Czech Technical University
Faculty of Transportation Sciences

Department of Mechanics and Materials

Study field: Transportation Systems and Technology

Modular Multi-process Control Software for Experimental Devices

MASTER'S THESIS

Author:	Bc. Václav Rada
Supervisors:	Ing. Petr Zlámal, PhD., Ing. Tomáš Fíla
Year:	2019

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta dopravní
děkan
Konviktská 20, 110 00 Praha 1

K618 .... Ústav mechaniky a materiálů

ZADÁNÍ DIPLOMOVÉ PRÁCE
(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení studenta (včetně titulů):

Bc. Václav Rada

Kód studijního programu a studijní obor studenta:

N 3710 – DS – Dopravní systémy a technika

Název tématu (česky):	**Modulární multiprocesová aplikace pro číslicové řízení experimentálních zařízení**
Název tématu (anglicky):	Modular Multi-process Control Software for Experimental Devices

Zásady pro vypracování

Při zpracování diplomové práce se řiďte osnovou uvedenou v následujících bodech:

- V rámci ústavu jsou vyvíjena a provozována experimentální zařízení obsahující společné funkční komponenty (krokové motory, polohovací osy, snímače fyzikálních veličin atd.), avšak mohou se podstatně lišit v jejich počtu, účelu i povaze řízení. V práci dojde k rozvoji softwaru, vyvinutým studentem v rámci bc. práce tak, aby umožnil uživatelsky snadnou a rychlou adaptovatelnost řídícího sw. na funkčně rozdílná experimentální zařízení.
- Cílem práce je vytvořit softwarové řešení založené na jazyku Python a platformě LinuxCNC umožňující pomocí zásuvných modulů a konfiguračních souborů snadnou adaptaci na konkrétní experimentální zařízení bez nutnosti znalosti programovacího jazyka. Řešení bude koncipováno jako multiprocesové, tj. budou navrženy a realizovány jednotlivé funkční celky (jádro, GUI atd.) s definovaným rozhraním, čehož může být s výhodou využito např. při řízení po síti.
- Funkčnost realizovaného řešení bude ověřena na reálném experimentálním zařízení.

ČVUT FD

Rozsah grafických prací:	nebyl stanoven
Rozsah průvodní zprávy:	minimálně 55 stran textu (včetně obrázků, grafů a tabulek, které jsou součástí průvodní zprávy)
Seznam odborné literatury:	G. Zaccone: Python Parallel Programming Cookbook, ISBN 978-1-78528-958-3 LinuxCNC Documentation (http://linuxcnc.org/docs/) M. Summerfield: Rapid GUI Programming with Python and Qt, ISBN 978-0134393333

Vedoucí diplomové práce: **Ing. Petr Zlámal, PhD.**
Ing. Tomáš Fíla

Datum zadání diplomové práce: **28. června 2018**
(datum prvního zadání této práce, které musí být nejpozději 10 měsíců před datem prvního předpokládaného odevzdání této práce vyplývajícího ze standardní doby studia)

Datum odevzdání diplomové práce: **28. května 2019**
a) datum prvního předpokládaného odevzdání práce vyplývající ze standardní doby studia a z doporučeného časového plánu studia
b) v případě odkladu odevzdání práce následující datum odevzdání práce vyplývající z doporučeného časového plánu studia

[illegible]

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE Fakulta dopravní L.S. 3

[illegible]

prof. Ing. Ondřej Jiroušek, Ph.D.
vedoucí
Ústavu mechaniky a materiálů

doc. Ing. Pavel Hrubeš, Ph.D.
děkan fakulty

Potvrzuji převzetí zadání diplomové práce.

Rada

Bc. Václav Rada
jméno a podpis studenta

V Praze dne 28. 6. 2018

Declaration

I hereby submit, for the evaluation and defence, the master's thesis elaborated at the CTU in Prague, Faculty of Transportation Sciences.

I have no relevant reason against using this work in the sense of §60 of Act No. 121/2000 Coll. on the Copyright and Rights Related to Copyright and on the Amendment to Certain Acts (the Copyright Act).

I declare I have accomplished my final thesis by myself and I have named all the sources used in accordance with the Guideline on the ethical preparation of university final theses.

In Prague, May 25, 2019 ....

Bc. Václav Rada

Acknowledgements

The thesis has been supported by the Operational Programme Research, Development and Education in project Engineering applications of microworld physics (CZ.02.1.01/ 0.0/0.0/16_019/0000766) and by the Grant Agency of the Czech Technical University in Prague (grant no. SGS15/225/OHK2/3T/16).

All the support is gratefully acknowledged.

Bc. Václav Rada

Title:	**Modular Multi-process Control Software for Experimental Devices**
Author:	Bc. Václav Rada
Study programme:	Technology in Transportation and Telecommunications
Study field:	Transportation Systems and Technology
Degree:	Master's Thesis
Year:	2019
Supervisors:	Ing. Petr Zlámal, PhD., Ing. Tomáš Fíla Department of Mechanics and Materials, Faculty of Transportation Sciences, Czech Technical University; Institute of Theoretical and Applied Mechanics of the Czech Academy of Sciences

Abstract: The proposed thesis enhances the functionality of the previous-generation control software developed in the Department of Mechanics and Materials and adds specific controls essential for the proper operation with newly developed experimental devices, such as the support for multiple sensors (load cells, temperature sensors, etc.), temperature control, force control, etc. The control software is a multi-process application based on a multi-process core which results in a rapid performance increase over the previous-generation. Modular architecture of the user interface enables the very effective adaptation to various experimental devices. Currently, the control software is fully utilised in controlling the experimental devices in the department, numerous scientific and engineering experiments have been performed and many valuable studies have been published.

Keywords: LinuxCNC, Python Interface, Python, control software, multiprocessing

Název:	**Modulární multiprocesová aplikace pro číslicové řízení experimentálních zařízení**
Autor:	Bc. Václav Rada
Studijní program:	Technika a technologie v dopravě a spojích
Obor:	Dopravní systémy a technika
Druh práce:	Diplomová práce
Rok vydání:	2019
Vedoucí práce:	Ing. Petr Zlámal, PhD., Ing. Tomáš Fíla Ústav mechaniky a materiálů, Fakulta dopravní, České vysoké učení technické v Praze; Ústav teoretické a aplikované mechaniky Akademie věd České republiky

Abstrakt:

Předložená práce zdokonaluje funkcionalitu předchozí generace řídicího software vyvinutého na Ústavu mechaniky a materiálů a rozšiřuje jej o ovládací prvky nutné pro řízení nově vzniklých experimentálních zařízení, jako je např. podpora měření při použití dvou a více siloměrů, použití teplotních čidel, teplotní a silové řízení apod. Nově vyvinutý software je multiprocesová aplikace, která se opírá o robustní multiprocesové jádro, což značně přispívá k vysokému výkonu aplikace. Modulární a dynamicky generované ovládací prvky uživatelského rozhraní umožňují velmi rychlou a efektivní adaptaci pro použití různých experimentálních zařízení. Řídicí software je v současné době plně využíván, byla díky němu provedena řada vědeckých i inženýrských měření a vznikla řada hodnotných publikací.

Klíčová slova: LinuxCNC, Python Interface, Python, řídicí software, multiprocessing

Contents

1 Introduction **14**

2 Theoretical background **16**

2.1 Stepper motor 16

2.1.1 Controlling stepper motors 17

2.1.1.1 Open-loop control system 19

2.1.1.2 Closed-loop control system 19

2.2 Encoders 20

2.2.1 Optical encoders 20

2.2.2 Magnetic encoders 21

2.2.3 Incremental and absolute encoders 21

2.2.3.1 Absolute encoders 21

2.2.3.2 Incremental encoders 21

2.3 LinuxCNC 22

2.3.1 Hardware abstraction layer 22

2.3.2 PID controller 22

2.3.2.1 Proportional term 23

2.3.2.2 Integral term 23

2.3.2.3 Derivative term 24

2.3.3 Hostmot2 driver 24

2.3.4 User interfaces 24

2.3.5 PyVCP 24

2.3.6 GladeVCP 25

2.3.7 Python Interface 25

2.3.8 Python HAL component 25

2.4 Python 25

2.4.1 Global interpreter lock 26

2.4.2 Threading module 26

2.4.3 Multiprocessing module 26

2.4.3.1 Multiprocessing pipes 27

2.4.3.2 Multiprocessing queues 27

2.4.4 PyQt 28

2.4.4.1 Qt Creator 28

2.4.4.2 Qt Designer 28

2.4.4.3 User Interface Compiler 29

2.4.5 PyQwt 29

2.4.6 Matplotlib 29

2.4.7 PyGnuplot 29

3 Initial state **30**

3.1 Straightforward adaptation to various experimental devices 31

3.2 Sensor support 32

3.3 Displacement-driven experiments 32

3.4 Obtaining and logging data 32

3.5 Real-time plotting and static plotting 33

3.6 Overview 33

4 Developed software **35**

4.1 Introduction 35

4.2 Performance gain 35

4.2.1 Control software core 36

4.2.1.1 Stat Poller 36

4.2.1.2 Data Logger 38

4.2.1.3 Data Keeper 38

4.2.1.4 Command Executor 40

4.2.1.5 I/O Manager 40

4.2.2 User Interface 40

4.2.2.1 Plotting performance benchmark 40

4.3 Custom Python script execution 42
4.3.1 Unix domain socket 44
4.3.2 Serialising modules 44
4.3.2.1 Pickle and cPickle modules 44
4.3.2.2 JSON module 45
4.3.2.3 List serialisation benchmark 46
4.3.2.4 Dictionary serialisation benchmark 49
4.3.2.5 Recapitulation 51
4.4 Enhanced Functionality 51
4.4.1 Introduction 51
4.4.2 Sensor support 51
4.4.3 Axes position bar 55
4.4.3.1 Encoder support 55
4.4.3.2 G92 offset 56
4.4.3.3 Positioning error 57
4.4.3.4 Homed status 57
4.4.4 Plot plugin 57
4.4.5 Force control 59
4.4.5.1 Common controls 61
4.4.5.2 Constant force command tab 62
4.4.5.3 Linear force command tab 63
4.4.5.4 Sine wave force command tab 64
4.4.5.5 Square wave force command tab 65
4.4.5.6 Triangle wave force command tab 66
4.4.5.7 Sawtooth wave force command tab 67
4.4.6 Temperature control 68
4.4.7 Rapo library 70

5 Case Studies **71**

5.1 Compression of a spongious sample in simulated physiological conditions . 71
5.2 Fracture analysis of sandstone 74

6 Work in Progress **77**

6.1 Remote control 77
6.1.1 TCP socket 78
6.1.2 Server Manager 78
6.1.3 Client Manager 78
6.1.4 Remote user interface 81
6.1.5 Remote script execution 81

7 Conclusion **84**

Appendices **89**

A Prime fatorisation script **89**

B Communication schemes description **94**

C Plot comparison script **99**

D Serialising modules comparison script **102**

List of Figures

2.1 Stepper motor feed with and without microstepping 17
2.2 STEP and DIR control signals scheme 18
2.3 STEP/DIR generated by a PC (CPU) 18
2.4 STEP/DIR generated by a motion controller 18
2.5 Closed-loop control system scheme 19
2.6 Optical encoder scheme, taken and edited from [1] 20
2.7 PID controller feedback loop 23
2.8 Multithreaded execution within a single process 27

3.1 Axes position bars 31
3.2 Plot plugin of the user interface 33

4.1 Results of the prime factorisation benchmark 37
4.2 The developed control software scheme 38
4.3 The control software core inter-process communication scheme 39
4.4 The control software user interface scheme 41
4.5 Matplotlib and PythonQwt plot comparison results 42
4.6 The API provider and rapo library connection scheme 43
4.7 Python list serialisation performance comparison 47
4.8 Python list deserialisation performance comparison 47

4.9 serialised Python list size comparison 48
4.10 Python dictionary serialisation performance comparison 49
4.11 Python dictionary deserialisation performance comparison 50
4.12 serialised Python dictionary size comparison 50
4.13 Experimental device for the 4-point bending 52
4.14 Load cell sensor initialising file content 52
4.16 The plugin for handling the sensors 53
4.15 Thermometer sensor initialising file content 53
4.17 The axes position plugin of the user interface 55
4.18 The axes position plugin of the user interface with G92 offset active 56
4.19 The plot plugin of the user interface 57
4.20 Force control flowchart 60
4.21 Force control plugin overall view 61
4.22 Force control plugin common controls 62
4.23 Constant force command tab 63
4.24 Linear force command tab 64
4.25 Sine wave force command tab 65
4.26 Square wave force command tab 66
4.27 Triangle wave force command tab 67
4.28 Sawtooth wave force command tab 68
4.29 Temperature control flowchart 69
4.30 Temperature control plugin of the user interface 70

5.1 Human bone specimen and loading device 72
5.2 Loading device exploded view in detail 72

5.3 Loading device exploded view in detail 73

5.4 Overall view of the control software 74

5.5 Experimental device for the 4-point bending mechanical test 74

5.6 Principle of the 4-point bending and on-the-fly CT 75

5.7 Results of the sandstone analysis 76

6.1 Scheme of the control software remote control 77

6.2 Control software core with connection to the Server Manager included . . . 79

6.3 Server Manager and Client Manager inter-process communication scheme . 80

6.4 Remote user interface architecture 82

6.5 Remote API provider and rapo library connection scheme 83

Acronyms

CNC	Computer Numerical Control.
CPU	Central Processing Unit.
CT	Computed Tomography.
DVC	Digital Volume Correlation.
FIFO	First in, first out.
FPS	Frames per second.
GIL	Global Interpreter Lock.
GUI	Graphical User Interface.
HAL	Hardware Abstraction Layer.
IDE	Integrated Development Environment.
IPC	Inter-process Communication.
JSON	Javascript Object Notation.
QML	Qt Modeling language.
UIC	User Interface Compiler.
WYSIWYG	what you see is what you get.

Chapter 1

Introduction

In recent years, Computer Numerical Control (CNC) systems have made an enormous strides. CNC machines have found utilisation in a wide range of applications (industry, medicine, etc.). Industrial CNC machines are typically aimed at the very effective and precise manufacturing of parts with complex shapes and have many other advantageous applications in various industrial fields. There are many CNC software solutions, which vary in price, performance or closed-source commercial (Siemens, FANUC, LabVIEW) and open-source solutions (LinuxCNC, Arduino - primarily for hobby operation).

Our research group, in the Department of Mechanics and Materials in the Faculty of Transportation Sciences at the Czech Technical University and at the Institute of Theoretical and Applied Mechanics of the Czech Academy of Sciences, tends to use open-source solutions, because the needs are different from conventional industrial CNC applications. Custom experimental devices used for advanced mechanical testing of materials are developed in the department. Therefore, some properties of the CNC software have to be operationally modified according to our requirements. For this purpose, closed-source commercial CNC software is not suitable for use with the devices. The purpose of the designs can be divided into three groups:

- mechanical loading machines (e. g., in-situ loading devices for X-ray computed tomography)
- positioning machines (e. g., optics and sample positioning)
- sample preparation devices (e. g., automatic grinders)

The devices are designed to be compact and portable. The devices are equipped with axes for the precise positioning in cooperation with high resolution encoders. Most of the devices feature various sensors, such as load cells and thermometers etc., used for measuring physical quantities, such as force or temperature.

For controlling the experimental devices, an open-source system LinuxCNC is used in the department. The open-source system is free to use and its functionality may be customised and extended to fit our requirements with use of the Python programming language through the LinuxCNC Python Interface. The Python Interface enables one to control the experimental devices directly using Python.

The performance of the control software is one of the key aspects in terms of reliablility and precise measurement and data acquisition. Modern CPUs contain numerous processing cores which provide great computing power when utilised properly in parallel. Another requirement on the control software is a modular design which provides a very straightforward adaptation for use with various experimental devices operated in the department including devices developed in the future.

The control software proposed in the thesis is a multi-process application with modular features and controls developed in Python programming language. Currently, the control software is fully used for performing various types of mechanical tests with great success and, thanks to this, many studies have been published.

Chapter 2

Theoretical background

Experimantal devices operated at our department use various types of actuators. The largest portion of them is equipped with stepper motors, due to their simplicity and low cost. The devices are equipped with optical or magnetic encoders which provide position feedback. For controlling the devices an open-source system LinuxCNC is used.

2.1 Stepper motor

A stepper motor is a brushless electric motor, which rotates in a number of equal steps. A stepper motor primarily consists of two parts: a stator and a rotor. The rotor is a permanent magnet[1] of a gear shape with a given number of teeth. The stator consist of coils, which can be magnetised in a certain order by an electric current and make the rotor turn to the closest stable position.

Magnetising the coils in a certain order is achieved by a stepper motor driver. The stepper motor driver takes low-level voltage impulses (commonly 5 V) on the input and produces a high-current signal which is delivered to the coils in the motor. The impulses on the driver input are usually in the form of STEP and DIR signals. A STEP signal is a square shaped signal and each STEP signal impulse[2] makes the rotor revolve to a fixed angle. This angle is called a step. Stepper motor drivers usually support a microstepping functionality

[1]Besides permanent magnet (PM) stepper motors, other types such as variable reluctance (VR) and hybrid synchronous stepper motors exist

[2]Or rather each rising or sinking edge of the STEP control signal

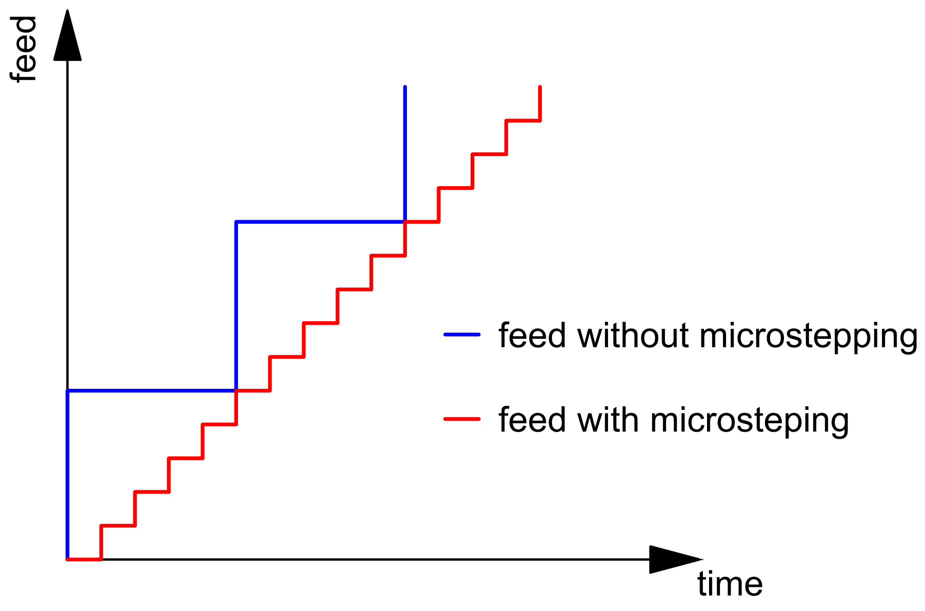


Figure 2.1: Stepper motor feed with and without microstepping

which provides a smoother motion of the stepper motor. In theory, the microstepping may increase the motion precision in case a load driven by the motor is well within its maximum capacity, see Figure 2.1

The DIR signal determines the direction of the stepper motor rotation. For instance, if the DIR signal is equal to 0 V, the motor rotates clockwise and if the DIR signal is equal to 5 V, the motor rotates the other direction - in this case counter-clockwise, see Figure 2.2.

2.1.1 Controlling stepper motors

Controlling stepper motors involve the precise generation of the STEP and DIR signals for the stepper motor driver.

The simplest approach is to generate the signals by the Central Processing Unit (CPU) of a PC and deliver it to the driver using an output port (e.g., parallel port), see Figure 2.3. However, this approach has a massive drawback. In this case, the signal generation is CPU-bound, therefore, it is very dependent on the CPU workload and can exhibit significant latency. If the CPU is stressed high, the STEP and DIR signals might not be generated by the CPU precisely in time, hence the system is affected by excessive latency.

Another approach is to generate STEP and DIR signals on a dedicated motion controller or another form of the real-time hardware layer. The motion controller enables decreasing

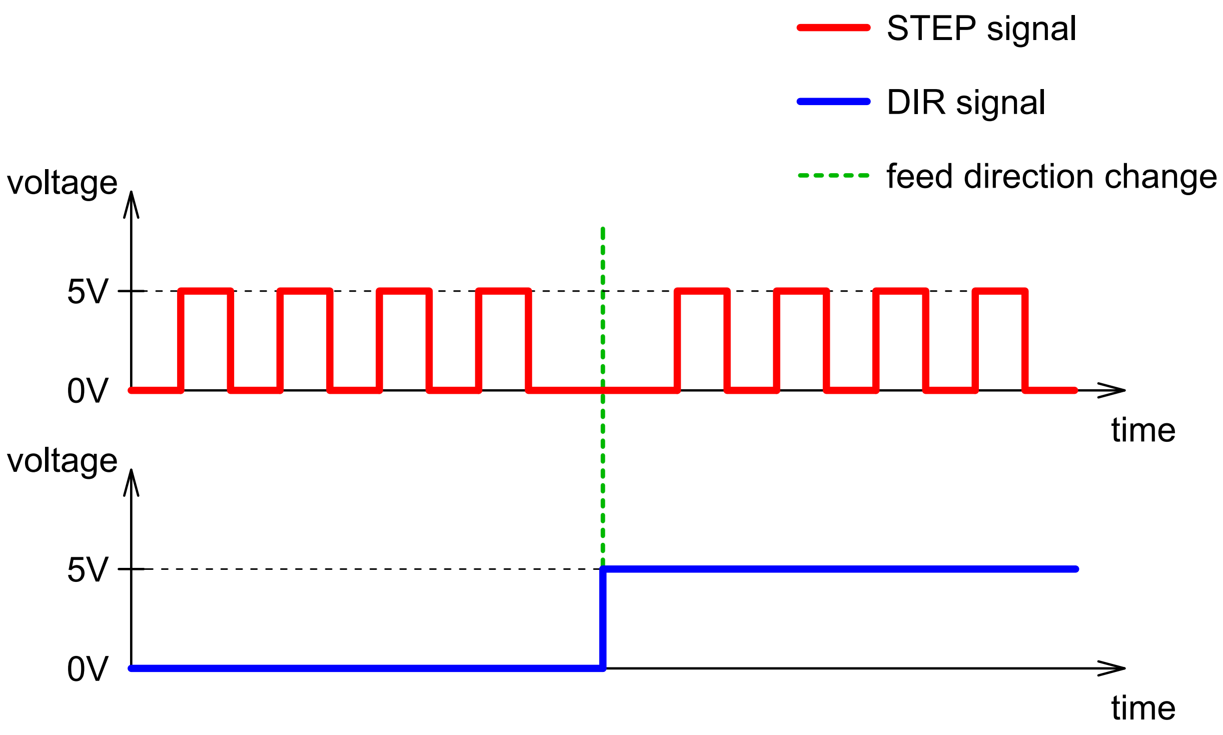


Figure 2.2: STEP and DIR control signals scheme

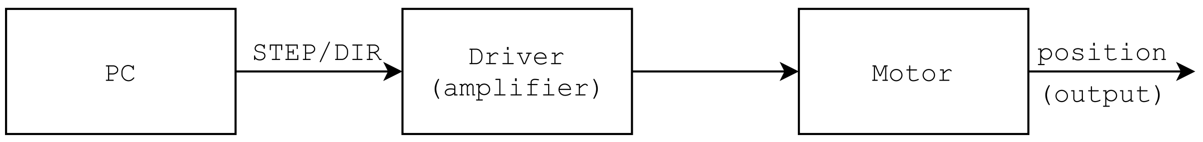


Figure 2.3: Stepper motor controlling scheme, the STEP/DIR generator is a PC (CPU)

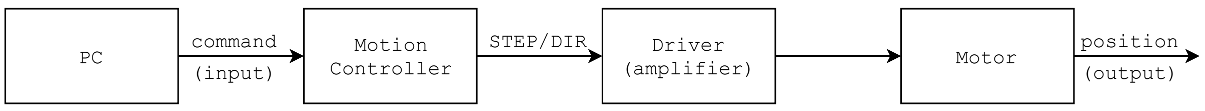


Figure 2.4: Stepper motor controlling scheme, the STEP/DIR generator is a motion controller

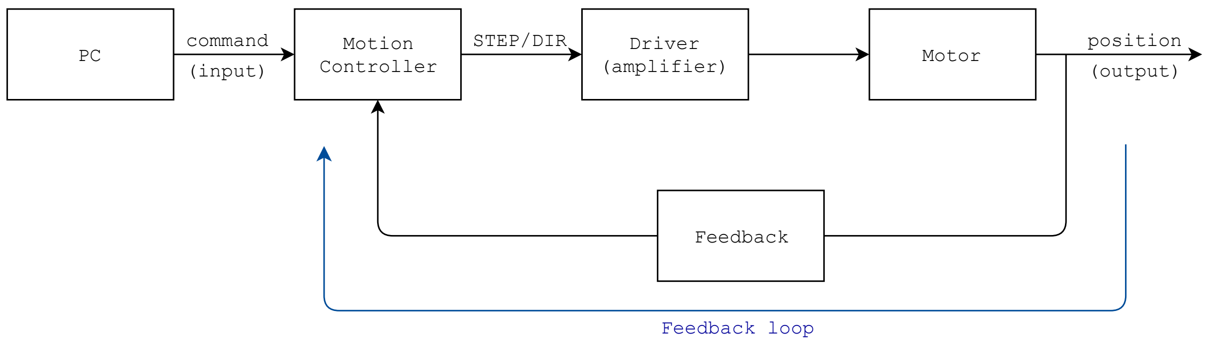


Figure 2.5: Closed-loop control system scheme

the system latency as the STEP and DIR signals are generated by the motion controller independently on a CPU workload. The scheme of this approach is shown in Figure 2.4

2.1.1.1 Open-loop control system

Stepper motors have an inherent ability to control the position, as the position can be determined by the number of steps to rotate. This makes them very easy to use without any feedback encoder, but the lack of an encoder limits its performance. In an open-loop control system, the stepper motor can drive a load which is well within its capacity, otherwise using a stepper motor beyond the limits may lead into positioning errors due to missed steps. An open-loop control system scheme is shown in Figures 2.3 and 2.4

2.1.1.2 Closed-loop control system

On the other hand, a closed-loop control system is based on an open-loop control system concept, but has one or more feedback loops. Closed loop systems are designed to automatically produce and maintain the intended position (command) by comparing it with the actual position (feedback). The difference between the command and the feedback determines the error which the control system must compensate for. A closed-loop control system scheme is shown in Figure 2.5

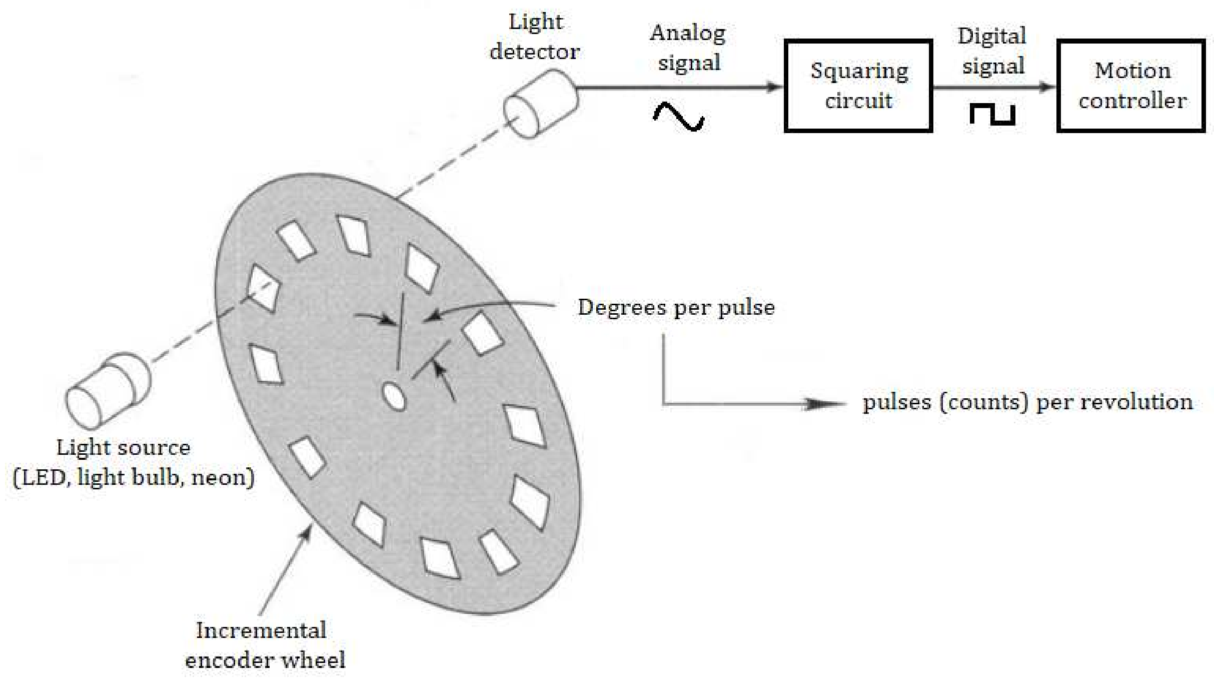


Figure 2.6: Optical encoder scheme, taken and edited from [1]

2.2 Encoders

An encoder is a sensor or transducer that encodes a position to an analog signal, which can then be decoded by a motion controller back into position. Encoders may work on various physical principles.

2.2.1 Optical encoders

Optical encoders are one of the most commonly used encoders in automation applications. Optical encoders are based on light detection as the light passes through an encoder wheel. A source of light (mostly LED) shines through an encoder wheel which has a series of slots in it. As the wheel rotates, the detector detects light passing though the slots. Each detection of the light exhibits the rotation of the encoder wheel by a defined angle. Optical encoders can achieve very high precision and are suitable for high feed rates. However, optical encoders are sensitive to contaminants such as dust, liquid and grease, also to shocks and vibrations, which makes them inconvenient for use in industrial environments.

2.2.2 Magnetic encoders

Magnetic encoders employ a magnetised scale and a read head. The read head can use either a Hall effect or a magnetoresistive sensor to detect signals generated by the magnetic code of the scale to provide position information. Unlike optical encoders, magnetic encoders are more resistant to environmental impacts, which makes them more suitable for use in dirty environments. However, the precision of magnetic encoders is lower compared to optical encoders.

2.2.3 Incremental and absolute encoders

Positioning tasks require precise position values to monitor or control the motion. In many applications, position sensing is undertaken using rotary encoders, also called shaft encoders or simply encoders. These sensors transform the mechanical angular position of a shaft or axle into an electronic signal that can be processed by a control system.

2.2.3.1 Absolute encoders

Absolute rotary encoders are capable of providing unique position feedback from the moment they are switched on. This is accomplished by scanning the position of a coded element. All positions in these systems correspond to a unique code. Even motion that occurs while the system is without power is translated into accurate position feedback once the encoder is powered up again.

2.2.3.2 Incremental encoders

Incremental rotary encoders generate an output signal each time the shaft rotates a defined angle. (The number of signals per turn defines the resolution of the device.) Each time the incremental encoder is powered on it begins counting from zero, regardless of where the shaft is. The initial homing procedure to a reference point is, therefore, necessary in all positioning tasks, both upon start up of the control system and whenever power to the encoder has been interrupted.

Experimental devices in the Department of Mechanics and Materials are controlled by an open-source system LinuxCNC.

2.3 LinuxCNC

LinuxCNC [2] is an open-source software system for the numerical control of CNC machines such as lathes, milling machines, cutting machines, robots, etc. Due to the precise control of the CNC machines, LinuxCNC requires real-time computing capabilities which are provided by real-time extensions of the operating system.

LinuxCNC uses a Hardware Abstraction Layer (HAL) to configure the control system hardware.

2.3.1 Hardware abstraction layer

The hardware abstraction layer [3] is a software subsystem which provides hardware abstraction. It allows applications to use the hardware of the system through a simple and abstract interface. For instance, the Hostmot2 driver is a package for the Hardware Abstraction Layer which provides abstraction of Mesa Electronics Anything I/O FPGA cards, which are used in our department. It features many other abstraction components such as a PID controller, etc. and also includes various tools, such as a virtual oscilloscope to examine real-time signals.

2.3.2 PID controller

A PID (proportional–integral–derivative) controller [4] is a control loop based on a feedback mechanism widely used in industrial control systems and various other applications requiring continuous control. A PID controller continuously calculates an error value $e(t)$ as the difference between the measured process variable $y(t)$ and the desired setpoint $r(t)$, see Formula 2.1. It applies a correction based on proportional (P), integral (I), and derivative (D) terms which constitute the manipulated variable $u(t)$, see Formula 2.2 and

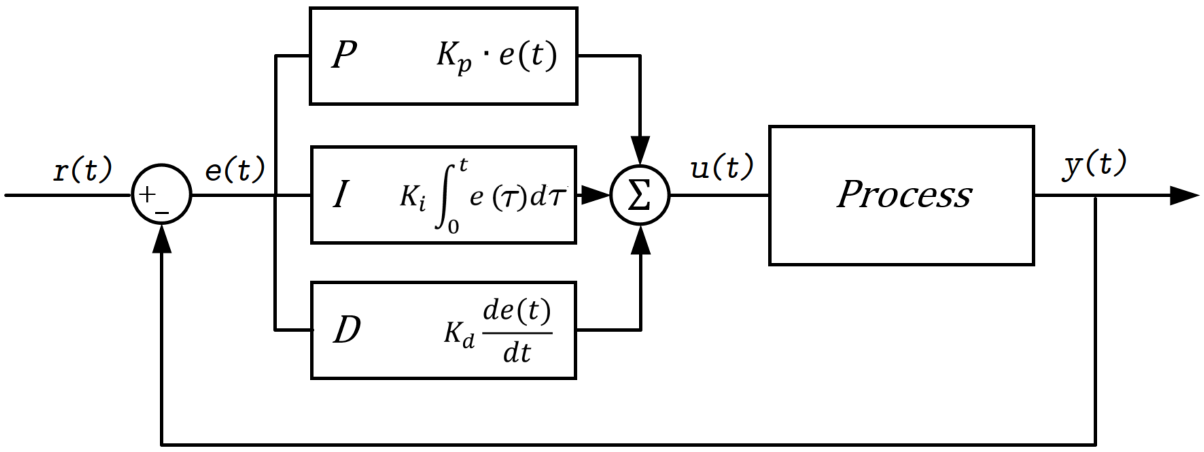


Figure 2.7: PID controller feedback loop

Figure 2.7.

$$e(t) = r(t) - y(t) \tag{2.1}$$

$$u(t) = K_p \cdot e(t) + K_i \cdot \int_0^t e(\tau)d\tau + K_d \cdot \frac{de(t)}{dt} \tag{2.2}$$

2.3.2.1 Proportional term

The proportional term $P = K_p \cdot e(t)$ is proportional to the current value of $e(t)$. The coefficient K_p is a proportional gain. The proportional term is the fundemantal term of the PID controller. The value of the proportional gain K_p is critical to the response rate and system stability. The proportional term alone cannot achieve a stable deviation between the command and feedback, so other terms are used.

2.3.2.2 Integral term

The integral term $I = K_i \cdot \int_0^t e(\tau)d\tau$ records the past values of $e(t)$ and intergrates them gradually to calculate the I term. Using this term allows one to reach a stable deviation between the command and feedback, often in exchange for a longer settling time.

2.3.2.3 Derivative term

The derivative term $D = K_d \cdot \frac{de(t)}{dt}$ estimates the future trend of $e(t)$. The derivative term represents the prediction element of the controller and allows the settling time to be shortened and the system response smoothened.

2.3.3 Hostmot2 driver

Hostomot2 [5] is an open-source driver developed by Mesa Electronics for FPGA Anything I/O motion control cards. It provides modules such as STEP/DIR generators, PWM generators, encoders (quadrature counters), etc. which can be loaded into HAL to connect these module instances to the I/O headers.

2.3.4 User interfaces

LinuxCNC comes natively preinstalled with various Graphical User Interfaces (GUIs), such as *Axis* - default user interface [6], *Touchy* - user interface used with touchscreens [7], etc. Preinstalled user interfaces are primarily designed for industrial CNC applications, therefore, they are not suitable for use with the custom experimental devices developed in our department because extended fuctionality is needed. User interfaces with additional functionality can be developed using LinuxCNC components such as PyVCP or Glade-VCP or it can be developed in the Python programming language based on the LinuxCNC Python Interface.

2.3.5 PyVCP

PyVCP (Python Virtual Control Panel) [8] is a package which provides additional functionality to native LinuxCNC GUIs. It is based on the same GUI toolkit (Tkinter) as the *Axis* user interface. PyVCP enables adding a custom panel on the right side of the *Axis* user interface. However, PyVCP is limited to setting and displaying HAL internals only.

2.3.6 GladeVCP

GladeVCP (Glade Virtual Control Panel) [9] is a LinuxCNC component which also extends functionality to native LinuxCNC GUIs. It uses Glade which is a WYSIWYG graphical user interface designer. GladeVCP is based on the GTK user interface toolkit. Unlike PyVCP, GladeVCP is not limited to interacting with HAL only, as an arbitary Python code can be executed.

2.3.7 Python Interface

LinuxCNC Python Interface [10] enables one to control devices directly using Python programming language by providing the linuxcnc module for Python. The module is compatible with Python version 2.x and module usage is very straightforward. It provides observing status variables of HAL such as axes position, axes velocity, analog/digital inputs/outputs and sending commands to it through Python. It uses three operating channels: a status channel, a command channel and an error channel.

2.3.8 Python HAL component

Custom variables of HAL such as encoder position can be observed using the custom HAL component [11] compatible with Python. For this purpose, the hal module for Python can be used. It provides connection which can be linked with HAL and share variables using the connection.

2.4 Python

Python [12] is an interpreted programming language supporting multiple programming paradigms such as object-oriented, functional, procedural and imperative. Python is a high-level programming language, it provides dynamic typing and automatic memory management by using garbage collector.

In 2019, Python features two incompatible versions, Python 2.x and Python 3.x. Python 3.x was first introduced in 2008 and is planned to replace Python 2.x in 2020 when Python

2.x will no longer be maintained by the developers. However, some Python modules are not forward compatible yet, such as the linuxcnc module, see subchapter 2.3.7 which might be an obstacle in upgrading from Python 2.x to Python 3.x.

Python has multiple implementations such as CPython, Jython, IronPython, etc. CPython is the default and most commonly used implementation of Python. CPython implementation is written in the C programming language and Python.

2.4.1 Global interpreter lock

Python default implementation CPython has a significant performance limitation due to the use of the Global Interpreter Lock (GIL) [13] which is a thread-safe mechanism to prevent parallel excution by threads within an interpreter process. It means that the Python threads cannot bring a performance gain by parallel execution, because the thread needs to acquire the lock in order to execute any instruction, so a multi-threaded execution cannot be faster than a single-threaded[3], see Figure 2.8. When the thread executes a certain number of Python virtual instructions or a specific time period elapses, the GIL is released and acquired by another thread.

2.4.2 Threading module

The threading module provides a high-level threading interface. Python threads are sometimes called light-weight processes as they do not require much memory overhead. Multiple threads within the same process share the same data space. It enables Python threads to share variables so they comunicate with each other much more easily than if they were separate processes. However, Python threads are limited by the GIL, see subchapter 2.4.1.

2.4.3 Multiprocessing module

The multiprocessing library provides the ability to spawn separate Python processes using an interface similar to the threading module. The module enables the true parallel execu-

[3]Certain computational performance-oriented libraries such as NumPy, SciPy might overcome this limitation in particular cases

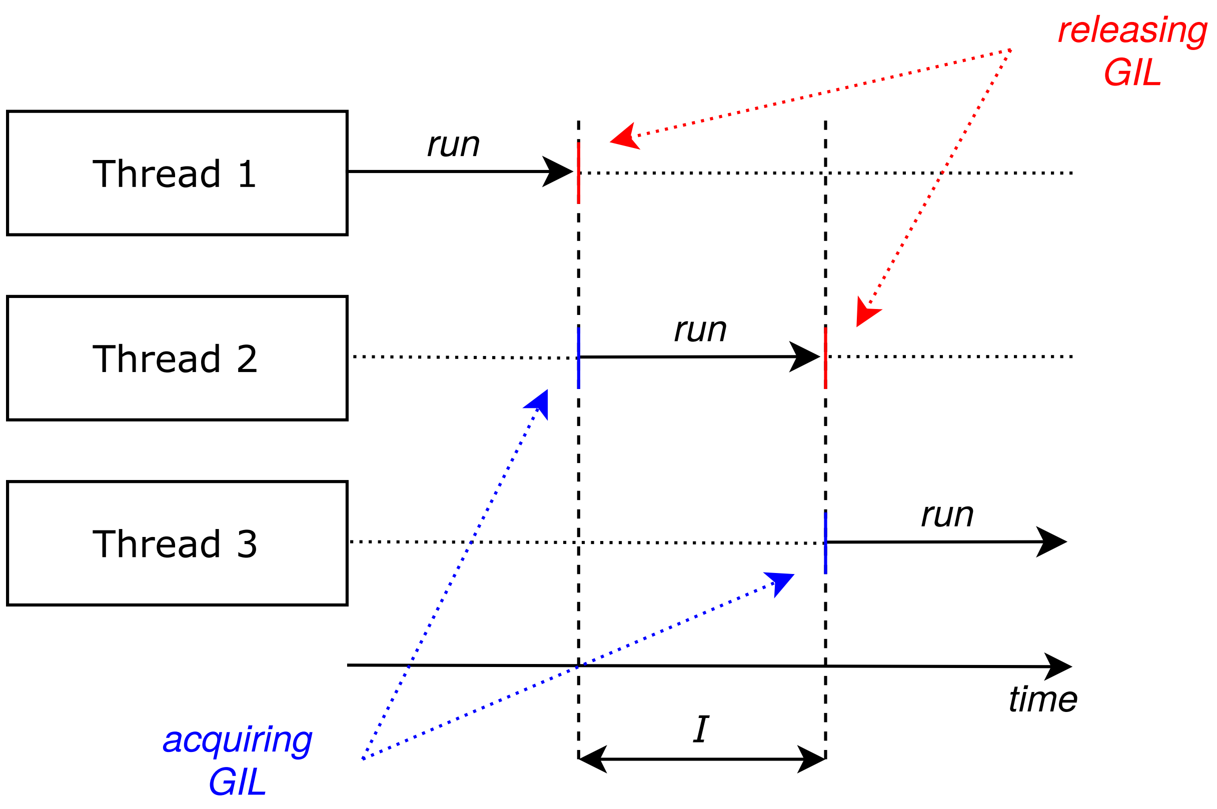


Figure 2.8: Multithreaded execution within a single process

tion of a Python code and may utilise multiple CPU cores. However, Python processes have distributed memory (does not share data space) which makes the interaction and communication between Python processes (Inter-process Communication (IPC)) challenging. The multiprocessing library provides various types of Inter-process communication mechanisms such as Queues, Pipes and synchronisation primitives such as locks.

2.4.3.1 Multiprocessing pipes

A multiprocessing pipe is one of the simplest types of IPC. It only connects two processes with each other. The pipe is bidirectional by default. It may also be unidirectional, thus it only allows sending messages by one process (producer) and only allows receiving the messages by the other process (consumer).

2.4.3.2 Multiprocessing queues

A multiprocessing queue is a multi-producer, multi-consumer First in, first out (FIFO) queue, i.e., unlike multiprocessing pipes, it enables the connection between multiple processes. It is implemented using multiprocessing pipes and locks/semaphores and a feeder thread. When data is put to the queue by a process, the data first comes to queue buffer and then the feeder thread distributes the data to the multiprocessing pipe leading to the appropriate process.

2.4.4 PyQt

PyQt [14] is a binding[4] of a Qt framework for Python. The Qt framework is a robust toolkit used for GUI development as well as multi-platform applications. Qt framework includes various development tools, such as the Qt Creator, the Qt Designer and the User Interface Compiler.

2.4.4.1 Qt Creator

Qt Creator is a C++ Integrated Development Environment (IDE) which is part of the Qt framework. Qt Creator provides features such as syntax highlighting, autocompletion, a visual debugger and integrates the Qt Designer for designing and building GUIs from Qt widgets[5].

2.4.4.2 Qt Designer

Qt Designer is a tool included in the Qt framework [15] which is used for designing and building GUIs in a WYSIWYG fashion. The GUI design can be saved in a platform-independent, XML-formatted (or rather QML[6]) file with *.ui extension. The file contains the whole user interface definition, which can be compiled into a source code using the User Interface Compiler.

[4]Binding is a wrapper library that bridges two programming languages. For instance it enables one to use a library developed in C/C++ programming language with Python.

[5]Widget is the foundation of all objects of the GUI.

[6]Qt Modeling language (QML) is a markup language used by the Qt framework for GUI declaration

2.4.4.3 User Interface Compiler

The User Interface Compiler (UIC) is a tool for compiling GUIs designed by the Qt Designer into a source code. The UIC natively compiles the *.ui file into a header file for use with the C++ programming language. In order to compile the *.ui file into the Python source code, PyQt provides a tool PyUIC which operates the UIC likewise.

2.4.5 PyQwt

PyQwt [16] is a Python binding for the Qwt (Qt Widgets for Technical Applications) [17] library. Qwt extends the Qt framework with widgets aimed at engineering and scientific applications, such as a widget to plot 2-dimensoinal data. It also features dials, compasses, thermometers, sliders, wheels or knobs to control or display values, etc.

2.4.6 Matplotlib

Matplotlib [18] is a plotting library for Python which produces publication quality figures in various formats, such as *.svg, *.eps, *.pdf, etc. It includes backends for use with various widget toolkits, such as Qt and GTK.

2.4.7 PyGnuplot

PyGnuplot [19] is a Python wrapper for Gnuplot [20]. Gnuplot is a multi-platform plotting library. It enables the generation of two-dimensional and three-dimensional figures and displays them directly on screen or saves them in various high quality image formats such as *.svg, *.eps, etc.

Chapter 3

Initial state

In 2017, a first-generation control software for experimental devices operated in the Department of Mechanics and Materials in the Faculty of Transportation Sciences at the Czech Technical University and at the Institute of Theoretical and Applied Mechanics of the Czech Academy of Sciences was developed as part of my Bachelor's thesis [21]. The control software was based on the Python Interface of the open-source system LinuxCNC [2]. The control software was developed using the Python programming language version 2.7. For user interface development, the Qt framework version 4.8, in cooperation with Python binding PyQt was used.

The main features of the first-generation control software are:

- straightforward adaptation to various experimental devices
- sensor support
- obtaining and logging data
- real-time plotting and static plotting
- displacement-driven experiment procedures

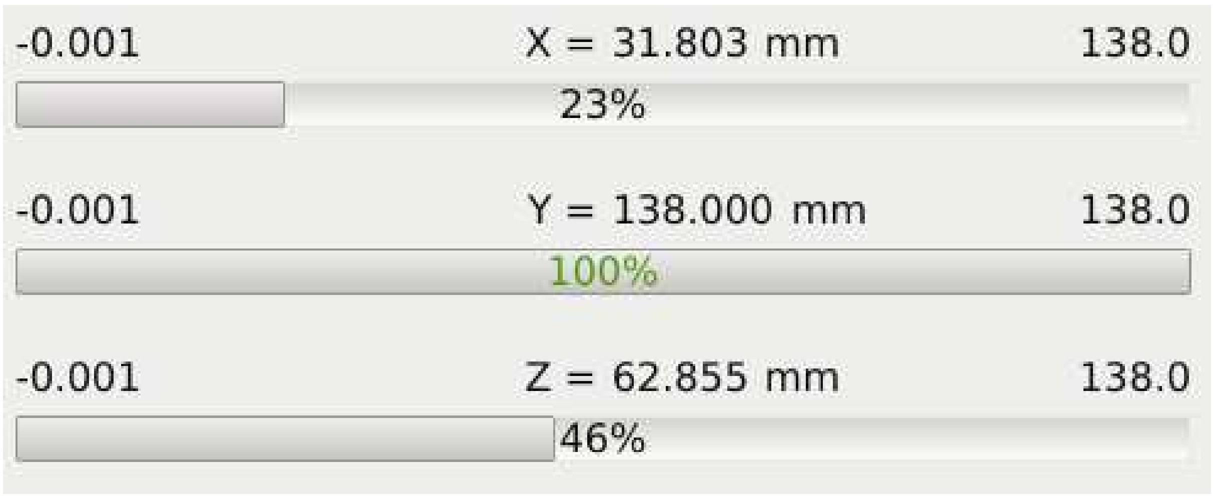


Figure 3.1: Axes position bars

3.1 Straightforward adaptation to various experimental devices

The devices operated in our department are equipped with common parts such as an actuator, an encoder, limit switches and with application specific equipment such as a load cell, etc. Each experimental device comes with its own LinuxCNC initialising file. The initialising file satisfies the device specifics, therefore, it is essential for the proper control software operation. The file includes various parameters specifying, for instance, the number of axes, which type of hardware is used for the data acquisition, etc.

The control software was designed as a set of separate plugins. There are plugins for common features of all devices such as an emergency-stop (E-STOP) button, a power button, a home position button, axes position display bars, force display bar, etc. On the other hand, there are plugins for specific applications such as a plugin for a displacement-driven experiment, etc. The user interface and the plugins inside of it are generated dynamically based on the machine initialising file. For instance, the parameters `AXES_ACTIVE` and `AXES_UNITS` correspond with the axes position diplay bars.

3.2 Sensor support

The control software has been used to control the laboratory devices in order to observe the mechanical properties of materials by obtaining data samples of physical quantities such as the force, position, etc. In our department, load cells based on strain gauges are used for force measurement.

For proper sensor use, each sensor is characterised by a set of constants (sensitivity, range, overload factor, etc.) and the control software must take them into account. In case of load cells, these contants refer to the tensometric bridge properties inside the load cell. The control software features an interface for one force sensor only which became a significant limitation (e.g., one of the newly developed experimental devices in our department, the four-point bending device [22] operates with two loading units and each of them is equipped with a force sensor).

3.3 Displacement-driven experiments

The control software enables one to perform displacement-driven experiments only. The displacement-driven experiment is a fundamental type of mechanical testing. The deformation of an experimental sample during the experiment is controlled by a crosshead movement. The displacement-driven experiment is very simple to perform, however it cannot be adjusted based on the sample response during the experiment.

3.4 Obtaining and logging data

The control software includes a discrete thread to obtain data, such as the force, the axes position, etc. Each data sample includes a unique timestamp to provide a time reference. The data samples are obtained in a loop with typicaly 0.02 seconds period within the loop which results in a sampling rate of 50Hz. The data samples are periodically saved to a text-based output file.

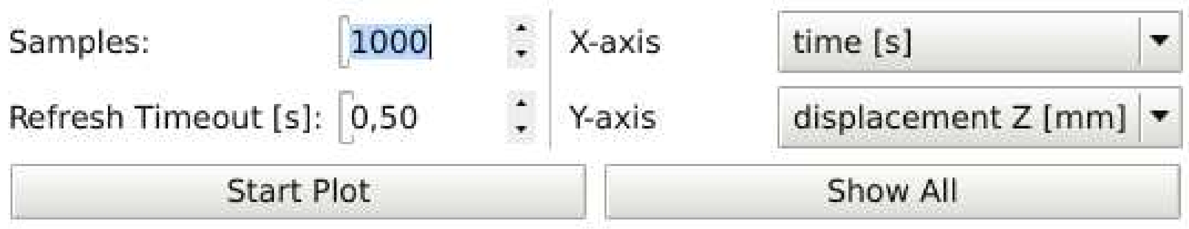

Samples: 1000
Refresh Timeout [s]: 0,50
X-axis time [s]
Y-axis displacement Z [mm]

Figure 3.2: Plot plugin of the user interface

3.5 Real-time plotting and static plotting

In order to visualise the data, the control software features a plotting functionality. It enables the real-time plotting (replotting the data periodically) and static plotting, see Figure 3.2. The user can configure the plotting parameters, such as the real-time plot refresh timeout, data series, etc.

It is based on the matplotlib library which produces high quality figures, however matplotlib has significantly limited performance which has become a drawback especially for real-time plotting.

3.6 Overview

The first-generation control software has been used in the Department of Mechanics and Materials in the Faculty of Transportation Sciences at the Czech Technical University and at the Institute of Theoretical and Applied Mechanics of the Czech Academy of Sciences for two years in full operation.

The most significant limitation of the control software is the sampling rate performance and the real-time plot performance due to the Python threads. It features two discrete threads within a single Python process. The first thread obtains and saves the data, the other thread is used for the real-time plotting. Referring to the GIL, the Python threads cannot bring any performance gain by the parallel execution in this scenario, see subchapter 2.4.1. Furthermore, the plotting library matplotlib is not very suitable for performance-oriented real-time plotting.

The control software does not feature displaying encoder feedback within the axes position bars, see Figure 3.1. In order to display the encoder feedback, an external GladeVCP panel is necessary to be used. The control software features support for a single force sensor and the plotting of a single data series which has become a limitation with a wider portfolio of experimental devices operated in the department.

Some of the newly developed experimental devices require support for multiple force sensors or support of various types of sensors apart from force sensors, such as thermometers. Moreover, these devices require more sophisticated and modular experimental procedures such as force-driven experiments, etc. for proper utilisation.

Chapter 4

Developed software

The new-generation control software developed as part of this Master's thesis is the successor to the first-generation control software described in the previous chapter.

4.1 Introduction

The main objective of the control software development was to improve performance, primarily the sampling rate and the real-time plot refresh rate capabilities. This required overcoming the Python threads limitations occuring in the first-generation control software by using separate Python processes instead of the threads. Furthermore, with a wider portfolio of experimental devices operated in the department, new demands for functionality came up.

4.2 Performance gain

Python processes enable the parallel execution of the Python code which may result in a rapid performance inrease. To demostrate the performance difference between the Python threads and the Python processes, a simple script for a benchmark was created, see Appendix A. The benchmark is based on a prime factorisation of a range of numbers utilising multiple threads or multiple processes and comparing the execution time. The prime factorisation benchmark was performed on a range of 1 million integers and run on the Intel Xeon W-2145 @4.5 Ghz (8-core, 16-thread) CPU. The result of the benchmark is shown in Figure 4.1.

The results confirm, that the Python threads do not bring any performance increase. In fact, the Python threads have a negative impact on the performance due to the switching between the threads (releasing and acquiring the GIL, see subchapter 2.4.1). The results also prove that the performance may be increased by the parallel execution of the code using separate Python processes.

The developed control software is based on a multi-process core consisting of processes with various funtionality and a single-process user interface. Besides the multi-process core and the user interface, the control software includes two more processes. The Server Manager process which is described in detail in subchapter 6.1 and the API Provider process which is described in subchapter 4.3. The control software architecture is shown in Figure 4.2.

4.2.1 Control software core

The control software core consists of five processes: a Stat Poller, a Command Executor, a Data Logger, a Data Keeper and an I/O Manager. Each process has a dedicated queue for receiving and processing messages from other processes. Therefore, the multi-process core is able to utilise multiple CPU cores, which led into a sampling rate increase from 50Hz up to 500Hz compared with the first-generation control software. A detailed scheme of the control software core is shown in Figure 4.3. A further description of the scheme can be found in Appendix B.

4.2.1.1 Stat Poller

The Stat Poller process is one of the crucial processes of the control software. It is based on the status channel and the error channel of the LinuxCNC Python Interface. The status channel provides access to all status variables of the device, the error channel checks if any error occurred. The Stat Poller extends the status channel functionality by including an interface for reading sensors the output signal, etc. It supports sensors operating on an electrical signal which is proportional to the applied excitation voltage (millivolts per

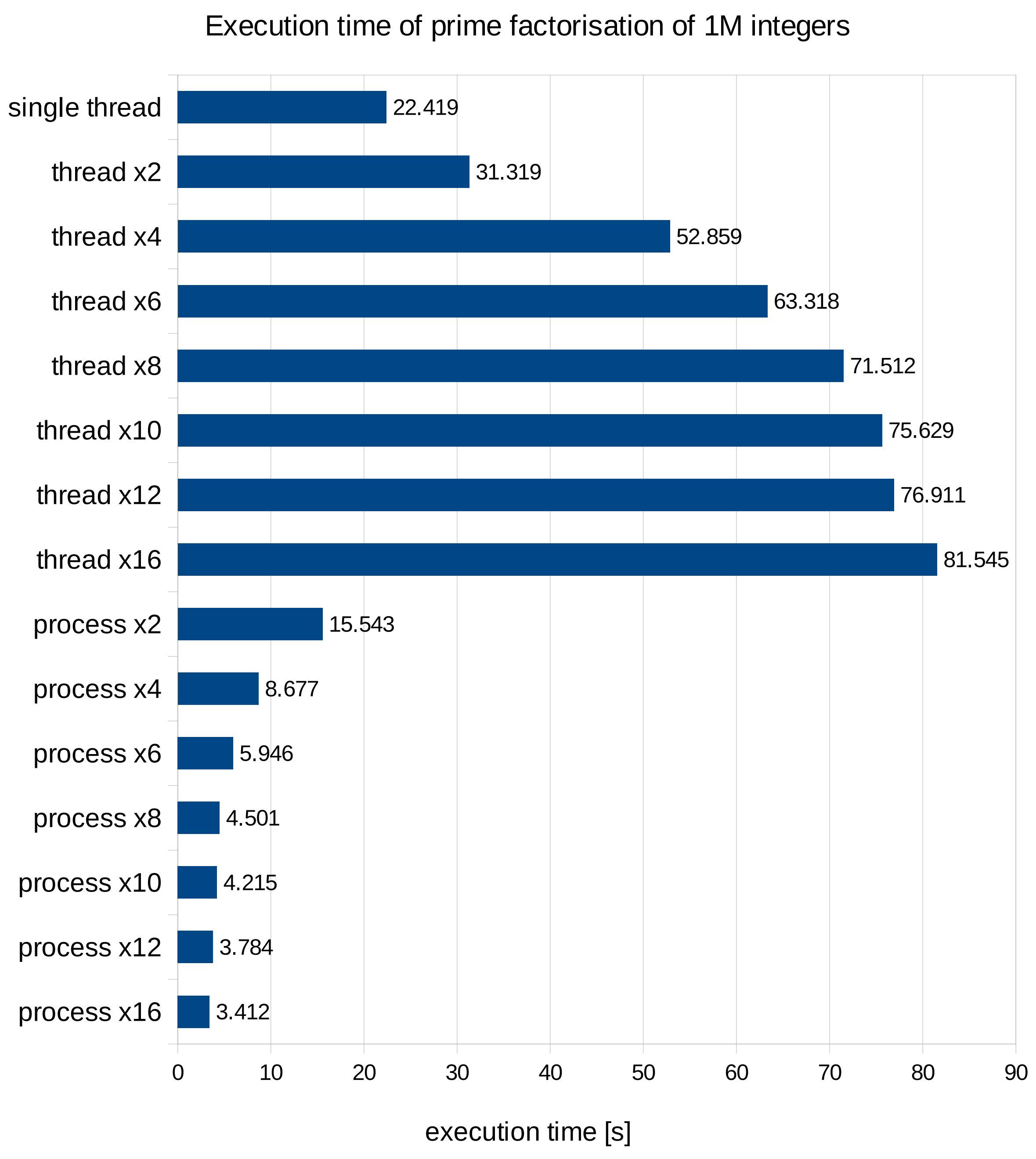


Figure 4.1: Results of the prime factorisation benchmark

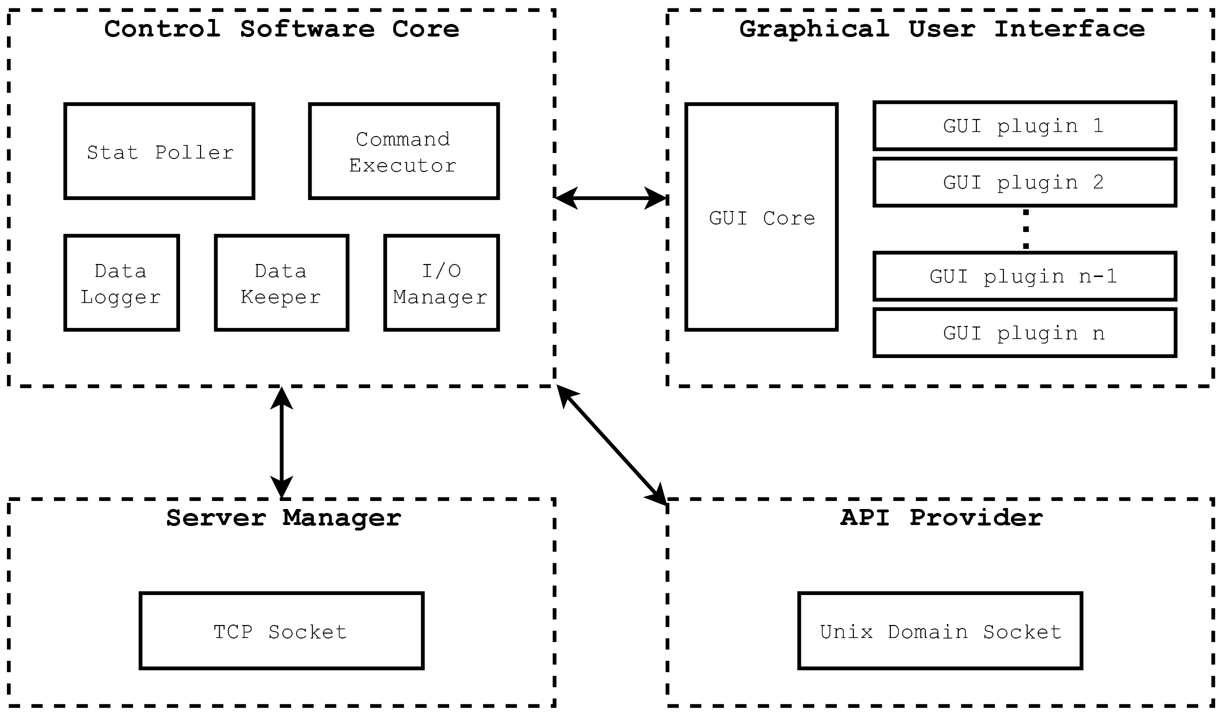


Figure 4.2: The developed control software scheme

volts output signal).

The process periodically calls the `poll` method of the LinuxCNC Python interface status channel and the error channel to obtain the status variables of the device, the eventual errors and reads out the sensors output signal. The process then sends the obtained variables to other processes of the control software core, such as the Data Logger and Data Keeper and to the user interface.

4.2.1.2 Data Logger

The Stat Poller process sends the obtained data to a queue leading to the Data Logger process. The Data Logger receives the data and saves is periodically to a plain-text-based output file.

4.2.1.3 Data Keeper

The Data Keeper receives the data from the Stat Poller and keeps it in an array. It provides a simple post-processing functionality, such as a floating average. The Data Keeper then sends the data, for instance, to the user interface in order to show the data in a graph.

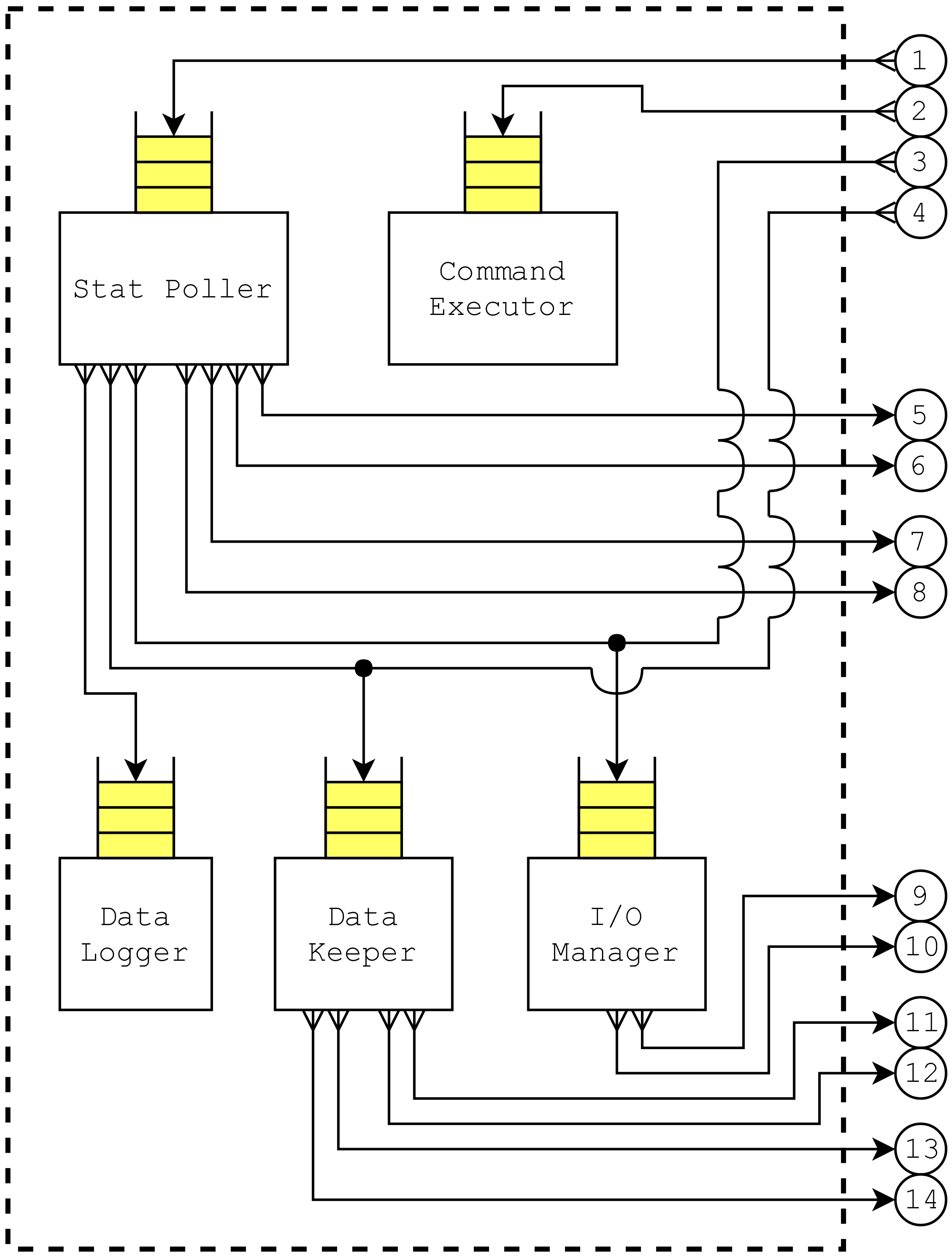


Figure 4.3: The control software core inter-process communication scheme

4.2.1.4 Command Executor

The Command Executor is a process based on the command channel of the LinuxCNC Python Interface. It provides the execution of the dynamically generated Python string commands using the `exec` statement.

4.2.1.5 I/O Manager

The I/O Manager provides the communication with the real-time HAL using the Python HAL component (see subchapter 2.3.1) or using analog and digital inputs/outputs. The Python HAL component is, for instance, used for experiments driven by non-linear displacement or force-driven experiments, see subchapter 4.4.5.

4.2.2 User Interface

The user interface of the control software runs separatedly from the control software core which results in a performance increase on the side of user interface as well. It is a single-process object connected with the control software core using various queues. The user interface is built ontop of user interface core, which provides all necessary functionality for communication with the control software core, see Figure 4.4 and Appendix B.

The user interface is designed as a set of separate plugins, which gives it an ability to be modular and makes it very effectively adjustable for particular applications. One of the plugins is a plotting plugin which has been a significant limitation of the first-generation control software, as it was developed on top of the matplotlib backend for the Qt framework. The plotting plugin of the new-generation control software is built on top of the PythonQwt library which provides more plotting performance over the matplotlib.

4.2.2.1 Plotting performance benchmark

To demostrate the plotting performance difference of PythonQwt and matplotlib, a simple benchmark was performed. It was run on the Intel Xeon W-2145 @4.5 Ghz (8-core, 16-thread) CPU. The benchmark is based on plotting a single period of a sinus function $f(x) = sin(x)$. The script used for the benchmark is shown in Appendix C.

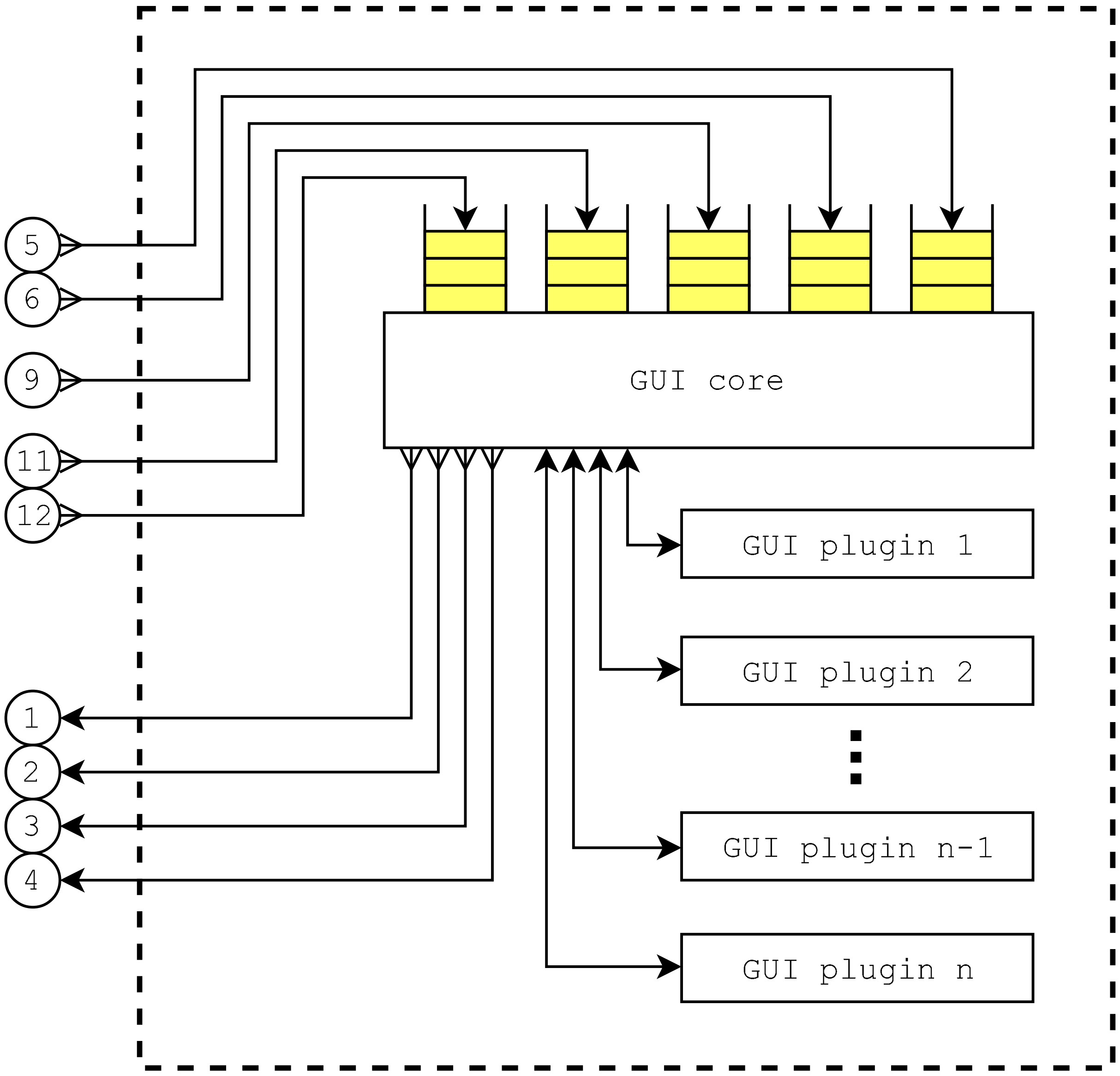


Figure 4.4: The control software user interface scheme

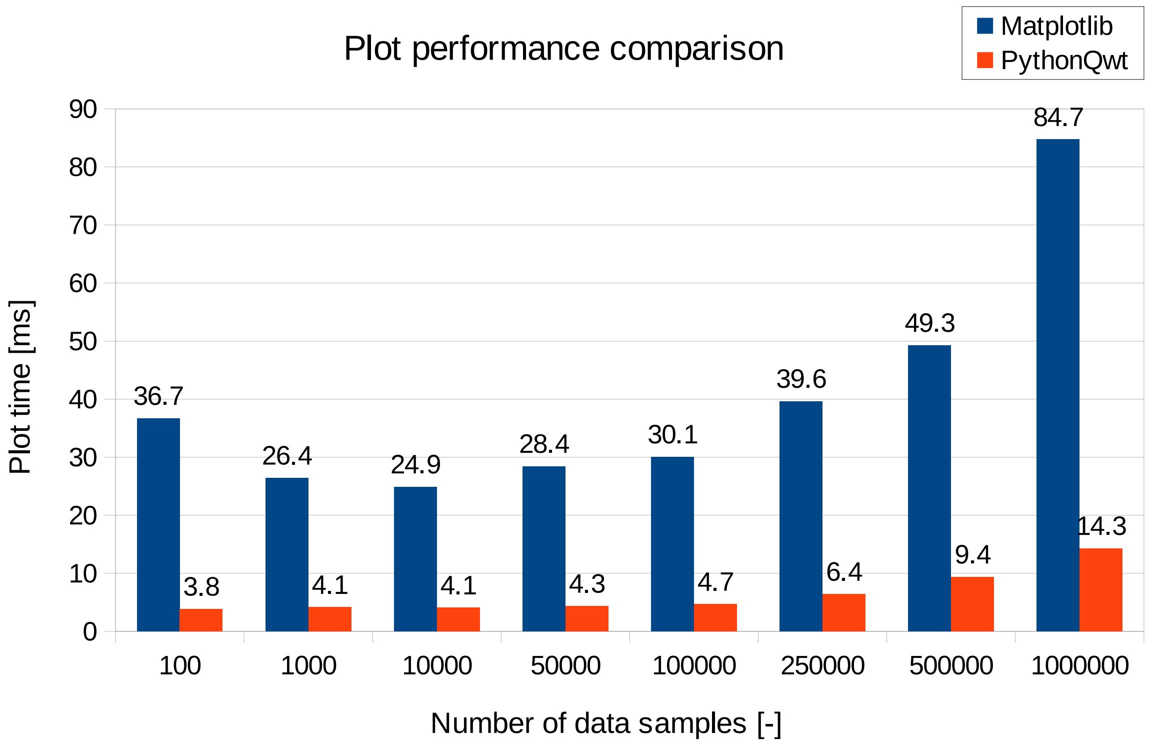


Figure 4.5: Matplotlib and PythonQwt plot comparison results

Unlike matplotlib which is written entirely in Python, PythonQwt is a Python wrapper (or binding) for the Qwt library which is written in C++, so it can deliver more performance over the matplotlib library.

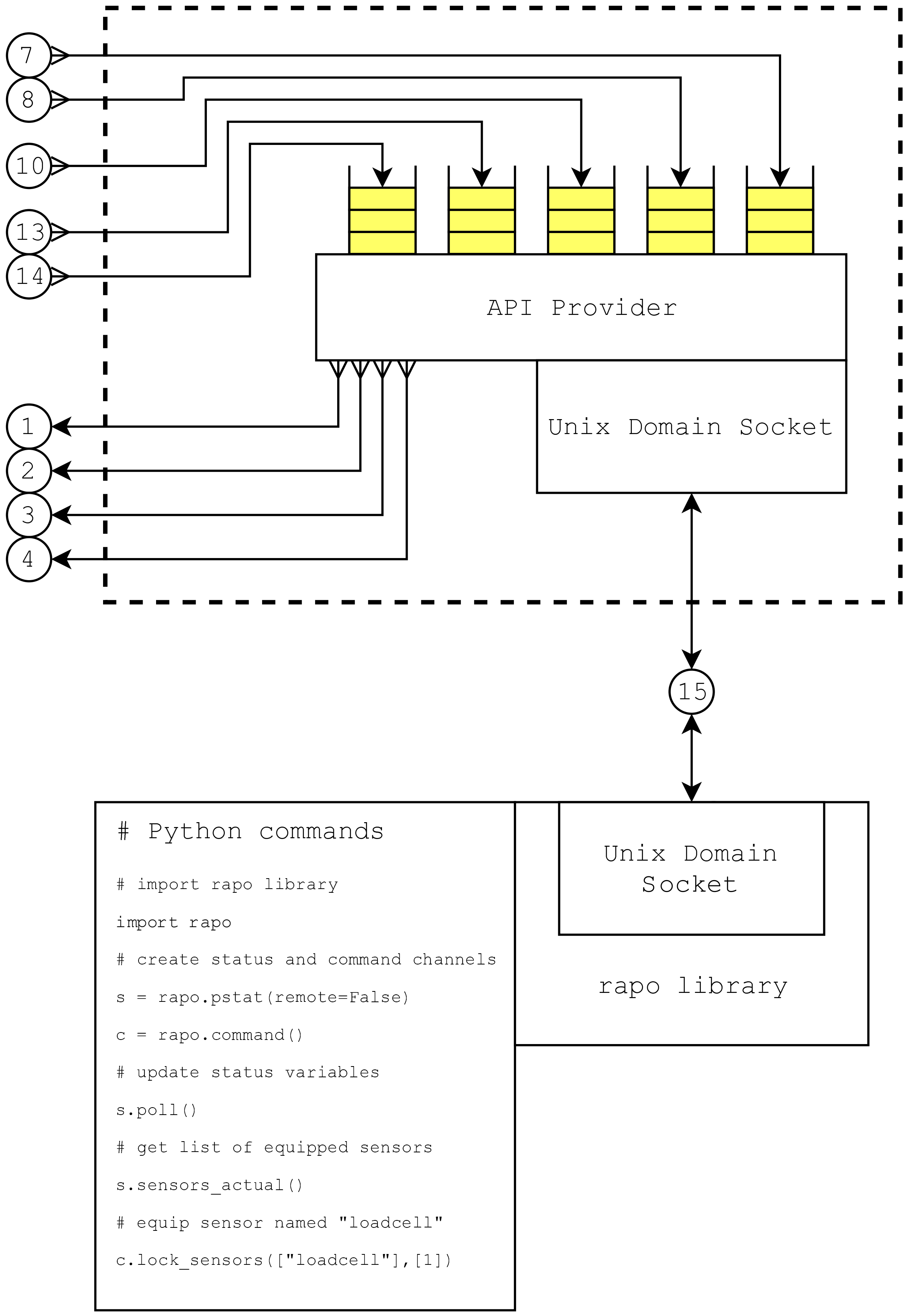
The performance increase also enabled the new-generation plot plugin to plot multiple data series in real-time.

4.3 Custom Python script execution

The control software also allows sending commands to the control software core through an external Python script. The external Python script execution can be used for the automated measurement, etc.

The communication of the external Python script with the core was implemented by an API Provider process which is connected with the control software core using queues, see Figure 4.2 and Figure 4.6 and Appendix B.

The external Python script is connected with the API Provider process using the custom



```
# Python commands

# import rapo library

import rapo

# create status and command channels

s = rapo.pstat(remote=False)

c = rapo.command()

# update status variables

s.poll()

# get list of equipped sensors

s.sensors_actual()

# equip sensor named "loadcell"

c.lock_sensors(["loadcell"],[1])
```

Figure 4.6: The API provider and rapo library connection scheme

`rapo` library (see subchapter 4.4.7) which features a Unix domain socket.

4.3.1 Unix domain socket

The Unix domain socket (or inter-process communication socket) is a communication endpoint for sending messages between processes running on the same host system. Unix domain sockets share the same semantics as network sockets, but Unix domain sockets do not connect via a hostname and port. They connect using a file system, thus, the whole communication occurs entirely within the operating system.

In order to send Python data objects such as a list, dictionary, etc. through the Unix domain socket, the data object must be firstly serialised into a stream of bytes format which can be pushed through the socket.

4.3.2 Serialising modules

Python features various serialising modules such as `pickle`, `cPickle`, `json`, etc. which provide protocols for serialising and deserialising Python data objects.

4.3.2.1 Pickle and cPickle modules

The pickle module is part of the Python standard library and is widely used for serialisation in Python. However, it is written entirely in Python which limits its performance and the data format used by pickle is Python-specific, therefore, it is not suitable for applications with interoperability requirements.

The Python standard library also features a cPickle module which provides the same fuctionality as the pickle module. Unlike pickle, cPickle is written in the C programming language, so cPickle gives more performance than pickle which makes it more suitable for performance-oriented applications. Pickle and cPickle feature various serialising protocols:

- Protocol 0 is the original ASCII protocol which is human-readable and is backwards compatible with other versions of Python.

- Protocol 1 is an obsolete binary format which is also backwards compatible. It has been substituted by protocol 2.
- Protocol 2 was introduced with Python version 2.3 and is the highest protocol of Python 2.x. It provides much more efficient serialisation of the new-style classes.[1]
- Protocol 3 was introduced in Python 3.0. It has explicit support for bytes objects and cannot be unpickled by Python 2.x.
- Protocol 4 was added in Python 3.4. It is the highest protocol of Python 3. It adds support for very large objects, pickling more kinds of objects, and data format optimisations.

4.3.2.2 JSON module

Javascript Object Notation (JSON) is a standardized format used for serialising data objects to a human-readable format. Unlike pickle and cPickle, JSON is a language independent data format derived from JavaScript.

It uses conventions that are compatible with programming languages including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal serialising format for applications with data-interchange and interoperability requirements.

The proposed control software is entirely developed in Python programming language, thus, any data-interchange capability is not required. The only requirement for the serialising module used within the developed software is to provide as much performance as possible. For that reason a benchmark comparing the serialising performance and memory consumption of the serialised object was performed.

The benchmark is based on serialising and deserialising Python list and dictionary objects. These objects were chosen purposefully as they are the data objects transfered through the Unix domain socket.

[1]A new-style class inherits from the object class and is a recommended option for creating a class in modern Python.

The list used for the serialisation benchmark was a range of one million integer numbers and the dictionary consisted of one million (key, value) pairs. In case of the pickle and cPickle modules, the performance of the different serialising protocols was also benchmarked. In addition to the performance benchmark, the size of the serialised data object is also a significant factor in terms of the memory consumption and data transfer performance, so the size of the serialised object was also compared. The whole script used for the benchmark is included in Appendix D. The benchmark was run on the Intel Xeon W-2145 @4.5 Ghz (8-core, 16-thread) CPU.

4.3.2.3 List serialisation benchmark

List serialisation performance is shown in Figure 4.7. The benchmark results show that the serialisation of the Python list using the pickle module is significantly slower regardless of the used serialising protocol compared to the JSON module or even the cPickle module. The cPickle module was able to provide the best performance out of the tested modules. Unlike pickle, the cPickle performance differed depending on the serialising protocol used. CPickle protocol 0, which is the default protocol is the slowest out of the protocols tested but it gives a comparable performance with JSON. cPickle protocols 1 and 2 have significantly better performance than cPickle protocol 0 and JSON.

In the case of the deserialisation performance, cPickle shows the best performance, the same as during the serialisation task. JSON performance is slightly lower, whereas the performance of the pickle module is incomparably worse. The list deserialisation performance is shown in Figure 4.8.

An additional aspect of the serialising module comparison was the serialised object size. The size of the object affects the data transfer performance and memory consumption. Pickle and cPickle protocols 1 and 2 serialise the data to a binary format which is the least memory-intensive compared to protocol 0 or JSON. Protocol 0 and JSON produce a human-readable ASCII-based format which consumes more memory compared to the binary format. The results of the serialised object size comparison is shown in Figure 4.9.

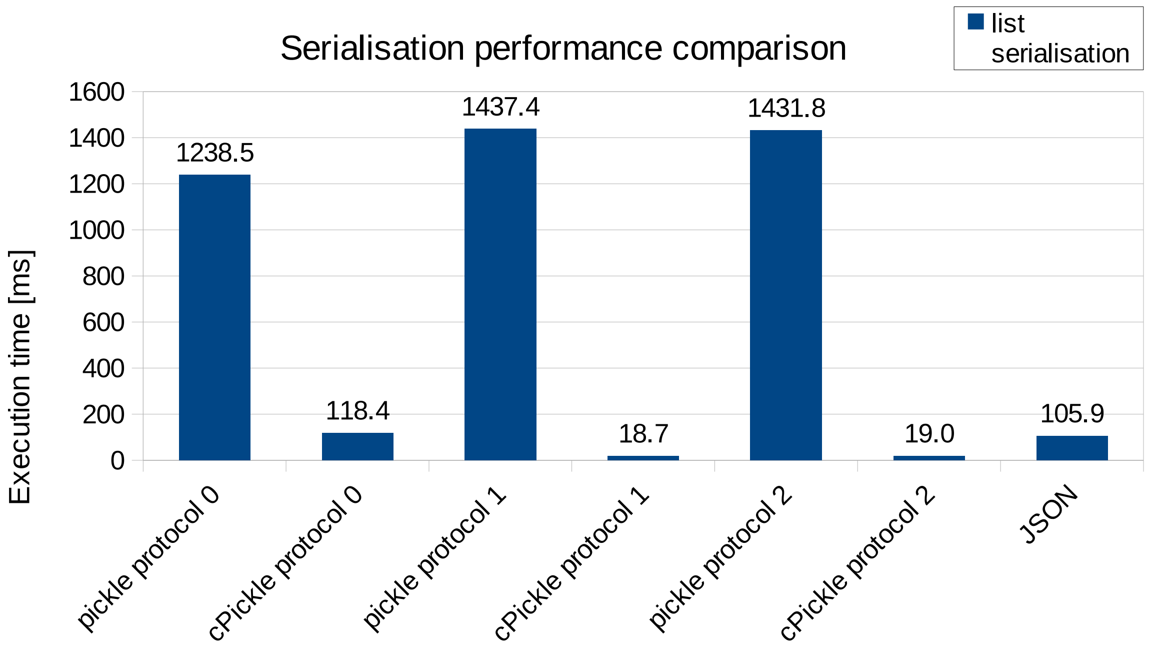


Figure 4.7: Python list serialisation performance comparison

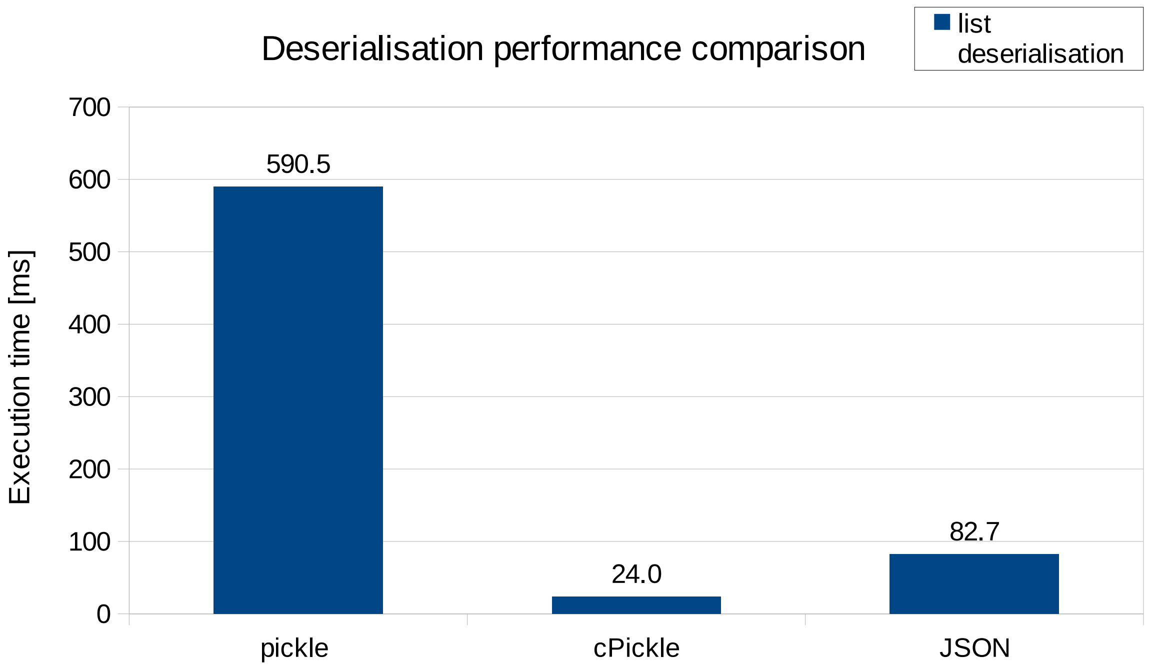


Figure 4.8: Python list deserialisation performance comparison

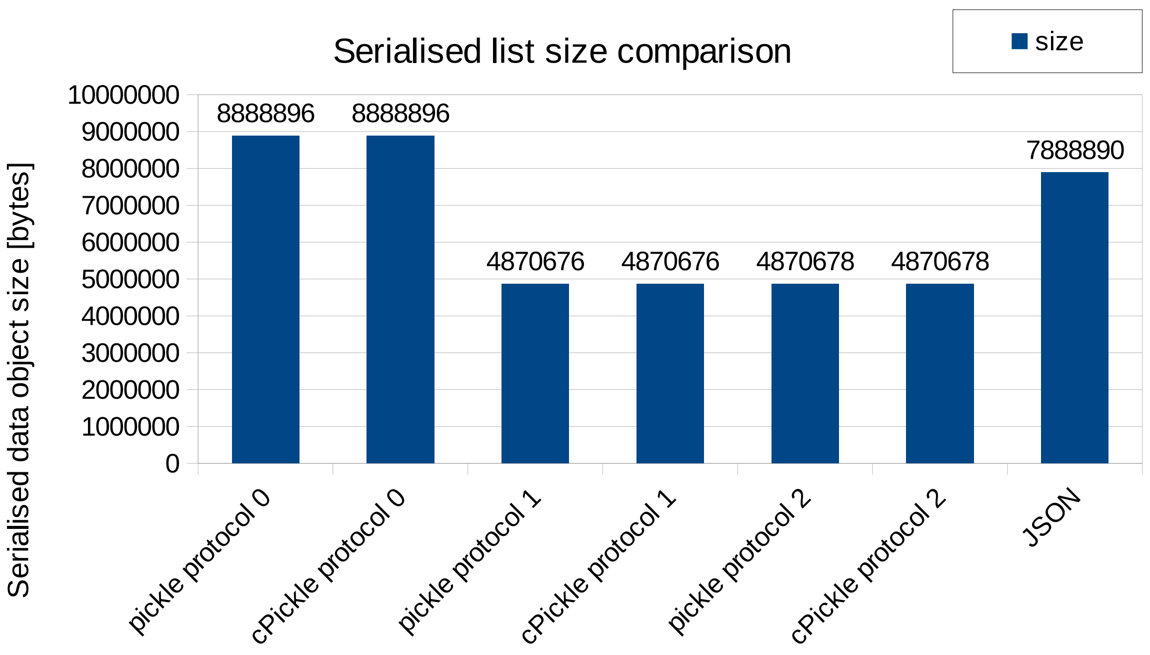


Figure 4.9: serialised Python list size comparison

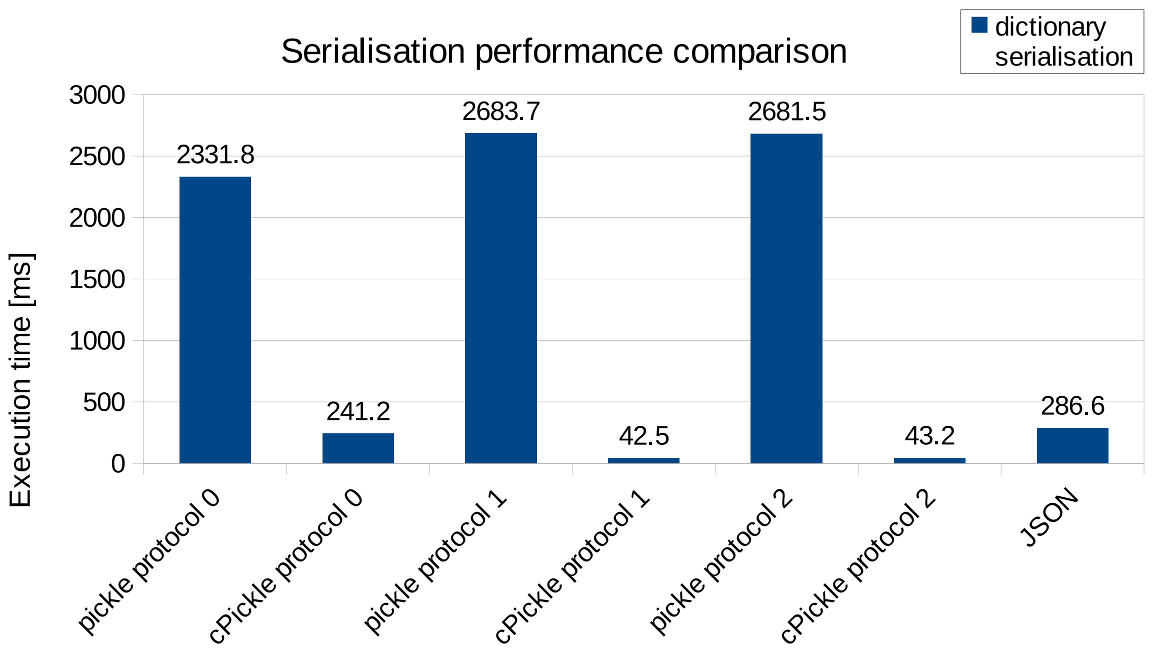


Figure 4.10: Python dictionary serialisation performance comparison

4.3.2.4 Dictionary serialisation benchmark

Besides the list serialisation perfomance comparison, a dictionary serialisation was also benchmarked. The results are very similar to the list serialisation. When it comes to the dictionary serialisation, the pickle module exhibits significantly worse results during the dictionary serialisation when compared to the cPickle and JSON modules. CPickle protocol 0 is slightly faster than JSON and in general the cPickle protocols are the fastest. The performance difference between cPickle protocols 1 and 2 is marginal, see Figure 4.10.

The cPickle module delivers the best dictionary deserialisation performance likewise in the list deserialisation task. JSON is second with a large gap, the worst results were achieved by the pickle module, see Figure 4.11.

In relation to the memory consumption, the JSON module has the lowest demands, the demands of cPickle protocols 1 and 2 are slightly higher. The serialised dictionary by the pickle module is approximately twice as large in size, see Figure 4.12.

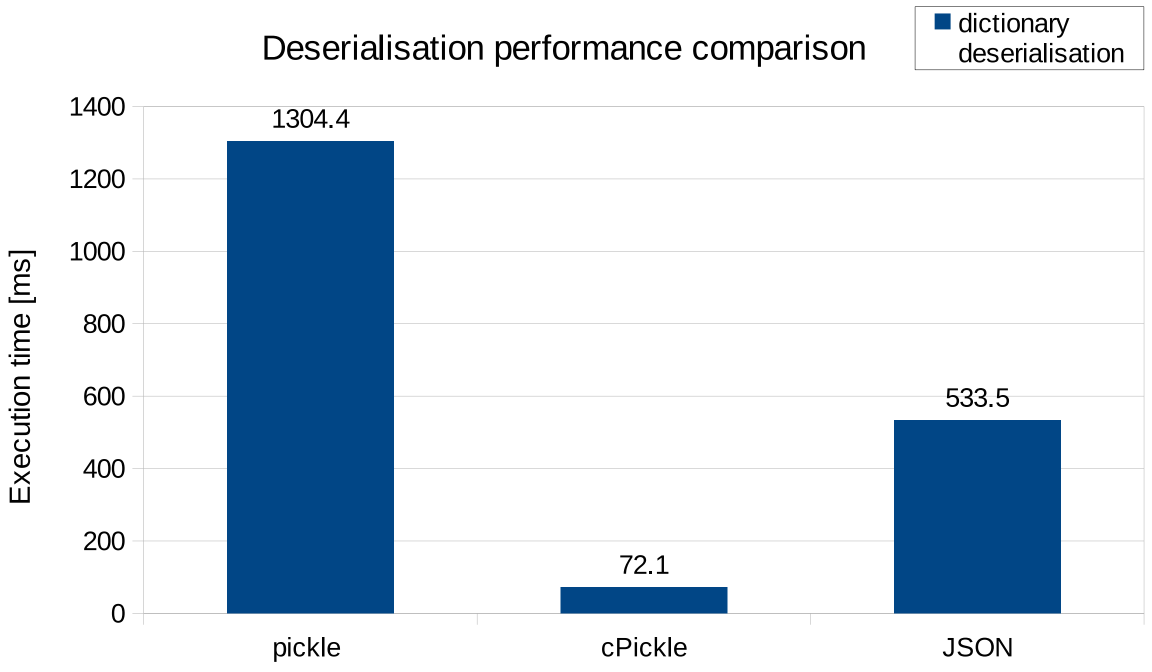


Figure 4.11: Python dictionary deserialisation performance comparison

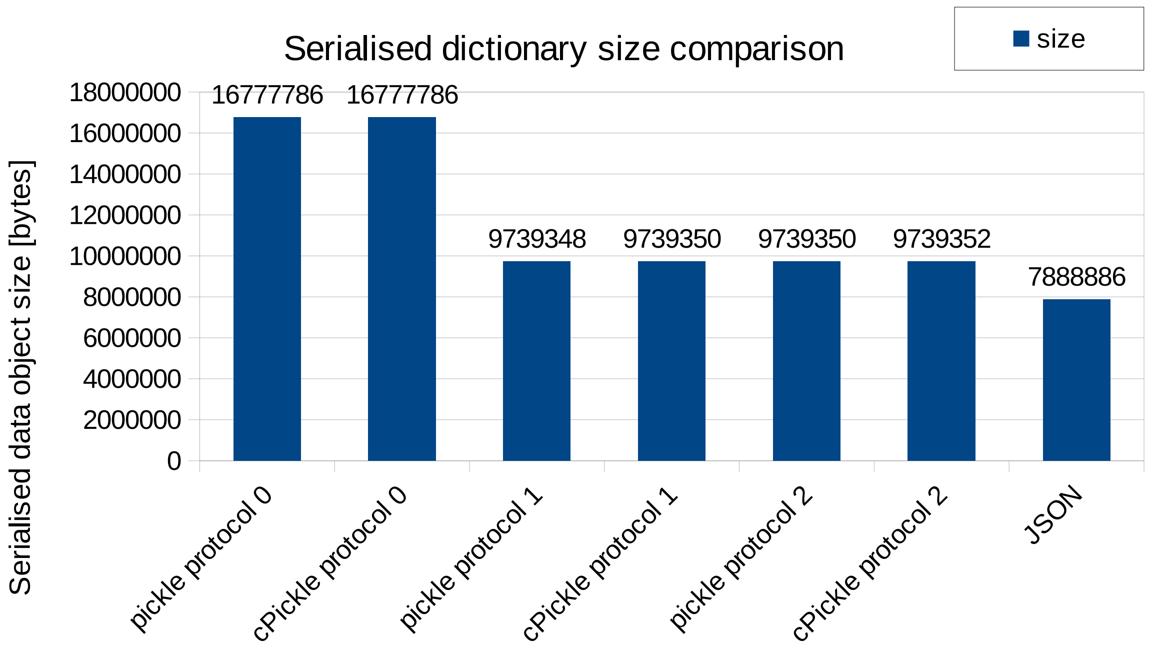


Figure 4.12: serialised Python dictionary size comparison

4.3.2.5 Recapitulation

The list and dictionary serialisation/deserialisation benchmarks pointed out that cPickle is the most suitable module for performance-oriented applications. Especially cPickle protocols 1 and 2 deliver much more performance compared to the pickle and JSON modules.

The developed control software does not require any interoperability capabilities as it is developed using the Python programming language only. The aim of the control software is to provide as much performance as possible which makes cPickle a perfect candidate for the serialising module. The benchmarks showed that cPickle is the most performance-oriented serialising module out of the benchmarked modules. In particular, the cPickle protocol 2 fulfils the performance requirements the best so it was chosen as the serialising module for the developed control software.

4.4 Enhanced Functionality

4.4.1 Introduction

The need to develop the proposed control system was not motivated by the performance limitations only, but also by progress in the device's construction and the use of advanced experimental procedures.

Since the first-generation software has been developed, various new experimental devices have been constructed which also involved developing a new set of functionality features, such as the support for multiple and various types of sensors or advanced experimental procedures, etc.

4.4.2 Sensor support

One of the newly developed experimental devices was a loading device intended to perform 4-point bending experiments [22]. This involves the usage of two load cells which the first-generation control software did not support, see Figure 4.13.

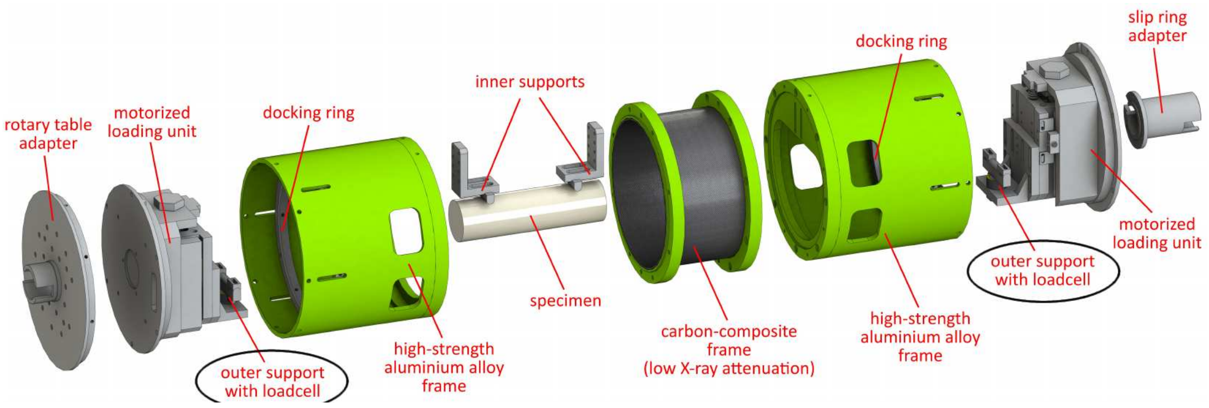

rotary table adapter
motorized loading unit
docking ring
inner supports
docking ring
slip ring adapter
motorized loading unit
outer support with loadcell
high-strength aluminium alloy frame
specimen
carbon-composite frame (low X-ray attenuation)
high-strength aluminium alloy frame
outer support with loadcell

Figure 4.13: Experimental device for the 4-point bending

The new-generation control software supports a theoretically unlimited number[2] of sensors. Moreover, another experimental device [23] involves the support for different types of sensors (load cells and thermometers) in terms of measuring the physical quantities, due to simultaneous force measurement and circulating fluid temperature measurement.

Each sensor is identified and initialised by the control software using its own initialising file with a specific header and parameters. The header (the first line of the initialising file starting with a # sign) determines the type of the sensor (load cell, thermometer, etc.), the parameters inside the file describe the important sensor parameters necessary for the realiable and precise measurement, such as sensitivity, etc. The load cell initialising file content is shown in Figure 4.14, the thermometer initialising file content is shown in Figure 4.15.

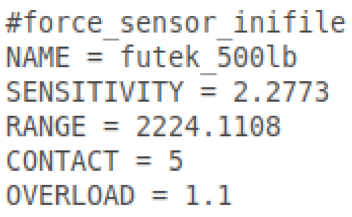

```
#force_sensor_inifile
NAME = futek_500lb
SENSITIVITY = 2.2773
RANGE = 2224.1108
CONTACT = 5
OVERLOAD = 1.1
```

Figure 4.14: Load cell sensor initialising file content

[2]For the sensor signal readout a LabJack T7 Pro (Labjack Corporation, USA) is used which features up to 14 analog input channels. With this data acquisition setup, the control system is able to handle up to 14 sensors, which is the hardware limitation.

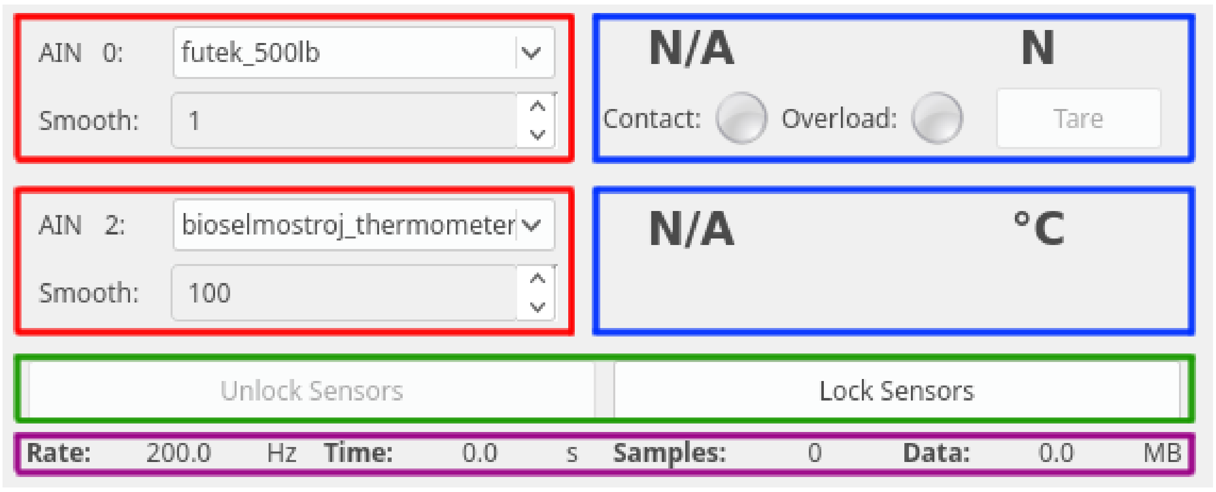


Figure 4.16: The plugin for handling the sensors

```
#thermometer_sensor_inifile
NAME = bioselmostroj_thermometer
RESISTANCE = 1001.805714
SENSITIVITY = 3.8216571428
```

Figure 4.15: Thermometer sensor initialising file content

These initialising files are loaded into a control software sensor database during the control software startup. The sensors plugin of the control software user interface provides the functionality for operating the sensors, see Figure 4.16.

The plugin for handling the sensors includes comboboxes[3] used to select actual sensors from the list of the loaded sensor initialising files. Underneath these fields are spinboxes[4] to specify a floating average window width, see the red boxes in Figure 4.16. The floating average is used to eliminate noise which may occur in the data. Based on initialising file contents of the chosen sensors, the display bars of the sensors are dynamically generated, see the blue boxes in Figure 4.16.

The display bar of the load cell (the upper blue box in Figure 4.16) consists of a label[5] for displaying the actual force with $[N]$ units. The display bar of the load cell also features

[3]QCombobox is a selection widget that displays the current item and can pop up a list of selectable items.

[4]QSpinBox allows the user to choose a value by clicking the up/down buttons or pressing up/down on the keyboard to increase/decrease the value currently displayed. The user can also type in the value manually.

[5]QLabel is a widget used for displaying the text or an image.

various signalisation mechanisms. For instance, it provides a diode[6] signaling whether the load cell is in contact with the tested sample. If the actual force is greater than the value of the $CONTACT$ parameter in the load cell initialising file, the diode changes from grey to orange. The other diode is used for the load cell overload signalisation. If the actual force is below the value of the $RANGE$ parameter, the diode remains grey. When the force exceeds the $RANGE$ limit, the diode becomes orange. If the force even exceeds the $RANGE * OVERLOAD$ value, the diode becomes red to signal the eventual load cell damage if the load continues to increase further. The control software automatically triggers the E-STOP (emergergency-stop) procedure and stops the machine to prevent the load cell damage whenever the load cell could be in danger due to a high load. The display bar includes a tare button[7] which tares the load cell based on the last data samples. The number of samples used for determining the tare value depends on the data acquisition rate and on the signal noise.

Unlike the load cell display bar, the display bar of the thermometer (the bottom blue box in Figure 4.16) includes a label for displaying the actual temperature with $[°C]$ units. Any other functionality is not needed.

The sensors plugin also includes two buttons to lock and unlock the sensors, see the green box of in Figure 4.16. The lock button is used to set the chosen sensors and start reading values from them. Until the sensors are not set/locked, all the controls of the user interface except the E-STOP and POWER buttons remain disabled, unable to send any commands to the control software core for safety reasons. The button for unlocking the sensors has inverse functionality. It unlocks the sensors which have been locked previously in order to stop reading sensor values or lock other sensors.

At the bottom of the sensors plugin a measured data status bar is located. The status bar provides simple information about the measurement, such as the data aquisition rate, the time elapsed by the measurement, the number of data samples obtained and the memory consumed by the data, see the purple box in Figure 4.16.

[6]A diode image put in the QLabel

[7]The QPushButton is perhaps the most commonly used widget in any graphical user interface. Pushing it (click) makes the button command the computer to perform some action.

4.4.3 Axes position bar

The axes position bar is a plugin of the user interface to show the actual position of the axes. It provides further information, such as the axis coordinate and axis units, see the purple box in Figure 4.17. The plugin also features labels to show the actual position stated by the LinuxCNC motion interpreter, see the value in bold in the red box in Figure 4.17. The plugin also indicates the actual position percentagewise using a progress bar[8], i.e., if the actual position reaches the minimum axis limit, the progress bar indicates 0 %. If the actual position reaches the maximum limit, the progress bar indicates 100 %. The axis minimum and maximum limits are shown within the plugin as well, see the orange boxes in Figure 4.17.

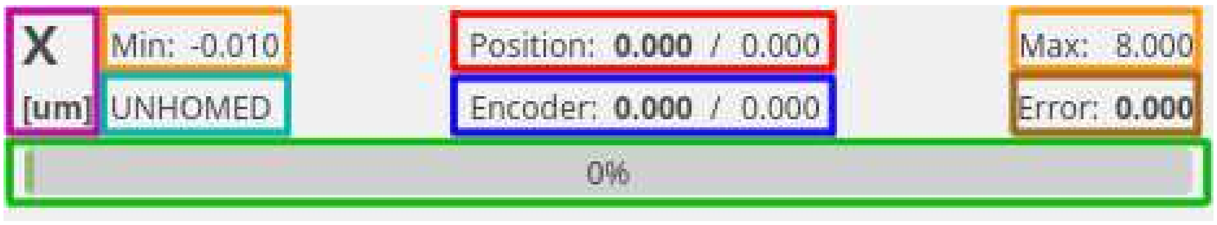


Figure 4.17: The axes position plugin of the user interface

The first-generation control software did not feature any functionality for showing the encoders position directly in the axes position bar. It was done using an external GladeVCP component, see subchapter 2.3.6.

4.4.3.1 Encoder support

The newly developed experimental devices are mostly equipped with multiple encoders. An encoder is a device which provides the position feedback, see subchapter 2.2 and subchapter 2.1.1.2.

The new-generation control software enables one to show encoders's position within the axes position bars (see the value in bold in the blue box in Figure 4.17) instead of using the external GladeVCP component. This required accessing the real-time HAL from Python

[8]QProgressBar is used to give the user an indication of the progress. The progress bar uses the concept of steps. It is set up by specifying the minimum and maximum possible step values, and it will display the percentage of the steps that have been completed.

in order to obtain the encoder position. It was achieved by linking the encoder position in HAL to the analog input of the LinuxCNC Python interface which can be accessed from Python directly.

4.4.3.2 G92 offset

The G92 command of the G-code[9] is used to set the start position (origin) offset for one or more axes. The first-generation did not support this functionality at all. The offset functinality is useful for the measurement to set the origin of the coordinate system when the experimental device reaches contact with the sample. Thus the experimental procedure begins with the position equal to zero.

The actual position and encoder position display areas (see the red and blue boxes in Figure 4.17) consist of two values each. The values are separated by a slash. The value to the left of the slash is the absolute actual position of the axis. The value to the right of the slash is the G92 position which is relative to the shifted origin of the axis. In Figure 4.17, the actual and the G92 positions are equal which means that the G92 offset is zero. If the G92 offset is present (i.e., the G92 offset is a non-zero value), the G92 position becomes bold to signal that the axis origin has changed.

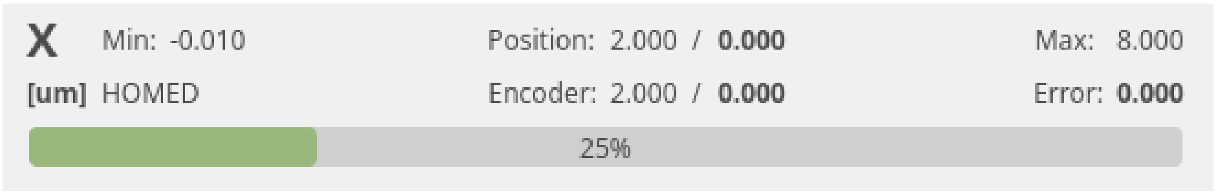

X Min: -0.010 Position: 2.000 / **0.000** Max: 8.000
[um] HOMED Encoder: 2.000 / **0.000** Error: **0.000**
25%

Figure 4.18: The axes position plugin of the user interface with G92 offset active

In Figure 4.18, the G92 offset is shown. The absolute axis position equals 2.000 µm, the G92 position (the position relative to the new axis origin) equals 0.000 µm. In this case, the G92 offset equals 2000 µm, see Figure 4.18.

[9]G-code is a common name for the most widely used numerical control (NC) programming language. It is mainly used in computer-aided manufacturing to control automated machine tools.

4.4.3.3 Positioning error

If an axis is equipped with an encoder, the positioning error can be determined. The positioning error is defined as the difference of the position given by the LinuxCNC motion interpreter and the position given by the feedback encoder. It is displayed within the axes position bar as well, see the brown box in Figure 4.17. The smaller the positioning error, the more precise the positioning is.

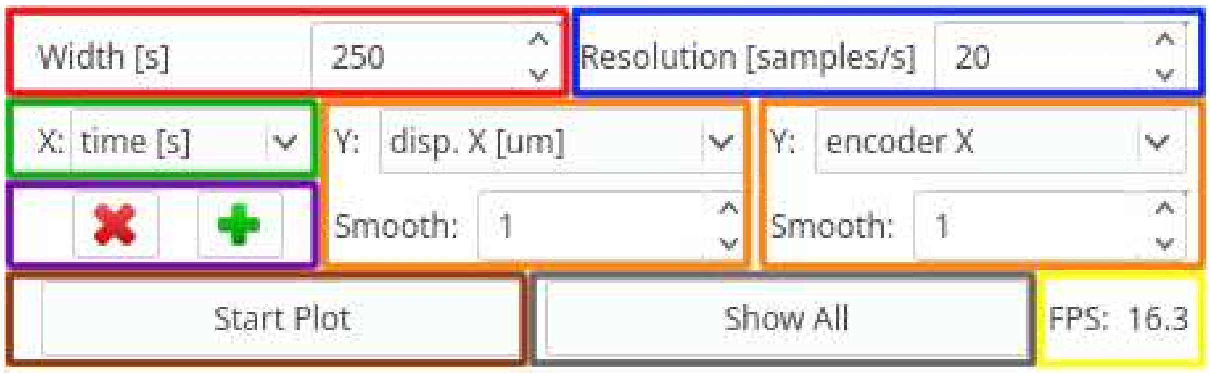
4.4.3.4 Homed status

The axes position bar of the new-generation control software also provides additional information, such as whether the axis is homed (i.e., a reference point has been found). Each axis should be homed properly in order to provide precise positioning. The axes position plugin displays the *HOMED* text if the axis is homed, otherwise it displays the *UNHOMED* text, see the cyan box in Figure 4.17.

4.4.4 Plot plugin

One of the main disadvantages of the first-generation control software was its limited real-time plot performance. The plot performance of the new-generation control software has been significantly improved thanks to the control software architecture described in subchapter 4.2.

Besides the performance improvement, the plot plugin of the user interface has been extended with new functionality.


Width [s] 250
Resolution [samples/s] 20
X: time [s]
Y: disp. X [um]
Y: encoder X
Smooth: 1
Smooth: 1

FPS: 16.3

Figure 4.19: The plot plugin of the user interface

One of the configuring parameters of the plot is the plot time window w, see the red box in Figure 4.19. The plot time window defines the time period of the past data samples shown within the plot. Another parameter is the plot resolution r. The resolution defines the number of data samples to plot within a single second of the plot time window. The higher the plot resolution, the more demanding the plotting is in terms of hardware resources due to larger number of data samples, see the blue box in Figure 4.19. Total number of the data samples n within the plot can be determined as:

$$n = w \cdot r, \tag{4.1}$$

where w is the plot time window and r is the plot resolution. The total number of the samples is used for the real-time plot performance prediction in formula 4.2.

The plot enables plotting of a multiple data series. The data series of the plot can be dynamically added with a plus sign button or removed with cross sign button, see the purple box in Figure 4.19. The data series share the same X-coordinate, the Y-coordinate is specific to the data series, see the green and orange boxes in Figure 4.19.

The plot plugin also features simple real-time processing functionality, such as a floating average. The *smooth* input field parameter defines the floating average window. If the *smooth* parameter equals 1, the floating average does not take effect due to the floating average window size. When using the floating average real-time processing, it is important to take the increased computational demands into consideration. Therefore, the floating average window size is limited up to 1000.

For the real-time plotting performance prediction, the plot plugin provides a simple indicator showing the possible Frames per second (FPS), see the yellow box in Figure 4.19. The FPS prediction is based on the plotting performance of the PythonQwt library. The results of the PythonQwt plotting benchmark (see subchapter 4.2) have been used to find a linear function:

$$T(n) = K \cdot C \cdot (A \cdot n + B), \tag{4.2}$$

where the function T describes the time period needed to plot n data samples. Coefficients A and B are given by linear regression based on the plotting benchmark. Coefficient C defines the number of data series in the figure and coefficient K is a safety factor.

The safety factor is primarily used to balance the user interface overall performance and fluency. In most of the applications, the safety factor $K = 3$, which reserves one third of the hardware resources to the real-time plotting functionality. Once the plotting parameters are configured, the real-time plot starts by using the start plot button, see the brown box in Figure 4.19.

Besides the real-time plotting functionality, the plot plugin also provides static plotting. The static plotting is realised separately from the user interface using the external plotting tootkit Gnuplot in cooperation with the Python wrapper PyGnuplot, see subchapter 2.4.7.

4.4.5 Force control

Another enhancement of the new-generation control software over the first-generation is the support for force-driven experiments. An experiment can be driven by a constant force command or by a force function. The control software uses the load cell feedback to calculate the axis velocity to copy the control command. The force control enables the controlling of the loading force independently to the displacement, thus, adapting the loading to a particular sample.

The force control is realised using the PID controller featured by HAL, see subchapter 2.3.2 and Figure 4.20. The setpoint $r(t)$ of the PID controller may be various time-dependent force functions, such as a linear function, periodic waves or a constant value. The error $e(t)$ is defined as the difference of the setpoint $r(t)$ and the actual force $y(t)$ measured by the load cell. Based on the proportional, intergral and derivarive terms, the manipulated variable $u(t)$ is determined. The manipulated variable $u(t)$ is the velocity command for the LinuxCNC motion interpreter. Based on the motion velocity $u(t)$, the measured force (output variable) is affected.

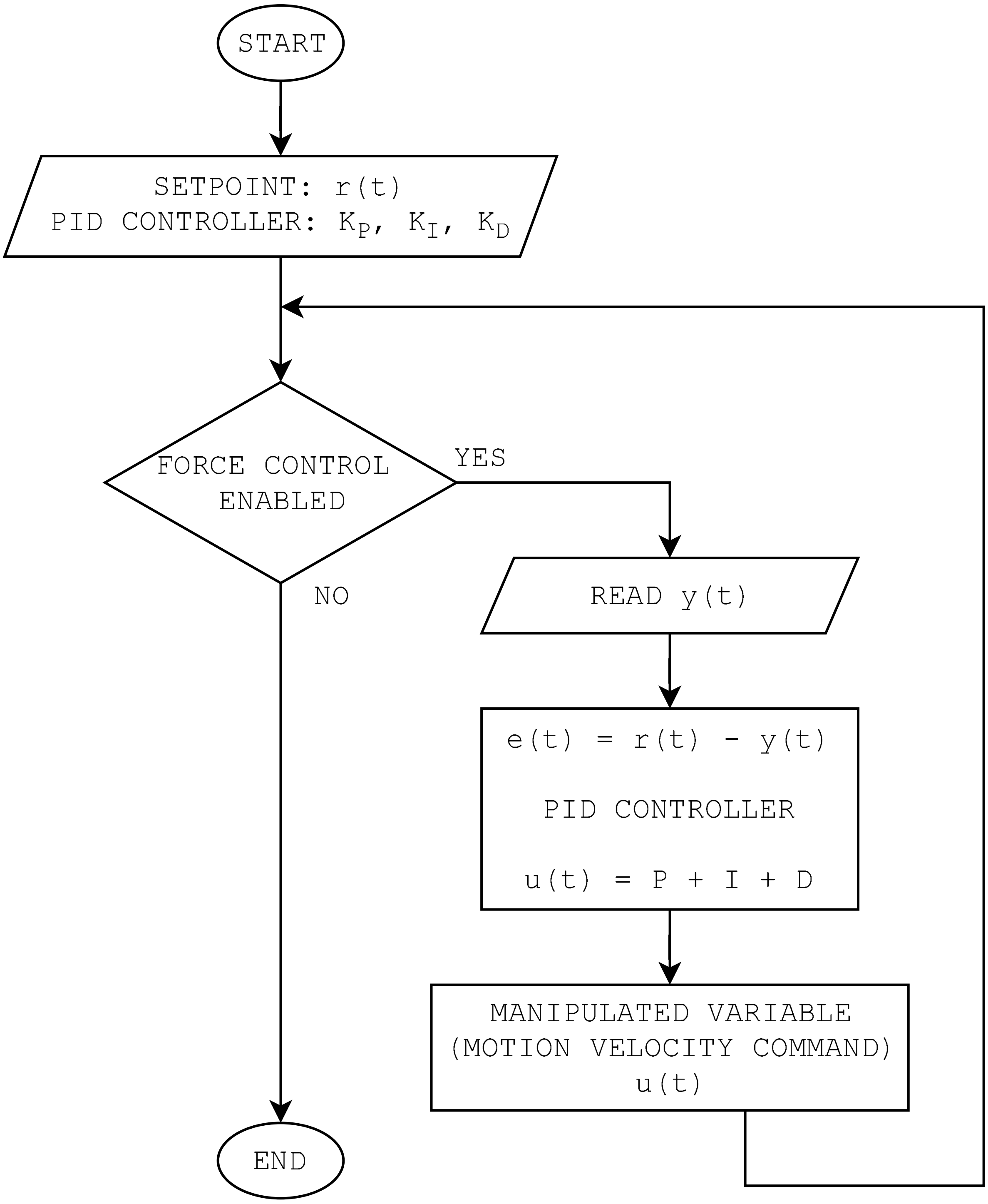
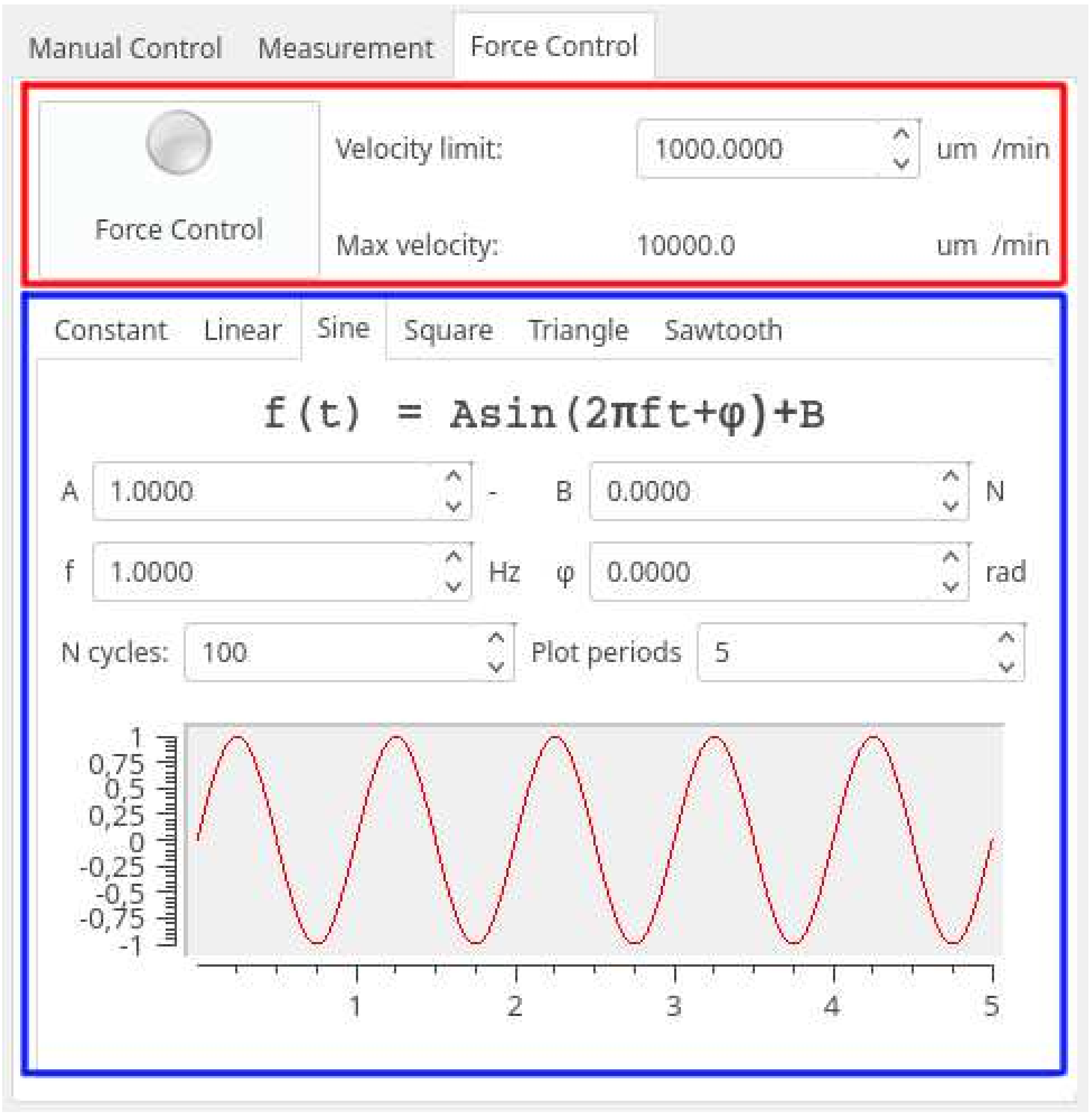


Figure 4.20: Force control flowchart

The force command plugin consists of controls common to all force command functions, such as sine, square, triangle and sawtooth waves (see the red box in Figure 4.21) and multiple tabs, each tab is dedicated to a specific force command function (see the blue box in Figure 4.21).



Force Control

Velocity limit: 1000.0000 um /min

Max velocity: 10000.0 um /min

$f(t) = A\sin(2\pi ft+\varphi)+B$

A 1.0000 -   B 0.0000 N

f 1.0000 Hz   φ 0.0000 rad

N cycles: 100   Plot periods 5

Figure 4.21: Force control plugin overall view

4.4.5.1 Common controls

Some controls of the force control plugin are common to all the force command tabs, see Figure 4.22.

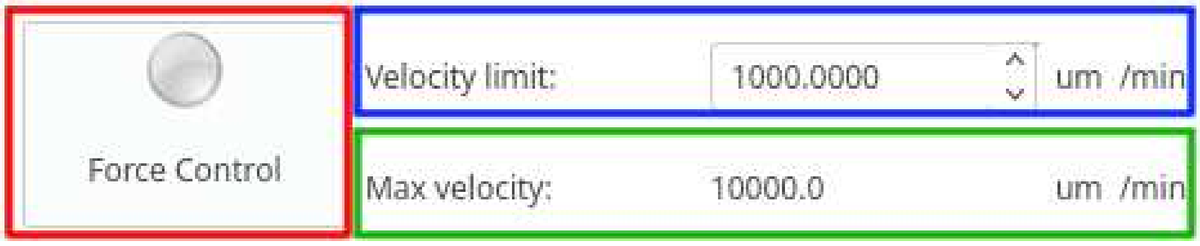


Figure 4.22: Force control plugin common controls

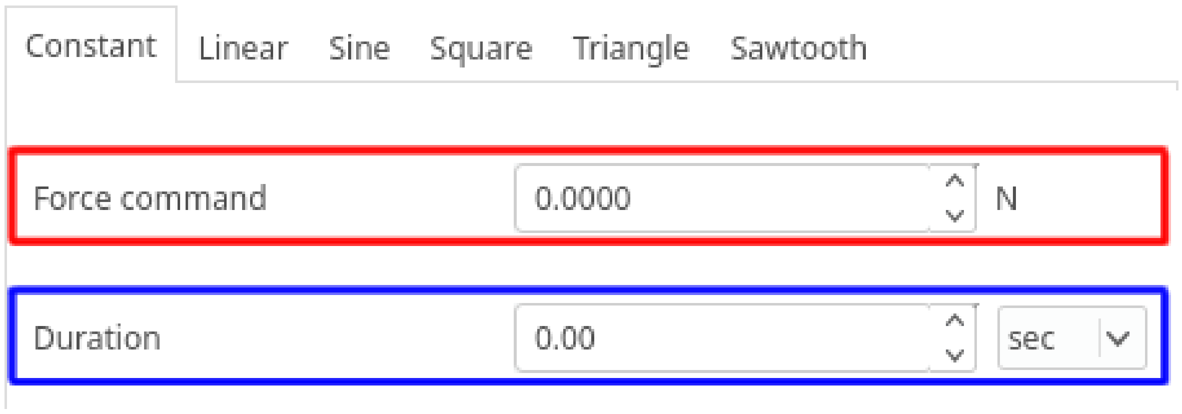
These controls consist of a button[10] to start or stop the force control measurement, see the red box in Figure 4.22. The button includes a diode to signal the force control status. If the force control is disabled, the diode remains grey. When the force control is activated, the diode becomes green. The force control features a PID controller which outputs a velocity command. The maximum motion velocity is an attribute of the axis which cannot be exceeded. The value of the maximum motion velocity is signalled in the common controls area, see the green box in Figure 4.22. The PID controller maximum output velocity might be adjusted by setting the custom velocity limit to the PID controller, see the blue box in Figure 4.22.

4.4.5.2 Constant force command tab

The force control plugin features a constant force command tab. The constant command is configured by putting the force value in a double spinbox[11], see the red box in Figure 4.23. The second parameter of the constant command is the command duration. When the force control is enabled a timer is trigerred. When the duration time expires, the force command is set back to zero and the procedure automatically stops.

[10]QToolButton is a special button that provides quick-access to specific commands or options. As opposed to a normal command button, a tool button usually does not show a text label, but shows an icon instead.

[11]QDoubleSpinBox allows the user to choose a value by clicking the up and down buttons or by pressing Up or Down on the keyboard to increase or decrease the value currently displayed. The user can also type in the value manually.



Force command 0.0000 N

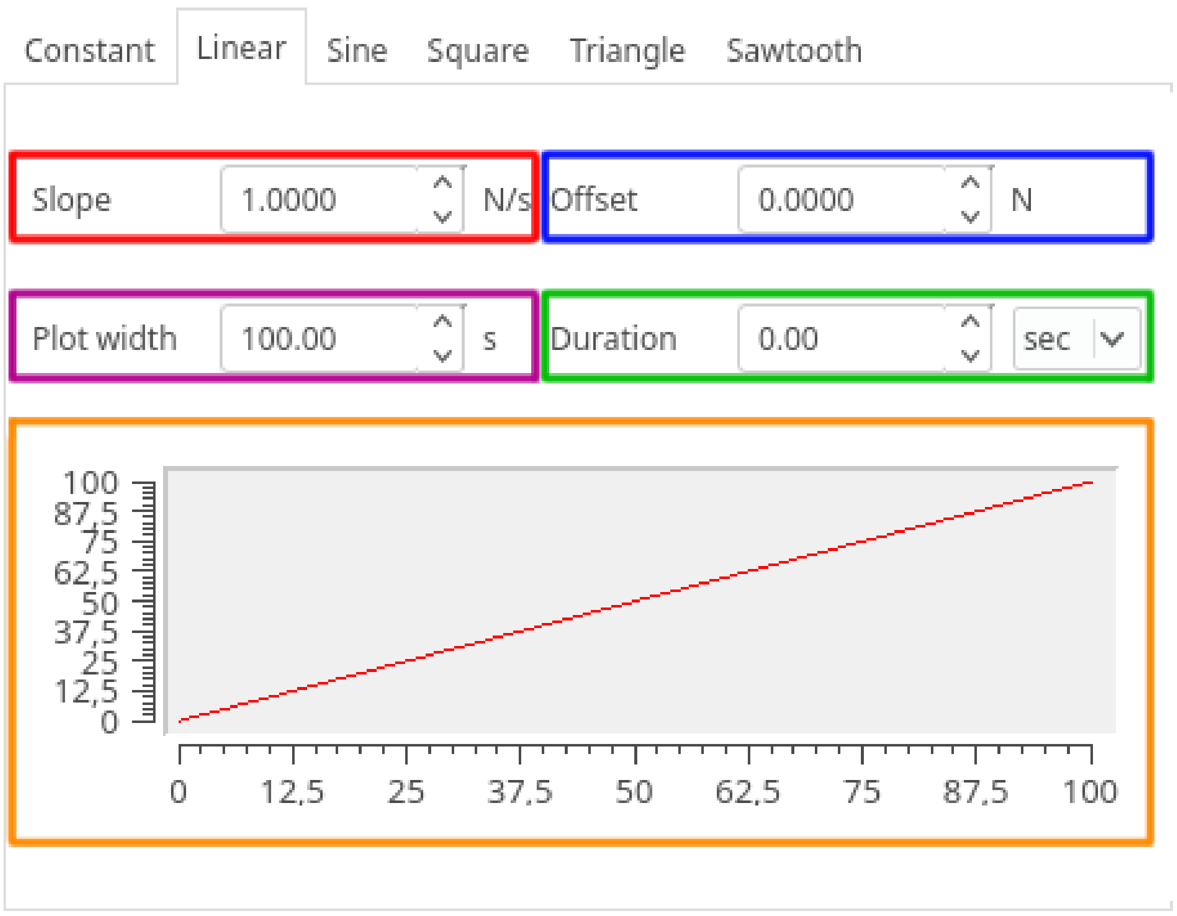
Duration 0.00 sec

Figure 4.23: Constant force command tab

4.4.5.3 Linear force command tab

Loading according to the linear function is another force command function $F(t) = A \cdot t + B$ implemented in the force command plugin. The linear function is defined by slope A in $[N/s]$ units, see the red box in Figure 4.24 and by offset B in $[N]$ units, see the blue box in Figure 4.24.

In order to visualise the force command as a function of time, a simple plot window is included, see the orange box in Figure 4.24. The plot window may be adjusted by the plot width parameter, see the purple box in Figure 4.24. As the force control is started, the force given by the force control function starts to grow. The user may define a duration parameter which automatically stops the force control when a certain ammount of time elapses, see the geen box in Figure 4.24.



Slope 1.0000 N/s
Offset 0.0000 N
Plot width 100.00 s
Duration 0.00 sec

Figure 4.24: Linear force command tab

4.4.5.4 Sine wave force command tab

The force control plugin includes various periodic force command functions. The most commonly used periodic force command is the sine wave function.

The sine wave force command function is defined as:

$$F(t) = A \cdot \sin(2 \cdot \pi \cdot f \cdot t + \varphi) + B, \tag{4.3}$$

where A is the sine wave amplitude (peak deviation), f is the ordinary frequency (the number of cycles per second), φ is the phase shift and B is the offset.

These parameters may be adjusted using spinboxes within the sine force command tab, see the red, blue, purple and green boxes in Figure 4.25. The force control function is visualised using a simple plot window, see the orange box in Figure 4.25. The number of periods shown within the plot window may be adjusted as well, see the grey box in Figure 4.25.

In order to start the force control, the number of cycles (periods) must be quantified. As soon as the number of cycles is reached, the force control automatically stops, see the brown box in Figure 4.25.

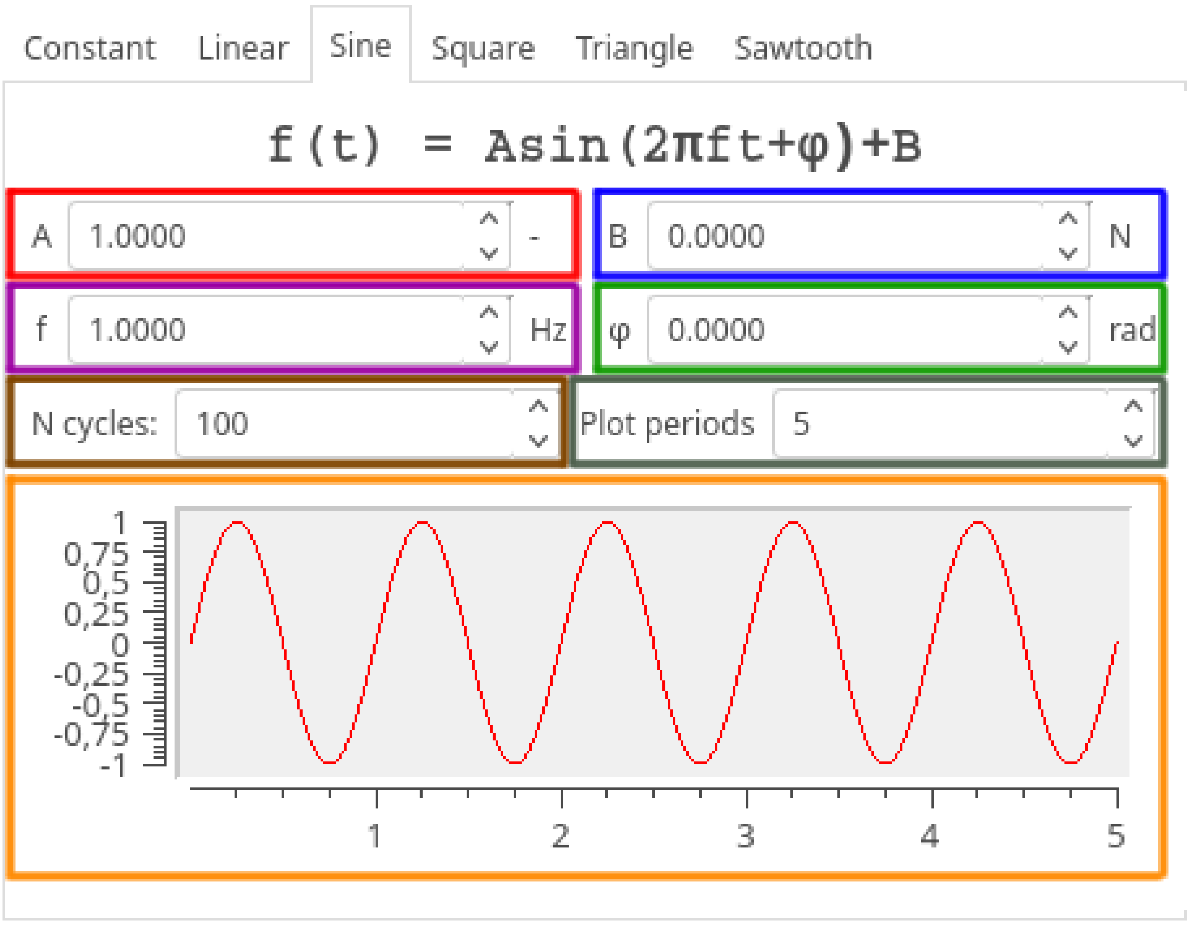


Figure 4.25: Sine wave force command tab

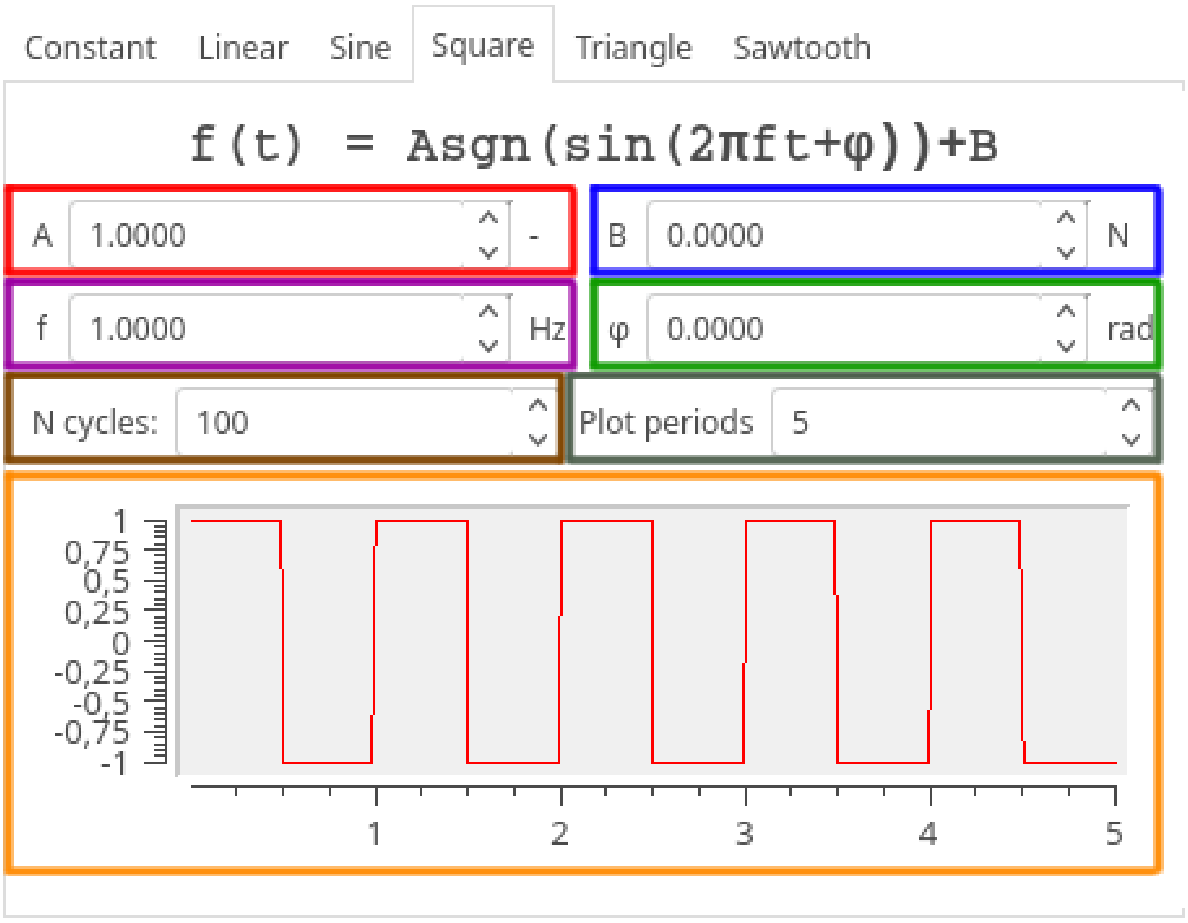
4.4.5.5 Square wave force command tab

Another periodic force command function is the square wave function. The square wave force command function is defined as:

$$F(t) = A \cdot \operatorname{sgn} \sin(2 \cdot \pi \cdot f \cdot t + \varphi) + B, \tag{4.4}$$

where A is the sine wave amplitude (peak deviation), f is the ordinary frequency (the number of cycles per second), φ is the phase shift and B is the offset.

The square wave parameters may be adjusted similarly to the previously introduced periodic function and the input fields of the user interface for the parameters have the same layout as well, see subchapter 4.4.5.4 and Figure 4.26.



A: 1.0000
B: 0.0000
f: 1.0000
φ: 0.0000
N cycles: 100
Plot periods: 5

Figure 4.26: Square wave force command tab

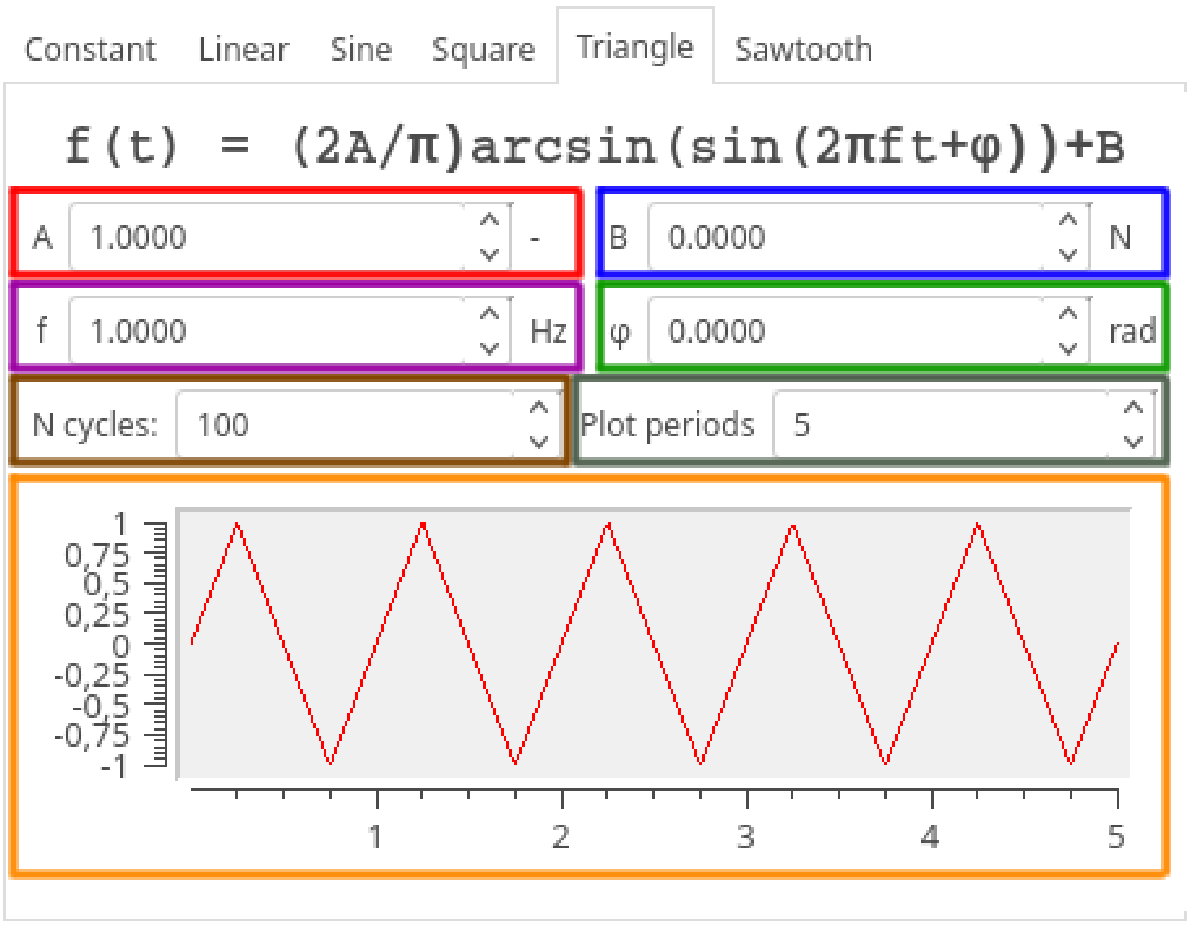
4.4.5.6 Triangle wave force command tab

The triangle wave force command function is defined as:

$$F(t) = \frac{2 \cdot A}{\pi} \cdot \arcsin\left[\sin\left(2 \cdot \pi \cdot f \cdot t + \varphi\right)\right] + B, \tag{4.5}$$

where A is the sine wave amplitude (peak deviation), f is the ordinary frequency (the number of cycles per second), φ is the phase shift and B is the offset.

The triangle wave parameters may be adjusted similarly to the previously introduced periodic function and the input fields of the user interface for the parameters have the same layout as well, see subchapter 4.4.5.4 and Figure 4.27.


Constant Linear Sine Square Triangle Sawtooth

f(t) = (2A/π)arcsin(sin(2πft+φ))+B

A 1.0000 -

B 0.0000 N

f 1.0000 Hz

φ 0.0000 rad

N cycles: 100

Plot periods 5

Figure 4.27: Triangle wave force command tab

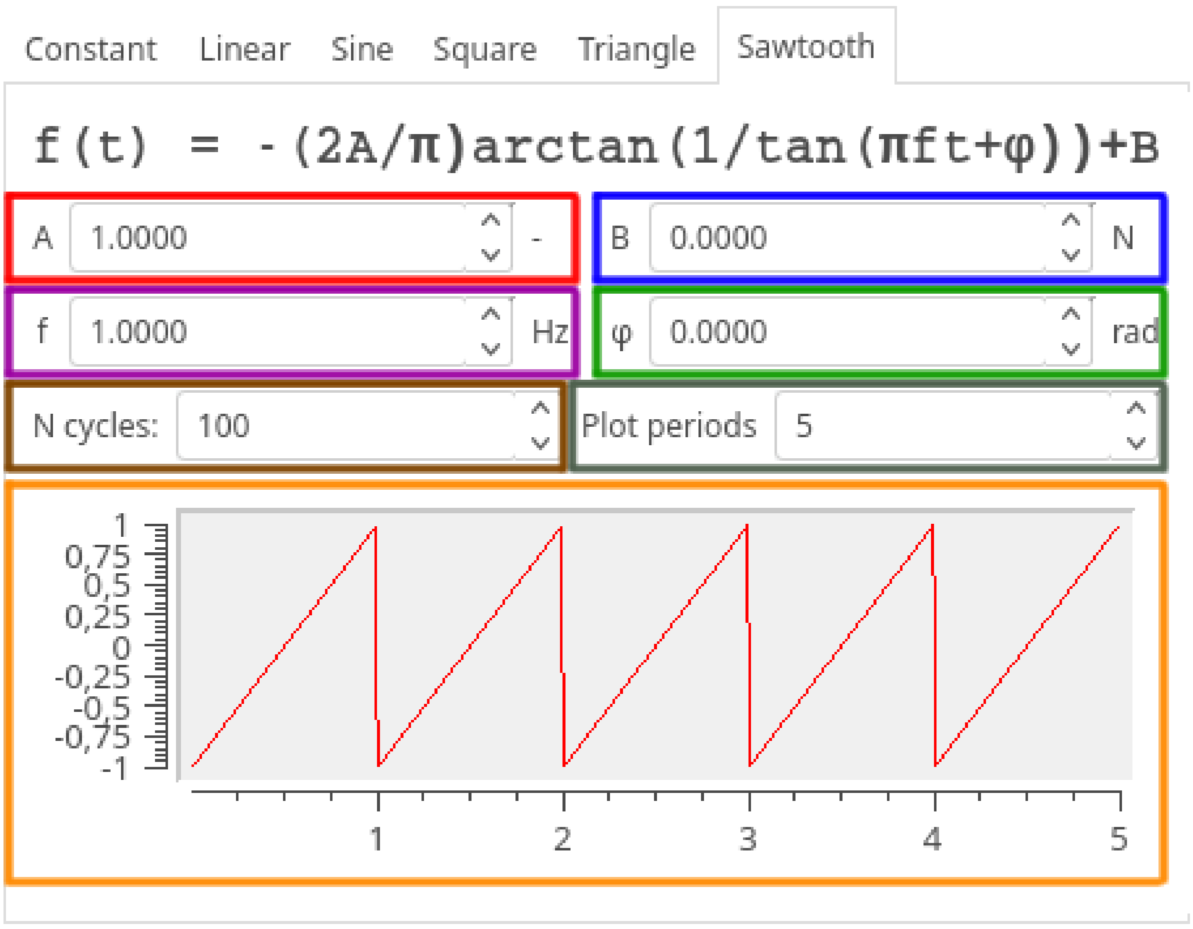
4.4.5.7 Sawtooth wave force command tab

The sawtooth wave force command function is defined as:

$$F(t) = -\frac{2 \cdot A}{\pi} \cdot \arctan\left[\frac{1}{\tan(\pi \cdot f \cdot t + \varphi)}\right] + B, \tag{4.6}$$

where A is the sine wave amplitude (peak deviation), f is the ordinary frequency (the number of cycles per second), φ is the phase shift and B is the offset.

Adjusting the sawtooth wave parameters is the same as in the cases of the other periodic functions, see subchapter 4.4.5.4 and Figure 4.28.



f(t) = -(2A/π)arctan(1/tan(πft+φ))+B

A 1.0000 -

B 0.0000 N

f 1.0000 Hz

φ 0.0000 rad

N cycles: 100

Plot periods 5

Figure 4.28: Sawtooth wave force command tab

4.4.6 Temperature control

The newly developed experimental devices also allow one to perform an experiment with the sample put in a controlled enviroment, because the mechanical properties of a material may vary dramatically based on enviromental conditions. To simulate these conditions, an observed sample can be submerged in a circulating liquid, such as a simulated body-fluid, artificial blood, water, degradation solutions - acids etc., with controlled temperature.

To control the temperature, the newly developed device is equipped with a heating plate. When the heating plate is under voltage, it heats the fluid to increase the temperature. As soon as the measured temperature $y(t)$ reaches the temperature setpoint $r(t)$, the heating plate is cut off from the voltage. Then the fluid starts to cool down to the ambient temperature. When the fluid temperature comes down to the setpoint minus hysteresis value $r(t) - h(t)$, the heating plate is put under voltage to increase the fluid temperature back again, see the temperature control flowchart in Figure 4.29.

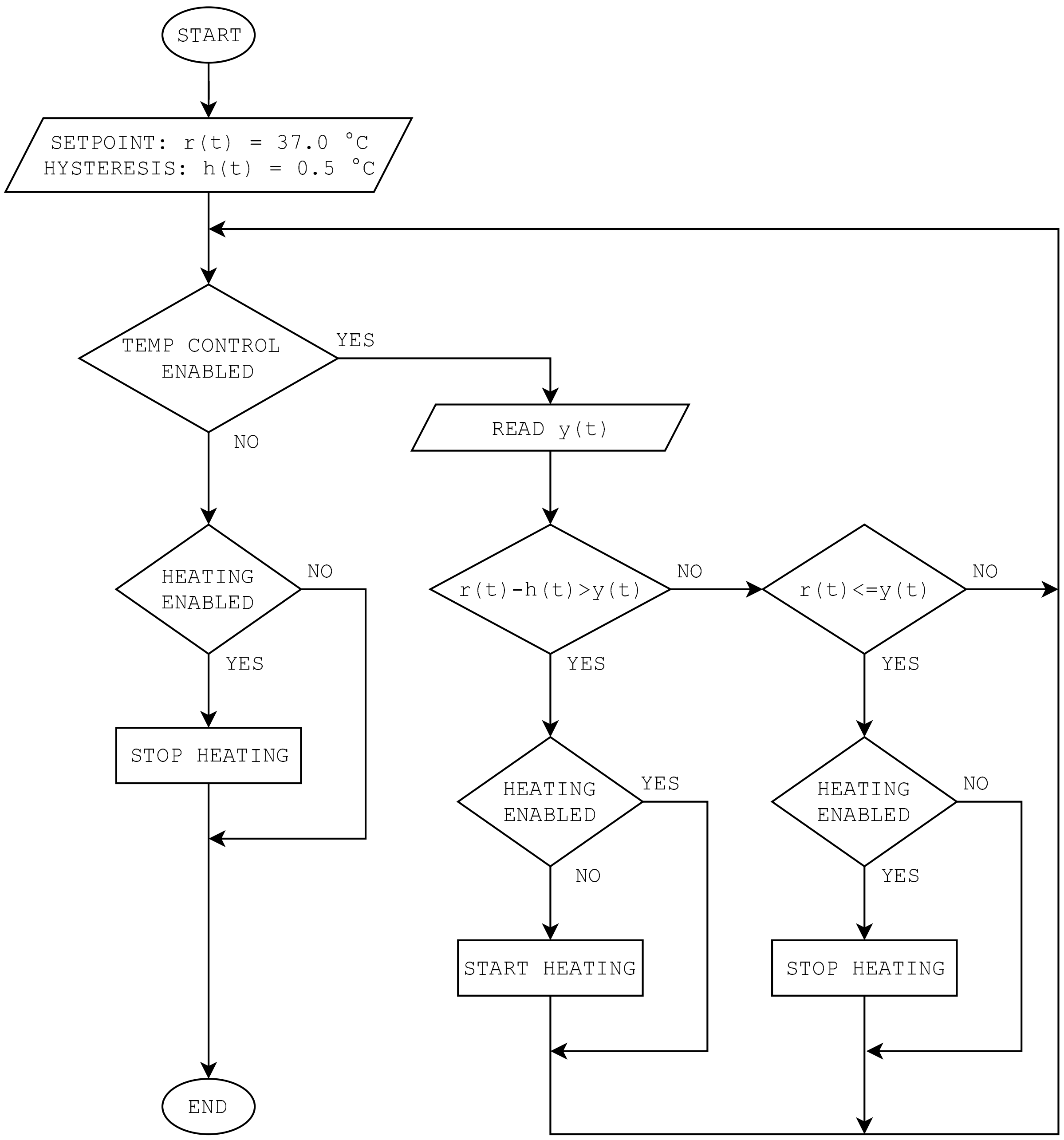


Figure 4.29: Temperature control flowchart

In order to keep the temperature within the whole loop (the so-called Bioreactor) uniform, the device is also equipped with a pump which circulates the fluid. These new features required developing a plugin for the user interface with a specific set of controls.

The plugin of the user interface features a button to switch on and switch off the pump, see the red box in Figure 4.30. The button includes a diode to signal whether the pump is switch on. Until the pump is switched, the diode remains grey. When the pump is switched on, the diode becomes green.

Another button included in the plugin is used to switch on and switch off the temperature control functionality. The button also includes a diode signalling whether the temperature control is active. When the temperature control is activated, the diode becomes green, otherwise it remains grey, see the blue box in Figure 4.30.

The plugin features another diode, signalling whether the heating plate is under voltage to heat the fluid. When the heating plate starts heating, the diode becomes orange, otherwise it remains grey, see the green box in Figure 4.30.

The temperature setpoint needed for the temperature control may be adjusted using a spinbox within the temperature control plugin, see the purple box in Figure 4.30.

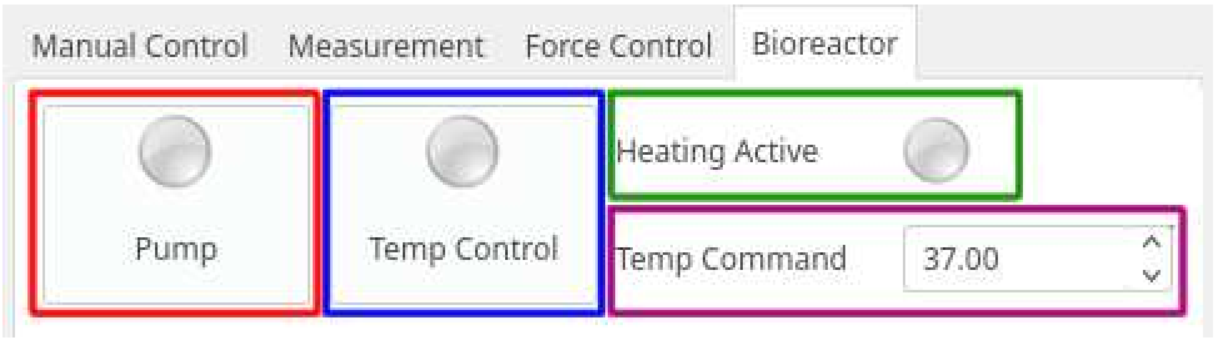


Figure 4.30: Temperature control plugin of the user interface

4.4.7 Rapo library

The control software comes with custom developed Python library which allows one to send commands to the control software core from an external Python script, see Figure 4.6. The library features the Unix domain socket connected with the API provider process.

The `rapo` library architecture is inspired by the LinuxCNC Python Interface. It includes various status channels and a command channel. The status channels provide status variables of the device, variables related with the measurement, such as the measured data, etc. The command channel allows one to send commands to the experimental device from the Python script in order to automate the experimental procedure.

Chapter 5

Case Studies

The newly developed control software found utilisation in controlling various experimental devices at the department and numerous experiments were successfully performed. Some of the experiments are presented in this chapter.

5.1 Compression of a spongious sample in simulated physiological conditions

In this study [23], an in-house designed table top loading device equipped with a bioreactor[1] is used for the in-situ compression of a human-bone sample in simulated physiological conditions, see Figures 5.1 and 5.2.

Fast on-the-fly 4D Computed Tomography (CT) together with a fast readout semiconductor detector are used as the tools for the advanced volumetric analysis of the deforming microstructure of the specimen, see Figure 5.3. Digital Volume Correlation (DVC) is employed as the method for the 3D strain analysis of the bone structure under loading.

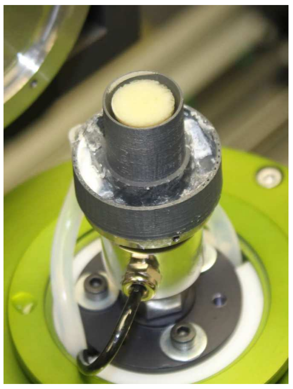
The loading device with the bioreactor was placed onto the rotary stage of the CT scanner. The geometry of the CT scanner was adjusted to a focus-object distance of 60 mm and a focus-detector distance of 300 mm. Thus, the nominal magnification was 5× with the

[1]The bioreactor is part of the loading device which can simulate the physiological conditions (temperature and flow) and it can be either used as an autonomous device or as an optional modular part of the loading device.

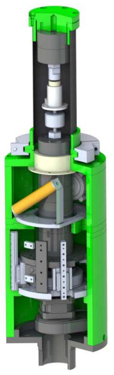
corresponding pixel size of 15 µm.

The displacement-driven compression of the specimen was conducted at a constant loading velocity of 0.25 µm/s. After the initial compression, three loading/unloading cycles were performed in the force range of 200 N − 400 N. At the end of the experiment, the structure was compressed to the nominal engineering strain of 2 %. The overall duration of the experiment was 3200 s.

(a) Tested human bone specimen



(b) Specimen submerged in the simulated body fluid



(c) Cutaway view of the in-situ loading device with bioreactor

Figure 5.1: Human bone specimen and loading device

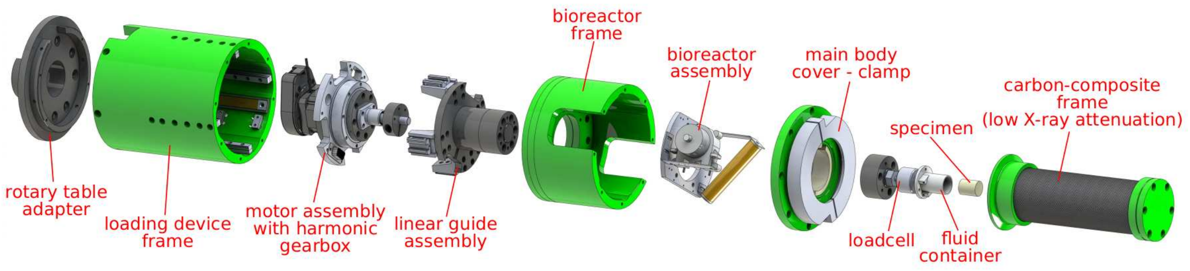


Figure 5.2: Loading device exploded view in detail

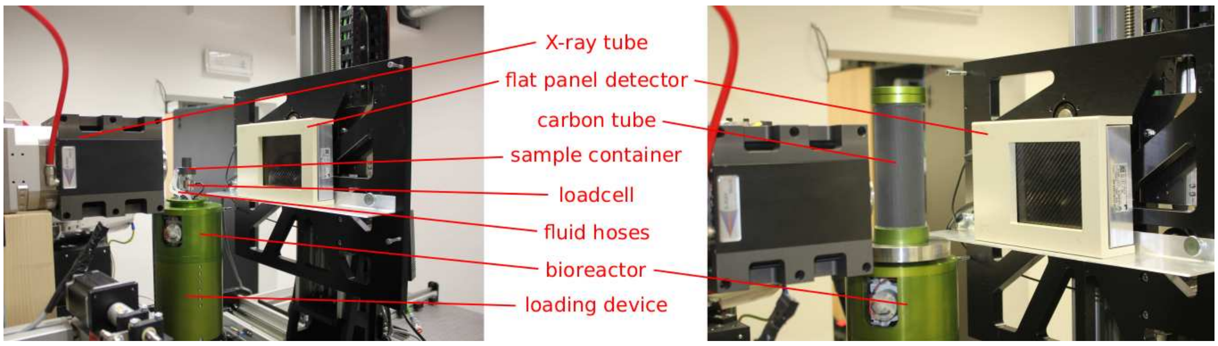
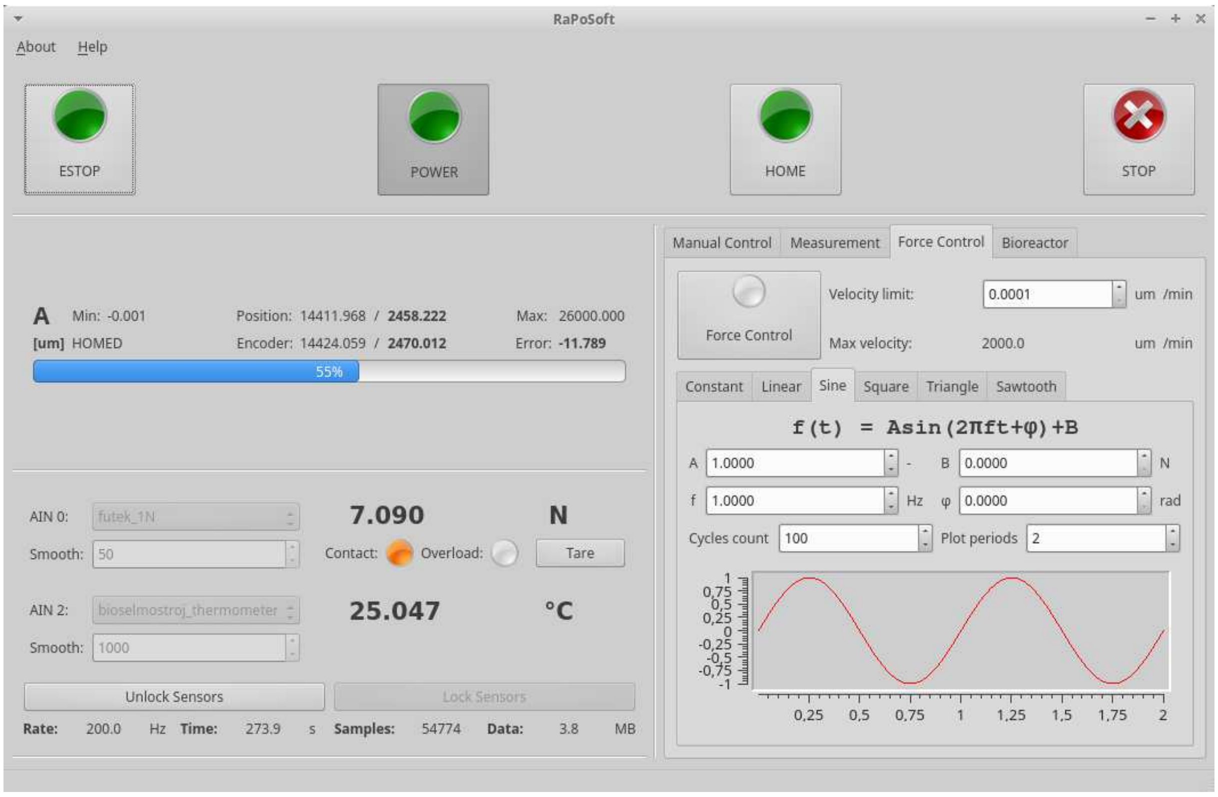


Figure 5.3: Loading device exploded view in detail

The newly developed features of the control software used to perform the experiment involve the simultaneous force and temperature measuring functionality described in detail in subchapter 4.4.2 and the temperature control with the controlled flow of the simulated body fluid, see subchapter 4.4.6.

During the experiment all of the features including the real-time plotting worked as it was supposed to. The data samples obtained by the control software were successfully logged and exported to the plain-text file. The control software proved that it is capable of controlling the experimental devices reliably, without any issues and the results showed that the measured data is correct. Overall view of the control software during the experiment is shown in Figure 5.4.



A Min: -0.001 Position: 14411.968 / **2458.222** Max: 26000.000
[um] HOMED Encoder: 14424.059 / **2470.012** Error: **-11.789**
55%

AIN 0: futek_1N 7.090 N
Smooth: 50 Contact: Overload:

AIN 2: bioselmostroj_thermometer 25.047 °C
Smooth: 1000

Rate: 200.0 Hz **Time:** 273.9 s **Samples:** 54774 **Data:** 3.8 MB

Force Control Velocity limit: 0.0001 um /min
Max velocity: 2000.0 um /min

f(t) = Asin(2πft+φ)+B
A 1.0000 - B 0.0000 N
f 1.0000 Hz φ 0.0000 rad
Cycles count 100 Plot periods 2

Figure 5.4: Overall view of the control software

5.2 Fracture analysis of sandstone

In this study, an in-house designed experimental device is used for the 4-point bending of weathered sandstone samples. The experimental device features two loading units, each of which is motorised by a stepper motor and equipped with a load cell. The frame of the device is made of high-strength aluminium alloy and carbon composite, see Figure 5.5.

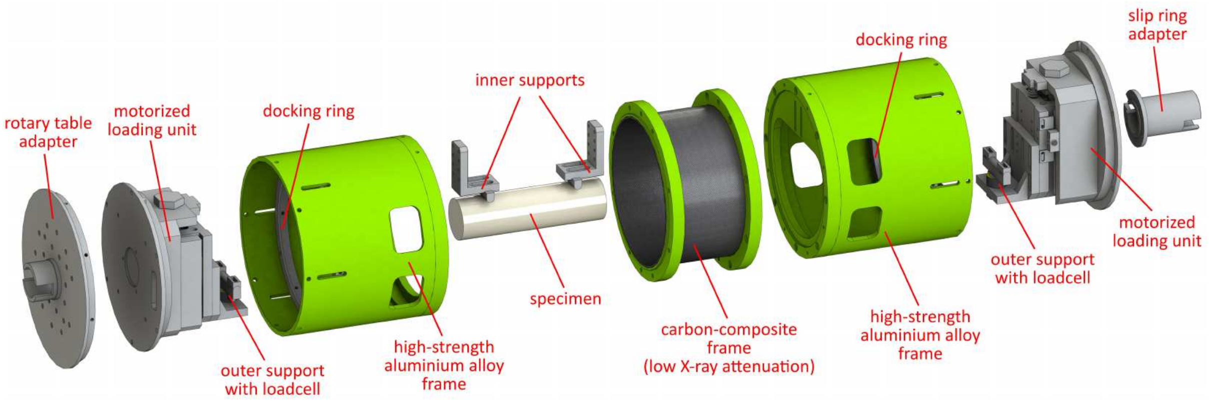

rotary table adapter
motorized loading unit
docking ring
inner supports
docking ring
slip ring adapter
outer support with loadcell
high-strength aluminium alloy frame
specimen
carbon-composite frame (low X-ray attenuation)
high-strength aluminium alloy frame
outer support with loadcell
motorized loading unit

Figure 5.5: Experimental device for the 4-point bending mechanical test

The device is intended to be used for the 4-point bending mechanical test paired with a CT scanner to perform on-the-fly 4D CT during the bending, see Figure 5.6. The CT scans may be used for the 3D strain analysis using the DVC.

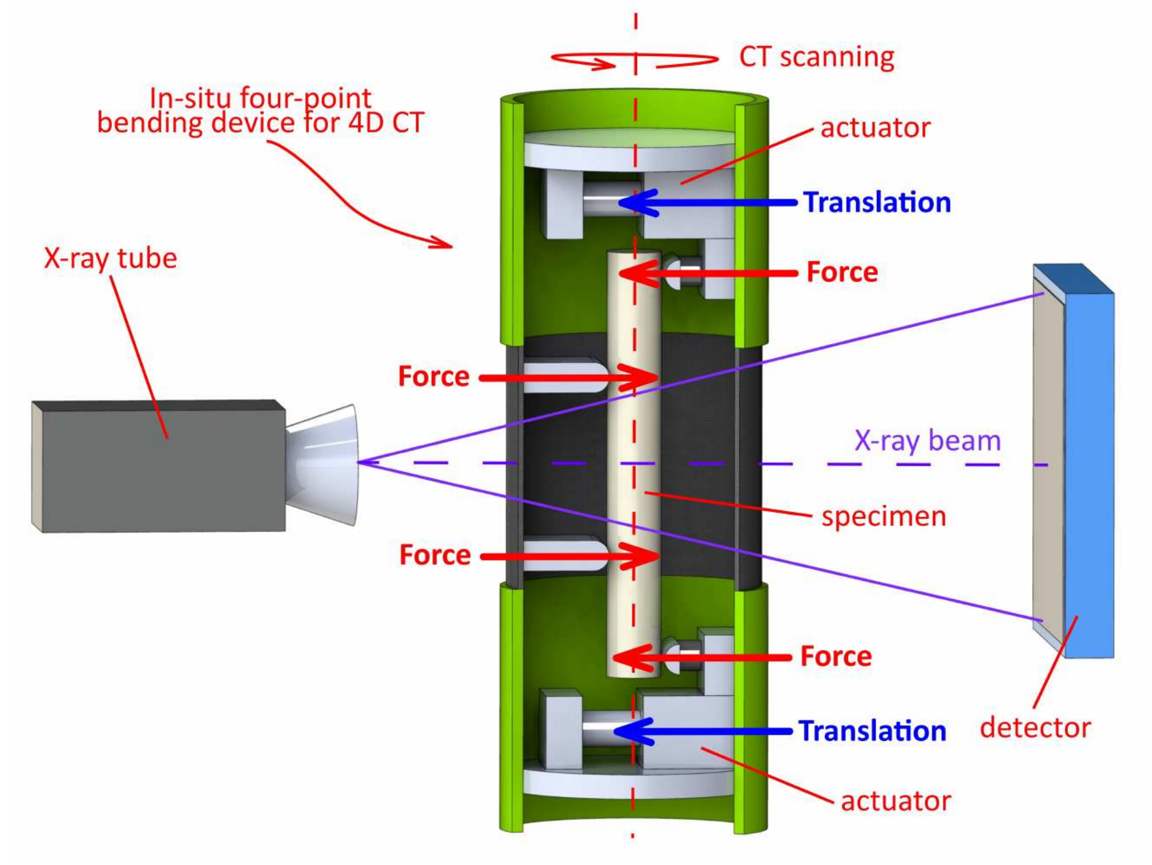


Figure 5.6: Principle of the 4-point bending and on-the-fly CT

The sandstone samples were very fragile, therefore a precise positioning was required, especially when approaching a contact force. The contact force was set to 5 N as the constant force control function. Maximum velocity (maximum output of the PID controller) was set to 10 µm/s. As the loading units reached the contact force, a displacement-driven experiment started. The loading velocity was 1 µm/s and the experiment stopped when the sandstone sample was broken in half.

Numerous series of sandstone samples were observed. Some of the samples were weathered by water, some of them were weathered by ice, some of the samples were observed intact. Results of the experiments are shown in Figures 5.7.

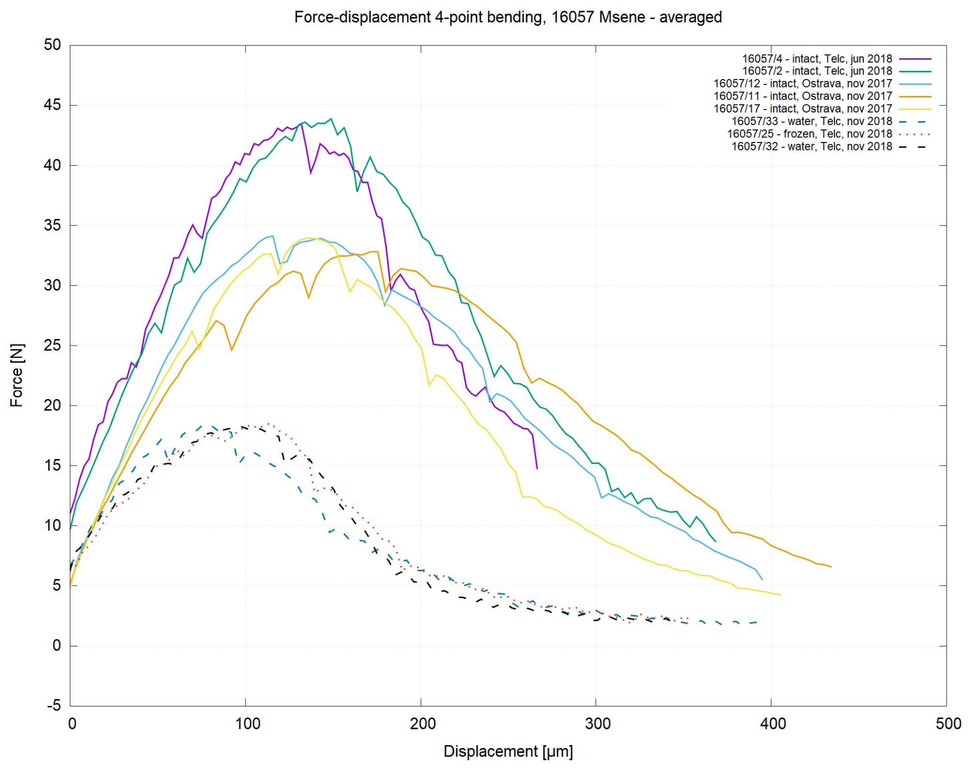


Figure 5.7: Results of the sandstone analysis

Chapter 6

Work in Progress

Currently, the control software is still being developed. New features are being implemented, such as a remote control.

6.1 Remote control

The remote control capabilities of the control software allow the user to control the experimental devices over the network in a client-server fashion, see Figure 6.1.

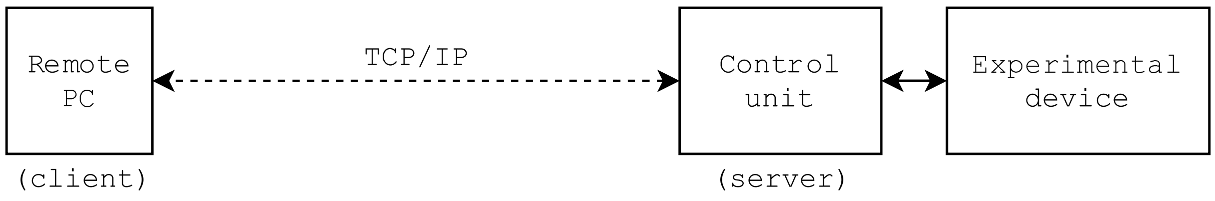


Figure 6.1: Scheme of the control software remote control

The control unit physically controlling the experimental device acts like a server. The control software core, including all the real-time demands and security procedures, such as limit switch supervision or load cell overload inspection, run within the control unit. The remote PC is used to send commands or requests for variables[1] to the control unit using the TCP stream socket.

[1]Variables, such as status variables, error status, measured data, etc.

6.1.1 TCP socket

A socket programming interface provides the routines required for interprocess communication between the applications, either on the local system (Unix domain socket, see subchapter 4.3.1) or spread in the TCP/IP based network environment. The TCP/IP connection is defined as an internet address (IPv4 or IPv6) and a port numerical value.

TCP sockets provide a reliable, nearly error-free data pipe between two endpoints. Both of the devices can send and receive streams of bytes so a serialising module must be used when sending the data structure, such as a Python list or dictionary, see subchapter 4.3.2.

One device, known as the client, creates a socket, connects to the server, and then begins sending and receiving data. On the other side, the server creates a socket and listens for the incoming connection from the client. Once a connection is initiated, the server accepts the connection, and then starts to send and receive data to and from the incoming client.

The control software includes a process of the so-called Server Manager. The Server manager is connected with other processes within the control software core which makes the control software architecture more complex than described in Figure 4.3, see Figure 6.2 and Appendix B.

6.1.2 Server Manager

The Server Manager process is part of the control software running within the control unit. It provides remote access over the network to the control software core using the TCP socket, see Figure 6.3 and Appendix B. The TCP socket of the Server Manager can be accessed by the remote PC through the so-called Client Manager process.

6.1.3 Client Manager

The Client Manager process runs within a client application on the remote PC and provides all functionality needed for communication with the Server Manager process running inside the control unit. The client application includes the Client Manager process which is connected with the remote user interface and with the remote API provider process.

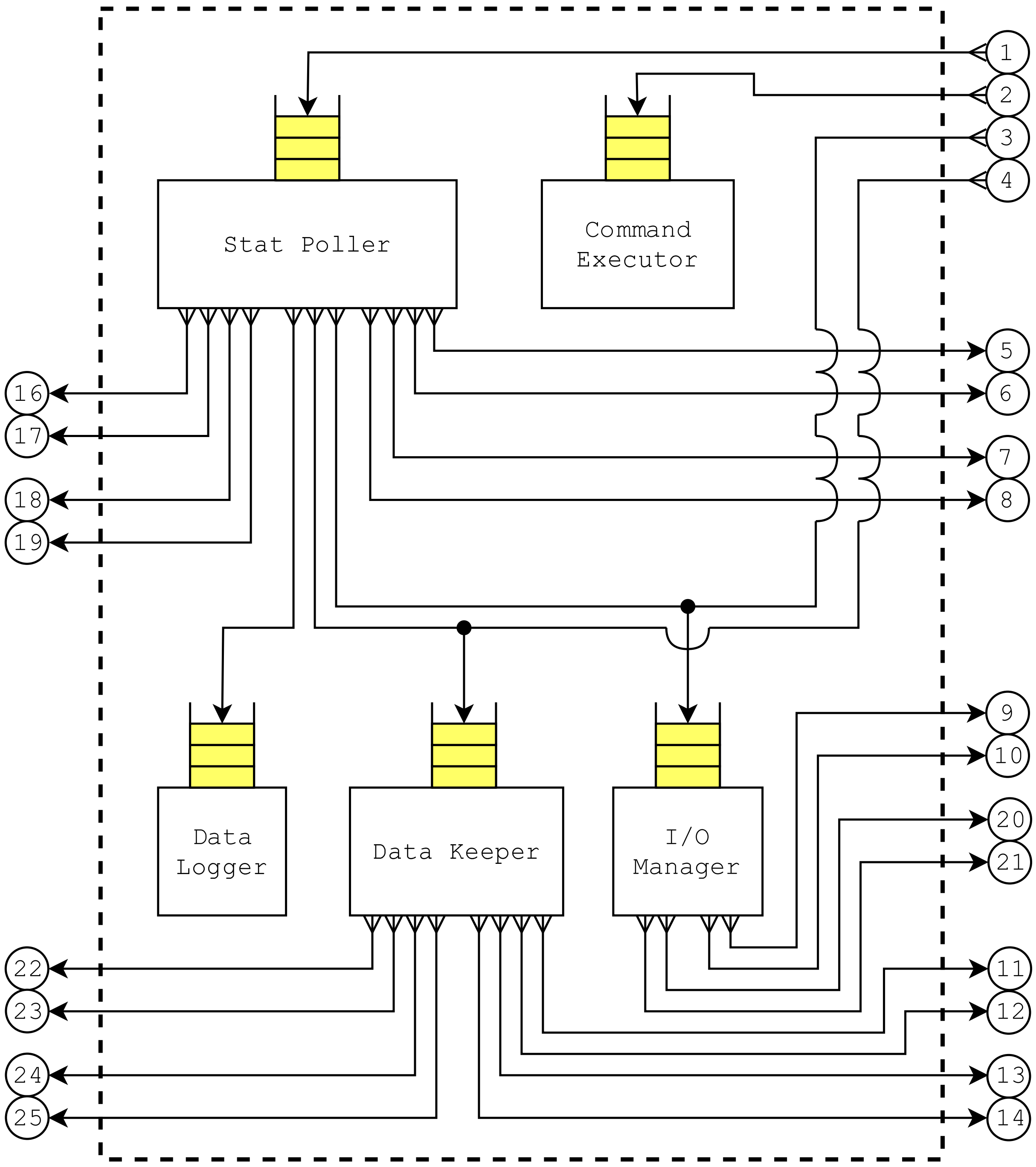


Figure 6.2: Control software core with connection to the Server Manager included

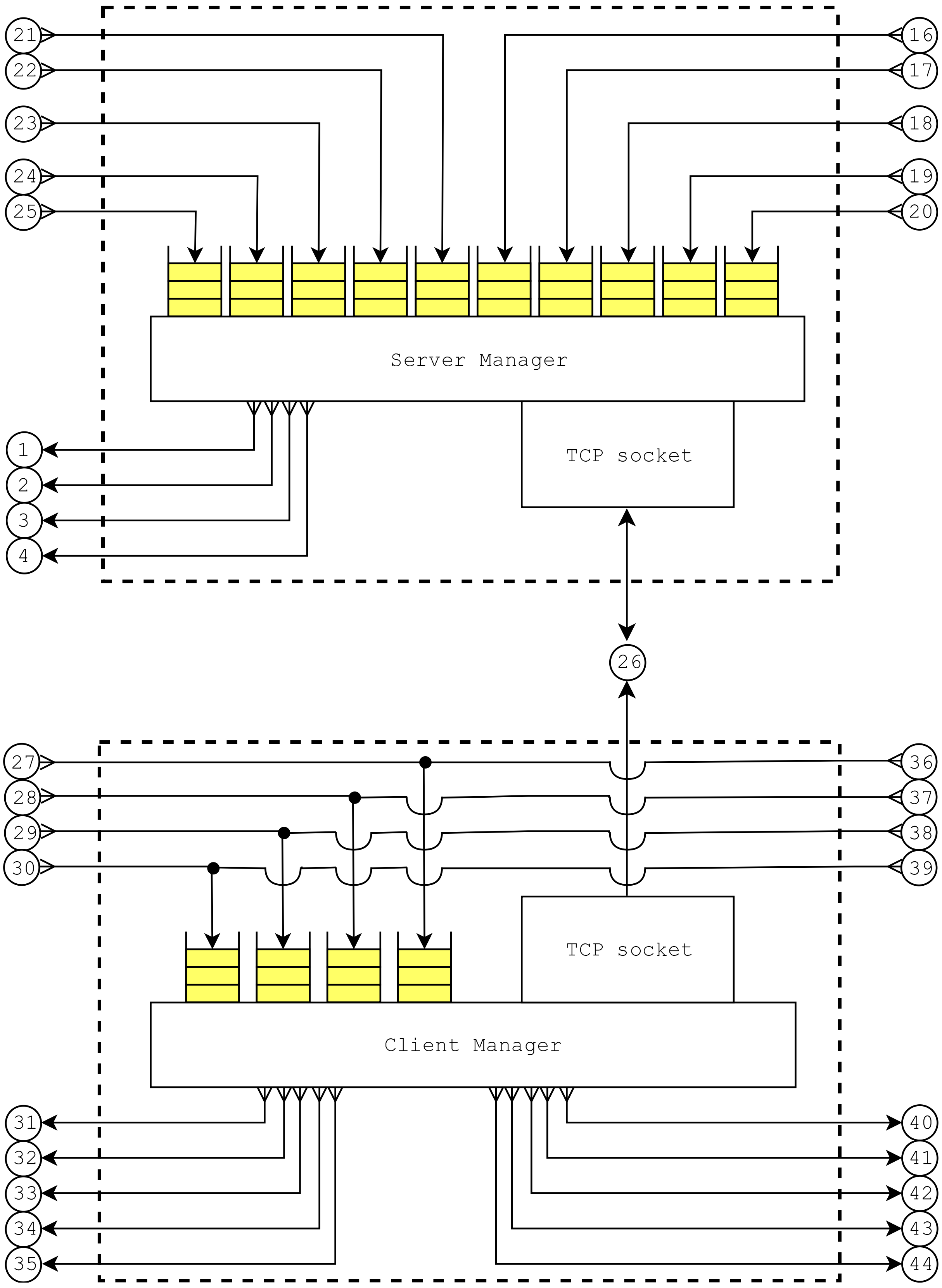


Figure 6.3: Server Manager and Client Manager inter-process communication scheme

6.1.4 Remote user interface

The remote user interface is connected with the Client Manager using multiprocessing queues. Once a command is sent from the remote user interface, it reaches the Client Manager which adds an identification stamp to the command and sends it through the TCP socket to the Server Manager running within the control unit. The Server Manager receives the command and reads the identification stamp. Based on the identification stamp, the Server Manager puts the command to an appropriate queue leading into a process suitable for executing the command.

Due to this identification stamp mechanism, the developed user interface may be connected either with the control software core directly within the control unit, or paired with the Client Manager within the client application. All the specific functionality required for sending commands remotely is included in Client Manager and Server Manager processes, which create a transfer layer for the remote commands. Notice, that the remote user interface architecture remains the same, it only connects to queues of the Client Manager, see Figure 6.4 and Appendix B.

6.1.5 Remote script execution

The remote application also includes the API provider process connected with the Client Manager. The API provider allows one to send commands from the remote Python script using the `rapo` library. The API provider and the `rapo` library are based on the same architecture shown in Figure 4.6 and described in subchapter 4.3 which makes them suitable for remote use as well. All the functionality needed for transfering commands from the remote API provider to the control software core is included in Client Manager and Server Manager processes, see Figure 6.5 and Appendix B.

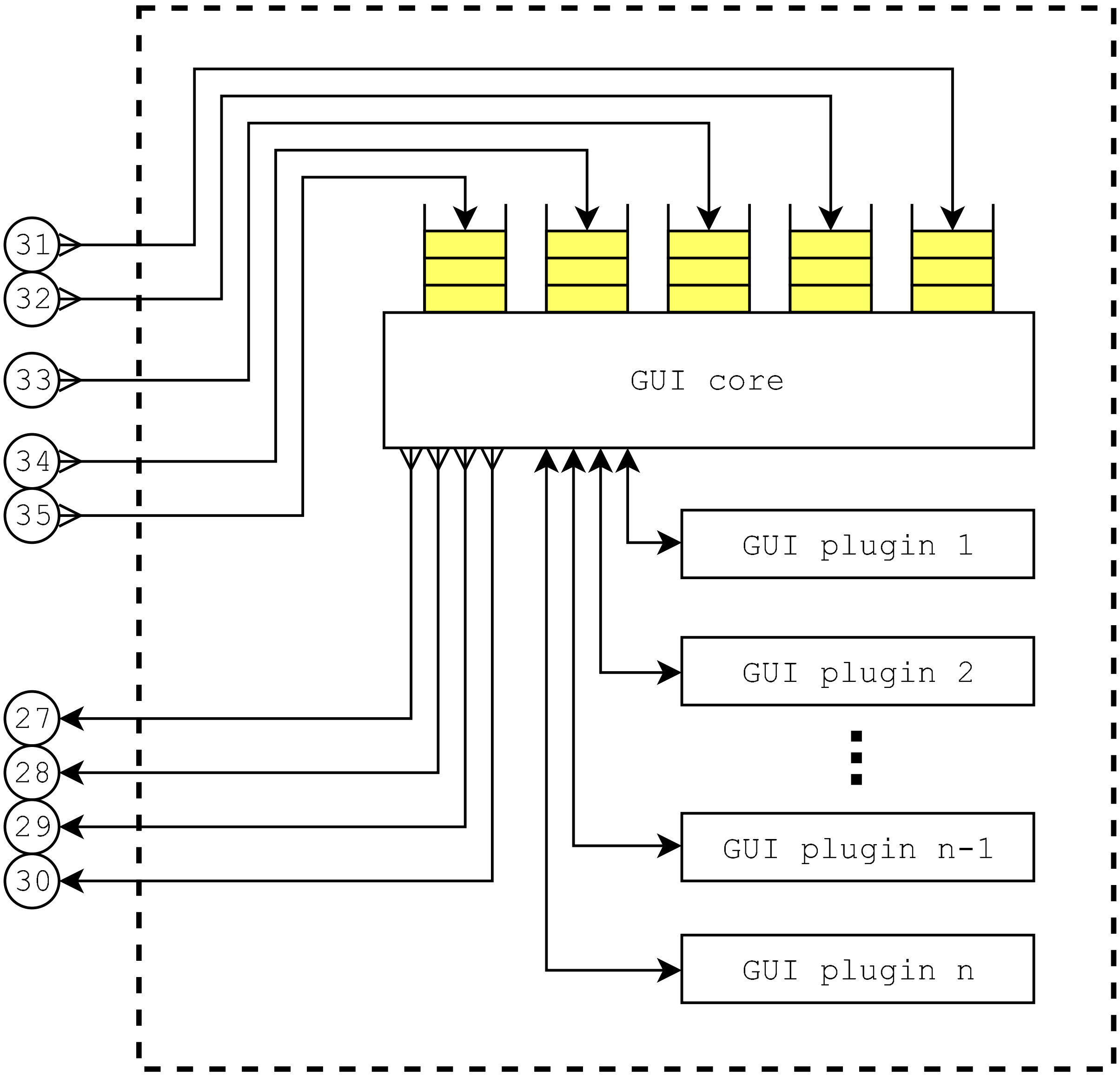
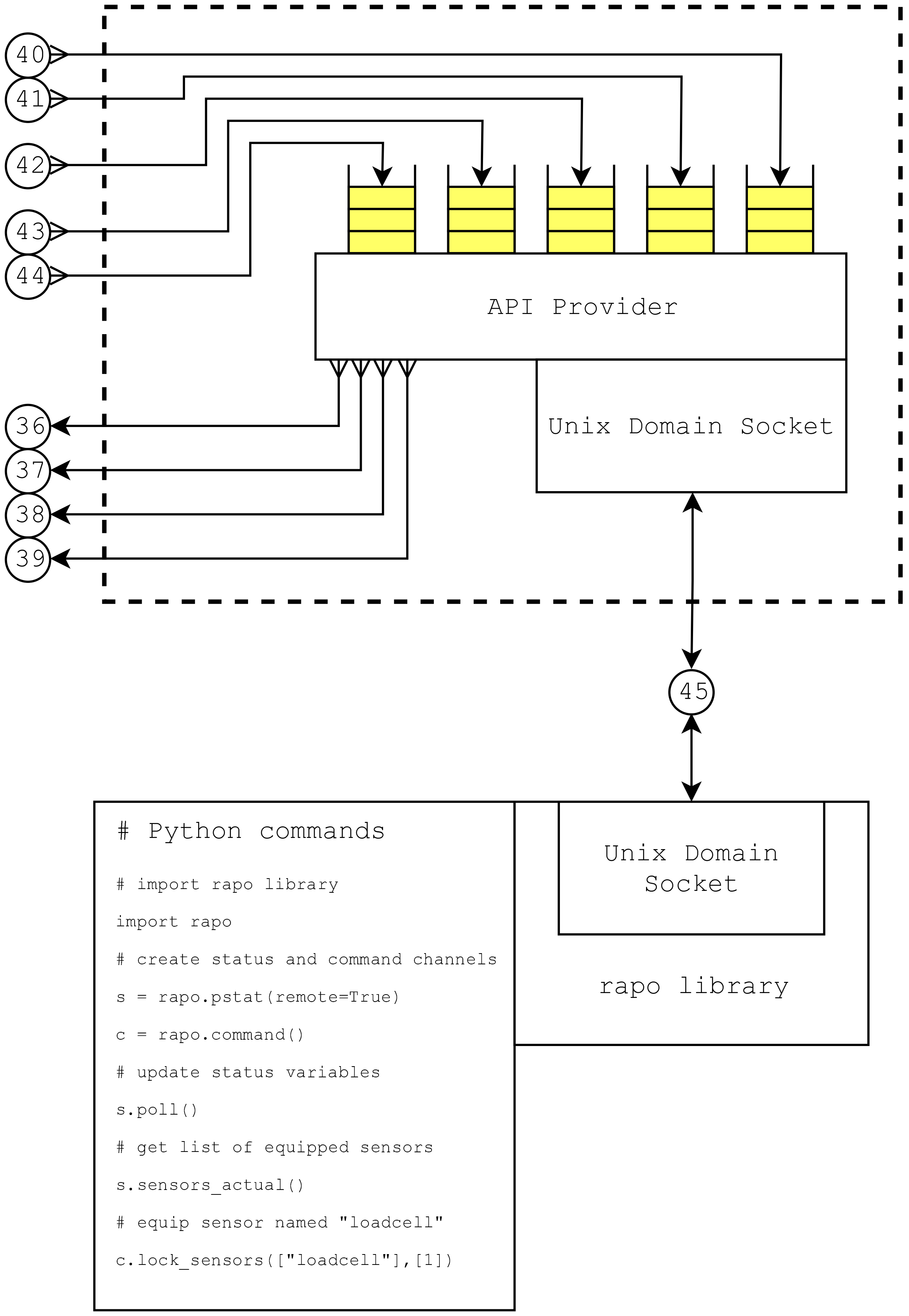


Figure 6.4: Remote user interface architecture



```
# Python commands

# import rapo library

import rapo

# create status and command channels

s = rapo.pstat(remote=True)

c = rapo.command()

# update status variables

s.poll()

# get list of equipped sensors

s.sensors_actual()

# equip sensor named "loadcell"

c.lock_sensors(["loadcell"],[1])
```

Figure 6.5: Remote API provider and rapo library connection scheme

Chapter 7

Conclusion

Within the proposed thesis, a modular multi-process control software for experimental devices operated in the Department of Mechanics and Materials in the Faculty of Transportation Sciences at the Czech Technical University and at the Institute of Theoretical and Applied Mechanics of the Czech Academy of Sciences was developed.

The newly developed control software replaced the first-generation control software published previously as part of my Bachelor's thesis. The new-generation control software gives much more performance than the first-generation control software as the new-generation is based on a multi-process architecture with a robust multi-process core and the control software functionality was also enahanced with new features to satisfy needs of the recently developed experimental devices, such as the support for various types of sensors (load cells, thermometers, etc.), encoders support, temperature control functionality or the support for performing force-driven experiments.

Furthermore, the control software features a scripting functionality which enables the execution of custom external Python scripts. These scripts can operate with the experimental device through the developed `rapo` library. The library allows one to monitor the status variables of the experimental device, send various commands to the device, etc. Therefore, the library may be used for creating various types of automated procedures, such as advanced experimental procedures, etc.

Currently, the control software has been utilised with a great success in controlling experimental devices in the department. Thanks to the control software, numerous experiments have been performed and many studies have been published. The control software has proved its long-term stability and reliability as several experimental procedures last for many hours and certain procedures took even more than one day of uninterrupted measurement.

The control software also participated at the LinuxCNC community meeting in Stuttgart, Germany in July 2018. Alhough the control software was still being developed during that time, the majority of the features had been implemented already. The software drew the attention of the community as it combines conventional precise CNC positioning capabilities with high-performance and high-precision data acquisition.

Still, the control software is being developed and new features are being added. The most recent feature in the development is the remote control which enables one to control the experimental devices remotely over a local network or even over the internet. Another feature that has come into consideration for future development is the real-time image processing functionality of the imaging data obtained during an experimental procedure, such as digital planar image correlation or digital volumetric image correlation.

Bibliography

[1] Eml 2023 – motor control lecture 3 – feedback sensor optical encoder. https://slideplayer.com/slide/6003777/. Accessed: 2019-03-03.

[2] T. LinuxCNC Team. *LinuxCNC Getting Started Guide*. Samurai Media Limited, 2016.

[3] Hal Introduction. http://linuxcnc.org/docs/html/hal/intro.html. Accessed: 2019-05-11.

[4] PID Controller. http://linuxcnc.org/docs/html/motion/pid-theory.html. Accessed: 2019-05-11.

[5] Mesa HostMot2 Driver. http://www.linuxcnc.org/docs/html/drivers/hostmot2.html. Accessed: 2019-05-11.

[6] AXIS GUI. http://linuxcnc.org/docs/html/gui/axis.html. Accessed: 2019-05-11.

[7] Touchy GUI. http://linuxcnc.org/docs/html/gui/touchy.html. Accessed: 2019-05-11.

[8] Python Virtual Control Panel. http://www.linuxcnc.org/docs/2.4/html/hal_pyvcp.html. Accessed: 2019-05-11.

[9] Glade Virtual Control Panel. http://linuxcnc.org/docs/html/gui/gladevcp.html. Accessed: 2019-05-11.

[10] LinuxCNC Python Interface Documantation. http://linuxcnc.org/docs/2.6/html/common/python-interface.html. Accessed: 2019-05-11.

[11] Creating Userspace Python Components. http://linuxcnc.org/docs/html/hal/halmodule.html. Accessed: 2019-05-11.

[12] What is Python? Executive Summary. https://www.python.org/doc/essays/blurb/. Accessed: 2019-05-11.

[13] G. Zaccone. *Python Parallel Programming Cookbook*. Packt Publishing Ltd., 35 Livery Street, Birmingham B3 2PB, UK, 2015.

[14] PyQt Documentation. https://wiki.python.org/moin/PyQt. Accessed: 2019-05-11.

[15] Qt Documentation. http://doc.qt.io/qt-5.9/index.html. Accessed: 2019-05-11.

[16] PythonQwt Manual. https://pythonhosted.org/PythonQwt/. Accessed: 2019-05-11.

[17] Qwt - Qt Widgets for Technical Applications. http://qwt.sourceforge.net/. Accessed: 2019-05-11.

[18] J. D. Hunter. Matplotlib: A 2D Graphics Environment. *Computing In Science & Engineering* **9**(3):90–95, 2007. DOI:10.1109/MCSE.2007.55.

[19] B. Schneider. PyGnuplot: Python wrapper for Gnuplot. https://pypi.org/project/PyGnuplot/. Accessed: 2019-05-11.

[20] T. Williams, C. Kelley, many others. Gnuplot 5.2: an interactive plotting program. http://gnuplot.sourceforge.net/, 2019.

[21] V. Rada. Software pro řízení strojů a experimentálních zařízení. https://dspace.cvut.cz/handle/10467/73180?locale-attribute=en, Bachelor's Thesis, Czech Technical University in Prague, Faculty of Transportation Sciences, Department of Mechanics and Materials, 2017.

[22] P. Koudelka, T. Fíla, D. Kytýř, et al. Novel device for 4-point flexural testing of quasi-brittle materials during 4d computed tomography. In *Structural Integrity*, pp. 27–32. Springer International Publishing, 2018. DOI:10.1007/978-3-319-91989-8_5.

[23] T. Fíla, J. Šleichrt, D. Kytýř, et al. Deformation analysis of the spongious sample in simulated physiological conditions based on in-situ compression, 4d computed

tomography and fast readout detector. *Journal of Instrumentation* **13**(11):C11021–C11021, 2018. DOI:10.1088/1748-0221/13/11/c11021.

Appendix A

Prime fatorisation script

Listing A.1: Prime fatorisation script

```
"""
how many numbers? 1000000
single thread: 22.4188029766 seconds
2 threads: 31.318999052 seconds
4 threads: 52.8587779999 seconds
6 threads: 63.3178188801 seconds
8 threads: 71.5121889114 seconds
10 threads: 75.6294119358 seconds
12 threads: 76.9113698006 seconds
16 threads: 81.545334816 seconds
2 processes: 15.542855978 seconds
4 processes: 8.67697715759 seconds
6 processes: 5.94609308243 seconds
8 processes: 4.50127601624 seconds
10 processes: 4.2152929306 seconds
12 processes: 3.7843940258 seconds
16 processes: 3.41232895851 seconds
"""

import time, math
from multiprocessing import Process, Queue
import threading
```

```
def factorize(n):
    """
    A factorization method. Take integer 'n', return list of factors.
    """
    if n < 2:
        return []
    factors = []
    p = 2

    while True:
        if n == 1:
            return factors

        r = n % p
        if r == 0:
            factors.append(p)
            n = n / p
        elif p * p >= n:
            factors.append(n)
            return factors
        elif p > 2:
            # Advance in steps of 2 over odd numbers
            p += 2
        else:
            # If p == 2, get to 3
            p += 1


def plain_factorizer(nums):
    """
    Single threaded method factorizing list of numbers
    :param nums: list of numbers to factor
    :return: dict, key is a factorized integer, value is list of factors
    """
    return {n: factorize(n) for n in nums}


def thread_worker(nums, outdict):
    """
    The worker function, invoked in a thread.
    :param nums: list of numbers to factor
    :param outdict: results are placed in outdict
    """
    for n in nums:
        outdict[n] = factorize(n)


def threaded_factorizer(nums, nthreads):
    """
    Method factorizing list of numbers using n threads.
    :param nums: list of numbers to factor
    :param nthreads: number of threads to utilize
    :return: dict, key is a factorized integer, value is list of factors
    """
    # Each thread will get 'chunksize' nums and its own output dict
    chunksize = int(math.ceil(len(nums) / float(nthreads)))
    threads = []
    outs = [{} for _ in range(nthreads)]

    for i in range(nthreads):
        # Create each thread, passing it its chunk of numbers to factor and output dict.
        t = threading.Thread(target=thread_worker, args=(nums[chunksize * i:chunksize * (i + 1)],
            ↪ outs[i]))
        threads.append(t)
        t.start()

    # Wait for all threads to finish
    for t in threads:
        t.join()

    # Merge all partial output dicts into a single dict and return it
    return {k: v for out_d in outs for k, v in out_d.iteritems()}


def process_worker(nums, out_q):
    """
    The worker function, invoked in a process.
    :param nums: :param nums: list of numbers to factor
    :param out_q: results are pushed to the queue
    """
    outdict = {}
    for n in nums:
        outdict[n] = factorize(n)
    out_q.put(outdict)



def multiprocess_factorizer(nums, nprocs):
    """
    Method factorizing list of numbers using n processes.
    :param nums: list of numbers to factor
    :param nprocs: number of processes to utilize
    :return: dict, key is a factorized integer, value is list of factors
    """
    # Each process will get 'chunksize' nums and a queue to put his out dict into
    out_q = Queue()
    chunksize = int(math.ceil(len(nums) / float(nprocs)))
    procs = []

    for i in range(nprocs):
        p = Process(
                target=process_worker,
                args=(nums[chunksize * i:chunksize * (i + 1)],
                      out_q))
        procs.append(p)
        p.start()

    # Collect all results into a single result dict. We know how many dicts with results to expect.
    resultdict = {}
    for i in range(nprocs):
        resultdict.update(out_q.get())

    # Wait for all worker processes to finish
    for p in procs:
        p.join()

    return resultdict
```

```
def main():
    """
    Main method of the script, performs benchmark
    :return: exit code 0
    """
    # test performance for n threads and processes from the lists
    N_THREADS = [2, 4, 6, 8, 10, 12, 16]
    N_PROCESSES = [2, 4, 6, 8, 10, 12, 16]

    # get the numbers to factorize as user input
    N = input("how many numbers? ")
    nums = range(N)

    # benchmark factorizing by single thread
    t0 = time.time()
    plain_factorizer(nums)
    t1 = time.time()
    print("single thread: {} seconds".format(t1 - t0))

    # benchmark factorizing by threads
    for nthreads in N_THREADS:
        t0 = time.time()
        threaded_factorizer(nums, nthreads)
        t1 = time.time()
        print("{} threads: {} seconds".format(nthreads, t1 - t0))

    # benchmark factorizing by processes
    for nprocs in N_PROCESSES:
        t0 = time.time()
        multiprocess_factorizer(nums, nprocs)
        t1 = time.time()
        print("{} processes: {} seconds".format(nprocs, t1 - t0))

    return 0


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

Appendix B

Communication schemes description

① Connection to a queue leading to the Stat Poller, used for sending commands to the Stat Poller

② Connection to a queue leading to the Command Executor, used for sending commands to the Command Executor

③ Connection to a queue leading to the I/O Manager, used for sending commands to the I/O Manager

④ Connection to a queue leading to the Data Keeper, used for sending commands to the Data Keeper

⑤ Connection to a queue leading to the GUI core, used for sending machine status variables from the Stat Poller to the GUI

⑥ Connection to a queue leading to the GUI core, used for sending eventual error messages from the Stat Poller to the GUI

⑦ Connection to a queue leading to the API Provider, used for sending machine status variables from the Stat Poller to the API Provider

⑧ Connection to a queue leading to the API Provider, used for sending eventual error messages from the Stat Poller to the API Provider

⑨ Connection to a queue leading to the GUI core, used for sending I/O status variables from the I/O Manager to the GUI

(10) Connection to a queue leading to the API Provider, used for sending I/O status variables from the I/O Manager to the API Provider

(11) Connection to a queue leading to the GUI core, used for sending measured data variables from the Data Keeper to the GUI

(12) Connection to a queue leading to the GUI core, used for sending data to plot from the Data Keeper to the GUI

(13) Connection to a queue leading to the API Provider, used for sending measured data variables from the Data Keeper to the API Provider

(14) Connection to a queue leading to the API Provider, used for sending data to plot from the Data Keeper to the API Provider

(15) Connection of the API Provider and rapo library through the Unix domain socket

(16) Connection to a queue leading to the Server Manager, used for sending machine status variables from the Stat Poller through the Server Manager and Client Manager to the remote GUI

(17) Connection to a queue leading to the Server Manager, used for sending eventual error messages from the Stat Poller through the Server Manager and Client Manager to the remote GUI

(18) Connection to a queue leading to the Server Manager, used for sending machine status variables from the Stat Poller through the Server Manager and Client Manager to the remote API Provider

(19) Connection to a queue leading to the Server Manager, used for sending eventual error messages from the Stat Poller through the Server Manager and Client Manager to the remote API Provider

(20) Connection to a queue leading to the Server Manager, used for sending I/O status variables from the I/O Manager through the Server Manager and Client Manager to the remote GUI

(21) Connection to a queue leading to the Server Manager, used for sending I/O status variables from the I/O Manager through the Server Manager and Client Manager to the remote API Provider

(22) Connection to a queue leading to the Server Manager, used for sending measured data variables from the Data Keeper through the Server Manager and Client Manager to the remote GUI

(23) Connection to a queue leading to the Server Manager, used for sending data to plot from the Data Keeper through the Server Manager and Client Manager to the remote GUI

(24) Connection to a queue leading to the Server Manager, used for sending measured data variables from the Data Keeper through the Server Manager and Client Manager to the remote API Provider

(25) Connection to a queue leading to the Server Manager, used for sending data to plot from the Data Keeper through the Server Manager and Client Manager to the remote API Provider

(26) Connection of the Server Manager and Client Manager through the TCP socket

(27) Connection to a queue leading to the Client Manager, used for sending commands from the remote GUI to the Stat Poller through the Client Manager and Server Manager

(28) Connection to a queue leading to the Client Manager, used for sending commands from the remote GUI to the Command Executor through the Client Manager and Server Manager

(29) Connection to a queue leading to the Client Manager, used for sending commands from the remote GUI to the I/O Manager through the Client Manager and Server Manager

(30) Connection to a queue leading to the Client Manager, used for sending commands from the remote GUI to the Data Keeper through the Client Manager and Server Manager

(31) Connection to a queue leading to the remote GUI core, used for sending machine status variables from the Stat Poller to the remote GUI

(32) Connection to a queue leading to the remote GUI core, used for sending eventual error messages from the Stat Poller to the remote GUI

(33) Connection to a queue leading to the remote GUI core, used for sending I/O status variables from the I/O Manager to the remote GUI

(34) Connection to a queue leading to the remote GUI core, used for sending measured data variables from the Data Keeper to the remote GUI

(35) Connection to a queue leading to the remote GUI core, used for sending data to plot from the Data Keeper to the remote GUI

(36) Connection to a queue leading to the Client Manager, used for sending commands from the remote API Provider to the Stat Poller through the Client Manager and Server Manager

(37) Connection to a queue leading to the Client Manager, used for sending commands from the remote API Provider to the Command Executor through the Client Manager and Server Manager

(38) Connection to a queue leading to the Client Manager, used for sending commands from the remote API Provider to the I/O Manager through the Client Manager and Server Manager

(39) Connection to a queue leading to the Client Manager, used for sending commands from the remote API Provider to the Data Keeper through the Client Manager and Server Manager

(40) Connection to a queue leading to the remote API Provider, used for sending machine status variables from the Stat Poller to the remote API Provider

(41) Connection to a queue leading to the remote API Provider, used for sending eventual error messages from the Stat Poller to the remote API Provider

(42) Connection to a queue leading to the remote API Provider, used for sending I/O status variables from the I/O Manager to the remote API Provider

(43) Connection to a queue leading to the remote API Provider, used for sending measured data variables from the Data Keeper to the remote API Provider

(44) Connection to a queue leading to the remote API Provider, used for sending data to plot from the Data Keeper to the remote API Provider

(45) Connection of the remote API Provider and rapo library on the remote device through the Unix domain socket

Appendix C

Plot comparison script

Listing C.1: Plot comparison script

```
"""
PythonQwt: plotting 100 data samples took 0.00383186340332 seconds
Matplotlib: plotting 100 data samples took 0.0366899967194 seconds
PythonQwt: plotting 1000 data samples took 0.00513195991516 seconds
Matplotlib: plotting 1000 data samples took 0.0264139175415 seconds
PythonQwt: plotting 10000 data samples took 0.00405383110046 seconds
Matplotlib: plotting 10000 data samples took 0.0248889923096 seconds
PythonQwt: plotting 50000 data samples took 0.00433802604675 seconds
Matplotlib: plotting 50000 data samples took 0.0284330844879 seconds
PythonQwt: plotting 100000 data samples took 0.00466799736023 seconds
Matplotlib: plotting 100000 data samples took 0.0300550460815 seconds
PythonQwt: plotting 250000 data samples took 0.00639510154724 seconds
Matplotlib: plotting 250000 data samples took 0.0396320819855 seconds
PythonQwt: plotting 500000 data samples took 0.00936698913574 seconds
Matplotlib: plotting 500000 data samples took 0.0492820739746 seconds
PythonQwt: plotting 1000000 data samples took 0.0142869949341 seconds
Matplotlib: plotting 1000000 data samples took 0.0847151279449 seconds
"""

from PyQt5.QtWidgets import QApplication
from qwt import QwtPlot, QwtPlotCurve

from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgg as FigureCanvas
from matplotlib.figure import Figure
```

```
import numpy as np

import time
import sys


def plot_qwt(n_samples):
    """
    Plot n data samples using PythonQwt
    :param n_samples: number of samples
    :return: exit code
    """
    # create QApplication and Qwt plot widget
    app = QApplication(sys.argv)
    gui = QwtPlot()
    # prepare data samples
    x = np.arange(0.0, 2*np.pi, 2*np.pi/n_samples)
    y = np.sin(x)
    # start measuring plot time
    t0 = time.time()
    curve = QwtPlotCurve()
    curve.setData(x, y)
    curve.attach(gui)
    t1 = time.time()
    # print results
    print "PythonQwt: plotting {} data samples took {} seconds".format(n_samples, t1-t0)
    # gui.show() # uncomment to show user interface

    # return sys.exit(app.exec_()) # uncomment to start QApplication event loop


def plot_matplotlib(n_samples):
    """
    Plot n data samples using matplotlib
    :param n_samples: number of samples
    :return: exit code
    """
    # create QApplication and matplotlib widget
    app = QApplication(sys.argv)
    figure = Figure()
    subplot = figure.add_subplot(111)
    gui = FigureCanvas(figure)
    # prepare data samples
    x = np.arange(0.0, 2*np.pi, 2*np.pi/n_samples)
    y = np.sin(x)
    # start measuring plot time
    t0 = time.time()
    subplot.plot(x, y)
    gui.draw()
    t1 = time.time()
    # gui.show() # uncomment to show user interface
    # print results
    print "Matplotlib: plotting {} data samples took {} seconds".format(n_samples, t1-t0)

    # return sys.exit(app.exec_()) # uncomment to start QApplication event loop


def main():
    """
    Main method of the script
    :return: exit code 0
    """
    for n in [100, 1000, 10000, 50000, 100000, 250000, 500000, 1000000]:
        plot_qwt(n)
        plot_matplotlib(n)

    return 0


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

Appendix D

Serialising modules comparison script

Listing D.1: Serialising modules comparison script

```
"""
pickle.dumps - protocol 0: type <type 'list'> took 1.23852205276 seconds, size 8888896 B
cPickle.dumps - protocol 0: type <type 'list'> took 0.1183581352223 seconds, size 8888896 B
pickle.dumps - protocol 1: type <type 'list'> took 1.43744206429 seconds, size 4870676 B
cPickle.dumps - protocol 1: type <type 'list'> took 0.0186970233917 seconds, size 4870676 B
pickle.dumps - protocol 2: type <type 'list'> took 1.43182492256 seconds, size 4870678 B
cPickle.dumps - protocol 2: type <type 'list'> took 0.0189759731293 seconds, size 4870678 B
pickle.loads: type <type 'list'> took 0.590484857559 seconds
cPickle.loads: type <type 'list'> took 0.0240499973297 seconds
json.dumps: type <type 'list'> took 0.105890989304 seconds, size 7888890 B
json.loads: type <type 'list'> took 0.0827050209045 seconds
pickle.dumps - protocol 0: type <type 'dict'> took 2.33178496361 seconds, size 16777786 B
cPickle.dumps - protocol 0: type <type 'dict'> took 0.241229057312 seconds, size 16777786 B
pickle.dumps - protocol 1: type <type 'dict'> took 2.68368887901 seconds, size 9739348 B
cPickle.dumps - protocol 1: type <type 'dict'> took 0.0424699783325 seconds, size 9739350 B
pickle.dumps - protocol 2: type <type 'dict'> took 2.68153905869 seconds, size 9739350 B
cPickle.dumps - protocol 2: type <type 'dict'> took 0.0431780815125 seconds, size 9739352 B
pickle.loads: type <type 'dict'> took 1.30437397957 seconds
cPickle.loads: type <type 'dict'> took 0.0720989704132 seconds
json.dumps: type <type 'dict'> took 0.286577939987 seconds, size 17777780 B
json.loads: type <type 'dict'> took 0.533478021622 seconds
"""
```

```
import pickle
import cPickle
import json

import time
import sys


def dumps_pickle(data, protocol):
    """
    Pickle/serialize data with pickle module using a protocol
    :param data: data to pickle/serialize
    :param protocol: protocol used for pickling/serialization
    :return: pickled/serialized data
    """
    return pickle.dumps(data, protocol=protocol)


def loads_pickle(data):
    """
    Unpickle/deserialize data with pickle module
    :param data: data to unpickle/deserialize
    :return: unpickled/deserialized data
    """
    return pickle.loads(data)


def dumps_cPickle(data, protocol):
    """
    Pickle/serialize data with cPickle module using a protocol
    :param data: data to pickle/serialize
    :param protocol: protocol used for pickling/serialization
    :return: pickled/serialized data
    """
    return cPickle.dumps(data, protocol=protocol)


def loads_cPickle(data):
    """
    Unpickle/deserialize data with cPickle module
    :param data: data to unpickle/deserialize
    :return: unpickled/deserialized data
    """
    return cPickle.loads(data)


def dumps_json(data):
    """
    Pickle/serialize data with json module
    :param data: data to pickle/serialize
    :return: pickled/serialized data
    """
    return json.dumps(data)


def loads_json(data):
    """
    Unpickle/deserialize data with json module
    :param data: data to unpickle/deserialize
    :return: unpickled/deserialized data
    """
    return json.loads(data)


def main():
    """
    Main method of the script
    :return: exit code 0
    """
    PROTOCOLS = [0, 1, 2]
    DATA = [list(range(1000000)), dict(zip(range(1000000), range(1000000)))]

    for data in DATA:
        # benchmark various pickle and cPickle protocols
        for protocol in PROTOCOLS:
            # benchmark pickle.dumps
            t0 = time.time()
            pickled = dumps_pickle(data, protocol)
            t1 = time.time()
            print "pickle.dumps␣−␣protocol␣{}:␣type␣{}␣took␣{}␣seconds,␣size␣{}␣B".format(
                ↪ protocol, type(data), t1−t0, len(pickled))
            # benchmark cPickle.dumps
            t0 = time.time()
            pickled = dumps_cPickle(data, protocol)
            t1 = time.time()
            print "cPickle.dumps␣−␣protocol␣{}:␣type␣{}␣took␣{}␣seconds,␣size␣{}␣B".format(
                ↪ protocol, type(data), t1−t0, len(pickled))
        # benchmark pickle.loads
        t0 = time.time()
        unpickled = loads_pickle(pickled)
        t1 = time.time()
        print "pickle.loads:␣type␣{}␣took␣{}␣seconds".format(type(data), t1 − t0)
        # benchmark cPickle.loads
        t0 = time.time()
        unpickled = loads_cPickle(pickled)
        t1 = time.time()
        print "cPickle.loads:␣type␣{}␣took␣{}␣seconds".format(type(data), t1−t0)
        # benchmark json.dumps
        t0 = time.time()
        pickled = dumps_json(data)
        t1 = time.time()
        print "json.dumps:␣type␣{}␣took␣{}␣seconds,␣size␣{}␣B".format(type(data), t1−t0, len(
            ↪ pickled))
        # benchmark json.loads
        t0 = time.time()
        unpickled = loads_json(pickled)
        t1 = time.time()
        print "json.loads:␣type␣{}␣took␣{}␣seconds".format(type(data), t1−t0)

    return 0


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