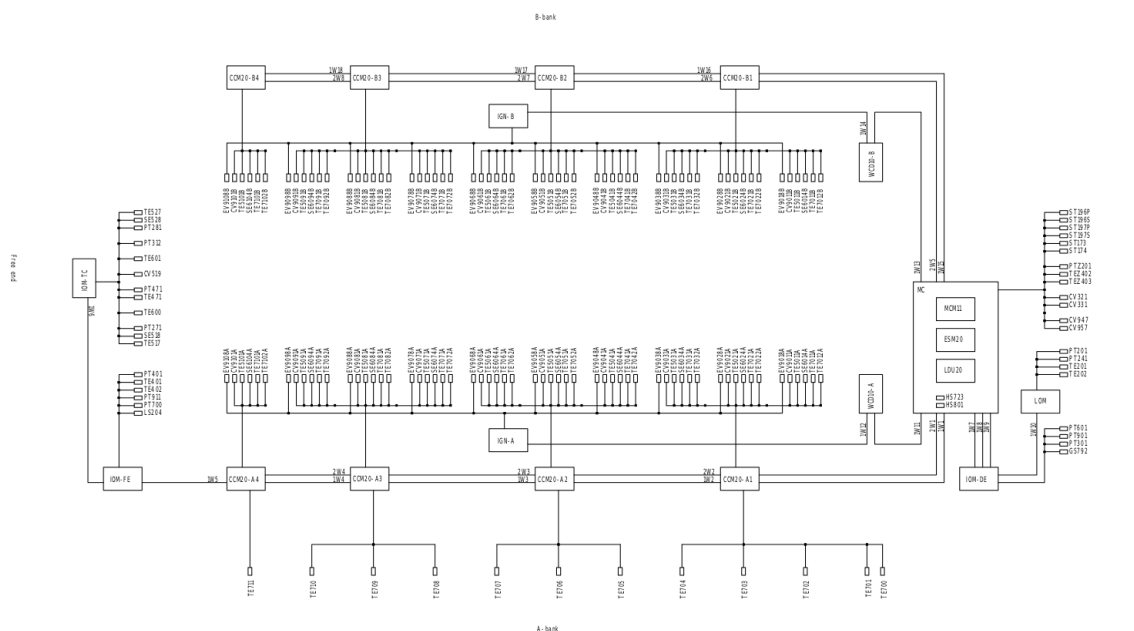


Figure 3. Schematic of the UNIC engine control system

## UNIC modules

The major modules that the UNIC engine control system comprises:

### MCM-11 Main Control Module

The MCM is the main module of the engine and it controls among other things the starting and stopping of the engine. The MCM also controls many of the other modules in the UNIC system and acts as the main data gathering point for this thesis.

### CCM-20 Cylinder Control Module

As implied by the name, CCM handles functions related to cylinders. It controls fuel injection and cylinder specific measurements. One CCM can handle injections for up to three cylinders simultaneously and can act as a backup for another three cylinders. The CCM is also capable of functioning in limp mode if the CAN buses fail or if the MCM fails.

### IOM-10 Input Output Module

IOM is used for collecting the signals from the various sensors mounted around the engine and it also handles some logical functions, such as waste-gate/by-pass control and others.

### ESM-20 Engine Safety Module

The ESM is an important module, if not the most important. It handles functions related to the safety of the engine. Also related to the engine safety, the module handles speed measuring and monitoring. Other functions include e.g. monitoring the lube oil pressure and the cooling water temperature.

**PDM** Power Distribution Module

The PDM name refers to the fact that this module distributes power to the various modules and electronics on the engine. It is designed with redundancy in mind to ensure power to the rest of the system and it has two supplies, one primary and one backup. It has short circuit protection, EMC filtering, earth fault detecting and detection of over voltages.

**LCP** Local Control Panel

This panel controls the engine locally and it consists of the following parts.

**LDU-20** Local Display Unit

LDU is the operator interface for the UNIC system and this also acts as communication interface to other networks. It displays sensor data, engine modes, failures and an event log.

**WIP-11** Wärtsilä Instrument Panel

The WIP displays the most important measurements such as the lube oil pressure, the engine speed and others.

**WCP-10** Wärtsilä Control Panel

The WCP contains switches and buttons for starting, stopping and resetting.

## 2.4 Knock control

The effects of knocking lead to the need of effective knock control. Usually the control methods involve retarding the ignition angle or limiting the boost pressure on turbo-charged engines [9]. These methods are executed based on heuristically determined levels. Determining one of these levels is the wanted end-result of this thesis.

Disregarding the specific adaptation circuitry depending on the measurement used, a somewhat uniform methodology can be applied in the signal processing stage. The next subsection will describe the implementation with the mechanical vibration sensor.

The knock detection algorithm must be able to adapt to a number of different variables to generate the optimum ignition timing close to the knock threshold. The mounting of the sensor (mechanical) including how clean the sensor is and the design of the engine affect which frequency modes will be detectable by the sensor. This usually means that the transfer function between the cylinder and the sensor is different for each cylinder. This will make absolute and relative magnitudes of the vibrational modes different for each cylinder [5].

Another thing that must be accounted for is the non-knocking reference signal amplitude which changes with higher RPM. If the engine speed is increased, the background vibration level will increase.



Finally, the engine characteristics will change with time. Parts wear down and tolerances change between parts, which can lead to changes in the magnitudes detected by the sensor. Background vibration can also be higher for a given engine speed.

### 2.4.1 Signal processing

The signal from the sensor must be processed to get the information about the signal strength in the required frequency range. If a tuned sensor is used with the resonant peak close to the resonant frequency of the cylinder, no further signal processing will be needed. In other situations, either some filtering technique (analogue or digital) or a spectral estimation must be used, which is the norm.

Analogue filtering is a cheap and easy way to process the signal, but because it lacks precision and because digital processing power is relatively cheap today it is not widely used. Digital filters are practical and can be shared across engine configurations.

Spectral analysis, e.g. the Fast Fourier Transform or the Discrete Fourier Transform, is more widely used today as it provides higher frequency resolving powers than a digital filter.

Below an example of how the signal processing procedures, with broadband knock sensors, would be implemented using DFT. The first step of the signal processing starts with the DFT.

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot e^{-j2\pi k[n]/N} \quad (2.2)$$

where:

|        |  |
|--------|--|
| $k$    | = frequency index                              |
| $n$    | = sample time index                            |
| $N$    | = sample block length                          |
| $X[k]$ | = amplitude of sinusoid at frequency index $k$ |
| $x[n]$ | = time domain representation of a signal       |

As processors cannot easily represent complex exponentials the DFT equation is required to be represented in a way that can be implemented in a DSP. This is done by leveraging Euler's relationship between complex exponentials and sinusoids:

$$e^{-j\Theta} = \cos \Theta - j \sin \Theta \quad (2.3)$$

Substitution gives:

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot \cos(2\pi k[n]/N) - j \sin(2\pi k[n]/N) \quad (2.4)$$

Finally, separating the real and imaginary part gives two equations that can easily be programmed:

$$X_R[k] = \sum_{n=0}^{N-1} x[n] \cdot \cos(2\pi k[n]/N) \quad (2.5a)$$

$$X_I[k] = \sum_{n=0}^{N-1} x[n] \cdot \sin(2\pi k[n]/N) \quad (2.5b)$$

The actual form of the difference equation on the DSP:

$$X_R[k] = X_R[k] + x[n] \cdot \cos(2\pi k[n]/N) \quad (2.6a)$$

$$X_I[k] = X_I[k] + x[n] \cdot \sin(2\pi k[n]/N) \quad (2.6b)$$

As the summation of the sample block is completed, the signal strength of every frequency range is computed as per the standard Pythagorean equation:

$$X[k]^2 = X_R[k]^2 + X_I[k]^2 \quad (2.7)$$

The square root is usually not calculated, as it provides no additional information.

The main advantage of the DFT over the FFT is that the calculations are spread out over the entire sample block. All samples must be stored in memory before FFT calculations can be performed whereas the DFT can be calculated one sample at a time, because there is no linkage between samples. This advantage is negligible when considering the speed of the DFT compared to the FFT. By using big O notation to express the arithmetical operations used by the algorithms the reason becomes apparent why the FFT is used. FFT requires  $ON * \log N$  operations while the DFT requires  $O(N^2)$  operations [10].

The most commonly used FFT algorithm, the Cooley-Turkey algorithm, is in its most common form (radix-2 decimation-in-time) where the DFT has been rearranged into two pairs, a sum over the even-numbered indices  $n = 2m$  and a sum over odd-numbered indices  $n = 2m + 1$ :

$$X_k = \sum_{m=0}^{N/2-1} x_{2m} e^{-\frac{2\pi i}{N}(2m)k} + \sum_{m=0}^{N/2-1} x_{2m+1} e^{-\frac{2\pi i}{N}(2m+1)k}. \quad (2.8)$$

from here the common multiplier  $e^{-\frac{2\pi i}{N}k}$  is factored out:

$$X_k = \underbrace{\sum_{m=0}^{N/2-1} x_{2m} e^{-\frac{2\pi i}{N/2}mk}}_{\text{DFT of even-indexed part of } x_m} + e^{-\frac{2\pi i}{N}k} \underbrace{\sum_{m=0}^{N/2-1} x_{2m+1} e^{-\frac{2\pi i}{N/2}mk}}_{\text{DFT of odd-indexed part of } x_m} = E_k + e^{-\frac{2\pi i}{N}k} O_k. \quad (2.9)$$

which leads to:

$$X(k) = \begin{cases} E_k + e^{-\frac{2\pi i}{N} \cdot k} \cdot O_k & \text{if } k < N/2 \\ E_{k-N/2} + e^{-\frac{2\pi i}{N} \cdot k - N/2} \cdot O_{k-N/2} & \text{if } k \geq N/2 \end{cases} \quad (2.10)$$

Without going into too much detail on the algorithm, the implementation basically expresses the DFT of length  $N$  recursively in terms of two DFTs of size  $N/2$ , which is the core of the radix-2 DIT Fast Fourier Transform. The speed gain comes from re-using intermediate results from previous computations, hence the need for the whole sample block in memory.

The advantages with FFT over digital filters, such as a Finite Impulse Response filter, are the same as with the DFT. Computing one  $N$ -tap FIR filter requires  $N$  samples and  $N$  coefficients to be stored in memory.  $N$  multiplies and  $N$  summations are required each sample period. This limits the frequency ranges and the frequency discrimination in the range for a given CPU due to the increased computation burden [5].

The actual knock values used in this thesis are the result of some signal processing in the corresponding CCM module of the cylinder. The samples are collected from the sensor through a 16-bit A/D converter at a sampling frequency of 25 kHz. Samples are only collected during a preconfigured measuring window, usually between top-dead center and 50 crank angle degrees.

These signals are offset so the resulting signal is oscillating around zero. After that the signal is given a preconfigured gain (due to sensor mounting, mechanical variations, etc). The signal is zero-padded as the FFT requires the number of samples to be the power of two. The signal is windowed with a Hamming window before the FFT is performed. Magnitudes are calculated and the magnitudes of the preconfigured frequencies are identified. The average of these magnitudes is then sent to the application level where the detection strategy is applied.

### 2.4.2 Detection strategy

There are several knock detection strategies with varying complexity, sophistication and computational overhead.

A simple strategy is to use the magnitudes generated by the signal processing for the

detection algorithm [11]. More complex strategies can involve rate of change of the magnitudes used with predictive algorithms or using fuzzy logic as a control strategy.

The simplest strategy involves comparing the magnitudes to predefined reference levels to indicate knock intensity. Wärtsilä uses three knock levels: none, light and heavy [12].

If the knock status of a cylinder is light knock the ignition timing of that cylinder will be retarded. The ignition timing is retarded until the knock status is none or until the max value of the offset is reached. If the cylinder is still in a state of light knock the engine control system will start retarding the main fuel injection (MFI) duration until the knock status is changed from light to none.

If the knock status is heavy knock the MFI duration is retarded instantly and for SG engines the ignition timing is retarded to a minimum instantly. Heavy knocking also involve safety functions that can start a shutdown of the engine.

In some applications a preventive control strategy might be desirable using the rate of change of the magnitudes. The predictor can be as simple as to calculate the difference between the current and the previous measurement. This difference is the one compared to reference values. This approach is not always optimal as the cycle-by-cycle values can be quite varied and it would lead to problems when defining reference levels and a possibly unstable system. A solution to that would be greater time difference between the measurements but that would lead to slower reaction time in the control system. Changing the engine speed would have significant effect on the result.

## 3 Python

Python is a general-purpose, dynamic, garbage-collected programming language used in a wide variety of application domains. Python supports multiple programming paradigms, primarily object-oriented but the imperative style can be used without extra effort and supports to a lesser extent a functional style of programming.

Python syntax utilizes whitespace indentation, instead of curly braces, to delimit blocks of code resulting in readable code by convention.

Python uses duck typing (more details about that later on in the thesis) and has typed objects but variables are not explicitly typed. In Python everything is an object[13] and that is one of the cornerstones of the language. Python has a dynamic typing system that takes care of variable-types and are automatically declared at run-time and can change during the execution of the program.

The Python programming language has a wide adoption across many domains. There are specialized libraries for many different things from scientific, numerical and even symbolic calculations to different Web frameworks.

Python is backed by a large open-source community and many big corporations around the world use Python, such as Google, CERN and NASA. Presently the creator of the language, Guido van Rossum, works for Google and has been named "BDFL"(Benevolent Dictator For Life) which means he has the final say on every decision regarding the language.

Python is also used as a backend language for SAGE (mathematical software) which has a bold goal which the website states:

*Creating a viable free open source alternative to Magma, Maple, Mathematica and Matlab.* SAGE motto

SAGE uses nearly 100 libraries with capabilities for high quality plotting and complex calculations.

In my opinion, one of the drawbacks with Python is that the code is not compiled in the general sense of the word and no errors, beside indentation and import errors, are caught before running the program and then only when the erring method is called. This leads

to the need of rigorous testing of every single function and class if developing critical applications, because Python will crash spectacularly when trying to for example add the integer 1 to a string as shown in the next section. For testing purposes Python includes modules for unit-testing and these are the recommended way to prove conformity to specifications.

### 3.1 Typing system

In Python everything is an object. An object is an entity that has an identity, a value, a type and one or more bases. This is more easily explained by an example:

```
>>> two = 2
>>> type(two)
<type 'int'>
>>> type(type(two))
<type 'type'>
>>> type(two).__bases__
(<type 'object'>,)
>>>
```

Here we give an integer the name *two*, it has a type of *int* and we see that the *int* type is of the type *type*. Finally we see that it is based on an object.

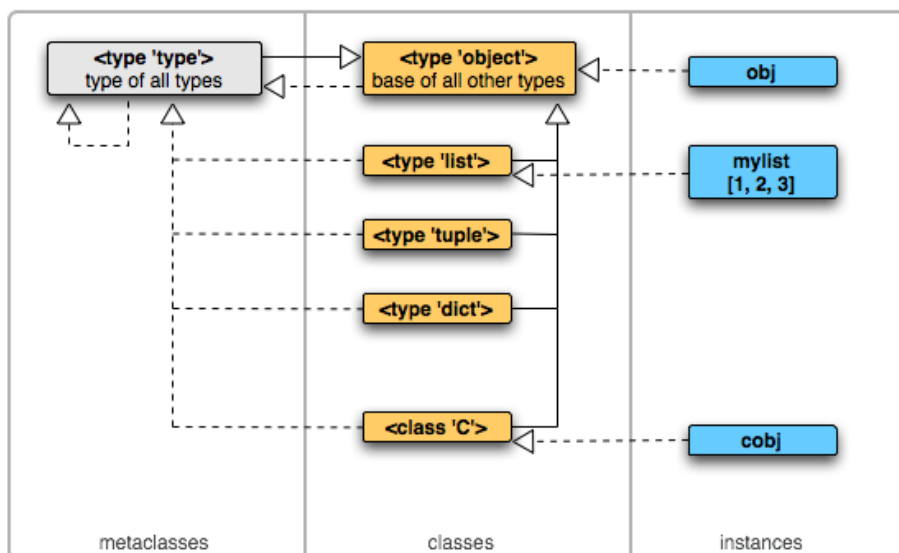


Figure 4. Simplified Python type map

Figure 4 shows a basic object map of the Python language. This is again more easily explained with an example:

```

>>> list
<type 'list'>
>>> list.__class__
<type 'type'>
>>> list.__bases__
(<type 'object'>,)
>>> tuple.__class__, tuple.__bases__
(<type 'type'>, (<type 'object'>,,))
>>> dict.__class__, dict.__bases__
(<type 'type'>, (<type 'object'>,,))
>>>
>>> mylist = [1,2,3]
>>> mylist.__class__
<type 'list'>
>>>

```

Here we have the built-in *list* object which we see is a type of *type* and is a subclass of *object*. This is the same as the *tuple* and *dict*. Finally we create an instance of the *list* class and its type is *list*.

Python uses duck typing which is a style of dynamic typing. With duck typing an object's attributes determine the valid semantics regarding the type, rather than explicit relationship to some type object. The name refers to the duck test, which can be phrased as follows:

*When I see a bird that walks like a duck and swims like a duck and quacks like a duck,  
I call that bird a duck.*

James Whitcomb Riley

Duck-typing proponents say that with duck-typing the programmer avoids tests using *type()* or *isinstance()*. Typically this means that Python employs the EAFP (Easier to Ask Forgiveness than Permission) style of programming. The result of this is shown in the next section.

## 3.2 Interactivity

Python comes with a built-in interpreter and is great tool for testing code, running small calculations or executing Python scripts. Python scripts are written as normal text with the extension *.py*. You can write functions beforehand in a normal text editor and import the file to the interpreter workspace and execute the functions interactively in the interpreter. Most of the code developed for this thesis has been run in the interpreter as a stand-alone function before being implemented in the main program.

```

>>> 2 + 2
4
>>> def sum(a, b):
...     return a + b
...
>>> sum(1, 2)
3
>>> sum(1.8, 2.9)
4.7
>>> sum("hello ", "world")
'hello world'
>>> sum(1, "hello")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in sum
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>>

```

The example above shows the simple use of the interpreter, defining (def) of a function and the duck-typing in effect. The function does not care what types are used with the caveat that they mix together. The example also shows the traceback of an error when using the function illegally.

The defined function *sum* is actually a built-in function of Python and is used for summing up the contents of a list.

The function defined above could be made somewhat more type-safe by trying to type-cast the return-values to matching types. This goes against the "Pythonic" way of doing things, but has to be used in some cases to ensure that the correct type is returned from a function/method.

```

>>> def sum(a, b):
...     return int(a) + int(b)
...
>>> sum(1, 2)
3
>>> sum(1, "2")
3
>>> sum(1.8, 2.9)
3
>>> sum(1, "hello")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in sum
ValueError: invalid literal for int() with base 10: 'hello'
>>>

```



As shown above this would work with strings that can be changed to an integer but would give incorrect answers when using double, as the double function (everything is an object, even types) takes the integer part from the input and returns that, although not crashing. As before you really cannot add an integer, in this case 1 to the string "hello" in a way that would make sense.

### 3.3 Standard library

Python includes a large standard library (STL) and is often jokingly described as a batteries included Python philosophy, although the STL is not mandatory for using Python. The STL includes many tools for many different tasks.

The Python STL includes modules for a vast array of different applications ranging from numerical and mathematical modules to modules for data persistence (databases). The Python STL is used in this thesis for almost everything beside the graphical user interface(GUI) even if the STL includes bindings for a GUI toolkit that could have been used but was not, for reasons discussed in the next section.

#### 3.3.1 Extending Python

Extending Python is easy with the Python Package Index (PyPI). During the writing of this thesis there were 18212 packages in the package index. Python packages or modules can be written in either C (Python is written in C), Python or C++. Package managers for the use of this package index have been written and the one that seemed the most popular at the time of writing was "Pip Installs Packages. PIP is easy to use and has a simple syntax for installing, updating and uninstalling packages from the PyPI.

### 3.4 wxPython

wxPython is the graphical user interface (GUI) toolkit used in this thesis and it is a wrapper for the wxWidgets toolkit.

The wxPython toolkit was chosen because it is bundled in the WECSplorer Wizard package which provides the Python environment used in this thesis. The Python STL includes bindings for the TK/TCL toolkit but as the toolkit is somewhat cumbersome to use, especially for complicated programs like the one developed in this thesis, and as there was a lack of advanced widgets, the decision was made not to use it.

Other notable contenders for the position of the GUI toolkit were Python wrappers for the Qt-framework (PyQt, PySide). PyGTK was ultimately not even considered because of the extra dependencies and it would not have any real advantages either. PyQt and PySide are excellent toolkits for developing GUIs and using the GUI designer Qt-Designer could

have notably shortened the development time of the program GUI. PyQt or PySide would also have been "future-safe", as the next version of WECSplorer (UNITool) is based on Qt. Ultimately it was decided against these to keep the package size down and to simplify deployment.

In this thesis the GUI is hand-coded although some tools exist that could help in the designing and which had some functionality for automatic code generation. However they were not used, as the basis was to learn as much as possible and as the design-tool had a few shortcomings.

### 3.4.1 wxPython architecture

wxPython is a Python wrapper for the C++ wxWidgets cross-platform GUI toolkit. wxWidgets in turn is a wrapper for the platform native libraries using the platform specific APIs (win32, cocoa, gtk, etc.) to generate windows and widgets as seen in Figure 5. The native APIs tend to be too cumbersome for general use, hence the wrapper for an abstraction layer between the programmer and the native libraries [14]. This gives wxPython a native look on every platform and the same code can be run on different platforms without any major changes.

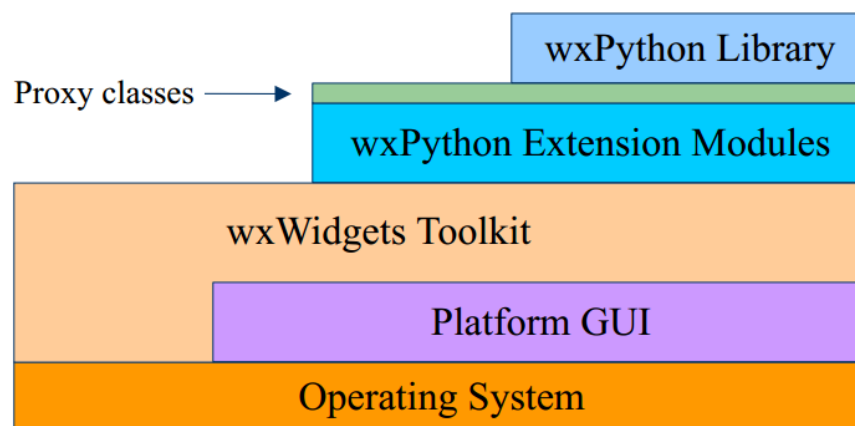


Figure 5. Simplified wxPython architecture

### 3.4.2 wxPython modules

wxPython consists of five basic modules:

**Windows** The windows module consists of various windows (panels, dialogues, frames and more), which form the base of any program.

**GDI** The Graphics Device Interface is a module with a set of classes that relate to the manipulation of fonts, colors, images and more.

**Core** The core module consists of the base classes used in development e.g. the object class which is the base class for all other classes in wxPython, sizer classes that control widget layout, event classes etc.

**Misc** Miscellaneous classes and functions e.g. logging, system setting etc.

**Controls** The controls module consists of the common widgets detailed below.

Widgets in wxPython can be divided into some logical groups e.g. base widgets, top level widgets, containers, dynamic widgets and static widgets. Examples of a few widgets that could be considered dynamic widgets are buttons, toggle buttons, check boxes, combo boxes and scrollbars.

### 3.4.3 wxPython inheritance

In wxPython there is a relation between widgets. This relation is from the inheritance of the base classes. Inheritance is a part of the object oriented programming paradigm. Widgets form a hierarchy because a widget can and will inherit functionality from parent widgets. Using a button as an example of the inheritance:



Figure 6. Inheritance of a button

Figure 6 shows the button widget which inherits functionality from four different base classes. The closest class is the *wx.Control* class which consists of methods related to control. After that the *wx.Window* class which gives the appearance to the button and the *wx.EvtHandler* class gives functionality related to events to the button. The final class is *wx.Object* which is the base class of all widgets in wxPython.

Compared to for example a dialogue class, an instance of that class would not have inherited from the control class as it has no control elements.

### 3.4.4 wxPython events

Like most GUI toolkits, wxPython is an event-based toolkit which means that every event on the GUI must have an action bound to it. Events in wxPython are handled differently compared to its competitors PyQt, PySide and PyGTK. Events in PyQt and its likes are connected directly to functions/methods via a signal/slot system meaning for example that all buttons have different signals that are triggered with different actions. These signals are bound to slots which in turn trigger the methods/functions that are connected to the slot.

wxPython handles events in a more complicated way, which can become confusing at times. When an action occurs the event is propagated through the originating widget all the way through its parents until a bound method is found.

The *Bind* method is used to specify event bindings in wxPython. It tells the system what kind of event to look for and also where to look for it. For example, in this code where *self* is a *wx.Frame* or another container control,

```
self.Bind(wx.EVT_BUTTON, self.OnButton, self.button)
```

the binding tells the system to call *self.OnButton* when it sees a *wx.EVT\_BUTTON* event delivered to the frame from *self.button*. The button may be many layers down in the hierarchy and hence the event has to propagate through all the layers of the GUI.

The other binding behaves somewhat differently when compared to the first:

```
self.button.Bind(wx.EVT_BUTTON, self.OnButton)
```

This binding method tells the system to immediately call *self.OnButton* for a *wx.EVT\_BUTTON* event that originates from *self.button*. This then clears the event and no parents have any possibility to see the event unless the method *Skip()* is called. This can be a wanted trait but sometimes it can be beneficial to have the event propagate through the layers and trigger different event handlers as with the first binding method.

## 4 Development and realization

This chapter details the implementation of the Knock Tuning Wizard program made with Python using wxPython as the GUI-framework. The program is executed from the WECSplorer interface and runs within the built-in Python environment.

### 4.1 Design

The design principle for the project has been iterative as no base design was ever formally written. During the writing of this thesis, the source code has had two major rewrites of most methods and functions, changing the layout of the program and improving the readability of the code. Adherence to the PEP8 and standard design principles of object oriented programming formed the basis. However, without experience in object oriented principles and graphical user interface design, the project has been changed and adapted to new specifications throughout the process. The program could have been laid out as a procedural program without the need of any classes, not including the user interface, but the design was more coherent with classes and provided some encapsulation. The whole project is loosely based on the MVC design pattern, which has diminished with each iteration of the program.

A few key points/functions to the program had specific requirements, detailed in later sections of the thesis, and the other functionality has been added as found necessary.

#### 4.1.1 Graphical User Interface

The design of the GUI was established quite early in the project as the program heavily depends on interactivity with the user. The initial thought was to follow the standard layout for a wizard, i.e. one screen after another. However, this paradigm does not really work when tuning an engine as the user has to see old data and be able to jump from cylinder to cylinder without a predetermined pattern of tuning.



Figure 7. Wizard paradigm

After failed experiments with the wizard paradigm in the beginning, a new design approach was tried with a base main window that spawns child windows as needed. For example, for the actual tuning process, a new window corresponding to the selected cylinder needs to be spawned when the actual tuning is to be started. This paradigm is what most users should be familiar with and be able to use without any extensive training.

A few crude sketches were made and finally coded with wxPython to an easily modifiable GUI. The GUI is designed so that every widget is inside a sizer which automatically resizes the widgets and keeps their placement relative to the window.

The wxPython framework contains many different sizers, for example the `wx.BoxSizer` is a sizer that depending on a parameter at initialization, will place the containing widgets horizontally or vertically. Other sizers used in this project were different grid sizers which put widgets in a grid with spacing defined with parameters. The use of sizers instead of static placement will enable the use of the program on different screen sizes and full-screen without misaligned widgets ruining the user interface.

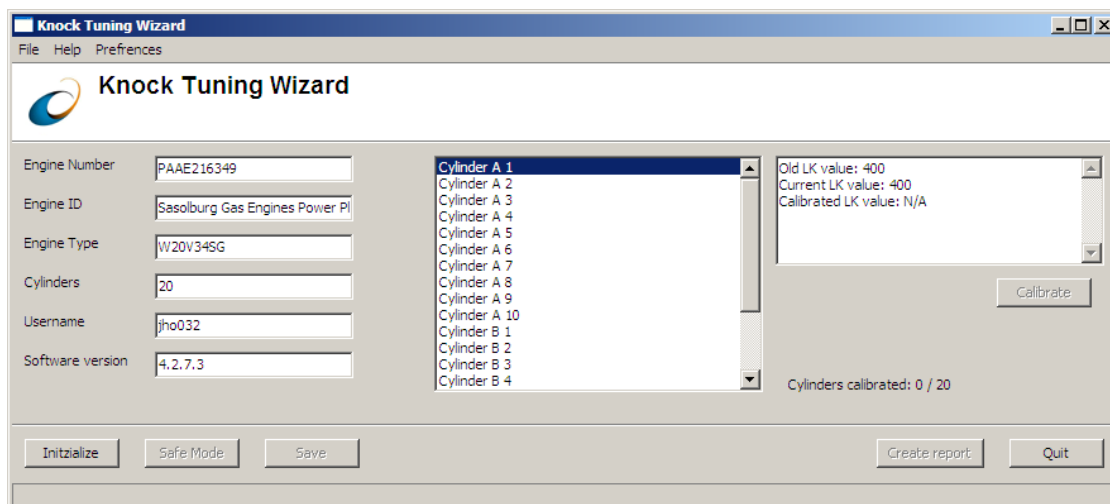


Figure 8. The main window of the program

#### 4.1.2 MVC - Model-View-Controller

The MVC design pattern creates applications that separate different aspects of the application. This design pattern is excellent for GUI programming and web-pages as the UI (the view) are separated from the variables (the model) and the logic (the controller) of the program. The project is based on this as the code for storing most of the variables is separated from the logic of the main program. This would be the equivalent of the model in MVC.

The eventhandler which is the main class in the project could be seen as the controller, because it contains most of the bound functions from the GUI in the project and delegates information from the model to the view. The UI (view) is mostly separated from everything else and contains a minimal amount of functions related to the actual runtime of the

program. A few key methods still exist in the UI files but these are mostly methods that kill the window internally. In this context the model part of the MVC paradigm would be the engine and cylinder classes, which contain some necessary logic but act mostly as data storage.

Separating the view from the rest of the code enables future maintainers of the program to take the logic without any user interface dependencies and with minimal rewriting to implement another GUI-framework such as PyQt.

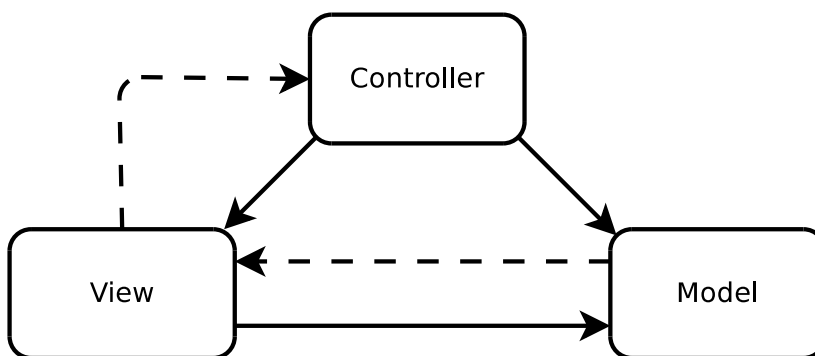


Figure 9. The MVC design pattern

### 4.1.3 PEP8

PEP8 (Python Enhancement Proposal) is the style guide recommended for Python developers to keep the code clean and readable. The style guide recommends for instance four spaces for indentation and no tabs to keep the code looking the same on every text editor. PEP8 also recommends the programmer to keep the line length under 79 characters, which might seem odd and is a bit restricting considering today's monitors. Nevertheless there are instances where this limitation will be good for everyone that might read the code from somewhere, especially with newer laptops and tablets that do not have the screen real-estate to read really long lines of code.

## 4.2 Class and module design

As has been mentioned earlier only a few functions had specific requirements. These are detailed in the next section and the rest of the requirements were implemented when considered necessary.

Classes were designed to reflect the physical world, i.e. one class for the engine and one class for the cylinder. The cylinder class is then instanced according to the amount of cylinders on the engine at runtime. One main class for the entire program was designed containing the logic of the program. The API was designed as a module to encourage code-reuse.

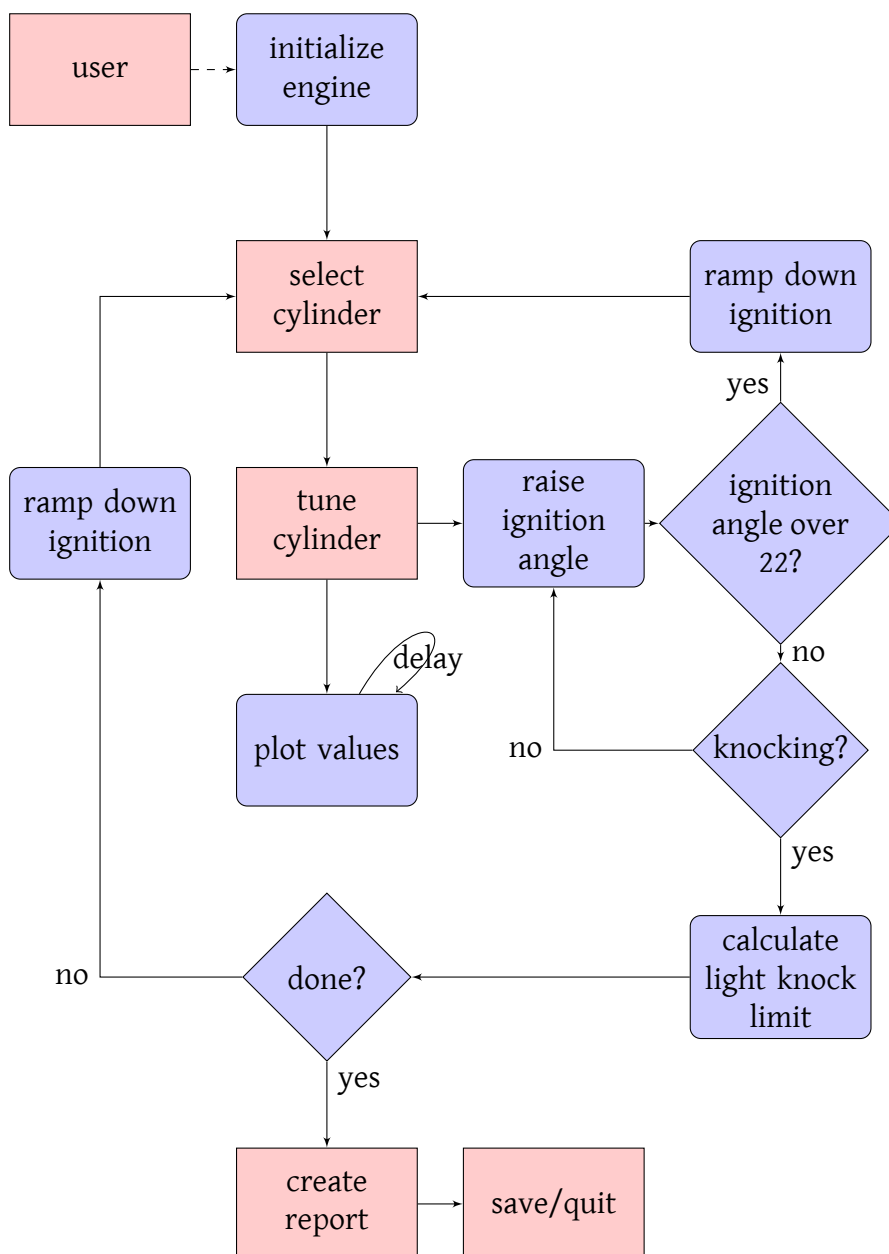


Figure 10. Flowchart representation of the program

#### 4.2.1 API module

The core of the program is the API. The API connects to the WECSplorerUT application which connects to the engine for live values. The API uses a COM (Component Object Model) interface to connect to the WECSplorerUT application.

This requires the connection to be initialized before using the API methods with the Python module *win32com*, thus no native Python connection exists.

The original API uses a syntax close to this:

`function_name("module_name", "dataname_of_symbol", row, col, amount)` and as such is quite cumbersome to use when called many times. Most of the API methods also have string values as return types, which may lead to some unwanted behavior in this



specific Python application. To simplify the whole procedure a Python wrapper for most of the API functions was written, with explicit return values depending on the called method. This goes against the duck-typing feature of Python as string values would be coerced to integer or float types depending on how the value is used. However, to simplify and minimize the need for error handling, a stricter type design was used. This design is still being evaluated and might be changed to a more flexible model which uses duck-typing in a later version of the program.

The wrapper is also designed with some logic in its initialization method. When initialized the wrapper determines the used module name for the engine, if an older configuration should be used and if the engine is using UNIC or WECS. This is done to select the correct configuration file containing the data-names for the symbols used in the program.

A few examples of the API wrapper methods for fetching an integer either as a single value or from a matrix:

```
api.get_integer("dataname_for_symbol")
```

returns an integer through the *win32com* connection with the row and column values of zero.

```
api.get_integer_matrix("dataname_for_symbol", row, column)
```

returns an integer from the matrix at row and column values which are usually automatically determined by the active cylinder class instance detailed later.

### 4.2.2 Engine class

The engine class is designed to mostly just contain the data relevant to the engine and to the program. The class could be designed as a simple dictionary and a few methods in any other class or a dictionary and a few functions. To keep the naming and design close to the physical design a class seemed to be a good choice. The overhead of using a class, because it can be slower in some cases, has no significance because it has no time-critical methods. The most important method that the class has and which is critical to the functionality of the rest of the program is initializing the API. The class is one of the first instanced at startup which in turn will trigger the API to connect to the WECSplorer program.

The methods in this class use the API for simple data gathering of values relating to the whole engine which are used later when creating the report of the tuning process. The gathered values include:

- Engine parameters
  - Engine nickname
  - Engine number

- Engine type
- ...
- Variables
  - Load during test
  - Global ignition angle
  - Turbocharger speeds
  - ...
- ...

This list is of course not a comprehensive list of variables stored in the class.

### 4.2.3 Cylinder class

The cylinder class is the controlling class for everything related to cylinders. This includes the logic for increasing the ignition angles, calculating the light knock limit and more.

One instance is created for every cylinder on the engine. These instances hold data gathered during the tuning process, e.g. the final ignition angle before calculating knock, the data names for the knock sensor for the specific cylinder and more.

The most critical function of the class is the calculating of the light knock limit and the ramp down methods for the ignition angle. The class also has some failsafe methods, such as writing a small XML file with some relevant data after the cylinder is tuned. This is so data can be recovered if for some unforeseen reason the program or something else crashes, which would lead to data loss of everything done to the engine. This check is done while instancing the classes and, if any data exists that is not over two days old, it will be recovered and the user can continue tuning the cylinder from where he left off.

This class uses the API extensively for many different functions, from simple data gathering to setting the ignition angle offset for the current cylinder. Even if the engine class is the one creating the connection for the API, the cylinder class uses the connection most extensively.

Variable types in this class are enforced in the extension that Python allows for, i.e. everything is type-casted to ensure the correct type. This is to negate issues relating to for example the ignition angle, which is used in some methods as a string and in other methods as a float and even sometimes as an integer.

### 4.2.4 Eventhandler, the main class

The brains of the program are collected into a single class. This class handles all the different windows and dialogues in the program, creating the instances, passing the

variables to them and finally destroying them when finished. The class also handles the instancing of the engine and cylinder classes and the main window of the program which is the parent to all the dialogues and the subwindows and is the bind point for 98% of the events in the program.

The class is designed procedurally so that the source can be read from top to bottom and the methods and events should, but do not always, correspond to the execution of the program itself. Because the class does not have direct access to the API it just acts as a middle ground for the events propagating from the user interface that are bound to local methods, which in turn call on engine or cylinder class methods with API access. The return values from those calls are usually passed directly or processed and passed to the user interface.

Some critical methods do exist in the class such as the handling of the timers that poll for the knock value from the engine (these are in the cylinder class but are called from the eventhandler class). The class also keeps checks on the ignition angle and aborts if the angle rises to too high a value. Another important function is to consolidate all the data that gets passed to the functions creating the report. The data is gathered from the engine instance and the cylinder instances.

#### **4.2.5 External data handling**

The program must be able to handle some external data in the form of configuration files used for setting the variables for the program instead of hard-coding them into the program. This is to ease the changing of values without diving into the complete code.

The external files are written in XML by a Python STL module *elemtree*. The configuration file contains different settings used throughout the program, which can also be modified during runtime. Other XML files contain data names (the raw path for a variable used in WECSplorer) for the many symbols used in the program.

The class is mainly designed for parsing the configuration file but it also creates the project folders if none exists because it is one of the first class instanced. These folders will contain screen-shots and general cylinder data gathered after calibration and saved for later access. This data will be used as a failsafe, which means that no data will be lost if the program or WECSplorer crashes and also on larger tuning projects that can't be finished in a day. The data will stay valid for two days, so the user of the program can continue with the tuning of the engine where he left off, if for instance, the tuning did not finish in one day.

The two day limit is there to minimize the external conditions (temperature, humidity, etc.) from skewing the tuning process. If the two day limit is exceeded, the program simply ignores any data saved.

The main configuration file is parsed fully and the contents are divided into dictionaries according to the element/subelement they are in. These dictionaries are then passed

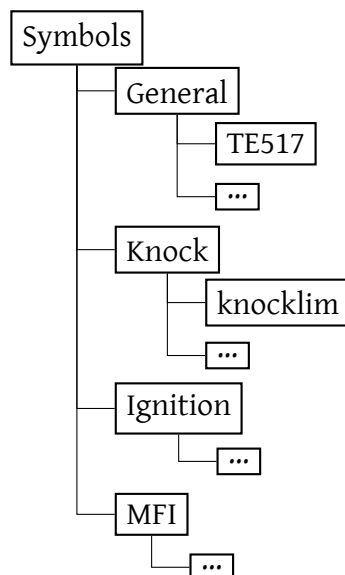


Figure 11. Node system in the XML files

directly to functions which themselves get the setting/dataname they need. The other option is to directly access the required setting/dataname key and passing the return value to the function.

#### 4.2.6 Data visualization

The program has to have some means of visualizing the knock values in real time. The most reasonable way was to embed some plotting library into the program. Some alternatives were tested such as *matplotlib* and some other alternatives.

*Matplotlib* produces excellent quality graphics and was the prime contender for the spot, but it was too slow in plotting values in the time frames that the program requires. This update-rate may be under 50 ms. The implementation would also have been less than ideal, regarding the code needed and integrating it into the user interface.

The final library used was the wxPython built-in *wx.Plot* which was fast enough for the program and easy to use and implement in the program. The only issues were that the WECSplorer Python distribution contained an old version of the *wx.Plot* file. This was solved by including a replacement in the program and using that instead of the integrated version.

The data that the plotting library has to handle is in the form of list of tuples containing the sample number and the knock value for that particular sample. This list is extended to a predefined number of values. After the list has been filled to the limit, the oldest value is discarded as new values are added to the end of the list. The list can be described as a FILO list (First In, Last Out).

This data is supplemented with a running average calculated for a predetermined but

|           |   |           |             |
|-----------|---|-----------|-------------|
|           | [(1,400), (2,564), (3,453), (4,537), (...)] | (200,486) | first plot  |
| (1,400),  | [(2,564), (3,453), (4,537), (5,570), (...)] | (201,510) | second plot |
| discarded |   |           |             |

Figure 12. The data list

configurable window as follows

$$\text{average} = \frac{1}{N} \sum_{i=0}^N x(i) \quad (4.1)$$

where  $x(i)$  is the list of tuples and N is the amount configured for averaging. The result can be seen in Figure 13.

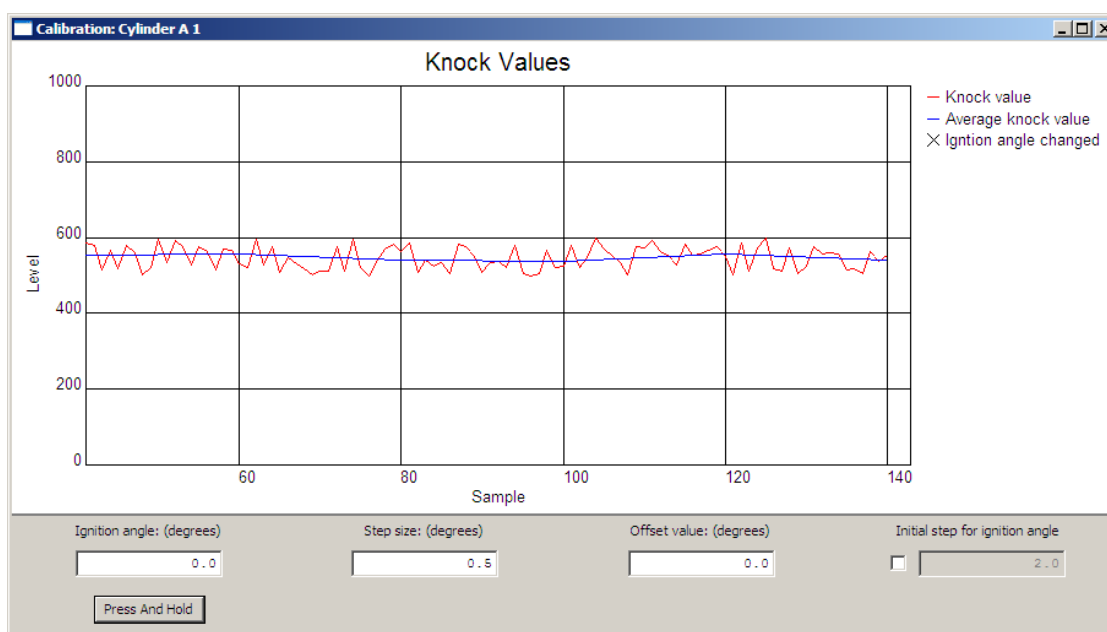


Figure 13. The tuning screen

Finally the calculated light knock limit is appended as a line to the plotted data which is seen in Figure 14. There is also another line in the same figure indicating the user changed light knock limit (if the calculated limit for some reason is not correct).

This was implemented because a suitable limit is hard to calculate without assumptions and sometimes a heuristically determined limit might be more suitable.

### 4.3 Critical requirements

The program has few requirements but they are critical for the functionality of the program and for avoiding damage to the engine. The most critical of these is the controlling of the ignition angle when a knock occurs, which can seriously damage the engine and its parts.

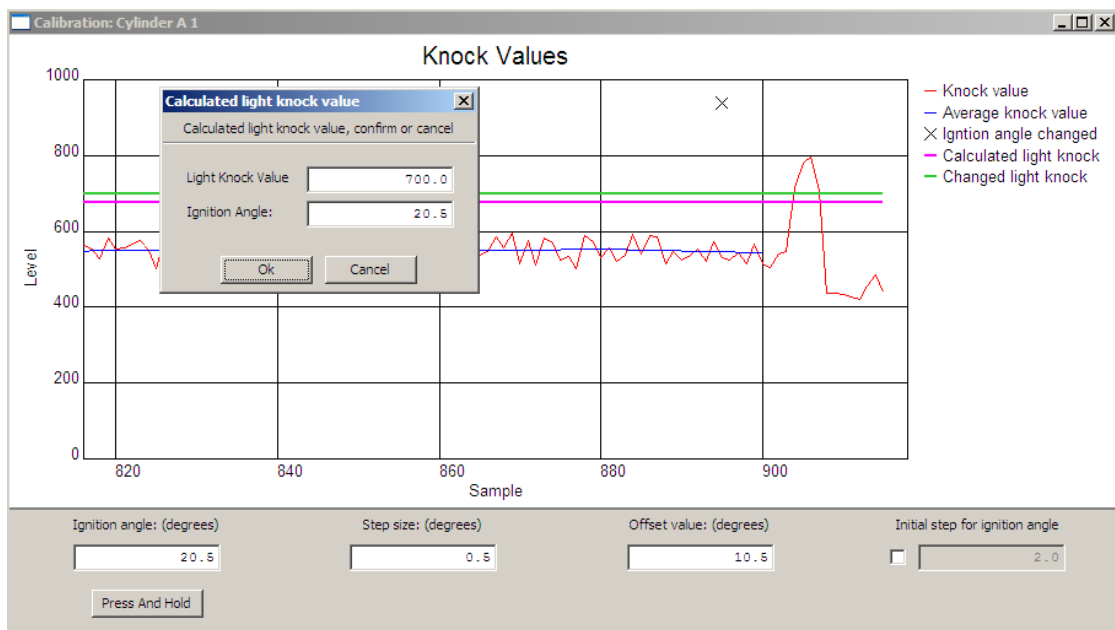


Figure 14. Tuning complete

### 4.3.1 Ignition angles

Ignition angles on an engine cannot be increased more than five degrees at a time because of the possible issues with the timings on the cylinder as detailed earlier. This is usually not a problem as the ignition angle is usually incremented in steps of 0.5 degrees. Because of the small (default) step-size the need for an initial step became apparent. This allows the users to increment the first step up to four degrees to decrease the time to the knocking threshold.

When the tuning is done the ignition angle has to be brought down as fast as possible to the original angle. When the knocking threshold has passed one generally does not want to stay in that situation. Here there is also the same problem with the limits on the step-size of the ignition angle. This is avoided by ramping down the ignition angle in a few iterations, which is done without user interaction and only if the angle is raised more than five degrees during tuning.

There is a hard coded maximum limit of 22 degrees offset on the ignition angle. If the engine is not in a knocking state at this angle there is most likely something wrong with the data gathering from the sensor or knocking will not occur at all.

The reasons for this might be among other things the high quality of fuel or some other external things. The general consensus is that if no knocking occur before 22 degrees offset there is no point in increasing it more as the information will be unreliable. At 22 degrees the tuning will fail with an error message to the user and ramp down the ignition angle.

### 4.3.2 Setting the knock limits

When the engine is initialized for tuning the light knock limits for every cylinder is raised to 1000. This is done to disable the engine control system from disturbing the tuning process, which requires getting the engine to a knocking state where the engine control system should activate and try to bring the engine out of the knocking state. If the light knock limit was not changed the tuning would be impossible.

The values are reverted to the values that were saved before raising the limits when the program exits, unless the user specifically saves the new tuned values. When saving, the user is given a dialogue with the calibrated cylinders to choose from. The choice is to select which of the revised light knock limits are to be permanently saved in the engine control system. Even the heavy knock limit must be raised for the duration of the tuning, but this

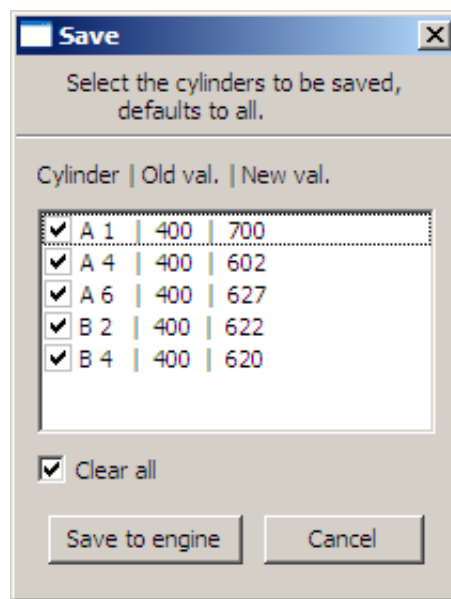


Figure 15. The saving dialogue

is only done cylinder-wise. This is to prevent accidental shutdown of the engine if the unaltered heavy knock limit was reached. The engine control system automatically goes for a shutdown if heavy knock is achieved. This could be considered a catastrophic failure. A shutdown of the engine is to be avoided at all costs, as it is time consuming to start up an engine and the engine isn't really in an uncontrolled state of heavy knocking.

### 4.3.3 Calculating the light knock limits

The method that does the calculating of the light knock limit suggestion works with the data available, the saved knock values. The data used for calculating the light knock limit is the last 30% of the knock values. Those values should hold the value of the knock that caused the user to stop the tuning process and they should minimize the effect of the rising mean value which occurs with a raised knock angle. These values are stored and accessed from a list of values containing tuples which in their turn contain both the

sample number and the knock value. These are accessed with a simple list comprehension statement.

```
data = [x[1] for x in self.dataPoints[-(int(0.3*len(self.dataPoints))):]]
```

This statement takes the values with the index 1 from the last 30% using list slicing. The values are used to calculate the standard deviation of the knock values with

$$\sigma = \sqrt{\frac{1}{n} \sum_{i=0}^n (x_i - \mu)^2}, \text{ where } \mu = \frac{1}{n} \sum_{i=0}^n (x_i) \quad (4.2)$$

The actual code does this in three steps, first by calculating the mean

$$\text{mean} = \sum_{i=0}^n \text{data}_i \quad (4.3)$$

Then the mean value is used with a lambda expression to calculate the variance which is mapped over the data extracted earlier. Mathematical notation would be

$$\text{mapped} = \sum_{i=0}^n \text{data}(i) - \text{mean}^2 \quad (4.4)$$

This result is finally summed up and squared for a standard deviation.

$$\text{stdev} = \sqrt{\frac{1}{n} \sum_{i=0}^n \text{mapped}_i} \quad (4.5)$$

The standard deviation is just used as a fail safe for the actual calculation tries to take all knock values that have occurred since the last two raises of the ignition angle using list slices.

These values are sorted in a descending order which means using a sort function and reversing the output. From these values the highest ones are saved (up to 40 values) and using the top few (four by default) a mean value is calculated and then multiplied to get a limit of percentage of the top (90% by default).

$$\text{result} = \frac{1}{4} \sum_{i=0}^4 \text{top}_i \cdot 0.9 \quad (4.6)$$

The failsafe then checks that the result is one and a half standard deviations over the mean value, otherwise the light knock limit suggested will be two and a half times the standard



deviation over the mean. This is to ensure that the suggested light knock limit is never under the standard deviation of the knock values, as that could lead to confusion among the users and in a worst case scenario a cylinder could be set with a light knock limit that is too low for efficient running and sooner or later it would require re-tuning.

## 4.4 Reporting

Reporting for calibrated engines is required. It is generated with the open-source *reportlab* package for report generation. The function generating the layout for the report is passed a dictionary with all the relevant data.

The main page of the report is designed so that it contains a table with general information about the engine. Then one or two additional tables containing information about the calibrated cylinders including old light knock limits, new light knock limits, MFI offsets and more.

The report also includes detailed pages for every cylinder that has been calibrated on the engine, including a screen-shot of the final plot which shows the latest knock values and indicates the selected limit.

The actual layout for the report is handled by three functions. One for laying out the first page which includes definition for page widths, logo placement and more. The second function lays out the pages after the first, which uses almost the same parameters as the first function. This means all pages in the report use the same margins and widths. The last function adds the content which is just appended to a list with different parameters and the whole list is then parsed by the *reportlab.SimpleDocTemplate* method which generates the final PDF file.

## 4.5 Documentation

Every project needs documentation or at the very least it should have some sort of manual. For this program a user guide and a developer guide were compiled. The user guide contains images of the program and a walkthrough of tuning a cylinder. It was compiled with Sphinx, a Python documentation generator.

The developer's guide was also compiled using Sphinx. Sphinx uses RST files for content and supports auto-documentation of source files. Sphinx is capable of generating HTML pages or PDFs using  $\text{\LaTeX}$ .

The auto-documentation functionality parses source files and generates professional source documentation that includes non-private methods and classes and their respective doc-strings. Doc-strings are comments directly below the declaration of a function that usually explain how a method/function works or should be used.

An example of the output could look something like this:

```
get_ign (dataname)  
Returns the value from the Ignition Manual adjust map  
Parameters dataname (str) – The dataname of the ignition angle variable.  
Return type float.
```

*Figure 16. Sphinx auto-documentation output*

The Sphinx generator eases the source documentation as it parses the actual source of the program and changes are automatically included on a new compile of the documentation. The downside with Sphinx, which might lead to the need of an alternative documentation that is essentially a copy of the generated documentation, is the need of an  $\text{\LaTeX}$  environment installed on the developer's computer. This could be an issue with Wärtsilä computers, as no compiler exists in the standard software allowed on the computers.

## 5 Results

The result of this thesis is a functional program that complies with the requirements and goes beyond the requirements in some respects. The resulting code is about 3000 lines of source code not counting the ReportLab library and the newer wx.plot classes that were incorporated into the program. Beside the actual program a user guide with basic usage and developer guide with detailed information about the program implementation have been written, which are to be included in a more public release. The actual developer's guide might be redundant as I will most likely maintain the actual program in the future.

The program is structured in a way that the program is split into different files, according to the functionality of the program. The program could have been written in a single file, but it would have been quite hard to maintain in the long run. As an example the user interface files are in a separate folder and are split to mostly one window per file, and so the folder has five different files each corresponding to a window in the program. This simplifies the following MVC paradigm, which dictates that the logic of the program should be separated from user interface. This will make the program easier to update to another user interface framework, as discussed earlier.

The program has not at the time of writing been tested on a live engine. It has only been tested on a test platform that simulates a running engine. To note about the testing-rig is that it does not simulate knock (in the version used), which was obviously detrimental to the testing of the program. This was overcome with randomly generated values that the program itself generated during runtime. Live knocking was briefly simulated by bashing a knock sensor against the rig to generate some values that were recorded by the program. The current implementation of the program is actually without any testing framework at all but these are to be incorporated in a later version.

The program follows the PEP8 coding standard as far as it was applicable. Some lines, such as data names, were hard or impossible to implement while keeping the line length under 79 characters. Long lines can be hard to follow and error prone in Python as code is delimited by indentation and line breaks.

The actual thesis has been generated with the latex typesetting system, which uses plain text files as sources with different keywords for sections, subsections and more. These text files are used in generating the final output. Latex simplifies writing as the sources for the

text usually only contain basic formatting delimiters such as sections and subsections. The rest of the source files are just content for the actual thesis. This thesis uses multiple files (one for each chapter) and a main file which contain the rules for generating the document.

Both thesis sources and program sources are version controlled by GIT, which is a distributed version control system. This has lead to easier development and experimentation, as reverting to previous working revisions is just a few commands away. The GIT repositories are stored on-line in a private repository and a copy of the repository has at all times been maintained on three different computers not including the working code on the work computer (which did not have GIT installed). Thus the measures against hardware failure in development phase have been followed.

## 6 Discussion

The program has required a lot of time in designing and creating the user interface and the actual program. The lack of a clear specification lead to some surprises and one complete rewrite of the program. The rewrite was also based on the awful code I had written up to that point. By awful code I mean hacks and barely functional code that did what it should but in a convoluted and non-apparent way. This kind of code was prevalent in the beginning phases of the project as previous knowledge on GUI programming was severely lacking on my part. The specifications were discussed and changed throughout the development of the program. The project started as an idea without any clear vision on how the implementation should work.

Python was known to me but wxPython and the design process of a somewhat larger scale program were mostly unknown to me. A huge chunk of the time spent on this project was on researching wxPython and the Python standard library for suitable functions. The live plotting was one thing that was especially puzzling to find a suitable framework that could update with the required speed of the program. Ironically there was a plotting class in wxPython that is not heavily advertised. It is fast enough but requires many rewrites and a lot of research to be able to implement it correctly.

Software development can arguably be described as an evolutionary process and is changed iteration after iteration based on what works and what doesn't. This has been the way I have made the program. Starting with a basic idea on how the function should work and tack on more functionality in iterations after previous goals or ideas have been implemented and found to work.

The following phase in the evolution of this program would be to include a testing framework for the critical functions. However, as further optimizations are already planned, which might change some of the fundamental functions in some way, the usefulness of unit testing at this stage is debatable. Some optimizations like changing lists to queues could speed up some parts by a factor of two and allow for larger sets of data to be collected without any performance decrease. These optimizations are already tested but not implemented. Other planned changes are refactoring code to improve the general design of the program and minimize function calls that have a huge overhead in terms of clock-cycles.

This thesis has been extremely educational, on both the theoretical side and the practical side of the project. I have learned a lot about the engine control system used at Wärtsilä, the Python programming in general and object oriented programming.

# Bibliography

- [1] H.N. Gupta. *Fundamentals of Internal Combustion Engines*. PHI Learning, 2006.
- [2] Jack Erjavec. *Automotive technology: a systems approach*. Cengage Learning, 2005.
- [3] Seppo Mäkinen. *The Effects of Knocking on the W34SG engine*. Tech. rep. Wärtsilä Finland, R&D, Calculation & Simulation, 2011. eprint: DBAB409026 (Wärtsilä IDM).
- [4] C. S. Draper. *Acoustical Analysis of the Pressure Waves Accompanying Detonation in the Internal-Combustion Engine*. Tech. rep. The Journal of the Acoustical Society of America, 1939.
- [5] Thomas G. Horner. *Engine Knock Detection Using Spectral Analysis Techniques With a TMS320 DSP*. Tech. rep. Texas Instruments, 1995.
- [6] Uwe Kiencke and Lars Nielsen. *Automotive Control Systems, For Engine, Driveline, and Vehicle*. 2nd. Springer Berlin Heidelberg New York, 2005.
- [7] H. Zhao and N. Ladommatos. “Engine performance monitoring using spark plug voltage analysis”. In: *Proceedings of the Institution of Mechanical Engineers* (1997).
- [8] H. Johansson J. Auzin and J. Nytomt. *Ion-gap sense in misfire detection, knock and engine control*. Tech. rep. Society of Automotive Engineers (SAE), 1995.
- [9] Jack Keebler. “Turbocharger with a brain”. In: *Popular Science* (1982).
- [10] Richard G. Lyons. *Understanding Digital Signal Processing*. Prentice Hall, 2010.
- [11] W. H. Nailon Mohamed Anas R. J. Paling and D. R. S. Cumming. *Real-time Combustion Knock Processing Using a Single Instruction Multiple Data Automotive PowerPC System-on-a-Chip*. Tech. rep. Embedded Controller Division, Transportation, Standard Products Group, Motorola SPS Institute for System Level Integration, an academic collaborative venture of the Universities of Edinburgh, Glasgow, Heriot-Watt, and Strathclyde, 1995.
- [12] T. Kaas. *Knock Control, Functional Description*. Software. 2009. eprint: DAAB102405 (Wärtsilä IDM).
- [13] Mark Pilgrim. *Dive Into Python*. 2004. URL: <http://www.diveintopython.net> (visited on 01/10/2011–01/11/2011). Published under the GNU Free Documentation License, original author no longer supports website.

- [14] *wxPython API Reference*. 2011. URL: <http://wxpython.org/docs/api/> (visited on 01/07/2011–01/12/2011).
- [15] Mark Lutz. *Learning Python*. 4th. OReilly Media, Inc., 2009.
- [16] Mark Lutz. *Programming Python*. 4th. OReilly Media, Inc., 2011.
- [17] *The Python Language Reference*. 2011. URL: <http://docs.python.org/reference/index.html> (visited on 01/07/2011–01/12/2011).
- [18] *The Python Standard Library*. 2011. URL: <http://docs.python.org/library/> (visited on 01/07/2011–01/12/2011).
- [19] Timo Auranen. *Knock Control, Application Module Release*. Tech. rep. Version 5.1. Wärtsilä, 2011. eprint: DBAB409026 (Wärtsilä IDM).