Cavro XP3000 GUI Documentation

Release 1.0

Nikos Koukis

July 15, 2014

Pump3000 is a GUI implementation for communicating with the Cavro XP3000 pump series. It was implemented in Python with the use of PySide. This guide should provide a way of setting up the software on your computer and give you a first taste on how to use it efficiently.

CHAPTER

ONE

CONTENTS:

1.1 Getting Started

There are 2 ways of running the XP3000 GUI:

- Running the *Pump3000.exe* [Windows only]
- Running *from source* [Python required]

Either way the user must first download the necessary files for the software, located at http://www.github.com/bergercookie/Pump3000. This can be done either by downloading the project locally from the github page or by cloning the project

The user can *download the software* by visiting the github page: http://www.github.com/bergercookie/Pump3000 and then pressing the TODO<Download the Desktop> Button.

See picture <TODO> for an example case

<TODO> insert the image

In order to **clone** the project the user must first make sure that git is installed on the platform. If it isn't then installing it requires issuing:

```
sudo apt-get install git "for Linux users - command-line
sudo port install git "MacOS users - command-line
TODO installing for windows
```

Then in order to clone the project the user can run the git clone command:

```
git clone http://wwww.github.com/bergercookie/Cavro-Pump-XP3000-GUI.git
```

1.2 Hardware Configuration

1.2.1 Equipment used

For the hardware communication to be achieved the following equipment is suggested:

- USB -> Serial (RS-232) adapter
- RS-232 -> RS-485 converter
- Power cables (Power + GND)
- Jumper wires (at least 2 Female to Male)
- Breadboard (optional)

1.2.2 Connecting the XP3000

The serial communication with the pump was implemented using the RS-485 serial protocol. Here is a sample configuration that was used:

TODO

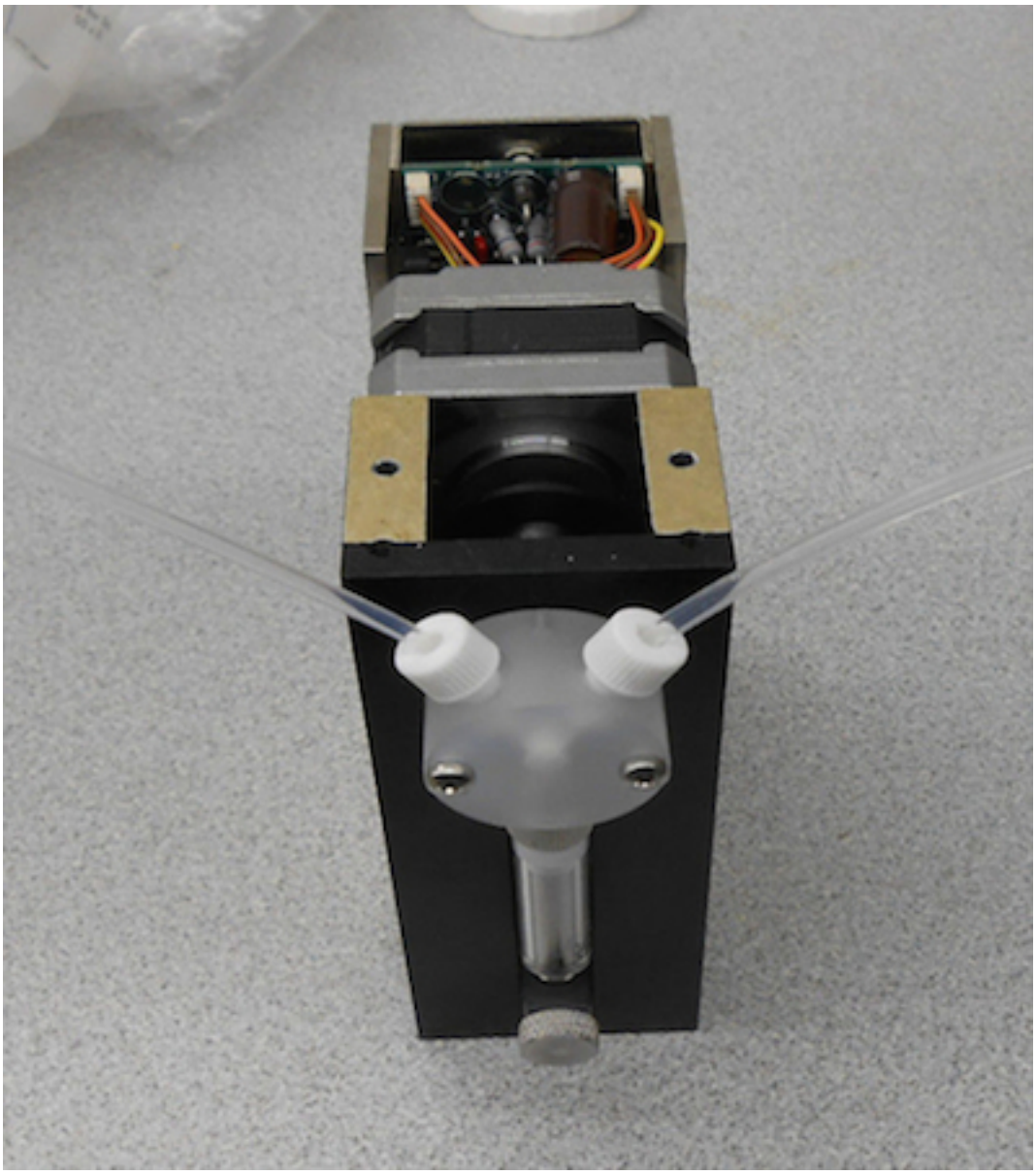

Figure 1.1: Suggested Hardware Communication to the pump

The needed steps to archive a similar connection would be the following:

1. First you have to supply the pump with the necessary power. As described in the manual the XP3000 requires a **24VDC power supply** with a current rating of at least **1.5A**. As seen in hardware-conf, during the development part, the power was supplied indirectly to the pump first by running the supply cables into a breadboard and then using extra jumpers to connect to pump pins #1 (power) and #9(GND)

2. For the serial communication, you need to connect the RS-485A, RS-485B signals from the pump PCB into the RS-485+, RS-485- of the data terminal (PC) used for the communication. Take extra notice when it comes to connecting the jumpers from the serial adapter to the pump PCB corresponding pins

To sum it up, here is the typical RS-485 pinout for the pump [1], as it is described above

Role	Pin No	Signal to Connect
Power	#1 [Power]	24 VDC
	#4 [GND]	Ground (of the power supply)
Protocol Signals	#11 [RS-485A]	RS-485 (+)
	#12 [RS-385B]	RS-485 (-)

1.3 Software Configuration

By now the user should have a copy of the project on his machine. If not refer to section: 'Getting Started' TODO!! <Cross referense>

1.3.1 Running the .exe

Running the executable version of the project is as simple as running the Pump3000.exe located in the <location_to_project>/build folder.

<TODO> insert the image

1.3.2 Running from source

In case the user wants to run the software *from source*, a basic python distribution must be installed on the platform, (already preconfigured on most *NIX systems). The user must also have the following packages installed:

- pyserial
- PySide

After this configuration, the user can run the software from the command-line [*NIX] command-prompt [Windows]:

```
python Pump3000.py
```

Note that the user must first go to the folder, the Pump3000.py is located.

<TODO> insert the image

1.3.3 Using the software

The first thing the user should do, if he doesn't know the serial connector to the pump, is figure out the correct port:

- On **Windows** machines this can be done by getting the *'Ports (COM & LPT)'* tab:

  ```
  Start Menu > right-click "My Computer" > select "Manage" >  Click on the "Device Manager"

  (On the "Device Manager") Click "Ports (COM & LPT)" tab > select the port your connector is on
  ```

- On ***NIX** machines this can be done the following way:

[1] For more information regarding the pump pinout consult the pump manual

```
cd /dev

ls -lart| grep tty
```

This will give you a list of the currently available ports. Ejecting and reinserting the connector to the computer and in the meantime running the second command again will help you recognise the port the pump is running on.

<TODO insert images!!>

Warning: Make sure that you have selected the correct port, otherwise the pump will not respond and will not raise any error Message

After that you are ready to run the software (whichever way you want). You should be directed to the New_Device window where the port connected to the pump must be selected <TODO image ref>

<TODO> insert the image

If you have selected the correct port, the connection to the pump is established and the software will automatically initialize the pump, by moving the plunger to the upper position.

You are now in the Main Window. From here you can:

- Command a volume delivery,
- Change the speed of the plunger,
- Issue a quick command to the pump (halt, push_all, etc)

<TODO> image

From the main window you can navigate to a series of other dialogs:

- **Editor's Tab**

  The Editor's Tab gives the user the ability to issue a series of commands to the pump. These commands are supplied in the "Pump Commands" page. The user can also issue raw pump commands in the same way.

  A typical example of issued commands would be the following:

  ```
  pump.property_set('speed', '5')

  # Python Comments, write as many as you want
  # Empty lines don't matter
  # Raw commands as well
  /1?2R\r

  pump.send_Command('A0')
  ```

- **History**

  From here the user can see all the commands sent to the pump which can be devided to 2 types:

  - Commands issued by the user
  - Commands issued by the software to decide pump status

- **Syringe Size**

  The user can decide the syringe size.

- **Reports**

  Gives the user an overview of the pump currently configured settings

- **Pump Parameters**

  The user can change certain parameters of the plunger movement such as “Top Velocity”, “Slope” etc.

  Port

  The user can configure the port that the pump is connected to. This window is also summoned at the start of the Pump30000

1.4 Pump Commands

In this section the basic commands for communication with the pump are explained. You can issue a series of these commands from the editor’s tab. For the full list of available commands see the code Walkthrough section.

Note: ‘self’ is a Python class arguement and shouldn’t be issued by the user.

---

pump.send_Command(self, command, bits_on_return = 0)

Arguements:

```
command        : string
bits_on_return : int [Optional]
```

Description

Sending Commands to the pump. Consult the manual for a detailed list of the pump commands for the RS-485 Protocol. Using this method you must have already defined the pump address to send to (Via the main window or the Editor tab) and you should issue neither the Run character [R] nor the terminating character

Examples:

```
pump.send_Command('A3000', 10) -> Move the plunger to abs. position 3000, read back 10 bits
pump.send_Command('T') -> Terminate plunger move in progress
```

pump.connect_new(self, port_name)

Arguements:

```
port_name: string
```

Description

Connect to the given port address.

Example:

```
pump.connect_new('/dev/tty.-SerialPort1-1') -> Configuring a serial connection on OSX
```

pump.initialize_pump(self, output = ‘right’)

Arguements:

```
output: string [Optional]

    values: 'left', 'right'
```

Description

Initialize the pump. The user can optionally issue the output valve. By default if not given otherwise, the output valve is set to the right.

Examples:

```
pump.initialize_pump() -> Initializing the pump with default output valve (right)
pump.initialize_pump(output = 'left') -> Initializing the pump with output valve set to left
```

pump.valve_command(self, position)

Arguements:

```
position: string

    values: 'I' (Input)
            'O' (Output)
            'B' (Bypass)
```

Description

Set the valve to certain position.

Examples:

```
pump.valve_command('B') -> Set the valve to Bypass position
```

pump.property_set(self, a_proprty, value)

Arguements:

```
a_property : string
value      : int
```

Description

Set a certain property for the plunger. The following properties are available for modification. These are listed with regards to the command that should be issued and the range of the values permitted:

```
'speed'                 : 'S', [1  , 40]
'backlash'              : 'K', [0  , 31]
'slope                  : 'L', [1  , 20]
'start_velocity'        : 'v', [50 , 1000]
'top_velocity'          : 'V', [5  , 5800]
'cutoff_velocity'       : 'c', [50 , 2700]
'cuttof_velocity_steps' : 'C', [0  , 25]
```

The user is encouraged to consult the manual for an overview on the commands above

Examples:

```
pump.property_set('speed', '10') -> Set the plunger speed to 10
pump.property_set('backlash', '15') -> set the backlash steps to 15
```

pump.volume_command(self, direction = 'P', vol = '0', special=None)

Arguements:

```
direction : string

    Values: 'P'
            'D'

vol       : string [Microliters]

special   : string [Optional]
```

```
    Values: 'push_all'
            'pull_all'
```

Description

Volume pushing / drawing mechanism. The user can issue a volume delivery as well as issue a special push / pull all action. In case the special action is given, the 'vol' argument is neglected

Examples:

```
pump.volume_command(direction = 'D', vol = '5') -> Dispense 5 microlitres
pump.volume_command(special = 'push_all') -> Dispense fluid volume
```

pump.ser.write(comand)

Arguements:

```
command: string
```

Description

This command should be used when a direct serial command has to be sent to the pump. User must issue the pump to which he is addressing to as well as the terminating character It is advisable that the user should prefere higher level commands such as 'send_Command' which doesn't require the prefix & suffix characters, and also check the availability of the pump

Examples:

```
pump.ser.write('/2ZR\r') \verb|->| Initialize the pump with the address 1
```

Note: As seen in the Examples section of pump.ser.write, the user should refer to the pump address + 1.

1.5 Code Walkthrough

This purpose of this section is to give you the basic information about the GUI implementation with the use of Python and PySide. For those who aren't familiar coding in Python or haven't heard of GUI development using PySide and Qt may find the links below useful:

-Official Python website: https://www.python.org/

-PySide: Python Bindings for Qt: http://qt-project.org/wiki/pyside

1.5.1 Full Python Code

Script #1: pump_model.py

```
# Tue Jun 3 19:49:03 EEST 2014, nickkouk

"""

This is the pump model for the Cavro XP3000 Pump series.

It is designed to implement all the basic controls of the pump independent of
the way the user interacts with it (GUI, CLI, etc).

"""
```

```
# Imports

from __future__ import division
__import__('serial.urlhandler.protocol_loop') # Special import, needed for py2exe
import serial
import threading
import sys
from exceptions_module import BusyPump
import time
import signal
import Queue

class Pump():
    def __init__(self, addr, com = 'serial'):


        self.addr = '/%s' %addr
        self.term = 'R\r'
        self.com = com

        # Control Related properties
        self.history = [ ]
        self.interval = 2
        self.timeout_time = 0.2

        self.stop_flag = False
        signal.signal(signal.SIGINT, self.stop_thread)


        # sending mechanism
        self.exc_mode = 'interactive'
        self.questions_out = Queue.Queue(-1)
        self.answers_ret = Queue.Queue(-1)
        self.answers_lock = threading.Lock()
        self.update_sema = threading.Semaphore(value=1)

        # Dictionary for holding the pump properties
        self.status = {"absolute_pos": '',\
                "top_vel": '',\
                "cutoff_vel": '',\
                "actual_pos": '', \
                "starting_vel": '',\
                "backlash_steps": '',\
                "fluid_sensor": '',\
                "buffer_status": '',\
                "version": '',\
                "checksum": ''}


        # Hold local properties
        self.own_status = {"plung_pos_mine": 0,\
                "valve_pos": '0',\
                "syringe_size": 50,\
                "steps_tot": 3000}

        self.correspondance = {"speed": ['S', 1, 40],\
                "backlash": ['K', 0, 31],\
                "slope": ['L', 1, 20],\
```

```
            "start_velocity": ['v', 50, 1000],\
            "top_velocity": ['V', 5, 5800],\
            "cutoff_velocity": ['c', 50, 2700],\
            "cutoff_velocity_steps": ['C', 0, 25],\
            }


    # Start the delivery thread
    self.deliveryThread = delivery_thread(self)
    self.deliveryThread.start()
    print "THREAD STARTED"


    self.connect_new(port_name = 'loop://')

# New serial connection
def connect_new(self, port_name):
    """Function for configuring a new serial connection."""

    try:
        self.ser = serial.Serial(port = port_name,\
                baudrate = 9600,\
                parity = 'N',\
                stopbits = 1,\
                bytesize = 8,\
                timeout = self.timeout_time)
    except serial.SerialException:
        self.ser = sys.modules['serial.urlhandler.protocol_loop'].Serial('loop://',\
            timeout = self.timeout_time)
    except:
        print sys.exc_info()[0]

    finally:
        self.initialize_pump()


# Initialization Phase
def initialize_pump(self, output = 'right'):
    """Initialize (the newly) configured pump"""

    try:
        self.ser.open()
        print "Opened the Serial port {} successfully".format(self.ser.port)
    except:
        pass

    # These commands should be sent when the pump first gets set
    if output == 'right':
        commands = ['S10', 'Z']
    else:
        commands = ['Y', 'S10']
    for command in commands:
        self.send_Command(command, 10)

    # Actions after every initialization
    self.history = [ ]
    self.update_values(initialize = True)
```

```
    # Valve Position
    def valve_command(self, position):
        if position == "out":
            answer = self.send_Command('O', 10)
            self.own_status["valve_pos"] = 'O'
        elif position == "in":
            answer = self.send_Command('I', 10)
            self.own_status["valve_pos"] = 'I'
        elif position == "bypass":
            answer = self.send_Command('B', 10)
            self.own_status["valve_pos"] = 'B'
        else:
            print "NO SUCH VALUE for position available"
        return answer

    # Plunger Functions
    def property_set(self, a_property, value):
        # Checking the validity of the command
        min_value, max_value = self.correspondance[a_property][1:]
        if int(value) >= min_value and int(value) <= max_value:
            # Sending the Command to the queue
            command = self.correspondance[a_property][0]
            value = "%s" %value
            print "SENDING THE COMMNAND"
            return_stat = self.send_Command("{}".format(\
                command + value), 10)
            return return_stat
        else:
            return "OUT OF BOUNDS"

    def volume_command(self, direction = 'P', vol = 0, special = None):
        """
        This is the volume command.

        The volume_command function first decides if it can
        deliver the needed volume,
        then if the volume given is a valid one,
        calls the move_plunger method with the needed arguments
        to deliver the volume.

        """

        valid = "True"
        status = "Done"

        if special:
            if special == 'push_all':
                self.send_Command('A0')
                self.own_status["plung_pos_mine"] = 0
            else:
                self.send_Command('A3000')
                self.own_status["plung_pos_mine"] = 3000

        else:
            if not vol.isdigit():
                return (False, "Please enter a numerical value")

            vol = float(vol)
```

```
            steps = int(self.own_status["steps_tot"] / \
                    self.own_status["syringe_size"] * vol)

            if direction == 'D':
                if self.own_status["plung_pos_mine"] - steps < 0:
                    valid = False
                    status = "Not a valid Value"
                else:
                    self.own_status["plung_pos_mine"] -= steps
                    status = self.send_Command(direction + "%s" %steps, 10)

            else:
                if self.own_status["plung_pos_mine"] +\
                        steps > self.own_status["steps_tot"]:
                    valid = False
                    status = "Not a valid Value"
                else:
                    self.own_status["plung_pos_mine"] += steps
                    status = self.send_Command(direction + "%s" %steps, 10)

        return (valid, status)

    # Pump related Settings update method
    def update_values(self, initialize = False):
        """
        This is the parameters update method.

        The purpose of this function is to constantly update the settings
        related to the pump, should be run by a thread periodically
        """

        self.update_sema.acquire()
        print "ENTERED THE SEMAPHORE"
        self.update_thread = threading.Thread(\
                target = self.actual_update_method,\
                args = (initialize,))
        self.update_thread.start()
    def actual_update_method(self, initialize):
        try:
            # reading info mechanism
            self.status["absolute_pos"] = self.send_Command('?', 10)[0][3:]
            self.status["actual_pos"] = self.send_Command('?4', 10)[0][3:]
            self.status["starting_vel"] = self.send_Command('?1', 10)[0][3:]
            self.status["top_vel"] = self.send_Command('?2', 10)[0][3:]
            self.status["cutoff_vel"] = self.send_Command('?3', 10)[0][3:]
            self.status["backlash_steps"] = self.send_Command('?12', 10)[0][3:]
            self.status["fluid_sensor"] = self.send_Command('?22', 10)[0][3:]
            self.status["buffer_status"] = self.send_Command('?F', 10)[0][3:]
            self.status["version"] = self.send_Command('?&', 10)[0][3:]
            self.status["checksum"] = self.send_Command('?#', 10)[0][3:]
        except TypeError:
            print "In actual_update_method,\n{}".format(sys.exc_info()[0])

        abs_pos = self.status["absolute_pos"][:-3]
        try:
            if 0 <= int(abs_pos) <= 3000:
                self.own_status["plung_pos_mine"] = int(abs_pos)
                print "Plunger Position is set: {}".format(self.own_status["plung_pos_mine"])
```

```
            else:
                print "Out of range value reported by pump!"
                pass
        except ValueError:
            print sys.exc_info()[0]

        self.update_sema.release()
        print "EXITED THE SEMAPHORE"
    # Supplementary Functions
    def terminate_execution(self):
        self.send_Command('T')
        time.sleep(0.1)
        self.update_values() # So that the plunger position may refresh itself

    def stop_thread(self, signum=0, fname=0):
        print "C-c caught!!"
        self.stop_flag = True
        sys.exit(1)

    def change_mode(mode):
        """ Function for changing the mode the commands are executed.

        Must be invoked at the end of the editor commands"""
        self.exc_mode = mode
        print "the mode has changed to {}".format(self.exc_mode)

    def send_Command(self, command, bits_to_read = 0):
        """Major mechanism for sending the commands to the pump.

        Stores the coming commands on a queue from which they are sent to the pump.
        Decides upon waiting for the answer if on interactive mode, or not.
        Finally returns the answer to the calling function
        """

        full_command = self.addr + command + self.term
        try:
            self.answers_lock.acquire()
            self.questions_out.put(full_command)
            self.history.append(full_command)
            if self.exc_mode == 'interactive':
                if bits_to_read:
                    answer = self.answers_ret.get(timeout = 1)
                else:
                    self.answers_ret.get(timeout = 0.5)
                    print "NO NEED TO READ"

        except Queue.Empty:
            print "Answer Queue is empty!!"
        except:
            print "ERROR in SEND_COMMAND:"
            print sys.exc_info()[0]
            self.stop_thread()
        finally:
            self.answers_lock.release()
            if self.exc_mode == 'interactive':
                if 'answer' in locals():
                    return answer
            else:
```

```
                return 0

class delivery_thread(threading.Thread):
    """
    Class for sending the queued commands to the pump.

    When initialized, the instance of the class inherits the pump instance as
    well as the send & return queues.

    """

    def __init__(self, pump):
        threading.Thread.__init__(self)
        self.pump = pump
        self.forSending = self.pump.questions_out
        self.forGivingBack = self.pump.answers_ret

    def run(self):
        while not self.pump.stop_flag:
            try:
                com_to_send = self.forSending.get(timeout = 3)
                self.pump.ser.write(com_to_send)
                time.sleep(0.1)
                answer = self.pump.ser.read(11)
                done, answer = self.push_command(com_to_send, answer)
                if self.pump.exc_mode == 'interactive':
                    self.forGivingBack.put((answer, done))
            except Queue.Empty:
                time.sleep(1)
                print "Questions Queue is empty"

    def push_command(self, *QA):
        com_to_send = QA[0]
        answer = QA[1]
        if self.pump.exc_mode == 'editor':
            # editor Mode
            while self.pump_busy(com_to_send, answer):

                self.pump.ser.write(com_to_send)
                time.sleep(0.1)
                answer = self.pump.ser.read(11)
            done = True
        else:
            # interactive Mode
            if self.pump_busy(com_to_send, answer):
                done = False
            else:
                done = True
        print "\n\ndone = {0}\nmode = {1}\nquestion = {2}\nanswer = {3}".format(done, self.pump.exc_m
        return (done, answer)

    def pump_busy(self, *QA):
        """
        Determines if the pump is busy for the specific command to send.

        Accepts as input the command to the pump the answer returned and returns
        to the caller if the command has been accepted by the pump
        """
```

```
        question = QA[0]
        answer = QA[1]

        print 'The answer is {}'.format(answer)
        if 'o' in answer:
            return True
```

Script #2: Pump3000.py

```
#!/usr/bin/env python
# Mon May 26 23:48:35 EEST 2014, nickkouk
"""
================================================================================
                                    About
================================================================================

Pump3000 was developed by Nikos Koukis (nickkouk@gmail.com). Its main purpose is
to offer a simple Graphical User Interface for communication with the Cavro XP3000
pumps. It was created primarily for scientific usage for the "Systems Biology and
Bioengineering" Laboratory of the National Technical University of Athens
(NTUA).

For further assistance see the documentation on http://https://github.com/bergercookie

For bug reports, and suggestions either raise an issue on the Github page of the
project, or contact me in person (nickouk@gmail.com)

================================================================================
                          Redistribution Policy
================================================================================

This is a free software, licensed under LGPLv2. It can be redistributed and
modified as long as the freedom to use is passed on to the recipients. For complete
policy of terms, consult a version of the license, either on the documentation page:
https://github.com/bergercookie/Cavro-Pump-XP3000-GUI/blob/master/LICENSE or on
the GNU page: https://www.gnu.org/licenses/lgpl-2.1.html

================================================================================
            Python Implementation for the Cavro XP3000 GUI.
                              Pump3000
================================================================================

The GUI is designed with regards to the MVC (Model View Controller)
UI architectural pattern. Inside this module the View - Controller behaviors
are implemented while the Model behavior is implemented in the imported module: pump_model.py


"""


# proper division for python 2.7
from __future__ import division

# Usual importing stuff for PySide
from PySide.QtGui import *
from PySide.QtCore import *
```

```
# Module imports
import sys
import time

import pump_model
from classes_used import HistoryDialog,\
        ReportsDialog,\
        NewDevDialog,\
        SyringePickDialog,\
        ParametersChange,\
        AboutDialog,\
        AboutDialog2

# Qt-Designer compiled python code
import python_gui
import python_settings
import history_settings
import device_configuration


class MainWindow(QMainWindow, python_gui.Ui_MainWindow):

    def __init__(self, parent=None):
        super(MainWindow, self).__init__(parent)
        self.setupUi(self)

        self.__appname__ = "Pump3000"
        self.setWindowTitle(self.__appname__)

        # Initialize a pump instance
        # You define how to start the pump
        self.pump = pump_model.Pump(3, 'serial')

        # MainWindow related parameters
        self.quick_action = 'push_all'
        self.editors_open = 1
        self.lcdNumber.display(10)

        #open fd
        self.open_files = [ ]
        self.save_time = 0
        self.modified_time = -1

        # Setting tabs titles
        self.tabWidget.setTabText(0, 'Controls')
        self.tabWidget.setTabText(1, 'Editor - {}'.format(self.editors_open))

        # Reverting any changes not shown in the Window
        self.revertQtimer = QTimer(self)
        self.revertQtimer.setInterval(60000)
        self.revertQtimer.start()

        # Initialize the port configuration
        self.newDev()

        # Set up a timer for periodical refresh of the pump parameters
        self.refresh_status = True
        self.refreshQtimer = QTimer(self)
```

```
        self.refreshQtimer.setInterval(60000)
        self.refreshQtimer.start()

        # Other dialogs instances
        self.reports_window = ReportsDialog(self.pump,\
                self,\
                parent = None)

        # saving the previous status of commands to revert in case pump busy
        self.prev_plung_pos = 10
        self.prev_valve_btn = self.output_btn
        self.valve_correspond = {"out": self.output_btn,\
                "in": self.input_btn,\
                "bypass": self.bypass_btn
                }

        # Valve Signals
        self.connect(self.output_btn,\
                SIGNAL("clicked()"),\
                self.valve_status)
        self.connect(self.input_btn,\
                SIGNAL("clicked()"),\
                self.valve_status)
        self.connect(self.bypass_btn,\
                SIGNAL("clicked()"),\
                self.valve_status)

        # Plunger Movement
        self.connect(self.speed_slider,\
                SIGNAL("sliderReleased()"),\
                self.speed_control)
        self.connect(self.speed_slider,\
                SIGNAL("valueChanged(int)"),\
                self.lcd_display)
        self.connect(self.volume_button,\
                SIGNAL("clicked()"),\
                self.volume_control)
        self.connect(self.volume_prompt,\
                SIGNAL("textEdited(QString)"),\
                self.enable_button)

        # Quick CommandsComboBox
        self.connect(self.quick_combobox,\
                SIGNAL("currentIndexChanged(int)"),\
                self.quick_combobox_fun)
        self.connect(self.address_combobox,\
                SIGNAL("currentIndexChanged(int)"),\
                self.set_address)
        self.connect(self.make_it_so,\
                SIGNAL("clicked()"),\
                self.make_it_so_function)

        # Other Dialogs
        self.connect(self.actionReports,\
                SIGNAL("triggered()"),\
                self.reports_window_open)
        self.connect(self.actionHistory,\
                SIGNAL("triggered()"),\
```

```
        self.history_window_open)
self.connect(self.actionNew_Device,\
        SIGNAL("triggered()"),\
        self.newDev)
self.connect(self.actionSyringe_Size,\
        SIGNAL("triggered()"),\
        self.newSyringe)
self.connect(self.actionPump_Parameters,\
        SIGNAL("triggered()"),\
        self.parameters_change_open)
self.connect(self.actionLeft_Valve,\
        SIGNAL("triggered()"),\
        self.init_valve_left)
self.connect(self.actionRight_Valve,\
        SIGNAL("triggered()"),\
        self.init_valve_right)
self.connect(self.actionAbout,\
        SIGNAL("triggered()"),\
        self.about_software)
self.connect(self.actionLicense,\
        SIGNAL("triggered()"),\
        self.about_license)
self.connect(self.actionCommands,\
        SIGNAL("triggered()"),\
        self.editor_help)

# QTimer events
self.connect(self.refreshQtimer,\
        SIGNAL("timeout()"),\
        self.update_pump_values)
self.connect(self.revertQtimer,\
        SIGNAL("timeout()"),\
        self.revert_GUI_values)

# Editor related action buttons
self.connect(self.actionSave,\
        SIGNAL("triggered()"),\
        self.save_btn)
self.connect(self.actionSaveAs,\
        SIGNAL("triggered()"),\
        self.saveAs_btn)
self.connect(self.actionOpen,\
        SIGNAL("triggered()"),\
        self.open_btn)
self.connect(self.textEdit,\
        SIGNAL("textChanged()"),\
        self.update_modify_time)
self.connect(self.run_script_btn,\
        SIGNAL("clicked()"),\
        self.run_script)
self.connect(self.clear_editor_btn,\
        SIGNAL("clicked()"),\
        self.clear_editor)

# Visualizing the pump
if sys.platform[:3] == 'win': #Running on windows, most probably executable
    self.pixmap = QPixmap("../../Images/cavro.jpg")
else:
```

```
            self.pixmap = QPixmap("../Images/cavro.jpg")

        self.pixmap_item = QGraphicsPixmapItem(self.pixmap)
        self.scene = QGraphicsScene()
        self.scene.setBackgroundBrush(Qt.gray)
        self.scene.addItem(self.pixmap_item)
        self.graphicsView.setScene(self.scene)

    def closeEvent(self, event):
        if self.save_time <= self.modified_time:
            quit_msg = "The editor script has been modified,\nDo you still want to continue"
            reply = QMessageBox.question(self, self.__appname__,\
                    quit_msg,\
                    QMessageBox.Yes,\
                    QMessageBox.No)

            if reply == QMessageBox.Yes:
                event.accept()
            else:
                event.ignore()
        else:
            try:
                self.refreshQtimer.stop()
                self.pump.stop_thread()
            except:
                pass
            # Close open file descriptors
            for a_file in self.open_files:
                a_file.close()
                print "closing {}".format(a_file)
            self.pump.stop_thread()
            self.close()

    def save_btn(self):
        dir = "."
        text = self.textEdit.toPlainText()
        try:
            self.file_fd.write(text);self.file_fd.seek(0)
            self.save_time = time.time()
        except AttributeError:
            self.saveAs_btn()

    def saveAs_btn(self):
        dir = "."
        text = self.textEdit.toPlainText()
        fileObj = QFileDialog.getSaveFileName(self, self.__appname__, dir=dir, filter="Text Files (*
        fileName = fileObj[0]
        try:
            self.file_fd = open(fileName, mode="w")
            self.open_files.append(self.file_fd)
            self.file_fd.write(text);self.file_fd.seek(0);
            self.save_time = time.time()
        except IOError: # Cancel was pressed
            pass

    def open_btn(self):
        dir = "."
        fileObj = QFileDialog.getOpenFileName(self, self.__appname__ + " Open File Dialog", dir=dir,
```

```
        fileName = fileObj[0]

        try:
            file_fd = open(fileName, "r")
            self.open_files.append(file_fd)
            read = file_fd.read()
            self.textEdit.setText(read)
            file_fd.close()
        except IOError:
            pass

    def update_modify_time(self):
        self.modified_time = time.time()

    def run_script(self):
        self.pump.exc_mode = 'editor'
        text = self.textEdit.toPlainText()
        script_commands = text.split('\n')
        script_commands = filter(lambda x: x != '', script_commands)
        script_commands.append('pump.change_mode(interactive)')
        try:
            self.pump.update_values()
            for i in range(len(script_commands)):
                if script_commands[i][:4] == 'pump':
                    eval("self." + script_commands[i])
                    print "self.%s" %script_commands[i]
                elif script_commands[i][0] == '/':
                    self.pump.send_Command(script_commands[i])
                else:
                    # TODO The user currently may not issue common python commands yet
                    eval(script_commands[i])
                    print script_commands[i]
        except:
            print sys.exc_info()[0]


    def clear_editor(self):
        self.textEdit.clear()

    def init_valve_left(self):
        self.pump.initialize_pump(output = 'left')
        self.speed_slider.setValue(10)
    def init_valve_right(self):
        self.pump.initialize_pump(output = 'right')
        self.speed_slider.setValue(10)


    # Qtimer event handling
    def update_pump_values(self):
        """
        First call the refresh method of the reports_window instance
        which in turn calls the update_values on the pump
        """
        self.reports_window.refresh()

    def revert_GUI_values(self, doit = False):
        """
        Function for applying the GUI properties to the pump.
```

```
        The user can call this function to make sure that the status on screen
        do correspond to the actual pump properties. The function should always
        be called when the pump was previously running on editor mode and the
        user switches back to interactive mode.
        """

        if doit:
            self.pump.exc_mode = 'interactive'

        if self.pump.exc_mode  == 'interactive':
            print "Setting the GUI values.\ndoit = {0}\nexc_mode ={1}".format(doit, self.pump.exc_mod
            self.valve_status()
            speed_val = self.speed_slider.value()
            self.pump.property_set('speed', '%s' % speed_val)
            self.pump.update_values()


    def cancel_timer(self):
        """
        The purpose of the cancel_timer is to cancel the status refreshing.
        Can be used by the user to manually update the pump status
        """

        self.refreshQtimer.stop()
        self.refresh_status = False
        print "TIMER closed"

    # Enabling the volume button
    def enable_button(self):
        if self.volume_prompt.text().isdigit():
            self.volume_button.setEnabled(True)
        else:
            self.volume_button.setEnabled(False)


    # Functions for either fulling or emptying the whole syringe
    def quick_combobox_fun(self):
        if self.quick_combobox.currentIndex() == 0:
            self.quick_action = 'push_all'
        elif self.quick_combobox.currentIndex() == 1:
            self.quick_action = 'pull_all'
        elif self.quick_combobox.currentIndex() == 2:
            self.quick_action = 'terminate'
        elif self.quick_combobox.currentIndex() == 3:
            self.revert_GUI_values()
    def make_it_so_function(self):
        if self.quick_action == 'terminate':
            self.pump.terminate_execution()
            self.revert_GUI_values()
        else:
            if self.bypass_btn.isChecked():
                answer = "Bypass Mode is ON"
                QMessageBox.warning(self, self.__appname__, answer)
                return
            else:
                self.pump.volume_command(special = self.quick_action)
                label = "Quick Action {action} executed".format(action = self.quick_action)
                self.command_label_show(label)
```

```
    # Opening Dialogs
    def editor_help(self):

        fd1 = open('../text_files/pump_commands_html', 'r')
        text = fd1.read()
        self.open_files.append(fd1)
        self.editor_help_win = AboutDialog(text,\
                "Editor Commands",\
                parent = None)
        self.editor_help_win.show()
        self.editor_help_win.raise_()

    def about_license(self):

        fd1 = open('../text_files/lgpl-2.1.txt', 'r')
        text = fd1.read()
        self.open_files.append(fd1)
        self.about_dialog = AboutDialog(text, "Software License")
        self.about_dialog.exec_()

    def about_software(self):

        fd1 = open('../text_files/about.txt', 'r')
        text = fd1.read()
        self.open_files.append(fd1)
        self.about_soft = AboutDialog2(text, "About the Software")
        self.about_soft.exec_()

    def parameters_change_open(self):
        """ Opens a ParametersChange Instance."""

        self.parameters_window = ParametersChange(self.pump, self)
        self.parameters_window.show()
        self.parameters_window.raise_()

        self.connect(self.parameters_window,\
                SIGNAL("accepted()"),\
                self.parameters_window.update_pump_param)



    def history_window_open(self):
        """ Opens a HistoryDialog Instance"""

        self.history_window = HistoryDialog(self.pump)
        self.history_window.show()
        self.history_window.raise_()
        self.history_window.refresh_Btn.click()

    def reports_window_open(self):
        """ Opens a ReportsDialog Instance"""


        self.connect(self.reports_window.noRefresh,\
                SIGNAL("clicked()"),\
                self.reports_window.tick_refresh)

        self.reports_window.refresh()
```

```
        self.reports_window.show()
        self.reports_window.raise_()

    # Device Configuration
    def newDev(self):
        self.dev_window = NewDevDialog(self.pump, self)
        self.dev_window.exec_()

    def newSyringe(self):

        self.syringe_window = SyringePickDialog(self.pump, self)
        self.syringe_window.show()
        self.syringe_window.exec_()

        self.connect(self.syringe_window,\
                SIGNAL("accepted()"),\
                self.syringe_window.select_new_syringe())

    def set_address(self):
        self.pump.addr = '/%s' %self.address_combobox.currentText()
        label = "Addressing to pump #{}".format(self.pump.addr[-1])
        self.command_label_show(label)

    # Valve Status
    def valve_status(self):
        """ Configures the valve of the pump
        Determines which valve is pressed,
        sends the command,
        updates the label """

        if self.output_btn.isChecked():
            pos = "out"
        elif self.input_btn.isChecked():
            pos = "in"
        else:
            pos = "bypass"

        answer, done = self.pump.valve_command(pos)
        if done:
            label = "Valve position changed to {valve_pos}".format(\
                    valve_pos = self.pump.own_status["valve_pos"])
            self.command_label_show(label)
            self.prev_valve_btn = self.valve_correspond[pos]
        else:
            self.prev_valve_btn.click()


    # Plunger speed
    def speed_control(self):
        # Changing the velocity of the plunger

        speed = self.speed_slider.value()
        self.pump.property_set("speed", speed)
        label = "Plunger Speed Changed: {}".format(speed)
        self.command_label_show(label)

    def lcd_display(self):
        speed = self.speed_slider.value()
```

```
        self.lcdNumber.display(speed)

    def volume_control(self):
        """function for calling the pump.volume_command method."""

        volume = self.volume_prompt.text()
        if self.bypass_btn.isChecked():
            answer = "Bypass Mode is ON"
            QMessageBox.warning(self, self.__appname__, answer)
            return
        if self.PushBtn.isChecked():
            direction = "D"
        elif self.PullBtn.isChecked():
            direction = "P"
        #print "volume: {0}\n volume_type: {1}\n direction: {2}: ".format(volume, type(volume), direc
        (done, answer) = self.pump.volume_command(direction, volume)
        #print "done: {0}\n answer: {1}".format(done, answer)

        if not done:
            QMessageBox.warning(self, self.__appname__, answer)
        else:
            if direction == "D":
                label = "Pushed {volume}".format(volume = volume)
            else:
                label = "Drew {volume}".format(volume = volume)
            self.command_label_show(label)
        self.volume_prompt.selectAll()


    # The statusline
    def command_label_show(self, a_string):
        self.command_label.setText(a_string)


if __name__ == '__main__':
    app = QApplication(sys.argv)
    window= MainWindow()
    window.show()
    app.exec_() # Event-loop of the application
```

Script #3: classes_used.py

```
#  Thu May 29 02:59:41 EEST 2014, nickkouk

"""
In this module a set of supplementary classes are set.

These classes are used primarily for the window_controller functions and assist
in the UI setup
"""

# proper division for python 2.7
from __future__ import division

# Usual importing stuff for PySide
from PySide.QtGui import *
from PySide.QtCore import *
```

```
# Module imports
import sys
import os
from serial.tools.list_ports import comports

# Qt-Designer compiled python code
import python_gui
import python_settings
import history_settings
import device_configuration
import parameters_change
import about_dialog
import syringe_pick


class ParametersChange(QDialog, parameters_change.Ui_Dialog):
    """
    This is the ParametersChange Class.

    When an instance of this class is generated the ParametersChange
    window pops up in the screen
    """

    def __init__(self, pump, window, parent = None):
        super(ParametersChange, self).__init__(parent)
        self.setupUi(self)
        self.pump = pump

    def update_pump_param(self):
        pairs = [('top_velocity', self.Top_Velocity_Edit_2.text()),
                 ('cutoff_velocity', self.Cutoff_Velocity_Edit_2.text()),
                 ('backlash', self.Backlash_Steps_Edit_2.text()),
                 ('start_velocity', self.Start_Velocity_Edit_2.text()),
                 ('slope', self.SlopeEdit.text())
                 ]

        valid_prs = filter(lambda x: x[1].isdigit(), pairs)
        for pair in valid_prs:
            answer = self.pump.property_set(*pair)
            print "In ParametersChange:\nasnwer = {0},value = {1}".format(answer, pair[1])

class HistoryDialog(QDialog, history_settings.Ui_Dialog):
    def __init__(self, pump, parent = None):
        super(HistoryDialog, self).__init__(parent)
        self.setupUi(self)

        self.pump = pump
        self.__appname__ = "Command History"
        self.setWindowTitle(self.__appname__)

        self.connect(self.refresh_Btn,\
                SIGNAL("clicked()"),\
                self.refresh)
        self.connect(self.commands_only,\
                SIGNAL("clicked()"),\
                self.refresh)
        self.connect(self.clear_history_btn,\
                SIGNAL("clicked()"),\
```

```
                self.clear_history)
        self.connect(self.user_com_btn,\
                SIGNAL("clicked()"),\
                self.refresh)

    def clear_history(self):
        self.pump.history = [ ]
        self.refresh()

    def refresh(self):
        wanted = self.pump.history
        wanted_string = ''
        if self.user_com_btn.isChecked():
            wanted = filter(lambda x: '?' not in x, self.pump.history)
        if self.commands_only.isChecked():
            for i in wanted:
                wanted_string += "{}\\r\n".format(i[:-1])
        else:
            for i in range(len(wanted)):
                wanted_string += "{0}:\t {1}\\r\n".format(i+1, wanted[i][:-1])

        self.history_edit.setPlainText(wanted_string)

class ReportsDialog(QDialog, python_settings.Ui_Dialog):

    def __init__(self, pump, window, parent=None):
        super(ReportsDialog,  self).__init__(parent)
        self.setupUi(self)

        self.__appname__ = "Reports Screen"
        self.setWindowTitle(self.__appname__)
        self.pump = pump
        self.window = window

        self.connect(self.refresh_interval_edit,\
                SIGNAL("textEdited(QString)"),\
                self.enable_button)

        self.connect(self.refresh_now_button,\
                SIGNAL("clicked()"),\
                self.refresh)

        self.refresh_interval_edit.setText("%s"\
                % int((self.window.refreshQtimer.interval() / 1000)))

        # Setting the refresh interval manually
        self.connect(self.refresh_interval_button,\
                SIGNAL("clicked()"),\
                self.setRefreshTime)

    # Enabling the volume button
    def enable_button(self):
        if self.refresh_interval_edit.text().isdigit():
            self.refresh_interval_button.setEnabled(True)
        else:
            self.refresh_interval_button.setEnabled(False)

    def refresh(self):
```

```
        """ The refresh function shows the pump major statistics.

        The refresh function is periodically run using the QTimer refreshQtimer
        When the timer timeouts the stats are fetched from the pump
        """

        self.pump.update_values()
        stats = self.pump.status

        self.Actual_Position_Edit.setText(stats["actual_pos"])
        self.Backlash_Steps_Edit.setText(stats["backlash_steps"])
        self.Cutoff_Velocity_Edit.setText(stats["cutoff_vel"])
        self.Position_Edit.setText(stats["absolute_pos"])
        self.Start_Velocity_Edit.setText(stats["starting_vel"])
        self.Top_Velocity_Edit.setText(stats["top_vel"])
        self.Checksum_Edit.setText(stats["checksum"])
        self.Fluid_Sensor_Edit.setText(stats["fluid_sensor"])
        self.Buffer_Status_Edit.setText(stats["buffer_status"])
        self.Version_Edit.setText(stats["version"])

    def setRefreshTime(self):
        text = self.refresh_interval_edit.text()
        if text.isdigit():
            self.window.refreshQtimer.setInterval(\
                    eval(text) * 1000)

            self.refresh_interval_edit.setText("%s"\
                    % int((self.window.refreshQtimer.interval() / 1000)))
            print "Timer interval Set: {} microseconds".format(eval(text) * 1000)
        else:
            QMessageBox.warning(self, self.__appname__, "Not a valid input")

        self.refresh_interval_edit.selectAll()

    def tick_refresh(self):
        if self.noRefresh.isChecked():
            self.window.cancel_timer()
            self.window.refresh_status = False
            self.window.scene.setForegroundBrush(\
                    QBrush(Qt.lightGray, Qt.CrossPattern))

        else:
            self.window.refresh_status = True
            self.window.refreshQtimer.start()
            self.window.scene.setForegroundBrush(\
                    Qt.NoBrush)


class NewDevDialog(QDialog, device_configuration.Ui_Dialog):

    def __init__(self, pump, window,  parent=None):
        super(NewDevDialog,  self).__init__(parent)
        self.setupUi(self)

        self.__appname__ = "Device Configuration"
        self.buttonBox =QDialogButtonBox(QDialogButtonBox.Ok
                |QDialogButtonBox.Cancel)
        self.buttonBox.accepted.connect(self.connect_with_port)
```

```
        self.buttonBox.rejected.connect(self.reject)
        self.horizontalLayout.addWidget(self.buttonBox)

        self.setWindowTitle(self.__appname__)
        self.comports = comports
        self.pump = pump
        self.window = window
        ports_available = list(self.comports())
        self.listWidget.addItem('loop://')
        for i in range(len(ports_available)):
            self.listWidget.addItem(ports_available[i][0])

        self.indexes = {"0": 0, "1": 1, "2": 2, "3": 3, "4":4,\
                "5": 5, "6": 6, "7": 7, "8": 8, "9": 9, "A": 10,\
                "B": 11, "C": 12, "D": 13, "E": 14, "F": 15}

    def connect_with_port(self):
        """Passes the selected item into the connect_new method of the pump."""

        try:
            port = self.listWidget.currentItem().text()
            address = '/%s' %self.address_box.currentText()
            self.pump.addr = address
            self.window.address_combobox.setCurrentIndex(\
                    self.indexes[address[-1]])
            self.pump.connect_new(port)
            text = "Port changed to %s\n Addressing to pump #%s" % (port,\
                    address[-1])
            self.window.command_label.setText(text)
            self.accept()
        except:
            text = "Parameters weren't set correctly!\n{}".format(sys.exc_info())
            self.window.command_label.setText(text)

class SyringePickDialog(QDialog, syringe_pick.Ui_Dialog):

    def __init__(self, pump, window, parent=None):
        super(SyringePickDialog, self).__init__(parent)
        self.setupUi(self)

        self.__appname__ = "Syringe Configuration"
        self.setWindowTitle(self.__appname__)
        self.pump = pump
        self.window = window

        syringe_sizes = ['50 micro', '100 micro', '250 micro', '500 micro', \
                '1000 micro', '5000 micro']

        for i in range(len(syringe_sizes)):
            self.listWidget.addItem(syringe_sizes[i])

    def select_new_syringe(self):
        """Passes the selected item into the connect_new method of the pump."""
        syringe = self.listWidget.currentItem().text().split()[0]
        self.pump.syringe_size = syringe
        text = "Syringe size is set to {size}".format(size = self.pump.syringe_size)
        self.window.command_label.setText(text)
```

```
        self.pump.update_values()

class AboutDialog(QDialog, about_dialog.Ui_Form):

    def __init__(self, text, appname, parent = None):
        super(AboutDialog, self).__init__(parent)
        self.setupUi(self)

        self.__appname__ = appname
        self.setWindowTitle(self.__appname__)
        self.setFixedSize(800, 500)
        self.text = text
        self.load_text()

    def load_text(self):
        self.textBrowser.setHtml(self.text)

class AboutDialog2(QDialog, about_dialog.Ui_Form):

    def __init__(self, text, appname, parent = None):
        super(AboutDialog2, self).__init__(parent)
        self.setupUi(self)

        self.__appname__ = appname
        self.setWindowTitle(self.__appname__)
        self.text = text
        self.load_text()
        self.QtButton = QPushButton("About Qt")
        self.horizontalLayout.addWidget(self.QtButton)

        self.connect(self.QtButton,\
                SIGNAL("clicked()"),\
                self.about_qt)

    def load_text(self):
        self.textBrowser.setText(self.text)
    def about_qt(self):
        QMessageBox.aboutQt(self, title = "About Qt")
```

1.6 License

The current software is licensed under LGPLv2.1 as is PySide, the Python Bindings for Qt with which it was developed. The purpose of the license is to encourage the free distribution of the project and to encourage everyone interested to come in and contribute to it. The license is provided in the GNU website https://www.gnu.org/licenses/lgpl-2.1.html

1.7 About

The software is the outcome of my individual project for the Systems Biology & Bioengineering Research Laboratory of the Mech. Engineering Department of NTUA

1.7.1 Tools Used

- The project was written in Python .
- For the GUI development I used the PySide Qt Bindings and the Qt-Designer application.
- The documentation was written in Sphinx
- All the code as well as the documentation was editted with the use of Vim

1.8 Contact Information

For information, bug reports or just to drop a comment about the software, either contact me on github or use one of the methods below:

- Mobile: +30 6978952969
- email: nickkouk@gmail.com

Otherwise you can contact the Systems Biology and Bioengineering Lab:

Department of Mechanical Engineering, Ktirio M
National Technical University of Athens
Heroon Polytechniou 9
15780 Zografou, GREECE
tel: +30 210 7721516

CHAPTER

TWO

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*